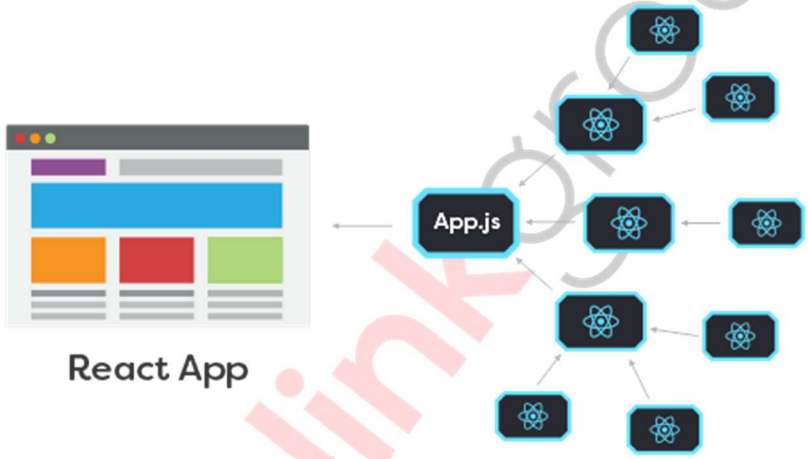
React komponente

Nakon priče o React elementima, spremni smo da detaljnije razmotrimo osobine centralnih figura koje čine korisničko okruženje React aplikacija. Naravno, reč je o komponentama, a nakon načelnog upoznavanja u uvodnoj lekciji ovoga modula, sada će komponentama biti posvećena puna pažnja.

Komponentna React arhitektura

Korisničko okruženje se posredstvom React aplikacije gradi kao hijerarhija komponenata. Aplikacije uglavnom poseduju jednu glavnu komponentu unutar koje se objedinjuju ostale komponente od kojih je sačinjeno korisničko okruženje (slika 11.1).



Slika 11.1. Komponentna arhitektura React aplikacije

U prethodnim lekcijama smo se već načelno upoznali sa React komponentama, tako da verovatno već posedujete osnovnu ideju o tome šta su komponente i šta one donose prilikom kreiranja React aplikacija. React komponente omogućavaju da se UI jedne aplikacije podeli na logičke, izolovane celine, koje je moguće upotrebljavati proizvoljan broj puta.

React komponente se mogu kreirati na dva načina:

- korišćenjem funkcija ili
- korišćenjem klasa.

Razlike između komponenata koje se dobijaju korišćenjem navedenih načina su primarno sintaksne. Drugim rečima, korišćenjem oba navedena pristupa dobijaju se komponente gotovo identičnih osobina. Razlika je u načinu kreiranja i pristupima koji se koriste za njihovo konfigurisanje. Mi ćemo se prvo pozabaviti kreiranjem React komponenata korišćenjem funkcija, dok će drugi deo ove lekcije biti posvećen klasnim komponentama React aplikacije.

Pitanje

React komponente je moguće kreirati korišćenjem:

a) objekata
b) klasa
c) običnih objekata
d) nizova

Objašnjenje:

React komponente se mogu kreirati na dva načina: korišćenjem funkcija i korišćenjem klasa.

Kreiranje React komponenata korišćenjem funkcija

Najjednostavniji način za kreiranje React komponente jeste definisanje JavaScript funkcije, koja kao svoju povratnu vrednost emituje React element:

```
function SayHelloWorld() {
  return <h1>Hello World</h1>;
}
```

Kako bi se ovakva komponenta prikazala unutar DOM strukture, neophodno je kreirati React element, koji će predstavljati ovakvu komponentu (*u prethodnoj lekciji je rečeno da React elementi mogu da predstavljaju DOM elemente, odnosno elemente koji postoje unutar HTML jezika, ali i korisnički definisane komponente*). Evo kako izgleda React element koji reprezentuje upravo kreiranu komponentu:

```
const elem = <SayHelloWorld />;
```

Primer je identičan već prikazanim primerima kreiranja React elemenata, a jedina razlika je u definisanju elementa sa nazivom koji upućuje na upravo kreiranu funkciju, koja predstavlja React komponentu.

Renderovanje je moguće obaviti kao i svaki put do sada:

```
function SayHelloWorld() {
  return <h1>Hello World</h1>;
}

const elem = <SayHelloWorld />;

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

Na ovaj način će prezentacija komponente biti umetnuta unutar `root` elementa u `index.html` dokumentu.

Prosleđivanje podataka korišćenjem svojstava (props)

React komponente koje se definišu pomoću funkcija mogu da kao svoju ulaznu vrednost prihvate i parametar kojim se definišu svojstva (**props**) komponente. Na taj način je vrlo lako moguće jednoj komponenti proslediti neke podatke:

```
function SayHello(props) {
  return <h1>Hello {props.name}</h1>;
}
```

Komponenta je sada delimično modifikovana. Promenjen je njen naziv, koji sada glasi `SayHello`. Ona prihvata i jedan parametar koji je imenovan kao `props`. Unutar komponente se pored reči *Hello* ispisuje i vrednost svojstva `name`, `props` objekta.

React Props

React Props su parametri koje React komponente mogu da prime definisanjem proizvoljnih HTML atributa.

Ovako kreiranoj komponenti sada se neki podaci mogu proslediti na sledeći način:

```
const elem = <SayHello name="Ben" />;
```

Bitno je da primetite da je na elementu koji predstavlja komponentu definisan atribut `name`, sa vrednošću `Ben`. Takav atribut će zajedno sa vrednošću da bude pretvoren u svojstvo `props` objekta. Unutar komponente se prosleđenoj vrednosti može pristupiti na sledeći način:

```
props.name
```

Evo kompletnog primera kreiranja i renderovanja React komponente sa jednim proizvoljnim svojstvom:

```
function SayHello(props) {
  return <h1>Hello {props.name}</h1>;
}

const elem = <SayHello name="Ben" />;

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

Primer će na stranici da proizvede ispis *Hello Ben*. *Hello* je definisano unutar komponente, dok se vrednost *Ben* komponenti prosleđuje prilikom njenog pridruživanja jednom React elementu.

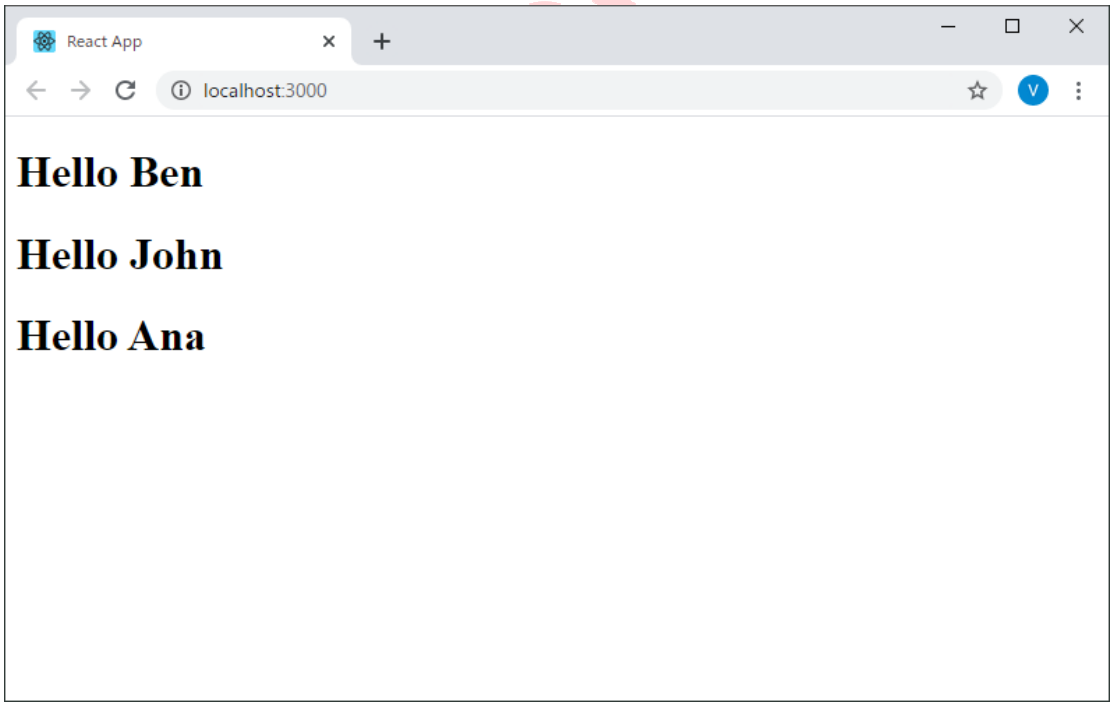
Ponovna upotrebljivost komponenata

Osnovna mogućnost komponenata jeste ponovna upotrebljivost. To praktično znači da je jednu React komponentu moguće upotrebiti proizvoljan broja puta. Takvo nešto će biti ilustrovano na primeru već viđene komponente:

```
function SayHello(props) {
  return <h1>Hello {props.name}</h1>;
}

const elem = <div>
  <SayHello name="Ben" />
  <SayHello name="John" />
  <SayHello name="Ana" />
</div>;

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

Sada je nešto ranije kreirana komponenta `SayHello` upotrebljena tri puta unutar jednog React elementa. Pri tome je svakoj komponenti prosleđena drugačija vrednost svojstva `name`. Sva tri elementa koji predstavljaju komponente objedinjeni su unutar jednog okružujućeg `div` elementa, zato što React elementi uvek moraju imati jedan koreni element.

Na ovaj način se unutar web pregledača dobija prikaz kao na slici 11.2.



Slika 11.2. Prikaz koji je produkt ponovne upotrebe jedne React komponente

Smeštanje jedne komponente unutar neke druge

React komponente je moguće smeštati unutar drugih komponenata i na taj način kreirati kompoziciju komponenata. Evo kako se efekat prikazan slikom 11.2. može postići kompozicijom komponenata:

```
function SayHello(props) {
  return <h1>Hello {props.name}</h1>;
}

function App() {
  return (<div>
    <SayHello name="Ben" />
    <SayHello name="John" />
    <SayHello name="Ana" />
  </div>);
}

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

U primeru se sada kreiraju dve komponente – `SayHello` i `App`. `App` je glavna komponenta aplikacije, unutar koje je za formiranje prikaza tri puta iskorišćena komponenta `SayHello`. Efekat će biti identičan kao na slici 11.2.

Grupisanje višelinijskog JSX koda nakon naredbe return

Obratite pažnju na to da je povratna vrednost funkcije `App()`, koja predstavlja React komponentu, postavljena unutar zagrada – (). Postavljanje povratne vrednosti unutar zagrada se praktikuje kada se pomoću JSX koda formira element čiji je sadržaj formatiran u više redova. Naime, ukoliko se povratna vrednost ne smesti unutar zagrada, neka izvršna okruženja mogu, odmah nakon prvog prelaska u novi red, automatski da ubace karakter za završavanje naredbe (tačka zapeta - ;).

Kreiranje komponenata korišćenjem klasa

Nešto ranije je rečeno da React obezbeđuje dva osnovna načina za kreiranje komponenata – korišćenje funkcija ili klasa. Do sada su prikazani primeri komponenata koje se kreiraju upotrebom običnih JavaScript funkcija, a u nastavku ćemo se upoznati sa pravilima za kreiranja klasnih React komponenata.

Klasne React komponente se predstavljaju klasama koje nasleđuju baznu klasu – `React.Component`. Evo kako može izgledati najjednostavnija komponenta prikazana na početku ove lekcije, kada se definiše pomoću klase:

```
class SayHelloWorld extends React.Component {
  render() {
    return <h1>Hello World</h1>;
  }
}
```

Osnovna razlika između funkcijskih i klasnih komponenata jeste sintaksa za njihovo kreiranje. Sve ostalo je identično, odnosno klasne komponente omogućavaju da se obavi sve ono što je već prikazano u dosadašnjem toku ove lekcije – ponovna upotrebljivost, kompozicija, mogućnost prihvatanja svojstava…

Kada su u pitanju razlike između funkcijskih i klasnih komponenata, za početak možete videti da se prezentacija jedne klasne komponente definiše kao povratna vrednost `render()` metode. Stoga kod, koji je do sada bio povratna vrednost funkcije, sada biva prebačen unutar `render()` metode klasne komponente.

Kada su u pitanju svojstva (**props**) koja se mogu prosleđivati komponentama, ona unutar klasnih komponenata postaju svojstva objektne instance koja se smešta unutar svojstva `props`:

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

ReactDOM.render(<SayHello name="Ben" />,
document.getElementById('root'));
```

Primer je identičan već prikazanom primeru u kome je ovakva komponenta kreirana korišćenjem JavaScript funkcije. Stoga će i efekat unutar dokumenta biti identičan.

Stanje komponenata

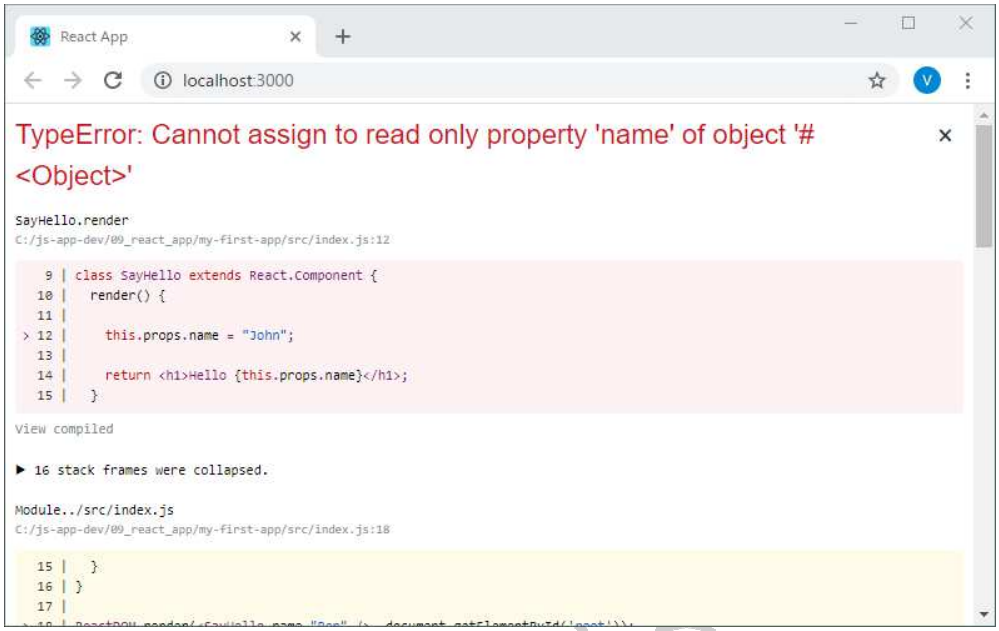
Dva osnovna načina za definisanje podataka komponenata jesu korišćenje:

a) svojstava (**props**) i
b) stanja (**state**).

U dosadašnjem toku lekcije ilustrovan je rad sa svojstvima. Mogli ste da vidite da je reč o mehanizmu koji omogućava da se React komponentama proslede neki podaci. Ipak, vrlo je bitno znati da su svojstva koja komponente dobijaju iz spoljašnjeg okruženja podaci koji su namenjeni samo za čitanje. Njihovu vrednost nije moguće menjati unutar komponenata:

```
class SayHello extends React.Component {
  render() {

    this.props.name = "John";

    return <h1>Hello {this.props.name}</h1>;
  }
}

ReactDOM.render(<SayHello name="Ben" />,
document.getElementById('root'));
```

Unutar komponente sada je obavljeno redefinisanje prosleđenog `name` svojstva. Nakon izgradnje aplikacije, unutar web pregledača ćete moći da vidite jasnu poruku o grešci (slika 11.3).



Slika 11.3. Greška koja se dobija prilikom pokušaja redefinisanja svojstava unutar React komponente

React svojstva su namenjena za predstavljanje podataka koji se tokom izvršavanja aplikacije neće promeniti. Ipak, postojanje dinamičkih podataka unutar JavaScript aplikacija apsolutni je imperativ, te je stoga potpuno jasno da React poseduje još jedan mehanizam koji se prilikom izgradnje aplikacija koristi za modelovanje podataka koji se menjaju. Takav mehanizam se unutar React aplikacije naziva stanje.

React stanje (**state**) omogućava da se definišu lokalni podaci jedne komponente, koji će se tokom izvršavanja aplikacije menjati. Svaka promena takvih podataka jeste signal za React, da je potrebno da osveži prezentaciju jedne komponente.

Evo kako može da izgleda jedna komponenta unutar koje su korišćenjem stanja definisani lokalni podaci:

```
class RandomNumGenerator extends React.Component {

  constructor(props) {
    super(props);
    this.state = { code: Math.random().toString(36).substring(7) };
  }

  render() {
    return <h1>Your random code is: {this.state.code}</h1>;
  }
}
```

Stanje komponente se definiše unutar konstruktora klase, tako što se postavlja vrednost svojstva **state**, koje poseduje svaka klasna React komponenta. Svojstvu `state` se dodeljuje objekat sa svojstvom `code`, čija će vrednost biti šestocifreni, nasumično generisani kod.

Konstruktor klasnih React komponenata

Sada prvi put unutar jedne klasne React komponente definišemo konstruktor. Naime, unutar JavaScript klasa nije obavezno definisati konstruktor. Kad se takvo nešto ne učini, klasa će imati podrazumevani konstruktor bez parametara. Ipak, kada se unutar jedne React komponente eksplicitno definiše konstruktor, kao što je to učinjeno u prikazanom primeru, neophodno je da on kao svoj ulazni parametar prihvati `props` objekat. Takođe, prva naredba unutar konstruktora treba da bude ona koja se odnosi na pozivanje konstruktora roditeljske klase, korišćenjem ključne reči `super`. Konstruktoru roditeljske klase je potrebno proslediti `props` svojstvo, kako bi React mogao na pravi način da izvrši konfigurisanje komponente.

Kako bismo pokazali da je ovako definisano stanje zaista i moguće promeniti za vreme izvršavanja aplikacije, primer će biti modifikovan na sledeći način:

```
class RandomNumGenerator extends React.Component {

  constructor(props) {
    super(props);
    this.state = { code: Math.random().toString(36).substring(7) };
    this.generateNewCode = this.generateNewCode.bind(this);
  }
  generateNewCode() {
    this.setState({ code: Math.random().toString(36).substring(7) });
  }
  render() {
    return (
      <div>
        <h1>Your random code is: {this.state.code}</h1>
        <button onClick={this.generateNewCode}>Generate new
code</button>
      </div>
    )
  }
}
```

Sada je unutar prezentacije komponente dodat još jedan element. Reč je o `button` elementu kojim će korisnik samostalno moći da inicira generisanje novog koda. U prikazanom primeru je neophodno da primetite sledeće:

- prezentacija koja se emituje iz `render()` metode upakovana je unutar jednog okružujućeg elementa, zato što prezentacija komponente može imati samo jedan koreni element,
- povratna vrednost `render()` metode je smeštena unutar zagrada – (), zato što se JSX definiše u više redova,

- na `button` elementu je obavljena pretplata na `click` događaj, a tom prilikom je `generateNewCode()` funkcija postavljena kao ona koja će obraditi događaj,
- unutar `generateNewCode()` metode je definisana logika koja će generisati novi kod i na taj način promeniti unutrašnje stanje komponente.

Bitno je da primetite da se stanje jedne komponente ne može promeniti direktnom intervencijom nad svojstvima koja su upisana unutar stanja, već je neophodno koristiti specijalnu metodu koja je namenjena za obavljanje takvog posla – **setState()**.

Pozivanjem metode `setState()` daje se signal komponenti da je potrebno da izmeni svoju prezentaciju, što će kao posledicu imati ponovno renderovanje onih delova koji su se promenili, pa će na kraju korisnik unutar svog web pregledača videti takvo izmenjeno, novo stanje (animacija 11.1).

Animacija 11.1. Promena unutrašnjeg stanja React komponente

Obrada događaja unutar komponenata

Upravo prikazani primer pokazao je i kako se može rukovati događajima unutar klasne komponente. Obavljena je pretplata na `click` događaj `button` elementa, na već prikazani način, a kao funkcija koja će takav događaj obraditi definisana je metoda `generateNewCode()` koja je smeštena unutar klase koja predstavlja komponentu.

Ipak, bitno je da znate da se klasne metode u JavaScript jeziku podrazumevano ne vezuju za instance takve klase. To za rezultat ima nemogućnost korišćenja ključne reči `this` unutar metoda za obradu događaja. Zbog toga je u primeru obavljeno vezivanje instance (objekta) za metodu `generateNewCode()`, korišćenjem sledećeg JavaScript koda:

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

Vezivanje metode i instance se obavlja korišćenjem JavaScript metode `bind()`. Metoda `bind()` kreira novu funkciju, unutar koje će ključna reč `this` da ukazuje na objekat koji se metodi `bind()` prosledi prilikom pozivanja. Na taj način će, unutar metode za obradu događaja, biti omogućeno korišćenje ključne reči `this`, koja će ispravno da ukazuje na trenutnu instancu klase.

Kretanje podataka kroz React aplikaciju

React aplikacije su uglavnom sastavljene iz većeg broja komponenata, koje grade jednu kompoziciju. Tipično, neke komponente poseduju svoje stanje, dok druge sopstveno stanje ne poseduju. Tako se može reći da React omogućava kreiranje dva osnovna tipa komponenata:

- komponente koje poseduju sopstveno stanje (engl. *stateful components*) i
- komponente bez sopstvenog stanja (engl. *stateless components*).

Međusobna saradnja komponenata i deljenje podataka između njih osnova je funkcionisanja složenijih React aplikacija. U takvim situacijama nijedna komponenta ne može znati da li neka druga komponenta poseduje sopstveno stanje. Stanje je lokalno i enkapsulirano unutar jedne komponente i nijedna komponenta, osim one koja ga poseduje, ne može mu direktno pristupiti. Sada se kao logično nameće pitanje – *kako se podaci kreću kroz React aplikaciju, s obzirom na to da je stanje lokalno*?

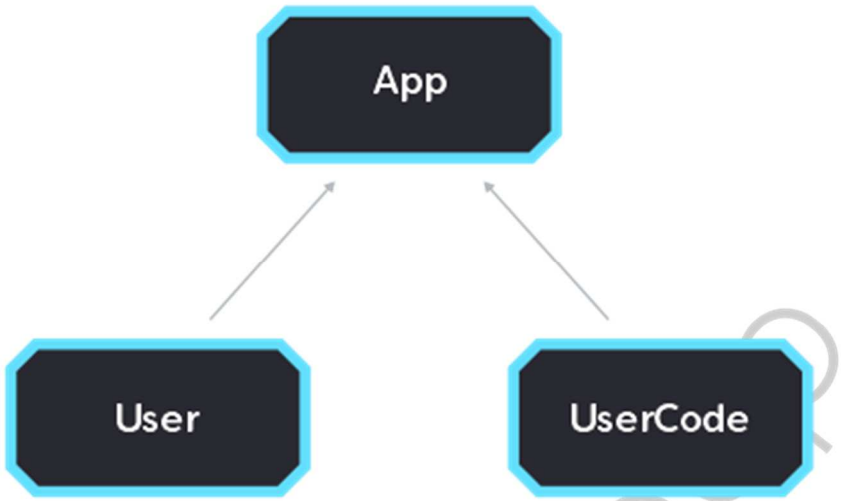
Jedna komponenta koja poseduje sopstveno stanje može da odluči da takvo stanje prosledi nekoj od komponenata koje se nalaze unutar nje. Prosleđivanje se obavlja korišćenjem već viđenog mehanizma, koji podrazumeva upotrebu svojstava (props). Na taj način komponenta koja prima podatke neće moći da zna da li su takvi podaci stanje roditeljske komponente ili možda neka svojstva koja je i ona dobila iz spoljašnjeg okruženja.

Kretanje podataka kroz React komponente i njihova međusobna saradnja biće prikazani na sledećem primeru:

```
class User extends React.Component {

  render() {
    return (
      <div>
        <p>{this.props.user.name}</p>
        <p>{this.props.user.email}</p>
        <p>{this.props.user.balance}</p>
        <p>{this.props.user.code}</p>
      </div>
    )
  }

}

class UserCode extends React.Component {

  render() {
    return (
      <div>
        <p>Change user code</p>
        <button onClick={this.props.generateNewCode}>Generate new
code</button>
      </div>
    )
  }
```

```
}

class App extends React.Component {

  constructor(props) {
    super(props);
    this.state = {
      name: "Ben Torrance",
      email: "ben@email.com",
      balance: 55353.93434,
      code: "5n4n5n"
    };

    this.generateNewCode = this.generateNewCode.bind(this);
  }

  generateNewCode() {
    this.setState({ code: Math.random().toString(36).substring(7) });
  }

  render() {
    return (
      <div>
        <User user={this.state} />
        <UserCode generateNewCode={this.generateNewCode} />
      </div>
    )
  }

}


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

Prikazanim kodom kreiraju se tri React komponente – glavna komponenta `App` i dva dodatne komponente – `User` i `UserCode`. Komponente `User` i `UserCode` koriste se za renderovanje delova korisničkog okruženja. One se objedinjuju unutar komponente `App`, a zatim se ona umeće unutar HTML dokumenta. Tako je struktura komponenata u ovom primeru kao na slici 11.4.



Slika 11.4. Struktura komponenata iz primera

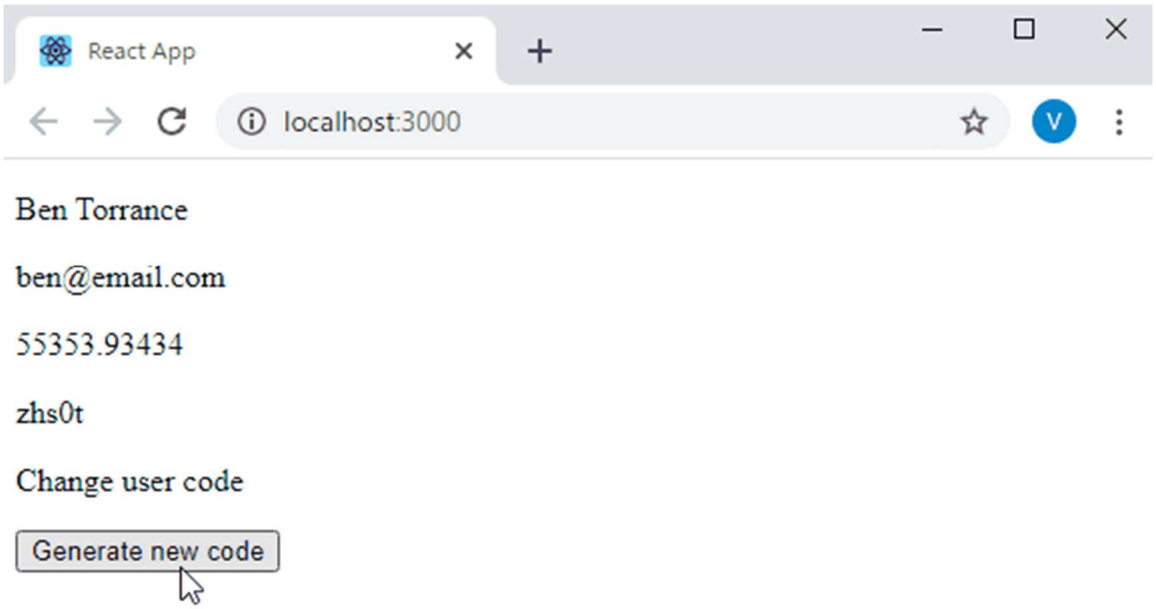
Prikazani primer ilustruje dve veoma važne osobine komunikacije između komponenata: prosleđivanje podataka iz komponente `App` u komponentu **`User`** i obradu događaja komponente **`UserCode`** unutar komponente **`App`.**

Prosleđivanje podataka iz komponente `App` u komponentu `User`

Jedina komponenta koja poseduje stanje u prikazanom primeru jeste komponenta `App`. Stanje se ogleda u podacima kojima se predstavlja jedan korisnik (`user`). Takvi podaci se prosleđuju komponenti `User` korišćenjem svojstava (props). Tako komponenta `User` dobija podatke od svoje roditeljske komponente bez ikakvog znanja o njihovom poreklu.

Obrada događaja komponente `UserCode` unutar komponente `App`

Komponenta `UserCode` je kreirana kako bi se omogućila izmena svojstva `code`, koje je sastavni deo podataka o korisniku. `UserCode` komponenta poseduje samo jedan `button` element sa pretplatom na `click` događaj, ali funkcija za obradu takvog događaja nije definisana unutar `UserCode` komponente, već unutar roditeljske `App` komponente. Tako primer ilustruje korišćenje svojstava (props) za prosleđivanje referenci na funkcije. Korišćenjem svojstva `generateNewCode`, komponenti `UserCode` je prosleđena referenca na funkciju koja se nalazi unutar roditeljske komponente. Klikom na `button` element za izmenu korisničkog koda biće aktivirana `generateNewCode()` metoda iz `App` klase, a zatim će unutar takve metode da bude ažurirano samo jedno svojstvo (`code`) stanja glavne komponente. Na kraju, možete da vidite da se izmene adekvatno propagiraju i unutar `User` komponente koja se koristi za prikaz podataka jednog korisnika (animacija 11.2).



Animacija 11.2. Primer komunikacije između više React komponenata

Definisanje komponenata unutar zasebnih fajlova

U jednoj od prethodnih lekcija, prilikom generisanja React projekta korišćenjem `create-react-app` alata, mogli ste da vidite da su komponente bile smeštane unutar zasebnih `.js` fajlova. Smeštanje komponenata unutar zasebnih fajlova omogućava bolju organizovanost koda i lakše održavanje, te se stoga takvo nešto uvek praktikuje prilikom razvoja realnih React aplikacija.

Kako bi se jedna React komponenta smestila unutar zasebnog fajla, prvo je potrebno kreirati prazan `.js` fajl. Zatim je klasu kojom se komponenta predstavlja potrebno prebaciti unutar takvog fajla:

```
class User extends React.Component {

    render() {
        return (
            <div>
                <p>{this.props.user.name}</p>
                <p>{this.props.user.email}</p>
                <p>{this.props.user.balance}</p>
                <p>{this.props.user.code}</p>
            </div>
        )
    }
}
```

Ipak, ovo nije dovoljno kako bi komponenta uspešno funkcionisala. Neophodno je na početku dokumenta uključiti i dva React modula:

```
import React from 'react';
import ReactDOM from 'react-dom';

class User extends React.Component {

    render() {
        return (
            <div>
                <p>{this.props.user.name}</p>
                <p>{this.props.user.email}</p>
                <p>{this.props.user.balance}</p>
                <p>{this.props.user.code}</p>
            </div>
        )
    }

}
```

Na ovaj način komponenta će uspešno funkcionisati, ali neće moći da se iskoristi izvan fajla u kome je definisana. Kako bi se omogućilo njeno korišćenje izvan fajla, nju je neophodno eksportovati, dodavanjem odgovarajuće naredbe na kraj fajla:

```
import React from 'react';
import ReactDOM from 'react-dom';

class User extends React.Component {

    render() {
        return (
            <div>
                <p>{this.props.user.name}</p>
                <p>{this.props.user.email}</p>
                <p>{this.props.user.balance}</p>
                <p>{this.props.user.code}</p>
            </div>
        )
    }

}

export default User;
```

Sada je `User` komponenta definisana unutar zasebnog fajla. Poseduje sve elemente koji će omogućiti njeno nesmetano korišćenje. Kako bi se, kao i do sada, iskoristila unutar `App` komponente, nju je neophodno prethodno importovati:

```
import User from './User';
```

React ToDo aplikacija

Sve što smo o aplikaciji React naučili u prethodnim lekcijama iskorišćeno je za praktičnu realizaciju ToDo aplikacije. Kompletan proces realizacije je prikazan u video-lekciji koja se nalazi u prilogu. Stoga ne propustite da pogledate na koji način se React razvoj razlikuje od već viđenih pristupa koji su podrazumevali korišćenje Vanilla JS-a i Vuea.

Rezime

- Korisničko okruženje se posredstvom React aplikacije gradi kao hijerarhija komponenata.
- React komponente se mogu upotrebljavati proizvoljan broj puta.
- React komponente se mogu kreirati korišćenjem funkcija ili klasa.
- React komponenti je moguće proslediti neke parametre korišćenjem svojstava (props).
- Komponenta koja se kreira u obliku funkcije, svojstva (props) dobija kao ulazni parametar takve funkcije.
- React Props su parametri koje React komponente mogu da prime definisanjem proizvoljnih HTML atributa.
- React komponente je moguće smeštati unutar drugih komponenata i na taj način kreirati kompoziciju komponenata.
- Kada se pomoću JSX koda formira element čiji je sadržaj formatiran u više redova, praktikuje se njegovo smeštanje unutar zagrada ispred naredbe `return`.
- Klasne React komponente se predstavljaju klasama koje nasleđuju baznu klasu – `React.Component`.
- Prezentacija jedne klasne komponente definiše se kao povratna vrednost `render()` metode.
- Svojstva (props) unutar klasnih komponenata postaju svojstva objektne instance koja se smešta unutar svojstva `props`.
- Dva osnovna načina za definisanje podataka komponenata jesu korišćenje: svojstava (props) i stanja (state).
- Svojstva (props) su podaci koji su namenjeni samo za čitanje, odnosno njihovu vrednost nije moguće promeniti unutar komponenata.
- React stanje (state) omogućava da se definišu lokalni podaci jedne komponente, koji će se tokom izvršavanja aplikacije menjati.
- Stanje komponente se definiše unutar konstruktora klase, tako što se postavlja vrednost svojstva `state`.
- Kada se unutar jedne React komponente eksplicitno definiše konstruktor, neophodno je da on kao svoj ulazni parametar prihvati `props` objekat i da njega prosledi prilikom poziva konstruktora roditeljske klase.
- Izmena podataka upisanih u stanje komponente obavlja se korišćenjem metode `setState()`.
- Klasne metode se u JavaScript jeziku podrazumevano ne vezuju za instance takve klase, te je stoga prilikom obrade događaja unutar klasnih komponenata neophodno metodi za obradu događaja ručno dodeliti vrednost ključne reči `this`, korišćenjem JavaScript metode `bind()`.
- Smeštanje komponenata unutar zasebnih `.js` fajlova omogućava bolju organizovanost koda i lakše održavanje.