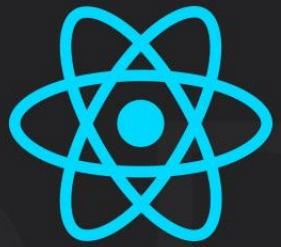


React

Promises
Wyświetlanie zdalnych danych
Routing po stronie klienta

Promise

obietnica 🙏

Obietnica na co?

Na to, że dane, do których chcemy się dostać, będą już gotowe i na miejscu.

O co chodzi? Kod JavaScript jest wykonywany synchronicznie, linijka po linijce. Skąd mamy pewność, że jakieś dane są gotowe?

```
const sum = (a, b) ⇒ a + b;


let a = 5;

let b = 10;

let suma = sum(a + b);

console.log(suma);
```

Mamy wszystko na miejscu. Wiemy, że **a** to 5, **b** to 10, **suma** wykorzystuje znane nam funkcję **sum** i zmienne **a**, **b**. Wszystko jest wiadome przy kompilacji.

W jaki sposób jakiś z tym elementów może nie być gotowy?

Wyobraźmy sobie, że nasz pewien znajomy ma nam do oddania pieniądze. Co więcej, obiecał, że je nam odda (szok). My sobie już pomyśleliśmy co zrobimy z tymi pieniędzmi jak je nam odda, np. kupimy kurs na uDemy w celu pouczenia się Reacta. Załóżmy, że chcemy przedstawić tę sytuację w kodzie.

```
let pieniazki = 0;


setTimeout(() ⇒ {
  pieniazki += 100;
}, 10000);


Udemy.kupKurs('React od podstaw', pieniazki);
```

Zakładamy, że na początku nie mamy żadnych pieniędzy i znajomy odda nam je po 10 sekundach.

Czyli poczekamy 10 sekund, dostaniemy pieniądze i wtedy kupimy kurs “React od podstaw” za te pieniądze. Czy tak to zadziała?

W żadnym razie.

Problem polega na tym, o czym wspomniałem wcześniej - **synchroniczność kodu**.

Ustalamy, że **nie mamy** na początku pieniędzy. Ustalamy potem, że po **10 sekundach** tych pieniędzy nam przybędzie. Potem jednak **od razu** przechodzimy do **kupna kursu**. Kod nie czeka aż przybędzie nam pieniędzy, a przecież takie coś chcemy osiągnąć!

Jak się z tym uporać?

```
let pieniazki = 0;


setTimeout(() => {
  pieniazki += 100;
}, 10000);


Udemy.kupKurs('React od podstaw', pieniazki);
```

```
let pieniazki = 0;

setTimeout(() ⇒ {
  pieniazki += 100;

  Udemy.kupKurs('React od podstaw', pieniazki);
}, 10000);
```

Wrzuciliśmy również zakup kursu do opóźnienia. W tym przypadku faktycznie będziemy mieli w końcu pieniądze przy kupowaniu kursu. Problem rozwiązany!

Wyobraźmy sobie teraz inną sytuację. Co zrobić, gdy nie wiemy, kiedy dostaniemy dostaniemy te pieniądze? Albo co w przypadku, kiedy czekamy aż trzy osoby nam oddadzą pieniądze, bo wtedy dopiero będzie nas stać na kurs? Albo co jeżeli ktoś nie dotrzyma obietnicy i nie dostarczy nam pieniędzy?

Promise

obietnica 🙏

Asynchroniczność!

A co jeśliby poczekać aż pieniądze dotrą i wtedy je wykorzystać na kupienie kursu, a w tym samym czasie mógłbym załatwiać swoje sprawy?

Obiekt **Promise** pozwala nam wykonywać operacje na danych, których z początku nie znamy, ale wiemy, że zostaną nam dostarczone. Nawet jeżeli nie zostaną dostarczone to **Promise** udostępnia nam możliwość poradzenia sobie z taką sytuacją.

Takich nieznanych z początku danych możemy mieć wiele, ale dzięki **Promise'om** możemy je **asynchronicznie** obsługiwać.

```
function oczekujPieniedzy(ilosc) {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      let czyDostane = Math.random() ≥ 0.5;
      if (czyDostane)
        resolve(ilosc);
      else
        reject('Nie dostaniesz pieniędzy');
    }, 2000);
  });
}

oczekujPieniedzy(100)
  .then(pieniadze => Udemu.kupKurs('React od podstaw', pieniadze)
  .catch(console.error);
```

Promise może zostać spełniony lub nie. W tym kodzie o jego spełnieniu decyduje zmienna **czyDostane**, przez którą mamy równe prawdopodobieństwo, że Promise zostanie spełniony czy też nie.

Kiedy **Promise** ma zostać spełniony, wywołujemy metodę **resolve**. Jako argument przyjmuje to, co ma zostać przekazane później do metody **then** (then wywołuje się w przypadku spełnienia Promise'a).

Promise nie jest spełniany kiedy wywołana zostanie metoda **reject**, która jako argument przyjmuje najczęściej komunikat błędu. Niespełnienie **Promise'a** jest wychwytywane przez metodę **catch**.

Fetch API – interfejs do pobierania danych

Podstawowym elementem jest metoda **fetch**, która jako jedyny wymagany argument przyjmuje adres url (bezwzględny lub względny) i wysyła **asynchroniczne** żądanie do serwera za pomocą metody GET.

Metoda fetch może także wysyłać dane (np. metodą POST).

Posiada również metody Response oraz .json()

Wspierana przez wszystkie współczesne przeglądarki (nie zalicza się do nich IE).

fetch() wywołuje i tworzy promise (obietnicę)

Przykłady użycia fetch:

```
fetch('http://google.com/weather');
fetch('src/index.js');
```

Metoda **fetch** tworzy **promise** (na początku nierozstrzygniętą). Następnie po rozstrzygnięciu może wywołać metodę then (gdy warunki obietnica zostanie spełniona) lub catch (gdy nie zostanie spełniona).

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

then i catch mozna zapisać na kilka sposobów

```
fetch('http://google.com/weatherapi')
    .then(res => console.log('Success!'))
    .catch(err => console.log(`Error ${err}`))

fetch('http://google.com/weatherapi')
    .then(res => console.log('Success!'), err => console.log(`Error ${err}`))

fetch('http://google.com/weatherapi')
    .then(res => console.log('Success!'))
    .then(null, err => console.log(`Error ${err}`))
```

Promise i fetch w prawdziwym życiu

Załóżmy, że:

- zamawiasz jedzenie w restauracji (metoda fetch),
- dostajesz numer zamówienia (promise - obietnica, że dostajesz jedzenie),
- czekasz - obietnica nie jest rozstrzygnięta
- odbierasz zamówienie - obietnica jest rozstrzygnięta,
- jeżeli Twoje zamówienia spełnia Twoje wymagania (lub nie spełnia, ale zostało zrealizowane) to obietnica została spełniona (status OK),
- jeżeli okazało się, że zamówienie nie może zostać zrealizowane to obietnica jest rozstrzygnięta ze statusem ODRZUCONA

Promise chain, czyli łańcuch obietnic

- w pierwotnym zamówieniu zamówiłeś szklankę wody,
- teraz prosisz o dolewkę (uruchamia się kolejny promise),
- czas oczekiwania na wodę to status promise nierozstrzygnięty,
- następnie kelner dostarcza Ci dobrą/złą dolewkę lub mówi, że jestem problem... (rozstrzygnięcie)

Promise chain w kodzie

```
1. fetch('http://google.com/weatherapi')
2.     .then(response => response.json())
3.     .then(result => console.log(result))
4.     .catch(err => console.log(`Error ${err}`))
```

1. Wywołanie metody **fetch** (powstaje **promise**)

Obietnica dotarła prawidłowo

Obietnica odrzucona

2. Obrobienie obiektu response na format json

Obietnica odrzucona

Obietnica dotarła prawidłowo

4. Wywołanie metody catch.

Obietnica odrzucona

3. Wyświetlenie response

Obiekt Response

Metoda then zwraca obiekt response

```
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"
```

Obiekt Response

Błąd 404 nie powoduje negatywnego rozstrzygnięcia

```
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 niemożliwości dostanie się do danej strony (np. nieprawidłowe określenie domeny) czy ze względów bezpieczeństwa. Błąd 404 nie powoduje negatywnego rostrzygnięcia.

Fetch json – przetworzenie danych w React

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

1. Wysyłamy zapytanie o dane
2. Przetwarzamy obiekt Response na dane Javascript
3. Operujemy na przetworzonych danych
4. Obsługa błędu

Fetch json – przetworzenie danych w React

`res.json()` - przetwarza zawartość JSON z odpowiedzi. Obiekt zwracany z metody fetch (Response) zawiera wiele elementów, między innym właściwość body, która posiada dane, ukryte jednak i niedostępne wprost. musimy te dane wyopdrębnić (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.

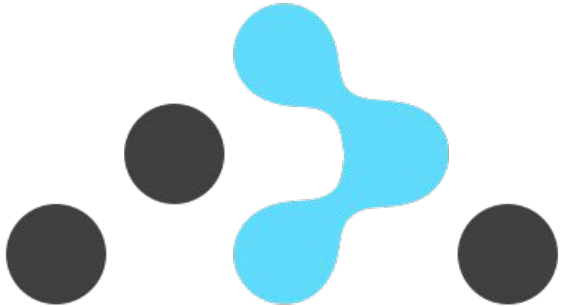
`res.text()` - jak metoda json, przy czym tu wyodrębnieniu ulega tekst (dobre dla txt czy xml), który w formie stringa będzie przekazany do metody then kolejnej obietnicy.

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

Routing

Dzięki routingowi jesteśmy w stanie tworzyć aplikację posiadającą kilka widoków.

Tak naprawdę niewiele aplikacji w internecie posiada jedną stronę (widok). W tej prezentacji przedstawimy w jaki sposób za pomocą React Routing można stworzyć aplikację z kilkoma widokami.



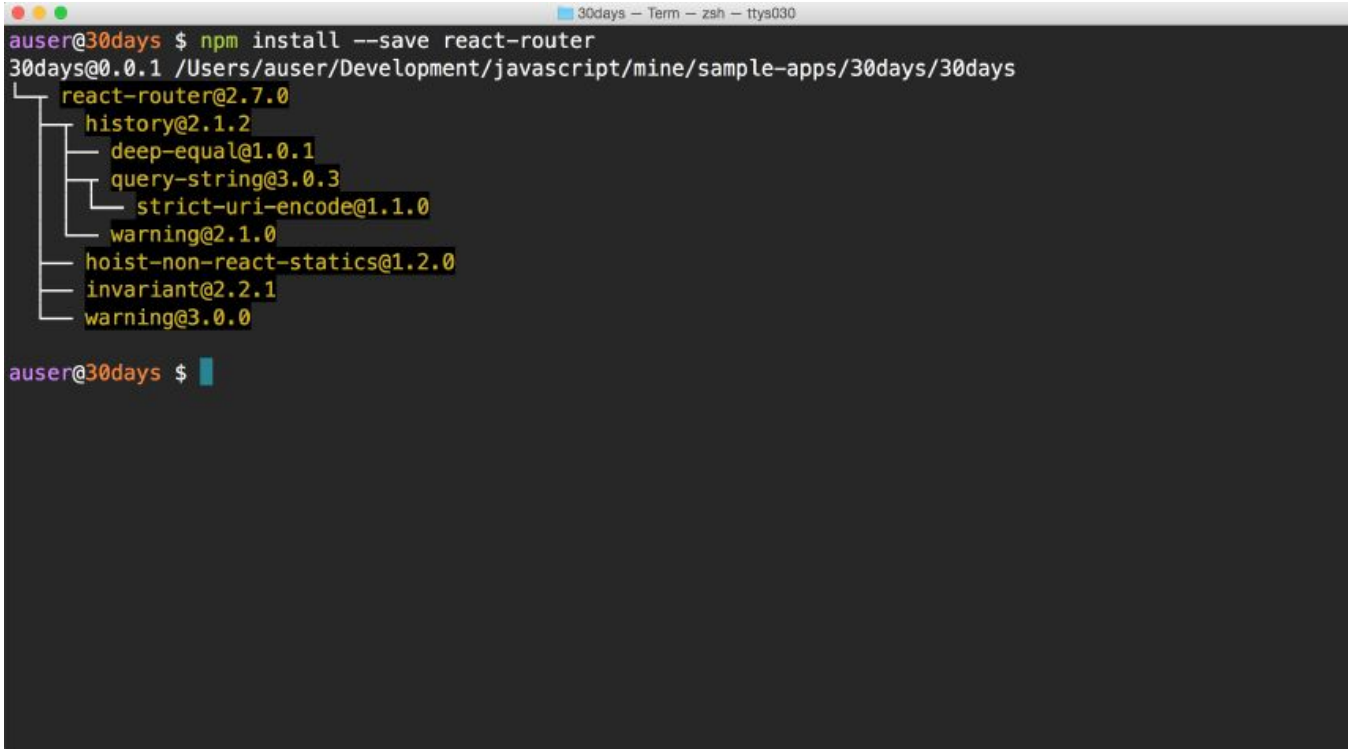
REACT ROUTER

React-Router

Posłużymy się w tym celu popularną biblioteką react-router (https://github.com/ReactTraining/react-router) do obsługi różnych odnośników. Żeby móc z niej skorzystać, musimy zainstalować npm package manager na swoim komputerze.

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

Pomyślna instalacja npm package manager



Struktura aplikacji

Z nowo zainstalowanym react-routerem, zaimportujemy kilka paczek z biblioteki i zaktualizujemy architekturę naszej aplikacji. Zanim wykonamy te kroki, spójrzmy *jak* i *dla*czego projektujemy naszą aplikację w ten sposób.

Konceptualnie aplikacja w React powinna być strukturą drzewa korzystającą z komponentów i zagnieżdżonych w nich pod-komponentów. Używając routingu w single page application możemy myśleć o różnych częściach składowych strony jako o komponentach-dzieciach. Tworząc aplikację w ten sposób możemy dynamicznie przełączać się pomiędzy kolejnymi częściami drzewa aplikacji posługując się innymi komponentami.

Innymi słowy, zdefiniujemy komponent Reacta jako komponent *root* (nadrzędny) elementów przekierowujących. Dzięki temu możemy przekazać bibliotece React informację, aby zmieniła ona wyświetlany widok aplikacji, co tak naprawdę podmieni cały komponent na inny tak, jakby podmieniała stronę renderowaną przez serwer.

Inicjalizacja routingu w aplikacji

Do ustanowienia routingu w naszej aplikacji posłuży nam komponent **BrowserRouter**. Dzięki niemu, wszelkie komponenty-dzieci będą podlegały routingowi. Aby określić jaki widok otrzymamy w zależności od wpisanego przez nas adresu URL, użyjemy komponentu **Route**.

```
import { BrowserRouter as Router, Route } from "react-router-dom";

const App = () => {
  return (
    <Router>
        {/* Tu umieścimy komponenty Route */}
    </Router>
  );
};
```

BrowserRouter jest importowany w tym przykładzie pod nazwą **Router**.

Inicjalizacja routingu w aplikacji

Komponent **Route** przyjmuje jako propsy między innymi **component** i **path** (**component** określa co jest renderowane, kiedy jesteśmy w ścieżce **path**). Route w przykładowym kodzie ma dodatkowy prop **exact** ustawiony na *true*, co oznacza, że komponent *Home* wyrenderuje się tylko wtedy, gdy ścieżka będzie dokładnie równa "/" bez niczego więcej (np. "*/home*" już nie przejdzie, choć ukośnik znajduje się na początku ścieżki).

```
import { BrowserRouter as Router, Route } from "react-router-dom";


const Home = () => (
  <div>
    <h1>Witaj świecie!</h1>
  </div>
);
const App = () => {
  return (
    <Router>
        <Route path="/" exact={true} component={Home}/>
    </Router>
  );
};
```

Przykład routingu na kodzie

localhost:3000

Witaj świecie!

localhost:3000/about

```
import { BrowserRouter as Router, Route } from "react-router-dom";


const Home = () => (
  <div>
    <h1>Witaj świecie!</h1>
  </div>
);
const App = () => {
  return (
    <Router>
        <Route path="/" exact={true} component={Home}/>
    </Router>
  );
};
```

Przełączanie widoku

Zimportujmy kolejny komponent - **Link**. Jest to hiperłącze, które przeniesie nas do konkretnej ścieżki, która wiąże się z jakimś **Route**'m. Dla przykładu, dodajemy kolejny widok *About*, do którego przełączymy się za pomocą hiperłącza "*coś o mnie*".

```
import { BrowserRouter as Router, Route, Link } from "react-router-dom";

const Home = () => (
  <div>
    <h1>Witaj świecie!</h1>
    <Link to="/about">Coś o mnie</Link>
  </div>
);
const About = () => (
  <div>
    <h1>O mnie</h1>
  </div>
);

const App = () => {
  return (
    <Router>
      <Route path="/" exact={true} component={Home}/>
      <Route path="/about" component={About}/>
    </Router>
  );
};
```

Przykład Link'a

localhost:3000

Witaj świecie!

Coś o mnie

localhost:3000/about

O mnie

```
import { BrowserRouter as Router, Route, Link } from "react-router-dom";

const Home = () => (
  <div>
    <h1>Witaj świecie!</h1>
    <Link to="/about">Coś o mnie</Link>
  </div>
);
const About = () => (
  <div>
    <h1>O mnie</h1>
  </div>
);

const App = () => {
  return (
    <Router>
      <Route path="/" exact={true} component={Home}/>
      <Route path="/about" component={About}/>
    </Router>
  );
};
```

Przekazywanie prop'sów do komponentów

Komponenty **Route** mają także prop'a podobnego do **component**, mianowicie **render**. Pozwala on na bardziej złożone renderowanie widoków, między innymi dlatego, że możemy przekazać renderowanemu komponentowi kolejne prop'sy (w tym przypadku w *About* dajemy prop'a *name="Paweł"*).

```
import { BrowserRouter as Router, Route, Link } from "react-router-dom";

const Home = () => (
  <div>
    <h1>Witaj świecie!</h1>
    <Link to="/about">Coś o mnie</Link>
  </div>
);
const About = ({name}) => (
  <div>
    <h1>O mnie</h1>
    <p>Jestem {name} 😃</p>
  </div>
);
const App = () => {
  return (
    <Router>
      <Route path="/" exact={true} component={Home}/>
      <Route path="/about" render={() => <About name="Paweł"/>}/>
    </Router>
  );
};
```

Przykład render'a

localhost:3000/about

O mnie

Jestem Paweł 😃

```
import { BrowserRouter as Router, Route, Link } from "react-router-dom";

const Home = () => (
  <div>
    <h1>Witaj świecie!</h1>
    <Link to="/about">Coś o mnie</Link>
  </div>
);
const About = ({name}) => (
  <div>
    <h1>O mnie</h1>
    <p>Jestem {name} 😃</p>
  </div>
);
const App = () => {
  return (
    <Router>
      <Route path="/" exact={true} component={Home}/>
      <Route path="/about" render={() => <About name="Paweł"/>}/>
    </Router>
  );
};
```

Dziękujemy za uwagę!

Więcej na temat routingu dowiesz się z:

https://github.com/ReactTraining/react-router/tree/master/docs
https://www.newline.co/fullstack-react/