

# React

---

Aleksandra Poręba   Grzegorz Podsiadło

20 maja 2020

Wydział Fizyki i Informatyki Stosowanej  
ul. Reymonta 19  
30-055 Kraków  
Polska

1. React
2. Wdrożenie React do aplikacji
3. Komponenty - JSX, właściwości, stan oraz cykl życia
4. React Router
5. Redux
6. Podsumowanie
7. Bibliografia

# React

---

**React** jest **biblioteką** utworzoną przez Facebook na potrzeby budowania interfejsów użytkownika (UI).

Cechy specyficzne dla React.js:

1. Wykorzystuje niezależne **komponenty**, które mogą posiadać własny **stan**.
2. Wykorzystuje **Virtual DOM**, który przyspiesza renderowanie zmian interfejsu.
3. Możliwość pisania interfejsu w **JSX** - składniowo podobny do XML z możliwością wykorzystania JS
4. **Ułatwia pracę w grupie**, gdyż każdy programista może pracować nad swoimi komponentami.
5. Skupiamy się na tworzeniu serwisów typu **SPA**.

# Wdrożenie React do aplikacji

---

# Dodanie React do aplikacji

Skoncentrujemy się na dodaniu Reacta do naszej aplikacji na dwa sposoby:

1. Przy pomocy skryptów hostowanych w ramach **CDN**<sup>1</sup>
2. Przy pomocy managera pakietów - na przykład **npm**

Dodanie React przy pomocy CDN jest bardzo proste, potrzebne będzie dodanie do strony w minimalnej wersji dwóch skryptów:

```
1 <!-- Załadowanie Reacta przy pomocy CDN. -->
2 <!-- Wersje deweloperskie ".../development.js" oraz produkcyjne ".../production.min.js". -->
3 <script src="https://unpkg.com/react@16/umd/react.development.js" crossorigin></script>
4 <script src="https://unpkg.com/react-dom@16/umd/react-dom.development.js" crossorigin></script>
```

---

<sup>1</sup>ang. content delivery network

# Dodanie React do aplikacji przy pomocy CDN

Konieczne jest również utworzenie "kontenera", w którym będą znajdowały się elementy tworzone przy pomocy Reacta.

- Wystarczy utworzenie prostego `div` z unikalnym `id`, do którego będziemy się odnosić.
- Takich kontenerów tworzyć możemy dowolną ilość, zwykle jednak ograniczymy się do jednego.
- Zawartość kontenera nie ma znaczenia. React i tak zastąpi jego całą zawartość w poddrzewie DOM.

```
1 <body>
2   <div id="react_root"></div>
3   <script src="HelloWorld.js"></script>
4 </body>
```

# Dodanie React do aplikacji przy pomocy npm

Kolejnym sposobem - bardziej polecanym jest wykorzystanie menedżera pakietów, w naszym przypadku `npm`, można pobrać go [pod tym adresem](#).

- Twórcy Reacta przygotowali wygodne środowisko, które można utworzyć przy pomocy polecenia `create-react-app`.

```
npx create-react-app <nazwa_aplikacji>  
cd <nazwa_aplikacji>  
npm start
```

- Utworzony projekt zawiera szkielet aplikacji oraz wszystkie potrzebne komponenty, w tym kompilator Babel.
- Po uruchomieniu live server przy pomocy `npm start` można na żywo obserwować dokonywane zmiany po zapisaniu kodu, bez konieczności odświeżania strony, pod adresem `localhost:3000`.



## Komponenty - JSX, właściwości, stan oraz cykl życia

---

# Elementy Reacta

**Element** jest podstawowym blokiem, z którego zbudowane są aplikacje napisane w React. Przykład takiego podstawowego elementu znajdują się poniżej:

```
1 const element = <h1>Hello, world</h1>;
```

Każdy element opisuje to, co chcielibyśmy widzieć na ekranie. Elementy Reacta, w przeciwieństwie do elementów drzewa DOM są zwykłymi obiektami, które są proste do utworzenia.

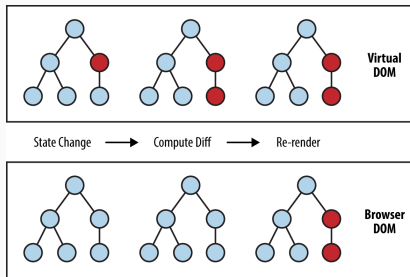
React używa do renderowania elementów używa konceptu wirtualnego DOM.

Najprostsze wykorzystanie Reacta, wyrenderowanie utworzonego elementu w głównym węźle Reacta.

```
1 ReactDOM.render(  
2   element,  
3   document.getElementById('root')  
4 );
```

# Wirtualny DOM

React zarządza swoim własną wirtualną reprezentacją drzewa DOM, za każdym razem gdy element zarządzany przez Reacta się zmienia - aktualizowane jest wirtualne drzewo, które porównywane jest z poprzednim i aktualizowane są tylko potrzebne elementy. Dzięki temu wirtualna reprezentacja działa szybciej od standardowej.



**Rysunek 1:** Działanie virtual DOM.

# JSX?

- JSX jest rozszerzeniem do języka JavaScript, szeroko wykorzystywanym w React, pozwala na wykorzystywanie JS wewnątrz napisanych komponentów.
- Wyglądem przypomina HTML oraz XML, jednak istnieją drobne różnice.
- JSX jest używany do tworzenia elementów Reacta, jednak nie jest do tego niezbędny. Tak naprawdę jest on kompilowany przez Babel do wywołań zwykłych funkcji JS tworzących elementy.

```
1 const element = <div><h1>Hello,</h1> <p> world!</p></div>
```

```
1 const element = React.createElement("div", null, React.createElement("h1", null, "Hello,"), " ",  
  React.createElement("p", null, " world!"));
```

Wyprodukowany przez JSX kod można sprawdzić online, przy pomocy [Babeljs](#).

W JSX można umieścić dowolne wyrażenie JS wewnątrz nawiasów {}.

```
1 const name = 'Grzegorz Podsiadlo';
2 const element = <h1>Hello, {name}</h1>;
3
4 ReactDOM.render(
5   element,
6   document.getElementById('root')
7 );
```

Ponieważ po skompilowaniu JSX zamienia się na zwykłe wywołania funkcji, możliwe jest wprowadzenie logiki do naszego UI.

```
1 function getGreeting(user) {
2   if (user) {
3     return <h1>Hello, {formatName(user)}!</h1>;
4   }
5   return <h1>Hello, Stranger.</h1>;
6 }
```

Atrybut klasy z HTML otrzymujemy poprzez `className`, nie `class`.

JSX jest odporny na ataki typu Injection, gdyż samoistnie sprawdza uciekanie znaków specjalnych wewnątrz otrzymywanych napisów.

# JSX w przypadku wykorzystania CDN

Wykorzystanie JSX w przypadku rozwoju aplikacji z pakietami CDN jest możliwe, jednak **nie jest zalecane, gdyż spowalnia to wykonanie kodu.**

Należy dodać pakiet Babela przy pomocy CDN:

```
1 <script src="https://unpkg.com/babel-standalone@6/babel.min.js"></script>
```

Oraz **tag zmieniający typ skryptu** , w którym JSX będzie wykorzystywany.

```
1 <script type="text/babel" > ... </script>
```

Kod napisany przy pomocy JSX będzie tak naprawdę tłumaczony na odpowiednie wywołania funkcji w JS.

# Renderowanie elementów

Na potrzeby rysowania elementów konieczne jest utworzenie węzła drzewa DOM, które będzie korzeniem dla wirtualnego DOM Reacta.

```
1 <div id="root"></div>
```

By wyrenderować element w węźle głównym należy użyć funkcji `ReactDOM.render`

```
1 const element = <h1>Hello, world</h1>;  
2 ReactDOM.render(element, document.getElementById('root'));
```

Wszystkie elementy Reacta są **immutable**, by zaktualizować UI trzeba utworzyć nowy element i przekazać go do funkcji renderującej.

# Komponenty

Komponenty są głównymi klockami budującymi Reacta, każdy komponent jest zbiorem elementów oraz logiki. Pozwala na rozbicie UI w niezależne od siebie elementy które są odpowiedzialne za własny stan.

Koncepcyjnie celem komponentów jest przyjmowanie parametrów nazywanych właściwościami (ang. properties) i zwrócenie widoku składającego się z elementów Reacta.

Komponent jest klasą, która dziedziczy po `React.Component` oraz posiada metodę `render`, która mówi jaka jest jego reprezentacja w postaci elementów.

```
1
2 class Welcome extends React.Component {
3   render() {
4     return <h1>Hello, {this.props.name}</h1>;
5   }
6 }
```

Dane wejściowe przekazane do komponentu można uzyskać poprzez `this.props`



# Komponent w postaci funkcji

- W React proste komponenty można przedstawić w postaci funkcji - taki komponent jest tym samym co zwykły komponent z użyciem klasy posiadający jedynie funkcję render.
- Komponent ten nie posiada własnego stanu, może przyjmować properties i musi zwracać to co ma zostać wyrenderowane.

```
1 function Welcome(props) {  
2   return <h1>Hello, {props.name}</h1>;  
3 }
```

# Renderowanie komponentu

Komponentów można używać tak, jak elementów, nazwa komponentu jest w tym przypadku nazwą węzła:

```
1 const element = <Welcome name="Grzegorz" />;
```

React przekazuje atrybuty elementu jako **properties** do komponentu.

Komponenty mogą używać innych komponentów:

```
1 function Welcome(props) {  
2   return <h1>Hello, {props.name}</h1>;  
3 }  
4  
5 function App() {  
6   return (  
7     <div>  
8       <Welcome name="Grzegorz" />  
9       <Welcome name="Aleksandra" />  
10    </div>  
11  );  
12 }  
13  
14 ReactDOM.render(  
15   <App />,  
16   document.getElementById('root')  
17 );
```

- **Properties każdego elementu są read-only**, czasami jednak potrzebujemy wprowadzić dodatkową logikę.
- Każdy komponent może mieć stan, który jest obiektem JS determinującym jak renderowany jest komponent oraz jak się zachowuje.
- Można tworzyć stan całej aplikacji ("application level") używając context API.

```
1 var state = {  
2   title: 'xxx',  
3   body: 'xxx',  
4   isFeatured: true  
5 };
```

Dostęp do stanu poprzez `this.state`.

```
1 render() {  
2   return (  
3     <div>  
4       <h1>Hello, world!</h1>  
5       <h2>It is {this.state.date.toLocaleTimeString()}.</h2>  
6     </div>  
7   );  
8 }
```

Ustawienie stanu przez metodę `setState()`, która jednocześnie informuje, że należy przerenderować komponent.

```
1 this.setState({comment: 'Hello'})
```

Komponenty Reacta posiadają specjalne metody, wykonywane w określonych momentach, co przydatne jest często do alokowania lub dealokowania wykorzystywanych zasobów.

Przykład dwóch takich metod:

- `componentDidMount()` - wykonuje się gdy render skończył renderować drzewo DOM po raz pierwszy.
- `componentWillUnmount()` - wykonuje się gdy DOM wyprodukowany przez render jest usuwany.

# React Router

---

**React Router** to narzędzie do obsługi dynamicznego routingu w aplikacjach ReactJS.

Składa się z komponentów umieszczonych w pakietach `react-router` oraz `react-router-dom`, za pomocą których definiujemy nawigację aplikacji.

Umożliwia on:

- dostęp do widoków za pomocą adresu URL,
- użycie API history,
- dynamiczne ładowanie komponentów bez odświeżania strony.

- **Routing statyczny** polega na początkowym stworzeniu puli adresów, które następnie wywołujemy według potrzeb.
- Każdorazowo odpytywany jest serwer, który wyszukuje odpowiednie dopasowanie.
- W **routingu dynamicznym** ścieżki aktualizowane są podczas renderowania aplikacji.
- Dzieje się to u klienta zgodnie z zasadą działania SPA, bez potrzeby wysyłania zapytań do serwera.



# Komponenty - BrowserRouter, Route

- **Browser router** - podstawowy komponent nawigacji, który za pomocą zdarzeń API history HTML5, takich jak `pushState`, `replaceState`, czy `popState` synchronizuje interfejs z aktualnym adresem URL.
- **Route** - zawiera wzorzec ścieżki oraz odpowiadający mu komponent.
- Składnia: `<Route path='/adres' > <Komponent /> </ Route>`
- Dodatkowo można dodać takie atrybuty jak `exact` - dopasowanie musi być dokładne lub `sensitive` - powoduje, że wzorzec sprawdza małe i wielkie litery.

Wzorzec	Ścieżka	Dopasowanie
<code>\about</code>	<code>\about</code>	Tak
<code>\about</code>	<code>\about\contact</code>	Nie

**Tablica 1:** Działanie atrybutu `exact`

# Komponenty - Link, NavLink

- **Link** - komponent, który umożliwia nawigację w aplikacji. Renderuje tag `<a>` z podanym adresem.
- **NavLink** - jest wersją komponentu Link, która dodaje dodatkowy styl, gdy element zostanie wyrenderowany.
- Składnia: `<Link to='/adres'> Opis </ Link>`

```
1 import { NavLink } from 'react-router-dom'
2
3 const Links = () => {
4   return (
5     <div>
6       <NavLink to='/'>Home</ NavLink>
7       <NavLink to='/about'>About</ NavLink>
8     </div>
9   )
10 }
```

# Komponenty - Switch

- **Switch** - renderuje pierwszy napotkany komponent, którego wzorec pasuje do adresu. W przeciwnym przypadku załadowane zostaną wszystkie dopasowane komponenty.
- Kolejność ścieżek również ma znaczenie - ścieżki, które są prefiksami dla innych, powinny znajdować się na końcu.
- Switch pozwala też na wyspecyfikowanie domyślnego komponentu, gdy nie wystąpiło żadne dopasowanie.

```
1 import { Route, Switch } from 'react-router-dom'
2
3 const Routes = () => {
4   return (
5     <Switch>
6       <Route path="/about"><About /></Route>
7       <Route path="/"><Home/></Route>
8     </Switch>
9   )
10 }
```

Podsumowując wszystkie te informacje jesteśmy w stanie stworzyć dynamiczne routowanie w aplikacji React.

```
1 import { BrowserRouter, Switch, Route } from 'react-router-dom'
2
3 const App = () => {
4   return (
5     <BrowserRouter>
6       <div className="App">
7         <Switch>
8           <Route path="/signin" component={SignIn} />
9           <Route path="/signup" component={SignUp} />
10          <Route path="/" component={Dashboard} />
11        </Switch>
12      </div>
13    </BrowserRouter>
14  );
15 }
```

Aby zainstalować bibliotekę 'react-router-dom', która eksportuje wszystkie komponenty i funkcje z react-router oraz doda komponenty *DOM aware* można użyć managera pakietów npm:

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

managera yarn:

```
1 yarn add react-router-dom
```

lub linku CDN (*Content Delivery Network*), na przykład:

```
1 <script src="https://cdnjs.cloudflare.com/ajax/libs/react-router-dom/5.1.2/react-router-dom.min.js" integrity="sha256-Ga/RV3YJI+cd1/ML8yitEoluFHUJZ7HTH90az8f0FZU=" crossorigin="anonymous"></script>
```

# Redux

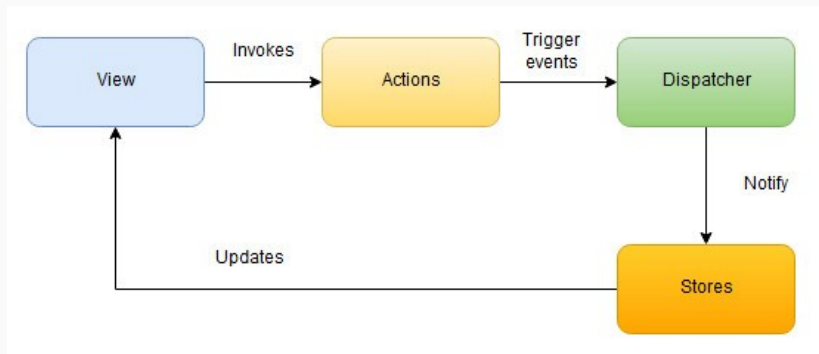
---

Architektura **Flux** powstała jako alternatywa dla wzorca MVC. Oparta jest o jednokierunkowy przepływ danych, wszystkie dane mają ten sam cykl życia.

Składa się ona z 4 elementów:

- **Store** - magazyn w którym przechowywany jest stan aplikacji,
- **View** - widok, czyli interfejs użytkownika, który generowany jest na podstawie informacji z magazynu,
- **Action** - akcja, informacja o zdarzeniu w systemie,
- **Dispatcher** - wysłanie informacji o zajściu akcji.

Jedną z implementacji architektury Flux jest **Redux**.



**Rysunek 2:** Działanie architektury Flux



Tak jak zostało już wspomniane Redux jest implementacją architektury Flux. Opisują go 3 podstawowe zasady:

- **Single source of truth**

Stan aplikacji trzymany jest w globalnym magazynie.

- **State is read-only**

Jedynym sposobem aby zmienić stan jest *akcja* - widoki nie mogą zmieniać stanu!

- **Changes are made with pure functions**

Zmiany wyzywalane przez akcje dokonują się za pomocą *reducers*, które są funkcjami obliczającymi nowy stan na podstawie akcji i stanu aktualnego.

Redux nie implementuje *dispatcher* z architektury Flux, jest on częścią magazynu.

W Reduxie *reducers* nigdy nie zmieniają danych - tworzą nowe obiekty, które są zwracane.

**Akcją** nazywana jest informacja o jakimś zdarzeniu. Przekazywana jest ona do magazynu.

```
1 const SET_NAME = 'SET_NAME'
```

Akcje możemy wywołać za pomocą funkcji `dispatch`.

```
1 dispatch({type: 'SET_NAME', name})
```

# Kreator akcji

Gdy po wywołaniu akcji chcemy wykonać dodatkowe operacje można użyć **kreatora akcji**.

```
1 function setName(name) {  
2   console.log(name)  
3   return {  
4     type: SET_NAME,  
5     name  
6   }  
7 }
```

Kreatora można wywołać bezpośrednio:

```
1 dispatch(setName(name))
```

albo za pomocą *bound action creator*.

```
1 const boundSetName = name => dispatch(setName(name))  
2 boundSetName(name)
```

**Reducers** zawierają reakcje na akcje. Na podstawie poprzedniego stanu i akcji generują nowy stan.

```
1 const userReducer = (state = initialState, action) => {  
2   if (action.type === SET_NAME) {  
3     return {  
4       ...state,  
5       name: action.name  
6     }  
7   }  
8  
9   return state  
10 }
```

Aby zadbać o zachowanie poprzedniego stanu można skopiować obiekt za pomocą `Object.assign()` albo użyć operatora `...` (*spread operator*).

# Tworzenie magazynu

Aby stworzyć magazyn w aplikacji React potrzebne będą funkcje oraz komponenty z bibliotek `redux` i `react-redux`.

Magazyn stworzony zostanie za pomocą funkcji `createStore`.

```
1 import { createStore } from 'redux'
2 import { Provider } from 'react-redux'
3 import rootReducer from '../rootReducer'
4
5 const store = createStore(rootReducer);
6
7 ReactDOM.render(
8   <Provider store={store}>
9     <App />
10   </Provider>,
11   document.getElementById('root')
12 );
```

Funkcja ta przyjmuje jako argument reducer, który będzie interreagował z danym magazynem.

# Tworzenie Reducer

Magazyn ma swój stan początkowy, który jest domyślnym argumentem reducera, na przykład:

```
1  const initialState = {
2      name: '',
3      friends: [],
4  }
5
6  const rootReducer = (state = initialState, action) => {
7      return state
8  }
9
10 export default rootReducer;
```

Gdy chcemy zrobić więcej niż jeden reducer powinniśmy je połączyć w jeden za pomocą funkcji `combineReducers`.

```
1  import {combineReducers} from 'redux'
2
3  const rootReducer = combineReducers ({
4      user: userReducer,
5      event: eventReducer
6  })
7
8  export default rootReducer;
```

# Łączenie komponentów z magazynem

Łączenie komponentów z magazynem odbywa się za pomocą biblioteki `react-redux`. Udostępnia ona funkcję `connect`, która jako argument przyjmuje komponent, a następnie opakowuje go w **Higher Order Component**. Tak opakowany komponent ma możliwość połączenia z magazynem, który wcześniej przekazywany był do komponentu 'Provider'.

```
1 import {connect} from 'redux-redux'
2
3 class Home extends Component {
4     render(){ return() }
5 }
6
7 export default connect()(Home);
```

Aby przekazać stan z magazynu do komponentu potrzebna jest jeszcze funkcja mapująca stan na atrybuty. Programista może wybrać które elementy mają być mapowane. Jest ona argumentem funkcji `connect`.

```
1 const mapStateToProps = (state) => {  
2     return {  
3         userName: state.name,  
4     }  
5 }  
6  
7 export default connect(mapStateToProps)(Home);
```

Po takich zmianach w komponencie można korzystać z `props.userName`.



# Wywoływanie akcji

Do zmieniania stanu magazynu należy użyć akcji. Aby wywołać je z komponentu należy stworzyć funkcję, która będzie mapowała akcje do atrybutów.

```
1 const mapDispatchToProps = (dispatch) => {  
2   return {  
3     setName: (name) => { dispatch({type: SET_NAME, name: name}) }  
4   }  
5 }  
6  
7 export default connect(mapStateToProps, mapDispatchToProps)(Home);
```

Teraz, gdy zostanie wywołana funkcja `props.setName()` wykona się podana akcja `SET_NAME`.

Aby dodać Reduxa do własnej aplikacji React potrzebne są dwie biblioteki `redux` oraz `react-redux`.

```
1 npm install redux  
2 npm install react-redux
```

# Podsumowanie

---

**Pytania?**

# Bibliografia

---



Flux documentation.

<https://facebook.github.io/flux/>.

[Online].



React a javascript library for building user interfaces.

<https://reactjs.org>.

[Online].



React virtual dom explained in simple english.

<https://programmingwithmosh.com/react/react-virtual-dom-explained/>.

[Online].



Redux documentation.

<https://redux.js.org/>.

[Online].