

Kurs Front-End **Developer**
ReactJS

ReactJS



ReactJS to biblioteka do budowania interfejsu użytkownika w JavaScript (niektórzy nazywają to Framework 😊). Stworzona została przez programistę Facebook – Jordana Walke.

ReactJS zdobył ogromną popularność wśród programistów i jest jedną z tych bibliotek, w których całkiem przyjemnie się pisze kod 😊 Jak zostało to już wspomniane, raczej nie jest kompletnym Framework'iem, a biblioteką.

ReactJS – JSX + Babel.js

JSX - jest rozszerzeniem składni języka JavaScript, który z wyglądu przypomina HTML wewnątrz JavaScript 😊

Budując aplikacje w ReactJS, kod pisze się w JSX. Składnia ta nie jest wspierana bezpośrednio przez przeglądarki, więc potrzebujemy narzędzia, które JSX przetworzy na kod zrozumiały dla przeglądarek.

Babel.js - zamienia JSX na kod przeglądarkowy. Dodatkowo zamienia on także kod napisany w standardzie ES6 na kod zrozumiały także dla starszych przeglądarek – czyli napisany w ES5.



```
let name = "Krystian";  
let elementHeading = <h1>Witaj, {name}</h1>; // Witaj Krystian  
let sumaParagraph = <p>{ 2 + 2 }</p>; // 4
```

ReactJS – Uruchomienie kodu JSX (I-***)

Aby zacząć programować w ReactJS wystarczy do pliku HTML dopiąć 3 biblioteki JavaScript w sekcji `<head>`: `react`, `react-dom` i `babel.js`.

```
<script src="https://unpkg.com/react/umd/react.development.js"></script>
<script src="https://unpkg.com/react-dom/umd/react-dom.development.js"></script>
<script src="https://unpkg.com/babel-standalone/babel.js"></script>
```

`Babel.js` nie działa automatycznie we wszystkich skryptach na stronie. Kod JSX, który ma być obsługiwany przez `babel.js` wymaga oznaczenia przez pomocy atrybutu `type` w znaczniku `<script>`. Wartością atrybutu `type` może być `text/babel` lub `text/jsx`.

```
<script type="text/babel">
  ReactDOM.render(
    <h1>Witaj Krystian!</h1>,
    document.getElementById("app")
  );
</script>
```

```
<script type="text/jsx">
  ReactDOM.render(
    <p>Akademia 108 &copy;</p>,
    document.getElementById("footer")
  );
</script>
```

ReactJS – node + npm (yarn)

Jednak najczęstszym sposobem tworzenia i pracy z aplikacjami napisanymi w ReactJS wymaga instalacji środowiska `node.js` oraz użycia menadżera pakietów `npm`, bądź `yarn` (przeważnie na maszynach UNIX'owych – OSX, Linux).

Instalacja `node.js` zgodnie z instrukcjami, zależnie od systemu operacyjnego

- Windows: <https://nodejs.org/en/download/package-manager/#windows>
- OSX: <https://nodejs.org/en/download/package-manager/#macos>
- Linux: <https://nodejs.org/en/download/package-manager/#debian-and-ubuntu-based-linux-distributions>



Następnie w terminalu (wierszu poleceń) sprawdzamy czy `node.js` i `npm` działają 😊

```
$ node --version && npm -v
```

W terminalu powinny wyświetlić się aktualne wersje `node.js` i `npm`.



Dla systemów UNIX'owych (OSX, Linux) ewentualna instalacja menadżera pakietów Yarn: <https://yarnpkg.com/lang/en/docs/install/#mac-stable>

I sprawdzenie czy `yarn` działa prawidłowo 😊

```
$ yarn -v
```

W terminalu powinna wyświetlić się aktualna wersja menadżera pakietów `yarn`.



ReactJS – create-react-app

Aby stworzyć nowy projekt ReactJS najlepiej użyć kreatora aplikacji React'owych `create-react-app` stworzonego przez programistów Facebook.

Instalacja globalna pakietu `create-react-app` za pomocą npm 😊

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

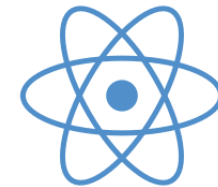
A następnie sprawdzenie, czy kreator działa prawidłowo 😊

```
$ create-react-app --version
```

W terminalu powinna wyświetlić się aktualna wersja kreatora `create-react-app`.

`create-react-app` – tworzy dla Ciebie strukturę katalogów i plików aplikacji ReactJS oraz zawiera wszystkie potrzebne na początek narzędzia (np. webpack). Dodatkowo pozwala od razu na uruchamianie, testowanie i budowanie aplikacji ReactJS.

Create React App



Official. No Setup. Minimal.

ReactJS – Pierwsza aplikacja ReactJS

Tworzymy pierwszą aplikację w ReactJS 😊

```
$ create-react-app react-counter
```

Uruchomi się kreator, który powinien stworzyć folder `react-app` z naszą aplikacją oraz zainstalować wszystkie pakiety potrzebne do uruchomienia aplikacji w ReactJS.

Następnie wchodzimy do folderu z utworzoną aplikacją i uruchamiamy wbudowany w naszą aplikację ReactJS serwer WWW. Po uruchomieniu się serwera, otworzy się domyślna przeglądarka z naszą aplikacją 😊

```
$ cd react-counter
```

```
$ npm start
```

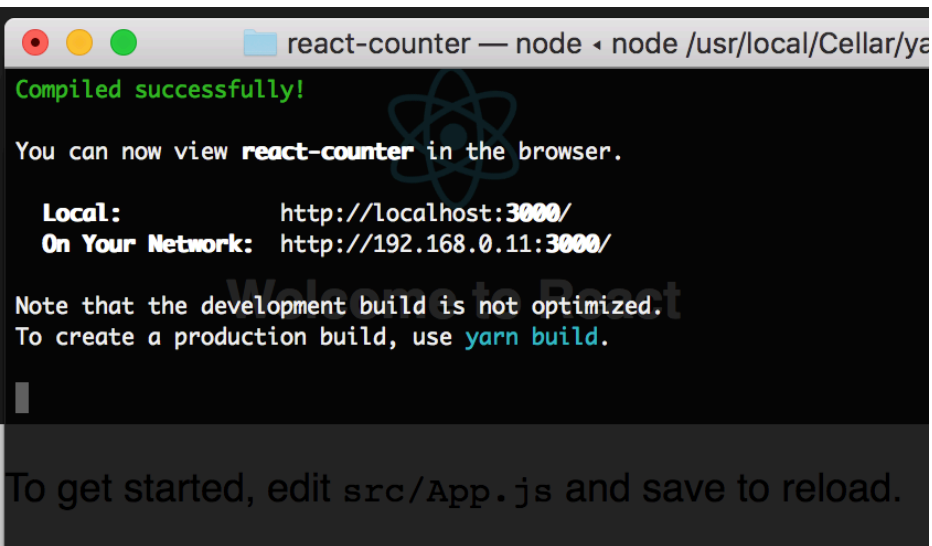
Lub `yarn start` jeśli używamy systemu UNIX'owego i mamy zainstalowany manager pakietów Yarn.

Ważne!

Kombinacja klawiszy `CTRL + C` zatrzymuje serwer WWW 😊 - Zrób to za każdym razem, gdy zamykasz okno konsoli/terminala - w przeciwnym wypadku serwer będzie dalej uruchomiony jako proces w tle i będzie zajmował port (domyślnie port 3000), pod którym uruchomiłeś serwer. Pojawiają się problemy z ponownym uruchomieniem serwera na domyślnym porcie 😊

ReactJS – Uruchomiony serwer z projektem

Widok uruchomionego serwera www w terminalu



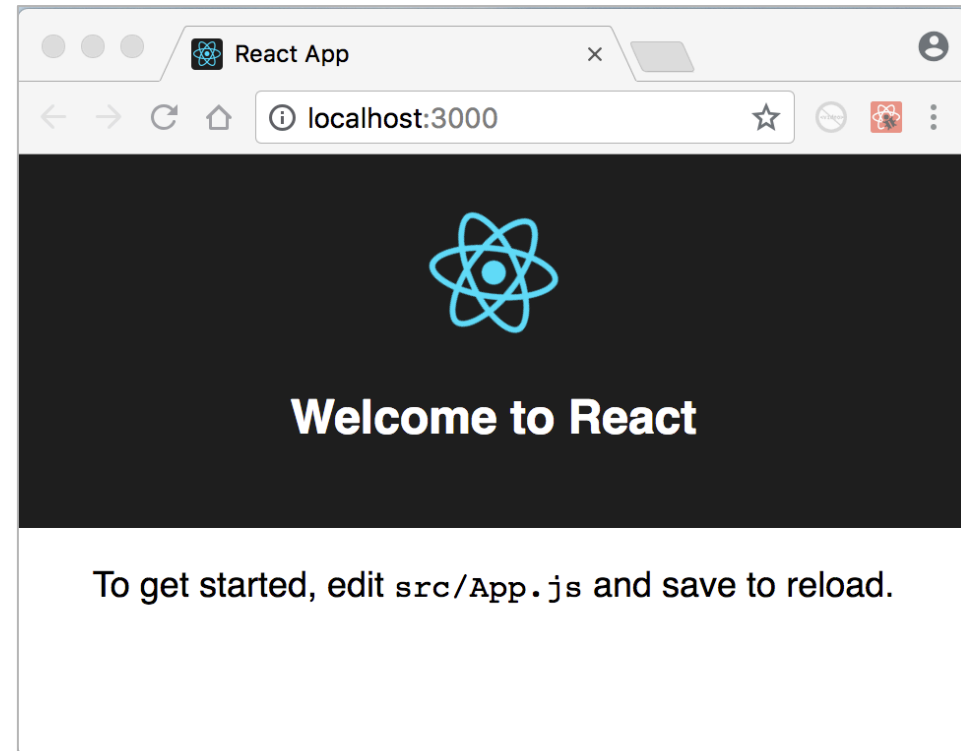
```
react-counter — node ◀ node /usr/local/Cellar/yarn
Compiled successfully!
You can now view react-counter in the browser.

Local:      http://localhost:3000/
On Your Network:  http://192.168.0.11:3000/

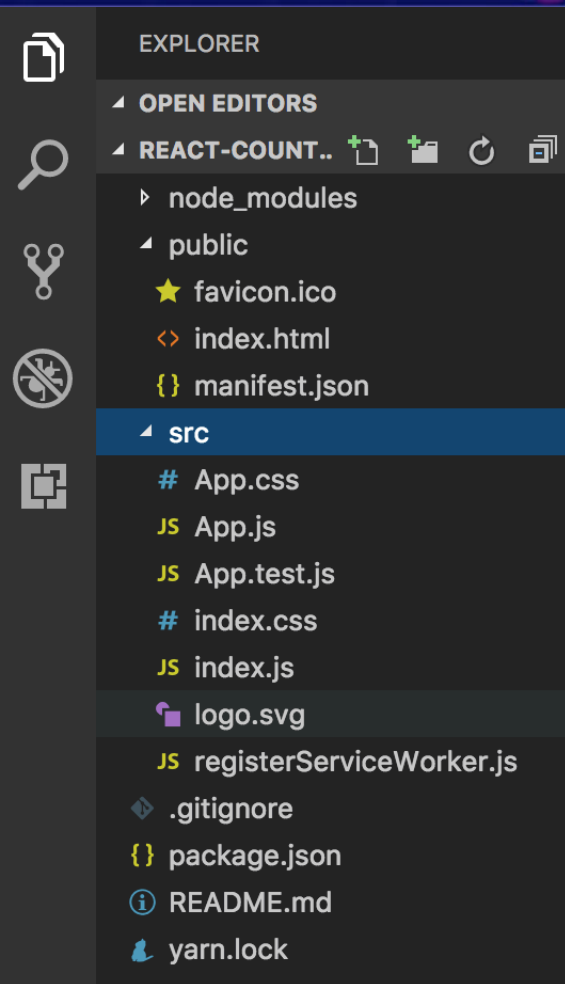
Note that the development build is not optimized.
To create a production build, use yarn build.

To get started, edit src/App.js and save to reload.
```

Widok uruchomionej aplikacji w przeglądarce



ReactJS – Struktura plików nowego projektu



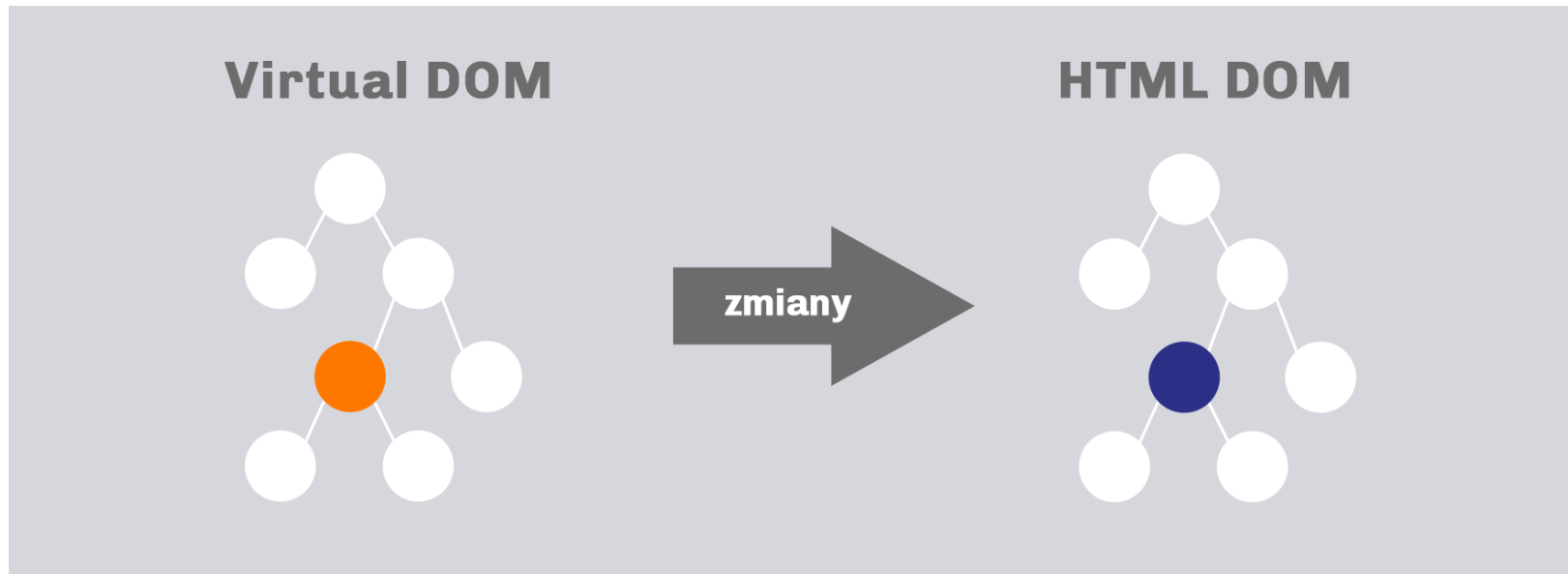
- `node_modules` – w tym folderze znajdują się wszystkie potrzebne do uruchomienia pakiety `node.js` – są tam wrzucane przez kreator podczas instalacji
- `public` – w tym folderze znajdują się pliki HTML ikonki itp.
- `src` – to najważniejszy folder, w którym znajdują się pliki Twojego projektu.
 - ✓ `index.js` – plik wejściowy aplikacji tzw. `entry point`. To tutaj właśnie komponent nadrzędny dodawany jest do DOM.
 - ✓ `App.js` – nadrzędny komponent aplikacji. To w nim znajduje się kod bezpośrednio importowany do pliku `index.js` i wyświetlany w przeglądarce
 - ✓ `serviceWorker.js` – plik zarządzający API, które pozwala na cache'owanie elementów. Stworzone z myślą o użytkownikach korzystających z aplikacji `offline`. Działa tylko w wersji produkcyjnej
- `.gitignore` – pliki ignorowane przez repozytorium GIT
- `package.json` – plik `node.js` zarządzający zależnościami bibliotek projektu
- `Readme.md` – plik zawierający opis Twojej aplikacji

ReactJS – VIRTUAL DOM

VirtualDOM jest wirtualną wersją oryginalnego drzewa DOM w HTML, zbudowaną w oparciu o to istniejące drzewo DOM wyświetlane przez przeglądarkę.

Dokonując zmiany w komponencie ReactJS, dokonujemy zmiany w VirtualDOM, a nie bezpośrednio w HTML'owym DOM. Na podstawie tych zmian następuje porównanie VirtualDOM z oryginalnym drzewem DOM.

Następnie aktualizowane są tylko te elementy DOM, które tego wymagają, a reszta pozostaje bez zmian. Dzięki temu zmieniane są rzeczywiście tylko te elementy, które naprawdę wymagają aktualizacji.



ReactJS – RENDEROWANIE KONTENTU

ReactJS renderuje zawartość na stronie. Odbyna się to za pomocą metody `render()` w obiekcie ReactDOM.

W przypadku aplikacji licznika, głównym miejscem, gdzie dodawany jest komponent aplikacji App jest element HTML `div` o wartości `id="root"` znajdujący się w pliku `index.html` w folderze `public`.

W pliku `index.js` tzw. entry point dodawany jest właśnie komponent App do znacznika `div` o `id="root"` znajdującym się w pliku `index.html` w folderze `public`.

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

// plik index.js

```
import React from 'react';
import ReactDOM from 'react-dom';
import './index.css';
import App from './App';
import * as serviceWorker from './serviceWorker';

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

serviceWorker.unregister();
```

<!-- plik index.html -->

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>Counter App</title>
  </head>
  <body>
    <div id="root"></div>
  </body>
</html>
```

ReactJS – Komponenty

Komponent to obiekt ReactJS, który jest napisany w JSX i docelowo składa się z elementów React. Innymi słowy, kod napisany w JSX jest przetwarzany, a z niego tworzone elementy ReactJS (są one czystymi obiektami JS), które są węzłami w VirtualDOM. Każdy komponent może być reużywalny w innych komponentach aplikacji.

Komponentem w ReactJS może być zarówno klasa (odpowiada najczęściej za logikę aplikacji) jak i funkcja (odpowiada najczęściej za prezentację aplikacji).

W naszej aplikacji `react-counter` tworzymy plik `Counter.js` w folderze `src`, w którym umieścimy kod naszego nowego komponentu o nazwie `Counter`. Najpierw stworzymy komponent funkcyjny, a potem przerobimy go na komponent klasowy.

// komponent funkcyjny

```
import React from "react";
const Counter = () => {
  return(
    <div className="counter">Licznik</div>
  )
}
export default Counter;
```

// komponent klasowy

```
import React, {Component} from "react";
class Counter extends Component {
  render() {
    return(<div className="counter">Licznik</div>)
  }
}
export default Counter;
```

ReactJS – Komponenty

Aby móc zaimportować komponent `Counter` w innym miejscu w aplikacji, musimy go wcześniej z pliku `Counter.js` wyeksportować: `export default Counter;`

Kolejnym krokiem jest import komponentu `Counter` w komponencie głównym aplikacji `App`, a następnie musimy go wyrenderować w tym komponencie (np. `<Counter />`)

```
import React, { Component } from 'react';
import './App.css';
import Counter from './Counter';

class App extends Component {
  render() {
    return (
      <div className="App">
        <header className="App-header">
          <h1 className="App-title">Licznik w ReactJS</h1>
        </header>
        <Counter />
      </div>
    );
  }
}
export default App;
```

ReactJS – Komponenty

Ważne! W każdej klasie komponentu istnieje metoda `render()`, która może zwracać tylko jeden element HTML. Jeśli chcemy renderować więcej elementów HTML lub innych komponentów, to musimy objąć całą renderowaną zawartość w jeden znacznik HTML, będący kontenerem. W przeciwnym przypadku podczas kompilacji otrzymamy błąd:

```
./src/App.js
```

Syntax error: Adjacent JSX elements must be wrapped in an enclosing tag

// poprawnie

```
import React, { Component } from 'react';
import './App.css';
import Counter from './Counter';

class App extends Component {
  render() {
    return (
      <div className="App">
        <header className="App-header">
          <h1>Licznik w ReactJS</h1>
        </header>
        <Counter />
      </div>
    );
  }
}
export default App;
```

// BŁĘDNE

```
import React, { Component } from 'react';
import './App.css';
import Counter from './Counter';

class App extends Component {
  render() {
    return (
      <div className="App">
        <header className="App-header">
          <h1>Licznik w ReactJS</h1>
        </header>
      </div>
      <Counter />
    );
  }
}
export default App;
```

ReactJS – this.props

props (properties, właściwości komponentu) to obiekt, zawierający wszystkie atrybuty w JSX, przekazane do instancji komponentu.

Komponenty wykorzystują props'y podobnie jak elementy HTML wykorzystują atrybuty i służą do przekazania dodatkowych informacji do komponentów.

Dostęp do props'ów wewnątrz komponentu klasowego uzyskujemy poprzez odwołanie się przez słówko `this` i nazwę obiektu przechowującego jego właściwości – czyli props 😊. → `this.props`

```
// plik App.js

import React, { Component } from 'react';
import './App.css';
import Counter from './Counter';

class App extends Component {
  render() {
    return (
      <div className="App">
        <header className="App-header">
          <h1 className="App-title">Licznik w ReactJS</h1>
        </header>
        <Counter initialValue="108" />
      </div>
    );
  }
}
export default App;
```

```
// plik Counter.js

import React, {Component} from "react";

class Counter extends Component {
  render() {
    return(
      <div className="counter">
        Licznik: {this.props.initialValue}
      </div>
    )
  }
}

export default Counter;
```

ReactJS – Stan Komponentu – `this.state`

`state` (stan komponentu) to obiekt przechowujący aktualny stan komponentu. Dane przechowywane w obiekcie stanu komponentu są inicjowane w konstruktorze klasy tego komponentu poprzez odwołanie do niego `this.state` i mogą być uaktualniane podczas działania aplikacji za pomocą metody `this.setState()`.

Obiekt `state` posiadają tylko komponenty klasowe. Odczyt aktualnego stanu komponentu dokonujemy poprzez odwołanie do niego `this.state`, a aktualizować go można za pomocą metody `this.setState()`, która jako parametr przyjmuje obiekt lub funkcję, która zwraca (słowo kluczowe `return`) aktualne wartości tego obiektu.

Podczas aktualizacji stanu komponentu dochodzi do merge'owania (scalenia) przekazanego obiektu do metody `this.setState()` z aktualnym stanem komponentu, którego wartość można pobrać w klasie komponentu za pomocą wyrażenia `this.state`.

Jeśli podczas uaktualniania stanu komponentu prześlemy do metody `this.setState()` obiekt, którego właściwość do tej pory nie istniała w obiekcie `this.state`, to zostanie ona do niego dodana. Natomiast jeżeli właściwość już w nim istnieje, to jej wartość zostanie uaktualniona (nadpisana) – zakładając, że została ona zmieniona 😊

ReactJS – Stan Komponentu – this.state

ReactJS obserwuje zmiany stanów wszystkich komponentów klasowych.

W przypadku gdy stan danego komponentu się zmieni, ReactJS wywołuje ponownie metodę `render()` dla tego komponentu oraz jego wszystkich wszystkich komponentów potomnych, jeśli oczywiście takie posiada.

```
/* plik Counter.css */
```

```
.counter {  
  border: 1px solid darkblue;  
  padding: 30px;  
  width: 300px;  
  left: 0;  
  right: 0;  
  margin: 30px auto;  
}  
.counter span.value {  
  margin-left: 10px;  
  color: blue;  
}
```

```
// plik Counter.js
```

```
import React, {Component} from "react";  
import "./Counter.css";  
  
class Counter extends Component {  
  constructor(props) {  
    super(props);  
    this.state = {  
      counterValue: 0,  
    }  
  }  
  changeValue = () => {  
    this.setState( (prevState) => {  
      return({  
        counterValue: prevState.counterValue + 1,  
      })  
    });  
  }  
  render() {  
    return(  
      <div className="counter">  
        Licznik:  
        <span className="value">  
          {this.state.counterValue}  
        </span>  
        <button onClick={this.changeValue}>  
          Add 1  
        </button>  
      </div>  
    )  
  }  
}  
export default Counter;
```

ReactJS – Komunikacja pomiędzy komponentami

Komunikacja pomiędzy komponentami – istnieje nieraz potrzeba, aby komponenty miały w sobie zagnieżdżone inne komponenty i naturalnym jest, że muszą się między sobą komunikować.

Aby przekazać dane od rodzica do dziecka wykonujemy to za pomocą props'ów dziecka.

Rodzic → Dziecko (dane przekazujemy przez props'y dziecka)

Natomiast aby przekazać dane od dziecka do rodzica, musimy za pomocą props'ów dziecka przekazać metodę (tzw. callback) zdefiniowaną w klasie komponentu rodzica, a w klasie komponentu dziecka ją wywołać poprzez jego props'y, przekazując dane jako parametr tej metody.

Dziecko → Rodzic (przez props'y dziecka przekazujemy metodę rodzica tzw. callback)

Przebudujemy nasz licznik, aby komponent Counter miał dodatkowy komponent potomny (tzw. dziecko), który będzie służył do wyświetlania panelu z przyciskami Add 1 - Reset - ReInit (ButtonsPanel). W folderze src stworzymy dodatkowy plik dla nowego komponentu: ButtonsPanel.js.

ReactJS – Komunikacja pomiędzy komponentami

// plik ButtonsPanel.js

```
import React, {Component} from 'react';
import './ButtonsPanel.css';

class ButtonsPanel extends Component {

  resetOrReinitCounter = (reset) => {
    this.props.resetCounterValue(reset);
  }

  render() {
    return(
      <div className="buttons-panel">
        <button onClick={this.props.changeCounterValue}>Add 1</button>
        <button onClick={ () => this.resetOrReinitCounter(false) }>ReInit</button>
        <button onClick={ () => this.resetOrReinitCounter(true) }>Reset</button>
      </div>
    )
  }
}

export default ButtonsPanel;
```

/* plik ButtonsPanel.css */

```
button {
  margin: 10px 0 10px 10px;
  padding: 10px;
  background-color: #0066FF;
  color: #FFF;
  border: 1px solid #0066FF;
}
button:hover {
  background-color: #003399;
  border: 1px solid #003399;
  cursor: pointer;
}
```

Nowy komponent potomny (ButtonsPanel) dla komponentu Counter. ma 2 dodatkowe buttony. Pierwszy ReInit przywraca wartość licznika ustawionego przy inicjalizacji komponentu, a drugi Reset zeruje licznik. Każdy z tych buttonów uruchamia metodę resetCounter, która znajduje się w komponentcie rodzica (Counter.js) i jest przekazywana przez propsy do tego komponentu tj. ButtonsPanel.

ReactJS – Komunikacja pomiędzy komponentami

```
// plik Counter.js

import React, {Component} from 'react';
import './Counter.css';
import ButtonsPanel from './ButtonsPanel';

class Counter extends Component {
  constructor(props) {
    super(props);

    let initialValue = 0;

    if ( ! isNaN(this.props.initialValue) ) {
      initialValue = parseInt(this.props.initialValue);
    }

    this.state = {
      counterValue: initialValue,
    }
  }

  changeValue = () => {
    this.setState( (prevValue) => {
      return({
        counterValue: prevValue.counterValue + 1,
      });
    });
  }
}
```

// ciąg dalszy kodu pliku Counter.js na kolejnym slajdzie

Po lewej stronie jest zaktualizowana wersja komponentu `Counter`, która zawiera warunki sprawdzające, czy wartość `initValue` licznika została poprawnie ustawiona, oraz zawiera dodatkową metodę `resetCounter`, która zeruje wartość licznika lub ustawia mu wartość z inicjacji komponentu `Counter`.

Czy jest `Reset`, czy `ReInit` zależy od tego, który został kliknięty przycisk (`Reset`, czy `ReInit`) w komponencie pochodnym (`ButtonsPanel`) - Czyli jaka została przekazana wartość parametru (z komponentu potomnego `ButtonsPanel`) `resetCounter` w metodzie `resetCounter`.

ReactJS – Komunikacja pomiędzy komponentami

// ciąg dalszy treści pliku Counter.js (z poprzedniego slajdu)

```
resetCounter = (resetCounter) => {
  let initialValue = 0;

  if (!resetCounter) {
    if ( ! isNaN(this.props.initValue) ) {
      initialValue = parseInt(this.props.initValue);
    }
  }

  this.setState({
    counterValue: initialValue,
  })
}

render() {
  return(
    <div className="counter">
      Licznik:
      <span className="value">
        {this.state.counterValue}
      </span>
      <ButtonsPanel changeCounterValue={this.changeValue} resetCounterValue={this.resetCounter} />
    </div>
  );
}

export default Counter;
```

ReactJS – Thinking in React

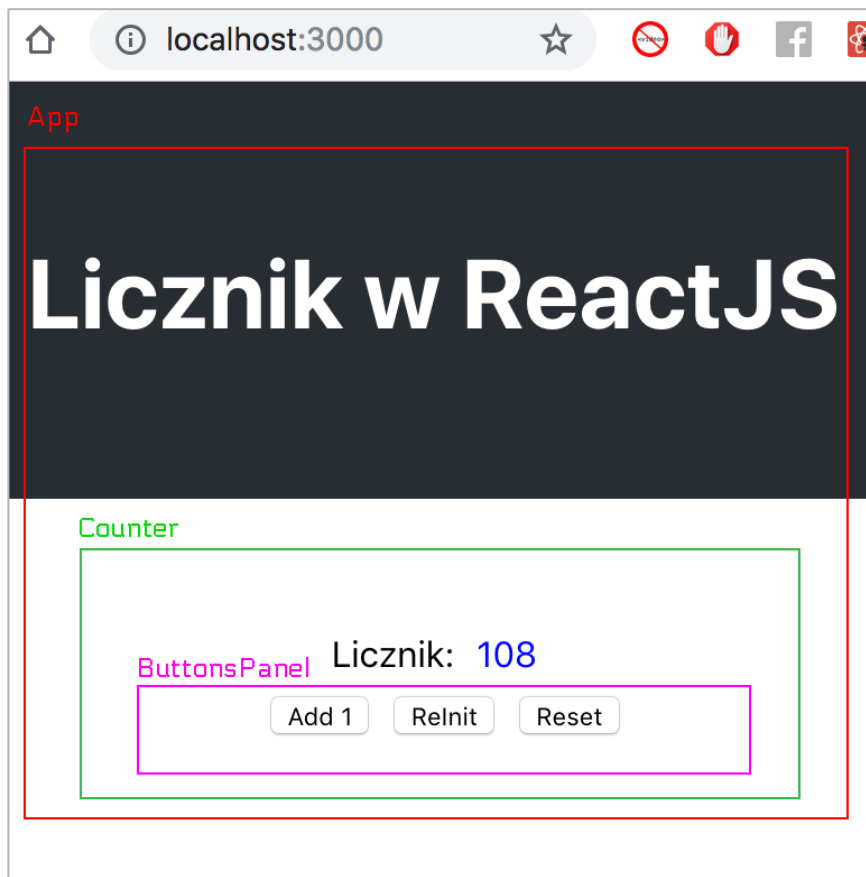
Thinking in React to sposób myślenia jak budować aplikacje w ReactJS. Ogólnie sprowadza się to do kilku kroków, którymi powinniśmy się kierować podczas tworzenia aplikacji za pomocą ReactJS ☺

1. Szkic aplikacji (tzw. Mockup) – dostajemy projekt od designer'a lub sami szkicujemy, co w aplikacji się będzie działo.
2. Podział Layout'u aplikacji na komponenty najlepiej wg. zasady SRP (Single Responsibility Principle) – czyli każdy komponent powinien robić tylko jedną rzecz.
3. Budowa w ReactJS statycznej wersji aplikacji – czyli stworzenie komponentów, ich wyglądu (css), wyświetlania danych ale bez interakcji z użytkownikiem. Nie używaj na tym etapie state, a staraj się tylko przekazywać dane od rodzica do dzieci poprzez props'y oraz potem je wyświetlać.
4. Identyfikacja jakie dane powinny być przechowywane w aplikacji – stan komponentu state. Kierujemy się zasadą DRY (Don't Repeat Yourself), czyli porcję danych przechowujemy i uaktualniamy tylko w jednym miejscu w aplikacji
5. Lokalizacja, w których komponentach powinny być przechowywane stany (state) – czyli który komponent powinien mieć obiekt state i go uaktualniać. Przeważnie komponent położony najwyżej w hierarchii komponentów powinien posiadać swój stan – state.
6. Komunikacja pomiędzy komponentami w górę – czyli od dzieci do rodzica. W komponencie nadrzędnym przekazujemy metodę (tzw. callback) do komponentu potomnego poprzez props'y ☺

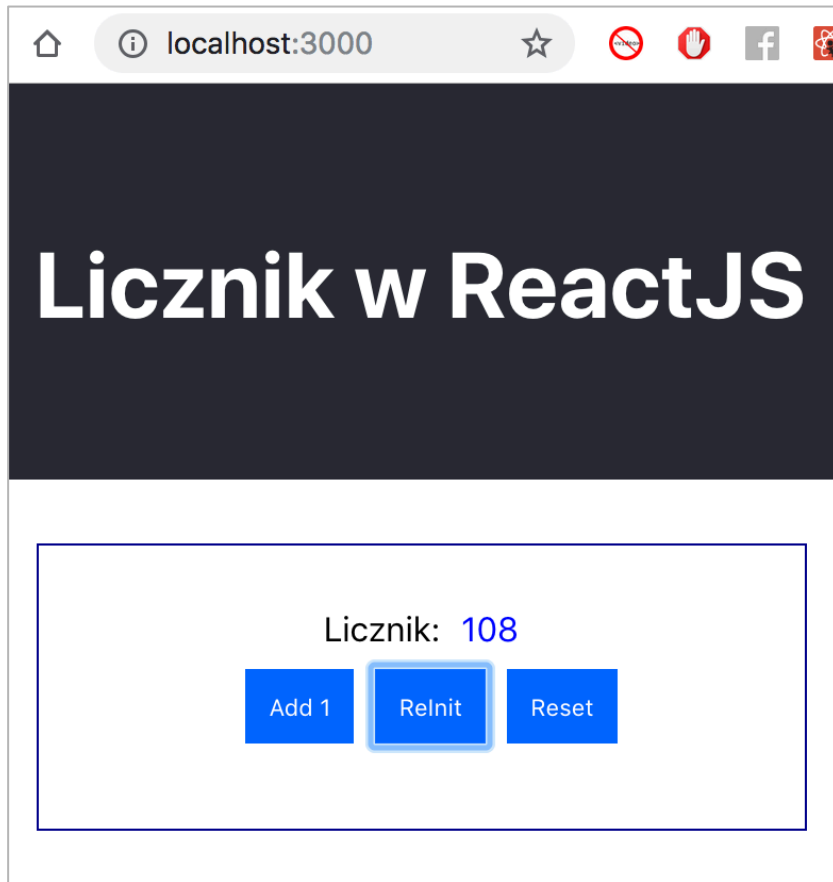
Polecamy przeczytanie artykułu z dokumentacji ReactJS pt. Thinking in React <https://reactjs.org/docs/thinking-in-react.html>. Bardzo dobrym tutorialiem do zagadnień związanych z ReactJS jest właśnie jego dokumentacja <https://reactjs.org/docs/getting-started.html>.

ReactJS – Thinking in React

Szkic Aplikacji Licznika z rozpisanymi komponentami



Gotowa działająca Aplikacja Licznika

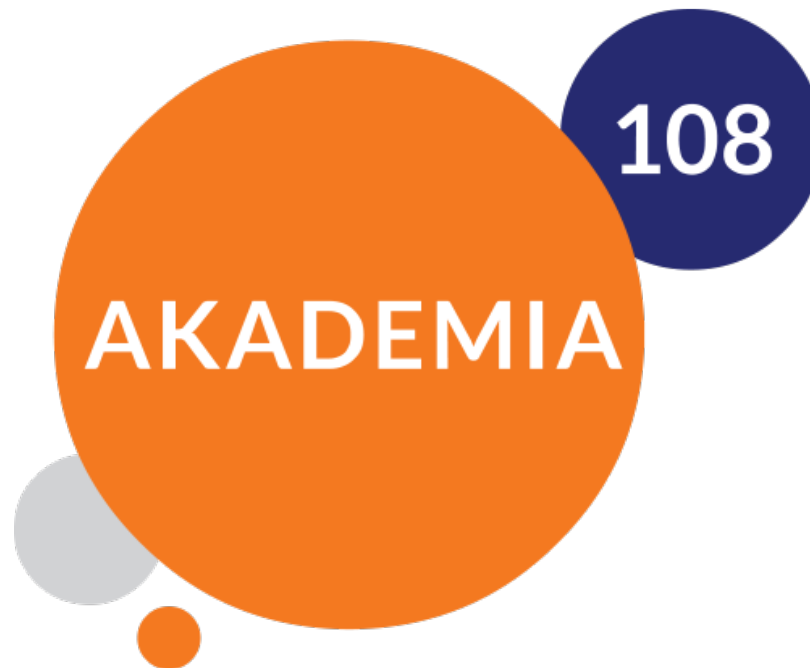


WARSZTATY – Lista użytkowników

Stwórz aplikację w `ReactJS` obsługującą moduł użytkowników.

Funkcjonalność aplikacji:

- dodawanie nowego użytkownika (wystarczy formularz, w którym jest pole do wprowadzania imienia)
- wyświetlanie listy użytkowników (np. lista ``)
- usuwanie użytkowników (po kliknięciu w użytkownika na liście, powinien on się usunąć)



Akademia 108

<https://akademia108.pl>