ReactJS - reaktivna arhitektura

Dragi polaznici, ova prezentacija nije primarno namijenjena za učenje već služi kao **pomoćni materijal**. Kao takva, ne može zamijeniti predavanja, literaturu kao niti vašu (pro)aktivnost na nastavi i konzultiranje s predavačem u slučaju potencijalnih pitanja i/ili nejasnoća.

ReactJS - reaktivna arhitektura

5. dio programa osposobljavanja za zanimanje *Front-end developer*

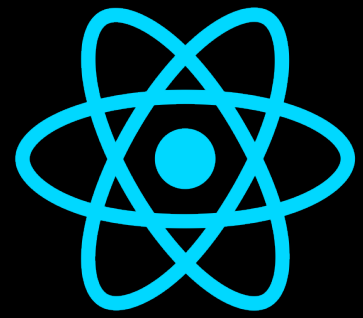
- 20 sati predavanja, 40 sati vježbi
- parcijalni ispit
- završni ispit

Cjeline

	Predavanja	Vježbe
Uvod u reaktivnu arhitekturu	4	10
Osnovni koncepti ReactJS-a	8	15
Napredni ReactJS	8	15

Literatura

- Wieruch,Robin. 2018. *The Road to learn React: Your journey to master plain yet pragmatic React.js.* CreateSpace Independent Publishing
- https://reactjs.org

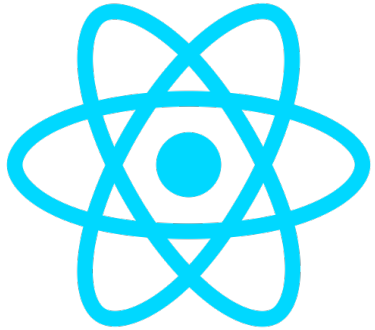




Uvod u reaktivnu arhitekturu

JavaScript

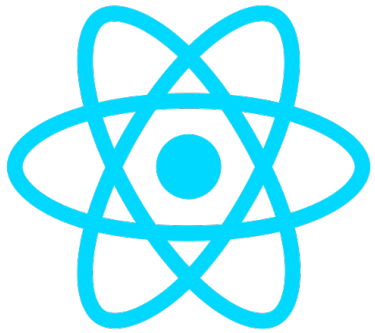
- vrlo moćan programski jezik, no samim time i složen za korištenje
- većina programskih jezika ima stroge konvencije i standarde pisanja koda, ali ne i JavaScript
- programiranje u čistom JavaScriptu postaje još složenije
- danas postoje mnoge biblioteke koje, definiranjem vlastitih konvencija i standarda, pojednostavljuju razvoj mrežnih aplikacija u JavaScriptu
- trenutno najpopularnija rješenja su Angular, React i Vue.js

ReactJS



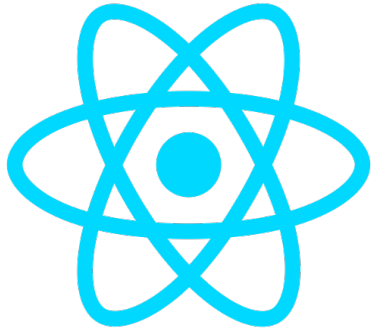
- vrlo popularna JavaScript biblioteka
- služi za izradu korisničkih sučelja (engl. *user interface - UI*)
- pomoću malih izoliranih dijelova koda (komponenti) stvaramo kompleksan, efikasan i fleksibilan UI
- razvio: Facebook Inc. 2013. godine

ReactJS



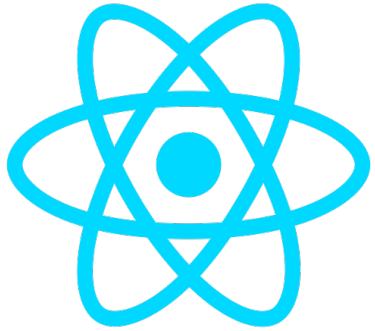
- razlikuje se od ostalih rješenja po tome što ostala rješenja imaju dodatke koji ih čine kompletnim razvojnim okvirima (engl. *frameworks*), dok je React samo programska biblioteka (engl. *library*) za izradu korisničkih sučelja

ReactJS



+ lakše ga je integrirati u već postojeća programska rješenja
+ velika i aktivna zajednica
+ podrška velike tvrtke (Facebook ga je stvorio te ga koristi u implementaciji Facebooka, Instagrama, WhatsAppa)
+ lako se uči

ReactJS



- za dodatnu funkcionalnost mora se kombinirati s drugim programskim bibliotekama
- React okolina stalno se mijenja
- postoje prijavljeni SEO (engl. *search engine optimization*) problemi

ReactJS - alternative

Angular

+ najzreliji od navedena 3 *frameworka*
+ detaljna dokumentacija
+ kompletna razvojna okolina
+ nove (i poboljšane) verzije *frameworka* izlaze vrlo često

Angular

- teško se uči
- nove verzije *frameworka* izlaze vrlo često - unose nove paradigme i stvaraju poteškoće prilikom prelaska sa stare na novu verziju

Vue.js

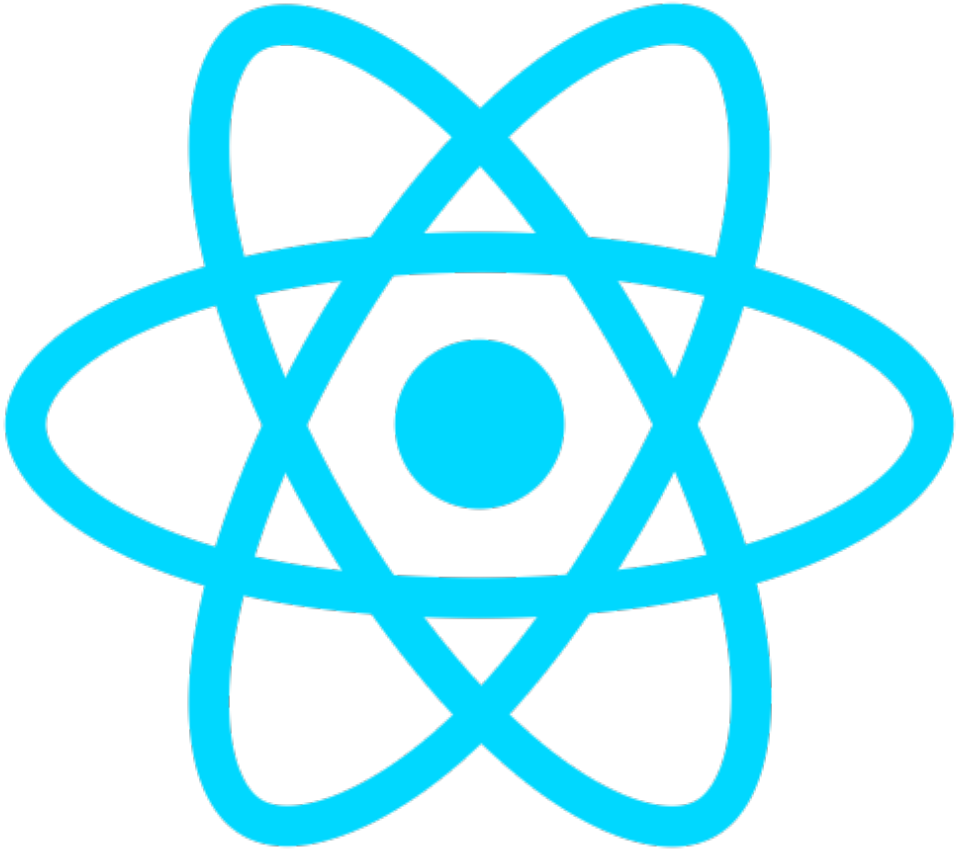
+ *lightweight* - samo 18kB nakon kompresije
+ omogućava veću fleksibilnost te ga je stoga lakše naučiti nego React ili Angular
+ lako je izraditi predloške za višekratnu upotrebu (engl. *reusable templates*)
+ ima sličnosti i s Angularom i s Reactom

Vue.js

- nedostatak resursa
- brza evolucija
- velika fleksibilnost dopušta lošiju kvalitetu kôda te samim time konačnog rješenja

Zašto React?

- popularnost
- zrelost
- fleksibilnost
- lakoća učenja
- aktivna zajednica

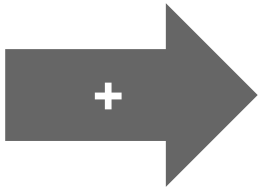


Zašto React?

U *Vanilla JavaScript-u* teško je reflektirati promjene u stanju aplikacije (promjene nastale korisničkom interakcijom)

Jednostavnija izrada i održavanje aplikacije nam ostavlja više vremena za implementiranje poslovne logike

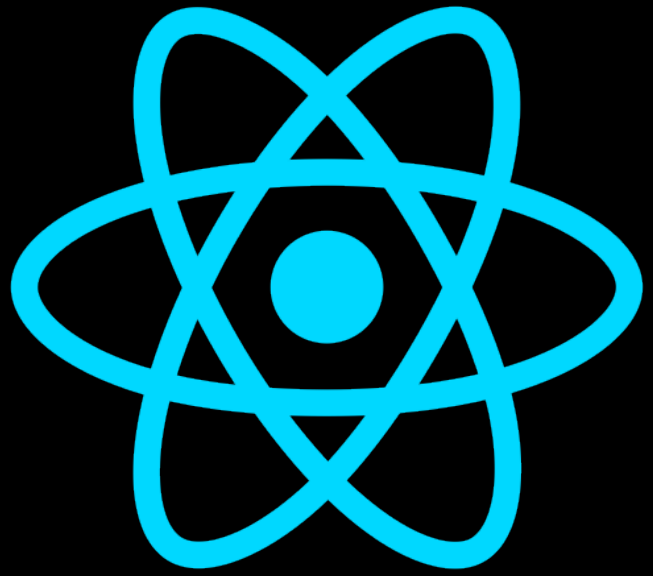
Velika i aktivna zajednica



Rezultirajući kôd ima bolje performanse jer se u pozadini koristi zrela i optimizirana biblioteka

Uvod u reaktivnu arhitekturu

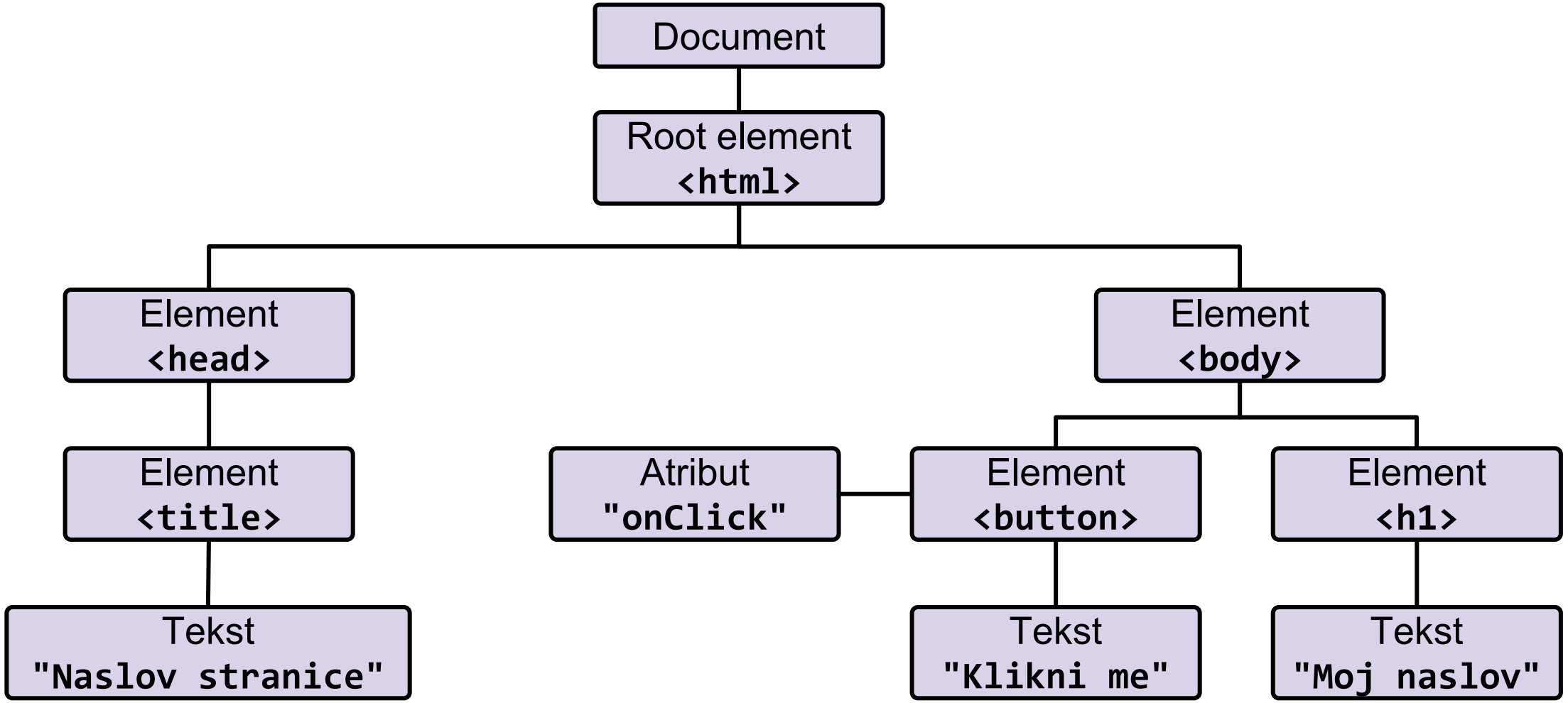
Virtual DOM



DOM

- DOM = *Document Object Model*
- W3C (*World Wide Web Consortium*) standard za pristupanje i ažuriranje sadržaja, strukture i stila dokumenata
- HTML DOM = apstrakcija HTML strukture stranice
- HTML elementi su posloženi u hijerarhijskom obliku (stablu), gdje svaki čvor stabla ima čvor roditelj (*parent*) i/ili čvor dijete (*child*)

HTML DOM stablo



HTML DOM

- HTML DOM je standard za dobivanje, promjenu, dodavanje ili brisanje elemenata HTML-a
- to znači da pomoću JavaScripta možemo kreirati dinamičke web-stranice:
  - dodavati, mijenjati, brisati elemente HTML-a i njihove atribute
  - mijenjati CSS web-stranice
  - dodavati ili mijenjati reakcije elemenata HTML-a na korisničku interakciju

HTML DOM - problem

- prilikom bilo koje promjene, generira se i prikazuje cijeli novi DOM
- stabla današnjih web-aplikacija su velika i dinamička, što znači da se DOM često ažurira
- to utječe na performanse aplikacije:
  - potrebno je puno procesorskog vremena
  - uvelike usporava aplikaciju

Rješenje:

React Virtual DOM!

Virtual DOM

- u svakom trenutku React rukuje dvjema instancama Virtualnog DOM-a:
    1. ažurno stanje Virtualnog DOM-a
    2. prethodno stanje Virtualnog DOM-a
- React koristi algoritam koji se naziva *The Diffing Algorithm* za uspoređivanje instanci Virtualnog DOM-a u svrhu pronalaženja najmanjeg broja razlika stanja prije i poslije ažuriranja
- React ažurira samo DOM elemente koji su se doista promijenili, ne dirajući ostale elemente

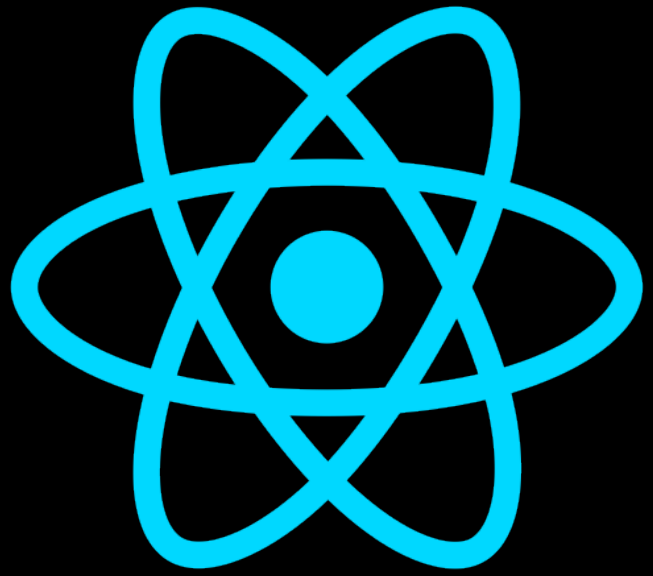
Primjer

- ako se na listi od deset zadataka želi označiti jedan kao završen, većina programskih okvira (*frameworka*) će nakon toga ponovo izgraditi cijelu listu, što je deset puta više nego što je potrebno
- React će, zbog implementacije Virtualnog DOM-a i *diffing* algoritm,a promijeniti samo jedan element u listi, te ostatak ostaviti nepromijenjenim
- na ovako malom primjeru to se i ne čini skupo, no današnje web-aplikacije koriste velik broj kompleksnih manipulacija ogromnog DOM stabla što dovodi do drastičnog porasta broja nepotrebnih ažuriranja

Literatura

- https://www.w3schools.com/js/js_htmldom.asp
- https://reactjs.org/docs/faq-internals.html
- https://reactjs.org/docs/reconciliation.html
- https://medium.com/coffee-and-codes/hey-react-what-is-the-virtual-dom-466ec333bf9a

Uvod u reaktivnu arhitekturu



ReactJS - razvojni alati

Razvojni alati

- *Command-line interface*
- Node.js
- Visual Studio Code

Command-line interface (CLI)

Command-line interface je program u kojem korisnik daje naredbe računalu u obliku uzastopnih redaka teksta. Najjednostavniji Windows *command-line interface* je Command Prompt.

Možemo koristiti Command Prompt ili Visual Studio Code Terminal.

Node.js

Potreban nam je kao JavaScript *runtime*, a s njim dolazi i snažan upravitelj paketima (engl. *package manager*), tj. programskim knjižnicama i razvojnim okvirima za node.js - **npm** (engl. *node package manager*).

Visual Studio Code

Odličan i besplatan uređivač kôd optimiziran za izradu modernih web-aplikacija. Dolazi s integriranom podrškom za verzioniranje kôda pomoću Gita te mnoštvom besplatnih ekstenzija koje se mogu instalirati da bi iskustvo programiranja bilo bolje.

Create React App

- Create React App je biblioteka koja omogućuje kreiranje React aplikacije samo jednom naredbom u CLI-ju:

```
npx create-react-app my-app
```

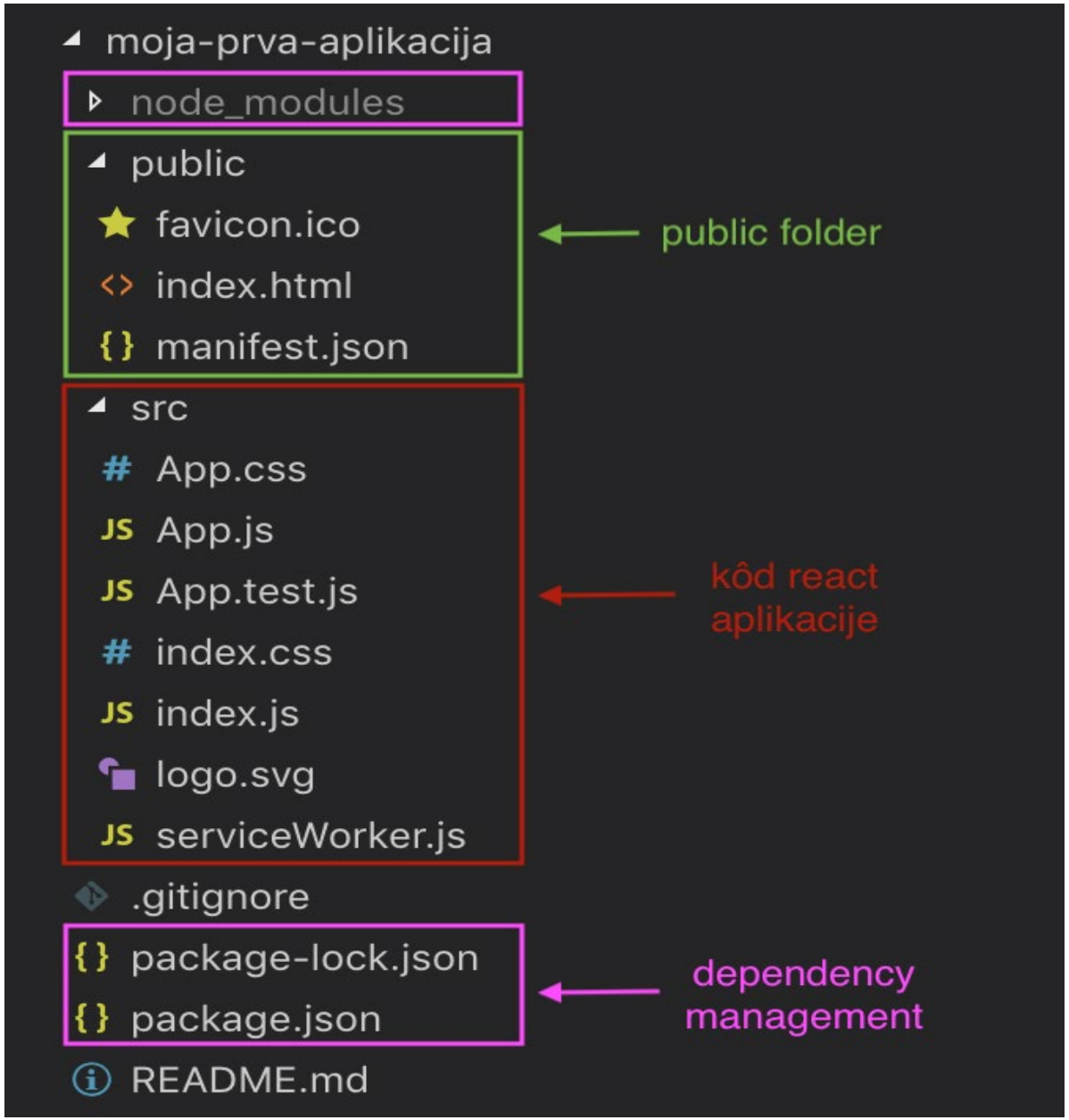
- rezultat ove naredbe je direktorij `my-app` unutar kojeg je organizirana cijela React aplikacija
- nije potrebno instalirati niti postavljati *Weback* i *Babel*, ti alati su već konfigurirani i aplikacija je spremna za pokretanje

Create React App

- kreirana aplikacija dolazi s nekoliko unaprijed definiranih skripti:
  - pokretanje aplikacije za lokalni razvoj: `npm start`
  - pokretanje testova: `npm run test`
  - pripremanje aplikacije za produkcijsku okolinu: `npm run build`
  - pokretanje aplikacije u produkcijskoj okolini: `serve -s build`
  - odvajanje od *defaultnih* postavki koje nudi *Create React App* (Webpack i Babel konfiguracija su sakrivene po *defaultu*): `npm run eject`

Vježba 5.1: Prva React Aplikacija

Struktura React aplikacije:



Struktura React aplikacije

1. `package.json`

- u ovoj datoteci možemo vidjeti eksterne pakete i skripte koje su dodane u našu aplikaciju
- trenutno su tu samo osnovne stvari koje naredba **`npx create-react-app`** radi za nas
- razvojem aplikacije i instalacijom dodatnih paketa **`package.json`** datoteka će se polako proširivati
- svi paketi instalirani su unutar direktorija **`node_modules`**

Struktura React aplikacije

2.public/index.html

- `<div id="root"></div>`
- ovdje će se biti ugniježđena naša React aplikacija

3.src/index.js

- `ReactDOM.render(<App />, document.getElementById('root'));`
- Ovom naredbom je React komponenta App postavljena na poziciju opisanu u prethodnoj točki

Struktura React aplikacije

`4.src/App.js`

- React komponenta koja predstavlja osnovnu komponentu i početnu točku prilikom izrade React web-aplikacije
- unutar ove komponente gnijezdimo druge React komponente te tako gradimo aplikaciju

Deploy React aplikacije

- kako bi se React aplikacija lokalno pokrenula u produkcijskom okruženju, potrebno je utipkati sljedeće naredbe:

```
1. npx create-react-app my-app
2. cd my-app
3. npm run build
4. serve -s build
```

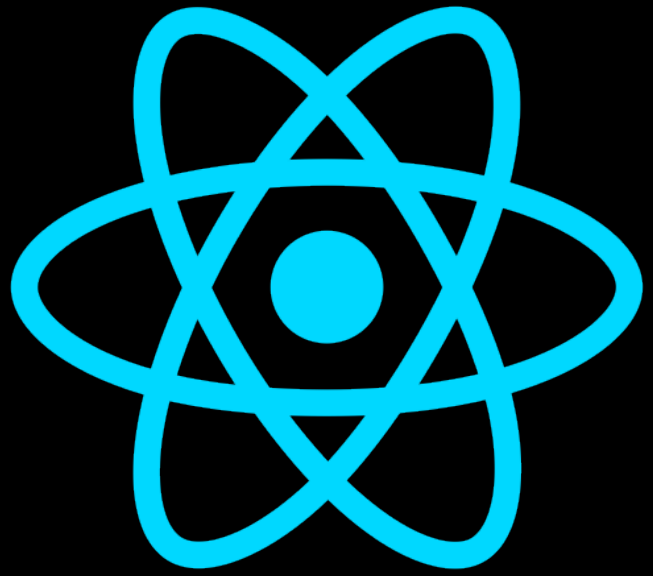
- otvoriti internetski preglednik na stranici http://<ip-adresa>:5000

Postavljanje React aplikacije na git

- React aplikacija sadrži mnogo dodatnih paketa instaliranih preko npm-a (sama React biblioteka nalazi se na npm-u)
- ti paketi nalaze se unutar direktorija node_modules
  - `node_modules` direktorij u sebi sadrži desetke tisuća datoteka
  - zbog toga sve akcije nad njime traju izrazito dugo (kopiranje, čak i brisanje direktorija)
  - zbog toga se direktorij `node_modules` ne postavlja na git repozitorij, već čuva samo lokalno
  - svi paketi zapisani su u datoteci `package.json` te se mogu jednostavno instalirati na drugom računalu

Literatura

- https://reactjs.org/docs/create-a-new-react-app.html
- https://github.com/facebook/create-react-app
- https://code.visualstudio.com/docs/languages/javascript
- https://code.visualstudio.com/docs/nodejs/reactjs-tutorial
- https://facebook.github.io/create-react-app/docs/setting-up-your-editor
- https://facebook.github.io/create-react-app/docs/using-the-public-folder



Uvod u reaktivnu arhitekturu

Dependency Management

npm - *node packet manager*

- npm je najveći svjetski registar besplatnih biblioteka i paketa
- sadrži više od 800 000 paketa
- mnogi programeri koriste npm
  - kako bi koristili javne *open-sourced* biblioteke
  - kako bi objavili vlastite biblioteke
- mnoga poduzeća koriste npm
  - na npm-u je moguće otvoriti privatni registar koji se može koristiti samo unutar organizacije

npm

- npm dolazi uz instalaciju Node.js-a, što znači da je potrebno instalirati Node.js kako bi se instalirao npm
- npm dolazi s vlastitim CLI-jem koji se može koristiti za instalaciju npm paketa:

```
npm install <ime-paketa>
```

- iako je najčešća primjena npm-a instalacija paketa, može se koristiti i za ažuriranje, konfiguriranje i brisanje paketa

npm

- uz instalaciju odabranog paketa, npm će instalirati i sve dodatne pakete koji su potrebni da bi odabrani paket funkcionirao ispravno (te pakete zovemo *dependencies*)
- paketi se na npm-u nalaze u arhiviranim datotekama
  - svaki paket sadrži metapodatke koji ga opisuju
  - metapodaci uključuju: naziv paketa, verziju paketa i listu svih *dependencyja*

package.json

- svi npm paketi koji pripadaju nekoj aplikaciji zapisani su u datoteci koja se zove *package.json*
- svaki dokument package.json sadrži atribute:
  - *name*: ime aplikacije
  - *version*: trenutna verzija aplikacije
  - *dependencies*: lista npm paketa koji pripadaju trenutnoj aplikaciji

package.json - primjer

```
{
 "name": "moja-prva-aplikacija",
 "version": "0.1.0",
 "private": true,
 "dependencies": {
   "react": "^16.8.6",
   "react-dom": "^16.8.6",
   "react-scripts": "3.0.1"
 },
"scripts": {
   "start": "react-scripts start",
   "build": "react-scripts build",
   "test": "react-scripts test",
   "eject": "react-scripts eject"
 },
}
```

package.json

- kako bi se instalirali svi *dependencies* paketi zapisani u package.json datoteci, potrebno je pozvati sljedeću naredbu:

```
npm install
```

- svi paketi bit će instalirani u direktoriju *node_modules*
- za dodavanje paketa u package.json datoteku, potrebno je pozvati sljedeću naredbu:

```
npm install --save <ime-paketa>
```

npm - osnovne naredbe

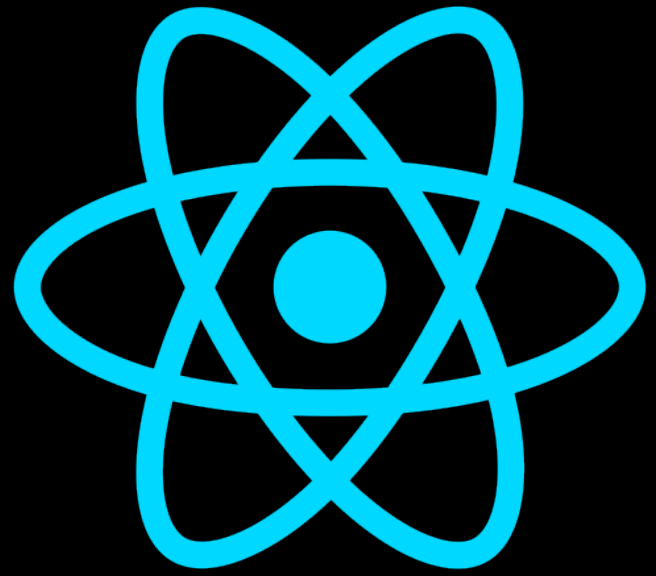
- **`npm install -g <ime-paketa>`** - instalacija paketa *globalno* - paket se može koristiti bilo gdje na računalu
- **`npm install --save <ime-paketa>`** - instalacija paketa u trenutni direktorij i zapisivanje u pripadajući package.json
- **`npm install --save-dev <ime-paketa>`** - instalacija paketa koji želimo koristiti samo prilikom razvoja, ali ne i prilikom izvršavanja aplikacije u produkciji
- **`npm list -g`** - lista globalno instaliranih paketa
- **`npm list`** - lista paketa instaliranih u trenutnom direktoriju
- **`npm uninstall -g <ime-paketa>`** - deinstalacija paketa iz *globalnog* direktorija
- **`npm uninstall --save <ime-paketa>`** - deinstalacija paketa i brisanje zapisa iz pripadajuće package.json datoteke

Vježba 5.2: *Dependency Management*

Literatura

- https://docs.npmjs.com/about-npm/
- *The Road to learn React*, str 18.

Uvod u reaktivnu arhitekturu



React Modules

JavaScript Modules

- JavaScript moduli su samostalni i odvojeni dio koda koji obavlja neku funkciju
- možemo ih dodati, mijenjati ili ukloniti prema potrebi, a pritom se ne narušava rad cijelog sustava
- ideja je rascjepkati kôd u više manjih datoteka koji se kasnije spoje u željenu funkcionalnost programa (aplikacije)
- jedna datoteka može sadržavati više od jednog modula

JavaScript Modules

- glavne prednosti:
  - održavanje - programski kôd puno je lakše izmijeniti kada je izdvojen u zasebnu cjelinu
  - mogućnost ponovne upotrebe - isti dio programske logike moguće je iskoristiti na više mjesta bez potrebe da se ista logika programira više puta
  - mogućnost korištenja vanjskih biblioteka (biblioteka koje je netko drugi napisao i objavio na javnom registru poput *npm*-a)

Import/export

Kako bi se programski kôd napisan u jednoj datoteci mogao koristiti u drugoj datoteci, potrebno ga je:

- ***exportati*** iz izvorne datoteke i
- ***importati*** u datoteci u kojoj ga želimo iskoristiti

Export

Dva su načina za napraviti *export*:

- ***default export*** - možemo imati samo jedan *default export* po datoteci, kod *importa* moramo specificirati ime *import* modula
- ***named export*** - možemo imati više *named exporta* po datoteci, ali kod *importa* moramo znati ime svakog modula

Default export

Person.js

```
const person = {
    name: "Max"
};
export default person;
```

Named export

Utility.js

```
export function sum(a, b) {
  return a + b;
}
export const num1 = 5;
export const num2 = 25;
```

Default / named import

App.js

```
import Person from './Person';
import { sum, num1, num2 } from './Utility';
```

Import/export

Export	Import
default export	`import Person from './Person';` `import BiloKojeIme from './Person';`

named export	`import { num1 } from "./Utility";`
	`import { num2 as BiloSto } from "./Utility";`
	`import * as bundled from "./Utility";`

- sami biramo ime
- imena su definirana *exportom*
* nam daje mogućnost da sve *exportane* funkcije i varijable stavimo u objekt kojem sami biramo ime
- prilikom *importa* ne moramo pisati ekstenziju datoteke (.js) iz koje *importamo* stvari

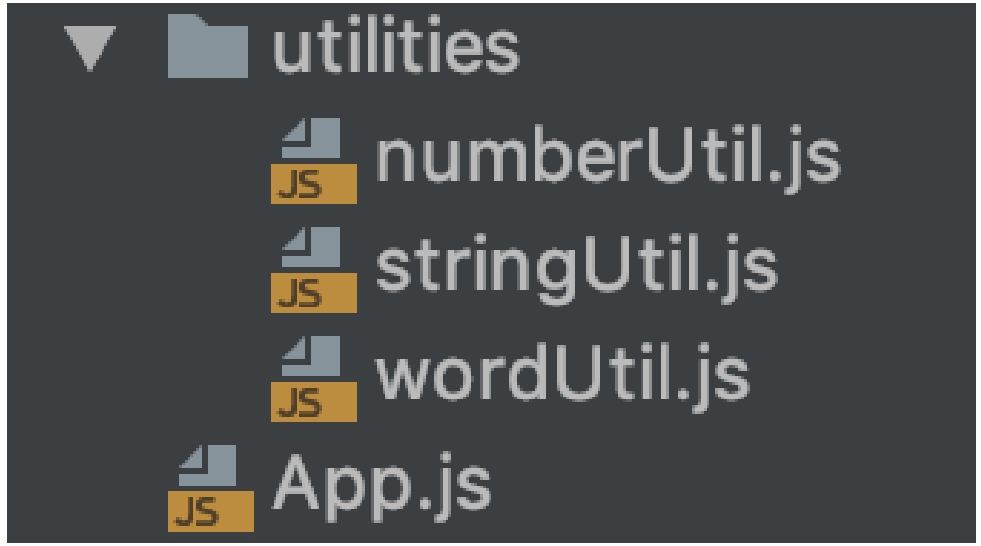
Vježba 5.3: *Import Export*

index.js

- često unutar jednog direktorija imamo puno JavaScript modula koje želimo koristiti u nekom drugom direktoriju ili datoteci (npr. unutar *App.js*)
- umjesto da *importamo* svaki potrebni modul iz zasebne datoteke, možemo unutar direktorija kreirati datoteku *index.js* u kojoj definiramo sve JavaScript module koje želimo *exportati* iz tog direktorija
- tada unutar *App.js* datoteke možemo *importati* sve module jednom naredbom, koristeći samo put do željenog direktorija (umjesto da za svaki modul moramo koristiti zasebnu naredbu *import*)

index.js - prije:

Struktura direktorija:



App.js

```
import NumberUtil from "./utilities/numberUtil";
import StringUtil from "./utilities/stringUtil";
import WordUtil from "./utilities/wordUtil";
```

index.js - poslije:

Struktura direktorija:

index.js

```
export { default as NumberUtil } from './numberUtil';
export { default as StringUtil } from './stringUtil';
export { default as WordUtil } from './wordUtil';
```

App.js

```
import { NumberUtil, StringUtil, WordUtil } from "./utilities";
```

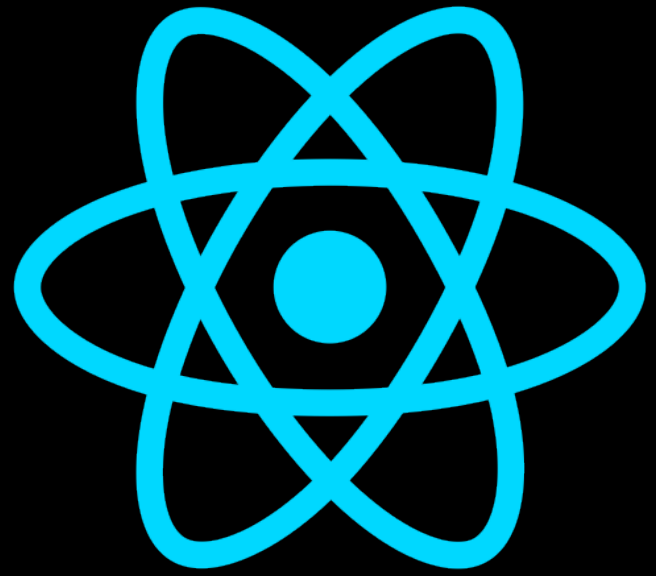
Vježba 5.4: Index.js

Literatura

- https://www.freecodecamp.org/news/javascript-modules-a-beginner-s-guide-783f7d7a5fcc/
- https://www.geeksforgeeks.org/javascript-importing-and-exporting-modules/
- https://alligator.io/react/index-js-public-interfaces/

Uvod u reaktivnu arhitekturu

Routing



Single-page apps

- web-aplikacije koje se sastoje od samo jedne stranice
- stranica u interakciji s korisnikom vrši dinamičko prepisivanje trenutnog sadržaja umjesto učitavanja nove stranice s poslužitelja
- ovakav pristup izbjegava prekid korisničkog iskustva učitavanjem uzastopnih stranica, čineći ponašanje aplikacije nalik onome *desktop* aplikacija

Single-page apps

- svi potrebni resursi (HTML, CSS, JavaScript, zvuk, video, slikovne datoteke) preuzimaju se učitavanjem jedne stranice ili se dinamički učitavaju po potrebi, kao odgovor na radnje korisnika
- stranica se ni u jednom trenutku ponovno ne učitava
- kontrola nad izvođenjem se ni u jednom trenutku ne prenosi na neku drugu stranicu

React Router

- React aplikacije su *single page* aplikacije - postoji samo jedna web-adresa
- ne postoji mogućnost navigacije preko web-adrese (URL), kao što je običaj kod „običnih” HTML stranica
- rješenje: React Router

React Router

- biblioteka koju moramo dodatno instalirati u našu React aplikaciju
- omogućava nam da pozivom na neku web-adresu postavimo React aplikaciju na unaprijed definirano stanje
- iako je aplikacija *single page*, dobivamo mogućnost navigacije putem web-adrese

React Router - instalacija

- za instalaciju React Router biblioteke u postojeću React aplikaciju, potrebno je u *command line interfaceu* (CLI) izvršiti sljedeću naredbu:

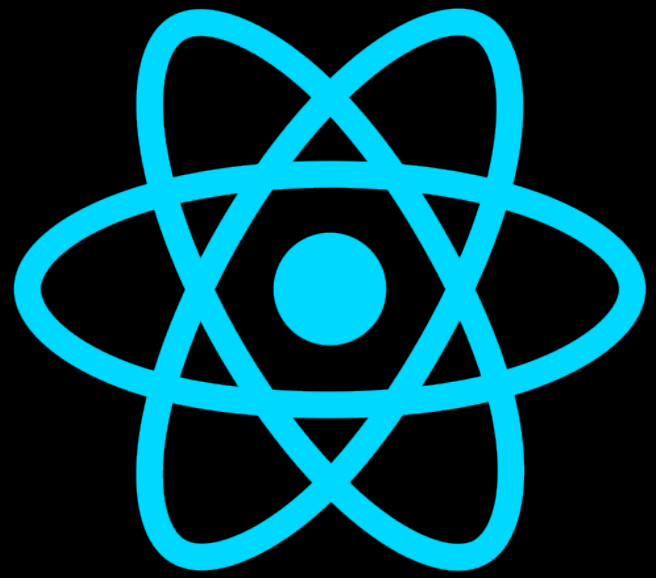
  ```
  npm install --save react-router-dom
  ```

- korištenje React Router biblioteke bit će objašnjeno u *Naprednim konceptima ReactJS-a*

Literatura

- https://reacttraining.com/react-router/web

Osnovni koncepti ReactJS-a



React komponente i JSX

JSX

- koristi se za opisivanje izgleda korisničkog sučelja u React aplikaciji
- ima sintaksu vrlo sličnu HTML-u te ga, uz znanje HTML-a vrlo lako rabe i početnici

```html
<div class="awesome" style="border: 1px solid red">
    <label for="name">Enter your name: </label>
    <input type="text" id="name" />
</div>
<p>Enter your HTML here</p>
```

HTML

```jsx
function ReactComponent() {
  return (
    <div>
      <div className="awesome" style={{ border: "1px solid red" }}>
        <label htmlFor="name">Enter your name: </label>
        <input type="text" id="name" />
      </div>
      <p>Enter your HTML here</p>
    </div>
  );
}
```

JSX

ALGEBRA

JSX

- svaka React komponenta mora vraćati neki JSX element:

```
const element = <h1>Hello world!</h1>;
```

- ako komponenta vraća više JSX elemenata, svaki mora biti ugniježđen u jednom vršnom elementu, najčešće elementu *** ***

JSX

- glavna prednost JSX-a u odnosu na običan HTML je što se uz opisivanja korisničkog sučelja može pisati i običan JavaScript (*JavaScript expression*)
- unutar JSX-a JavaScript kod je potrebno staviti unutar vitičastih zagrada {JavaScript code}

JavaScript unutar JSX-a:

```
function ReactComponent() {
    const animals = ['slon', 'zebra', 'pas'];
    const x = 5;
    const y = 3;

    return (
        <div>
            Ovo su moje zivotinje: {animals.join(', ')}
            <br />
            X + Y = {x + y}
        </div>
    )
}
```

ALGEBRA

Unutar vitičastih zagrada možemo pozivati i funckije:

```
function ReactComponent() {
    const user = {
            name: "Marko",
            surname: "Horvat"
        };

        const formatUser = user => {
            return `${user.name} ${user.surname}`;
        };

        return <p>Hello {formatUser(user)}</p>;
}
```

Vježba 5.5: JSX

Conditional rendering (uvjetovano iscrtavanje)

- u JSX-u nije moguće pisati *if* blokove
- umjesto toga, element koji želimo nacrtati možemo dodijeliti varijabli ili vratiti kao rezultat logičkog operatora *&&*

Conditional rendering

pomoću varijable:

```
function ReactComponent() {
    const user = "Ivan Horvat";

    const greetings =
user.startsWith("Ivan")
    ? <p>Hello Friend!</p>
    : <p>Hello Stranger!</p>;

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

Conditional rendering

pomoću logičkog operatora &&

```
function ReactComponent() {
        const user = "Ivan Horvat";
        const isFriend =
user.startsWith("Ivan");

        return (
            <div>
            {isFriend && <p>Hello
Friend!</p>}
            {!isFriend && <p>Hello Stranger!</p>}
        </div>
        );
}
```

Vjezba 5.6: *Conditional rendering*

ReactJS komponente

- razlikujemo:
  - komponente definirane funkcijom i komponente definirane klasom
  - složene (pametne, *statefull*) i jednostavne (prezentacijske, *stateless*) komponente

ReactJS komponente

- osnovni koncept Reacta - React je u svojoj jezgri biblioteka za stvaranje komponenata
- komponente su neovisni, ponovno iskoristivi dijelovi kôda
- tipična React web-aplikacija može biti prikazana kao stablo komponenti i to tako da postoji jedna korijenska (*root*) komponenta (*“App”*), i potencijalno beskonačna količina ugniježdenih komponenti

ReactJS komponente

- mogu biti funkcije ili klase
- primaju proizvoljne argumente (*properties* - skraćeno *props*), a vraćaju proizvoljni HTML
- konvencija nalaže da imena React komponenti počinju velikim početnim slovom

Funckijska komponenta

Komponenta klasa

ReactJS komponente

- svaka React komponenta može sadržavati nove React komponente
- u tom slučaju upotrebljavamo terminologiju *parent*/*child*
- *parent* komponenta u sebi sadrži *child* komponente
- React aplikacija je zapravo stablo ugniježđenih React komponenti

Parent/child komponente

```
function ChildComponent() {
    return <h1>Ja sam ChildComponent</h1>
}

function ParentComponent() {
    return (
        <div>
            <h1>Ja sam ParentComponent</h1>
            <ChildComponent />
        </div>
    )
}
```

ReactJS komponente

- komponente se najčešće rade u odvojenim datotekama
- konvencija nalaže da ime komponente počinje velikim početnim slovom, a pripadajuća datoteka (datoteka u kojoj se komponenta nalazi) ima isto ime kao i komponenta
- ekstenzija datoteke u kojoj se nalazi komponenta je *.js* ili *.jsx*

```jsx
import React from "react";

export default class WelcomeKlasa extends React.Component {
 render() {
   return (
     <h2>Welcome! Komponenta definirana klasom.</h2>;
   );
 }
}
```

WelcomeKlasa.js

```jsx
import React from "react";

export default function WelcomeFunkcija() {
 return <h2>Welcome! Komponenta definirana funckijom.</h2>;
}
```

WelcomeFunkcija.js

```jsx
import React from "react";
import WelcomeFunkcija from "./WelcomeFunkcija";
import WelcomeKlasa from "./WelcomeKlasa";

export default function App() {
 return (
   <div>
     <WelcomeFunkcija />
     <WelcomeKlasa />
   </div>
 );
}
```

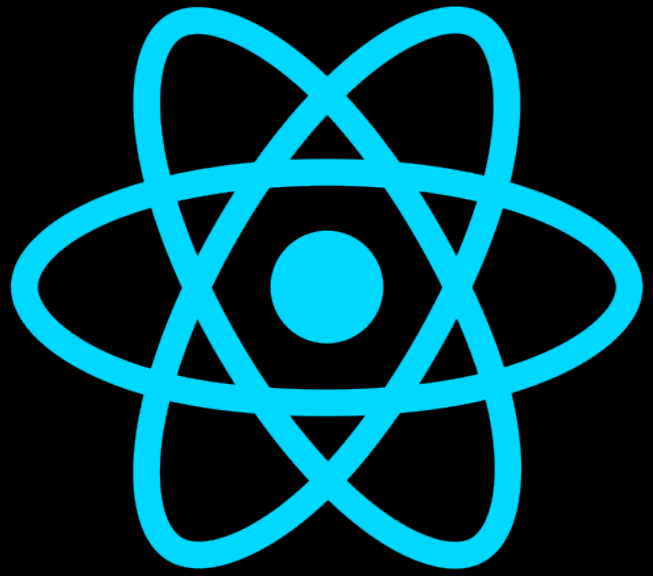
App.js

Vježba 5.7: Komponente

Literatura

- https://reactjs.org/docs/react-component.html
- https://reactjs.org/docs/components-and-props.htm
- https://reactjs.org/docs/introducing-jsx.html
- https://reactjs.org/docs/react-without-jsx.html

Osnovni koncepti ReactJS-a

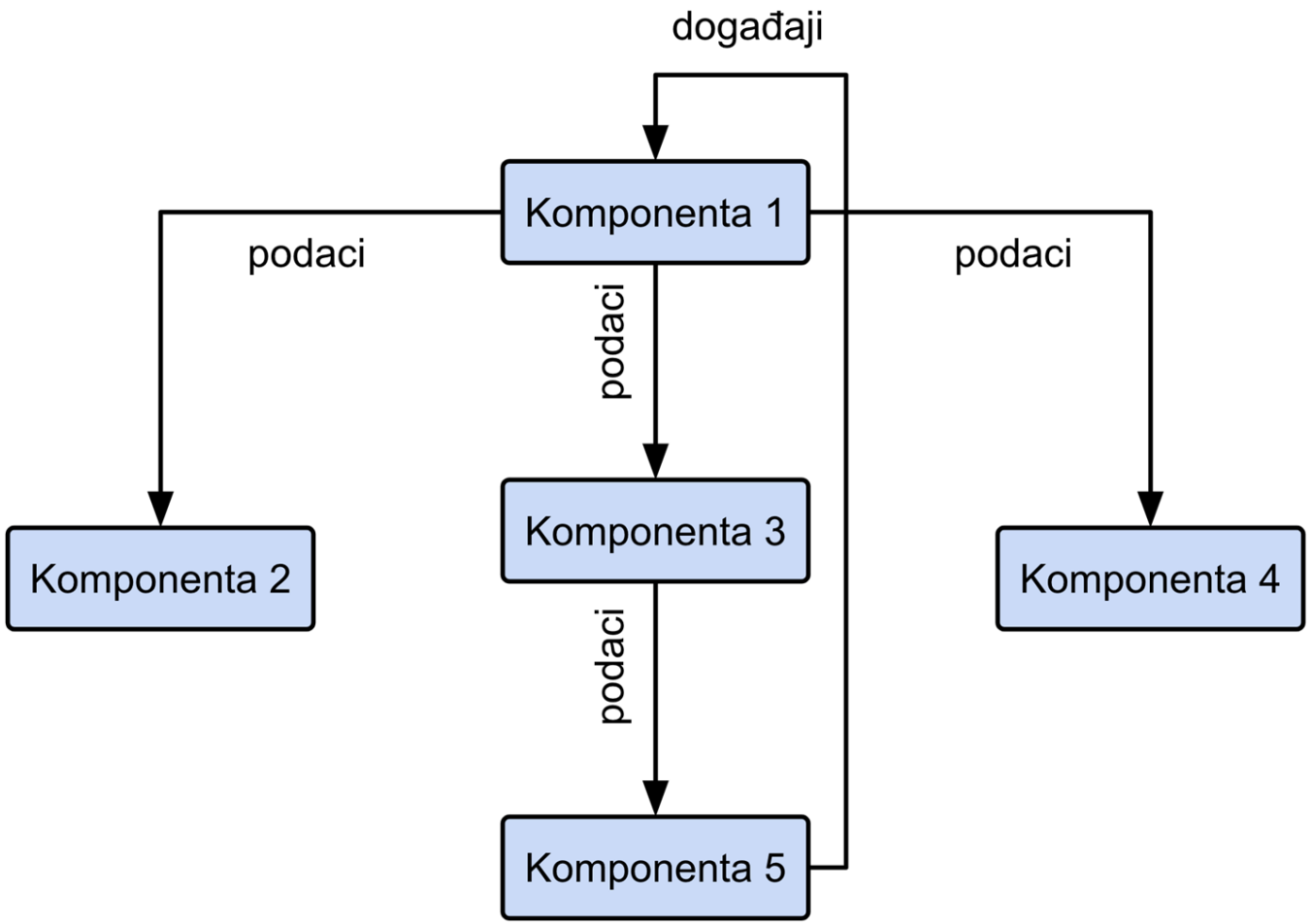


Svojstva (*props*)

Props

- *props* („propertyji”) su način na koji komponente međusobno komuniciraju
- *propsi* se koriste kako bi se informacije prenijele iz *parent* komponente u *child* komponentu
- protok podataka kroz *propse* je uvijek jednosmjeran - iz *parent* komponente u *child* komponentu
- *child* komponenta ne može i ne smije mijenjati *props* objekt koji je primila od *parenta*

Jednosmjeran protok podataka u React komponentama

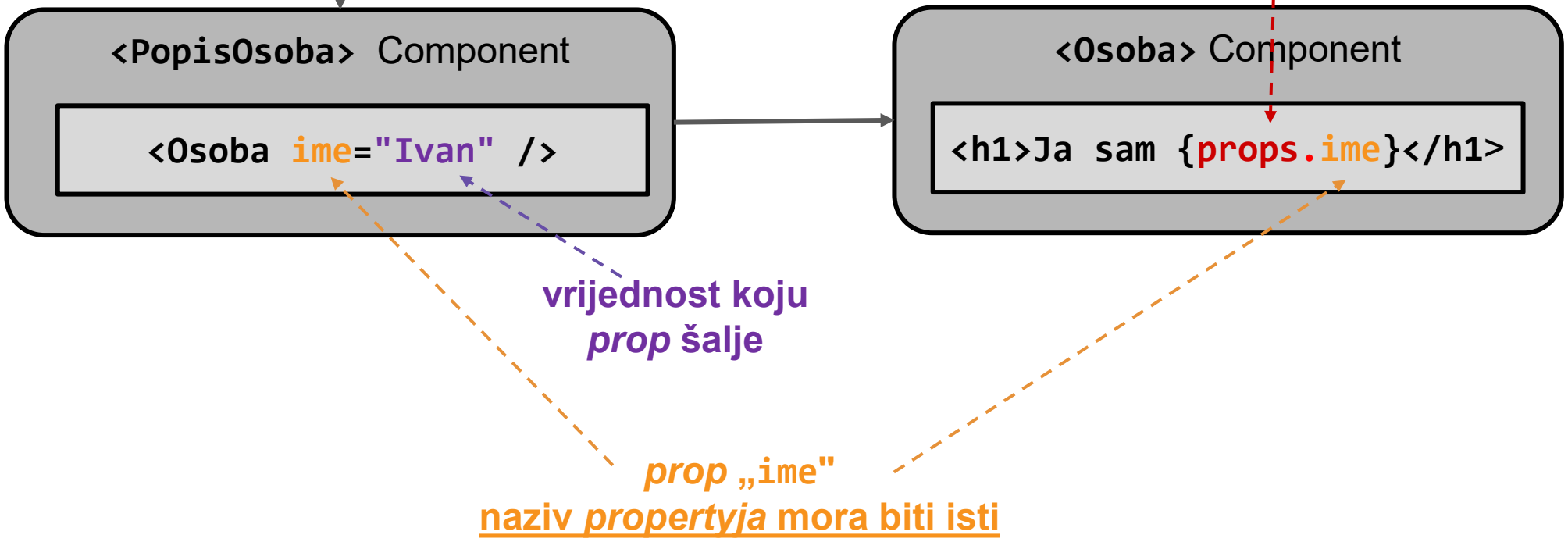


Props

promjene koje se događaju izvan komponente (podaci se šalju u komponentu)

argument poslan komponenti definiranoj funkcijom

u komponenti definiranoj klasom koristi se *this.props*



Props

```
function ChildComponent({ name }) {
    return <h1>Ja sam {name}</h1>
}

function ParentComponent() {
    return (
        <div>
            <h1>Ja sam ParentComponent</h1>
            <ChildComponent name="Child"/>
        </div>
    )
}
```

Props

- *propertyji* mogu biti bilo koji JavaScript objekt - primitivan ili složen objekt
- kako bi bilo lakše znati koje *propertyje* neka komponenta može primiti, koristimo object ***propTypes***
- kako bismo definirali *defaultne* vrijednosti nekih *propertyja*, koristimo objekt ***defaultProps***

Props - playground

- https://codepen.io/pen?&editable=true&editors=0010
- https://codepen.io/pen?&editable=true&editors=0010

PropTypes

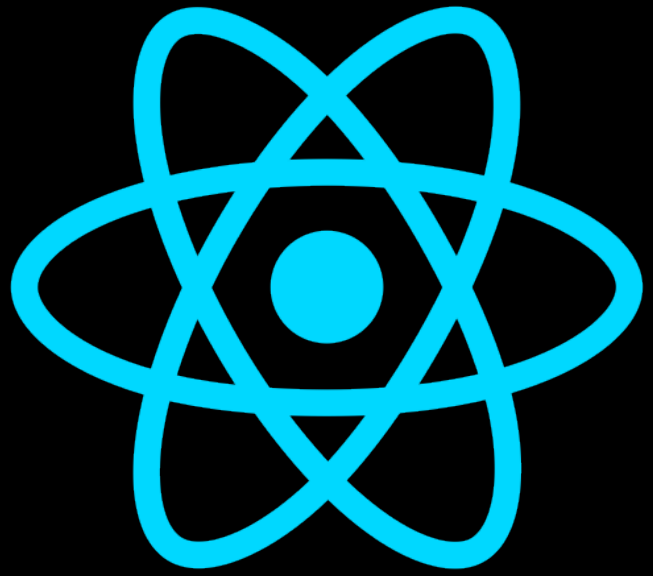
```jsx
import React from "react";
import PropTypes from "prop-types";

function MyComponent({ name, age, gender, interests, address }) {
  return (
    <div>
      <div>Ime: {name}</div>
      <div>Godine: {age}</div>
      <div>Spol: {gender}</div>
      <div>Interesi: {interests.join(", ")}</div>
      <div>Ulica: {address.street}</div>
      <div>Grad: {address.city}</div>
    </div>
  );
}

MyComponent.propTypes = {
  name: PropTypes.string,
  age: PropTypes.number,
  gender: PropTypes.oneOf(["M", "F"]),
  interests: PropTypes.arrayOf(PropTypes.number),
  address: PropTypes.shape({
    street: PropTypes.string,
    city: PropTypes.string
  })
};

MyComponent.defaultProps = {
  gender: "F"
};
```

Vježba 5.8: *Props*

Literatura

- https://reactjs.org/docs/components-and-props.htm
- https://reactjs.org/docs/typechecking-with-proptypes.html
- https://hackernoon.com/understanding-state-and-props-in-react-94bc09232b9c

Osnovni koncepti ReactJS-a



Stanja komponente (*state*)

State

- Postoje dvije vrste podataka koje definiraju komponentu:
  - **props** - komponenta ih dobiva izvana (od *parent* komponente) i ne može ih promijeniti
  - **state** - lokalno stanje komponente - komponenta ima kontrolu nad svojim stanjem i može ga mijenjati pomoću metode ***setState***

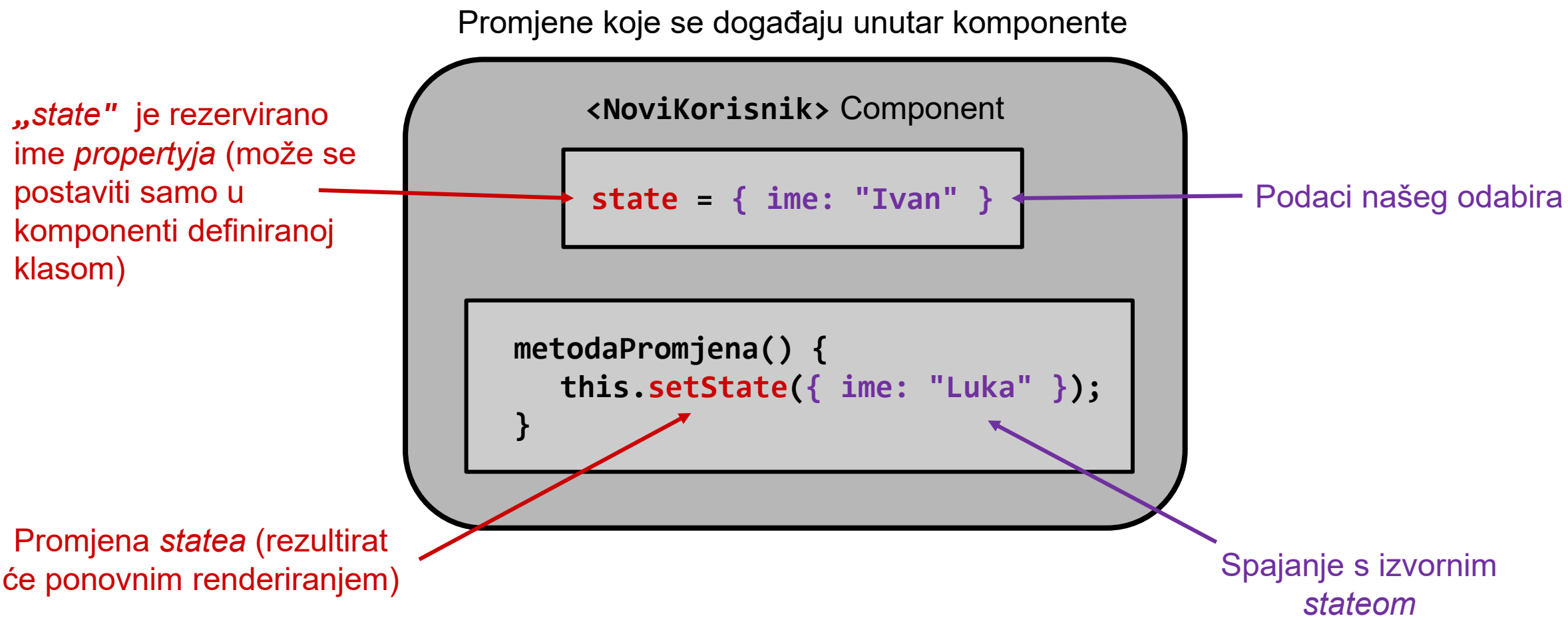
State komponente

```
class MyComponent extends React.Component {
  state = {
    city: "Zagreb",
    state: "Hrvatska"
  };

  render() {
    return (
      <div>
        <h1>Grad: {this.state.city}</h1>
        <h1>Država: {this.state.state}</h1>
      </div>
    );
  }
}
```

State

- state se smije mijenjati samo pomoću metode *setState*
- kada se izvrši metoda *setState*, komponenta se ponovno iscrtava (renderira) kako bi prikazala promjene u stanju
- ponovnim renderiranjem komponente, renderiraju se i sve njezine *child* komponente

State


Promjene koje se događaju unutar komponente
„state" je rezervirano ime propertyja (može se postaviti samo u komponenti definiranoj klasom)
<NoviKorisnik> Component
state = { ime: "Ivan" }
Podaci našeg odabira
metodaPromjena() {
this.setState({ ime: "Luka" });
}
Promjena statea (rezultirat će ponovnim renderiranjem)
Spajanje s izvornim stateom

```
class MyComponent extends React.Component {
  state = {
   city: "Zagreb",
   state: "Hrvatska"
  };

  changeState = () => {
    const newCity = this.state.city === "Zagreb" ? "Osijek" : "Zagreb";
    this.setState({ city: newCity });
  };


  render() {
   return (
     <div>
       <h1>Grad: {this.state.city}</h1>
       <h1>Država: {this.state.state}</h1>
       <button onClick={this.changeState}>Drugi grad</button>
     </div>
   );
  }
}
```

```
class MyComponent extends React.Component {
  state = {
    city: "Zagreb",
    state: "Hrvatska"
  };

  changeState = () => {
     const newCity = this.state.city === "Zagreb" ? "Osijek" : "Zagreb";
     this.setState({ city: newCity });
   };

   dontChangeLikeThis = () => {
     const newCity = this.state.city === "Zagreb" ? "Osijek" : "Zagreb";
     this.state.city = newCity;
   };

  render() {
    return (
      <div>
        <h1>Grad: {this.state.city}</h1>
        <h1>Država: {this.state.state}</h1>
        <button onClick={this.changeState}>Drugi grad</button>
         <button onClick={this.dontChangeLikeThis}>Drugi grad</button>
      </div>
    );
  }
}
```

State

- unutar *statea* mogu stajati bilo kakvi JavaScript objekti, primitivni ili složeni objekti
- kada uređujemo složeni objekt (polje ili objekt), nije moguće urediti samo vrijednost nekog elementa unutar tog objekta, već moramo trenutni objekt zamijeniti novim objektom koji sadrži promijenjene vrijednosti

State - playground

- https://codepen.io/gaearon/pen/zKRqNB?editors=0010

Vježba 5.9: *State*

State

- za kontroliranje protoka podataka imamo i vanjske *state containere* Redux ili MobX za promjenu *statea* umjesto pozivanja funkcije *setState*
- više o Redux i MobX bibliotekama bit će rečeno u *Naprednim konceptima ReactJS-a*

Statefull/Stateless komponente

- ***Statefull* komponente**
  - “pametne komponente”
  - imaju vlastito stanje (*state*)
  - prosljeđuju stanje *child* komponentama preko *propertyja*
- ***Stateless* komponente**
  - prezentacijske komponente
  - nemaju vlastito stanje (*state*)
  - primaju stanje aplikacije preko *propertyja* (*propsa*)
  - ako se promijeni neki *property*, ponovno se iscrtavaju

Statefull/Stateless komponente

- *Stateless* komponente preko svojih *propsa* mogu primiti novi *state statefull* komponente:
  - u ovom slučaju *statefull* komponenta je roditelj (*parent*), a *stateless* komponenta je njezino dijete (*child*)
  - ako se *state parenta* promijenio, promijenio se i *props* objekt koji je primila *child* komponenta
  - *child* komponenta će se ponovno renderirati radi promjene *props* objekta te će tako reflektirati promjenu stanja u *parentu*

```
function StatelessComponent({ color, count, onButtonClick }) {
 const style = { backgroundColor: color, padding: 20 };

 return (
   <div>
     <p>Broj klikova: {count}</p>
     <button style={style} onClick={onButtonClick}>
       Klikni me!
     </button>
   </div>
 );
}
```

```jsx
export default class StatefullComponent extends React.Component {
 state = { color: "red", count: 0 };

 handleButtonClick = () => {
   this.setState({ count: this.state.count + 1 });
 };

 render() {
   return (
     <div>
       <StatelessComponent
         count={this.state.count}
         color={this.state.color}
         onButtonClick={this.handleButtonClick}
       />
       <p>Mogu koristiti state i u parent komponenti: {this.state.count}</p>
     </div>
   );
 }
}
```

useState hook

- *hook* je nova funkcija koja se pojavila s posljednjom verzijom React biblioteke
- *useState hook* omogućuje korištenje stanja u komponentama definiranima funkcijom

useState hook

App.js prije

```
export default class App extends React.Component {
        state = { number: 0 };

    const handleChange = event => {
                this.setState({ number: event.target.value});
        };

        render() {
                <div>{this.state.number}</div>
    }
}
```

useState hook

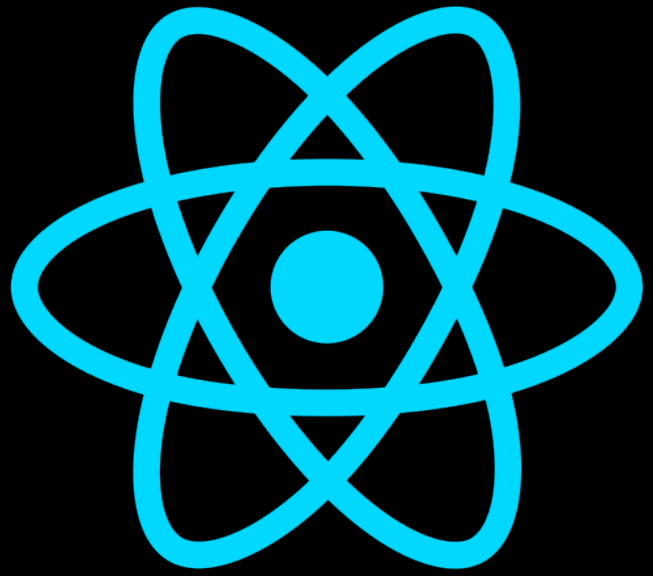
App.js poslije

```
export default class App extends React.Component {
        state = { number: 0 };

    const handleChange = event => {
                this.setState({ number: event.target.value});
        };

        render() {
                return <div>{this.state.number}</div>;
    }
}
```

Literatura

- https://reactjs.org/docs/state-and-lifecycle.html
- https://hackernoon.com/understanding-state-and-props-in-react-94bc09232b9c

Osnovni koncepti ReactJS-a



Događaji (*events*)

Događaji (*events*)

- događaji - interakcija između korisnika i aplikacije gdje svaka korisnička akcija ima unaprijed definiranu reakciju (set zadataka koje aplikacija mora izvršiti kao odgovor)
- kao i u HTML-u, postoji definirana lista događaja na koje React komponente mogu reagirati, npr: **`onClick`**, **`onChange`**, …
- aktiviranjem događaja izvršava se funkcija dodijeljena tom događaju - vrlo često ta funkcija mijenja stanje neke komponente ili cijele aplikacije

Najčešći *eventi*

HTML	React
`onclick`	`onClick`
`onchange`	`onChange`
`onmouseover`	`onMouseOver`
`onmouseout`	`onMouseOut`
`onkeydown`	`onKeyDown`
`onload`	`onLoad`

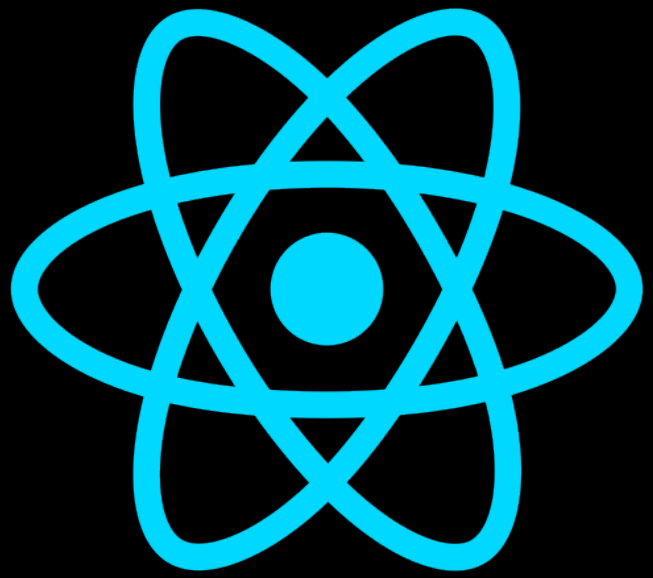
Primjer korištenja *eventa*:

```
function ReactComponent() {
   const handleButtonClick = (event) => {
     console.log("Button click!");
   };

   return (
      <button onClick={handleButtonClick}>
         Klikni me!
      </button>
   );
}
```

Literatura

- https://openclassrooms.com/en/courses/4286486-build-web-apps-with-reactjs/4286716-handle-events
- https://maksimivanov.com/posts/reactjs-handling-events/

Osnovni koncepti ReactJS-a



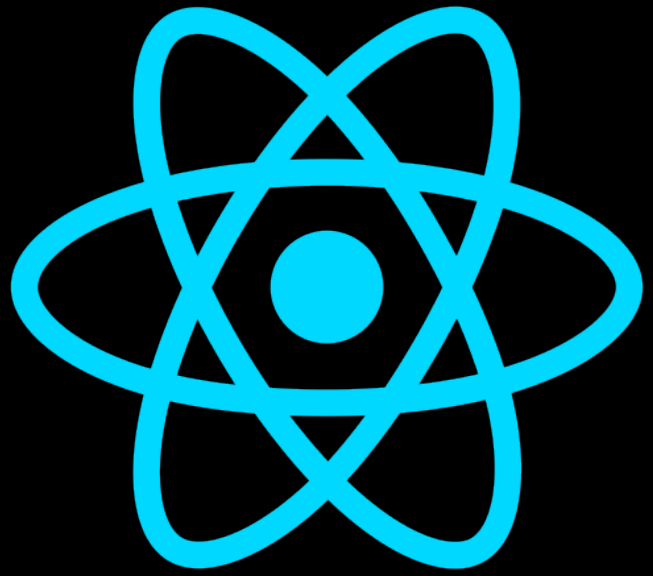
Vježbe - *state, props, events*

Vježba 5.10: *Set state*

Vježba 5.11: *useState hook*

Vježba 5.12: *Events*

Osnovni koncepti ReactJS-a

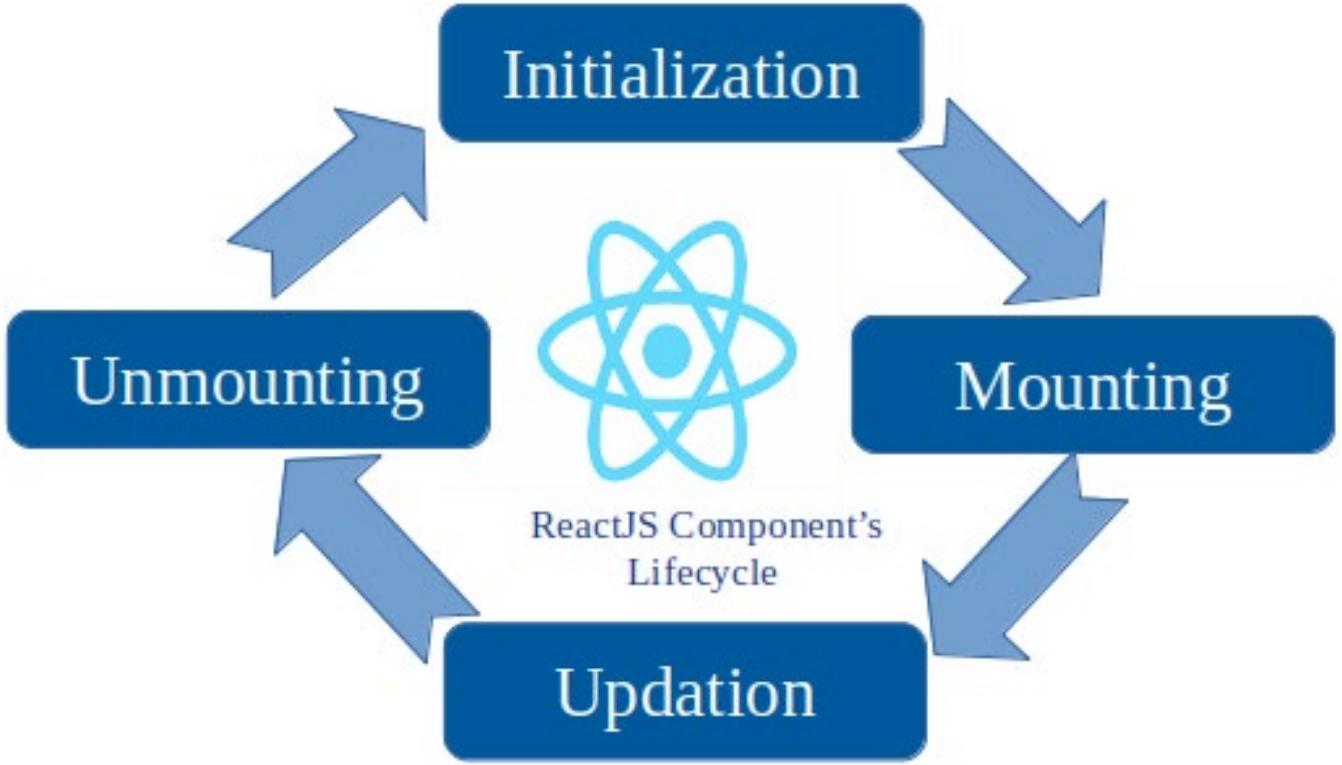


Životni ciklus komponente

Životni ciklus komponente (engl. *Lifecycle methods*)

- životni ciklus komponente može se definirati kao niz metoda koje se pozivaju u različitim fazama postojanja komponente
- te metode nazivamo *lifecycle methods*, a pozivaju se u sljedećim fazama:
  - inicijalizacija komponente (*initialization*)
  - postavljanje na DOM (*mounting*)
  - ažuriranje (*updating*)
  - brisanje iz DOM-a (*unmounting*)

Životni ciklus komponente (*Lifecycle methods*)



Inicijalizacija (*initialization*)

U ovoj fazi definira se početno stanje (*state*) komponente

```
class Initialize extends React.Component {
  constructor(props) {
    // poziv konstruktora klase React.Component
    super(props);
       // postavljanje početnog stanja komponente
    this.state = {
      date: new Date(),
      status: false
    };
  }
}
```

Postavljanje na DOM (*mounting*)

Mount je faza životnog ciklusa komponente koji se događa nakon što je incijalizacija komponente dovršena. U tom trenutku React komponenta se postavlja u DOM i po prvi put prikazuje (*renderira)* na web-stranici.

Dvije *lifecycle* metode se vezane za ovu fazu:

- **`componentWillMount`**
- **`componentDidMount`**

Postavljanje na DOM (*mounting*)

- **`componentWillMount`:**
  - ova metoda se poziva jednom neposredno prije nego što se komponenta ugradi na DOM (prije nego se pozove **`render`** metoda)
- **`componentDidMount`:**
  - ova metoda se poziva nakon što se komponenta ugradi na DOM; kao i metoda `componentWillMount`, izvršava se jednom u životnom ciklusu komponente; poziva se nakon metode **`render`**

Postavljanje na DOM (*mounting*)

```
class LifeCycle extends React.Component {
  componentWillMount() {
    console.log("Component will mount!");
  }

  componentDidMount() {
    console.log("Component did mount!");
    this.getList();
  }

  getList = () => {
    // method to make api call
  };

  render() {
    return (
      <div><h3>Hello mounting methods!</h3></div>
    );
  }
}
```

Ažuriranje (*updating*)

- treća faza kroz koju prolazi komponenta
- faza dolazi na red nakon faze postavljanja na DOM
- u ovoj fazi korisničkom interakcijom mijenjaju se svojstva (*propsi*) i stanja (*state*) komponente i stoga dolazi do ponovnog iscrtavanja (*renderiranja*) komponente

Ažuriranje (*updating*)

Metode na raspolaganju u ovoj fazi su:

- **`shouldComponentUpdate:`**
  - ova metoda određuje treba li komponentu ažurirati ili ne
  - standardna postavka metode je da uvijek vraća *true*
  - ako ne želimo da se komponenta ponovno iscrtava, potrebno je podesiti metodu tako da vraća *false*
  - ova metoda prima argumente **`nextProps`** i **`nextState;`** ovisno o tim argumentima, moguće je izračunati je li potrebno ponovno iscrtavanje komponente

Ažuriranje (*updating*)

- **`componentWillUpdate:`**
  - ova metoda se poziva jednom prije ponovnog iscrtavanja komponente
  - koristi se kada je potrebno izvršiti neki izračun prije ponovnog prikazivanja komponente, a nakon ažuriranja *propsa* i *statea*
  - kao i metoda **`shouldComponentUpdate`** prima argumente **`nextProps`** i **`nextState`**.

Ažuriranje (*updating*)

- **`componentDidUpdate:`**
  - ova metoda poziva se neposredno nakon što se komponenta ažurirala u DOM-u
  - prima argumente **`prevProps`** i **`prevState`**

Ažuriranje (*updating*)

- primjer: https://codepen.io/toroz/pen/mNXLNR?editors=0111

Brisanje iz DOM-a (*unmounting*)

Ovo je posljednja faza u životnom ciklusu komponente. U ovoj fazi komponenta se miče iz DOM-a. U ovoj fazi dostupna je samo jedna metoda:

- **`componentWillUnmount`**
  - poziva se prije nego što se komponenta uklanja iz DOMa
  - označava kraj životnog ciklusa komponente

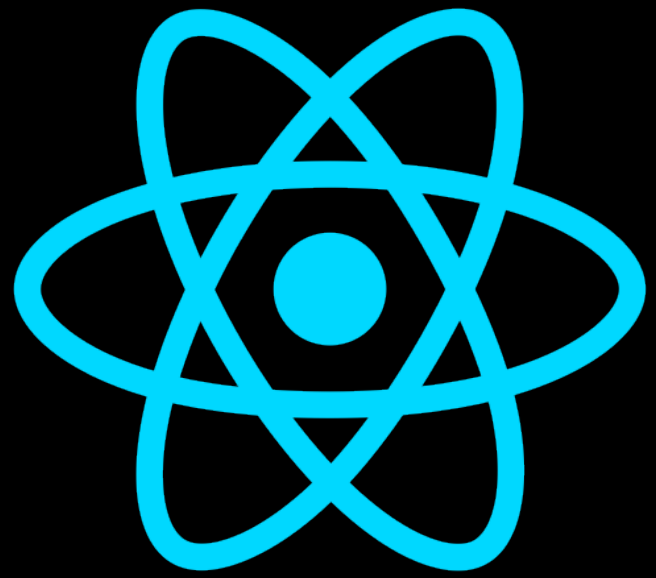
useEffect hook

- *useEffect hook* koristimo kada želimo uzrokovati neke nuspojave (engl. *side effects*) prije ili poslije iscrtavanja (renderiranja) komponente
- *useEffect hook* može zamijeniti sve *lifecycle* metode, odnosno, različitom implementacijom *useEffect hooka*, možemo ga pokrenuti u istim životnim fazama (funkcijske) komponente u kojima su se pokretale *lifecycle* metode u komponentama definiranima klasom
- primjer: https://codepen.io/psynewave/pen/NmzdRY?editors=0011

Literatura

- https://www.freecodecamp.org/news/how-to-understand-a-components-lifecycle-methods-in-reactjs-e1a609840630/
- https://www.geeksforgeeks.org/reactjs-lifecycle-components/
- https://www.freecodecamp.org/news/these-are-the-concepts-you-should-know-in-react-js-after-you-learn-the-basics-ee1d2f4b8030/
- https://reactjs.org/docs/state-and-lifecycle.html
- https://programmingwithmosh.com/javascript/react-lifecycle-methods/
- https://reactjs.org/docs/hooks-effect.html

Osnovni koncepti ReactJS-a



Liste

Liste

- u web-aplikacijama se vrlo često pojavljuju liste jednakih elemenata: kartice, tablice, padajući izbornici, …
- kako bismo napravili listu elemenata u Reactu, koristimo JavaScript polja i funkcije

Transformacija polja u JavaScriptu

```
const numbers = [1, 2, 3, 4, 5];
const updatedNumbers = numbers.map(number => {
 return number + 2;
});
console.log(updatedNumbers);


>> [3, 4, 5, 6, 7]
```

Liste (polja) u Reactu

U Reactu, transformacija polja u elemente je gotovo identična, a vrlo lako možemo napraviti kolekciju elemenata i uključiti ih u JSX koristeći vitičaste zagrade {}:

```
function NumberList() {
 const numbers = [1, 2, 3, 4, 5];
 return (
   <ul>
     {numbers.map(number => <li>{number}</li>)}
   </ul>
 );
}
```

Liste (polja) u Reactu

- kôd prikazan na prethodnom slajdu radi, međutim, ako otvorimo konzolu, vidjet ćemo sljedeće upozorenje:

```
"Warning: Each child in a list should have a unique
'key' prop. See https://fb.me/react-warning-keys for
more information."

Check the render method of `NumberList`." "" "
    in li (created by NumberList)
     in NumberList"
```

Liste (polja) u Reactu

- kako bismo riješili taj problem, uvodimo pojam ključeva (engl. *keys*)
- *property key* pomaže Reactu da prepozna elemente liste koji su se promijenili, dodali ili uklonili
- na taj način React može brzo i učinkovito odraditi DOM manipulacije - na promjenu jednog elementa u listi, promijenit će se samo taj DOM element jer React prema *propertyju key* zna prepoznati koji je to element

Liste (polja) u Reactu

- svaki *key* unutar liste mora biti unikatan, a najčešće je to ID podataka koje prikazujemo
- ako nemamo pristup unikatnom podatku za *key*, možemo koristiti *indeks* podataka unutar matrice, ali to se nikako ne preporučuje:
  - ako imamo podatke unutar matrice kojima se položaj (*indeks*) unutar matrice mijenja, to će negativno utjecati na performanse React aplikacije

Liste (polja) u Reactu - *playground*

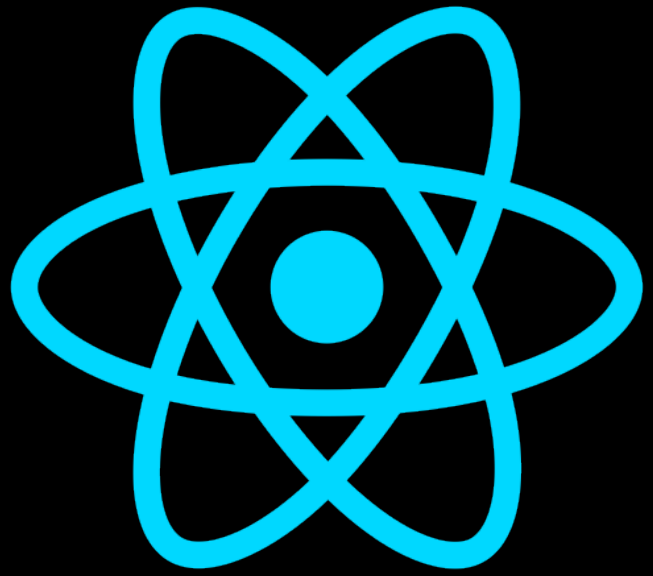
- https://codepen.io/gaearon/pen/jrXYRR?editors=0011

Vježba 5.13: Liste

Literatura

- https://reactjs.org/docs/lists-and-keys.html
- https://medium.com/@robinpokorny/index-as-a-key-is-an-anti-pattern-e0349aece318
- https://www.geeksforgeeks.org/reactjs-lists/

Osnovni koncepti ReactJS-a



Obrasci (*forms*)

Obrasci

- obrasci su među rijetkim HTML elementima koji su predviđeni za interakciju korisnika s nekom stranicom (aplikacijom)
- uobičajena uporaba:
  - prijava i registracija
  - pretraživanje
  - obrasci za kontakt
  - plaćanje sadržaja košarice

Obrasci

- elementi HTML obrazaca rade malo drugačije od ostalih DOM elemenata u Reactu jer je HTML element *form* već predviđen za primanje stanja
- na primjer, ovo je obrazac u običnom HTML-u koji prihvaća jedno ime:

```
<form>
        <label>
                Ime:
                <input type="text" name="name"
/>
        </label>
        <input type="submit" value="Potvrdi" />
</form>
```

Obrasci

- obrazac na prethodnoj stranici prima ime, a na potvrdu (engl. *form submit*) se učitava nova stranica
- jednako se ponašaju i obrasci u Reactu
- međutim, u većini slučajeva, prikladno je imati JavaScript funkciju koja ima pristup korisničkim podacima i upravlja potvrdom obrasca
- to znači da obrascima upravljamo na isti način kao što smo upravljali i događajima - događaju u obrascu dodjeljujemo funkciju, a okidanjem tog događaja automatski se poziva dodijeljenja funkcija

Obrasci

Primjer: React obrasca koji s funkcijama koje reagiraju na događaje promjene imena (onChange) i predavanja forme (onSubmit):

```
<form onSubmit={this.handleSubmit}>
        <label>
                Ime:
                <input name="name" onChange={this.handleChange}
value={this.state.name}/>
        </label>
        <input type="submit" value="Potvrdi" />
</form>
```

Obrasci - *playground*

- https://codepen.io/gaearon/pen/wgedvV?editors=0010
- https://codepen.io/gaearon/pen/VmmPgp?editors=0010

Vježba 5.14: Obrasci 1

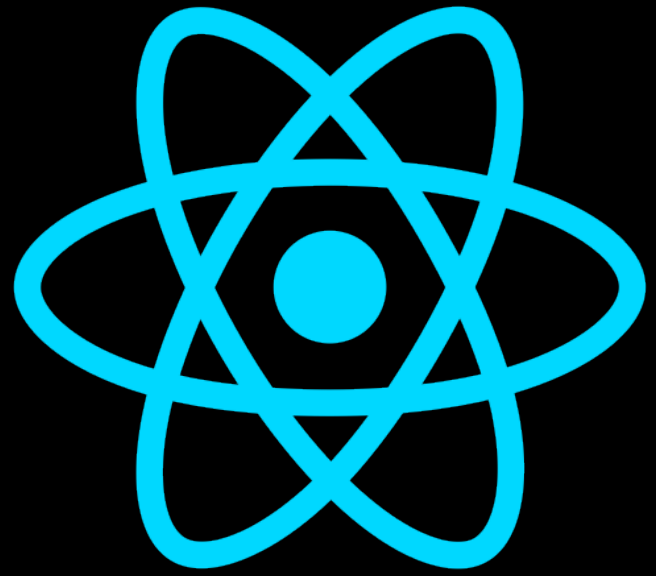
Vježba 5.15: Obrasci 2

Literatura

- https://reactjs.org/docs/forms.html
- https://flaviocopes.com/react-forms/
- https://itnext.io/keep-calm-and-handle-forms-in-react-js-52c67eea340e
- https://www.w3schools.com/react/react_forms.asp

Osnovni koncepti ReactJS-a

React Context



React Context

- pretpostavimo sljedeću konstrukciju React komponenata:

```
Komponenta 1
    |-- Komponenta 2
        |-- Komponenta 3
            |-- Komponenta 4
```

- također pretpostavimo da želimo *state* Komponente 1 proslijediti do Komponente 4; Pomoću *propertyja* (*propsa*), to moramo prosljeđivati redom: `Komponenta 1 -> Komponenta 2`; `Komponenta 2 -> Komponenta 3`; `Komponenta 3 -> Komponenta 4`

React Context

React Context nam omogućava da unutar React aplikacije imamo „globalne” varijable koje možemo prosljeđivati komponentama, bez prethodnog prosljeđivanja istog *parent* komponenti.

React Context

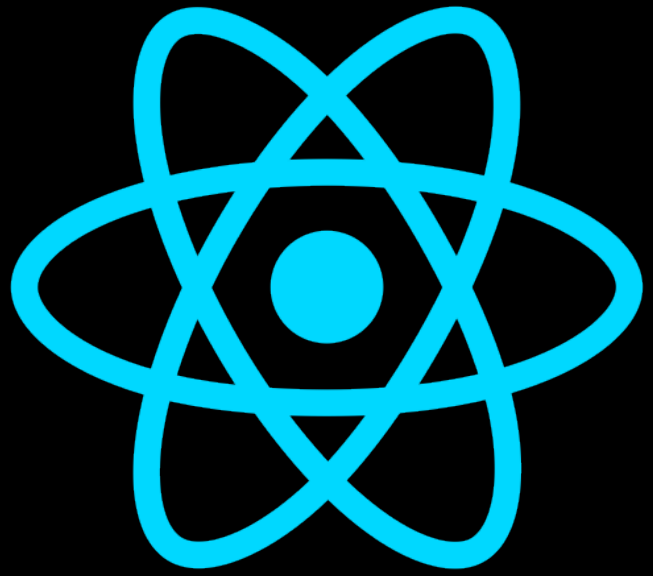
- za izradu objekta *Context* koristi se naredba:
  `const MyContext = React.createContext(defaultValue);`
- React.createContext nam u MyContext varijablu daje dvije komponente:
  - **`MyContext.Provider`** – komponenta koja pruža vrijednosti
  - **`MyContext.Consumer`** – komponenta koja koristi vrijednosti
- ako **`MyContext.Provider`** ne pruža nikakve podatke, **`MyContext.Consumer`** prima standardne vrijednosti (**`defaultValue`**) koje su definirane kod kreiranja *contexta*

Vježba 5.16: React Context

Literatura

- https://reactjs.org/docs/context.html
- https://www.taniarascia.com/using-context-api-in-react/

Osnovni koncepti ReactJS-a



React HOC - Komponente višeg reda

React HOC

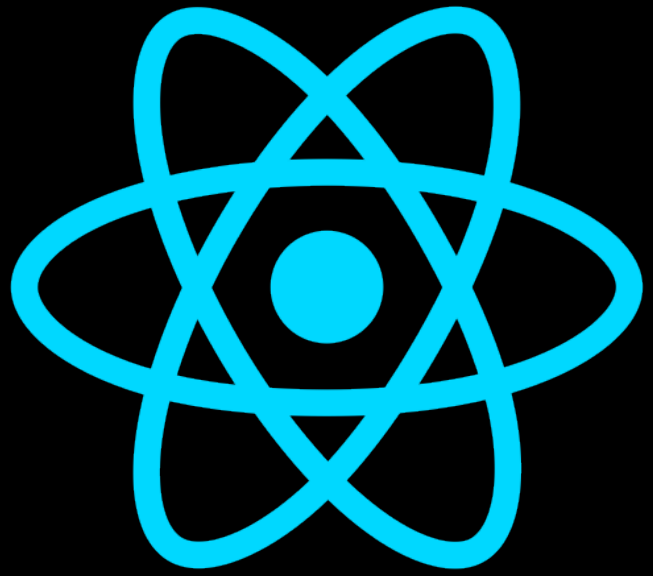
- komponente višeg reda - *Higher Order Components (HOC)*
- funkcija koja uzima komponentu i vraća novu komponentu
- napredna tehnika u Reactu za ponovnu upotrebu logike komponente
- obrazac koji proizlazi iz kompozicijske naravi Reacta
- mogu usporediti s funkcijama višeg reda u JavaScriptu, koje primaju druge funkcije kao argument, a isto mogu vraćati funkciju (map(), filter(), itd.)

Vježba 5.17: HOC

Literatura

- https://flaviocopes.com/react-higher-order-components/
- https://reactjs.org/docs/higher-order-components.html

Osnovni koncepti ReactJS-a



Stilizacija (CSS)

CSS u Reactu

- Najčešći načini za stilizaciju komponenti u Reactu:
  - CSS Stylesheet
  - Inline stilovi
  - CSS Modules
- Primjer komponente:

CSS *Stylesheet*

```jsx
import React from "react";
import "./ShadowCard.css";

export default function Card() {
 return (
   <p className="shadow-card ">
     Dobar dan!
   </p>
 );
}
```

ShadowCard.js

```css
.shadow-card {
        border-radius: 8px;
        box-sizing: border-box;
        margin: 24px;
        padding: 24px;
        box-shadow: 0 0 8px 0
darkgreen;
        width: 200px;
        color: darkgreen;
        font-weight: bold;
}
```

ShadowCard.css

ALGEBRA

Inline stilizacija

```
.shadow-card {
 border-radius: 8px;
 box-sizing: border-box;
 margin: 24px;
 padding: 24px;
 box-shadow: 0 0 8px 0 darkg
 width: 200px;
 color: darkgreen;
 font-weight: bold;
}
```

ShadowCard.js

```
import React from "react";

const cardStyle = {
    borderRadius: 8,
    boxSizing: "border-box",
    margin: 24,
    padding: 24,
    boxShadow: "0 0 8px 0 darkred",
    width: 200,
    color: "darkred",
   fontWeight: "bold"
};

export default function Card() {
      return <p style={cardStyle}>Dobar
dan!</p>;
}
```

ALGEBRA

CSS *Modules*

```jsx
import React from "react";
import styles from "./ShadowCard.module.css";

export default function Card() {
 return (
   <p className={styles["shadow-card"]}>
     Dobar dan!
   </p>
 );
}
```

ShadowCard.js

```css
.shadow-card {
  border-radius: 8px;
  box-sizing: border-box;
  margin: 24px;
  padding: 24px;
  box-shadow: 0 0 8px 0 darkgreen;
  width: 200px;
  color: darkgreen;
  font-weight: bold;
}
```

ShadowCard.module.css

ALGEBRA

React Bootstrap

- Bootstrap, kao jedna od najpopularnijih CSS biblioteka, ima svoju React inačicu: **`react-bootstrap`**
- sve komponente u toj biblioteci čiste su React komponente i mogu se *importirati* i koristiti u JSX-u kao da su lokalno kreirane komponente

React Bootstrap - instalacija

1. `npm install --save bootstrap react-bootstrap`

2. unutar src/index.js datoteke dodati liniju:

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

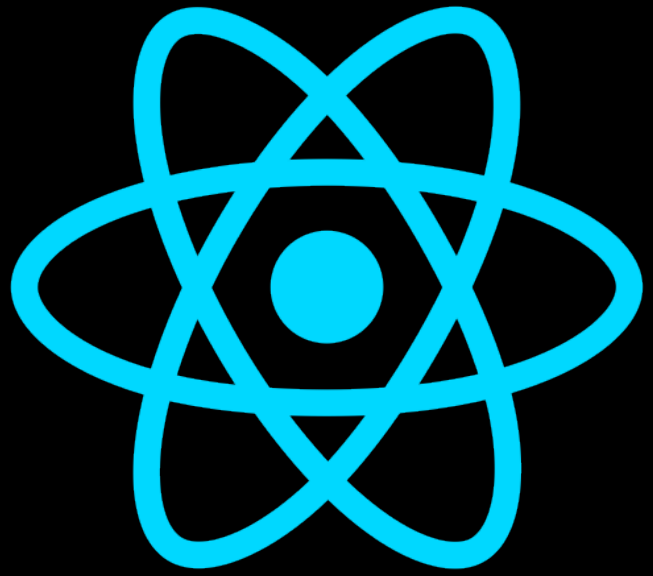
React Bootstrap - korištenje

```
import React from "react";
import Button from "react-bootstrap/Button";


export default function App() {
 return <Button variant="primary">Primary</Button>;
}
```

Literatura

- https://reactjs.org/docs/faq-styling.html
- https://www.w3schools.com/react/react_css.asp
- https://facebook.github.io/create-react-app/docs/adding-a-css-modules-stylesheet
- https://www.npmjs.com/package/classnames#usage-with-reactjs
- https://react-bootstrap.github.io/
- https://facebook.github.io/create-react-app/docs/adding-bootstrap

Napredni koncepti ReactJS-a



Strukturiranje React aplikacije

Strukturiranje React aplikacije

- aplikacija stvorena pomoću `create-react-app` već ima postavljenu osnovnu strukturu direkorija i datoteka
- međutim, kada stvaramo veće aplikacije, potrebno je tu strukturu dodatno razraditi
- obično razlikujemo sljedeće module (direktorije): *components*, *fragments (containers)*, *services (helpers, utilities)*

Direktorij *components*

- unutar direktorija *components* spremamo prezentacijske komponente
- to su komponente koje nemaju vlastito stanje već služe samo kao prezentacijski sloj
- u pravilu, jednostavne komponente, bez puno logike

Direktorij *fragments*

- *fragments* ili *containers*
- ovdje se nalaze složenije komponente, komponente koje imaju vlastito stanje ili su na neki način spojene na stanje aplikacije
- u takvim komponentama najčešće *importiramo* jednu ili više prezentacijskih komponenti iz direktorija *components* te ih slažemo u jednu cjelinu

Direktorij *services*

- *services, helpers, utilities*
- u ovom direktoriju ne nalaze se React komponente već obične JavaScript datoteke
- u tim datotekama čuvamo korisne funkcije koje se koriste kroz cijelu aplikaciju
- npr. datoteka *date.js* koja može sadržavati funkcije za manipulaciju i prikaz datuma

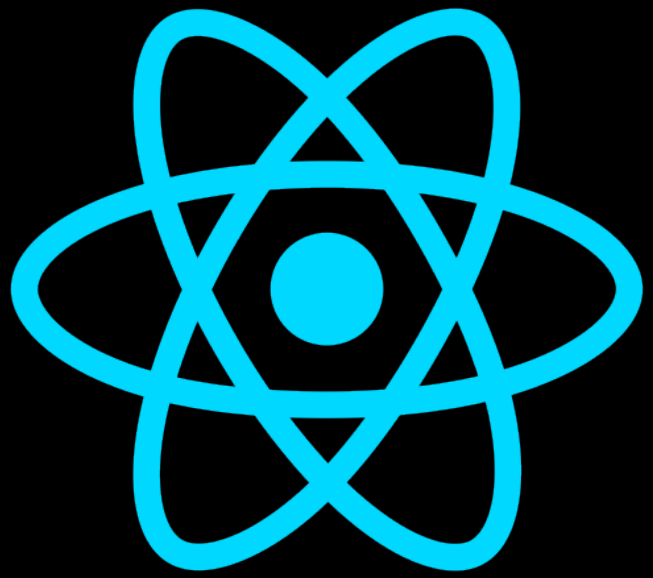
Organizacija direktorija prema funkcionalnosti

- vrlo velike aplikacije često imaju stotine komponenti
- u takvim aplikacijama nije dovoljno podijeliti datoteke unutar samo 3 direktorija jer će svaki od tih direktorija svejedno sadržavati velik broj datoteka i bit će nepregledan i teško održiv
- u takvim situacijama korisno je organizirati direktorije prema setovima funkcionalnosti

Organizacija direktorija prema funkcionalnosti

Primjer: pretpostavimo da radimo jednostavnu aplikaciju za naručivanje hrane:

- takva aplikacija ima nekoliko osnovnih koncepata: narudžba, restoran, korisnik...
- direktorije ćemo prvo podijeliti prema osnovnim setovima funkcionalnosti - imat ćemo direktorije: *orders*, *restorants, users*…
- u svakom od tih direktorija nalazit će se datoteke vezane samo za konkretnu funkcionalnost - npr. u *users* direktoriju bit će samo komponente i servisi vezani za koncept korisnika
- svaki od tih direktorija bit će organiziran prema strukturi *components/ fragments/services* kako bi i dalje postojala jasna razlika između složenih i jednostavnih komponenti te datoteka koje sadrže *utility* funkcije



Napredni koncepti ReactJS-a

Usmjeravanje u React aplikacijama (*React Router*)

Ponavljanje: *Single-page apps*

- web-aplikacije koje se sastoje od samo jedne stranice
- stranica u interakciji s korisnikom vrši dinamičko prepisivanje trenutnog sadržaja umjesto učitavanja nove stranice
- svi potrebni resursi (HTML, CSS, JavaScript, zvuk, video, slikovne datoteke) preuzimaju se učitavanjem jedne stranice ili se dinamički učitavaju po potrebi
- stranica se ni u jednom trenutku ponovno ne učitava
- React aplikacije su *single page* aplikacije

Ponavljanje: React Router

- biblioteka koja omogućava navigaciju po React aplikaciji putem web-adrese
- pozivom na neku web adresu, React aplikacija se otvara u unaprijed definiranom stanju:
  - npr. *my-react-app.com/pets* - React aplikacija se otvara u stanju u kojem prikazuje sve kućne ljubimce
  - npr. *my-react-app.com/pets*/dogs - React aplikacija se otvara u stanju u kojem prikazuje samo pse

React Router - instalacija

- Za instalaciju React Router biblioteke u postojeću React aplikaciju, potrebno je u *command line interfaceu* (CLI) izvršiti sljedeću naredbu:

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

React Router

U React Routeru razlikujemo dva osnovna koncepta:

1. **rute** (engl. *routes*) - služe za povezivanje React komponente s određenom web-adresom (rutom), odnosno određuju koja komponenta će se prikazati na kojoj web-adresi
2. **poveznice** (engl. *links*) - služe za navigiranje na određenu web-adresu unutar React aplikacije

React Router - *Route*

Kako bismo definirali rute, koristimo komponentu *Route* koju možemo *importirati* iz biblioteke *react-router-dom* :

```
import { Route } from 'react-router-dom';
...
 return (
          <div>
                    <Route exact path="/" component={Home} />
           <Route path="/pets/dogs" component={Dogs} />
          <Route path="/pets" component={Pets} />
      </div>
);
```

React Router - *Switch*

Unutar *Switch* komponente možemo ugnijezditi više *Route* komponenti. Time osiguravamo da će u jednom trenutku biti prikazana samo jedna komponenta.

```
import { Route, Switch } from 'react-router-dom';
...
 return (
        <Switch>
                <Route exact path="/" component={Home} />
         <Route path="/pets/dogs" component={Dogs} />
        <Route path="/pets" component={Pets} />
    </Switch>
);
```

React Router - *Link*

Komponenta *Link* omogućuje navigaciju unutar React aplikacije.

```
import { Link } from 'react-router-dom';
...
 return (
        <div>
                <Link to="/">Home</Link>
        <Link to="/pets">See all pets</Link>
                <Link to="/dogs">See just dogs</Link>
        </div>
);
```

React Router - *BrowserRouter*

Komponentu *BrowserRouter* treba staviti na sam vrh DOM-a React aplikacije kako bismo omogućili ispravan rad React Router biblioteke:

```
import { BrowserRouter } from 'react-router-dom';


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

React Router - dodatne komponente

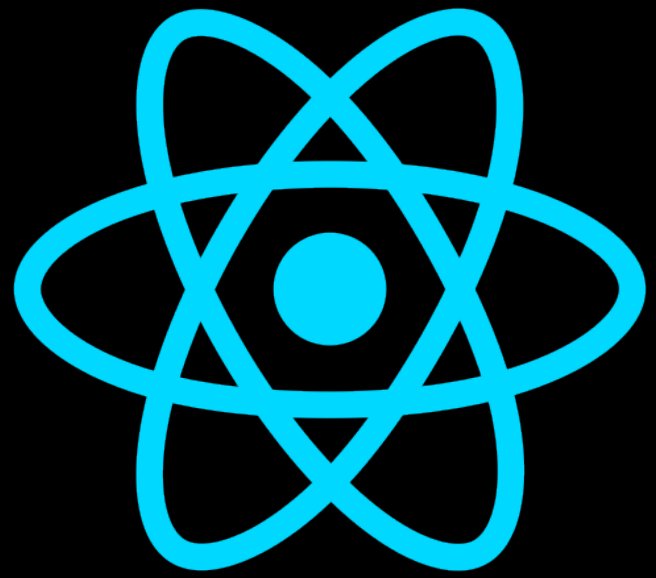
- *NavLink* - posebna vrsta komponente *Link* koju je moguće dodatno stilizirati za slučaj da trenutna web-adresa odgovara ruti na koju taj *link* vodi
- *Redirect* - komponenta koja omogućuje navigaciju na određenu rutu preko JSX-a

Vježba 5.18: React Router

Literatura

- https://reacttraining.com/react-router/web

Napredni koncepti ReactJS-a

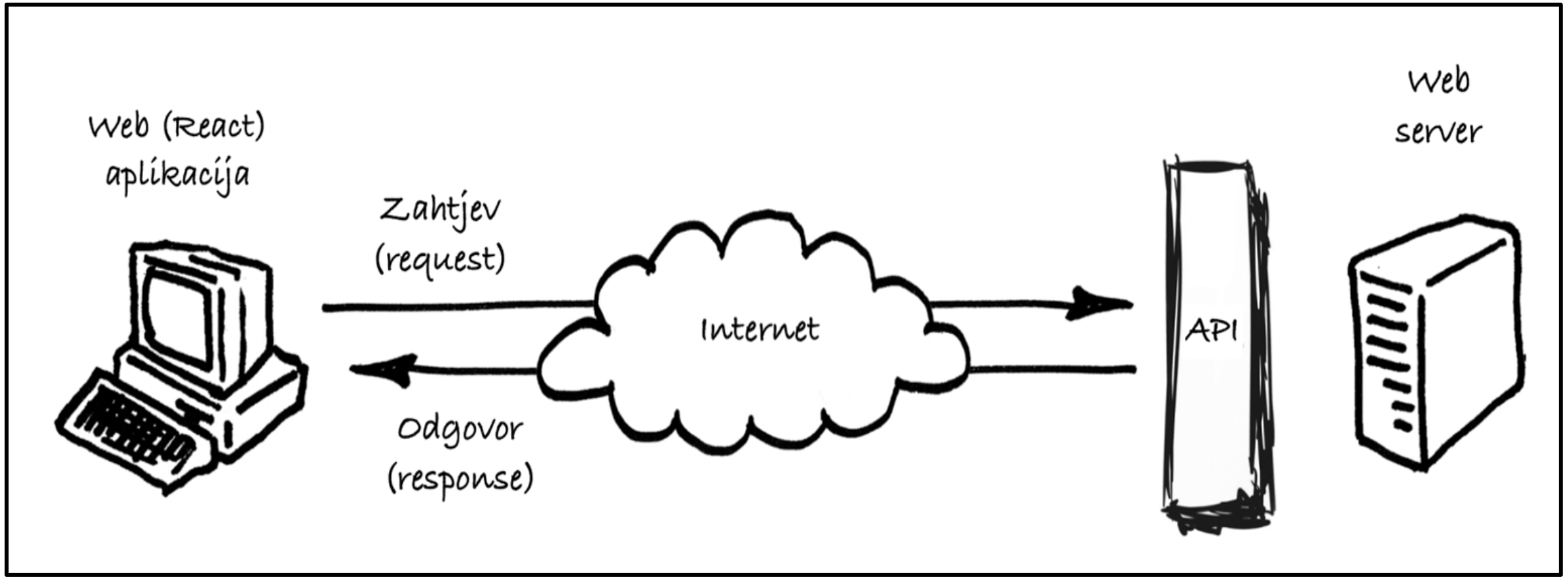


Povezivanje na API

API

- API = **A**pplication **P**rogramming **I**nterface
- API-ji omogućavaju aplikacijama da komuniciraju jedna s drugom
- nas zanimaju mrežni API-ji koji služe za komunikaciju web-aplikacija (React aplikacija) s poslužiteljem na kojem se nalaze podaci

API


Web (React)
aplikacija
Zahtjev
(request)
Internet
Odgovor
(response)
API
Web
server

JSON API

- JSON = **J**ava**S**cript **O**bject **N**otation
- JSON API prima zahtjeve i vraća odgovore u JSON formatu
- danas najpopularniji i vrlo rasprostranjen

RESTful API

- REST = **RE**presentational **S**tate **T**ransfer
- konvencija za organizaciju arhitekture mrežnih API-ja
- glavni koncept: *resurs*
- svaki resurs može biti:
  - dohvaćen (GET)
  - kreiran (POST)
  - promijenjen (PUT)
  - djelomično primijenjen (PATCH)
  - izbrisan (DELETE)

Vježba 5.19: REST API

GraphQL

- GraphQL = **Graph Q**uery **L**anguge, jezik upita baze graf
- moguće ga je koristiti u svakom slučaju kada poslužitelj (*server*) ili baza podataka podržavaju GraphQL programsko sučelje
- dopušta klijentu da točno definira upit za željene podatke
- GraphQL upiti uvijek vraćaju predvidljive rezultate
- Aplikacije koje koriste GraphQL brze su i stabilne jer one same kontroliraju podatke koje primaju, a ne poslužitelj

Apollo

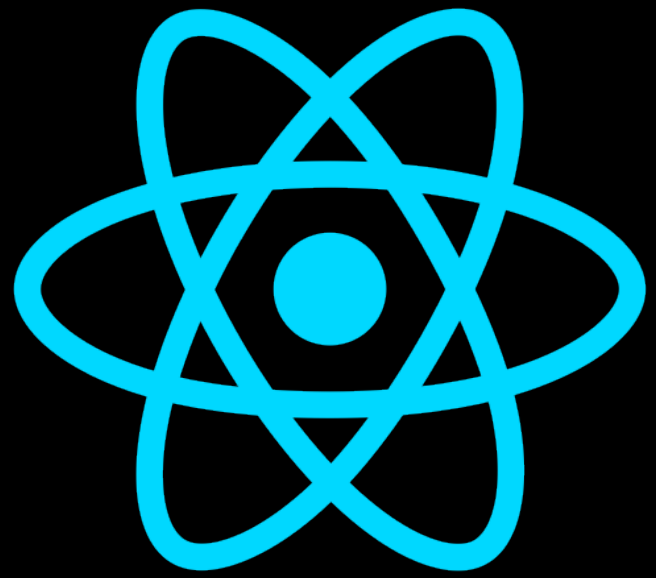
- jedna od najpopularnijih biblioteka koja nam nudi integraciju GraphQLa unutar React aplikacije
- pomoću biblioteke Apollo možemo definirati koja komponenta aplikacije zahtieva podatke s GraphQL API-ja
- komponenta koja traži podatke može biti u tri stanja:
  - učitavanje (*loading*)
  - uspješno završen upit (*success*)
  - pojavila se greška (*error*)

Vježba 5.20: GraphQL

Literatura

- https://medium.com/@perrysetgo/what-exactly-is-an-api-69f36968a41f
- https://www.restapitutorial.com/
- https://www.robinwieruch.de/react-fetching-data
- https://graphql.org/
- https://graphql.org/swapi-graphql
- https://www.apollographql.com/docs/react/essentials/get-started/

Napredni koncepti ReactJS-a



Napredno upravljanje stanjima

Skladišta stanja (engl. *stores*)

- u većim React aplikacijama često se pojavljuje potreba da više komponenti dijeli isto stanje aplikacije
- u jednostavnijem slučaju, možemo sve komponente koje dijele isto stanje staviti kao *child* komponente unutar iste *parent* komponente - tada *parent* komponenta preko *propsa* prosljeđuje isti dio stanja svim komponentama
- često to nije praktično rješenje te koristimo napredne biblioteke za upravljanje stanjima - MobX i Redux

Redux i MobX

- Redux i MobX su trenutno najpopularnije biblioteke koje za spremanje podataka koriste se skladišta stanja (*state store*)
- aplikacije najčešće imaju samo jedno skladište stanja koje se onda može sastojati od više manjih dijelova odnosno modula
- svaki dio se brine za odvojeni dio aplikacije i u pravilu oni bi trebali biti neovisni jedni o drugima

Skladišta stanja

Mobx	Redux
Biblioteka za jednostavno upravljanje stanjima uz pomoć TFRP (engl. *transparently applying functional reactive programming*)	JavaScript biblioteka za upravljanjem stanja unutar aplikacije.
Biblioteka napisana u JavaScriptu	Biblioteka napisana u ES6
Može se koristiti više skladišta stanja	Može se koristiti samo jedno skladište stanja
Uglavnom se koristi za jednostavnije i manje aplikacije	Uglavnom se koristi za kompleksne i velike aplikacije
Manje skalabilanosti od Reduxa	Koristi se za skalabilne aplikacije
Jako dobre performanse	Nije baš efikasan
Teže otklanjanje grešaka i poprilično loši razvojni alati	Lakše otklanjanje grešaka i izrazito dobri razvojni alati
Teže održavanje aplikacije	Lakše održavanje aplikacije

Redux

- biblioteka Redux je odličan primjer za skladište stanja jer je veoma ekspresivna i ima točno određen oblik
- Redux *store* (skladište stanja) sastoji se od više povezanih dijelova te ima dosta restrikcija kojih se treba pridržavati
- jedna od najvažnijih restrikcija je ta da može postojati samo jedan Redux *store*, te se ne smije direktno modificirati (postoje propisane konvencije načina na koji se smije mijenjati)

Redux

- nije dozvoljeno modificirati podatke u trenutnom stanju zato da biblioteka lakše može odrediti koje točno vrijednosti treba promijeniti i tako smanjiti broj komponenti koje će reagirati na promjenu podataka
- svaka promjena koja se želi pohraniti u Redux *store* mora proći kroz funkciju *reducer*
- *reduceri* su jedan od osnovnih koncepata Reduxa

Redux - *reducers*

- svaki *reducer* kao parametre prima trenutno stanje i akciju koja ga je pokrenula
- svaki *reducer* vraća novo stanje i pritom ne modificira ulazne podatke
- najčešće se radi kopija trenutnog stanja i ono se zatim modificira te vraća kao izlaz iz *reducera*

Redux - *reducers*

- Redux će koristiti izlaz *reducer* funkcije za analizu promjena te će promijeniti *redux store* prema tome
- sve *reducer* funkcije moraju biti čiste (engl. *pure*) što znači da, osim što ne smiju modificirati ulazne parametre, ne smiju vršiti nikakve asinkrone akcije
- ako se asinkrono mijenja vrijednost stanja, biblioteka neće moći reagirati na promjenu

Redux - *reducers*

```
function todoApp(state = initialState, action) {
        switch (action.type) {
                case 'SET_VISIBILITY_FILTER':
                        return Object.assign({}, state, { visibilityFilter:
action.filter });
        default:
                return state;
        }
}
```

Redux - *reducers*

- u većini aplikacija najčešće će postojati samo jedna *reducer* funkcija koja će, ovisno o akciji koja ju je pozvala, vratiti drugačiji rezultat
- ako se *reducer* pokrene s akcijom na koju ne zna odgovoriti, dužan je vratiti stanje koje je primio kao parametar ili, ako to stanje nije predano, dužan je vratiti inicijalno stanje aplikacije

Redux - *actions*

- akcije su funkcije koje se pozivaju kako bi se izvršile promjene na *redux store*
- *reduceri* reagiraju na akcije i u ovisnosti o njima, mijenjaju stanje
- primjer jednog poziva funkcije *reducer*:

```
store.dispatch({ type : 'SET_VISIBILITY_FILTER' , filter : '42' });
```

Redux - *connect*

- *connect* je HOC komponenta koju pruža biblioteka Redux i u nju je moguće ugnijezditi bilo koju React komponentu kako bi dobila pristup i mogućnost promjene stanja aplikacije koje je spremljeno u Redux skladištu

Redux - *connect*

- *connect* prima dvije funcije:
  - **`mapStateToProps`** - funkcija koja ima pristup Redux stanju te može ekstrahirati bilo koji dio stanja; komponenta ugniježđena u *connect HOC* primit će željeni dio stanja kroz *propse*
  - **`mapDispatchToProps`** - funkcija koja omogućuje okidanje akcija namijenjenih za promjenu stanja aplikacije (spremljenog unutar Redux skladišta); po okidanju akcije, Redux biblioteka pozvat će sve registrirane *reducer* funkcije te će se promijeniti očekivani dio stanja

Redux - useSelector, useDispatch

- umjesto HOC *connect* komponente, možemo koristiti i *hookove* **`useSelector`** i **`useDispatch`**
- **`useSelector`** (umjesto mapStateToProps) - služi za dohvaćanje željenog dijela stanja aplikacije koji je zapisan u Redux skladištu
- **`useDispatch`** (umjesto mapDispatchToProps) - omogućuje okidanje akcija koje služe za promjenu Redux stanja

Vježba 5.21: Redux

Vježba 5.22: Redux & Redux Thunk

MobX

- izašao 2015. godine
- biblioteka pisana u JavaScriptu
- može imati više od jednog skladišta (*store*) za pohranu podataka
- uglavnom se koristi za manje i jednostavnije aplikacije
- manje je skalabilan od Reduxa, ali daje bolje performanse na manjim aplikacijama

MobX

- dozvoljava korištenje većeg broja skladišta podataka koja su međusobno neovisna
- skladište podataka čini jedna klasa koja se instancira samo jednom (*singleton*) i sadrži sve podatke koji bi se trebali nalaziti u skladištu zajedno s metodama (koje mogu biti i asinkrone) koje izravno mijenjaju stanje skladišta
- kako bi se objasnio rad biblioteke Mobx potrebno je prvo objasniti pojam dekoratora u programskom jeziku JavaScript

JavaScript dekoratori

- dekoratori su funkcije sa specijaliziranom namjenom, koriste se za modificiranje ili za dodavanje dodatnih informacija na klase, njezine atribute ili metode
- dekoratori su trenutno u drugoj fazi postupka uvrštenja novih funkcija u standard JavaScript
- dekoratore je jednostavno prepoznati po znaku “@” ispred imena dekoratora
- korištenje dekoratora nije obavezno jer se ne nalaze u standardu programskog jezika JavaScript; u zamjenu, moguće je korištenje funkcijskog oblika dekoratora

JavaScript dekoratori

- kako bi se omogućilo korištenje dekoratora, potrebno je dodati i dodatak *transform-decorators-legacy* za modul *Babel*
- funkcija modula *Babel* je da prevodi JavaScript kôd pisan u novijem standardu jezika JavaScript u kôd pisan u nekom starijem standardu

JavaScript dekoratori

Primjer dekoratora:

```
import { observable } from 'mobx' ;

class OrderLine {
        @observable price = 0 ;
    @observable amount = 1 ;
    @computed get total() {
        return this .price * this .amount;
    }
```

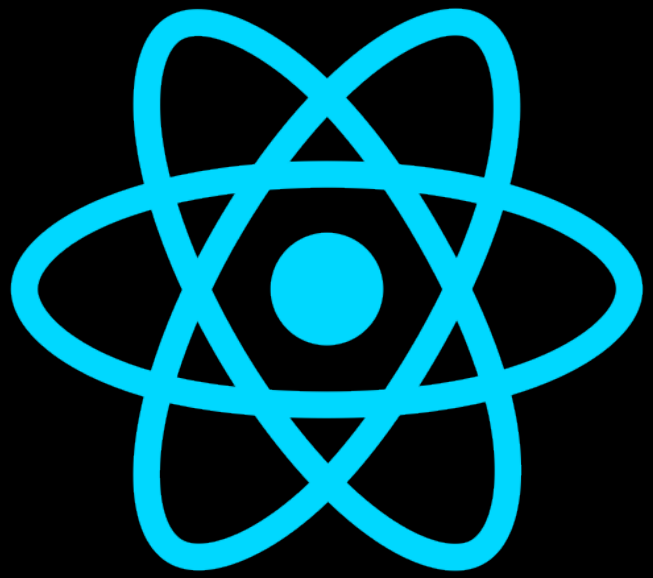
MobX dekoratori

- **`@observable`**: postavlja se ispred atributa u skladištu čija vrijednost se može u nekom trenutku promijeniti i tako promijeniti stanje aplikacije (OrderLine.price i OrderLine.amount u prethodnom primjeru)
- **`@computed`**: predstavlja vrijednosti koje nemaju izravnu vrijednost, ali se mogu izračunati korištenjem drugih vrijednosti iz iste klase skladišta (OrderLine.total u prethodnom primjeru)
- **`@action`**: opisuje metodu unutar klase skladišta koja mijenja stanje aplikacije (na sinkroni ili asinkroni način)
- **`@observer`**: koristi se nad komponentama koje trebaju reagirati na promjene unutar navedenog skladišta

Vježba 5.23: MobX

Literatura

- https://www.educba.com/mobx-vs-redux/
- https://codeburst.io/mobx-vs-redux-with-react-a-noobs-comparison-and-questions-382ba340be09
- https://mobx.js.org/getting-started.html



Napredni koncepti ReactJS-a

Redux-Saga

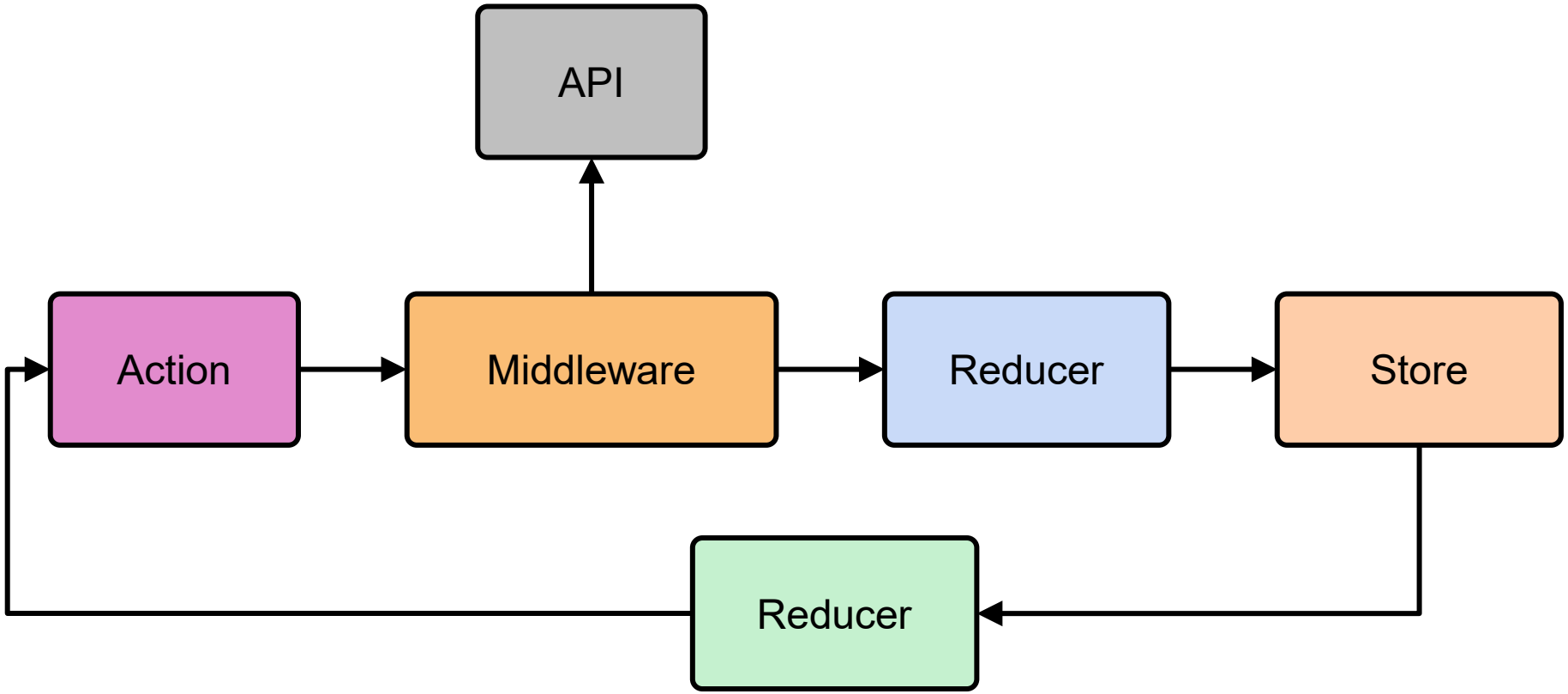
Redux-Saga

- biblioteka koja nadograđuje koncept Reduxa
- olakšava primjenu nuspojava aplikacije (*side effects* - dohvaćanje podataka, pristupanje *cacheu* internetskog preglednika, ...)
- omogućuje bolje performanse, lako testiranje te donosi alate za lakše pronalaženje pogrešaka
- instalacija: `npm install --save redux-saga`

Redux-Saga

- saga je poput zasebne niti (engl. *thread*) u React aplikaciji, odgovorna isključivo za nuspojave
- može se pokrenuti, pauzirati ili otkazati iz glavne aplikacije normalnim Redux akcijama
- svaka saga ima potpuni pristup Redux stanju
- svaka saga može slati Redux akcije

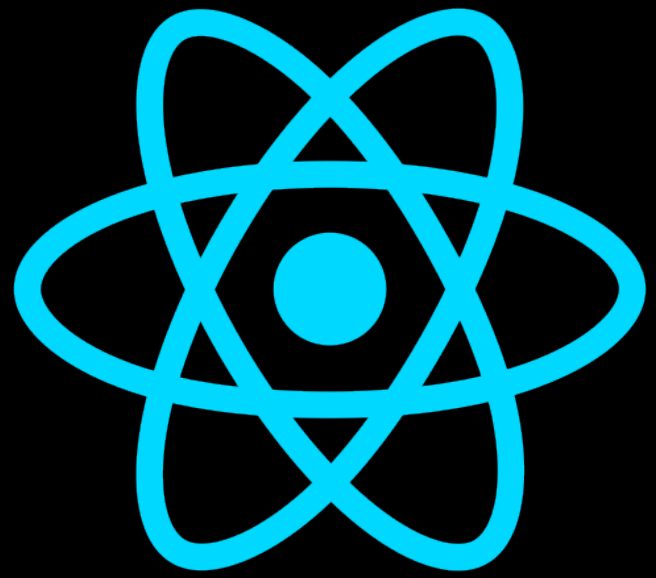
Redux-Saga



Literatura

- https://redux-saga.js.org
- https://scalac.io/redux-saga-handle-side-effects-2/
- https://medium.com/@adlusk/a-newbs-guide-to-redux-saga-e597d8e6c486

Napredni koncepti ReactJS-a



React Native

React Native

- *React Native* je JavaScript biblioteka za izradu višeplatformnih mobilnih aplikacija korištenjem slične metodologije kao i ReactJS
- podržane platforme su Android i iOS
- koristi se za izradu nekih od najpopularnijih mobilnih aplikacija, kao što su Facebook, Instagram, Skype i druge
- programiranjem s React Native ne stvara se mrežna ni hibridna aplikacija, već prava mobilna aplikacija koja se ne razlikuje od izvorne pisane u Javi / Kotlinu (Android) ili Objective C / Swiftu (iOS)

React Native

- za razliku od React biblioteke koja koristi mješavinu jezika JavaScript i HTML za prikaz korisničkog sučelja nazvanu JSX, za mobilne se aplikacije ne smije koristiti HTML jer ne postoji DOM kao na web-stranici
- umjesto toga koriste se posebne React Native komponente koje služe kao adapter između JavaScripta i izvornog koda platforme
- JavaScript logika se ne prevodi u izvorni kod, već se pokreće prilikom izvođenja aplikacije

React Native

Osnovni elementi	
Element:	**Opis:**
`<View>`	Ekvivalent je `<div>` u web-aplikacijama
`<TouchableOpacity>`	Pretvara bilo koju komponentu unutar *tagova* u površinu koja reagira na dodir, te može pritiskom izvršiti neku akciju
`<Text>`	Služi za ispisivanje teksta
`<Image>`	Prima relativnu adresu slike te ju prikazuje

React Native

- React filozofija prenosi se na React Native tako što korisničko sučelje postaje funkcija trenutnog stanja, što je ključan element React biblioteke
- prilikom promjene stanja, React brine o osvježavanju dijelova sučelja koji ovise o toj promjeni
- React Native pruža mogućnost pisanja dijela aplikacije koristeći izvorno programiranje za ciljanu platformu:
  - na taj se način dijelovi koje je teže prilagoditi React sustavu mogu napisati koristeći Javu/Kotlin ili Objective C / Swift.

React Native

- *Hello World* primjer:

  https://facebook.github.io/react-native/docs/tutorial

Literatura

- https://facebook.github.io/react-native/
- https://medium.com/@alexmngn/from-reactjs-to-react-native-what-are-the-main-differences-between-both-d6e8e88ebf24

Vježba 5.24: *TODO app*

Vježba 5.25: *TODO app* - napredno

Hvala na pažnji!