

Budowanie wydajnych i nowoczesnych aplikacji webowych w oparciu o React i powiązane technologie

Prowadzący szkolenie:

Mateusz Kulesza

Senior Software Developer,
Team Leader, Scrum Master
Project Manager
Konsultant i szkoleniowiec



Ustalenia:

- Cel i plan szkolenia
- Obowiązki bieżące
- Pytania, dyskusje, potrzeby
- Elastyczność zagadnień

Wieloletnie doświadczenie komercyjne w pracy z :

- HTML5, CSS3, SVG, EcmaScript 5 i 6
- jQuery, underscore, backbone.js
- canjs, requirejs, dojo ...
- Grunt, Gulp, Webpack, Karma, Jasmine ...
- Angular.JS, Angular, **React**, RxJS, Flux

Zarządzanie pakietami



Node Package
Manager

- system zarządzania zależnościami dla server-side js
- zależności opisywane z dokładnością do wersji w pliku `package.json`
- `npm install` - instaluje pakiety, których jeszcze nie ma w projekcie
- `npm update` - sprawdza, czy istnieją nowsze wersje pakietów + instaluje
- `npm install nazwa-pakietu --save-dev` - instaluje pakiet, dodaje go do `package.json`



JavaScript 2015

“JavaScript next”

~~ECMAScript 6?~~

ECMAScript 2015

- moduły
- dużo dobrego „cukru składniowego”
- leksykalny zasięg (let) i wiele innych
- można używać...

już! **BABEL**

<https://babeljs.io/>

Webpack

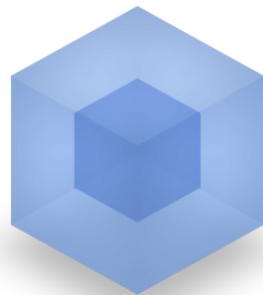
- module bundler
- obsługuje wiele formatów modułów: ES2015, AMD, CommonJS (npm)
- traktuje wszystko jak moduły (np. scss, html, grafiki)
- React Hot Loader
- dobrze współpracuje z popularnymi task runnerami (Gulp, Grunt)
- de facto standard w środowisku React, popularny również poza

```
// instalacja webpack-cli, globalnie npm
```

```
install webpack --global
```

```
// oraz instalacja lokalna
```

```
npm install webpack --save-dev
```



webpack
MODULE BUNDLER

WebPack - Konfiguracja

```
module.exports = {  
  entry: [  
    './js/index.js'  
  ],  
  output: {  
    path: __dirname + '/static/',  
    filename: 'bundle.js'  
  },  
  plugins: [],  
  module: {  
    rules: [{  
      test: /\.js$/, use: ['babel-loader'], exclude: /node_modules/  
    }]  
  },  
  devtool: 'source-map'  
};
```

Babel - Konfiguracja z .babelrc

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

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

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


Funkcje Anonimiwe

(Lambda)

// Domyślnie - zwraca wyrażenie

```
var odds = myArr.map(v => v + 1);  
var nums = myArr.map((v, i) => v + i);  
var pairs = myArr.map(v => (  
    {even: v, odd: v + 1}  
));
```

// Deklaracje umieszczamy w klamrach

```
nums.filter(v => {  
    if (v % 5 === 0) {  
        return true;  
    }  
});
```

// Leksykalne this

```
var bob = {  
    _name: "Bob",  
    _friends: [],  
    getFriends() {  
        return this._friends.forEach(f =>  
            this._name + " knows " + f  
        )  
    }  
}
```

Destrukturyzacja

```
// list matching
```

```
var [a, , b] = [1,2,3];
```

```
// object matching
```

```
var { op: a, lhs: { op: b }, rhs: c }
```

```
  = getASTNode() // i.e. { op: 'a', lhs: {op: 'b' }, rhs: 'c' }}
```

```
// object matching shorthand
```

```
var {op, lhs, rhs} = getASTNode()
```

```
// Can be used in parameter position
```

```
function g({name: x}) {
```

```
  console.log(x);
```

```
}
```

```
g({name: 5})
```

Default, ...Spread i ...Rest

```
function f(x, y = 12) {  
  // domyślna wartość y (jeśli y === undefined)  
  return x + y;  
}  
f(3) === 15;  
function f(x, ...y) {  
  // y jest tablicą pozostałych wartości  
  return x * y.length;  
}  
f(3, "hello", true) === 6;  
function f(x, y, z) {  
  return x + y + z;  
}  
// przekazanie każdego elementu tablicy osobno  
f(...[1, 2, 3]) === 6;
```

Dynamiczny Literał

```
var obj = {  
  __proto__: theProtoObj,  
  // === 'handler: handler'  
  handler,  
  // === toString: function toString() {  
    toString() {  
      // Super calls  
      return "d " + super.toString();  
    },  
  // Dynamiczne nazwy własności  
  [ 'prop_' + (() => 42)() ]: 42  
};
```

Rozszerzony obiekt Array

```
// konwertuje obiekt tablicopodobny na prawdziwą tablice
```

```
Array.from(document.querySelectorAll('*'))
```

```
// Tworzy nowa Tablicę, podobnie do new Array(), ale ma inne zachowanie dla 1 parametru
```

```
Array.of(1, 2, 3)
```

```
[0, 0, 0].fill(7, 1) // [0, 7, 7]
```

```
[1, 2, 3].find(x => x == 3) // 3
```

```
[1, 2, 3].findIndex(x => x == 2) // 1
```

```
[1, 2, 3, 4, 5].copyWithin(3, 0) // [1, 2, 3, 1, 2]
```

```
["a", "b", "c"].entries() // iterator [0,"a"],[1,"b"],[2,"c"]
```

```
["a", "b", "c"].keys() // iterator 0, 1, 2
```

```
["a", "b", "c"].values() // iterator "a", "b", "c"
```

Moduły

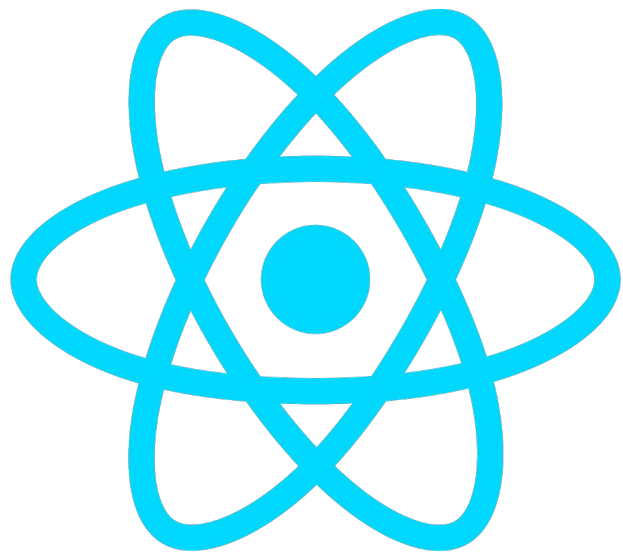
```
//----- lib.js -----  
export const sqrt = Math.sqrt;  
export function square(x) {  
    return x * x;  
}  
export function diag(x, y) {  
    return sqrt(square(x) + square(y));  
}  
  
export default diag;
```

```
//----- main.js -----  
import { square, diag } from 'lib';  
console.log(square(11)); // 121  
  
// -- lub cały moduł jako zmienna --  
import * as lib from 'lib';  
console.log(lib.square(11)); // 121  
  
// -- lub domyślny obiekt ( diag )  
import diagonal from 'lib';
```

Obietnice (Promise)

```
function timeout(duration = 0) {  
    return new Promise((resolve, reject) => {  
        setTimeout(resolve, duration);  
    })  
}
```

```
var p = timeout(1000).then(() => {  
    return timeout(2000);  
}).then(() => {  
    return Promise.reject("hmm... błąd!");  
}).catch(err => {  
    return Promise.all([timeout(100), timeout(200)]);  
}).then((result) => {  
    console.log("Wynik to: " + result);  
})
```



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

```
<body>
  <div id="app"></div>

  <script src="react/react.js"></script>
  <script src="react/react-dom.js"></script>

  <script>
    var root = React.createElement('div', null, 'Hello!');
    ReactDOM.render(root, document.getElementById("app"));
  </script>
</body>
```

React VirtualDOM

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!**

React VirtualDOM

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

- **react-dom** - DOM przeglądarki
- React Native - natywne aplikacje iOS i Android
- react-blessed - terminal
- gl-react - WebGL
- react-canvas - element HTML Canvas

Atrybuty i klasy elementów

```
React.createElement('div', {  
  
  // Atrybuty elementu podajemy jako klucz:wartość  
  id: 'rootElem',  
  
  // Klasy CSS dodajemy używając className.  
  // ( Słowo class jest zarezerwowane w JavaScript )  
  className: 'root-element',  
  
  // Style elementu podajemy jako obiekt.  
  // Dzięki czemu łatwiej jest dynamicznie aktualizować pojedyncze style  
  style:{  
    borderTop: '1px solid black'  
  }  
  
}, 'Hello!'); // Zawartość elementu podajemy jako trzeci argument
```

React JSX

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

```
const Section = <section>
  <h1>Lista zadań</h1>
  <h2 id="todos" className="subtitle">Na dziś</h2>
  <ul>
    <li>Zrobić Zakupy</li>
    <li>Nauczyć się React!</li>
  </ul>
</section>
```

```
React.render( Section, document.getElementById('app'))
```

Dynamiczna treść

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

```
var section = {  
  title: 'Zadania',  
  subtitle: 'Na dziś'  
}
```

```
React.render(<section>  
  <h1>{ section.title }</h1>  
  <h2 id="todos" className="subtitle">{ section.subtitle }</h2>  
</section>, document.getElementById('app'))
```

Renderownie kolekcji

... a także dynamiczne kolekcje elementów:

```
section.items = [ {id:1, name:'Zakupy'}, {id:2, name:'React'} ];
```

```
var Section = <section>
```

```
  <h1>{ section.title }</h1>
```

```
  <h2 id="todos" className="subtitle">{ section.subtitle }</h2>
```

```
  <ul>
```

```
    { section.items.map( item => <li key={item.id}>{item.name}</li> ) }
```

```
  </ul>
```

```
</section>;
```

```
React.render(Section , document.getElementById('app'));
```

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

Komponenty

Podstawowym blokiem aplikacji w React są tzw. Komponenty.

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

Klasa komponentu zawiera kod JavaScript kontrolujący wygląd i zachowanie elementu.

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

`<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. Jako że poza ciągami znaków możemy przekazywać dowolne obiekty, parametry te nazywamy właściwościami komponentu (Component properties, albo krócej - props)

Definiowanie komponentów


Klasa ES5	<pre>var MyComponent = React.createClass({ render: function(){ return <div> dowolny DOM, oraz {this.props.tekst} </div> } })</pre>
Klasa ES6	<pre>class MyComponent extends React.Component{ render() { return <div> dowolny DOM, oraz {this.props.tekst} </div> } }</pre>
Komponent bezstanowy Funkcja	<pre>const MyComponent = (props) => { return <div> dowolny DOM, oraz {props.tekst} </div> }</pre>

Komponenty - state, props

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


```
var Todo = ( props ) => <div>
  <h3> { props.tekst } </h3>
</div>;

React.render(<div>
  <Todo tekst="Zakupy"></Todo>
  <Todo tekst={tododata.name}></Todo>
  <Todo tekst={ "Todo" + uuid }></Todo>
</div>, document.getElementById( 'app' ))
```

An arrow points from the `props` argument in the `React.render` call to the `props` parameter in the `Todo` component definition.

```
class Counter extends React.Component{
  componentDidMount(){
    setInterval(()=>{
      this.setState({
        counter: this.state.counter + 1
      })
    },1000);
  }

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

An arrow points from the `this.state.counter` property access inside the `setInterval` callback to the `this.state.counter` property access inside the `render` method.

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

Domyślne state i props

```
class Clock extends React.Component {  
  // Domyślne props, gdy nie zostaną przekazane z nadrzędnego komponentu:  
  static defaultProps = {  
    name: "Hello"  
  };  
  
  constructor(props) {  
    super(props);  
    // Domyślny stan  
    this.state = {date: new Date()};  
  }  
}
```

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:

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

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

this.replaceState({...}) natomiast nadpisuje cały obiekt **this.state**

Zdarzenia

Zdarzenia w React nasłuchujemy bezpośrednio na renderowanych elementach:

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

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

```
handleChange: function(event){  
  this.setState({  
    myvalue: event.target.value  
  });  
}
```

Możemy też wykonać kod bezpośrednio:

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

Metody cyklu życia

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

Montowanie:

// Przed zamontowaniem w DOM:

~~UNSAFE_componentWillMount~~()

// Po zamontowaniu w DOM:

componentDidMount()

// Przed usunięciem z DOM:

componentWillUnmount()

// Przed Zmianą DOM

getSnapshotBeforeUpdate(prevProps = {}, prevState = {})

Aktualizowanie:

// Komponent dostał nowe properties:

~~UNSAFE_componentWillReceiveProps~~(newProps = {})

static getDerivedStateFromProps(newProps = {})

// Jeśli zwróci false, react pominie renderowanie:

shouldComponentUpdate(newProps={}, newState={})

// Komponent będzie renderowany, nie zmieniaj stanu

~~UNSAFE_componentWillUpdate~~()

// Komponent się wyrenderował, DOM jest stabilny

componentDidUpdate()

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:

```
this.refObj = createRefObj()  
...  
<input ref="refObj" />
```

Odwołujemy się w kodzie poprzez:

```
ReactDOM.findDOMNode( this.refObj.current ).focus()  
ReactDOM.findDOMNode( this.refObj.current ).value
```

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. `<input ref={ (elem) => elem.focus() } />`

React Hooks

```
import React, { useState } from 'react';

function Example() {
  // useState zwraca tablice z 2 elementami
  // zmienną 'count' i funkcją 'setCount':
  const [count, setCount] = useState(0);

  return (
    <div>
      {/* Możemy odwołać się do zmiennej stanowej: */}
      <p>You clicked {count} times</p>

      {/* Możemy wywołać funkcję setter setCount: */}
      <button onClick={() => setCount(count + 1)}>
        Click me
      </button>
    </div>
  );
}
```

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.

`const [aktualnaWartosc, funkcjaZmianyStanu] = useState(wartoscPoczatkowa)`

useEffect

```
import React, { useState } from 'react';

function Example() {
  // useState zwraca tablice z 2 elementami
  // zmienną 'count' i funkcją 'setCount':
  const [count, setCount] = useState(0);

  return (
    <div>
      {/* Możemy odwołać się do zmiennej stanowej: */}
      <p>You clicked {count} times</p>

      {/* Możemy wywołać funkcję setter setCount: */}
      <button onClick={() => setCount(count + 1)}>
        Click me
      </button>
    </div>
  );
}
```

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.

`const [aktualnaWartosc, funkcjaZmianyStanu] = useState(wartoscPoczatkowa)`

Zagnieżdżanie 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** ;

```
var Section = (props) => <section>
  <h1>{ section.title }</h1>
  {props.children}
</section>;
```

```
React.render(<Section title="Zagnieżdżony Todo">
  Dowolna treść, także komponenty <Todo />
</Section>
, document.getElementById('app'));
```

Formularze

Dodanie do pola formularza atrybutu `value=""` zamienia to pole w tzw. pole kontrolowane, czyli pole którego stan jest powiazany 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 sposoby na zmianę wartości tego pola to zmiana stanu komponentu, czyli skorzystanie z metody `setState`:

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



```
handleChange = e => this.setState({myValue:e.target.value })
```

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

Router

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

```
import React from 'react'
import { BrowserRouter as Router, Route, Link } from 'react-router-dom'
import { Home, About, Topics } from './my-components'

const Routing = () => <Router>
  <div>
    <Route exact path="/" component={Home}/>
    <Route path="/about" component={About}/>
    <Route path="/topics" component={Topics}/>
  </div>
</Router>
```

Routing jest komponentem, renderujemy go w dowolnym miejscu w kodzie JSX

Linkowanie

Router pozwala tworzyć parametryzowane ścieżki oraz dynamiczne linki:

```
<ul>
  <li><Link to="/netflix">Netflix</Link></li>
  <li><Link to="/zillow-group">Zillow Group</Link></li>
  <li><Link to="/yahoo">Yahoo</Link></li>
  <li><Link to={some.dynamic.id}>Dynamic link with some {some.dynamic.id}</Link></li>
</ul>
```

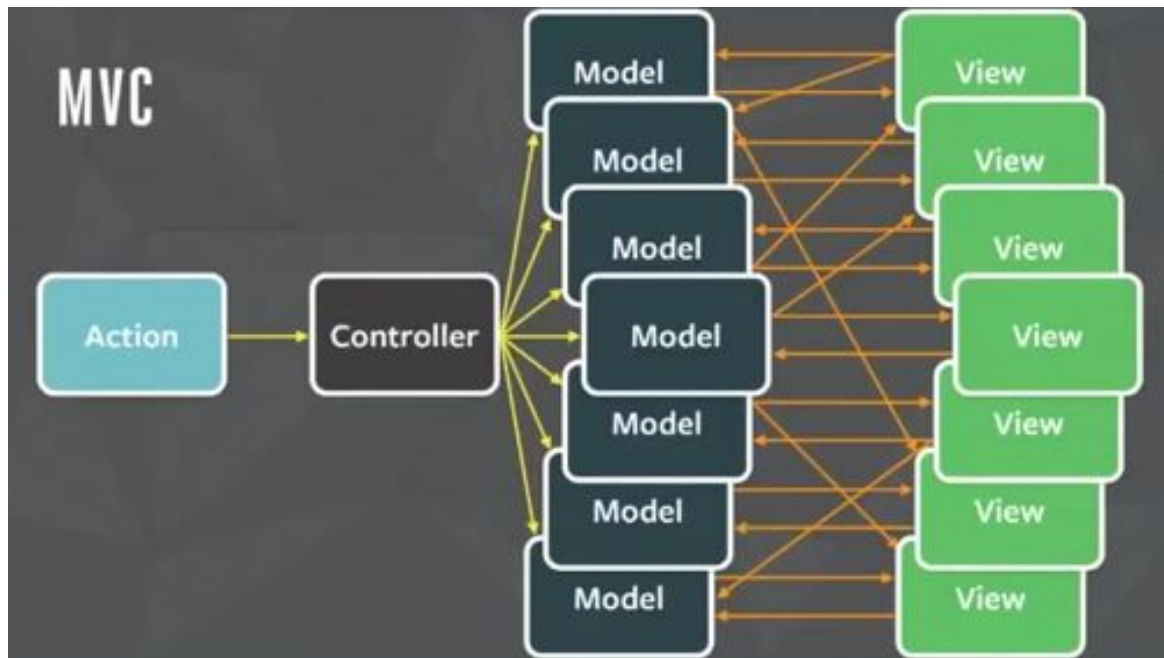
```
<Route path="/:id" component={Child}/>
```

```
const Child = ({ match }) => (
  <div>
    <h3>ID: {match.params.id}</h3>
  </div>
)
```



Flux

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.

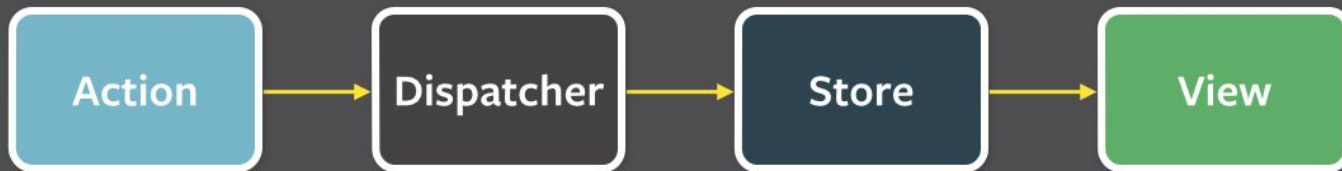
Trudno jest określić dokładnie kierunek przepływu danych.

Łatwo o błędy.

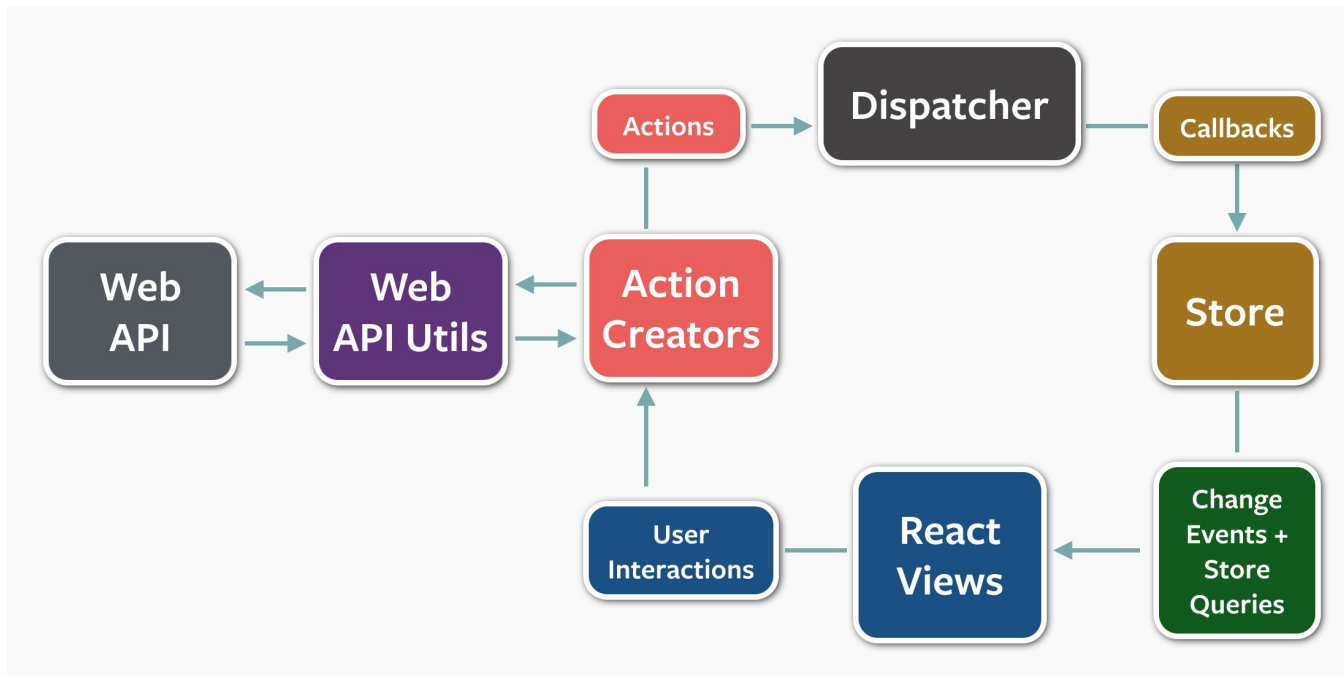
Flux

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 “jedynym 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



Flux w praktyce



Akcje

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

```
const ADD_FLIGHT = {  
  type: 'ADD_FLIGHT',  
  payload: {  
    id: 3542,  
    name: 'WAW/SFO'  
  }  
}
```

```
const REMOVE_FLIGHT = {  
  type: 'REMOVE_FLIGHT',  
  payload: {  
    id: 3542  
  }  
}
```

```
const UPDATE_FLIGHT = {  
  type: 'UPDATE_FLIGHT',  
  payload: {  
    id: 3542,  
    name: 'WAW/SFO',  
    status: 'active'  
  }  
}
```

Dyspozytor

Akcje są obiektami zawierającymi przynajmniej 2 pola:

```
var flightDispatcher = new Dispatcher();
```

```
// Magazyn przechowujący informacje o lotach
```

```
var FlightsStore = {flights: []};
```

```
// Rejestrujemy funkcję odpowiedzialną za aktualizowanie Magazynu
```

```
flightDispatcher.register( flightStoreActionHandler )
```

```
// Dyspozycja akcji przekaże ją do zarejestrowanych magazynów, które zaktualizują stan
```

```
// a następnie poinformują widoki o zmianie stanu
```

```
flightDispatcher.dispatch({  
  type: 'ADD_FLIGHT', payload: { id: 3542, name: 'WAW/SFO' }  
})
```

Magazyn

Magazyn składa się z 2 części:

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

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

```
class FlightStore extends EventEmitter{
  constructor(){
    this.store = { flights: [] }
  }
  handleAction(action){
    switch(action.type){
      case 'ADD_FLIGHT':
        this.store.flights.push( action.payload )
        break;
    }
    this.notifyViews( this.state );
  }
  ...
}
```



Redux

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

```
let state = null;

function dispatch(action) {
  const newState = reducer(state, action);

  if (newState !== state) {
    state = newState;
    reRenderViews(state)
  }else{
    // brak zmian - nie aktualizujemy
  }
}
```

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

```
import { createStore } from 'redux'

function counter(state = 0, action) {
  switch (action.type) {
    case 'INCREMENT':
      return state + 1
    case 'DECREMENT':
      return state - 1
    default:
      return state
  }
}

let store = createStore(counter)
store.subscribe(() => console.log(store.getState()))

store.dispatch({ type: 'INCREMENT' }) // 1
store.dispatch({ type: 'INCREMENT' }) // 2
store.dispatch({ type: 'DECREMENT' }) // 1
```


Reducers

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

```
function reducer(state, action) {  
  switch (action) {  
    case 'INC':  
      return {...state, counter: state.counter + 1 };  
    case 'DEC':  
      return {...state, counter: state.counter - 1 };  
    default:  
      return state;  
  }  
}
```

Nie modyfikuj obiektów bezpośrednio: ~~state.counter = state.counter + 1~~

Action Creators

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

```
const addIngredient = (recipe, name, quantity) => ({  
  type: 'ADD_INGREDIENT',  
  recipe,  
  name,  
  quantity  
});
```

W ten sposób dużo wygodniej możemy wprowadzać zmiany stanu aplikacji:

```
store.dispatch(addIngredient('Omelette', 'Eggs', 3));
```

Zagnieżdżanie reduktorów

Reduktory możemy zagnieżdżać:

```
const recipesReducer = (recipes, action) => {
  switch (action.type) {
    case 'ADD_RECIPE':
      return recipes.concat({
        name: action.name
      });
  }
  return recipes;
};
```

```
const ingredientsReducer = (ingredients, action) => {...}
```

```
const rootReducer = (state, action) => {
  return Object.assign({}, state, {
    recipes: recipesReducer(state.recipes, action),
    ingredients: ingredientsReducer(state.ingredients, action)
  })
}
```

... lub “kombinować” :

```
export default combineReducers({
  recipes: recipesReducer,
  ingredients: ingredientsReducer
});
```

Middleware

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

```
import { createStore, applyMiddleware } from 'redux';
import rootReducers from 'reducers/root';

const loggingMiddleware = ({ getState, dispatch }) => (next) => (action) => {
  console.log(`Action: ${ action.type }`, action);
  next(action);
};

const initialState = {...};

export default createStore(
  rootReducers,
  initialState,
  applyMiddleware(loggingMiddleware)
);
```

Efekty uboczne

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

```
function fetchData(url, callback) {
  fetch(url)
    .then((response) => {
      if (response.status !== 200) {
        console.log(`Error fetching data: ${ response.status }`);
      } else {
        response.json().then(callback);
      }
    })
    .catch((err) => console.log(`Error fetching data: ${ err }`))
}

const apiMiddleware = ({ dispatch }) => next => action => {
  if (action.type === FETCH_MY_DATA) {
    fetchData(URL, data => dispatch(setMyData(data)));
  }

  next(action);
};
```

Popularne middleware:

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

Redux - React

```
import { connect } from 'react-redux'

const mapStateToProps = (state) => {
  return {
    todos: state.todos
  }
}

const mapDispatchToProps = (dispatch) => {
  return {
    onTodoClick: (id) => {
      dispatch(toggleTodo(id))
    }
  }
}

const VisibleTodoList = connect(
  mapStateToProps,
  mapDispatchToProps
)(TodoList)
```

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

```
import { Provider } from 'react-redux'

let store = createStore(todoApp)

render(
  <Provider store={store}>
    <App />
  </Provider>,
  document.getElementById('root')
)
```

Normalizacja

```
{
  posts : {
    byId : {
      "post1213" : {
        id : "post1",
        author : "user1",
        body : ".....",
        comments : ["comment1", "comment2"]
      },
      ...
    }
  },
  allIds : ["post1213", "post2425", ...]
},
comments : {
  byId : {
    "comment1675" : {
      id : "comment1",
      author : "user2",
      comment : ".....",
    },
    ...
  },
  allIds : ["comment1675", "comment254", ...]
},
...
}
```

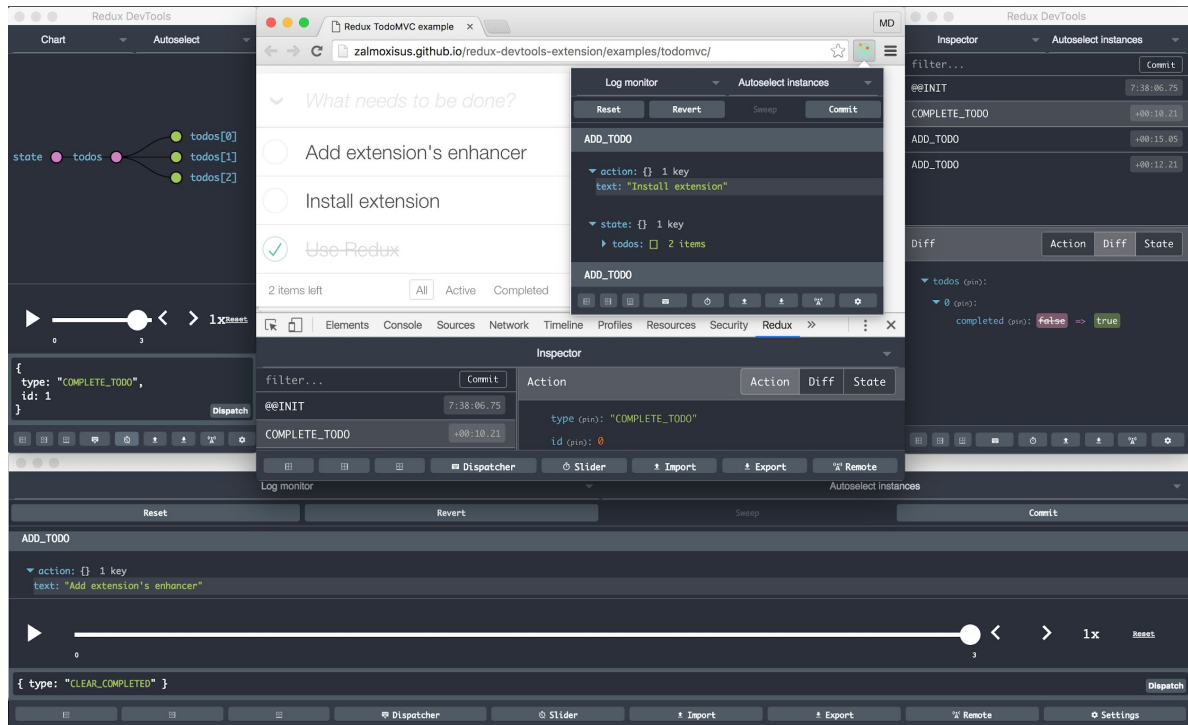
Normalizacja oznacza, że każdy element danych przechowywany jest w Magazynie Redux zawsze tylko i wyłącznie raz.

Jeżeli obiekt występuje w kilku miejscach w aplikacji wystarczy przechować jego identyfikator i typ.

Następnie przy renderowaniu można odtworzyć wszystkie wystąpienia używając identyfikatorów:

```
posts.allIds.map( id => posts.byId[id] )
```

Redux Devtools





Dziękuję za uwagę!

Pytania? ;-)