

# Konfiguracja projektu React i pierwsze szlify

Damian Banach

Mateusz Jagiełło

# Prolog - Instalacja node.js

Do pracy z React'em jest wymagany node.js. Można go pobrać ze strony:

<https://nodejs.org/en/>

Instalacja jest bardzo prosta, a w celu sprawdzenia czy node.js prawidłowo został zainstalowany należy wpisać w konsoli wiersza polecen polecenie:

„node -v”

```
C:\Users\Damian>node -v  
v16.14.0
```

# Krok 1 - Stworzenie nowego projektu za pomocą Create React App

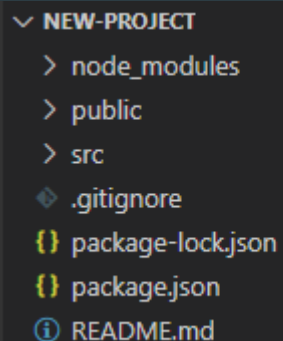
Wraz z zainstalowaniem node.js mamy do dyspozycji pakiet zarządzania aplikacją, a mianowicie - „npm”. Pakiet ten udostępnia nam narzędzie „npx”, które uruchamia pakiety wykonywalne. Do stworzenia pierwszego projektu wystarczy wpisać następującą komendę:

```
npx create-react-app nazwa-projektu
```

Po stworzeniu nowego projektu w oknie terminala otrzymamy komunikat o poprawnym stworzeniu nowego projektu i wskazówki o kilku przydatnych komendach.

# Krok 2 - React Scripts

Po stworzeniu nowego projektu możemy ujrzeć następującą strukturę:



```
▼ NEW-PROJECT
  > node_modules
  > public
  > src
  ◆ .gitignore
  {} package-lock.json
  {} package.json
  ⓘ README.md
```

**node\_modules** zawiera zewnętrzne biblioteki JavaScript

**public** to „korzeń” projektu, który zawiera podstawowe pliki html, json oraz pliki obrazów.

**src** tutaj jest zawarty kod React JavaScript i w tym pliku będziemy przechowywać owoce naszej pracy.

**.gitignore** jak sama nazwa mówi tu są informacje o ignorowanych plikach przez Gita.

**README.md** zawiera wiele informacji na temat Create React App. W tym też pliki w późniejszym czasie będziemy uwzględniać informacje o naszym projekcie.

# package-lock.json oraz package.json

Plik package-lock.json służy do sprawdzenia czy pakiety są w tej samej wersji. Innymi słowy gdy ktoś inny instaluje nasz projekt można się upewnić czy są te same zależności.

Plik package.json zawiera metadane dotyczące projektu, czyli tytuł, numer wersji i zależności. Są w nim zawarte również skrypty, których można użyć w projekcie.

# Build Script

Do skompilowania naszego kodu w użyteczny pakiet wystarczy użyć w terminalu komendy:

## `npm run build`

Po zastosowaniu tej komendy w strukturze naszego projektu pojawi się nowy folder o nazwie **build** w którym są zawarte pliki kompilacji. W pliku `.gitignore` można zauważyć, że folder **build** jest ignorowany przez git. To dlatego, że folder **build** to tylko zminimalizowana i zoptymalizowana wersja innych plików i nie ma potrzeby używania wersji kontrolnych i lepiej uruchomić polecenie build.

# Test Script

Do przeprowadzania testów na swoim projekcie można wykorzystać polecenie

## npm test

Komenda ta uruchomi testy na naszym projekcie. Dodatkowo w terminalu otrzymamy komunikat, że możemy poprzez dane klawisze uruchomić dane testy np.:

f - testy zakończone niepowodzeniem

o - testy związane ze zmienionymi plikami

p - filtrowanie według nazwy pliku

t - filtrowanie według nazwy testu

enter - uruchomienie testowego uruchomienia

Klawiszem q wychodzimy z trybu testowania.

# Eject Script

Eject Script odpowiada za skopiowanie naszych zależności oraz plików konfiguracyjnych do naszego projektu oraz daje nam pełną kontrolę nad kodem.

Dzięki temu poleceniu można zmienić narzędzenia do budowania i konfiguracji projektu.

Jest to też operacja jednokierunkowa, a więc po jej wykonaniu nie da rady wrócić do poprzednich konfiguracji.



## Krok 3 - Uruchomienie Serwera

W celu uruchomienia serwera aby móc oglądać zmiany, które poczynamy na naszym kodzie wystarczy w terminalu wpisać następująca komendę:

**npm start**

Po chwili odpali nam się przeglądarka, a w niej domyślny projekt, który utworzyliśmy.

# Krok 4 - Modyfikacja Strony Głównej

W celu modyfikacji strony głównej należy przejść do folderu **public**

Plik **manifest.json** to zestaw metadanych opisujący nasz projekt, a plik **robots.txt** zawiera informacje dla robotów indeksujących.

W celu edycji strony głównej należy otworzyć plik **index.html**. Plik ten jest krótki, a jest to spowodowane przez Reacta, który wstrzykuje całą strukturę HTML za pomocą JavaScript.

Na początek możemy zmienić tytuł na dowolnie inny i zapisać zmiany. Zmiany w przeglądarce zajdą automatycznie bez potrzeby odświeżania karty przeglądarki.

Następnie zmienimy w divie wartość id z „root” na „base”. Zmiana ta spowoduje, że nic nie zobaczymy, a jest to spowodowane tym, że React szuka elementu o id root do rozpoczęcia projektu.

# Krok 5 - Modyfikacja Aplikacji

W celu modyfikacji aplikacji należy przejść do katalogu **src**.

Do zmodyfikowania stylu naszej aplikacji należy udać się do pliku **App.css**.

W pliku index.js:

- ▶ Poprzez import React konwertujemy kod JSX na JavaScript
- ▶ Import ReactDOM odpowiada za łączenie z podstawowymi elementami z katalogu **public**.

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

Kod ten nakazuje znalezienie Reactowi elementu o id root i wstrzyknięciu tam kodu Reacta.

Import arkusza stylu index.css odpowiada za rozkaz Webpackowi dołączenia tego kodu do skończonego skompilowanego pakietu.

# Hello World!

Przechodząc do pliku App.js możemy edytować widniejący na naszej stronie główny paragraf

Edit `<code>src/App.js</code>` and save to reload.

Zmieniając tą część dokonamy naszej pierwszej zmiany w komponencie Reacta.

# Import plików

W pliku App.js dochodzi do następującego importu:

```
import logo from './logo.svg';
```

...

```
<img src={logo} className="App-logo" alt="logo" />
```

...

Za każdym razem gdy przekazany atrybut nie jest stringiem lub liczbą należy użyć nawiasów klamrowych, inaczej React potraktuje go jako JavaScript, a nie ciąg. W tym przypadku nie jest importowany obraz, a jest odwołanie do obrazu. Kiedy Webpack zbuduje projekt ustawi on odpowiednią ścieżkę do obrazu. Można to łatwo zauważyć w kodzie źródłowym w przeglądarce.

## Krok 6 - Kompilacja kodu

Kiedy dokonaliśmy wcześniej kosmetycznych zmian możemy skompilować kod w pakiet, który można byłoby wdrożyć na serwer za pomocą polecenia

```
npm run build
```

W folderze build pojawi się plik index.html, w którym znajdą się połączone pliki w sposób jak najbardziej zminimalizowany.

# Tworzenie elementów JSX

Przykład składni JSX:

```
const element = <h1>Witaj, świecie!</h1>;
```

Taką składnię nazywamy JSX i jest to rozszerzenie składni JavaScriptu o możliwość wstawiania znaczników. React przyjmuje ideę, że logika związana z prezentacją danych jest z natury rzeczy powiązana z innymi elementami logiki biznesowej UI: sposobami przetwarzania zdarzeń w aplikacji, tym, jak stan aplikacji zmienia się w czasie, jak również tym, jak dane są przygotowywane do wyświetlenia.

Zamiast sztucznie rozdzielać technologie, umiejscawiając znaczniki oraz logikę aplikacji w osobnych plikach, React wprowadza podział odpowiedzialności poprzez wprowadzenie luźno powiązanych jednostek, nazywanych “komponentami”, które zawierają zarówno znaczniki HTML, jak i związaną z nimi logikę.



# „Hello, World” i zwracanie elementu

```
function App() {  
  return <h1>siema</h1>  
}
```

Kiedy zwracamy JSX z funkcji, musimy zwrócić pojedynczy element. Ten element może mieć „dzieci”, ale musi być jeden element najwyższego poziomu.


```
function App() {  
  return (  
    <h1>siema</h1>  
    <p>nauka jsx</p>  
  );  
}
```

# Style w JSX

Jako iż JSX jest JavaScriptem, ma kilka ograniczeń. Jednym z ograniczeń jest to, że JavaScript ma zarezerwowane słowa kluczowe. Oznacza to, że nie można użyć pewnych słów w dowolnym kodzie JavaScript.

Jednym z zastrzeżonych słów jest `class`. React omija to zastrzeżone słowo zmieniając go nieznacznie. Zamiast dodawać atrybut `class`, dodajemy atrybut `className`.

```
function App() {  
  return (  
    <div className='container'>  
      <h1 id='siema'>siema</h1>  
      <p>nauka jsx</p>  
    </div>  
  );  
}
```

```
.container{  
  background-color:  yellow;  
  display: flex;  
  flex-direction: column;  
  align-items: center;  
}
```

# Zmienne i funkcje jako atrybuty

Zaletą JSX jest to, że chociaż wygląda jak HTML, ma moc JavaScripta. Można przypisywać zmienne i odwoływać się do nich w swoich atrybutach.

Wykorzystuje się do tego nawiasy { }.

```
function App() {  
  const przywitanie='siema';  
  return (  
    <div className='container'>  
      <h1 id={przywitanie}>siema</h1>  
      <p>nauka jsx</p>  
    </div>  
  );  
}
```

# Dodawanie zdarzeń do elementu-KOD

```
<ul>
  <li>
    <button onClick={event=>alert(event.target.id)}>
      <span id="emoji1" role="img">😊</span>
    </button>
  </li>
  <li>
    <button onClick={event=>alert(event.target.id)}>
      <span id="emoji2" role="img">😐</span>
    </button>
  </li>
  <li>
    <button onClick={event=>alert(event.target.id)}>
      <span id="emoji3" role="img">😓</span>
    </button>
  </li>
</ul>
```

```
button {
  font-size: 200px;
  border: 0;
  padding: 0;
  background: none;
  cursor: pointer;
}
ul { display: flex;
  padding: 0;
}
li {
  margin: 0 20px;
  list-style: none;
  padding: 0;
}
```

# Rodzaje zdarzeń-przykłady

## Common HTML Events

Here is a list of some common HTML events:

Event	Description
onchange	An HTML element has been changed
onclick	The user clicks an HTML element
onmouseover	The user moves the mouse over an HTML element
onmouseout	The user moves the mouse away from an HTML element
onkeydown	The user pushes a keyboard key
onload	The browser has finished loading the page

Zdarzenia w React piszemy metodą camelCase (onClick zamiast onclick).

# Redukowanie kodu i zwiększenie jego czytelności.

JSX nie ogranicza Cię do składni HTML. Daje ci również możliwość aby używać JavaScript bezpośrednio w znacznikach. Przerabiamy kod na tablicę, z której będziemy pobierać elementy. Tablica obiektów emoji:

```
const emojis = [  
  {  
    emoji: '😄',  
    name: "emoji1"  
  },  
  {  
    emoji: '😐',  
    name: "emoji2"  
  },  
  {  
    emoji: '😞',  
    name: "emoji3"  
  }  
]
```

# Tablica elementów-kod

```
▶ const emojis = [  
▶   {  
▶     emoji: '😊',  
▶     name: "emoji1"  
▶   },  
▶   {  
▶     emoji: '😐',  
▶     name: "emoji2"  
▶   },  
▶   {  
▶     emoji: '😞',  
▶     name: "emoji3"  
▶   }  
▶ ];
```

# Redukowanie kodu i zwiększenie jego czytelności.

Aby stworzyć komponenty React, musimy przekonwertować dane do JSX. Aby to zrobić musimy zmapować dane i zwrócić element JSX. Jest kilka rzeczy, o których musisz pamiętać podczas pisania kodu.

Najpierw grupa elementów musi być otoczona kontenerem np. `<div>`.

Po drugie, każdy element wymaga specjalnej właściwości o nazwie **key**. **Key** musi być unikalnym elementem danych, który React może wykorzystać do śledzenia elementów. Metoda `.map()` tworzy nową tablicę zawierającą wyniki wywołania funkcji dla każdego elementu wywołującej tablicy.

```
...  
const names = [  
  "Atul Gawande",  
  "Stan Sakai",  
  "Barry Lopez"  
];  
  
return(  
  <div>  
    {names.map(name => <div key={name}>{name}</div>)}  
  </div>  
)  
...
```



# Emoji-map kod

```
▶ <ul>
▶   {
▶     emojis.map(emoji => (
▶       <li key={emoji.name}>
▶         <button onClick={displayemo}>
▶           <span role="img" id={emoji.name}>{emoji.emoji}</span>
▶           </button>
▶         </li>
▶       ))
▶   }
▶ </ul>
```

# Warunkowe wyświetlanie elementów

Są chwile, kiedy będziesz potrzebować komponentu, aby pokazać informacje w niektórych przypadkach, a w innych nie. Na przykład możesz chcieć wyświetlać pewne informacje o koncie dla administratora, których nie chcesz pokazać normalnemu użytkownikowi. Jako że korzystamy z JavaScriptu możemy do tego wykorzystać operatory logiczne. Przykład:

```
let zalogowany=true;
```

```
{zalogowany && <button>Wyloguj się</button>}
```

Wartość zmiennej zalogowany będzie decydować o wyświetlaniu guzika na stronie.