ReactJS

Škalac, Lorena

Sveučilište J.J. Strossmayera u Osijeku
Odjel za matematiku
Diplomski studij: Matematika i računarstvo

Lorena Škalac

ReactJS

Diplomski rad

Osijek, 2017.

Sveučilište J.J. Strossmayera u Osijeku
Odjel za matematiku
Diplomski smjer: Matematika i računarstvo

Lorena Škalac

ReactJS

Diplomski rad

Mentor: izv. prof. dr. sc. Domagoj Matijević

Osijek, 2017.

Sadržaj

Sažetak

Cilj ovog rada je čitatelja upoznati s osnovama sintakse ReactJS-a, alata i praksa potrebnih za izradu funkcionalne React aplikacije. Navode se prednosti koje React donosi kao što su korištenje virtualnog DOM-a u svrhu ažuriranja prikaza aplikacije, podijela aplikacije na manje ponovne upotrebljive dijelove koje se nazivaju komponente te automatsko renderiranje koje se događa na svaku promjenu podataka. Budući da su komponente srž svake React aplikacije, veliki dio govori o njihovoj strukturi, toku podataka među njima koje se izvršava pomoću posebnih parametara komponente zvanih *props* te posebnom *state* objektu komponente, mjestu koje sadrži promijenjive podatke komponente. Kako se spomenuta automatska renderiranja ne bi događala više nego što je potrebno te time narušila performanse aplikacije, rad također uključuje kriterije koji će pomoći u strukturiranju optimalne aplikacije. Rad završava praktičnim dijelom, jednostavnom Planner aplikacijom za praćenje zadataka i zapisivanje bilježaka koja će ukratko, na stvarnom primjeru, prikazati odrađivanje određenih akcija unutar podstabla React komponenti.

Ključne riječi: komponenta, props parametri, state objekt, renderiranje, virtualan DOM

Summary

Aim of this paper is to introduce the reader with basics of ReactJS syntax, tools and practices needed to make a functional React application. Some of Reacts advantages are listed, for example use of virtual DOM that is used to update application view, application partitioning to small reusable parts that are called components and auto re-rendering which is run on every data alteration. Since components are the very core of every React application, big part of this paper talks about their structure and data flow between them which is driven by special component parameters called *props* and special *state* object of the component data. In case the aforementioned auto-renderings wouldn't happen more than it's needed, lowering the performance on the way, paper also includes criterion which will help to structure an optimal application. Paper concludes with the practical part, a simple Planner application that can track tasks and mark notes, which will in short present, on the real example, execution of specific action inside React components subtree.

Keywords: component, props parameters, state object, rendering, virtual DOM

Uvod

Današnje web aplikacije su većinom interaktivne pa programiranje njihovih korisničkih sučelja može biti dosta težak zadatak. U svrhu izvršavanja korisnikovih zahtjeva, izazvanih nekom od interakcija, koriste se kombinacije svojstava i funkcija čistog JavaScript jezika, JavaScript framework-a i biblioteka. Jedna od često korištenih biblioteka u te svrhe je jQuery. Međutim, kako aplikacije rastu te korisnikovi zahtjevi postaju složeniji, ne zvuči baš idealno da se na svaki korisnikov zahtjev elementi DOM-a manipuliraju ručno, primjerice na svako dodavanje elemenata u niz, koji je na sučelju prikazan u obliku tablice, potrebno je i dodati DOM element koji predstavlja redak tablice, kao što to zahtjeva jQuery. Budući da je takav kôd vrlo često vezan samo za određeni zahtjev, kao za prošli primjer specifičan broj stupaca, klase i slično, teško ga je ponovno upotrijebiti te na kraju uopće i pratiti. Ovakve situcije su ujedno i najveći izvor grešaka web aplikacija, odnosno pojavljuju se u pokušaju sinkronizacije modela podataka sa DOM-om.

Kako bi se dani problemi riješili pomoću jednog alata, izrađen je ReactJS koji osigurava da svaka promjena nad podacima uzrokuje ponovno renderiranje prikaza aplikacije. Takvim će pristupom aplikacija korisniku u svakom trenutku prikazivati njeno stvarno stanje.

Osim dobrobiti koje donosi aplikaciji, učenje i korištenje React-a će i svakog programera učiniti uspješnijim. Naime, zbog React-ovog poticanja na korištenje svojstava koje dolaze iz najnovijih verzija JavaScript-a te koncepta funkcijskog programiranja, programer će postati bolji u korištenju JavaScript programskog jezika. Također, budući da je strukturiran kao skupina manjih komponenti koje se nastoje kreirati tako da budu odgovorne za jedan dio funkcionalnosti, React će programera podučiti boljem strukturiranju aplikacija.

1 O React-u

React su kreirali inženjeri Facebook-a u svrhu dobivanja aplikacije koja ima sljedeća dva bitna svojstva:

- **Skalabilnost** - sposobnost sustava da nastavi dobro funkcionirati neovisno o promjeni njegove veličine.
- **Održivost** - sposobnost sustava da se nakon doživljenog pada lako i brzo vrati u operativno stanje.

Objavljen je 2013. godine kao JavaScript biblioteka za kreiranje korisničkih sučelja.

1.1 Zašto biblioteka, a ne framework?

Iako ima mnoge odlike framework-a mnogi su protiv mišljenja da je React kompletan framework. Službena stranica ga navodi kao JavaScript biblioteku za izradu korisničkih sučelja. Kako bi se pobliže shvatio razlog donošenja ove odluke, potrebno je opisati ključne razlike između framework-a i biblioteke.

JavaScript framework je arhitekturalni pattern pisan u JavaScript programskom jeziku koji definira dizajn aplikacije. To znači da će se pri izradi aplikacije koja koristi framework nužno morati pratiti struktura koju on specificira. Neki od poznatih JavaScript framework-a su:

- AngularJS
- Knockout.js
- Ember.js
- Vue.js

JavaScript biblioteka je skup funkcija pisanih u JavaScript programskom jeziku čije korištenje olakšava izradu JavaScript aplikacija. Može ih napisati svatko, objediniti u paket te dati drugima na korištenje. Obično zamjenjuju dijelove kôda koji se često ponavljaju. Neki od poznatih JavaScript biblioteka su:

- jQuery

- D3.js
- Moment.js
- Underscore.js

Programeri pišu kôd za framework kako bi izradio aplikaciju, dok biblioteku koriste kako bi napisala kôd kojim će moći izraditi aplikaciju. To je ujedno i ključna razlika između framework-a i bilioteke.

Facebook inženjeri su pri kreiranju ReactJS-a izabrali korištenje drugačijeg pristupa od onog koji se koristi u široko poznatom AngularJS-u i ostalim MVC frameworcima. Primjerice u Angular-u 1, izrađivanje aplikacije se sastoji od izrade HTML templatea, kontrolera koji njima upravljaju, servisa koji sadrže logiku koja se može koristiti među kontrolerima i direktiva koje predstavljaju ponovno upotrebljive dijelove kôda. Koristeći samo Angular 1 i prateći njegovu strukturu moguće je izraditi cijelu funkcionalnu aplikaciju bez potrebe za pomoćnim bibliotekama. S druge strane, React nema posebne strukture. Fokusiran je samo na view sloj aplikacije te će zahtijevati korištenje ostalih biblioteka kako bi se mogla izraditi funkcionalna React aplikacija. To znači da je React jedna velika JavaScript biblioteka te se kao takva može koristiti u kombinacijama s ostalim bibliotekama i frameworcima.

Iako nije jedan od njih, često se može pronaći u usporedbama performansi različitih JavaScript framework-a. Iz različitih mjerenja koja su provedena u te svrhe pokazalo se da je React u rangu s najpoznatijima od njih od kojih neke čak i nadmašuje. Sljedeća slika, dobivena primjenom filtera na stranici:
`http://www.stefankrause.net/js-frameworks-benchmark6/webdriver-ts-results/table.html`,
prikazuje rezultate mjerenja renderiranja view-a nakon izvedbe određenih akcija u četiri najpoznatija framework-a te React-u. Rezultati su izraženi u milisekundama.

Name	angular-v4.1.2-keyed	ember-v2.13.0	knockout-v3.4.1	react-v15.5.4-keyed	vue-v2.3.3-keyed
create rows Duration for creating 1000 rows after the page loaded.	193.1 ± 7.9 (1.2)	344.6 ± 13.8 (2.1)	358.2 ± 9.0 (2.1)	188.9 ± 10.9 (1.1)	166.7 ± 8.6 (1.0)
replace all rows Duration for updating all 1000 rows of the table (with 5 warmup iterations).	197.4 ± 5.3 (1.2)	292.7 ± 12.1 (1.7)	369.7 ± 19.6 (2.2)	201.0 ± 6.4 (1.2)	168.5 ± 5.0 (1.0)
partial update Time to update the text of every 10th row (with 5 warmup iterations).	13.0 ± 4.5 (1.0)	17.1 ± 1.7 (1.1)	14.0 ± 4.5 (1.0)	16.5 ± 2.3 (1.0)	17.3 ± 2.9 (1.1)
select row Duration to highlight a row in response to a click on the row. (with 5 warmup iterations).	3.4 ± 2.3 (1.0)	8.6 ± 2.4 (1.0)	11.4 ± 2.6 (1.0)	8.8 ± 3.4 (1.0)	9.3 ± 1.7 (1.0)
swap rows Time to swap 2 rows on a 1K table. (with 5 warmup iterations).	13.4 ± 1.0 (1.0)	16.4 ± 1.5 (1.0)	14.7 ± 1.2 (1.0)	14.7 ± 0.9 (1.0)	18.3 ± 1.5 (1.1)
remove row Duration to remove a row. (with 5 warmup iterations).	46.1 ± 3.2 (1.0)	49.1 ± 3.2 (1.1)	48.3 ± 2.3 (1.0)	47.2 ± 3.2 (1.0)	52.6 ± 2.7 (1.1)
create many rows Duration to create 10,000 rows	1946.0 ± 41.8 (1.2)	2569.3 ± 56.3 (1.6)	3485.6 ± 114.0 (2.2)	1852.4 ± 29.0 (1.2)	1587.5 ± 33.9 (1.0)
append rows to large table Duration for adding 1000 rows on a table of 10,000 rows.	324.6 ± 10.1 (1.0)	524.2 ± 23.6 (1.6)	5127.3 ± 132.0 (15.8)	345.6 ± 10.4 (1.1)	399.5 ± 11.0 (1.2)
clear rows Duration to clear the table filled with 10.000 rows.	379.9 ± 11.3 (1.5)	303.9 ± 74.7 (1.2)	579.6 ± 47.5 (2.3)	398.4 ± 8.2 (1.6)	254.5 ± 5.0 (1.0)
startup time Time for loading, parsing and starting up	84.3 ± 2.6 (1.5)	245.2 ± 5.6 (4.3)	60.3 ± 2.5 (1.1)	70.0 ± 2.9 (1.2)	56.6 ± 2.5 (1.0)
slowdown geometric mean	1.14	1.50	1.83	1.13	1.06

Slika 1: Vrijeme izvršavanja akcija u određenim frameworcima izraženo u milisekundama

Uz to što ima veliku brzinu renderiranja, iz slike se može vidjeti da ona ne ovisi o veličini podataka na kojemu se izvodi. Naime, mjerenje navodi da geometrijska sredina usporavanja React-a, obzirom na veličinu podataka, iznosi 1.13 što znači da je Facebook-ova namjera za dobivanje skalabilne biblioteke bila uspješna. Razlog tome je što React, pri ažuriranju DOM-a, ažurira samo ono što se stvarno promijenilo te pritom koristi virtualan DOM koji posjeduje znatno jeftinije operacije od ekvivalntnih na pravom DOM-u. Kako ga je moguće koristiti u kombinaciji sa drugim frameworcima, poželjno ga je koristiti kao view sloj aplikacija koja koriste framework-e sa slabijim performansama na tom području.

2 EcmaScript 6 / EcmaScript 7

Kao što je u prethodnom poglavlju zaključeno, ReactJS je biblioteka. Pisana je u Javascript skriptnom programskom jeziku kojeg je 1995. godine kreirao tadašnji inženjer Netscape-a Brendan Eich. Službeno je objavljen 1996. godine zajedno s Netscape-om 2, web preglednikom tvrtke Netscape. Originalan naziv mu je bio LiveScript no ubrzo je preimenovan u JavaScript u svrhu stjecanja popularnosti kroz tadašnju popularnost Java programskog jezika. Iz tog razloga ljudi miješaju ta dva jezika iako imaju malo toga zajedničkog.

Danas je JavaScript najčešće korišten jezik u izradi web aplikacija te je uz HTML i CSS jedna od tri jezgrene tehnologije većine web aplikacija. Izvršava se na klijentskoj strani aplikacije u svrhu definiranja kako će se ona ponašati u trenutku pojave određenog događaja. Potrebno je naglasiti da se JavaScript koristi i u okolinama koje ne uključuju pojavu web preglednika kao što je primjerice serverski orjentirano okruženje Node.js. Posjedovanje znanja o ovom jeziku postala je bitna i neizbježna vještina svakog web ili mobile programera.

Ovakvo svjetsko prihvaćanje JavaScript-a, kao primarnog programskog jezika klijentske strane web aplikacija, dovelo je do formacije njegovog standarda, odnosno specifikacije koja je nazvana Ecmascript ili skraćeno ES. Trenutno aktualno izdanje je EcmaScript 7 na koje se može gledati kao na sedmu verziju JavaScript programskog jezika.

EcmaScript 6 standard je doživio značajno proširenje sa čak oko 20-ak skupina novih svojstava dok je za usporedbu EcmaScript 7 imao tek dva. React potiče korištenje svojstva najnovijih verzija JavaScript-a stoga će u radu ona biti često korištena. Potrebno je naglasiti da, iako je aktualan EcmaScript 7 standard, kada bude riječ o novim svojstvima u radu referirat će se na EcmaScript 6 svojstva jer većina njih dolazi iz tog standarda.

2.1 Kratki osvrt na EcmaScript 6 svojstva korištenih u radu

Class

JavaScript je jezik u kojemu ne postoje klase kao što je to inače u ostalim objektno-

orjentiranim jezicima. Kada je riječ o nasljeđivanju, u JavaScript-u objekt nasljeđuje objekt korištenjem sintakse bazirane na *prototyp* nasljeđivanju. Izlaskom ES6 standarda uvodi se riječ *class* u JavaScript jezik. Nova *class* sintaksa ne uvodi novi model nasljeđivanja već samo omogućuje jednostavniju i intuitivniju sintaksu za kreiranje objekta (kasnije u tekstu klasa) i proces postojećeg nasljeđivanja. Klase se, kao i u većini ostalih objektno orjentiranih programskih jezika, kreiraju pomoću riječi *class*. Svaka na ovaj način kreirana klasa treba imati metodu *constructor()* koja se poziva u trenutku kada se instancira novi objekt te klase koristeći sintaksu *new className()*. Proširenja klasa su moguća korištenjem sintakse *Child extends Parent*.
U React-u, ovo svojstvo služi za kreiranje komponenti, neovisnih i ponovno upotrebljivih dijelova kôda koji su srž svake React aplikacije.

Const & Let

Obje nove ključne riječi *let* i *const* služe za deklariranje varijabla.
Const služi za deklariranje konstanti, odnosno nepromjenjivih varijabla kojima se kasnije ne želi dodijeliti nova vrijednost. Deklarira li se konstanta na ovaj način te joj se zatim pokuša promijeniti vrijednost, program će upozoriti na pogrešku.

```
1.  const pi = 3.14;
2.  pi = 3.14159265359; // Assignment to constant variable
```

U slučaju kada varijabla može mijenjati vrijednost, za njeno deklariranje se koristi riječ *let*. Ovakvo deklariranje varijable ima slična svojstva kao pomoću standardne riječi *var*. Razlika je u tome što se deklariranjem pomoću riječi *let* područje postojanja varijable ograničava na blok kôda u kojem je deklarirana čime se mogu izbjeći neželjene pogreške u aplikaciji.

```
1.  let x = 0;
2.  if (true){
3.     var y = 1;
4.     let z = 2;
5.  }
6.  console.log(x); // 0
7.  console.log(y); // 1
8.  console.log(z); // z is not defined
```

Arrow funkcije

ES6 uvodi novu sintaksu za pisanje anonimnih funkcija korištenjem arrow funkcija. Dobile su naziv po ⇒ znaku koji pruža sažetost definicije funkcije. Arrow funkcije u JavaScript-u su zapravo funkcije koje su u drugim programskim jezicima poznate pod nazivom lambda funkcije. Njihova prednost u React-u, osim sažetosti, je u u načinu vezanja *this* objekta unutar funkcija. One nemaju *this* objekt te zbog toga koriste *this* objekt parent objekta. U React-u, korištenje arrow funkcija zbog toga će automatski vezati objekt *this*, unutar funkcija koje ne dolaze iz React biblioteka, na komeponentu unutar koje je funkcija definirana.
Sintaksa definiranja arrow funkcija je takva da se prije znaka ⇒ stavljaju parametri fukcije, a iza njega tijelo funkcije.

```
1.  var sum = (a,b) => {
2.     return a + b;
3.  };
4.  var sum2 = (a,b) => (a + b);
5.
6.  console.log(sum(1,2)); // 3
7.  console.log(sum2(1,2)); // 3
```

Ukoliko u tijelu funkcije postoji samo return izraz, definicija funkcije se može dodatno sažeti tako da se riječ return ispusti te se rezultat koji želi biti vraćen stavi u okrugle zagrade iza znaka ⇒.

Object.assign()

Metoda *Object.assign()* se koristi za kopiranje vrijednosti svojstava s jednog ili više izvora objekta na ciljani objekt. Prima bilo koji broj argumenata s naznakom da prvi referencira ciljani objekt dok ostali izvorne objekte. Metoda kao rezultat vraća ciljani objekt.

```
1.  const source = {
2.    name: "Test",
3.    value: 0
4.  };
5.
6.  var a = Object.assign({}, source, {
7.    value: 1
8.  });
9.
```

```
10. console.log(a);
11. // Object {
12. //   "name": "Test",
13. //   "value": 1
14. //}
```

U prethodnom primjeru, prvi objekt pridružen funkciji *Object.assign()* predstavlja novi objekt kojeg će funkcija vratiti, drugi predstavlja objekt s kojega će kopirati svojstva na prvi novostvoreni objekt te treći predstavlja objekt koji će ažurirati novostvoreni objekt s kopiranim vrijednostima drugog objekta.
Kao rezultat dobiven je objekt koji je jednak izvornom objektu s ažuriranom vrijednošću svojstava.
Prethodno je spomenuto da se varijablama koje su definirane sa riječi *const* ne može ponovno dodijeliti vrijednosti pa je ova metoda prikladna kada se radi s konstantnim podacima.

3 Komponente

Srž svake React aplikacije čine komponente. One su samostalne i ponovno upotrebljive cjeline kôda koje sadrže markup, logiku te stil na jednom mjestu. Na njih se može gledati kao na funkcije jer primaju ulazne *props* parametre te kao izlaz vraćaju React element koji opisuje što se treba prikazati na zaslonu. U trenutku kada se ulazni parametar komponente promijeni, React ponovno renderira tu komponentu. To svojstvo osigurava da izgled komponente na zaslonu bude jednak onome što joj se proslijedi, odnosno da za dani ulaz uvijek daje isti izlaz.

Komponente treba izrađivati na način da je svaka odgovorna za samo jedan dio funkcionalnosti. Ukoliko se u kôdu naiđe na neku za koju vrijedi suprotno to sugerira da se ona može razbiti na više komponenti. Takva podjela odgovornosti učinit će komponente jednostavnima te omogućiti da budu pogodne za ponovnu upotrebu.

3.1 Načini definiranja komponenti

U React-u, komponente se mogu definirati na dva načina:

Kao EcmaScript klase

U ovom slučaju komponenta je EcmaScript klasa koja nasljeđuje React klasu *React.Component*. Sintaksa kreiranja komponente koristeći ovaj način je sljedeća:

```
1.  class MyComponent extends React.Component{
2.     render() {
3.        return (
4.            <p> Hello world! </p>
5.         );
6.      }
7.  }
```

Korištenjem React.createClass() metode

U ovom slučaju za kreiranje komponente koristi se ugrađena React metoda *createClass()*. Sintaksa kreiranja komponenata koristeći ovaj način je sljedeća:

```
1.  const MyComponent = React.createClass({
2.      render(){
```

```
3.          return(
4.              <p> Hello world! </p>
5.          );
6.      }
7.  });
```

Oba načina deklariranja komponenti su prihvatljiva te su razlike minimalne. Razlog postajanja dva načina je taj što je React nastao u vrijeme kada JavaScript nije imao ugrađen sustav za kreiranje klasa pa je uveo svoj *React.createClass*. S pojavom EcmaScript 6 standarda komponente je moguće kreirati i koristeći *class* sintaksu.

React metoda *render()* korištena u prethodnim primjerima je jedina nužna metoda React komponente. React koristi vraćenu vrijednost te metode kako bi zaključio što treba renderirati na zaslonu aplikacije.

3.2 JSX

U primjerima kreiranja komponenti može se uočiti da izraz koji metoda *render()* vraća ne izgleda kao tradicionalni JavaScript. Ona vraća dio koda koji je napisan pomoću JSX-a. Skraćenica JSX dolazi od izraza JavaScript Syntax Extenstion te se radi o ekstenziji sintakse JavaScript-a koju je napravio Facebook kako bi pisanje markupa komponenti bilo omogućeno u poznatoj HTML sintaksi. Sljedeći primjer prikazuje dio kôda za kreiranje React elementa pomoću JSX ekstenzije:

```
1.  <div className="red-box">
2.      <p> Hello world. </p>
3.   </div>
```

Korištenje JSX-a nije nužno te se kao alternativa može koristiti direktno čisti JavaScript. U tom slučaju za kreiranje React elemenata koristi se *React.createElement()* funkcija koja prima tri argumenta. Prvi argument označava tip elementa, drugi svojstva elementa te treći djecu elementa. Ekvivalentan React element iz prethodnog primjera bez korištenja JSX ekstenzije pisao bi se na sljedeći način:

```
1.  React.createElement('div', {className: 'red-box'},
2.      React.createElement('p', null, 'Hello world.')
3.  );
```

Način kreiranja React elemenata koristeći *React.createElement()* funkcije je pogodan

za male komponente no u situacijama kada je komponenta sastavljena od mnogo ugnježđenih elemenata, sintaksa može postati nečitljiva te se tada preporuča korištenje JSX ekstenzije.

Budući da preglednici ne znaju pročitati JSX, on je pretvoren u JavaScript pomoću dodatnih alata. Najćešće korišten alat za ovu pretvorbu je Babel.

3.2.1 Babel

Babel je JavaScript kompajler koji razumije JSX. On pretvara JSX sintaksu u Vanilla ES5 JavaScript koji će svaki preglednik moći interpretirati. Također, većina web preglednika ne podržava sva svojstva EcmaScript 6 standarda te se u svrhu njihove pretvorbe u ES5 sintaksu također koristi Babel.

Budući da će JavaScript konstantno evoluirati, alat kao Babel je poželjan u projektu jer će programeri uvijek moći iskoristiti prednosti koje im pružaju nova svojstva. U suprotnom će na to morati pričekati jer je web preglednicma potrebno duže vrijeme za ažuriranje strojeva koji omogućuju direktno korištenje novih svojstava jezika.

3.3 Dodavanje komponente na DOM

Nakon izrade React komponenti, potreban je način na koji će se one povezati sa vanjskim svijetom aplikacije. U tu svrhu koristi se *ReactDOM.render()* funkcija koja dolazi iz *react.dom* biblioteke.

Način kojim će se React-u reći da komponenta treba biti umetnuta na određeni dio DOM-a je ta da se *ReactDOM.render()* funkciji prosljede dva argumenta, prvi koji označava React element koji se želi renderirati te drugi koji govori na koje mjesto u DOM-u se prvi argument želi renderirati.
Pretpostavljajući da negdje u DOM-u aplikacije postoji element sa id atributom *myApp*, sintaksa renderiranja komponente *MyComponent* unutar tog elementa je sljedeća:

```
1.  ReactDOM.render(<MyComponent />,
2.  document.getElementById('myApp'));
```

Nakon ovog koraka komponenta će biti vidljina na prikazu aplikacije.

U primjeru se također vidi razlika u pisanju React elemenata i HTML tagova u JSX sintaksi. Naime, HTML tagovi započinju malim slovom dok React elementi započinju velikim slovom. Sintaksom *<MyComponent />* komponenta *MyComponent* je pretvorena u React element te kao takva prosljeđena *ReactDOM.render()* funkciji koja će ju znati umetnuti u pravi DOM.

4 Dinamičke komponente

Dinamičke komponente su komponente koje će se dinamički renderirati ovisno o podacima koji su im prosljeđeni.

4.1 Hijerarhija komponenti i tok podataka unutar nje

Među komponentama u React aplikacijama postoji hijerarhija pa postoje parent i child komponente. Kako bi neka komponenta B postala child komponenta komponenti A, potrebno ju je smjestiti u *render()* funkciju komponente A. Komponenta A tada postaje parent komponenta komponenti B.
Tok podataka unutar takve strukture uvijek ide od vrha prema dnu što znači da će svaka parent komponenta prosljeđivati podatke child komponentama. Takva struktura ujedno omogućuje parent komponenti da dinamički renderira bilo koji broj child komponenti od kojih će svaka imati svoje jedinstvene atribute. Prosljeđivanje podataka između komponenti omogućeno je pomoću parametara koji se nazivaju *props*.

4.1.1 Props parametri komponenti

U trenutku kada parent komponenta renderira child komponentu može joj proslijediti *props* parametre o kojima ona ovisi. Prosljeđivanje podataka se provodi na način da se na child komponentu, koja se nalazi unutar *render()* funkcije parent komponente, doda atribut te njemu pridružena vrijednost pri čemu će atribut označavati ime *prop* parametra. U sljedećem primjeru *ParentComponent* komponenta prosljeđuje *ChildComponent* komponenti *props* parametre *title* i *imageUrl*.

```
1.  class ParentComponent extends React.Component{
2.      const imageUrl = "someImage.jpg";
3.      render(){
4.          return(
5.              <ChildComponent
6.                  title = "Hello world!"
7.                  imageUrl = {imageUrl}
8.              />
9.          );
10.     }
11. }
```

Generalno, u JSX-u svaka vrijednost atributa mora biti odvojena:

- Ili vitičastim zagradama - govore JSX-u da se unutar njih nalazi JavaScript izraz. Izraz se prvo rješava te se vrijednost koju on vraća pridružuje atributu.
- Ili navodnicima - koriste se za stringove koji se direktno pridružuju atributu.

Svrha toga što parent komponenta šalje podatke child komponenti je ta da ih ona na neki način koristi. U Reactu, komponenta može pristupiti svim prosljeđenim *props* parametrima pomoću objekta *this.props*. *this.props* objekt iz prethodnog primjera, unutar *ChildComponent* komponente je sljedeći:

```
1.  this.props = {
2.    title: "Hellow world!",
3.    imageUrl: "someImage.jpg"
4.  }
```

Svojstva *this.props* objekta se mogu koristiti unutar child komponente kako bi renderirala dijelove sučelja koji ovise o njihovim vrijednostima. Koriste se na način da se na mjestu, gdje se žele umetnuti, zapišu unutar vitičastih zagrada. Primjerice, ako se vrijednosti svojstava *this.props* objekta komponente *ChildComponent* žele umetnuti kao sadržaj ili atribut nekog elementa, provodi se na sljedeći način:

```
1.  class ChildComponent extends React.Component{
2.      render(){
3.          return(
4.              <div> {this.props.title} </div>
5.              <img src={this.props.imageUrl} />
6.          );
7.      }
8.  }
```

Iz primjera je vidljivo da je sintaksa kojom se vrijednost *prop* parametra umeće kao sadržaj elementa jednaka sintaksi za pridruživanje vrijednosti atributu elementa.

This objekt unutar *render()* funkcije povezan je s klasom React komponente u kojoj se nalazi pa kada se pristupa *this.props* objektu unutar *render()* funkcije pristupa se *props* parametrima te komponente.

4.1.2 Renderiranje većeg broja child komponenti

Kako bi neka komponenta renderirala veći broj child komponenti, prvi korak je da generira niz tih komponenti i spremi ih u neku vrijablu. U tu svrhu se koristiti

map() funkcija.
map() funkcija prolazi nizom elemenata nad kojim je pozvana te funkciju, koju ona prima kao argument, poziva na svakom elementu tog niza. Tek kada završi s posljednjim elementom niza, *map()* funkcija vraća novi niz koji sadrži vraćene vrijednosti izvršene argument funkcije na svakom elementu niza.

U slučaju generiranja niza child komponenti, *map()* funkcija se poziva na nizu podataka koji opisuju child komponente te za svaki element tog niza prosljeđena funkcija kreira child komponentu. Krajnji rezultat ove operacije biti će novokreirani niz child komponenti čija veličina je jednaka broju elemenata niza podataka koji je opisivao te komponente.

Nakon generiranja niza child komponenti preostaje reći *render()* funkciji da vrati niz novonastalih komponenti na način koji je prikazan u sljedećem primjeru.

```
1.  class ComponentList extends React.Component{
2.      render(){
3.          const childComponents = childArray.map((child) => (
4.              <ChildComponent
5.                  key={child.id}
6.                  title={child.title}
7.                  imageUrl={child.imagUrl}
8.              />
9.          ));
10.         return(
11.             <div id="childComponents">
12.                 {childComponents}
13.             </div>
14.         );
15.     }
16. }
```

U primjeru je korišten specijalan *prop* parametar komponenti koji se naziva *key*. U slučaju renderiranje više child komponenti preporuča se njegovo korištenje jer ga React koristi kako bi kreirao jedinstvenu vezu za svaku instancu komponente. To će dodatno optimizirati algoritam pretraživanja koji React koristiti kako bi pronašao elemente koji se trebaju ažurirati. Kako vrijednost *key* parametra treba biti jedinstvena, za nju se najčešće koristi identifikator objekta.

5 Interaktivno React korisničko sučelje

Korisnička sučelja današnjih web aplikacija su većinom interaktivna. Ona omogućuju korisniku da putem neke vrste interaktivnog događaja pošalje zahtjev aplikaciji. Primjerice, na klik dugmeta za ažuriranje neke komponente korisnik očekuje da se otvori forma za ažuriranje.

U React-u, child komponente mogu čitati *props* parametre koji su im prosljeđeni, ali ih ne mogu modificirati. Razlog tome je što one nisu njihovi vlasnici već su to parent komponente. Za child komponente *props* parametri služe samo za čitanje te su s njihovog aspekta oni nepromjenjivi podaci. Takva struktura podržava React-ovu ideju o toku podataka u jednom smjeru što znači da ukoliko se želi napraviti neka promjena nad podacima da će se oni mijenjati od vrha hijerarhije te se nakon toga ponovno prosljeđivati prema nižim razinama.

Budući da se interaktivni događaj korisnika najčešće događa na nekoj od child komponenti, potreban je način na koji će ona signalizirati taj zahtjev parent komponenti, koja je vlasnik podataka nad kojima zahtjev želi izvršiti promjenu. U tu svrhu parent komponenta će putem *prop* parametra, komponenti na kojoj se izvodi zahtjev, uz podatke prosljediti i funkciju koja se nazivna *callback*. Takva funkcija ima samo jednu namjenu i to je komunikacija child komponente s parent komponentom. U trenutku kada korisnik pošalje zahtjev, komponenta na kojoj se on izvodi će pozvati *prop* parametar na kojem se nalazi *callback* funkcija. Poziv će teći sve do parent komponente unutar koje će *callback* funkcija pozvati funkciju unutar koje je definirana logika za odrađivanje željenog zahtjeva. Nakon što se ova funkcija izvrši te napravi željene modifikacije nad podacima, parent komponenta će ažurirane podatke ponovno proslijediti prema child komponentama.

Callback funkcija koja služi za komunikaciju child komponenti s parent komponentama je jedna od onih koje ne dolaze iz React biblioteka te se unutar takvih *this* objekt ne referencira na komponentu. Razlog tome je što funkcije koje ne dolaze iz React biblioteka imaju različit način vezanja *this* objekta unutar funkcije.

5.1 Vezanje objekta this unutar funkcija

U JavaScript-u specijalan *this* objekt ima različito vezanje ovisno o kontekstu. Primjerice, unutar *render()* funkcije *this* objekt referencira na komponentu u kojoj se nalazi dok unutar spomenute *callback* funkcije, koja služi za komuniciranje child komponente sa parent komponentom, referencira na *null* vrijednost. Razlog tome je što unutar svake funkcije iz React biblioteka, kao što je *render()*, React automatski veže *this* objekt unutar funkcije na komponentu. Za svaku drugu funkciju vezanje objekta *this* je potrebno izvesti manualno. To se može postići na dva načina:

Unutar konstruktora

Constructor() je jedinstvena funkcija JavaScript klase koja se poziva svaki put kada se radi nova instanca te klase. Budući da je svaka komponenta React klasa, prvo što će React napraviti pri incijaliziranju komponente je to da će pozvati konstruktor skupa sa *props* parametrima komponente. Zbog tog svojstva upravo je konstruktor mjesto gdje se treba obaviti vezanje objekta *this* unutar funkcije na komponentu. Način na koji se to provodi je sljedeći:

```
1.  class MyComponent extends React.Component{
2.      constructor(props){
3.          super(props);
4.
5.          this.myFunction = this.myFunction.bind(this);
6.      }
7.  };
```

Koristeći paket Babel-a i arrow funkcije

Standardna sintaksa za definiranje anonimnih funkcija će vezati objekt *this*, koji se koristi unutar funkcije, na globalan objekt. Ukoliko se za definiranje anonimnih funkcija koriste arrow funkcije uz uključen *transfrom-class-properties* Babel paket, one će vezati isti taj objekt unutar zatvorenog konteksta, u ovom slučaju na komponentu. U tom slučaju objekt *this* nije potrebno vezati manualno što uvelike pojednostavljuje posao.

Uvođenjem *callback* funkcija omogućeno je slanje zahtjeva za modifikacijom nad podacima od child komponente do parent komponente koja je vlasnik tih podataka.

Kako bi se ta modifikacija i izvršila, unutar komponente je potrebno mjesto na kojem će se to odvijati. U React-u, to je objekt koji se naziva *state*.

5.2 Korištenje state-a

U React-u postoje dva tipa podataka o kojima komponenta ovisi. To su *props* i *state* objekti. *Props* objekt prosljeđen je od parent komponente te je za child komponentu nepromjenjiv kroz cijeli njen životni ciklus. Za podatke koji će se mijenjati koristi se *state* objekt. On je privatan za komponentu te se može ažurirati pomoću funkcije *setState()*.

Budući da je svaka komponenta renderirana kao funkcija ovisna o *this.props* i *this.state* objektima, state ima važno svojstvo u React-u jer će se na svako njegovo ažuriranje komponenta ponovno renderirati. Renderiranje komponenti je deterministički proces jer će za dani skup svojstava this.props i this.state objekta komponenti, komponenta uvijek biti renderirana na isti način. Ovaj pristup omogućava konzistenciju korisničkog sučelja sa stvarnim stanjem podataka.

Radi performansi te jednostavnosti aplikacije, *state* objekt komponente treba biti prezentiran sa što manje izvora podataka. U tu svrhu može se koristiti sljedeći kriterij koji će dati odgovor na to treba li promatrani podatak biti sadržan na state objektu.

5.2.1 Kriterij za donošenje odluke treba li podatak biti sadržan na state objektu

- Je li promatrani podatak prosljeđen od parent komponente putem *props* parametara? Ako je odgovor pozitivan, vjerojatno nije sadržan na *state* objektu.

- Mijenja li se promatrani podatak s vremenom? Ako je odgovor negativan, vjerojatno nije sadržan na *state* objektu.

- Može li se promatrani podatak izračunati na temelju nekih drugih podataka sadržanih na *state* objektu ili na temelju *props* parametara u danoj komponenti? Ako je odgovor pozitivan, nije sadržan na *state* objektu.

Nakon zaključivanja koji podaci trebaju biti sadržani na *state* objektu, potrebno je zaključiti u kojim komponentama oni trebaju živjeti. Stoga za svaki podatak *state* objekta, treba izvršiti sljedeće:

1. Identificirati sve komponente koje renderiraju kôd koji ovisi o *state* objektu.
2. Pronaći zajedničku vlasnik komponentu, odnosno komponentu iznad svih komponenti koje trebaju *state* objekt u hijerahiji.
3. *State* objekt treba živjeti na zajedničkoj vlasnik komponenti. Ukoliko iz nekog razloga nema smisla da *state* objekt živi u toj komponenti, *state* objekt treba živjeti u novoj komponenti koja će se dodati u hijerarhiju iznad zajedničke vlasnik komponente. Njena namjena će naprosto biti posjedovanje *state* objekta.

5.2.2 Popunjavanje state objekta podacima

Praksa je uvijek postaviti inicijalno stanje *state* objekta radi izbjegavanja pogreške koju React javlja kada se *state* objekt još nije popunio podacima. Budući da je *constructor()* metoda komponente koja se prva poziva pri njenom kreiranju, predstavlja najbolje mjesto za definiranje inicijalnog stanja *state* objekta. Sljedeći primjer prikazuje inicijalno stanje *state* objekta s podatkom *data* koji predstavlja prazan niz podataka.

```
1.  constructor(props){
2.      super(props);
3.
4.      this.state = {
5.          data: []
6.      };
7.  };
```

Ukoliko se u komponenti ne koristi konstruktor u druge svrhe, uz dodatno uključen Babel paket *transform-class-properties*, inicijalno stanje *state* objekta može biti definirano na početku definicije komponente.

U ovom trenutku komponenta je sposobna koristiti podatke *state* objekta no budući da je jedini podatak na *state* objektu prazan niz, niti jedna komponenta na nižoj razini hijerarhije se neće renderirati. Podatke sadržane na *state* objektu je potrebno popuniti nakon što se komponenta montira na DOM. U tu svrhu koristi se React-ova

metoda *componentDidMount()* koje se poziva upravo u tom trenutku.

Jedino mjesto gdje se *state* objektu pridružuju podaci pomoću znaka jednakosti je prilikom definiranja inicijalnog stanja. Za ostale se modifikacije koristi *setState()* metoda.

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

Kronološki slijed događaja koji nastupaju od postavljanja inicijalnog *state* objekta do njegovog ažuriranja:

1. Komponenta se montira na stranicu sa podacima koji se nalaze na inicijalnom state objektu.
2. State objekt se puni podacima.
3. Komponenta se ponovno renderira sa popunjenim podacima.
4. Podaci se prikazuju na zaslonu.

Prethodni koraci se događaju u brzini koja je neprimjetna korisniku.

5.2.3 Ažuriranje nepromjenjivog state objekta

Unatoč tome što komponenta može ažurirati *state* objekt, preporuča se smatrati *state* objekt nepromjenjivim objektom. Kako bi se ipak udovoljilo korisnikovom zahtjevu za modifikacijom podataka, koji su sadržani na state objektu, koriste se JavaScript funkcije koje, umjesto da rade modifikacije na prosljeđenom objektu, vraćaju novi. Najčešće takve funkcije koje se koriste u svrhu provedbe CRUD operacija nad podacima su:

- **concat()** – metoda koja spaja dva ili više niza. Koristi se za dodovanje novih elemenata u niz.

```
1.  var immutableArray = [1,2,3]
2.  var newArray = immutableArray.concat([4]);
3.  console.log(immutableArray); // [1,2,3]
4.  console.log(newArray); // [1,2,3,4]
```

- **filter()** – metoda koja se provodi na nizu te kao argument prima funkciju koja služi kao test. Kao rezultat vraća novi niz koji sadrži elemente niza elemenata na kojoj je pozvana te koji su prošli prosljeđenu test funkciju. Koristi se za brisanje elemenata niza.

```
1.  var immutableArray = [1,2,3];
2.  var newArray = immutableArray.filter((x => x % 2 !== 0));
3.  console.log(immutableArray); // [1,2,3]
4.  console.log(newArray); // [1,3]
```

- **Object.assign()** – metoda objašnjena unutar poglavlja EcmaScript 6 svojstva. Koristi se za ažuriranje svojstava postojećih podataka.

6 Virtualni DOM

DOM je skraćenica za Document Object Model. Apstrakcija je strukturiranog teksta programerima znanog kao HTML kôd. Elementi HTML-a predstavljaju čvorove u DOM-u. Za jedan HTML moguće je imati više DOM objekata, primjerice kada je ista stranica učitana na više tabova web preglednika.

Prema strukturi HTML dokumenta, DOM je uvijek strukturiran kao stablo. Takva struktura omogućuje lak no ne i brz prijelaz po njemu. Stabla DOM-a današnjih aplikacija su ogromna te budući da je većina njih dinamička, manipulacija DOM-a čini njihovu srž. Uz to što je jedna od najskupljih operacija DOM-a, većina frameworka ažurira DOM više nego što je potrebno.

Primjerice, ukoliko se na listi od deset zadatka želi označiti jedan kao završen, većina frameworka će nakon toga ponovno izgraditi cijelu listu što je deset puta više nego potrebno. Na ovako malom primjeru to se i ne čini skupo no stvarna situacija današnjih web aplikacija je ta da koriste veliki broj kompleksnijih manipulacija ogromnog DOM-a što dovodi do drastičnog porasta broja nepotrebnih ažuriranja.

React rješava navedene probleme koristeći virtualan DOM na način da za svaki DOM objekt u React-u postoji njegova virtualna reprezentacija, odnosno pojednostavljena kopija.

Bazu virtualnog DOM-a čine React elementi, jednostavni objekti koji opisuju izgled DOM-a. React komponente nemaju pristup virtualnom DOM-u pa se prije umetanja u virtualan DOM trebaju pretvoriti u njih, ili sa

```
React.createElement(MyComponent);
```

ili ekvivalento pomoću JSX sintakse

```
<MyComponent />
```

Virtualan DOM ima ista svojstva kao i pravi DOM no nedostaje mu mogućnost direktne promjene stanja na zaslonu. Taj nedostatak u ovom slučaju postaje prednost jer čini manipulaciju pravim DOM-om mnogo bržom.

6.1 Proces ažuriranje DOM-a u React aplikacijama

Kronološki slijed akcija

1. U React-u, na svaku promjenu nad podacima kreira se novi virtualan DOM. To znači da će se virtualni DOM ponovno kreirati od nule na svaki poziv funkcije *setState()* na nekoj od komponenti. Taj proces je jako brz i ne utječe na performanse aplikacije.

2. U svakom trenutku, React rukuje dvjema instancama virtualnog DOM-a. Jedna predstavlja ažurirano stanje virtualnog DOM-a te druga njegovo prethodno stanje. React koristi algoritam koji se naziva *diff* za uspoređivanje instanci virtualnog DOM-a u svrhu pronalaženja najmanjeg broja razlika stanja prije i poslije ažuriranja. Složenost *diff* algoritma je linearna ili *O(n)* što znači da će njegova složenost rasti proporcionalno veličini inputa u oznaci *n*.

3. Nakon što je pomoću *diff* algoritma dobio podatak o promjenama koje su nastale, React ažurira samo te elemente na pravom DOM-a ne dirajući ostale.

Nakon provedenih koraka React bi u primjeru liste bio dovoljno pametan da ažurira samo jedan izmijenjen zadatak s liste.

Uz to što korištenje virtualnog DOM-a povećava porformanse ažuriranja DOM-a, nije potreban prevelik dodatan posao kada se ažuriranje želi postići. Naime, kao što je već spomenuto, ažuriranje virtualnog DOM-a će se dogoditi na svaku promjenu podataka, odnosno mjesta na kojima se oni drže a to je u React-u *state* objekt.

7 React u praksi

Ovaj rad je popraćen praktičnim dijelom, Planner aplikacijom čiji se kôd može pronaći na linku:
`https://github.com/lskalac/Diplomski-React-Asp.Net_WebApi`.

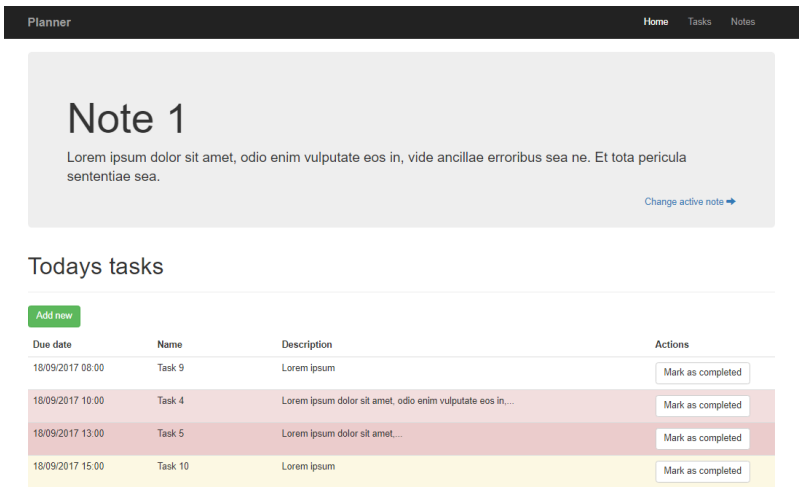
7.1 O aplikaciji

Aplikacija predstavlja jednostavan planer za praćenje zadataka i dodavanja bilješki koji sadrži tri glavne stranice.

HomePage

HomePage je početna stranica koja sadrži:

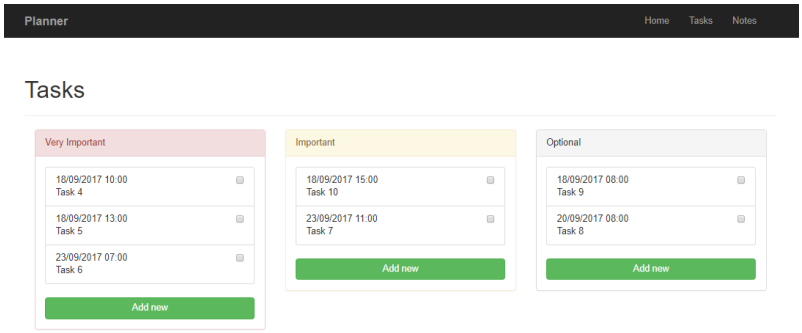
- Sekciju sa bilješkom koju je korisnik postavio kao aktivnu te link koji vodi na stranicu na kojoj se ona može promijeniti.
- Tablicu današnjih zadataka koji se mogu označiti kao završeni.



Slika 2: Naslovna stranica

TaskPage

TaskPage je stranica koja sadrži tri sekcije koje predstavljaju zadatke odvojene po prioritetima. Korisnik može dodati novi zadatak u svaku sekciju te postojeći označiti kao završen.



Slika 3: Stranica zadataka

NotePage

NotePage je stranica koja sadrži sve bilješke koje je korisnik unio. Korisnik može dodati novu, ažurirati ili obrisati postojeću bilješku. Također, ovo je mjesto na kojem korisnik može označiti bilješku kao aktivnu kako bi ona bila vidljiva na naslovnoj stranici.

Slika 4: Stranica bilježaka

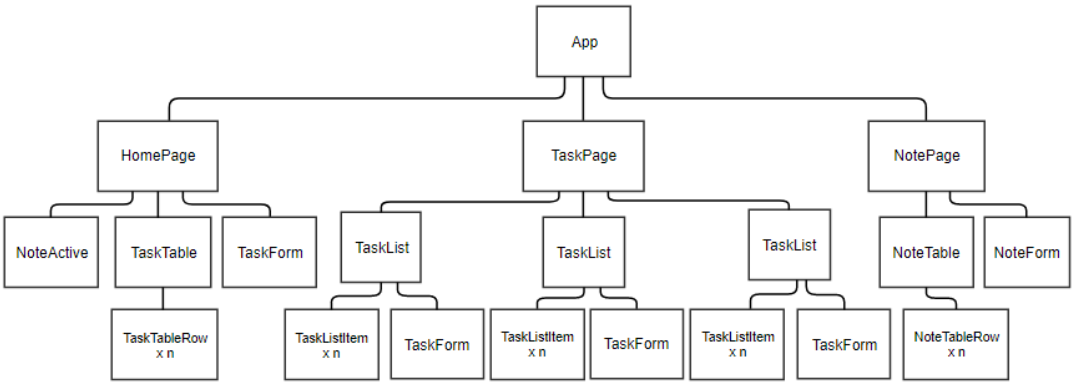
7.2 Korištene tehnologije i paketi

Planner je ReactJS aplikacija čija je server strana pisana u Asp.Net-u. Na klijentskoj strani su korištene biblioteke, koje nisu spomenute u radu, kao što su:

- axios - za stvaranje HTTP zahtjeva.
- bootstrap - za uređivanje elemenata DOM-a.
- rect-router - za navigaciju među stranicama aplikacije.
- webpack - za grupiranja modula aplikacije.

7.3 Hijerarhija komponenti

Budući da se React aplikacija sastoji od ugnježđenih komponenti, bitno je dobro postaviti relaciju među njima. Iz tog razloga se prije prije pisanja kôda preporuča odvojiti vrijeme za crtanje osnovnog dijagrama komponenti. Dijagram komponenti Planner aplikacije prikazan je na sljedećoj slici:



Slika 5: Dijagram komponenti aplikacije

Prateći kriterij prethodno spomenut o radu, koji govori o tome na kojem mjestu u hijerahiji *state* objekt treba živjeti, dodana je *App* komeponenta koja je zajednička komponenta *HomePage*, *TaskPage* i *NotePage* komponenti. Razlog tome je što

HomePage komponenta koristi podatke koji se koriste unuutar *TaskPage* i *NotePage* komponente pa je *state* objekt, koji sadrži zajedničke podatke, potrebno smjestiti na razinu iznad njih, odnosno unutar *App* komponente.

7.4 Izrada podstabla NotePage

Ovo poglavlje pokazat će kako izgleda podstablo s korijenom u komponenti *NotePage* te kako se unutar njega izvode određene akcije.
Kôd *NotePage* komponente izgleda ovako:

```
import React from 'react';
import PageTitle from '../common/PageTitle';
import NoteTable from '../note/NoteTable';
import NoteForm from '../note/NoteForm';

class NotePage extends React.Component {
    render() {
        return (
            <div>
                <PageTitle title="Notes" />
                <div className="panel panel-default">
                    <div className="panel-heading">
                        <NoteForm onFormSubmit={this.props.
                           onNoteFormSubmit} />
                    </div>
                    <div className="panel-body">
                        <NoteTable
                            notes={this.props.notes}
                            onDelete={this.props.onNoteDelete}
                            onMark={this.props.onNoteMark}
                        />
                    </div>
                </div>

            </div>
        );
    }
}

export default NotePage;
```

U prve tri linije kôda uvezeni su određeni moduli o kojima ovisi ova komponenta, React modul te tri komponente. Prva uvezena komponenta je *PageTitle* koja se prosljeđuju dva *props* parametra, *title* i *subititle*.

```
import React from 'react';
```

```
const PageTitle = (props) => {
  return (
    <div className="page-header">
      <h1> {props.title} <small> {props.subtitle} </small> </h1>
    </div>
  );
};

export default PageTitle;
```

Unutar *NotePage* komponente, *PageTitle* komponenti je *prop* parametru *title* prosljeđen string *"Notes"* pa će renderirana komponenta na prikazu aplikacije jednostavno biti element koji sadrži riječ *Notes*.

Slika 6: Renderirana PageTitle komponenta

Sljedeća korištena komponenta je *NoteTable* komeponenta koja služi za ispisvanje svih bilješki unutar tablice. Njezin kôd izgleda ovako:

```
import React from 'react';
import NoteTableRow from '../note/NoteTableRow';

class NoteTable extends React.Component {
  render() {
    const noteTableRows = this.props.notes.map((note) => (
      <NoteTableRow
        key={note.NoteId}
        id={note.NoteId}
        title={note.Title}
        text={note.Text}
        onDelete={this.props.onDelete}
        onMark={this.props.onMark}
      />
    ));
    return (
      <div className="table-responsive">
        <table className="table table-stripped table-hover">
          <thead>
            <tr>
              <th className="col-lg-2"> Title </th>
              <th className="col-lg-7"> Text </th>
```

```
              <th className="col-lg-3"> Action </th>
            </tr>
          </thead>
          <tbody>
            {noteTableRows}
          </tbody>
        </table>
      </div>
    );
  }
}

export default NoteTable;
```

Budući da tablica ne ispisuje jedan redak, redak tablice treba predstavljati komponentu. U tu svrhu je izrađena *NoteTableRow* komponenta.

```
import React from 'react';

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

    this.handleDelete = this.handleDelete.bind(this);
    this.handleMark = this.handleMark.bind(this);
  }

  handleDelete() {
    this.props.onDelete(this.props.id);
  }

  handleMark() {
    this.props.onMark(this.props.id);
  }

  render() {
    return (
      <tr>
        <td> {this.props.title} </td>
        <td> {this.props.text} </td>
        <td>
          <button type="button" className="btn btn-default btn-sm m
              -r-xs"> Edit </button>
          <button type="button" className="btn btn-default btn-sm m
              -r-xs" onClick={this.handleMark}> Mark as active </
              button>
          <button type="button" className="btn btn-danger btn-sm"
              onClick={this.handleDelete}> Delete </button>
        </td>
```

```
      </tr>
    );
  }
};

export default NoteTableRow;
```

Nakon njenog kreiranja, *NoteTable* komponenta će moći renderirati bilo koji broj *NoteTableRow* komponenti opisanih *this.props.notes* parametrom. Na ovaj način je pokazano kako u React-u izvršiti *for each* svojstvo koje se može pronaći u drugim frameworcima, primjerice u Angular-u *ng-repeat*.

Slika 7: Renderirana NoteTable komponenta

Na prethodnoj slici, svaki redak tablice zajedno s okidačima za akcije, čini jednu *NoteTableRow* komponentu.

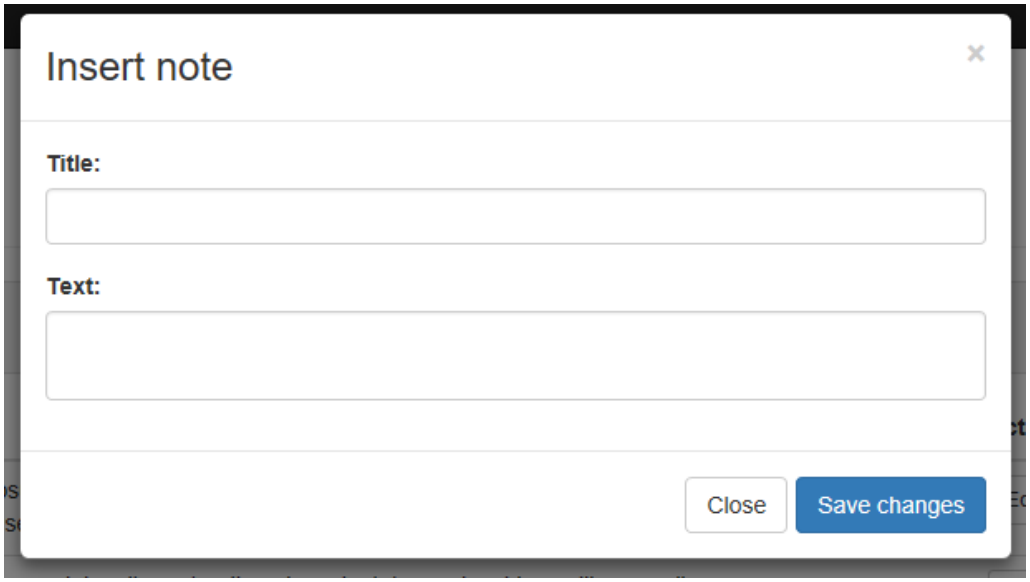
NotePage komponenta također sadrži *NoteForm* komponentu koja sadrži dugme za otvaranje modala s formom za dodavanje nove ili editiranje postojeće bilješke. Njen kôd izgleda ovako:

```
import React from 'react';

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

    const title = this.props.title || "";
    const text = this.props.text || "";
    this.state = {
      title: title,
      text: text
    }

    this.handleTitleChange = this.handleTitleChange.bind(this);
    this.handleTextChange = this.handleTextChange.bind(this);
```

```
    this.handleSubmit = this.handleSubmit.bind(this);
  }

  handleTitleChange(e) {
    this.setState({ title: e.target.value });
  }

  handleTextChange(e) {
    this.setState({ text: e.target.value });
  }

  handleSubmit() {
    this.props.onFormSubmit({
      NoteId: this.props.id,
      Title: this.state.title,
      Text: this.state.text
    });
  }

  render() {
    const modalTitle = this.props.id ? "Edit note" : "Insert note";
    const buttonText = this.props.id ? "Edit" : "Add new";
    const buttonClass = this.props.id ? "btn btn-default btn-sm" :
       "btn btn-success";
    return (
      <div>
        <div className="input-group col-lg-2" data-toggle="modal"
           data-target="#noteModal">
          <button className={buttonClass}> {buttonText} </button>
        </div>

    <div className="modal fade" id="noteModal" tabindex="-1" role="
       dialog" aria-labelledby="myModalLabel" aria-hidden="true">
          <div className="modal-dialog">
            <div className="modal-content">
              <div className="modal-header">
                <button type="button" className="close" data-
                   dismiss="modal" aria-hidden="true">&times;</
                   button>
                <h3 className="modal-title" id="myModalLabel"> {
                   modalTitle} </h3>
              </div>
              <div className="modal-body">
                <form>
                  <div className="form-group">
                    <label for="title"> Title: </label>
                    <input type="text" className="form-control" id=
                       "title" maxlength="255"
                    value={this.state.title} onChange={this.
```

```
                        handleTitleChange} />
                    </div>
                    <div className="form-group">
                      <label for="text"> Text: </label>
                      <textarea type="text" className="form-control"
                         id="text"
                      value={this.state.text} onChange={this.
                         handleTextChange} > </textarea>
                    </div>
                  </form>
                </div>
                <div className="modal-footer">
                  <button type="button" className="btn btn-default"
                     data-dismiss="modal">Close</button>
                  <button type="button" className="btn btn-primary"
                     onClick={this.handleSubmit}>Save  changes</
                     button>
                </div>
              </div>
            </div>
          </div>

        </div>
      );
  }
}

export default NoteForm;
```

Renderirana *NoteForm* komponenta će renderirati dugme koje sadrži tekst *Add new* te će nakon njegovog pritiska renderirati formu za ispunjavanje svojstava bilješke.



Slika 8: Renderirana NoteForm komponenta nakon pritiskanja dugmeta

Unutar *NoteForm* komponente korisnik može pritisnuti dugme *Save changes* nakon čega će očekivati da nova bilješka bude dodana u tablicu bilješki. Radi toga je unutar *App* komponente definirana funkcija *handleNoteFormSumit* koja će tu bilješku dodati na *state* objekt te je *callback* na nju poslan *NoteForm* komponenti pomoću prop parametra *onNoteFormSubmit*. Taj *prop* parametar prosljeđen je *onClick* događaju dugmeta unutar *NoteForm* komponente pa će ona njegovim pritiskom signalizirati *App* komponenti da provede dodavanje bilješke na *state* objekt. *App* komponenta je ujedno komponenta na kojoj se izvršava sva logika nad podacima jer je ona vlasnik *state* objekte. Njen kôd izgleda ovako:

```
import React, { PropTypes } from 'react';
import Header from './common/Header';
import axios from 'axios';
import Notifications, { notify } from 'react-notify-toast';

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

    this.state = {
      notes: [],
      activeNote: {},
      tasksByPriority: [],
      todaysTasks: []
```

```
  };

  this.handleNoteDelete = this.handleNoteDelete.bind(this);
  this.handleNoteMark = this.handleNoteMark.bind(this);
  this.handleNoteFormSumit = this.handleNoteFormSumit.bind(this);
  this.handleTaskClose = this.handleTaskClose.bind(this);
  this.handleTaskFormSubmit = this.handleTaskFormSubmit.bind(this
      );
}

componentDidMount() {

  axios.get('http://localhost:52975/api/note')
    .then((response) => {
      let activeNote = response.data.find(function (note) {
        return note.IsActive === true;
      });
      if (!activeNote)
        activeNote = {
         Title: "Decide. Commit. Succeed."
        };
        this.setState({
          notes: response.data,
            activeNote: activeNote
          });
        });

  axios.get('http://localhost:52975/api/task/priority')
    .then((response) => {
      this.setState({
        tasksByPriority: response.data
      });
    });

  axios.get('http://localhost:52975/api/task/today')
    .then((response) => {
      this.setState({
        todaysTasks: response.data
      });
    });
}

handleNoteDelete(noteId) {
  axios.delete('http://localhost:52975/api/note/' + noteId)
    .then(function (response) {
      notify.show("Note removed successfully.", "success");
        this.setState({
          notes: this.state.notes.filter(function (note) { return
              note.NoteId !== noteId })
```

```
        });
    }.bind(this))
    .catch(function (error) {
      notify.show("An error occurd.", "error");
    });
}

handleNoteMark(noteId) {
  axios.put('http://localhost:52975/api/note/mark', { NoteId:
     noteId })
          .then(function (response) {
              notify.show("Note marked as active successfully.",
                 "success");
              this.setState({
                  activeNote: this.state.notes.find(function (
                     note) {
                      return note.NoteId === noteId;
                  })
              });
          }.bind(this))
          .catch(function (error) {
              notify.show("An error occurd.", "error");
          });
  }

handleNoteFormSumit(note) {
 axios.post('http://localhost:52975/api/note', note)
  .then(function (response) {
    notify.show("Note inserted successfully.", "success");
    note.NoteId = response.data;
    this.setState({
      notes: this.state.notes.concat(note)
    });
    $('#noteModal').modal('hide');
 }.bind(this))
 .catch(function (error) {
    notify.show("An error occurd.", "error");
 });
}

handleTaskClose(taskId) {
 axios.put('http://localhost:52975/api/task/' + taskId + '/close'
    , {taskId : taskId})
   .then(function (response) {
      notify.show("Task closed successfully.", "success");
      this.setState({
        tasksByPriority: this.state.tasksByPriority.map((task) =>
           {
          if (task.TaskId === taskId) {
```

```
            return Object.assign({}, task, {
              IsCompleted: true
            });
          } else {
            return task;
          }
        })
      });
    }.bind(this))
    .catch(function (error) {
      notify.show("An error occurd.", "error");
    });
}

handleTaskFormSubmit(task) {
  axios.post('http://localhost:52975/api/task', task)
    .then(function (response) {
      notify.show("Task inserted successfully.", "success");
      task.TaskId = response.data;
      this.setState({
        tasksByPriority: this.state.tasksByPriority.concat(note)
      });
      $('#taskModal').modal('hide');
    }.bind(this))
    .catch(function (error) {
      notify.show("An error occurd.", "error");
    });
}

render() {
  const children = React.Children.map(this.props.children, (child
    ) => {
    return React.cloneElement(child, {
      notes: this.state.notes,
      activeNote: this.state.activeNote,
      onNoteDelete: this.handleNoteDelete,
      onNoteMark: this.handleNoteMark,
      onNoteFormSubmit: this.handleNoteFormSumit,
      tasksByPriority: this.state.tasksByPriority,
      todaysTasks: this.state.todaysTasks,
      onTaskClose: this.handleTaskClose,
      onTaskFormSubmit: this.handleTaskFormSubmit
    })
  });
  return (
    <div>
      <Notifications />
      <Header />
      <div className="container">
```

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

App.propTypes = {
  children: PropTypes.object.isRequired
};

export default App;
```

Završetak

Danas postoji mnogo JavaScript biblioteka koje uvelike olakšavaju posao u svijetu izrade aplikacija. Naravno, svaka ima svoje prednosti i nedostatke kao što je i u slučaju framework-a pa se traženje najboljih ne može provesti generalno. Traženje najboljeg framework-a ili biblioteke ovisi o aplikaciji. Postavlja se pitanje koja su svojstva framework-a ili biblioteke za nju jako bitna te koja se mogu zanemariti.

Osobno, na poslu još uvijek radim s AngularJS-om 1 te nisam uočila da se renderiranja prikaza aplikacije događaju u nekom drastično velikom vremenu. Možda se razlike vide tek na velikim podacima, kao što ih Facebook ima.

Međutim, što se tiče praksa koje se koriste pri pisanju React komponenti mogu reći da sam ih pomalo počela primjenjivati u svom načinu pisanja kôda. Pokušavam većinu kôda napisati generički kako bi bio pogodan za ponovnu upotrebu. Podatke ili funkcije koje su mi potrebne na više mjesta u aplikaciji nastojim smjestiti na neku zajedničku razinu u hijerarhiji što i meni i ostalima uvelike olakšava snalaženje u kôdu.

Također, sviđa mi se što je u React-u HTML i JavaScript kôd u jednom dokumentu pa se tako u procesu pisanja određene funkcionalnosti ne treba snalaziti po više dokumenata odjednom.

Ako se pravilno koristi, React može imati velike dobrobiti unutar aplikacije. Ovdje se ne misli samo na brzinu i prostorni trošak već i na prakse kojima uči što za rezultat može dati kvalitetnu i brzu aplikaciju sastavljenu od ponovno upotrebljivih cjelina unutar čijeg će se kôda svatko u budućnosti moći lako snaći i dodati neke promjene.

Literatura

[1] 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

[2] https://facebook.github.io/react/

[3] https://www.codecademy.com/articles/react-virtual-dom

[4] http://www.hongkiat.com/blog/ecmascript-6/l

[5] http://reactkungfu.com/2015/10/the-difference-between-virtual-dom-and-dom/

[6] https://hackernoon.com/virtual-dom-in-reactjs-43a3fdb1d130

[7] https://codepen.io/somethingkindawierd/post/
es6-arrow-functions-this

[8] http://www.stefankrause.net/wp/?p=431

[9] https://app.pluralsight.com/library/courses/
react-redux-react-router-es6/table-of-contents

Životopis

Rođena sam 22.01.1993. godine u Virovitici. 2007. godine završila sam osnovnu školu u mjestu prebivališta Špišić Bukovici. Iste godine sam upisala Strukovnu školu u Virovitici, ekonomski smjer. Tijekom osnovne i srednje škole sudjelovala sam na mnogim natjecanjima te školskim i izvanškolskim aktivnostima. 2011. godine na maturi sam odabrala matematičke i informatičke fakultete te iste godine uspješno upisala nastavnički smjer matematike na Odjelu za matematiku Sveučilišta Josipa Jurja Strossmayera u Osijeku. Nakon nepunih tri godine odlučujem se prebaciti na preddiplomski smjer sa ciljem upisivanja diplomskog smjera Matematika i računarstvo. Te iste godine uspješno završavam preddiplomski studij uz završni rad Fermatov teorem beskonačnog spusta pod mentorstvom izv. prof. dr. sc. Ivana Matića. S početkom sljedeće akademske godine upisujem diplomski studij matematike na smjeru Matematika i računarstvo. U ljetu 2015. godine odrađujem 2 mjeseca prakse u Plavoj tvornici u Virovitici na poziciji frontend developera. U studenom 2016. godine počinjem raditi u Osječkoj software tvrtki Mono na poziciji software developer.