Metode poboljšanja izvođenja React mrežnih aplikacija

Milić, Nikola

Master's thesis / Diplomski rad

2020

Degree Grantor / Ustanova koja je dodijelila akademski / stručni stupanj: **Josip Juraj Strossmayer University of Osijek, Faculty of Humanities and Social Sciences / Sveučilište Josipa Jurja Strossmayera u Osijeku, Filozofski fakultet**

Permanent link / Trajna poveznica: https://urn.nsk.hr/urn:nbn:hr:142:650167

Rights / Prava: In copyright / Zaštićeno autorskim pravom.

Download date / Datum preuzimanja: **2024-08-08**

FILOZOFSKI FAKULTET
SVEUČILIŠTE JOSIPA JURJA STROSSMAYERA U OSIJEKU

Repository / Repozitorij:

FFOS-repository - Repository of the Faculty of Humanities and Social Sciences Osijek

zir.nsk.hr

SVEUČILIŠTE JOSIPA JURJA STROSSMAYERA U OSIJEKU

dabar
DIGITALNI AKADEMSKI ARHIVI I REPOZITORIJI

Sveučilište J.J. Strossmayera u Osijeku

Filozofski fakultet

Dvopredmetni diplomski studij Nakladništva i informacijske tehnologije

Nikola Milić

Metode poboljšanja izvođenja React mrežnih aplikacija

Diplomski rad

Mentor doc.dr.sc. Tomislav Jakopec

Osijek, 2020

Sveučilište J.J. Strossmayera u Osijeku
Filozofski fakultet Osijek
Odsjek za informaijske znanosti
Dvopredmetni diplomski studij Nakladništva i informacijske tehnologije

---

Nikola Milić

Metode poboljšanja izvođenja React mrežnih aplikacija

Diplomski rad

Društvene znanosti, informacijske i komunikacijske znanosti,
informacijsko i programsko inženjerstvo

doc.dr.sc. Tomislav Jakopec

Osijek, 2020

Prilog: Izjava o akademskoj čestitosti i o suglasnosti za javno objavljivanje

Obveza je studenta da donju Izjavu vlastoručno potpiše i umetne kao treću stranicu završnog odnosno diplomskog rada.

IZJAVA

Izjavljujem s punom materijalnom i moralnom odgovornošću da sam ovaj rad samostalno napravio te da u njemu nema kopiranih ili prepisanih dijelova teksta tuđih radova, a da nisu označeni kao citati s napisanim izvorom odakle su preneseni.
Svojim vlastoručnim potpisom potvrđujem da sam suglasan da Filozofski fakultet Osijek trajno pohrani i javno objavi ovaj moj rad u internetskoj bazi završnih i diplomskih radova knjižnice Filozofskog fakulteta Osijek, knjižnice Sveučilišta Josipa Jurja Strossmayera u Osijeku i Nacionalne i sveučilišne knjižnice u Zagrebu.

U Osijeku, datum 30.09.2020.

Nikola Milić, 0122220625
ime i prezime studenta, JMBAG

Sažetak

Rad obrađuje temu poboljšanja izvođenja React aplikacija. React je JavaScript biblioteka koja služi za izgradnju korisničkih sučelja. React aplikacije sačinjene su od zasebnih komponenti - JavaScript klasa i funkcija, a s obzirom na to mogu se podijeliti na klasne i funkcijske komponente. Komponente se kroz svoj životni ciklus u aplikaciji mogu više puta ažurirati, njihovi parametri stanja i svojstava se mijenjaju, a prilikom svake od tih promjena dolazi do novog iscrtavanja komponente. U tom području može doći do usporenog izvođenja pojedinih komponenti te samim time cjelokupne aplikacije. Navedene vrste komponenti razlikuju se u načinu korištenja i ažuriranja parametara stanja, načinu upravljanja životnim ciklusom te u samoj sintaksi. Upravljanje životnim ciklusom komponente i ažuriranje parametara stanja komponente ključno je za postizanje visoke razine efikasnosti izvođenja komponente. Tomu se može pristupiti korištenjem metoda životnog ciklusa `shouldComponentUpdate()` i `componentDidUpdate()` kada su u pitanju klasne komponente, te korištenjem React *hook* funkcija kada su u pitanju funkcijske kako bi se izbjegla nepotrebna iscrtavanja komponenti. Iscrtavanja se mogu izbjeći i primjenom `React.PureComponent` funkcija kod klasnih komponenti te `React.memo()` funkcija višeg reda kod funkcijskih. Ostale metode koje se mogu primijeniti za postizanje boljeg izvođenja komponenti su primjena uvjetne logike i smanjivanje ukupnog broja pozivanja funkcija koje mogu dovesti do iscrtavanja komponente. Kako bi se uopće moglo pristupiti upravljanju životnim ciklusom komponenti nužno je primjenjivati principe ne-mutiranja podataka. Izvođenje pojedinih komponenti kao i cjelokupne aplikacije može se ispitati alatima React DevTools i Google Lighthouse.

Ključne riječi: *React, komponente, izvođenje, poboljšanje*

Sadržaj

Uvod ... 1
Alati za ispitivanje izvođenja React aplikacije ... 2
React DevTools ... 2
Components ... 3
Profiler ... 3
Prikaz životnog ciklusa (Flamegraph) ... 4
Redani prikaz (Ranked) ... 4
Prikaz korisničkih interakcija ... 5
Lighthouse ... 5
React komponente ... 7
Životni ciklus komponente ... 9
Klasne komponente ... 9
Funkcijske komponente ... 10
Usporedba klasne i funkcijske komponente ... 11
Metode poboljšanja izvođenja aplikacije ... 14
Metoda shouldComponentUpdate() ... 14
Metoda componentDidUpdate() ... 16
React.memo() ... 16
React.PureComponent ... 17
Ne-mutirajući podaci ... 18
Spread operatori za nizove i objekte ... 20
Uvjetno iscrtavanje ... 21
Izjave if / if else ... 21
Uvjetni operatori ... 22
Logički AND (&&) operator ... 23
Smanjivanje broja poziva funkcija ... 23
Primjena metoda poboljšanja izvođenja ... 25
Funkcionalnost aplikacije ... 26
Struktura aplikacije ... 26
Područja poboljšanja izvođenja ... 27
Dohvaćanje podataka iz lokalne memorije ... 27

Ažuriranje podataka u lokalnoj memoriji 28
Dodavanje i uklanjanje objekata 29
Ažuriranje objekata 30
Iscrtavanje komponenti 32
Pretraživanje 35
Zaključak 38
Literatura 39

Uvod

React (React.js) je Javascript biblioteka otvorenog koda zaštićena MIT licencom[3] čija je svrha izgradnja korisničkih sučelja. Kod je dostupan u *facebook/react* repozitoriju na Githubu (https://github.com/facebook/react). Za potrebe ovog diplomskog rada korištenja je inačica Reacta v16.13.1 (19.03.2020.) koja je u trenutku pisanja ovog rada ujedno i najnovija. React biblioteka razvijena je od strane Facebooka. Glavna karakteristika Reacta je njegova zasnovanost na komponentama. Komponente imaju višestruku svrhu – razdvajanje dijelova korisničkog sučelja i razdvajanje samoga koda. Svaka komponenta u sebi može sadržavati više potkomponenti, a samim time može biti dio neke komponente ili se može pojavljivati u više različitih komponenti. Ovakav pristup omogućuje intuitivnu izgradnju složenih korisničkih sučelja, no on u isto vrijeme zahtijeva dobro poznavanje metoda i praksi koje povećavaju efikasnost izvođenja aplikacije. U slučaju da se takve metode ne primjenjuju, korisničko sučelje aplikacije može postati sporo. Postoje brojni načini na koje se može povećati efikasnost izvođenja React aplikacije, a cilj ovog rada je prikazati, analizirati te opisati ponašanje osnovnih metoda i principa koji pomažu u postizanju navedenoga cilja. Osim poboljšanja brzine izvođenja same aplikacije, primjena principa obrađenih u ovom radu može imati pozitivan utjecaj na razvojni proces same aplikacije pošto neki od njih pomažu u pronalasku i otklanjanju sintaksnih pogrešaka u kodu.

Rad je podijeljen na nekoliko poglavlja. Prvo poglavlje obrađuje alate za ispitivanje izvođenja React aplikacije. Navedeni alati mogu pomoći u otklanjanju problema u vidu brzine izvođenja. Ti alati pojavljuju se u vidu proširenja za mrežne preglednike, razvojnih alata integriranih u mrežne preglednike te metoda sadržanih unutar React biblioteke.

Iduće poglavlje obrađuje temu React komponenti - gradivnih blokova same aplikacije. Kontrolom životnog ciklusa komponente moguće je precizno odrediti ponašanje određene komponente, njene reakcije na promjene unutar same aplikacije, kako i kada će se komponenta prikazati te brojne druge mogućnosti. Primjena dobrih praksi u ovom području osnova je za efikasno izvođenje cjelokupne aplikacije. Kako se React komponente mogu pojavljivati u obliku JavaScript funkcije i klase koje zahtijevaju različite pristupe upravljanju životnim ciklusom komponente, važno je dobro poznavanje tih različitosti.

[3] facebook/react. URL: https://github.com/facebook/react/blob/master/LICENSE (20-08-2020)

Središnji dio rada obrađuje nekoliko najvažnijih metoda koje se mogu primijeniti u poboljšanju izvođenja aplikacije. Aplikacija je dostupna u Github repozitoriju nmilic96/diplomski_rad (https://github.com/nmilic96/diplomski_rad) te na adresi https://nmilic96.github.io/diplomski_rad/. Odnosi se na React metode, komponente i funkcije kao što su metode shouldComponentUpdate() i componentDidUpdate(), React.memo() komponenta višeg reda i React.PureComponenta vrsta komponente, zatim principi ne-mutiranja podataka, načini postizanja uvjetnog iscrtavanja i načini smanjivanja broja pozivanja funkcija.

Poslijednji dio rada prikazuje primjenu navedenih metoda poboljšanja izvođenja aplikacije na primjeru stvarne aplikacije. Ovdje se prikazuju i međuovisnosti navedenih metoda i komponenti aplikacije u postizanju visoke efikasnosti izvođenja aplikacije.

Svi primjeri iz mrežnog preglednika napravljeni su u pregledniku Google Chrome, inačica Chrome 84.

Alati za ispitivanje izvođenja React aplikacije

Ispitivanje izvođenja React mrežnih aplikacije može se vršiti putem alata dostupnih u mrežnim preglednicima. U narednim primjerima prikazat će se testiranja u dvije faze – ispitivanje lokalne, nezapakirane inačice aplikacije i testiranje zapakirane inačice aplikacije spremne za postavljanje na mrežno mjesto (production build). Najtočnije podatke o stvarnoj kvaliteti izvođenja aplikacije moguće je dobiti testiranjem zapakirane inačice. React u razvojnoj konzoli prikazuje brojna upozorenja, primjerice, ako se u komponentu uvoze komadi koda koji se zapravo ne koriste u samoj komponenti. Ova upozorenja mogu biti od velike važnosti za uočavanje problema kod izvođenja aplikacije, no ona također sa sobom nose određenu težinu i čine aplikaciju većom i samim time sporijom. Zapakirana inačica aplikacije ne sadrži ta upozorenja.[4]

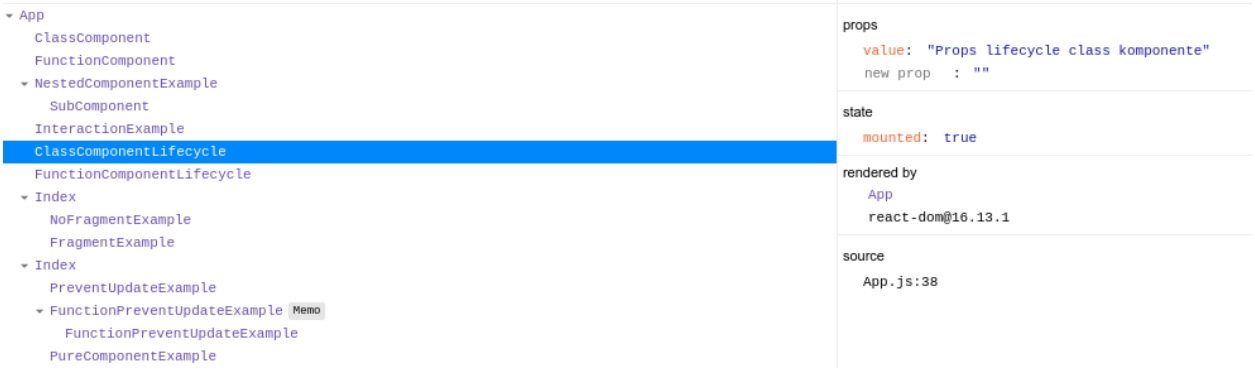
React DevTools

Alat React DevTools najbolje je koristiti u prvoj fazi ispitivanja – ispitivanje nezapakirane inačice aplikacije. Alat je razvijen od strane Facebooka i dostupan je kao proširenje za Google Chrome i Mozilla Firefox mrežne preglednike. Izvorni kod proširenja dostupan je u facebook/react repozitoriju na Githubu. Alat se sastoji od dva dijela – kartice Components i Profiler.

[4] Optimizing Performance. URL: https://reactjs.org/docs/optimizing-performance.html (20-08-2020)

Components

Kartica Components nudi uvid u hijerarhijsku strukturu aplikacije, odnosno prikaz svih React komponenti koje sačinjavaju aplikaciju. Za svaku komponentu aplikacije moguće je dobiti uvid u njene parametre stanja i svojstava, hook funkcije, te podatke o izvoru iscrtavanja komponente, kao i o nadređenoj komponenti u kojoj se određena komponenta nalazi.



Slika 1: Hijerarhijska struktura aplikacije u Components prikazu

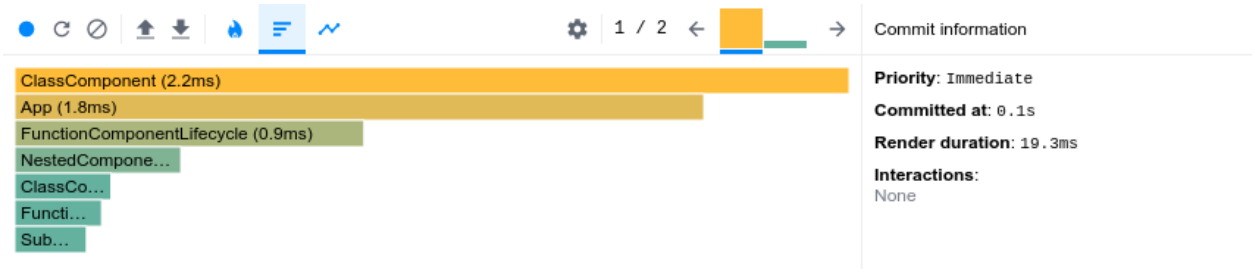
Profiler

Kartica Profiler daje detaljan uvid u životni ciklus komponenti, vremenske točke u kojima se određene komponente iscrtavaju i vrijeme potrebno za iscrtavanje komponente, promjene parametara stanja i svojstava, korisničke interakcije i ostale korisne informacije. Tako je moguće vrlo jednostavno prepoznati komponente koje se nepotrebno više puta iscrtavaju ili one komponente čije bi se vrijeme iscrtavanja moglo smanjiti. Profiler kartica podijeljena je u tri kategorije:

Prikaz životnog ciklusa (Flamegraph)

- Moguće je prikazati životni ciklus komponenti za svako iscrtavanje aplikacije.
- Slika 2 prikazuje prvo iscrtavanje aplikacije, od ukupno dva iscrtavanja.
- Iscrtavanje je trajalo 19.3ms.

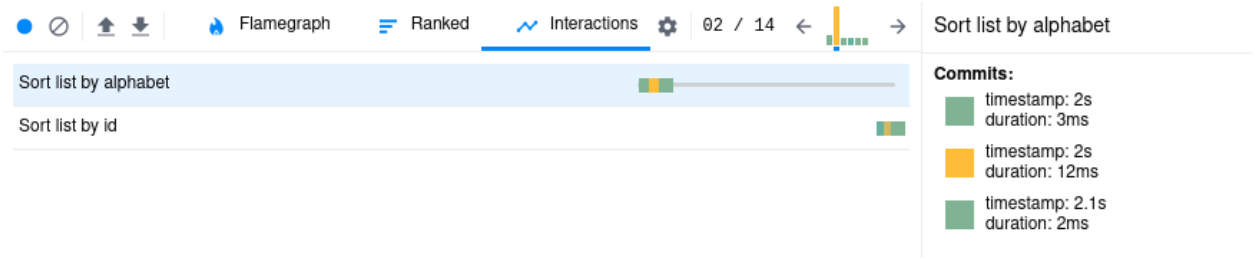
Redani prikaz (Ranked)



Slika 3: Redani prikaz iscrtavanja

- Prikazuju se iscrtavanja komponenti u prvom iscrtavanju aplikacije.
- Iscrtavanja se redaju od najdužeg prema najkraćem.

Prikaz korisničkih interakcija



Slika 4: Prikaz korisničkih interakcija

- Za korištenje zahtijeva primjenu React API-a za praćenje korisničkih interakcija – interaction tracing API, koji je dostupan u paketu `scheduler`.
- Prikazuju se korisničke interakcije za one funkcije kod kojih se primjenjuje `scheduler` API.
- Daje uvid u trajanje korisničkih interakcija, moguće je vidjeti prilikom kojeg iscrtavanja aplikacije se dogodila interakcija te u kojem trenutku.

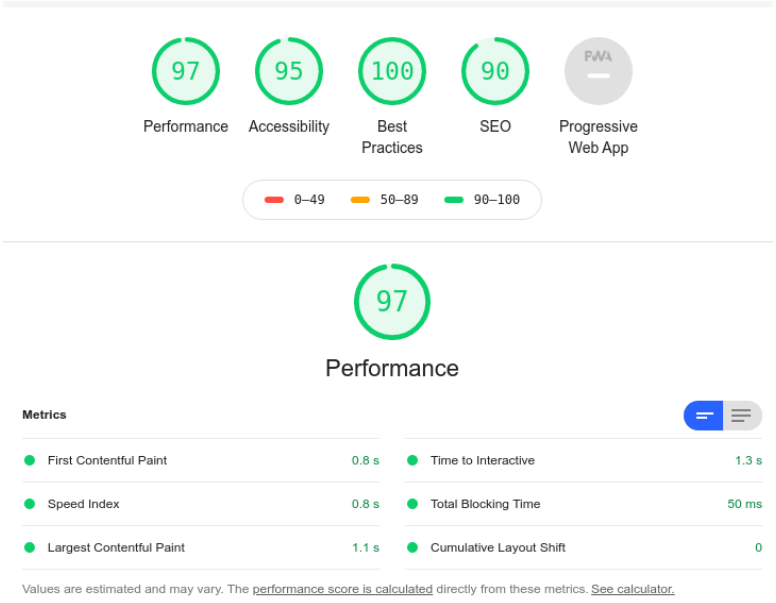
```
const sortByAlphabet = () => {
      trace('Sort list by alphabet', performance.now(), () => {
            setSortType({
                  type: 'alphabet'
            });
      });
};
```

Primjer 1. Korištenje interaction tracing API-a.

Lighthouse

Lighthouse je automatizirani alat otvorenog koda, dostupan u GoogleChrome/lighthouse repozitoriju na Githubu (https://github.com/GoogleChrome/lighthouse). Verzija alata korištena u ovom radu je Lighthouse 6.0.0. Namjena alata je testiranje izvođenja aplikacije, pristupačnosti, optimizacije za mrežne tražilice, optimizacija progresivnih mrežnih aplikacija i ostalo. Alat je moguće pokrenuti unutar razvojnih alata Google Chrome preglednika, kroz razvojnu konzolu preglednika ili kao Node modul. Alat provodi testiranja u pet kategorija:

- Brzina izvođenja (Performance)
- Pristupačnost (Accessibility)
- Primjena najboljih praksi (Best Practices)
- Optimizacija za mrežne tražilice (SEO)
- Optimizacija progresivne mrežne aplikacije (Progressive Web App)



Slika 5: Prikaz rezultata testiranja u alatu Lighthouse

Kategorija rezultata brzina izvođenja od posebnog je značaja za testiranje u narednim primjerima

te će se ovaj rad usredotočiti na ovu kategoriju. Rezultat u ovoj kategoriji računa se na osnovu šest mjerenja:

- **Prvi prikaz sadržaja (First Contentful Paint)** – mjeri se vrijeme potrebno mrežnog pregledniku da iscrta prvi djelić DOM sadržaja. Rezultat se računa tako što se vrijeme potrebno za ocrtavanje prvog sadržaja uspoređuje s vremenima stvarnih mrežnih mjesta.[5]
- **Brzinski indeks (Speed Index)** – mjeri brzinu vizualnog prikaza sadržaja u vremenu učitavanje stranice. Lighthouse stvara videozapis učitavanja mrežne stranice u pregledniku i radi izračun vizualnog napretka kroz kadrove videozapisa.[6] Konačni izračun se vrši tako što se uspoređuje histogram trenutnog kadra s histogramom posljednjeg kadra.[7]
- **Prikaz najvećeg sadržaja (Largest Contentful Paint)** – Mjeri se vrijeme potrebno da se na zaslonu iscrta najveći vidljivi slikovni ili tekstni blok.[8]
- **Vrijeme do interaktivnosti (Time to Interactive)** – mjeri se vrijeme potrebno da aplikacija postane u potpunosti interaktivna. Potpuna interaktivnost podrazumijeva da aplikacija prikazuje uporabljiv sadržaj, da su *event handleri* registrirani za većinu elemenata stranice te da stranica na korisničke radnje reagira unutar 50 milisekundi.[9]
- **Ukupno vrijeme čekanja (Total Blocking Time)** – ukupni vremenski period u kojemu aplikacija ne može primati korisničke unose kao što su klikovi, dodiri zaslona ili pritisci tipki na tipkovnici.[10]
- **Ukupan pomak rasporeda (Culmulative Layout Shift)** – radi se o mjeri vizualne stabilnosti aplikacije, daje informacije o tome koliko često se korisnici susreću s neočekivanim promjenama i pomacima u rasporedu aplikacije.[11]

Navedene kategorije vrijedno je imati na umu kroz proces razvoja pošto ukazuju na najčešće izvore lošeg izvođenja aplikacije. Može ih se smatrati ciljevima prema kojima se treba usmjeriti u razvojnom procesu. Korisnicima je bitno u što kraćem vremenskom roku prikazati potpun sadržaj

[5]First Contentful Paint, 2019. URL: https://web.dev/first-contentful-paint/ (19-08-2020)
[6]Speed Index, 2019. URL: https://web.dev/speed-index/ (19-08-2020)
[7]Speedline. URL: https://github.com/paulirish/speedline (19-08-2020)
[8]Walton, P. Largest Contentful Paint (LCP), 2019. URL: https://web.dev/lcp/ (19-08-2020)
[9]Isto.
[10]Total Blocking Time, 2019. URL: https://web.dev/lighthouse-total-blocking-time/ (19-08-2020)
[11]Walton P; Mihajlija, M. Culmulative Layout Shift (CLS), 2019. URL: https://web.dev/cls/ (19-08-2020)

aplikacije te im omogućiti aktivnu interakciju s aplikacijom uz izbjegavanje naglih promjena u vizualnom rasporedu aplikacije u procesu učitavanja.

React komponente

Komponente (React.Component) predstavljaju osnovni gradivni blok React aplikacija. Komponente služe za odvajanje koda u manje dijelove koji predstavljaju zasebne dijelove korisničkog sučelja koji se mogu ponovno koristiti. Komponente u sebi mogu sadržavati više pod-komponenti. Pojedina komponenta može biti opisana kao Javascript klasa ili Javascript funkcija. Komponenta može primati parametre svojstava i vratiti React JSX elemente koji opisuju što bi se trebalo prikazati na zaslonu.12

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

class ClassComponent extends Component {
      render() {
                  return (
                        <div className="component">
                              <h3>Primjer 1: Osnovna klasna
komponenta</h3>
                              <p>Ovo je klasna komponenta</p>
                        </div>
                  );
      }
}

export default ClassComponent;
```

Primjer 2. Klasna komponenta

```
import React from 'react'

function FunctionComponent() {
```

[12]Components and Props. URL: https://reactjs.org/docs/components-and-props.html (20-08-2020)

```
      return (
                  <div className="component">
                        <h3>Primjer 2: Osnovna funkcijska
komponenta</h3>
                        <p>Ovo je funkcijska komponenta</p>
                  </div>
      )
}

export default FunctionComponent
```

Primjer 3. Funkcijska komponenta

Kako je vidljivo iz navedenih primjera, između dvije vrste komponenti postoje neke razlike. Prva vidljiva razlika je razlika u samoj sintaksi. Funkcijska komponenta obična je JavaScript funkcija koja prihvaća parametar svojstava kao argument i vraća React JSX element. S druge strane, klasna komponenta zahtjeva proširivanje iz **React.Component** klasne definicije te definiranje metode **render()** unutar same komponente.

Životni ciklus komponente

Životni ciklus React komponente odvija se kroz četiri stadija: inicijalizacija, uglavljivanje, ažuriranje i uklanjanje. U stadiju inicijalizacije, React komponenta postavlja početne vrijednosti parametara stanja i svojstava. U stadiju uglavljivanja, React pretvara vrijednosti koje vraća React komponenta u stvarne DOM elemente. Stadij ažuriranja događa se kada komponenta prima nove vrijednosti u vidu **p**arametara stanja i svojstava. Prilikom promjene vrijednosti, poziva se metoda **render()**. Zadnji stadij je uklanjanje komponente prilikom kojeg se komponenta uklanja iz DOM-a. U ovom stadiju mogu se prekinuti svi mrežni zahtjevni i praćenja JavaScript događaja koji se vežu uz tu komponentu. [13]

[13]Gonzales, A. Life Cycle Hooks in React, 2019. URL: https://medium.com/@anisgonzales/life-cycle-hooks-in-react-9f1690bcd91b (20-08-2020)

Klasne komponente

Do nedavno su klasne komponente predstavljale jedini mogući način definiranja komponente koja može imati svoj parametar stanja te koja ima pristup metodama životnog (*lifecycle* metode). Te metode pružaju mogućnost definiranja slijeda događaja vezanih uz tu komponentu u odnosu na to kada je ona iscrtana, nadograđena ili uklonjena.[14] One mogu pokretati novi ili zamijeniti stari kod na određenim vremenskim točkama u procesu. Ove metode mogu se podijeliti u četiri glavne skupine prema stadijima životnog ciklusa komponente:

Mounting (uglavljivanje komponente) – metode koje se pozivaju kada je instanca komponente stvorena i uglavljena u DOM.[15]

- constructor()
- render()
- componentDidMount()

Updating (ažuriranje komponente) – ažuriranje komponente uzrokovano promjenama parametara stanja ili svojstava.[16]

- render()
- componentDidUpdate()

Unmounting (uklanjanje komponente) – postoji samo jedna metoda koja se poziva kada se komponenta uklanja iz DOM-a.[17]

- ComponentWillUnmount()

Error Handling (upravljanje pogreškama) – metode koje se pozivaju u slučaju kada se dogodi pogreška tijekom iscrtavanja, u nekoj od metoda životnog ciklusa ili u konstruktoru neke od podkomponenti. [18]

- componentDidCatch()

[14]Copes, F. The React Handbook, str. 118. URL: https://flaviocopes.nyc3.digitaloceanspaces.com/react-handbook/react-handbook.pdf
[15]React.Component. URL: https://reactjs.org/docs/react-component.html (20-08-2020)
[16]Isto.
[17]Isto.
[18]Isto.

Ovo nije potpun popis metoda za upravljanje životnim ciklusom. U kasnijim primjerima bit će prikazane metode koje su posebno bitne za poboljšanje brzine izvođenja aplikacije.

Korištenjem ovih metoda komponentama se pridodaje dinamičnost te se postiže visok stupanj kontrole nad istima. Sve navedene metode vrlo su bitne u procesu poboljšanja izvođenja React aplikacije.

Funkcijske komponente

React inačica 16.8 po prvi puta uvodi React Hooks funkcije. *Hook* je funkcija koja funkcijskoj komponenti daje pristup parametrima stanja i metodama životnog ciklusa. Prije njihove pojave, funkcijske komponente mogle su primati samo parametre svojstava, te nisu mogle sadržavati nikakav oblik unutarnjeg parametra stanja ili kontrole nad životnim ciklusom komponente.[19] Zanimljiva karakteristika *hook* funkcija je mogućnost stvaranja vlastitih funkcija koje, primjerice, mogu pomoći u međusobnom dijeljenju logike kroz više komponenti.[20] React sadrži nekoliko ugrađenih *hook* funkcija koje se mogu primijeniti za kontrolu parametara stanja i životnog ciklusa funkcijske komponente:

useState - omogućuje dodjeljivanje parametara stanja funkcijskoj komponenti.

useEffect - omogućuje izvršavanje takozvanih nuspojava (*side effect*) u funkcijskoj komponenti koje imaju istu svrhu kao metode componentDidMount(), componentDidUpdate(), i componentWillUnmount(). Te funkcionalnosti mogu biti dohvaćanje podataka, praćenje mreže ili promjene u DOM-u. [21]

Usporedba klasne i funkcijske komponente

Sljedeći primjer prikazuje osnovnu primjenu metoda životnog ciklusa u klasnoj komponenti:

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

class ClassComponentLifecycle extends Component {
     constructor(props) {
          super(props);
```

[19]Hooks at a Glance. URL: https://reactjs.org/docs/hooks-overview.html (21-08-2020)
[20]Eluwande, Y; Murray, N. An Introduction to Hooks in React, 2018. URL: https://www.newline.co/fullstack-react/articles/an-introduction-to-hooks-in-react/ (21-08-2020)
[21] Using the Effect Hook. URL: https://reactjs.org/docs/hooks-effect.html (21-08-2020)

```
            this.state = {
                mounted: false
            };
        }



        componentDidMount() {
            this.setState({ mounted: true });
        }

        render() {
            if (this.state.mounted) {
                return (
                    <div className="component">
                        <h3>Primjer 3: Životni ciklus klasne
                        komponente</h3>
                        <p>status - komponenta je učitana</p>
                    </div>
                );
            } else {
                return null;
            }
        }
}

export default ClassComponentLifecycle;
```

Primjer 4. Životni ciklus klasne komponente

- U konstruktoru komponente definira se početni parametar stanja te komponente – objekt koji sadrži svojstvo `mounted: false`.
- Metoda `componentDidMount()` u sebi sadrži metodu `setState()` koja ažurira parametar stanja komponente s novim vrijednostima – `mounted: true`.

- Metoda componentDidMount() poziva se pri uglavljivanju komponente u DOM te se time događa ažuriranje parametra stanja.
- render() metoda sadrži if else izjavu koja u ovisnosti o parametru stanja određuje što će vratiti ta komponenta. U ovom slučaju, komponenta vraća vrijednost tek kada je uglavljena u DOM.

Istu funkcionalnost možemo postići primjenom **useState** i **useEffect** *hook* funkcija u funkcijskoj komponenti:

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

function FunctionComponentLifecycle() {
      const [ mounted, setMounted ] = useState(false);

      useEffect(() => {
                  setMounted(true);
      }, []);

      if (mounted) {
                  return (
                        <div className="component">
                              <h3>
                                    Primjer 4: Životni funkcijske
                                    komponente
                              </h3>
                              <p>status - komponenta je učitana</p>
                        </div>
                  );
            } else {
                  return null
      }
}
```

```
export default FunctionComponentLifecycle;
```

Primjer 5. Životni ciklus funkcijske komponente

- Na najvišoj razini komponente definira se varijabla koja sadrži trenutni parametar stanja – `mounted` i funkciju koja ažurira taj parametar – `setMounted()`, a čija početna vrijednost je `false`.
- Funkcija `setMounted()` poziva se unutar `useEffect` *hook* funkcije te ona ujedno predstavlja efekt te funkcije.
- `useEffect` *hook* funkcija će se pozvati svaki put kada se parametri stanja ili svojstava komponente ažuriraju. Kako bi se ostvarilo ponašanje kao kada se koristi `componentDidMountMetoda()`, funkciji je potrebno dodati drugi argument - `[ ]` (prazan niz). Argument sadrži niz uvjeta za pozivanje `useEffect` funkcije. Ako je ta lista prazna, funkcija se pokreće samo nakon prvog iscrtavanja nadređene komponente.

Kada se ove dvije vrste komponenti usporede, jasno je da je za poboljšanje izvođenja istih potrebno primijeniti različite pristupe kako bi se postigli istovjetni rezultati. Primjeri opisani u ovom poglavlju prilično su jednostavni i sastavni su dio React aplikacije. Opisane metode životnog ciklusa važno je razumjeti kako bi se moglo uspješno pristupiti poboljšanju izvođenja složenijih komponenti koje će biti prikazane u narednim primjerima.

Metode poboljšanja izvođenja aplikacije

Metoda shouldComponentUpdate()

Metoda koja Reactu daje do znanja imaju li trenutne promjene parametara stanja ili svojstava komponente ikakav utjecaj na podatke koje vraća ta komponenta. Glavna svrha metode je poboljšanje izvođenja komponente. Zadana funkcionalnost klasnih komponenti je da se iscrtavaju na svaku promjenu u navedenim parametrima, što u većini slučajeva predstavlja željeno ponašanje.[22] Metoda `shouldComponentUpdate()` poziva se prije iscrtavanja u trenutku primanja novih parametara stanja i svojstava, u stadiju ažuriranja komponente. U doslovnom smislu, ova metoda određuje bi li se komponenta trebala ažurirati. Zadana vrijednost koju metoda vraća je `true`. Metoda omogućuje usporedbu `this.props` sa `nextProps` te `this.state` sa `nextState`, odnosno

[22]React.Component. URL: https://reactjs.org/docs/react-component.html#shouldcomponentupdate (21-08-2020)

usporedbu trenutnih parametara stanja i svojstava s onim nadolazećim. Ako usporedba navedenih parametara vrati vrijednost false, komponenta se neće ponovno iscrtati.[23] Vraćanje false vrijednosti sprječava izvođenje UNSAFE_componentWillUpdate(), render() i componentDidUpdate() metoda. Korištenjem ove metode moguće je postići visoku razinu kontrole nad brojem ukupnih iscrtavanja komponente. Kod korištenja ove metode nije poželjno vršiti duboke usporedbe jednakosti među parametrima. Primjer duboke usporedbe jednakosti je pretvaranje niza u JSON string objekt i usporedba jednakosti takvih string objekata.[24]

```
export class PreventUpdateExample extends Component {
      constructor(props) {
                  super(props);
      }

      shouldComponentUpdate(nextProps) {
                  return nextProps !== this.props
      }

      render() {
                  return (
                        <div>
                              vrijednost: {this.props.value}
                        </div>
                  )
      }
}

export default PreventUpdateExample
```

Primjer 6. Korištenje shouldComponentUpdate() metode u klasnoj komponenti

[23]Isto.
[24]Isto.

- Komponenta u navedenom primjeru sadrži shouldComponentUpdate() metodu u kojoj se provjerava jednakost this.props i nextProps parametara. U slučaju da su parametri različiti, metoda će vratiti vrijednost true te će se komponenta ponovno iscrtati. Dok god su parametri isti, komponenta se neće iscrtati.

Metoda componentDidUpdate()

Metoda componentDidUpdate() poziva se odmah nakon ažuriranja komponente. Slično kao i kod shouldComponentUpdate() metode, ova metoda omogućava usporedbu prevState i prevProps parametara s onim trenutnim. setState() funkcija se u ovoj metodi može pozvati odmah, ali mora biti zamotana unutar uvjetne izjave. U suprotnom će se stvoriti beskonačno ažuriranje parametra stanja.

```
componentDidUpdate(prevState) {
      if (this.state !== prevState) {
            this.setState({value: 'update'})
      }
}
```

Primjer 7. Korištenje componentDidUpdate() metode

- setState() funkcija poziva se samo ako this.state i prevState parametri nemaju istu referencu.

React.memo()

Kada su u pitanju funkcijske komponenta, funkcionalnost metode shouldComponentUpdate() može se oponašati korištenjem React.memo() komponente višeg reda koja vrši memoizaciju vrijednosti. Međutim, ova komponenta može pratiti samo promjene parametara svojstava funkcijske komponente. useState i useHook funkcije će i dalje rezultirati novim iscrtavanjem komponente svaki put kada se izvrše.[25] Kao i u pitanju shouldComponentUpdate() metode, glavna svrha ove metode je poboljšanje izvođenja

[25]React Top-Level API. URL: https://reactjs.org/docs/react-api.html#reactmemo (21-08-2020)

komponente.[26] Zadano ponašanje React.memo() komponente je napraviti plitku usporedbu složenih objekata sadržanih u parametru svojstava. React.memo() komponenta kao argument prima komponentu čiji se parametar svojstava želi memoizirati. U slučaju da je potreban viši stupanj kontrole nad usporedbom parametra svojstava, komponenti je moguće proslijediti prilagođenu funkciju kao drugi argument.[27]

```
import React from 'react'

function FunctionPreventUpdateExample(props) {
          return (
               <div>
                    vrijednost: {props.value}
               </div>
          )
}

function areEqual(prevProps, nextProps) {
          return prevProps.value === nextProps.value
}

export default React.memo(FunctionPreventUpdateExample, areEqual)
```

Primjer 8. Korištenje React.memo komponente višeg reda

- Komponenta FunctionPreventUpdateExample ugrađuje se u React.memo() komponentu višeg reda koja izvršava memoizaciju parametra svojstava.
- Funkcija areEqual() radi provjeru jednakosti stare i nove vrijednosti parametra svojstava te se dodaje React.memo() komponenti kao drugi argument.
- Za razliku od shouldComponentUpdate() metode, novo iscrtavanje će se dogoditi ako funkcija areEqual() vrati vrijednost true.

[26]Isto.
[27]Isto.

Dodjeljivanjem drugog argumenta funkcionalnost React.memo() komponenta vrlo je slična funkcionalnosti shouldComponentUpdate(). Ako se ostavi u zadanom stanju, sličnija je React.PureComponent komponenti.

React.PureComponent

React.PureComponent vrsta je komponente slična React.Component komponenti, odnosno klasnim komponentama. Razlikuju se u tome što React.PureComponent komponenta zadano primjenjuje shouldComponentUpdate() metodu kako bi vršila plitke usporedbe parametara stanja i svojstava. U slučaju da render() metoda ponovno vrati isti rezultat za iste parametre stanja i svojstava, ta komponenta neće se ponovno iscrtavati. [28]

```
export class PureComponentExample extends React.PureComponent { ... }


export default PureComponentExample
```

Primjer 9. Korištenje React.PureComponent komponente

- Za razliku od obične klasne komponente, kod ove vrste komponente proširuje se React.PureComponent.
- shouldComponentUpdate() metoda ugrađena je u komponentu. Iscrtavanje komponente dogodit će se samo ako je nova vrijednost parametra svojstava drukčija od one stare.

Ova vrsta komponente ograničena je na slučajeve kada parametri stanja ili svojstava sadrže jednostavne podatkovne strukture. Razlog tomu je nemogućnost zadane shouldComponentUpdate() metode da izvršava duboke usporedbe. Navedenom problemu može se doskočiti primjenom principa ne-mutiranja podataka za uspješno izvršavanje plitkih usporedbi ugniježđenih podatkovnih struktura.

Ne-mutirajući podaci

Ne-mutirajuća varijabla ona je varijabla čija se vrijednost ne može promijeniti nakon što je stvorena. Primjer takve varijable je string. Kada se mijenja, zapravo se stvara novi string koji

[28]React Top-Level API. URL: https://reactjs.org/docs/react-api.html#reactpurecomponent (21-08-2020)

se pridodaje varijabli istog imena. Dakle, varijabla je nepromjenjiva, a kako bi se ažurirala njena vrijednost potrebno je stvoriti novu varijablu.[29]

Isti princip vrijedi za objekte i nizove. Kako bi se napravila promjena u nizu potrebno je stvoriti novi niz nadovezivanjem starog niza s novom vrijednosti. Objekt se nikada ne ažurira, već se stvara nova inačica tog objekta.[30] Navedeni principi vrijede za parametre stanja ili svojstava komponente. U klasnoj komponenti se parametar stanja ne bi trebao ažurirati izravno već putem `setState()` metode.

Nepromjenjivost podataka posebno je bitna kada se radi o plitkim usporedbama podataka, kao što je slučaj kod `React.PureComponent` komponente. Svaki objekt ima svoju referencu, odnosno ključ koji ga razlikuje od drugih objekata. Izravnom promjenom podataka unutar objekta ne stvara se novi objekt. Ako se ne stvori novi objekt s novom referencom, prilikom plitke usporedbe promijenjeni objekt i dalje će biti prepoznat kao isti, jer on to zapravo i jest.[31]

```
constructor(props) {
      super(props);
      this.state = {
                  list: ['list_item']
      }
}

componentDidMount() {
      const list = this.state.list;
      list.push('new_list_item');
      this.setState({list: list})
}
```

Primjer 10. Primjena principa ne-mutiranja podataka

[29]Copes, F. The React Handbook, str. 55. URL: https://flaviocopes.nyc3.digitaloceanspaces.com/react-handbook/react-handbook.pdf
[30]Isto.
[31]Grzywaczewski, M. Pros and Cons of using immutability with React.js, 2015. URL: https://reactkungfu.com/2015/08/pros-and-cons-of-using-immutability-with-react-js/ (22-08-2020)

- U navedenom primjeru `list` niz unutar parametra stanja se izravno mijenja dodavanjem nove varijable u taj niz. Prilikom izravne promjene vrijednost objekta se mijenja dok referenca objekta ostaje ista.
- `setState()` funkcija postavlja novu vrijednost `list` svojstvu parametra stanja, ali ta vrijednost i dalje ima istu referencu te ju komponenta prilikom plitke usporedbe prepoznaje kao isti objekt. Iz tog se razloga neće izvršiti novo iscrtavanje komponente, unatoč tome što je vrijednost parametra stanja zapravo drukčija.

U većini slučajeva, parametri stanja i svojstava pojavljuju se u obliku objekata i nizova. Kako bi se u radu s istima primijenili principi ne-mutiranja, moguće je koristiti neke JavaScript metode:

Spread operatori za nizove i objekte

Spread operatori (...) omogućuju dodavanje elemenata u niz, spajanje nizova i objekata te raščlanjivanje niza u argumente poziva funkcije.[32] Korištenjem *spread* operatora se stvara novi niz, a stari niz ostaje netaknut, što omogućava primjenu principa ne-mutiranja podataka:

```
componentDidMount() {
      const list = this.state.list;
      const newList = [...list, ['new list item']];
      this.setState({list: newList});
}
```

Primjer 11. Izbjegavanje mutacije podataka pri ažuriranju niza unutar parametra stanja

- Za razliku od prethodnog primjera, u ovom primjeru nije došlo do izravnog mijenjanja svojstva `this.state.list` pošto se korištenjem *spread* operatora stvorio niz.
- Novi niz je kroz `setState()` funkciju primijenjen u `list` svojstvo parametra stanja.

Spread operatori mogu se primijeniti i u radu s objektima. Sintaksa je vrlo slična kao i u radu s nizovima:

```
componentDidMount() {
      const object = this.state.object;
```

[32]Derek, A. How to use the spread operator (…) in JavaScript, 2019. URL: https://medium.com/coding-at-dawn/how-to-use-the-spread-operator-in-javascript-b9e4a8b06fab (22-08-2020)

```
        const newState = {...object, {new_propery: 2}};
        this.setState(newState);
}
```

Primjer 12. Izbjegavanje mutacije podataka pri ažuriranju objekta unutar parametra stanja

- Korištenjem *spread* operatora objektu se pridodaje novo svojstvo te se stvara nova inačica tog objekta.

Primjena principa ne-mutiranja podataka kritična je za uspješno izvođenje metoda životnog ciklusa., te je posebno bitna kad su u pitanju komponente kod kojih se želi smanjiti broj iscrtavanja. Izravno mutiranje parametara stanja ili svojstava ne mora nužno dovesti do neuspješnog rada aplikacije ili pojedinih komponenti, ali je velika mogućnost da hoće.

Uvjetno iscrtavanje

Uvjetno iscrtavanje React komponenti moguće je postići primjenom if, if else ili switch izjava te korištenjem logičkih i uvjetnih operatora. Na taj se način komponente mogu iscrtavati samo kada su zadovoljeni traženi uvjeti, čime je moguće izbjeći nepotrebna iscrtavanja komponenti.[33] Primjena uvjetne logike posebno je korisna kada se koristi zajedno s metodama životnog ciklusa komponente. Kod uvjetnog iscrtavanja komponente najčešće se koriste:

Izjave if / if else

Primjer korištenja ovakve izjave može se pronaći u ranijim primjerima ovog rada (vidi Primjer 3 i 4).

```
if (this.state.mounted) {
      return (
                  <div className="component">
                        <h3>Primjer 3: Životni ciklus klasne
                              komponente</h3>
                        <p>status - komponenta je učitana</p>
                  </div>
            );
```

[33]Conditional Rendering. URL: https://reactjs.org/docs/conditional-rendering.html (22-08-2020)

```
} else {
            return null;
}
```

Primjer 13. Korištenje if else izjave za uvjetno iscrtavanje

- Komponenta će se iscrtati tek kada vrijednost svojstva this.state.mounted iznosi true.
- U suprotnom se neće iscrtati ništa, zato što komponenta vraća vrijednost null.

Ista takva logika može se primijeniti za prikazivanje grafike učitavanja:

```
if (this.state.mounted) {
      return (
                  <div className="component">...</div>
            );
} else {
            return <Loading />;
}
```

Primjer 14. korištenje if else izjave za prikaz grafike učitavanja

- Komponenta će se prikazati tek kada svojstvo this.state.mounted iznosi true.
- Dok god je ta vrijednost false, bit će prikazana komponenta Loading koja služi za prikaz grafike učitavanja.

Uvjetni operatori

if else izjave se prema zadanom ne mogu primijeniti unutar JSX sintakse. U slučajevima kada se želi postići takva funkcionalnost mogu se primijeniti uvjetni operatori:

```
render() {
      return (
                  <div>
                        {this.state.mounted ?
                              <div className="component">...</div>
                        :
                              <Loading />
```

```
                    }
                </div>
            );
}
```

Primjer 15. Korištenje uvjetnih operatora za prikaz grafike učitavanja

- Kao i u prethodnom primjeru, `Loading` prikazuje se sve dok svojstvo `this.state.mounted` ne iznosi `true`.

Logički AND (&&) operator

Obično se koristi s *Booleovim* vrijednostima. Uvjet je ispunjen samo ako tražene vrijednosti iznose `true`.[34] Može se koristiti umjesto `if` izjave ili s logičkim operatorima unutar JSX sintakse:

```
render() {
    return (
            <div>
                {this.state.mounted && this.state.list.length > 0 ?
                    <div className="component">...</div>
                :
                    <Loading />
                }
            </div>
    );
}
```

Primjer 16. Korištenje operatora && zajedno sa uvjetnim operatorom

- U navedenom primjeru `&&` operator koristi s uvjetnim operatorom.
- Tako se može ostvariti iscrtavanje komponente samo kada ona zadovoljava više uvjeta.
- U navedenom primjeru vrijednost svojstva `this.state.mounted` mora biti `true` te dužina niza `this.state.list` mora biti veća od 0.

[34]Logical AND (&&). URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Logical_AND (22-08-2020)

Smanjivanje broja poziva funkcija

Debouncing je naziv metode kojom se može smanjiti ukupan broj pozivanja funkcije. Ova je metoda omogućava upravljanje događajima koji se događaju jedan za drugim. Ako je vremenski razmak između događaja manji od nekog zadanog vremena, onaj prvi događaj se neće izvesti. Kada se dogodi neki događaj nakon zadanog vremenskog intervala, funkcija se izvodi. Drugim riječima, *debouncing* određuje da se funkcija ne može ponoviti dok nije prošao zadani vremenski period.[35]

```
export const debounce = (func, wait) => {
      let timeout;
            return function executedFunction(...args) {
                  const later = () => {
                  clearTimeout(timeout);
                  func(...args);
            };

            clearTimeout(timeout);
            timeout = setTimeout(later, wait);
      };
};
```

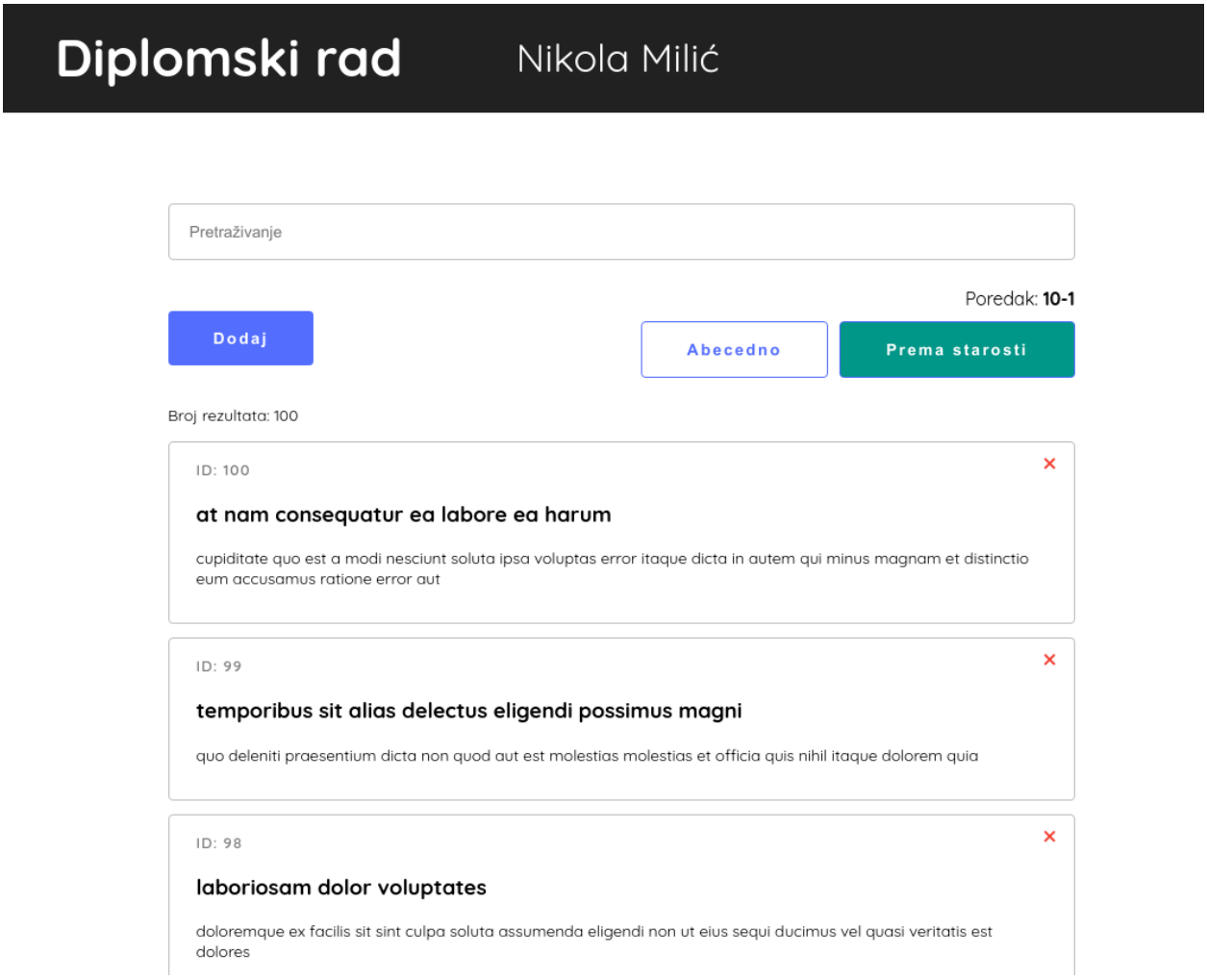
Primjer 17. Debouncing funkcija

- Funkcija debounce() sadrži dva argumenta – func i wait. Funkcija debounce() prima funkciju za koju se želi napraviti *debounce* (argument func). Drugi argument predstavlja vremenski interval (argument wait).
- Pozivom debounce() funkcije pokreće se odbrojavanje (setTimeout()). Ako odbrojavanje dosegne zadano vrijeme (wait), debounce() funkcija vraća željenu funkciju pozvanu kroz argument.

[35]Understanding Debouncing && Throttling, 2020. URL: https://medium.com/javascript-in-plain-english/understanding-debouncing-throttling-2a0a5e9cc74a (23-08-2020)

Primjena metoda poboljšanja izvođenja

Ovaj diplomski rad prati primjer aplikacije koja prikazuje primjenu svojstava navedenih kroz rad. Primjer je stvoren isključivo kako bi se predstavile obrađene metode te ona nema neku stvarnu uporabu. Radi se o aplikaciji koja izlistava JSON podatke pohranjene u lokalnoj memoriji korisnika. JSON je sačinjen od 100 objekata stvorenih pomoću alata JSONPlaceholder REST API-a čija je namjena stvaranje lažnih podataka koji se zatim mogu koristiti u svrhu ispitivanja i stvaranja prototipa. API je dostupan u typicode/jsonplaceholder Github repozitoriju (https://github.com/typicode/jsonplaceholder).[36]



Slika 6: Prikaz aplikacije

[36]JSONPlaceholder. URL: https://github.com/typicode/jsonplaceholder (24-08-2020)

Funkcionalnost aplikacije

Funkcionalnost aplikacije je sljedeća:

- Prilikom prvog iscrtavanja aplikacije podaci se pohranjuju u lokalnu memoriju korisnika u obliku JSON *stringa*, osim ako u lokalnoj memoriji korisnika već postoje podaci.
- Komponenta `Index` prilikom iscrtavanja dohvaća podatke iz lokalne memorije. Oni se iz JSON *stringa* pretvaraju u JavasScript niz sačinjen od objekata. Svaki objekt u nizu sadrži svojstva `userId`, `id`, `title`, i `body`.
- Niz objekata pretvara se u DOM. Svaki objekt predstavlja jednu „objavu". Na svakoj objavi prikazani su identifikacijski broj (`id`), naslov (`title`) i opis (`body`).
- Klikom na objavu, ona se označava kao aktivna te se omogućuje uređivanje naslova i opisa, prilikom čega se podaci u lokalnoj memoriji ažuriraju s novim nizom.
- Svaka objava sadrži gumb za brisanje. Klikom na taj gumb objava se uklanja iz niza te se podaci u lokalnoj memoriji ažuriraju s novim nizom.
- U aplikaciji postoji gumb za dodavanje nove objave. Klikom na gumb objava se dodaje u niz, na sam početak, te se i u ovom slučaju vrši ažuriranje lokalnih podataka.
- Postoji mogućnost redanja objava. Objave se može redati abecedno ili prema identifikacijskom broju.
- Objave je moguće pretraživati unosom riječi u polje za pretraživanje. Provjerava se postoji li tražena riječ u `title` i `body` svojstvima objekata koji predstavljaju objave.

Struktura aplikacije

Aplikacija je sačinjena od nekoliko komponenti, a ima sljedeću strukturu:

- `App` – glavna komponenta koja predstavlja samu React aplikaciju
  - `Index` – indeksna komponenta aplikacije koja sadrži sve ostale komponente
    - `Search` – sadrži polje za pretraživanje niza
    - `Sort` – komponenta koja sadrži gumbe za odabir redanja popisa objava

- `Post` – komponenta objave
- `Loader` – komponenta koja sadrži grafički prikaz učitavanja

Područja poboljšanja izvođenja

Dodavanje, brisanje, uređivanje i pretraživanje objava radnje su prilikom kojih dolazi do ažuriranja cijelog niza podataka. Kako početna dužina niza iznosi sto objekata, te radnje mogu biti poprilično spore u vidu efikasnosti izvođenja te je kod njih posebno bitno primijeniti metode navedene u radu.

Naredni primjeri navode radnje u aplikaciji kod kojih se primjenjuju metode poboljšanja izvođenja.

Dohvaćanje podataka iz lokalne memorije

Podaci koji se koriste u aplikaciji nalaze se u `posts.json` datoteci. Podaci se prilikom prvog iscrtavanja `Index` komponente ažuriraju u `data` parametar stanja te se pohranjuju u lokalnu memoriju korisnika. Ako su prilikom prvog iscrtavanja podaci prisutni u lokalnoj memoriji, dohvaćaju se iz lokalne memorije i postavljaju kao vrijednost `data` parametra stanja. Razlog pohranjivanja i korištenja podataka iz lokalne memorije aplikacije je to što pohrana podataka u lokalnu memoriju omogućuje naknadno uređivanje istih.

```
const [ data, setData ] = useState(null);
useEffect(() => {
      if (!localStorage.getItem('data')) {
            localStorage.setItem('data',
JSON.stringify(postsData));
            setData(postsData)
      } else {
            setData(JSON.parse(localStorage.getItem('data')))
      }
}, []);
```

Primjer 18. Dohvaćanje podataka iz lokalne memorije

- Prilikom prvog iscrtavanja `Index` komponente uvjetnom izjavom `if else` provjerava se postojanje podataka u lokalnoj memoriji korisnika.

- Ako podaci nisu prisutni u lokalnoj memoriji, u memoriju se pohranjuju podaci iz posts.json datoteke, te se također postavljaju kao vrijednost data parametra stanja.
- Ako podaci jesu prisutni, kao vrijednost data parametra stanja postavljaju se podaci dohvaćeni iz lokalne memorije.

U navedenom primjeru primjenjuje se uvjetna logika te se upravlja životnim ciklusom komponente – radnja se vrši na samom početku životnog ciklusa i ne ponavlja se sve dok korisnik ne osvježi prozor aplikacije. Na ovaj se način izbjegava stalno ažuriranje i dohvaćanje podataka iz lokalne memorije. Umjesto toga, prilikom rada s aplikacijom, korisnik radi s podacima unutar data parametra stanja.

Ažuriranje podataka u lokalnoj memoriji

Podaci u lokalnoj memoriji ažuriraju se prilikom svake promjene data parametra stanja, uz uvjet da je zadan action parametar stanja. Tako se izbjegava ažuriranje podataka u lokalnoj memoriji prilikom pretraživanja (koje sadrži setData() funkciju), ali se omogućava ažuriranje prilikom dodavanja, brisanja ili ažuriranja objekata unutar data niza.

```
const [ data, setData ] = useState(null);
const [ action, setAction ] = useState(null);
useEffect(() => {
      if (action) {
            localStorage.setItem('data', JSON.stringify(data));
      }
      return () => {
            setAction(null);
      };
}, [ data ]);
```

Primjer 19. Ažuriranje podataka u lokalnoj memoriji

- useEffect *hook* funkcija sadrži drugi argument kojim se određuje da se poziva samo kada se ažurira data parametar stanja.
- Uvjetna izjava if određuje da se podaci u lokalnoj memoriji ažuriraju samo ako action parametar stanja ima zadanu vrijednost.
- useEffect funkcija vraća funkciju kojom se action parametru stanja zadaje vrijednost null.

Ažuriranje podataka primjer je primjene uvjetne logike (sadrži li `action` parametar stanja vrijednost) i upravljanja životnim ciklusom komponente (poziva se nakon ažuriranja `data` parametra stanja).

Dodavanje i uklanjanje objekata

Prilikom dodavanja i uklanjanja objekata primjenjuje se princip ne-mutiranja podataka kako bi se ostvarilo uspješno iscrtavanje niza u DOM. Za dodavanje objekata u niz odgovorna je funkcija `addNewPost()` koja se poziva klikom na gumb „Dodaj”.

```
const addNewPost = () => {
      let newPost = {
            id: handlePostId(data),
            title: 'Nova objava',
            body: ''
      };

      let newData = [ newPost, ...data ];
            setData(newData);
            setAction('add');
};
```

Primjer 20. Primjena principa ne-mutiranja podataka u dodavanju niz

- Mutiranje podataka prilikom dodavanja objekta u niz, odnosno stvaranja nove objave izbjegava se stvaranjem novog niza primjenom *spread* operatora.
- U varijabli `newData` spaja varijabla `newPost` (koja predstavlja objekt koji se dodaje u niz) s `data` parametrom stanja. Time se stvara novi niz.
- Funkcija `setData(newData)` sadrži novi niz kao argument te tako ažurira `data` parametar stanja.
- Funkcija `setAction('add')` određuje radnju koja se izvršava prilikom dodavanja novog objekta u niz.

Funkcija `removePost()` odgovorna je za uklanjanje podataka iz niza te primjenjuje ne-mutiranje na vrlo sličan način kao i funkcija iz prethodnog primjera:

```
const removePost = (index) => {
      let filteredPosts = data.filter((item) => item.id !==
                              index);
      setData(filteredPosts);
      setAction('remove');
};
```

Primjer 21. Ne-mutiranje podatka prilikom uklanjanja objekta iz niza

- Objekt se niza uklanja tako što se Javascript filter metodom vraćaju svi objekti u nizu koji nemaju id svojstvo jednako id svojstvu objekta koji se želi ukloniti. Tako se stvara novi niz koji sadrži sve objekte osim ciljanog objekta.
- Novi niz se setData(filteredPosts) funkcijom ažurira u data parametar stanja.
- Kao i u prethodnom primjeru, setAction('remove') funkcija određuje izvršenju radnju.

Ne-mutiranjem podataka parametara stanja osigurava se njihovo uspješno iscrtavanje u DOM te izvođenje metoda životnog ciklusa komponenti, čije je uspješno izvođenje nužno u smanjivanju broja iscrtavanja Post komponenti. Dodavanjem vrijednosti action parametru stanja omogućava se uvjetno ažuriranje podataka u lokalnoj memoriji.

Ažuriranje objekata

Ažuriranje objekata vrši se prilikom uređivanja naslova i opisa objave. Naslov i opis objave se mogu urediti samo kada active parametar stanja post komponente sadrži vrijednost true. Prilikom uređivanja objave, mijenjaju se svojstva parametra stanja – title i body. Kada se promjene ta svojstva, ažurira se vrijednost data parametra stanja Index komponente. Nakon ažuriranja tog parametra, podaci se pohranjuju u lokalnu memoriju. Cijeli proces nešto je složeniji i vrši se kroz nekoliko koraka:

```
<input
      type="text"
      defaultValue={this.state.title}
      onChange={(e) => {
            this.setState({
                  title: e.target.value,
                  action:'update'
```

```
            });
      }}
      placeholder={'Dodaj naslov'}
/>
```

Primjer 22. Ažuriranje title parametra stanja unosom teksta u Input polje

- Prilikom unosa teksta u Input polje poziva se funkcija onChange() koja u sebi poziva setState() funkciju. title svojstvo parametra stanja ažurira se vrijednošću Input polja te action svojstvo parametra stanja dobiva vrijednost 'update'.

```
componentDidUpdate() {
      if (this.state.action) {
            let data = [ ...this.props.data ];
            let item =
                  data.find((item) => item.id === this.state.id);
            item.title = this.state.title;
            item.body = this.state.body;
            this.props.setData(data);
            this.props.setAction('update');

            return () => {
                  this.setState({ action: null });
            };
      }
}
```

Primjer 23. componentDidUpdate() metoda poziva funkciju updateData()

- componentDidUpdate() metoda ažurira data i action parametre stanja Index komponente pod uvjetom da action parametar stanja Post komponente ima zadanu vrijednost.

- Korištenjem *spread* operatora stvara se novi data niz. Korištenjem Javascript find() metode u nizu se pronalazi objekt koji sadrži id čija je vrijednost jednaka id parametru stanja Post komponente.
- Pronađenom objektu zadaju se nove vrijednosti title i body svojstava. Te vrijednosti jednake su this.state.title i this.state.body svojstvima Post komponente.
- this.props.setData() i this.propssetAction() funkcije su Post komponenti proslijeđene kao parametri svojstva. Njihovim pozivom u Index komponenti ažuriraju se data i action parametri stanja.

U navedenom primjeru se item.title i item.body svojstva izravno mutiraju. Ovakvo izravno mutiranje podataka nema nikakvog utjecaja na React aplikaciju pošto se ne radi o objektu vrijednosti parametara stanja ili svojstava.

Iscrtavanje komponenti

Index komponenta je funkcijska komponenta čije se iscrtavanje ograničava korištenjem React.memo() metode. Ova metoda koristi se još u Search i Sort komponentama, koje su također funkcijske komponente. U svim primjerima se koristi bez drugog argumenta.

```
export default React.memo(Index);
```

Primjer 24. Korištenje React.memo metode()

Loader komponenta je klasna komponenta vrste *PureComponent*.

```
class Loader extends PureComponent {}
```

Primjer 25. PureComponent komponenta u aplikaciji

Na ovaj način se kod navedenih komponenti iscrtavanje događa samo prilikom ažuriranja njihovih parametara stanja ili svojstava.

Post komponenta je komponenta koja se u aplikaciji najviše puta iscrtava, pošto svaka iscrtana Post komponenta predstavlja objekt iz data niza. Početni data niz u aplikaciji sadrži sto objekata, što znači da se prilikom početnog iscrtavanja aplikacije Post komponenta iscrtava sto puta. Kako bi se umanjio utjecaj tih iscrtavanja u componentDidMount() metodi definira se iscrtavanje sadržaja komponente ovisno o njenoj vidljivosti u prozoru preglednika. Komponenta će se i dalje

iscrtavati, ali neće dohvaćati parametre stanja. Na taj se način smanjuje utjecaj velikog broja komponenti na učitavanje aplikacije.

```
async componentDidMount() {
     const observer = new IntersectionObserver(
          ([ item ]) => {
               if (item.intersectionRatio === 1) {
                    this.setState({
                         visible: true,
                         title: this.props.postData.title,
                         body:
this.props.postData.body.replace(/\                              n/g,
' '),
                         id: this.props.postData.id
                    });
                    }
               },
               { root: null, rootMargin: '100px', threshold: 1.0 }
     );
     if (this.ref.current) {observer.observe(this.ref.current)}
}
```

Primjer 26. Korištenje componentDidMount() metode za definiranje uvjetne vidljivosti komponente

- Unutar componentDidMount() update definira se intersectionObserver API čija je svrha praćenje vidljivosti komponente u prozoru preglednika.
- Kada je komponenta vidljiva, setState() funkcijom određuju se svojstva parametra stanja komponente – visible, title, body i id.
- observer() funkciji prosljeđuje se referenca na element čiju se vidljivost želi pratiti, u ovom slučaju to je div element u koji je zamotano tijelo komponente.

Drugi korak koji se poduzima kako bi se postigao ovakav način uvjetnog iscrtavanja je primjena uvjetnih operatora unutar render() metode:

```
return (
          <div ref={this.ref} className={`loading-wrapper $
               {this.state.visible ? 'visible' : 'hidden'}`}>
               {this.state.visible && (...tijelo komponente)}
          </div>
     );
}
```

Primjer 27. Korištenje uvjetnih operatora unutar render() metode

- Tijelo komponente zamotano je i div element s klasom loading-wrapper koji u ovisnosti o this.state.visible parametru sadrži dodatnu klasu – visible ili hidden. hidden klasa elementu određuje visinu 110px. Kada element ne bi imao određenu visinu, uvjetan prikaz komponente ne bi bio moguć.
- U ovisnosti u this.state.visible parametru prikazuje se tijelo komponente. Tijelo komponente prikazuje se samo onda kada taj parametar sadrži neku vrijednost.

U navedenom primjeru primjenjuju se uvjetni operatori i logički operator *AND (&&)*.

Kod post komponente bitno je primijeniti i metodu shouldComponentUpdate() kako se svaka Post komponenta ne bi ponovno iscrtavala prilikom radnji pretraživanja, dodavanja i brisanja objekata iz data niza, promjene redoslijeda prikaza objava ili čak uređivanja naslova i sadržaja pojedinih objava.

```
shouldComponentUpdate(nextProps, nextState) {
     return (
          nextProps.postData !== this.props.postData ||
          nextProps.active !== this.props.active ||
          this.state !== nextState ||
          this.state.visible !== nextState.visible
     );
}
```

Primjer 28. korištenje shouldComponentUpdate() metode u Post komponenti

- Metoda `shouldComponentUpdate()` postavlja uvjete pod kojima će se `Post` komponenta iscrtati.
  - Ako su `nextProps.postData` i `this.props.PostData` parametri različiti
  - Ako su `nextProps.active` i `this.props.active` parametri različiti
  - Ako su `this.state` i `nextState` parametri različiti
  - Ako su `this.state.visible` i `nextState.visible` parametri različiti
- Komponenta će se tako ažurirati samo kada se zadani nadolazeći parametri razlikuju od onih trenutnih.

U navedenom primjeru čvrsto se ograničava iscrtavanje `Post` komponente. Komponenta se iscrtava samo kada se zadovolji četiri zadana uvjeta.

Pretraživanje

Pretraživanje se u aplikaciji vrši unosom teksta u polje za upis. Podaci se pretražuju samim unosom, nije prisutan gumb za potvrdu pretraživanja. Kada se radi o pretraživanju velikih nizova podataka, ovakav oblik pretraživanja može biti zahtjevan i dovesti do lošeg izvođenja aplikacije, pošto se veliki niz podataka mora obraditi na svaki unos slovnog znaka. U aplikaciji se taj problem rješava korištenjem *debouncing* funkcije. Funkcija u primjeru identična je onoj iz ranije navedenog primjera (vidi Primjer 17.)

```
export const debounce = (func, wait) => {
      let timeout;
      return function executedFunction(...args) {
            const later = () => {
                  clearTimeout(timeout);
                  func(...args);
            };

      clearTimeout(timeout);
            timeout = setTimeout(later, wait);
      };
};
```

Primjer 29. Primjer debounce() funkcije

Funkcija se poziva prilikom unosa teksta u input polje:

```
<input
      ref={ref}
      type="text"
      placeholder="Pretraživanje"
      className="search__input"
      onChange={debounce(() => {
            handleInput()
      }, 220)}
/>
```

Primjer 30. Korištenje debounce funkcije u pretraživanju

- Funkcija se poziva prilikom onChange događaja input elementa. Prvi argument je arrow funkcija koja poziva handleInput() funkciju. Drugi argument je zadani vremenski interval koji mora proći kako bi se funkcija iz prvog argumenta pozvala (220ms).

handleInput() funkcija vrši pretraživanje data niza:

```
const handleInput = () => {
      if (ref.current) {
            let searchTerm = ref.current.value.toLowerCase().trim();
            let posts = JSON.parse(localStorage.getItem('data'));
            let filteredPosts = posts.filter(
                  (item) =>
                  item.title.toLowerCase().includes(searchTerm) ||
                  item.body.toLowerCase().includes(searchTerm)
            );
            props.setData(filteredPosts);
      }
};
```

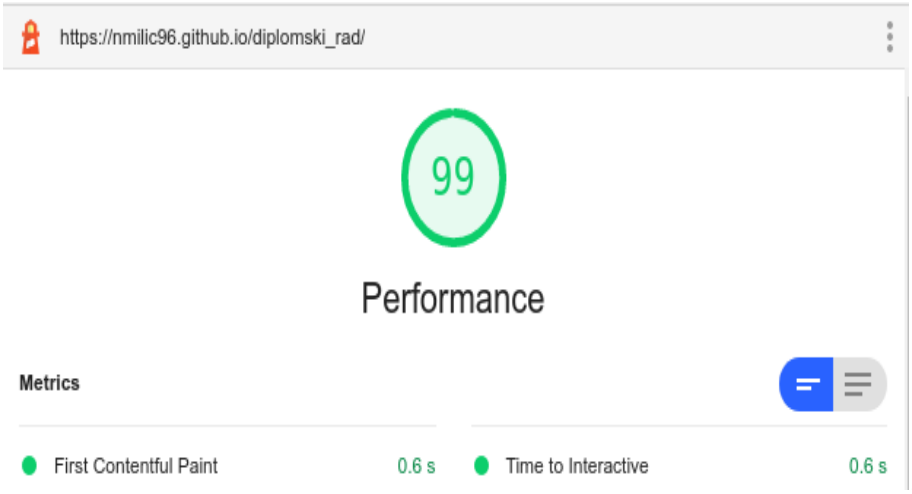
Primjer 31. Funkcija za pretraživanje data niza

- Pretraživanje data niza vrši se korištenjem Javascript `filter()` metode. U nizu se filtriraju oni objekti čija `title` ili `body` svojstva sadrže uneseni termin (`searchTerm`).
- Filter metoda stvara novi niz koji se zatim pomoću funkcije `props.setData()` postavlja u `data` parametar `Index` komponente.

U navedenom primjeru pretraživanje data niza vrši se nakon unosa teksta u input polje, samo kada je nakon zadnjeg znakovnog unosa prošlo 220ms.

Rezultat primjene metoda poboljšanja izvođenja

Rezultat primjene navedenih metoda očituje se u rezultatu ispitivanja efikasnosti izvođenja u alatu Lighthouse. Ukupna ocjena iznosi 99 što se može smatrati vrlo visokim rezultatom. U primjeru je ispitano izvođenje zapakirane inačice aplikacije:



Zaključak

Razdvajanje React aplikacije u komponente omogućuje bolju preglednost koda i podržava dobre prakse u razvoju aplikacije. Kod ovakvog pristupa razvoju aplikacije važno je imati na umu načine na koji te komponente utječu jedne na druge, kao i njihov utjecaj na cjelokupnu aplikaciju. U razvoju se mogu koristiti klasne i funkcijske komponente. Odabir vrste komponenti koje će se koristiti ima utjecaj na pristup poboljšanju izvođenja aplikacije pošto se one razlikuju u pristupu kontroli životnog ciklusa komponente, kao i u načinu definiranja parametara stanja. Ono što je bitno je to da se zahvaljujući React *hook* funkcijama kod obje vrste komponenti može postići jednaka razina efikasnosti izvođenja. Kao bitne razlike između te dvije vrste komponenti se može istaknuti nešto jednostavnija sintaksa funkcijskih komponenti, te nešto jednostavnije upravljanje životnim ciklusom klasnih komponenti uz primjenu `shouldComponentUpdate()` i `componentDidUpdate()` metoda. Iz tog razloga može se preporučiti korištenje funkcijskih komponenti u slučajevima kad nije potrebna detaljna kontrola nad životnim ciklusom i iscrtavanjem komponente te upotreba klasnih komponenti kada to jest slučaj. Važno je naglasiti da sve nabrojane metode i principi nisu u potpunosti nužni za uspješno izvođenje aplikacije. Principe ne-mutiranja podataka nužno je primijeniti, dok se metode `shouldComponentUpdate()`, `componentDidUpdate()`, funkcije višeg reda kao što je `React.memo()`, principe uvjetnog iscrtavanja te principe smanjivanja broja poziva funkcija primjenjuje kako bi se postigla što viša efikasnost izvođenja aplikacija. Kako je vidljivo iz rezultata ispitivanja izvođenja u alatu Lighthouse, izvođenje aplikacije dobilo je vrlo visoku ocjenu. Ovakav rezultat postignut je uspješnom primjenom metoda poboljšanja izvođenja React mrežnih aplikacija.

Literatura

1. Components and Props. URL: https://reactjs.org/docs/components-and-props.html (20-08-2020)
2. Conditional Rendering. URL: https://reactjs.org/docs/conditional-rendering.html (22-08-2020)
3. Copes, F. The React Handbook, str. 1-207. URL: https://flaviocopes.nyc3.digitaloceanspaces.com/react-handbook/react-handbook.pdf
4. Derek, A. How to use the spread operator (…) in JavaScript, 2019. URL: https://medium.com/coding-at-dawn/how-to-use-the-spread-operator-in-javascript-b9e4a8b06fab (22-08-2020)
5. Eluwande, Y; Murray, N. An Introduction to Hooks in React, 2018. URL: https://www.newline.co/fullstack-react/articles/an-introduction-to-hooks-in-react/ (21-08-2020)
6. facebook/react. URL: https://github.com/facebook/react/blob/master/LICENSE (20-08-2020)
7. First Contentful Paint, 2019. URL: https://web.dev/first-contentful-paint/ (19-08-2020)
8. Gonzales, A. Life Cycle Hooks in React, 2019. URL: https://medium.com/@anisgonzales/life-cycle-hooks-in-react-9f1690bcd91b (20-08-2020)
9. Grzywaczewski, M. Pros and Cons of using immutability with React.js, 2015. URL: https://reactkungfu.com/2015/08/pros-and-cons-of-using-immutability-with-react-js/ (22-08-2020)
10. Hooks at a Glance. URL: https://reactjs.org/docs/hooks-overview.html (21-08-2020)
11. JSONPlaceholder. URL: https://github.com/typicode/jsonplaceholder (24-08-2020)
12. Logical AND (&&). URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Logical_AND (22-08-2020)

13. Optimizing Performance. URL: https://reactjs.org/docs/optimizing-performance.html (20-08-2020)

14. React Top-Level API. URL: https://reactjs.org/docs/react-api.html#reactmemo (21-08-2020)

15. React.Component. URL: https://reactjs.org/docs/react-component.html (20-08-2020)

16. Speed Index, 2019. URL: https://web.dev/speed-index/ (19-08-2020)

17. Speedline. URL: https://github.com/paulirish/speedline (19-08-2020)

18. Total Blocking Time, 2019. URL: https://web.dev/lighthouse-total-blocking-time/ (19-08-2020)

19. Understanding Debouncing && Throttling, 2020. URL: https://medium.com/javascript-in-plain-english/understanding-debouncing-throttling-2a0a5e9cc74a (23-08-2020)

20. Using the Effect Hook. URL: https://reactjs.org/docs/hooks-effect.html (21-08-2020)

21. Walton P; Mihajlija, M. Culmulative Layout Shift (CLS), 2019. URL: https://web.dev/cls/ (19-08-2020)

22. Walton, P. Largest Contentful Paint (LCP), 2019. URL: https://web.dev/lcp/ (19-08-2020)