

UNIVERZITET U BEOGRADU
MATEMATIČKI FAKULTET



Pavle A. Cvejović

ANALIZA KARAKTERISTIKA BIBLIOTEKE REACT.JS

master rad

Beograd, 2023.

Mentor:

dr Vladimir Filipović, redovan profesor
Univerzitet u Beogradu, Matematički fakultet

Članovi komisije:

dr Filip Marić, redovan profesor
Univerzitet u Beogradu, Matematički fakultet

dr Aleksandar Kartelj, docent
Univerzitet u Beogradu, Matematički fakultet

Datum odbrane: _____

*Zahvaljujem se mentoru, profesoru dr. Vladimiru
Filipoviću na saradnji, korisnim primedbama i
sugestijama i posvećenom vremenu.*

*Ovaj rad posvećujem porodici za ljubav i podršku tokom
studija i pisanja rada.*

Naslov master rada: Analiza karakteristika biblioteke React.js

Rezime: U današnjem digitalnom dobu, interaktivnost i brza reakcija korisničkog interfejsa postali su ključni elementi za uspeh veb aplikacija. Upravo tu ulogu preuzima React.js, jedna od najinovativnijih biblioteka za izgradnju korisničkih interfejsa koja je potpuno transformisala način na koji razvijamo moderne veb aplikacije. Predstavljen javnosti od strane *Facebook*-a 2013. godine, *React.js* je postao vodeći izbor za razvoj *frontend* dela aplikacija u industriji, prepoznat po svojoj *efikasnosti*, *fleksibilnosti*, *modularnosti* i posebno svojoj sposobnosti da omogući *ponovnu upotrebu komponenti*.

U radu se izučava *JSX*, *sintaksički šećer* koji se koristi u *React.js*-u, kombinujući *HTML* i *Javascript* na jedinstven način, što omogućava lakše čitanje koda i optimizaciju performansi aplikacija. Analiziraju se i *React kuke* (eng. *React Hooks*), funkcije koje su promenile način na koji programeri pristupaju upravljanju stanjima i životnim ciklusima komponenti. Njihovo razumevanje i pravilna upotreba su ključ za efikasnu izgradnju složenih aplikacija. Takođe će se analizirati koncept *virtuelnog DOM-a* koji *React.js* koristi, inovaciju koja je bitno doprinela njegovoj efikasnosti. Kako bi se dublje razumela prednost *React.js*-a, ovaj rad će sadržati i poređenje sa *Angular* radnim okvirom, drugim popularnim alatom za izradu veb aplikacija. Programski kod aplikacije pisane u ova dva alata biće javno dostupan kao softver otvorenog koda i nalaziće se na *GitHub* repozitorijumu na adresi: <https://github.com/pavle99/React-Angular-comparison>.

U kombinaciji, ove analize pružiće temeljno razumevanje prednosti *React.js*-a kao tehnologije za razvoj modernih veb aplikacija, kao i uvid u razloge njegove sve veće popularnosti u industriji razvoja softvera.

Ključne reči: računarstvo, veb aplikacije, *Javascript*, *React.js*

Sadržaj

1	Uvod	1
1.1	Motivacija	1
1.2	Istorijski pregled	2
2	Virtuelni <i>DOM</i>	5
2.1	Šta je Virtuelni <i>DOM</i> ?	5
2.2	Kako funkcioniše virtuelni <i>DOM</i> ?	6
2.2.1	Atribut <i>key</i>	7
2.3	Prednosti Virtuelnog <i>DOM</i> -a u odnosu na stvarni <i>DOM</i>	8
3	<i>JSX</i>	10
3.1	Uvod u <i>JSX</i>	10
3.2	Mehanizam Prevoda <i>JSX</i> -a u <i>JavaScript</i>	10
3.3	Upotreba <i>JSX</i> -a u <i>React.js</i> -u	12
3.4	Pravila <i>JSX</i> -a	14
3.5	Prednosti <i>JSX</i> -a u odnosu na čisti <i>HTML</i>	15
4	Komponente i <i>Props</i>	17
4.1	Uvod u <i>React</i> komponente	17
4.2	Definisanje komponenti	18
4.3	Klasne komponente	22
4.4	Funkcijske komponente	23
4.5	<i>Props</i> : Prenos podataka između komponenti	24
4.6	Komponente višeg reda	27
5	<i>React kuke</i>	31
5.1	Uvod u <i>React kuke</i>	31
5.1.1	<i>StrictMode</i>	32

5.2	Ugrađene <i>React</i> kuke	32
5.2.1	useState	32
5.2.2	useEffect	35
5.2.3	useRef	40
5.2.4	useMemo i useCallback	44
5.2.5	useReducer	46
5.2.6	useContext	50
5.3	Pravila korišćenja <i>React</i> kuka	53
5.4	Pravljenje sopstvenih <i>React</i> kuka	54
5.5	Tranzicija sa klasnih na funkcijske komponente	55
6	Rutiranje u <i>React.js</i>	57
6.1	Uvod u rutiranje	57
6.2	Implementacija rutiranja u <i>React.js</i>	57
6.3	<i>React Router</i>	58
6.3.1	Postavljanje rutiranja u aplikaciji	58
6.3.2	Prikazivanje dece unutar roditeljske komponente	61
6.3.3	Rutiranje sa parametrima	62
6.3.4	Navigacija i redirekcija	63
6.3.5	Lenjo učitavanje	67
7	Podrška za <i>TypeScript</i>	70
7.1	Kako integrisati <i>TypeScript</i> u <i>React.js</i> projekat	70
7.2	<i>TypeScript</i> podrška za komponente i props	71
7.2.1	Generičke komponente	73
7.3	Primena <i>TypeScript</i> -a u <i>JSX</i> : Pregled <i>TSX</i> -a	75
8	Alati za pakovanje modula: <i>Webpack</i> i <i>Vite</i>	78
8.1	Uvod u alate za pakovanje modula	78
8.2	<i>Webpack</i>	79
8.2.1	Primer korišćenja <i>Webpack</i> -a u <i>React.js</i> aplikaciji	79
8.3	<i>Vite</i>	81
8.3.1	Primer korišćenja <i>Vite</i> -a u <i>React.js</i> aplikaciji	82
8.4	Poređenje <i>Vite</i> -a i <i>Webpack</i> -a	82
8.5	Tehnike za smanjenje veličine bundle-a	83

9	Poređenje <i>React.js</i>-a i <i>Angular</i>-a	85
9.1	Ukratko o <i>Angular</i> -u	85
9.2	Pregled aplikacije	86
9.3	Postavljanje projekta	87
9.3.1	Postavljanje <i>React.js</i> projekta preko <i>Vite</i>	87
9.3.2	Postavljanje <i>Angular</i> projekta koristeći <i>Angular CLI</i>	87
9.4	Izrada aplikacije	87
9.4.1	Lista beleški	88
9.4.2	Završavanje i brisanje beleške	92
9.4.3	Dodavanje beleške	97
9.4.4	Dohvatanje podataka sa servera i dinamičko ažuriranje liste	100
9.4.5	Rutiranje i navigacija	105
9.5	Rezime	115
10	Zaključak	117
	Literatura	118

Glava 1

Uvod

1.1 Motivacija

Razvoj *frontend* dela veb aplikacija igra ključnu ulogu u stvaranju zanimljivih i interaktivnih korisničkih iskustava na vebu. To uključuje dizajniranje i implementaciju korisničkog interfejsa, osiguranje odzivnosti na različitim uređajima i optimizaciju performansi. Da bi ubrzali proces razvoja i povećali produktivnost, programeri često koriste *Javascript* radne okvire (*eng. framework*). *Javascript* radni okviri su moćni alati koji programerima pružaju unapred izgrađene komponente, biblioteke i alate za pojednostavljivanje razvoja *frontend* dela veb aplikacija. Ovi radni okviri nude strukturiran pristup koji omogućava programerima da efikasno upravljaju složenostima modernih veb aplikacija. Promovišu ponovno korišćenje koda, modularnost i održivost, čime omogućavaju izgradnju skalabilnih i robustnih aplikacija.

Frontend radni okviri su se ranije suočavali sa značajnim izazovima - efikasnim upravljanjem i ažuriranjem korisničkog interfejsa kada se promene podaci na kojima se bazira. Tradicionalni pristupi često su podrazumevali direktno manipulisanje *DOM*-om, što je bilo ne samo nepraktično već je dovodilo do problema performansi i neefikasnog prikazivanja (*eng. render-ovanja*).

React.js se pojavio kao rešenje za ove probleme i preuzeo vodeću ulogu u razvoju *frontend* dela veb aplikacija. On je uveo koncept **virtuelnog *DOM***-a, lagane (*eng. lightweight*) reprezentacije stvarnog *DOM*-a. Korišćenjem ovog virtuelnog *DOM*-a, *React.js* može efikasno utvrditi minimalne promene koje su potrebne i ažurirati samo neophodne komponente, što dovodi do poboljšanja performansi.

Pored toga, *React.js* je usvojio **deklarativni** stil programiranja, gde progra-

meri opisuju željeni korisnički interfejs na osnovu njegovog stanja. Ovaj pristup pojednostavljuje razvoj korisničkog interfejsa eliminisanjem potrebe za ručnom manipulacijom *DOM*-a i imperativnim kodom. Umesto toga, programeri mogu da se fokusiraju na definisanje željenog stanja korisničkog interfejsa i da puste *React.js* da efikasno obavlja prikazivanje i ažuriranje.

Komponentna arhitektura *React.js*-a takođe igra značajnu ulogu u njegovoj popularnosti. Razbijanjem korisničkog interfejsa na ponovno iskoristive, autonomne **komponente**, programeri mogu da grade složene korisničke interfejse komponovanjem manjih blokova. Ovaj pristup promovise ponovno korišćenje koda, modularnost i lakše održavanje, što rezultuje skalabilnijim i održivijim aplikacijama.

React.js se prvenstveno koristi za razvoj **aplikacija jedne strane (SPA)**. Aplikacije jedne strane omogućavaju brže učitavanje veb stranica tako što preuzimaju samo potrebne podatke i komponente tokom interakcije korisnika, umesto da svaki put učitavaju celu stranicu. *React.js*, sa svojim efikasnim virtuelnim *DOM*-om i sposobnošću selektivnog ažuriranja komponenti, olakšava izgradnju i održavanje aplikacija jedne strane, pružajući korisnicima glatko i interaktivno iskustvo.

Takođe, obimni **ekosistem** *React.js*-a koji je podržan od strane programerske zajednice dodatno doprinosi njegovoj popularnosti. Ove biblioteke, kao što su *Redux* za upravljanje stanjem i *React Router* za rutiranje, pružaju dodatne funkcionalnosti i olakšavaju razvojni proces, čime *React.js* postaje sveobuhvatno i moćno rešenje za razvoj *frontend* dela veb aplikacija.

Na kraju, bitno je napomenuti da je *React.js* često pogrešno klasifikovan kao *radni okvir*, ali on je zapravo *biblioteka*. Za razliku od radnih okvira, koji obuhvataju kompletno rešenje i diktira određene šablone razvoja, biblioteke poput *React.js* su modularne, fleksibilne i koncentrišu se na jednu oblast, u ovom slučaju, na izgradnju korisničkih interfejsa, i programerima omogućava da odaberu ostale tehnologije koje žele da koriste u svojim projektima. Ova fleksibilnost je još jedan od razloga zašto je *React.js* toliko popularan među programerima.

1.2 Istorijski pregled

React.js, moćna *Javascript* biblioteka za konstrukciju korisničkih interfejsa, razvijena je u kompaniji *Facebook* 2011. godine s ciljem da se unapredi proces razvoja unutar njihove platforme. Glavni pokretač kreiranja *React.js*-a bila je sve veća kompleksnost *Facebook*-ove baze koda. Kako je njihova aplikacija postajala

sve složenija, upravljanje ažuriranjima i održavanje efikasnosti postalo je ogroman zadatak. Ovaj izazov je motivisao *Facebook*-ovog inženjera, *Jordana Walke*-a, da razvije efikasan prototip nazvan *FaxJs*, koji je bio inspirisan *XHP*-om, okvirom za *HTML* komponente za *PHP* koji je koristio *Facebook*.

Međutim, iako je *FaxJs* doneo određena poboljšanja, suočio se sa izazovima dinamičkih veb aplikacija koje zahtevaju česte komunikacije sa serverom. Ove interakcije su rezultirale promenama u prikazu veb aplikacije svaki put kada se promeni stanje, što je rezultiralo gubitkom svih prethodno sačuvanih informacija u *DOM*-u. Ova neefikasnost je imala negativan uticaj na performanse i korisničko iskustvo, što je podstaklo *Walke*-a da nastavi sa inovacijama.

Kao odgovor na ove probleme, *Walke* je razvio *Javascript* biblioteku usmerenu na kreiranje korisničkih interfejsa, nazvanu *React.js*. Ovo rešenje je imalo za cilj da učini stanje veb komponenti predvidljivijim i lakše upravljivim, garantujući glađe funkcionisanje aplikacija velikih razmera.

React.js je prvi put predstavljen široj zajednici programera na *JSConf US*-u 2012. godine. Do 2013. godine, *Facebook* je objavio *React.js* pod *MIT* licencom, omogućavajući programerima širom sveta da koriste ovu novu tehnologiju u svojim projektima. S vremenom, *React.js* je postao sve popularniji među programerima zbog svog efikasnog algoritma razlikovanja (*eng. diffing*), arhitekture zasnovane na komponentama i mehanizma za jednosmernu vezu podataka. Njegova popularnost je dodatno porasla sa objavom *React Native*-a 2015. godine, platforme koja je omogućila programerima da razvijaju mobilne aplikacije koristeći *React.js*.

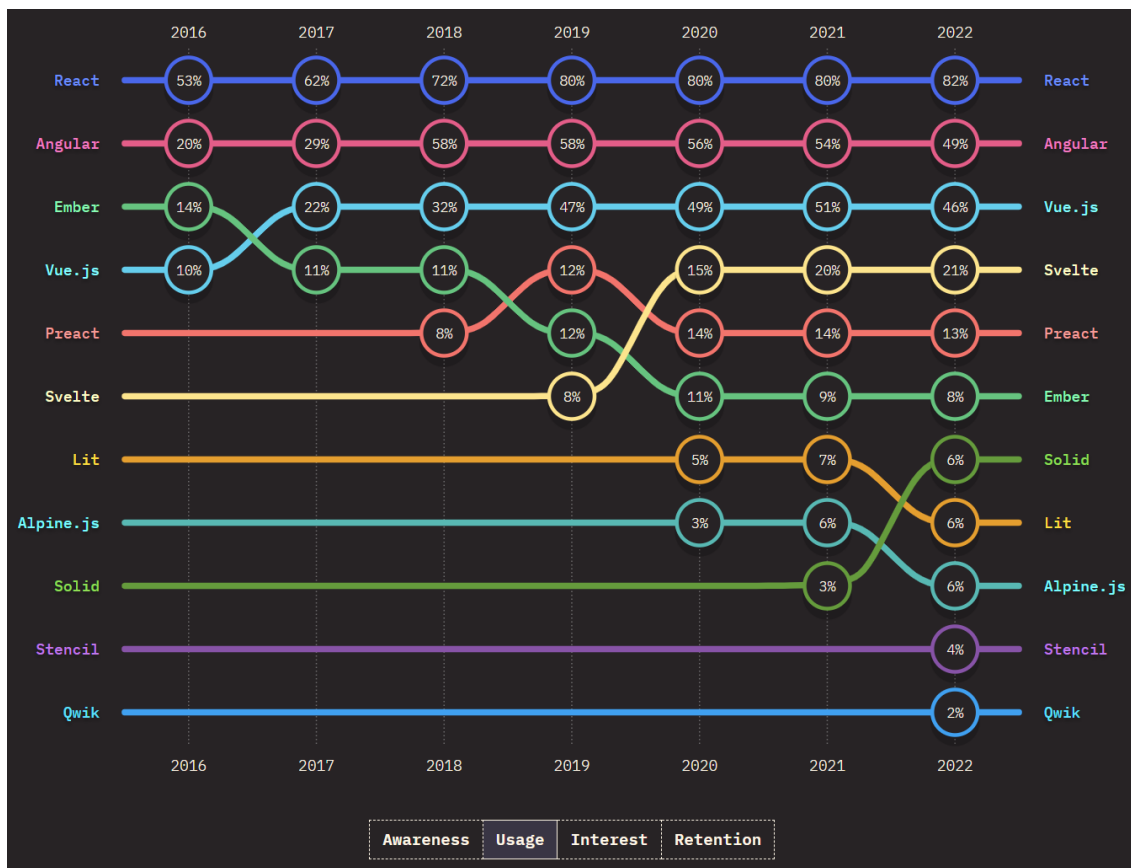
U ranim danima *React.js*-a, *klasne komponente* bile su standardni način za stvaranje ponovo upotrebljivih komponenti i upravljanje njihovim životnim ciklusom i **stanjem**. Međutim, sintaksa i semantika *Javascript* klasa mogu biti zbunjujuće, posebno za početnike. Štaviše, upravljanje složenim stanjima i sporednim efektima unutar klasnih komponenti je moglo biti nezgodno.

Kako bi se proces pojednostavio, *Facebook* je u *React.js* verziji 16.8 uveo *Hook*-ove (**kuke**), funkcionalnost koja je predstavljala značajan paradigmatki pomak. *Hook*-ovi su omogućili programerima da koriste stanje i funkcionalnosti životnog ciklusa bez klasa, umesto toga pišući svoje komponente kao čiste funkcije. Ovo je učinilo kod čitljivijim i održivijim, čineći *React.js* još privlačnijim programerima. Osim toga, kuke su pružile način za izdvajanje i ponovnu upotrebu logike sa stanjima između komponenti, unapređujući ponovnu upotrebljivost koda.

Uvođenje kuka i prelazak na funkcijske komponente dodatno su učvrstili pozici-

ju *React.js*-a kao vodeće *Javascript* biblioteke. Tranzicija nije samo revolucionirala način na koji programeri pišu *React.js* aplikacije, već je takođe osigurala kontinuiranu dominaciju 1.1 *React.js*-a u sferi razvoja korisničkih interfejsa. Do sada, iako su klasne komponente još uvek podržane u *React.js*-u, funkcijske komponente i kuke postali su preporučena praksa.

Dalja evolucija *Javascript* ekosistema donela je razvoj *metaframework*-a kao što su *Next.js* i *Remix*, koji se izgrađuju na osnovi *React.js*-a, pružajući poboljšane performanse i dodatne funkcionalnosti. Ovi alati, koji su zapravo preporučeni od strane zvanične *React.js* dokumentacije, doprinose fleksibilnosti i modularnosti *React.js*-a, potvrđujući njegov status kao vodeće biblioteke za izgradnju korisničkih interfejsa i ukazujući na buduće mogućnosti u svetu razvoja korisničkih interfejsa.



Slika 1.1: Evolucija korišćenja *Javascript frontend* biblioteka i radnih okvira u poslednjih 10 godina. [5]

Glava 2

Virtuelni *DOM*

2.1 Šta je Virtuelni *DOM*?

Virtuelni *DOM* (skraćeno, *VDOM*) označava ključni koncept u svetu savremenog veb razvoja, imajući ulogu kopije *DOM*-a u memoriji. Koncept *VDOM*-a je nastao i popularizovan kroz *Javascript* biblioteku, *React.js*. Ovaj koncept je podstakao paradigmatnu promenu u načinu na koji se veb aplikacije razvijaju i ažuriraju, pružajući efikasno rešenje za komplikacije i probleme povezane sa direktnom manipulacijom *DOM*-a.

Konceptualno, *VDOM* je apstraktna struktura podataka. Ona predstavlja **stablo** koje čuva elemente zajedno sa njihovim atributima i sadržajem, predstavljene kao *Javascript* objekte čije stvaranje i manipulacija su relativno jednostavni procesi. Glavni cilj ove apstrakcije je da poboljša performanse kompleksnih operacija korisničkog interfejsa, posebno onih koje zahtevaju česta ažuriranja.

React.js je imao ključnu ulogu u usvajanju i evoluciji koncepta *VDOM*-a koji je integralni deo dizajnerske filozofije *React.js*-a jer predstavlja jedan od temelja zbog kojeg biblioteka efikasno prikazuje (*eng. render*) korisničke interfejse, čineći *React.js* moćnim alatom za razvoj veb aplikacija.

Pored *VDOM*-a, postoji i koncept *Shadow DOM*-a, deo standarda veb komponenti, koji omogućava enkapsulaciju *CSS*-a i *Javascript*-a, stvarajući izolovane *DOM* podsisteme. Iako oba pružaju mehanizme za bolje upravljanje *DOM*-om, njihova upotreba i ciljevi su različiti. *VDOM* predstavlja apstraktnu verziju stvarnog *DOM*-a koja optimizuje performanse, dok *Shadow DOM* služi za izolaciju komponentata i obezbeđuje da stilovi i skripte korišćeni u jednoj komponenti ne mogu neželjeno uticati na druge komponente.

2.2 Kako funkcioniše virtuelni *DOM*?

VDOM omogućava efikasno ažuriranje korisničkog interfejsa koje se postiže procesom poznatim kao **pomirenje** (*eng. reconciliation*), koji se implementira koristeći heuristički algoritam poznat kao algoritam razlikovanja.

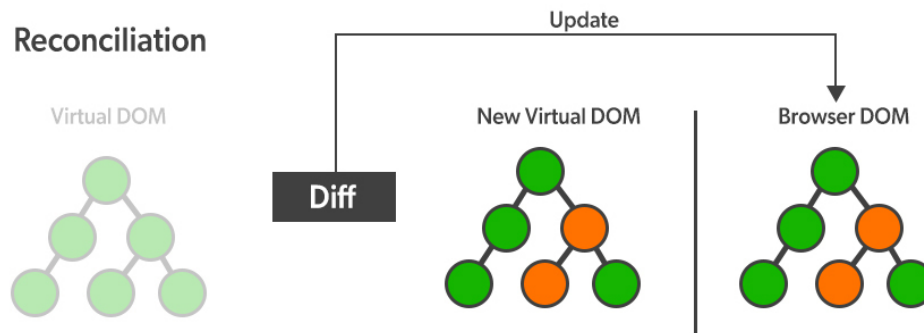
U računarskoj nauci, izazov određivanja minimalnog broja operacija za transformaciju jednog stabla u drugo je dobro proučen problem. Najnapredniji algoritmi za ovaj problem imaju vremensku složenost od $O(n^3)$ [2], gde je n broj elemenata u stablu. Ovaj kubni porast vremena sa veličinom stabla čini takve algoritme nepraktičnim za korisnički interfejs biblioteke kao što je *React.js*, gde bi prikazivanje 1000 elemenata zahtevalo red veličine od milijardu poređenja, što bi dovelo do značajnog usporavanja korisničkog interfejsa.

React.js rešava ovaj problem implementiranjem **heurističkog** algoritma sa vremenskom složenošću $O(n)$, na osnovu dve ključne pretpostavke. Prva pretpostavka je da će dva elementa različitih tipova proizvesti različita stabla. Druga pretpostavka je da programeri mogu nagovestiti koji dete elementi (*eng. children*) mogu biti stabilni kroz različite prikaze sa atributom **key**. Ove pretpostavke omogućavaju *React.js*-u da značajno smanji broj operacija potrebnih za ažuriranje *DOM*-a, održavajući visoke performanse čak i kada se bavi velikim brojem elemenata.

Kada se *React.js* aplikacija prvi put pokrene, kreiraju se i *VDOM* i stvarni *DOM*. *React.js* radi na osnovu **obrazaca posmatrača** (*eng. observable pattern*), i kada dođe do promene stanja, ažurira čvorove u *VDOM*-u. Nakon toga, *React.js* upoređuje *VDOM* sa stvarnim *DOM*-om i ažurira promene¹.

Algoritam razlikovanja olakšava proces pomirenja. Proverava korene elemente za promene, a ažuriranja zavise od tipova korenskih elemenata. Ako se tip korenskog elementa promeni, *React.js* odbacuje staro stablo i gradi novo, što rezultuje potpuno ponovnom izgradnjom stabla. Ako tip promenjenog elementa ostane isti, *React.js* proverava attribute obe verzije i ažurira samo čvorove koji su se promenili, bez menjanja stabla. Komponenta će biti ažurirana samo ako se promeni tip korenskog elementa ili ako se promene atributi. Ako se tip korenskog elementa ne promeni, ali se promeni tip nekog od deteta, *React.js* će zameniti samo promenjeno dete, a ostala deca će ostati nepromenjena. Ovaj proces se nastavlja sve dok se ne provere sva deca. Ako se neko dete doda ili ukloni, *React.js* će ažurirati samo pro-

¹Pogledati sliku ispod



Slika 2.1: Vizualizacija procesa ažuriranja virtuelnog i stvarnog *DOM*-a u *React.js*

menjenu deca, a ostala deca će ostati nepromenjena. Ovo se postiže korišćenjem atributa *key* koji omogućava *React.js*-u da identifikuje promenjenu decu.

2.2.1 Atribut *key*

Prikazivanje dece u *React.js*-u može predstavljati određene izazove, posebno kada su u pitanju performanse. Prema podrazumevanim postavkama, kada *React.js* vrši pomirenje dece nekog *DOM* čvora, on iterira kroz obe liste dece istovremeno i generiše *mutaciju* kada postoji razlika. Ovaj pristup dobro funkcioniše kada se element dodaje na kraj dece, ali može biti neefikasan kada se element umetne na početak. U takvim slučajevima, *React.js* bi mutirao svako dete umesto da shvati da može da zadrži određena podstabla netaknuta.

Da bi rešio ovaj problem, *React.js* podržava atribut *key*. Kada deca imaju ključeve, *React.js* koristi ključ da upari decu u originalnom stablu sa decom u narednom stablu. Ovo čini proces konverzije stabla efikasnijim, jer *React.js* sada zna koji elementi su se samo pomerili, umesto da su kreirani ili uništeni.

Atribut *key* bi idealno trebao biti **jedinstveni identifikator**. Može direktno da dolazi iz podataka ako element koji prikazujete već ima jedinstveni *ID*. Na primer:

```
<li key={item.id}>{item.name}</li>
```

Ako to nije slučaj, moguće je dodati novo *ID* svojstvo u dati objekat ili heširati neke delove sadržaja tako da se na osnovu njih generiše ključ. Ključ elementa ne mora biti globalno jedinstven, već samo jedinstven u okviru nivoa roditelja. Dva različita niza elemenata mogu imati iste ključeve, a dva deteta istog roditelja ne mogu imati iste ključeve jer će ih onda React tretirati kao iste elemente pri pomirenju.

Još jedna opcija je da se koristi **indeks** stavke u nizu kao ključ što se neretko može videti u tutorialima kada nema izvore jedinstvenih identifikatora za svaki element za upotrebu kao ključ, ali ovo nije preporučljivo jer može dovesti do problema sa performansama jer će se ključ menjati pri svakom prikazivanju komponente a glavni razlog za korišćenje ključeva je identifikacija elemenata kroz različita prikazivanja pa ako se ključ menja pri svakom prikazivanju, *React.js* će tretirati element kao novi pri svakom prikazivanju, što dovodi do nepotrebnih ponovnih prikazivanja i potencijalnog gubitka stanja. Ovaj problem se takođe javlja i pri preuređivanju (*eng. reorder*) elemenata, jer će pomeranje elementa niza promeniti i ključ tog elementa.

Zaključno, iako atribut `key` može značajno optimizovati performanse *React.js* aplikacija, važno je pravilno ga koristiti. Korišćenje jedinstvenih identifikatora iz podataka kao ključeva je dobra praksa, dok bi trebalo izbegavati korišćenje psuedonasumičnih vrednosti ili indeksa kao ključeva kad god je to moguće.

2.3 Prednosti Virtuelnog *DOM*-a u odnosu na stvarni *DOM*

VDOM u *React.js*-u nudi nekoliko prednosti u odnosu na stvarni *DOM*:

1. **Efikasnost:** *VDOM* omogućava *React.js*-u da minimizira ažuriranja stvarnog *DOM*-a, što je spora operacija u okruženjima pregledača. Izračunavanjem razlika između trenutnog i novog *VDOM*-a, korišćenjem algoritma razlikovanja, *React.js* može ažurirati samo delove stvarnog *DOM*-a koji su se promenili. Ovo dovodi do značajnih poboljšanja performansi.
2. **Jednostavnost:** *React.js* apstrahuje direktnu manipulaciju *DOM*-om, pružajući programerima jednostavniji programski model. Programeri mogu pisati kod kao da se cela stranica ažurira pri svakoj promeni, dok *React.js* brine o prikazivanju samo neophodnih komponenti.

3. **Predvidljivost:** Algoritam za pomirenje *React.js*-a osigurava da su ažuriranja komponenti predvidljiva. Programeri mogu biti sigurni da će se njihove komponente prikazivati i ponašati kako se očekuje, bez obzira na kompleksnost aplikacije.

Međutim, treba uzeti u obzir i **kompromise**. *React.js* bi teoretski mogao ponovo ažurirati celu aplikaciju pri svakoj akciji. Iako bi to dalo isti krajnji rezultat, bilo bi manje efikasno. U praksi, *React.js* primenjuje razlike prateći pravila navedena u prethodnim odeljcima, što čini uobičajene slučajeve upotrebe bržim.

Trenutna implementacija algoritma za pomirenje omogućava izražavanje činjenice da se podstablo premestilo među svojom braćom i sestrama, ali ne i da se premestilo negde drugde. Ovo ograničenje generalno nije problem, jer se korisnički interfejsi obično ne menjaju drastično između uzastopnih ažuriranja. Međutim, to je nešto što treba imati na umu prilikom dizajniranja složenih aplikacija.

Glava 3

JSX

3.1 Uvod u *JSX*

JSX nije ni niska ni *HTML*, već sintaksička ekstenzija za *JavaScript* koja se koristi u *React.js*-u za opisivanje strukture korisničkog interfejsa. *JSX* je skraćenica za *JavaScript XML*, što ukazuje na njegovu sličnost sa *XML* i *HTML* sintaksom. Iako nije obavezno koristiti *JSX* prilikom razvoja *React.js* aplikacija i postoje drugi načini pisanja UI komponenti, on je postao standard u *React.js* ekosistemu zbog svoje čitljivosti i efikasnosti.

U nastavku će biti prikazano kako se *JSX* prevodi u *JavaScript*, kako se koristi u *React.js*, koja su pravila za njegovo korišćenje i koje su prednosti u odnosu na čisti *HTML*.

3.2 Mehanizam Prevoda *JSX*-a u *JavaScript*

U okviru razmatranja *JSX*-a u *React.js* biblioteci, neophodno je objasniti proces njegove transformacije u običan *JavaScript* kod. Ova transformacija je obavezna jer veb pregledači nisu sposobni da direktno interpretiraju *JSX*. Iako *JSX* predstavlja proširenje *JavaScript* sintakse koje omogućava pisanje koda sličnog *HTML*-u, on nije validan *JavaScript*. Da bi se ova prepreka prevazišla, koristi se niz alata za prevod, poznatih kao *transpajleri*. Među najpopularnijim alatima za ovu svrhu su *Babel* i *TypeScript*[11]. Ovi alati transformišu *JSX* u pozive funkcije `React.createElement()`, koji su u stanju da interpretiraju veb pregledači.

Ukoliko je potrebno samo prevesti *JSX* u *JavaScript* sa *Babel*-om, to se može uraditi na sledeći način:

1. Prvo je potrebno instalirati `babel-cli` alat koristeći sledeću komandu: `npm install -g babel-cli`
2. Dalje, potrebno je instalirati `babel-preset-react` paket: `npm install babel-preset-react`
3. Opciono, može se instalirati i `babel-preset-es2015` paket: `npm install babel-preset-env` ako se koristi *ES6* sintaksa
4. U korenom direktorijumu projekta, potrebno je kreirati `.babelrc` datoteku sa sadržajem:

```
{
  "presets": ["react", "es2015"]
}
```

Pritom se `es2015` može opciono navesti, u zavisnosti od toga da li se koristi *ES6* sintaksa

5. Na kraju, potrebno je pokrenuti `babel` komandu: `babel src -d build`

U procesu razvoja *React* aplikacija, često se koriste alati koji automatizuju mnoge aspekte, uključujući i transformaciju *JSX*-a u *JavaScript* kao što su *Create React App* i *Vite* o kome će biti reči u poglavlju 8.

Ako na primer postoji sledeći *JSX* kod u aplikaciji:

```
import React from 'react';

function Primer() {
  return <h1 className="greeting">Zdravo, svete!</h1>;
}
```

on će biti preveden u poziv `React.createElement()` funkcije koja vraća *JavaScript* objekat koji predstavlja *DOM* element:

```
import React from 'react';

function Primer() {
  return React.createElement(
    'h1',
    {className: 'greeting'},
    'Zdravo, svete!'
  );
}
```

U prethodnom primeru, iako React biblioteka nije eksplicitno korišćena u originalnom kodu, bilo ju je obavezno uvesti jer se koristila funkcija `React.createElement()` u prevedenom kodu. Od *React.js* verzije 17 pojavila i nova verzija *JSX* transformacije [10] koja ne zahteva uvoz *React* biblioteke i ne forsira menjanje dosadašnje sintakse što znači da je potpuno kompatibilna unazad sa svim postojećim *JSX* kodom.

Ovo se postiže tako što umesto da se koristi `React.createElement()`, koristi se funkcija `jsx` iz `react/jsx-runtime` paketa koju kompajler automatski uvozi. Ovaj novi pristup donosi niz prednosti, uključujući potencijalno manju veličinu paketa i pojednostavljenje učenja *React.js*-a. Prethodni primer bi onda izgledao ovako:

```
function Primer() {
  return <h1 className="greeting">Zdravo, svete!</h1>;
}
```

a prevod u *JavaScript* kod bi bio sledeći:

```
import {jsx as _jsx} from 'react/jsx-runtime';

function Primer() {
  return _jsx(
    'h1',
    {className: 'greeting'},
    'Zdravo, svete!'
  );
}
```

3.3 Upotreba *JSX*-a u *React.js*-u

React.js prihvata činjenicu da je logika prikazivanja inherentno povezana sa ostalom logikom korisničkog interfejsa: kako se događaji obrađuju, kako se stanje menja tokom vremena, i kako se podaci pripremaju za prikaz. Umesto veštačkog razdvajanja tehnologija stavljanjem *markup*-a i logike u odvojene fajlove, *React.js* rukuje ovim izazovom kroz koncept „komponentata” koje prirodno integrišu oba aspekta.

JSX se koristi u *React.js* za definisanje komponenti. Komponente su osnovni blokovi *React.js* aplikacija, a *JSX* omogućava da se one definišu na način koji je intuitivan za programere koji su upoznati sa *HTML*.

React.js ne zahteva korišćenje *JSX*-a, ali većina ljudi ga smatra korisnim kao vizuelnu pomoć pri radu sa UI unutar *JavaScript* koda. Takođe omogućava *React.js*-u da prikaže korisnije poruke o greškama i upozorenjima.

U *JSX*-u možemo koristiti *JavaScript* izraze unutar vitičastih zagrada:

```
const fullName = 'Pera Peric';
const element = <h1>Zdravo, {fullName}</h1>;
```

Unutar vitičastih zagrada može se koristiti bilo koji validan *JavaScript* izraz. Na primer, poziv funkcije:

```
function formatName(user) {
  return user.firstName + ' ' + user.lastName;
}

const user = {
  firstName: 'Pera',
  lastName: 'Peric'
};

const element = (
  <h1>Zdravo, {formatName(user)}!</h1>
);
```

Sam *JSX* je takođe validan *JavaScript* izraz (jer nakon kompilacije on postaje poziv funkcije `React.createElement()`). Stoga se može koristiti u naredbama grananja i petljama:

```
function renderData(params) {
  const data = getData(params);
  if (data.error !== null) {
    return <h1>Greska: {data.error.message}</h1>;
  } else {
    return <p>{data.content}</p>
  }
}
```

Da bi se definisao atribut, može se koristiti sintaksa `<element atribut="vrednost">`. Atributi koji se koriste u *HTML*-u se mogu koristiti i u *JSX*-u¹. Takođe, umesto vrednosti atributa pod navodnicima možemo koristiti *JavaScript* izraz unutar vitičastih zagrada: `<element atribut={izraz}>`. Ako je vrednost atributa `true`, onda se atribut može navesti bez vrednosti: `<element atribut>`.

JSX elementi mogu imati i dečije elemente:

```
const element = (
```

¹Sa malim izmenama, pogledati sekciju 3.4.

```
<div>
  <h1>Zdravo, svete!</h1>
  <h2>Danas je {new Date().toLocaleDateString()}</h2>
</div>
);
```

Ovde su `<h1>` i `<h2>` dečiji elementi (*eng. children*) `<div>` elementa. Ukoliko element nema dečije elemente, ne mora se eksplicitno zatvarati kao u *HTML*-u, već se može zatvoriti koristeći `/>` sintaksu: `<element/>`.

Još će biti napomenuto da *JSX* pruža **zaštitu od napada putem ubrizgavanja skripti** (*eng. script injection*). Uvek je bezbedno prikazivati vrednosti koje dolaze iz promenljivih u *JSX*-u jer *React.js DOM* automatski zaštićuje (bezbedno prikazuje) sve vrednosti u *JSX*-u pre nego što ih prikaže. Takođe, on briše sve *HTML* elemente iz vrednosti promenljivih. Ovo ponašanje obezbeđuje da aplikacija bude bezbedna od XSS napada (*eng. cross-site-scripting*).

3.4 Pravila *JSX*-a

Kao što je već pomenuto, *JSX* nije potreban za pisanje *React.js* aplikacija, ali se smatra korisnim. Ukoliko se koristi, moraju se poštovati neka pravila:

1. **Uvek mora postojati jedan koreni element:** Kada se vraća više elemenata, onda se oni moraju nalaziti kao deca jednog roditeljskog elementa. Na primer, sledeći kod je neispravan:

```
function App() {
  return (
    <h1>Zdravo, svete!</h1>
    <h2>Danas je {new Date().toLocaleDateString()}</h2>
  );
}
```

Da bi se ovo ispravilo, potrebno je dodati još jedan element koji će biti roditelj dva postojeća elementa:

```
function App() {
  return (
    <div>
      <h1>Zdravo, svete!</h1>
      <h2>Danas je {new Date().toLocaleDateString()}</h2>
    </div>
  );
}
```

```
}  

```

Umesto `<div>` elementa, može se koristiti i `<React.Fragment>` element. Ovaj element neće biti prikazan u *DOM*-u, ali će omogućiti da se grupišu deca u jedan roditeljski element. Takođe, može se koristiti i skraćena sintaksa `<>` umesto `<React.Fragment>`.

2. **Za (skoro²) sve attribute se koristi kamilja notacija:** Na primer, `class` atribut se zove `className` u *JSX*-u, a `stroke-width` atribut se zove `strokeWidth`.

Ovo se radi jer se *JSX* prevodi u *JavaScript* i atributi napisani u *JSX*-u postaju ključevi *JavaScript* objekata. U komponentama, često će se koristiti prilagođeni (*eng. custom*) atributi u okviru *props* objekta, a oni će se onda koristiti kao ključevi objekata. Ali *JavaScript* ima ograničenja na imena promenljivih. Na primer, njihova imena ne mogu sadržati crtice ili biti rezervisane reči poput `class`, pa se zbog toga koristi *camelCase* sintaksa.

3. **Svi elementi moraju biti zatvoreni:** Čak iako element nema dečije elemente, on se mora zatvoriti. Na primer, `` element u *HTML*-u se ne zatvara, ali u *JSX*-u se mora zatvoriti: ``. Ovo se radi jer se *JSX* prevodi u *JavaScript* i nezatvoreni element `` bi rezultirao sintaksnom greškom u *JavaScript*-u pa se kod ne bi mogao izvršiti. Zbog toga se elementi bez dečijih elemenata moraju zatvarati koristeći `<element/>` sintaksu.

3.5 Prednosti *JSX*-a u odnosu na čisti *HTML*

JSX nudi brojne prednosti u odnosu na običan *HTML*, posebno kada se koristi u kontekstu razvoja aplikacija u *React.js* i ako je programeru poznat *HTML* i *JavaScript*, *JSX* je veoma jednostavan za učenje. Neke od ključnih prednosti su:

- **Integracija sa *JavaScript*-om:** Za razliku od običnog *HTML*-a, *JSX* omogućava direktnu integraciju sa *JavaScript*-om. Ovo znači da se mogu koristiti *JavaScript* izrazi direktno unutar *JSX*-a, što omogućava dinamičko kreiranje i manipulaciju interfejsa.

²Jedini atributi koji se pišu kao u *HTML*-u su `ariaa--` i `datad--` atributi. Ovo je zbog toga što su oni deo *DOM API*-ja, a ne *JavaScript API*-ja.

- **Kompilacija u vreme izgradnje:** *JSX* se kompilira u *JavaScript* kod u vreme izgradnje aplikacije, što znači da se greške mogu otkriti pre nego što se kod pokrene u pretraživaču. Ovo može pomoći u sprečavanju grešaka u vreme izvršavanja.
- **Poboljšana saradnja između programera i dizajnera:** Budući da *JSX* izgleda slično kao *HTML*, dizajneri i *frontend* programeri mogu lakše saradivati. Dizajneri mogu pisati *markup* koji izgleda poznato, a programeri mogu uključiti taj *markup* direktno u svoje *React* komponente.
- **Komponentni pristup:** *JSX* podržava koncept komponenti, što omogućava modularnost i ponovnu upotrebu koda. Svaka komponenta može imati svoj vlastiti *JSX* kod, što omogućava jasno razdvajanje funkcionalnosti i olakšava održavanje koda.
- **Prevenција XSS napada:** *JSX* automatski zaštićuje sve vrednosti koje se ugrađuju u *JSX*, što pomaže u sprečavanju *XSS* (*cross-site scripting*) napada. Ovo je značajna prednost u odnosu na običan *HTML*, gde je potrebno ručno upravljati bezbednošću.
- **Efikasnost:** *JSX* se prevodi u efikasne pozive `React.createElement()`, što omogućava brže ažuriranje *DOM*-a i bolje performanse u poređenju sa direktnim manipulacijama *HTML*-om.
- **Čitljivost i održivost:** Kôd napisan u *JSX*-u je često čitljiviji i lakši za održavanje od običnog *HTML*-a, posebno za složene interfejse. *JSX* omogućava jasno strukturiranje koda i olakšava razumevanje toka podataka.

Glava 4

Komponente i *Props*

4.1 Uvod u *React* komponente

Komponente su jedan od ključnih koncepta *React.js*-a i predstavljaju temelj na kojem se grade korisnički interfejsi. Na webu, *HTML* omogućava da se kreiraju bogato strukturirani dokumenti sa svojim ugrađenim skupom elemenata, kao što su `<h1>` i ``. Ovi *HTML* elementi se mogu posmatrati kao osnovni građevinski blokovi veb stranica.

Međutim, *React.js* uvodi dodatni nivo apstrakcije kroz koncept **komponenti**. Komponente su prilagođeni elementi korisničkog interfejsa koji se kreiraju kombinovanjem *HTML*-a, *CSS*-a i *Javascript*-a. Na primer, formular za prijavu na veb stranici može se pretvoriti u komponentu `<LoginForm />` koja se može prikazati gde god je potrebna autentifikacija korisnika na vašoj aplikaciji.

Ove komponente, poput *HTML* elemenata, mogu se komponovati, uređivati i ugnjezditi kako bi se dizajnirale celine stranice. Na primer, stranica za prodavnicu u e-trgovini koja se poseti može biti sastavljena od *React* komponenti.

Komponenta u *React.js*-u vraća *JSX*, što znači da je ona validan *Javascript* objekat. Ovo znači da se komponente mogu koristiti na različite načine, kao što su:

- **Prikazivanje u *DOM*-u:** Komponente se mogu prikazivati u *DOM*-u na isti način kao i *HTML* elementi.
- **Prikazivanje u drugim komponentama:** Komponente se mogu prikazivati u drugim komponentama, kao što je pokazano u primeru 4.2.

- **Generisanje u statički *HTML*:** Komponente se mogu generisati (*eng. render*) u statički *HTML*, što je korisno za optimizaciju za pretraživače (*eng. search engine optimization, SEO*) i prikazivanje na uređajima koji ne podržavaju *Javascript*.
- **Pretvaranje u niske:** Komponente se mogu pretvoriti u niske, što je korisno za testiranje.
- **Dodeljivanje komponenti promenljivima:** Komponente se mogu dodeliti promenljivoj (na isti način kao i *HTML* elementi), što je korisno za uslove i iteracije:

```
const LoginForm = <LoginForm />;
const App = <div>{LoginForm}</div>;
```

Kako se projekat razvija i postaje sve veći, može se uočiti da se komponente koje su već napravljene mogu ponovno koristiti za različite delove aplikacije. Na primer, ako je kreirana komponenta `<LoginForm />` za formular za prijavu, ona se može koristiti na svakom mestu u aplikaciji gde je potrebna prijava korisnika. Ovo ne samo da ubrzava razvoj, već i osigurava doslednost u dizajnu i funkcionalnosti.

Ukoliko nije od suštinskog značaja da se ima potpuno prilagođen dizajn, preporučljivo je koristiti prethodno napravljene komponente koje su dostupne kroz različite biblioteke. *React.js* ima veliku zajednicu otvorenog koda (*eng. open source*) koja je razvila i podelila hiljade ovakvih komponenti. Biblioteke kao što su *Chakra UI* i *Material UI* nude širok spektar komponenti, od dugmića i formulara, do kompletnih šablona za navigaciju i izgled stranice. Ove biblioteke mogu značajno ubrzati proces razvoja, omogućavajući fokusiranje na jedinstvene aspekte aplikacije umesto da se iznova kreiraju uobičajeni UI elementi.

4.2 Definisane komponente

Definisanje komponente u *React.js*-u podrazumeva kreiranje funkcija koje vraćaju određeni *JSX* ili klasa koja nasleđuje `React.Component` i implementira metodu `render()` koja vraća *JSX*. Na primer, može se definisati komponenta koja prikazuje sliku. Ova komponenta se može koristiti u bilo kojem delu aplikacije gde je potrebno prikazati ovu sliku. Komponenta je definisana tako što vraća *JSX markup* koji predstavlja sliku sa određenim `src` i `alt` atributima:

```
function Image() {  
  return ;  
};
```

Osim tradicionalnog načina definisanja funkcijskih komponenti, možemo ih definisati i kao *lambda* funkcije. Na primer, komponenta `<Image />` se može ekvivalentno definisati kao *lambda* funkcija na sledeći način:

```
const Image = () => {  
  return (  
      
  );  
};
```

Važno je napomenuti da komponente mogu uključivati druge komponente, stvarajući na taj način složene strukture. Ovo omogućava kreiranje složenih korisničkih interfejsa kroz kombinovanje jednostavnijih komponenti, na primer možemo kreirati komponentu `<ImageGallery />` koja uključuje više `<Image />` komponenti:

```
const ImageGallery = () => {  
  return (  
    <div>  
      <Image />  
      <Image />  
      <Image />  
    </div>  
  );  
};
```

Međutim, iako komponente mogu uključivati druge komponente, definicije komponenti ne bi trebalo da budu ugneždene jedna u drugu. Svaka komponenta treba da bude definisana na vrhu nivoa. Ovo omogućava jasnu strukturu koda i olakšava održavanje. Na primer, sledeći pristup nije dobar:

```
const ImageGallery = () => {  
  const Image = () => {  
    return ;  
  };  
  
  return (  
    <div>  
      <Image />  
      <Image />  
      <Image />  
    </div>  
  );  
};
```

```
);  
};
```

jer u ovom slučaju komponenta `<Image />` nije dostupna za ponovnu upotrebu u drugim delovima aplikacije. Umesto toga, komponenta `<Image />` treba da bude definisana na vrhu nivoa, a komponenta `<ImageGallery />` treba da je koristi.

Kada jedna komponenta treba da primi podatke od druge komponente, ti podaci se prosleđuju putem posebnog mehanizma koji se naziva *props*. Ovaj koncept je ključan za razumevanje načina na koji podaci teku kroz *React* aplikaciju i biće detaljnije objašnjen u sekciji 4.5.

Da bi se komponenta koristila u ostalim delovima aplikacije, mora se izvršiti izvoz (*eng. export*) te komponente iz fajla u kojem je definisana. Na primer, komponenta `<Image />` se može izvesti iz fajla `Image.js` na sledeći način:

```
const Image = () => {  
  return ;  
};  
  
export default Image;
```

Kako su komponente *Javascript* objekti one se mogu i **uslovno prikazivati**. Na primer, komponenta `<Image />` se može prikazati samo ako je neki uslov ispunjen:

```
const Image = () => {  
  return ;  
};  
  
const App = () => {  
  const condition = Math.random() > 0.5;  
  
  return (  
    <div>  
      {condition ? <Image /> : null}  
    </div>  
  );  
};
```

Ovaj kod možemo i skratiti koristeći logički operator `&&`:

```
const Image = () => {  
  return ;  
};  
  
const App = () => {  
  const condition = Math.random() > 0.5;  
  
  return (  
    <div>  
      {condition && <Image /> : null}  
    </div>  
  );  
};
```

```
<div>
  {condition && <Image />}
</div>
);
};
```

Bitno je napomenuti da treba biti oprezan pri korišćenju operatora `&&` sa vrednostima koje se izračunavaju kao netačne (*eng. falsy*), poput broja 0. Naime, ako je uslov izraza sa leve strane operatora `&&` *falsy* vrednost, sam izraz će biti ocenjen kao ta *falsy* vrednost, što može dovesti do neželjenog prikazivanja jer se evaluacija konjukcije završava čim jedan od argumenata bude izračunat kao netačan. Na primer, kod sledećeg koda:

```
const MyComponent = () => {
  const count = 0;

  return (
    <div>
      {count && <p>Brojac je veci od 0.</p>}
    </div>
  );
}
```

Biće generisan sledeći *HTML*:

```
<div>
  0
</div>
```

Ovo je zato što je vrednost `count` promenljive jednaka 0, koja je *falsy* vrednost, pa će sam izraz `count && <p>Count is greater than zero.</p>` biti izračunat kao 0. Da bi se izbegao ovaj problem, treba koristiti ternarni operator `?:`:

```
const MyComponent = () => {
  const count = 0;

  return (
    <div>
      {count ? <p>Count is greater than zero.</p> : null}
    </div>
  );
}
```

U suštini, definisanje komponente u *React.js*-u podrazumeva kreiranje samostalne jedinice koda koja predstavlja deo korisničkog interfejsa, može se ponovno

koristiti i može uključivati druge komponente. Ovo omogućava modularnost, ponovnu upotrebu koda i jasnu strukturu aplikacije.

4.3 Klasne komponente

U prethodnim sekcijama, komponente su bile definisane kao funkcije koje vraćaju *JSX* koje su postale osnovni način definisanja komponenti u *React.js*-u. Međutim, pre nego što su funkcijske komponente postale dominantne¹, **klasne komponente** su bile standard za kreiranje *React* komponenti.

Definisanje klasne komponente podrazumeva kreiranje klase koja nasleđuje `React.Component`. Ova klasa mora da implementira metodu `render()` koja vraća *JSX*. Na primer, komponenta `<Image />` se može definisati kao klasa na sledeći način:

```
class Image extends React.Component {
  render() {
    return (
      
    );
  }
}
```

Klasne komponente su bile jedini način da se koriste određene funkcionalnosti *React.js*-a, kao što su *metode životnog ciklusa* (eng. *lifecycle methods*), pre uvođenja *React kuka*. Metode životnog ciklusa su funkcije koje se automatski pozivaju u različitim tačkama u „životnom ciklusu” komponente, kao što su montiranje (eng. *mounting*) (kada se komponenta prvi put prikazuje), ažuriranje (eng. *updating*) (kada se promene atributi (eng. *props*)² ili stanje) i demontiranje (eng. *unmounting*) (kada se komponenta uklanja iz *DOM*-a).

Iako su klasne komponente sada uglavnom zastarele i *React* tim preporučuje korišćenje funkcijskih komponenta za nove projekte [9], još uvek je važno razumeti kako rade klasne komponente, jer se i dalje koriste u mnogim postojećim projektima.

¹Klasne komponente su počele da se zanemaruju sa uvođenjem *React kuka* u *React.js* verziji 16.8, koja je objavljena u februaru 2019. godine. Više o ovome u poglavlju 5.

²Objekat *props* će biti detaljnije objašnjen u sekciji 4.5

4.4 Funkcijske komponente

Funkcijske komponente, takođe poznate kao komponente bez stanja (*eng. stateless components*), su moderniji način definisanja komponenti u React-u. One su predstavljene u React verziji 0.14 ali su stekle još veće mogućnosti sa uvođenjem *React kuka*. Funkcijske komponente inicijalno nisu mogle da drže stanje a sa uvođenjem *React kuka*, funkcijske komponente sada mogu koristiti stanje i druge funkcionalnosti koje su prethodno bile dostupne samo klasnim komponentama. Stekle su popularnost zbog svoje jednostavnosti i konciznosti. Ove komponente su samo *Javascript* funkcije koje vraćaju *JSX*, prihvataju jedan argument objekta *props* i vraćaju *React* element.

Funkcijske komponente bi trebalo da budu čiste funkcije³ u smislu da ne menjaju svoje *props* ili stanje direktno. Umesto toga, one vraćaju novi *JSX* na osnovu *props*-a i stanja koje su primile. Ovo omogućava *React.js*-u da efikasno upravlja prikazivanjem i ažuriranjem komponenti.

Kao primer funkcijske komponente, uzeta je komponenta `<Image />` koja je prethodno bila definisana kao klasa a sada će biti prebačena u funkcijsku komponentu.

```
function Image() {
  return (
    
  );
};
```

Kao što se može videti, funkcijska komponenta je samo *Javascript* funkcija koja vraća *JSX*. Funkcijske komponente su lakše za čitanje i testiranje jer su to obične *Javascript* funkcije bez stanja ili metoda životnog ciklusa. Možemo predvideti njihov izlaz za dati ulaz jer nemaju interno stanje koje bi moglo uticati na prikazivanje.

U prethodnih nekoliko godina, *React* tim i zajednica su se kretali ka funkcijskim komponentama, a sa uvođenjem *React kuka*, funkcijske komponente su sada preporučeni način za kreiranje komponenti u React-u. *React* tim je izjavio da nema planove da ukloni klasne komponente iz React-a zbog kompatibilnosti

³Čiste funkcije su funkcije koje daju isti rezultat za iste argumente i ne izazivaju nikakve sporedne efekte. To znači da ne menja ništa izvan svog dometa, uključujući argumente koje prima, globalne promenljive ili stanje komponente.

unazad (*eng. backwards compatibility*), ali u budućnosti će nove funkcionalnosti biti uglavnom orijentisane ka funkcijskim komponentama.

Iako tranzicija može zahtevati neko vreme za prilagođavanje, prelazak na funkcijske komponente može doneti brojne prednosti, uključujući čistiji i čitljiviji kod, lakše testiranje i bolju optimizaciju performansi.

4.5 *Props*: Prenos podataka između komponenti

U *React.js*-u, komponente mogu primiti podatke od roditeljskih komponenti putem **atributa**, koja se nazivaju *props*. *Props* omogućavaju komponentama da budu fleksibilne i dinamičke, jer im se mogu proslediti različiti podaci pri svakom prikazivanju. *Props* se prosleđuju komponenti kao objekat, a komponenta ih prima kao argument funkcije. Na primer, ukoliko postoji komponenta `<Product/>` koja treba da prikaže informacije o proizvodu, podaci o samom proizvodu mogu biti prosleđeni toj komponenti putem *props*-a:

```
const Product = (props) => {
  return (
    <div>
      <h1>{props.productName}</h1>
      <p>{props.description}</p>
    </div>
  );
};

const App = () => {
  return (
    <div>
      <Product
        name="React"
        description="Javascript biblioteka za kreiranje korisnickih
                    interfejsa"
      />
      <Product
        name="Angular"
        description="Typescript radni okvir za kreiranje korisnickih
                    interfejsa"
      />
    </div>
  );
};
```

U ovom primeru, komponenti su prosleđena dva atributa, `productName` i `description`, koji čine svojstva objekta *props*, koji je komponenta primila kao argument. Ovim

GLAVA 4. KOMPONENTE I PROPS

svojstvima je moguće pristupiti unutar komponente, koristeći `props.productName` i `props.description`.

Radi čitljivosti koda, *props* objekat se može destrukuirati (*eng. destructuring*⁴) u argumentu funkcije kako bi se izvukle pojedinačne vrednosti umesto da se koristi `props.name` i `props.description`:

```
const Product = ({ productName, description }) => {
  return (
    <div>
      <h1>{productName}</h1>
      <p>{description}</p>
    </div>
  );
};
```

Kao i kod običnih *JavaScript* objekata, moguće je definisati zadatu vrednost za svojstvo unutar *props*-a. Ukoliko komponenti nije prosleđena vrednost za tu osobinu, ona će biti `undefined`, ali je definisati podrazumevanu vrednost za to svojstvo. Na primer, u slučaju da komponenti `<Product/>` nije dodeljena osobina `description`, moguće je odrediti zadatu vrednost za to svojstvo:

```
const Product = ({ productName, description = "Bez opisa" }) => {
  return (
    <div>
      <h1>{productName}</h1>
      <p>{description}</p>
    </div>
  );
};
```

U slučaju kada se *props* prenosi kao objekat, moguće je iskoristiti (...) operator⁵ kako bismo sve osobine objekta preneli komponenti. Na primer, ukoliko postoji objekat `product` koji sadrži svojstva `productName` i `description`, ovaj objekat može biti prosleđen komponenti `<Product/>` putem (...) operatora:

```
const product = {
  productName: "React",
  description: "Javascript biblioteka za kreiranje korisnickih interfejsa"
};

const App = () => {
  return (
```

⁴Destructuiranje je sintaksna konstrukcija uvedena u *ES6* koja omogućava izvlačenje podataka iz objekta ili niza i dodeljivanje tih podataka u nove promenljive.

⁵Operator razdvajanja (*eng. spread*) (...) je takođe sintaksna konstrukcija uvedena u *ES6* koja omogućava da se niz ili objekat razdvoje na pojedinačne elemente.


```

    <div>
      <Product {...product} />
    </div>
  );
};

```

U situaciji gde je potrebno da komponenta prikazuje decu elemente, dostupno je svojstvo `props.children` za pristupanje deci elementima komponente. Ovaj *prop* je uvek dostupan u komponenti, čak i ukoliko deca nisu eksplicitno prosleđena, i obuhvata sadržaj koji se nalazi unutar otvorenih i zatvorenih elemenata prilikom korišćenja komponente. Njegova funkcionalnost je posebno korisna kada želimo da definišemo dinamički unutrašnji sadržaj komponente, koji se može razlikovati iz upotrebe u upotrebu. Deca mogu biti bilo šta: tekst, druge komponente, *HTML* elementi ili kompleksnije strukture. Na primer, ako postoji komponenta `<Product/>` koja prikazuje informacije o proizvodu i potrebno je da se prikaže slika proizvoda, slika može biti prosleđena kao dete ovoj komponenti:

```

const Product = ({ productName, description, children }) => {
  return (
    <div>
      <h1>{productName}</h1>
      <p>{description}</p>
      {children}
    </div>
  );
};

const App = () => {
  return (
    <div>
      <Product
        productName="React"
        description="Javascript biblioteka za kreiranje korisnickih
          interfejsa"
      >
        
      </Product>
    </div>
  );
};

```

Potrebno je imati na umu da je `children` *prop* dizajniran da prima *JSX* elemente, komponente i tekstualni sadržaj. Takođe, `children` može da prihvati i funkcije, ali samo ako one vraćaju tipove podataka pomenute ranije. Na primer, ukoliko je potrebno prikazati dinamički izračunatu cenu proizvoda, može se kom-

ponenti `<Product/>` proslediti funkcija koja vraća *JSX* element:

```
const Product = ({ productName, description, children }) => {
  const price = calculatePrice(productName);

  return (
    <div>
      <h1>{productName}</h1>
      <p>{description}</p>
      {children({ price })}
    </div>
  );
};
```

```
const App = () => {
  return (
    <div>
      <Product
        name="React"
        description="Javascript biblioteka za kreiranje korisnickih
          interfejsa"
      >
        {({ price }) => <p>{price}</p>}
      </Product>
    </div>
  );
};
```

4.6 Komponente višeg reda

Komponente višeg reda (*eng. Higher-Order Component (HOC)*) su moćan koncept u React-u koji omogućava ponovno korišćenje logike komponenti i poboljšava njihovu funkcionalnost. *HOC* je funkcija koja prima komponentu kao argument i vraća modifikovanu verziju te komponente.

Glavna svrha *HOC*-a je omogućavanje deljenja logike i funkcionalnosti između više komponenti. Na ovaj način, umesto da se kopira i ponavlja ista logika u više komponenti, logika se može izdvojiti u *HOC* i primeniti na potrebne komponente. Ovime se doprinosi čistijem kodu, smanjenju dupliranja koda i poboljšanoj organizaciji. Takođe, postiže se veća fleksibilnost i konfigurabilnost, budući da se različiti *HOC*-ovi mogu primeniti na istu komponentu kako bi se prilagodio njen rad različitim scenarijima.

Važno je napomenuti da *HOC*-ovi nemaju svoju vizuelnu reprezentaciju, već se koriste za rukovanje sa logikom i sa funkcionalnošću komponenti. Oni su sna-

žan alat koji pruža fleksibilnost i unapređuje ponovno iskoristivost komponenti u React-u.

Jedan jednostavan primer *HOC*-a bi bio uslovni prikaz komponente. Ovaj *HOC* prima komponentu kao argument i vraća novu komponentu koja prikazuje prosleđenu komponentu samo ako je ispunjen određeni uslov što uklanja potrebu da se kopira i ponavlja ista logika u više komponenti:

```
const withConditionalRendering = (Component) => {
  return (props) => {
    const { condition } = props;

    return condition ? <Component {...props} /> : null;
  };
};

const Product = ({ name, description, price, discount }) => {
  return (
    <div>
      <h1>{name}</h1>
      <p>{description}</p>
      <p>Originalna cena: ${price}</p>
      <p>Cena sa popustom: ${price * (1 - discount)}</p>
    </div>
  );
};

const ProductWithConditionalRendering = withConditionalRendering(Product);

const App = () => {
  const product = {
    name: "React",
    description: "A \textit{Javascript} library for building user
      interfaces",
    price: 100,
    discount: 0.1,
  };

  return (
    <div>
      <ProductWithConditionalRendering
        condition={product.discount > 0}
        {...product}
      />
    </div>
  );
};
```

U ovom primeru specifično, prikazivaće se komponenta `<Product/>` samo ako je taj proizvod na popustu, tj. ako je vrednost polja `discount` veća od 0.

Još jedan veoma koristan primer *HOC*-a je `withLoading` koji prikazuje indikator učitavanja dok se komponenta učitava. Ovaj *HOC* prima komponentu kao argument i vraća novu komponentu koja prikazuje indikator učitavanja dok se komponenta učitava. Ovaj *HOC* je izuzetno koristan jer omogućava prikazivanje indikatora učitavanja za bilo koju komponentu koja se učitava, bez potrebe da se ista logika kopira i reprodukuje unutar više komponenti.

```
const withLoading = (Component) => {
  return (props) => {
    const [isLoading, setIsLoading] = useState(true);

    useEffect(() => {
      setTimeout(() => {
        setIsLoading(false);
      }, 2000);
    }, []);

    return isLoading ? <div>Loading.</div> : <Component {...props} />;
  };
};

const Product = ({ name, description, price, discount }) => {
  return (
    <div>
      <h1>{name}</h1>
      <p>{description}</p>
      <p>Originalna cena: ${price}</p>
      <p>Cena sa popustom: ${price * (1 - discount)}</p>
    </div>
  );
};

const ProductWithLoading = withLoading(Product);

const App = () => {
  const product = {
    name: "React",
    description: "A \textit{Javascript} library for building user interfaces",
    price: 100,
    discount: 0.1,
  };

  return (
    <div>
      <ProductWithLoading {...product} />
    </div>
  );
};
```

U ovom primeru su spomenute *React kuke* `useState` i `useEffect` koje će biti detaljnije objašnjene u sekcijama 5.2.1 i 5.2.2 ali sve što one rade ovde je da inicijalno postave vrednost `isLoading` promenjive na `true` čime se prikazuje indikator učitavanja a nakon 2 sekunde se postavi vrednost promenjive na `false` čime *HOC* prikazuje komponentu koja je prosleđena kao argument.

Glava 5

React kuke

5.1 Uvod u *React kuke*

Jedan od najznačajnijih koraka u razvoju *React.js*-a bio je uvođenje *React kuka* u verziji 16.8, objavljenoj u februaru 2019. godine.

Pre uvođenja *React.js*-a, komponente u *React.js*-u su se uglavnom pisale kao klase, koje su koristile metode životnog ciklusa za upravljanje stanjima i efektima. Metode životnog ciklusa, kao što su `componentDidMount`, `componentDidUpdate` i `componentWillUnmount`, bile su ključne za upravljanje različitim fazama životnog ciklusa komponente. Međutim, ove metode su često bile izvor konfuzije za mnoge programere, posebno za one koji su tek počinjali sa *React*-om. Takođe, korišćenje metode životnog ciklusa često je dovodilo do problema sa organizacijom koda, jer je logika koja je povezana mogla biti raspršena po različitim metodama.

React kuke su predstavljene kao rešenje za ove probleme. One omogućavaju korišćenje stanja i drugih *React* funkcionalnosti bez potrebe za pisanjem klasa. Sa *React kukama*, može se izvući logika stanja iz komponente, tako da se može testirati nezavisno i ponovno koristiti. *React kuke* omogućavaju da se povezana logika grupiše zajedno, što čini kod čistijim i lakšim za razumevanje.

Ovaj pristup je postao veoma popularan među programerima i sada je standardna praksa u razvoju *React* aplikacija. Bitno napomenuti da *React kuke* ne zamenjuju prethodno stečeno znanje o konceptima u *React.js*-u, već pružaju direktniji *API* za *React* koncepte koji su tu bili duže vreme: *props*, *state*, *context*, *refs*, i *lifecycle*. Kako će kasnije biti pokazano, *React kuke* takođe nude novi moćan način za kombinovanje ovih koncepta.

U narednim sekcijama ovog poglavlja, detaljnije će biti istraženo kako se koriste

React kuke, kao i njihove prednosti i moguće izazove.

5.1.1 StrictMode

Strogi režim (*eng. Strict Mode*) je alatka korisna za razvoj koja pomaže u otkrivanju potencijalnih problema u aplikaciji. Kada je aktiviran, *React* će izvršiti dodatne provere i upozoriti na probleme koji bi mogli proći nezapaženo.

Pri korišćenju nekih od kuka u kombinaciji sa strogim režimom u *React*-u, koji je podrazumevano uključen u svim novim *React* aplikacijama, možda ćete primetiti da se funkcijske komponente **pozivaju dva puta** umesto jednom tokom razvoja. Ovo je namerno ponašanje dizajnirano da pomogne u hvatanju neželjenih efekata.

Jedna od stvari koje strogi režim radi je da dva puta poziva `render` metode funkcijskih komponenti. Ovakvo ponašanje po dizajnu pomaže u otkrivanju nečistih funkcija tako što dvostruko pozivanje `render` metode u strogom režimu omogućava *React.js*-u da uporedi rezultate oba prikazivanja sa istim *props*-ovima i ukoliko se rezultati razlikuju, *React.js* će prijaviti upozorenje.

Važno je napomenuti da se ovo dvostruko isrtavanje događa samo u razvojnom režimu. U produkciji, strogi režim neće dva puta pozivati `render` metode. Takođe, ovo dvostruko isrtavanje neće imati nikakav uticaj na efekte ili ažuriranja stanja - oni se i dalje primenjuju samo jednom.

5.2 Ugrađene *React kuke*

U ovoj sekciji će biti objašnjene najčešće korišćene *React kuke* ugrađene u samoj *React.js* biblioteci i koje su dostupne svim *React* aplikacijama. U nastavku će biti istraženo kako se koriste ove kuke i kako se mogu koristiti za upravljanje stanjem i efektima u *React* aplikacijama.

5.2.1 `useState`

Ova kuka se koristi za upravljanje stanjem u funkcijskim komponentama i omogućava komponenti da „pamti” informacije, poput korisničkog unosa. Ova funkcija kao argument prima početnu vrednost stanja i vraća niz sa dva elementa: trenutno stanje i funkciju za ažuriranje stanja. Funkcija za ažuriranje stanja može da prima novu vrednost stanja ili funkciju koja vraća novu vrednost stanja. Kroz ovu funk-

ciju za ažuriranje, stanje komponente se menja i dolazi do ponovnog prikazivanja komponente. Na primer:

```
import React, { useState } from 'react';

function Example() {
  const [count, setCount] = useState(0);

  return (
    <div>
      <button onClick={() => setCount(count + 1)}>
        Inkrementiraj
      </button>
      <p>Kliknuo si dugme {count} puta</p>
    </div>
  )
}
```

U prethodnom primeru, `count` je trenutno stanje, a `setCount` je funkcija za ažuriranje stanja. Kada korisnik klikne na dugme, `setCount` će se pozvati sa novom vrednošću stanja, koja će biti `count + 1`. *React.js* će ponovo prikazati komponentu i prikazati novu vrednost stanja.

Bitno je spomenuti da u *React.js*-u postoji mehanizam koji se naziva „grupno ažuriranje” (*eng. batch update*) koji efikasno ažurira stanje. Kada pozovete `setState` (ili `set` funkciju iz `useState` kuke) više puta unutar istog ciklusa životnog događaja (npr. unutar istog rukovaoca događaja (*eng. event handler*) ili `useEffect` kuke), *React.js* će grupisati ova ažuriranja u samo jedan ciklus ažuriranja umesto da pokreće višestruke cikluse ažuriranja. Ovo poboljšava performanse aplikacije.

U slučaju da se prethodni primer izmeni tako da se klikom na dugme brojač povećava za 2, putem dva poziva funkcije `setCount(count + 1)`, očekivani rezultat je da će brojač biti povećan za 2 nakon klika na dugme „Increment”. Međutim, klikom na dugme, brojač se povećava samo za 1. To je zato što *React.js* grupiše ova dva poziva `setCount` u jedan ciklus ažuriranja, a oba poziva koriste istu vrednost `count` kao osnovu za ažuriranje.

Da bi povećali brojač za 2, treba se koristiti funkcionalni oblik `setCount`, koji prima funkciju čiji je argument prethodno stanje i koja vraća novu vrednost stanja: `setCount(prevCount => prevCount + 1)`. Sada, svaki poziv `setCount` koristi najnoviju vrednost stanja kao osnovu za ažuriranje, tako da brojač pravilno povećava za 2.

Nadalje, `set` funkcija iz `useState` kuke (u nastavku, `setState`) je asinhrona

funkcija, jer ako bi bila sinhrona a radi se neka skupa operacija to bi ostavilo veb pregledač blokiranim dok se ne završi ažuriranje stanja, što znači da se promene stanja ne odražavaju odmah.

Važno je razumeti ovo jer može dovesti do neočekivanih rezultata ako se ne koristi pravilno. Na primer, pristupanjem ažuriranom stanju odmah nakon pozivanja `setState` verovatno će se dobiti stara vrednost stanja, jer se promena još nije odrazila (delovaće kao da vrednost „kasni” jedan korak). Razlog za to je što *React.js* može odložiti ažuriranje stanja za kasnije radi optimizacije performansi. Za izvršavanje neke akcije odmah nakon ažuriranja stanja, može se koristiti `useEffect` kuka o kojoj će biti više rečeno u nastavku.

Ukoliko je početno stanje rezultat skupe operacije (npr. poziv *API*-ja), može se proslediti funkcija koja vraća početno stanje. Ova funkcija će se pozvati samo prilikom inicijalnog prikazivanja komponente, tako da će se izbegnuti nepotrebno izvršavanje funkcije za inicijalizaciju stanja. Na primer:

```
const [state, setState] = useState(() => {
  const initialState = someExpensiveComputation();
  return initialState;
});
```

U nekim situacijama, moguće je da postoji promenljiva koja se može izračunati na osnovu stanja, ali koja ne zahteva sopstvenu kuku. Na primer, možda postoji lista korisnika i treba prikazati samo one korisnike koji zadovoljavaju određeni kriterijum. Umesto da se koriste dve *useState* kuke - jedna za sve korisnike i druga za filtrirane korisnike - može se koristiti samo jedan *useState* za sve korisnike, a zatim izračunati filtrirane korisnike na osnovu tog stanja.

Evo kako bi to moglo izgledati¹:

```
import React, { useState, useEffect } from 'react';

function UserList() {
  const [users, setUsers] = useState(userListFromServer);
  const [searchTerm, setSearchTerm] = useState('');

  const filteredUsers = users.filter(user =>
    user.name.toLowerCase().includes(searchTerm.toLowerCase())
  );

  return (
    <div>
      <input
```

¹Ovde koristimo izmišljenu funkciju `fetchUser` koja vraća podatke o korisniku sa servera.

```
        type="text"
        placeholder="Pretraži korisnike"
        value={searchTerm}
        onChange={event => setSearchTerm(event.target.value)}
      />

      {filteredUsers.map(user => (
        <div key={user.id}>{user.name}</div>
      ))}
    </div>
  );
}

export default UserList;
```

U datom primeru, koristi se `useState` kuka za `searchTerm`, koji predstavlja trenutni termin pretrage. Zatim se izračunava `filteredUsers` filtriranjem liste korisnika na osnovu `searchTerm`. Kada se `searchTerm` promeni, komponenta će se ponovno ažurirati, `filteredUsers` će se ponovo izračunavati, i prikazivaće se samo korisnici koji zadovoljavaju kriterijum pretrage. Dodatni `useState` za `filteredUsers` nije korišćen, jer se njegova vrednost može izračunati direktno iz `users` i `searchTerm`.

Na kraju, treba napomenuti da se u striktnom modu, funkcije unutar `useState` kuke pozivaju dva puta. Ovo se dešava zato što strogi režim dva puta prikazuje komponentu kako bi proverio da li se stanje menja između ta dva prikazivanja. Ovo dvostruko prikazivanje neće imati uticaja na efekte ili ažuriranja stanja - oni se i dalje primenjuju samo jednom.

5.2.2 `useEffect`

`useEffect` je jedan od najvažnijih kuka koja omogućava izvršavanje „sporednih efekata” u funkcijskim komponentama. Sporedni efekti su operacije koje ne utiču direktno na izlaz komponente, ali su ključne za njenu funkcionalnost. To uključuje rad sa mrežom (na primer, dohvatanje podataka), rad sa *DOM*-om, postavljanje tajmera, pretplate (*eng. subscriptions*), i druge operacije koje se obično izvršavaju u metodama životnog ciklusa u klasnim komponentama.

`useEffect` kuka prima dva argumenta: funkciju koja sadrži sporedne efekte i niz zavisnosti. Funkcija koja sadrži sporedne efekte će se izvršiti nakon svakog prikazivanja komponente, uključujući i prvo. Drugi argument, niz zavisnosti, je opcion, i služi da kontroliše kada će se funkcija sa sporednim efektima izvršiti.

Postoje 3 slučaja:

- Niz zavisnosti se ne navede - funkcija sa sporednim efektima će se izvršiti nakon svakog prikazivanja komponente. Taj slučaj je ilustrovan primerom 5.1.
- Niz zavisnosti se navede i nije prazan - funkcija sa sporednim efektima će se izvršiti nakon prvog prikazivanja komponente i nakon svakog prikazivanja komponente u kojem se promenila neka od vrednosti iz niza zavisnosti. Taj slučaj je ilustrovan primerom 5.2.
- Niz zavisnosti se navede i prazan je - funkcija sa sporednim efektima će se izvršiti samo jednom, nakon prvog prikazivanja komponente. Taj slučaj je ilustrovan primerom 5.3.

Kod 5.1: Primer korišćenja `useEffect` kuke bez niza zavisnosti.

```
useEffect(() => {  
  console.log('Ovo se ispisuje pri prvom prikazivanju komponente i nakon  
    bilo koje izmene u stanju komponente.');
```

Kod 5.2: Primer korišćenja `useEffect` kuke sa nizom zavisnosti.

```
useEffect(() => {  
  console.log('Ovo se ispisuje pri prvom prikazivanju komponente i nakon  
    svakog prikazivanja komponente u kojem se promenila vrednost promenjive  
    count.');
```

Kod 5.3: Primer korišćenja `useEffect` kuke sa praznim nizom zavisnosti.

```
useEffect(() => {  
  console.log('Ovo se ispisuje samo pri prvom prikazivanju komponente.');
```

Funkcija koja sadrži sporedne efekte može u sebi koristiti resurse koji se moraju „čistiti” (*eng. cleanup*), kao što su pretplate i tajmeri. Pretplate su mehanizmi za dinamičko ažuriranje podataka iz izvora i moraju se očistiti kako bi se oslobodili resursi. Ova funkcija može da vrati funkciju koja će se izvršiti pre sledećeg prikazivanja komponente, i pre nego što se izvrši funkcija sa sporednim efektima. Ova

funkcija se često zove „funkcija za čišćenje” i korisna je za „čišćenje” resursa koje je funkcija sa sporednim efektima koristila. Na primer:

```
useEffect(() => {
  const subscription = subscribeToSomething();
  return () => {
    subscription.unsubscribe();
  };
});
```

Čest razlog zbog kog se koristi `useEffect` kuka je dohvaćanje podataka sa *API*-ja (eng. *Application Programming Interface*). U ovom kontekstu, `async/await` se koristi za asinhrono dohvaćanje podataka, omogućavajući čekanje završetka operacije pre nego što se ažurira stanje. Na primer, ukoliko se želi dohvatiti podatak o korisniku sa servera, može se koristiti `useEffect` kuka sa praznim nizom zavisnosti, kako bi se dohvatilo samo jednom, pri prvom prikazivanju komponente:

```
import React, { useState, useEffect } from 'react';

function ProfilePage({ userId }) {
  const [user, setUser] = useState(null);

  useEffect(async () => {
    await fetchUser(userId).then((response) => setUser(response.data));
  }, []);

  return (
    <div>
      <h1>{user.name}</h1>
      <h2>{user.email}</h2>
    </div>
  );
}
```

Ovde, `await` čeka da se asinhrona operacija dohvaćanja podataka sa servera uspešno završi pre nego što se ažurira stanje `user` objekta sa novim podacima. Takođe, ovaj kod se može malo poboljšati dodavanjem indikatora učitavanja dok se podaci ne dohvate, kao i indikatora za greške u slučaju neuspešnog dohvaćanja podataka.

```
import React, { useState, useEffect } from 'react';

function ProfilePage({ userId }) {
  const [user, setUser] = useState(null);
  const [isLoading, setIsLoading] = useState(false);
  const [isError, setIsError] = useState(false);
```

```
useEffect(async () => {
  setIsLoading(true);
  await fetchUser(userId)
    .then((response) => {
      setUser(response.data);
    })
    .catch((error) => {
      setIsError(true);
    })
    .finally(() => {
      setIsLoading(false);
    });
}, []);

if (isLoading) {
  return <div>Loading...</div>;
}

if (isError) {
  return <div>Something went wrong...</div>;
}

return (
  <div>
    <h1>{user.name}</h1>
    <h2>{user.email}</h2>
  </div>
);
}
```

Može se primetiti da je u prethodnim primerima bila korišćena `async` funkcija kao prvi argument `useEffect` kuke. Ovo je moguće jer `useEffect` kuka može da prima i asinhronu funkciju kao prvi argument ali sama kuka ne može biti asinhrona funkcija.

Još jedan veoma čest slučaj upotrebe `useEffect` kuke je kada treba da se **izvrši neka akcija odmah nakon što se promenila promenjiva** dobijena iz `useState` kuke. Već je viđeno da se funkcija `setState` ne izvršava odmah nakon poziva, već se čeka sledeće prikazivanje komponente. Na primer, ukoliko bi želeli da ispišemo novi broj u konzolu odmah nakon što se promenljiva `count` promeni, naivno bi napisali sledeći kod:

```
const [count, setCount] = useState(0);

const increment = () => {
  setCount(count + 1);
  console.log(count);
};
```

```
return (  
  <button onClick={increment}>  
    Increment  
  </button>  
);
```

Međutim, pošto je `setCount` asinhrona funkcija, ona ne ažurira broj odmah. Umesto toga, zakazuje ažuriranje da se dogodi u budućnosti. Kada se pozove `console.log(count)`, `count` još uvek ima svoju staru vrednost, pa se stara vrednost i ispiše u konzolu. Ukoliko bi se nastavilo sa pozivanjem `console.log(count)` unutar funkcije `increment`, u konzoli bi se neprestano prikazivala prethodna vrednost promenljive `count`, stvarajući utisak o jednom koraku zakašnjenja.

Da bi se pravilno ispisala nova vrednost, može se koristiti `useEffect` kuka sa nizom zavisnosti koji sadrži promenljivu `count`:

```
const [count, setCount] = useState(0);  
  
useEffect(() => {  
  console.log(count);  
}, [count]);  
  
const increment = () => {  
  setCount(count + 1);  
};  
  
return (  
  <button onClick={increment}>  
    Increment  
  </button>  
);
```

Neke od najčešćih grešaka koje se prave prilikom korišćenja `useEffect` je da se ne navedu sve promenjive koje se koriste u `useEffect` funkciji u nizu zavisnosti. Ovaj problem se može rešiti pomoću alata za statičku analizu koda, a drugi problem, koji je dosta teži za rešavanje, je kada se u nizu zavisnosti navede previše promenljivih. U tom slučaju, `useEffect` će se izvršavati nakon svake promene bilo koje od promenljivih navedenih u nizu zavisnosti, što može dovesti do neočekivanog ponašanja aplikacije. Ukoliko se u nizu zavisnosti navede previše promenljivih, to je verovatno znak da je potrebno podeliti `useEffect` na više manjih `useEffect` kuka. Kao posledica ovog problema, takođe je moguće ući u beskonačnu petlju gde se `useEffect` izvršava nakon svake promene promenljive, a promenljiva se menja u `useEffect` funkciji, na primer:

```
const [count, setCount] = useState(0);

useEffect(() => {
  setCount(count + 1); // Ovo dovodi do beskonacnog ponovnog azuriranja jer
  // se promenjiva count konstantno menja
}, [count]);

return (
  <button onClick={increment}>
    Increment
  </button>
);
```

Još će biti napomenuto da ukoliko je potrebno da se neka funkcija izvrši pre nego što se komponenta prvi put prikaže na ekranu, može se koristiti `useLayoutEffect` kuka koja se samo po tome razlikuje od `useEffect` kuke. Mana ove kuke je što se korisniku neće prikazati ekran sve dok se ne izvrši `useLayoutEffect` funkcija.

5.2.3 useRef

React kuka `useRef` je kuka koja omogućava **pristup i interakciju** sa *DOM* elementima ili čuvanje promenljive vrednosti koja se ne menja između ažuriranja. Kuka `useRef` prima jedan argument - inicijalnu vrednost promenljive koja se čuva, označenu kao `initialValue`, i vraća objekat sa `.current` svojstvom koje je inicijalizovano sa prosleđenim argumentom. Ovaj objekat će ostati konstantan kroz sve cikluse ažuriranja komponente. To znači da je moguće postaviti `.current` na bilo koju vrednost i ona će ostati nepromenjena između ažuriranja. Ova karakteristika je korisna za čuvanje vrednosti koja mora opstati kroz cikluse ažuriranja komponente, izbegavajući pritom izazivanje ponovnog ažuriranja pri promeni vrednosti.

Kada se koristi za pristup *DOM* elementima, `useRef` se koristi u kombinaciji sa `ref` atributom u *JSX*-u. Kuka `useRef` se prvo poziva da bi se kreirao `ref` objekat, a zatim se taj objekat prosleđuje `ref` atributu elementa. Ovo omogućava pristup tom elementu izvan `render` metode, na primer, u `useEffect` ili funkcijama za obradu događaja. Primer upotrebe:

```
import React, { useRef } from 'react';

function MyComponent() {
  const myRef = useRef(null);

  function handleClick() {
```

```
// Pristup DOM elementu
console.log("DOM element", myRef.current);
}

return (
  <div>
    <button ref={myRef} onClick={handleClick}>
      Klikni me
    </button>
  </div>
);
}

export default MyComponent;
```

U ovom primeru, `useRef` se koristi za pristup button elementu. Kada se klikne na dugme, `handleClick` funkcija će se pozvati i ispisati *DOM* element na konzolu.

`useRef` kuka se takođe može koristiti za **čuvanje promenljivih vrednosti koje se ne menjaju između ažuriranja**. Ova funkcionalnost je korisna kada je potrebno sačuvati neku vrednost kako bi preživela ciklus ažuriranja komponente, ali se ne želi izazvati ponovno ažuriranje kada se ta vrednost promeni, na primer:

```
import React, { useRef, useEffect } from 'react';

function MyComponent() {
  const renderCount = useRef(0);

  useEffect(() => {
    renderCount.current = renderCount.current + 1;
  });

  return (
    <div>
      Komponenta MyComponent je prikazana {renderCount.current} puta
    </div>;
  );
}

export default MyComponent;
```

U ovom primeru, `useRef` se koristi za čuvanje broja ažuriranja komponente. Kada se komponenta ažurira, `useEffect` će se pozvati i povećati broj ažuriranja. Međutim, promena vrednosti `renderCount.current` neće izazvati ponovno ažuriranje komponente.

Još jedan primer gde se `useRef` često koristi je u kombinaciji sa `forwardRef` funkcijom iz *React API*-ja. Funkcija `forwardRef` se koristi pri pravljenju sopstve-

nih (*eng. custom*) komponenti i omogućava da se `ref` prosledi kroz komponentu do deteta, što omogućava roditeljskoj komponenti da pristupi `ref`-u deteta. Ovo je korisno kada roditeljska komponenta treba da interaguje sa detetom, na primer da dobije vrednost input polja nakon podnošenja formulara:

```
import React, { useRef } from 'react';

const CustomInput = React.forwardRef((props, ref) => (
  <input ref={ref} {...props} />
));

function MyComponent() {
  const inputRef = useRef();

  const handleSubmit = (e) => {
    e.preventDefault();
    const value = inputRef.current.value;
    console.log(value);
    // Ovde se mogu izvršiti dalje akcije sa vrednoscu input polja
  };

  return (
    <div>
      <form onSubmit={handleSubmit}>
        <label>
          Unesite tekst:
          <CustomInput type="text" ref={inputRef} />
        </label>
        <button type="submit">Potvrdi</button>
      </form>
    </div>
  );
}

export default MyComponent;
```

U ovom primeru, `CustomInput` je *custom* komponenta koja koristi `forwardRef` da bi omogućila roditeljskoj komponenti da prosledi `ref`. Komponenta `MyComponent` koristi `useRef` da bi kreirala `ref`, a zatim prosleđuje taj `ref` `CustomInput` komponenti. Kada se formular podnese (*eng. submit*), `handleSubmit` funkcija će se pozvati i u njoj se može pristupiti vrednosti `CustomInput` komponente pomoću `inputRef.current.value`.

Razlika između useRef i useState

Iako useRef i useState mogu izgledati slično na prvi pogled, jer oba omogućavaju čuvanje vrednosti između ažuriranja, postoje ključne razlike između njih. Kada se koristi useState da bi se promenilo stanje, to će izazvati ponovno ažuriranja komponente. Sa druge strane, promena vrednosti .current useRef objekta **neće izazvati ponovno ažuriranje**. Ovo čini useRef idealnim za čuvanje vrednosti koje treba da prežive ciklus ažuriranja, ali koje ne bi trebalo da izazovu ponovno ažuriranje kada se promene.

Sa useState, trenutna vrednost stanja je dostupna direktno iz povratne vrednosti kuke. Sa useRef, trenutna vrednost je dostupna kao .current svojstvo useRef objekta. Kuka useState prima inicijalnu vrednost koja se koristi samo pri prvom ažuriranju. Ako useState prima funkciju, ta funkcija će se pozvati samo jednom pri prvom ažuriranju. Sa druge strane, useRef prima inicijalnu vrednost koja se koristi za postavljanje .current svojstva useRef objekta pri svakom ažuriranju.

useRef je idealan kada treba da se čuva vrednost koja treba da preživi ciklus ažuriranja, ali koja ne bi trebalo da izazove ponovno ažuriranje kada se promeni. To je takođe korisno kada treba da se pristupi *DOM* elementima. Kuka useState se koristi kada treba da se čuva stanje koje može izazvati ponovno ažuriranje kada se promeni. Ovo je korisno za čuvanje vrednosti koje su direktno povezane sa onim što se prikazuje u render metodi.

Jedan od čestih slučajeva kada je useRef bolji izbor od useState je kada je potrebno pratiti trenutnu vrednost input polja, ali bez izazivanja ponovnog ažuriranja pri svakoj promeni vrednosti.

Na primer, ako se pretpostavi da postoji formular sa input poljem i potrebno je pratiti korisnikov unos, ali bez izazivanja ponovnog ažuriranja pri svakoj promeni vrednosti. Ovo se može postići upotrebom useRef:

```
import React, { useRef, useEffect } from 'react';

function MyComponent() {
  const inputRef = useRef(null);

  function handleInputChange() {
    console.log(inputRef.current.value);
  }

  return (
```

```
<div>
  <input ref={inputRef} onChange={handleInputChange} />
</div>
);
}

export default MyComponent;
```

U gornjem primeru, `useRef` se koristi za čuvanje reference na `input` element. Kada se vrednost inputa promeni, `handleInputChange` funkcija će se pozvati i ispisati trenutnu vrednost inputa na konzolu. Međutim, promena vrednosti inputa neće izazvati ponovno ažuriranje komponente, što bi bio slučaj kada bismo koristili `useState` za praćenje vrednosti inputa.

5.2.4 useMemo i useCallback

`useMemo` i `useCallback` su dve kuke u *React.js*-u koje omogućavaju **keširanje** (*eng. memoization*), što može biti ključno za optimizaciju performansi aplikacije. Obe kuke primaju kao prvi argument funkciju i takođe primaju kao drugi argument niz zavisnosti koji funkcioniše na isti način kao i niz zavisnosti u `useEffect` kuki - ako se neka od vrednosti u nizu zavisnosti promeni, keširana vrednost će se ponovo izračunati. Ako se prosledi prazan niz zavisnosti, keširana funkcija će se pozvati samo pri prvom prikazivanju komponente u kojoj se poziva `useMemo` ili `useCallback`.

`useMemo` je kuka koja omogućava **keširanje skupih izračunavanja**. Na primer, ako postoji funkcija koja vrši složena izračunavanja na osnovu određenih promenljivih, može se koristiti `useMemo` da se spreči ponovno izračunavanje tih vrednosti sve dok se te promenjive ne promene. Ovo se postiže tako što `useMemo` prima funkciju koja vrši izračunavanje i niz zavisnosti, i vraća keširanu vrednost koja se ažurira samo kada se promene zavisnosti.

Čest slučaj kada se koristi `useMemo` je kada se treba izračunati neka vrednost na osnovu stanja komponente. Na primer, može se pretpostaviti postojanje komponente koja prikazuje listu korisnika, zajedno sa `input` poljem koje omogućava filtriranje liste korisnika po njihovim imenima. Filtrirana lista korisnika se može sačuvati putem `useMemo` kako bi se izbeglo ponovno računanje liste svaki put kada se promeni stanje `input` polja:

```
import React, { useState, useMemo } from 'react';

function MyComponent({ users }) {
```

```

const [searchTerm, setSearchTerm] = useState('');

const filteredUsers = useMemo(() => {
  return users.filter(user => user.name.includes(searchTerm));
}, [searchTerm, users]);

function handleSearchTermChange(event) {
  setSearchTerm(event.target.value);
}

return (
  <div>
    <input value={searchTerm} onChange={handleSearchTermChange} />
    <ul>
      {filteredUsers.map(user => (
        <li key={user.id}>{user.name}</li>
      ))}
    </ul>
  </div>
);
}

export default MyComponent;

```

S druge strane, `useCallback` je kuka koja omogućava **keširanje funkcija**. Ovo je posebno korisno kada se prosleđuju *callback* funkcije kao *props* u dečije komponente. Bez keširanja, svaki put kada se roditeljska komponenta prikazuje, nova instanca *callback* funkcije bi se kreirala, što bi moglo izazvati nepotrebno ažuriranje dečije komponente. Kuka `useCallback` rešava ovaj problem tako što prima funkciju i niz zavisnosti, i vraća keširanu verziju te funkcije koja se ažurira samo kada se promene zavisnosti.

Čest slučaj kada se koristi ova kuka je kada se prosleđuje funkcija kao *prop* u optimizovanu dečiju komponentu² koja se oslanja na referentnu jednakost za sprečavanje nepotrebnih ažuriranja što znači da će se promene pratiti samo na nivou referenci, a ne i unutar samih objekata. Na primer:

```

import React.js, { memo, useState, useCallback } from 'react';

function ChildComponent({ onClick }) {
  console.log('ChildComponent je prikazan!');
  return (
    <button onClick={onClick}>Klikni me</button>
  );
}

```

²`memo` je funkcija višeg reda iz *React.js API*-ja koja omogućava keširanje *React.js* komponenti. Kada se omota komponenta sa `memo`, *React.js* će preskočiti ponovno prikazivanje te komponente i ponovo koristiti poslednje keširano stanje ako se *props* nisu promenili.

```
}  
  
const MemoizedChildComponent = memo(ChildComponent);  
  
function ParentComponent() {  
  const [count, setCount] = useState(0);  
  
  const increment = useCallback(() => {  
    setCount(count => count + 1);  
  }, []);  
  
  return (  
    <div>  
      <p>Kliknuli ste {count} puta</p>  
      <MemoizedChildComponent onClick={increment} />  
    </div>  
  );  
}  
  
export default ParentComponent;
```

Iako su `useMemo` i `useCallback` slični u tome što oba omogućavaju keširanje i prihvataju niz zavisnosti, ključna razlika je u tome što `useMemo` kešira vrednost, dok `useCallback` kešira funkciju.

Obe ove kuke su korisne alatke za optimizaciju performansi, ali treba ih koristiti oprezno da ne bi došlo do prekomernog keširanja koje može dovesti do prekomerne upotrebe memorije. Iako keširanje može poboljšati performanse sprečavanjem nepotrebnih izračunavanja, ona takođe ima trošak. Svaka keširana vrednost ili funkcija zauzima prostor u memoriji. Ako se previše stvari kešira, može se završiti sa prekomernom upotrebom memorije, što može usporiti vašu aplikaciju. Takođe, nije potrebno sve funkcije ili izračunavanja keširati. Ako funkcija nije skupa za izvršavanje ili se ne poziva često, možda nema potrebe za keširanjem. Slično tome, ako vrednost koja se izračunava ne menja se često, možda nema potrebe za keširanjem.

5.2.5 useReducer

`useReducer` je kuka u *React.js*-u koja omogućava **upravljanje složenijim stanjima i prelazima između stanja**. Ova kuka je alternativa `useState`-u i koristi se kada je logika za ažuriranje stanja složenija ili kada se sledeće stanje zavisi od prethodnog.

`useReducer` prima dva argumenta: `reducer` funkciju i početno stanje, a vraća

niz sa trenutnim stanjem i `dispatch` funkcijom. *Reducer* funkcija je funkcija koja prima trenutno stanje i akciju, a zatim vraća novo stanje. Funkcija `dispatch` se koristi za pokretanje akcija koje ažuriraju stanje i prima akciju kao argument.

Osnovni oblik korišćenja `useReducer` izgleda ovako:

```
const [state, dispatch] = useReducer(reducer, initialState);
```

Dok bi `reducer` funkcija mogla izgledati ovako:

```
function reducer(state, action) {
  switch (action.type) {
    case 'ACTION_TYPE':
      const newState = ...
      // logika za ažuriranje stanja
      return newState;
    default:
      throw new Error();
  }
}
```

`dispatch` funkcija je mehanizam kojim se pokreću akcije. Akcija je obično objekat koji sadrži `type` polje, i može sadržati dodatne podatke. Kada se `dispatch` funkcija pozove sa akcijom, `useReducer` pokreće `reducer` funkciju sa trenutnim stanjem i datom akcijom.

```
dispatch({ type: 'ACTION_TYPE', /* dodatni podaci */ });
```

`useReducer` je posebno koristan u slučajevima složenog stanja koje uključuje više vrednosti ili kada naredno stanje zavisi od prethodnog stanja. Takođe, ima koristi u situacijama kada su dečije komponente optimizovane i oslanjaju se na *referentnu jednakost* kako bi sprečila nepotrebna ažuriranja. U ovim slučajevima, `dispatch` funkcija iz `useReducer`-a garantuje da će se referenca na nju promeniti samo ukoliko se promeni i sama `dispatch` funkcija.

Upravljanje formularima je čest primer upotrebe `useReducer` kuke. Kada se radi sa formom koja ima više polja, može biti korisno koristiti `useReducer` za upravljanje stanjem svih polja.

Evo jednostavnog primera formulara za prijavu:

```
import React.js, { useReducer } from 'react';

const initialState = {
  username: '',
  password: '',
};

function reducer(state, action) {
```

```
switch (action.type) {
  case 'field': {
    return {
      ...state,
      [action.fieldName]: action.payload,
    };
  }
  case 'reset':
    return initialState;
  default:
    return state;
}
}

function LoginForm() {
  const [state, dispatch] = useReducer(reducer, initialState);
  const { username, password } = state;

  function handleInputChange(e) {
    dispatch({
      type: 'field',
      fieldName: e.currentTarget.name,
      payload: e.currentTarget.value,
    });
  }

  function handleReset() {
    dispatch({ type: 'reset' });
  }

  function handleSubmit(e) {
    e.preventDefault();
    // Ovde se mogu slati podaci na server...
    console.log(`Submitting: ${username}, ${password}`);
  }

  return (
    <div>
      <form onSubmit={handleSubmit}>
        <input
          type="text"
          name="username"
          value={username}
          onChange={handleInputChange}
        />
        <input
          type="password"
          name="password"
          value={password}
          onChange={handleInputChange}
        />
      </form>
    </div>
  );
}
```

```
    <button type="submit">
      Submit
    </button>
    <button type="button" onClick={handleReset}>
      Reset
    </button>
  </form>
</div>
);
}

export default LoginForm;
```

U ovom primeru, `useReducer` se koristi za upravljanje stanjem `username` i `password` polja. Kada se vrednost bilo kog polja promeni, funkcija `handleInputChange` se poziva, koja zatim poziva `dispatch` funkciju sa odgovarajućom akcijom. *Reducer* funkcija ažurira stanje na osnovu ove akcije. Takođe postoji `reset` akcija koja resetuje stanje formulara na početno stanje. Kada se formular podnese, funkcija `handleSubmit` se poziva, koja bi onda poslala podatke formulara na odgovarajući *API*.

Kada se koristi `useReducer`, postoji nekoliko važnih stvari na koje treba obratiti pažnju kako bi se osiguralo da se stanje pravilno ažurira i da se izbegnu potencijalne zamke.

- *Reducer* funkcija mora biti *čista funkcija*: *reducer* funkcija koja se prosleđuje `useReducer` kuki treba da bude čista funkcija. To znači da za iste ulazne vrednosti uvek treba da vraća isti rezultat, i da ne bi trebalo da ima sporedne efekte. Ovo je važno jer *React.js* može da pozove *reducer* funkciju više puta pre nego što ažurira UI, pa ako *reducer* funkcija ima sporedne efekte, mogu se pojaviti neočekivani problemi.
- Direktna mutacija stanja treba da se izbegava: Prilikom ažuriranja stanja unutar *reducer* funkcije, preporučuje se izbegavanje direktnih promena trenutnog stanja. Umesto toga, treba vratiti novi objekat koji predstavlja novo stanje. Pravljenje direktnih promena u stanju može dovesti do neplaniranih problema, jer *React.js* koristi referentnu jednakost za procenu promene stanja i potrebu za ponovnim ažuriranjem.
- Korišćenje akcija za ažuriranje stanja: Pri upotrebi `useReducer` kuke, preporučuje se upotreba akcija za promenu stanja. Akcija obično predstavlja

objekat koji sadrži polje `type` koje opisuje vrstu akcije, a takođe može sadržati dodatne podatke potrebne za ažuriranje stanja. Kroz korišćenje akcija, omogućava se jasno definisanje svih mogućih načina na koje stanje može biti izmenjeno, olakšavajući razumevanje i održavanje koda.

- Korišćenje `useReducer` je preporučljivo kada je logika za ažuriranje stanja kompleksna. Iako `useReducer` može biti od koristi za rukovanje složenim stanjima njegovo korišćenje može biti nepotrebno za jednostavne situacije. Ako je logika za ažuriranje stanja jednostavna, `useState` bi mogao biti povoljniji izbor. Kuka `useReducer` postiže najveću korist pri postojanju složenih stanja koja obuhvataju više vrednosti ili kada naredno stanje zavisi od prethodnog.

5.2.6 `useContext`

U istoriji razvoja *React.js*-a, upravljanje globalnim stanjem u aplikacijama često je predstavljalo izazov. Pre nego što je *React.js* predstavio *Context API* i `useContext` kuku, upravljanje globalnim stanjem često je zahtevalo upotrebu dodatnih biblioteka kao što je *Redux*. Ove biblioteke su pružale način za skladištenje stanja koje je dostupno svim komponentama u aplikaciji, ali su takođe dodale složenost i dodatni kod.

Uvođenjem *Context API*-ja i `useContext` kuke, *React.js* je omogućio upravljanje globalnim stanjem direktno unutar same biblioteke. Ovo je omogućilo programerima da dele stanje između komponenti bez potrebe za prosleđivanjem *props*-a kroz hijerarhiju komponenti, što je rezultiralo čistijim i lakšim za održavanje kodom.

Ovaj napredak je bio značajan korak napred za *React.js*, jer je omogućio programerima da efikasno upravljaju globalnim stanjem bez potrebe za uključivanjem dodatnih biblioteka. Međutim, *Redux* i dalje ima svoje mesto u ekosistemu *React.js*-a, posebno za složene aplikacije sa naprednim potrebama za upravljanje stanjem.

Kuka `useContext` omogućava pristup vrednosti objekta *Context* iz bilo koje komponente bez potrebe za prosleđivanjem *props*-a kroz hijerarhiju komponenti. Ovo je posebno korisno kada postoji „globalno” stanje koje treba da bude dostupno širom aplikacije, ili kada postoje „duboko” ugneždene komponente koje treba da pristupe istim podacima.

Da bi se koristio `useContext`, prvo je potrebno da se kreira *Context* objekat koristeći `createContext()` funkciju iz `react` paketa. Ova funkcija vraća *Context* objekat koji može da koristi sa `useContext`.

Evo jednostavnog primera kako se `useContext` koristi:

```
import React, { createContext, useContext } from 'react';

// Kreiranje Context objekta
const MyContext = createContext();

function MyComponent() {
  // Koriscenje useContext da se pristupi trenutnoj vrednosti Context-a
  const contextValue = useContext(MyContext);

  return <div>{contextValue}</div>;
}

function MyApp() {
  return (
    // Postavljanje trenutne vrednosti Context-a
    <MyContext.Provider value="Hello, world!">
      <MyComponent />
    </MyContext.Provider>
  );
}

export default MyApp;
```

U ovom primeru, `MyComponent` koristi `useContext` da pristupi trenutnoj vrednosti `MyContext`. Vrednost objekta *Context* se postavlja koristeći `MyContext.Provider`.

Tehnički, `useContext` funkcija prihvata *Context* objekat (vrednost koja se vraća iz `createContext`) i vraća trenutnu vrednost objekta *Context*. Trenutna vrednost objekta *Context* je vrednost koju je najbliži `MyContext.Provider` postavio. Ako ne postoji `MyContext.Provider` u hijerarhiji iznad komponente koja poziva `useContext`, tada će se vratiti početna vrednost koja je prosledena `createContext()`.

Važno je napomenuti da kada *React.js* prikazuje komponentu koja poziva `useContext`, ona će se ponovno prikazati kada se vrednost objekta *Context* promeni. To podrazumeva da je potrebno obratiti pažnju na performanse prilikom korišćenja `useContext`, posebno ukoliko se vrednost objekta *Context* često menja.

Jedan čest slučaj upotrebe `useContext` je za upravljanje temom u aplikaciji. Na primer, može se imati „svetlu” i „tamnu” temu i želi se da korisnici mogu da prebacuju između ove dve teme. Ovo se može postići koristeći `useContext`. Evo

jednostavnog primera:

```
import React, { createContext, useContext, useState } from 'react';

const ThemeContext = createContext('light');

function ThemedButton() {
  const theme = useContext(ThemeContext);

  return (
    <button style={{ background: theme === 'dark' ? '#000' : '#fff' }}>
      Ja sam {theme === 'dark' ? 'tamno' : 'svetlo'} dugme
    </button>;
  );
}

function MyApp() {
  const [theme, setTheme] = useState('light');

  const toggleTheme = () => {
    setTheme(prevTheme => prevTheme === 'light' ? 'dark' : 'light');
  };

  return (
    <ThemeContext.Provider value={theme}>
      <ThemedButton />
      <button onClick={toggleTheme}>Toggle theme</button>
    </ThemeContext.Provider>
  );
}

export default MyApp;
```

U ovom primeru, `ThemedButton` koristi `useContext` da pristupi trenutnoj vrednosti `ThemeContext`. Vrednost objekta *Context* se postavlja koristeći `<ThemeContext.Provider>`. Kada se komponenta `ThemedButton` prikazuje, ona će prikazati dugme koje ima klasu koja odgovara trenutnoj temi. Takođe postoji dugme koje menja temu kada se klikne na njega, što izaziva ponovno prikazivanje `ThemedButton` sa novom temom.

Kada komponenta koristi `useContext` da pristupi *Context*-u, ona će se ponovno prikazati svaki put kada se vrednost objekta *Context* promeni. Ako postoji komponenta koja se često ponovno prikazuje zbog promena u objektu *Context*, to može dovesti do problema sa performansama. Da bi se izbeglo ovo, može se koristiti keširanje ili razdvajanje objekta *Context* na više manjih *Context* objekata tako da komponente mogu da se pretplate samo na delove objekta *Context* koji su im potrebni.

Struktura objekta *Context* može imati veliki uticaj na performanse aplikacije. Ako postoji veliki objekat kao vrednost objekta *Context* i samo jedno polje tog objekta se često menja, sve komponente koje koriste taj *Context* će se ponovno prikazivati svaki put kada se to polje promeni, čak i ako im nije potrebno to polje. Da bi se izbeglo ovo, može se razdvojiti *Context* objekat na više manjih *Context* objekata, tako da svaki *Context* sadrži samo jedno polje.

Iako *Context* može biti veoma moćan alat za upravljanje globalnim stanjem, važno je koristiti ga sa oprezom. Ako se prekomerno koristi *Context*, to može dovesti do nepotrebnih ažuriranja i može učiniti aplikaciju teže za razumevanje i održavanje. Pre nego što se odluči da se koristi *Context*, treba razmotriti da li postoji jednostavniji način za postizanje istog rezultata, kao što je podizanje stanja ili korišćenje propova.

Konačno, važno je napomenuti da je moguće koristiti ekstenzije za veb pregledače, *React.js DevTools*, koja omogućava inspekciju *React.js* komponenti u hijerarhiji stranice u pregledaču. Ova alatka je izuzetno korisna za debugovanje *React.js* aplikacija i može pružiti dublji uvid u to kako se stanje i propovi menjaju tokom vremena. Jedna od ključnih karakteristika *React DevTools*-a je mogućnost da se vidi trenutna vrednost objekta *Context* za odabranu komponentu. Ovo može biti veoma korisno kada se koristi `useContext` kuka, jer omogućava da se tačno vidi koje vrednosti su dostupne komponenti kroz *Context*.

5.3 Pravila korišćenja *React kuka*

React kuke su dizajnirane da se koriste unutar funkcijskih komponenti. Razlog za to je što *React kuke* omogućavaju funkcijskim komponentama da koriste funkcionalnosti koje su prethodno bile dostupne samo klasnim komponentama, kao što su upravljanje stanjem i metode životnog ciklusa (*eng. lifecycle metode*). *React* tim je postavio dva ključna pravila³ koja treba pratiti prilikom korišćenja *React.js*-a:

- **Treba koristiti kuke na vrhu nivoa:** Ne treba da se koriste kuke unutar petlji, uslova ili ugneždenih funkcija. Ovo pravilo obezbeđuje da se kuke pozivaju u istom redosledu svaki put kada se prikaže komponenta. To je ono što omogućava *React.js*-u da pravilno čuva stanje između više prikazivanja.

³Da bi se lakše pridržavali ovih pravila, može se koristiti linter plugin *eslint-plugin-react-hooks* koje je objavio *React* tim, koji automatski proverava kod za kršenje ovih pravila.

- **Treba koristiti kuke samo iz *React* funkcijskih komponenti:** Ne treba pozivati kuke iz običnih *Javascript* funkcija. Umesto toga, može se pozvati kuka iz *React* funkcijskih komponenti. Takođe, može se pozvati *React kuka* iz sopstvenih implementacija kuka.

5.4 Pravljenje sopstvenih *React kuka*

Sopstvena (*eng. custom*) kuka u *React.js*-u je funkcija koja počinje sa „use” i može koristiti druge kuke. Ona nije deo *React API*-ja, već je obrazac koji se pojavljuje iz *React*-ovog kompozicionog modela. Ideja je da ako komponenta ima složenu logiku koja uključuje efekte i stanje, ta logika se može izdvojiti u ponovno upotrebljivu funkciju, tj. sopstvenu kuku.

Pravljenje sopstvenih kuka je slično definisanju bilo koje druge funkcije u *Javascript*-u, ali ime sopstvene kuke treba da počinje sa „use”. Ovo je konvencija koju *React* koristi da bi identifikovao funkcije koje su sopstvene kuke. Jedan slučaj gde bi se mogla upotrebiti sopstvena kuka je u situacijama kada treba omogućiti korisnicima da kopiraju neki tekst na *clipboard* sa jednim klikom, i da se prikaže poruku (ili indikacija) da je tekst uspešno kopiran a zatim nakon nekoliko sekundi da se ta poruka (ili indikacija) ukloni. Umesto da se ova logika implementira u svakoj komponenti gde je potrebno omogućiti kopiranje, može se izdvojiti logika u `useCopy` kuku i ponovno je upotrebiti gde god je potrebno:

```
const useCopy = () => {
  const [copied, setCopied] = useState(false);

  useEffect(() => {
    if (copied) {
      const timeout = setTimeout(() => {
        setCopied(false);
      }, 2000);

      return () => clearTimeout(timeout);
    }
  }, [copied]);

  return [copied, setCopied] as const;
};

export default useCopy;
```

Ova kuka se sada može koristiti u bilo kojoj komponenti koja zahteva funkcionalnost kopiranja teksta. Na primer:

```
import React from 'react';
import useCopy from './useCopy';

const TextComponent = () => {
  const [copied, setCopied] = useCopy();

  const handleCopyClick = () => {
    navigator.clipboard.writeText('Ovo je tekst za kopiranje. ');
    setCopied(true);
  };

  return (
    <div>
      <button onClick={handleCopyClick}>Kopiraj tekst</button>
      {copied && <span>Tekst je uspesno kopiran!</span>}
    </div>
  );
};

export default TextComponent;
```

Kao i kod ugrađenih kuka, pridržavanje *React.js*-ovih smernica i pravila je važno prilikom kreiranja sopstvenih kuka. To uključuje pozivanje kuka na vrhu komponenata i koristeći ih samo unutar *React* funkcijskih komponenti.

5.5 Tranzicija sa klasnih na funkcijske komponente

Kako se *React* razvijao, preporuka za kreiranje komponenti se promenila. Dok su klasne komponente bile standard u ranijim verzijama *React.js*-a, uvođenje *React kuka* u *React* verziji 16.8 je omogućilo funkcijskim komponentama da koriste stanje i druge *React* funkcionalnosti bez potrebe za kreiranjem klase. Ovo je dovelo do preporuke da se za nove komponente koriste funkcijske umesto klasnih komponenata.

Tranzicija sa klasnih na funkcijske komponente može biti postepen proces. *React* tim ne preporučuje da se postojeće klasne komponente prepravljaju u funkcijske[9], osim ako već postoji potreba za refaktorisanjem koda. Umesto toga, preporuka je da se za nove komponente koriste funkcijske komponente.

Kada se prelazi sa klasnih na funkcijske komponente, važno je razumeti kako se koncepti iz klasnih komponenti prenose na funkcijske. Na primer, stanje kompo-

nente, koje se u klasnim komponentama upravlja kroz `this.state` i `this.setState`, u funkcijskim komponentama se upravlja kroz *React kuku* `useState`. Slično tome, metode životnog ciklusa iz klasnih komponentata se mogu simulirati kroz *React kuku* `useEffect` u funkcijskim komponentama.

Glava 6

Rutiranje u *React.js*

6.1 Uvod u rutiranje

Rutiranje predstavlja jedan od ključnih aspekata u razvoju veb aplikacija. Osnovna svrha rutiranja je omogućavanje navigacije između različitih komponenti ili stranica u okviru jedne veb aplikacije. U kontekstu *aplikacija jedne strane (SPA)*, kao što su one koje se razvijaju pomoću *React.js*, rutiranje igra posebno važnu ulogu.

U tradicionalnom veb razvoju, svaka nova stranica koju korisnik poseti rezultira novim zahtevom prema serveru. Međutim, u aplikacijama jedne strane, sve stranice se učitavaju samo jednom, a rutiranje se koristi za dinamičko menjanje sadržaja bez potrebe za ponovnim učitavanjem stranice što dovodi do boljeg korisničkog iskustva.

6.2 Implementacija rutiranja u *React.js*

React.js sam po sebi ne nudi ugrađene mogućnosti za rutiranje. Umesto toga, za ovu svrhu se koriste dodatne biblioteke, od kojih je najpopularnija *React Router*[6]. *React Router* je bogata i fleksibilna biblioteka koja omogućava rutiranje u *React.js* aplikacijama, pružajući brojne mogućnosti za kontrolu navigacije i prikaza komponenti.

U narednim sekcijama biće detaljnije objašnjeno kako se koristi *React Router*, kako se postavlja, kao i različite tehnike i metode koje se mogu koristiti za efikasno rutiranje u *React.js* aplikacijama.

6.3 *React Router*

React Router je dizajniran da se koristi u pregledaču, gde koristi *HTML5 History API* za kontrolu navigacije korisnika. Ova biblioteka pruža skup komponenti koje omogućavaju definisanje ruta i navigaciju između njih kao i različite metode za manipulaciju istorijom pregledača. Najbitnije funkcionalnosti koje pruža će biti detaljnije objašnjene u narednim sekcijama.

6.3.1 Postavljanje rutiranja u aplikaciji

Kako je *React Router* biblioteka koja nije ugrađena u *React.js*, potrebno je prvo instalirati je u aplikaciju. Ovo se može uraditi pomoću *npm* alata, na sledeći način:

```
npm install react-router-dom
```

Sledeći korak je **definisanje ruta** u našoj aplikaciji. Ovo se postiže korišćenjem `createBrowserRouter` funkcije koja prima dva argumenta: niz ruta i opcionalni objekat sa konfiguracijama. Svaka ruta je objekat koji sadrži sledeće ključeve:

- `path`: Niska koja definiše putanju rute.
- `element`: *React.js* komponenta koja se prikazuje kada se poseti ruta.
- `loader`: Opciona funkcija koja se koristi za učitavanje podataka za rutu.
- `children`: Opcioni niz ruta koje predstavljaju pod-rute.

Jedan jednostavan primer definisanja ruta u *React.js* aplikaciji dat je u nastavku:

```
// siteRoutes.jsx
import { createBrowserRouter } from 'react-router-dom';

import HomePage from './pages/HomePage';
import AboutPage from './pages/AboutPage';
import MemberPage from './pages/MemberPage';

export const router = createBrowserRouter([
  {
    path: '/',
    element: <HomePage />,
  },
  {
    path: '/about',
```

```

    element: <AboutPage />,
    children: [
      {
        path: 'member',
        element: <MemberPage />,
      },
    ],
  },
]);

```

```

// App.jsx
import { RouterProvider } from 'react-router-dom';
import { router } from './siteRoutes';

function App() {
  return (
    <div>
      // ... ostatak aplikacije
      <RouterProvider router={router} />
      // ... ostatak aplikacije
    </div>
  );
}

```

U ovom primeru, kreiran je ruter sa dve glavne rute: početna stranica (/) i stranica o nama (/about). Stranica o nama ima pod-rutu member (/about/member). Kada se poseti ruta /, prikazuje se HomePage komponenta, a kada se poseti ruta /about/member, prikazuje se MemberPage komponenta. Na kraju, koristi se RouterProvider komponenta unutar App komponente za pružanje rutera svim komponentama u aplikaciji.

Kao alternativu ovome, opcija je korišćenje createRoutesFromElements ukoliko je preferenca da se rute definišu kao *JSX* umesto niza objekata. To se ostvaruje kroz upotrebu <Route> komponente koja prihvata path i element *props*. Takođe, <Route> komponenta može biti iskorišćena za **definisane pod-rute**. Primer upotrebe je prikazan u nastavku:

```

// siteRoutes.jsx
import { createBrowserRouter, createRoutesFromElements, Navigate, Route } from
  'react-router-dom';

import HomePage from './pages/HomePage';
import AboutPage from './pages/AboutPage';
import MemberPage from './pages/MemberPage';

export const router = createBrowserRouter(
  createRoutesFromElements(
    <React.Fragment>

```

```
    <Route path="/" element={<HomePage />} />,
    <Route path="/about" element={<AboutPage />} />,
    <Route path="member" element={<MemberPage />} />,
  </Route>
</React.Fragment>
)
);
```

```
// App.jsx
import { RouterProvider } from 'react-router-dom';
import { router } from './siteRoutes';

function App() {
  return (
    <div>
      // ... ostatak aplikacije
      <RouterProvider router={router} />
      // ... ostatak aplikacije
    </div>
  );
}
```

Alternative za BrowserRouter

Pored `createBrowserRouter`, *React Router* v6 nudi i druge funkcije za kreiranje rutera, uključujući `createHashRouter` i `createMemoryRouter`.

Funkcija `createHashRouter` je korisna ako nije moguće konfigurisati veb server da usmerava sav saobraćaj ka *React Router* aplikaciji. Umesto korišćenja normalnih *URL*-ova, koristiće se deo *URL*-a nakon znaka hash (`#`) za upravljanje „aplikacijskim *URL*-om”. Ova funkcija se koristi identično kao i `createBrowserRouter`. Napomenimo još da je korišćenje hash *URL*-ova generalno loša praksa, ali je korisno znati da postoji ova alternativa.

S druge strane, `createMemoryRouter` upravlja sopstvenim *stack*-om istorije u memoriji, umesto da koristi istoriju pregledača. Ovo je posebno korisno za testiranje i alate za razvoj komponenti poput Storybook-a¹, ali se može koristiti i za pokretanje *React Router*-a u bilo kom okruženju koje nije pregledač.

Ove alternative pružaju fleksibilnost prilikom kreiranja rutera u zavisnosti od specifičnih potreba aplikacije ili okruženja.

¹Storybook je popularan alat za razvoj korisničkih interfejsa koji omogućava izolovano kreiranje i testiranje komponenti u okruženju nezavisnom od glavne aplikacije.

6.3.2 Prikazivanje dece unutar roditeljske komponente

Ako bismo želeli da se `MemberPage` komponenta **prikazuje unutar** `AboutPage` komponente, potrebno je da se u `AboutPage` komponenti koristi `Outlet` komponenta. Komponenta `Outlet` se koristi za prikazivanje pod-ruta. Primer korišćenja `Outlet` komponente dat je u nastavku:

```
// AboutPage.jsx
import * as React from 'react';
import { Outlet } from 'react-router-dom';

function AboutPage() {
  return (
    <div>
      <h1>About Page</h1>
      <Outlet />
    </div>
  );
}

export default AboutPage;
```

```
// MemberPage.jsx
import * as React from 'react';

function MemberPage() {
  return (
    <div>
      <h1>Member Page</h1>
    </div>
  );
}
```

U ovom primeru, `MemberPage` komponenta će se prikazivati unutar `AboutPage` komponente kada se poseti ruta `/about/member`. *HTML* kod koji će se generisati u ovom slučaju je:

```
<div>
  <h1>About Page</h1>
  <div>
    <h1>Member Page</h1>
  </div>
</div>
```

6.3.3 Rutiranje sa parametrima

Parametri rutiranja predstavljaju ključni deo dinamičkog rutiranja u *React Router*-u. Omogućavaju tretiranje dela *URL*-a kao promenljive, što doprinosi kreiranju fleksibilnijih navigacionih struktura.

Definisanje parametra rutiranja se postiže korišćenjem dvotačke (:) u putanji rute. Na primer, ruta za profil korisnika može se definisati na sledeći način:

```
{
  path: '/user/:id',
  element: <UserPage />,
}
```

U ovom slučaju, `:id` predstavlja parametar rutiranja. Kada se poseti *URL* poput `/user/123`, *React Router* mapira ovu rutu na `UserPage` komponentu, a `123` postaje vrednost `id` parametra.

Za pristup vrednosti parametra rutiranja unutar *React* komponente, koristi se `useParams` kuka iz *React Router*-a. Na primer u `UserPage` komponenti koja prikazuje profil korisnika, `useParams` kuka može se koristiti na sledeći način:

```
import { useParams } from 'react-router-dom';

function UserPage() {
  const { id } = useParams();
  // ...
}
```

U ovom slučaju, `useParams` vraća objekat sa svim parametrima rutiranja. Kada se poseti `/user/123`, `id` će biti `123`.

Parametri rutiranja su izuzetno korisni za kreiranje dinamičkih ruta, kao što su stranice profila korisnika, stranice proizvoda, itd. Omogućavaju da se ista komponenta koristi za prikazivanje različitih sadržaja, u zavisnosti od parametra u *URL*-u.

Opcioni dinamički parametri

Opcioni dinamički parametri u *React Router*-u omogućavaju tretiranje dela puta kao promenljive, ali je taj deo puta opcion. To znači da ruta može da se poklapa sa *URL*-om bez obzira da li je parametar prisutan ili ne.

Definisanje opcionog dinamičkog parametra se postiže korišćenjem dvotačke (:) i upitnika (?) u putanji rute. Na primer, rutu za profil korisnika možemo definisati tako da ako ne postoji parametar `id`, onda se prikazuje stranica o profilu trenutno

prijavljenog korisnika, a ako postoji, onda se prikazuje stranica o profilu korisnika sa datim id-om. To se postiže na sledeći način:

```
{
  path: '/user/:id?',
  element: <UserPage />,
}
```

U ovom slučaju, `:id?` predstavlja opcioni dinamički parametar. Kada se poseti URL poput `/user/123` ili samo `/user`, *React Router* mapira ovu rutu na `UserPage` komponentu. Ako je 123 prisutan, to će biti vrednost `id` parametra. Ako nije, `id` će biti `undefined`:

```
import { useParams } from 'react-router-dom';

function UserPage() {
  const { id } = useParams();

  const userId = id ? id : 'current';

  const [userData, setUserData] = useState(null);

  useEffect(() => {
    fetchUser(userId)
      .then((data) => setUserData(data));
  }, [userId]);

  // ...
}
```

U ovom slučaju, `useParams` vraća objekat sa svim parametrima rutiranja. Ako se poseti `/user/123`, `id` će biti 123. Ako se poseti samo `/user`, `id` će biti `undefined`.

Opcioni dinamički parametri su korisni kada želimo da ista ruta pokriva više različitih URL-ova, u zavisnosti od toga da li je određen parametar prisutan ili ne.

6.3.4 Navigacija i redirekcija

Navigacija i redirekcija su dve ključne funkcionalnosti koje *React Router* pruža za upravljanje prelascima između različitih stranica u aplikaciji.

Za **navigaciju između ruta**, *React Router* pruža `useNavigate` kuku. Ova kuka vraća funkciju koja se može koristiti za navigaciju do različitih ruta i ona je ključna za programsku navigaciju u aplikaciji. Ova funkcija omogućava prelazak na drugu rutu bez potrebe za korišćenjem komponente `Link` ili direktnim menjanjem

URL-a pomoću `window.location` iz globalnog objekta `window`. Na primer, funkcija `navigate` može se koristiti na sledeći način:

```
import { useNavigate } from 'react-router-dom';

function NavigationComponent() {
  const navigate = useNavigate();

  return (
    <button onClick={() => navigate('/about')}>
      Idi na About stranicu
    </button>
  );
}
```

Funkcija `navigate` prima jedan neophodan i jedan opcioni argument. Prvi argument je putanja do koje želite da navigirate. Ova putanja može biti apsolutna (npr. `/about`) ili relativna (npr. `about`). Ako je putanja relativna, ona će biti rešena u odnosu na trenutnu rutu. Možemo koristiti `../` za navigaciju do roditeljske rute ili `/` za navigaciju do korenske rute.

Drugi argument je opcioni objekat sa opcijama za navigaciju. Ovaj objekat može sadržati sledeća polja:

- `replace`: Ako je postavljeno na `true`, trenutna ruta će biti zamenjena novom rutom u istoriji pregledača, umesto da se doda nova stavka u istoriju.
- `state`: Ovo je proizvoljni objekat koji možete proslediti novoj ruti. Ovaj objekat će biti dostupan u polju `state` od `location` objekta koji se dobije korišćenjem `useLocation` kuke. Ovo je korisno za prosleđivanje dodatnih informacija o stanju aplikacije kada se prelazi na novu rutu.
- `relative`: Ovo je bubovska vrednost koja određuje da li će putanja biti rešena relativno u odnosu na trenutnu rutu.

Evo jednog malo složenijeg primera koji koristi ove opcije:

```
// Home.jsx
import { useNavigate } from 'react-router-dom';

function HomePage() {
  const navigate = useNavigate();

  return (
    <div>
```

```

        <button onClick={() => navigate('../about', { replace: true,
            state: { from: 'home' } })}>
            Idi na About stranicu
        </button>
    </div>
);
}

export default HomePage;

```

```

// About.jsx
function AboutPage() {
    const location = useLocation();
    const { from } = location.state;

    return (
        <div>
            <p>{from ? `Dosao si sa ${from} stranice` : 'Dosao si direktno na
                ovu stranicu'}</p>
        </div>
    );
}

export default AboutPage;

```

U ovom primeru, kada se klikne na dugme, korisnik će biti preusmeren na stranicu AboutPage i trenutna ruta će biti zamenjena novom rutom u istoriji pregledača. Takođe, u state objektu koji se prosleđuje novoj ruti, biće prosleđena informacija o tome da li je korisnik došao na stranicu AboutPage sa stranice HomePage.

Zaštita ruta redirekcijom

Zaštita ruta predstavlja ključnu funkcionalnost u većini veb aplikacija. Ovo je posebno važno za aplikacije koje sadrže osetljive podatke ili funkcionalnosti dostupne samo autentifikovanim korisnicima.

Na primer, razmatrajući veb aplikaciju za upravljanje članstvom u teretani, može postojati stranica članova (MemberPage) na kojoj članovi mogu videti informacije o svom članstvu, rezervisati treninge, pratiti svoj napredak i slično. Jasno je da ove informacije i funkcionalnosti treba da budu dostupne samo autentifikovanim članovima.

Za zaštitu MemberPage, moguće je koristiti useEffect i useNavigate kuke u kombinaciji. Kada se MemberPage komponenta prikazuje, useEffect proverava da

li je korisnik autentifikovan. Ako korisnik nije autentifikovan, `navigate` funkcija ga preusmerava na stranicu za prijavu (`LoginPage`):

```
import { useEffect } from 'react';
import { useNavigate } from 'react-router-dom';

function MemberPage() {
  const navigate = useNavigate();
  const isAuthenticated = // logika za proveru da li je korisnik
    autentifikovan

  useEffect(() => {
    if (!isAuthenticated) {
      navigate('/login');
    }
  }, [navigate, isAuthenticated]);

  return (
    // ...
  );
}
```

Ovaj pristup osigurava da samo autentifikovani korisnici mogu pristupiti `MemberPage`. Ako neautentifikovani korisnik pokuša da pristupi ovoj stranici, biće preusmeren na `LoginPage`. Ovo ne samo da štiti osetljive podatke i funkcionalnosti od neautorizovanog pristupa, već i poboljšava korisničko iskustvo, jer korisnici su jasno usmereni ka stranici za prijavu kada pokušaju da pristupe funkcionalnostima koje zahtevaju autentifikaciju.

Korisničko iskustvo se može još više poboljšati tako što bi se dodao argument koji se na primer zove `redirectedFrom` prilikom preusmeravanja korisnika na stranicu za prijavu. Ovaj argument može se kasnije koristiti za vraćanje korisnika na stranicu sa koje je preusmeren nakon uspešne prijave:

```
import { useEffect } from 'react';
import { useNavigate, useLocation } from 'react-router-dom';

function MemberPage() {
  const navigate = useNavigate();
  const location = useLocation();
  const isAuthenticated = // logika za proveru autentifikacije
  const state = { redirectedFrom: location.pathname };

  useEffect(() => {
    if (!isAuthenticated) navigate('/login', { state });
  }, [navigate, isAuthenticated, location.pathname]);
}
```

```
    return // ...  
  }
```

U ovom primeru, koristi se `useLocation` kukica da se dobije trenutna putanja iz polja `pathname` i prosledi kao `redirectedFrom` u `state` objektu kada se korisnik preusmerava na stranicu za prijavu.

Na stranici za prijavu, moguće je koristiti `useLocation` da se pristupi `redirectedFrom` putanji i `useNavigate` da se preusmeri korisnik na tu putanju nakon uspešne prijave:

```
import { useEffect } from 'react';  
import { useNavigate, useLocation } from 'react-router-dom';  
  
function LoginPage() {  
  const navigate = useNavigate();  
  const location = useLocation();  
  const isAuthenticated = // logika za proveru da li je korisnik  
    autentifikovan  
  
  useEffect(() => {  
    if (isAuthenticated) {  
      const redirectedFrom = location.state?.redirectedFrom || '/default';  
      navigate(redirectedFrom);  
    }  
  }, [navigate, isAuthenticated, location.state]);  
  
  return (  
    // ...  
  );  
}
```

U ovom primeru, kada se korisnik uspešno prijavi, koristi se `navigate` funkcija da se preusmeri na putanju iz `redirectedFrom` ili na podrazumevanu putanju ako `redirectedFrom` ne postoji.

6.3.5 Lenjo učitavanje

Lenjo učitavanje je tehnika koja može poboljšati performanse aplikacije tako što odlaže učitavanje nekih delova aplikacije dok nisu potrebni. U kontekstu *React Router*-a, ovo se obično odnosi na odgađanje učitavanja komponenti ruta dok ruta nije posetila.

React ima ugrađenu podršku za lenjo učitavanje kroz `React.lazy` i `Suspense`. `React.lazy` omogućava da se dinamički učita komponenta kao funkcija koja vraća

Promise. *Suspense* omogućava da se prikaže neki rezervni sadržaj dok se *Lazy* komponenta učitava.

Koristeći `createBrowserRouter` i `createRoutesFromElements` u kombinaciji sa `React.lazy` i `Suspense`, možemo implementirati lenjo učitavanje za rute na sledeći način:

```
import * as React from 'react';
import * as ReactDOM from 'react-dom';

import { createBrowserRouter, createRoutesFromElements, RouterProvider,
  Navigate, Route } from 'react-router-dom';

const HomePage = React.lazy(() => import('./pages/HomePage'));
const AboutPage = React.lazy(() => import('./pages/AboutPage'));
const MemberPage = React.lazy(() => import('./pages/MemberPage'));

const router = createBrowserRouter(
  createRoutesFromElements(
    <React.Fragment>
      <Route path="/" element={<HomePage />} />,
      <Route path="/about" element={<AboutPage />} />,
      <Route path="/member" element={<MemberPage />} />,
    </Route>
  </React.Fragment>
)
);

function App() {
  return (
    <div>
      // ... ostatak aplikacije
      <React.Suspense fallback={<div>Ucitavanje...</div>}>
        <RouterProvider router={router} />
      </React.Suspense>
      // ... ostatak aplikacije
    </div>
  );
}

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

U ovom primeru, `HomePage`, `AboutPage` i `MemberPage` komponente se dinamički učitavaju kada se pristupi odgovarajućoj ruti. Ovo znači da će se `HomePage` komponenta učitati kada se pristupi `/`, `AboutPage` komponenta kada se pristupi `/about` i `MemberPage` komponenta kada se pristupi `/about/member`.

`Suspense` komponenta prikazuje sadržaj zadat atributom `fallback` dok se *lenje komponente* učitavaju. U ovom primeru, prikazuje se `Ucitavanje...` dok se

učitava HomePage, AboutPage ili MemberPage komponenta.

Ova tehnika može značajno poboljšati performanse aplikacije, posebno za velike aplikacije sa mnogo ruta. Učitavanjem samo onih komponenti koje su potrebne za trenutnu rutu, smanjuje se količina koda koja se mora inicijalno preuzeti, što rezultira bržim vremenima učitavanja.

Glava 7

Podrška za *TypeScript*

7.1 Kako integrisati *TypeScript* u *React.js* projekat

S obzirom da je *React.js* napisan u *Javascript*-u, on nema podršku za tipiziranje i automatsko dovršavanje (*eng. autocomplete*) koje pružaju statičko tipizirani jezici. Ovde na scenu stupa *TypeScript*, statičko tipizirani nadskup *Javascript*-a.

Pretpostavka je da je čitaoc već upoznat sa osnovama *TypeScript*-a, stoga osnovni koncepti i funkcionalnosti *TypeScript*-a neće biti detaljno objašnjavani. Umesto toga, fokus će biti na tome kako se *TypeScript* koristi u kontekstu *React.js*, i kako se može integrisati u *React.js* projekat. Kao i van *React.js*, *TypeScript* i u *React*-u donosi prednosti statičke tipizacije *Javascript*-a, uključujući poboljšane alate za veće baze koda, bolju saradnju između programera kao i veću robusnost i čitljivost koda. Međutim, važno je napomenuti da *React.js* ne podržava *TypeScript* *nativno*. To znači da ne možete jednostavno početi pisati *TypeScript* kod u *React.js* projektu bez dodatne postavke. Važnu ulogu u integraciji *TypeScript*-a u *React.js* igra i tim volontera, poznat kao *DefinitelyTyped* tim, koji održava i ažurira *TypeScript* definicije za širok spektar *Javascript* biblioteka, uključujući i *React.js*¹.

Čak i sa ovim smetnjama, integracija *TypeScript*-a u *React.js* projekat je olakšana kroz upotrebu određenih alata koji podržavaju *TypeScript* „iz kutije” (*eng. out of the box*). Jedan takav alat je *Create React App*, alat za komandnu liniju (*eng. command-line (CLI) tool*) za kreiranje novih *React.js* aplikacija, koja ima

¹*DefinitelyTyped* je repozitorijum na *GitHub*-u koji sadrži visoko kvalitetne *TypeScript* definicije. Ove definicije omogućavaju *TypeScript*-u da razume strukturu postojećih *Javascript* biblioteka, kao što je *React.js*.

ugrađenu podršku za *TypeScript*. Korišćenjem *Create React App* sa *TypeScript* šablonom, programeri mogu brzo postaviti novi *React.js* projekat koji podržava *TypeScript*.

Još jedan alat koji olakšava integraciju *TypeScript*-a u *React.js* projekat je *Vite*, jedan od novijih alata za izgradnju *frontend* dela veb aplikacija, koji takođe nudi podršku za *TypeScript* „iz kutije”. *Vite* pruža lakšu, bržu i fleksibilniju razvojnu okolinu za moderne veb projekte²

U ovom poglavlju detaljnije će se razmotriti korišćenje *TypeScript*-a u kontekstu *React*-a. Biće istaknute razlike između *JSX* i *TSX*, objašnjen način definisanja *React.js* komponenti pomoću *TypeScript*-a, koncept generičkih komponenti u *TypeScript*-u i upotreba *TypeScript*-a sa kukama. Takođe će se razmotriti najbolje prakse za upotrebu *TypeScript*-a u projektima zasnovanim na *React.js*-u i kako se suočiti sa uobičajenim izazovima i problemima.

7.2 *TypeScript* podrška za komponente i props

Kada se definiše *React.js* komponenta u *TypeScript*-u, može se koristiti interfejs ili tip da definiše oblik *props*-a koje komponenta očekuje. Zamislimo da se razvija korisnički interfejs za *e-commerce* platformu. Postoji komponenta *ProductCard* koja prikazuje informacije o proizvodu. Ova komponenta očekuje prop *product* koji je objekat sa svojstvima *id*, *productName*, *description*, *price*, i *imageUrl*.

U *TypeScript*-u, može se definisati interfejs za *product* i koristiti ovaj interfejs da se tipiziraju *props* za *ProductCard* komponentu:

```
interface Product {
  id: number;
  productName: string;
  description: string;
  price: number;
  imageUrl: string;
}

interface IProductCardProps {
  product: Product;
}
```

```
const ProductCard: React.FC<IProductCardProps> = ({ product }) => {
  return (
```

²O *Create React App* i *Vite* će biti detaljnije diskutovano u poglavlju 8, gde će se istražiti kako ovi alati pomažu u postavljanju i upravljanju procesom izgradnje *React.js* projekata.

```

<div>
  <img src={product.imageUrl} alt={product.productName} />
  <h2>{product.productName}</h2>
  <p>{product.description}</p>
  <p>{product.price} USD</p>
</div>
);
};

```

Ako se pokuša koristiti ova komponenta bez prosleđivanja `product prop`-a, *TypeScript* će prijaviti grešku. Na primer, sledeći kod će izazvati grešku:

```
<ProductCard /> // Greska: fali prop 'product'
```

Slično tome, ako se pokuša proslediti `product prop` koji nije pravilnog tipa, *TypeScript* će takođe prijaviti grešku. Na primer, sledeći kod će izazvati grešku:

```
<ProductCard product={{ id: 1, productName: "Product 1", price: 20 }} /> //
Greska: fali prop 'description' i 'imageUrl'
```

U ovom primeru, `product prop` nema `description` i `imageUrl` svojstva, što je u suprotnosti sa `Product` interfejsom koji je definisan.

Ova upozorenja su veoma korisna jer omogućavaju otkrivanje i ispravljanje grešaka pre nego što kod dođe do produkcije. Takođe, mnogi moderni editori koda (*eng. code editor*) imaju integraciju sa *TypeScript*-om i mogu prikazati ova upozorenja direktno u *editor*-u, što dodatno poboljšava iskustvo razvoja.

U ovom primeru, ako se pokuša koristiti `ProductCard` komponenta sa `product prop`-om koji ne odgovara `Product` interfejsu, *TypeScript* će prijaviti grešku. Ovo može pomoći u otkrivanju i ispravljanju grešaka pre nego što kod dođe do produkcije.

Ovo je samo jedan primer kako *TypeScript* može poboljšati sigurnost tipova i kvalitet koda u *React.js* projektima. Postoje mnoge druge situacije u kojima *TypeScript* može biti koristan, uključujući tipizaciju *state*-a, *context*-a, i sopstvenih kuka.

Neke od preporuka za korišćenje *TypeScript*-a u *React.js* projektima su:

- **Interfejsi ili tipovi:** I interfejsi i tipovi mogu se koristiti za definisanje oblika *props*-a u *React* komponentama. Generalno u praksi, interfejsi se najčešće koriste za definisanje oblika *props*-a komponenti kao i objekata koji se koriste u komponentama, dok se tipovi koriste za definisanje oblika vrednosti koje nije moguće predstaviti kao objekat, kao što primitive vrednosti, nizovi, i

funkcije i pomoćni tipovi kao što su `Partial<T>`, `Pick<T, K>` koji vraćaju novi tip na osnovu postojećeg tipa ili interfejsa.

- **Imenovanje interfejsa za *props*:** Konvencija je da se interfejsi za *props* završavaju sa „*Props*”, a počinju sa „*I*” da bi se izbegla konfuzija. Na primer, interfejs za *props* za `ProductCard` komponentu može se nazvati `IProductCardProps`. Ova konvencija pomaže u jasnom razlikovanju interfejsa za *props* od drugih interfejsa i tipova.
- **Korišćenje `React.FC<T>` ili direktno tipiziranje *props*-a:** `React.FC<T>` je generički tip koji predstavlja funkcijsku komponentu koja očekuje *props* definisane tipom `T`. Korišćenje `React.FC<T>` automatski dodaje `children` u tip *props*-a, čak i ako komponenta ne koristi `children`. S druge strane, direktno tipiziranje *props*-a pruža više kontrole nad kojim *props*-ima vaša komponenta očekuje. U praksi, izbor između `React.FC<T>` i direktnog tipiziranja *props*-a je sličan kao izbor interfejsa ili tipova i često zavisi od ličnih preferenci i konvencija tima.

7.2.1 Generičke komponente

Generičke komponente su moćan koncept u *TypeScript*-u koji omogućava definisanje komponenti koje mogu raditi sa različitim tipovima podataka. Ovo je posebno korisno kada se želi kreirati komponenta koja može da prihvati različite tipove *props*-a, ali i dalje želi da očuva sigurnost i automatsko zaključivanje tipova.

U nastavku, biće napravljena generička `select` komponentu koja prikazuje padajuću listu sa opcijama sa labelom. Ova komponenta će prihvatati `options` prop koji je niz objekata sa `value` i `label` svojstvima. Takođe, komponenta će prihvatati `onChange` prop koji je funkcija koja se poziva kada se izabere opcija. Ova komponenta će biti generička tako da može raditi sa različitim tipovima podataka za `value` *prop*:

```
// CustomSelect.tsx
import React, { ChangeEvent } from 'react';

interface ICustomSelectProps<T> {
  label: string;
  value: T;
  options: Array<{ value: T; label: string }>;
  onChange: (value: T) => void;
}
```



```

function CustomSelect<T>({
  label,
  value,
  options,
  onChange,
}: CustomSelectProps<T>) {
  const handleSelectChange = (event: ChangeEvent<HTMLSelectElement>) => {
    const selectedValue = event.target.value as T;
    onChange(selectedValue);
  };

  return (
    <div>
      <label>{label}</label>
      <select value={value} onChange={handleSelectChange}>
        {options.map((option) => (
          <option key={option.label} value={option.value}>
            {option.label}
          </option>
        ))}
      </select>
    </div>
  );
}

export default CustomSelect;

// Primer upotrebe u drugoj komponenti

import React, { useState } from 'react';
import CustomSelect from './CustomSelect';

interface Color {
  id: number;
  productName: string;
}

const colors: Color[] = [
  { id: 1, colorName: 'Red' },
  { id: 2, colorName: 'Green' },
  { id: 3, colorName: 'Blue' },
];

function App() {
  const [selectedColor, setSelectedColor] = React.useState<Color | null>(null)
  ;

  return (
    <div>
      <CustomSelect

```

```

    label="Choose a color"
    value={selectedColor}
    options={colors => colors.map(color => ({ value: color, label:
        color.colorName })))
    onChange={({color}) => setSelectedColor(color)} // Dovoljno je samo
        proslediti setSelectedColor
  />
</div>
)

```

S obzirom na to da nizu `colors` nedostaje svojstvo `label` kako bi se uklopio u tip `options prop`-a, potrebno je izvršiti mapiranje u niz objekata sa svojstvima `value` i `label`, jer je to očekivani format za ovu komponentu. Nakon što je izmenjeni niz `colors` prosleđen kao `options prop`, *TypeScript* automatski zaključuje da je tip argumenta `color` u funkciji `onChange` sada `Color`. Ovo se događa jer je definisan interfejs `ICustomSelectProps` kao `CustomSelectProps<T>` i `Color` je korišćen kao tip argumenta `T`. Ovaj koncept pruža moćnu mogućnost da se kreiraju komponente koje mogu operisati sa različitim tipovima podataka, istovremeno zadržavajući sigurnost i automatsko zaključivanje tipova.

Pri korišćenju komponente u drugim komponentama, može se i eksplicitno navesti tip argumenta `T`:

```

<CustomSelect<Color>
  label="Choose a color"
  value={selectedColor}
  options={colors => colors.map(color => ({ value: color, label: color.name }
    )}
  onChange={({color}) => setSelectedColor(color)}
/>

```

7.3 Primena *TypeScript*-a u *JSX*: Pregled *TSX*-a

TSX, *TypeScript* XML, je ekvivalent *JSX*-a za *TypeScript*. *TSX* omogućava pisanje sintaksi sličnoj HTML-u direktno u *TypeScript* kodu, ali sa dodatnim prednostima koje *TypeScript* pruža. Na primer, ako se pretpostavi da postoji jednostavna komponenta koja prikazuje listu korisnika gde svaki korisnik ima `id`, `name`, i `email`, u *TSX*-u, se može definisati tip za korisnika i koristiti ovaj tip direktno u *TSX* sintaksi:

```

type User = {

```

```

    id: number;
    name: string;
    email: string;
};

type UserListProps = {
  users: User[];
};

function UserList({ users }: UserListProps) {
  return (
    <ul>
      {users.map((user: User) => ( // Iako nije neophodno, ovde je
        eksplicitno naveden tip user-a
        <li key={user.id}>
          {user.name} ({user.email})
        </li>
      ))}
    </ul>
  );
}

export default UserList;

```

Kada se koristi *TypeScript* u *React.js* projektu, važno je napomenuti da fajlovi koji sadrže *TSX* sintaksu imaju ekstenziju `.tsx`. Ovo nije slučaj sa *JSX*-om, gde fajlovi koji sadrže *JSX* sintaksu mogu imati ekstenziju `.jsx` ili `.js` (ali je `.jsx` ekstenzija često korisna za razlikovanje fajlova koji koriste *JSX* sintaksu).

Razlog zašto je ovo važno je taj što *TypeScript* kompajler koristi ekstenziju fajla da bi odredio kako da tretira sadržaj fajla. Kada *TypeScript* kompajler vidi fajl sa ekstenzijom `.tsx`, on zna da treba da očekuje *TSX* sintaksu u tom fajlu, i da treba da proveri tipove u skladu sa tim.

Ako se pokuša koristiti *TSX* sintaksa u fajlu sa `.ts` ekstenzijom, *TypeScript* kompajler će prijaviti grešku, jer ne očekuje da vidi *JSX* sintaksu u standardnom *TypeScript* fajlu.

Stoga, kada se radi sa *TypeScript*-om i *React.js*, važno je uvek koristiti `.tsx` ekstenziju za fajlove koji sadrže *TSX* sintaksu. Ovo će osigurati da kod bude pravilno tretiran i kompajliran, te da se iskoriste sve prednosti koje *TypeScript* pruža kada se koristi sa *React.js*.

Napomenimo još da ako se koriste lambda funkcije u *TypeScript*-u za definisanje generičkih funkcija unutar `.tsx` fajla, potrebno je dodati zarez posle generičkog tipa. Ova je neophodno kako bi *TypeScript* parser mogao pravilno da razlikuje *JSX* elemente od generičkih tipova. Bez zareza, *TypeScript* bi mogao pogrešno da

protumači `<T>` kao *JSX* element. Komponentu `CustomSelect` iz prethodne sekcije bi mogli definisali i na sledeći način:

```
const CustomSelect = <T,>({
  label,
  value,
  options,
  onChange,
}: CustomSelectProps<T>) => {
  // ...
};
```

Glava 8

Alati za pakovanje modula: *Webpack* i *Vite*

8.1 Uvod u alate za pakovanje modula

U ranim danima *React.js*-a, postavljanje novog projekta bilo je prilično složeno. Jedan od problema je što veb pregledači ne mogu direktno da interpretiraju *JSX*, pa je potrebno koristiti alate za pakovanje modula (*eng. bundlers*) kao što su *Webpack* ili *Vite* da se *JSX* transpiluje¹ u običan *Javascript*.

Pored toga, *React.js* aplikacije često koriste *ES6* module za organizaciju koda, ali ni oni nisu direktno podržani u svim veb pregledačima (niti starijim verzijama *node.js*-a). Takođe, *React.js* aplikacije često koriste *CSS* module, slike, i druge resurse, koji takođe moraju da se obrade pre nego što se mogu koristiti u veb pregledaču. Sve ovo znači da je postavljanje novog *React.js* projekta bilo prilično složeno i zahtevalo je dosta konfiguracije.

Ovde na scenu stupaju alati za pakovanje modula koji su olakšali ovaj proces na nekoliko načina. Prvo, oni su omogućili automatizaciju procesa transpilacije *JSX*-a i *ES6* modula, kao i uključivanje *CSS*-a, slika, i drugih resursa. Drugo, oni su omogućili optimizaciju koda za produkcionu *build*, uključujući minifikaciju i kompresiju koda, razdvajanje koda (*eng. code splitting*), i lenjo učitavanje (*eng. lazy loading*). Treće, oni su omogućili napredne funkcije za razvoj, kao što je dinamička zamena modula uživo (*eng. hot module replacement (HMR)*).

¹Transpilacija je proces konverzije koda napisanog u jednom programskom jeziku u ekvivalentan ili skoro ekvivalentan kod u drugom programskom jeziku. U kontekstu *Javascript*-a, transpilacija se često koristi za konverziju novijih funkcija *Javascript*-a koje nisu podržane u svim pregledačima (kao što su *ES6* moduli ili *JSX*) u starije verzije *Javascript*-a koje su široko podržane.

Danas, alatke kao što su *Create React App*, *Next.js*, i *Gatsby.js* koriste *Webpack* ili *Vite* da bi olakšale postavljanje novih *React.js* projekata. Ove alatke dolaze sa prethodno konfigurisanim alatima za pakovanje modula, tako da programeri mogu da počnu da pišu *React.js* kod bez potrebe za ručnim konfigurisanjem ovih alatki.

8.2 *Webpack*

Webpack se instalira kao lokalna zavisnost u projektu koristeći `npm`. Nakon instalacije, konfiguracija *Webpack*-a se obično vrši putem `webpack.config.js` fajla. Ovaj fajl može da sadrži različite konfiguracije za različite aspekte vašeg projekta, uključujući ulazne i izlazne putanje, *loader*-i, *plugin*-ove, i druge opcije.

Loader-i u *Webpack*-u se koriste za obradu različitih tipova fajlova i pretvaraju ih u validne module koje *Webpack* može da obradi. Na primer, *babel-loader* se koristi za transpilaciju *Javascript* koda, dok *css-loader* omogućava uvoz *CSS*-a direktno u *Javascript*. *Plugin*-ovi pružaju širok spektar funkcionalnosti i koriste se za sve, od optimizacije *bundle*-a do ubrizgavanja promenljivih okruženja (*eng. environment variables*) u kod.

Webpack takođe podržava i **razdvajanje koda** koje omogućava da se aplikacija podeli na manje delove (*eng. chunks*) koji se mogu učitavati i izvršavati asinhrono. Ovo može da poboljša performanse aplikacije tako što će korisnici preuzimati samo kod koji im je potreban u datom trenutku. **Lenjo učitavanje** je sličan koncept, ali se odnosi na asinhrono učitavanje funkcionalnosti aplikacije samo kada je to potrebno. Ovo može da uključi učitavanje komponenti na zahtev, ili učitavanje velikih biblioteka samo kada su potrebne.

Jedna od najkorisnijih funkcionalnosti *Webpack*-a je *HMR* koja omogućava da se **izmene u modulima primene uživo**, bez potrebe za punim osvežavanjem stranice. Ovo može da ubrza razvoj tako što omogućava programerima da vide efekte izmena koda odmah.

8.2.1 Primer korišćenja *Webpack*-a u *React.js* aplikaciji

Da bi se dodatno ilustrovala upotreba *Webpack*-a, biće razmotren primer jednostavne *React.js* aplikacije. U ovom primeru, *Webpack* će se koristiti za pakovanje *React.js* aplikacije, dok će *Babel* biti korišćen za transpilaciju *JSX* koda u običan *Javascript*.

Prvi korak je inicijalizacija projekta korišćenjem komande `npm init` u terminalu. Ovo će kreirati `package.json` fajl koji će sadržati informacije o projektu i njegovim zavisnostima. Nakon toga, potrebno je instalirati *Webpack* i *Webpack CLI* alat globalno da bi se mogla koristiti `webpack` komanda, kao i potrebnih *React* i *Babel* paketa:

```
npm install -g webpack webpack-cli
npm install react react-dom
npm install -D @babel/core
npm install -D @babel/preset-env @babel/preset-react babel-loader
```

Koristi se opcija `-D` da se označi da su ovi paketi instalirani kao razvojne zavisnosti, što znači da će biti instalirani samo u razvojnom okruženju i neće biti uključeni u finalni paket.

Dalje, konfiguracija *Webpack*-a se vrši putem `webpack.config.js` fajla. Osnovna konfiguracija može izgledati ovako:

```
module.exports = {
  entry: './src/index.js',
  output: {
    filename: 'bundle.js',
    path: __dirname + '/dist'
  },
  module: {
    rules: [
      {
        test: /\.js$/,
        exclude: /node_modules/,
        use: {
          loader: 'babel-loader',
          options: {
            presets: ['@babel/preset-env', '@babel/preset-react']
          }
        }
      }
    ]
  }
};
```

U `src/index.js` fajlu, može se kreirati jednostavna *React* komponenta:

```
import React from 'react';
import ReactDOM from 'react-dom';

const App = () => {
  return <h1>Zdravo svete!</h1>;
};
```

```
};  
  
ReactDOM.render(<App />, document.getElementById('root'));
```

Nakon kreiranja ovih fajlova, izvršavanje komande `webpack` u terminalu će generisati `bundle.js` fajl u `dist` direktorijumu. Ovaj fajl će sadržati sve potrebne module i može se uključiti u `HTML` fajl unutar `<script>` elementa.

8.3 Vite

Vite je moderni alat za pakovanje modula i alatka za izgradnju veb aplikacija koju je kreirao *Evan You*, osnivač *Vue.js*². Ime „*Vite*” dolazi iz francuskog jezika i znači „brzo”, što odražava glavni cilj ovog alata - pružanje bržeg i lakšeg razvojnog iskustva.

Instalacija *Vite*-a je jednostavna i može se obaviti slično *Webpack*-u, koristeći `npm`. *Vite* dolazi sa predefinisanim šablonima za različite radne okvire i biblioteke, uključujući i *React.js*. Nakon instalacije, konfiguracija *Vite*-a se obavlja putem `vite.config.js` fajla. Za razliku od *Webpack*-a, *Vite* ima mnogo jednostavniju i lakšu konfiguraciju.

Zanimljivo je da *Vite* sebe opisuje kao „*non-bundler*”, umesto da koristi tradicionalni alat za pakovanje modula, *Vite* koristi *ES Modules* za razvojnu izgradnju, što omogućava brže ponovno učitavanje i manje početno vreme učitavanja u odnosu na tradicionalne alate za pakovanje modula. Takođe, *Vite* ima dobru podršku za *ES Modules*, što omogućava bolju kompatibilnost sa modernim *Javascript* bibliotekama i funkcionalnostima.

Za produkcionu *build*, *Vite* koristi *Rollup*, još jedan popularni alat za pakovanje modula. *Rollup* je dizajniran sa fokusom na efikasnost i generisanje malih *bundle*-a, što ga čini idealnim za produkcionu *build*. Kombinacija *Vite*-a za razvoj i *Rollup*-a za produkciju pruža optimalnu kombinaciju brzine i efikasnosti.

Kao i *Webpack*, *Vite* takođe podržava *HMR*. Ovo omogućava da se izmene u kodu odmah prikažu u pregledaču bez potrebe za punim osvežavanjem stranice. Međutim, zbog „*non-bundling*” pristupa, to jest, zahvaljujući korišćenju *ES Modules*, *Vite* pruža veoma ugodno *HMR* iskustvo u odnosu na *Webpack* jer kada se fajl promeni, samo taj fajl i fajlovi koji direktno zavise od njega se ponovo učitavaju,

² *Vue.js* je još jedan popularni *Javascript* radni okvir za izgradnju korisničkih interfejsa.

umesto celog bundle-a. Ovo može da rezultira znatno bržim vremenima ponovnog učitavanja, posebno za velike aplikacije.

8.3.1 Primer korišćenja *Vite*-a u *React.js* aplikaciji

Kao i kod *Webpack*-a, biće razmotren primer jednostavne *React.js* aplikacije koja će biti izgrađena pomoću *Vite*. *Vite* već nudi predefinisane šablone za različite radne okvire i biblioteke, uključujući i *React.js*, tako da je postavljanje novog projekta jednostavno.

Prvo se treba instalirati *Vite CLI* globalno koristeći komandu `npm install -g create-vite`.

Nakon toga, može se kreirati novi *React.js* projekat koristeći komandu `create-vite react-vite-example`. Ovo će kreirati novi direktorijum `react-vite-example` sa predefinisanim *Vite* konfiguracionim fajlovima.

Sada se trebaju instalirati zavisnosti projekta koristeći `npm install`.

Na kraju, da bi se napravio paket za produkciju, koristi se komanda `npm run build` koji poziva komandu `vite build` u pozadini. Ovo će kreirati direktorijum `dist` sa paketom koji se može servirati kao statički veb sajt.

8.4 Poređenje *Vite*-a i *Webpack*-a

Kada se vrši poređenje između *Webpack*-a i *Vite*-a, nekoliko ključnih aspekata dolazi u obzir:

- *Performanse*: *Vite* generalno pruža brže vreme učitavanja tokom razvoja zahvaljujući svom pristupu kao „*non-bundler*”, koji koristi *ES Module (ESM)* za brže vreme učitavanja i ponovnog učitavanja. S druge strane, iako *Webpack* može biti sporiji za inicijalno učitavanje, nudi snažne optimizacije za produkcionu *build*, što može rezultirati manjim vremenima učitavanja u produkciji.
- *Konfigurabilnost*: *Webpack* je poznat po svojoj visokoj konfigurabilnosti, što može biti prednost za složene projekte koji zahtevaju specifične funkcionalnosti ili optimizacije. Međutim, *Vite* teži jednostavnosti i lakoći upotrebe, sa manje potrebe za ručnom konfiguracijom, što može biti prednost za manje složene projekte ili za programere koji preferiraju brži setup.

- **Podrška za različite funkcije:** Oba alata za pakovanje modula podržavaju širok spektar funkcija, uključujući *HMR*, razdvajanje koda i lenjo učitavanje. Međutim, mogu postojati razlike u načinu na koji ove funkcije rade ili se konfigurišu u svakom alatu za pakovanje modula, što može uticati na izbor alata za pakovanje modula u zavisnosti od specifičnih potreba projekta.

Odluka između *Webpack*-a i *Vite*-a zavisiće od specifičnih potreba projekta. Ukoliko su prioritet brzina razvoja i jednostavnost konfiguracije, *Vite* bi mogao biti bolji izbor. S druge strane, u slučaju složenih projekata koji zahtevaju visoku konfigurabilnost i specifične funkcije, *Webpack* bi mogao biti bolji izbor.

8.5 Tehnike za smanjenje veličine bundle-a

Smanjenje veličine *bundle*-a je ključno za poboljšanje performansi *React.js* aplikacija, posebno za one sa velikim brojem komponenti i modula. Postoji nekoliko tehnika koje se mogu koristiti za ovu svrhu:

- **Razdvajanje koda (*Code Splitting*):** Ova tehnika podrazumeva podeľu koda na manje delove (ili „*chunks*“) koji se mogu učitavati i izvršavati asinhrono. Ovo može poboljšati performanse aplikacije tako što će korisnici preuzimati samo kod koji im je potreban u datom trenutku.
- **Lenjo učitavanje (*Lazy Loading*):** Slično razdvajanju koda, lenjo učitavanje omogućava asinhrono učitavanje funkcionalnosti aplikacije samo kada je to potrebno. Ovo može uključivati učitavanje komponenti na zahtev, ili učitavanje velikih biblioteka samo kada su potrebne.
- ***Tree Shaking*:** Ova tehnika uklanja neiskorišćeni kod iz finalnog *bundle*-a, smanjujući njegovu ukupnu veličinu.

Takođe se može smanjiti vreme izgradnje paketa tako što će se koristiti efikasniji alati za pakovanje modula (kao što je *Vite*), optimizovati konfiguracija alata za pakovanje modula, i koristiti keširanje za ponovnu upotrebu prethodno izgrađenih modula.

Ako se želi lakše *debug*-ovati *React.js* aplikacija, koriste se izvorne mape (*eng. source maps*). One predstavljaju fajlove koji mapiraju izvorni kod na minifikovani (ili transpilovani) kod koji se izvršava u pregledaču. Ovo omogućava alatima za

debug-ovanje da prikažu izvorni kod, čak i kada se izvršava minifikovani kod, što olakšava pronalaženje i ispravljanje grešaka. Većina modernih alata za pakovanje modula, uključujući *Webpack* i *Vite*, podržava generisanje izvornih mapa.

Glava 9

Poređenje *React.js*-a i *Angular*-a

Motivacija za upoređivanje biblioteke *React.js* i radnog okvira *Angular* proizlazi iz želje da se detaljno razumeju različite mogućnosti i pristupi koje svaka tehnologija nudi. Ovo poređenje pruža dublje uvide u načine implementacije osnovnih i naprednih funkcionalnosti u obe tehnologije, što je od suštinskog značaja za donošenje obrazloženih odluka o izboru tehnologije u budućim projektima. Konkretno, fokusiraće se na izradu aplikacije za praćenje beleški u oba okruženja. Ova aplikacija služi kao praktičan primer koji omogućava istraživanje ključnih aspekata kao što su koncept komponenta, upravljanje stanjem, navigacija i autentifikacija što bi trebalo da pomogne čitaocu da bolje razume praktične razlike između korišćenja *React.js*-a i *Angular*-a a ne samo teorijske.

Poseban naglasak biće stavljen na pitanje sigurnosti, naročito na zaštićenim rutama i upravljanju neautorizovanim pristupom. Ovo je od suštinskog značaja jer se *React.js* i *Angular* različito nose sa ovim izazovima, što može biti odlučujući faktor u izboru odgovarajuće tehnologije za buduće projekte.

9.1 Ukratko o *Angular*-u

Angular je platforma i okvir za izradu klijentskih aplikacija jedne strane koristeći *HTML* i *TypeScript*. Razvijen i održavan od strane *Google*-a, kombinuje deklarativne šablone, ubrizgavanje zavisnosti (*eng. dependency injection*) i integrisane alate kako bi se rešili izazovi u razvoju. Neće se detaljno ulaziti u objašnjavanje funkcionalnosti *Angular*-a, jer je on dobro obrađen na kursu „Programiranje za veb”. Više informacija o kursu i skripti može se pronaći u literaturi navedenoj pod [7].

9.2 Pregled aplikacije

U ovom delu će biti predstavljena struktura i funkcionalnosti aplikacije za praćenje beleški koja će biti implementirana u *React.js*-u i *Angular*-u. Funkcionalnosti koje će biti implementirane u obe aplikacije su sledeće:

1. Lista beleški:

- Prikaz svih beleški sa naslovom, kratkim opisom i dugmićima za brisanje i završavanje beleške.

2. Dodavanje, završavanje i brisanje beleške:

- Korišćenje `useState` (u *React.js*-u) ili povezivanje događaja i atributa (u *Angular*-u) za upravljanje unosom.
- Dugme za brisanje pored svake beleške.
- Dugme za završavanje beleške pored svake beleške.
- Formular za unos naslova i opisa beleške.

3. Dohvatanje podataka sa servera i dinamičko ažuriranje liste:

- Simulacija bekenda koristeći *JSON Server* i korišćenje `fetch`-a (u *React.js*-u) ili `HttpClient`-a (u *Angular*-u) za dohvatanje podataka.
- Korišćenje kuka životnog ciklusa (u *Angular*-u) ili `useEffect` (u *React.js*-u) za praćenje promena i ažuriranje liste.

4. Rutiranje i navigacija:

- Implementacija osnovne navigacije između stranica.
- Simulacija prijave koristeći *React Context API* i *Angular* servisa za upravljanje korisničkim podacima, dok se `localStorage` koristi za čuvanje tih podataka kroz osvežavanje stranice.
- Zaštićene rute koje preusmeravaju neautorizovane korisnike na stranicu za prijavu.

Ovom strukturom, aplikacija će efikasno demonstrirati osnovne funkcionalnosti i razlike između *React.js*-a i *Angular*-a.

9.3 Postavljanje projekta

9.3.1 Postavljanje *React.js* projekta preko *Vite*

U prethodnom poglavlju su spomenuta 2 alata za pakovanje modula, *Webpack* i *Vite*. S obzirom da aplikacija koja se bude pravila neće biti velika, *Vite* će biti korišćen kao alat za pakovanje modula za *React.js* zbog svoje brzine. *React.js* projekat može se postaviti koristeći *Vite* CLI na sledeći način:

1. Instalacija *Vite CLI* globalno: `npm install -g create-vite`
2. Kreiranje novog *React.js* projekta: `create-vite react-example --template react-ts`
3. Ulazak u direktorijum projekta i instalacija zavisnosti: `cd react-example && npm install`
4. Pokretanje razvojnog servera: `npm run dev`

9.3.2 Postavljanje *Angular* projekta koristeći *Angular CLI*

Angular CLI je standardni alat za inicijalizaciju i razvoj *Angular* aplikacija. Evo koraka za postavljanje *Angular* projekta:

1. Instalacija *Angular CLI* alata globalno: `npm install -g @angular/cli`
2. Kreiranje novog *Angular* projekta ¹: `ng new angular-example`
3. Ulazak u direktorijum projekta: `cd angular-example`
4. Pokretanje razvojnog servera: `ng serve`

9.4 Izrada aplikacije

Pre nego što se započne sa izradom aplikacije, važno je napomenuti da *Vite* (za *React.js*) i *Angular CLI* prilikom inicijalizacije projekta generišu određeni

¹Ova komanda će Vas pitati koje karakteristike želite da uključite u projekat. Za potrebe ovog poređenja, možete prihvatiti *Angular* rutiranje i *CSS* kao stilizacijskim jezikom.

podrazumevani sadržaj. Da bi se imala čista osnova za aplikaciju, preporučuje se brisanje ovog podrazumevanog sadržaja.

Za *Vite*, brisanje se može izvesti na sledeći način:

1. Otvoriti glavnu komponentu aplikacije, `App.tsx`.
2. Ukloniti sav sadržaj unutar povratne vrednosti `return` funkcije, osim glavnog kontejnera.
3. Ukloniti sve nepotrebne uvoze na vrhu datoteke.
4. Ukloniti *CSS* datoteke `index.css` i `App.css` i izbrisati uvoz ovih datoteka u `index.tsx` i `App.tsx`.

Za *Angular*, brisanje se može izvesti na sledeći način:

1. Otvoriti glavnu komponentu aplikacije koja se nalazi u `src/app/app.component.html`.
2. Ukloniti sav *HTML* sadržaj unutar ove datoteke.
3. U datoteci `src/app/app.component.ts`, ukloniti sve nepotrebne uvoze i atribute unutar klase `AppComponent`.

Nakon ovih koraka, dobija se čista osnova za dalju izradu aplikacije za praćenje beleški.

9.4.1 Lista beleški

Prvi korak u izradi aplikacije je implementacija liste beleški. Ova lista će prikazivati sve beleške koje su dodate od strane korisnika sa naslovom, kratkim opisom i dugmićima za brisanje i završavanje beleške. Ova lista će biti prikazana na glavnoj stranici aplikacije.

U početku će se korisiti lažni podaci za beleške, sa interfejsom koji izgleda ovako:

```
interface INote {
  id: number;
  title: string;
  description: string;
  completed: boolean;
}
```

React.js

Unutar App.tsx fajla dodati sledeći kod:

```
import Note from "../components/Note";
import { INote } from "../types/note";

function App() {
  const notes: INote[] = [
    {
      id: 1,
      title: "Sastanak sa timom",
      description: "Pripremiti pitanja za sastanak sa timom u vezi novog projekta.",
      completed: false,
    },
    {
      id: 2,
      title: "Kupovina namirnica",
      description: "Kupiti sveze voce, povrce i mlecne proizvode u prodavnici.",
      completed: false,
    },
  ];

  return (
    <div className="notes-container">
      <h1 className="notes-list-title">Lista beleški</h1>
      <ul className="notes-list">
        {notes.map((note) => (
          <li key={note.id} className="note-item">
            <h2 className="note-title">{note.title}</h2>
            <p className="note-description">{note.description}</p>
            <p className="note-status">
              Status: {note.completed ? "Završeno" : "Nije završeno"}
            </p>
            <button className="toggle-status-button">
              Oznaci kao {note.completed ? "Nije završeno" : "Završeno"}
            </button>
            <button className="delete-note-button">Izbrisi</button>
          </li>
        ))}
      </ul>
    </div>
  );
}

export default App;
```

Može se videti da je lista beleški prikazana koristeći `.map()` funkciju za iteraciju

kroz svaku belešku i prikazivanje iste na ekranu. Ova beleška se može izolovati u posebnu komponentu koja će se zvati `Note.tsx` i izgledaće ovako:

```
import { INote } from "../types/note";

const Note = ({ note }: { note: INote }) => {
  return (
    <li className="note-item">
      <h2 className="note-title">{note.title}</h2>
      <p className="note-description">{note.description}</p>
      <p className="note-status">
        Status: {note.completed ? "Završeno" : "Nije završeno"}
      </p>
      <button className="toggle-status-button">
        Oznaci kao {note.completed ? "Nije završeno" : "Završeno"}
      </button>
      <button className="delete-note-button">Izbrisi</button>
    </li>
  );
};

export default Note;
```

Dok će se `App.tsx` izmeniti na sledeći način:

```
// ostatak je isti kao i ranije

return (
  <div className="notes-container">
    <h1 className="notes-list-title">Lista beleški</h1>
    <ul className="notes-list">
      {notes.map((note) => (
        <Note key={note.id} note={note} />
      ))}
    </ul>
  </div>
);
```

Primećuje se da je atribut `key` prebačen na komponentu `Note.tsx` a ne u `` element. To je zato što `key` atribut mora biti postavljen na najviši element u iteraciji. U ovom slučaju, to je `<Note>` komponenta.

Angular

Da bi se prikazala lista beleški, potrebno je dodati sledeći kod u `app.component.html`:

```
<div class="notes-container">
<h1 class="notes-list-title">Lista beleški</h1>
<ul class="notes-list">
  <li *ngFor="let note of notes" class="note-item">
```

```
<h2 class="note-title">{{ note.title }}</h2>
<p class="note-description">{{ note.description }}</p>
<p class="note-status">
  Status: {{ note.completed ? "Završeno" : "Nije završeno" }}
</p>
<button class="toggle-status-button">
  Oznaci kao {{ note.completed ? "Nije završeno" : "Završeno" }}
</button>
<button class="delete-note-button">Izbrisi</button>
</li>
</ul>
</div>
```

I u `app.component.ts` dodati sledeći kod:

```
import { Component } from "@angular/core";
import { INote } from "./note";

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})

export class AppComponent {
  title = 'angular-example';

  notes: INote[] = [
    {
      id: 1,
      title: 'Sastanak sa timom',
      description:
        'Pripremiti pitanja za sastanak sa timom u vezi novog projekta.',
      completed: false,
    },
    {
      id: 2,
      title: 'Kupovina namirnica',
      description: 'Kupiti sveže voće, povrće i mlecne proizvode u prodavnici.',
      completed: false,
    },
  ];
}
```

Nakon ovih koraka, korisnik će biti u mogućnosti da vidi sve svoje beleške na glavnoj stranici aplikacije.

U *Angular*-u, može se direktno dodati nova komponenta koristeći Angular CLI komandu: `ng generate component note`. Ova komanda će kreirati novu kompo-

mentu u direktorijumu `app/note` i dodati je u `app.module.ts` fajl. Fajl `note.component.ts` izgleda ovako:

```
import { Component, Input } from '@angular/core';
import { INote } from 'src/types/note';

@Component({
  selector: 'app-note',
  templateUrl: './note.component.html',
  styleUrls: ['./note.component.css'],
})
export class NoteComponent {
  @Input() note!: INote;
}
```

Dok fajl `note.component.html` izgleda ovako:

```
<h2 class="note-title">{{ note.title }}</h2>
<p class="note-description">{{ note.description }}</p>
<p class="note-status">
  Status: {{ note.completed ? "Završeno" : "Nije završeno" }}
</p>
<button class="toggle-status-button">
  Oznaci kao {{ note.completed ? "Nije završeno" : "Završeno" }}
</button>
<button class="delete-note-button">Izbrisi</button>
```

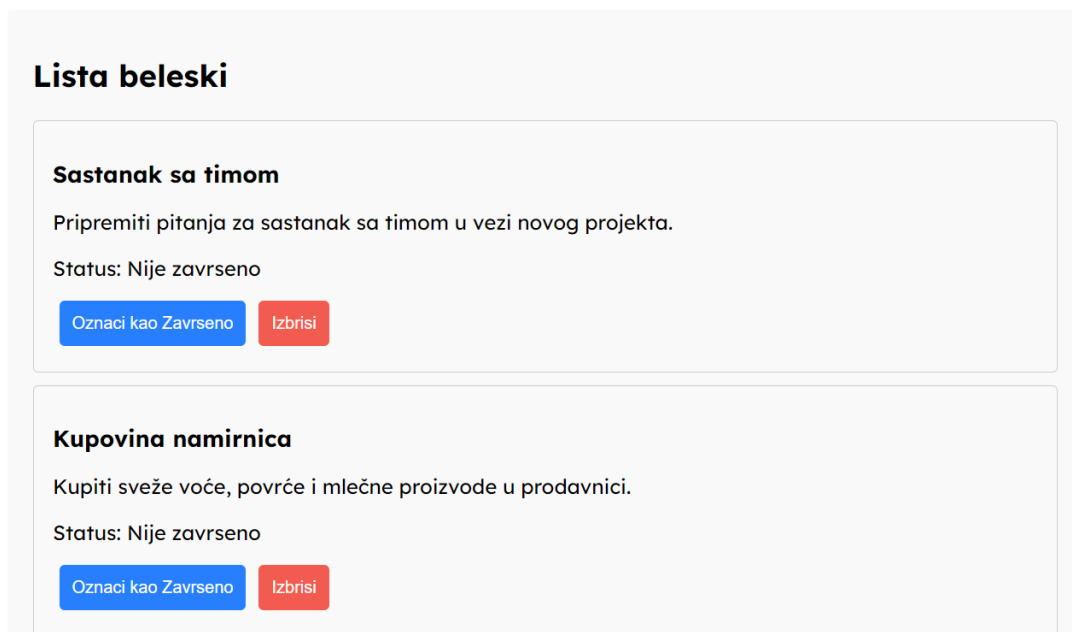
Pošto je polje `notes` definisano u `AppComponent` klasi, potrebno je da se ono prosledi komponenti `NoteComponent` kao *input* parametar. Ovo se postiže korišćenjem `@Input` dekoratora a u `app.component.html` fajlu komponenta `NoteComponent` se može koristiti na sledeći način:

```
// ostatak je isti kao i ranije
<ul class="notes-list">
  <li *ngFor="let note of notes" class="note-item">
    <app-note [note]="note" />
  </li>
</ul>
```

Nakon ovih izmena, veb strana će izgledati kao na slici 9.1.

9.4.2 Završavanje i brisanje beleške

S obzirom na to da će sada biti omogućeno korisnicima da dodaju, brišu i završavaju beleške, potrebno je da stanje niza svih beležaka bude praćeno na odgovarajući način.



Slika 9.1: Lista beleški

React.js

U *React.js*-u, koristi se `useState` kuka za praćenje stanja niza beleški koje se koriste u funkcijama dodavanja, brisanja i završavanja beleške. Da bi se omogućilo završavanje i brisanje beleški, kod se menja na sledeći način:

```
// App.tsx
import { useState } from "react";
import Note from "../components/Note";

function App() {
  const [notes, setNotes] = useState([
    {
      id: 1,
      title: "Sastanak sa timom",
      description: "Pripremiti pitanja za sastanak sa timom u vezi novog projekta.",
      completed: false,
    },
    {
      id: 2,
      title: "Kupovina namirnica",
      description: "Kupiti sveže voće, povrće i mlečne proizvode u prodavnici.",
      completed: false,
    },
  ]);
}
```

```

function toggleCompleteNote(id: number) {
  const newNotes = notes.map((note) => {
    if (note.id === id) {
      return {
        ...note,
        completed: !note.completed,
      };
    }
    return note;
  });
  setNotes(newNotes);
}

function deleteNote(id: number) {
  const newNotes = notes.filter((note) => note.id !== id);
  setNotes(newNotes);
}

return (
  <div className="notes-container">
    <h1 className="notes-list-title">Lista beleski</h1>
    <ul>
      {notes.map((note) => (
        <Note key={note.id} note={note} toggleCompleteNote={
          toggleCompleteNote} deleteNote={deleteNote} />
      ))}
    </ul>
  </div>
);
}

export default App;

```

```

// Note.tsx
import { INote } from "../types/note";

interface INoteProps {
  note: INote;
  toggleCompleteNote: (id: number) => void;
  deleteNote: (id: number) => void;
}

const Note = ({ note, toggleCompleteNote, deleteNote }: INoteProps) => {
  return (
    <li className="note-item">
      <h2 className="note-title">{note.title}</h2>
      <p className="note-description">{note.description}</p>
      <p className="note-status">
        Status: {note.completed ? "Završeno" : "Nije završeno"}
      </p>
    </li>
  );
}

```

```

    <button onClick={() => toggleCompleteNote(note.id)} className="
      toggle-status-button">
      Oznaci kao {note.completed ? "Nije završeno" : "Završeno"}
    </button>
    <button onClick={() => deleteNote(note.id)} className="
      delete-note-button">
      Izbrisi
    </button>
  </li>
);
};

export default Note;

```

Može se primetiti da je `useState` kuka korišćena za praćenje stanja niza beleški. Takođe, funkcije `toggleCompleteNote` i `deleteNote` su prosledene komponenti `Note` kao *props*. Ove funkcije će biti pozvane kada korisnik klikne na dugme za završavanje ili brisanje beleške. Takođe, kao *props* za komponentu `Note` je interfejs posebno izdvojen zbog veće preglednosti koda.

Angular

U *Angular*-u, koristi se povezivanje događaja za praćenje stanja niza beleški koje se koriste u funkcijama dodavanja, brisanja i završavanja beleške. Da bi se omogućilo završavanje i brisanje beleški, kod komponente `app` se menja na sledeći način:

```

// app.component.html
<h1 class="notes-list-title">Lista beleski</h1>
<ul class="notes-list">
  <li *ngFor="let note of notes" class="note-item">
    <app-note
      [note]="note"
      [deleteNote]="deleteNote"
      [toggleCompleteNote]="toggleCompleteNote"
    />
  </li>
</ul>

```

```

// app.component.ts
import { Component } from '@angular/core';
import { INote } from 'src/types/note';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css'],

```

```

})
export class AppComponent {
  title = 'angular-example';

  notes: INote[] = [
    {
      id: 1,
      title: 'Sastanak sa timom',
      description:
        'Pripremiti pitanja za sastanak sa timom u vezi novog projekta.',
      completed: false,
    },
    {
      id: 2,
      title: 'Kupovina namirnica',
      description: 'Kupiti sveze voce, povrce i mlecne proizvode u prodavnici.',
      completed: false,
    },
  ];

  toggleCompleteNote = (id: number) => {
    this.notes = this.notes.map((note) => {
      if (note.id === id) {
        return {
          ...note,
          completed: !note.completed,
        };
      }
      return note;
    });
  };

  deleteNote = (id: number) => {
    this.notes = this.notes.filter((note) => note.id !== id);
  };
}

```

dok se kod komponente note menja na sledeći način:

```

// note.component.html
<h2 class="note-title">{{ note.title }}</h2>
<p class="note-description">{{ note.description }}</p>
<p class="note-status">
  Status: {{ note.completed ? "Zavrшено" : "Nije zavrшено" }}
</p>
<button (click)="toggleCompleteNote(note.id)" class="toggle-status-button">
  Oznaci kao {{ note.completed ? "Nije zavrшено" : "Zavrшено" }}
</button>
<button (click)="deleteNote(note.id)" class="delete-note-button">
  Izbrisi

```

```
</button>
```

```
import { Component, Input } from '@angular/core';
import { INote } from 'src/types/note';

@Component({
  selector: 'app-note',
  templateUrl: './note.component.html',
  styleUrls: ['./note.component.css'],
})
export class NoteComponent {
  @Input() note!: INote;

  @Input() toggleCompleteNote!: (id: number) => void;

  @Input() deleteNote!: (id: number) => void;
}
```

Može se primetiti da je niz `notes` ostao nepromenjen, samo su funkcije `toggleCompleteNote` i `deleteNote` dodate kao metode klase `AppComponent`. Ove funkcije će biti pozvane kada korisnik klikne na dugme za završavanje ili brisanje beleške što je omogućeno korišćenjem povezivanja događaja putem (`click`) atributa.

9.4.3 Dodavanje beleške

Dalje, potrebno je omogućiti korisniku da doda novu belešku. To se može postići dodavanjem formulara za unos beleške na glavnu stranicu aplikacije. Formular će se sastojati od polja za unos naslova i opisa beleške, kao i dugmeta za dodavanje nove beleške. Prikazivaće se na glavnoj stranici aplikacije iznad liste beleški.

React.js

U *React.js*-u, samo je potrebno dodati formu za unos beleške na glavnu stranicu aplikacije u fajlu `App.tsx`. Ovaj formular će se sastojati od polja za unos naslova i opisa beleške, kao i dugmeta za dodavanje nove beleške:

```
// ostatak je isti kao i ranije

function addNote(e: React.FormEvent<HTMLFormElement>) {
  e.preventDefault();

  const data = new FormData(e.currentTarget);

  const newNote = {
```



```

    id: notes.length + 1,
    title: (data.get("title") || "Prazan naslov") as string,
    description: (data.get("description") || "Prazan opis") as string,
    completed: false,
  };

  setNotes([...notes, newNote]);

  e.currentTarget.reset();
}

return (
  <div className="notes-container">
    <form onSubmit={addNote} className="add-note-form">
      <input name="title" type="text" placeholder="Title" className="
        note-input" />
      <input name="description" type="text" placeholder="Description"
        className="note-input" />
      <button type="submit" className="add-note-button">
        Dodaj belesku
      </button>
    </form>
    <h1 className="notes-list-title">Lista beleski</h1>
    <ul className="notes-list">
      {notes.map((note) => (
        <Note key={note.id} note={note} toggleCompleteNote={
          toggleCompleteNote} deleteNote={deleteNote} />
      ))}
    </ul>
  </div>
);

```

Angular

Da bi se iskoristile funkcionalnosti formulara u *Angular*-u, potrebno je uvesti `ReactiveFormsModule` u `app.module.ts` fajl:

```

import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';

import { AppRoutingModule } from './app-routing.module';
import { AppComponent } from './app.component';
import { NoteComponent } from './note/note.component';
import { ReactiveFormsModule } from '@angular/forms';

@NgModule({
  declarations: [AppComponent, NoteComponent],
  imports: [BrowserModule, AppRoutingModule, ReactiveFormsModule],
  providers: [],

```

```
bootstrap: [AppComponent],
})
export class AppModule {}
```

Dalje, kao i kod *React.js*-a, u *Angular*-u je potrebno dodati formu za unos beleške na glavnu stranicu aplikacije u fajlu `app.component.html`. Ovaj formular će se sastojati od polja za unos naslova i opisa beleške, kao i dugmeta za dodavanje nove beleške:

```
<form (ngSubmit)="addNote()" [formGroup]="addNoteForm" class="add-note-form">
  <input
    type="text"
    name="title"
    placeholder="Title"
    formControlName="title"
    class="note-input"
  />
  <input
    type="text"
    name="description"
    placeholder="Description"
    formControlName="description"
    class="note-input"
  />
  <button type="submit" class="add-note-button">Dodaj belesku</button>
</form>
<!-- ostatak je isti kao i ranije -->
```

U `app.component.ts` fajlu, potrebno je dodati funkciju `addNote` koja će biti pozvana kada korisnik klikne na dugme za dodavanje nove beleške kao i inicijalizaciju formulara za dodavanje beleške (koja se u ovom slučaju zove `addNoteForm`):

```
// ostatak je isti kao i ranije

addNoteForm = this.formBuilder.group({
  title: '',
  description: '',
});

addNote = () => {
  const note: INote = {
    id: this.notes.length + 1,
    title: this.addNoteForm.value.title || 'Prazan naslov',
    description: this.addNoteForm.value.description || 'Prazan opis',
    completed: false,
  };
  this.notes = [...this.notes, note];
  this.addNoteForm.reset();
};
```

Dakle, korišćenjem vezivanja atributa (*eng. attribute binding*), `addNoteForm` će biti prosleđen formularu za unos beleške. Takođe, korišćenjem `formControlName` atributa, polja za unos naslova i opisa beleške će biti vezana za odgovarajuće attribute u formularu za dodavanje beleške.

Nakon ovih izmena, veb strana će izgledati kao na slici 9.2.

Title

Description

Dodaj belesku

Lista beleski

Sastanak sa timom
Pripremiti pitanja za sastanak sa timom u vezi novog projekta.
Status: Nije završeno
Oznaci kao Završeno Izbrisi

Kupovina namirnica
Kupiti sveže voće, povrće i mlečne proizvode u prodavnici.
Status: Nije završeno
Oznaci kao Završeno Izbrisi

Slika 9.2: Formular za dodavanje beleške

9.4.4 Dohvatanje podataka sa servera i dinamičko ažuriranje liste

Do sada su se koristili statički podaci na frontendu za prikazivanje liste beleški. Sledeći korak je da se ove beleške dohvate sa servera i prikažu u listi. Ovo se može postići koristeći `fetch` (u *React.js*-u) ili `HttpClient` (u *Angular*-u). Oba ova alata su klijentske biblioteke koje omogućavaju slanje HTTP zahteva sa klijenta. Ove biblioteke su veoma slične i imaju sličan interfejs.

Simulacija bekenda

Pošto ne postoji bekennd za ovu aplikaciju, koristiće se *JSON Server*[12] da simulira bekennd. *JSON Server* je alat koji omogućava kreiranje lažnog bekenda za testiranje i izradu prototipa. Ovaj alat može se instalirati globalno koristeći `npm install -g json-server` i pokrenuti koristeći `json-server --watch db.json`, čime će se kreirati lažni bekennd koji će biti dostupan na adresi `http://localhost:3000`.

Podaci o beleškama nalaze se u fajlu `db.json`, a resursima se pristupa putem adrese `http://localhost:3000/notes`. Na osnovu ovih podataka, može se početi sa izradom lista beleški koja se dohvata sa servera u obe aplikacije.

React.js

Umesto da se u fajlu `App.tsx` koristi statički niz beleški kao inicijalno stanje `useState` kuke, ono će sada biti prazan niz, a pomoću `useEffect` kuke će se dohvatiti beleške sa servera i ažurirati stanje. Kuka `useEffect` se poziva nakon što se komponenta montira:

```
// ostatak je isti kao i ranije

useEffect(() => {
  fetch("http://localhost:3000/notes")
    .then((res) => res.json())
    .then((res) => setNotes(res));
}, []);

const [notes, setNotes] = useState<INote []>([]);

// ostatak je isti kao i ranije
```

U funkcijama `toggleCompleteNote`, `deleteNote` i `addNote` će se sada slati HTTP zahtevi kako bi se ažurirali podaci na serveru a stanje `notes` niza će se ažurirati tek nakon što se dobije odgovor od servera:

```
// ostatak je isti kao i ranije

function toggleCompleteNote(id: number) {
  const note = notes.find((n) => n.id === id);

  if (note) {
    const updatedNote = { ...note, completed: !note.completed };
    fetch(`http://localhost:3000/notes/${id}`, {
      method: "PATCH",
      headers: {
        "Content-Type": "application/json",
      },
    });
  }
}
```

```
    },
    body: JSON.stringify(updatedNote),
  })
  .then((res) => res.json())
  .then((newNote) => setNotes((prevNotes) => prevNotes.map((note) => (
    note.id === id ? newNote : note))));
}
}

function deleteNote(id: number) {
  setNotes((prevNotes) => prevNotes.filter((note) => note.id !== id));

  fetch(`http://localhost:3000/notes/${id}`, {
    method: "DELETE",
  });
}

function addNote(e: React.FormEvent<HTMLFormElement>) {
  e.preventDefault();

  const data = new FormData(e.currentTarget);

  const newNote = {
    title: (data.get("title") || "Prazan naslov") as string,
    description: (data.get("description") || "Prazan opis") as string,
    completed: false,
  };

  fetch("http://localhost:3000/notes", {
    method: "POST",
    headers: {
      "Content-Type": "application/json",
    },
    body: JSON.stringify(newNote),
  })
  .then((res) => res.json())
  .then((newNote) => setNotes([...notes, newNote]));

  e.currentTarget.reset();
}

// ostatak je isti kao i ranije
```

Angular

Prvo je potrebno uvesti HttpClientModule u AppModule:

```
import { HttpClientModule } from "@angular/common/http";
// ostatak je isti kao i ranije
```

```
@NgModule({
  // ostatak je isti kao i ranije
  imports: [HttpClientModule],
})

// ostatak je isti kao i ranije
```

Zatim, generiše se *resolver* koji će dohvatiti beleške sa servera pre nego što se komponenta prikaže. Prvo, koristi se *Angular CLI* komanda `ng generate resolver notes`, zatim ovaj fajl se otvara i menja da izgleda ovako:

```
import { HttpClient } from '@angular/common/http';
import { Injectable } from '@angular/core';
import {
  ActivatedRouteSnapshot,
  Resolve,
  RouterStateSnapshot,
} from '@angular/router';
import { Observable } from 'rxjs';

@Injectable({
  providedIn: 'root',
})
export class NotesResolver implements Resolve<any> {
  constructor(private http: HttpClient) {}

  resolve(
    route: ActivatedRouteSnapshot,
    state: RouterStateSnapshot
  ): Observable<any> {
    return this.http.get('http://localhost:3000/notes');
  }
}
```

Dalje se menja fajl `app-routing.module.ts` kako bi ruta za beleške koristila ovaj *resolver*:

```
// ostatak je isti kao i ranije

const routes: Routes = [
  { path: '', component: LoginComponent },
  {
    path: 'notes',
    component: NotesComponent,
    canActivate: [AuthGuard],
    resolve: { notes: NotesResolver },
  },
];

// ostatak je isti kao i ranije
```

Konačno, menja se fajl `app.component.ts` tako da polje `notes` više nije statički niz beleški, već je niz beleški koji se dohvata iz *resolver*-a:

```
// ostatak je isti kao i ranije
export class NotesComponent {
  notes: INote[] = [];

  constructor(
    private loginService: LoginService,
    private router: Router,
    private formBuilder: FormBuilder,
    private route: ActivatedRoute
  ) {
    this.notes = this.route.snapshot.data['notes'];
  }

  // ostatak je isti kao i ranije
}
```

U funkcijama `toggleCompleteNote`, `deleteNote` i `addNote` će se sada slati HTTP zahtevi kako bi se ažurirali podaci na serveru a stanje `notes` niza će se ažurirati tek nakon što se dobije odgovor od servera:

```
// ostatak je isti kao i ranije

private baseUrl = 'http://localhost:3000/notes';

constructor(
  private loginService: LoginService,
  private router: Router,
  private formBuilder: FormBuilder,
  private route: ActivatedRoute,
  private http: HttpClient
) {
  this.notes = this.route.snapshot.data['notes'];
}

toggleCompleteNote = (id: number) => {
  const note = this.notes.find((n) => n.id === id);
  if (note) {
    const updatedNote = { ...note, completed: !note.completed };
    this.http
      .put<INote>(`${this.baseUrl}/${id}`, updatedNote)
      .subscribe((newNote) => {
        this.notes = this.notes.map((n) => (n.id === id ? newNote : n));
      });
  }
};

deleteNote = (id: number) => {
```

```

    this.http.delete(`${this.baseUrl}/${id}`).subscribe((response) => {
      this.notes = this.notes.filter((note) => note.id !== id);
    });
  };

  addNote = () => {
    const note = {
      title: this.addNoteForm.value.title || 'Prazan naslov',
      description: this.addNoteForm.value.description || 'Prazan opis',
      completed: false,
    };
    this.http.post<INote>(this.baseUrl, note).subscribe((newNote) => {
      this.notes = [...this.notes, newNote];
      this.addNoteForm.reset();
    });
  };
};

// ostatak je isti kao i ranije

```

9.4.5 Rutiranje i navigacija

Sada kada je lista beleški implementirana sa funkcionalnostima dodavanja, brisanja i završavanja beleški, potrebno je zaštititi stranicu sa listom beleški tako što će korisnici inicijalno biti preusmereni na stranicu za prijavu. Da bi to bilo omogućeno, prvo se moraju uvesti rute u obe aplikacije.

React.js

Da bi se uvele rute u *React.js* aplikaciju, koristiće se `react-router-dom` biblioteka koja je bila spomenuta u poglavlju 6. Da bi se ova biblioteka instalirala, potrebno je izvršiti sledeću komandu: `npm install react-router-dom`.

Nakon što je biblioteka instalirana, u fajlu `siteRoutes.tsx` će biti definisane rute za aplikaciju²:

```

import React from "react";

import { createBrowserRouter, createRoutesFromElements, Route } from "
  react-router-dom";

import Login from "./pages/Login";
import Notes from "./pages/Notes";

export const router = createBrowserRouter(

```

²U ovom primeru nije korišćeno lenjo učitavanje jer je aplikacija mala pa se neće primetiti razlike u performansama.


```

createRoutesFromElements(
  <React.Fragment>
    <Route path="/" element={<Login />} />,
    <Route path="/notes" element={<Notes />} />
  </React.Fragment>
)
);

```

Fajl `App.tsx` će biti izmenjen na sledeći način:

```

import { RouterProvider } from "react-router-dom";
import { router } from "./siteRoutes";
import React from "react";

function App() {
  return (
    <React.Fragment>
      <RouterProvider router={router} />
    </React.Fragment>
  );
}

export default App;

```

Ono što je ostalo je da se implementiraju stranice za prijavu i beleške. Ove stranice će biti implementirane u fajlovima `pages/Login.tsx` i `pages/Notes.tsx`. Kako je prethodni sadržaj fajla `App.tsx` zapravo predstavljao listu beseski, on će sada biti prebačen u fajl `pages/Notes.tsx` dok će fajl `pages/Login.tsx` izgledati ovako:

```

function Login() {
  return <div>Login</div>;
}

export default Login;

```

Stanje prijave korisnika se treba pratiti u okviru celog sajta. Da bi se to postiglo, korišće se *Context API* koji je spomenut u sekciji 5.2.6. Dodatno, informacije o korisniku će biti sačuvane i u `localStorage`-u kako bi se održala informacija o prijavi čak i ako se stranica osveži. *Provider* za korisnika će biti implementiran u fajlu `siteContext.tsx`:

```

import React from "react";

interface IUserContext {
  user: string | null;
  setUser: (user: string | null) => void;
}

```

```

export const UserContext = React.createContext<IUserContext>({
  user: null,
  setUser: () => {},
});

export const UserProvider = ({ children }: { children: React.ReactNode }) => {
  const [user, setUser] = React.useState<string | null>(localStorage.getItem("
    user"));

  return <UserContext.Provider value={{ user, setUser }}>{children}</
    UserContext.Provider>;
};

export const useUser = () => {
  const context = React.useContext(UserContext);

  if (context === undefined) {
    throw new Error("useUser must be used within a UserProvider");
  }

  return context;
};

```

Sada će cela aplikacija biti obavijena *Provider*-om za korisnika u fajlu `App.tsx` kako bi se informacijama o korisniku moglo pristupiti bilo gde u aplikaciji:

```

// ostatak je isti kao i ranije
<UserProvider>
  <RouterProvider router={router} />
</UserProvider>

```

Dalje, na Login stranicu će biti dodat jednostavni formular za prijavu koja će korisnika preusmeriti na listu beleški korišćenjem `useNavigate` kuke nakon što se prijavi i sačuvati njegove podatke korišćenjem `useUser` kuke i `localStorage`-a:

```

import { useNavigate } from "react-router-dom";
import { UserContext } from "../siteContext";
function Login() {
  const { setUser } = useUser();

  const navigate = useNavigate();

  function login(e: React.FormEvent<HTMLFormElement>) {
    e.preventDefault();

    const data = new FormData(e.currentTarget);

    if (!data.get("username") || !data.get("password")) {
      alert("Morate uneti korisnicko ime i lozinku.");
    }
  }
}

```

```

    return;
  }

  const user = {
    username: data.get("username") as string,
    password: data.get("password") as string,
  };

  setUser(user.username);
  localStorage.setItem("user", user.username);

  navigate("/notes");
}

return (
  <div className="login-page-wrapper">
    <div className="login-container">
      <h1 className="login-title">Prijava</h1>
      <form onSubmit={login} className="login-form">
        <input type="text" placeholder="Korisnicko ime" name="username"
          className="login-input" />
        <input type="password" placeholder="Sifra" name="password"
          className="login-input" />
        <button type="submit" className="login-button">
          Prijavi se
        </button>
      </form>
    </div>
  </div>
);
}

export default Login;

```

Notes.tsx će biti izmenjen tako da prikazuje poruku dobrodošlice korisniku:

```

function Notes() {
  const { user, setUser } = useUser();

  // ostatak je isti kao i ranije

  const logout = () => {
    setUser("");
    localStorage.removeItem("user");
  };

  return (
    <div>
      <div className="header">
        <h2 className="welcome-message">Dobrodosao {user}!</h2>
        <button onClick={logout} className="logout-button">

```

```
        Odjavi se
      </button>
    </div>

    /* ostatak je isti kao i ranije */
  </div>
);
}

export default Notes;
```

Ono što je još ostalo je da se zaštiti ruta za listu beležaka tako što će se korisnici preusmeriti na stranicu za prijavu ukoliko nisu prijavljeni kao i da korisnike koji pokušaju da pristupe stranici za prijavu a već su prijavljeni, preusmeri na stranicu sa listom beleški. Ovo se može postići korišćenjem `useNavigate` kuke:

```
// Notes.tsx
function Notes() {
  const { user, setUser } = useUser();

  const navigate = useNavigate();

  useEffect(() => {
    if (!user) navigate("/");
  }, [user, navigate]);

  // ostatak je isti kao i ranije
}

export default Notes;
```

```
// Login.tsx
function Login() {
  const { user, setUser } = useUser();

  const navigate = useNavigate();

  useEffect(() => {
    if (user) navigate("/notes");
  }, [user, navigate]);

  // ostatak je isti kao i ranije
}

export default Login;
```

Angular

U *Angular*-u, koristiće se `@angular/router` biblioteka. Prilikom inicijalizacije projekta ova biblioteka je već bila instalirana pa nema potrebe za dodatnom instalacijom. Za početak, biće generisane dve komponente koje će predstavljati stranice za prijavu i beleške. Ove komponente će biti generisane korišćenjem Angular CLI komande: `ng generate component login` i `ng generate component notes`. Ove komponente će biti dodate u `app-routing.module.ts` fajl:

```
import { NgModule } from '@angular/core';
import { RouterModule, Routes } from '@angular/router';
import { LoginComponent } from './login/login.component';
import { NotesComponent } from './notes/notes.component';

const routes: Routes = [
  { path: '', component: LoginComponent },
  { path: 'notes', component: NotesComponent },
];

@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule],
})
export class AppRoutingModule {}
```

Sada, kada su rute definisane, sadržaj `app.component.html` fajla će biti izmenjen tako da sadrži samo `<router-outlet/>` element a prethodni sadržaj `app.component.html` i `app.component.ts` fajlova će biti prebačen u `notes.component.html` i `notes.component.ts` fajlove.

Dalje, biće kreiran servis koji će obrađivati funkcionalnost prijave i čuvati informacije o korisniku. Ovaj servis će biti kreiran korišćenjem Angular CLI komande: `ng generate service login`. Ovaj servis će izgledati ovako:

```
import { Injectable } from '@angular/core';
import { BehaviorSubject } from 'rxjs';

@Injectable({
  providedIn: 'root',
})
export class LoginService {
  private userSubject = new BehaviorSubject<string | null>(
    localStorage.getItem('user')
  );
  user$ = this.userSubject.asObservable();

  login(username: string): void {
    localStorage.setItem('user', username);
  }
}
```

```

    this.userSubject.next(username);
  }

  logout(): void {
    localStorage.removeItem('user');
    this.userSubject.next(null);
  }

  get userValue(): string | null {
    return this.userSubject.value;
  }
}

```

U fajl `login.component.html` će biti dodat jednostavni formular za prijavu:

```

<div class="login-page-wrapper">
  <div class="login-container">
    <h1 class="login-title">Prijava</h1>
    <form [formGroup]="loginForm" (submit)="login()" class="login-form">
      <input
        type="text"
        placeholder="Korisnicko ime"
        formControlName="username"
        class="login-input"
      />
      <input
        type="password"
        placeholder="Sifra"
        formControlName="password"
        class="login-input"
      />
      <button type="submit" class="login-button">Prijavi se</button>
    </form>
  </div>
</div>

```

A u fajl `login.component.ts` će biti dodata funkcija `login` koja će biti pozvana kada korisnik klikne na dugme za prijavu koja će proveriti da li su uneti podaci za prijavu, sačuvati korisnika korišćenjem `LoginService` servisa i preusmeriti korisnika na stranicu sa listom beleški korišćenjem `Router` servisa:

```

import { Component } from '@angular/core';
import { Router } from '@angular/router';
import { LoginService } from '../login.service';
import { FormBuilder, Validators } from '@angular/forms';

@Component({
  selector: 'app-login',
  templateUrl: './login.component.html',
  styleUrls: ['./login.component.css'],

```

```

})
export class LoginComponent {
  constructor(
    private loginService: LoginService,
    private router: Router,
    private formBuilder: FormBuilder
  ) {}

  loginForm = this.formBuilder.group({
    username: ['', Validators.required],
    password: ['', Validators.required],
  });

  login() {
    const { username, password } = this.loginForm.value;

    if (this.loginForm.invalid || !username || !password) {
      alert('Morate uneti korisnicko ime i lozinku.');
      return;
    }

    this.loginService.login(username);
    this.router.navigate(['/notes']);
  }
}

```

Fajl `notes.component.html` će biti izmenjen tako da prikazuje poruku dobrodošlice korisniku i prikazuje dugme za odjavu:

```

<div class="header">
  <h2 class="welcome-message">Dobrodošao {{ user$ | async }}!</h2>
  <button (click)="logout()" class="logout-button">Odjavi se</button>
</div>

<!-- ostatak je isti kao i ranije -->

```

Fajl `note.component.ts` je izmenjen tako da sadrži polje `user$` i metod `logout`:

```

// ostatak je isti kao i ranije

user$ = this.loginService.user$;

logout(): void {
  this.loginService.logout();
  this.router.navigate(['']);
}

```

Korišćen je *async pipe* kako bi se prikazalo ime korisnika. Ovaj operator će se automatski pretplatiti (*eng. subscribe*) na `user$` promenljivu i prikazati ime korisnika čim se ona promeni bez manuelnog pozivanja `subscribe` metode.

Kao i kod *React.js* aplikacije, ostalo je da se zaštiti ruta za listu beležaka tako što će se korisnici preusmeriti na stranicu za prijavu ukoliko nisu prijavljeni kao i da korisnike koji pokušaju da pristupe stranici za prijavu a već su prijavljeni, preusmeri na stranicu sa listom beleški. Ovo se postiže korišćenjem `CanActivate` interfejsa. Prvo će biti kreiran `AuthGuard` koji implementira `CanActivate` interfejs. Koristi se *Angular CLI* komanda `ng generate guard auth`. Zatim menjamo sadržaj `auth.guard.ts` fajla:

```
import { Injectable } from '@angular/core';
import { CanActivate, Router } from '@angular/router';
import { LoginService } from '../login.service';

@Injectable({
  providedIn: 'root',
})
export class AuthGuard implements CanActivate {
  constructor(private loginService: LoginService, private router: Router) {}

  canActivate(): boolean {
    if (this.loginService.userValue) {
      return true;
    }
    this.router.navigate(['/']);
    return false;
  }
}
```

Dalje, u `app-routing.module.ts` fajlu, dodaje se `canActivate` atribut za rutu koju treba zaštititi:

```
// ostatak je isti kao i ranije
{ path: 'notes', component: NotesComponent, canActivate: [AuthGuard] },
// ostatak je isti kao i ranije
```

I u `login.service.ts` fajlu će biti dodat *getter* za `userSubject` koji proverava da li je korisnik prijavljen:

```
// ostatak je isti kao i ranije
get userValue(): string | null {
  return this.userSubject.value;
}
```

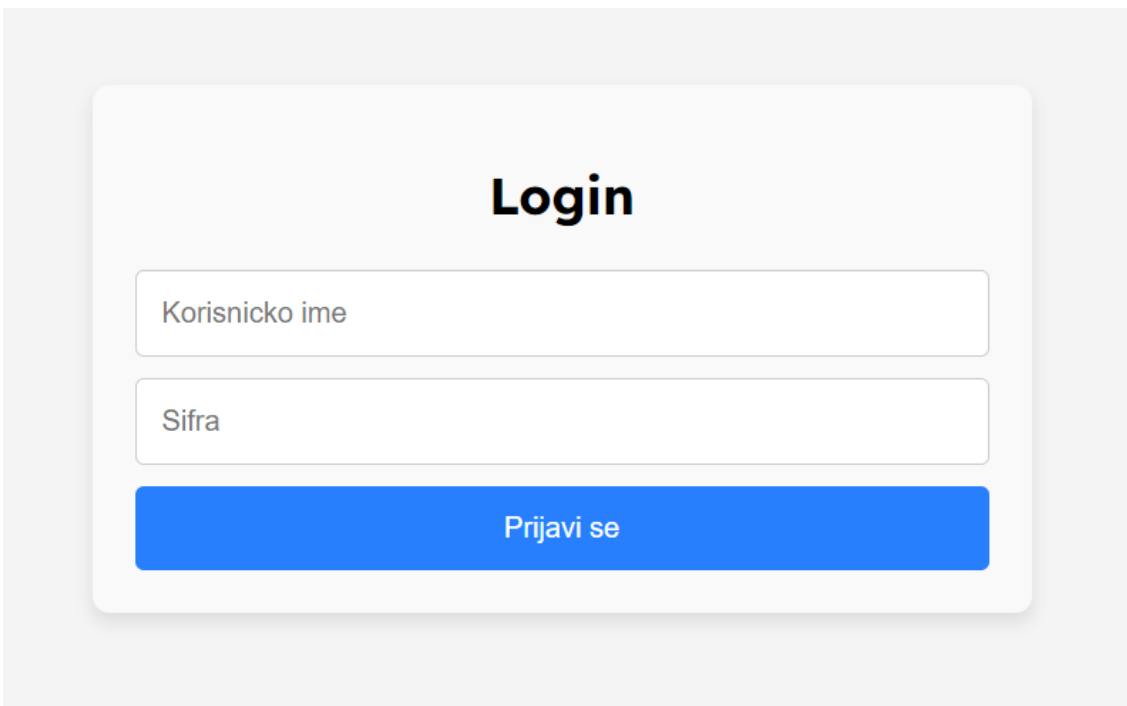
Konačno, u `login.component.ts` fajlu, dodaje se provera da li je korisnik već prijavljen i ako jeste, preusmerava se na stranicu sa listom beleški:

```
// ostatak je isti kao i ranije
constructor(
  private loginService: LoginService,
```



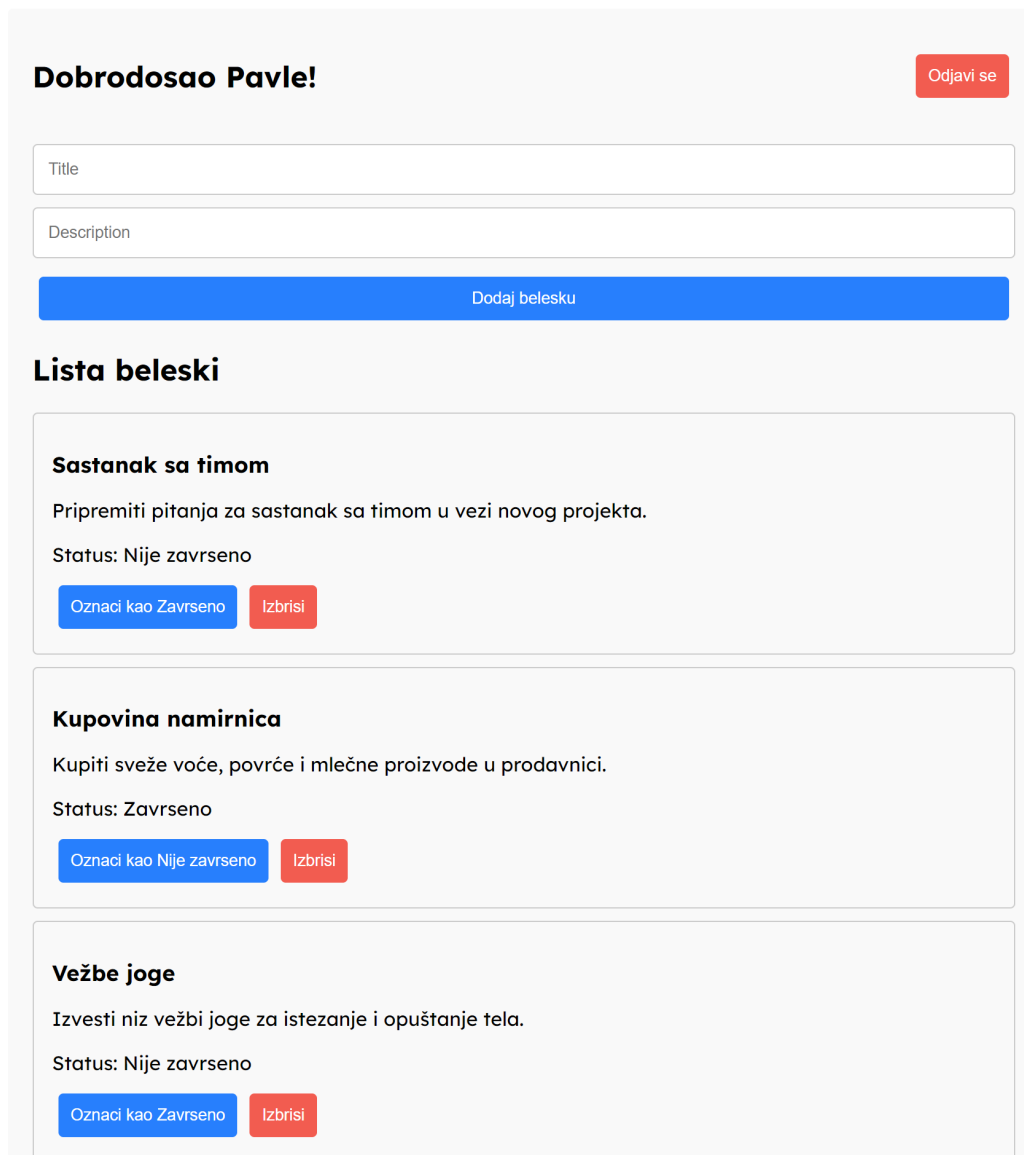
```
private router: Router,  
private formBuilder: FormBuilder  
) {  
  if (this.loginService.userValue) {  
    this.router.navigate(['/notes']);  
  }  
}  
// ostatak je isti kao i ranije
```

Nakon ovih izmena, strana za prijavu će izgledati kao na slici 9.3 dok će strana sa listom beleški izgledati kao na slici 9.4.



The image shows a login form with a white background and rounded corners, centered on a light gray background. At the top, the word "Login" is written in a bold, black, sans-serif font. Below the title, there are two input fields stacked vertically. The first field is labeled "Korisnicko ime" (Username) and the second is labeled "Sifra" (Password). Both fields have a light gray border and a white background. Below the input fields is a prominent blue button with the text "Prijavi se" (Login) in white, centered on the button.

Slika 9.3: Strana za prijavu



Slika 9.4: Strana sa listom beleški

9.5 Rezime

Iako su dve aplikacije za čuvanje beleški, jedna napravljena u *React.js*-u a druga u *Angular*-u, identične iz ugla korisnika, postoji nekoliko razlika u implementaciji. Ove razlike su uglavnom posledica različitih pristupa u razvoju aplikacija. *React.js* je biblioteka, dok je *Angular* radni okvir. Ovo znači da *React.js* nudi samo osnovne funkcionalnosti i potrebno je instalirati dodatne biblioteke kako bi se postigla ista funkcionalnost kao u potpunom radnom okviru poput *Angular*-a. S druge strane, *Angular* nudi mnogo više funkcionalnosti koje su ugrađene direktno u sam radni

okvir. Ove razlike su najuočljivije u implementaciji rutiranja i navigacije, gde *Angular* nudi ove funkcionalnosti „iz kutije”, dok je za *React.js* potrebno koristiti neku od biblioteka za rutiranje (u ovom projektu, to je konkretno `react-router-dom`).

Jedna od ključnih razlika između *React.js* i *Angular* je i pristup stilu programiranja. Za *React*, ne postoji konkretan stil programiranja ili set najboljih praksi koji je propisan od strane tima koji je razvio biblioteku. Ovo može biti prednost jer pruža veću fleksibilnost, ali takođe može dovesti do nekonzistentnosti u većim timovima. S druge strane, *Angular* dolazi sa jasno definisanim stilom programiranja i najboljim praksama koje je postavio sam tim koji je razvio okvir. Ovo može olakšati saradnju u timovima i osigurati konzistentnost koda.

Kada je reč o učenju, mnogi smatraju da je *React.js* lakši za početnike jer je primarno usmeren na kreiranje korisničkih interfejsa. S druge strane, *Angular* može biti izazovniji za nove korisnike zbog svoje sveobuhvatne arhitekture i složenijih koncepta poput dvosmernog povezivanja podataka, injekcije zavisnosti i modula.

Takođe, u ovom projektu to nije pokazano, ali u slučaju da je aplikacija dosta složenija, i postoji potreba za praćenjem globalnog stanja, za *React.js* bi bolja opcija bila korišćenje neke biblioteke za upravljanje stanjem kao što je *Redux* ili *Zustand* umesto *Context API* zbog performansi, dok za *Angular* ovo nije potrebno jer on nudi ugrađeno rešenje za upravljanje stanjem kroz *observables* i *service*. Kao i malopre, prednost ovoga za *React.js* je što postoji fleksibilnost u izboru biblioteka za rutiranje i upravljanje stanjem, omogućavajući programerima da odaberu onu koja najbolje odgovara specifičnim potrebama projekta dok je mana što je potrebno više vremena za istraživanje i učenje ovih biblioteka.

Glava 10

Zaključak

U ovom master radu pružen je detaljan pregled *React.js*-a, koji se smatra jednim od najdominantnijih biblioteka za veb u savremenom vremenu. Uvodna motivacija i istorijski pregled ne samo da postavljaju temelje za razumevanje ključnih aspekata *React.js*-a, već i definišu i nagoveštavaju sadržaj koji će biti obrađen u radu, kao i razloge postojanja svih tih aspekata unutar *React.js*-a. Detaljno su analizirani *Virtuelni DOM*, *JSX*, komponente, kuke, rutiranje, podrška za *TypeScript*, alati za pakovanje modula poput *Vite*-a i *Webpack*-a, i na kraju, izvršeno je poređenje *React.js*-a sa *Angular*-om kako bi se istakle prednosti i mane *React.js*-a u odnosu na drugu popularnu biblioteku, tj. radni okvir, za veb.

Jedan od glavnih razloga za dominaciju *React.js*-a u svetu veb razvoja jeste njegova sposobnost da pruži fleksibilne, skalabilne i modularne rešenja za izgradnju korisničkih interfejsa. Kroz korišćenje koncepta *Virtuelnog DOM*-a, omogućeno je efikasno ažuriranje i renderovanje komponenata. *JSX* nudi intuitivan način za pisanje UI komponenata, dok *kuke* olakšavaju upravljanje stanjem i efektima, čime se pojednostavljuje i ubrzava razvoj aplikacija. Fleksibilnost *React.js*-a i njegov bogat ekosistem, uključujući alate kao što su *Webpack* i *Vite*, ističu se kao ključne prednosti. U poređenju sa *Angular*-om, naglašena je fleksibilnost i modularnost koje *React* može da pruži. *React.js* nastavlja da se razvija i prilagođava, ali njegove osnovne karakteristike i principi ostaju nepromenjeni, čime se potvrđuje njegova važnost u svetu veb razvoja.

Nakon proučavanja ovog rada, čitaocu bi trebalo da bude omogućeno razumevanje osnovnih i naprednijih koncepta *React.js*-a, čime će on biti opremljen za kreiranje *frontend* dela veb aplikacija koristeći ovu biblioteku. *React.js* nastavlja da se razvija i prilagođava, ali njegove osnovne karakteristike i principi ostaju nepromenjeni, čime se potvrđuje njegova važnost u svetu veb razvoja.

Literatura

- [1] Anthony Accomazzo, Nate Murray, and Ari Lerner. *Fullstack React: The Complete Guide to ReactJS and Friends*. Fullstack.io, 2017.
- [2] Philip Bille. A survey on tree edit distance and related problems. *Theoretical Computer Science*, 337(1-3):217–239, 2005.
- [3] Facebook. Zvanična dokumentacija biblioteke *React.js*. on-line at: <https://reactjs.org/>.
- [4] Adam Freeman. *Pro React 16*. Apress, 2019.
- [5] Sacha Greif. State of *Javascript*. on-line at: <http://stateofjs.com/>.
- [6] Michael Jackson. React Router. on-line at: <https://reactrouter.com/>.
- [7] Anđelka Zečević Nikola Ajzenhamer. Programiranje za veb, 2022. on-line at <https://matfpvbe.gitlab.io/vezbe/>.
- [8] React Team. Novi sajt biblioteke *React.js*. on-line at: <https://react.dev/>.
- [9] React Team. React Functional Components, 2019. on-line at: <https://legacy.reactjs.org/docs/hooks-faq.html#should-i-use-hooks-classes-or-a-mix-of-both>.
- [10] React Team. Introducing the New JSX Transform, 2020. on-line at: <https://reactjs.org/blog/2020/09/22/introducing-the-new-jsx-transform.html>.
- [11] TypeScript Team. Typescript JSX transpilation to Javascript, 2021. on-line at: <https://www.typescriptlang.org/docs/handbook/jsx.html>.
- [12] typicode. JSON Server. on-line at: <https://github.com/typicode/json-server>.

Biografija autora

Pavle Cvejović rođen je 10. decembra 1999. godine, u Pančevu. Osnovne studije završio je na Matematičkom fakultetu Univerziteta u Beogradu, smer Informatika 2022. godine sa prosečnom ocenom 9,18 gde je stekao zvanje diplomiranog informatičara. Nakon toga, nastavio je svoje obrazovanje upisavši master studije na istom fakultetu.

Pavle je fakultet upisao vođen snažnim interesovanjem za matematiku i algoritme. Tokom studija, počeo je da radi u firmi **QCerris**, gde je po prvi put bio izložen brojnim veb tehnologijama kao što su *Django*, *FastAPI*, *React.js* i *Next.js*. Kroz rad sa ovim tehnologijama, u njemu se počela razvijati strast prema veb razvoju kako je on sve dublje i dublje učio o svakoj od ovih tehnologija. Posebno ga je privukao *React.js* zbog njegove fleksibilnosti, moćnog ekosistema i sposobnosti da se prilagodi različitim projektima. Ova strast i prepoznavanje potencijala *React.js*-a inspirisali su ga da izabere ovu temu za svoj master rad.

Pored formalnog obrazovanja i profesionalnog angažmana, Pavle je nezavisno radio na brojnim veb projektima. Njegovi projekti su pružili alate i resurse koji su omogućili drugima da lakše kreiraju i razvijaju svoje projekte koristeći prethodno pomenute tehnologije.