Praktični primeri pristopov za izboljšavo hitrosti in delovanja React aplikacij

Leon Pahole

Povio Inc., San Francisco, United States
leon.pahole@poviolabs.com

Sinopsis Knjižnica React omogoča implementacijo sodobnih interaktivnih spletnih aplikacij. Temelji na komponentah, ki predstavljajo vizualne elemente v aplikaciji. Komponente se povezujejo v hierarhijo in definirajo logiko ter vizualno predstavitev aplikacije. Skozi proces razvoja lahko večanje števila komponent in njihove kompleksnosti vodita v upočasnjeno delovanje aplikacije. V prispevku smo predstavili pristope za analizo in izboljšavo delovanja spletnih aplikacij v knjižnici React. Razložili smo uporabo in interno delovanje knjižnice React ter predstavili tehnike za izvedbo performančnih analiz aplikacij. Na podlagi tega smo pripravili več praktičnih primerov optimizacij. Na koncu smo podali lastno mnenje in nekaj nasvetov na temo optimizacij v procesu razvoja spletnih aplikacij.

Ključne besede:
React
spletne aplikacije
performančna analiza
optimizacija delovanja

Univerzitetna založba
Univerze v Mariboru

ISBN 978-961-286-639-6
DOI https://doi.org/10.18690/um.feri.10.2022.6

1 Uvod

Napredek na področju spletnih tehnologij je v zadnjih letih povzročil rast števila spletnih aplikacij. Na to nakazuje tudi dejstvo, da so spletne tehnologije JavaScript, HTML in CSS na vrhu lestvice najpopularnejših tehnologij v letu 2022 [1,2].

Podjetja vse več svojih produktov razvijajo v obliki spletnih aplikacij, saj lahko na tak način dosežejo širšo publiko, hkrati pa so aplikacije dostopne kadarkoli, na katerikoli napravi. Takšne aplikacije po večini niso namenjene zgolj statični predstavitvi informacij, temveč služijo kot interaktiven produkt s pripadajočo poslovno logiko in profesionalnim izgledom.

Grajenje takšnih aplikacij zahteva previdno planiranje, fleksibilnost logike in enostavno programsko kodo, saj je razvoj dolgotrajen in se velikokrat nikoli ne zaključi. Hkrati je treba poskrbeti za sprejemljivo hitrost izvajanja aplikacije, predvsem, če je ta namenjena javni uporabi. Statistike obnašanj uporabnikov na spletu namreč konsistentno kažejo, da hitrost nalaganja in delovanja strani vplivata na njeno kredibilnost in sposobnost zadrževanja uporabnikov (angl. user retention) [3].

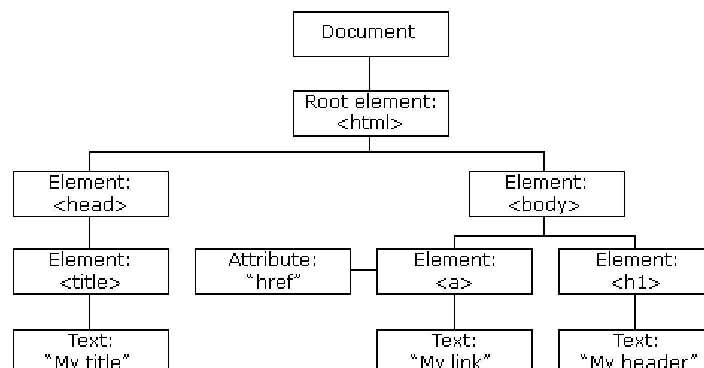
V tem prispevku bomo predstavili knjižnico React.js (v nadaljevanju React) kot eno izmed rešitev za grajenje interaktivnih, hitrih in za vzdrževanje prijaznih spletnih aplikacij. Predstavili bomo uporabo in interno delovanje knjižnice, nato pa na podlagi primerov razložili nekaj tehnik za optimizacijo hitrosti delovanja React aplikacij.

2 O knjižnici React

Želimo implementirati interaktivno spletno aplikacijo. To zahteva manipulacijo z elementi na spletni strani, ne da bi pri tem le-to osvežili - npr. ob kliku na gumb odpremo pojavno okno in skrijemo gumb.

2.1 Vmesnik DOM

Struktura spletne strani je definirana s pomočjo označevalnega jezika HTML. Brskalniki v osnovi ponujajo programski vmesnik za upravljanje s strukturo strani, imenovan DOM (document object model) [4]. V DOM je struktura strani modelirana kot drevo elementov (slika 1). Vsak element ima lahko nič, enega ali več otrok in, razen korena drevesa, natanko enega starša. S pomočjo programskega jezika JavaScript lahko komuniciramo z vmesnikom DOM in manipuliramo z drevesno strukturo; npr. pridobimo podatke o enem ali večih elementih, jim spreminjamo lastnosti ter jih dodajamo ali brišemo.



Slika 1: Vizualizacija drevesa DOM.
Vir: [5].

Zgornja leva stran slike 2 prikazuje primer programske kode, ki komunicira z DOM. Tak pristop se odsvetuje za razvoj kompleksnih spletnih aplikacij iz naslednjih razlogov:

- Koda je manj berljiva in strukturirana. Razvijalec več časa posveti temu, *kako* (imperativno) razviti določen vmesnik, kot temu, *kaj* (deklarativno) naj ta vmesnik vsebuje.
- Če nismo pozorni na implementacijske podrobnosti, je lahko pristop počasen. V primeru, da sprememba v DOM zahteva spremembo v postavitvi strani ali izgledu elementov, bo brskalnik sprožil procesa prilagoditve strukture (angl. reflow) ali izrisa (angl. repaint), ki sta časovno potratna [6]. Z določenimi tehnikami [7] lahko sicer optimiziramo to obnašanje, vendar te optimizacije kodo še dodatno obtežijo, kar nas vodi nazaj v problem zmanjšane berljivosti in nestrukturiranosti.

Slika 2: Različni načini ustvarjanja komponent v spletni aplikaciji.
Vir: lasten.

2.2 Zakaj React?

React je odprtokodna knjižnica za enostavno grajenje uporabniških vmesnikov. Knjižnica ni omejena zgolj na razvoj spletnih aplikacij z delom z DOM v brskalnikih, je pa za ta namen najpogosteje uporabljena [8]. S knjižnico React lajšamo težave, ki jih prinaša neposredno upravljanje z DOM. React delo z DOM abstrahira in programerju omogoči prijaznejši, deklarativni vmesnik za implementacijo spletnih aplikacij. Poleg tega React uporablja algoritme, ki optimizirajo komunikacijo z DOM tako, da so spremembe čim manj potratne [9].

2.3 Komponente

Osnovni gradniki v Reactu so komponente, ki predstavljajo vizualne elemente v aplikaciji [10]. V praksi vmesnik razdelimo na več manjših komponent, kjer ima vsaka komponenta svojo (idealno le eno) nalogo. Komponente lahko velikokrat tudi ponovno uporabimo. Aplikacijo tvori več hierarhičnih komponent tako, da nastane drevo komponent. Komponente se v večini primerov preslikajo v več DOM elementov, kar pomeni, da drevo komponent ni enako drevesu DOM.

Slike 3, 5 in 7 prikazujejo primere komponent. Komponenta je zgolj JavaScript funkcija[1] (ti. funkcijska komponenta). V hierarhiji lahko otrok prejme lastnosti (angl. props) od starša. Komponente tipično opravijo določen izračun ali pripravijo podatke, nato pa vrnejo nekaj, kar izgleda kot HTML. V resnici gre zgolj za

[1] Alternativno se komponente lahko implementira tudi v obliki ES6 razredov.

poenostavljeno sintakso (angl. syntactic sugar), imenovano JSX, ki se med procesom grajenja (angl. build) prevede v klice funkcije *createElement*, ki je del knjižnice React [11].

Slika 2 prikazuje JSX sintakso (levo v sredini) in pripadajoče klice *createElement* (levo spodaj). Opazimo, da se za vsak element v JSX uporabi en klic s parametri *(značka, props, otroci)*. Klici so gnezdeni, s čimer se definira hierarhija (npr. *div* je starš *span*). Rezultat klica *createElement* je React element. Vsaka komponenta vrne en korenski React element in poljubno število otrok. Desna stran slike 2 prikazuje rezultat klica *createElement*. Vidimo lahko, da je React element v ozadju zgolj JavaScript objekt.

Komponente so same po sebi zgolj funkcije. Da povežemo komponente z React-om, je potrebno uporabiti funkcijo *createRoot*, kateri pošljemo korensko komponento v hierarhiji (v našem primeru *Parent*) [12]. S tem bo React izrisal podano hierarhijo komponent v izbran element.

2.4 Izris komponent

Rezultat klica funkcijske komponente je objekt, ki predstavlja React element. Komponente združimo v hierarhijo in jih pošljemo v React s pomočjo *createRoot*. Kako React iz hierarhije komponent izriše strukturo strani?

2.4.1 Prvi izris

Najprej bomo opisali postopek za prvi izris - ob nalaganju strani [13].

Pretvorba v React elemente. React najprej pridobi React elemente za vsako izmed komponent v hierarhiji. To stori tako, da rekurzivno evalvira (angl. render[2]) vsako funkcijo, ki predstavlja funkcijsko komponento, začenši s korensko komponento. Iz vsake komponente pridobi en React element.

Gradnja virtualnega DOM. React iz pridobljenih React elementov zgradi drevesno strukturo, imenovano virtualni DOM (angl. virtual DOM). Virtualni DOM ni dejanski DOM v brskalniku, temveč interna podatkovna struktura v Reactu v obliki JavaScript objekta. S takšno strukturo je delo lažje, kot z dejanskim DOM.

Generiranje mutacij (angl. mutation). React bo dobljen virtualni DOM pretvoril v zaporedje mutacij (sprememb), ki se morajo izvesti na DOM, da bo doseženo želeno stanje. Ker gre za prvi izris, bo potrebno ustvariti prav vse elemente v hierarhiji. Poudariti je treba, da React sam po sebi ne kliče DOM vmesnika brskalnika, temveč zgolj pripravi seznam mutacij v neodvisni obliki.

Manipulacija DOM v brskalniku. V tem delu se seznam mutacij pretvori v dejanske klice DOM (kot na sliki 2, zgoraj levo). Mutacije so aplicirane na optimalen način. Dejanske klice DOM generira knjižnica, ločena od Reacta, imenovana ReactDOMClient[3]. To pomeni, da je React dejansko neodvisen od okolja, v katerem se izvaja. Poleg uporabe v spletnih aplikacij je npr. popularen tudi za razvoj mobilnih aplikacij (React native).

2.4.2 Stanje komponent

Zgoraj opisani postopek velja za prvi izris. Namen interaktivnih aplikacij je, da se skozi čas stanje v komponentah spreminja; npr. ob kliku na gumb se v tabeli prikažejo podatki iz zalednega sistema. Poskusimo implementirati komponento, ki šteje, kolikokrat se je gumb kliknil. Leva stran slike 3 prikazuje naš prvi poskus.

---

[2] V angleški terminologiji prihaja do nekaj konfliktne terminologije glede besed render in rerender. Ti besedi namreč lahko označujeta tako klice funkcij, ki jih opravlja React, kot tudi izris vsebine na zaslon brskalnika. V tem prispevku v prvem kontekstu uporabljamo besedo evalvacija [komponente], v drugem pa izris.
[3] Pred React verzijo 18 je to bila knjižnica ReactDOM [14].

Ta koda ne deluje, saj React ob izrisu (tako prvem, kot vseh naknadnih) funkcijske komponente kliče (ti. evalvacija), kot se kličejo navadne funkcije. To pomeni, da se bo spremenljivka *count* ob vsakem izrisu ponastavila na 0. Poleg tega React ne bo zaznal spremembe tako deklarirane spremenljivke, zato koda v funkciji *onClick* v bistvu ne naredi ničesar.

Potrebujemo način, da Reactu sporočimo, da želimo določeno spremenljivko ohraniti skozi izrise, saj takšna spremenljivka predstavlja stanje naše aplikacije (v tem primeru števec na gumbu). Hkrati želimo, da sprememba stanja povzroči ponoven izris. V Reactu se takšno stanje imenuje state. V funkcijskih komponentah ga lahko deklariramo s pomočjo vgrajene funkcije *useState*. Funkcija *useState* je ena izmed funkcij, ki služi kot vmesnik z Reactom (angl. hook). Prejme začetno vrednost stanja in vrne dva podatka - prvi je trenutna vrednost stanja, drugi pa funkcija, ki nam omogoča spremembo stanja in posledično proženje procesa ponovnega izrisa. Desna stran slike 3 ponazarja delujoč primer števca.

Slika 3: Levo primer napačne uporabe stanja, desno primer pravilne uporabe stanja.
Vir: lasten.

2.4.3 Ponovni izris

Ko se aplikacija prvič izriše na zaslonu, postane interaktivna. V poglavju 2.4.2 smo implementirali enostaven števec na klik. Kaj se zgodi, ko se gumb izriše in nanj kliknemo? Postopek je podoben, kot pri prvem izrisu, in se imenuje postopek ponovnega izrisa (angl. rerender) [13].

Označevanje sprememb. Ob kliku se pokliče funkcija *setCount* z novo vrednostjo števca, kar označi komponento *Counter* kot spremenjeno (angl. dirty). React bo najprej preveril, ali se je stanje dejansko spremenilo (na primer, če bi klicali *setCount(0)*, bi stanje ostalo enako in bi komponenta ostala nespremenjena).

Pretvorba spremenjenih komponent v React elemente[4]. React bo v hierarhiji komponent poiskal vse spremenjene komponente. Za vsako izmed teh komponent bo evalviral pripadajočo funkcijsko komponento. V našem primeru se bo evalvirala funkcija za komponento *Counter*, kjer pa bo spremenljivka counter nastavljena na posodobljeno vrednost (ob prvem kliku bo to 1). Rezultat bo React element, kjer bo edina razlika vsebina elementa *button*. Na tak način se evalvirajo vse komponente, ki so označene za posodobitev. Iz rezultatov se zgradi nov virtualni DOM (nespremenjene komponente ohranijo enake elemente, kot v prejšnjem izrisu).

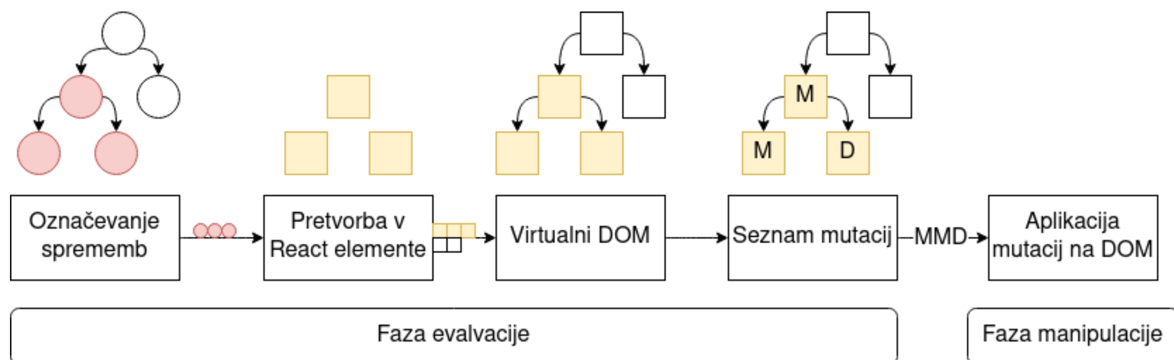
Primerjava Virtualnih DOM dreves in generiranje mutacij. React si interno hrani Virtualni DOM iz prejšnjega izrisa. V tem koraku bo primerjal prejšnji virtualni DOM s trenutnim. Ker je poseganje v dejanski DOM brskalnika potratno, je cilj ugotoviti minimalni nabor mutacij, ki bodo ustrezno pretvorile prejšnje drevo v trenutno drevo. React ta proces imenuje usklajevanje (angl. reconciliation) [15]. Algoritem za iskanje sprememb na drevesih (angl. diffing algorithm) je hevrističen s kompleksnostjo O(n). Usklajevalnik (angl. reconciler), uporabljen v Reactu, se

[4] React bo, če je le možno, več sprememb stanja vključil v en cikel ponovnega izrisa (angl. batching). V React 18 je podpora za batching postala še boljša [17].

imenuje Fiber. V primeru komponente *Counter* bo edina sprememba v drevesu to, da se spremeni vsebina elementa *button*.

Manipulacija DOM v brskalniku. Postopek je enak, kot pri prvem izrisu, vendar se tokrat spremeni občutno manj elementov. Postopek usklajevanja je eden izmed glavnih razlogov, da je React tako hiter, saj je število dragih mutacij v DOM minimizirano.

Zgoraj opisani postopki izrisa se pogosto konceptualno ločujejo na dve fazi: fazo evalvacije (angl. render phase) in fazo manipulacije (angl. commit phase). V fazo evalvacije sodijo vsi koraki do vključno z izračunom mutacij, v fazo manipulacije pa apliciranje generiranih manipulacij na DOM brskalnika [16]. Slika 4 prikazuje povzetek postopka za izris.



Slika 4: Postopek izrisa v knjižnici React.
Vir: lasten.

2.4.4 Rekurzivna evalvacija otrok

V poglavju 2.4.3 smo zapisali, da React pri ponovnem izrisu evalvira le komponente, ki so označene kot spremenjene. Kaj se pri tem zgodi z otroki komponente? Oglejmo si na primeru komponente *Counter* z dvema otrokoma (slika 5).

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

  const onClick = () => {
    setCount(count + 1);
  };

  return (
    <div>
      <button onClick={onClick}>Click</button>
      <DisplayText />
      <CounterText count={count} />
    </div>
  );
}

function DisplayText() {
  return <p>Hello, OTS 2022!</p>;
}

function CounterText(props) {
  return <p>Clicked: {props.count}!</p>;
}
```

Slika 5: Komponenta z dvema otrokoma.
Vir: lasten.

Ali se bosta otroka ob spremembi stanja *count* v *Counter* izrisala ponovno? Lahko bi predvidevali, da se *DisplayText* ne bo izrisal, saj ni odvisen od stanja starša, medtem ko se *CounterText* bo, saj je odvisen od stanja *count* v staršu. Ali je to pravilno razmišljanje? Odvisna je od tega, kaj razumemo pod besedo "izris". Pojdimo skozi postopek.

Označevanje sprememb. Ob klicu *setCounter* se bo komponenta *Counter* označila kot spremenjena.

Pretvorba spremenjenih komponent v React elemente. Komponenta *Counter* se bo evalvirala. Proces evalvacije bo evalviral tudi vse otroke *Counter* ne glede na to, ali so dejansko odvisni od spremenjenega stanja. Otroci bodo evalvirani rekurzivno, kar pomeni, da bi se evalvirali tudi morebitni otroci komponent *DisplayText* in *CounterText*. Rezultat tega koraka so torej evalvacije komponent *Counter*, *DisplayText* in *CounterText*. Rekurzivna evalvacija poteka samo po hierarhiji navzdol, zato se komponenta, ki je v hierarhiji nad *Counter*, ne bi evalvirala.

Primerjava Virtualnih DOM dreves in generiranje mutacij. Po ustvarjanju virtualnega DOM bo React generiral zgolj dve spremembi: spremembo vsebine v komponentah *Counter* in *CounterText*.

Opazimo, da je pri *DisplayText* prišlo do zanimivega pojava: komponenta se je evalvirala v fazi evalvacije, vendar v njej ni bilo sprememb, zato ni prispevala k seznamu mutacij za fazo manipulacije. Tak pojav imenujemo nepotrebna evalvacija (angl. unnecessary render), saj evalvacija komponente ni uvedla sprememb v DOM [18].

3 Optimizacija delovanja v knjižnici React

V poglavju 2 smo opisali postopek izrisa v knjižnici React. Pri optimizaciji delovanja React aplikacij je cilj bodisi skrajšati dolžino postopka enega izrisa, ali pa zmanjšati veliko število izrisov skozi čas, v kolikor je to smiselno. Preden si ogledamo primere, predstavimo tehnike za merjenje hitrosti delovanja React aplikacij.

3.1 Tehnike za merjenje hitrosti delovanja in razhroščevanje postopka izrisa

Za merjenje hitrosti je uporabno orodje React Developer tools (slika 6), ki je na voljo v obliki binarnega paketa ali razširitve za brskalnik. Na podlagi izkušenj predlagamo razširitev za brskalnik Chrome.

Orodje React Developer tools uporabimo tako, da za določen čas snemamo podatke za profiliranje [19]. Med snemanjem aplikacijo uporabljamo, s čimer se zajemajo podatki o izrisih. Po snemanju so nam na voljo podatki:

- seznam izrisov in porabljen čas,
- graf, ki za vsak izris prikazuje hierarhijo komponent, skupaj s podatki o tem, kako dolgo je trajal izris posamezne komponente in ali se je komponenta sploh spremenila v DOM (angl. flamegraph),
- pogled na individualno komponento in vzrok, zakaj se je evalvirala.

Slika 6: Uporaba React Developer tools.
Vir: lasten.

Pri uporabi orodja React Devtools ponavadi snemamo del aplikacije, ki je na vizualni pogled najpočasnejša. Ko posnamemo dovolj podatkov, v seznamu izrisov poiščemo vrhove in velike čase (npr. nad 30 milisekund). S pomočjo grafa nato ugotovimo, katera komponenta je vzela največ časa in zakaj se je evalvirala.

Poleg grafa je uporabna funkcionalnost orodja React Developer tools tudi možnost označevanja izrisov (angl. highlight updates when components render), ki vizualno označi komponente, ko se izrisujejo. Na tak način lahko identificiramo okvirne dele aplikacije, ki predstavljajo morebitna ozka grla, nakar jih kvantificiramo s snemanjem.

Poleg orodja React Developer tools so pogosto uporabni tudi izpisi v kodi komponent (angl. logs). Na tak način se bo v brskalnik izpisal niz znakov ob evalvaciji komponente. Po izkušnjah je tak pristop primeren za iskanje točnega vzroka počasnega delovanja, ko s pomočjo grafa ugotovimo, katere komponente so problematične.

3.2 Primeri vzorcev optimizacij

Primeri, ki sledijo, so namenjeni demonstraciji delovanja postopka izrisa in tehnik za optimizacijo.

3.2.1 Primer: enakost stanja

Imamo predvajalnik videov in komponento *VideoTime*, v kateri bi želeli prikazovati trenutni čas videa v sekundah. Predvajalnik nam trenutni čas sporoča v enoti milisekunda vsakih 50 milisekund. Ob zagonu predvajalnika si bodo časi torej sledili tako: 0, 50, 100, 150 itn. Leva stran slike 7 prikazuje neoptimalno implementacijo.

```
function VideoTime() {
  const [time, setTime] = useState(0);

  const onTimeChange = (timeMs) => {
    setTime(timeMs);
  };

  useVideo(onTimeChange);

  return <p>Time: {Math.floor(time / 1000)}</p>;
}
```

```
function VideoTime() {
  const [time, setTime] = useState(0);

  const onTimeChange = (timeMs) => {
    setTime(Math.floor(timeMs / 1000));
  };

  useVideo(onTimeChange);

  return <p>Time: {time}</p>;
}
```

Slika 7: Levo primer neoptimalne, desno primer boljše rešitve.
Vir: lasten.

S pomočjo označevanja evalvacij v orodju React Developer tools ali s pomočjo izpisov lahko opazimo, da se komponenta *VideoTime* evalvira vsakih 50 milisekund.

Razmislimo, kaj se dogaja v postopku izrisa za časa 0 in 50 milisekund. Ob klicu *setTime(0)* se komponenta označi kot spremenjena, zato jo React v naslednjem ciklu evalvira. Postopek evalvacije izračuna tudi vrednost *Math.floor(0 / 1000)*, ki milisekunde pretvori v sekunde in rezultat zaokroži navzdol. Rezultat je vrednost 0. Ker gre za prvi izris, se bo komponenta v celoti izrisala na DOM v fazi manipulacije (poglavje 2.4.1).

Ob klicu *setTime(50)* se ponovi podoben postopek. V tem primeru se evalvira izraz *Math.floor(50 / 1000)*, katerega rezultat je ponovno 0. Pri postopku primerjave virtualnega DOM-a bo React ugotovil, da ni potrebno ničesar spremeniti, saj je element *p* že v prejšnjem izrisu imel vsebino 0. Podoben postopek se bo ponavljal vse do klica *setTime(1000)* čez eno sekundo, kjer se bo vsebina spremenila na 1.

Opazimo, da imamo vsako sekundo le eno manipulacijo na DOM in 19 nepotrebnih evalvacij, ki sicer spremenijo stanje *time*, ampak ne povzročijo izrisa na DOM. Rešitev je enostavna - klic *Math.floor* prestavimo iz funkcije v klic *setTime* (desna stran slike 7).

Sedaj se bo pri časih 0, 50 in 100 *setTime* klical s parametri 0, 0, 0 in ne 0, 0.05 in 0.1. React bo pri klicu *setTime* zaznal, da se stanje ni spremenilo (saj je 0 enako 0) in ne bo evalviral komponente. Komponenta se bo sedaj evalvirala le enkrat na sekundo[5].

Primer lahko še dodatno optimiziramo tako, da pred klicem *setTime* primerjamo zaokroženo vrednost s trenutno vrednostjo *time*. Priporočamo takšno rešitev, kljub temu pa je na prejšnji rešitvi dobro ponazorjen efekt spremembe stanja na enako vrednost.

3.2.2 Primer: otrok v lastnostih

Kot opisano v poglavju 2.4.4, se ob evalvaciji komponente rekurzivno evalvirajo tudi vsi njeni otroci. Želeli bi se izogniti nepotrebnim evalvacijam otrok, ko na otroka sprememba stanja v staršu ne vpliva.

Manj znan pristop, s katerim lahko to dosežemo, je pošiljanje otroka v starša preko lastnosti (props). V tem primeru potrebujemo še eno starševsko komponento *CounterParent* (slika 8).

```
function CounterParent() {
  return <Counter textComponent={<DisplayText />} />;
}

function DisplayText() {
  return <p>Hello, OTS 2022!</p>;
}

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

  const onClick = () => {
    setCount(count + 1);
  };

  return (
    <div>
      <button onClick={onClick}>Click {count}</button>
      {props.textComponent}
    </div>
  );
}
```

Slika 8: Uporaba otroka v lastnostih za preprečevanje nepotrebnih evalvacij.
Vir: lasten.

Ko se v *Counter* spremeni *count*, se starš *CounterParent* ne bo evalviral. Hkrati vemo, da komponenta ne more spreminjati svojih lastnosti. Iz tega sledi zaključek, da se lastnost *textComponent* pri spremembi stanja v komponenti *Counter* zagotovo ni spremenila. React lahko torej varno izpusti evalvacijo komponente *CounterText*.

3.2.3 Primer: preprečevanje nepotrebnih evalvacij

Rešitve v prejšnjem primeru ni vedno možno uporabiti, poleg tega pa lahko pristop hierarhijo komponent naredi manj razumljivo. V tem primeru bomo razložili alternativo s pomočjo komponente *Counter*, ki vsakih 50 milisekund inkrementira števec (slika 9). Poleg tega ima komponenta 4 otroke:

- *Text* zgolj izriše statičen tekst. Ta komponenta ima še enega otroka *TextChild*, ki prav tako samo izriše statični tekst.

[5] React pravzaprav bo evalviral komponento prvič, ko bo sprememba stanja enaka. To je varnostni mehanizem [20].

- *ResetButton* ima preko funkcije *onClick*, poslane kot lastnost, povezavo v starša, s katerim se stanje števca ponastavi ter inkrementira število klikov.
- *ComputedText* prejme in izriše objekt s številom klikov na gumb ter fakulteto tega števila.
- *CounterText* izriše vrednost števca.

S pomočjo orodja React Developer tools takoj opazimo, da se izrisujejo prav vsi otroci zaradi rekurzivne evalvacije. Preden nadaljujemo, analizirajmo optimalne čase izrisa za vse komponente:

- *Text*, *TextChild* in *ResetButton* niso odvisni od stanja v komponenti *Counter*, zato bi se v optimalnem primeru izrisali samo enkrat, na začetku.
- *ComputedText* je sicer odvisen od stanja v *Counter*, ampak le od števila klikov *clickCount*, zato bi se v optimalnem primeru izrisal le ob spremembi *clickCount* (tj. ob kliku na gumb *ResetButton*).
- *CounterText* je odvisen od stanja števca *count*, zato se mora izrisati vsakič, ko se spremeni stanje *count*.

```
const Counter = () => {
  const [count, setCount] = useState(0);
  const [clickCount, setClickCount] = useState(0);

  const onIncrement = () => {
    setCount((count) => count + 1);
  };

  const onReset = () => {
    setCount(0);
    setClickCount((clickCount) => clickCount + 1);
  };

  useIncrement(onIncrement);

  const data = {
    clickCount,
    factorial: factorial(clickCount),
  };

  return (
    <div>
      <Text />
      <ResetButton onClick={onReset} />
      <ComputedText data={data} />
      <CounterText count={count} />
    </div>
  );
};
```

```
const Text = () => {
  return (
    <div>
      <span>Hello,</span>
      <TextChild />
    </div>
  );
};

const TextChild = () => {
  return <span>OTS 2022!</span>;
};

const ResetButton = (props) => {
  return <button onClick={props.onClick}>Reset</button>;
};

const ComputedText = (props) => {
  return <p>{JSON.stringify(props.data)}</p>;
};

const CounterText = (props) => {
  return <p>Counter: {props.count}</p>;
};
```

Slika 9: Komponenta *Counter* za primer preprečevanja nepotrebnih evaluacij.
Vir: lasten.

Optimizacija komponent *Text* in *TextChild*. Komponenta *Text* se izrisuje kljub temu, da ni odvisna od stanja *Counter*. Ko se evalvira *Counter*, se evalvira *Text*, in rekurzivno tudi *TextChild*. Gre za nepotrebno evalvacijo, saj se vsebina komponent ne spreminja. React ponuja mehanizem *React.memo*, s katerim lahko preprečimo evalvacijo komponente, v kolikor so izpolnjeni pogoji [21]:

- evalvacija je sprožena iz starša,
- stanje komponente se ni spremenilo,
- lastnosti (props) komponente se niso spremenile.

Komponento *Text* objamemo (angl. wrapping) s funkcijo *React.memo* (slika 11), kar vrne novo komponento. Ko se evalvira komponenta *Counter*, se komponenta *Text* ne bo, saj so izpolnjeni vsi pogoji za preprečevanje evalvacije. Poudarimo, da komponente *TextChild* ni potrebno objeti z *React.memo*, saj bo veriga rekurzivne evalvacije ustavljena že na komponenti *Text*.

Optimizacija komponente *ResetButton*. Podobno kot pri komponenti *Text*, se tudi lastnosti komponente *ResetButton* ne spreminjajo, ko se evalvira *Counter*, torej lahko ponovno uporabimo *React.memo*. Ugotovili bomo, da se bo kljub uporabi *React.memo* komponenta *ResetButton* še vedno evalvirala ob vsaki evalvaciji *Counter*. To pomeni, da eden izmed pogojev za preprečevanje evalvacije ni dosežen. V tem primeru je lahko to zgolj pogoj, da se lastnosti komponente niso spremenile.

Komponenta *ResetButton* ima le eno lastnost - funkcijo *onClick*. Na prvi pogled bi lahko rekli, da se ta funkcija ob evalvacijah *Counter* ne spreminja, vendar temu ni tako. V komponenti *Counter* deklariramo funkcijo *onReset*, ki

predstavlja lastnost *onClick* za *ResetButton*. Ko se *Counter* evalvira, se bo izvedel tudi ta del kode, kar pomeni, da se bo ob vsaki evalvaciji *Counter* ustvari nova funkcija, ki se priredi spremenljivki *onReset*.

React.memo[6] pri detektiranju sprememb uporablja JavaScript funkcijo *Object.is* [22]. Ta funkcija preveri referenčno enakost (angl. referential equality) parametrov. V primeru enakih vrednosti primitivnih tipov v jeziku JavaScript [23] bo funkcija vrnila pričakovani rezultat (leva stran slike 10), pri objektih pa le, če gre za referenci na enak objekt (desna stran slike 10). Enako pravilo velja za polja in funkcije, ki so v jeziku JavaScript pravzaprav le posebne vrste objektov.

```
Object.is(1, 1); // true
Object.is(1, 2); // false
Object.is("OTS", "OTS"); // true
Object.is("", ""); // true
Object.is(null, undefined); // false
```

```
Object.is({}, {}); // false - različna referenca!

const arr = [];
const arr2 = arr;
Object.is(arr, arr2); // true - enaka referenca!
```

Slika 10: Demonstracija delovanja *Object.is* funkcije.
Vir: lasten.

V našem primeru se ob vsaki evalvaciji *Counter* ustvari nova instanca funkcije *onReset*, ki ni enaka instanci iz prejšnje evalvacije, zaradi česar *Object.is* v *React.memo* zazna spremembo lastnosti, kar evalvira funkcijo *ResetButton*.

Potrebujemo mehanizem, s katerim lahko skozi evalvacije komponente *Counter* ohranimo enako referenco na funkcijo *onReset*, tako da bo *Object.is* ustrezno prepoznal referenčno enakost. Rešitev je funkcija *useCallback* (slika 11), ki izvede tako imenovano memoizacijo (angl. memoization) nad funkcijo *onReset* in skozi evalvacije ohranja njeno referenco.

Optimizacija komponente *ComputedText*. Pri optimizaciji komponente *ComputedText* pridemo do podobnega zaključka, kot pri komponenti *ResetButton*. Tudi v tem primeru *React.memo* sam ne bo preprečil nepotrebnih evalvacij, saj se objekt *data* ponovno ustvari ob vsaki evalvaciji komponente *Counter*. Razlika je zgolj v tem, da v tem primeru želimo ohraniti referenco na objekt, namesto na funkcijo. Ekvivalent *useCallback* za objekte je funkcija *useMemo*. Funkcija *useMemo* bo ohranjala vrednost poljubne spremenljivke (v našem primeru je to objekt *data*) skozi evalvacije, dokler se ne spremeni ena izmed spremenljivk, naštetih v polju, ki je podan kot drug parameter v klic *useMemo*. Ob spremembi se bo ponovno evalvirala funkcija, podana kot prvi parameter. V našem primeru se bo to zgodilo ob kliku na gumb *ResetButton*. V tem primeru bo prišlo do evalvacije *Counter* in *ComputedText* (zaradi spremenjene reference na objekt *data*), kar pa je v tem primeru pravilno obnašanje, saj je *ComputedText* odvisen od števila klikov v *Counter*.

Optimizacija komponente *CounterText*. Komponenta *CounterText* je odvisna od spremenljivk stanja *count* in *clickCount* v komponenti *Counter*. Pri tem se spremenljivka *count* spreminja najpogosteje. To pomeni, da se komponente ne splača optimizirati, saj se bodo v vsakem primeru njene lastnosti spremenile ob vsaki evalvaciji.

[6] Enak algoritem se uporablja tudi pri detektiranju spremenjenih stanj v komponentah v prvem koraku postopka izrisa.

```
const Counter = () => {
  const [count, setCount] = useState(0);
  const [clickCount, setClickCount] = useState(0);

  const onIncrement = () => {
    setCount((count) => count + 1);
  };

  const onReset = useCallback(() => {
    setCount(0);
    setClickCount((clickCount) => clickCount + 1);
  }, []);

  useIncrement(onIncrement);

  const data = useMemo(
    () => ({
      clickCount,
      factorial: factorial(clickCount),
    }),
    [clickCount]
  );

  return (
    <div>
      <Text />
      <ResetButton onClick={onReset} />
      <ComputedText data={data} />
      <CounterText count={count} />
    </div>
  );
};
```

```
const Text = React.memo(() => {
  return (
    <div>
      <span>Hello,</span>
      <TextChild />
    </div>
  );
});

const TextChild = () => {
  return <span>OTS 2022!</span>;
};

const ResetButton = React.memo((props) => {
  return <button onClick={props.onClick}>Reset</button>;
});

const ComputedText = React.memo((props) => {
  return <p>{JSON.stringify(props.data)}</p>;
});

const CounterText = (props) => {
  return <p>Counter: {props.count}</p>;
};
```

Slika 11: Optimizirana komponenta *Counter*.
Vir: lasten.

3.2.4 Primer: optimiziranje faze manipulacije

Vse dozdajšnje optimizacije smo izvajali v sklopu faze evalvacije. V tem poglavju bomo predstavili dva primera optimizacije v fazi manipulacije. V poglavju 2.4.3 smo omenili, da je algoritem usklajevanja hevrističen, in sicer za optimalnost izvajanja naredi dve predpostavki [9]. Razumevanje obeh predpostavk je ključnega pomena za optimizacijo faze manipulacije.

Predpostavka 1: elementa z različnima tipoma generirata drugačna drevesa.

Imejmo komponento, ki ob prvem izrisu ustvari React element z elementom *div* in tremi otroci *span*, ob naslednjem izrisu pa *div* zamenja z *section* (otroci ostanejo enaki). React bo v procesu usklajevanja ugotovil, da se je tip starša spremenil iz *div* v *section*, zato bo vse otroke rekurzivno izbrisal in jih ponovno ustvaril v DOM kljub temu, da se niso spremenili. To je posledica predpostavke 1.

Rekurzivni izbris vseh otrok v hierarhiji je zelo počasna operacija, predvsem, če ima starš veliko število otrok. Da se izognemo počasnemu delovanju, se v praksi skušamo izogniti spremembam tipov starševskih komponent.

Predpostavka 2: razvijalec lahko z lastnostjo *key* označi enake komponente v ponovnem izrisu.

Imejmo komponento, ki vrne element *ul* in tri otroke *li*. V naslednjem izrisu dodamo še enega otroka *li* na konec seznama. V procesu usklajevanja bo React primerjal elemente po vrsti od zgoraj navzdol in pravilno generiral eno manipulacijo: dodajanje novega otroka *li* na konec seznama.

Oglejmo si, kaj pa se zgodi, če element *li* dodamo na začetek seznama. V tem primeru bo primerjanje elementov od zgoraj navzdol pri vsakem elementu *li* zaznalo spremembo, zato bo React zgeneriral spremembo prvih dveh elementov *li* in dodajanje zadnjega elementa *li* (slika 12).

```
<ul>
  <li>OTS</li>
  <li>2022</li>
</ul>
```

```
<ul>
  <li>Hello</li>
  <li>OTS</li>
  <li>2022</li>
</ul>
```

Slika 12: Brez uporabe lastnosti *key* - React ne najde ujemanja med elementi.
Vir: lasten.

```
<ul>
  <li key="ots">OTS</li>
  <li key="2022">2022</li>
</ul>
```

```
<ul>
  <li key="hello">Hello</li>
  <li key="ots">OTS</li>
  <li key="2022">2022</li>
</ul>
```

Slika 13: Z uporabo lastnosti *key* - React najde ujemanje.
Vir: lasten.

Težavo lahko rešimo s pomočjo lastnosti *key*, ki predstavlja identiteto komponente v seznamu (slika 13). V kolikor elementom podamo unikatno lastnost *key*, bo react pare iskal po lastnosti *key*, namesto od zgoraj navzdol. V prejšnjem primeru bo tako pravilno našel zgolj eno spremembo: dodajanje elementa *li* na prvo mesto seznama.

Priporočljivo je, da za lastnost *key* uporabimo unikaten identifikator. Indeksi polja niso dobri kandidati za vrednost *key*, saj pri vstavljanju na sredino ali pri menjavi elementov v seznamu elementi menjajo lastnost *key* [24].

3.3 Primer iz produkcije

Predstavljajmo si aplikacijo, ki predvaja video posnetek. Uporabniki lahko na poljubni točki v posnetku dodajo komentar. Komentarji se prikažejo v seznamu poleg posnetka v času, ko so bili dodani in do 5 sekund po tem času. Komentarje lahko brišemo. Prikazujemo tudi lastno časovnico videa, ki se posodablja ob predvajanju videa.

V aplikaciji se moramo povezati na čas videa. Video nam čas sporoča v obliki milisekund približno vsakih 10 milisekund. V polju hranimo podatke o dodanih komentarjih. Ob vsaki spremembi časa je potrebno izračunati komentarje, ki se prikazujejo na zaslonu. Prav tako je treba prilagoditi časovnico.

Pri planiranju smo ugotovili, da se bodo vsi komentarji velikokrat nepotrebno izrisali, saj se čas videa spreminja okoli 50-krat na sekundo. V ta namen smo uporabili *React.memo*. Poleg tega smo morali uporabiti še *useCallback*, da smo zagotovili nespremenljivost funkcijskih referenc.

Kljub temu, da so se ponekod v aplikaciji še pojavljale nepotrebne evalvacije, z optimizacijo nismo nadaljevali, saj nadaljnje optimizacije ne bi prinesle večje vrednosti za produkt.

3.4 Kdaj izvesti optimizacije?

Kdaj se lotiti izvajanja optimizacij? Mnenj na to temo je veliko, nekateri razvijalci so zagovorniki optimizacij šele, ko je opazno počasno delovanje, spet drugi pa optimizirajo že v začetku razvoja. Na koncu je najpomembneje, da se odločimo po lastni presoji glede na situacijo.

Naše mnenje je, da je potrebno najti balans med izvedbo optimizacij v razvojnem procesu in vrednostjo, ki jo to prinaša za produkt.

V kolikor pišemo prototip ali dokaz koncepta, potem je vrednost optimizacij praktično ničelna, zato optimizacij ne izvajamo. V kolikor pa razvijamo produkt v produkciji, pa optimizacije vkomponiramo v razvojni proces, vendar na način, ki ni časovno potraten in je osredotočen le na pomembne stvari. V fazi planiranja razvoja funkcionalnosti identificiramo komponente, ki bi se lahko pogosto izrisovale. Med razvojem pogosto uporabljamo funkcionalnost vizualizacije izrisov, s katerimi dobimo hiter vizualni feedback o morebitnih ozkih grlih. Na tak način lahko že v zgodnji fazi preprečimo počasno delovanje.

Dobro se je zavedati, da nepotrebnih evalvacij ni potrebno vedno naslavljati. V večini primerov gre za komponente, katerih evalvacije so dovolj enostavne ali pa dovolj redke, da ne prispevajo veliko k času izrisa. V teh primerih lahko optimizacije celo poslabšajo hitrosti komponent, saj ima vsaka optimizacija v ozadju določeno režijo, ki prav tako zavzame nekaj časa v izrisu.

Optimizacije se najbolj splačajo na komponentah, ki se pogosto evalvirajo z enakimi lastnostmi. V teh primerih bo večina evalvacij komponente nepotrebnih, tako da bo *React.memo* v kombinaciji z *useMemo* in *useCallback* drastično znižal število evalvacij komponente.
Velikokrat se optimizacije lahko izvede brez uporabe eksplicitnih orodij za optimizacijo, npr. s tehniko otroka v lastnosti ali z premestitvijo hierarhije komponent.

Priporočamo, da se po vsaki optimizaciji s pomočjo React Developer tools preveri, ali je optimizacija dejansko pospešila delovanje.

4 Zaključek

V prispevku smo predstavili React kot orodje, ki poenostavi grajenje modernih interaktivnih aplikacij s pomočjo deklarativne sintakse in strukturiranih komponent. S pomočjo analize internih algoritmov v knjižnici React smo ugotovili, zakaj je React že v izvoru zelo hiter. Z znanjem o internem algoritmu in pravilih za izris smo nato pripravili primere vzorcev optimizacij ter predstavili en primer iz produkcijskega okolja. Na koncu smo podali nekaj praktičnih izkušenj in napotkov za optimizacije v knjižnici React.

Literatura

[1] https://bootcamp.berkeley.edu/blog/most-in-demand-programming-languages, 11 Most In-Demand Programming Languages in 2022, obiskano 27. 7. 2022.

[2] https://survey.stackoverflow.co/2022/#most-popular-technologies-language, Stack Overflow Developer Survey 2022, 2022, obiskano 27. 7. 2022.

[3] Kristen Baker, https://blog.hubspot.com/marketing/page-load-time-conversion-rates, 11 Website Page Load Time Statistics You Need, 2022, obiskano 27. 7. 2022.

[4] https://developer.mozilla.org/en-US/docs/Glossary/DOM, DOM (Document Object Model), 2022, obiskano 27. 7. 2022.

[5] https://www.w3schools.com/js/js_htmldom.asp, JavaScript HTML DOM, 2022, obiskano 27. 7. 2022.

[6] Anshul Parmar, https://www.linkedin.com/pulse/reflow-repaint-manipulating-dom-responsibly-anshul-parmar, Reflow and Repaint: Manipulating DOM responsibly, 2019, obiskano 27. 7. 2022.

[7] Gopalakrishnan, https://dev.to/gopal1996/understanding-reflow-and-repaint-in-the-browser-1jbg, Understanding Reflow and Repaint in the browser, 2020, obiskano 27. 7. 2022.

[8] https://www.geeksforgeeks.org/reactjs-reactdom/, ReactJS | ReactDOM, 2021, obiskano 27. 7. 2022.

[9] https://reactjs.org/docs/reconciliation.html, Reconciliation, obiskano 27. 7. 2022.

[10] https://reactjs.org/docs/components-and-props.html, Components and Props, obiskano 27. 7. 2022.

[11] https://reactjs.org/docs/introducing-jsx.html, Introducing JSX, obiskano 27. 7. 2022.

[12] https://reactjs.org/docs/react-dom-client.html#createroot, ReactDOMClient, obiskano 27. 7. 2022.

[13] https://blog.isquaredsoftware.com/2020/05/blogged-answers-a-mostly-complete-guide-to-react-rendering-behavior/, A (Mostly) Complete Guide to React Rendering Behavior, 2020, obiskano 27. 7. 2022.

[14] https://reactjs.org/blog/2022/03/08/react-18-upgrade-guide.html#updates-to-client-rendering-apis, How to Upgrade to React 18, obiskano 27. 7. 2022.

[15] Kishan Sheth, https://hackernoon.com/virtual-dom-reconciliation-and-diffing-algorithm-explained-simply-ycn34gr, Virtual DOM, Reconciliation And Diffing Algorithm Explained Simply, 2021, obiskano 27. 7. 2022.

[16] Bryan Almaraz, https://dev.to/thee_divide/reconciliation-react-rendering-phases-56g2, 2020, obiskano 27. 7. 2022.

[17] Lakindu Hewawasam, https://blog.bitsrc.io/automatic-batching-in-react-18-what-you-should-know-d50141dc096e, 2022, obiskano 27. 7. 2022.

[18] https://www.debugbear.com/blog/react-rerenders, Optimizing React performance by preventing unnecessary re-renders, 2021, obiskano 27. 7. 2022.

[19] https://www.geeksforgeeks.org/react-developer-tools/, React Developer Tools, 2021, obiskano 27. 7. 2022.

[20] https://reactjs.org/docs/hooks-reference.html#bailing-out-of-a-state-update, Bailing out of a state update, obiskano 27. 7. 2022.

[21] https://reactjs.org/docs/react-api.html#reactmemo, React.memo, obiskano 27. 7. 2022.

[22] Valentino Gagliardi, https://www.valentinog.com/blog/react-object-is/, Demystifying Object.is and prevState in React useState, 2021, obiskano 27. 7. 2022.

[23] https://developer.mozilla.org/en-US/docs/Glossary/Primitive, Primitive, obiskano 27. 7. 2022.

[24] Robin Pokorny, https://robinpokorny.medium.com/index-as-a-key-is-an-anti-pattern-e0349aece318, Index as a key is an anti-pattern, 2015, obiskano 27. 7. 2022.