Klijentske web-aplikacije i biblioteka React

Šelendić, Matija

Master's thesis / Diplomski rad

2020

Degree Grantor / Ustanova koja je dodijelila akademski / stručni stupanj: **University of Zagreb, Faculty of Science / Sveučilište u Zagrebu, Prirodoslovno-matematički fakultet**

Permanent link / Trajna poveznica: https://urn.nsk.hr/urn:nbn:hr:217:514024

Rights / Prava: In copyright / Zaštićeno autorskim pravom.

Download date / Datum preuzimanja: **2025-03-27**

Repository / Repozitorij:

Repository of the Faculty of Science - University of Zagreb

SVEUČILIŠTE U ZAGREBU

PRIRODOSLOVNO–MATEMATIČKI FAKULTET

MATEMATIČKI ODSJEK

Matija Šelendić

KLIJENTSKE WEB-APLIKACIJE I BIBLIOTEKA REACT

Diplomski rad

Voditelj rada:
izv. prof. dr. sc. Zvonimir Bujanović

Zagreb, veljača, 2020

Ovaj diplomski rad obranjen je dana _____ pred ispitnim povjerenstvom u sastavu:

1. _____, predsjednik
2. _____, član
3. _____, član

Povjerenstvo je rad ocijenilo ocjenom _____.

Potpisi članova povjerenstva:

1. _____
2. _____
3. _____

Sadržaj

Sadržaj **iii**

Uvod **1**

1 Biblioteka React **2**
1.1 Instalacija 2
1.2 Prva aplikacija 3
1.3 JSX 5
1.4 Komponente i svojstva 7
1.5 Unutarnje stanje komponente 9
1.6 Životni ciklus komponente 13
1.7 Upravljanje događajima 15
1.8 Hook funkcije 20

2 Često korišteni dodaci Reactu **22**
2.1 Paket react-router-dom 22
2.2 Paket reactstrap 24
2.3 Paket axios 25

3 Složena web-aplikacija **27**
3.1 Aplikacija za upravljanje razvojem agilnih projekata 27
3.2 Struktura direktorija 33
3.3 Backend i baza podataka 34

Bibliografija **36**

Uvod

React je besplatna biblioteka otvorenog koda (eng. *open-source*) za programski jezik JavaScript čija je glavna svrha izrada korisničkog sučelja (UI), odnosno, fokusirana je na klijentsku stranu web-aplikacije. Razvoj UI-a pomoću React-a svodi se na definiranje pojedinih komponenata sučelja od kojih svaka ima određeno interno stanje i reagira na događaje. Te komponente se onda koriste u izradi kompletnog korisničkog sučelja. No kako se programska logika kod modernih web-aplikacija sve više premješta sa serverske na klijentsku stranu, tako i uloga razvojnih okvira poput React-a postaje puno više od same izrade UI-a. Tipična aplikacija koja koristi React tako implementira i usmjeravanje (eng. *routing*), te koristi pakete kao što su paket *reactstrap* za lakši dizajn korisničkog sučelja ili paket *axios* za upite na server. Biblioteku React održava Facebook te zajednica individualnih programera.

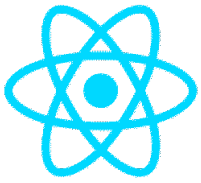
U prvom poglavlju opisujemo osnove biblioteke React. Govorimo o njenoj instalaciji i kreiranju nove web-aplikacije iz temelja. Objasnit ćemo poseban dodatak sintakse JavaScripta koji podsjeća na HTML jezik i razloge zašto ga je dobro koristiti. Zatim, ulazimo u detaljan opis nezavisnih dijelova korisničkog sučelja (tzv. React komponente) te kako React koristi DOM (*Document Object Model*) sučelje za njihov prikaz. Nadalje, govorimo o životnom ciklusu i vijeku tih komponenata, njihovim svojstvima i njihovom unutarnjem stanju. Na kraju ovog poglavlja nešto ćemo reći o upravljanju događajima te tzv. Hooks funkcijama.

U drugom poglavlju opisujemo dodatne pakete koji se često koriste u sklopu React aplikacija. Opisat ćemo pakete *react-router-dom* za usmjeravanje, *reactstrap* za lakši dizajn korisničkog sučelja te *axios* za spremanje i dohvat podataka iz baze.

U trećem poglavlju opisujemo složenije web-aplikacije u React-u. Za potrebe ovoga rada kreirana je web-aplikacija za upravljanje razvojem proizvoljnih agilnih projekata. Na početku opisujemo funkcionalnost same web-aplikacije, kako se ona upotrebljava i koje su njezine mogućnosti te prikazujemo implementaciju koncepata iz prvog poglavlja.

Poglavlje 1

Biblioteka React



Slika 1.1: Logotip projekta React

React je besplatna biblioteka otvorenog koda za programski jezik JavaScript koja omogućava razvoj korisničkih sučelja SPA (eng. *single-page application*). SPA aplikacija je tip web-aplikacije koja u interakciji s korisnikom dinamički mijenja dijelove stranice, za razliku od tradicionalnih web-aplikacija koje učitavaju cijele nove stranice s poslužitelja. One se pokreću u browseru i ne zahtijevaju ponovno učitavanje nakon korištenja. Neke poznate SPA aplikacije su Gmail, Facebook te GitHub.

React omogućava sastavljanje korisničkog sučelja pomoću elemenata koji se nazivaju komponente. Njih je moguće iznova koristiti u aplikaciji proizvoljno veliki broj puta. Takvo sastavljanje UI-a postoji već duže vrijeme, ali React je prvi koji je to omogućio koristeći samo osnovnu funkcionalnost JavaScripta, bez vanjskih biblioteka (tzv. *Vanilla JS*). Taj pristup pokazao se lakši za održavanje, proširivanje i višestruko korištenje.

Prvo izdanje Reacta bilo je 2013. godine. Danas React pripada među najpopularnije razvojne okvire u JavaScriptu te ga koriste vodeća tehnološka poduzeća kao što su Facebook, Netflix i Airbnb.

1.1 Instalacija

Postoji više načina kako možemo napraviti aplikaciju u Reactu ili uvesti React u već postojeću aplikaciju. Preporuča se Facebook-ov paket *create-react-app*. Da bismo ga isko-

ristili moramo instalirati *Node.js* i *npm* (eng. *node package manager*). *Npm* je upravitelj paketima za programski jezik JavaScript i dolazi u sklopu razvojne okoline *Node.js* koju možemo preuzeti na adresi `https://nodejs.org/en/`.

Npm služi za brzu instalaciju eksternih paketa (biblioteka ili cijelih razvojnih okolina) direktno iz komandne linije. Paket *create-react-app* može se instalirati globalno i zatim kreirati aplikacija korištenjem sljedeće dvije naredbe:

```
1 npm install -g create-react-app
2 create-react-app example-app
```

Ako ne želimo globalno instalirati paket *create-react-app*, preporuča se korištenje npx-a (tzv. *npm package runner*):

```
1 npx create-react-app example-app
```

Ova naredba stvorit će potrebne datoteke za polaznu aplikaciju u React-u unutar korisnikovog trenutnog direktorija bez instalacije paketa *create-react-app*.

1.2 Prva aplikacija

Nakon stvaranja prve aplikacije pojavio se novi direktorij s nazivom *example-app* koji sadrži bazičnu aplikaciju istog imena. Novostvoreni direktorij inicijalno sadrži tri direktorija i četiri datoteke. Objasnimo:

- **README.md**: Sadrži informacije o pokretanju aplikacije te linkove na React dokumentaciju.
- **node_modules/**: Sadrži sve pakete instalirane na računalo korištenjem naredbe *npm install*. Naredbom *create-react-app* inicijalno se instalirao velik broj paketa koje možemo naći u ovom direktoriju.
- **package.json**: Sadrži listu svih paketa o kojima ovisi naša aplikacija, inicijalne skripte i ostale konfiguracije projekta.
- **package-lock.json**: Automatski generirana datoteka koja sadrži točne verzije paketa. Datoteka *package.json* ne mora sadržavati točno specificiranu verziju paketa.
- **.gitignore**: Popis svih datoteka i direktorija koji nisu uključeni u git repozitorij kada se on koristi.

- **public**/: Sadrži datoteke koje se koriste pri početnom pokretanju aplikacije. Najvažnija datoteka je *index.html*. Ona sadrži programski kod u HTML-u koji definira što će se prikazati u web-aplikaciji nakon njenog pokretanja.
- **src**/: Sadrži sav izvorni kod aplikacije.

Za pokretanje aplikacije smjestimo se unutar njenog korjenskog direktorija i pokrenimo skriptu *start* definiranu u datoteci *package.json*.

```
1  cd example-app
2  npm start
```

Izvršavanjem ove naredbe pokrenut će se web-preglednik unutar kojega možemo vidjeti inicijalni prikaz naše web-aplikacije *example-app* (slika 1.2).

Slika 1.2: Inicijalna stranica nove web-aplikacije

Dobili smo poruku s uputom da promijenimo datoteku *App.js* unutar direktorija *src*. Nakon što to napravimo i spremimo promjene, one će se odraziti i u aplikaciji.

1.3 JSX

HTML

HTML (*Hypertext Markup Language*) je standardni jezik za izradu web-stranica. Sve web-stranice na današnjem internetu implementirane su pomoću HTML-a.

Dokument napisan u HTML-u sastoji se od niza ugnježđenih elemenata. Pojedini elementi imaju različite uloge, pa neki opisuju naslove unutar dokumenta (na primjer, element *h1*), neki paragrafe teksta (element *p*), a neki slike (element *img*). Svaki element sastoji se od početnih i završnih oznaka (tzv. *tag-ova*) između kojih je naveden sadržaj elementa. Na primjer, naslov na web-stranici bismo mogli postaviti ovako:

```
<h1> Ovo je naslov ...</h1>
```

Prikaz elemenata

Najjednostavniji primjer aplikacije napisane pomoću React-a prikazan je u sljedećem isječku koda:

```
import React from 'react';
import ReactDOM from 'react-dom';
ReactDOM.render(
  <h1>Hello, world!</h1>,
  document.getElementById('root')
);
```

Uočimo da se ovdje koristi pomoćni paket *react-dom* koji je za nas instalirala naredba *create-react-app*. Ovaj paket sadrži metode za upravljanje DOM-om od kojih je najpoznatija metoda *render* čija svrha je da kontrolira sadržaj unutar određenog čvora stabla DOM-a. Metoda *render* uzima dva parametra: prvi je tzv. React element napisan korištenjem JSX-a, a drugi je neki čvor stabla u DOM-u (u našem slučaju čvor s identifikatorom *'root'*). Ako zamijenimo cijeli sadržaj datoteke *index.js* unutar direktorija *src* s gornjim isječkom koda te pokrenemo aplikaciju, na ekranu ćemo vidjeti naslov *'Hello, world!'*.

JSX

Isječak koda u gornjem odlomku napisan je u programskom jeziku JavaScript. Mogli bismo pomisliti da je prevoditelj tokom prevođenja trebao javiti grešku jer smo pokušali

u metodu *render* proslijediti element *<h1>* koji nije dio JavaScript-a, već HTML-a. No greške nije bilo. O čemu se ovdje radi?

React koristi ekstenziju programskog jezika JavaScript koja se naziva JSX (*JavaScript XML*). Ta ekstenzija koristi sintaksu koja podsjeća na HTML/XML koju smo vidjeli u prošlom primjeru u obliku *h1* elementa koji smo proslijedili kao prvi parametar u metodu *render*. Tijekom prevođenja aplikacije, sav kod napisan korištenjem JSX-a će se pretvoriti u odgovarajući JavaScript kod. Prevođenje automatski radi alat *Babel*, Javascript prevoditelj koji se instalirao pokretanjem naredbe *create-react-app*. Tako će gornji kod biti pretvoren u:

```
1    React.createElement('h1', null, 'Hello, world!');
```

Ovdje se poziva metoda *createElement* iz paketa *react* koja prima tri parametra. Prvi je string koji određuje koji će se HTML element ubaciti kao čvor u DOM stablo, drugi je JSON objekt koji sadrži sva svojstva koja se predaju elementu (ovdje *null* pošto nema proslijeđenih svojstava, o tome više kasnije), dok je treći lista istih takvih objekata čvorova koji će predstavljati djecu toga čvora u DOM stablu (ovdje samo string *'Hello, world!'* umjesto jednočlane liste). Funkcija *createElement* vraća JSON objekt sljedećeg tipa:

```
1    {
2      type: 'h1',
3      props: {
4        children: 'Hello, world!'
5      }
6    }
```

Ovakvi JSON objekti nazivaju se React elementi. Možemo zamisliti kako se slaganjem ovih objekata može stvoriti proizvoljno veliki JSON objekt koji bi precizno opisao odnos roditelja i djece čvorova koji čine stablo u DOM-u. Takav cjelokupni objekt šalje se kao parametar u već spomenutu metodu *render* klase *ReactDOM* iz paketa *react-dom* koji onda iz toga kreira izgled web-stranice, tj. korisničko sučelje.

Korištenje JavaScript izraza u JSX-u

Unutar JSX koda dozvoljeno je ubaciti Javascript izraze. Pogledajmo sljedeći primjer.

```
1    import React from 'react';
2    import ReactDOM from 'react-dom';
3    var name = 'John Smith';
4    const element = <h1>Hello, {name}!</h1>;
5    ReactDOM.render(element, document.getElementById('root'));
```

Inicijalizirali smo varijablu *element* (napisan u JSX-u) u koji smo koristeći vitičaste zagrade ubacili prethodno inicijaliziranu varijablu *name*. Ako datoteku *index.js* u našoj aplikaciji zamijenimo s gornjim kodom te pokrenemo aplikaciju, vidjet ćemo naslov *'Hello, John Smith!'*.

JSX dozvoljava da se unutar vitičastih zagrada unese bilo koji validan JavaScript izraz (npr. *1 + 1*, *name* ili *getName()*).

Prednosti i mane

Kao i svaki programski jezik i JSX ima svoje prednosti i mane. Mnogi programeri koji se prvi puta susreću s React-om imaju problema s JSX sintaksom jer dopušta miješanje jezika JavaScript i XML/HTML-a. Taj pristup je nov i isprva može biti neintuitivan i konfuzan. Doduše, većina programera hvali JSX jer značajno olakšava posao razvoja aplikacije. Korištenjem JSX-a programer ne mora razmišljati o JSON objektima i inicijalizaciji svojstva *children* svaki puta kada kreira React element. Uz to, u obliku HTML-a nudi puno ljepši vizualni prikaz elemenata.

Treba napomenuti da ekstenziju JSX nije obavezno koristiti. I bez nje se može raditi s React-om, ako se koristi metoda *createElement* kao zamjena za stvaranje elemenata, ali taj pristup se ne preporuča.

1.4 Komponente i svojstva

U gornjem primjeru ubacili smo *<h1>* element u čvor s identifikatorom *root*. To nije bilo zahtjevno iz razloga što smo imali element koji sadrži samo jedno dijete, kratki tekst *'Hello, world!'*. Zamislimo da smo trebali prikazati element koji ima više djece, deset ili stotinu? Kako bismo to izveli? Taj problem rješavaju komponente.

React komponente su nezavisne i ponovno upotrebljive gradivne cjeline svake React aplikacije. Jednostavno rečeno, jedna komponenta je JavaScript klasa ili funkcija koja prima proizvoljan broj parametara koji se nazivaju svojstva (eng. *property*) i vraća React element koji opisuje kako će izgledati određeni dio UI-a. Drugim riječima, možemo je promatrati kao objekt koji obuhvaća naše React elemente i dodaje im programsku logiku — u programerskoj terminologiji takav objekt nazivamo *wrapper*.

Funkcijska komponenta

Najjednostavniji način za stvaranje jedne komponente je pisanjem funkcije u JavaScriptu:

```
import React from 'react';
function Greeting() {
  return <h1>Hello, world!</h1>;
}
```

Upravo smo stvorili funkcijsku komponentu bez ulaznih argumenata. Oni nam ovdje nisu bili potrebni. No u većini slučajeva, funkciji ćemo htjeti proslijediti neke ulazne podatke, po potrebi napraviti neku transformaciju nad njima i tek onda vratiti rezultat. Za to služi argument *props*. On predstavlja objekt s ulaznim podacima. Modificirajmo gornji primjer tako da pošaljemo svojstvo *name*:

```
import React from 'react';
function Greeting(props) {
  return <h1>Hello, {props.name}!</h1>;
}
```

Komponenta kao klasa

Drugi način pomoću kojeg možemo stvoriti komponentu je korištenjem klase u JavaScriptu (ovdje koristimo standard ECMAScript 6):

```
class Greeting extends React.Component {
  render() {
    return <h1>Hello, {this.props.name}</h1>;
  }
}
```

Stvaranje komponente pomoću klase radi se tako da se naslijedi klasa *React.Component* pomoću ključne riječi *extends*. Unutar te klase obavezno je implementirati metodu *render*. Nju ne treba miješati s već viđenom metodom iz klase *ReactDOM* koja ubacuje React element u čvor stabla u DOM-u.

Prikazivanje (renderiranje) komponenti

Prisjetimo se, u gornjem primjeru stvorili smo sljedeći React element.

```
var name = 'John Smith';
const element = <h1>Hello, {name}!</h1>;
```

No to nije jedini način kako možemo stvoriti React element. Validan je i sljedeći kod:

```
const element = <Greeting name="John Smith">;
```

Ovdje se dogodilo nekoliko stvari. JSX je uočio našu komponentu koju smo prethodno stvorili te dodatni par (koji nas podsjeća na atribut u HTML-u) u obliku ključ-vrijednost (*name="John Smith"*). Koristeći taj par (moglo ih je biti i više) inicijaliziran je objekt *props* i proslijeđen je kao ulazni parametar u našu komponentu, te ovisno o kojoj vrsti komponente se radi, nastao je objekt kojim je inicijalizirana varijabla *element*. Ako je komponenta *Greeting* bila funkcijska komponenta, onda je JSX samo vratio vrijednost te funkcije, a ako se radilo o klasi, onda je vratio vrijednost render metode te klase. U oba slučaja rezultat je isti.

Važno je napomenuti da se u oba načina kreiranja komponente mora vratiti samo jedan objekt. Drugim riječima, React komponenta vraća samo jedan React element koji nije ništa drugo nego JavaScript objekt koji, kako smo vidjeli gore, sadrži tip i svojstva, među kojima je i objekt *children*. Dakle, nije moguće napraviti komponentu koja vraća tri komponente jednu ispod druge. Ako to želimo učiniti, te tri komponente moramo zamotati u neki drugi HTML element. Slijedi primjer:

```
const element =
  <div>
    <Greeting name="John" />
    <Greeting name="Mark" />
    <Greeting name="Anne" />
  </div>
```

1.5 Unutarnje stanje komponente

Za sada znamo da svojstva unutar komponente utječu na izgled korisničkog sučelja. To je samo po sebi korisno, no kod njih postoji jedan nedostatak. Ta svojstva ne mogu se mijenjati unutar komponente u kojoj se nalaze (eng. *immutable*). Ako želimo promijeniti izgled UI-a onda je potrebno uništiti komponentu te stvoriti novu kojoj ćemo predati nove vrijednosti za objekt *props*. Kako bi se to izbjeglo, React uvodi koncept unutarnjeg stanja komponenata (eng. *state*). Prema mnogima, ovo je najvažniji i najkorisniji koncept koji se koristi u izgradnji React aplikacija.

Stanje je promjenjivi (eng. *mutable*) skup podataka sadržan unutar objekta *state* neke komponente. Svaka komponenta ima svoj vlastiti skup stanja tj. ti podaci za nju su privatni. Drugim riječima, stanje se može izmjenjivati samo unutar komponente koja ih sadrži. Kao

i kod svojstava, stanje je varijabla koja se može ubaciti u JSX unutar metode *render* te tako utjecati na izgled korisničkog sučelja. Ono što razlikuje objekt *state* od objekta *props* je činjenica da promjenom podatka *state* unutar komponente, ako je taj podatak korišten u prikazu UI-a (*render*), dolazi do automatskog ponovnog renderiranja te komponente u DOM bez njenog uništavanja. Važno je napomenuti da se mijenjaju jedino dijelovi UI-a na koje utječe promijenjeni podatak. Ostatak DOM-a ostaje nepromijenjen.

Jedna komponenta može imati proizvoljno mnogo stanja. Njih dohvaćamo po ključu. Ključ je svojstvo objekta *state* u komponenti. Dakle, ako imamo objekt *state* s ključem *name* (kao u primjeru gore) možemo ga dohvatiti s *this.state.name*.

Inicijalizacija i korištenje stanja

Do 2019. godine i uvođenja tzv. *state hook-a*, stanje se nije moglo koristiti unutar funkcijske komponente. Ako je komponenta zahtijevala korištenje stanja, ona je morala biti napisana kao klasa jer se njemu pristupalo pomoću ključne riječi *this*.

Stanja unutar komponente mogu se dohvaćati i mijenjati po volji. No prije svega mora se postaviti inicijalna vrijednost svakog podatka koji želimo dodati u skup stanja. To se radi unutar konstruktora komponente.

Inicijalizirajmo komponentu *Greeting* s inicijalnim stanjem:

```
class Greeting extends React.component {
  constructor(props) {
    super(props);
    this.state = {
      name: props.name
    };
  }
  render() {
    return (
      <h1>Hello, {this.state.name}!</h1>
    );
  }
}
```

Komponente unutar konstruktora kao prvu naredbu moraju pozvati konstruktor svoje bazne klase kojem predaju objekt *props*.

I sada kreiramo komponentu:

```
<Greeting name="John Smith" />
```

Unutar svog objekta *state* ova komponenta će imati ključ *name* inicijaliziran s vrijednošću *'John Smith'*.

Mijenjanje stanja

Za vrijeme korištenja aplikacije promjena stanja unutar komponente događa se jako često. Najčešći primjer je tijekom upravljanja događaja. Naime, korisnik aplikacije pritiskom na gumb ili utipkavanjem teksta u *input* komponentu može izazvati promjenu stanja određene komponente te time promijeniti izgled korisničkog sučelja. Kako ćemo upravljanje događaja opisati u jednoj od sljedećih sekcija, ovdje ćemo se poslužiti JavaScript metodom *setTimeout* kojom ćemo simulirati neki događaj (izvršiti promjenu stanja) da bismo vidjeli promjene UI-a.

Promjena stanja vrši se pozivom metode *setState* kojoj se prosljeđuje objekt koji sadrži novo željeno stanje. Metoda *setState* njime zamjenjuje staro te zatim poziva metodu *render*. Važno je napomenuti da je unutar proslijeđenog objekta dovoljno poslati samo one podatke iz skupa stanja koje je potrebno promijeniti, a ne sve koje postoje unutar objekta *state*. React za nas radi spajanje.

Unutar konstruktora u gornjem kodu dodat ćemo sljedeću naredbu:

```
1     setTimeout(this.changeState(), 2000);
2
```

te ćemo izvan konstruktora, ispred metode *render* inicijalizirati metodu *changeState* koja će poslužiti kako bismo promijenili stanje komponente:

```
1     changeState() {
2       this.setState({
3         name: 'John Wayne'
4       });
5     }
```

U našoj inicijalnoj aplikaciji zamijenimo sav kod iz datoteke *index.js* sa sljedećim:

```
1     import React from 'react';
2     import ReactDOM from 'react-dom';
3
4     class Greeting extends React.Component {
5       constructor(props) {
6         super(props);
7
8         this.state = {
9           name: props.name
```

```
10          };
11
12          setTimeout(() => this.changeState(), 2000);
13        }
14
15        changeState() {
16          this.setState({
17            name: 'John Wayne'
18          });
19        }
20
21        render() {
22          return (
23            <h1>Hello, {this.state.name}!</h1>
24          );
25        }
26      }
27
28      ReactDOM.render(<Greeting name="John Smith"/>, document.
    getElementById('root'));
```

te pokrenimo aplikaciju. Vidjet ćemo ime inicijalizirano u konstruktoru te njegovu promjenu nakon dvije sekunde. Štoviše, prilikom ovog procesa komponenta se nije uništila, već ponovno prikazala.

Ako sada istu stvar pokušamo napraviti direktnim mijenjanjem stanja u metodi *changeState* naredbom:

```
1     this.state.name = "John Wayne";
```

vidjet ćemo da ovim načinom nećemo postići ponovno renderiranje komponente pa u UI-u neće biti promjene.

Kod korištenja metode *setState* treba biti oprezan jer ona može mijenjati stanje asinkrono. Zbog performansi, React može spojiti nekoliko poziva metode *setState* te samo jednom izvršiti metodu *render*. Zbog ovoga se ne treba oslanjati na prethodna stanja tijekom računanja novog stanja. Na primjer, ako u komponenti imamo brojač, sljedeći kod može krivo promijeniti varijablu *counter*:

```
1     this.setState({
2       counter: this.state.counter + this.props.increment
3     });
```

Kako bi se zaobišle ovakve greške, koristi se drugi oblik metode *setState* koja umjesto objekta prima funkciju. Ta funkcija prima prethodno stanje kao prvi argument i svojstva kao drugi. Prikazujemo ispravan način korištenja:

```
1       this.setState((state, props) => ({
2         counter: state.counter + props.increment
3       }));
```

Neiskusnim programerima može biti teško odlučiti hoće li pojedine podatke koristiti kao svojstva ili ih držati u skupu stanja. Ponovimo: svojstva su nepromijenjiva i predana su komponenti od roditelja kao podaci ili funkcije koje reagiraju na događaje (tzv. *"event handleri"*), dok je skup stanja promjenjiv te se pojedina stanja često šalju kao svojstva u djecu pripadne komponente.

1.6 Životni ciklus komponente

Svaka komponenta u Reactu prolazi kroz niz događaja. Te događaji nisu ništa drugo nego metode unutar komponente koje se izvršavaju u određenom redoslijedu i tako simuliraju njen životni ciklus (od trenutka kada je komponenta stvorena pa do njenog uništenja).

Metode životnog ciklusa dijelimo na 3 skupine:

- **Metode montiranja**: Pozivaju se u vrijeme kada se komponenta stvara i ubacuje u DOM.
- **Metode ažuriranja**: Pozivaju se kada želimo promijeniti objekte *props* ili *state* u komponenti.
- **Metode uništenja**: Komponenta koja se poziva kada želimo maknuti komponentu iz DOM-a.

Često korištene metode

Daleko najčešće korištena metoda životnog ciklusa komponente, već spomenuta u ovom radu, je metoda *render*. Ona se nalazi u svakoj klasi koja nasljeđuje klasu *React.Component*. Obavezno ju je implementirati. Kao što samo ime kaže, ova se metoda brine za prikazivanje (renderiranje) komponente u korisničko sučelje. Metoda *render* se izvršava za vrijeme stvaranja komponente i ažuriranja njenog stanja. React zahijeva da je metoda *render* čista. To znači da se u njoj ne smije mijenjati unutarnje stanje. Kad bi se to dogodilo pomoću metode *setState*, dogodila bi se beskonačna petlja (*render* $\Rightarrow$ *setState* $\Rightarrow$ *render* $\Rightarrow$ ...). Ako je potrebno mijenjati stanje, postoje druge metode životnog ciklusa koje tome služe.

Kada je komponenta pomoću metode *render* postavljena u DOM, na red dolazi metoda *componentDidMount*. Ona može poslužiti za dohvat podataka iz baze i dozvoljava korištenje metode *setState*.

Sljedeća metoda o kojoj govorimo je *componentDidUpdate*. Ona se poziva neposredno nakon što se ažuriranje stanja dogodilo. Ova metoda služi za upravljanje DOM-om u trenutku kada se mijenja stanje. I u njoj se često implementiraju pozivi za dohvat podataka. Dozvoljava se pozivanje metode *setState*, ali unutar nekog uvjeta jer će se inače dogoditi beskonačna petlja.

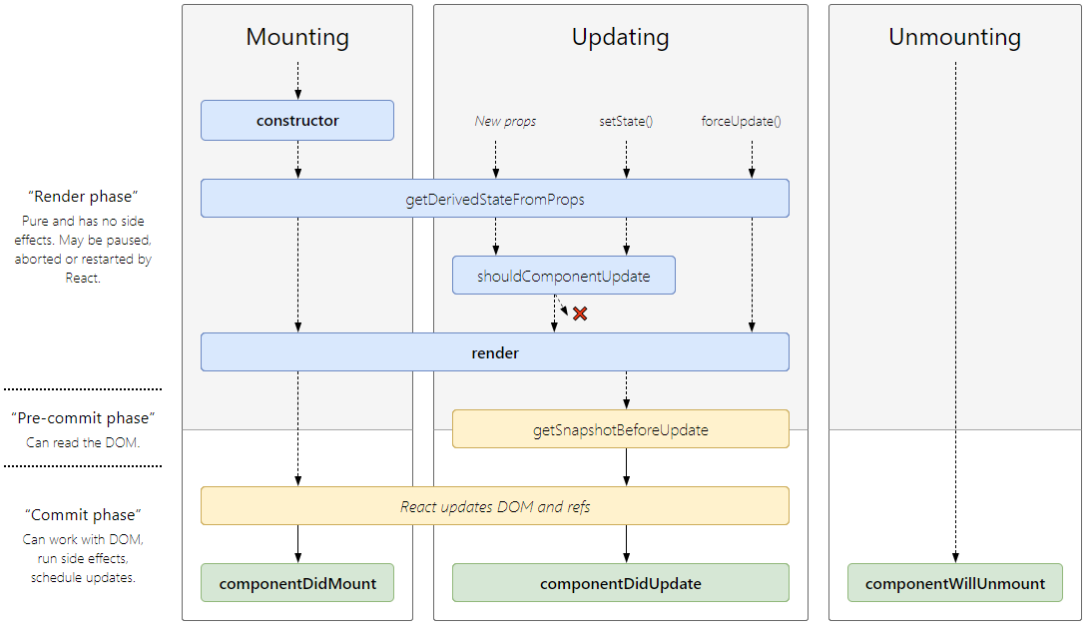
Neposredno prije uništavanja komponente pozvat će se metoda *componentWillUnmount*. Ako je potrebno izvršiti neku akciju čišćenja poput uništavanja event listenera ili brisanja iz *local storage-a*, ova metoda je mjesto u kojoj se to može učiniti. Naravno, nije moguće mijenjati stanje komponente u njoj pošto se nakon njenog izvršavanja komponenta uništava.

Rjeđe korištene metode

Druge metode životnog ciklusa koje se rjeđe koriste su:

- **shouldComponentUpdate**: Korisna u slučaju ako ne želimo pozvati metodu *render* nakon što smo ažurirali stanje komponente. Služi za optimizaciju performansi.
- **getDerivedStateFromProps**: Statička metoda. Koristi se kada stanje ovisi o promijeni svojstava. Vraća objekt kojim će se ažurirati stanje u slučaju kada se promjeni neko svojstvo. Može vratiti *null* ako se ne mijenja stanje.
- **getSnapshotBeforeUpdate**: Poziva se neposredno prije ažuriranja UI-a. Vrijednost koju vraća proslijeđuje se u metodu *componentDidUptate*. Primjer korištenja je mijenjanje veličine prozora.

Navedene metode mogu se vidjeti na dijagramu 1.3.



Slika 1.3: Metode životnog ciklusa

1.7 Upravljanje događajima

Do sada smo u našim primjerima radili samo prikaz podataka. Drugim riječima, nismo omogućavali nikakvu interakciju s korisnikom kao što je pritisak gumba ili unos nekog teksta. U ovom poglavlju uvodimo događaje.

HTML događaji (eng. *events*) su akcije koje se odvijaju nad HTML elementima u DOM-u. Koristeći JavaScript moguće je reagirati na te događaje. Elementima se specificiraju atributi čija imena predstavljaju akcije nad njima i kao vrijednost im se dodjeljuje JavaScript metoda koja će se izvršiti nakon što se ta akcija dogodi. Ta metoda naziva se upravitelj događajem (eng. *event handler*). Često se još naziva *event listener*.

Upravljanje događajima u Reactu vrši se na sličan način kao i upravljanje događajima na HTML elementima u DOMu. Postoje dvije razlike:

- React koristi *camelCase* sintaksu za pisanje imena događaja.
- U React-u se predaje metoda, a ne string kao što je kod HTML događaja.

Vezanje objekta this unutar funkcija

Često je potrebno na neki događaj reagirati promjenom stanja komponente. Kako bi to bilo moguće, to stanje je prvo potrebno dohvatiti koristeći objekt *this*.

Pogledajmo sljedeći primjer:

```
1  class CustomInput extends React.Component {
2    constructor(props) {
3      super(props);
4      this.state = {
5        text: props.initialText
6      };
7    }
8
9    render() {
10     return (
11       <div>
12         <input value={this.state.text} onChange={function(event) {
13           this.setState({
14             text: event.target.value
15           });
16         }}/>
17         <h1>Text: {this.state.text}</h1>
18       </div>
19     );
20   }
21 }
22
23 ReactDOM.render(
24   <CustomInput initialText=""/>, document.getElementById('root'));
```

U gornjem primjeru u korisničkom sučelju prikazujemo jedan input HTML element te naslov koji prikazuje tekst unutar tog input elementa. U skup stanja dodali smo svojstvo *text* čije inicijalno stanje ovisi o varijabli *initialText*. Definirali smo upravitelj događajem koji će na promjenu teksta u input elementu promijeniti svojstvo *text* unutar objekta *state* u komponenti. Također smo input elementu zadali inicijalnu vrijednost pomoću atributa *value*.

Ako sada pokrenemo aplikaciju te počnemo tipkati, dobit ćemo grešku da je objekt *this undefined*. Drugim riječima, u funkciji ne postoji referenca na instancu klase *CustomInput*. Kako bismo ovo popravili, potrebno je povezati objekt *this* unutar funkcije. Drugim riječima, kako je upravitelj događajem tzv. *callback metoda*, ona mora znati kontekst na koji se ključna riječ *this* odnosi pa ga moramo zadati. To za nas radi metoda *bind*.

Prvo izvucimo naš upravitelj događajem u zasebnu callback metodu unutar klase *CustomInput*:

```
handleKeyPress(event) {
  this.setState({
    text: event.target.value
  });
}
```

zatim promijenimo vrijednost *onChange* attributa u gornjem kodu:

```
    <input value={this.state.text} onChange={this.handleKeyPress} />

```

i za kraj, dodajmo sljedeću naredbu u konstruktor koja čini povezivanje:

```
this.handleKeyPress() = this.handleKeyPress.bind(this);
```

Pokrenimo aplikaciju. Objekt *this* je prepoznat i na unos teksta u input element mijenja se objekt *state* te na ekranu vidimo ispisan taj tekst. Mogli bismo i logirati objekt *this* u konzolu unutar metode *handleKeyPress* kako bismo vidjeli da se uistinu radi o instanci klase *CustomInput*.

U gornjem primjeru unutar upravitelja događaja *handleKeyPress* proslijedili smo parametar *event*. Taj objekt je zaslužan za promjenu stanja naše komponente *CustomInput*.

Kako bismo objasnili, možemo ispisati taj event:

```
handleKeyPress(event) {
  console.log(event);
  ...
}
```

Pokrenimo aplikaciju, počnimo tipkati i pogledajmo u konzolu. Uočimo da se ispisala instanca klase *SyntheticEvent*.

SyntheticEvent

Rad s objektima tipa *Event* direktno iz DOM-a može biti izuzetno zahtjevno iz razloga što razni web-preglednici na različiti način implementiraju te događaje. Ovisno o kojem web-pregledniku se radi, objekt *event* koji se predaje upravitelju događaja može imati različite metode i svojstva. Kako korisnici aplikacija moraju svjedočiti istom ponašanju neovisno

o tome koji web-preglednik koriste, programeri tih aplikacija moraju biti na oprezu i pisati razne uvjetne naredbe u aplikaciji kako bi to omogućili. Kako bi se to izbjeglo, razvojni tim biblioteke React je uveo omotač oko nativnih *event* objekata web-preglednika kojeg naziva *SyntheticEvent*. Taj pristup čini te objekte konzistentnima neovisno o web-pregledniku. React zamota određeni DOM *event* u tu klasu te njenu instancu šalje upraviteljima događaja.

Neka od svojstava klase *SyntheticEvent*, kojue možemo vidjeti ispisane u konzoli, su:

- **target**: instanca klase *DOMEventTarget*, predstavlja HTML element nad kojim je događaj pokrenut.
- **nativeEvent**: instanca klase *DOMEvent*, predstavlja nativni *event* objekt preglednika.

Najčešća metoda klase *SyntheticEvent* koja se koristi je *preventDefault* koja spriječava izvršavanje predefinirane (*defaultne*) akcije u web-pregledniku.

Više informacija o klasi *SyntheticEvent*, kao i ukupna lista događaja koje React podržava može se naći u [9].

Razmjena podataka između komponenata

Do sada smo koristili svojstva kako bismo određene varijable iz roditelja proslijedili komponenti djetetu te ih tamo koristili za gradnju UI-a. No varijable nisu jedina stvar koja se može proslijediti. Moguće je kao svojstvo proslijediti i metodu koja se po potrebi može pozivati.

Kao demonstraciju ćemo kreirati komponentu *Names* te u njoj pozivati dvije instance komponente *CustomInput* iz prošlog primjera koju ćemo malo izmijeniti tako da ćemo je napraviti kao funkcijsku komponentu iz razloga što nam u njoj više neće biti potrebno spremanje teksta:

```
1  function CustomInput(props) {
2    return (
3      <div>
4        <label>Name: </label>
5        <input onChange={props.handleChange} />
6      </div>
7    );
8  }
```

Ova komponenta služi samo za unos imena koje ćemo prikazati u komponenti *Names*:

```
class Names extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      firstName: '',
      secondName: ''
    };
    this.handleFirstNameChange = this.handleFirstNameChange.bind(this);
    this.handleSecondNameChange = this.handleSecondNameChange.bind(this)
    ;
  }

  handleFirstNameChange(event) {
    this.setState({
      firstName: event.target.value
    });
  }

  handleSecondNameChange(event) {
    this.setState({
      secondName: event.target.value
    });
  }

  render() {
    return (
      <div>
        <h3>First name: {this.state.firstName}</h3>
        <h3>Second name: {this.state.secondName}</h3>
        <CustomInput handleChange={this.handleFirstNameChange}/>
        <CustomInput handleChange={this.handleSecondNameChange}/>
      </div>
    );
  }
}
```

Sada možemo ubaciti tu komponentu u DOM:

```
ReactDOM.render(<Names/>, document.getElementById('root'));
```

U gornjem kodu mogu se vidjeti dvije metode, *handleFirstNameChange* i *handleSecondNameChange*. One će biti korištene kao upravitelji događajima pa se moraju vezati za objekt *this* unutar konstruktora. Te metode proslijeđuju se kao svojstvo *handleChange*

(atribut objekta props) u komponentu *CustomInput* gdje se onda dalje proslijeđuju input elementu kao upravitelji za događaj *onChange*. Nakon što korisnik unese neki tekst u prvu input komponentu poziva se callback metoda *handleFirstNameChange* iz komponente *Names* koja mijenja atribut *firstName* u njenom skupu stanja. Kako metoda *setState* interno poziva metodu *render*, na ekranu se vidi promjena. Isto se postiže unosom teksta u drugu input komponentu. Ovaj puta se poziva metoda *handleSecondNameChange* te se postavlja stanje *secondName*.

S jedne strane imamo proslijeđivanje podataka djeci koristeći svojstva. No sada je jasan i drugi smjer toka podataka: od djeteta prema roditelju. Taj tok radi se pomoću proslijeđivanja metoda u djecu koje se onda iznutra mogu pozivati.

Više informacija o React biblioteci, komponentama i upravljanju događaja može se vidjeti u [10], [11].

1.8 Hook funkcije

U ovom poglavlju ukratko ćemo opisati tzv. *Hook* funkcije, dodatak Reactu 2019. godine kojim se značajno olakšava razvoj aplikacija.

Hook funkcije su specijalne funkcije koje omogućavaju korištenje stanja i metoda životnog ciklusa unutar funkcijskih komponenata. Postoji nekoliko takvih funkcija od kojih je najpopularnija metoda *useState*.

useState

Način funkcioniranje metode *useState* najbolje ćemo demonstrirati primjerom:

```
1  import React, {useState} from 'react';
2  import ReactDOM from 'react-dom';
3
4  function CustomInput() {
5    const [text, setText] = useState('');
6
7    return (
8      <div>
9        <input onChange={(e) => setText(e.target.value)} />
10       <p>Text: {text}</p>
11     </div>
12   );
13 }
14 ReactDOM.render(<CustomInput/>, document.getElementById('root'));
```

Koristimo funkcijsku komponentu *CustomInput* koja sadrži element input te ovisno o tome što se u nju upiše prikazuje taj tekst na ekranu.

Kako u funkcijskim komponentama nije moguće koristiti objekt *state* ovdje smo se poslužili Hook funkcijom *useState*. Ona nam omogućuje da deklariramo varijable stanja (koji su analogni svojstvima objekta *state* u klasnim komponentama) unutar naše funkcijske komponente. Vraća niz od dva elementa: prvi element je vrijednost željenog trenutnog stanja, a drugi je funkcija kojom se to stanje ažurira. Koristimo sintaksu destrukcije niza kako bismo ih dohvatili u zasebne lokalne varijable proizvoljnog imena ([*text*, *setText*]). Metodi *useState* proslijeđujemo jedan parametar, a to je inicijalna vrijednost stanja. U našem primjeru to je prazan string.

useEffect

Još uvijek postoji nedostatak funkcijskih komponenti u odnosu na komponente koje su kreirane kao klase, a to je nemogućnost pisanja metoda životnog ciklusa. Tu uskače Hook funkcija *useEffect*. Ona omogućava da se unutar funkcijskih komponenata definira dodatan kod (npr. dohvat podataka iz baze), koji će se izvršiti nakon što se DOM ažurirao.

U gornjem primjeru promijenili smo inicijalni tekst i te dodali metodu *useEffect*:

```
1    const [text, setText] = useState('Initial text');
2    useEffect(() => {
3      setText('I was set inside useEffect()');
4    });
```

Kao i *useState*, da bismo koristili metodu *useEffect* potrebno je napraviti *import* iz paketa *react*. Ona prima jedan parametar, a to je funkcija koja će se izvršiti nakon što se DOM ažurira. Kada smo pokrenuli ovaj primjer uočili smo da se u DOMu na trenutak ispisalo *"Initial text"* te da se zatim izvršila metoda *useEffect* koja je promijenila taj tekst. Dakle ova Hook metoda radi istu stvar kao i *componentDidMount* unutar klasne komponente.

Važno je napomenuti da je u funkcijskoj komponenti dozvoljeno više puta pozivati funkcije *useState* i *useEffect* tj. imati više različitih varijabli stanja ili poduzeti više različitih akcija nakon renderiranja u DOM. Postoje i neka pravila o korištenju Hook funkcija. Korektno ih je pozivati samo unutar funkcijskih komponenata i nije ih dozvoljeno koristiti u petljama, uvjetnim naredbama ili ugniježđenim funkcijama.

Osim Hook funkcija *useState* i *useEffect*, postoje mnoge druge. Njih, kao i način za stvaranje vlastitih Hook funkcija, moguće je vidjeti na službenoj React web stranici [9].

Poglavlje 2

Često korišteni dodaci Reactu

Izgradnja kompleksnijih web-aplikacija zahtijeva više od samog prikazivanja komponenti u DOM-u. Drugim riječima, uz biblioteku React koja to omogućuje potrebno je unutar projekta imati uključene dodatne pakete (tzv. *dependency*). Neke od čestih dodataka React aplikacijama su paketi koje se brinu za usmjeravanje, mogućnost dizajniranja aplikacije te spremanje i dohvaćanje podataka iz baze.

Postoji veliki broj dodatnih paketa koji se svakodnevno koriste u raznim web-aplikacijama. Besplatni su i mogu se preuzeti i uključiti u projekt na isti način na koji smo preuzeli i paket *create-react-app* iz prvog poglavlja, a to je koristeći *npm*.

NPM (skr. *node package manager*) je najveći svjetski registar softverskih paketa. Sadrži više od 800,000 paketa. Može ih se vidjeti u [6].

U ovom poglavlju ukratko ćemo objasniti popularne pakete *react-router-dom*, *react-strap* i *axios*. Njih ćemo koristiti u aplikaciji koja će biti opisana u trećem poglavlju.

2.1 Paket react-router-dom

Paket *react-router-dom* je standardni paket koji služi za usmjeravanje u React aplikacijama. Drugim riječima, njegov jednostavan API omogućuje prikaz određenih komponenti React aplikacije ovisno o URL-u. Uključuje se u projekt pomoću sljedeće naredbe:

```
1  npm install react-router-dom --save
```

Sadrži tri vrste komponenata:

- **Router**: Najčešće korišten je *BrowserRouter*. Sadrži programsku logiku o preusmjeravanju.

- **Route**: Komponente *Switch* ili *Route*. Ovisno o ruti u URL-u određuje koja će se komponenta prikazati u korisničkom sučelju.
- **Navigation**: Komponente *Link*, *NavLink* ili *Redirect*. Služe za preusmjeravanje (navigaciju).

Prikazujemo jednostavan primjer korištenja:

```
import {BrowserRouter, Switch, Route, Link} from "react-router-dom";
<BrowserRouter>
  <div>
    <Link to="/">Home</Link>
    <Link to="/about">About</Link>
    <Link to="/dashboard">Dashboard</Link>
  <Switch>
    <Route exact={true} path="/" component={Home}/>
    <Route path="/about" component={About}/>
    <Route path="/dashboard" component={Dashboard}/>
  </Switch>
  </div>
</BrowserRouter>
```

U gornjem kodu naveli smo tri linka i tri rute omotane u komponentu *BrowserRouter*. Komponente tipa *Route* obavezno je omotati komponentom tipa *Router*. Ruta može biti proizvoljno mnogo i dodijeljuju im se dva svojstva: *path* i *component*. Ako se tijekom korištenja aplikacije u URL unese path koji odgovara path-u unutar specificirane rute, komponenta *Router* će za nas u UI-u prikazati komponentu koju smo naveli za tu rutu.

Komponenta *Link* ima istu namjenu kao i element *<a>* u HTML-u. Klikom na njega mijenja se URL ovisno o vrijednosti svojstva *to* koja je proslijeđena u komponentu *Link*. Dakle, u gornjem primjeru, klikom na link *Dashboard*, promijenit će se path unutar URL-a u */dashboard*. Komponenta *BrowserRouter* će to prepoznati i prikazati komponentu *Route* sa svojstvom *path="/dashboard"*. Komponenta *Route* tada će prikazati komponentu proslijeđenu kao vrijednost svojstva *component*, a to je komponenta *Dashboard*.

Primjerimo da smo koristili i komponentu *Switch*. Ona služi za prikazivanje isključivo jedne komponente tipa *Route*. Ako bismo gornji kod napisali bez komponente *Switch*, onda bi se pritiskom mišem na link *Dasboard* u UI-u prikazale dvije komponente (*Home* i *Dashboard*). To se događa zato što se komponenta *Route* prikazuje ako je početni dio path-a u URL-u jednak svojstvu *path* koji joj je proslijeđen. Kako za URL path */dashboard* to vrijedi za obje komponente *Home* i *Dashboard*, obje će se prikazati u DOM-u. Kako bi se to izbjeglo, komponente *Route* zamataju se u komponentu *Switch* te se komponentama dodijeljuje svojstvo *exact* (ovdje je dovoljno to učiniti samo jednom).

Gornjim primjerom tek smo zagrebali u paket *react-router-dom*. On nudi još puno mogućnosti za usmjeravanje kao što su prikaz komponenti koristeći URL parametre, korištenje Router-a unutar Router-a (tzv. *nesting*), zaštićene rute i upravljanje greškama. Više o paketu *react-router-dom* može se vidjeti u [7].

2.2 Paket reactstrap

Većina današnjih korisnika interneta očekuje da web-aplikacije koje koriste budu jednostavne za korištenje i ugodne oku. Kako bi zadržali korisnike, razvojni timovi web-aplikacija sadrže pojedince koji se bave isključivo dizajnom njihovih aplikacija (tzv. *web-designer*). Oni su često i sami programeri tih aplikacija. Kako bi njihov posao bio jednostavniji, koriste mnoge pomoćne pakete koji sadrže već gotove dizajnirane komponente poput gumba, elementa za unos teksta (tzv. *input field*) ili odabir datuma (tzv. *date picker*). Za razvoj web-aplikacija u React-u jedan od čestih paketa koji se koristi za dizajniranje aplikacije je paket *reactstrap*.

Instalacija se radi na sljedeći način:

```
npm install --save bootstrap
npm install --save reactstrap
```

Kako bismo koristili komponente iz paketa *reactstrap* potrebno je instalirati i paket *bootstrap*. To je najčešće korištena biblioteka za responzivni prikaz web-stranica koja omogućuje lagano dodavanje CSS klasa elementima u DOM-u pa tako i manipuliranje njihovim izgledom. Paket *reactstrap* koristi paket *bootstrap* za dizajniranje svojih komponenata.

Kako bi se CSS klase iz paketa *bootstrap* primijenile na komponente iz paketa *reactstrapa* potrebno ih je unutar datoteke *index.js* uključiti sljedećom naredbom:

```
import 'bootstrap/dist/css/bootstrap.css';
```

Kao primjer, pogledajmo način korištenja komponente *Dropdown* iz paketa reactstrap:

```
<Dropdown isOpen={dropdownOpen} toggle={toggle}>
  <DropdownToggle caret>
    Dropdown
  </DropdownToggle>
  <DropdownMenu>
    <DropdownItem>Action one</DropdownItem>
    <DropdownItem>Action two</DropdownItem>
  </DropdownMenu>
</Dropdown>
```

Komponenta *Dropdown* ovdje ima dva svojstva: *isOpen* — varijabla tipa boolean koja određuje je li padajući izbornik otvoren te *toggle* — metoda koja se poziva nakon što se padajući izbornik otvori i zatvori (unutra eksplicitno mijenjamo vrijednost *dropdownOpen* varijable kako bi otvarali ili zatvarali padajući izbornik).

Paket *reactstrap* sadrži još puno komponenata: gumbe, forme, tabove, spinnere. Neke od njih koristit ćemo i u aplikaciji unutar trećeg poglavlja. Više o komponentama paketa *reactstrap* može se vidjeti u [3].

2.3 Paket axios

U kompleksnijim web-aplikacijama koriste se pozadinski procesi na poslužitelju (tzv. *backend*) te baze podataka. Način komuniciranja klijentskih web-aplikacija s poslužiteljima je korištenjem HTTP protokola. Taj proces odvija se tako da klijentski dio web-aplikacije pošalje upit (eng. *request*) poslužitelju (koji zatim komunicira s bazom podataka) te čeka njegov odgovor (eng. *response*). Najčešće korištene HTTP metode su GET, POST, PUT, DELETE za čitanje, slanje, uređivanje ili brisanje određenih podataka u bazi (tzv. *CRUD metode*). O HTTP protokolu više se može saznati na sljedećem linku: [5].

Programski jezik JavaScript ima ugrađeni API koji to omogućuje (tzv. *Fetch API*). U sljedećem primjeru koristit ćemo besplatni poslužitelj *randomuser.me* koji će vratiti slučajno odabranog korisnika iz baze.

Pošaljimo upit poslužitelju koristeći Fetch API:

```
1  const url = 'https://randomuser.me/api/?results=1';
2  fetch(url)
3  .then(response => response.json())
4  .then(function(data) {
5     console.log(data);
6  })
7  .catch(function(error) {
8    console.log(error);
9  });
```

U gornjem primjeru pozivamo Fetch API i proslijeđujemo mu URL kojeg smo definirali kao konstantu iznad. Metoda *fetch* automatski koristi GET metodu HTTP protokola. Nakon što smo dobili odgovor poslužitelj (tzv. *response*), on se prvo mora pretvoriti u JSON (skr. *JavaScript Object Notation*) oblik objekta. Ako se dogodila pogreška, izvršit će se dio

koda unutar *catch* bloka. U oba slučaja ispisujemo poslužiteljev odgovor u konzolu. Više infomacija o Fetch API-u može se vidjeti u [5].

Axios je paket koji ima istu svrhu kao i Fetch API, no jednostavnije ga je koristiti i nudi dodatne mogućnosti poput presijecanja zahtijeva (eng. *request interception*). To je mjesto za logiranje, autentifikaciju i dr.

Instalacija paketa axios radi se na sljedeći način:

```
npm install axios --save
```

Primjer korištenja je sličan kao kod Fetch API-ja:

```
import axios from 'axios';
const url = 'https://randomuser.me/api/?results=1';
axios.get(url)
.then(function (response) {
  console.log(response);
})
.catch(function (error) {
  console.log(error);
})
```

Da bismo koristili axios potrebno je napraviti *import*.

Jedna od razlika Axios-a i Fetch API-ja je ta što kod axios-a nije potrebno pretvarati response u JSON objekt. Axios to radi za nas.

Više o paketu axios može se saznati u [2].

Poglavlje 3

Složena web-aplikacija

Kako bismo demonstrirali koncepte iz prvoga poglavlja, za potrebe ovoga rada napravili smo složeniju web-aplikaciju za upravljenje razvojem agilnih projekata. Pojasnimo malo što znači agilni razvoj.

Postoje dvije vrste metoda za razvoj softvera: klasične i agilne. Kod klasičnih metoda, razvoj softvera se promatra kao pažljivo planirani proces koji ima svoj početak i kraj. U tom procesu se utvrđuju zahtjevi te se radi plan za realizaciju softvera. Određuje se raspored ljudi i resursa po aktivnostima. Zahtijeva se dokumentiranje softvera. Klasične metode koriste se kod razvoja velikih projekata kod kojih je potrebno imati sve korisničke zahtjeve unaprijed određene.

Za razliku od klasičnih metoda koje postoje od 70-ih godina 20. stoljeća, agilne metode pojavile su se tek u 21. stoljeću. Najpoznatija takva metoda naziva se *Scrum* metoda. Njena karakteristika je da se softver razvija u brzim i kratkim iteracijama od kojih jedna, u pravilu, traje dva tjedna (tzv. *sprint*). U svakoj iteraciji, programerima se dodijeljuju određeni zadaci u obliku kartica teksta. U razvojnom timu postoji član koji je predstavnik korisnika te on određuje i raspoređuje te zadatke po prioritetima. Nakon završenog sprinta, predstavnik korisnika provjerava status tih zadataka te ovisno o tome, određuje nove zadatke za sljedeći sprint.

Namjera agilnih metoda je ubrzati razvoj softvera i izbjeći nepotrebnu administraciju. One su se pokazale korisnima u slučajevima kada se radi o malim i kratkoročnim projektima, obično u to spada izrada mobilnih i web-aplikacija.

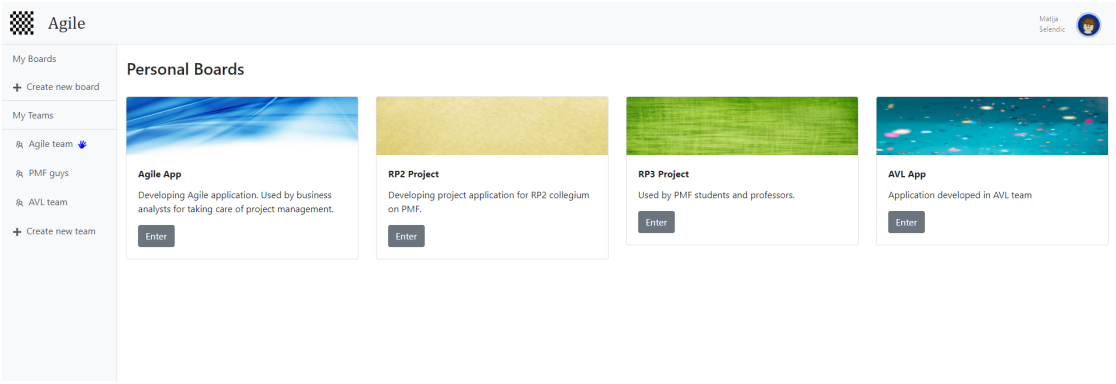
3.1 Aplikacija za upravljanje razvojem agilnih projekata

Kako bismo demonstrirali rad agilnih projekata razvili smo aplikaciju *Agile* u kojoj smo omogućili korisniku razvoj i upravljanje proizvoljnih projekata uz pomoć razvojnog tima kojim on također može upravljati. Kao u agilnim metodama razvoja softvera, aplikacija

pruža mogućnost jednostavnog stvaranja zadataka unutar određenog projekta u obliku kartica teksta, njihovog dodijeljivanja članovima razvojnog tima te pregled tih zadataka po kategorijama i njihovog trenutnog statusa. Također, omogućuje članovima tima komunikaciju o pojedinim zadacima u obliku komentara.

Aplikacija je nastala po uzoru na postojeću aplikaciju Trello [8]. Izvorni kod dostupan je na priloženom CD-u te se nalazi u repozitoriju na Githubu [1].

Korištenje aplikacije zahtijeva ulogiravanje prethodno registriranog korisnika. Za tu svrhu napravljene su dvije komponente: *Login* i *Register*. One imaju forme za unos potrebnih podataka za registraciju i ulogiravanje u aplikaciju. Nakon što se korisnik ulogirao, prikazat će mu se lista timova kojima trenutno pripada i projekata na kojima trenutno radi (slika 3.1).



Slika 3.1: Naslovna stranica

Osnovne komponente

Nakon ulogiravanja, aplikacija se sastoji od četiri različite React komponente koje se mijenjaju korištenjem paketa *react-router-dom* te dvije koje su stalno pristutne (na njih router ne utječe):

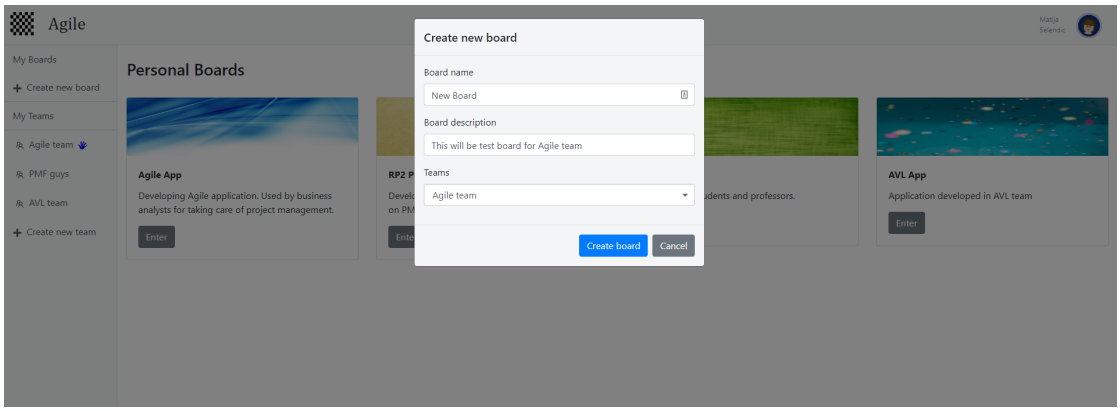
Stalno pristutne komponente u korisničkom sučelju su:

- **Naslovna traka** (eng. *topbar*): Prikazana na samom vrhu UI-a. Prikazuje logo i ime aplikacije u lijevom kutu, te ime, prezime i fotografiju ulogiranog korisnika u desnom kutu.
- **Bočna traka** (eng. *sidebar*): Prikazana lijevo u UI-u. Prikazuje link na projekte reprezentirane u obliku panoa (eng. *board*), te razvojne timove u kojima se korisnik nalazi. Osim toga, omogućuje stvaranje novih projekata ili novog tima.

Na naslovnoj traci korisnik može pritisnuti na svoju ikonu. Otvorit će se padajući izbornik s dvije akcije: *Profile* i *Logout*. Pritiskom na akciju *Profile* prikazat će se komponenta

koja prikazuje informacije o profilu korisnika, a pritiskom na *Logout* korisnik će se izlogirati iz aplikacije.

Na bočnoj traci moguće je stvoriti novi projekt ili tim. Ako korisnik to odluči napraviti, otvorit će se komponenta *Modal* u kojem će korisnik morati unijeti podatke o novom projektu ili timu. Za novi projekt potrebno je unijeti ime i opis projekta te zadati tim koji će raditi na tom projektu. Za novi tim, korisnik će morati unijeti ime tima te navesti članove. Dodavanje novog projekta možemo vidjeti na slici 3.2.



Slika 3.2: Stvaranje novog projekta

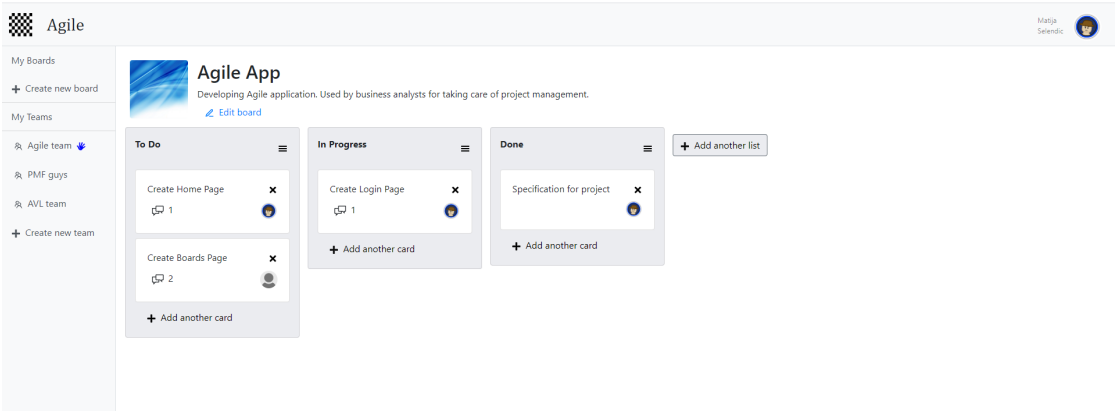
Na naslovnoj stranici aplikacije, u donjem desnom dijelu korisničkog sučelja možemo vidjeti listu projekata (u obliku komponenata *Card* iz paketa *reactstrap*) na kojima ulogirani korisnik trenutno radi. Zahvaljujući paketu *react-router-dom* koji iz URL-a prepoznaje je da se radi o naslovnoj stranici, prikazana je komponenta *Boards*.

Postoji četiri komponente koje se prikazuju u aplikaciji ovisno o rutama u URL-u:

- **Boards**: Izlistani su svi projekti na kojima korisnik radi. Path u URL-u: *"/"*.
- **Board**: Prikazuje izgled jednog projekta, sastoji se od kategorija u kojima se nalaze zadaci. Path kojim se otvara je *"/board/:id"*.
- **Team**: Prikazuje informacije o razvojnom timu. Moguće je pregledati projekte na kojima razvojni tim radi te članove tog tima. Path: *"/team/:id"*.
- **Profile**: Omogućuje prikaz informacija o trenutno ulogiranog korisniku. Path: *"/profile"*.

Već smo vidjeli komponentu *Boards*. Opišimo malo ostale komponente.

Kako bismo otvorili komponentu Board, pritisnimo gumb *Enter* na jednoj od kartica na naslovnoj stranici. Prikazat će se više informacija o odabranom projektu (slika 3.3).



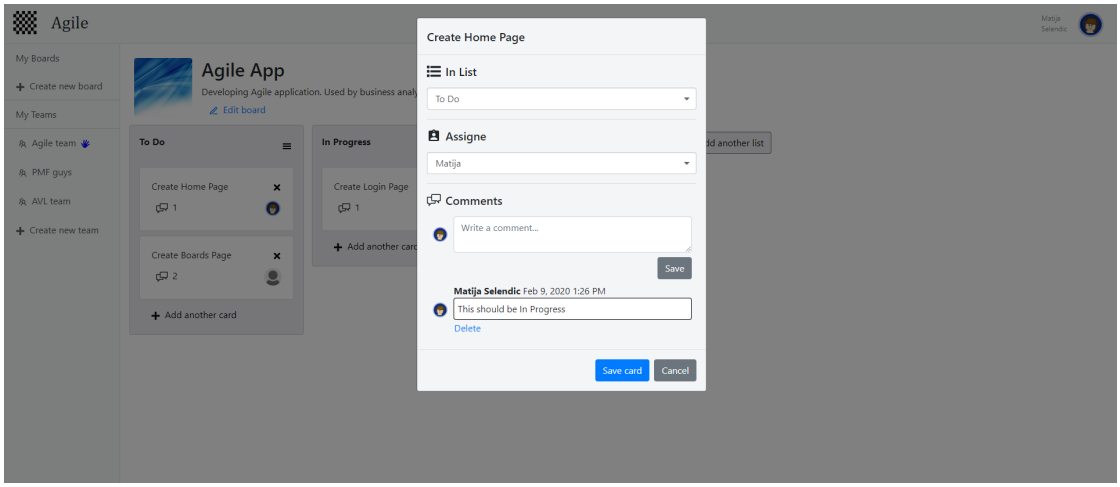
Slika 3.3: Komponenta Board

Komponenta *Board* sastoji se od dva dijela. Prvi dio (na slici gore) prikazuje osnovne informacija o projektu: ime projekta, opis, sliku te tim koji na njoj radi. Isto tako omogućuje kreatoru projekta da te osnovne informacije uredi ili da obriše projekt.

U drugom dijelu komponente *Board* (na slici dolje) možemo vidjeli tri stupca koji predstavljaju liste. Jedan projekt može imati proizvoljno mnogo lista. Unutar jednog panoa moguće ih je dodavati, brisati ili mijenjati redoslijed. Na liste možemo gledati kao na kategorije u kojima se nalaze korisnički zadaci. Svaka lista sadrži proizvoljan broj korisničkih zadataka u obliku kartica teksta. Lista ima naziv i poredak te listu zadataka. U jednu listu moguće je dodavati nove zadatke te ih brisati.

Korisnički zadaci su predstavljeni React komponentom *Card*. Jedan zadatak može raditi samo jedan član razvojnog tima. Na gornjoj slici unutar kartice možemo vidjeti fotografiju člana razvojnog tima kojem je dodijeljen zadatak koji je reprezentiran tom karticom. Kartica može sadržavati i proizvoljan broj komentara. Ako postoje komentari, na kartici je prikazan njhov broj.

Pritiskom mišem na karticu otvorit će se komponenta *Modal* u kojoj se može vidjeti više informacija o kartici (slika 3.4).



Slika 3.4: Komponenta Card

Dodavanjem komponente *Modal* iz paketa *reactstrap* u *render* metodu komponente otvorit će se skočni prozor (eng. *pop-up*). Na gornjoj slici može se vidjeti prozor koji prikazuje više informacija o kartici. Korisnik može vidjeti u kojoj listi (kategoriji) se nalazi kartica te istu prebaciti u neku drugu listu (npr. prebaciti je iz liste *To Do* u listu *In progress*). Uz to, korisnik može promijeniti člana tima koji trenutno radi na zadatku.

Komponenta *Card* sadrži i listu komponenti *Comment* koje predstavljaju prethodno napisane komentare članova razvojnog tima na taj zadatak. Komponenta *Comment* prikazuje ime, prezime i sliku člana tima koji je taj komentar napisao, datum kada je komentar napisan, te njegov sadržaj. Komentari su sadržani unutar state objekta komponente *Card*. Korisnik može pregledati listu komentara te napisati novi komentar. Isto tako može i brisati svoje komentare.

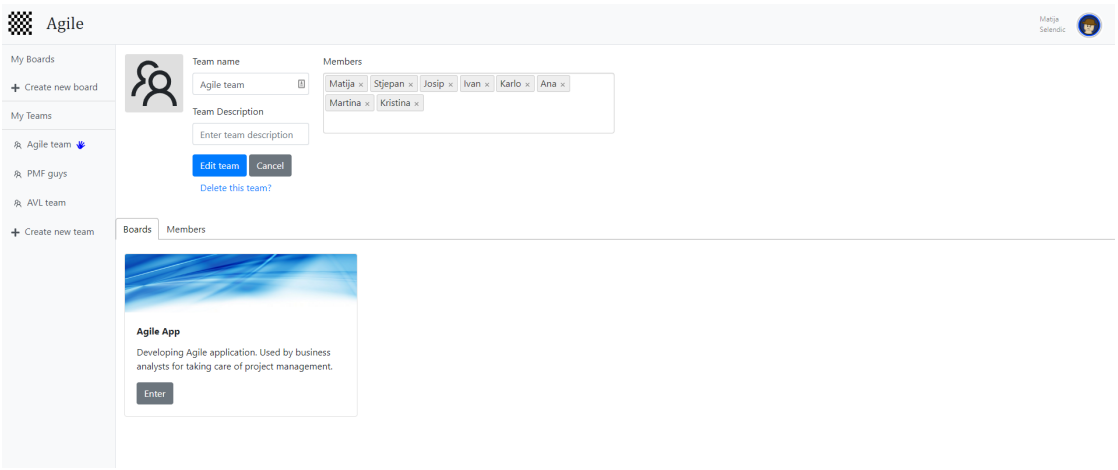
Pogledajmo kako komponenta Card prikazuje komentare unutar svoje render metode:

```
1     render() {
2       return (
3         ...
4         {this.state.comments.map(comment => (
5         <Comment key={comment.id} comment={comment} deleteComment={(
      comment) => this.deleteComment(comment)}/>
6         ))}
7       );
8     }
9   })
```

Unutar objekta *state* komponente *Card* sadržana je lista *comment* koja sadrži informacije o svim komentarima ove kartice. Ta lista dohvaćena je iz baze koristeći paket *axios*.

Unutar metode *render* primjetit ćemo petlju koja za svaki komentar instancira komponentu *Comment*. Prilikom instanciranja proslijeđujemo joj dva svojstva: komentar koji želimo prikazati i upravitelj događaja *deleteComment* koji će se izvršiti nakon pritiska mišem na gumb *delete* unutar komponente *Comment*. JSX je dovoljno pametan da tako dobivenu listu komponenti *Comment* posloži jednu iza druge unutar metode *render*. U tom redoslijedu ćemo ih vidjeti u korisničkom sučelju.

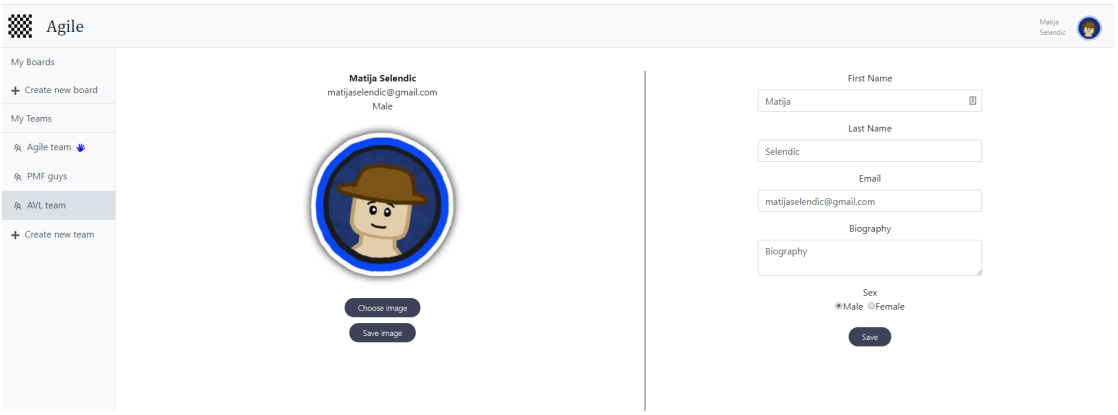
Sljedeće ćemo ukratko opisati komponentu *Team*. Do nje možemo doći preko bočne trake pritiskom na tim (slika 3.5).



Slika 3.5: Komponenta Team

Prvi dio komponente *Team* sličan je kao i komponenta *Board*. Prikazuje ime, opis i sliku razvojnog tima kao i njegove članova. Omogućava autoru tima da uredi te informacije i da dodaje ili briše članove iz njega. U donjem dijelu komponente možemo vidjeti komponentu *Tabs* iz paketa *reactstrap*. U jednom tab-u prikazani su svi projekti na kojima razvojni tim radi, a u drugom njegovi članovi.

Zadnja komponenta koju ćemo prikazati je komponenta *Profile*. Do nje je moguće doći pritiskom slike profila na naslovoj traci te odabrati *Profile* u padajućem izborniku (slika 3.6).



Slika 3.6: Komponenta Profile

Komponenta *Profile* sastoji se od dva dijela. Na lijevoj strani prikazani su osnovni podaci o ulogiranom korisniku: ime, prezime, email i spol. Uz to je omogućen upload slike. Na desnoj strani moguće je editirati te podatke uz dodatak biografije. Promjenom nekog podatka i pritiskom na gumb *Save* promjene će se odraziti na lijevoj strani komponente i u ostatku aplikacije, a promjenom slike, promjena će biti vidljiva i u naslovnoj traci.

3.2 Struktura direktorija

Aplikaciju smo započeli korištenjem naredbe *create-react-app* (objašnjeno u prvom poglavlju). Ta naredba je u korijenskom direktoriju aplikacije, između ostalog, stvorila direktorij *src* u kojem je sadržan sav izvorni kod aplikacije.

Složenije web-aplikacije u React-u, pa tako i naša, sadrže puno programske logike i velik broj komponenata. Kako bi se programeri lakše snalazili, postoji pravilo razdvajanja koda u poddirektorije unutar direktorija *src*.

Unutar tog direktorija napravili smo četiri direktorija:

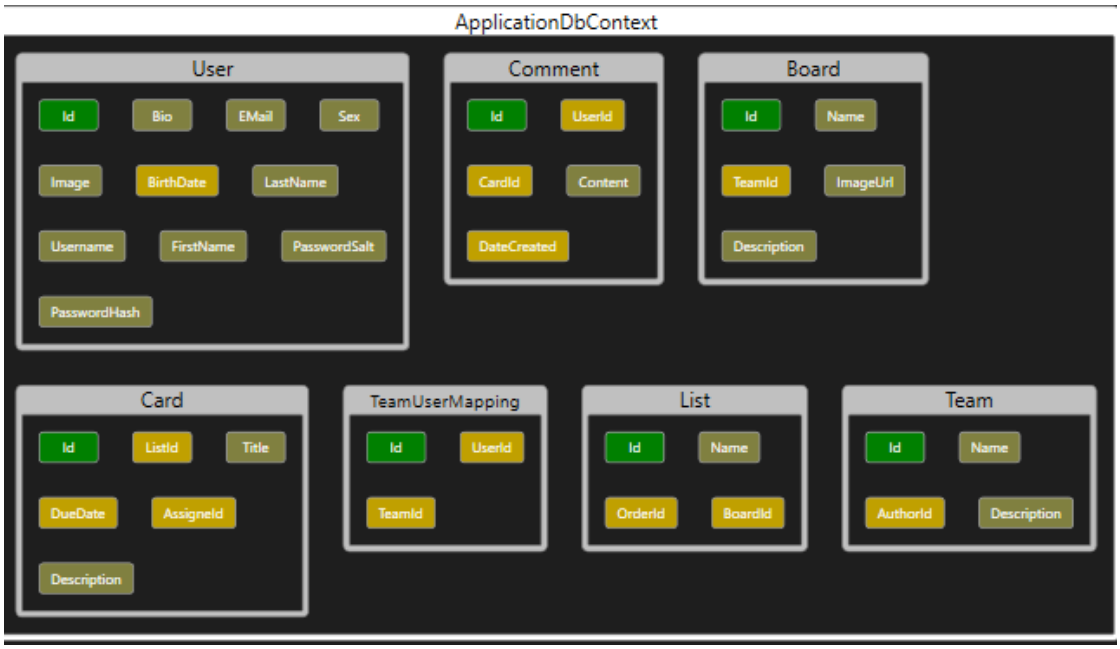
- **Assets**: U ovom direktoriju nalaze se sve slike, animacije, css koji koristimo unutar naše aplikacije.
- **Components**: Sadrži većinu komponenti koje su korištene u aplikaciji i njihovu programsku logiku.
- **Services**: Sadrži kod koji povezuje aplikaciju sa poslužiteljem.
- **Shared**: Komponente koje su korištene na više mjesta u aplikaciji (npr. naslovna i bočna traka).

3.3 Backend i baza podataka

U većini web-aplikacija koriste se pozadinski procesi na poslužitelju (tzv. *backend*) te baze podataka. Za kraj ovoga rada ukratko ćemo opisati što smo koristili za backend i bazu podataka u našoj aplikaciji. Backend smo izgradili koristeći Visual Studio 2019. te Entity Framework Core. EF Core je razvojna okolina otvorenog koda koja omogućuje programerima da rade na aplikacijama koje zahtijevaju velike količine podataka koristeći samo objekte (kao instance klasa koje reprezentiraju tablice u bazama podataka u koje su spremaju ti podaci) umjesto da se fokusiraju direktno na rad sa bazama podataka. Dakle, Entity Framework za nas obavlja rad sa bazom podataka u pozadini i pri tome koristi tehniku ORM (eng. *Object Relation Mapping*). Više o EF Core-u može se pročitati u [4].

Skoro svaka komponenta u našoj aplikaciji koristi paket *axios* kako bi poslala zahtjev backendu preko kojeg bi napravila neke transformacije nad bazom podataka. Te transformacije mogu biti spremanje novog komentara u bazu ili dohvaćanje liste svih korisnika aplikacije i sl. Kako bi aplikacija ispravno radila, backend mora biti upaljen kako bi mogao odgovarati na te zahtjeve. Backend sadrži sedam kontrolera, za svaku tablicu u bazi po jedan.

Za kraj ovog rada prikazujemo i objašnjavamo dijagram baze podataka (slika 3.7).



Slika 3.7: Model baze podataka

Gornju sliku generirali smo unutar Visual Studia i prikazali koristeći *DGML Viewer* eksteziju. Na slici možemo vidjeti sedam tablica u bazi koje koristimo u našoj aplikaciji.

- **User**: Predstavlja korisnika aplikacije.
- **Board**: Predstavlja jedan projekt. Ima strani ključ na tablicu *Team* koji radi na tom projektu.
- **List**: Označava jednu kategoriju u projektu. Ima strani ključ na tablicu *Board* kojoj pripada.
- **Card**: Predstavlja zadatak. Ima strani ključ na korisnika koji izvršava taj zadatak i listu u kojoj se nalazi.
- **Team**: Jedan razvojni tim. Ima strani ključ na korisnika koji predstavlja autora.
- **TeamUserMapping**: Povezuje korisnika i razvojni tim. Korisnik aplikacije može biti u više timova te jedan tim može imati više korisnika (tzv. *n-n veza*).
- **Comment**: Predstavlja komentar unutar kartice. Ima strani ključ na nju.

Svim tablicama primarni ključ je identifikator *Id*.

Bibliografija

[1] *Agile aplikacija*, https://github.com/shenky1/React-Agile/tree/initial, 2020, posjećeno u siječnju 2020.

[2] *Dokumentacija paketa axios*, https://github.com/axios/axios, 2020, posjećeno u siječnju 2020.

[3] *Dokumentacija paketa reactstrap*, https://reactstrap.github.io/, 2020, posjećeno u siječnju 2020.

[4] *Entity Framework Core dokumentacija*, https://docs.microsoft.com/en-us/ef/core, 2020, posjećeno u siječnju 2020.

[5] *MDN web dokumentacija*, https://developer.mozilla.org/en-US/docs/Web, 2020, posjećeno u siječnju 2020.

[6] *Npm dokumentacija*, https://www.npmjs.com/, 2020, posjećeno u siječnju 2020.

[7] *React router dokumentacija*, https://reacttraining.com/react-router/web/, 2020, posjećeno u siječnju 2020.

[8] *Trello web-aplikacija*, https://trello.com/home, 2020, posjećeno u siječnju 2020.

[9] *Web stranice razvojnog okvira React*, https://reactjs.org/docs, 2020, posjećeno u siječnju 2020.

[10] A. Mardan, *React Quickly: Painless web apps with React, JSX, Redux, and GraphQL*, 2017.

[11] R. Wieruch, *The Road to learn React: Your journey to master plain yet pragmatic React.js*, 2017.

Sažetak

U ovom radu prezentirana je biblioteka React. Ona je napisana u programskom jeziku JavaScript, otvorenog je koda te je besplatna za korištenje.

U prvom poglavlju opisani su osnovni koncepti Reacta. Na samom početku objašnjeno je kako se provodi njegova instalacija te stvaranje prve aplikacije koristeći npm-ov paket *create-react-app*. Nakon toga se govorilo o komponentama, neovisnim gradivnim cjelinama svake React aplikacije te kako ih React koristi za izgradnju korisničkog sučelja. Opisane su metode životnog ciklusa komponente, njihova svojstva, unutarnje stanje te kako njihovim korištenjem više komponenti može međusobno komunicirati. Na kraju poglavlja objašnjeno je kako React upravlja događajima te novi koncept *hooks* funkcija koje uvelike olakšavaju razvoj web-aplikacija.

U drugom poglavlju su opisani neki dodatni paketi koji se koriste u većini React web-aplikacija: react-router-dom, reactstrap i axios.

U trećem poglavlju je za potrebe ovoga rada opisana složenija web-aplikacija za upravljanje razvojem proizvoljnih agilnih projekata. Na početku je opisana funkcionalnost same web-aplikacije, kako se ona upotrebljava i koje su njezine mogućnosti te je pokazana implementacija koncepata iz prvog poglavlja.

Summary

In this thesis React library was presented. React is free to use and open-source library written in JavaScript programming language.

In the first chapter, the main concepts of React were described. The chapter starts by installation of React and the creation of first application using the npm package *create-react-app* was explained. The main topic of the chapter are React components: independent, reusable building blocks. We also explain the way that React uses them to render and compose user interface. Its lifecycle methods were described, as well as properties, and the internal state of React components, along with its use for establishing communication of multiple components. At the end of the chapter, it was shown how React manages events, as well as *hooks*, functions that were recently added to React which made application development in React much easier.

In chapter two, several external packages were explained which are typically used within most React applications. Those packages are react-router-dom, reactstrap and axios.

In chapter three, implementation of a more complex web-application was described. Its purpose is to manage development of agile projects. It was explained how to use the application and its features. Also, concepts from chapter one are demonstrated in practice.

Životopis

Rođen sam 19. rujna 1994. godine u Zagrebu. Završio sam Osnovnu školu Markuševec 2009. godine nakon čega upisujem XV. gimnaziju u Zagrebu. Završio sam sa školovanjem 2013. godine te tada upisujem preddiplomski studij matematike na Prirodoslovno-matematičkom fakultetu u Zagrebu koji sam završio 2017.godine kada sam stekao titulu prvostupnika matematike. Neposredno nakon toga upisao sam diplomski studij Računarstva i matematike, također na Prirodoslovno-matematičkom fakultetu u Zagrebu.

Od srpnja 2018. godine radim studentski posao u AVL-AST kao frontend developer koristeći razvojnu okolinu Angular te biblioteku React.