

MATEUSZ JABŁOŃSKI, 4-6.04.2022

TWORZENIE APLIKACJI ZUŻYCIEM REACT

MATEUSZ JABŁOŃSKI

- programowaniem zajmuję się już od 2011 roku
- pracuję na stanowisku senior fullstack developer
- zajmuję się głównie: javascript (React i Angular) oraz java

USTALENIA:

- Cel i plan szkolenia
- Oczekiwania
- Pytania oraz dyskusje
- Elastyczność zagadnień



CO NAS CZEKA?

DZIEŃ 1.

- NPM oraz Webpack
- ECMAScript 2015+
- Typescript
- Idea Virtual DOM
- Podstawy React

DZIEŃ 2.

- CSS w React
- Praca z formularzami
- Routing
- Dobre praktyki
- Komunikacja z BE

DZIEŃ 3.

- Redux
- MobX
- Testowanie aplikacji

REPOZYTORIUM

github.com/matwjablonski/deloitte-react



X minut

ZARZĄDZANIE PAKIETAMI





NPM?

- Newton's Principia Mathematica
- Nuclear Powered Mushroom
- Neurosis Prevention Mechanism
- Neurotic Pink Mongooses
- Naughty Push Message
- Nectar of the Programming Masses
- Node Package Manager

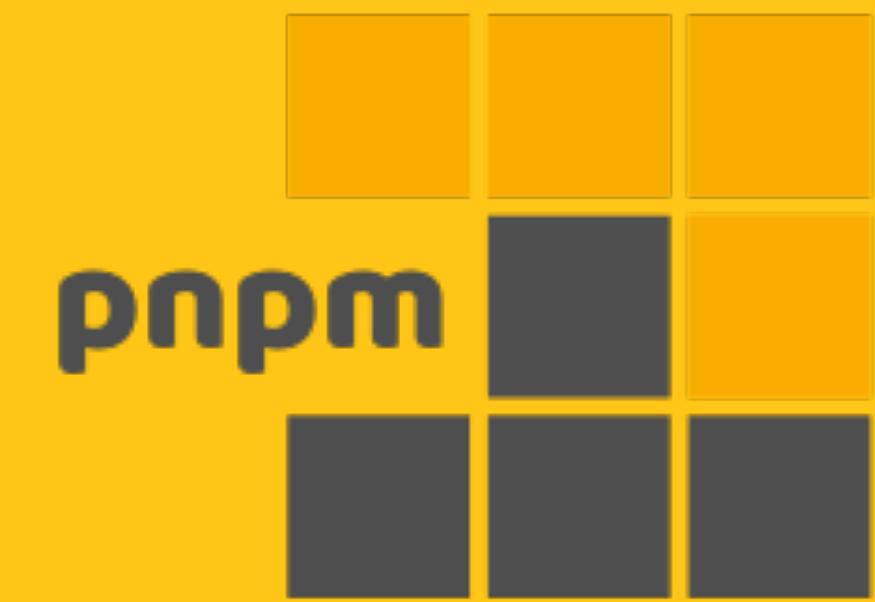


NPM?

- system zarządzania zależnościami
- zależności opisywane z dokładnością do wersji w pliku `package.json`
- `npm install` - instaluje pakiety, których nie ma w projekcie
- `npm update` - sprawdza czy istnieją nowe wersje pakietów i je instaluje
- `npm install nazwa-pakietu --save-dev` - instaluje nowy pakiet, dodaje go za listy zależności zgodnie w `package.json` z flagą
- `npx nazwa-pakietu` - uruchamia pakiet, nawet jeśli nie został wcześniej zainstalowany
- `package-lock.json` - wygenerowany plik podczas instalacji, zawiera informacje o wszystkich zainstalowanych zależnościach (`npm ci`)



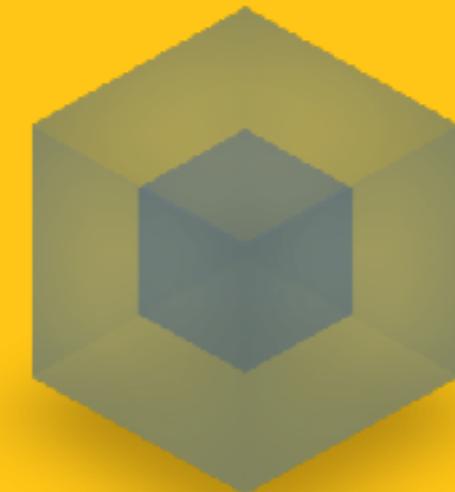
yarn



CZY MAMY INNE MOŻLIWOŚCI?

The background of the image is filled with a variety of colorful cakes and pastries, including a yellow cake with kiwi slices, a pink cake with white stripes, a chocolate cake with cherries, and a cake with a large dollop of white cream and a dark chocolate heart. The pastries are arranged in a dense, overlapping pattern.

WEBPACK



webpack
MODULE BUNDLER

WEBPACK?

- **module bundler**
- **obsługuje wiele formatów modułów (ES2015, AMD, CommonJS (np. Node.js))**
- **traktuje wszystko jak moduły (np. scss, html, grafiki)**
- **bardzo dobrze współpracuje w popularnymi task runnerami (Grunt, Gulp)**
- **standard w środowisku React**
- **React Hot Loader / React Fast Refresh (React Native)**



webpack
MODULE BUNDLER

```
1 module.exports = {
2   ...
3     entry: [
4       './js/index.js'
5     ], output: {
6       path: __dirname + '/static/',
7       filename: 'bundle.js'
8     },
9     plugins: [],
10    module: {
11      rules: [
12        {
13          test: /\.js$/,
14          use: ['babel-loader'],
15          exclude: /node_modules/
16        }
17      ]
18    },
19    devtool: 'source-map'
20  };
21
```

BABEL

Konfigurację dla transpilatora babel możemy zdefiniować raz tworząc w projekcie plik .babelrc

- presets - określają reguły transformacji kodu
- plugins - pozwalają rozszerzać mechanizmy babel

```
B .babelrc > ...
1 {
2   "presets": ["react", "es2015", "stage-0", "react-hmre"],
3   "plugins": ["react-hot-loader/babel", "transform-class-properties"]
4 }
```

A chocolate cake with pistachios and chocolate shavings on a silver platter.

ECMASCRIPT 2015+



- ECMA International - organizacja odpowiedzialna za standaryzację takich języków jak ECMAScript, C#, Eiffel
- Javascript jako implementacja języka ECMAScript (inne implementacje to JScript, ActionScript)

JS

ES2015

- **moduły (import / export)**
- **klasy (lukier składniowy)**
- **let, const**
- **forEach, map, reduce, filter, some, any**
- **arrow functions**
- **babeljs.io - kompilator JavaScript**

ARROW FUNCTIONS

```
1 // Domyślnie – zwraca wyrażenie  
2  
3 const odds = myArr.map(v => v + 1);  
4 const nums = myArr.map((v, i) => v + i);  
5  
6 const pairs = myArr.map(v => ({  
7   even: v,  
8   odd: v + 1,  
9 }));  
10  
11 // Deklaracje umieszczamy w klamrach  
12  
13 nums.filter(v => {  
14   if (v % 5 === 0) {  
15     return true;  
16   }  
17  
18   return false;  
19 });  
20
```

```
21 // Leksykalne this  
22  
23 const bob = {  
24   _name: 'Bob',  
25   _friends: [],  
26   getFriends() {  
27     return this._friends;  
28   }  
29 }  
30  
31 // Zmiana kontekstu wywołania  
32  
33 function a() {  
34   return this;  
35 }  
36  
37 element.addEventListener('click', a); // this === element  
38  
39  
40 const a = () => {};  
41 element.addEventListener('click', a); // this === window  
42
```

JS

DEFAULT EXPORT / NAMED EXPORT

```
1 // Named export
2
3 export const someExampleFunction = () => {};
4 import { someExampleFunction } from './';
5
6 // Default export
7
8 const someExampleFunction = () => {};
9 export default someExampleFunction;
10
11 import anyNameWeWant from './';
12 import { default as anyNameWeWant } from './';
13 .
```

DESTRUKTURYZACJA

```
1 const [a, , c] = [1, 2, 3];
2
3 const {
4   name,
5   age,
6   getName,
7   skills: { programming, languages, other }
8 } = person;
9
10 function showName({ name }) {
11   return name;
12 }
13 showName(person); // for person === { name: 'Bob' } it returns „Bob”
14
```

DEFAULT, SPREAD, REST

JS

```
1  function addTwo(a, b = 10) {  
2    return a + b;  
3  }  
4  addTwo(2); // 12  
5  
6  
7  function f(a, ...rest) {  
8    return a + rest.length;  
9  }  
10 f(2, "abc", true); // 4  
11  
12  
13 function g(x, y, z) {  
14   return x + y + z;  
15 }  
16 g(...[1, 2, 3]); // 6  
17
```

JS

OBIETNICE (PROMISE)

```
1
2  function timeout(duration = 0) {
3      return new Promise((resolve, reject) => {
4          setTimeout(resolve, duration);
5      })
6  }
7
8  const p = timeout(1000)
9      .then(() => {
10         return timeout(2000);
11     })
12     .then(() => {
13         throw new Error('hmm');
14     })
15     .catch(err => {
16         return Promise.all([timeout(100), timeout(200)]);
17     });
18
```

DYNAMICZNY LITERAŁ

```
1 const obj = {  
2     __proto__: theProtoObj,  
3  
4     // === `handler: handler`,  
5     handler,  
6  
7     // === `toString: function toString() {}`  
8     toString() {  
9         // Super calls  
10        return 'd ' + super.toString();  
11    },  
12  
13    // Dynamiczne nazwy właściwości  
14    [ 'prop_' +(() => 42)(): 42  
15  };  
16
```

ROZSZERZONY OBIEKT ARRAY

```
1 // konwertuje obiekt tablicopodobny na prawdziwą tablicę
2
3 Array.from(document.querySelectorAll('*'));
4
5 // tworzy nową tablicę – podobnie do new Array(), ale ma inne zachowanie dla jednego parametru
6
7 Array.of(1, 2, 3); // [1, 2, 3]
8 Array.of(2); // [2]
9
10 new Array(1, 2, 3); // [1, 2, 3]
11 new Array(2); // [ , ]
12
13 [0, 0, 0].fill(7 ,1); // [0, 7, 7]
14 [1, 2, 3].find(x => x === 3); // 3
15 [1, 2, 3].find(x => x === 2); // 1
16 [1, 2, 3, 4, 5].copyWithin(3, 0); // [1, 2, 3, 1, 2]
17 ["a", "b", "c"].entries(); // iterator [0, "a"], [1, "b"], [2, "c"]
18 ["a", "b", "c"].keys(); // iterator 0, 1, 2
19 ["a", "b", "c"].values(); // iterator "a", "b", "c"
20
```

ZADANIE

task/ecmascript



30 minut



TYPESCRIPT



- **stworzony i rozwijany przez Microsoft**
- **nadzbiór języka Javascript**
- **dodaje silne typowanie**
- **usprawnia proces developmentu**
- **dodane typy nie są przenoszone do kodu produkcyjnego**



- dodatkowa para oczu, która patrzy razem z nami na kod i szuka błędów
- świetne podpowiadanie składni oraz współpraca narzędzi na znacznie wyższym poziomie niż w czystym JS
- refaktoryzacja to nie problem - o spójność kodu dba kompilator
- gwarancja zawartości stałych i zmiennych - patrząc w typy możemy wywnioskować co znajduje się w danym miejscu (bez uruchamiania aplikacji)
- typy w TS to jednocześnie zawsze aktualna dokumentacja kodu
- TS to również sposób na zapisywanie ustaleń i kontraktów pomiędzy komponentami, zewnętrznymi serwisami



- nie rozwiązuje problemów JSa
- TS sprawdza typy tylko w czasie kompilacji
- bywa niechlujny (w szczególności w zależnościach)



TYPY

- **boolean**
- **string**
- **number**
- **array**
 - *let array: number[];*
- **object**
- **tuple**
 - *let tuple: [number, boolean, string];*
- **any**
- **unknown**
- **never**
- **void**
- **enum**



INFERENCJA TYPÓW

```
const title: string = "To jest tytuł";  
const title = "To jest tytuł";
```

TS domyśli się jaki jest typ dla zmiennej `title`. Jeśli nie przypiszemy typu i nie zostanie on rozpoznany, wówczas TS przypisze do zmiennej typ `any`.

TYPOWANIE FUNKCJI

TS

```
1 type generateTitleType = (text: string) => string;
2
3 const generateTitleA: (text: string) => string = (text) => "To jest tytuł!" + text;
4 const generateTitleB: generateTitleType = (text) => "To jest tytuł!" + text;
5
6
```

KLASY

```
1  class Car {  
2      private name: string;  
3  
4      constructor(name: string) {  
5          this.name = name;  
6      }  
7  
8      public getName() {  
9          return this.name;  
10     }  
11 }  
12  
13 const car = new Car("VW");  
14 car.name; // Error  
15 car.getName(); // VW  
16
```

TS

INTERFEJSY

TS

```
1  enum Doors {
2      THREE = 3,
3      FIVE = 5,
4  }
5
6  interface Car {
7      name: string;
8      enginePower: number;
9      doors: Doors;
10     color?: string;
11 }
12
13 let car: Car;
14
15
```

ZADANIE

task/typescript



20 minut

PRZERWA



**45 minut
13:25 - 14:10**



REACT

REACT



React jest biblioteką służącą do budowania dynamicznych, złożonych interfejsów użytkownika w sposób deklaratywny i modułowy.

- Zdejmuje z programisty odpowiedzialność za renderowanie oraz aktualizowanie stanu DOM
- Pozwala deklarować strukturę, a także logikę wyświetlania treści w sposób deklaratywny, czyli bardziej naturalny niż w metodach obiektowych czy imperatywnych
- Pozwala umieścić zarówno strukturę, jak i logikę wyświetlania w tym samym miejscu za pomocą wyłącznie JavaScript. Dzięki czemu programista **ma do dyspozycji pełne możliwości języka JavaScript**, a przy tym nie musi poznawać specjalnej (często ograniczonej) składni języków szablonów.

React nie renderuje zmian bezpośrednio, ale poprzez tzw. **Virtual-DOM**

HELLO REACT



```
1 <body>
2   <div id="app"></div>
3
4   <script src="https://unpkg.com/react/umd/react.development.js" crossorigin></script>
5   <script src="https://unpkg.com/react-dom/umd/react-dom.development.js" crossorigin></script>
6
7   <script>
8     var root = React.createElement('div', null, 'Hello!');
9     ReactDOM.render(root, document.getElementById("app"));
10  </script>
11 </body>
12
```

VIRTUAL DOM



VirtualDOM to uproszczona reprezentacja obiektów DOM (Document Object Model), które są w przeglądarce. Cykl renderowania wygląda następująco:

- React buduje deklaratywnie drzewo Virtual DOM, a następnie renderuje je jako przeglądarkowy DOM
- Przy każdej zmianie stanu React buduje ponownie całe drzewo Virtual DOM komponentu
- ReactDOM porównuje aktualne i nowe drzewo, a następnie wprowadza zmiany tylko w tych miejscach DOM, które faktycznie potrzebują zmiany.

Dzięki zminimalizowaniu operacji na DOM React aktualizuje widok **błyskawicznie!**

VIRTUAL DOM

Pozwala generować abstrakcyjny DOM, który można następnie “renderować” UI na wielu różnych platformach:

- **react-dom** - DOM przeglądarki
- React Native - natywne aplikacje iOS i Android
- react-blessed - terminal
- react-canvas - element HTML Canvas
- react-vr / react 360 - aplikacje 3D



ATRYBUTY I KLASY ELEMENTÓW



```
1 React.createElement(  
2     // pierwszy argument to nazwa "tagu", który tworzymy  
3     'div',  
4     // drugi argument to jego właściwości  
5     {  
6         // atrybuty podejamy jako pary klucz: wartość  
7         id: 'rootElement',  
8         // klasy CSS dodajemy używając className zamiast class  
9         className: 'root-element-class',  
10        // style elementu podajemy jako obiekt JS  
11        style: {  
12            borderTop: '1px solid black',  
13        }  
14    },  
15    // trzeci argument to zawartość elementu  
16    "Hello!"  
17 );  
18
```

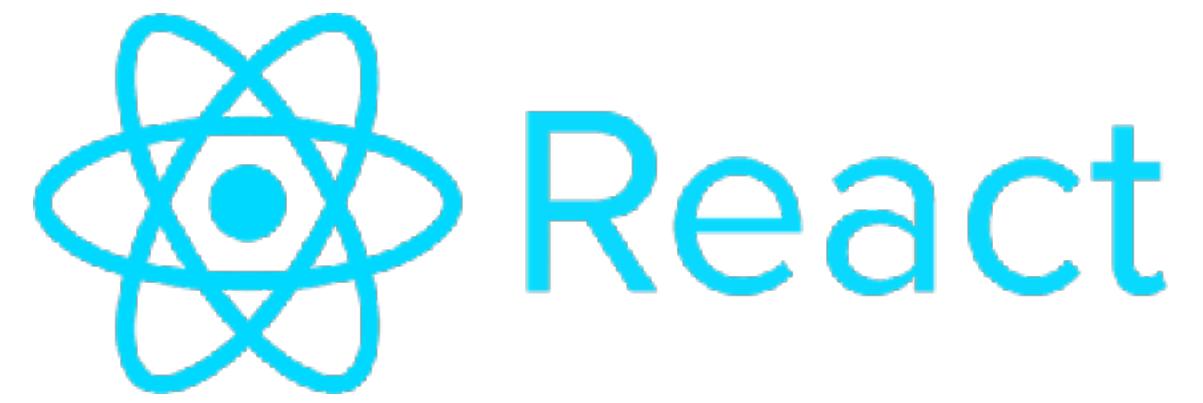
REACT JSX



Korzystając z Transpilacji oraz rozszerzenia JSX możemy budować elementy VirtualDOM w sposób dużo wygodniejszy:

```
1  const Section = <section>
2      <h1>Lista zadań</h1>
3      <h2 id="todos" className="subtitle">Na dziś</h2>
4      <ul>
5          <li>Zrobić zakupy</li>
6          <li>Nauczyć się React!</li>
7      </ul>
8  </section>;
9
10 React.render(Section, document.getElementById('app'));
11
```

DYNAMICZNA TREŚĆ



Korzystając z JSX możemy mieszać statyczną strukturę dokumentu z dynamiczną treścią:

```
1  const section = {  
2    title: 'Zadania',  
3    subtitle: 'Na dziś'  
4  }  
5  
6  const Section = <section>  
7    <h1>{section.title}</h1>  
8    <h2 id="todos" className="subtitle">{section.subtitle}</h2>  
9    <ul>  
10      <li>Zrobić zakupy</li>  
11      <li>Nauczyć się React!</li>  
12    </ul>  
13  </section>;  
14  
15  React.render(Section, document.getElementById('app'));  
16  
17
```



RENDEROWANIE KOLEKCJI

Możemy też renderować dynamiczną kolekcję elementów.

```
1  const section = {  
2      title: 'Zadania',  
3      subtitle: 'Na dziś',  
4      items: [ { id: 1, name: 'Zrobić zakupy'}, { id: 2, name: 'Nauczyć się React!' }],  
5  }  
6  
7  const Section = <section>  
8      <h1>{section.title}</h1>  
9      <h2 id="todos" className="subtitle">{section.subtitle}</h2>  
10     <ul>  
11         {section.items.map(item => <li key={item.id}>{item.name}</li>)}  
12     </ul>  
13 </section>;  
14  
15 React.render(Section, document.getElementById('app'));  
16  
17
```

ATRYBUT KEY



Elementom kolekcji należy przekazać **atrybut key**, który musi być zawsze unikalny i stabilny. Dzięki temu React może optymalnie dokonywać aktualizacji list, używając minimalnej liczby operacji.

Jeśli klucz będzie się zmieniał React przerenderuje ponownie całą listę.

Za aktualizację drzewa DOM odpowiada algorytm różnicujący. Proces w którym odbywa się aktualizacja nazywa się rekoncyliacją.

KOMPONENTY



Podstawowym blokiem aplikacji w React są tzw. Komponenty.

Komponent jest elementem w drzewie DOM, który jest zarządzany przez React.

Funkcja / Klasa komponentu zawiera kod JavaScript kontrolujący wygląd i zachowanie elementu.

Instancja komponentu to właśnie element, który został wyrenderowany przez React przy użyciu tej klasy lub funkcji.

```
1 <MyElement option={zmienna} title="tekst">Treść</MyElement>
```

Komponenty mogą być zagnieżdżane w dowolne struktury podobnie jak HTML czy XML.

Także, jak w HTML możemy przekazywać do komponentu dane jako atrybuty. Poza ciągami znaków możemy przekazywać dowolne obiekty. Parametry te nazywamy właściwościami komponentu (Component properties, albo krócej - props).



DEFINOWANIE KOMPONENTÓW

Pseudoklasa ES6

```
1 var createReactClass = require('create-react-class');
2 var Greeting = createReactClass({
3   render: function() {
4     return <h1>Witaj, {this.props.name}</h1>;
5   }
6});
```

Klasa ES6

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

DEFINOWANIE KOMPONENTÓW



Komponent funkcyjny

Wcześniej określany jako bezstanowy komponent funkcyjny (stateless function component). Od wprowadzenia Hook'ów komponent funkcyjny może mieć swój state i nim zarządzać.

```
1  const Greeting = (props) => {  
2      return <h1>Witaj, {props.name}</h1>  
3  }  
4
```

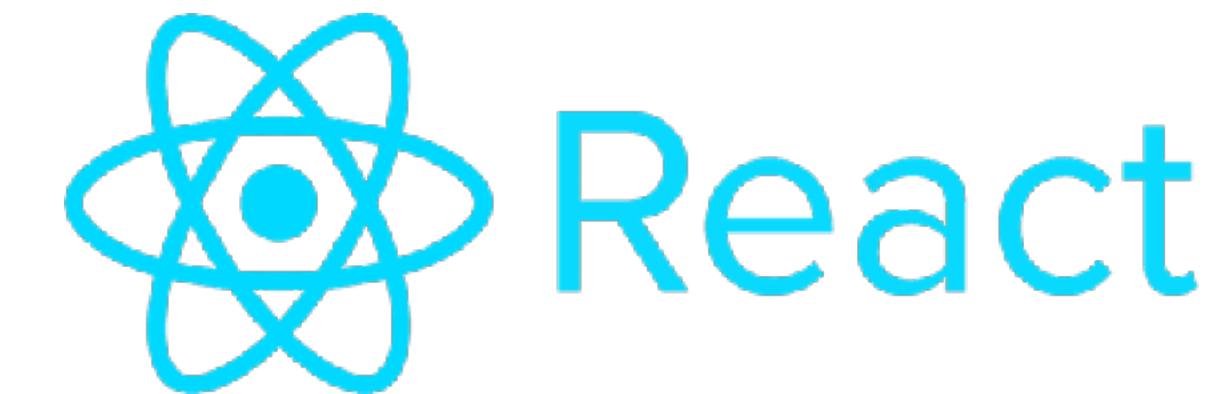
ZADANIE

task/react-basics



20 minut

KOMPONENTY KLASOWE - STATE I PROPS



React „reaguje” na zmiany danych i wywołuje ponowne renderowanie virtualDOM. Renderowanie odbywa się, gdy przekazujemy nowe właściwości do komponentu (**props**) lub gdy zmieniamy jego wewnętrzny stan (**state**).

```
1 const Todo = props => <div>
2   <h3>{props.title}</h3>
3 </div>;
4
5 React.render(<div>
6   <Todo title="Przywitaj się"></Todo>
7 </div>)
8
```

```
1 class Counter extends React.Component {
2   componentDidMount() {
3     setInterval(() => {
4       this.setState({
5         counter: this.state.counter + 1,
6       })
7     }, 1000);
8   }
9
10  render() {
11    return <div>{this.state.counter}</div>
12  }
13}
14
15
```

Przekazanie nowych properties do elementu lub zmiana wewnętrznego stanu powoduje **ponowne wyrenderowanie komponentu**. Props ani state nie wolno modyfikować ręcznie.

DOMYŚLNE STATE I PROPS



```
1  class Clock extends React.Component {  
2      // domyślne props, gdy nie zostaną przekazane  
3      // z nadziednego komponentu  
4      static defaultProps = {  
5          name: 'Hello',  
6      }  
7  
8      constructor(props) {  
9          super(props);  
10         // domyślny stan  
11         this.state = { date: new Date() };  
12     }  
13 }  
14  
15
```

```
1  function Clock(props) {  
2      return <h3>{props.name}</h3>  
3  }  
4  
5  // domyślne propsy, gdy nie zostaną  
6  // przekazane z komponentu nadziednego  
7  
8  Clock.defaultProps = {  
9      name: 'Hello'  
10 }  
11
```

MODYFIKOWANIE STANU



React musi “wiedzieć” kiedy stan komponentu się zmienił. **Nigdy nie modyfikujemy stanu bezpośrednio:**

~~this.state.name = “to nie zadziała”~~

Do modyfikacji stanu służy funkcja **setState**:

`this.setState({ name: “Hello” })`

Metoda **setState działa asynchronicznie**. Pobranie wartości `this.state.name` natychmiast po ustawieniu stanu może nie dać poprawnego rezultatu. Jeśli nowy stan zależy od poprzedniego, użyj funkcji:

```
1  this.setState(function(prevState, props) {  
2    |    return { counter: prevState.counter + props.increment}  
3  });
```

Zmiany są łączone - Wartość `name` nie ulegnie zmianie gdy zmieniamy `counter`.

`this.replaceState(...)` natomiast nadpisuje cały obiekt **this.state**

TYPOWANIE KOMponentu KLASOWEGO



```
5  interface AppProps {}  
6  
7  interface AppState {}  
8  
9  
10 class App extends React.Component<AppProps, AppState> {  
    |
```

ZDARZENIA



Zdarzenia w React nasłuchujemy bezpośrednio na renderowanych elementach:

```
1 <input value={this.state.myvalue} onChange={this.handleChange} />
```

Obsługujemy je w kodzie komponentu, obiekt zdarzenia zostanie przekazany jako parametr:

```
4 handleChange: (event) => {
5   this.setState({
6     myvalue: event.target.value,
7   });
8 }
```

Mögemy też wykonać kod bezpośrednio:

```
10 <div onClick={e => this.myClick(e)}>Press me!</div>
```

ZAGNIEJDZANIE KOMPONENTÓW



Komponenty mogą być zagnieżdżane - tj. dowolny kod stanowiący element JSX przekazany pomiędzy znacznikiem otwierającym a zamykającym komponentu będzie dostępny jako obiekt JSX w zmiennej **props.children**

```
1  const Section = (props) => <div>
2    <h1>{props.title}</h1>
3    {props.children}
4  </div>;
5
6  React.render(
7    <Section title="Tytuł sekcji">
8      Dowolna treść, także komponenty
9    </Section>,
10   document.getElementById('app')
11 );
12
```

CREATE REACT APP



CREATE REACT APP

```
> node_modules
  ●
  ↘ public
    ★ favicon.ico
    ◁ index.html
    🖼 logo192.png
    🖼 logo512.png
    { } manifest.json
    Ⓜ robots.txt
  ↘ src
    # App.css
    JS App.js
    JS App.test.js
    # index.css
    JS index.js
    📺 logo.svg
    JS reportWebVitals.js
    JS setupTests.js
    .gitignore
    { } package-lock.json
    { } package.json
    ⓘ README.md

      3   "version": "0.1.0",
      4   "private": true,
      5   "dependencies": {
      6     "@testing-library/jest-dom": "^5.16.3",
      7     "@testing-library/react": "^12.1.4",
      8     "@testing-library/user-event": "^13.5.0",
      9     "react": "^18.0.0",
     10    "react-dom": "^18.0.0",
     11    "react-scripts": "5.0.0",
     12    "web-vitals": "^2.1.4"
     13  },
     ▷ Debug
     "scripts": {
     14   "start": "react-scripts start",
     15   "build": "react-scripts build",
     16   "test": "react-scripts test",
     17   "eject": "react-scripts eject"
     18 },
     "eslintConfig": {
     19   "extends": [
     20     "react-app",
     21     "react-app/jest"
     22   ]
     23 },
     24 },
     25 },
     26 "browserslist": {
     27   "production": [
```



ZADANIE

task/state



20 minut

METODY CYKLU ŻYCIA



Każdy komponent ma tzw. “Cykl życia”, czyli etapy przez jakie przechodzi od jego utworzenia do jego usunięcia z widoku.

Montowanie:

```
1 // przed zamontowaniem w DOM
2 UNSAFE_componentWillMount()
3
4 // po zamontowaniu w DOM
5 componentDidMount()
6
7 // przed usunięciem z DOM
8 componentWillUnmount()
```

Aktualizowanie:

```
1 // Komponent dostał nowe properties:
2 UNSAFE_componentWillReceiveProps(nextProps = {})
3 getDerivedStateFromProps(nextProps = {})
4
5 // Jeśli zwróci false, react pominie renderowanie:
6 shouldComponentUpdate(nextProps = {}, newState = {})
7
8 // Komponent będzie renderowany, nie zmieniaj stanu
9 UNSAFE_componentWillUpdate()
10
11 // Komponent się wyrenderował, DOM jest stabilny
12 componentDidUpdate()
13
14 // Przed zmianą DOM
15 getSnapshotBeforeUpdate(prevProps = {}, prevState = {})
16
```

METODY CYKLU ŻYCIA



Odmontowywanie:

```
1 // Komponent zostaje usunięty z drzewa DOM  
2 componentWillUnmount()
```

Obsługa wyjątków:

```
1 // Metody zostaną wywołane w razie wystąpienia wyjątku podczas renderowania,  
2 // w metodzie cyklu życia lub w konstruktorze dowolnych komponentów potomnych  
3 getDerivedStateFromError()  
4 componentDidCatch()
```

ZADANIE

task/lifecycle



20 minut

OPEROWANIE NA DOM



Chociaż nie jest wskazane manipulowanie DOM wygenerowanym przez React, to jest taka możliwość. Aby uzyskać dostęp do wybranych elementów DOM, używamy referencji:

```
1  class InputText extends React.Component {  
2      constructor() {  
3          this.refObj = React.createRef()  
4      }  
5  
6      componentDidMount() {  
7          this.refObj.current.focus();  
8      }  
9  
10     render() {  
11         return <input ref="refObj" />  
12     }  
13 }  
14
```

OPEROWANIE NA DOM



Do referencji możemy odwołać się również za pomocą:

```
15     ReactDOM.findDOMNode(this.refObj.current)
```

Powyższy sposób jest uznany jako „wyjście awaryjne”. Ta funkcja jest odradzana, ponieważ zaburza abstrakcję struktury komponentów.

Pamiętaj, że referencje dostępne będą dopiero, gdy element jest wyrenderowany. Zalecane jest używanie referencji w odpowiednich metodach cyklu życia.

Możesz też wykonać kod bezpośrednio, np.

```
10     render() {
11       return <input ref={(elem) => elem.focus()} />
12     }
```

REACT HOOKS



Funkcje w JavaScript nie mają “stanu”. Tzn. każde wywołanie ma dostęp tylko do parametrów (props) i domkniętych zmiennych (closure).

Aby funkcja komponentu przy kolejnym renderowaniu DOM (JSX) mogła odwołać się do wcześniejszych wartości stanu musimy “odwołać” się do stanu używając specjalnych funkcji - React Hooks.

```
1 import React, { useState } from 'react';
2
3 const Example = () => {
4   const [count, setCount] = useState(0);
5
6   return (
7     <div>
8       <p>Kliknąłeś {count} razy!</p>
9       <button onClick={() => setCount(count + 1)}></button>
10    </div>
11  )
12}
13
14 const [aktualnaWartość, funkcjaZmianyStanu] = useState(wartoscPoczatkowa);
15
```

USE EFFECT



Oprócz DOM “renderować” możemy także tzw. “efekty uboczne”.

Po każdym wyrenderowaniu DOM możemy wykonywać imperatywne operacje na poza komponentem używając `useEffect`.

Na przykład możemy ustawić tytuł okna przeglądarki / tytuł karty, pobrać dane z serwera itp.

```
3  const WindowTitleUpdater = () => {
4      const [title, setTitle] = useState('Tytuł');
5
6      useEffect(() => {
7          document.title = title;
8      })
9
10     return <div>
11         <input value={title} onChange={e => setTitle(e.target.value)} />
12     </div>
13 }
14
```



USE EFFECT - DESTRUKTOR

Każde wyrenderowanie komponentu powoduje ponowne uruchomienie efektu.
Działa podobnie jak: **componentDidMount** oraz **componentDidUpdate**

```
3  const WindowResizeWatcher = () => {
4      const [windowSize, setWindowSize] = useState(0);
5
6      useEffect(() => {
7          const resizeHandler = () => {
8              setWindowSize(window.innerWidth);
9          }
10
11         window.addEventListener('resize', resizeHandler);
12
13         return () => {
14             window.removeEventListener('resize', resizeHandler);
15         }
16     })
17
18     return <div>{windowSize}px</div>
19 }
```

USE EFFECT - WARUNKOWE EFEKTY



Drugi argument **useEffect** to tablica zależności, od których zależy wywołanie efektu. Nie przekazując drugiego argumentu **useEffect** wykona się po każdym renderze. Przekazanie pustej tablicy oznacza, że nasz efekt nie jest zależny od jakichkolwiek zależności, więc wykona się tylko raz. Przekazanie wartości do tablicy oznacza, że przy zmianie zależności efekt się wykonać.

```
6  useEffect(() => {}, []); // wykona się raz
7
8  useEffect(() => {}); // wykona się przy każdym renderze
9
10 useEffect(() => {}, [props.title, value]); // wykona się, gdy któraś z zależności się zmieni
11
```

ZADANIE

task/effects



20 minut

USE LAYOUT EFFECT



Zachowanie podobne do `useEffect`. Różnica polega na tym, że jest uruchamia się synchronicznie po mutacji w strukturze DOM.

Przydatny, gdy chcemy aby widok był zgodny z danymi w sytuacji, gdy operujemy bezpośrednio na strukturze DOM.

INNE HOOKI



- `useContext`
- `useRef`
- `useReducer`
- **`useMemo`** - memoizacja wartości
- **`useCallback`** - memoizacja callbacków
- **`useImperativeHandle`** - dostosowuje wartość instancji, jaka przekazywana jest komponentom przodków, kiedy używają właściwości `ref`
- **`useDebugValue`** - może zostać użyty do wyświetlania etykiet dla własnych hooków w narzędziu React DevTools.
- **`useId`** - generuje unikalne ID
- **`useTransition`** - możemy określić, która aktualizacja stanu ma wyższy priorytet
- **`useDefferedValue`** - może wyświetlić starą wartość, zanim nowa będzie gotowa

ZADANIE

task/hooks



30 minut

FORMULARZE



Dodanie do pola formularza atrybutu `value=""` zamienia to pole w tzw. pole kontrolowane, czyli pole którego stan jest powiązany ze stanem komponentu i tylko komponent może ten stan zmienić.

Zapis w formie `<input value={this.state.myValue} />` sprawia, że pole uniemożliwia ręczną zmianę swojego stanu. Jedyne sposób na zmianę wartości tego pola to zmiana stanu komponentu, czyli skorzystanie z metody **setState**:

```
1 <input value={this.state.myValue} onChange={this.handleChange} />
2
3 handleChange = e => this.setState({myValue:e.target.value });
4
```

Pole, które nie posiada atrybutu value to tzw. **pole niekontrolowane**. Elementy typu `<select>` czy `<textarea>` także korzystają z atrybutu value.

ZADANIE

task/forms



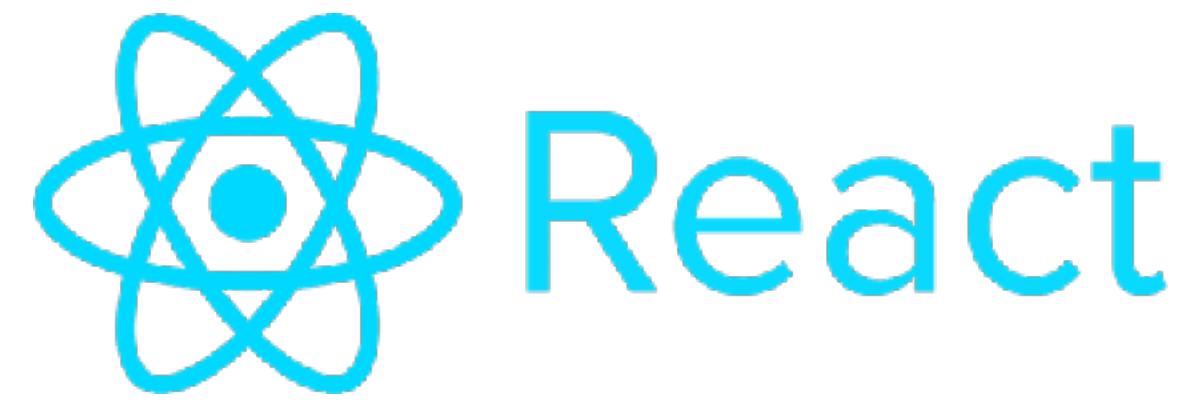
30 minut

REACT ROUTER DOM



Router pozwala “podmienić” renderowany na stronie komponent zależnie od aktualnego adresu url w przeglądarce:

```
 9  function App() {
10    return (
11      <Router>
12        <div className="App">
13          <Nav />
14          <Routes>
15            <Route path='/' element={<HomePage />} />
16            <Route path='/contact' element={<ContactPage />} />
17          </Routes>
18        </div>
19      </Router>
20    );
}
```



Router pozwala tworzyć parametryzowane ścieżki oraz dynamiczne linki:

```
13 const Nav = <ul>
14   |   <li>
15   |   |   <Link to="/netflix">Netflix</Link>
16   |   </li>
17   |   <li>
18   |   |   <Link to="/yahoo">Yahoo</Link>
19   |   </li>
20   |   <li>
21   |   |   <Link to="/youtube">Youtube</Link>
22   |   </li>
23   |   <li>
24   |   |   <Link to={some.dynamic.id}>Dynamiczny link</Link>
25   |   </li>
26 </ul>;
```

```
5  const ChildComponent = () => {
6    const params = useParams();
7
8    return (
9      |   <div>ID: {params.id}</div>
10     )
11 }
```

```
28   <Route path="/:id" component={ChildComponent} />
```

ZADANIE

task/router



20 minut

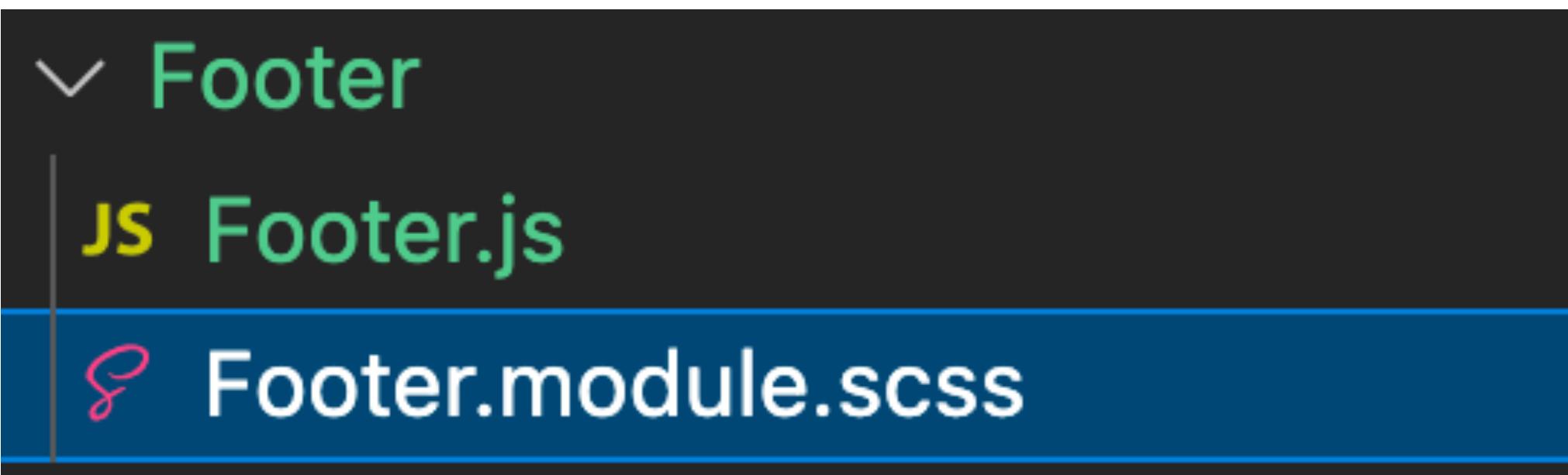
A close-up photograph of a variety of wrapped candies. The candies are wrapped in shiny, translucent paper that reflects light, giving them a metallic appearance. They come in various colors including yellow, orange, red, green, and pink. Some have intricate patterns like waffles or stripes. The candies are piled together, creating a vibrant and sweet-looking composition.

STYLE
css

CSS MODULES



Pozwala korzystać z preprocessorów takich jak Sass. Pliki ze stylami są traktowane jako moduły. Każdy komponent powinien mieć swój własny plik modułu ze stylami. Style globalne przechowujemy w osobnym katalogu.



```
1 import styles from './Footer.module.scss'
```

STYLED COMPONENTS



Biblioteka do tworzenia stylów w formie komponentów. Komponenty wizualne możemy dodawać jako dedykowane do naszego komponentu lub jako samodzielne komponenty wizualne występujące w naszej aplikacji.

```
1 import styled from 'styled-components';
2
3 export const HeaderWrapper = styled.div`
4   background: #efefef;
5   padding: 2rem;
6`
```

```
6   |   <HeaderWrapper>
7   |   |   <Nav />
8   |   </HeaderWrapper>
```

CSS IN JS



```
import jss from 'jss'
import preset from 'jss-preset-default'
import color from 'color'

// One time setup with default plugins and settings.
jss.setup(preset())

const styles = {
  '@global': {
    body: {
      color: 'green'
    },
    a: {
      textDecoration: 'underline'
    }
  }
}.
```

ZADANIE

task/styles



30 minut

KOMUNIKACJA Z API



PROMISE



```
7  const fetchCode = () => {
8    fetch('https://api.postcodes.io/random/postcodesaa')
9      .then(res => {
10        return res.json()
11      })
12      .then(data => {
13        if (data.status === 404) {
14          handleError();
15        }
16        setPostCode(data.result);
17      })
18      .catch((err) => {
19        handleError();
20      })
21      .finally(() => {
22        setIsLoading(false);
23      })
24    }
```

ASYNC/AWAIT



```
26 |   const fetchCode = async () => {
27 |     try {
28 |       const data = await fetch('https://api.postcodes.io/random/postcodesss');
29 |       const res = await data.json();
30 |
31 |       setPostCode(res.result);
32 |     } catch(err) {
33 |       handleError();
34 |     } finally {
35 |       setIsLoading(false);
36 |     }
37 |   }
```

KOMUNIKACJA Z API



Zapytania do API są wykonywane po wyrenderowaniu komponentu. Zapytania możemy wykonywać z poziomu **useEffect**.

```
26 > const fetchCode = async () => { ...
37 }
38 ▶
39     useEffect(() => {
40         fetchCode();
41     }, [])
42 }
```

ZADANIE

task/api-solution



30 minut

FLUX



sages

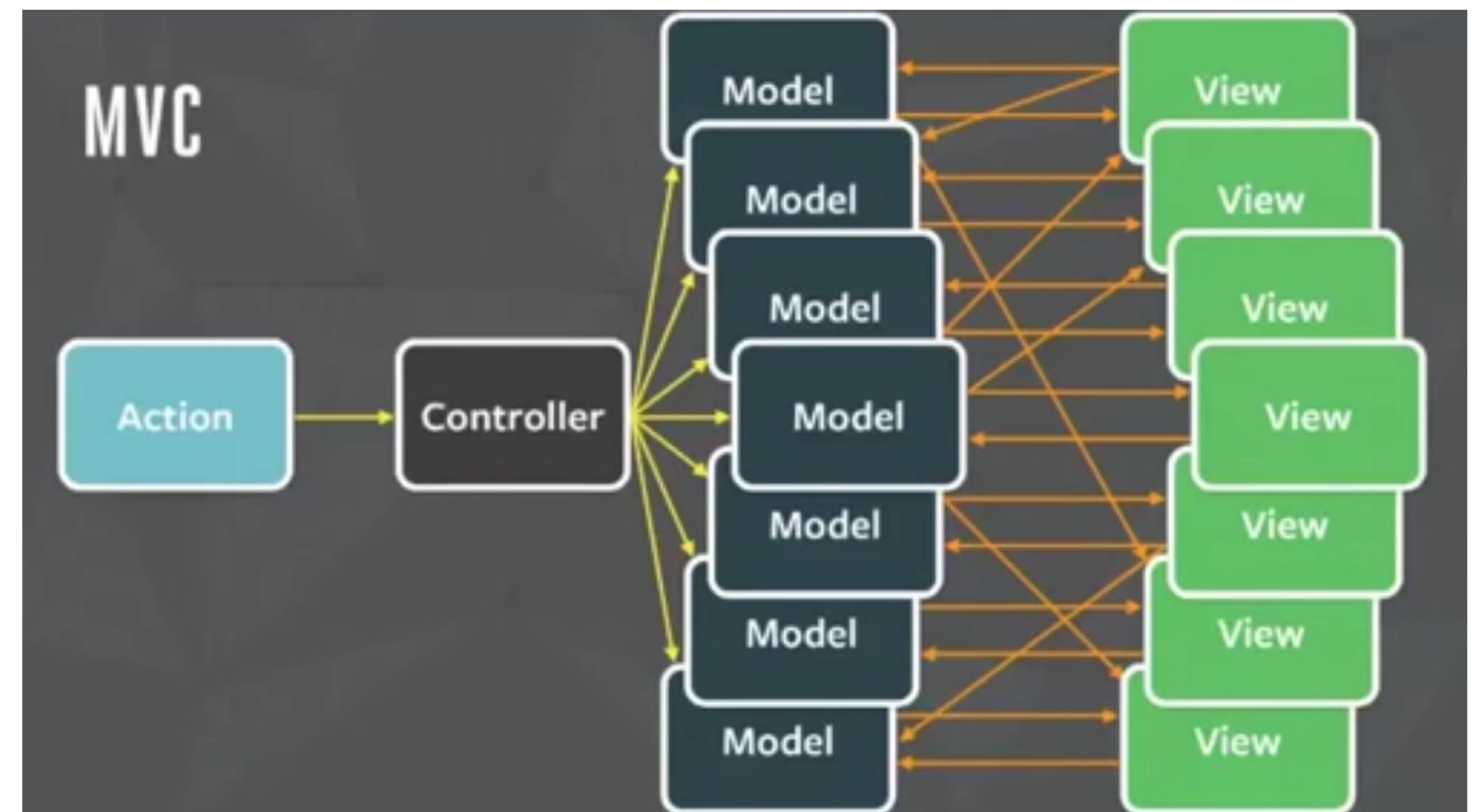
PROBLEM Z MVC



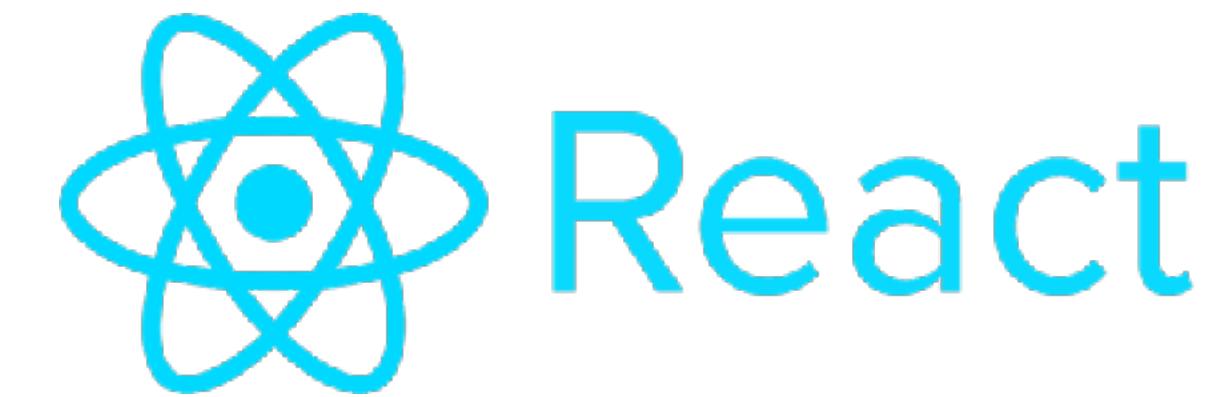
Modele i widoki tworzą wiele dwukierunkowych powiązań.

Akcja użytkownika może wpływać na wiele modeli i wiele widoków, które mogą zmieniać kolejne modele... itd.

Trudno jest określić dokładnie kierunek przepływu danych. Łatwo o błędy.

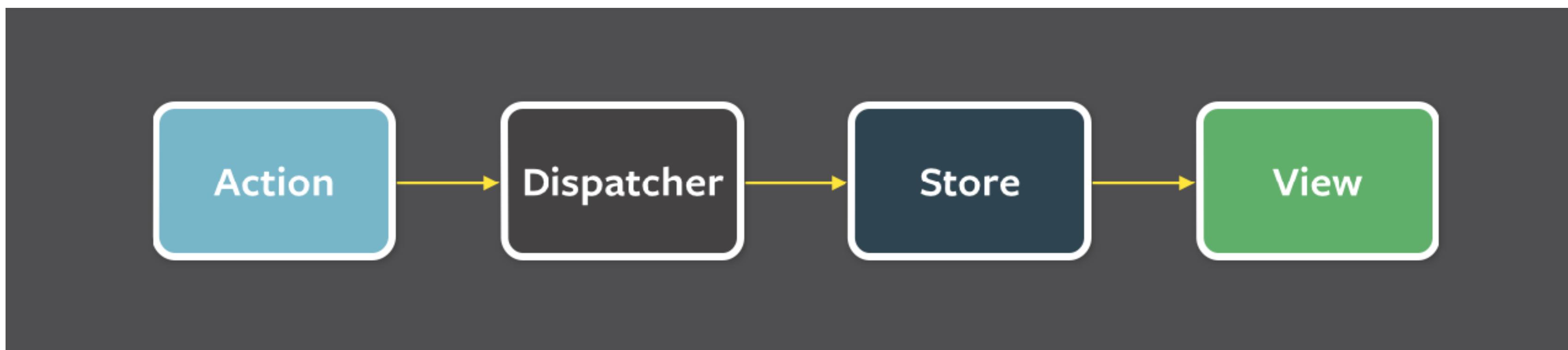


FLUX

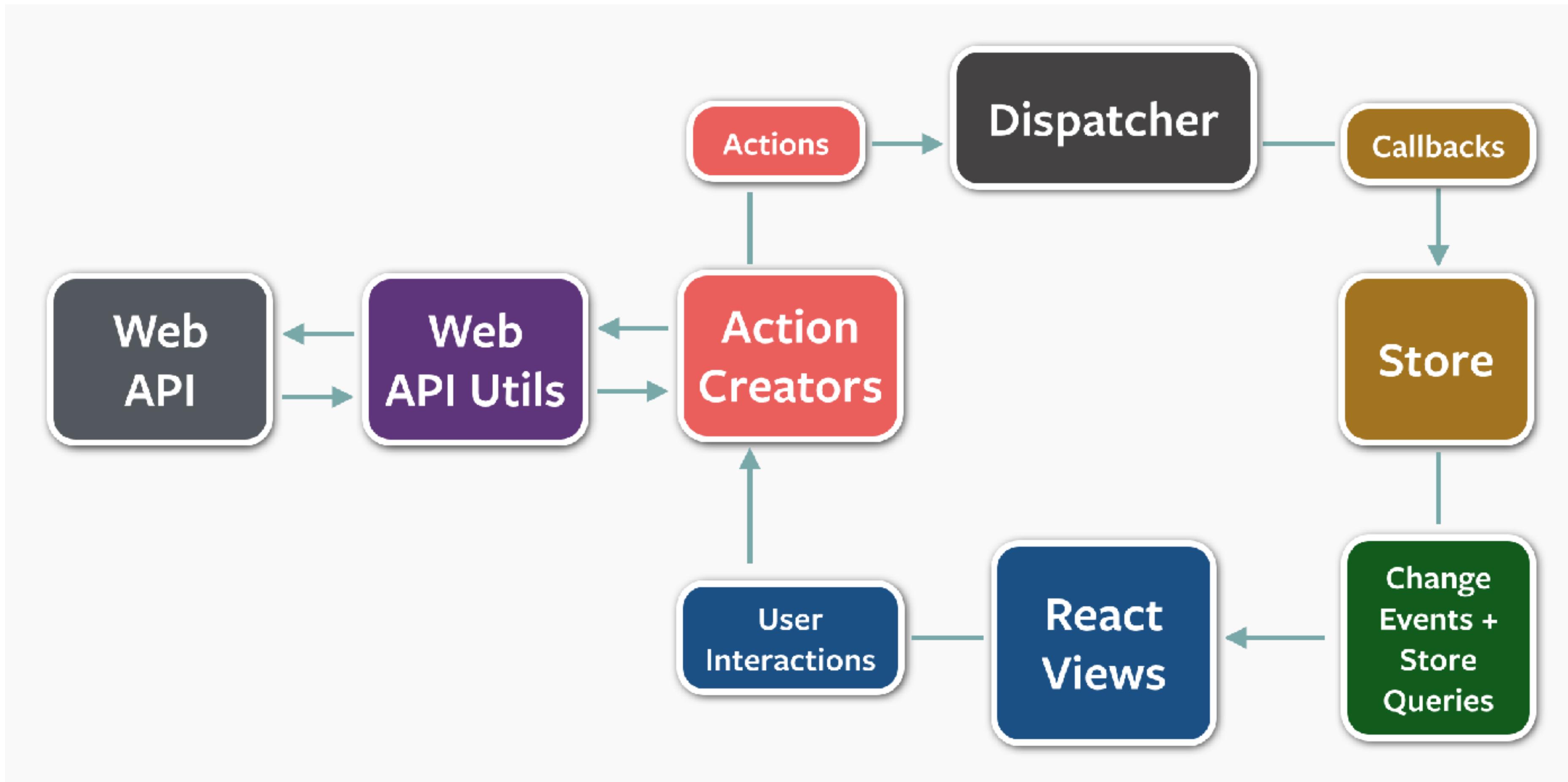


Architektura Flux zakłada jednokierunkowy przepływ danych.

- Akcje stanowią jedyny sposób zmiany stanu aplikacji (Action)
- Dyspozytor (Dispatcher) przekazuje akcje do odpowiednich Magazynów stanu (Store)
- Magazyn jest “jednym źródłem prawdy”, zmienia swój stan w reakcji na akcje. Stan aplikacji pobierany jest z magazynów i przekazywany do widoków.
- Widok (View) obserwuje magazyny i renderuje zmiany w aplikacji ele i widoki tworzą wiele dwukierunkowych powiązań.



FLUX W PRAKTYCE



Akcje są obiektami zawierającymi przynajmniej 2 pola:

- `type` - Typ akcji, wg którego magazyn wie jak przetworzyć ładunek (payload)
- `payload` - ładunek akcji, czyli parametry akcji, np. dane do zapisania

```
1  const ADD_FLIGHT = {  
2      type: 'ADD_FLIGHT',  
3      payload: {  
4          id: 3542,  
5          name: 'WAW/SFO'  
6      }  
7  }
```

```
9  const REMOVE_FLIGHT = {  
10     type: 'REMOVE_FLIGHT',  
11     payload: {  
12         id: 3542,  
13     }  
14 }
```

```
16 const UPDATE_FLIGHT = {  
17     type: 'UPDATE_FLIGHT',  
18     payload: {  
19         id: 3542,  
20         name: 'WAW/SFO',  
21         status: 'active'  
22     }  
23 }
```

MAGAZYN



Magazyn składa się z 2 części:

- Obiektu przechowującego stan
- Funkcji obsługującej przychodzące akcje, modyfikującej stan magazynu i powiadamiającej o zmianach

Dobrą praktyką jest stworzenie jednego obiektu (klasy), który spełnia te role.

```
1  class FlightStore extends EventEmitter {  
2      constructor() {  
3          this.store = { flights: [] }  
4      }  
5  
6      handleAction(action) {  
7          switch(action.type){  
8              case 'ADD_FLIGHT':  
9                  this.store.flights.push(action.payload)  
10                 break;  
11             }  
12         }  
13         this.notifyViews(this.state);  
14     }  
15 }  
16 }
```

KORZYŚCI

Dzięki niemutowalnym strukturom danych wykrywanie zmian staje się bardzo proste i wydajne - wystarczy porównać cały obiekt z jego poprzednią wersją. Jeśli jest to ten sam obiekt to nic się nie zmieniło i nie ma potrzeby aktualizacji widoku.



```
1  let state = null;
2
3  function dispatch(action) {
4      const newState = reducer(state, action);
5
6      if (newState !== state) {
7          state = newState;
8
9          this.notifyViews(this.state);
10     } else {
11         // brak zmian – nie aktualizujemy
12     }
13 }
14
```

REDUX



REDUX

Redux jest podobną do Flux architekturą, inspirowaną rozwiązaniami funkcyjnymi. Jego głównymi założeniami są:

- Jedno źródło stanu aplikacji
- Niemutowalny stan
- Redukowanie listy akcji do aktualnego spójnego stanu może wpływać na wiele modeli i wiele widoków, które mogą zmieniać kolejne modele... itd.



Redux

```
1 import { createStore } from 'redux'  
2  
3 function counter(state = 0, action) {  
4   switch (action.type) {  
5     case 'INCREMENT':  
6       return state + 1  
7     case 'DECREMENT':  
8       return state - 1  
9     default:  
10       return state  
11   }  
12 }  
13 let store = createStore(counter)  
14  
15 store.subscribe(() => console.log(store.getState()))  
16 store.dispatch({ type: 'INCREMENT' }) // 1  
17 store.dispatch({ type: 'INCREMENT' }) // 2  
18 store.dispatch({ type: 'DECREMENT' }) // 1  
19
```

REDUCERS

Reducer może pracować z zagnieżdżonym stanem. Należy jednak pamiętać aby zwracać każdorazowo nowy obiekt, jeśli została w nim wprowadzona jakakolwiek zmiana.



```
1  function reducer(state, action) {  
2      switch (action) {  
3          case 'INC':  
4              return { ...state, counter: state.counter + 1 };  
5          case 'DEC':  
6              return { ...state, counter: state.counter - 1 };  
7          default:  
8              return state;  
9      }  
10 }  
11
```

Nie modyfikuj obiektów bezpośrednio:

```
12  
13     state.counter = state.counter + 1; // ŹLE  
14
```



Redux

ACTION CREATORS

Aby ułatwić tworzenie akcji i nie zapomnieć o istotnych polach tworzymy funkcje - kreatory akcji.

W ten sposób dużo wygodniej możemy wprowadzać zmiany stanu aplikacji

```
1  const addIngredient = (recipe, name, quantity) => ({  
2    type: 'ADD_INGREDIENT',  
3    recipe,  
4    name,  
5    quantity  
6  });  
7  
8  store.dispatch(addIngredient('Omelette', 'Eggs', 3));  
9
```



Redux

ZAGNIEŻDŻANIE REDUKTORÓW

Reduktory możemy zagnieździć:

```
1 const recipesReducer = (recipes, action) => {
2   switch (action.type) {
3     case 'ADD_RECIPE':
4       return recipes.concat({
5         name: action.name
6       });
7     }
8
9   return recipes;
10};
11 const ingredientsReducer = (ingredients, action) => {}
12
13 const rootReducer = (state, action) => {
14   return Object.assign({}, state, {
15     recipes: recipesReducer(state.recipes, action),
16     ingredients: ingredientsReducer(state.ingredients, action)
17   })
18 }
```

lub kombinować:

```
1 export default combineReducers({
2   recipes: recipesReducer,
3   ingredients: ingredientsReducer
4 });
5
6
```

MIDDLEWARE



Middleware pozwala “przechwycić” akcje i zmodyfikować je, wysłać zależne akcje oraz wykonać inne operacje dla każdej wysłanej akcji:

```
1 import { createStore, applyMiddleware } from 'redux';
2 import rootReducer from 'reducers/root';
3
4 const loggingMiddleware = ({ getState, dispatch }) => (next) => (action) => {
5   console.log(`Action: ${action.type}`, action);
6   next(action);
7 }
8
9 const initialState = {};
10
11 export default createStore(
12   rootReducer,
13   initialState,
14   applyMiddleware(loggingMiddleware)
15 );
16
17
18
```

EFEKTY UBOCZNE

Middleware można także wykorzystać do wykonywania efektów ubocznych, takich jak zapytywanie serwera:

```
1  function fetchData(url, callback) {
2      fetch(url)
3          .then((response) => {
4              if (response.status !== 200) {
5                  console.log(`Error fetching data: ${ response.status }`);
6              } else {
7                  response.json()
8                      .then(callback);
9              }
10         })
11     .catch((err) => console.log(`Error fetching data: ${ err }`));
12 }
13
14 const apiMiddleware = ({ dispatch }) => next => action => {
15     if (action.type === FETCH_MY_DATA) {
16         fetchData(URL, data => dispatch(setMyData(data)));
17     }
18     next(action);
19 };
20
21
```

Popularne middleware:

- redux-thunk
- redux-promise
- redux-saga

REACT - REDUX

Dzięki `connect()` podłączenie komponentów do magazynu redux-store możemy wykonać automatycznie używając Providera:

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



Redux

```
1 import { connect } from 'react-redux'  
2  
3 const mapStateToProps = (state) => {  
4   return {  
5     todos: state.todos  
6   }  
7 }  
8  
9 const mapDispatchToProps = (dispatch) => {  
10  return {  
11    onTodoClick: (id) => {  
12      dispatch(toggleTodo(id))  
13    }  
14  }  
15 }  
16  
17 const VisibleTodoList = connect(  
18  mapStateToProps,  
19  mapDispatchToProps  
20 )(TodoList)  
21  
22
```

MOBX

sages

ALTERNatywa dla REDUXA



*Events invoke actions.
Actions are the only thing that modify state and may have other side effects.*

State is observable and minimally defined. Should not contain redundant or derivable data. Can be a graph, contain classes, arrays, refs, etc.

Computed values are values that can be derived from the state using a pure function. Will be updated automatically by MobX and optimized away if not in use.

Reactions are like computed values and react to state changes. But they produce a side effect instead of a value, like updating the UI.

```
@action onClick = () => {
  this.props.todo.done = true;
}
```

```
@observable todos = [
  { title: "learn MobX",
    done: false
  }
]
```

```
@computed get completedTodos() {
  return this.todos.filter(
    todo => todo.done
  )
}
```

```
const Todos = observer({ todos } =>
  <ul>
    todos.map(todo => <TodoView ... />
  </ul>
)
```



OBSERWOWALNY STAN I WARTOŚCI WYLICZALNE

```
1 import { observable, computed } from 'mobx';
2
3 class Store {
4   @observable name = 'Jan';
5   @computed get decorated() {
6     return `${this.name} Kowalski!`;
7   }
8 }
9
10 export default Store;
11
```



INTEGRACJA Z REACT

```
1 import { observer } from "mobx-react"
2 import { useState } from "react"
3
4 const TimerView = observer(() => {
5   const [timer] = useState(() => new Timer());
6   return <span>Seconds passed: {timer.secondsPassed}</span>
7 })
8
9 ReactDOM.render(<TimerView />, document.body)
10
```

TESTOWANIE





JEST + TESTING-LIBRARY/REACT

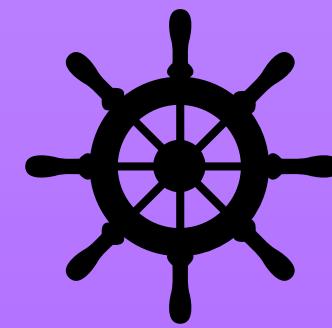
```
2 import { render, screen } from '@testing-library/react';
3 import App from './App';
4
5 Run | Debug
6 test('renders learn react link', () => {
7   render(<App />);
8   const linkElement = screen.getByText(/To jest moja aplikacja/i);
9   expect(linkElement).toBeInTheDocument();
10});
```

CODE:XXXXXX



kahoot.it

ANKIETA



10 minut
tinyurl.com/yc5bjvym

DZIĘKUJĘ ZA UWAGĘ

- mail@mateuszjablonski.com
- mateuszjablonski.com
- [github.com/matwjablonski/
deloitte-react](https://github.com/matwjablonski/deloitte-react)

