



create-react-app

ŚRODOWISKO DEWELOPERSKIE DLA REACT
GOTOWA KONFIGURACJA I PODSTAWOWY PROJEKT

optymalizacja projektu i kodu oraz nowe rozwiązania w projekcie



HOME | ABOUT | DOWNLOADS | DOCS | GET INVOLVED | SECURITY | NEWS | FC

Node.js® is a JavaScript runtime built on [Chrome's V8 JavaScript engine](#).

Download for Windows (x64)

10.15.0 LTS

Recommended For Most Users

11.6.0 Current

Latest Features

[Other Downloads](#) | [Changelog](#) | [API Docs](#) [Other Downloads](#) | [Changelog](#) | [API Docs](#)

Or have a look at the [Long Term Support \(LTS\) schedule](#).

Sign up for [Node.js Everywhere](#), the official Node.js Monthly Newsletter.

Zaczynamy od node (+npm), terminala i polecenia

It sets up your development environment so that you can use the latest JavaScript features, provides a nice developer experience, and optimizes your app for production. You'll need to have Node ≥ 6 and npm ≥ 5.2 on your machine. To create a project, run:

```
npx create-react-app my-app  
cd my-app  
npm start
```

PROBLEMS | OUTPUT | DEBUG CONSOLE | TERMINAL

1: powershell



Windows PowerShell

Copyright (C) Microsoft Corporation. All rights reserved.

```
PS C:\Users\iMikser\Desktop\UdemyReactKurs\Kurs\Sekcja4_cra\example> npx create-react-app to_do_app
```

create-react-app - CLI i interpretery wiersza poleceń

CLI (Command Line Interface) - interfejs wiersza poleceń

```
PS C:\Users\iMikser\Desktop\UdemyReactKurs\Kurs\Sekcja4_cra\example> npx create-react-app to_do_app
```

`npx create-react-app .`
//instalacja w katalogu, którym jesteśmy

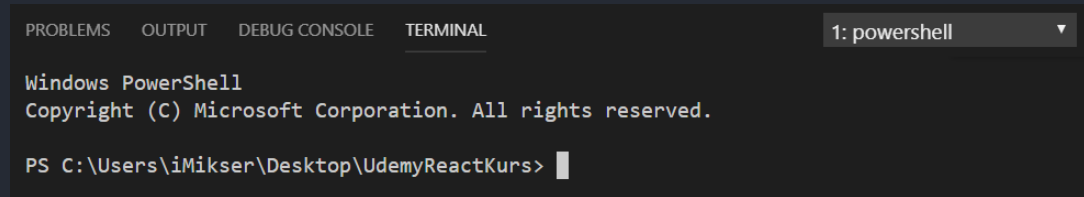
`cd nazwa-katalogu`
//wejście do katalogu

`cd..`
//poziom wyżej (wyjście z katalogu)

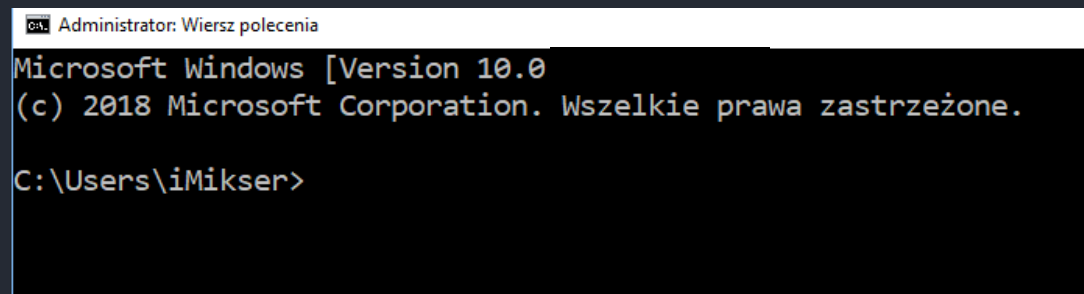
`cls (clear)`
//czyszczenie konsoli

`[strzała w górę]`
// cofnięcie się do poprzedniej komendy

Terminal z Windows Powershell (Bash w Mac i Linux)



CMD (wierz polecenia w Windows)



create-react-app - instalacja 3 pakietów

```
PS C:\Users\iMikser\Desktop\UdemyReactKurs\Kurs\Sekcja4_cra\example> npx create-react-app to_do_app

Creating a new React app in C:\Users\iMikser\Desktop\UdemyReactKurs\Kurs\Sekcja4_cra\example\to_do_app.

Installing packages. This might take a couple of minutes.
Installing react, react-dom, and react-scripts...
```

CRA instaluje bibliotekę React, bibliotekę React DOM i dodatkowe paczki wraz z konfiguracją (mnóstwo innych rzeczy)

```
Installing packages. This might take a couple of minutes.
Installing react, react-dom, and react-scripts...

+ react-dom@16.7.0
+ react@16.7.0
+ react-scripts@2.1.2
added 1726 packages from 667 contributors and audited 35709 packages
```

react-scripts ... (Obecna wersja CRA > 2)

Pod pozycją react-scripts w dependencies w pliku `package.json` kryje się konfiguracja projektu i mnóstwo paczek (m.in. Babel czy Webpack), które znajdziemy w katalogu `node_modules`.

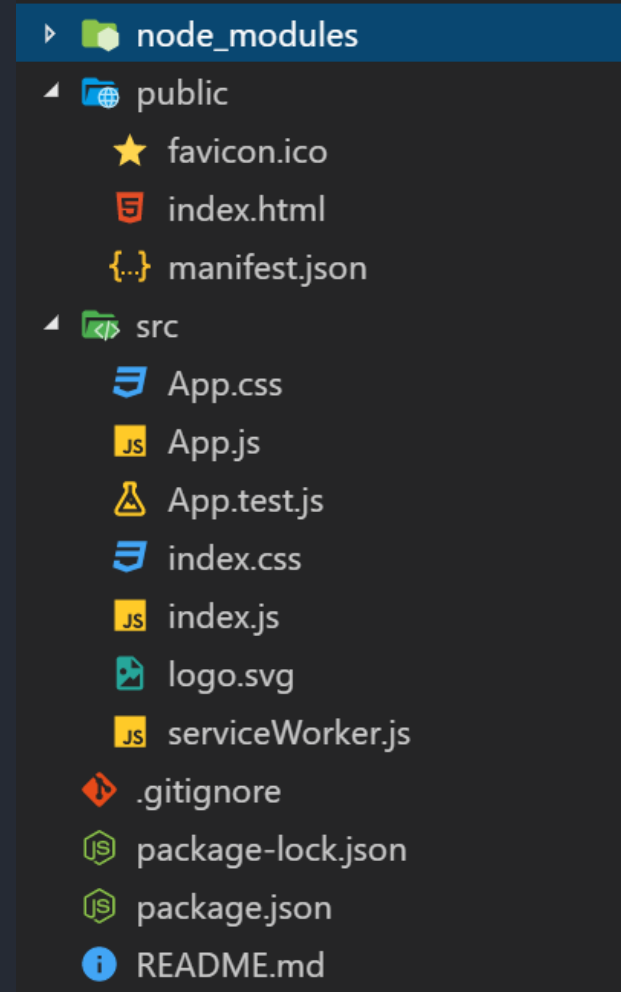
To właśnie ten element odpowiada za przygotowanie środowiska developerskiego dla naszego projektu.

```
"dependencies": {  
  "react": "^16.6.0",  
  "react-dom": "^16.6.0",  
  "react-scripts": "2.1.1"  
},
```

create-react-app - przykładowa aplikacja

Powstaje przykładowa aplikacji ze stworzoną (sugerowaną) strukturą kodu.

Ta struktura i zawartość plików w kolejnych wersjach CRA delikatnie się zmieniała i może też zmieniać się w przyszłości.



create-react-app - uruchomienie serwera i aplikacji

```
PS C:\Users\iMikser\Desktop\UdemyReactKurs\Kurs\Sekcja4_cra\example> cd to_do_app
PS C:\Users\iMikser\Desktop\UdemyReactKurs\Kurs\Sekcja4_cra\example\to_do_app> npm start
```

Starting the development server...



Edit src/App.js and save to reload.

[Learn React](#)

create-react-app - co to jest

Create React App is a comfortable environment for learning React, and is the best way to start building a new single-page application in React.

It sets up your development environment so that you can use the latest JavaScript features, provides a nice developer experience, and optimizes your app for production.

Your environment will have everything you need to build a modern single-page React app.

-- dokumentacja React

create-react-app - gotowe środowisko programistyczne

Nie musimy nic konfigurować (ale możemy). Używane bardzo często przy produkcji aplikacji. Środowisko jest oparte na:

- development server - lokalny serwer deweloperski
- Webpack (bundling/pakietowanie)
- Babel (kompilowanie JSX i ES od wersji 6 kompilowane do 5)
- ESLint (sprawdzanie kodu)
- możliwość doinstalowania dodatkowych paczek (np. bootstrap, jQuery itd.)

To prawda, że całe takie środowisko można przygotować/skonfigurować samemu od zera, ale najczęściej nie ma to sensu. CRA pozwala się skupić na tym co ważne - na naszej aplikacji.

create-react-app - lokalny serwer deweloperski (Webpack Dev Server)

npm start

localhost:3000 - adres pod którym domyślnie uruchomi się nasza aplikacja.

Tworzenie uproszczonej produkcyjnej wersji aplikacji w pamięci (łączenie plików do bundle.js i umieszczanie w pliku html). Ten plik jest tworzony po każdej zmianie.

Czasami potrafi trochę przymulić ;)

```
    <title>React App</title>
  .. <style type="text/css"> == $0
    body {
      margin: 0;
      padding: 0;
      font-family: -apple-system, BlinkMacSystemFont, "Segoe UI", "Rob
        "Ubuntu", "Cantarell", "Fira Sans", "Droid Sans", "Helvetica N
        sans-serif;
      -webkit-font-smoothing: antialiased;
      -moz-osx-font-smoothing: grayscale;
    }

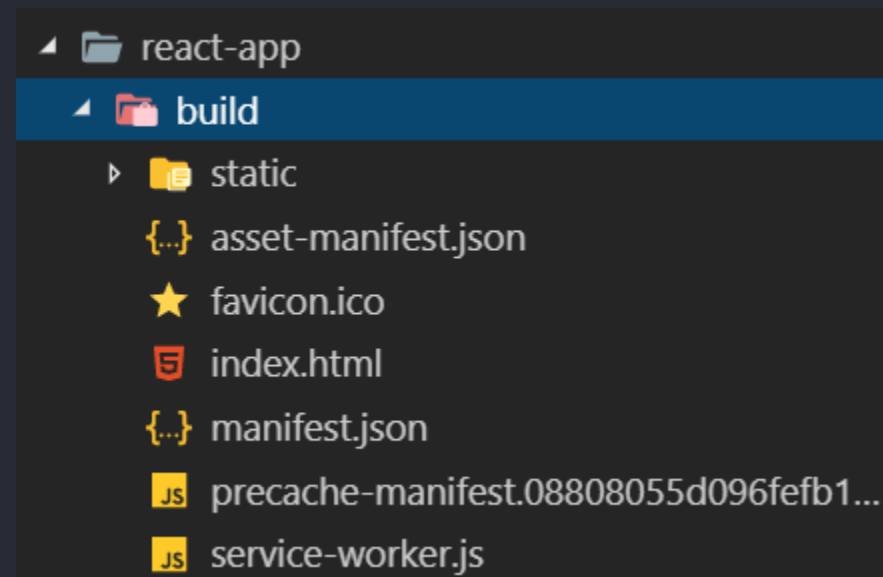
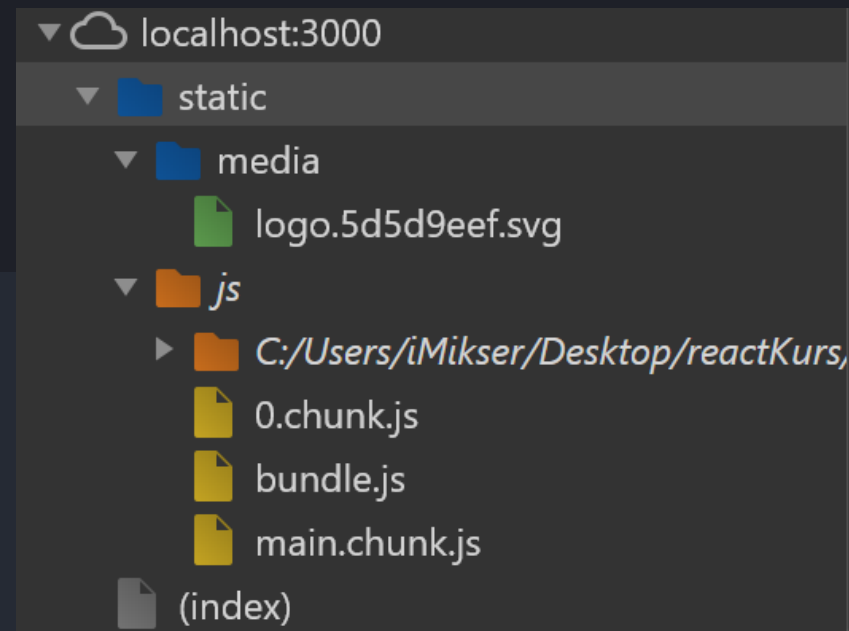
    code {
      font-family: source-code-pro, Menlo, Monaco, Consolas, "Courier
        monospace;
    }
  </style>
  ▶ <style type="text/css">...</style>
  </head>
```

Do pliku index.html zostaje dodane przede wszystkim:

```
<script src="/static/js/bundle.js"></script>
<script src="/static/js/0.chunk.js"></script>
<script src="/static/js/main.chunk.js"></script>
```

create-react-app - Webpack

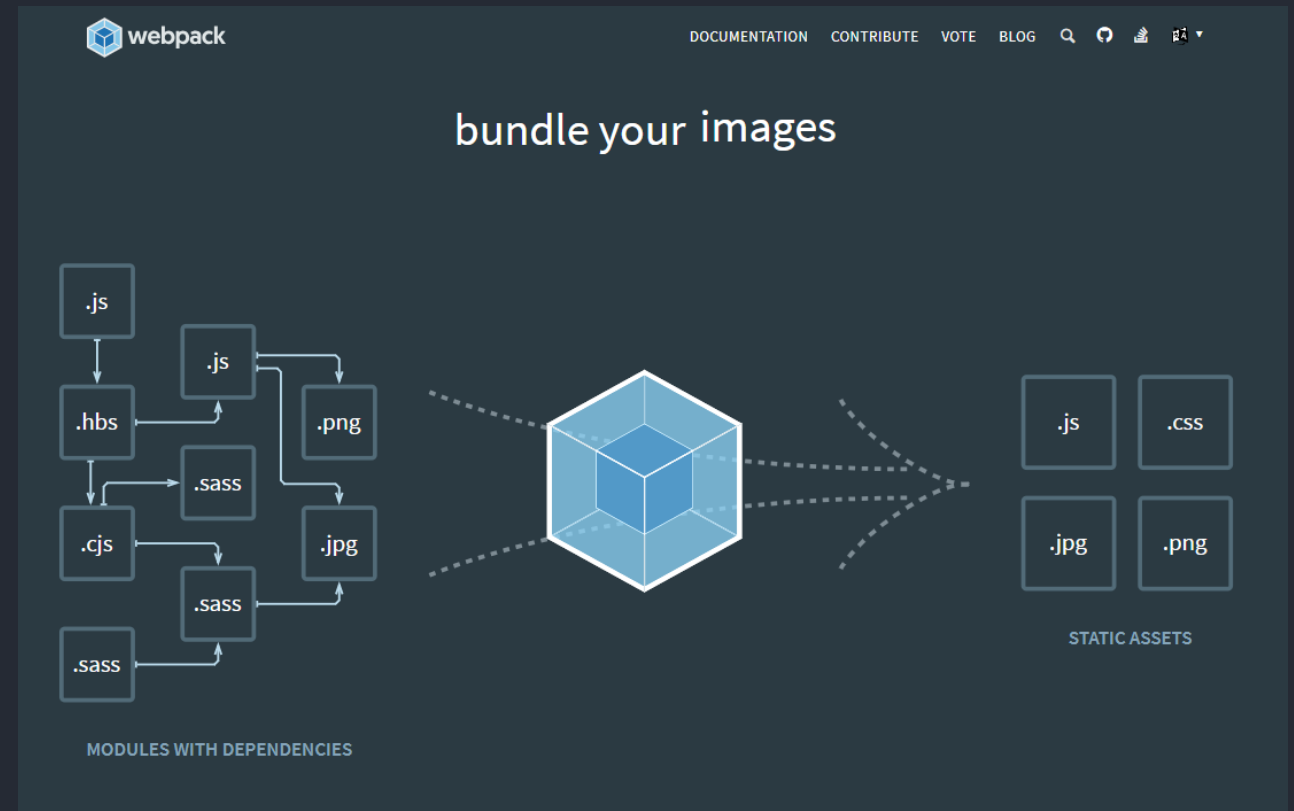
Ten lokalny serwer, łączenie plików (bundling, to podstawowe zadanie webpacka), możliwość użycia modułów, zarządzanie rozszerzeniami wewnątrz aplikacji, tworzenie produkcyjnej (gotowej do publikacji) wersji aplikacji (polecenie `npm run build` utworzy folder /build) - za wszystko odpowiedzialny w naszym środowisku CRA jest **Webpack**.



create-react-app - Webpack

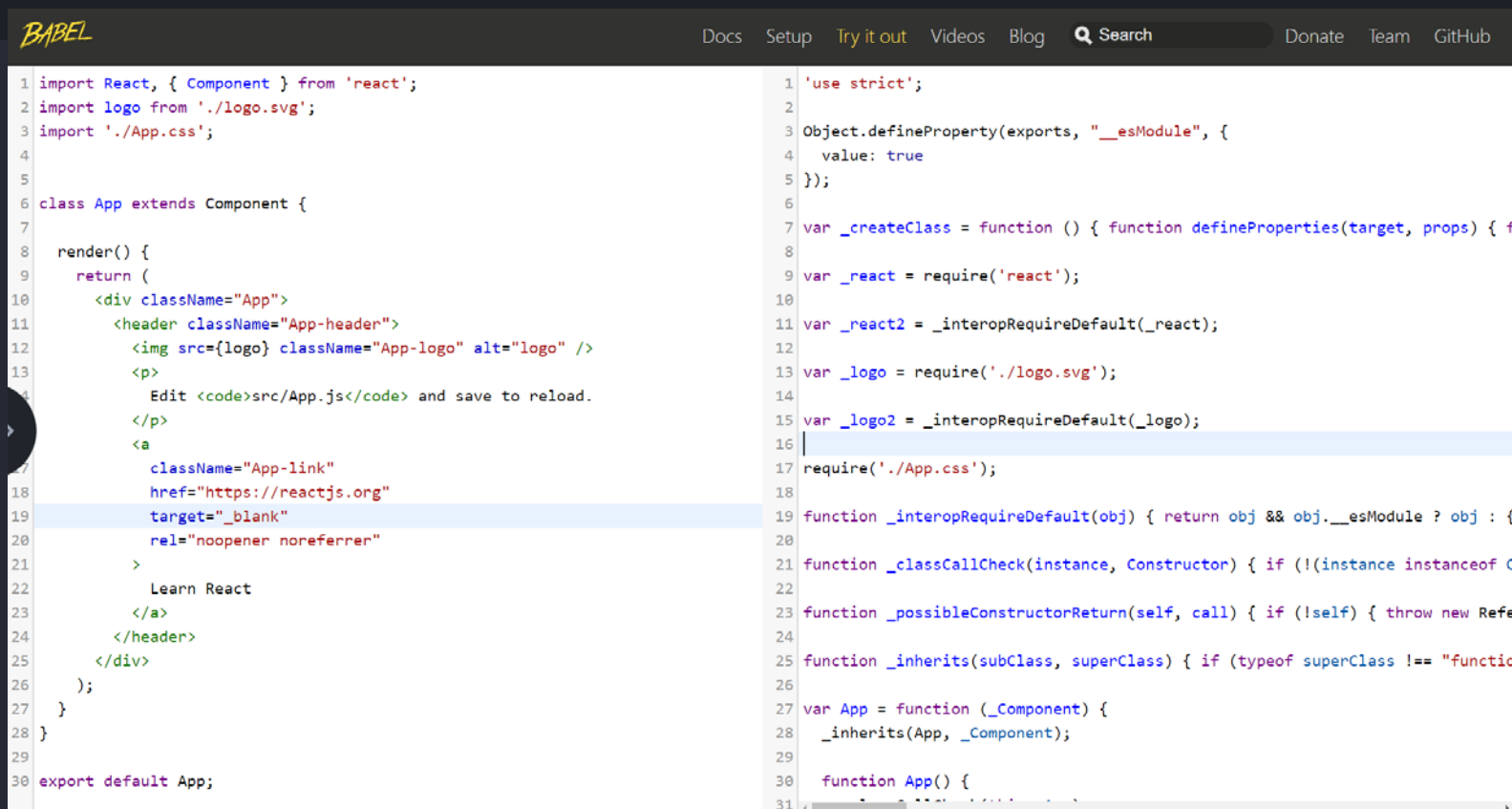
Webpack jest fundamentem środowiska tworzonego przez CRA, ale oczywiście może być i jest używany poza CRA i poza React.

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



create-react-app - Babel

Zmienia kod napisany w ES6 (i późniejszych specyfikacja) oraz kod napisany z użyciem składni JSX na ES5.



The screenshot displays the Babel REPL interface, which is used for transforming code. The left pane shows the input code, and the right pane shows the output code after transformation.

Input Code (Left Pane):

```
1 import React, { Component } from 'react';
2 import logo from './logo.svg';
3 import './App.css';
4
5
6 class App extends Component {
7
8   render() {
9     return (
10       <div className="App">
11         <header className="App-header">
12           <img src={logo} className="App-logo" alt="logo" />
13           <p>
14             Edit <code>src/App.js</code> and save to reload.
15           </p>
16           <a
17             className="App-link"
18             href="https://reactjs.org"
19             target="_blank"
20             rel="noopener noreferrer"
21           >
22             Learn React
23           </a>
24         </header>
25       </div>
26     );
27   }
28 }
29
30 export default App;
```

Output Code (Right Pane):

```
1 'use strict';
2
3 Object.defineProperty(exports, "__esModule", {
4   value: true
5 });
6
7 var _createClass = function () { function defineProperties(target, props) { f
8
9 var _react = require('react');
10
11 var _react2 = _interopRequireDefault(_react);
12
13 var _logo = require('./logo.svg');
14
15 var _logo2 = _interopRequireDefault(_logo);
16
17 require('./App.css');
18
19 function _interopRequireDefault(obj) { return obj && obj.__esModule ? obj : {
20
21 function _classCallCheck(instance, Constructor) { if (!(instance instanceof C
22
23 function _possibleConstructorReturn(self, call) { if (!self) { throw new Refe
24
25 function _inherits(subClass, superClass) { if (typeof superClass !== "function
26
27 var App = function (_Component) {
28   _inherits(App, _Component);
29
30   function App() {
31
```

create-react-app - ESLint









Łatwiej znajdziemy problemy!

Lintery to programy/dodatki do monitorowania (online) i analizowania kodu. Create-react-app dostarcza nam skonfigurowany ESLint, czyli jeden z najpopularniejszych analizatorów, który ostrzega nas przed złymi praktykami (złamaniem zasad, również określonych przez programistę) i błędami w kodzie.

ESLint analizuje nasz kod JavaScript (ES, JSX)

ECMAScript 6

These rules relate to ES6, also known as ES2015:

	<code>arrow-body-style</code>	require braces around arrow function bodies
	<code>arrow-parens</code>	require parentheses around arrow function arguments
	<code>arrow-spacing</code>	enforce consistent spacing before and after the arrow in arrow functions
✓	<code>constructor-super</code>	require <code>super()</code> calls in constructors
	<code>generator-star-spacing</code>	enforce consistent spacing around <code>*</code> operators in generator functions
✓	<code>no-class-assign</code>	disallow reassigning class members
	<code>no-confusing-arrow</code>	disallow arrow functions where they could be confused with comparisons
✓	<code>no-const-assign</code>	disallow reassigning <code>const</code> variables
✓	<code>no-dupe-class-members</code>	disallow duplicate class members
	<code>no-duplicate-imports</code>	disallow duplicate module imports
✓	<code>no-new-symbol</code>	disallow <code>new</code> operators with the <code>Symbol</code> object
	<code>no-restricted-imports</code>	disallow specified modules when loaded by <code>import</code>
✓	<code>no-this-before-super</code>	disallow <code>this</code> / <code>super</code> before calling <code>super()</code> in constructors
	<code>no-useless-computed-key</code>	disallow unnecessary computed property keys in object literals
	<code>no-useless-constructor</code>	disallow unnecessary constructors
	<code>no-useless-rename</code>	disallow renaming import, export, and destructured assignments to the same name
	<code>no-var</code>	require <code>let</code> or <code>const</code> instead of <code>var</code>

create-react-app - ESLint

Przydatne wsparcie w znajdowaniu błędów. Przykładowe błędy/problemy

JS App.js react-app\src 2

✖ [eslint] Unexpected use of 'name'. [no-restricted-globals] (12, 21)

⚠ [eslint] 'logo' is defined but never used. [no-unused-vars] (2, 8)

JS App.js my-new-app-react\src 1

⚠ [eslint] 'div' is assigned a value but never used. [no-unused-vars] (12, 11)

50



51

</>

PROBLEMS

1

OUTPUT

DEBUG CONSOLE

TERMINAL

Filter. Eg: text, **/*.ts, !**/node_modules/**

JS App.js timekeeper\src\components 1

⚠ [eslint] img elements must have an alt prop, either with meaningful text, or an empty string for decorative images. [jsx-a11y/alt-text] (50, 9)

create-react-app - w praktyce, dlaczego warto.

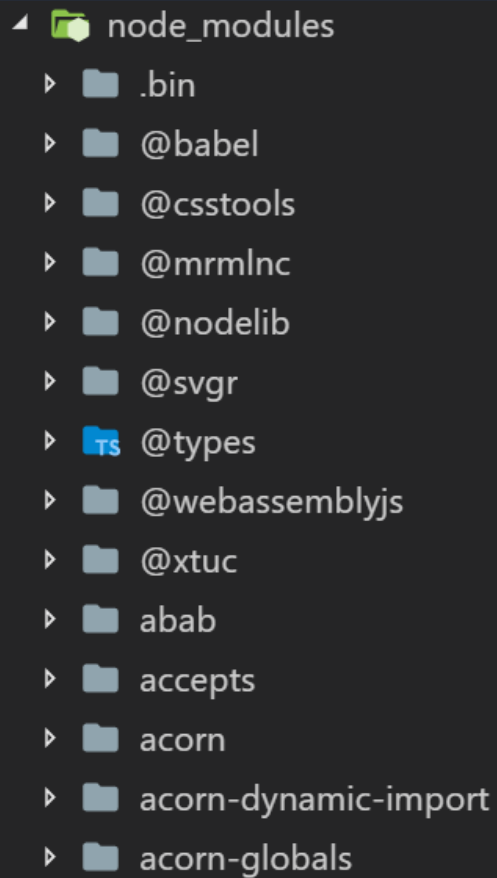
- nie potrzeba dodatkowej konfiguracji
- gotowa struktura dla SPA
- wsparcie dla modułów JavaScript
- wsparcie dla składni React, JSX, ES6
- Autoprefiksy CSS (nie potrzeba używać -webkit- i innych), wsparcie dla Sass
- dostęp do olbrzymiej ilości bibliotek (npm)
- serwer deweloperski
- optymalizowanie i kompilowanie JS, CSS, i obrazków do wersji produkcyjnej (katalog /build)

... i wiele wiele mniej i bardziej istotnych rzeczy na które natrafisz jeśli będzie w przyszłości pracował z React (na przykład testy czy Git)

create-react-app - struktura

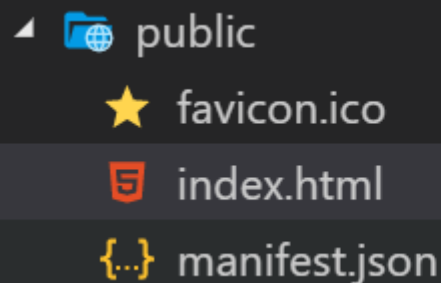
- ▶ node_modules
- ▲ public
 - ★ favicon.ico
 - index.html
 - {...} manifest.json
- ▲ src
 - App.css
 - App.js
 - App.test.js
 - index.css
 - index.js
 - logo.svg
 - serviceWorker.js
- .gitignore
- package-lock.json
- package.json
- README.md

/node-modules

- 
- node_modules
 - .bin
 - @babel
 - @csstools
 - @mrmlnc
 - @nodelib
 - @svgr
 - @types
 - @webassemblyjs
 - @xtuc
 - abab
 - accepts
 - acorn
 - acorn-dynamic-import
 - acorn-globals

W tym miejscu przechowywane są wszystkie paczki. Jeśli dodamy jakąś nową paczkę (np. bibliotekę jQuery), to ona również zostanie umieszczona w katalogu node_modules.

/public/index.html



```
public
  favicon.ico
  index.html
  manifest.json
```

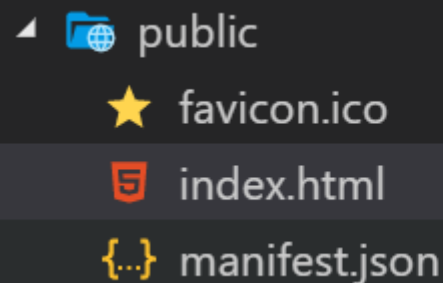
Zwróć uwagę, że w kodzie pliku index.html nie ma żadnych `<script>` czy `css`-a. Nie są dodawane tutaj też bezpośrednie biblioteki (choć oczywiście mogą być).

Jednak po kompilacji i łączeniu (bundling), która nastąpi po uruchomieniu na serwerze lokalnym (czy po przygotowaniu wersji produkcyjnej) ten plik będzie wyglądał już nieco inaczej (dodaje `css` czy bundlowane moduły).

```
<title>React App</title>
</head>
<body>
  <noscript>You need to enable JavaScript
  to run this app.</noscript>
  <div id="root"></div>
  <!--
    This HTML file is a template.
    If you open it directly in the browser,
    you will see an empty page.

    You can add webfonts, meta tags, or
    analytics to this file.
```

/public/manifest.json



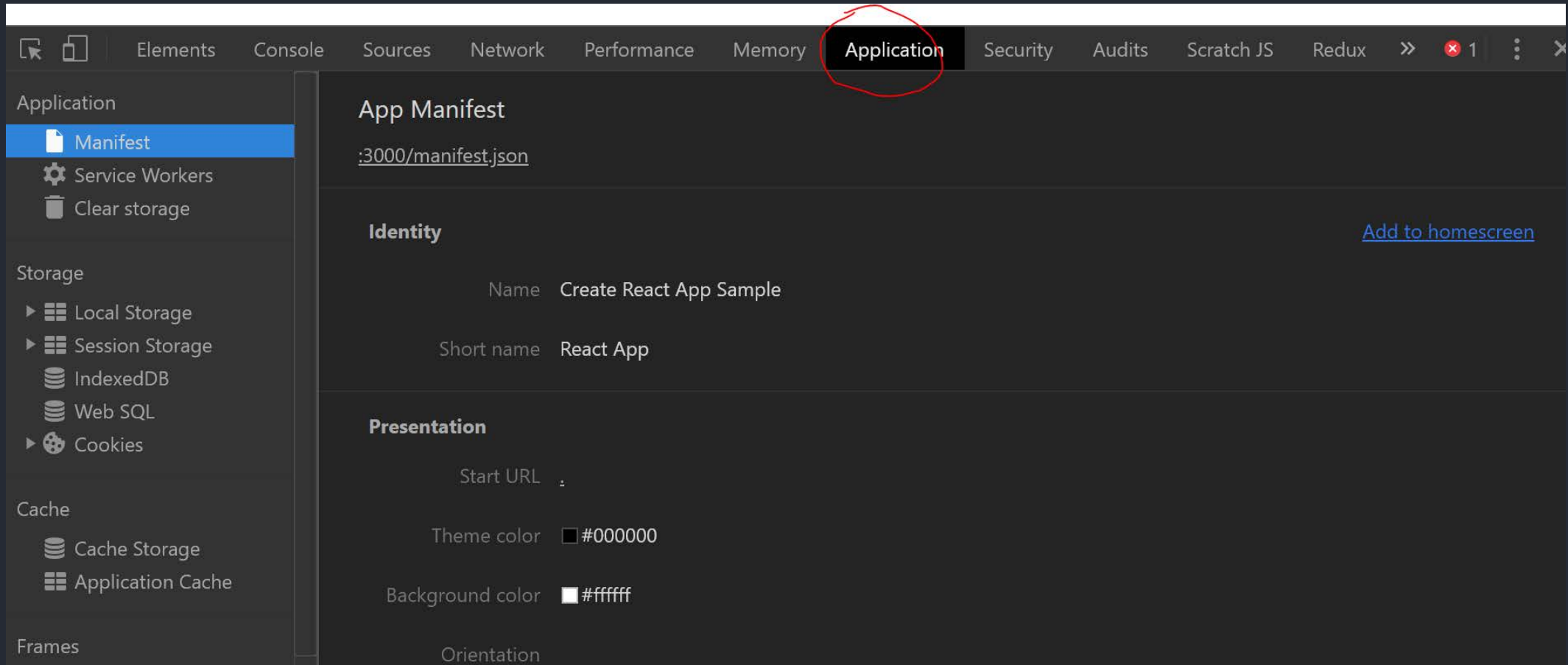
Plik w formacie json, informujący przeglądarkę jak ma zachować się po dodaniu (zainstalowaniu) aplikacji do ekranu głównego (mobile i desktop).

W wersji którą widzimy tutaj to nazwa, ikona, adres początkowy, color tła przed wczytaniem aplikacji, color elementów przeglądarki jak pasek (theme-color) wreszcie po otwarciu aplikacji nie chcemy by pojawił się pasek (display: standalone)

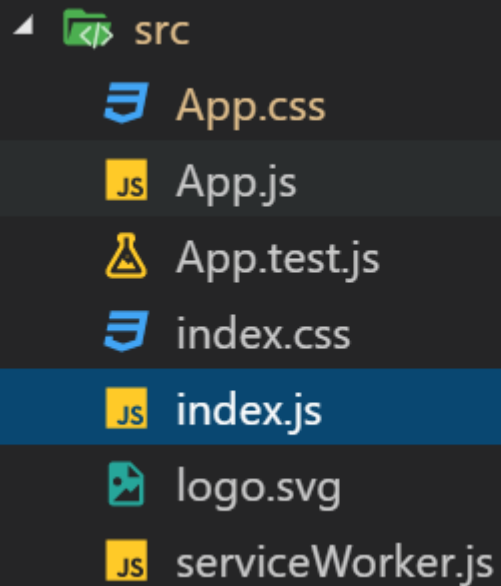
```
<link rel="manifest"
href="%PUBLIC_URL%/manifest.json" />
```

```
{
  "short_name": "React App",
  "name": "Create React App Sample",
  "icons": [
    {
      "src": "favicon.ico",
      "sizes": "64x64 32x32 24x24 16x16",
      "type": "image/x-icon"
    }
  ],
  "start_url": ".",
  "display": "standalone",
  "theme_color": "#000000",
  "background_color": "#ffffff"
}
```

/public/manifest.json



/src/index.js



```
src
├── App.css
├── App.js
├── App.test.js
├── index.css
├── index.js
├── logo.svg
└── serviceWorker.js
```

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

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

/src



katalog jest bundlowany (odpowiada za to Webpack). Bundlowanie (pakietowanie), to robienie jednego czy kilku plików z bardzo wielu. Przy okazji - katalogu /public nie jest bundlowany.

/src/serviceWorker.js

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

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

```
// If you want your app to work offline and load faster, you can change
// unregister() to register() below. Note this comes with some pitfalls.
// Learn more about service workers: http://bit.ly/CRA-PWA
```

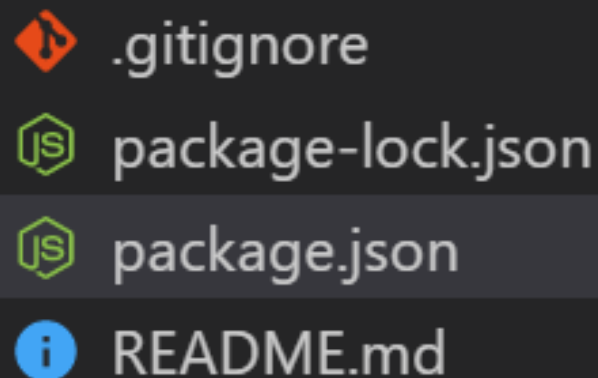
```
serviceWorker.unregister();
```

Związane z dostępem do aplikacji w trybie offline (gdy użytkownik nie jest podpięty do sieci). Domyślnie wyłączone.

Dotyczy sytuacji gdy tworzymy PWA czyli Progressive Web App tj. stronę (aplikację) internetową, która przypomina aplikacje natywne - są, choćby w podstawowym zakresie, dostępne offline i można w nich integrować choćby notyfikacje (push notification).

PWA wymaga service workera, manifestu i https.

package.json



plik z ustawieniami (konfiguracją) naszego projektu. Przede wszystkim widzimy tu zainstalowane paczki (zależności).

W skryptach skróty, których możemy użyć w terminalu np. npm start czy npm run build.

```
{
  "name": "app-english",
  "version": "0.1.0",
  "private": true,
  "dependencies": {
    "react": "^16.7.0",
    "react-dom": "^16.7.0",
    "react-scripts": "2.1.2"
  },
  "scripts": {
    "start": "react-scripts start",
    "build": "react-scripts build",
    "test": "react-scripts test",
    "eject": "react-scripts eject"
  },
  "eslintConfig": {
    "extends": "react-app"
  },
  "browserslist": [
    ">0.2%",
    "not dead",
    "not ie <= 11",
    "not op_mini all"
  ]
}
```

.gitignore

Co Git (system kontroli wersji) ma ignorować, jakie katalogi/pliki (te które nie są potrzebne). Te elementy nie będą brały udziału przy operacji commit.

Zapraszam do osobnego (DARMOWEGO) kursu Git na Udemy.



.gitignore

```
# See https://help.github.com/articles/ignoring-files/ for more about
ignoring files.

# dependencies
/node_modules
/.pnp
.pnp.js

# testing
/coverage

# production
/build

# misc
.DS_Store
.env.local
.env.development.local
.env.test.local
```

To na start wystarczy...

Moduły w JavaScript (od ES6)

Bez modułów kod tak naprawdę nie jest do końca wyodrębniony (hermetyzacja nie jest pełna). Jeśli nie korzystamy z modułów, to jeden plik tworzy jeden zakres globalny w którym znajdują się zasięgi lokalne.

Moduł nie jest częścią zasięgu globalnego (sam w sobie tworzy niezależny zakres). Zapewnia to większe bezpieczeństwo, ale też pozwala zachować lepszą strukturę kodu.

By korzystać ze zmiennych, funkcji (itd) z jednego moduły w innym module trzeba je (lub cały moduł) eksportować i importować.

Moduły w środowisku CRA

O co chodzi, jak korzystać, przykłady

Zarysujemy tutaj tematykę modułów w takim stopniu w jakim pracujemy w środowisku create-react-app.

Biblioteki, paczki w module

By użyć czegoś w module musimy "to coś" do niego zaimportować.
(pliki css, grafiki, biblioteki, których użyjemy w module).

Zaimportować możemy jednak tylko to, co eksportowaliśmy w innym module.

Export modułów w React

//opcja 1

```
export default class MyComponent extends Component {  
    ...  
}
```

//opcja 2

```
class MyComponent extends Component {  
    ...  
}  
export default MyComponent
```

Export modułów w React - default

//opcja 1

```
export default class MyComponent extends Component {}
```

//opcja 2

```
export default MyComponent
```

Pojedyncza wartość (w module tylko jedna wartość może przypisana do default) może być nią np. klasa czy funkcja (w React więc najczęściej komponent).

Default oznacza, że jest to domyślnie eksportowany element z modułu. Dzięki temu prościej jest taki element importować.

import wartości domyślnej

```
//Moduł 1
```

```
export default App
```

```
//Moduł 2
```

```
import App from './App';
```

```
// import Counter from './App';
```

Nazwa jest dowolna (nie musi być taka sama jak przy eksporcie), ponieważ importowanie w ten sposób oznacza importowanie domyślnego elementu eksportowanego z modułu

Import modułów, pakietów

```
import React from 'react';  
import ReactDOM from 'react-dom';
```

```
import { render } from 'react-dom';  
//możemy wtedy zamiast ReactDOM.render(/.*..*/), po prostu render(/.*...*/)  
import { render as rdom } from 'react-dom';  
//wtedy nawet rdom()
```

```
//w jednej instrukcji  
import React, { Component, Fragment } from 'react';
```

```
//osobno  
import React from 'react';  
import {Component} from 'react';
```

Import - destruktywizacja?

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

//Wygląda jak destruktywizacja, ale nazywa się to [listą wiązań do zaimportowania](#). Dzięki temu możemy się do tych obiektów odwołać bezpośrednio.

Component, zamiast React.Component

```
class App extends Component
```

//Przykłady

```
import React from 'react';
```

```
import { Component } from 'react';
```

```
import { Fragment, Component } from 'react';
```

```
<Fragment>
```

```
    <div>...</div>
```

```
</Fragment>
```

Import - wiązanie

```
import React, { Component } from 'react';  
import $ from 'jquery'; //z pakietu (node_modules)  
//import jq from 'jquery'; //nazwa może być inna
```

```
import logo from '../img/logo.svg';  
import './Component/App.css';
```

```
import App from './App'; //nie jest to konieczne, ale dobrze by importowaniu modułów w React  
używać tych samych nazw co eksportowane klasy
```

Wiązanie działa jak deklaracja const. Nazwa jest zastrzeżona.

import biblioteki zewnętrznych

1. instalacja biblioteki w projekcie
2. import biblioteki w module
3. użycie biblioteki

import biblioteki zewnętrznych

1. instalacja biblioteki w projekcie (nie tylko React) - instrukcje w dokumentacji. Przykład jQuery

Downloading jQuery using npm or Yarn

jQuery is registered as [a package](#) on [npm](#). You can install the latest version of jQuery with the npm CLI command:

```
1 | npm install jquery
```

As an alternative you can use the [Yarn](#) CLI command:

```
1 | yarn add jquery
```

This will install jQuery in the `node_modules` directory. Within `node_modules/jquery/dist/` you will find an uncompressed release, a compressed release, and a map file.

Terminal

```
PS C:\Users\iMikser\Desktop\reactKurs\movie-app-api> npm install jquery
```

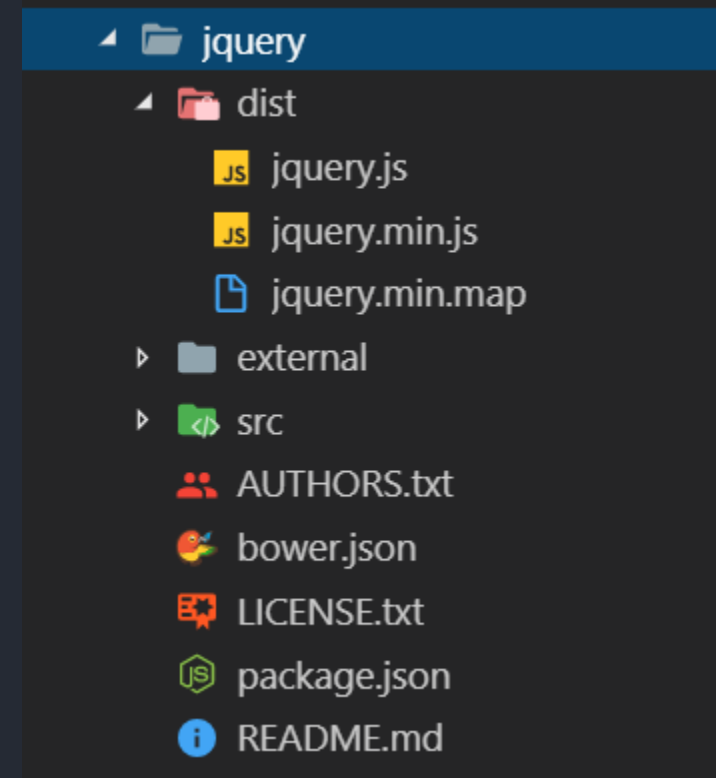
```
PS C:\Users\iMikser\Desktop\reactKurs\movie-app-api> npm install jquery@2
```

import biblioteki zewnętrznych

1. instalacja biblioteki w projekcie
2. import biblioteki w module

w katalogu node_modules
w package.json

```
"dependencies": {  
  "jquery": "^2.2.4",  
  "react": "^16.6.1",  
  "react-dom": "^16.6.1",  
  "react-scripts": "2.1.1"  
},  
"scripts": {  
  "start": "react-scripts start",  
  "build": "react-scripts build",  
  "test": "react-scripts test",  
  "eject": "react-scripts eject"  
}
```



import biblioteki zewnętrznych

1. instalacja biblioteki w projekcie
2. import biblioteki w module
3. użycie bibliteki

```
import $ from 'jquery'; //wielkość liter ma znaczenie
```

```
componentDidMount() {  
    $('div').css({ backgroundColor: "red" })  
}
```


Import modułów

importowanie defaultowego elementu z komponentu

```
import MyComponent from './MyComponent';
```

//nazwa dowolna

//miejsce wskazuje na moduł (w tym wypadku ten sam katalog)

Zwróćmy uwagę na to jak zbudowana jest ścieżka.

'MyComponent' - szukał by w pakietach

'./MyComponent' - szuka pliku (nie potrzeba dodawać .js)

Przejdźmy do kolejnego zagadnienia...

Pobieranie i wyświetlanie danych w React. Użycie metody `fetch` i pobranie danych za pomocą `API`.

Najpierw w kodzie projekt, potem trochę teorii.



Dane w React

Krótkie teoretyczne wprowadzenie (lub przypomnienie) dotyczące
AJAX, JSON, XMLHttpRequest, Promises, fetch(), API

*React nie interesuje się skąd są dane i za pomocą jakiego sposobu są pobierane.
React ma je dostać, przetworzyć, wykorzystać do tworzenia widoku i wyświetlić.*



AJAX

AJAX - czym jest

AJAX, czyli asynchroniczne żądanie do serwera (wysłanie/pobranie danych). Współcześnie pod tą nazwą kryją się różne techniki i technologie, oparte na zdarzeniach czy obietnicach (promises).

W AJAX można korzystać z różnych formatów danych. Początkowa AJAX opierała się przede wszystkim o format XML (stąd i nazwa Asynchronous JavaScript and XML) dziś takim podstawowym formatem jest JSON.

Ps. XML przypomina HTML i jest językiem znaczników (*Extensible Markup Language*)

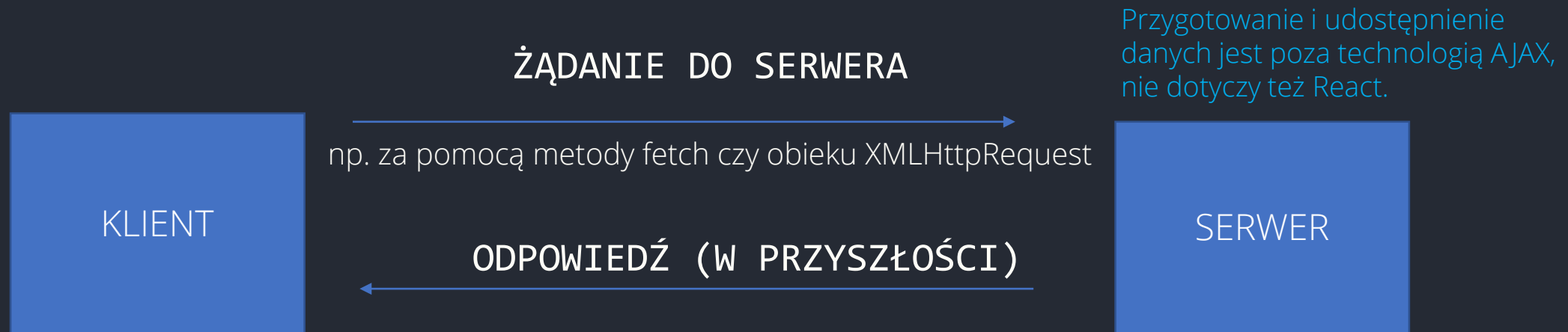
AJAX to m.in. XMLHttpRequest i fetch

Początkowa AJAX był oparty na obiekcie `XMLHttpRequest`, a jego główną zaletą i podstawowym zadaniem była możliwość `wczytania zawartości części strony bez odświeżenia całej strony`.

Współcześnie AJAX oferuje znacznie więcej sposobów. Najpopularniejszym (również w React) i dostępnym natywnie we współczesnych przeglądarkach jest funkcja `fetch` (i szerzej Fetch API) do asynchronicznego pobierania danych z wykorzystaniem mechanizmu obietnic.

Asynchroniczne żądanie

- żądanie otrzymania danych nie blokuje aplikacji.



Działanie aplikacji nie jest blokowane do czasu otrzymania odpowiedzi

Dane są przekazywane najczęściej w formacie JSON

Aplikacja wie o otrzymaniu odpowiedzi i może ją przetworzyć
(wywołanie zwrotne, promise)



JSON

JSON - format danych

JavaScript Object Notation - sposób prezentacji/przechowywania danych podobny do obiektu w JavaScript.

Dane są przechowywane w formie tekstu (tak są przesyłane między serwerem a klientem), który może zostać przetworzony/konwertowany/parsowany na obiekt JavaScript (może być także tablicą).

Bardzo popularny. Elastyczny. Dobrze się z nim pracuje. Łatwo konwertuje JSON na obiekt i obiekt na JSON.

JSON - metody

Warto zapamiętać `JSON.parse(string)` i `JSON.stringify(obiekt)`

```
const users = '[{"name": "Adam", "age": 20}, {"name": "Anna", "age": 30}]'
const obj = JSON.parse(users)
```

```
(2) [{...}, {...}] ⓘ
  ▶ 0: {name: "Adam", age: 20}
  ▶ 1: {name: "Anna", age: 30}
    length: 2
  ▶ __proto__: Array(0)
```

```
const json = JSON.stringify(obj)
```

```
"[{"name": "Adam", "age": 20}, {"name": "Anna", "age": 30}]"
```

JSON - zasady

```
{
  "lesson": "animals",
  "words": [
    {
      "id": 1,
      "en": "cat",
      "pl": "kot"
    },
    {
      "id": 2,
      "en": "dog",
      "pl": "pies"
    },
    {
      "id": 3,
      "en": "fish",
      "pl": "ryba"
    }
  ]
}
```

Nazwa właściwości w podwójnym cudzysłowie.

Brak przecinka po ostatniej właściwości.

Dopuszczone: string, number, boolean, null, object, array (ale nie funkcja czy undefined)

Jeśli w pliku, to rozszerzenie `.json`



API

API - Application Programming Interface

By dwie rzeczy mogły się ze sobą komunikować potrzebny jest interfejs np. pralka (czy każde urządzenie elektroniczne)

W świecie programowania i web developmentu najczęściej mówiąc o interfejsie mamy na myśli:

- **UI** w aplikacji/stronie - interakcja użytkownika i programu.
- **API** - interakcja między aplikacjami/programami/serwerami.

* ps. DOM to też API udostępnione przez przeglądarkę dzięki czemu mamy w naszej aplikacji dostęp do elementów i możemy je edytować.

API - wymowa i nazwy

API czyli Api czy EjPiAj ?

HTML czyli HTML czy EjczTiEmEl ?

AJAX - ajax, adżaks czy EjDżaks?

XMLHttpRequest a więc xml http rikłest czy eksEmEl EjczTiTiPi Rikłest ?

Promise czy obietnica?

API - nasza aplikacja często z niego korzysta

Założenie: w naszym przypadku mówimy o API w kontekście aplikacji, ale pamiętajmy, że samo API nie musi być związane ze stronami/aplikacjami i opierać się o połączenie sieciowe. API to sposób na pobranie czy wymianę danych między klientem a serwerem (który jest przygotowany do udostępniania danych w taki sposób, że odpowie na żądanie).

W skrócie: Jakaś aplikacja, poprzez http i URI udostępnia na zewnątrz interfejs za pomocą którego inne aplikacje mogą pobierać czy wysyłać dane. "Nasza aplikacja" wysyła żądanie, które jest przetwarzane w "innej aplikacji" i odpowiedź jest zwracana do "naszej aplikacji"

API - nasza aplikacja często z niego korzysta

Komunikacja w (web) API odbywa się z wykorzystaniem HTTP i metody, z których najpopularniejsza jest GET, inne to POST, DELETE, PUT ...) i adresów url (tzw. `endpoint'y`). Dzięki temu możemy otrzymać (lub wysłać) dane. Adres może zawierać dodatkowe parametry.

GET `https://www.udemy.com/api-2.0/courses/238934?fields[course]=title,headline`

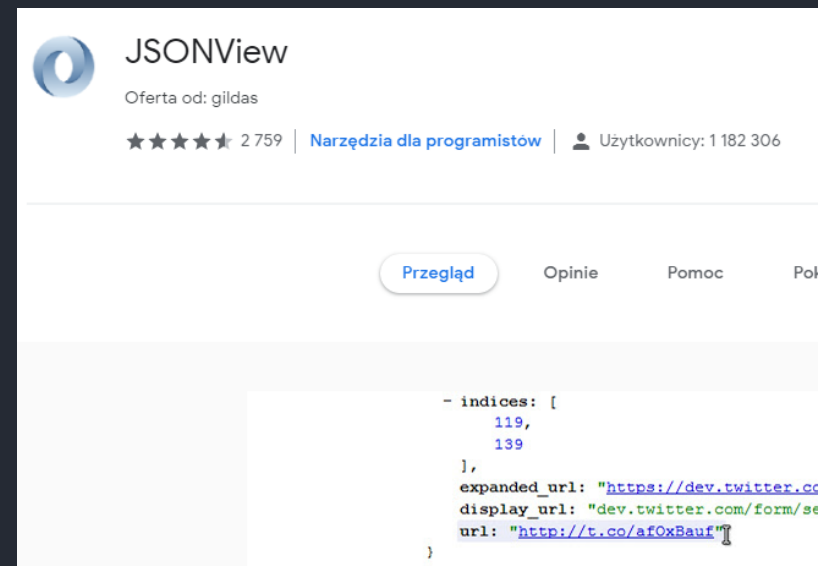
```
{  
  _class: "course",  
  id: 238934,  
  title: "Pianoforall - Incredible New Way To Learn Piano & Keyboard",  
  headline: "Learn Piano in WEEKS not years. Play-By-Ear & learn to Read Music. Pop, Blues, Jazz, Ballads, Improvisation, Classical"  
}
```


<https://www.udemy.com/api-2.0/courses/238934>

```
{"_class": "course", "id": 238934, "title": "Pianoforall - Incredible New Way To Learn Piano & Keyboard", "url": "/pianoforall-incredible-new-way-to-learn-piano-keyboard/", "is_paid": true, "price": "609,99 zł", "price_detail": {"amount": 609.99, "currency": "PLN", "price_string": "609,99 zł", "currency_symbol": "zł"}, "visible_instructors": [{"_class": "user", "title": "Robin Hall", "name": "Robin", "display_name": "Robin Hall", "job_title": "Piano Teacher (200,000 students online), Artist, Therapist,", "image_50x50": "https://udemy-images.udemy.com/user/50x50/5753906_1b3f_4.jpg", "image_100x100": "https://udemy-images.udemy.com/user/100x100/5753906_1b3f_4.jpg", "initials": "RH", "url": "/user/robinhall13/"}], "image_125_H": "https://udemy-images.udemy.com/course/125_H/238934_4d81_5.jpg", "image_240x135": "https://udemy-images.udemy.com/course/240x135/238934_4d81_5.jpg", "is_practice_test_course": false, "image_480x270": "https://udemy-images.udemy.com/course/480x270/238934_4d81_5.jpg", "published_title": "pianoforall-incredible-new-way-to-learn-piano-keyboard"}
```

```
{
  _class: "course",
  id: 238934,
  title: "Pianoforall - Incredible New Way To Learn Piano & Keyboard",
  url: "/pianoforall-incredible-new-way-to-learn-piano-keyboard/",
  is_paid: true,
  price: "609,99 zł",
  price_detail: {
    amount: 609.99,
    currency: "PLN",
    price_string: "609,99 zł",
    currency_symbol: "zł"
  },
  visible_instructors: [
    - {
      _class: "user",
      title: "Robin Hall",
      name: "Robin",
      display_name: "Robin Hall",
      job_title: "Piano Teacher (200,000 students online), Artist, Therapist,",
      image_50x50: "https://udemy-images.udemy.com/user/50x50/5753906_1b3f_4.jpg",
      image_100x100: "https://udemy-images.udemy.com/user/100x100/5753906_1b3f_4.jpg",
      initials: "RH",
      url: "/user/robinhall13/"
    }
  ],
  image_125_H: "https://udemy-images.udemy.com/course/125_H/238934_4d81_5.jpg",
  image_240x135: "https://udemy-images.udemy.com/course/240x135/238934_4d81_5.jpg",
  is_practice_test_course: false,
}
```

Na górze widok "normalny" w przeglądarce.
Po prawej to samo gdy skorzystamy z [viewer-a](#) do JSON-a w przeglądarce



REST API - proste, wydajne, skalowalne, niezawodne

Przy okazji warto wiedzieć, że takie API (jak to Udemy) często określamy jako **RESTful API** czy po prostu **REST API**. REST API/RESTful API cechuje się tym, że spełnia kilka fundamentalnych, określonych i zapisanych zasad dla architektury **REST** (**R**epresentational **S**tate **T**ransfer)

API, Web API, REST API i RESTful - reasumując

Często zdarza się, że mówiąc API, Web API, REST czy RESTful API mamy to samo na myśli - choć pojęcia te różnią się od siebie

Pamiętajmy, że **API** jest szerokim pojęciem (może dotyczyć np. interfejsu między programami, API oprogramowania czajnika, HTML5 APIs czy DOM API), **Web API** dotyczy komunikacji za pomocą protokołu HTTP między klientem a serwerem, **REST** jest zbiorem zasad a **RESTful API** (czy po prostu **REST API**) jest API opartym na http, który spełnia te zasady.



TECHNIKI AJAX

Różne techniki AJAX

Funkcje wbudowane w przeglądarkę

`XMLHttpRequest()` - technika stworzona na początku tego wieku, w nazwie sugeruje się XML, ale można oczywiście skorzystać też z formatu JSON.

`fetch()` - oparte na obietnicach (promises, ES6) - prawdopodobnie najpopularniejsza metoda do asynchronicznych żądań do serwera.

Inne popularne rozwiązania do AJAX (biblioteki)

`axios` - biblioteka która także korzysta z promisów. Popularna w React (npm install axios), ale nie tylko. Działa w podobny sposób do metody fetch.

`jQuery` - metoda `$.ajax()` - opiera się na `XMLHttpRequest`, ale jest bardzo przyjemna (przez wiele lat ważny powód by używać jQuery).



XMLHttpRequest

XMLHttpRequest - podstawowa metoda (już od IE9)

```
const xhr = new XMLHttpRequest(); //tworzymy obiekt XMLHttpRequest, który  
odpowiada za wszystkie etapy AJAX (definiowanie, wysyłanie, przechwycenie  
odpowiedzi, działanie)
```

```
xhr.open('GET', 'https://adresprzykladowy.pl', true); //metoda przygotowująca  
żądanie. Jak (czyli jaka metoda http), gdzie (również ścieżka relatywna) i  
czy asynchronicznie (true)
```

```
xhr.send(null); //wysyłamy żądanie (można przesłać też informacje do serwera  
tutaj; można nie używać żadnego argumentu)
```

```
xhr.onload = () => { } //po uzyskaniu odpowiedzi następuje zdarzenie onload,  
do którego przypisujemy funkcję.
```

XMLHttpRequest - zdarzenie onload

```
xhr.onload = () => { } //po uzyskaniu odpowiedzi
```

Wewnątrz funkcji możemy skorzystać z różnych właściwości obiektu, które na tym etapie są już uzupełnione danymi. Mamy dostęp m.in. do właściwości :

```
xhr.status //np. 200, wtedy wszystko ok
```

```
xhr.response (xhr.responseText) //odpowieź najczęściej w formie stringa
```

```
xhr.onload = () => {  
    if(xhr.status === 200) {  
        const data = JSON.parse(xhr.response) //parsowanie/deserializacja (JSON-a na obiekt) za pomocą metody parse  
        /* zrób to i to z tym obiektem */  
    }  
}
```


kod odpowiedzi z serwera (status)

Kilka, kilkanaście istotnych, które możemy obsłużyć.

200 - dostaliśmy (klient) odpowiedź i wszystko jest ok.

301 - trwałe przeniesienie zasobu na inny adres

404 - strona nie została odnaleziona (coś użytkownik zrobił nie tak)

500 - wewnętrzny błąd serwera (coś się zepsuło na serwerze)



Metoda fetch

i obietnice

Fetch API - interfejs do pobierania danych

Podstawowym elementem Fetch API jest **metoda fetch**, która jako jedyny wymagany argument przyjmuje **adres url** (bezwzględny/względny) i wysyła asynchroniczne żądanie do serwera (metoda GET domyślnie). Metoda fetch może też wysyłać dane (np. użyć metody POST).

Metoda fetch jest wspierana we wszystkich współczesnych przeglądarkach. Za współczesną przeglądarkę nie uważa się już IE ;).

* inne elementy Fetch API to np. obiekt Response czy metoda .json()

fetch() - wywołanie i tworzenie obietnicy

Jeden argument wymagany. URL naszego zasobu.

```
fetch('http://someapi.com/weather/v2/poland')  
fetch('data/users.js')
```

Metoda fetch tworzy obietnicę (promise), która po rostrzygnięciu (początkowo jest nierozstrzygnięta) może wywołać **metodę then** (gdy promise został spełniony) lub **catch** (gdy nie został spełniony).

fetch() - rezultat i wywołanie metod

```
fetch("http://some-cool-api.com/weather/v2/poland")  
  .then(/* kod gdy obietnica "zakończona" pozytywnie */ )  
  .catch(/* kod gdy obietnica "zakończona" negatywnie */ )
```

Poprawny, ale trochę mniej czytelny, jest też zapis w jednej linii

```
fetch(`http://amazingapi.com/weather/api/cities`).then(/* kod  
gdy obietnica rostrzygnięta pozytywnie*/ ).catch(/* kod gdy  
obietnica rostrzygnięta negatywnie */ )
```

then i catch - metody po rozstrzygnięciu obietnicy

```
fetch(`http://someapi.com/weather/v2/poland`)  
  .then(res => console.log('ok'))  
  .catch(err => console.log(`błąd ${err}`))
```

//to samo w jednej metodzie. Dwa argumenty, drugi to rozstrzygnięcie negatywne (zamiast catch)

```
fetch(`http://someapi.com/weather/v2/poland`)  
  .then(res => console.log('ok')), err => console.log(`błąd ${err}`))
```

//to samo w dwóch metodach then (zamiast catch użyjemy then, ale z pierwszym argumentem ustawionym na null)

```
fetch(`http://someapi.com/weather/v2/poland`)  
  .then(res => console.log('ok'))  
  .then(null, err => console.log(`błąd ${err}`))
```

O co chodzi - zrozumienie przez przykład

Najlepszy sposób na zrozumienie idei działania obietnic. to przykład zamawianie jedzenia czy wizyta w urzędzie...

1. kupujesz jedzenie i dostajesz Twój numer (np. metoda fetch, żądanie do serwera). Numer pełni rolę obietnicy - obietnicy, która brzmi dostaniesz swoje jedzenie (póki co jest nie wiesz czy dostaniesz, ale czekasz)
2. w przyszłości zostaniesz wezwany, Twój numer będzie wyczytany. Podchodzisz do okienka. Twoja obietnica jest rostrzygnięta.
 - Jeśli dostałeś jedzenie wszystko jest ok. Być może dostałeś nie dokładnie to co chciałeś, ale jedzenie (w przypadku fetch, odpowiedź serwera) masz. Stan obietnicy to rostrzygnięta spełniona.
 - Może się jednak zdarzyć, że podejdziesz z numerkiem i dostaniesz informację. Zgubiliśmy Państwa zamówienie, albo płatność została odrzucona. Taki błąd nadal oznacza, że obietnica została rostrzygnięta, ale ma stan odrzucona.

O co chodzi - chaining

Najlepszy sposób na zrozumienie idei działania obietnic. to przykład zamawianie jedzenia czy wizyta w urzędzie...

1. kupujesz jedzenie i dostajesz Twój numer (np. metoda fetch, żądanie do serwera). Numer pełni rolę obietnicy - obietnicy, która brzmi dostaniesz swoje jedzenie (póki co jest nie wiesz czy dostaniesz, ale czekasz)

2. w przyszłości zostaniesz wezwany, Twój numer będzie wyczytany. Podchodzisz do okienka. Twoja obietnica jest rostrzygnięta.

-- Jeśli dostałeś jedzenie wszystko jest ok. Być może dostałeś nie dokładnie to co chciałeś, ale jedzenie (w przypadku fetch, odpowiedź serwera) masz. Stan obietnicy to rostrzygnięta spełniona.

-- Może się jednak zdarzyć, że podejdziesz z numerkiem i dostaniesz informację. Zgubiliśmy Państwa zamówienie, albo płatność została odrzucona. Taki błąd nadal oznacza, że obietnica została rostrzygnięta, ale ma stan odrzucona.

Dodatkowo zobaczmy przykład **łańcucha obietnic** (promise chain). Załóżmy, że kupiłeś też napój (w pierwotnym zamówieniu). Prosisz teraz kelenera by dolał Ci wody do kubka (kolejny promise). Czekasz aż przyniesie wodę (w tym czasie promise jest nierostrzygnięty). Wtedy on przychodzi z pustym/pełnym/z problemem. Następuje rostrzygnięcie. I tu historia może pisać się dalej...

Fetch API - metoda fetch w praktyce

```
fetch(`http://someapi.com/weather/v2/poland`)  
  .then(response => response.json())  
  .then(result => console.log(result))  
  .catch(err => console.log(`błąd ${err}`))
```

Zapis podstawowy (jeszcze bez uzględniania statusów odpowiedzi (404, 301, 500, 200)). Z then skorzystamy dwukrotnie (o ile obie obietnice są rostrzygnięte pozytywnie). Najpierw by wyodrębnić dane (w tym wypadku json) z obiektu Odpowiedzi (Response) a potem by coś z nimi zrobić (w tym wypadku tylko wyświetlić w konsoli). Na końcu znajduje się metoda catch, która przechwyci i obsłuży nam błąd (przez błąd należy w tym przykładzie rozumieć sytuację gdy, któraś obietnica zostanie rostrzygnięta negatywnie, tzn, zmieni swój stan na "odrzucona").

Fetch API - metoda fetch w praktyce

```
fetch(`http://someapi.com/weather/v2/poland`)  
  .then(response => response.json())  
  .then(result => console.log(result))  
  .catch(err => console.log(`błąd ${err}`))
```

wywołanie fetch (powstaje promise) -> uzyskanie rostrzygnięcia (założmy sukces, czyli informacje z serwera) -> wywołanie metody then (pierwszej), do której przekazywany jest **obiekt odpowiedzi (Response)**, który możemy przetworzyć. Metoda then również tworzy promise -> po wykonaniu (zakładamy w przykładzie sukces) wywoływany jest kolejny then do którego przekazana została wartość zwracana z poprzedniej obietnicy (z obietnicy tworzonej przez response.json()).

Jeśli na jakimś etapie łańcuch promise zmienia stan z nierozstrzygniętego na rozstrzygnięty odrzucony, to wywołana będzie metoda catch (zazwyczaj umieszczamy ją na końcu).

Promise (obietnica) - czym jest

Obietnica (podobnie jak model zdarzeń i wywołań zwrotnych) definiuje **co ma być wykonane w przyszłości**. Obietnica może zakończyć się sukcesem lub niepowodzeniem. Stan początkowy obietnicy to **nierostrzygnięta**. Po rostrzygnięciu obietnica może zmienić swój stan na rostrzygnięta **spełniona** i rostrzygnięta **odrzucona**.

Obietnica reprezentuje **przyszły rezultat asynchronicznej operacji**. Są alternatywą dla wywołań zwrotnych. Upraszczają proces i pozwalają pozbyć się tzw. piekła wywołań zwrotnych (chodzi m.in. o to, że przy skomplikowanym kodzie, wywołania zwrotne stają się bardzo mało czytelne i trudne w utrzymaniu/rozwoju aplikacji).

Obietnica, to ważny aspekt zmian wprowadzonych w ES6. W naszym kursie warto o nich powiedzieć chociaż podstawe rzeczy, ponieważ obietnice są ważnym elementem pracy z metodami pobierania/wysyłania danych (fetch, biblioteka axios).

Promise (obietnica) - implementacja w fetch()

```
fetch('data.txt').then(/* obiekt Response jest częścią Fetch API*/)
```

```
const promise = fetch('data.txt');  
promise.then(/* obiekt Response jest częścią Fetch API*/)
```

W tej sytuacji (wywołanie metody fetch) zwracany jest promise czyli obiekt, który zostanie w przyszłości (po otrzymaniu odpowiedzi) uzupełniony o rezultat. Obietnica ma cykl życiowy składający się z trzech stanów: oczekujący, w toku (ang. [pending](#)), oraz rozwiązany zakończony sukcesem (ang. [resolved](#)) i rozwiązany odrzucony (ang. [rejected](#)). Póki jest oczekująca określamy ją jako nierozstrzygniętą ([unsettled](#)), gdy jest do niej przypisane rostrzygnięcie (a więc resolved albo rejected), to mówimy, że jest rostrzygnięta ([settled](#)).

Nasze rostrzygnięcie w metodzie fetch oprócz informacji o stanie dostarcza też dodatkowych informacji w postaci [obiektu Response](#) - są to właściwości i metody do przetworzenia odpowiedzi oraz pobrane dane (nie dostępne wprost)

Promise (obietnica) - metoda then

Za pomocą metody `then` (również metody `catch` dla błędu), można określić sposób działania gdy zmienia się stan obietnicy (czyli obietnica nierozstrzygnięta staje się rozstrzygnięta) i podjąć różne działania gdy to rozstrzygnięcie oznacza, że jest spełniona lub odrzucona.

```
const promise = fetch('data.txt');
```

```
//w przyszłości, po zmianie stanu, wykona się metoda then  
//Możliwość podania dwóch argumentów, pierwszy na funkcje wykonywana po  
sukcesie, drugi, opcjonalny, na funkcje po odrzuceniu.
```

```
promise.then(  
  response => { console.log("ok", response) },  
  err => { console.log('błąd', err); }  
)
```

metoda fetch - parametry response i err

```
const promise = fetch('data.txt');

promise.then(
  response => { console.log(response.ok, response.status) },
  err => { console.log('błąd', err); })
}
```

Parametry response i err (nazwy dowolne) przy metodzie fetch.

Konwencja jest jednak taka, że argument w pierwszym then dla metody fetch nazywamy response, result lub res. W chwili wywołania metody (then) do parametru jest przekazywany obiekt Response, więc nazwa response lub skrót res wydają się być właściwie.

Obiekt Response metody fetch()

```
const promise = fetch(url);
```

```
promise.then( res => console.log(res) )
```

▼ Response

body: (...)

bodyUsed: true

► headers: Headers {}

ok: true

redirected: false

status: 200

statusText: "OK"

type: "cors"

url: "http://numbersapi.com/23/year?json"

Obietnica spełniona, ale... 404

```
const promise = fetch('data.txt');  
  
promise.then( res => console.log(res) )
```

```
► GET http://127.0.0.1:5500/react/fetch/trening/data.txt 404 (Not Found)
```

```
► Response {type: "basic", url: "http://127.0.0.1:5500/react/fetch/trening/data.txt", redirected: false, status: 404, ok: false, ...}
```

W metodzie fetch obietnica będzie rostrzygnięta jako odrzucona w przypadku niemożności dostanie się do danej strony (np. nieprawidłowe określenie domeny) czy ze względów bezpieczeństwa. Błąd 404 nie powoduje negatywnego rostrzygnięcia.

Promise (obietnica) - metoda then

```
const promise = fetch('data.txt');
```

```
//dwa argumenty (pierwszy obietnica spełniona, drugi gdy odrzucana)
```

```
promise.then(  
  res => console.log(res),  
  err => console.log(err)  
)
```

//alternatywnie - then z jednym argumentem (gdy obietnica spełniona) i metoda catch, która jest wywoływana gdy obietnica jest odrzucona (zmienia stan na rozstrzygnięty odrzucony). Oba przykłady (ten i ten powyżej) dadzą ten sam efekt.

```
promise.then(res => console.log(res))  
promise.catch(err => console.log(err))
```

obietnice - składnia łanucha

```
fetch(`http://someapi.com/weather/v2/poland`)  
  .then(response => console.log(response))  
  .catch(err => console.log(`błąd ${err}`))
```

//to samo, ale wyżej wygodniej

```
const data = fetch(`http://someapi.com/weather/v2/poland`)  
data.then(response => console.log(response))  
data.catch(err => console.log(`błąd ${err}`))
```

Adres względny jako argument

```
const data = fetch('data.txt')
```

```
data.then(response => console.log(response))
```

```
data.catch(err => console.log(`błąd ${err}`))
```

--- to samo poniżej, ale szybciej (i wygodnie przy większej liczbie obietnic).

```
fetch('data.txt')
```

```
  .then(response => console.log(response))
```

```
  .catch(err => console.log(`błąd ${err}`))
```

wywołanie then i catch a nowy promise

Zapamiętaj.

Każde wywołanie then i catch zwróci nową obietnicę.

łańcuch obietnic i przekazywanie wartości

```
fetch('data.json') //tworzy obietnicę
```

`.then(response => "coś")` //po sukcesie (rostrzygnięcie, obietnica spełniona, dotyczy obietnicy stworzonej przez metodę `fetch`) wywołuje metodę `then`, która zwraca obietnicę. Możemy też przekazać coś do tej obietnicy. Do tej metody jest przekazywany obiekt `Response`, który znajduje się w przekazanym argumencie.

`.then(result => console.log(result))` //po sukcesie pokaże "coś" (bo to zostało przekazane) i zwróci kolejną obietnicę (nie musi już być obsługiwane, więc nie trzeba tworzyć kolejnego `then`). Do kolejnego `then` (jeśli by był) przekazany zostałby `undefined`, bo on jest zwracany.

`.catch(err => console.log("błąd" + err))` //gdy na którymś etapie nastąpi błąd.

fetch() - błąd w adresie

```
fetch('http://')  
  .then(response => "coś")  
  .then(result => console.log(result))  
  .catch(err => console.log("błąd " + err))
```

Wywołanie catch.

błąd TypeError: Failed to execute 'fetch' on 'Window': Failed to parse URL from http://

fetch() - błąd bezpieczeństwa (CORS policy)

```
fetch('http://api.example-fetch.com/api')  
  .then(response => "coś")  
  .then(result => console.log(result))  
  .catch(err => console.log("błąd " + err))
```

Wywołanie catch.

błąd Access to fetch at 'https://websamuraj.pl/api/course/v2/tags' from origin 'http://127.0.0.1:5500' has been blocked by CORS policy: No 'Access-Control-Allow-Origin' header is present on the requested resource. If an opaque response serves your needs, set the request's mode to 'no-cors' to fetch the resource with CORS disabled.

błąd TypeError: Failed to fetch

wyodrębnienie z obiektu Response tekstu

```
fetch('data.txt')  
  .then(res => res.text())  
  .then(result => console.log(result))  
  .catch(err => console.log(err))
```

Efekt w przypadku spełnienia obietnicy (1 i 2): wydrukowanie w konsoli zawartości pliku tekstowego

Efekt w przypadku odrzucenia obietnicy: wydrukowanie w konsoli błędu

-- pierwszy then przy zmianie stanu obietnicy na "spełniona". Wyodrębniamy z obiektu response zawartość tekstową i zwracamy ją do nowej obietnicy, która powstaje przy metodzie then (i catch)

-- drugi then wywołuje się gdy zmienia się stan w drugiej obietnicy. Do drugiego promise przekazujemy wyodrębnioną zawartość i coś z nią robimy (w tym przypadku pokazujemy w konsoli)

-- catch pełni rolę zabezpieczającą. Jeśli, któraś z dwóch obietnic zmieni stan z nierozstrzygnięty na odrzucony, wtedy wywoła się metoda catch

fetch - json i przetworzenie w React

```
fetch('data.json')  
  .then(response => response.json())  
  .then(result => {  
    const users = result.users.slice(0,100);  
    this.setState({users});  
  })  
  .catch(err => console.log(err))
```

przypomnijmy, można w jednej linii

```
fetch('data.json').then(res =>  
  res.json()).then(result =>  
    console.log(result.users)).catch(err =>  
      console.log(err))
```

Przykładowe właściwości obiektu Response

```
fetch(url)
  .then(res => console.log(res.status, res.ok, res.statusText))
    // np 200, true, "OK"
    //404, false, "Not Found"
  .catch(error => console.log(error, 'Pojawił się błąd!'))
}
```

Polecane rozwiązanie w metodzie fetch

```
const url = 'http://example.api.com/v3/cars';
fetch(url)
  .then(response => {
    if (response.ok) {
      return response; //lub od razu response.json()
    }
    throw Error(response.statusText);
    // pamiętajmy, że powstaje nowa obietnica (metoda then ją tworzy) -
    // użycie throw spowoduje, że nowa obietnica uzyska stan reject czyli
    // wywoła się w naszym przykładzie metoda catch
  })
  .then(response => /* coś robimy z odpowiedzią np. this.setState */)
  .catch(error => console.log(error, 'Houston, mamy problem'))
  // Error: Not Found, Houston, mamy problem
}
```

Fetch API - przykładowe metody do odczytu danych z obiektu Response

```
fetch(url)
  .then(res => res.json())
  .then(data => console.log(data))
}
```

`res.json()` - przetwarza zawartość JSON z odpowiedzi. Obiekt zwracany z metody `fetch` (Response) zawiera wiele elementów, między innymi właściwość `body`, która posiada dane, ukryte jednak i niedostępne wprost. musimy te dane wyodrębnić (odczytać) i zamienić je na obiekt JavaScript (o ile są w formacie JSON). Taki wyodrębniony obiekt jest w przykładzie przekazany jako argument przy wywołaniu metody `then` dla kolejnej obietnicy.

`res.text()` - jak metoda `json`, przy czym tu wyodrębnieniu ulega tekst (dobre dla `txt` czy `xml`), który w formie stringa będzie przekazany do metody `then` kolejnej obietnicy.

`res.blob()` - do pracy z obrazkami

metoda fetch() - drugi argument (opcjonalny)

```
const option = {  
  method: 'post',  
  body: JSON.stringify(data),  
}
```

Drugi argument (opcjonalny) metody fetch, pełni rolę konfiguracyjną. Jest obiektem, który może zawierać różne właściwości np. metodę http (domyślnie jest GET).

```
fetch(url, option)  
  .then(response => response.json())  
  .then(data => console.log(data))  
}
```



Pobieranie danych w React

Kiedy pobierać dane w React poprzez Web API

`componentDidMount` - to dobre miejsce do ustawienia pierwszego asynchronicznego żądania (pobrania) danych z endpointu (jakiegoś url) i aktualizacji stanu (`setState`). Spowoduje to ponowne wywołanie metody `render` i ewentualnie przekazanie nowych wartości jako props-y do komponentów dzieci.

Nie powinniśmy do tego używać metody `constructor` (pobranie jest asynchroniczne, więc nie możemy zainicjalizować danych w ten sposób) czy metody `render` (nie możemy użyć `setState` bezpośrednio w `render` i `constructor`).

Pobieranie z Web API w metodzie constructor - problemy

constructor()

(1) Użycie setState bezpośrednio w konstruktorze

Warning: Can't call setState on a component that is not yet mounted.

(2) Użycie żądania asynchronicznego do pobrania danych inicjalizujących

```
this.state = {  
  people: this.fetchData()  
}
```

// w render właściwość this.state.people będzie miała wartość undefined ponieważ render wywoła się przed wynikiem metody asynchronicznej

Pobieranie z Web API w metodzie render - problemy

render()

Dwa problemy po użyciu setState

(1)Warning: Cannot update during an existing state transition (such as within `render`). Render methods should be a pure function of props and state.

(2)Uncaught Error: Maximum update depth exceeded. This can happen when a component repeatedly calls setState inside componentWillUpdate or componentDidUpdate.

Kiedy pobierać dane w React poprzez Web API cz.2

Oprócz `componentDidMount` najczęściej korzystam jeszcze z:

- `funkcji obsługująca zdarzenie` (kliknięcie, wysłanie formularza, skrolowanie strony itd.)
- `componentDidUpdate` (po kolejnych zmianach). Pamiętajmy że by nie doszło do pętli musimy użyć tu warunku.

Gdzie (w którym komponencie) pobierać dane

Nic odkrywczego. W komponencie, który jest wspólny (jest nadrzędny) dla wszystkich komponentów, które będą potrzebowały tych danych.

Przejdźmy do kolejnych projektów

Zadanie z Web API (przetrenujemy jeszcze raz)

To Do App - projekt listy zadań.

Formularz z walidacją.