Matematički fakultet

Univerzitet u Beogradu

MASTER RAD

Elektronske lekcije o radnom okviru ReactJS

Student: Milina Smiljanić 1065/2019

Mentor: prof. dr Miroslav Marić

Beograd, 2021.

Sadržaj

Uvod

Napredak tehnologije i sve veća dostupnost interneta doprineli su razvoju i ekspanziji veb aplikacija. Ključni razlog za njihovu popularnost je sposobnost ažuriranja i održavanja bez potrebe za distribucijom i instalacijom softvera na potencijalnim klijentskim računarima. Sve nove funkcije se sprovode na serveru i automatski se dostavljaju korisnicima.

Osim što značajno pojednostavljuju, ubrzavaju i olakšavaju svakodnevne poslovne procese i aktivnosti, danas su veb aplikacije postale pomoćno sredstvo i u obrazovanju. Budući da su veb aplikacije postale nezaobilazni deo svakodnevnog života, njihova upotreba i njihov broj su u stalnom porastu, a pojavljuju se i nove veb tehnologije za izradu i poboljšanje istih. Dostupan je veliki broj platformi za učenje veb programiranja, ali su one uglavnom napisane na engleskom jeziku. To ponekad predstavlja prepreku za početnike u ovoj oblasti. Rešenje ovog problema se nalazi u materijalima dostupnim na srpskom jeziku, a deo su elektronske platforme *eŠkola veba*. Na platformi *eŠkola veba* se nalaze materijali za učenje trenutno napopularnijih veb tehnologija i radnih okvira. Sve lekcije su besplatne i javno dostupne na adresi http://edusoft.matf.bg.ac.rs/eskola_veba/#/home.

Softverski razvoj veb aplikacija, odnosno razvoj korisničkih interfejsa za njih, uglavnom je vezan za programski jezik JavaScript. Budući da je izrada složenijih aplikacija u programskom jeziku JavaScript prilično komplikovana, sve više se izrađuju radni okviri (eng. *framework*) koji značajno olakšavaju kreiranje istih, uz znatno manje linija koda. Radni okviri pružaju sintaksu koja je jednostavna kao i skup često korišćenih funkcionalnosti koje služe za rešavanje problema koji nastaju prilikom razvoja veb aplikacija. Osim što olakšavaju kreiranje koda aplikacije, gde je akcenat na samoj arhitekturi jezika, savremeni radni okviri omogućavaju i prilagodljiv veb dizajn. To je veoma značajno zbog povećanog broja uređaja na kojima je veličina ekrana mala.

Glavna uloga JavaScript radnih okvira je da olakša razvoj veb aplikacija koje pri tome treba da budu jasne i intuitivne za korišćenje. ReactJS je radni okvir koji se koristi za kreiranje korisničkih interfejsa za jednostranične veb aplikacije (eng. *single page application*, skraćeno SPA). Zahvaljujući svojim karakteristikama, ReactJS omogućava razvoj aplikacija koje obezbeđuju promenu podataka bez potrebe za ponovnim učitavanjem stranice, čineći samu aplikaciju brzom, jednostavnom i skalabilnom. Radni okvir ReactJS sadrži obrasce za kreiranje veb strana i veb aplikacija zasnovanih na jeziku za označavanje hiperteksta HTML, na jeziku za formatiranje CSS i na skript jezicima JavaScript i Sass. Elektronske lekcije o ovom radnom okviru se nalaze u okviru kursa ReactJS na platformi *eŠkola* veba. Napisane na srpskom jeziku i imaju metodički pristup, tako da su pristupačne za početnike u ovoj oblasti.

U ovom radu će biti predstavljena struktura elektronske platforme na kojoj se nalaze lekcije. Na početku će biti prikazani ključni delovi arhitekture SPA, kao i prednosti i nedostaci ovakvog pristupa za izradu veb aplikacija. Nakon toga će biti opisan format JSX[1] i zašto je on

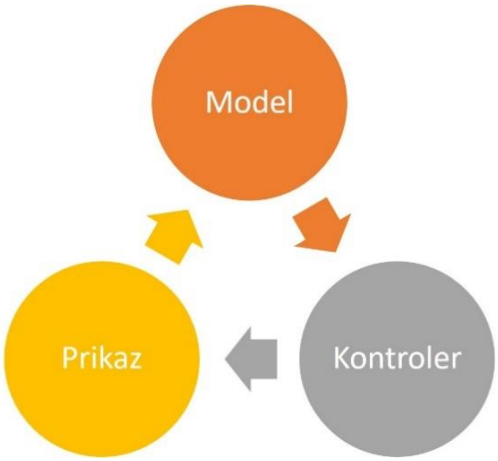
[1] Sintaksno proširenje programskog jezika JavaScript. Omogućava da se u radnom okviru ReactJS koriste HTML oznake.

značajan prilikom definisanja vizuelnih elemenata. Jedan od osnovnih koncepata radnog okvira ReactJS je struktuiranje nezavisnih komponenti prilikom kreiranja dinamičkog korisničkog okruženja, što će detaljnije biti objašnjeno u narednim poglavljima. Biće predstavljeni načini za kreiranje ReactJS okruženja, kao i glavne mogućnosti koje pruža ovaj radni okvir.

1 Arhitekturalni šablon MVC

Arhitekturalni šablon MVC (eng. *Model-View-Controller*) opisuje način kreiranja aplikacija pri čemu se mora voditi računa o kriterijumima koje je potrebno zadovoljiti. Suština MVC šablona je razdvajanje prezentacijskog dela aplikacije od koda koji se bavi podacima. Logika aplikacije je strukturirana tako da pruža mogućnost jednostavnog unapređivanja koda. Arhitektura MVC razdvaja aplikaciju na tri grupe komponenti: **model**, **prikaz** (eng. ***view***) i **kontroler** (eng. ***controller***). Pomoću ovog obrasca se zahtevi korisnika preusmeravaju na kontroler koji je zadužen za komunikaciju sa modelom. Model sadrži podatke kao što su informacije o objektima iz baze podataka. Sadrži i poslovnu logiku definišući operacije koje se mogu primeniti nad tim podacima. Model se ne može direktno pozvati, već se kontroler obraća modelu kada od njega zahteva potrebne podatke. Zatim model obrađuje tražene podatke i vraća ih natrag kontroleru. Kontroler prosleđuje podatke komponenti zaduženoj za prikaz, koji ih dalje prikazuje korisniku. Komponenta prikaz samo prikazuje informacije, dok kontroler obrađuje korisnički unos i bira izlazne podatke. Prikaz MVC šablona se nalazi na slici 1.

Radni okvir ReactJS se bavi slojem prikaza, pri čemu je bitno da svi vizuelni elementi budu ažurirani, a prilagodljivost modela i kontrolera omogućava programeru odabir poznatih alata i tehnologija. Sa ovakvom arhitekturom ReactJS postaje koristan ne samo za pravljenje novih veb aplikacija, već i za unapređivanje koda postojećih aplikacija.



Slika 1 – MVC arhitekturalni šablon

2 Veb aplikacije u jednoj strani

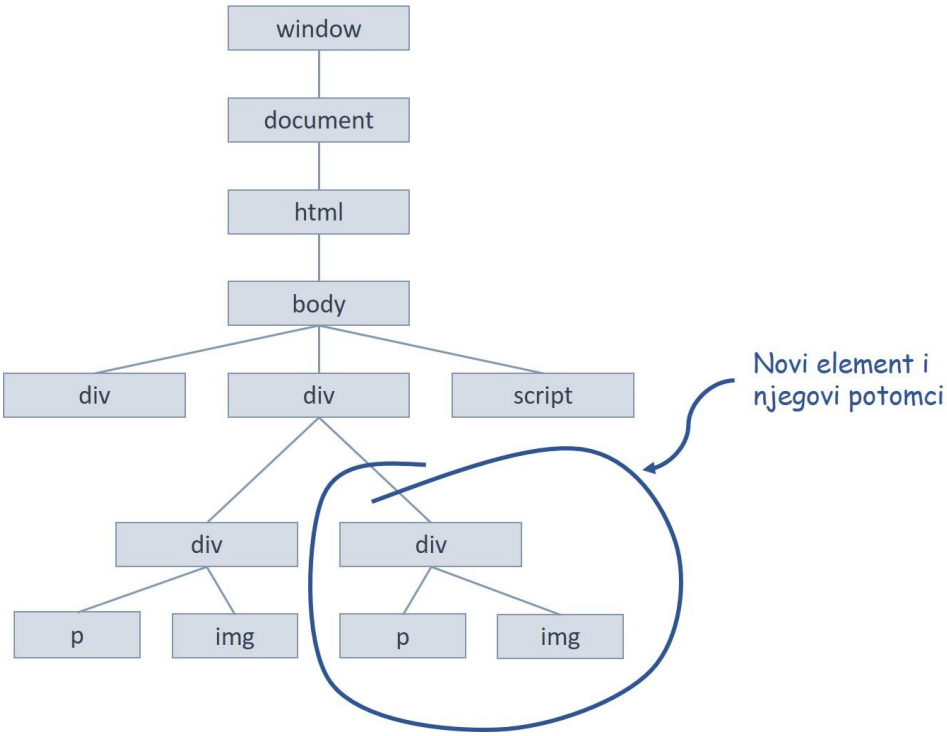
Tradicionalni pristup razvoju veb aplikacija podrazumeva da se nakon svake akcije korisnika šalje zahtev serveru, nakon čega se generiše nova stranica. Ovakav pristup ima za posledicu celokupno ponovno učitavanje stranice, što je proces koji korisniku oduzima vreme. Danas je model aplikacije koji zahteva kretanje između pojedinačnih strana zastareo. Nasuprot tome, moderne veb aplikacije navigaciju vrše u jednoj strani, pri čemu se učitavanje stranice vrši samo prilikom inicijalnog pokretanja aplikacije. Dalja promena okruženja su parcijalna, odgovarajući resursi se učitavaju dinamički i obično su posledica neke od akcija korisnika. Većinom je poslovna logika prebačena na klijentsku stranu, što omogućava brze promene stanja, dok server služi za trajno čuvanje podataka i dostavljanje podataka klijentu. Popularne aplikacije kao što su *Gmail, Facebook, Twitter,* itd. su sve jednostranične aplikacije. U tim aplikacijama sadržaj se dinamički prikazuje, bez ponovnog učitavanja stranice ili prelaska na novu stranicu.

Kod izrade veb aplikacija u jednoj strani, potrebno je vreme za usklađivanje podataka sa korisničkim okruženjem. Npr. da li ukloniti polje za pretragu nakon što korisnik učita neki sadržaj, ili koji vizuelni elementi će ostati na stranici, a koji će se ukloniti nakon što korisnik izvrši neku akciju. Pored toga, izazov u radu sa modernim veb aplikacijama predstavlja i rad sa objektnim modelom dokumenta (eng. *Document Object Model* - DOM)[2] koji je veoma spor.

Upravljanje DOM-om predstavlja način reagovanja na korisnikove akcije i prikazivanja novog sadržaja. Pozivanje elemenata, dodavanje potomaka elemenata (slika 2) i obavljanje drugih operacija nad DOM-om su stvari koje se najsporije obavljaju u pregledaču. Uprkos ovim nedostacima, jednostranične aplikacije su veoma popularne i čine budućnost izrade veb aplikacija. Zahvaljujući pojavi mnogih nezavisnih radnih okvira i biblioteka za skript jezik JavaScript, razvoj veb aplikacija u jednoj strani danas nije komplikovan.

Radni okviri koji koriste SPA principe su ReactJS, AngularJS, ExtJS, VueJS i drugi.

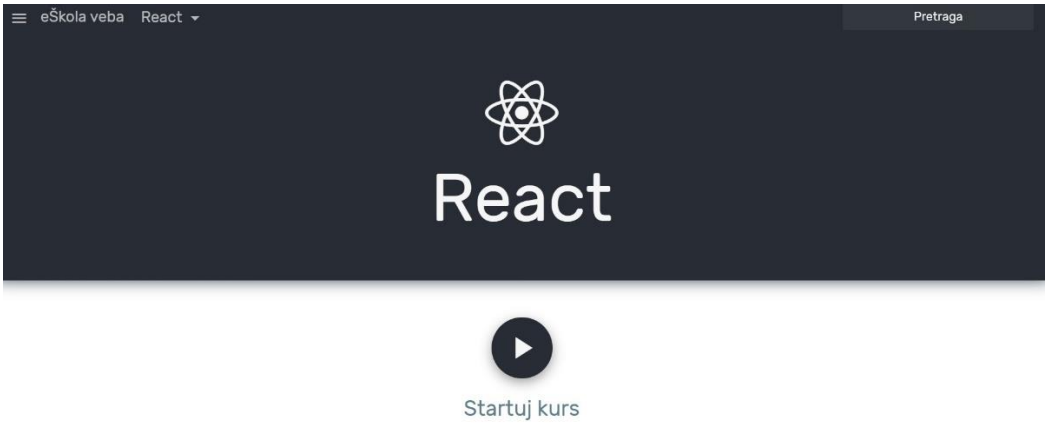
[2] Strukturna prezentacija HTML ili XML dokumenata čiji su elementi organizovani u stablo u kome svaki čvor predstavlja jedan element



Slika 2 – Dodavanje potomaka

3 Elektronske lekcije o radnom okviru ReactJS

Postoji veliki broj interaktivnih sadržaja dostupnih na internetu koji mogu da pomognu u savladavanju ovog radnog okvira. Lekcije o radnom okviru ReactJS, kreirane za potrebe ovog rada, dostupne su u okviru elektronskog kursa i deo su projekta *eŠkola veba* koji se razvija na Matematičkom fakultetu u Beogradu. Kurs je koncipiran tako da sadrži neophodnu teoriju, konstruktivne primere za praktično učenje, ali i mogućnost da korisnik na kraju svake lekcije stečeno znanje testira kroz pažljivo odabrane primere. Kurs je dostupan na adresi http://edusoft.matf.bg.ac.rs/eskola_veba/#/course-details/react. Klikom na dugme "Startuj kurs" pokreće se prva lekcija kursa (slika 3).



Slika 3 - Elektronske lekcije o radnom okviru ReactJS

Svaka lekcija se sastoji iz tri segmenta: prvi segment se odnosi na teorijski sadržaj, drugi segment se odnosi na interaktivni primer koji se može direktno pokrenuti (testirati) i izmeniti, a treći segment je prikaz navedenog primera u pregledaču. Ovakav prikaz lekcija pruža interaktivnost u učenju i ima za cilj da korisnicima omogući lakše usvajanje znanja. Na slici 4 je dat prikaz jedne lekcije, pri čemu je crvenom bojom označen prvi segment, zelenom bojom je označen drugi segment, dok je ljubičastom bojom označen treći segment lekcije.

Slika 4 - Prikaz jedne lekcije u radnom okviru ReactJS

4 Istorija radnog okruženja ReactJS

Još 2011. godine programeri u kompaniji *Facebook* počeli su da se suočavaju sa problemima oko održavanja koda. Kako je aplikacija koju su koristili dobijala sve veći broj funkcija, vremenom je njom bilo teško upravljati jer se suočila sa brojnim ažuriranjima. Njihov kôd je zahtevao nadogradnju kako bi postao efikasniji. Tim povodom je Jordan Walke, softverski inženjer u kompaniji *Facebook* razvio prototip koji je proces učinio efikasnijim i nazvao ga *FaxJs*. Na njegov razvoj je uticao XHP[3]. Međutim, dinamičke veb aplikacije zahtevaju mnogo povratnih putanja do servera, a XHP ne rešava ovaj problem. To znači da se veb aplikacija ponovo iscrtava kada se promeni stanje, što dovodi do toga da korisnik izgubi sve informacije prethodno skladištene u DOM-u. Ovaj proces nije dobar, kako za performanse tako i za korisničko iskustvo. Jordan Walke je radio na prototipu kako bi proces učinio efikasnijim. Tako je nastala biblioteka skript jezika JavaScript za izgradnju korisničkih interfejsa i nazvana je ReactJS. Vremenom je ova biblioteka poprimila sve više korisnika, a u maju 2013. Jordan Walke je svetu predstavio ReactJS u okviru *JS ConfUS* konferencije i sam kôd učinio javno dostupnim.

U službenoj dokumentaciji je navedeno da je ReactJS biblioteka programskog jezika JavaScript i da služi za razvoj korisničkih interfejsa. Vremenom je, zbog velike popularnosti, zajednica otvorenog koda razvila mnoštvo paketa koje proširuju biblioteku kako bi se omogućile funkcionalnosti koje su nedostajale. Danas je fleksibilnost biblioteke čini kompatibilnom s nizom alata čijom se integracijom postiže potpuna funkcionalnost radnog okvira.

[3] Sintaksno proširenje programskog jezika PHP. Omogućava upotrebu HTML sintakse u programskom jeziku PHP.

5 EcmaScript 6

Budući da je JavaScript najčešće korišćen programski jezik za izradu klijentske strane veb aplikacija, došlo je do razvoja njegovog standarda, odnosno specifikacije koja je nazvana EcmaScript ili skraćeno ES. Vremenom je EcmaScript prošao kroz mnoge izmene i poboljšanja, ali je svaka izmena napravljena tako da prethodna verzija bude kompatibilna sa novom verzijom. Šesti EcmaScript standard, odnosno ES6 je značajno poboljšanje skript jezika JavaScript. Radni okvir ReactJS većinom podrazumeva upotrebu ES6 standarda, zbog boljih performansi.

5.1 Kratak pregled ES6 svojstava korišćenih u radu

5.1.1 Klase

U jeziku JavaScript objekti se nasleđuju upotrebom sintakse bazirane na *prototype* nasleđivanju. Izlaskom ES6 standarda uvodi se reč *class* u skript jezik JavaScript. Ključna reč *class* ukazuje na to da je u pitanju specijalna vrsta funkcije, gde se umesto ključne reči *function* koristi *class*, a svojstva se dodeljuju unutar metode *constructor*. Metoda *constructor* se poziva kada se instancira novi objekat te klase upotrebom rezervisane reči *new*. Na ovaj način se ne uvodi novi model nasleđivanja, već se obezbeđuje jednostavna i intuitivna sintaksa za kreiranje objekta i proces postojećeg nasleđivanja. Ako se u narednim primerima uporede delovi koda, može se uočiti da je bolja čitljivost ES6 sintakse (kôd 2) u odnosu na ES5 sintaksu (kôd 1).

Kôd 1 – Sintaksa ES5

```
function Osoba(ime) {
    this.ime = ime;
    this.objavi = function () {
        console.log("Ovo je " + this.ime);
    }
}
```

Kôd 2 - Sintaksa ES6

```
class Osoba {
 constructor(ime) {
  this.ime = ime;
 }
 objavi() {
  console.log("Ovo je " + this.ime);
 }
}
```

5.1.2 Uvođenje promenljivih uz upotrebu ključnih reči let i const

Jedno od svojstava standarda ES6 je mogućnost uvođenja promenljivih uz upotrebu ključnih reči *let* i *const*. Ključna reč *const* se koristi prilikom uvođenja konstanti, odnosno promenljivih kojima se kasnije ne može ponovo dodeliti novi sadržaj. U slučaju kada varijabla može da menja vrednosti, prilikom deklaracije se navodi *let*. Na ovaj način se domen postojanja promenljive ograničava na blok koda u kome je definisana.

5.1.3 Arrow funkcije

ES6 standard uvodi novu sintaksu za pisanje funkcija koje su poznate pod nazivom *arrow* funkcije. Dobile su naziv po znaku => koji obezbeđuje kraći zapis. U drugim programskim jezicima su poznate pod nazivom lambda funkcije. Osim što obezbeđuju sažetost definicije funkcije, njihova prednost u odnosu na druge funkcije se ogleda u tome što ne definišu objekat *this*, tj. on zadržava referencu na globalni objekat, unutar kojeg je *arrow* funkcija definisana.

Kod ovih funkcija može da se izostavi ključna reč *return* i u pojedinim slučajevima mogu da se izostave zagrade. Male zagrade mogu da se izostave ukoliko postoji samo jedan ulazni parametar, a vitičaste zagrade mogu da se izostave ako u telu funkcije postoji samo jedan izraz. Kôd 3 prikazuje zapis funkcije uz pomoć standardne sintakse i kao *arrow* funkcije.

Kôd 3 – Zapis funkcije uz pomoć standardne sintakse i kao *arrow* funkcije

```
// zapis funkcije uz pomoc standardne sintakse
var pomnozi = function (x, y) {
  return x * y;
 };
// arrow funkcija
const pomnozi = (x, y) => x * y;
```

5.1.4 Kopiranje vrednosti svojstava upotebom metode *Object.assign*

Metoda *Object.assign* se koristi za kopiranje vrednosti svojstava jednog ili više izvornih objekata na ciljani objekat. Prvi parametar ove metode predstavlja referencu na ciljani objekat (eng. *targetObject*), dok ostali parametri predstavljaju izvorne objekte (eng. *sourceObject*). Metoda kao rezultat vraća ciljani objekat. Kôd 4 prikazuje upotrebu metode *Object.assign.*

Kôd 4 - Dodavanje vrednosti svojstava ciljanom objektu

```
const test = {
  ime: 'Jovan',
  godine: 30,
  rezultat: 'pozitivan'
 };
 var test2 =
   Object.assign( { }, test, {status: 'mirovanje', rezultat: 'negativan'} );
 console.log(test);
 // { ime: 'Jovan', godine: 30, rezultat: 'pozitivan' }

 console.log(test2);
 // { ime: 'Jovan', godine: 30, rezultat: 'negativan', status: 'mirovanje'}
```

U kodu 4 prvi argument funkcije *Object.assign()* predstavlja ciljani objekat koji će funkcija da vrati, drugi argument je objekat čija se svojstva kopiraju u ciljani objekat, dok treći argument funkcije predstavlja objekat čija svojstva ažuriraju prethodna. Rezultat je novi objekat koji odgovara izvornom objektu sa ažuriranim svojstvima.

Ova metoda je pogodna za rad sa konstantnim podacima, budući da se promenljivama koje su definisane kao konstante ne mogu menjati vrednosti.

Više informacija o ES6 standardima koji se koriste u radnom okviru ReactJS se može naći u [9].

6 Definisanje vizuelnih komponenti

Modularnost, kompaktnost i samostalnost su ključna svojstva dobro struktuiranih aplikacija. ReactJS ova svojstva primenjuje na kreiranje vizuelnih elemenata aplikacije. Osnovna ideja rada u ovom radnom okviru je da se vizuelni elementi podele na manje komponente, kako bi one bile ponovo iskoristive i kada se podaci koje prikazuju promene.

Definisanje vizuelnih elemenata u programskom jeziku JavaScript je prilično komplikovano i zahteva dosta poziva funkcije *createElement*. ReactJS omogućava da se vizuelne komponente definišu pomoću sintakse slične jeziku HTML, nazvane JSX (eng. JavaScript XML). Umesto pisanja koda za definisanje korisničkog okruženja, sintaksa JSX podrazumeva upotrebu odgovarajućih HTML oznaka.

Kôd 5 predstavlja primer upotrebe JSX sintakse u radnom okviru ReactJS.

Kôd 5 - Primer upotrebe *JSX* sintakse

```
ReactDOM.render(
  <div>
    <h1> Naslov </h1>
    <h3> Podnaslov </h3>
  </div>,
  destination
);
```

Kôd 6 prikazuje drugačiji način kreiranja ReactJS okruženja.

Kôd 6 - Kreiranje vizuelnih elemenata upotrebom funkcije *createElement*

```
ReactDOM.render(React.createElement
  "div",
  null,
  React.createElement(
    "h1",
     null,
    "Naslov"
  ),
  React.createElement(
    "h1",
     null,
    "Podnaslov"
  )
), destination);
```

Sintaksa JSX omogućava jednostavno definisanje vizuelnih komponenti, a ipak zadržava moć i fleksibilnost koju nudi skript jezik JavaScript.

Nije bitno da li je aplikacija napisana pomoću radnog okvira ReactJS ili neke druge biblioteke za JavaScript, krajni rezultat mora da bude kombinacija koda koja podrazumeva upotrebu jezika HTML, CSS i JavaScript. Zato je potrebno definisati način na koji će se JSX konvertovati u odgovarajući JavaScript kôd.

Više o upotrebi JSX sintakse u radnom okruženju ReactJS se može naći u [5].

7 Podešavanje radnog okruženja

Postoje dva načina za kreiranje radnog okruženja ReactJS. Prvi način podrazumeva upotrebu okruženja Node.js[4] i odgovarajućih alata, dok se drugi način ostvaruje uključivanjem okruženja iz distribucionog sistema servera CDN[5]. Oba načina imaju svoju primenu u praksi.

Prvi način je način na koji se moderni razvoj veba danas odvija. Pored prevođenja sintakse JSX u odgovarajući JavaScript kôd, ovaj pristup omogućava korišćenje modula, boljih alata za izgradnju i mnoštvo drugih funkcionalnosti koje olakšavaju izradu složenih veb aplikacija. Detaljnije o ovom načinu kreiranja radnog okruženja može se naći u [13].

Drugi način uključuje brzu i direktnu putanju koja se brine za prevođenje sintakse JSX u odgovarajući JavaScript kôd kada se strana učita. To se realizuje tako što se u zaglavlju HTML dokumenta navode linkovi pomoću kojih se pristupa udaljenim fajlovima. Linkovi koji se u ovom slučaju navode pokazuju na:

- kompajliranu osnovnu ReactJS bibiloteku,
- kompajliranu biblioteku koja je neophodna za rad sa DOM elementima,
- kompajliranu biblioteku *Babel*[6].

[4] Node.js je višeplatformsko radno okruženje otvorenog koda za izvršavanje programskog jezika JavaScript na serverskoj strani.

[5] CDN (eng. *Content Delivery Network*) je mreža za distribuciju sadržaja. Predstavlja veliki distribucioni sistem servera zastupljenih širom interneta i korisnicima garantuje brzo učitavanje sadržaja.

[6] Babel je kompajler (prevodilac) programskog jezika JavaScript. Uglavnom se koristi za prevođenje ECMAScript 2015+ koda u kompatibilnu verziju skript jezika JavaScript koja se koristi u mnogim pregledačima ili okruženjima. Što se tiče radnog okruženja ReactJS, značajna je njegova sposobnost da konvertuje JSX u odgovarajući JavaScript kôd.

Kôd 7 prikazuje linkove ka navedenim fajlovima.

Kôd 7 - Uvođenje radnog okruženja ReactJS pomoću sistema servera *CDN*

```
      <!– biblioteka React-->
        <script src= "https://unpkg.com/react@16/umd/react.production.min.js"></scri
pt>
      <!– biblioteka React-dom -->
        <script src= "https://unpkg.com/react-dom@16/umd/react-
dom.production.min.js">
        </script>
      <!–biblioteka Babel -->
        <script src="https://unpkg.com/babel-
standalone@6.15.0/babel.min.js"></script>
```

Da bi se omogućilo pravilno funkcionisanje veb strane, potrebno je da se u zaglavlju navede i element <meta> sa sledećim atributima: <meta name="viewport" content="width=device-width, initial-scale=1.0">. Ovaj element omogućava prilagođavanje širini ekrana, dajući instrukcije pregledaču kako da kontroliše i skalira dimenziju veb stranice. Atribut width određuje širinu strane i može mu se dodeliti konkretna vrednost piksela. Izraz width=device-width određuje da širina strane bude jednaka širini ekrana. Atribut initial-scale=1.0 postavlja inicijalnu vrednost prikaza veb strane prilikom njenog prvog učitavanja i sprečava podrazumevano uvećanje.

8 Kreiranje veb strane korišćenjem okruženja ReactJS

U prethodnom odeljku su razmotrena dva načina koja omogućuju da ReactJS aplikacija postane nešto što pregledač razume. U nastavku ovog rada radno okruženje ReactJS će biti uključivano iz distribucionog sistema servera *CDN*, sve dok ne bude eksplicitno rečeno drugačije.

Kôd 8 prikazuje kreiranje veb strane korišćenjem radnog okvira ReactJS.

Kôd 8 - Kreiranje veb strane u radnom okruženju ReactJS

```
<!DOCTYPE html>
<html>
 <head>
  <title>Uvod u React</title>
  <meta name="viewport" content="width=device-width, initial-scale=1.0" />
  <script src="https://unpkg.com/react@16/umd/react.production.min.js">
  </script>
  <script src="https://unpkg.com/react-dom@16/umd/react-
dom.production.min.js">
  </script>
  <script src="https://unpkg.com/babel-standalone@6.15.0/babel.min.js">
  </script>
 </head>
 <body>
  <script type="text/babel">
   ReactDOM.render(
    <h1>Zdravo svete!</h1>,
     document.body
   );
  </script>
 </body>
</html>
```

Rezultat pokretanja aplikacije koja je predstavljena u kodu 8 je ispisivanje naslova *Zdravo, svete!* u podrazumevanom pregledaču. Kako bi se sintaksa JSX prevela u odgovarajući JavaScript kôd, potrebno je atribut *type* u oznaci *script* podesiti na vrednost text/babel. Da bi se element prikazao, poziva se funkcija *ReactDOM.render()* koja dolazi iz biblioteke koja uključuje rad sa DOM elementima. Navodi se između skript tagova i jedna je od najčešćih metoda koje se koriste u razvoju ReactJS aplikacija.

Metoda *render* zahteva dva argumenta:

1. Elemente sintakse JSX koji treba da budu prikazani
2. Lokaciju DOM elementa u koju će se ispisati odgovarajući JSX kôd

Posmatrano unutar koda 8, kada se metoda *render* izvrši, oznaka *h1* i sadržaj unutar nje će biti postavljeni unutar elementa *body*. Međutim, nije preporučljivo sadržaj predviđen za ispis postavljati direktno u element *body*, naročito ako se ReactJS koristi u kombinaciji sa drugim JavaScript bibliotekama i radnim okvirima. Uobičajeno je da se napravi poseban element. On se tretira kao nov korenski element i služi kao odredište za metodu *render*. U tu svrhu će biti dodat element *div* čija će vrednost *id* biti *container*. Kada je element *div* definisan, potrebno je još izmeniti metodu *render* da koristi kreirani element umesto element *document.body*.

Kôd 9 prikazuje upotrebu novog elementa koji služi kao odredište za metodu *render*.

Kôd 9 – Odredište za metodu *render* je nov korenski element

```
<body>
 <div id="container"></div>
 <script type="text/babel">
  ReactDOM.render(
   <h1>Zdravo svete!</h1>,
   document.querySelector("#container")
  );
 </script>
</body>
```

9 Komponente

Komponente predstavljaju jedan od glavnih načina za definisanje vizuelnih elemenata aplikacije. One su nezavisne i ponovo upotrebljive celine koje treba struktuirati tako da svaka od njih bude odgovorna samo za jedan deo funkcionalnosti. Takva podela funkcionalnosti će ih učiniti jednostavnim i pogodnim za ponovnu upotrebu. Pored toga, manje komponente se lakše testiraju i održavaju, a sama aplikacija se brže ažurira.

Komponente su u osnovi HTML elementi, ali ReactJS omogućuje neka dodatna svojstva za bolju manipulaciju tih elemenata.

9.1 Načini definisanja komponenti

Komponente mogu biti napisane kao funkcije ili kao klase. Naziv komponente mora počinjati velikim slovom, jer bi u suprotnom ReactJS smatrao da se radi o standardim HTML elementima.

9.1.1 Funkcijske komponente

Najjednostavniji način za definisanje jedne komponente je pisanjem funkcije u skript jeziku JavaScript.

Kôd 10 - Komponenta definisana pomoću funkcije

```
function Pozdrav() {
  return <h2>Zdravo svete!</h2>;
}
```

Kôd 10 prikazuje funkcijsku komponentu bez ulaznih parametara. Međutim, u većini slučajeva je funkciji potrebno proslediti neke ulazne podatke, eventualno izvršiti neke promene nad njima i zatim vratiti rezultat.

Komponenti se prilikom inicijalizacije dodeljuju svojstva (eng. *properties*, skraćeno *props*). Za definisanje komponente dovoljno je napisati funkciju koja može da ima argument *props* (kako bi bilo moguće pristupiti svojstvima) i vraća JSX element koji opisuje kako će se komponenta prikazati u okviru DOM-a.

Kôd 11 predstavlja funkcijsku komponentu koja koristi svojstvo *props.*

Kôd 11 – Funkcijska komponenta kojoj se dodeljuje svojstvo *ime*

```
function Pozdrav(props) {
  return <h2>Zdravo {props.ime}!</h2>;
}
```

9.1.2 Klasne komponente

Drugi način za definisanje komponente je pomoću klase (kôd 12) u programskom jeziku JavaScript[7].

Kôd 12 – Komponenta definisana pomoću klase

```
class Pozdrav extends React.Component {
  render() {
    return <h2>Zdravo, {this.props.ime}!</h2>;
  }
}
```

Stvaranje komponente na ovaj način zahteva nasleđivanje klase *React.Component* pomoću ključne reči *extends*. Unutar klase je neophodno implementirati metodu *render*. Kao povratnu vrednost ona vraća opis izgleda te komponente što je kôd pisan JSX sintaksom, isto kao u slučaju funkcijske komponente. Ova metoda se razlikuje od metode iz klase *ReactDOM*, čija je uloga da ubaci element u čvor stabla u DOM-u.

Budući da je komponenta definisana kao klasa, unutar metode *render* se koristi objekat *this* da bi se omogućio pristup svojstvima te komponente.

[7] U radu se koristi ES6 standard programskog jezika JavaScript.

9.2 Prikazivanje komponenti

Bilo da je komponenta definisana kao funkcija ili kao klasa, inicijalizacija komponente je ista. Potrebno je proslediti vrednost svojstva kao deo poziva za komponentu. Svojstva se dodaju na isti način na koji se dodaju atributi standardnim HTML elementima. To funkcioniše tako što se doda atribut sa istim imenom kao što je ime svojstva, nakon čega sledi vrednost koja se prosleđuje. Prilikom prikazivanja komponente mora se vratiti samo jedan element. Ukoliko ima više poziva komponente, potrebno ih je obaviti jednim zajedničkim HTML elementom (kôd 13). To može biti i prazan tag.

Kôd 13 – Svojstvo koje se u ovom primeru prosleđuje komponenti *Pozdrav* je *ime*

```
ReactDOM.render(
  <div>
    <Pozdrav ime="Sara" />
    <Pozdrav ime="Relja" />
    <Pozdrav ime="Iva" />
  </div>,
  document.querySelector("#container")
);
```

Svojstva unutar komponente utiču na izgled korisničkog okruženja. Ta svojstva se ne mogu menjati unutar same komponente. Ukoliko korisnik želi da promeni izgled korisničkog okruženja potrebno je da uništi istu komponentu i kreira novu kojoj će predati druge vrednosti.

9.3 Inicijalizacija i korišćenje stanja

Komponente definisane u [9.1] su komponente bez stanja. Imaju svojstva i to je zapravo način na koji komponente međusobno komuniciraju. Informacije koje se ovim putem prosleđuju se ne mogu menjati. U mnogim interaktivnim rešenjima, to nije ono čemu se teži. Poželjno je da mogu da se menjaju aspekti komponenti, npr. prilikom vraćanja nekih podataka sa servera. Zbog toga je potreban neki drugi način da se unutar komponente čuvaju podaci koji prevazilaze svojstva. Zapravo, potreban je način za čuvanje podataka koji se mogu menjati. To se postiže upotrebom stanja.

Stanje je promenljivi skup podataka sadržan unutar objekta *state* neke komponente. Komponenta može unutar sebe da inicijalizuje, menja i koristi stanje. Ono u čemu se razlikuju komponente definisane kao funkcije i komponente definisane kao klase je upotreba stanja. Sve do 2019. godine i pojave funkcija *react hooks*[8], stanje nije moglo da se koristi unutar funkcijske komponente. Ukoliko je komponenta zahtevala upotrebu stanja, morala je biti napisana kao klasa.

Kôd 14 prikazuje komponentu *Brojac* koja koristi stanje.

Kôd 14 – Komponenta *Brojac* ilustruje upotrebu stanja

```
class Brojac extends React.Component {
  state = {
    brojac: 0
  }
  render() {
    return (
      <h1>{this.state.brojac}</h1>
    )
  }
}
```

U okviru koda 14 komponenta *Brojac* sadrži objekat *state*, unutar koga je definisana promenljiva *brojac* kao njegovo svojstvo. Opis izgleda komponente se definiše u okviru funkcije *render*.

U skript jeziku JavaScript prilikom rada sa klasom može da se pozove konsktruktor. Konstruktor je mesto gde se inicijalizuje stanje komponente. Sa stanovišta radnog okvira ReactJS, svejedno je koji se od ova dva načina primenjuje za kreiranje komponente i njenog stanja.

Kôd 15 prikazuje drugi način definisanja komponente *Brojac*.

[8] *React Hooks* su funkcije koje su postale dostupne pojavom verzije React 16.8. Upotreba ovih funkcija značajno olakšava razvoj aplikacija. Njihov značaj se ogleda u mogućnosti korišćenja stanja i životnog ciklusa unutar funkcijskih komponenti.

Kôd 15 – Komponenta *Brojac* uključuje upotrebu konstruktora za definisanje stanja

```
class Brojac extends React.Component {
  constructor(props) {
    super(props);

    this.state = {
      brojac: 0
    };
  }
  render() {
    return (
      <h1>{this.state.brojac}</h1>
    )
  }
}
```

9.3.1 Promena stanja komponente

Radni okvir ReactJS ažurira korisnični interfejs na osnovu stanja komponente. Stanje se može promeniti u samoj komponenti pomoću metode *setState*. Njoj se prosleđuje objekat koji sadrži novo stanje. Unutar prosleđenog objekta je dovoljno navesti samo one podatke iz skupa stanja koje je potrebno promeniti. Nakon svake promene stanja poziva se *render* metoda i prikazuje novo stanje komponente. Promenom objekta *state* unutar komponente dolazi do automatskog ponovnog prikazivanja te komponente u DOM. Tom prilikom se menjaju samo delovi korisničkog okruženja na koji utiče promenjeni podatak, pri čemu ostatak DOM-a ostaje nepromenjen. Kôd 16 prikazuje komponentu *Brojac* koja ilustruje promenu stanja. Argument metode *setState* je objekat koji sadrži svojstvo koje se integriše u objekat *state*.

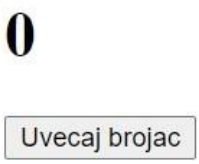
Kôd 16 - Promena stanja komponente *Brojac*

```
class Brojac extends React.Component {
  constructor(props) {
    super(props);

    this.state = {
      brojac: 0,
    };
  }
  uvecajBrojac = () => {
    this.setState({
      brojac: this.state.brojac + 1,
    });
  };

  render() {
    return (
      <div>
        <h1>{this.state.brojac}</h1>
        <button onClick={this.uvecajBrojac}> Uvecaj brojac </button>
      </div>
    );
  }
}
```

Kada se pokrene kôd 16, na veb stranici će biti prikazan brojač i dugme koje omogućava povećanje vrednosti brojača (slika 5). Vrednost brojača se menja zahvaljujući promeni stanja. Primer dat u okviru koda 16 se može testirati na elektronskoj stranici kursa kojoj se može pristupiti putem linka http://edusoft.matf.bg.ac.rs/eskola_veba/#/course/react/react_komponente.



Slika 5 – Pritiskom na dugme menja se vrednost brojača

Uglavnom se promena stanja komponente postiže upotrebom događaja. Korisnik aplikacije pritiskom na dugme ili unošenjem teksta u *input* polje može izazvati promenu stanja komponente i na taj način promeniti izgled korisničkog okruženja.

Više o struktuiranju komponenti i upravljanju njihovim stanjem se može naći u [2].

10 Upravljanje događajima

Upravljanje događajima u radnom okviru ReactJS se ostvaruje tako što se HTML elementima dodaju atributi koji predstavljaju akcije nad njima, a vrednost koja im se dodeljuje je metoda koja će se pozvati nakon što se ta akcija izvrši. Metoda koja se poziva se navodi unutar vitičastih zagrada. U kodu 16, metoda *uvecajBrojac* je definisana kao *callback* metoda. Ukoliko bi ona bila napisana kao obična funkcija, onda se pritiskom na dugme (slika 5) ne bi povećavala vrednost brojača. Zbog toga je u ovom slučaju neophodno koristiti metodu *bind* unutar konstruktora (kôd 17). U radnom okviru ReactJS vrednost za objekat *this* se ne odnosi na element koji je pokrenuo događaj. Ta vrednost je nedefinisana i zbog toga se mora zadati kontekst na koji se ključna reč *this* odnosi. To se postiže upotrebom metode *bind*.

Kôd 17 – Promena stanja komponente *Brojac* pomoću metode *bind*

```
class Brojac extends React.Component {
  constructor(props) {
    super(props);
    this.state = { vrednost: 0 };
    this.uvecajBrojac = this.uvecajBrojac.bind(this);
  }
  uvecajBrojac() {
    this.setState({
      vrednost: this.state.vrednost + 1
    })
  }
  render() {
    return (
      <div>
        <h1>{this.state.vrednost}</h1>
        <button onClick={this.uvecajBrojac}> Uvecaj brojac </button>
      </div>
    )
  }
}
```

Kada se zada događaj, kao što je to slučaj sa događajem *onClick* u kodu 17, ne radi se direktno sa običnim DOM događajima. Radi se sa tipom događaja koji je specifičan za radni okvir ReactJS, a poznat je pod nazivom sintetički događaj (eng. *SyntheticEvent*). Procedure za obradu događaja neće dobiti matične argumente događaja kao što su događaji miša, tastature, itd. One uvek dobijaju argumente sintetičkog događaja koji obavijaju matične događaje pregledača. Taj pristup čini te objekte kompatibilnim bilo kom pregledaču.

Najčešća metoda klase *SyntheticEvent* koja se koristi je *preventDefault*. Ona sprečava izvršavanje podrazumevane (neželjene) akcije prilikom pokretanja pregledača.

Više informacija o *događajima* u radnom okviru ReactJS se može naći u [1].

11 Životni ciklus komponente

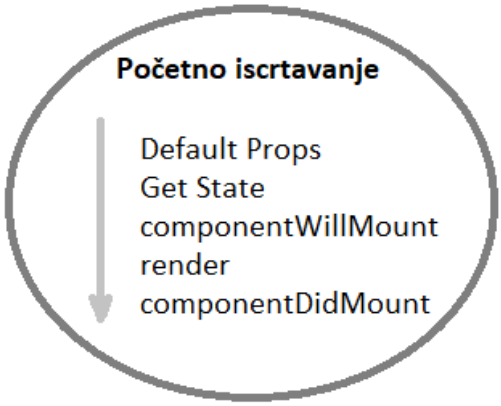
Metode životnog ciklusa komponente su specijalne procedure za obradu događaja koje se pozivaju u određenim trenucima života komponente. Kao i kod procedura za obradu događaja, može se pisati kôd koji će obavljati razne stvari u tim trenucima. Metode životnog ciklusa komponente se mogu svrstati u tri grupe:

1. metode iscrtavanja,
2. metode ažuriranja i
3. metode uništenja.

Metode iscrtavanja se pozivaju kada se komponenta inicijalizuje i smešta u DOM. Metode ažuriranja se pozivaju pri promeni svojstva i stanja komponente, dok se metode uništenja pozivaju prilikom uklanjanja komponente iz DOM-a.

11.1 Metode iscrtavanja

Kada komponenta treba da započne svoj životni ciklus pozivaju se sledeće metode (slika 6):



Slika 6 - Metode životnog ciklusa koje se inicijalno pozivaju

11.1.1 Unošenje podrazumevanih svojstava

Unošenje podrazumevanog svojstva omogućava da se komponenti zada podrazumevana vrednost za svojstvo *props*. Komponenti *Pozdrav* se može zadati svojstvo *ime* na način koji je prikazan u kodu 18. Kôd 18 se izvršava pre nego što komponenta bude definisana.

Kôd 18 – Zadavanje podrazumevanog svojstva komponente

```
Pozdrav.defaultProps = {
 ime: "Jovan",
};
```

11.1.2 Unošenje podrazumevanog stanja

Unošenje podrazumevanog stanja komponente se odnosi na zadavanje podrazumevane vrednosti za objekat *state* kao deo stvaranja komponente (kôd 19).

Kôd 19 – Objekat stanja je definisan svojstvom brojac čija je vrednost 0

```
constructor(props){
  super(props);

  this.state = {
    brojac: 0
  }
}
```

Metoda koja se poziva pre nego što komponenta bude iscrtana u DOM je metoda *componentWillMount*. Ako se pozove metoda *setState* unutar ove metode, komponenta se neće ponovo iscrtavati. Nakon nje se poziva metoda *render*. To je ujedno i najčešće korišćena metoda životnog ciklusa. Ona služi za prikazivanje komponente prilikom kreiranja korisničkog okruženja. Metoda *render* se izvršava za vreme inicijalizacije komponente i prilikom ažuriranja njenog stanja. Unutar same metode se ne poziva metoda *setState*, jer bi u tom slučaju došlo do izvršavanja beskonačne petlje (*render => setState => render => setState => render*...).

Odmah nakon što se komponenta iscrta i postavi u DOM poziva se metoda *componentDidMount*. Ova metoda se obično koristi kada se pristupa nekim podacima u bazi i dozvoljava korišćenje metode *setState*.

Sa izuzetkom metode *render*, navedene metode životnog ciklusa se mogu pokrenuti samo jednom. Kada su komponente dodate u DOM, one se potencijalno mogu ponovo ažurirati i ponovo iscrtavati kada se promene objekti svojstva ili stanja same komponente.

Kôd 20 prikazuje korišćenje metode *componentDidMount*.

Kôd 20 – Upoteba metode životnog ciklusa *componentDidMount*

```
constructor(props){
  super(props);

  this.state = {
    data: []
  }
}

componentDidMount(){
  this.setState({
    data: [1, 2, 3, 4, 5]
  });
}
```

U kodu 20 se zadaje inicijalno stanje podatkom koji je prazan niz. Nakon definisanja inicijalnog stanja, poziva se metoda *componentDidMount*, kako bi se *state* objekat popunio podacima.

Hijerarhija događaja od postavljanja inicijalnog *state* objekta do njegovog ažuriranja:

- komponenta se iscrtava u DOM sa podacima koji se nalaze na inicijalnom *state* objektu,
- objekat *state* se ažurira,
- komponenta se ponovo iscrtava u DOM sa drugim podacima.

11.2 Faza ažuriranja

Jedna od najčešće korišćenih metoda tokom promene stanja komponente je metoda *componentDidUpdate*. Ova metoda se poziva nakon ažuriranja komponente i pozivanja metode *render*. Ako treba da se izvrši neki kôd nakon ažuriranja, onda se smešta unutar ove metode.

Kôd 21 prikazuje upotrebu metoda *componentDidMount* i *componentDidUpdate*.

Kôd 21 – Upotreba metoda *componentDidMount* i *componentDidUpdate*

```
class Header extends React.Component {
  constructor(props) {
    super(props);
    this.state = { boja: "crvena" };
  }
  componentDidMount() {
    setTimeout(() => {
      this.setState({ boja: "zuta" })
    }, 3000)
  }
  componentDidUpdate() {
    document.getElementById("root").innerHTML =
      "Nakon ažuriranja, omiljena boja je " + this.state.boja;
  }
  render() {
    return (
      <div>
        <h1>Moja omiljena boja je {this.state.boja}</h1>
        <div id="root"></div>
      </div>
    );
  }
}
```

Sadržaj veb stranice nakon pokretanja koda 21, prikazan je na slici 7. Nakon tri sekunde sadržaj veb stranice se menja i prikazan je na slici 8.

Slika 7 – Sadržaj veb stranice nakon pokretanja *koda 21*

Slika 8 – Sadržaj promenjene veb stranice nakon pokretanja koda 21

Kada se komponenta montira i smešta u DOM, boja je "crvena". Nakon što je komponenta smeštena u DOM, tajmer menja stanje i boja postaje "žuta". Metoda *componentDidUpdate* se poziva nakon što je izvršena promena stanja i ispisuje poruku u prazan *div* element.

11.3 Faza uništenja komponente

Kada se metoda uklanja iz DOM-a aktivna je samo jedna metoda životnog ciklusa, a to je *componentWillUnmount*. Ukoliko je potrebno izvršiti uklanjanje osluškivača događaja ili zaustavljanje tajmera, to se može učiniti unutar ove metode. Kada se ova metoda pozove, komponenta se uklanja iz DOM-a.

Više informacija o životnom ciklusu komponenti se može naći u [4].

12 React Hooks

React Hooks su funkcije koje su postale dostupne pojavom verzije React 16.8, početkom 2019. godine. Upotreba ovih funkcija značajno olakšava razvoj aplikacija. Njihov značaj se ogleda u mogućnosti korišćenja stanja i životnog ciklusa unutar funkcijskih komponenti. Pre pojave ove verzije radnog okvira ReactJS, nije bilo moguće definisati stanje unutar komponenti koje nisu napisane kao klase.

Najčešće korišćena funkcija među njima je ***useState***. Ona omogućuje inicijalizaciju i promenu stanja funkcijske komponente, analogno svojstvima objekta *state* u komponentama koje su definisane kao klase.

12.1 Korišćenje objekta kao promenljive stanja pomoću funkcije *useState*

Kôd 22 prikazuje upotrebu objekta kao promenljive stanja pomoću funkcije *useState*.

Kôd 22 – Primer upotrebe funkcije *useState*

```
const { useState } = React;
   const App = () => {
    const [poruka, postaviPoruku] = useState({ tekst: "" });
    return (
     <div>
      <p>{poruka.tekst}</p>
      <input type="text"
       onChange={(event) => {
        poruka.test = event.target.value;
        postaviPoruku(poruka);  //nije ispravno
       }}
       placeholder="unesi tekst"
      />
     </div>
    );
   };
```

Umesto da kreira novi objekat, kôd unutar događaja *onChange* transformiše postojeći objekat stanja (kôd 22). U radnom okviru ReactJS to je isti objekat. Da bi promena stanja funkcionisala, potrebno je kreirati objekat na sledeći način (kôd 23).

Kôd 23 – Događaj *onChange* transformiše postojeći objekat stanja

```
        onChange={(event) => {
           const novaPoruka = {tekst: event.target.value};
           postaviPoruku(novaPoruka);
        }}
```

U kodu 23 funkcijska komponenta *App* sadrži element *input* i zavisno od sadržaja koji se unese, ispisuje isti na ekranu.

Funkcija *useState* vraća niz od dva elementa: prvi element je promenljiva koja predstavlja svojstvo, a drugi je funkcija kojom se to stanje ažurira. Funkcija *useState* kao argument zahteva jedan parametar koji je inicijalna vrednost svojstva. U kodu 23 argument funkcije *useState* je objekat. Za razliku od komponenata koje su definisane kao klase, u funkcijskim komponentama stanje ne mora biti objekat, već bilo koji tip podataka. Svaki segment stanja sadrži sopstvenu vrednost, koja može da bude objekat, niz, logička vrednost, itd. Argument početnog stanja koristi se samo tokom prvog prikazivanja komponente u DOM.

Funkcija koju vraća *useState* ne nadovezuje objekte kao što je to slučaj sa funkcijom *setState* unutar komponenata koje su definisane kao klase.

Kôd 24. Funkciji *useState* je prosleđen objekat koji sadrži više od jednog svojstva

Kôd 24 – Argument funkcije *useState* je objekat koji sadrži dva svojstva

```
const { useState } = React;

   const App = () => {
    const [poruka, postaviPoruku] = useState({tekst: "", id: 1 });

    return (
     <div>
      <p>{poruka.id}: {poruka.tekst}</p>
      <input
       type="text"
       onChange={(event) => {
        const novaPoruka = {tekst: event.target.value};
        postaviPoruku(novaPoruka);
       }}
       placeholder="unesi tekst"
      />
     </div>
    );
   };
```

Sadržaj veb stranice nakon pokretanja HTML dokumenta koji je dat u kodu 24 je prikazan na slici 9.

Slika 9 – Sadržaj veb stranice nakon pokretanja koda 24

Nakon unosa teksta u *input* polje (slika 9), dati sadržaj se ispisuje u liniji iznad, ali će se oznaka koja označava svojstvo izbrisati. Ovakav način pozivanja događaja *onChange*, koji samo sadrži svojstvo *tekst*, učiniće da svojstvo *id* nestane. Naime, ReactJS je zamenio postojeći objekat stanja {tekst: " ", id: 1 }, objektom {tekst: event.target.value}. Jedan od načina za rešenje ovog problema je da se događaj *onChange* napiše kao što je to u kodu 25. Na taj način će se omogućiti pristup svim svojstvima objekta, a svojstvo *tekst* će dobiti novu vrednost.

Kôd 25 – Događaj *onChange* transformiše postojeći objekat stanja

```
onChange={(event) => {
   const val = event.target.value;
   postaviPoruku(prethodnoStanje => {
     return { ...prethodnoStanje, tekst: val }
   });
}}
```

Isti rezultat se postiže korišćenjem metode *Object.assign* (kôd 26).

Kôd 26 – Metoda *Object.assign* transformiše postojeći objekat stanja

```
onChange={(event) => {
   const val = event.target.value;
   postaviPoruku(prethodnoStanje => {
     return Object.assign({}, prethodnoStanje, { tekst: val });
   });
}}
```

12.2 Korišćenje funkcije *useState* u radu sa nizovima

Kôd 27 prikazuje korišćenje funkcije *useState* u radu sa nizovima.

Kôd 27 – ReactJS aplikacija koja prikazuje korišćenje funkcije *useState* u radu sa nizovima

```
const { useState } = React;

 function App(){

  const[ime, dodajIme] = useState(["Jovan","Marko"]);
  const[novoIme, dodajNovoIme] = useState("");

  const sacuvajIme =() =>{
    dodajIme([...ime, novoIme]);
    dodajNovoIme("");
  }

  return(
   <div className="container">
    {ime.map(element=><li>{element}</li>)}
    <input
       type="text"
       onChange={(event)=>dodajNovoIme(event.target.value
       value={novoIme}
       placeholder="novo ime"/>
    <button onClick={sacuvajIme}>Dodaj</button>
   </div>
   )
  }
```

Sadržaj internet stranice nakon pokretanja HTML dokumenta koji je dat u kodu 27 prikazan je na slici 10.

Slika 10 – Sadržaj veb stranice nakon pokretanja koda 27

Nakon unosa novog imena u okviru *input* polja, potrebno je pokrenuti dugme *Dodaj* i to ime će se prikazati u okviru liste, odmah ispod postojećih imena (slika 10).

U okviru prvog poziva funkcije *setState* inicijalna vrednost stanja je niz imena. Događaju *onClick* je prosleđena metoda *sacuvajIme* koja dodaje ime na već postojeći niz imena. Za dodavanje elemenata u niz korišćen je *spread* operator. Imena su prikazana u okviru nenumerisane liste, dok je za pristup pojedinačnim članovima niza korišćena funkcija *map*.

Više informacija o *hooks* funkciji *useState* se može naći u [7].

12.3 Funkcija *useEffect*

Hooks funkcije su pojednostavile upravljanje metodama životnog ciklusa unutar funkcijskih komponenti. Najčešće korišćena među njima je funkcija *useEffect*.

Funkcija *useEffect* se podrazumevano pokreće nakon prvog iscrtavanja komponente u DOM, ali i nakon svakog ažuriranja. Kao argument može da ima jedan ili dva parametra. Prvi je funkcija povratnog poziva, a drugi parametar je niz i on je opcioni. Ukoliko se ne navede drugi parametar, sporedni efekti[9] se pokreću nakon svakog prikazivanja komponente. Inače, ukoliko se kao drugi parametar navede prazan niz, ovi efekti se pokreću jednom, nakon inicijalnog prikazivanja (renderovanja). Najveći značaj funkcije *useEffect* je to što obezbeđuje pokretanje sporednih efekata nezavisno od prikazivanja.

[9] Ako funkcijska komponenta vrši izračunavanja koja ne ciljaju izlaznu vrednost, onda se ti proračuni nazivaju sporedni efekti. Primeri sporednih efekata su pristup podacima iz baze, direktna manipulacija DOM-om, korišćenje funkcija *SetTimer, SetInterval,* itd.

Drugi argument funkcije *useEffect* kontroliše kada se sporedni efekti pokreću i zavisno od toga razlikuju se sledeći slučajevi:

1. ukoliko se ne navede argument, sporedni efekti se pokreću nakon svakog prikazivanja,
2. ukoliko je argument prazan niz, sporedni efekti se pokreću jednom, nakon incijalnog prikazivanja,
3. ukoliko argument sadrži svojstva *props* i *state*, sporedni efekti se pokreću samo kada se njihove vrednosti promene.

12.3.1 Upotreba funkcije *useEffect* na način na koji se sporedni efekti pokreću jednom, nakon inicijalnog prikazivanja

Kôd 28. Funkcija *useEffect* pokreće sporedni efekat samo jednom, nakon inicijalnog prikazivanja.

Kôd 28 – Komponenta *Pozdrav* koristi funkciju *useEffect*

```
const { useEffect } = React;

function Pozdrav({ ime }) {
  const poruka = `Zdravo, ${ime}!`;

  useEffect(() => {
    document.title = 'Stranica sa pozdravom'; // sporedni efekat
  }, []);

  return <div>{poruka}</div>;
}
```

Čak i ako se komponenta ponovo poziva sa različitim svojstvom imena (kôd 29), sporedni efekat se pokreće samo jednom, nakon prvog prikazivanja. Ovaj način upotrebe funkcije *useEffect* unutar funkcijske komponente se može uporediti sa metodom životnog ciklusa *ComponentDidMount* unutar komponente klase.

Kôd 29 – Inicijalizacija komponente *Pozdrav* definisane u *kodu 28*

```
// Prvo prikazivanje
<Pozdrav ime="Stefan" />  // pokrece se sporedni efekat

// Drugo prikazivanje, svojstvo props se menja
<Pozdrav ime="Jovan" />  // ne pokrece se sporedni efekat

// Trece prikazivanje, svojstvo props se menja
<Pozdrav ime="Marko" />  // ne pokrece se sporedni efekat
```

12.3.2 Upotreba funkcije *useEffect* na način na koji se sporedni pokreću samo kada se vrednosti *props* i *state* promene

Kôd 30. Funkcija *useEffect* pokreće sporedni efekat kada se vrednosti *props* i *state* promene.

Kôd 30 – Drugi argument funkcije *useEffect* su vrednosti svojstava *props* i *state*

```
const { useEffect } = React;
function App({ prop }) {
  const [state, setState] = useState();
  useEffect(() => {
    // sporedni efekti koriste vrednosti props i state
  }, [prop, state]);
  return <div>....</div>;
}
```

Svaki put kada sporedni efekti koriste vrednosti *props* i *state*, isti se moraju navesti kao argumenti niza koji je drugi parametar funkcije *useEffect*. Funkcija povratnog poziva se poziva nakon što se promene izvrše na DOM-u ako i samo ako se neke od vrednosti *props* i *state* promene. Korišćenje argumenta zavisnosti u funkciji *useEffect* omogućava kontrolu poziva sporednih efekata, nezavisno od ciklusa prikazivanja komponente. To je suština funkcije *useEffect*.

Kôd 31. Funkcija *useEffect* će pokrenuti sporedni efekat ako se odgovarajuće svojstvo promeni.

Kôd 31 - Drugi argument funkcije *useEffect* je svojstvo *ime*

```
const { useEffect } = React;

function Pozdrav({ ime }) {
  const poruka = `Zdravo, ${ime}!`;

  useEffect(() => {
    document.title = 'Zdravo, ${ime}'; // sporedni efekat
  }, [ime]);

  return <div>{poruka}</div>;
}
```

Funkcija *useEffect* pokreće sporedni efekat nakon početnog (inicijalnog) prikazivanja, a nakon sledećih prikazivanja samo ako se vrednost svojstva promeni (kôd 32).

Kôd 32 – Inicijalizacija komponente *Pozdrav* definisane u *kodu 31*

```
// Prvo prikazivanje
<Pozdrav ime="Jovan" />  // pokrece se sporedni efekat

// Drugo prikazivanje, svojstvo props se menja
<Pozdrav ime="Stefan" />  // pokrece se sporedni efekat

// Trece prikazivanje, svojstvo props se menja
<Pozdrav ime="Stefan" />  // ne pokrece se sporedni efekat

// Cetvrto prikazivanje, svojstvo props se menja
<Pozdrav ime="Marko" />  // pokrece se sporedni efekat
```

Ovaj način upotrebe funkcije *useEffect* unutar funkcijske komponente se može uporediti sa metodom životnog ciklusa *ComponentDidUpdate* unutar komponente klase.

12.3.3 Čišćenje nakon izvršenja sporednih efekata

Neki sporedni efekti zahtevaju čišćenje. Ako funkcija povratnog poziva koja je argument funkcije *useEffect* vraća neku funkciju, tada se u toj funkciji vrši čišćenje nakon tog sporednog efekta (kôd 33).

Kôd 33 – Čišćenje nakon sporednog efekta

```
useEffect(() => {
  // sporedni efekat
  return function izbrisi() {
  // uklanjanje nakon sporednog efekta
  };
 }, []);
```

Čišćenje nakon sporednog efekta se odvija na sledeći način:

1. Nakon inicijalnog prikazivanja, funkcija *useEffect* poziva povratnu funkciju. Funkcija za čišćenje se ne poziva;
2. Tokom sledećih prikazivanja, *useEffect* poziva funkciju koja se odnosi na čišćenje nakon izvršenja prethodnog sporednog efekta, a zatim pokreće trenutni sporedni efekat;
3. Nakon uklanjanja komponente, *useEffect* poziva funkciju za čišćenje nakon izvršenja poslednjeg sporednog efekta.

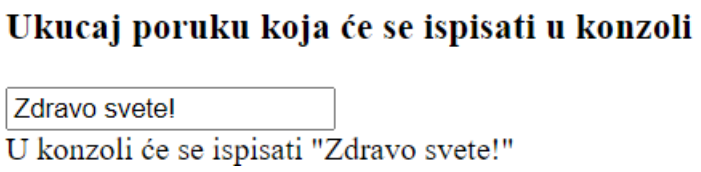
Kôd 34. Sporedni efekti se javljaju prilikom upotrebe metode *setInterval*.

Kôd 34 – Funkcijska komponenta *App* poziva funkciju *PonoviPoruku*. Argument funkcije *useEffect* je metoda *setInterval* pomoću koje konzola beleži svake dve sekunde svaku poruku koja je uneta u polje za unos teksta

```
const { useEffect, useState } = React;


function PonoviPoruku({ poruka }) {
  useEffect(() => {
    setInterval(() => {
      console.log(poruka);
    }, 2000);
  }, [poruka]);
  return <div>U konzoli će se ispisati "{poruka}"</div>;
}
function App() {
  const [poruka, postaviPoruku] = useState("Zdravo svete!")
  return (
    <div>
      <h3>Ukucaj poruku koja će se ispisati u konzoli</h3>
      <input
        type="text"
        value={poruka}
        onChange={(e) => postaviPoruku(e.target.value)}
      />
      <PonoviPoruku poruka={poruka} />
    </div>
  );
}
```

Sadržaj veb stranice nakon pokretanja HTML dokumenta koji je dat u kodu 34 je prikazan na slici 11.



Slika 11 – Sadržaj veb stranice nakon pokretanja koda 34

Zatim je potrebno otvoriti konzolu i pratiti promene. Konzola beleži svake dve sekunde svaku poruku koja je ikada otkucana u polju predviđenom za unos teksta. Ovo je slučaj kada je potrebno ukloniti sporedni efekat. To se postiže tako što se vrati funkcija koja zaustavlja prethodni *tajmer* (kôd 35). Na taj način se samo poslednja izmena evidentira u konzoli. Više informacija o sporednim efektima se može naći u [8].

Kôd 35 – Funkcija *useEffect* koristi metodu *setInterval* na način na koji se samo poslednja izmena evidentira u konzoli

```
useEffect(() => {
  const id = setInterval(() => {
      console.log(poruka);
    }, 2000);
    return () => {
      clearInterval(id);
    };
  }, [poruka]);
  return <div>U konzoli ce se ispisati "{poruka}"</div>;
}
```

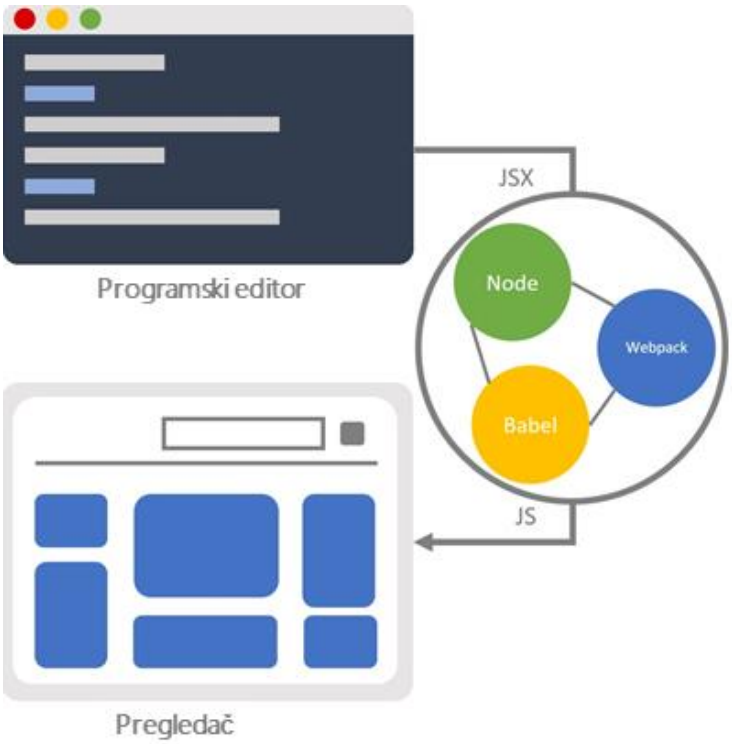
13 Korišćenje biblioteke *Create React App* za podešavanje okruženja

Za potrebe ovog rada, radno okruženje ReactJS je uključivano pomoću distribucionog sistema servera CDN. To je podrazumevalo pristup odgovarajućim skript datotekama u zaglavlju HTML dokumenta. Pomenuti način stvara brzu i direktnu putanju koja obezbeđuje prevođenje sintakse JSX u odgovarajući JavaScript kôd. Loša strana samog pristupa su performanse. Pored toga što se bavi učitavanjem strane, pregledač je odgovoran i za prevođenje sintakse JSX u JavaScript kôd (slika 12). To je proces koji oduzima vreme.

Slika 12 – Pregledač transformiše odgovarajući JSX kôd

Drugi način kreiranja radnog okruženja ReactJS podrazumeva da se konverzija sintakse JSX u odovarajući JavaScript kôd obavlja kao deo izgradnje aplikacije, što se ostvaruje kombinacijom alata Node.js, Webpack[10] i Babel (slika 13).

[10] *Webpack* je paket modula otvorenog koda za skript jezik JavaScript. Prevodi JSX kôd, ReactJS elemente i njihove zavisnosti i prevodi ih u jedan JavaScript dokument.



Slika 13 – Kombinacija odgovarajućih alata je zaslužna za prevođenje sintakse JSX u JavaScript kôd

Pre nekoliko godina, podešavanje razvojnog okruženja je uključivalo ručno konfigurisanje ovih alata. Projekat *Create React*[11] je znatno pojednostavio postupak podešavanja radnog okruženja. Za početak je potrebno instalirati najnoviju verziju okruženja Node.js. Zatim, u komandnoj liniji uneti sledeću naredbu

```
npm install -g create-react-app
```

i pritisnuti Enter/Return.

Nakon nekog vremena, kada se instaliranje završi, u komandnoj liniji je potrebno pristupiti direktorijumu u kome će se napraviti nov projekat i uneti naredbu

```
create-react-app zdravosvete.
```

Rezultat izvršenja date komande je projekat nazvan *zdravosvete*. Da bi se projekat testirao, potrebno je pristupiti direktorijumu *zdravosvete* pomoću naredbe

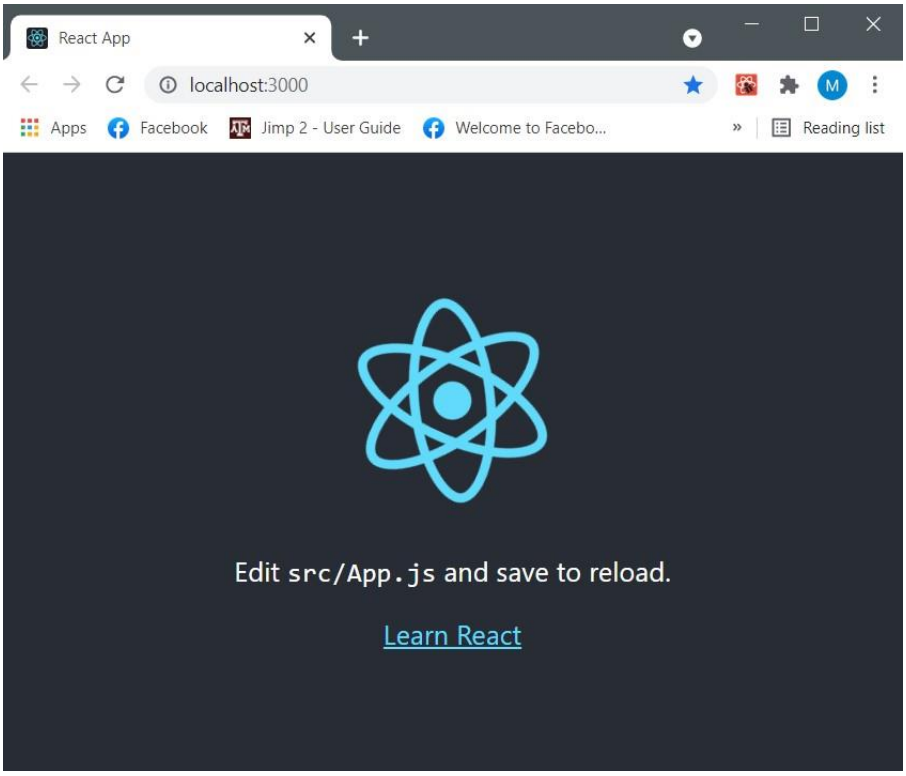
```
cd zdravosvete.
```

Nakon toga je moguće testirati aplikaciju unošenjem komande

```
npm start.
```

[11] *Create – React* je postupak koji predstavlja način kreiranja radnog okruženja React.js. Dostupan je na adresi https://github.com/facebookincubator/create-react-app.

Projekat će biti izgrađen, lokalni veb server pokrenut (slika 14).

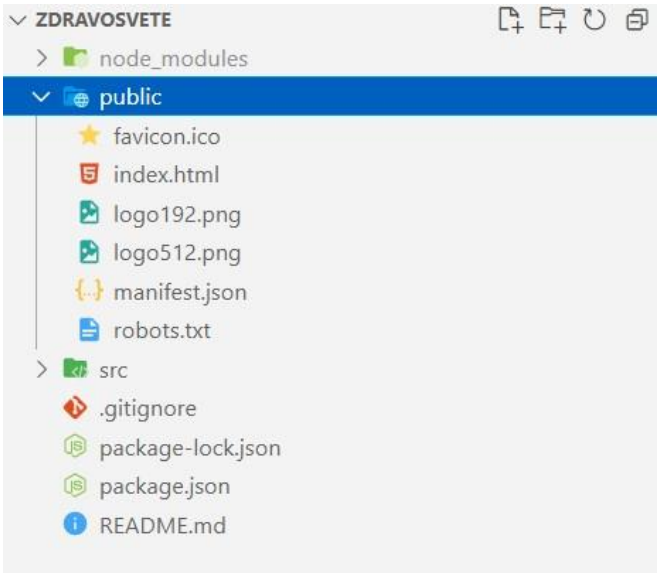


Slika 14 – Lokalni veb server

Dobijeni projekat je podrazumevani sadržaj koji generiše komanda

create - react - app.

Struktura datoteka i direktorijuma nakon pokretanja aplikacije *zdravosvete* izgleda kao na slici 15.



Slika 15 – Struktura datoteka i direktorijuma

Datoteka *index.html* u direktorijumu *public* se otvara u pregledaču. Sadržaj datoteke *index.html*, sa uklonjenim komentarima je dat u kodu 36. Svaka datoteka *index.html* sadrži jedan DOM element, podrazumevano nazvan *root* u koji se smeštaju ostali elementi.

Kôd 36 – Sadržaj datoteke *index.html*

```
<!DOCTYPE html>
<html lang="en">
 <head>
  <meta charset="utf-8" />
  <link rel="icon" href="%PUBLIC_URL%/favicon.ico" />
  <meta name="viewport" content="width=device-width, initial-scale=1" />
  <meta name="theme-color" content="#000000" />
  <meta name="description"
   content="Web site created using create-react-app" />
  <title>React App</title>
 </head>
 <body>
  <div id="root"></div>
 </body>
</html>
```

Sadržaj React.js aplikacije i JSX kôd su smešteni u direktorijumu *src*. Polazište aplikacije je smešteno u datoteku *index.js* (kôd 37).

Kôd 37 – Sadržaj datoteke *index.js*

```
import React from 'react';
import ReactDOM from 'react-dom';
import './index.css';
import App from './App';
import reportWebVitals from './reportWebVitals';

ReactDOM.render(
 <React.StrictMode>
  <App />
 </React.StrictMode>,
 document.getElementById('root')
);

reportWebVitals();
```

Poziv za metodu *ReactDOM.render* (kôd 37) zahteva korenski element koji je naveden unutar datoteke *index.html*. Naredba *import* je deo modula u skript jeziku JavaScript. Cilj modula je da podeli funkcionalnost aplikacije na progresivno manje delove. Neki moduli koji se uvoze su deo koda u projektu. Drugi moduli su spoljni za projekat i to su biblioteke *React* i *React-DOM*. Pored njih, uvozi se i CSS datoteka, kao i komponenta koja se podrazumevano označava kao *App.js*.

Kôd 38 – Sadržaj datoteke *App.js*

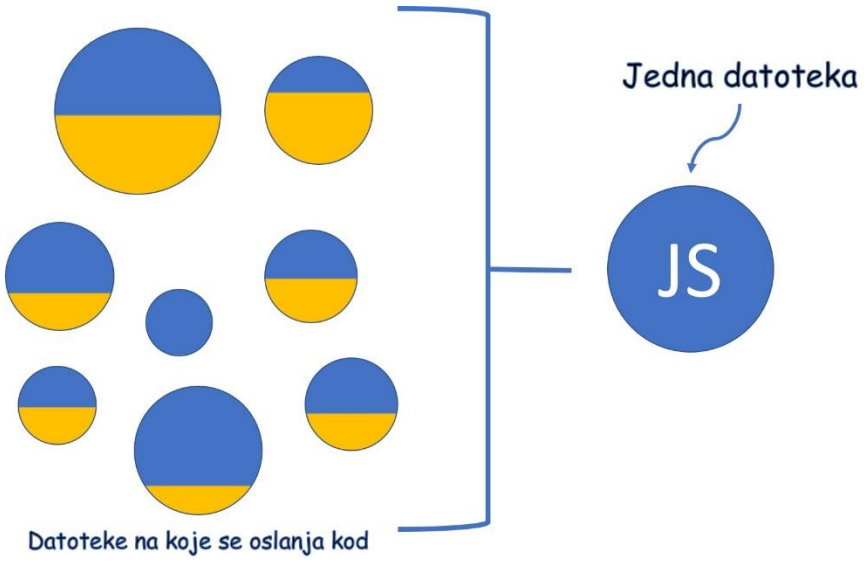
```
import logo from './logo.svg';
import './App.css';

function App() {
 return (
  <div className="App">
   <header className="App-header">
    <img src={logo} className="App-logo" alt="logo" />
   </header>
  </div>
 );
}

export default App;
```

Datoteka *App.js* (kôd 38) sadrži naredbu *export* i ime koje će projekat koristiti da identifikuje izvezeni modul. To ime se navodi prilikom uvoza modula *App* u druge delove projekta, kao što je datoteka *index.js*. Pored toga, datoteka *App.js* uvozi još i logo i CSS datoteku koje su neophodne za funkcionisanje ove strane.

Ovo je drugi način struktuiranja koda. Naredbe *import* i *export* olakšavaju upravljanje kodom aplikacije. Umesto da sve bude definisano u jednoj velikoj datoteci, aplikacija je podeljena na više funkcionalnih delova. Zavisno od redosleda učitavanja datoteka, postupak izrade može da optimizuje konačan rezultat na načine o kojima onaj ko piše aplikaciju ne mora da brine. Prilikom testiranja aplikacije, dešava se korak prevođenja u izvršnu verziju. Tom prilikom pregledač obrađuje datoteke i komponente koje se uvoze i prevodi ih u jednu datoteku sa svim bitnim delovima (slika 16). Više o ovom načinu podešavanja okruženja se može naći u [1].



Slika 16 – Pregledač obrađuje datoteke i komponente u jednu datoteku

13.1 Kreiranje aplikacije *ZdravoSvete*

Cilj aplikacije je da ispiše reči *Zdravo, svete!* na ekranu. Ono na šta se treba fokusirati je struktuiranje datoteka u projektu. Za početak, potrebno je u direktorijumu *src* obrisati sve datoteke, a potom napraviti novu datoteku sa imenom *index.js*. U tu datoteku uneti sadržaj koji je dat u kodu 39.

Kôd 39 – Sadržaj datoteke *index.js* za kreiranje aplikacije *ZdravoSvete*

```
import React from "react";
import ReactDOM from "react-dom";
import ZdravoSvete from "./ZdravoSvete";

ReactDOM.render(
  <ZdravoSvete/>,
  document.getElementById("root")
);
```

Moduli koji se uvoze su *React* i *ReactDOM*. Pored toga, uvozi se i komponenta *ZdravoSvete* koja se navodi u pozivu funkcije *ReactDOM.render*. U istom direktorijumu *src*, potrebno je kreirati datoteku *ZdravoSvete.js*, a u okviru nje komponentu *ZdravoSvete* (kôd 40).

Kôd 40 – Sadržaj datoteke *ZdravoSvete.js*

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


class ZdravoSvete extends Component {
  render(){
    return (
      <h1>Zdravo, svete! </h1>
    )
  }
}


export default ZdravoSvete;
```

Kôd 40 prikazuje datoteku *ZdravoSvete.js* koja sadrži dodatnu naredbu *import*, komponentu *ZdravoSvete* koja ispisuje određeni tekst na ekranu i kôd koji označava naredbu za izvoz komponente, kako bi modul kao što je *index.js*, mogao da joj pristupi.

Nakon što se sačuvaju sve izmene, može da se testira aplikacija. U komandnoj liniji se unese naredba

npm start.

Ukoliko je aplikacija već pokrenuta, ona se automatski ažurira u skladu sa poslednjim izmenama. Ukoliko se aplikacija zaustavila, kombinacijom tastera

CTRL + C

potrebno je zaustaviti proces i ponovo uneti naredbu

npm start.

Aplikaciji *ZdravoSvete* se mogu dodati određeni stilovi. To se postiže tako što se u direktorijumu *src* kreira datoteka pod nazivom *index.css* i u okviru nje navedu pravila za stilizovanje (kôd 41).

Kôd 41 – Stilizovanje odgovarajućeg ispisa aplikacije

```
body{
  display: flex;
  align-items: center;
  justify-content: center;
  min-height: 100vh;
  margin: 0;
}
h1{
  font-family: sans-serif;
  font-size: 36px;
  background-color:aliceblue;
}
```

Pored pravljenja opisa stilova, potrebno je u datoteci *index.js* navesti referencu na datoteku za stilizovanje. To se postiže tako što se u datoteci *index.js* doda odgovarajuća naredba za uvoz (kôd 42).

Kôd 42 – Dodavanje naredbe za uvoz stilova u datoteci *index.js*

```
import React from "react";
import ReactDOM from "react-dom";
import ZdravoSvete from "./ZdravoSvete";
import "./index.css"

ReactDOM.render(
  <ZdravoSvete/>,
  document.getElementById("root")
);
```

Nakon ovih izmena, veb strana će izgledati kao na slici 17.

Slika 17 – Sadržaj veb strane nakon pokretanja aplikacije ZdravoSvete

13.1.2 Pravljenje produkcione verzije

Do sada je razvoj aplikacije *ZdravoSvete* bio u razvojnom režimu. U tom režimu kôd nije minimizovan i neke stvari se pokreću u sporoj postavci. Ukoliko aplikaciju treba poslati korisniku, traži se najbrže i najkompaktinije rešenje. Zbog toga, nakon što se izvrši izgradnja aplikacije, u komandnoj liniji se unosi naredba

```
npm run build.
```

Toj naredbi je potrebno nekoliko minuta da napravi optimizovani skup datoteka. Posle toga se može izabrati opcija da se aplikacija pošalje na server ili samo da se lokalno testira pomoću odgovarajućeg Node.js paketa. Generisana JavaScript datoteka sadrži svu logiku koja je aplikaciji neophodna za rad.

14 Virtuelni DOM

Prema strukturi HTML dokumenta objektni model dokumenta ima oblik stabla. Kada se dinamički menja sadržaj veb stranice, menja se DOM. Pošto je menjanje DOM-a dosta sporo, radni okvir ReactJS nikada direktno ne menja isti. Umesto toga, koristi virtuelni DOM u memoriji. Na svaku promenu nad podacima, ReactJS stvara novi virtuelni DOM. Taj proces je brz i ne utiče na performanse.

Većina radnih okvira za skript jezik JavaScript ažurira DOM više puta nego što je zaista potrebno, čime se smanjuje efikasnost. To se javlja kao posledica kompleksnih manipulacija DOM-a usled promene stanja aplikacije, budući da se njen sadržaj prikazuje dinamički. Prednost radnog okvira ReactJS u odnosu na ostale okvire za skript jezik JavaScript je to što koristi virtuelni DOM, pri čemu vodi računa o ažuriranju pravog DOM-a kada je potrebno. On poredi izmene u virtuelnom DOM-u sa pravim, praveći najmanji broj potrebnih izmena kako bi sve bilo ažurno. Virtuelni DOM je ustvari kopija pravog DOM-a. ReactJS upravlja sa dva virtuelna DOM-a, jedan DOM predstavlja trenutno stanje, a drugi DOM predstavlja pređašnje stanje - odnosno stanje pre ažuriranja. Za upoređivanje najmanjeg broja razlika između instanci dva virtuelna DOM-a koristi se algoritam *diff*. Složenost ovog algoritma je linearna, što znači da se pravi DOM ažurira u najmanjem broju koraka. Naime, kada se komponenta inicijalizuje, koristi se *render* metoda koja generiše njen prikaz. Kada se sledeći put pozove ista metoda, ReactJS upoređuje trenutni prikaz sa prethodnim i generiše minimalni set promena na pravom DOM-u. Taj proces se naziva usklađivanje podataka (eng. *reconciliation*). Zbog toga se prednosti ovog radnog okvira najviše ističu kada se podaci koje prikazuju menjaju tokom vremena.

Cena implementacije virtuelnog DOM-a je veličina aplikacije. Postoje alati koji je mogu minimizovati, a najveću upotrebu ima Redux.[12] Uglavnom se u programiranju složenih aplikacija radni okvir ReactJS koristi za oblikovanje korisničkog okruženja, a biblioteka Redux za upravljanje stanjem aplikacije. Vezu među njima predstavlja paket react - redux.

[12] Biblioteka skript jezika JavaScript za upravljanje stanjem aplikacije sa minimalnim API-jem (eng. *Application Programming Interface*).

Zaključak

U današnje vreme postoji mnogo radnih okvira za skript jezik JavaScript koji u velikoj meri olakšavaju razvoj veb aplikacija. Izdvajanje najboljeg od njih se ne može generalizovati. Analiziraju se svojstva radnog okvira koja su za aplikaciju bitna, kao i ona koja se mogu zanemariti i shodno tome se bira radni okvir koji bi obezbedio najbolje performanse.

Jedan od najvećih kvaliteta radnog okvira ReactJS su brze DOM performanse. Pored toga, ovaj radni okvir je uneo dosta promena u razvoju veb aplikacija, poboljšavajući kvalitet koda čineći ga pogodnim za ponovnu upotrebu. Pošto je struktuiran kao skupina manjih komponenti, pri čemu treba nastojati da svaka od njih bude zadužena za određeni deo funkcionalnosti, radni okvir ReactJS doprinosi boljem struktuiranju koda. Glavni izazov u radu sa ovim radnim okvirom predstavlja upravljanje stanjem aplikacije. Svaki put kada se stanje promeni ili kada se prenese svojstvo, sve komponente koje su pod njegovim uticajem biće ponovo iscrtane. To dovodi do toga da se pojedine komponente nepotrebno iscrtavaju iako samo prenose podatke sa roditelja na potomka. Prilikom razvoja složenih aplikacija, zbog složenosti i veličine samog koda radni okvir ReactJS se koristi u kombinaciji sa bibliotekom Redux, pri čemu se Redux koristi za upravljanje stanjem aplikacije, a ReactJS za kreiranje korisničkog okruženja. Načini njihove upotrebe i uzajamnog funkcionisanja bi mogli da budu predmet daljeg istraživanja.

Elektronske lekcije o radnom okviru ReactJS, koje su deo platforme *e-Skola veba*, imaju za cilj da čitaoca osposobe za efikasno kreiranje veb stranica i jednostavnih aplikacija, da prikažu mogućnosti koje pruža ovaj radni okvir i olakšaju proces njegovog usvajanja. Interaktivnost sadržaja i mogućnost pokretanja koda iz lekcija doprinose iskoristivom potencijalu samih lekcija. Pored toga što sadrže mnoštvo primera koji olakšavaju usvajanje znanja, zadaci na kraju lekcije omogućavaju korisnicima da provere naučeno znanje. Elektronske lekcije imaju za cilj da doprinesu kvalitetu znanja kroz razvoj prakse koja vodi ka efikasnom rešavanju problema. Mogućnost kreiranja dinamičkog korisničkog okruženja pogodnog za održavanje čini ReactJS sve popularnijim izborom kada je odabir radnog okruženja u pitanju, a elektronske lekcije daju sistematičan pristup za njegovo izučavanje.

Literatura

[1] Kirupa Chinnathambi, Learning React, November 2016.

[2] Anthony Accomazzo, Ari Lerner, Nate Murray, Clay Allsopp, David Guttman, Tyler McGinnis, Fullstack React, The Complete Guide to ReactJS and Friends, Fullstack.io, San Francisco, 2017

[3] N. Jurić, M. Marić, ”e-Skola veba”, VII Simpozijum Matematika i primene, Matematički fakultet, Beograd, 5. novembar 2016.

[4] Sajt W3 Schools: https://www.w3schools.com/react/, pristupano jun 2021.

[5] Sajt radnog okvira ReactJS: https://reactjs.org/, pristupano jun 2021.

[6] Sajt Education Ecosystem: https://www.education-ecosystem.com/guides/programming/react-js/history, pristupano jun 2021.

[7] Blog Esteban Herrera: https://blog.logrocket.com/a-guide-to-usestate-in-react-ecb9952e406c/, pristupano jun 2021.

[8] Blog Dmitrija Pavlutina: https://dmitripavlutin.com/react-useeffect-explanation/, pristupano jun 2021.

[9] Sajt codeBlog: https://www.codeblog.rs/clanci.php?p=javascript_es6_sintaksa, pristupano jun 2021.