Razvoj web aplikacije u programskom okviru React.js

Lacko, Martin

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Martin Lacko

RAZVOJ WEB APLIKACIJE U PROGRAMSKOM OKVIRU REACT.JS

ZAVRŠNI RAD

Varaždin, 2020.

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Martin Lacko

Matični broj: 46271/17–R

Studij: Primjena informacijske tehnologije u poslovanju

RAZVOJ WEB APLIKACIJE U PROGRAMSKOM OKVIRU REACT.JS

ZAVRŠNI RAD

Mentor/Mentorica:

Prof. dr. sc. Dragutin Kermek

Varaždin, lipanj 2020.

Martin Lacko

Izjava o izvornosti

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

Autor/Autorica potvrdio/potvrdila prihvaćanjem odredbi u sustavu FOI-radovi

_____

Sažetak

Cilj ovog rada je opisati React.js programski okvir. Teorijski dio rada prikazat će osnove programskog okvira, postavljanje radnog okruženja, manipulaciju komponentama koje su karakteristične za *React* te korištenje posebnih parametara (*props*) i posebnog objekta (*state*). Bit će navedene prednosti *React*-a i dodaci koji doprinose poboljšanju funkcionalnosti web aplikacija. Praktični dio sastoji se od web aplikacije koja će prikazati korištenje *React*-a na stvarnom primjeru, a koja će služiti za provjeru originalnosti i dostupnosti imena poduzeća na području Republike Hrvatske. Provjera će se odvijati na temelju podataka iz sudskog registra, dostupnosti imena hrvatske domene, ali i najpoznatijih svjetskih domena te provjera dostupnosti korisničkog imena društvenih mreža. Izgradnja web aplikacije bit će opisana u teorijskom djelu završnog rada.

Ključne riječi: *Web, HTML5, CSS3, Javascript, React, Redux, Web aplikacija*

Sadržaj

1. Uvod

Svakodnevni razvoj poslovanja i potreba za digitalizacijom poslovnih sustava dovela je do većeg razvoja tehnologije. Svaki poslovni sustav želi imati što veću podršku IT sektora i poboljšati svoj rad te korisnicima pružati brže usluge. Zbog ovakvih potreba tehnologija je prisiljena ubrzano se razvijati i pružati podršku poslovanju. Razvojem novih poslovnih sustava stvara se digitalizirano rješenje, dok se stariji sustavi također nastoje nadomjestiti tehnologijom. Najčešći način digitalizacije nekog poslovnog sustava je realizacija putem interneta, tj. kroz web aplikacije.

Internet se definira kao svjetska međusobna povezanost pojedinih mreža. Tim mrežama upravlja vlada, industrija, akademija i privatne stranke. Prvotno, Internet je služio za povezivanje laboratorija koji su se bavili vladinim istraživanjima, a od 1994. godine se proširio te sada služi mnogim korisnicima diljem svijeta. Internet se mijenja svakim danom, a dvije stvari (inovacije) promijenile su način na koji ljudi koriste Internet danas. Prvotno, naglasak je prešao na komunikaciju ljudi putem interneta koja je pokrenuta 2004. godine, pojavom *Facebook*-a, koji je prerastao u svjetsku mrežu od preko 2,450 milijuna aktivnih korisnika. S druge strane, mobilna tehnologija omogućila je još veći doseg interneta, povećavajući broj korisnika interneta u cijelome svijetu. Danas, Internet je i dalje najdemokratskiji i najmasivniji medij [1].

Zbog sve veće potrebe za digitalnim rješenjima putem interneta, nastali su mnogi programski jezici i različiti programski okviri koji su pridonijeli lakšoj i bržoj izradi web aplikacija i pružanju što boljeg iskustva korisnicima. *Facebook*, kao vodeća društvena mreža, zbog povećanog broja korisnika i širenja cijele platforme, odlučio je izgraditi svoj vlastiti programski okvir za održavanje i poboljšanje rada – *React*.js.

Prva verzija *React*-a izašla je 29.05.2013. godine, a sam programski okvir razvija se svakim danom. Sve češća je upotreba *React*-a kao programskog okvira u raznim programerskim tvrtkama te je postao jedan od najpopularnijih okvira za razvoj brzih i efikasnih web aplikacija. Temelji se na komponentama od kojih je svaka odgovorna za stvaranje malog djela *HTML* koda koji se može ponovno upotrijebiti. Komponente se mogu ugnijezditi u druge komponente kako bi se lakše razvile složenije aplikacije. One su napisane u *Javascript*-u i mogu koristiti sve mogućnosti tog jezika (funkcije, petlje, itd.). Uz sam *React*, postoje i mnogi dodaci koji se mogu instalirati i koristiti zajedno u paketu, kako bi pridonijeli lakšem razvoju web aplikacija [2].

U ovom radu bit će prikazano kako instalirati i koristiti *React* i mnoge druge dodatke koji su potrebni za lakši razvoj web aplikacija. Također, na kraju ovog rada bit će predstavljen i opisan primjer vlastite web aplikacije i objašnjeni koraci razvoja koji će prikazati kako sve teorijski opisane funkcije rade u praksi.

2. Web aplikacije

Kako je Internet postao sve zreliji, aplikacije koje uključuju korisnička sučelja i interaktivne mogućnost postala su sve popularnija. Razvojni inženjeri u prošlosti su koristili različite aplikacije od tvrtki kao što su *Macromedia*, *Microsoft* i *Sun Microsystems*. Sada je sve manja upotreba računalnih aplikacija i cijeli sustav prebacio se na web, zbog lakšeg i bržeg razvoja i dostupnosti krajnjim korisnicima [8].

Kako bi se stvorilo dobro korisničko iskustvo, važno je odabrati dobru arhitekturu web aplikacija. Kod odabira često se zahtijeva detalja analiza arhitektonskih značajki i potreba korisnika između aplikacije s jednom stranicom (*SPA-Single Page Application*) i aplikacija s više stranica (*MPA-Multi Page Application*). *MPA* aplikacije su većinom namijenjene za veće sustave s velikim brojem različitih usluga, dok je *SPA* noviji pristup web aplikacijama i često se koristi kod jednostavnijih aplikacija s manje sadržaja [9].

Glavna razlika između *MPA* i *SPA* radnih ciklusa je u prirodi zahtjeva i odgovora. *MPA* životni ciklus završava primanjem odgovora, dok *SPA* životni ciklus traje i temelji se na radu korisnika kroz korisničko sučelje. *SPA* koristi *AJAX* (*Asynchronous JavaScript and XML*) kada šalje zahtjev serveru, a kao odgovor dobiva podatke (obično u obliku *JSON*-a) i male dijelove *HTML* koda. Nakon što podaci stignu s poslužitelja, klijentska strana će formirati primljeni sadržaj i prikazati ga na određenom mjestu [9].

2.1. HTML

HTML (*HyperText Markup Language*) je najosnovniji dio web stranica. On definira značenje i strukturu cijelog web sadržaja te čini sam kostur web stranice. Uz *HTML* koriste se i ostale tehnologije koje služe za opisivanje izgleda/prezentacije web stranice (*CSS*) ili funkcionalnosti/ponašanja web stranice (*Javascript*) [3].

Značenje riječi „*Hypertext*“ odnosi se na poveznice koje povezuju web stranice, bilo to unutar same web stranice (interne poveznice) ili između dvaju ili više različitih web mjesta (eksterne poveznice). Na veze se gleda kao na temelji aspekt weba. Prijenosom sadržaja na Internet i povezivanjem sa stranicama koje su stvorili drugi ljudi, postaje se aktivnim sudionikom u *World Wide Web-u* [3].

Značenje riječi „*Markup*“ koristi se za napomenu teksta, slika i drugog sadržaja koji se prikazuje u web pregledniku. *HTML* za označavanje koristi različite elemente poput `<head>`, `<title>`, `<body>`, `<header>`, `<footer>`, `<article>`, `<section>`, `<p>`,

, , <img>, <aside>, <audio>, <canvas>, <datalist>, <details>, <embed>, <nav>, <output>, <progress>, <video>, <ul>, <ol>, <li> i mnogi drugi. HTML element izdvaja se iz drugog teksta pomoću oznaka koje se sastoje od naziva elementa okruženog sa „<“ i „>“. Naziv elementa unutar oznaka ne razlikuje velika i mala slova pa se tako primjerice oznaka za naslov (title) može napisati malim ili velikim slovima (<title> ili <TITLE>) [3].

U sljedećih će nekoliko redova biti prikazano kako se koriste *HTML* elementi na primjeru jednostavne web stranice koja ispisuje tekst „Pozdrav svijetu“. Web stranica može se izgraditi u bilo kojem tekstualnom uređivaču. Datoteka u kojoj se pište *HTML* kod mora biti spremljena s nastavkom .html kako bi bila prepoznata kao *HTML* datoteka. Jednom kada se datoteka kreira, može biti otvorena u bilo kojem web pregledniku [4].

```
<!DOCTYPE html>
<html>
<head>
<title>Pozdrav!</title>
<meta charset="UTF-8">
</head>
<body>
<h1>Pozdrav svijetu!</h1>
<p>Ovo je jednostavna web stranica.</p>
</body>
</html>
```

Na samom početku, uz pomoć <!DOCTYPE html> definirana je verzija *HTML*-a koja je korištena, u ovom slučaju to je *HTML5*. Nakon toga koristeći element <html> otvoren je početak web stranice koji se sastoji od sekcije glave (<head>) i sekcije tijela (<body>). Unutar glave definira se naslov web stranice i pomoću <meta> elementa, dodijeljeni su pregledniku meta podaci o dokumentu (u ovom slučaju kodiranje znakova). U sekciji tijela nalazi se naslov <h1> u kojem se nalazi tekst „Pozdrav svijetu“, a odmah nakon naslona definiran je paragraf <p> koji ispisuje tekst „Ovo je jednostavna web stranica“ [4].

2.2. CSS

Cascading Style Sheets (*CSS*) jezik je stila koji se koristi kako bi se opisala prezentacija dokumenta koji je napisan u *HTML*-u ili *XML*-u (uključujući *XML* dijalekte poput *SVG*, *MathML* ili *XHTML*). *CSS* opisuje na koji se način elementi definirani pomoću *HTML*-a

moraju prikazati na web stranici. Također, on je jedan od osnovnih jezika otvorenog web-a i standardiziran je putem web preglednika u skladu sa *W3C* specifikacijom. *CSS* se razvijao s godinama pa tako postoje tri verzije *CSS*-a, a to su *CSS1*, *CSS2* i *CSS3* koji se danas najkorišteniji uz *HTML5*. Postoji i verzija *CSS4*, koja nikada nije postala službena verzija [5].

Tri su načina na koji se može koristiti *CSS* unutar *HTML* dokumenta:

- Interni stil – kod internog stila, sav *CSS* kod piše se unutar glave (<head>) u *HTML* dokumentu, a započinje elementom <style>
- Eksterni stil – kod eksternog stila sva *CSS* pravila pišu se u zasebnom dokumentu koji se sprema s nastavkom .css i vrijede samo u onom *HTML* dokumentu u kojem su integrirani (integriraju se pomoću elementa <link>)
- Stil u retku (*inline*) – kod stila u retku, pravila su ugniježđena u *HTML*-u, što bi značilo da se nalaze unutar samog *HTML* elementa

Najčešće korišteni način za korištenje *CSS*-a u većim projektima i na većim web stranicama je eksterni stil. Sljedeći primjer prikazuje *HTML* dokument i eksterni *CSS* dokument. Unutar *HTML* dokumenta nalazi se kod za ispis naslova „Pozdrav svijetu“, dok *CSS* dokument prikazuje pravila za dodavanje stila tom naslovu.

HTML dokument „pozdrav-svijetu.html“:

```
<!DOCTYPE html>
<html>
<head>
<title>Pozdrav!</title>
<meta charset="UTF-8">
<link rel="stylesheet" type="text/css" href="style.css">
</head>
<body>
<h1>Pozdrav svijetu!</h1>
<p>Ovo je jednostavna web stranica.</p>
</body>
</html>
```

CSS dokument „eksterni-stil.css“:

```
h1 {
color: green;
text-decoration: underline;
}
```

Kako bi *HTML* i *CSS* dokumenti bili povezani, unutar glave *HTML* dokumenta ugrađena je linija koda koja se odnosi na poveznicu s eksternim CSS dokumentom (`<link rel="stylesheet" type="text/css" href="style.css">`). Atribut `rel` određuje odnos između trenutnog dokumenta i povezanog dokumenta, `type` određuje tip dokumenta, dok `href` određuje mjesto povezanog dokumenta. Unutar *CSS* dokumenta definiran je selektor h1 kojem se određuje kako će se prikazati na web stranici, a unutar vitičastih zagrada pišu se *CSS* pravila koja definiraju izgled. U ovom primjeru naslovu se dodjeljuje zelena boja i tekst će biti podcrtan [6].

Želi li se isti takav primjer izgraditi korištenjem internog *CSS*-a, kod bi izgledao ovako:

```
<!DOCTYPE html>
<html>
<head>
<title>Pozdrav!</title>
<meta charset="UTF-8">
<style>
h1 {
color: green;
text-decoration: underline;
}
</style>
</head>
<body>
<h1>Pozdrav svijetu!</h1>
<p>Ovo je jednostavna web stranica.</p>
</body>
</html>
```

Uz klasičan CSS, danas se sve više koriste CSS pretprocesori kao što su LESS i Sass koji omogućavaju lakšu izradu dizajna web stranica. CSS pretprocesor je program koji omogućuje generiranje CSS-a iz jedinstvene sintakse pretprocesora. Oni sastavljaju kod koji je napisan pomoću posebnog prevoditelja i zatim ga koriste za izradu CSS datoteke na koju se nakon toga poziva glavni HTML dokument [7].

Za upotrebu CSS pretprocesora potrebno je na svoj web poslužitelj instalirati CSS prevodilac ili upotrijebiti CSS pretprocesor za sastavljanje u razvojnom okruženju i zatim prenijeti sastavljenu CSS datoteku na web poslužitelj [7].

2.3. Javascript

Javascript je jezik koji je izgrađen na *ECMAScript* standardu. Nastao je u svibnju 1995. godine, a kreirao ga je *Brendan Eich*. Ideja *Javascript*-a je bila da se glavni dijelovi klijentske strane weba implementiraju u Javu tekako bi se HTML učinio interaktivnijim [11].

Javascript se uglavnom koristi kao klijentski jezik i slovi kao objektno orijentirani skriptni jezik koji se koristi za interaktivnost web stranica (npr. složene animacije, skočni izbornici itd.). Postoje i naprednije verzije *Javascript*-a na strani poslužitelja, kao što je Node.js. On omogućuje dodavanje više funkcionalnosti na web mjesto od jednostavnog preuzimanja datoteka u stvarnom vremenu. Unutar okruženja domaćina (npr. web preglednik) *Javascript* se može povezati s objektima svog okruženja kako bi im se osigurala programska kontrola [11].

Javascript sadrži standardnu biblioteku objekata kao što su nizovi (eng, *Array*), datumi (eng, *Date*) i matematičke funkcije (eng, *Math*). Također sadrži niz jezičnih elemenata kao što su operatori, upravljačke strukture i izrazi. Jezgra *Javascript*-a može se proširiti za različite svrhe dopunjavanjem dodatnim objektima kao što su [11]:

- Na strani klijenta (eng. *Client-side*) *Javascript* proširuje jezgru pružajući predmete za upravljanje preglednikom i njegovim dokumentnim modelom (*DOM-Document Object Model*). Na primjer, proširenja na strani klijenta omogućuju aplikaciji da postavlja elemente u *HTML* obrazac i reagira na korisničke događaje kao što su klik miša, unos obrazaca i navigacija [11].
- Na strani poslužitelja (eng. *Server-side*) *Javascript* proširuje jezgru jezika pružajući predmete relevantne za pokretanje *Javascript*-a na poslužitelju. Na primjer, proširenja na strani poslužitelja omogućavaju aplikaciji da komunicira s bazom podataka, pruža kontinuitet podataka s jednog poziva na drugi u aplikaciji ili vrši manipulacije datotekama na poslužitelju [11].

Javascript se danas nalazi na gotovo svakoj web stranici. Dok HTML služi za formatiranje statičnog sadržaja, *Javascript* omogućuje kreiranje dinamičkih web stranica. Primjerice, HTML omogućuje prikaz podebljanog teksta, kreiranja gumbića, dok *Javascript* omogućuje promjenu teksta na stranici, različite skočne prozore (eng. *Pop-up*) i sl.

2.3.1. Uvod u Javascript

Javascript kod može biti prikazan bilo gdje unutar `<html>` elementa. Početak *Javascript* koda započinje otvaranjem elementa i zatvaranjem elemenata

`<script></script>`.Kod se može pisati unutar elemenata ili se može kreirati poseban dokument u kojem će biti smješten samo *Javascript* kod i koji se poziva uz pomoć atributa *src* unutar elementa `<script>` [10].

Sve kreirane skripte moraju privremeno negdje spremati podatke. Za privremeno spremanje podataka koriste se varijable. Varijable se definiraju ključnom riječi var uz koje dolazi ime varijable. Varijable mogu spremati različite tipove podataka. Varijablama se pridružuje vrijednost uz pomoć operatora pridruživanja. Primjer kreirane varijable s imenom „varijablaPrimjer“ i tekstualnom vrijednošću „Prva varijabla“ [10]:

```
var varijablaPrimjer = „Prva varijabla“;
```

Novi *ECMAScript6* standard omogućuje kreiranje varijable uz pomoć ključnih riječi let i *const*. Kod kreiranja varijable vrlo je bitno u novom *ES6* standardu uzeti u obzir kada će se varijabla definirati sa *const*, a kada sa *let* [11]:

- *const* ukazuje na nepromjenjivo stanje i da varijabla nikad ne mijenja svoju vrijednost
- *let* označava da se vrijednost varijable mijenja

Grananja u *Javascript*-u se odnose na seriju instrukcija koju računalo provjerava jednu po jednu. Definira se pomoću naredbi *if-else* ili *switch-case*. Kod tih naredbi provjerava se točnost uvjeta i prema tome se izvršavaju određene naredbe. Sljedeći kod prikazuje primjere *if-else* i *switch-case* grananja [10].

```
if(5 > 3) {
console.log("5 je veće od 3!");
}
else {
console.log("3 je veće od 5!");
}
switch(broj) {
case 1:
console.log("Broj je 1!");
break;
case 2:
console.log("Broj je 2!");
break;
default:
onsole.broj("Nije broj!");
```

```
break;
}
```

Javascript sadrži petlja koje provjeravaju uvjet. Ako je uvjet unutar petlja istinit, kod će se dalje izvršiti, tako dugo do kad uvjet postavljen unutar petlja ne bude lažan. Postoje tri vrste petlji: *for*, *while*, *do while* [10]. *For* petlja koristi brojač kao uvjet. Tako se petlji zadaje koliko će se puta izvršiti bez obzira na rezultat uvjeta. *While* petlja izvršava sve dok je uvjet istinit. Prvo slijedi provjera uvjeta i nakon tog slijedi izvršavanje petlja ako je uvjet istinit. S druge strane, do *while* petlja izvršava se barem jednom zbog naredbe do bez obzira na rezultat uvjeta te nakon tog slijede provjere kao u *while* petlji. Slijedi primjer svih petlji.

```
for(let i = 0; i < 10; i++) {
console.log(i);
}
```

U ovom primjeru *for* petlje, brojač kreće od vrijednosti 0 i izvršava se tako dugo do kad vrijednost ne postane manja od 10 te ispisuje rezultat.

```
while(a > b) {
console.log("a je veće od b");
}
```

U primjeru *while* petlje, provjerava se je li uvjet istinit, odnosno je li vrijednost a veća od vrijednosti b. Ako jest, skripta će ispisati tekst „a je veće od b“. U protivnom, petlja se više neće izvršiti.

```
do {
console.log("a je veće b");
}
while(a > b);
```

U primjeru *do while* petlje, prvo će se ispisati tekst „a je veće od b“, a nakon toga slijedi provjera uvjeta i u daljnjem ciklusu provjera tekst će se ispisati samo ako je uvjet istinit.

2.3.2. Objekti i polja

Objekti grupiraju skup varijabli i funkcija za stvaranje modela nečega što bi se prepoznalo iz stvarnog svijeta. U objektu, varijable i funkcije preuzimaju nova imena. Unutar objekta varijable postaju poznate kao svojstva. Svojstva govore stvari o objektu, kao što naziv hotela ili broj soba u njemu. Svaki pojedini hotel mogao bi imati različito ime i različit broj soba. Ako pak je funkcija dio objekta, ona se naziva metodom. Metode predstavljaju zadatke koji su povezani s objektom. Na primjer, može se provjeriti koliko je soba dostupno oduzimanje broja rezerviranih soba od ukupnog broja soba [12].

U sljedećem primjeru prikazan je objekt koji predstavlja hotel. Objekt ima četiri svojstava i jednu metodu. Pohranjen je u varijabli koja se naziva hotel. Metoda sadržava funkciju koja vraća vrijednost dostupnih soba na temelju oduzetog broja rezerviranih soba od ukupnog broja soba [12].

```
let hotel = {
ime: 'Hotel Park',
sobe: 40,
rezervirano: 25,
teretana: true,
provjeriDostupnost: function() {
return this.sobe - this.rezervirano;
}
}
```

Polje ili niz stvara se tako da se stvori varijabla i da joj se ime kao i svakoj drugoj varijabli. Vrijednosti u nizu dodaju se unutar uglatih zagrada, a svaka vrijednost u nizu odvojena je zarezom. Vrijednosti u nizu ne moraju biti istog tipa, što znači da se u niz mogu spremati tekstualne vrijednosti, brojevi ili logički tip podataka. Primjer nizova s različitim tipovima podataka [12]:

```
const boje = ['crvena', 'zelena', 'plava'];
const ulica = ['Varazdinska', 32];
```

Svaka stavka u nizu ima svoj automatski dodani broj koji se naziva indeks. Indeks se koristi za pristup određenim stavkama u nizu. Brojač kreće od vrijednosti 0 koja se dodijeli prvoj stavki u nizu. Primjer ispisa zelene boje u prethodno definiranom nizu koji sadrži boje [12]:

```
console.log(boje[1]); // Ispisuje 'zelena'
```

Svaki niz ima svojstvo koje se naziva duljina (eng. *length*), koje sadrži broj stavki u nizu. To svojstvo može biti korisno kod korištenja petlji ili u nekim drugim slučajevima. Primjer ispisa dužine niza [12]:

```
console.log(boje.length); // Ispisuje 3
```

Također, nizovima se mogu mijenjati vrijednosti stavke ili dodavati nove. Primjer dodavanja i promjene vrijednosti [12]:

```
const boje2 = […boje, 'bijela']; // Dodavanje vrijednosti na
kraj niza
```

```
const boje3 = ['crna', …boje2]; // Dodavanje vrijednosti na
početak niza
boje3[0] = 'zuta' // Promjena crne boje u žutu
```

2.3.3. Funkcije i klase

Funkcije omogućuju grupiranje niza izjava zajedno za izvođenje određenog zadatka. Kako bi se bolje organizirao kod, različiti dijelovi skripte koji ponavljaju isti zadatak, kreiraju se kao funkcija (umjesto da se više puta ponavlja isti kod). Izrazi u funkciji izvršavaju se prilikom učitavanja stranice, tako da funkcije nude i način pohrane koraka potrebnih za postizanje zadatka. Skripta tada može zatražiti funkciju da izvodi sve one korake kada su potrebni [12].

Javascript ima dvije kategorije funkcija [11]:

- Obična funkcija koja može imati nekoliko uloga
  - Realna funkcija
  - Metoda
  - Funkcija konstruktora
- Specijalizirana funkcija koja može imati samo jednu ulogu
  - Funkcija strelice koja može biti samo stvarna funkcija
  - Metoda koja može biti samo metoda
  - Klasa koja može biti samo konstruktor funkcija

Na primjer, može postojati zadatak koji će se izvršiti samo ako korisnik klikne na određeni element na stranici. Funkciji se dodjeljuje njeno ime koje bi trebalo opisivati zadatak koji ona izvršava. Funkcija se uvijek mora pozvati kako bi se morala izvršiti. Slijedi primjer funkcije koja zbraja dva broja koja prima kao parametre [12]:

```
function zbrojiBrojeve(a, b) {
return a + b;
}
zbrojiBrojeve(2, 3); // Funkcija vraća broj 5
```

Klase su kod *Javascript*-a dodane kod *ECMAScript6* standarda. Temeljni dio većine klasa je njezin konstruktor koji postavlja početno stanje svake instance i rukuje bilo kojim parametrima koji su proslijeđeni. Klase su u stvari „posebne funkcije“, i baš kao što se definiraju izrazi funkcija , sintaksa klase ima dvije komponente: izraze klasa i deklaracije klasa. Slijedi primjer definiranja nove klase [13]:

```
class NovaKlasa {
```

```
constructor(ime, prezime){
this.ime = ime;
this.prezime = prezime;
}
ispisImena(){
console.log('Moje ime je' + this.ime + ', a prezime' +
this.prezime);
}
}
const ispis = new NovaKlasa('Martin', 'Lacko');
ispis.ispisImena();
```

U primjeru je kreirana klasa pod nazivom „NovaKlasa“ koja sadrži konstruktor u kojem prima parametre. Također, klasa sadrži funkciju „ispisImena“ koja vraća tekst sa ispisom imena i prezimena koje konstruktor sprema u varijable. Na samom dnu, kreira se nova varijabla u kojoj se kreira nova instanca klase i dodjeljuju joj se parametri te se poziva funkcija za ispis imena.

2.3.4. Rad s DOM standardom

Model objekta dokumenta (eng. *Document Object Model - DOM*) specificira kako bi preglednici trebali stvoriti model *HTML* stranice i kako *Javascript* može pristupiti i ažurirati sadržaj web stranice dok je ona u prozoru preglednika. *DOM* nije niti dio *HTML*-a niti dio *Javascript*-a. On je zaseban skup pravila i ugrađeni je od strane svih glavnih proizvođača preglednika te pokriva dva glavna područja [12]:

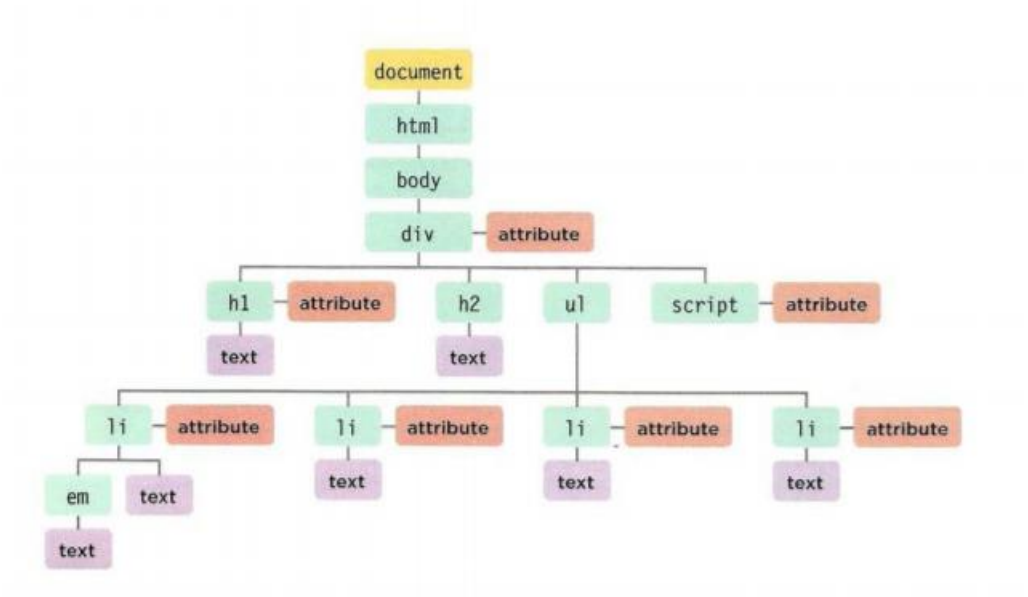
Izrada modela *HTML* stranica jedno je od dvaju glavnih područja. Kada preglednik učita web stranicu, tada stvara model stranice u memoriji. *DOM* se naziva objektni model jer je on model koji je načinjen od objekata. Svaki objekt predstavlja različit dio stranice učitane u prozoru preglednika [12].

Drugo glavno područje je pristup i promjena *HTML* stranica. *DOM* definira metode i svojstva za pristup i ažuriranje svakog objekta u ovom modelu, koji zauzvrat ažurira ono što korisnik vidi u pregledniku. Često ljudi *DOM* zovu Sučeljem za programiranje aplikacija (eng. *Application Programming Interface - API*). *API* dopušta programima (i skriptama) da međusobno komuniciraju [12]. DOM navodi što skripta može „pitati web preglednik o trenutnoj stranici“, i kako reći pregledniku što će korisniku biti prikazano.

Kako preglednik učitava web stranicu, on stvara model te stranice. Model se naziva *DOM* stablo i sprema se u memoriju preglednika. Sastoji se od četiri glavne vrste čvorova:

- Čvor dokumenta (eng. *The document node*)
- Čvor elementa (eng. *Element nodes*)
- Čvor atributa (eng. *Attribute nodes*)
- Čvor teksta (eng. *Text nodes*)

Svaki čvor je objekt s metodama i svojstvima. Skripte pristupaju i ažuriraju *DOM* stablo. Sve promjene na stablu *DOM* odražavaju se u pregledniku [12].



Slika 1. Stablo *Document Object Model*-a (Prema: Gilles Ruppert, Jack Moore, 2014.)

Pristupanje i ažuriranje *DOM* stabla uključuje dva koraka [12]:

1. Potrebno je pronaći čvor koji predstavlja element s kojim se želi raditi
2. Korištenje njegovog tekstualnog sadržaja, podređenih elemenata i atributa

Metode koje pronalaze elemente u *DOM* stablu nazivaju se *DOM* upiti (eng. *DOM queries*). Kada je potrebno raditi s nekim elementom više od jednom, trebalo bi ga spremiti u varijablu kao rezultat upita [12].

Slijedi primjer primjene *HTML* elementa pomoću *DOM*-a:

```
<div id="div1" style="background-color: black;">Element 1</div>
<!—HTML element -->
document.getElementById('div1').style.backgroundColor = „red“;
// Promjena boje elementa iz crne u crvenu
```

3. React

React je *Javascript* programski okvir koji je specijaliziran za pomoć programerima u izradi korisničkih sučelja. Kada se govori o web stranicama i web aplikacijama, korisnička sučelja predstavljaju zbirku zaslonskih izbornika, traka za pretraživanje, gumba i svega ostalog s čime netko koristi web mjesto ili aplikaciju. Prije *React*-a programeri su razvijali i gradili korisnička sučelja isključivo pomoću *Javascript*-a. To je označavalo dulje vrijeme razvoja i više mogućnosti za različite pogreške. Tako je 2011. godine inženjer na *Facebook*-u *Jordan Walke* kreirao React.js kako bi poboljšao i olakšao razvoj korisničkog sučelja [14].

Kako bi se izbjegla složenost interakcije i potreba za naknadnom obradom, *React* u potpunosti pruža potpunu obradu aplikacije. Temelji se na jednostavnosti tijeka rada. Sama ideja krenula je od toga da je *DOM* manipulacija skupa operacija i da bi trebala biti svedena na minimum. Također, prepoznaje se da bi ručna manipulacija *DOM* elementima mogla prouzrokovati velik broj programskog koda koji je sklon greškama, dosadan i ponavlja se. *React* to rješava na način da programeru daje virtualni *DOM* umjesto stvarnog *DOM*-a. Time obrađuje minimalan broj *DOM* operacija koje su potrebne za postizanje novog stanja [14].

Tijekom nastanka *React*-a, koncept komponenata postao je popularan. Svaka komponenta definira svoj izgled i stil pomoću *HTML*-a, *CSS*-a i *Javascript*-a. Jednom kad je komponenta definirana, može se koristiti u hijerarhiji komponenata za stvaranje cijele web aplikacije. Iako se *React* temelji na komponentama, okolni ekosustav ga čini fleksibilnim programskim okvirom [16].

React je također deklarativan, a to radi na način da tijekom promjene podataka, *React* konceptualno pritisne gumb za osvježavanje stranice i prepoznaje te ažurira samo promijenjene dijelove. Ovaj jednostavan tijek podataka, zajedno s jednostavnom logikom prikaza, čini razvoj pomoću *React*-a jednostavan i razumljiv. *React* koriste neke od najpopularnijih svjetskih usluga kao što su *Facebook*, *Instagram*, *Netflix*, *Alibaba*, *Yahoo*, *E-bay*, *Sony*, *Atlassian*, itd. [14].

3.1. Instalacija React-a

React je od samog početka osmišljen za postepeno usvajanje, što bi značilo da ga se može koristiti onoliko malo ili puno koliko je potrebno. Možda se primjerice želi dodati samo dio interaktivnosti na neku postojeću stranicu, a *React* komponente su sjajan način da se to postigne [15].

U nastavku će biti prikazano kako se na vrlo jednostavan način kreira *React* aplikacija bez posebnih instalacija. Prvi korak je da se kreira *HTML* stranica koja se želi uređivati i na koju se želi dodati *React*. U toj stranici kreira se `<div>` element kojem se doda poseban identifikator preko kojeg će se kasnije reći *Javascript*-u gdje da se prikaže komponenta [15].

Primjer web stranice:

```
<!-- ... postojeći HTML ... -->
<div id="react_komponenta"></div>
<!-- ... postojeći HTML ... -->
```

Sljedeći korak je dodavanje `<script>` elemenata u web stranicu, neposredno prije zatvaranja `</body>` elementa.

```
<!-- ... ostatak HTML stranice ... -->
<!-- React skripte -->
<script src=https://unpkg.com/react@16/umd/react.development.js
crossorigin></script>
<script src="https://unpkg.com/react-dom@16/umd/react-
dom.development.js" crossorigin></script>
<!-- Učitavanje React komponente -->
<script src="komponenta.js"></script>
</body>
```

Treći i posljednji korak u ovom procesu postavljanja *React* komponente je kreiranje novog dokumenta koji će se zvati komponenta.js. Unutar tog dokumenta kreira se nova komponenta i na kraju dokumenta potrebno je dodati dvije linije koda koje će povezati `<div>` element s *React* komponentom [15]:

```
constdomContainer=document.querySelector('#react_komponenta');
ReactDOM.render(e(Komponenta), domContainer);
```

Uz ovaj način ubacivanja *React* komponenata unutar web stranice, za veće projekte, odnosno web aplikacije, postoji instalacija *React* paketa na računalo i kreiranje cijelog *React* okruženja unutar komandnog prozora. Kako bi se instalirao *React* paket, potrebno je na računalu imati instaliran *Node.js*. Tako se u komandnom prozoru pokreće naredba za instalaciju [16]:

```
npm install react
```

React paket može se instalirati zasebno unutar datoteke gdje se želi kreirati web aplikacija ili pa globalno na cijelom računalu, a za to je potrebno u prethodnu naredbu dodati

`-g`. Nakon izvršene instalacije, uz pomoć npm naredbe postoji mogućnost postavljanja cijelog *React* okruženja, a to se izvršava naredbom [15]:

```
npm create-react-app naziv-aplikacije
```

Aplikacija koja je kreirana prethodnom naredbom ne upravlja sigurnosnom logikom ili bazama podataka. Ta naredba otvara cjevovod za izgradnju korisničkog sučelja koje se može spojiti sa bilo kojim pomoćnim pozadinama (eng. *backend-om*) [15].

3.2. Komponente

Najveći dio razvoja u *React*-u je stvaranje komponenata. Komponenta predstavlja dio aplikacije i može se definirati kao ES6 klasa koja proširuje osnovnu klasu *React.Component*. U svom minimalnom obliku, komponenta mora sadržavati metodu „vraćanje“ (eng. *render*) koja određuje kako se komponenta predstavlja *DOM*-u, odnosno koji će se podaci ili elementi prikazati i izvesti iz te komponente. Metoda vraća čvorove koji se mogu definirati *JSX* sintaksom kao *HTML* oznake koja će biti objašnjena niže. Slijedi primjer definiranja *React* komponente koja se naziva „Pozdrav“ [2]:

```
import React from 'react'
class Pozdrav extends React.Component {
render() {
return <h1>Pozdrav Svijetu!</h1>
}
}
export default Pozdrav
```

Primjer prikazuje kako se u komponenti unutar metode `render()` ispisuje tekst „Pozdrav Svijetu!“ u obliku *HTML* `<h1>` elementa.

Komponenta može biti izvezena unutar neke druge komponente ili direktno u *DOM*, koristeći ReactDOM.render [2]:

```
import React from 'react'
import ReactDOM from 'react-dom'
import Pozdrav from './Pozdrav'
ReactDOM.render(<Pozdrav/>, document.getElementById('main'))
```

Kako se *React* aplikacija temelji na kreiranju komponenata, i raščlanjivanju aplikacije na manje dijelove koji se mogu koristiti na više mjesta i tako zapravo se riješiti ponavljajućeg programskog koda, komponente se mogu pozivati unutar drugih komponenata. Komponenta

Pozdrav iz prethodnog može se pozvati i izvesti unutar neke druge komponente i na taj način ispisati naslov „Pozdrav Svijetu!“:

```
import React from 'react'
import Pozdrav from './Pozdrav'
class DrugaKomponenta extends React.Component {
render() {
return <Pozdrav />;
}
}
export default DrugaKomponenta
```

3.2.1. JSX (Javascript Syntax Extension)

Javascript Syntax Extension je *XML*-sintaktično proširenje na *ECMAScript* bez bilo kakve definirane semantike i dozvoljava pisanje HTML elemenata unutar *Javascript*-a i stavljanja tih elemenata u *DOM* bez potrebnih `createElement()` ili `appendChild()` metoda. Nije namijenjeno tome da ga implementiraju pretraživači niti da se ubacuje u sam spektar *ECMAScript*-a, već da ga koriste različiti procesori koji će ga transformirati u standardni *ECMAScript*. Svrha ove specifikacije je definirati poznatu i konzistentnu sintaksu. Umetanje nove sintakse u već postojeći jezik je rizičan pothvat. Kroz samostalnu specifikaciju olakšava se implementatorima drugih proširenja sintakse da razmotre *JSX* tijekom dizajniranja svoje vlastite sintakse. To bi omogućilo postojanje novih različitih proširenja sintakse. Ova specifikacija ne pokušava biti u skladu s bilo kojom *XML* ili *HTML* specifikacijom, nego je zamišljena kao *ECMAScript* značajka, a sličnost s *XML*-om služi samo za upoznavanje [17].

Primjer pisanja elemenata bez *JSX*-a:

```
const mojElement = React.createElement('h1', {}, 'Bez koristenja JSX!');

ReactDOM.render(mojElement,document.getElementById('root'));

Primjer pisanja elemenata sa JSX-om:

const mojElement = <h1>Primjer sa JSX!</h1>;

ReactDOM.render(mojElement,document.getElementById('root'));
```

Sa *JSX*-om mogu se pisati različiti izrazi unutar vitičastih zagrada. Izraz može biti *React* varijabla, svojstvo ili bilo kakav drugi validan *Javascript* izraz. *Javascript* će izvršiti izraz i vratiti rezultat. Primjer izraza sa *JSX*-om [2]:

```
const mojElement = <h1>5 + 5 = {5 + 5}</h1>; // Rezultat je: 5 + 5 = 10
```

3.2.2. Babel

Današnji preglednici još uvijek ne razumiju *ES6* ili *ES7* kod koji se piše. Zbog toga se koristi *Babel* kako bi se taj kod pretvorio u *ES5* sintaksu koju današnji preglednici razumiju i mogu pročitati. *Babel* omogućuje pisanje nove *Javascript* sintakse i također daje podršku za prethodno spomenuti *JSX*. Zbog navedenih korisnih stvari, vrlo se često koristi zajedno s *React*-om. Moderan je i dolazi s arhitekturom dodataka. Ima dodatke za različite *ES6* i *ES7* sintakse. *Babel* je potrebno instalirati u okruženje u kojem se razvija web aplikacija [14]:

```
npm install --save-dev @babel/preset-react
```

Tijekom kreiranja cijelog okruženja uz pomoć naredbe `create-react-app`, *Babel* će biti automatski instaliran i nije potrebno izvršavati samostalnu instalaciju. *Babel* pokušava koristiti najmanju moguću količinu koda bez ikakve ovisnosti o glomaznom trajanju. To nije uvijek moguće izvesti i uvijek postoje „labave“ opcije za specifične transformacije koje mogu kompliciratи specifičnost za čitljivost, veličinu datoteke i brzinu [14].

3.2.3. Dodavanje komponente na DOM

Nakon kreiranih komponenata, u kojima se koristi *JSX* sintaksa, *Babel* transformator za pretvorbu *ES6* ili *ES7* koda u *ES5* kod, potrebno je komponente povezati na *DOM*.

Kako bi se glavna početna komponenta prikazala na sučelju potrebno ju je postaviti direktno u *DOM*. To se radi na način da se koristi `ReactDom.render` u kojem se navode dva argumenta: komponenta i *DOM* čvor koji ukazuje na to gdje će se element prikazati [2].

Na primjer, postoji li negdje u aplikaciji `id` atribut naziva `main`, dodavanje komponente *Pozdrav* izgledalo bi ovako:

```
import React from 'react'
import ReactDOM from 'react-dom'
import Hello from './Hello'
ReactDOM.render(<Hello name="Billy James" />,
document.getElementById('main'))
```

3.3. Dinamičke komponente

Dinamičke komponente su komponente koje se prikazuju dinamički ovisno o tome koji su im podaci proslijeđeni. Podaci se mogu prosljeđivati s komponente roditelja prema komponenti djeteta ili u obrnutom smjeru.

3.3.1. Hijerarhija komponenata

Komponente u *React*-u mogu se odrediti hijerarhijski. Tako postoje komponente roditelji (eng. *parent*) i komponente djeca (eng. *child*). Kako prepoznati koje su komponente na kojoj hijerarhijskoj razini? Kako bi neka komponenta preuzela status komponente dijete potrebno ju je pozvati i prikazati unutar `render()` funkcije druge komponente koja će tako postati komponenta roditelj. Tok podataka između komponenata ide od vrha prema dnu, odnosno od roditelja prema djetetu, no ni obrnuti slijed podataka nije neobičan i također se koristi [2].

Lakši način za prosljeđivanje podataka je s roditelj komponente prema komponenti dijete. To je zapravo i prirodan tok podataka u *React*-u. Slijedi primjer u kojem se prosljeđuje poruka komponenti dijete.

```
import React from 'react';
// Komponenta Roditelj
class Roditelj extends React.Component {
render() {
const varijabla = 5;
return (
<div>
<Dijete poruka="poruka za komponentu dijete" />
<Dijete poruka={varijabla} />
</div>
);
}
}
// Komponenta Dijete
class Dijete extends React.Component {
render() {
return <h1>{this.props.poruka}</h1>
}
```

```
}
export default Roditelj;
```

U primjeru iznad, komponenta *Roditelj* poziva dva puta komponentu *Dijete*, u prvoj liniji koda joj šalje poruku, dok u drugoj liniji koda joj šalje vrijednost iz varijable. Kako bi komponenta dijete mogla dohvatiti vrijednost koja joj se šalje, toj vrijednosti se pristupa naredbom `this.props`.

S druge strane, slanje podataka iz komponente dijete natrag komponentu roditelji nije prirodan način na koji *React* funkcionira. Kako bismo to postigli potrebno je proslijediti funkciju vrijednost s komponente roditelja komponenti dijete, i u komponenti dijete pozvati tu funkciju. U primjeru je prikazano kako se radi promjena stanja tako što će se funkcija prenijeti na komponentu dijete i pozvati se unutar nje [2].

```
import React from 'react';
// komponennta Roditelj
class Roditelj extends React.Component {
constructor(props) {
super(props);
this.state = { zbroj: 0 };
this.outputEvent = this.outputEvent.bind(this);
}
promjenaZbroja() {
this.setState({ zbroj: this.state.zbroj++ });
}
render() {
return (
<div>
Zbroj: { this.state.zbroj }
<Dijete klikDogadaj={this.promjenaZbroja} />
</div>
);
}
}
// komponenta Dijete
class Dijete extends React.Component {
render() {
return (
```

```
<button onClick={this.props.promjenaZbroja}>
Dodaj
</button>
);
}
}
export default Roditelj;
```

3.4. State i props objekti

State objekti unutar komponente ključni su za upravljanje i komunikaciju podataka unutar aplikacije. Oni su zapravo predstavljeni kao *Javascript* objekti i imaju opseg razine komponente. *State* objekt nije nužno postavljati unutar svake komponente, već kada je to potrebno. Sljedeći primjer prikazuje postavljenje *state* objekta [2]:

```
var App = React.createClass({
state = {
pozdrav: 'Dobar dan!'
}
render: function(){
console.log(this.state.pozdrav); // ispisuje "Dobar dan!"
});
```

U ovom primjeru postavljen je *state* objekt na sam vrh komponente, gdje je zapravo i zamišljeno da se on definira. State-u se pristupa na način da se poziva `this.state` naredba i nakon nje se definira kojem svojstvu *state* objekta se želi pristupiti. U ovom primjeru pristupa se svojstvu pozdrav, pa se navodi `this.state.pozdrav`.

Uz to što se *state*-u može pristupiti, u *state* se mogu i postavljati različiti podaci. Za to se koristi metoda `setState()`. To je primarni način na koji se ažurira korisničko sučelje. Metoda zahtjeva obavezan parametar koji se naziva „ažurirač“ koji može biti i najčešće je objekt koji sadrži par ključ-vrijednost koji se spoje u *state*. Drugi parametar koji je opcionalan je funkcija koja se izvršava nakon uspješnog izvoženja `setState` metode. Ta funkcija može se koristiti u slučaju da se želi izvršiti neka radnja nakon što bude sigurno da se metoda izvršila uspješno [2].

Primjer korištenja metode:

```
this.setState({ pozdrav: 'Dobro jutro!' });
```

S druge strane, *props* objekt se koristi za prijenos podataka i metoda iz roditeljske komponente u podređenu komponentu, odnosno komponentu dijete. *Props* objekti su nepromjenjivi i omogućuju stvaranje komponenata koje se mogu više puta koristiti.

```
// komponenta roditelj
class Roditelj extends React.Component{
render() {
return <div>
<Child naslov1="Naslov 1" />
<Child naslov2="Naslov 2" />
</div>
}
}
// komponenta dijete
class Dijete extends React.Component{
render() {
return <div>
<h1>{this.props.naslov1}</h1>
<h2>{this.props.naslov2}</h2>
</div>
}
}
```

U ovom primjeru prikazano je prosljeđivanje dvaju naslova iz komponente roditelj u komponentu dijete. Kako bi se unutar podređene komponente moglo pristupiti *props* objektu, korištena je naredba `this.props`.

Glavna razlika između *props* i *state* objekata je u tome što su *props* objekti nepromjenjivi. Njih se ne bi trebalo ažurirati unutar komponente kojoj su proslijeđeni. Oni su u vlasništvu komponente u kojoj su definirani. S druge strane, *state* objekti su nešto što je u „privatnom vlasništvu“ komponente i mogu biti promijenjeni ovisno o interakcijama s vanjskim svijetom [14].

3.5. Virtualni DOM

Svaka web stranica je predstavljena kao interno stablo objekata, a taj se prikaz naziva *Document Object Model – DOM*. *DOM* je jezično neutralno sučelje koje programskim jezicima omogućava pristup *HTML* elementima. No rad s *DOM*-om je zapravo „poprilično

skupo“. Zbog toga je kod *React* programskog okvira kreiran virtualni *DOM* kao rješenje koje služi za praćenje onoga što se promijenilo u stvarnom *DOM*-u [14].

Koncept se temelji na tome da *React* uvijek čuva kopiju prikaza stvarnog *DOM*-a u memoriji. Kad god se nešto promijeni, primjerice kao *state* objekt, izračunava se novi primjerak *DOM*-a koji će biti kreiran s novim *state* objektom. Tada se izračunava razlika između originalne kopije virtualnog *DOM*-a i nove kopije virtualnog *DOM*-a. Razlika izračuna su zapravo one operacije koje se moraju napraviti na stvarnom *DOM*-u. Na ovaj način *React* ne mora napraviti velike promjene i ubrzava se cijeli proces rada web aplikacije [14].

U svakom trenutku *React* ima stanje aplikacije koje je predstavljeno kao virtualni *DOM*. Kad god se stanje aplikacije promjeni, koraci koji se izvršavaju su sljedeći:

1. Stvara se novi virtualni *DOM* koji predstavlja novo stanje aplikacije
2. Usporedba starog *DOM*-a sa novim virtualnim
3. Na temelju koraka broj dva, pronalazi se minimalan broj operacija za transformaciju starog virtualnog *DOM*-a u novi virtualni *DOM*
4. Nakon što su operacije pronađene, preslikavaju se u njihove ekvivalentne operacije na *HTML DOM*-u
5. *DOM* operacije se primjenjuju direktno na *HTML DOM* aplikacije, što značajno štedi vrijeme koje je potrebno da se obave promjene [2]

3.6. Svojstva React programskog okvira

Svojstva *React* programskog okvira zapravo su dodaci koje on koristi kako bi mu pomogla lakše savladati neke od zadataka koje mora izvršiti. Primjerice, svojstva služe kako bi pomogla u kretanju kroz web aplikaciju, slanje različitih zahtjeva, dohvaćanje *API*-ja (*Application programming interface*) ili pak lakšeg pisanja strukture *HTML*-a unutar programskog okvira.

3.6.1. React Aux

React-aux predstavlja komponentu za prikaz više elemenata. Kako ne bi kroz cijelu web aplikaciju bilo potrebno stavljati više elemenata unutar div bloka, *aux* pruža mogućnosti definiranja nevidljivog bloka unutar kojeg je moguće pisati normalan redoslijed *HTML* elemenata.

Kako bi se u *React*-u vratilo više elemenata unutar komponente potrebno ih je postaviti unutar pomoćnog elementa:

```
const Root = () => {
return <div>
<p>Pozdrav Svijetu!</p>
<p>Ja sam paragraf.</p>
</div>;
};
Rezultat ove komponente je:
<div>
<p>Pozdrav Svijetu!</p>
<p>ja sam paragraf.</p>
</div>
```

Kako bi se izbjeglo konstantno ponavljanje pomoćnog div elementa koji se zapravo u *HTML*-u vraća kao rezultat, svojstvo *react-aux* omogućilo je pomoćnu komponentu unutar koje je moguće pisati i vratiti više elemenata:

```
const Root = () => {
return <Aux>
<p>Pozdrav Svijetu!</p>
<p>Ja sam paragraf.</p>
</Aux>;
};
Kao rezultat ove komponente dobije se:
<p>Pozdrav Svijetu!</p>
<p>Ja sam paragraf.</p>
```

U ovom primjeru vraćeni su čisti *HTML* elementi bez nepotrebnog ponavljanja div bloka.

3.6.2. React router

React router je *React* svojstvo koje omogućuje kretanje unutar web aplikacije. Vrlo je važno ispravno ga implementirati i koristiti unutar koda. Kako bi se započelo sa korištenjem aplikacije, potrebno je unutar radnog okruženja instalirati *React router* [18]:

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

Nakon instalacije potrebno je imati kreirane komponente koje će prikazivati sadržaj i kroz koje će se biti moguće kretati. Slijedi primjer komponenata kojima upravlja *React router* [18]:

```
import React from "react";
import Aux from './Aux';
import {
BrowserRouter as Router,
Switch,
Route,
Link
} from "react-router-dom";
export default function App() {
return (
<Router>
<Aux>
<nav>
<ul>
<li>
<Link to="/">Početna</Link>
</li>
<li>
<Link to="/onama">O nama</Link>
</li>
<li>
<Link to="/korisnici">Korisnici</Link>
</li>
</ul>
</nav>
<Switch>
<Route path="/onama">
<Onama />
</Route>
<Route path="/korisnici">
<Users />
</Route>
<Route path="/">
<Pocetna />
</Route>
```

```
</Switch>
</Aux>
</Router>
);
}
function Pocetna() {
return <h2>Početna</h2>;
}
function Onama() {
return <h2>O nama</h2>;
}
function Korisnici() {
return <h2>Korisnici</h2>;
}
```

Ovaj primjer prikazuje tri komponente između kojih *React router* vrši kretanje. U ovom primjeru korištena je prethodno opisana komponenta *aux* koja omogućuje definiranje i vraćanje više elemenata unutar komponente. *Switch* element koji se nalazi unutar *router*-a zapravo prolazi kroz podređene elemente i traži prvi link koji se poklapa sa onim kojeg je korisnik odabrao kao sljedeću destinaciju na aplikaciji i tako zapravo daje *routeru* do znanja gdje mora usmjeriti korisnika, na koju komponentu. Link element zapravo je zamjena za a *HTML* element koji služi za navigaciju. Unutar *Link*-a određuje se ruta na koju će se *router* usmjeriti, dok se unutar *Route* elementa definiraju komponente koje će biti prikazane i određuje se *path* atribut koji označava link do komponente.

3.6.3. Axios

Gotovo svaka veća aplikacija koristi nekakav *API* pomoću kojeg dohvaća podatke u obliku *JSON* objekata. *Axios* je definiran kao *HTTP* klijent. Temelji se na obećanima (eng. *promise*) i stoga se može koristiti asinkroni kod i čekati za čitljiviji odgovor. Također, *Axios* omogućuje presretanje i otkazivanje zahtjeva, a postoji i ugrađena zaštita na strani klijenta od krivotvorenja zahtjeva na različitim web lokacijama. *Axios* je još jedno od svojstava koje se mogu priključiti *React*-u i potrebno ga je prvotno instalirati u samo okruženje [19]:

```
npm install axios
```

Axios nudi mogućnosti slanja i primanja zahtjeva. *GET* zahtjev omogućuje dohvaćanje podataka:

```
axios.get(`https://jsonplaceholder.typicode.com/users`)
.then(response => {
const persons = response.data;
this.setState({ persons });
})
```

Tijekom slanja zahtjeva potrebno je navesti *API* kojem će se pristupiti i iz kojeg će se uzimati podaci. Nakon toga postavlja se metoda `then` koja čeka da se zahtjev izvrši i kao argument prima podatke koji su dohvaćeni. Podaci su u primjeru spremljeni u *state* objekt.

POST zahtjevi omogućuju slanje podataka i primanje odgovora. Slijedi primjer *POST* zahtjeva:

```
const user = {
name: this.state.name
};
axios.post(`https://jsonplaceholder.typicode.com/users`, { user
})
.then(response => {
console.log(response.data);
})
```

U ovom primjeru definirana je varijabla *user* koja sadrži objekt u kojem je spremljeno ime korisnika koje se dohvatilo iz *state* objekta. Nakon toga šalje se *axios* zahtjev u kojem se navodi API kojem se šalju podaci, zatim se navodi varijabla *user* gdje su spremljeni podaci koji se šalju i nakon toga se u metodi `then` dohvaća odgovor zahtjeva.

4. React Redux

Kao uvod u ovo poglavlje potrebno je naglasiti da *Redux* nije dio *React*-a već ga mogu koristiti i drugi programski jezici ili okviri kao što su *Javascript*, *Angular*, *jQuery* i sl. *Redux* je knjižnica uz pomoć koje se unutar web aplikacije održava jedno stablo (*state* objekt) kojem se pristupa iz cijele aplikacije. Popularan je kod *React*-a zbog toga jer jako dobro funkcionira s njime i omogućuje da se korisničko sučelje temelji na *state* objektu koji se konstantno ažurira pozivajući akcije [20].

Redux je kao takav potrebno prvotno instalirati u radno okruženje kako bi ga mogli koristiti:

```
npm install react-redux
```

React aplikacija koja koristi *Redux* odvaja komponente koje koriste *state* objekt kao spremnik i prezentacijske komponente koje nemaju svoj vlastiti spremnik, već koriste *Redux* preko kojeg pristupaju glavnom *state* objektu. Glavne razlike između prezentacijskih komponenata i komponenata sa spremnikom su [20]:

Tabela 1. Razlika između prezentacijske i komponente sa *state* objektom (Prema: https://redux.js.org/basics/usage-with-react)

	PREZENTACIJSKA KOMPONENTA	**KOMPONENTA SA *STATE* OBJEKTOM**
SVRHA	Definirati izgled komponente (dizajn i sl.)	Omogućiti rad komponente (dohvaćanje podataka, promjena *state* objekta)
SVJESNA REDUX-A	Ne	Da
ČITANJE PODATAKA	Čita podatke sa *props* objekta	Pretplaćuje se na *Redux state*
PROMJENA PODATAKA	Pozivanje povratnih poziva sa *props* objekta	Pozivanje Redux akcija
IZGRAĐENA	Ručno	Generira ju Redux

Većina komponenata o kojima je do sad bilo pisano je bila prezentacijska, ali u velikim web aplikacijama koje podržavaju veliku količinu podataka potrebno je definirati

nekoliko komponenata koje će se povezati s *Redux*-om. Kako bi se komponente mogle povezivati s *Redux*-om potrebno je kreirati akcije.

4.1. Akcije

Akcije su objekti koji sadrže informacije na temelju kojih se može ažurirati glavno skladište, odnosno glavni *state* objekt *Redux*-a. Umjesto da se izravno promjeni stanje, specificiraju se mutacije koje će pomoću običnih objekata koji se nazivaju akcijama odraditi taj posao. Tada se izradi posebna funkcija koja se zove reduktor u kojoj se odlučuje kako i na koji način svaka radnja treba promijeniti stanje cijele aplikacije [20].

```
const mapDispatchProps = dispatch => {
return {
povecaj: () => dispatch({ type: 'POVECAJ' }),
smanji: () => dispatch({ type: 'SMANJI' })
};
}
```

U primjeru iznad prikazana je `mapDispatchProps` funkcija unutar koje se uz pomoć `dispatch` metode pozivaju akcije koje su zapisane kao funkcije. Akcije su nazvane `povecaj` i smanji i odnose se na tip akcije koji će kasnije biti zapisan u reduktoru koji će tako prepoznati koja se akcija pozvala i na temelju toga promijeniti glavni *state* objekt.

4.2. Reduktor

Reduktori su funkcije koje određuju kako će se promijeniti stanje aplikacije. Prethodno spomenute akcije opisuju samo ono što se dogodilo, ali ne opisuju kako se mijenja stanje aplikacije. Unutar *Redux*-a, stanje aplikacije se pohranjuje kao jedan objekt. Vrlo je važno da se prije nego što se postavi i napiše kod, razmisli o tome kako će taj objekt izgledati, odnosno kakav će biti njegov oblik. Slijedi primjer reduktora [20]:

```
const reduktor = (stanje = [], akcija) => {
switch (akcija.type) {
case POVECAJ:
return brojac+1;
case SMANJI:
return brojac-1;
default:
```

```
return brojac;
}
}
```

Unutar funkcije *recenice* nalazi se reduktor koji se temelji na *switch-case* uvjetima. Svaki *case* vraća tip akcije koji definira kako će se stanje funkcije promijeniti. Tako u ovom primjeru prvi uvjet povećava varijablu *brojac* za jedan, dok drugi uvjet smanjuje varijablu *brojac* također za jedan. *Default* vraća varijablu *brojac* bez da radi ikakve promjene nad njom.

Ako se *Redux* koristi unutar velike i složene web aplikacije potrebno je napraviti funkciju koja spaja manje reduktore u jedan veći reduktor kako bi se lakše moglo mijenjati stanje aplikacije i snalaziti unutar reduktora. Tada se unutar te funkcije pozivaju svi reduktori s istom akcijom [20].

4.3. Skladište

Skladište se definira kao objekt koji spaja sva stanja koja se mijenjaju unutar aplikacije. Ima sljedeće odgovornosti [20]:

- Omogućuje pristup *state* objektu putem metode `getState();`
- Omogućuje ažuriranje stanja putem akcije
- Registrira slušatelje (eng. *listeners*) putem pretplate (eng. *subscribe*)
- Rukuje ne-registracijom slušatelja putem funkcije koju je vratio pretplatnik (slušatelj)

Unutar svake *Redux* aplikacije postoji samo jedno skladište koje se definira kao objekt. Kada se želi podijeliti logika upravljanja podacima, koriste se reduktori umjesto manjih skladišta [20].

Vrlo je jednostavno postaviti skladište ako je već izrađen reduktor. Kod kombiniranja reduktora koristi se metoda `combineReducers()` koja se uvodi i prosljeđuje u `createStore()` [20]:

```
const pocetnoStanje = {
brojac: 0;
}
const store = createStore(reduktor, pocetnoStanje);
```

Varijabla *brojac* koja se koristi u prethodnim primjerima definirana je u skladištu, točnije unutar objekta *pocetnoStanje* kojem se pristupa pozivom funkcije `createStore()` koja je smještena unutar varijable *store*.

5. Razvoj web aplikacije

Kako bi sav teorijski sadržaj mogao biti prikazan u praksi, u ovom radu izrađena je web aplikacija u *React.js* programskom okviru. Web aplikacija služi za provjeru dostupnosti i originalnosti imena novog poduzeća na području Republike Hrvatske i nazvana je *Originality*. Aplikacija provjerava novo izmišljeno ime korisnika i pregledava postoje li već rezervirane domene na to ime, nalazi li se to ime u Sudskom registru te je li već to ime zauzeto na društvenim mrežama kao što su *Facebook*, *Instagram* i *Youtube*.

Slika 2. Logotip izrađene web aplikacije (vlastita izrada)

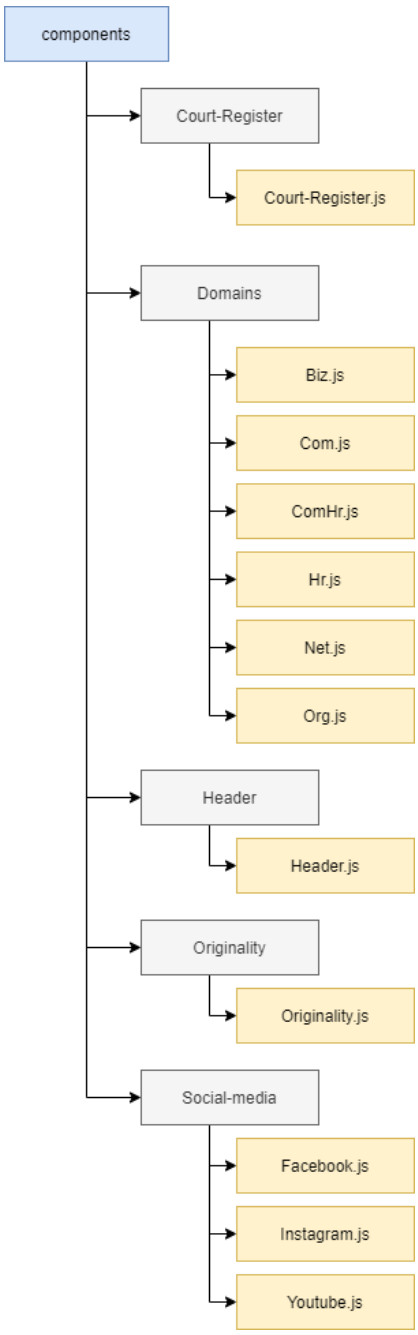
Ovu aplikaciju u potpunosti može koristiti bilo koji korisnik bez ikakve registracije ili prijave u sustav gdje se traže osobni podaci. Ona korisnicima u dva najjednostavnija koraka pruža potpunu uslugu. Potrebno je samo upisati novo osmišljeno ime i pritisnuti gumb za provjeru. Aplikacija je zamišljena za sve vrste korisnika i zbog toga je izrađena s vrlo jednostavnim sučeljem za korištenje.

Za provjeru dostupnosti imena, aplikacija koristi različite *API*-je za dohvaćanje potrebnih podataka kako bi se utvrdilo postoji li već registrirano ime ili ne. Stoga u ovoj web aplikaciji nije potrebno kreirati nikakvu bazu podataka već je bilo potrebno više pažnje posvetiti na organizaciju komponenata i čim brže dohvaćanje podataka.

5.1. Struktura web aplikacije

Kao što je navedeno, ova web aplikacija ne koristi bazu podataka i vrlo je bitno da je struktura web aplikacije dobro sastavljena. *React.js* programski je okvir koji omogućuje izgradnju pojedinih komponenata i sastavljanja tih komponenata u jednu aplikaciju. Tako je i

izrađena web aplikacija u ovome radu. Sljedeća slika prikazuje strukturu web aplikacije na temelju komponenata:



Slika 3. Struktura web aplikacije (vlastita izrada)

Web aplikacija sastoji se od pet datoteka koje sadržavaju nekoliko komponenata. Datoteke su podijeljene na sljedeći način: datoteka *Header* sadržava komponentu koja čini početno sučelje web aplikacije. Unutar datoteke *Domains*, sadržane su komponente koje se odnose na samu logiku provjere dostupnosti domena novog imena korisnika. *Social-media* datoteka sadržava komponente koje se odnose na društvene mreže i provjeru njihove dostupnosti, dok *Court-Register* datoteka sadržava komponentu za provjeru dostupnosti

imena u Sudskom registru. Na kraju je datoteka *Originality* koja sadržava komponentu koja daje cjelokupni rezultat svih navedenih provjera.

5.2. Razvoj komponenata

Prije razvoja komponenata i cijele web aplikacije potrebno je stvoriti okruženje rada. Najprije je potrebno pokrenuti skriptu koja će sama generirati sve potrebne dodatke i postaviti početnu komponentu na koju će se vezati sav razvoj aplikacije:

```
npm create-react-app originality
```

Nakon uspješne instalacije i prije samog kreiranja komponenata, instalirana su svojstva koja će pomoći u izgradnji ove web aplikacije. Svojstva koja su instalirana su *axios* za dohvaćanje podataka, *aux* za pisanje *HTML* elemenata unutar *ES6* sintakse i *React-router* za navigaciju unutar aplikacije.

```
npm install aux
npm install react-router
npm install axios
```

Kada je radno okruženje spremno, kreće se sa izgradnjom komponenata. Prva komponenta koja se gradi prema logičkom slijedu je komponenta *Header*. To je komponenta koja će većinskim dijelom sadržavati dizajn početnog sučelja. Na njoj će biti prikazan logotip aplikacije, izbornik, opis glavnih funkcionalnosti i polje za unos u koje korisnik upisuje ime koje želi provjeriti. Glavna funkcionalnost osim prikaza početnog sučelja ove komponente jest da sprema unos imena za provjeru, sprema ga u svoj *state* objekt i šalje ga komponenti *Results* koja će kasnije usmjeravati to ime prema komponentama unutar kojih se to ime provjerava. Tako će zapravo teći slijed jedine informacije koju korisnik šalje ovoj web aplikaciji. Ona će se slati od početne roditeljske komponente prema njihovim podređenim komponentama gdje će napokon biti korištena. Uz preuzimanje i spremanje korisnikovog podatka, komponenta *Header* radi još jednu posrednu funkciju, a to je da dohvaća podatke sa Sudskog registra kako bi oni bili spremni za korištenje kada glavni podatak stigne do komponente gdje se provjerava dostupnost imena u sudskom registru. Još jedan od glavnih razloga zašto se ta funkcija odvija baš u početnoj komponenti je zato što se komponenta *Header* ne mora ispočetka učitavati i konstantno može sadržavati podatke koji se dohvaćaju sa Sudskog registra te će tako povećati odnosno osigurati brži rad aplikacije.

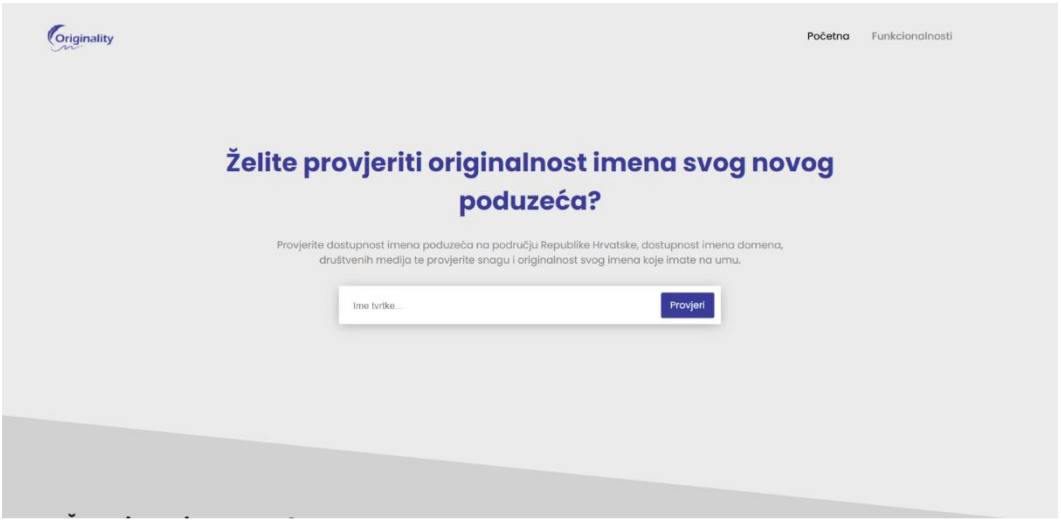
```
state = {
companyName: '',
courtRegisterData: ''
```

```
}

// Dohvaćanje podataka sa Sudskog registra
async componentDidMount() {
const response = await axios.get('https://sudreg-
api.pravosudje.hr/javni/tvrtka?offset=0&limit=161571',
{headers: {
'Ocp-Apim-Subscription-Key': '0dd386761a344b349ee2f6c14b562e4b'
}});
let courtRegisterData = await response.data;
this.setState({ courtRegisterData: courtRegisterData });
}

// Dohvaćanje upisanog imena
getCompanyNameHandler = (event) => {
const companyName = event.target.value;
this.setState({ companyName: companyName });
}
```

U cjeloživotnoj funkciji `componentDidMount` dohvaćaju se podaci sa Sudskog registra i spremaju se unutar *state* objekta pod nazivom *courtRegisterData*, dok se upisano ime korisnika dohvaća unutar funkcije `getCompanyNameHandler` i sprema se pod nazivom *companyName*.

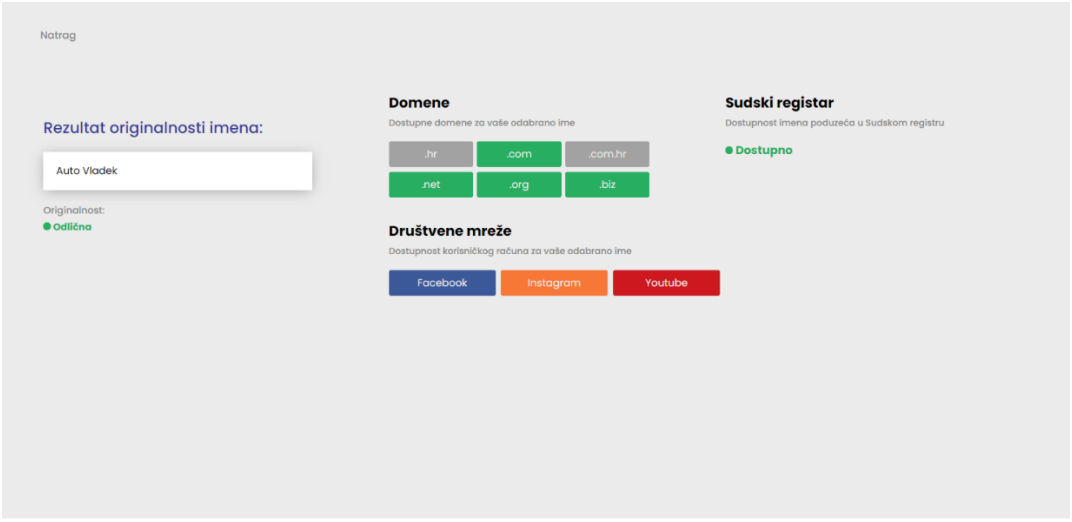
Unutar *Header* komponente kreiran je *router* koji usmjerava korisnika na stranicu rezultati nakon što upiše ime i pritisne gumb za provjeru originalnosti. Tu se definira komponenta *Results* kojoj se prosljeđuje upisano ime i podaci sa Sudskog registra.

```
<Route path="/result">
<Results companyName={this.state.companyName}
clearCompanyName={this.clearCompanyNameHandler}
courtRegisterData={this.state.courtRegisterData} />
</Route>
```



Slika 4. Prikaz sučelja *Header* komponente (vlastita izrada)

Druga prema logičkom slijedu je komponenta *Results*. To je komponenta koja prikazuje korisniku željeni rezultat. Ona prikazuje dostupnost domena, Sudskog registra, društvenih mreža, prikazuje upisano ime korisnika i na kraju i najvažnije od svega, kakva je originalnost imena koje je korisnik upisao. To je komponenta koja preuzima podatke o imenu od komponente *Header* i šalje ga svim komponentama koje su definirane unutar nje, a to su sve komponente koje su prikazane unutar strukture. Svaka od tih komponenata zapravo vraća neki *HTML* element kao rezultat, dok unutar komponente *Results* zapravo služi još za prikupljanje podataka o imenu.



Slika 5. Prikaz sučelja komponente *Results* (vlastita izrada)

Svaki od elemenata na ovoj slici sučelja je zapravo zasebna komponenta kao što se to može vidjeti u programskom kodu:

```
<div className="section-show-results">
```

```
<div className="row">
<div className="col span-1-of-2">
<h3>Domene</h3>
<p  className="results-p">Dostupne  domene  za  vaše  odabrano
ime</p>
<div className="row box">
<Hr name={props.companyName}/>
<Com name={props.companyName}/>
<ComHr name={props.companyName}/>
</div>
<div className="row box">
<Net name={props.companyName}/>
<Org name={props.companyName}/>
<Biz name={props.companyName}/>
</div>
</div>
<div className="col span-1-of-2">
<h3>Sudski registar</h3>
<p  className="results-p">Dostupnost  imena  poduzeća  u  Sudskom
registru</p>
<CourtRegister                          name={props.companyName}
data={props.courtRegisterData} />
</div>
</div>
```

Ovaj dio koda prikazuje gornju sekciju rezultata provjere dostupnosti domena i Sudskog registra. U ovom prikazu zapravo je definiran dizajn izgleda komponente *Results* i pozvane su sve komponente koje vrše provjeru: u ovom slučaju prikaza koda to su komponente *Hr*, *Com*, *ComHr*, *Biz*, *Net*, *Org* i *CourtRegister*. Svaka od tih komponenata pod nazivom *name* prima svojstvo *companyName* unutar kojeg se nalazi prethodno spremljeno ime novog poduzeća koje je upisao korisnik i koje želi provjeriti.

Nakon što su podaci proslijeđeni svim komponentama, može se početi vršiti provjera unutar svake komponente. Prvotno se kreiraju sve komponente koje šalju zahtjeve prema *API*-ju i koje vrše provjeru, dok se komponenta za provjeru originalnosti razvija posljednja, nakon što su svi podaci spremni. Svaka od komponenata ima svoj zasebni *state* objekt unutar kojeg se spremaju samo dvije vrste podataka. Prvi je podatak o dostupnosti imena

unutar te komponente, a drugi podatak je taj koji određuje vrši li se još uvijek provjera ili ne i prema tome komponenta zna hoće li vratiti konačan rezultat, odnosno *HTML* element ili će biti prikazan znak za učitavanje odnosno dohvaćanje podataka koji će dati korisniku do znanja da provjera tog segmenta još nije završena.

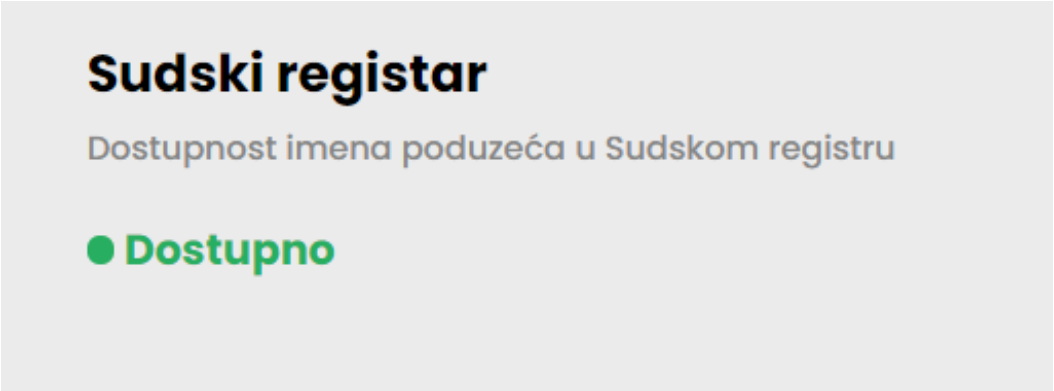
```
state = {
isRegistered: '',
loading: true
}
```

Komponenta *CourtRegister* tako unutar svog okruženja prima ime koje je upisao korisnik i vrši provjeru unutar Sudskog registra.

```
componentDidMount() {
let name = this.props.name;
let companyName = name.toUpperCase();
let courtRegisterData = this.props.data;

for(let i = 0; i < courtRegisterData.length; i++) {
if(companyName === courtRegisterData[i].naznaka_imena) {
this.setState({ isRegistered: 'Nedostupno', loading: false });
break;
}
else {
this.setState({ isRegistered: 'Dostupno', loading: false });
}
}
}

render() {
let isRegistered = this.state.isRegistered;
let courtRegister;

if(isRegistered === 'Dostupno' ) {
courtRegister = <p className="court-register-availability
available"><span className="dot-green"></span> Dostupno</p>;
this.props.handleCourtRegister();
}
```

```
else {
courtRegister = <p className="court-register-availability
taken"><span className="dot-red"></span> Nedostupno</p>;
this.props.handleCourtRegisterFalse();
}

if(this.state.loading) {
return <div className="loader"><Loader type="ThreeDots"
color="#3B3B98" height={50} width={50} /></div>
}
else {
return courtRegister;
}
}
```

Unutar funkcije `componentDidMount` prvo je potrebno postaviti dohvaćeno ime u velika tiskana slova jer su tako spremljeni podaci unutar *API*-ja koji služi za provjeru. Sljedeća stvar je da se napravi for petlja koja prolazi kroz svako ime koje se nalazi u registru i radi usporedbu s upisanim korisnikovim imenom. Ako pronađe ime zaustavlja se petlja i označava se postojanje imena, a u suprotnom slučaju postavlja se da je Sudski registar dostupan i unutar funkcije `render` vraća se paragraf p koji zapravo ispisuje tekst Dostupno / Nedostupno ovisno o povratnoj informaciji petlje.



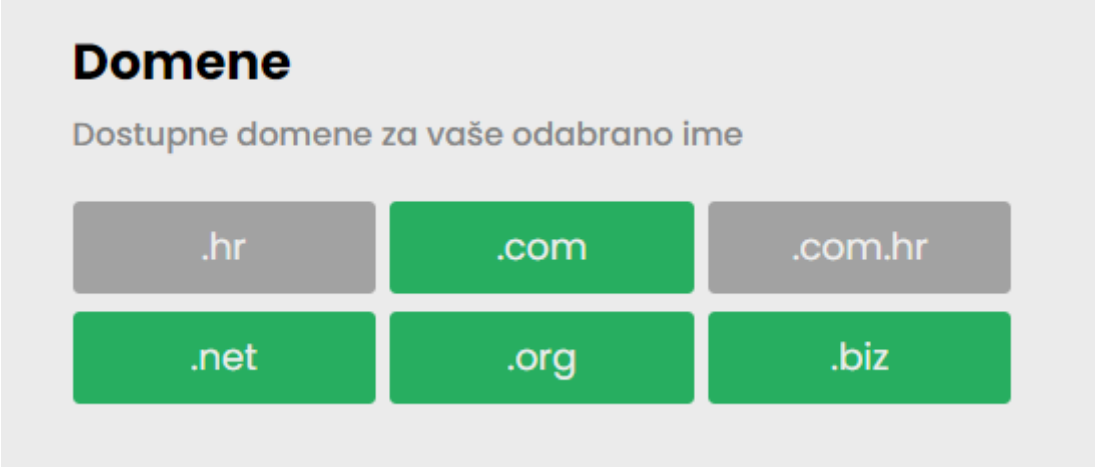
Slika 6. Prikaz dostupnosti imena u Sudskom registru (vlastita izrada)

Provjera dostupnosti šest navedenih domena vrši se zapravo u šest komponenata. U svaku komponentu proslijedi se ime i svaka komponenta zapravo šalje zahtjev na isti *API*, samo s drugim nazivom domene za koju se vrši provjera.

```
componentDidMount() {
```

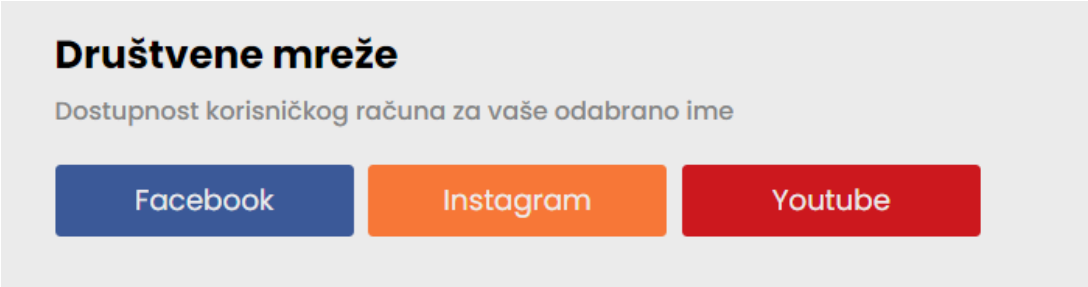
```
let name = this.props.name;
let companyName = name.toLowerCase().replace(/\s/g, '-');
let isRegistered;

axios.get('http://api.whoapi.com/?apikey=8918de443fc42a8c859df7
4469b74a0b&r=whois&domain='
+ companyName + '.com&ip=').then( res => {
isRegistered = res.data.registered;
this.setState({ registered: isRegistered, loading: false });
});
}

render() {
let com;

if(this.state.registered === false) {
com = <div className="col span-1-of-3 domain-box">.com</div>;
this.props.handleCom();
}
else {
com    =    <div    className="col    span-1-of-3    domain-box-
taken">.com</div>;
this.props.handleComFalse();
}

if(this.state.loading) {
return   <div   className="col   span-1-of-3   loader"><Loader
type="ThreeDots"    color="#3B3B98"    height={50}    width={50}
/></div>
}
else {
return com;
}
}
```

Komponente za provjeru domena šalju *axios* zahtjeve na određeni *API* s određenim podacima o imenu domene i vraćaju *JSON* objekt unutar kojeg je moguće vidjeti je li domena dostupna ili ne (*res.data.registered*). Rezultat *JSON* svojstva može biti *true* ako je domena slobodna ili *false* ako je domena zauzeta i taj se rezultat sprema unutar *state* objekta kako bi se unutar funkcije `render` moglo odrediti kakav će se *HTML* element vratiti kao rezultat. Provjera koja se radi odlučuje o tome hoće li element biti zelene boje ili sive.



Slika 7. Prikaz dostupnosti domena (vlastita izrada)

Za provjeru dostupnosti društvenih mreža korištene su pak komponente koje provjeravaju dostupnost, svaka zasebno za pojedinu društvenu mrežu. Ono što je specifično je da je u ovim komponentama korišten *API* gdje se gledalo vraća li on bilo kakav odgovor ili grešku i na temelju oznake greške bilo je lako moguće zaključiti radi li se o zauzetoj društvenoj mreži ili je to korisničko ime slobodno. Princip vraćanja rezultata u obliku *HTML* elementa je isti kao i kod domena.



Slika 8. Prikaz dostupnosti korisničkog imena na društvenim mrežama (vlastita izrada)

Nakon što su izvršene sve provjere dostupnosti imena po pojedinim komponentama, potrebno je korisniku dati do znanja kakav je ukupni rezultat svih provjera, odnosno kakva je originalnost imena koje je odabrao što je i glavna funkcionalnost ove web aplikacije. Kako bi se dobio odgovor na to, kreirana je prema logičkom slijedu posljednja potrebna komponenta.

To je komponenta *Originality*. Uz pomoć *React Redux*-a dobili smo na jednom mjestu sve rezultate dostupnosti svih komponenata.

```
const initialState = {
courtRegister: false,
bizDomain: false,
comDomain: false,
comHrDomain: false,
hrDomain: false,
netDomain: false,
orgDomain: false,
instagram: false,
facebook: false,
youtube: false
}
```

Ovo je početni *state* objekt unutar *reducer*-a. Unutar njega navedene su sve komponente koje se provjeravaju za dostupnost imena i početne vrijednosti svih stavljene su početno na *false*, što bi označavalo da nisu dostupne. Ono što rade funkcije unutar *reducer*-a je to da mijenjaju svako svojstvo u *true* / *false*, ovisno o tome kakav je rezultat komponente.

```
const reducer = (state = initialState, action) => {
switch(action.type) {
case 'COURT-REGISTER-TRUE':
return Object.assign({}, state, {
courtRegister: true
});

case 'COURT-REGISTER-FALSE':
return Object.assign({}, state, {
courtRegister: false
});

case 'BIZ_TRUE':
return Object.assign({}, state, {
bizDomain: true
```

```
});

case 'BIZ_FALSE':
return Object.assign({}, state, {
bizDomain: false
});

case 'COM_TRUE':
return Object.assign({}, state, {
comDomain: true
});

case 'COM_FALSE':
return Object.assign({}, state, {
comDomain: false
});

case 'COMHR_TRUE':
return Object.assign({}, state, {
comHrDomain: true
});

case 'COMHR_FALSE':
return Object.assign({}, state, {
comHrDomain: false
});

case 'HR_TRUE':
return Object.assign({}, state, {
hrDomain: true
});

case 'HR_FALSE':
```

```
return Object.assign({}, state, {
hrDomain: false
});

case 'NET_TRUE':
return Object.assign({}, state, {
netDomain: true
});

case 'NET_FALSE':
return Object.assign({}, state, {
netDomain: false
});

case 'ORG_TRUE':
return Object.assign({}, state, {
netDomain: true
});

case 'ORG_FALSE':
return Object.assign({}, state, {
netDomain: false
});
```

Ovo je prikaz samo djela *reducer*-a koji sadržava funkcije koje rade promjene za dostupnost svih domena koje se provjeravaju unutar aplikacije.

Komponenta *Originality* uzima sva ta svojstva *reducer*-a i smješta ih u jednu vanjsku funkciju `mapStateToProps`.

```
render() {
let availability;
const values = _.values(this.props);
const valuesTrue = values.filter(value => value === true);
const valuesFalse = values.filter(value => value === false);
```

```
if(valuesTrue.length > valuesFalse.length) {
availability = <p className="originality-result"><span
className="dot-green"></span> Odlična</p>;
}
else if(valuesTrue.length === valuesFalse.length) {
availability = <p className="originality-result-yellow"><span
className="dot-yellow"></span> Srednje dobro</p>;
}
else {
availability = <p className="originality-result-red"><span
className="dot-red"></span> Loše</p>;
}
return availability;
}
const mapStateToProps = state => {
return {
courtReg: state.courtRegister,
bizDomain: state.bizDomain,
comDomain: state.comDomain,
comHrDomain: state.comHrDomain,
hrDomain: state.hrDomain,
netDomain: state.netDomain,
orgDomain: state.orgDomain,
instagram: state.instagram,
facebook: state.facebook,
youtube: state.youtube
}
}
```

Kada su svi rezultati provjera spremni unutar ove funkcije, tada funkcija *render* započinje svoj rad i uzima sva ova svojstva te radi filtriranje nad njima. U zasebne varijable spremaju se svi filtrirani rezultati koji su *true* i svi koji imaju vrijednost *false*. Kada su vrijednosti podijeljene vrši se glavna provjera cijele aplikacije. Provjera radi na način da provjerava postoji li više dostupnih komponenata i tada vraća odličnu originalnost, ako pak su vrijednosti izjednačene tada originalnost bude srednje dobra, a ako je više negativnih rezultata, tj. *false* tada korisnik dobiva poruku da je originalnost njegovog imena loša.

6. Zaključak

U ovome radu opisan je *React.js* programski okvir, koji je samo jedan od mnogih popularnih *Javascript* modernih programskih okvira. Istaknute su prednosti, lakoća te brzina kreiranja web aplikacija. Opisane su osnove programiranja u *React*-u kao što su pisanje nove *ES6* sintakse i korištenje pomoćnih alata za pisanje *HTML* elemenata unutar *Javascript*-a. Opisano je kreiranje novih komponenata, spremanje podataka u *state* objekt koji predstavlja spremnik unutar komponente. Time je prikazano kako se zapravo *React* svodi na kreiranje komponenata koje se implementiraju jedna unutar drugih i koje imaju mogućnosti prenošenja različitih tipova podataka.

Podaci su vrlo važan segment i bez njih niti jedna web aplikacija ne bi imala smisla, pa je tako vrlo važno da se ti podaci mogu prenositi brzo i jednostavno kroz sve komponente. Za to služe svojstva koja se pozivaju kod definiranja komponenata i koja služe da se u njih spreme podaci. Podaci se najčešće prosljeđuju s komponente roditelja prema podređenim komponentama, no nije isključena opcije obrnutog slanja podataka što je i prikazanu u praktičnom djelu ovog rada. Također, za lakše dohvaćanje podataka s više komponenata na jednom izdvojenom mjestu, korišten je *Redux* koji uvelike pojednostavljuje upravljanje podacima pomoću svog glavnog *state* objekta i *reducer*-a.

Na kraju ovog rada prikazan je *React* u praksi te je tako izgrađena web aplikacija koja služi za provjeru dostupnosti imena novog poduzeća unutar Republike Hrvatske. Aplikacija ima funkciju provjere dostupnosti imena domena, društvenih mreža i unutar Sudskog registra. Web aplikacija je primjer kako zapravo koristiti sav teorijski sadržaj u praksi i primijeniti u izradi jednostavne web aplikacije koja je brza i koja kontrolira slanje podataka unutar komponente.

Zaključak ovog rada je da je *React.js* vrlo moderan i često korišten programski okvir koji omogućuje kreiranje velikih, ali brzih web aplikacija i koriste ga najveće kompanije za razvoj svojih produkata.

Popis literature

[1] „Internet Growth Statistics 1995 to 2019 – the Global Village Online“, Internetworldstats.com, 2020. [Na internetu]. Dostupno: https://www.internetworldstats.com/emarketing.htm [Pristupljeno: 9.7.2020.]

[2] Artemij Fedosejev „React.js Essentials“. Birmingham, Packt Publishing. 2015.

[3] „HTML: Hypertext Markup Language“, https://developer.mozilla.org/, 2020. [Na internetu]. Dostupno: https://developer.mozilla.org/en-US/docs/Web/HTML [Pristupljeno: 11.7.2020.]

[4] Bruce Lawson, Remy Sharp „Introducing HTML 5“. Berkley, New Riders. 2011.

[5] „CSS: Cascading Style Sheets“, https://developer.mozilla.org/, 2020. [Na internetu]. Dostupno: https://developer.mozilla.org/en-US/docs/Web/CSS [Pristupljeno: 11.7.2020.]

[6] Jon Duckett „HTML & CSS, Design and Build Websites“. Indianapolis, Crosspoint Boulevard. 2011.

[7] „CSS preprocessor“, https://developer.mozilla.org/, 2020. [Na internetu]. Dostupno: https://developer.mozilla.org/en-US/docs/Glossary/CSS_preprocessor [Pristupljeno: 11.7.2020.]

[8] Linda Dailey Paulson, „Building Rich Web Applications with Ajax“, semanticscholar.org, 2005. [Na internetu]. Dostupno: https://www.semanticscholar.org/paper/Building-Rich-Web-Applications-with-Ajax-Paulson/a4777cea6758969602098785d10a599a77fd0d30 [Pristupljeno: 12.7.2020.]

[9] Marin Kaluža, Krešimir Troskot, Bernard Vukelić, „Comparison of Front-End Frameworks for Web Applications Development“, hrcak.srce.hr, 2018. [Na internetu]. Dostupno: https://hrcak.srce.hr/199922 [Pristupljeno: 12.7.2020.]

[10] Stephen Blumenthal "Javascript For Beginners - Learn Programming with ease". CreateSpace Independent Publishing Platform. 2017.

[11] Dr. Axel Rauschmayer, „JavaScript for impatient programmers (ES2020 edition)“. Dr. Axel Rauschmayer, 2020.

[12] Gilles Ruppert, Jack Moore, "Javascript and JQuery - Interactive Front-End development", pdfdrive.com, 2014. [Na internetu]. Dostupno: https://www.pdfdrive.com/javascript-jquery-interactive-front-end-development-e150028047.html [Pristupljeno: 13.7.2020.]

[13] „JavaScript Notes for Professionals“, https://books.goalkicker.com/, 2019. [Na internetu]. Dostupno: https://books.goalkicker.com/JavaScriptBook/ [Pristupljeno: 10.8.2020.]

[14] Vipul A. M., Prathamesh Sonpatki „ReactJS by Example - Building Modern Web Applications with React“. Birmingham, Publishing Place. 2016.

[15] „Add React to a Website“, https://reactjs.org/ [Na internetu]. Dostupno: https://reactjs.org/docs/add-react-to-a-website.html#add-react-in-one-minute [Pristupljeno: 10.8.2020.]

[16] Robin Wieruch „The Road to React: Your journey to master plain yet pragmatic React.js“. Robin Wieruch, 2017.

[17] „Draft: JSX Specification“, https://facebook.github.io/, 2014. [Na internetu]. Dostupno: https://facebook.github.io/jsx/ [Pristupljeno: 11.8.2020.]

[18] „React Router: Declarative Routing for React.js“, https://reactrouter.com/ [Na internetu]. Dostupno: https://reactrouter.com/web/guides/quick-start [Pristupljeno: 15.8.2020.]

[19] „Using Axios with React“, https://www.digitalocean.com/, 2018. [Na internetu]. Dostupno: https://www.digitalocean.com/community/tutorials/react-axios-react [Pristupljeno: 15.8.2020.]

[20] „Getting started with Redux“, https://redux.js.org/ [Na internetu]. Dostupno: https://redux.js.org/introduction/getting-started [Pristupljeno: 16.8.2020.]

Popis slika

Popis tablica