

REACT

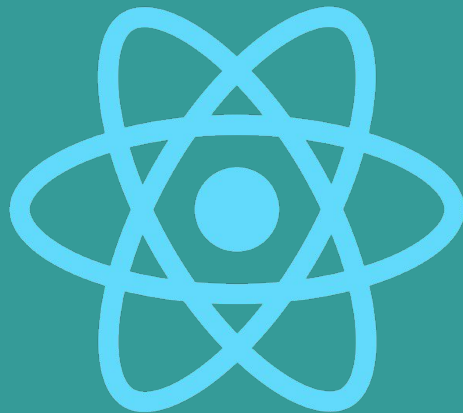


Hello

Kamil Richert

Engineering Manager w Atlassian

Co to w sumie jest?



javascriptowa biblioteka służąca do tworzenia interfejsów użytkownika
często wykorzystywana do tworzenia aplikacji typu **Single Page Application**

Główne założenia

Deklaratywny

React znacznie ułatwia tworzenie interaktywnych UI: zajmie się sprawną aktualizacją i ponownym renderowaniem odpowiednich komponentów.

Oparty na komponentach

Jako że logika komponentów pisana jest w JavaScriptcie, a nie w szablonach, przekazywanie skomplikowanych struktur danych i przechowywanie stanu aplikacji poza drzewem DOM staje się łatwiejsze.

Naucz się raz, używaj wszędzie

React działa w izolacji od reszty stosu technologicznego, dzięki czemu możesz w nim tworzyć nowe funkcjonalności, bez konieczności przepisywania istniejącego kodu.

React

Zalety

- ma wysoką wydajność, ponieważ opiera się na domenie wirtualnej
- jego komponenty mogą być wielokrotnie używane
- jest wszechstronny
- można go używać z dowolnym frameworkiem
- jest stabilny, stoi za nim spora społeczność, cały czas się rozwija
- stosuje jednokierunkowy przepływ danych
- pozwala na zbudowanie dynamicznego interfejsu
- może być używany przez początkujących programistów

React

Wady

- hmmm? ;)
- architektura - odpowiedzialność spada na nas
- JSX - trzeba się go nauczyć
- biblioteki dodatkowe - by tworzyć skomplikowane aplikacje trzeba ich sporo poznać
- prędkość rozwoju biblioteki - może się okazać dla kogoś problematyczne

React

React pozwala nam na budowanie aplikacji webowych, nie mylcie tego z aplikacjami mobilnymi. Jednak powstał odrębny bardzo podobny twór: **React Native**, który pozwala tworzyć aplikację mobilne z użyciem javascriptu.

Więcej info tutaj: <https://reactnative.dev/>

Create React App

to oficjalnie obsługiwany sposób tworzenia single page aplikacji w React. Oferuje nowoczesne ustawienia kompilacji bez konfiguracji.

Create react app

Co wchodzi w jego skład?

- **Menadżer pakietów** - **yarn** lub **npm**. Umożliwia on korzystanie z ogromnego ekosystemu dodatkowych pakietów. Pozwala łatwo je instalować i aktualizować.
- **Bundler** - **webpack**. Umożliwia on pisanie kodu modularnego i pakowania go w małe pakiety, aby zoptymalizować czas ładowania.
- **Kompilator** - **babel**. Pozwala on na stosowanie nowych wersji JavaScriptu przy zachowaniu kompatybilności ze starszymi przeglądarkami.

Create react app

Dostępne skrypty:

- **npm start**

Uruchamia aplikację w trybie deweloperskim. Otwórz adres **http://localhost:3000**, aby wyświetlić ją w przeglądarce. Strona zostanie załadowana ponownie, jeśli wprowadzisz zmiany (*hot-reload*). W konsoli zostaną również wyświetlone wszelkie błędy linta.

- **npm test**

Uruchamia testy aplikacji za pomocą biblioteki **Jest** w trybie *watch*, co oznacza, że testy się odpalą automatycznie po zmianie w kodzie.

Create react app

Dostępne skrypty:

- **npm run build**

Buduje aplikację w wersji produkcyjnego w folderze *build*. Prawidłowo łączy Reacta w trybie produkcyjnym i optymalizuje kompilację pod kątem najlepszej wydajności. Kompilacja jest zminimalizowana, a nazwy plików zawierają skróty.

- **npm run eject**

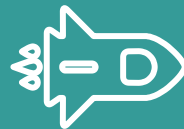
Uwaga: jest to operacja jednokierunkowa. Po jej wykonaniu nie można cofnąć! To polecenie spowoduje usunięcie ustawień kompilacji z projektu.

Create react app

Struktura plików:

```
my-app/  
  README.md  
  node_modules/  
  package.json  
  public/  
    index.html  
    favicon.ico  
  src/  
    App.css  
    App.js  
    App.test.js  
    index.css  
    index.js  
    logo.svg
```

ZADANIE

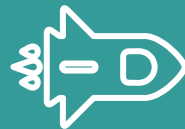


Stwórz aplikację reactową za pomocą **create-react-app**.

`npx create-react-app my-app`

Uruchom aplikację w trybie developerskim zgodnie z instrukcją wyświetlaną w konsoli i dokonaj zmiany tekstu.

ZADANIE



Stwórz produkcyjną wersję swojej aplikacji za pomocą
`npm run build`

Następnie przejdź do folderu *build* i otwórz plik **index.html**.

Create react app

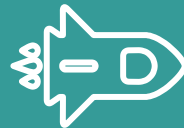
Uwaga! Konfiguracja w create-react-app jest tak stworzona, że webpack będzie próbował odczytać pliki z źródła systemu. W związku z tym musimy wskazać webpackowi skąd ma je odczytać i będzie to folder w której znajdują się aplikacja.

By to zrobić wystarczy dodać: **"homepage": "."** do package.json.

Create react app pozwala również na łatwe publikowanie aplikacji na gh-pages:

<https://typeofweb.com/react-js-na-github-pages-dzieki-create-react-app/>

ZADANIE



Dodaj **homepage** do package.json i odpal ponownie.

npm run build

Następnie przejdź do folderu *build* i otwórz plik **index.html**.

JSX

jest on nakładką na JavaScript, która dodaje możliwość wstawiania kodu html (lub komponentów React) bezpośrednio w kodzie, zamiast ciągu znaków.

JSX

Przykład elementu JSX:

```
const element = <h1>Hello world!</h1>;
```

JSX jest tłumaczony jeden do jednego na reactowe “elementy”. JSX kompiluje się za pomocą narzędzia Babel do Javascriptu i faktyczne elementy HTMLowe zostają stworzone za pomocą funkcji `createElement` i potem dodane do drzewa DOM przez funkcję renderującą.

React nie wymaga używania JSX, jednakże większość programistów uważa go za przydatne narzędzie, unaoczniające to, co dzieje się w kodzie javascriptowym operującym na interfejsach graficznych.

Uwaga: JSX to nie HTML!

JSX

Ponieważ JSX kompiluje się do Javascriptu to możemy w jego wnętrzu wykorzystywać zmienne czy osadzić dowolne wyrażenie Javascriptowe. By React wiedział, że chcesz wykorzystać zmienną to musisz dane wyrażenie otoczyć nawiasami klamrowymi.

Przykład elementu JSX z wyrażeniem:

```
const name = 'Janusz';  
const element = <h1>Witaj, {name}!</h1>;
```

Inaczej mówiąc: za pomocą nawiasów klamrowych “używamy” javascript w JSX.

JSX

JSX jest również wyrażeniem, więc możemy go używać jako wyniki funkcji warunkowej np przykład:

```
if (statement) { return <h1>Witaj, świecie!</h1>; }
```

W JSX możemy również przekazywać atrybuty do elementów HTML:

```
const element = <div tabIndex="0"></div>;  
const element = <img src={user.avatarUrl} />;
```

JSX

Znaczniki JSX mogą również zawierać elementy potomne:

```
const element = (  
  <div>  
    <h1>Witaj!</h1>  
    <h2>Dobrze cię widzieć.</h2>  
  </div>  
);
```

JSX

Jako że składni JSX jest bliżej do JavaScriptu niż do HTML-a, React DOM do nazywania argumentów używa notacji camelCase zamiast nazw atrybutów HTML-owych.

Przykładowo, w JSX **class** staje się **className**, zaś zamiast **tabindex** używamy **tabIndex**.

Uwaga! By korzystać z JSX w ramach danego komponentu musi być zaimportowany React: **import React from 'react';**
Jest tak ponieważ JSX jest kompilowany do metody `React.createElement`.

Od wersji 17 Reacta nie ma potrzeby dodawania tego importu.

JSX

W JSX musimy być zawsze zwrócony jeden element ze względu na to jak działa metoda `React.createElement`. Zatem tego typu zapis jest niepoprawny:

```
const element = (  
  <h1>Witaj!</h1>  
  <h2>Dobrze cię widzieć.</h2>  
);
```

By to poprawić możesz wykorzystać tag html'owy lub react'owy fragment:

```
const element = (  
  <>  
    <h1>Witaj!</h1>  
    <h2>Dobrze cię widzieć.</h2>  
  </>  
);
```

FUNKCJA RENDERUJĄCA

Aby wyrenderować reactowy element w węźle drzewa DOM, przekaz oba do **ReactDOM.render()**:

```
const root = ReactDOM.createRoot(  
  document.getElementById('root') as HTMLElement  
);  
root.render(  
  <React.StrictMode>  
    <App />  
  </React.StrictMode>  
);
```

ReactDOM?

Tak React tworzy swoje własne drzewo DOM. Nazywane to jest **Virtual DOM**.

VIRTUAL DOM

React przechowuje cały DOM aplikacji w pamięci, po zmianie stanu wyszukuje różnice między wirtualnym i prawdziwym DOM i aktualizuje zmiany.

VIRTUAL DOM

ReactDOM porównuje element i jego potomków do poprzedniego oraz nakłada tylko te aktualizacje drzewa DOM, które konieczne są do doprowadzenia go do pożądanego stanu.

Hello, world!

It is 12:26:47 PM.



```
▼ <div id="root">
  ▼ <div data-reactroot="
    <h1>Hello, world!</h1>
    ▼ <h2>
      <!-- react-text: 4 -->
      "It is "
      <!-- /react-text -->
      <!-- react-text: 5 -->
      "12:26:46 PM"
      <!-- /react-text -->
      <!-- react-text: 6 -->
      "."
      <!-- /react-text -->
    </h2>
  </div>
</div>
```

ZADANIE



Zobaczmy ten Virtual DOM!

Zainstaluj narzędzia deweloperskie od Reacta i sprawdź wirtualny DOM.

[react-developer-tools](#)

Komponent

Komponenty to niezależne i wielokrotnego użytku kawałki kodu. Służą temu samemu celowi co funkcje JavaScript, ale działają w izolacji i zwracają kod HTML za pośrednictwem funkcji renderującej, czyli zawierają zarówno znaczniki HTML, jak i związaną z nimi logikę.

Komponenty

Spróbujmy podzielić poniższą aplikację na komponenty:

☐ Only show products in stock

Name	Price
Sporting Goods	
Football	\$49.99
Baseball	\$9.99
Basketball	\$29.99
Electronics	
iPod Touch	\$99.99
iPhone 5	\$399.99
Nexus 7	\$199.99

Komponenty

Żółty - FilterableProductTable

Niebieski - SearchBar

Zielony - ProductTable

Turkusowy - ProductCategoryRow

Czerwony - ProductRow

☐ Only show products in stock

Name	Price
Sporting Goods	
Football	\$49.99
Baseball	\$9.99
Basketball	\$29.99
Electronics	
iPod Touch	\$99.99
iPhone 5	\$399.99
Nexus 7	\$199.99

Komponenty

W React'cie rozróżniamy 2 rodzaje komponentów:

komponent funkcyjny

```
function Welcome() {  
  return <h1>Cześć</h1>;  
}
```

komponent klasowy

```
class Welcome extends React.Component {  
  render() {  
    return <h1>Cześć</h1>;  
  }  
}
```

Komponenty

Nazwy komponentów zawsze tworzymy z dużej litery. Jest to związane z tym, że w JSX mała litera sugeruje, że jest to znacznik HTML zamiast komponentu.

By wykorzystać dany komponent musisz go zaimportować z danego pliku i osadzić w JSX podobnie jak w przypadku znaczników HTML.

```
import { Welcome } from './Welcome'
```

```
<div className="container">  
  <Welcome />  
</div>
```


Komponenty

Z reguły każdy komponent jest odrębny plikiem z rozszerzeniem js, w którym znajduje się definicja komponentu i jego **export**, by można go później było zaimportować.

Rozróżniamy 2 rodzaje exportów: export konkretnej wartości oraz export defaultowy.

```
export function Welcome() { ... }
```

```
import { Welcome } from ' ... ';
```

```
function Welcome() { ... }
```

```
export default Welcome;
```

```
import Welcome from ' ... ';
```

Oba exporty są równoznaczne, jedynie różnią się sposobem ich importowania. W przypadku default nazwa importu może być dowolna.

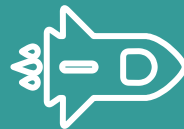
KOMPONENT FUNKCYJNY

Komponent funkcyjny

Nazywamy nimi zwykłą javascriptową funkcję, której nazwa zaczyna się z dużej litery i zwraca ona Reactowy element.

```
function Welcome() {  
  return <h1>Hello world!</h1>;  
}
```

ZADANIE



Stwórz swój pierwszy komponent funkcyjny **MyName**, który wyświetli:

“Moje imię to <TWOJE_IMIĘ>”

Dodaj go do głównego komponentu **<App />** w aplikacji, nie zapomnij o imporcie.

Komponent funkcyjny

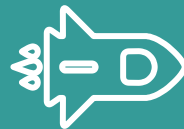
Każdy komponent funkcyjny przyjmuje pojedynczy argument “props” (który oznacza “właściwości”, z ang. *properties*), będący obiektem z danymi.

```
function Welcome(props) {  
  return <h1>Cześć, {props.name}</h1>;  
}
```

By przekazać właściwości do komponentu przekazujemy je podobnie jak atrybuty w tagach HTMLowych:

```
<Welcome name="Kamil" />
```

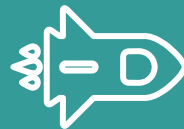
ZADANIE



Zmień komponent funkcyjny **MyName** na taki który wyświetli imię przekazane przez **props**.

Pamiętaj aby przekazać swoje imię w miejscu gdzie ten komponent jest użyty.

ZADANIE



Dodaj kolejny props zawierający Twoje nazwisko do komponentu **MyName** o nazwie *surname* i wykorzystaj go w wiadomości rozszerzając ją do wersji:

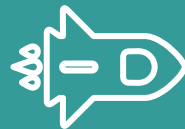
“Moje imię to <TWOJE_IMIĘ>, a nazwisko to <TWOJE_NAZWISKO>”

Komponent funkcyjny

Oczywiście komponent może zawierać w sobie logikę, na przykład jakiś warunek:

```
function Greeting(props) {  
  const isLoggedIn = props.isLoggedIn;  
  
  if (isLoggedIn) {  
    return <UserGreeting />;  
  }  
  
  return <GuestGreeting />;  
}
```


ZADANIE



Dodaj warunek do komponentu **MyName**, który w zależności od tego czy został podany props 'surname' wyświetli tekst:

Moje imię to ... albo Moje imię ... ,a nazwisko to ...

Użyj komponentu **MyName** w dwóch różnych wersjach w głównym komponencie **App**.

Stylowanie

CSS w React działają tak samo jak w przypadku zwykłego HTML'a z drobną różnicą w postaci dodawania stylów. Możemy dodać plik css lub dodać style inline.

W przypadku pliku css stwórz go i dodaj w nim stylowanie, następnie zaimportuj dany plik w komponencie, w którym chcesz użyć tych stylów.

App.css

```
.root {  
  color: red  
}
```

App.js

```
import './App.css';  
  
function App() {  
  return <div className='root'>Tekst</div>  
}
```

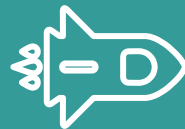
Stylowanie

Uwaga! Choć mogłoby się wydawać, że dzięki importom każdy komponent może korzystać jedynie ze styli zaimportowanych do niego to raz zaimportowane style są dostępne dla każdego komponentu.

Natomiast style inline dodajemy jako obiekt javascriptowy, gdzie dwuczłonowe nazwy zamiast rozdzielania średnikiem zamieniamy na camelCase:

```
<div style={{color: 'red', backgroundColor: 'white' }}>Tekst</div>
```

ZADANIE



Stwórz nowy komponent **Contact**, który wyświetli dane kontaktowe do Twojej osoby zgodnie z załączoną grafiką. Dane kontaktowe powinny zostać przekazane jako 1 obiekt do propsów w takiej postaci:

```
{  
  phone: '111 222 333',  
  address: { street: 'Słowackiego', city: 'Gdańsk', number: 37 },  
  email: 'moj@mail.com'  
}
```

TEL

+48 575 707 887

ADRES

ul. Słowackiego 37
Gdańsk Wrzeszcz
(obok Garnizonu)

EMAIL

kontakt@muzykon.pl

CSS Modules

Pliki CSS, w których wszystkie nazwy klas i nazwy animacji mają domyślnie zasięg lokalny. Dzieje się to poprzez automatyczne tworzenie unikatowej nazwy klasy w formacie `[nazwa pliku]_[nazwa klasy]__[hash]`. Oznacza to, że rozwiązuje problem nadpisywania nazw klas przez inne komponenty i można używać tej samej nazwy klasy dla każdego komponentu.

Vite obsługuje moduły CSS od razu przy użyciu konwencji nazewnictwa plików `[name].module.css`.

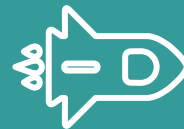
App.module.css

```
.root {  
  color: red  
}
```

App.js

```
import styles from './App.module.css';  
  
function App() {  
  return <div className={styles.root}>Tekst</div>  
}
```

ZADANIE



Przepisz Contact komponent na taki co używa CSS modules.

TEL

+48 575 707 887

ADRES

ul. Słowackiego 37
Gdańsk Wrzeszcz
(obok Garnizonu)

EMAIL

kontakt@muzykon.pl

Obsługa zdarzeń

Obsługa zdarzeń w Reakcie jest bardzo podobna do tej z drzewa DOM. Istnieje jednak kilka różnic w składni:

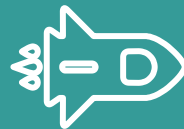
- zdarzenia reactowe pisane są camelCasem, a nie małymi literami.
- w JSX procedura obsługi zdarzenia przekazywana jest jako funkcja, a nie łańcuch znaków, czyli przekazujemy bez “()”

```
<button onClick={activateLasers}>
```

```
  Aktywuj lasery
```

```
</button>
```

ZADANIE



W nowo stworzonym komponencie **Contact** dodaj przycisk z napisem “Wyślij”, który po kliknięciu wyświetli alert z tekstem:

“Dziękuję! Zapraszam do mnie!”

Postaraj się wyświetlić w alercie dane z propsów:

“Dziękuję! Zapraszam do mnie przy <TWÓJ_ADRES>!”

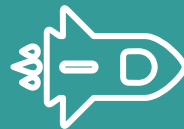
Children

Jeżeli przekażesz w JSX dzieci do naszego komponentu to są one dostępne w props pod nazwą **children**.

```
<Wrapper>  
  <div>Children</div>  
</Wrapper>
```

```
function Wrapper(props) {  
  return <div style={{ textAlign: 'center' }}>  
    {props.children}  
  </div>  
}
```

ZADANIE



Utwórz komponent Wrapper, który wyśrodkuje elementy w środku (za pomocą flexbox'a) i wyświetli przekazane do niego dzieci.

Wykorzystaj go w głównym komponencie `<App />` i otocz nim wszystkie stworzone dotychczas komponenty.

Listy i klucze

By wyświetlić listę elementów w Reakcie musimy przekształcić tę listę elementów za pomocą javascriptowej funkcji **map** w elementy JSX, zbudowaną kolekcję dodajemy do JSX w klamrach `{ }`:

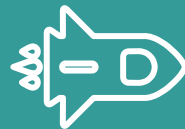
```
function Photos() {  
  const photos = ['photo1', 'photo2'];  
  
  return <ul>  
    {photos.map(photo => <li>{photo}</li>)}  
  </ul>  
}
```

Listy i klucze

Klucze (*key*) pomagają Reaktowi zidentyfikować, które elementy uległy zmianie, zostały dodane lub usunięte. Klucze powinny zostać nadane elementom wewnątrz tablicy, aby elementy zyskały stabilną tożsamość.

```
function Photos() {  
  const photos = [{ id: 1, name: 'photo1'}, { id: 2, name: 'photo2'}];  
  
  return <ul>  
    {photos.map(photo => <li key={photo.id}>{photo.name}</li>)}  
  </ul>  
}
```

ZADANIE



Stwórz komponent **MyFavouriteDishes**, który wyświetli listę Twoich ulubionych potraw (co najmniej 3). Potrawy powinny być przekazane jako props. Pamiętaj o unikalnych kluczach!

Następnie stwórz nowy komponent **AboutMe**, gdzie przeniesiesz komponenty **MyName** i **Contact** oraz umieść w nim **MyFavouriteDishes**. Dodaj go do głównego komponentu `<App />` w aplikacji.

STYLED COMPONENTS

Styled components

Jest to paczka, która daje nam sposób jak możemy ulepszyć CSS do stylizacji systemów komponentów React.

- śledzi, które komponenty są renderowane na stronie i wstrzykuje ich style i nic więcej, w pełni automatycznie
- brak błędów w nazwach klas: styled-components generuje unikalne nazwy klas dla twoich stylów
- łatwiejsze usuwanie CSS: może być trudno stwierdzić, czy nazwa klasy jest używana gdzieś w Twojej bazie kodu
- proste dynamiczne stylizowanie
- bezproblemowa konserwacja

Styled components

```
const Title = styled.h1`  
  font-size: 1.5em;  
  text-align: center;  
  color: palevioletred;  
`
```

```
return (  
  <Wrapper>  
    <Title>  
      Hello World!  
    </Title>  
  </Wrapper>  
);
```

```
const Wrapper = styled.section`  
  padding: 4em;  
  background: papayawhip;  
`
```


Styled components

Możesz przekazać funkcję ("interpolacje") do literału szablonu komponentu stylizowanego, aby dostosować go na podstawie jego propsów.

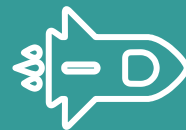
```
const Button = styled.button`  
  background: ${props => props.primary ? "palevioletred" : "white"};  
  color: ${props => props.primary ? "white" : "palevioletred"};  
  font-size: 1em;  
`;  
  
return (  
  <div>  
    <Button primary>Primary</Button>  
  </div>  
)
```

Styled components

Aby łatwo stworzyć nowy komponent, który dziedziczy styl innego, po prostu zapakuj go w konstruktor **styled()**.

```
const TomatoButton = styled(Button)`  
  color: tomato;  
  border-color: tomato;  
`;  
  
return (  
  <div>  
    <Button>Normal Button</Button>  
    <TomatoButton>Tomato Button</TomatoButton>  
  </div>  
);
```

ZADANIE



Stwórz nowy komponent **ContactStyled**, który będzie dokładną kopią komponentu **Contact**, ale wykonanego z użyciem styled-components.

TEL

+48 575 707 887

ADRES

ul. Słowackiego 37
Gdańsk Wrzeszcz
(obok Garnizonu)

EMAIL

kontakt@muzykon.pl

HOOKS

Hooki

Czym jest hook?

Hook jest specjalną funkcją, która pozwala „zahaczyć się” w wewnętrzne mechanizmy Reacta. Jego nazwa zaczyna się od use*. Działa jedynie w komponentach funkcyjnych od wersji React 16.8.

Prezentacja hooków na ReactConf: <https://youtu.be/dpw9EHDh2bM?t=686>

Stan

W komponentach Reactowych możemy zapisywać lokalny stan. W przypadku komponentów funkcyjnych wykorzystujemy hooki by to osiągnąć:

```
const [value, setValue] = useState(0)
```

value – wartość stanu

setValue – setter, ustawia wartość stanu

argument funkcji `useState` to początkowa wartość stanu (w tym przypadku 0)

Stan

Przykład użycia useState:

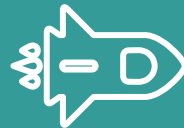
```
import React, { useState } from 'react';

const Component = () => {
  const [value, setValue] = useState(0);

  const increase = () => setValue(value + 1);

  return <>
    <h1>{value}</h1>
    <button onClick={increase}>increase</button>
  </>
}
```

ZADANIE



Stwórz komponent **Game**, który wyświetli:

“Witam w grze <NAZWA_GRY>”, gdzie nazwa gry zostaje przekazana przez props.

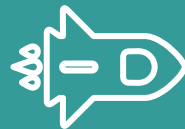
Pod przywitaniem wyświetl informacje:

“Twoja liczba punktów to: <PUNKTY>”,

gdzie punkty zostają dostarczone przez stan komponentu i na początku wynoszą 0.

Dodaj go do głównego komponentu `<App />` w aplikacji.

ZADANIE



Zmodyfikuj komponent **Game** poprzez dodanie przycisku, który po kliknięciu zwiększy ilość punktów o 5 oraz przycisk, który zmniejszy tą wartość o 5.

Gdy gracz będzie miał wartość punktów poniżej zera to punktacja powinna mieć kolor czerwony.

Effect

Czasami chcemy uruchomić jakiś dodatkowy kod po tym, jak React zaktualizuje drzewo DOM. Zapytania sieciowe, ręczna modyfikacja drzewa DOM czy logowanie. Możemy to osiągnąć przy pomocy kolejnego hooka: **useEffect**.

Poprzez użycie tego hooka mówisz Reactowi, że twój komponent musi wykonać jakąś czynność po jego wyrenderowaniu. React zapamięta funkcję, którą przekazano do hooka (będziemy odtąd odnosić się do niej jako naszego „efektu”), a potem wywoła ją, gdy już zaktualizuje drzewo DOM.

Effect

Przykład użycia useEffect:

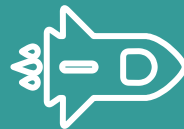
```
import React, { useState, useEffect } from 'react';

function Example() {
  const [count, setCount] = useState(0);

  useEffect(() => {    document.title = `Kliknięto ${count} razy`;  });

  return <div>
    <p>Kliknięto {count} razy</p>
    <button onClick={() => setCount(count + 1)}>
      Kliknij mnie
    </button>
  </div>
}
```

ZADANIE



Wykorzystując hook **useEffect** w komponencie Game tak by przy osiągnięciu 50 punktów wyświetlił się alert:

“Gratulacje! Wygrałeś w grę <NAZWA_GRY>”, gdzie nazwa gry jest wartością z propsów.

Effect

Jednak czasami chcemy by dany efekt wywołał się tylko raz - by to osiągnąć możemy dodać do **useEffect** drugi argument, którym jest tablica zależności. Co to oznacza? Jeśli wartość count ma wartość 5, a nasz komponent jest ponownie renderowany z count wciąż równym 5, React porówna [5] z poprzedniego renderowania i [5] z kolejnego renderowania. Ponieważ wszystkie elementy w tablicy są takie same (5 === 5), React pominie efekt.

```
useEffect(() => {  
  document.title = `Kliknięto ${count} razy`;  
}, [count]);
```

// Uruchom ponownie efekt tylko wtedy, gdy zmieni się wartość count

Effect

Zatem jeżeli chcemy wykonać daną operacją tylko raz po pierwszym wyrenderowaniu się komponentu to musimy przekazać w **useEffect** pustą tablicę zależności:

```
useEffect(() => {  
  console.log('Komponent się zamontował');  
}, []);
```

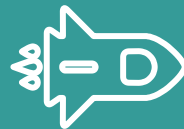
// Uruchom efekt tylko po pierwszym renderze

Pobieranie danych

Pobieranie danych w Reakcie najlepiej jak będzie odbywało się po pierwszym renderze komponentu. Dlaczego? Użytkownik dostaje już informację, że aplikacja działa i ewentualnie może coś zmienić w przypadku złego kliknięcia + możemy na przykład pokazać stan ładowania.

```
useEffect(() => {  
  fetch('.../posts').then(r => r.json()).then(posts => {  
    console.log('Pobrano: ' + posts);  
  });  
}, []);
```

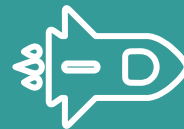
ZADANIE



Stwórz komponent **Users** i wykonaj w nim pobieranie z adresu:
<https://jsonplaceholder.typicode.com/users>.

Pobrane dane wyświetl w postaci listy:
"{name} {surname} works in {companyName}".

ZADANIE



Elementy listy w komponencie **Users** zamień na kolejny komponent o nazwie **User**, który otrzyma dane o użytkowniku w postaci props'a / ów.

Następnie dodaj w nim logikę aby tooltip (atrybut *title*) się wyświetlał po najechaniu na nazwę firmy (wykorzystaj *catchPhrase* z pobranych danych).

Effect

Jednak czasami chcemy coś posprzątać po komponencie lub chcemy wykonać jakąś akcję na koniec. W takim wypadku w funkcji dodanej do hook'a musimy zwrócić nową funkcję, która wywoła się na koniec efektu przy każdym kolejnym renderowaniu.

```
useEffect(() => {  
  ChatAPI.subscribe(props.id, handleStatusChange);  
  
  return () => {  
    ChatAPI.unsubscribe(props.id, handleStatusChange);  
  };  
});
```

Hooks

Każdy hook możesz wykorzystać wiele razy na przykład możesz użyć **useState** więcej niż raz by zapisać więcej niż 1 pole w stanie.

Zatem możesz też używać wielu efektów. Pozwala to na wydzielenie niepowiązanej logiki na osobne efekty.

Hooki pozwalają na dzielenie kodu na mniejsze fragmenty pod względem ich odpowiedzialności, a nie ze względu na nazwę metody cyklu życia. React wywoła *każdy* efekt użyty w komponencie w takiej kolejności, w jakiej został dodany.

Zaawansowane hooks

useCallback zwraca zapamiętaną funkcję zwrotną, która zmieni się tylko wtedy, gdy zmieni się któraś z zależności podanej w tablicy jako drugi argument.

```
const memoizedCallback = useCallback(() => { doSomething(a, b) }, [a, b]);
```

useMemo zwraca zapamiętaną wartość, która zostanie obliczona ponownie wartość tylko wtedy, gdy zmieni się któraś z zależności podanej w tablicy jako drugi argument. Ta optymalizacja pozwala uniknąć kosztownych obliczeń przy każdym renderowaniu.

```
const memoizedValue = useMemo(() => computeExpensiveValue(a, b), [a, b]);
```

KOMPONENT KLASOWY

Komponent klasowy

Do zdefiniowania komponentu możesz również użyć klasy ze standardu ES6. Z punktu widzenia Reacta nie ma różnicy pomiędzy nimi. Jednak różnią się od komponentów funkcyjnych w kilku kwestiach składniowych oraz w kwestii wydajności.

```
class Welcome extends React.Component {  
  render() {  
    return <h1>Hello world!</h1>;  
  }  
}
```

Uwaga! W komponentach klasowych jest wymagana funkcja **render**, która powinna zwrócić kawałek kodu JSX lub null.

Komponent klasowy - props

W przypadku klasy nie mamy dostępu do argumentów funkcji, więc propsy w komponencie klasowym odczytujemy poprzez referencje do **this**.

```
class Welcome extends React.Component {  
  render() {  
    return <h1>Cześć, {this.props.name}</h1>;  
  }  
}
```

Komponent klasowy - stan

Tak jak w przypadku komponentu funkcyjnego można zdefiniować lokalny stan w komponencie klasowy. Stan w komponencie klasowym zawsze jest obiektem.

Stan można zdefiniować na 2 sposoby

1. definicja w construttorze klasy
2. za pomocą skróconego zapisu - przez propercje klasy

Podobnie jak w przypadku propsów stan w komponencie klasowym odczytujemy poprzez referencje do **this**.

Komponent klasowy - stan

definicja w construtctorze:

```
class Score extends React.Component {  
  constructor(props) {  
    super(props);  
    this.state = { value: 10 }  
  }  
  render() {  
    return <h2>Wynik: {this.state.value}</h2>  
  }  
}
```

Komponent klasowy - stan

za pomocą skróconej wersji:

```
class Score extends React.Component {  
  state = { value: 10 }  
  
  render() {  
    return <h2>Wynik: {this.state.value}</h2>  
  }  
}
```

Komponent klasowy - stan

Stan możemy zmienić za pomocą metody `this.setState`, która przyjmuje jako argument albo obiekt, który zostanie scalony z aktualnym stanem (*albo funkcję, która jako argumenty przyjmuje state oraz props i zwróci zmieniony stan*).

```
this.setState({comment: 'Witam'});
```

Komponent klasowy - stan

Przykład użycia lokalnego stanu w komponencie klasowym:

```
class Score extends React.Component {  
  state = { value: 10 }  
  increase = () => this.setState({ value: this.state.value + 1 });  
  
  render() {  
    return <>  
      <h2>Wynik: {this.state.value}</h2>  
      <button onClick={this.increase}>increase</button>  
    </>  
  }  
}
```

Zwróć uwagę, że `increase` jest metodą klasy i by się do niej dostać trzeba poprzedzić wszystko zapisem **this.** .

Komponent klasowy - stan

Uwagi odnośnie stanu i metod:

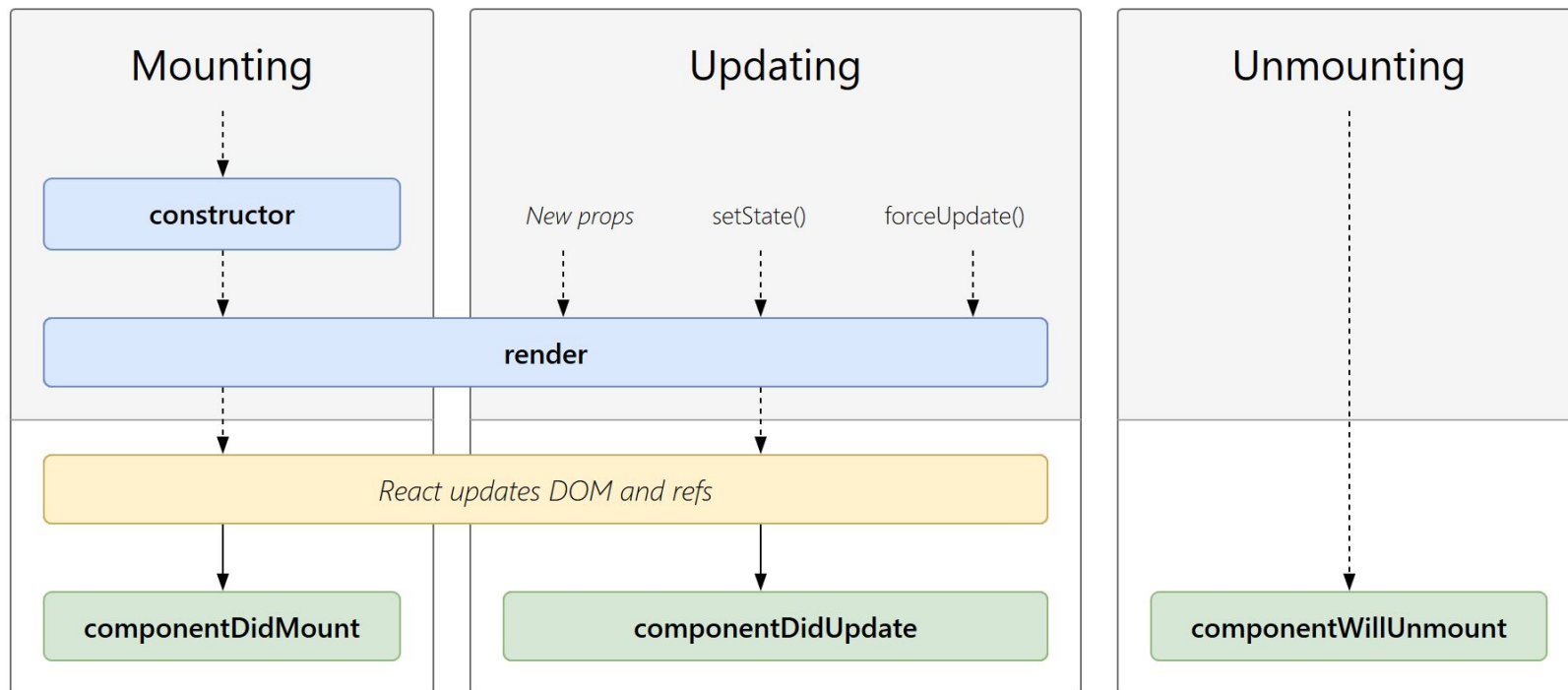
1. Nie modyfikuj stanu bezpośrednio, ponieważ taka zmiana nie spowoduje ponownego wyrenderowania komponentu.
2. Operacja `this.setState` jest asynchroniczna - zatem nie zostanie od razu wykonana, by mieć pewność, że coś zostanie wykonane po zmianie stanu to możemy dodać funkcję jak drugi argument, która zostanie wykonana zaraz po zmianie stanu:

```
this.setState({comment: 'Witam'}, () => { console.log(this.state) });
```

Więcej tutaj: <https://pl.reactjs.org/docs/faq-state.html>

3. Metody w komponencie klasowy mogą być zapisywane jako arrow function by uniknąć pisania **bind** tej metody w constructorze. Więcej tutaj:
<https://pl.reactjs.org/docs/faq-functions.html>

Cykl życia



Cykl życia

Metody cyklu życia można wykorzystać w ten sposób:

```
class Clock extends React.Component {  
  state = { date: new Date() }  
  
  componentDidMount() { }  
  
  componentDidUpdate() { }  
  
  componentWillUnmount() { }  
  
  render() {  
    return <h2>Aktualny czas: {this.state.date.toLocaleTimeString()}.</h2>  
  }  
}
```

Cykl życia a useEffect

```
class Example extends React.Component {  
  componentDidMount() {  
    console.log("I am mounted!");  
  }  
  render() {  
    return null;  
  }  
}
```

```
function Example() {  
  useEffect(() => {  
    console.log("mounted");  
  }, []);  
  return null;  
}
```


Cykl życia a useEffect

```
class Example extends React.Component {  
  componentDidMount() {  
    console.log("I am here!");  
  }  
  
  componentDidUpdate() {  
    console.log("I am here!");  
  }  
  
  render() {  
    return null;  
  }  
}
```

```
const Example = () => {  
  useEffect(() => {  
    console.log("I am here!");  
  })  
}
```

Cykl życia a useEffect

```
class Example extends React.Component {  
  componentWillUnmount() {  
    console.log("Bye, bye...");  
  }  
  
  render() {  
    return null;  
  }  
}
```

```
const Example = () => {  
  useEffect(() => {  
    return () => {  
      console.log("Bye, bye...");  
    };  
  });  
};
```

ZADANIE



Przepisz komponent Game na komponent klasowy.

Dokumentacja: <https://pl.reactjs.org/docs/state-and-lifecycle.html>

Formularze

W Reakcie elementy formularza HTML działają trochę inaczej niż pozostałe elementy DOM. Wynika to stąd, że elementy formularza same utrzymują swój wewnętrzny stan.

Natomiast w Reakcie zmienny stan komponentu jest zazwyczaj przechowywany we właściwości `state` danego komponentu. Jest on aktualizowany jedynie za pomocą funkcji `setterów`.

Formularze

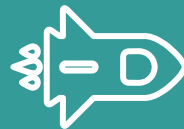
Możliwe jest łączenie tych dwóch rozwiązań poprzez ustanowienie reactowego stanu jako “wyłączniego źródła prawdy”. Wówczas reactowy komponent renderujący dany formularz kontroluje również to, co zachodzi wewnątrz niego podczas wypełniania pól przez użytkownika.

Element input formularza, kontrolowany w ten sposób przez Reacta, nazywamy “komponentem kontrolowanym”

Formularze

```
const NameForm = () => {  
  const [name, setName] = useState("");  
  
  const handleChange = (event) => {    setName(event.target.value); }  
  const handleSubmit = (event) => {  
    event.preventDefault();  
    alert('Podano następujące imię: ' + name);  
  }  
  
  return (  
    <form onSubmit={handleSubmit}>  
      <label> Imię:  
        <input type="text" value={name} onChange={handleChange} />  
      </label>  
      <input type="submit" value="Wyślij" />  
    </form>  
  );  
}
```

ZADANIE



Utwórz komponent z formularzem o nazwie **MyForm**, w którym użytkownik będzie mógł zapisać dane takie jak: nazwa, wiek, wybrać z listy rozwijanej płeć, wpisać komentarz.

Po wciśnięciu przycisku “wyślij” wyświetlić podane informacje w alercie. Strona nie powinna się przeładować.

Formularze

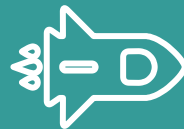
W Reakcie istnieją wiele zewnętrznych paczek ułatwiających pracę z formularzami:

- formik (<https://formik.org/>)
- react hook form (<https://react-hook-form.com/>)
- **react final forms** (<https://final-form.org/react>)

Upraszczają one codzienną pracę z formularzami, ułatwiając zarządzanie wartościami w formularzu, walidacją i jego stanem wewnętrznym. Dzięki bibliotece możesz w prosty sposób:

- podpiąć customowe walidacje które podejmiemy tak prosto jak walidacje natywne,
- podpiąć walidacje asynchroniczne
- sygnalizować użytkownikowi, które pola zostały już przez niego wypełnione
- obsługiwać tablice i obiekty jako pola formularza

ZADANIE



Utwórz komponent z poprzedniego zadania z użyciem jednej z wybranych paczek pod nazwą **MyBoostedForm**.

Pamiętaj by wpierw zainstalować daną paczkę.



Zod

Zod

Zod to biblioteka do deklarowanie dowolnego typu danych (schematu) przy użyciu TypeScript i sprawdzania jego poprawności.

Zod został zaprojektowany tak, aby był jak najbardziej przyjazny dla programistów. Celem jest wyeliminowanie zduplikowanych deklaracji typu.

Kilka innych aspektów:

- Zero zależności
- Działa w Node.js i we wszystkich nowoczesnych przeglądarkach
- Mały: 8kb
- Niemutowalne metody
- Zwięzły, łańcuchowy interfejs
- Działa również ze zwykłym JavaScriptem! Nie musisz używać TypeScript.

Zod

```
import { z } from "zod";
```

```
const mySchema = z.string();
```

```
mySchema.parse("tuna"); // => "tuna"
```

```
mySchema.parse(12); // => throws ZodError
```

```
mySchema.safeParse("tuna"); // => { success: true; data: "tuna" }
```

```
mySchema.safeParse(12); // => { success: false; error: ZodError }
```

Zod

```
import { z } from "zod";

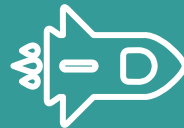
const User = z.object({
  username: z.string(),
});

User.parse({ username: "Ludwig" });

// extract the inferred type
type User = z.infer<typeof User>;

// { username: string }
```

ZADANIE



Dodaj walidację do **MyForm** z użyciem Zod.

Nazwa: required, string

Wiek: required, number

Płeć: required, male lub female

Komentarz: optional, string

Pamiętaj by wpierw zainstalować odpowiednie paczki.

REACT UI COMPONENTS

UI Components

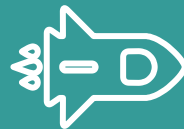
Pracując w Reactcie rzadko kiedy piszemy własne stylowanie. Korzystamy z czegoś co nazywamy UI Components, czyli komponenty odpowiadające za wygląd. W sieci jest wiele dostępnych darmowych paczek z takimi komponentami:

- material ui
- semantic ui
- react bootstrap

i wiele wiele innych:

<https://www.codeinwp.com/blog/react-ui-component-libraries-frameworks/>

ZADANIE



Wykorzystując jedną z paczek zmień komponenty **AboutMe** (bez Contact) oraz **Game**, tak by używał komponentów już z przygotowanymi wcześniej stylami.

Pamiętaj by wpierw zainstalować paczkę danych komponentów.

REACT ROUTER

React router

Routing po stronie serwera jest bardzo powszechną metodą obsługi routingu. **Nie jest to jednak część React Routera.** Jak to działa?

1. Użytkownik klika link na stronie internetowej.
2. Na ekranie pojawia się zupełnie inna strona. Ścieżka URL jest aktualizowana.
3. Routing po stronie serwera powoduje, że strona się odświeża, ponieważ wysuwamy kolejne żądanie do serwera, a on daje nam kompletnie nowy content do wyświetlenia.

Jeśli nowa strona nadal będzie miała informacje z nagłówka i stopki umieszczone na ekranie, to dlaczego mielibyśmy ponownie je ładować? Jednak serwer tego nie wie, dlatego ponownie je załaduje, mimo że nie musi.

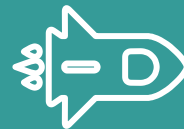
React router

Routing po stronie klienta polega na wewnętrznej obsłudze przekazów przez JS, który jest renderowany na stronie front endu. Jak to działa?

1. Użytkownik klika link na stronie internetowej.
2. Ścieżka URL jest aktualizowana. Na ekranie aktualizuje się tylko wymagana część strony.
3. Routing po stronie klient powoduje, że strona się nie odświeża, a jedynie wykonuje kolejne zapytania do serwera.

Dzięki temu uzyskujemy SPA, gdzie nie musimy ładować wielu stron. Ładujemy za to początkowe żądanie z naszymi początkowymi plikami HTML, CSS i JS z serwera. Używamy routingu po stronie klienta, aby zachować wszystkie zalety routing.

ZADANIE



Wykorzystując jedną z paczek podanych wyświetl nawigację dla naszej strony, w której dodasz przyciski: Home, About me, Forms, Game, Users.

React router

Mam dwa rodzaje paczek dla react-router'a:

- react-router
- react-router-dom

react-router-dom jest nadzbiorem react-router'a z przygotowanymi komponentami do pracy w przeglądarce

Router

Mam dwa typy routerów:

- BrowserRouter - dla dynamicznych zapytań
- HashRouter - dla statycznych requestów

W większości przypadków interesuje nas **BrowserRouter** ze względu na aplikacje typu SPA.

BrowserRouter jest komponentem, którym musi “owrapować” (okalać) naszą aplikację lub jej część w której chcemy skorzystać z routingu.

Routes, Route, Link, NavLink

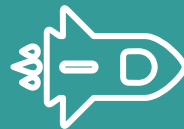
By stworzyć routing przy pomocy react-router-dom musimy jeszcze poznać kilka innych komponentów:

- **Routes** - zmienia odpowiednio widok bazując na url, znajduje wśród swoich dzieci odpowiedni komponent Route i go renderuje
- **Route** - przyjmuje ścieżkę jako props i zawiera zawartość, która ma się wyświetlić jak ścieżka pasuje do url
- **Link** - tworzy link w aplikacji (<a> tag), który może zmienić url
- **NavLink** - specjalny komponent Link, który może przyjąć odpowiednie style jak jest aktywna ścieżka do którego prowadzi

React router dom

```
import { BrowserRouter as Router, Routes, Route, Link } from "react-router-dom";  
function App() {  
  return <Router>  
    <nav> <ul>  
      <li> <Link to="/">Home</Link> </li>  
      <li> <Link to="/about">About</Link> </li>  
      <li> <Link to="/users">Users</Link> </li>  
    </ul> </nav>  
    <Routes>  
      <Route path="/about" element={<About />} />  
      <Route path="/users" element={<Users />} />  
      <Route path="/" element={<Home />} />  
    </Routes>  
  </Router>  
}
```

ZADANIE



Dodaj **BrowserRouter** do naszej aplikacji - “owrapuj” aplikację w `index.js`.

Dodaj **Routes** z odpowiednimi **Route**’ami, które będą prowadzić odpowiednio do Home, About me, Game, Forms, Users. Pod ścieżką *Home* wyświetl przywitanie “*Witaj na naszej stronie!*”.

Sprawdź czy po wprowadzeniu odpowiedniej ścieżki widok się zmienia.

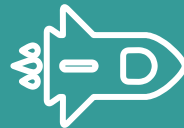
ZADANIE



Użyj komponentu **Link** w nawigacji by po kliknięciu w jej element prowadziła nas pod odpowiedni url.

Sprawdź czy po kliknięciu elementu nawigacji zmienia się ścieżka url i widok strony.

ZADANIE



Zamień **Link** na **NavLink** i użyj w nim props'a **isActive** przekazanego do `className` i `style`. W przypadku `className` Pamiętaj by stworzyć i zaimportować klasę css, która będzie odpowiednio stylować aktywny link.

Link do dokumentacji: <https://reactrouter.com/en/main/components/nav-link>

Sprawdź jak zmienia się nawigacja po wpisaniu ręcznie ścieżki lub po kliknięciu w nawigację.

React router props

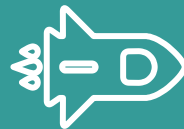
React router dostarcza nam propsy, które możemy wykorzystać w naszej aplikacji:

- params - parametry url
- location - reprezentuje aktualny url
- history - historia przeglądarki, która można użyć do nawigacji strony

By mieć do nich dostęp musimy wykorzystać odpowiednie hooki:

- useParams
- useLocation
- useNavigate

ZADANIE



Stwórz nowy komponent **Sign**, wykorzystaj go w routing pod ścieżką `/sign`.

Dodaj w nim formularz logowania: email i hasło z przyciskiem zaloguj.

Po kliknięciu Zaloguj wykonaj przekierowanie na stronę główną `/home` za pomocą hooka **useNavigate**.

React router props

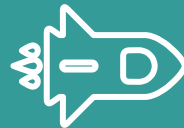
W url możemy również przetrzymywać dane. By je odczytać musimy odpowiednio zdefiniować ścieżkę:

```
<Routes>  
  <Route path="/user/:username" element={<User />} />  
</Routes>
```

I następnie możemy łatwo z użyciem hook'a odczytać te parametry.

```
function User(props) {  
  const params = useParams();  
  
  return <h1>Hello {params.username}!</h1>;  
}
```

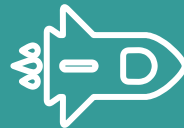
ZADANIE



Stwórz nowy komponent **UserDetails**, który będzie renderował się na ścieżce: **path="/users/:id"**. Będzie na podstawie ścieżki wyświetlał informację: *"Witaj użytkowniku o id: <ID_Z_URL>"*.

Zacznij od stworzenia komponentu i wyświetlenia go pod daną ścieżką, a następnie postaraj się wyświetlić podane w url id.

ZADANIE

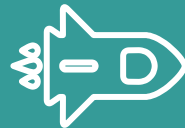


Dodaj w **UserDetails** pobieranie danych użytkownika na podstawie odczytanego id z hook'a. Wyświetl jego imię i nazwisko.

Dane możesz pobrać z adresu:

<https://jsonplaceholder.typicode.com/users/1> , gdzie id to cyfra podana na końcu.

ZADANIE



Zmień w komponencie **Users** elementy listy na elementy klikalne, które będą linkami do ścieżki gdzie znajduje się **UserDetails** pod odpowiednim url.

Wykorzystaj do tego komponent **Link** z react-router-dom.

DZIELENIE KODU

DZIELENIE KODU

Wraz ze wzrostem objętości kodu twojej aplikacji, rośnie również jej objętość a co za tym idzie czas ładowania. Aby uniknąć problemu zbyt dużego pakietu, warto już na początku o tym pomyśleć i rozpocząć “dzielenie” swojej paczki. Dzielenie kodu to funkcja wspierana przez narzędzia takie jak Webpack czy Vite, które mogą tworzyć wiele pakietów doładowywanych dynamicznie w czasie wykonania kodu aplikacji.

Dzielenie kodu twojej aplikacji ułatwi ci użycie “leniwego ładowania” do wczytywania jedynie tych zasobów które są aktualnie wymagane przez użytkownika zasobów, co może znacznie poprawić wydajność twojej aplikacji. Mimo że nie zmniejszysz w ten sposób sumarycznej ilości kodu, unikniesz ładowania funkcjonalności zbędnych dla użytkownika w danym momencie, co przełoży się na mniejszą ilość kodu do pobrania na starcie aplikacji.

IMPORT

Najprostszym sposobem na wprowadzenie podziału kodu do twojej aplikacji jest użycie dynamicznego wariantu funkcji **import()**.

```
import { add } from './math';
```

```
console.log(add(16, 26));
```

```
import("./math").then(math  
=> {  
  console.log(math.add(16,  
26));  
});
```

Gdy Webpack/Vite natknie się na taką składnię, automatycznie zacznie dzielić kod w twojej aplikacji.

REACT LAZY

Funkcja `React.lazy` pozwala renderować dynamicznie importowane komponenty jak zwykłe komponenty.

Przed:

```
import OtherComponent from './OtherComponent';
```

Po:

```
const OtherComponent = React.lazy(() => import('./OtherComponent'));
```

Powyższy kod automatycznie załaduje paczkę zawierającą `OtherComponent` podczas pierwszego renderowania komponentu. `React.lazy` jako argument przyjmuje funkcję, która wywołuje dynamiczny `import()`.

REACT SUSPENSE

“Lazy” komponent powinien zostać wyrenderowany wewnątrz Suspense, dzięki któremu na czas ładowania możemy wyświetlić komponent zastępczy (np. wskaźnik ładowania) poprzez props fallback tego komponentu, który akceptuje dowolny element reactowy.

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

```
const OtherComponent = React.lazy(() => import('./OtherComponent'));
```

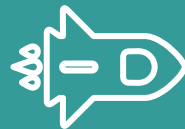
```
function MyComponent() {  
  return (  
    <Suspense fallback={<div>Wczytywanie...</div>}>  
      <OtherComponent />  
    </Suspense>  
  );  
}
```

REACT SUSPENSE

Decyzja o tym, w których miejscach podzielić kod aplikacji, może okazać się kłopotliwa. Zależy ci na miejscach, że wybierasz miejsca, które równomiernie podzielą twoje pakiety, ale nie chcesz zepsuć doświadczeń użytkownika.

Dobrym punktem startowym są ścieżki (ang. routes) w aplikacji. Większość ludzi korzystających z Internetu przyzwyczajona jest, że przejście pomiędzy stronami zajmuje trochę czasu. Dodatkowo, zazwyczaj podczas takiego przejścia spora część ekranu jest renderowana ponownie. Można więc założyć, że użytkownik nie będzie wykonywał żadnych akcji na interfejsie podczas ładowania.

ZADANIE



Wykorzystaj “leniwe” ładowanie poprzez użycie React **lazy** i **Suspense** podczas ładowania komponentów na nowej ścieżce w react routerze.

Dodaj **Spinner** podczas ładowania strony na środku ekranu.

CONTEXT

Context

Kontekst umożliwia przekazywanie danych wewnątrz drzewa komponentów bez konieczności przekazywania ich przez właściwości każdego komponentu pośredniego.

W typowej aplikacji reactowej dane przekazywane są z góry w dół (od rodzica do dziecka) poprzez właściwości. Może się to jednak okazać zbyt uciążliwe w przypadku niektórych danych i bardzo by obciążyło wydajność aplikacji poprzez za dużo ilość przeładowań. By to rozwiązać stworzono konteksty do współdzielenia danych, które można uznać za “globalne” dla drzewa komponentów, takich jak informacje o zalogowanym użytkowniku, schemat kolorów czy preferowany język.

Context

Kontekst można stworzyć za pomocą metody: **React.createContext**. Metoda ta tworzy obiekt kontekstu. Gdy React renderuje komponent, który zasubskrybował się do tego kontekstu, będzie przekazywać mu aktualną wartość z najbliższego “dostawcy” (Provider) powyżej w drzewie.

```
const MyContext = React.createContext(defaultValue);
```

Stworzony obiekt kontekstu składa się z dwóch elementów: **Provider** (“dostawca”) i **Consumer** (“konsument”).

Uwaga! Argument `defaultValue` jest używany **tylko** gdy komponent odczytujący z kontekstu nie ma nad sobą żadnego dostawcy.

Context

Każdy obiekt kontekstu posiada własny komponent dostawcy **Provider**, który pozwala komponentom odczytującym na zasubskrybowanie się na zmiany w tym kontekście.

`<MyContext.Provider value={/* jakaś wartość */}>`

Wartość przekazana przez dostawcę we właściwości `value` będzie trafiała do “konsumentów” **Consumer** tego kontekstu znajdujących się poniżej w drzewie. Jeden dostawca może być połączony z wieloma konsumentami.

Wszyscy konsumenci znajdujący się poniżej dostawcy będą ponownie renderowani przy każdej zmianie właściwości **value**.

Context

Przykład tworzenia kontekstu:

```
const TextColorContext = React.createContext('black');
```

```
const App = () => {  
  const [textColor, setTextColor] = useState('black');  
  
  return <>  
    <TextColorContext.Provider value={textColor}>  
      <Content />  
    </TextColorContext.Provider>  
    <button onClick={() => setTextColor('red')}>Red color</button>  
    <button onClick={() => setTextColor(black)}>Black color</button>  
  </>  
}
```

Context

`createContext` również generuje komponent konsumenta **Consumer**, który subskrybuje się na zmiany w kontekście, czyli nasłuchuje na zmiany w kontekście.

Potomkiem **Consumer'a** musi być funkcja. Funkcja ta otrzymuje aktualną wartość z kontekstu i zwraca węzeł `reactowy`. Argument `value` przekazany do tej funkcji będzie równy właściwości `value` najbliższego dostawcy tego kontekstu powyżej w drzewie.

Kontekst zamiast z użyciem komponentu `Consumer'a` może być odczytany również za pomocą hook'a **`useContext`**.

Jeśli ponad komponentem nie ma żadnego dostawcy, zostanie użyta wartość `defaultValue` przekazana do `createContext()`.

Context

Przykład odczytania kontekstu z użyciem hook'a:

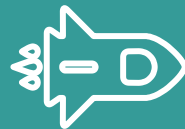
```
const ThemedText = (props) => {  
  const textColor = useContext(TextColorContext);  
  
  return (  
    <span style={{color: textColor}}>  
      Ten tekst ma kolor z kontekstu  
    </span>  
  );  
}
```


Context

Przykład odczytania kontekstu z użyciem komponentu Consumer'a:

```
const ThemedText = (props) => (  
  <TextColorContext.Consumer>  
    {value => <span style={{color: value}}>Ten tekst ma kolor z kontekstu</span> }  
  </TextColorContext.Consumer>  
)
```

ZADANIE



W naszej aplikacji stwórz kontekst o nazwie **ThemeContext**, który będzie dostarczał dwa motywy - czarny i jasny (lub inne). W nawigacji powinny znaleźć się dwa przyciski służące do zmiany motywu, a aplikacja przy zmianie powinna zmieniać tło i kolor treści.

REACT QUERY

React Query

Uproszczona obsługa stanu danych w aplikacjach React

W dzisiejszych czasach wiele aplikacji internetowych wymaga ładowania danych z serwera. Często ta operacja jest powtarzana wielokrotnie, co prowadzi do spowolnienia działania aplikacji i obciążenia serwera. React Query to biblioteka, która pozwala na łatwe pobieranie, cache'owanie i synchronizowanie danych w aplikacjach React.

- Uproszczenie obsługi stanu danych w aplikacji
- Automatyczne odświeżanie danych
- Przechowywanie danych w pamięci podręcznej
- Obsługa błędów
- Synchronizacja danych między różnymi komponentami

React Query

Uproszczona obsługa stanu danych w aplikacjach React

W dzisiejszych czasach wiele aplikacji internetowych wymaga ładowania danych z serwera. Często ta operacja jest powtarzana wielokrotnie, co prowadzi do spowolnienia działania aplikacji i obciążenia serwera. React Query to biblioteka, która pozwala na łatwe pobieranie, cache'owanie i synchronizowanie danych w aplikacjach React.

- Uproszczenie obsługi stanu danych w aplikacji
- Automatyczne odświeżanie danych
- Przechowywanie danych w pamięci podręcznej
- Obsługa błędów
- Synchronizacja danych między różnymi komponentami

React Query

React Query automatycznie przechowuje dane w pamięci podręcznej, co pozwala na szybszy dostęp do tych danych przy kolejnych żądaniach. Można również skonfigurować czas trwania cache'owania.

```
const { isLoading, error, data } = useQuery('posts', async () => {  
  const response = ...  
}, {  
  cacheTime: 10000 // cache for 10 seconds  
});
```

Oferuje również łatwe sposoby na obsługę błędów podczas pobierania danych. Można skonfigurować automatyczne ponawianie prób pobrania danych w przypadku błędu.

```
const { isLoading, error, data } = useQuery('posts', async () => {  
  const response = ...  
}, {  
  retry: 3 // retry up to 3 times on error  
});
```

React Query

React Query umożliwia łatwe modyfikowanie danych i wysyłanie ich na serwer. Można skonfigurować automatyczne aktualizowanie tych danych na stronie klienta.

```
const { mutate } = useMutation(async (newPost) => {  
  const response = await fetch('https://jsonplaceholder.typicode.com/posts', {  
    method: 'POST',  
    headers: {  
      'Content-Type': 'application/json'  
    },  
    body: JSON.stringify(newPost)  
  });  
  await response.json();  
}, {  
  onSuccess: () => queryClient.invalidateQueries('posts')  
});
```

React Query

React Query pozwala na łatwą synchronizację danych między różnymi komponentami. Wystarczy skorzystać z hooka `useQueryClient` i wywołać metodę `setQueryData`.

```
const queryClient = useQueryClient();
```

```
const handleDelete = async (id) => {  
  await fetch(`https://jsonplaceholder.typicode.com/posts/${id}`, {  
    method: 'DELETE'  
  });  
  queryClient.setQueryData('posts', (oldData) => oldData.filter((post) => post.id !== id));  
};
```


useQuery

Parametr **queryKey** jest wymagany i określa klucz, pod którym zostaną zcache'owane pobrane dane. Klucz może być łańcuchem znaków lub tablicą, jeśli zapytanie wymaga przekazania parametrów.

Parametr **queryFunction** jest również wymagany i określa funkcję, która będzie wywoływana w celu pobrania danych. Funkcja ta może zwracać dane bezpośrednio lub obiekt Promise z danymi.

Parametr options jest opcjonalny i pozwala na dostosowanie różnych parametrów, takich jak czas trwania cache'owania, re-fetching, obsługa błędów itp.

Wynikiem wywołania hooka useQuery są trzy zmienne:

- **isLoading** - wartość logiczna, która określa, czy zapytanie jest w trakcie wykonywania,
- **error** - obiekt błędu, który jest zwracany, jeśli zapytanie zakończy się niepowodzeniem,
- **data** - dane zwrócone przez zapytanie.

```
const { isLoading, error, data } = useQuery(queryKey, queryFunction, options);
```

useMutation

Parametr **mutationFunction** jest wymagany i określa funkcję, która będzie wywoływana w celu wysłania żądania. Funkcja ta może zwracać dane bezpośrednio lub obiekt Promise z danymi.

Parametr **options** jest opcjonalny i pozwala na dostosowanie różnych parametrów, takich jak dostępność mutacji, obsługa błędów, zwalnianie lock'a itp.

Wynikiem wywołania hooka useMutation są dwie zmienne:

- **mutate** - funkcja, która wywołuje mutację,
- **isLoading** - wartość logiczna, która określa, czy mutacja jest w trakcie wykonywania,
- **error** - obiekt błędu, który jest zwracany, jeśli mutacja zakończy się niepowodzeniem,
- **data** - dane zwrócone przez mutację.

Po wywołaniu funkcji **mutate**, React Query wykonuje mutację i aktualizuje cache w zależności od ustawień. Na przykład, gdy mutacja typu POST zostanie wykonana, biblioteka może automatycznie dodać nowo utworzony obiekt do cache'a bez potrzeby dodatkowego pobierania danych z serwera. W przypadku mutacji typu DELETE, biblioteka może usunąć odpowiedni obiekt z cache'a

```
const [mutate, { isLoading, error, data }] = useMutation(mutationFunction, options);
```

REACT



Starcie ostateczne!

Starcie ostateczne

Stworzymy nową aplikację korzystając z całej poznanej póki co wiedzy i postaramy się zbudować odpowiednią architekturę aplikacji.

Nowa aplikacja będzie polegała na stworzeniu strony internetowej restauracji z burgerami, w której użytkownik będzie mógł się zalogować i sprawdzić menu oraz ustawić swoje dane konta (w tym zdjęcie).

Uwaga! Do wszystkich zapytań do bazy danych Firebase będziemy wykorzystywać **react query**. Firebase wystawia nam swoje dane poprzez *databaseURL*. By dostać się do danych za pomocą protokołu HTTP musimy dodać na końcu zawsze **.json**.

Na przykład:

<https://rest-api-b6410.firebaseio.com/persons.json>

Burgers menu

Stwórzmy tabelę, która będzie przedstawiać menu naszej nowo otwartej burgerowni Firebase Burgers.

1. Wygeneruj tabele z użyciem UI components.
2. Pobierz dane z serwera i wyświetl w konsoli.
3. Użyj pobranych danych w tabeli.

Firestore Burgers

Name	Ingredients	Price
Hot	beef,bread,BBQ,jalapeno	22
Hawai	beef,bread,pineapple	20
Specjal	beef,bread,BBQ	15

Burgers details

Dodajmy podstronę, na której wyświetlimy dane i zdjęcie burgera.

1. Stwórz nowy component i podepnij go pod odpowiednią ścieżkę.
2. Zmień nazwę burgera w tabeli na Link i przekieruj na odpowiednią ścieżkę.
3. W widoku detalicznym pobierz dane o burgerze i je wyświetl. Pamiętaj o stanie ładowania i błędach.

Firestore Burgers

Name	Ingredients	Price
Hot	beef,bread,BBQ,jalapeno	22
Hawai	beef,bread,pineapple	20
Specjal	beef,bread,BBQ	15

Burgers admin

Pora przygotować panel administratora.

1. Stwórz nową tabelę (na potrzeby zajęć możesz skopiować z poprzedniego zadania).
2. Dodaj przyciski Edit i Delete.
3. Na dole tabeli dodaj przycisk (+), który otworzy modala z formularzem do dodania nowego burgera.

Firestore Burgers

Name	Ingredients	Price		
Hot	beef,bread,BBQ,jalapeno	22	EDIT	DELETE
Hawai	beef,bread,pineapple	20	EDIT	DELETE
Specjal	beef,bread,BBQ	15	EDIT	DELETE

Burgers admin

Dodajmy interakcję!

Wykonaj request do serwera by stworzyć nowego burgera w modalu.

Dodajmy walidację formularza z użyciem react-hook-forms i zod.

Pamiętaj by odświeżyć listę po wykonaniu zapytania.

Firestore Burgers

Name	Ingredients	Price		
Hot	beef,bread,BBQ,jalapeno	22	EDIT	DELETE
Hawai	beef,bread,pineapple	20	EDIT	DELETE
Specjal	beef,bread,BBQ	15	EDIT	DELETE

Burgers admin

Kolejna interakcja

Wykonaj request po kliknięciu Delete.
(*) Dodaj potwierdzenie przed
usunięciem - może być alert.confirm.

Pamiętaj by za każdym razem
odświeżyć listę po operacji.

Zastanów się jak możemy zrobić Edit z
założeniem, że edycja ma się odbywać
inline.

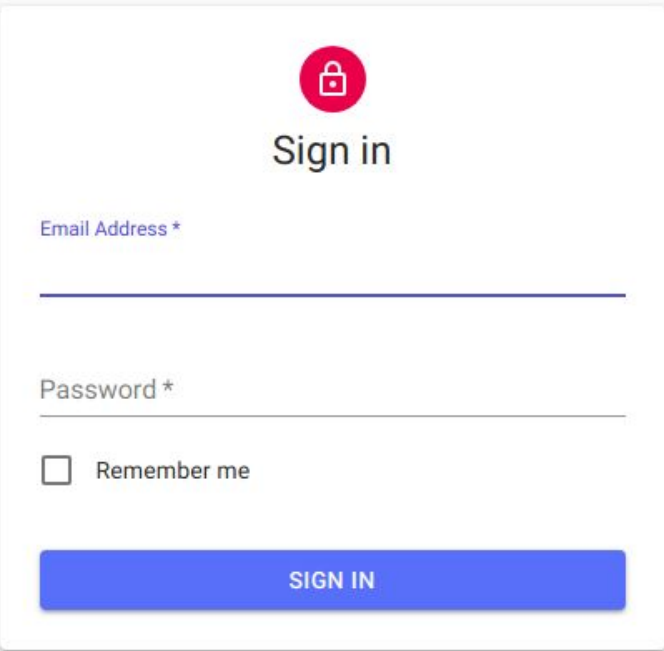
Firestore Burgers


Name	Ingredients	Price		
Hot	beef,bread,BBQ,jalapeno	22	EDIT	DELETE
Hawai	beef,bread,pineapple	20	EDIT	DELETE
Specjal	beef,bread,BBQ	15	EDIT	DELETE

Logowanie

Dodaj komponent logowania się do firebase.

1. Razem wykonajmy inicjalizację aplikacji firebase.
2. Stwórz formularz do rejestracji i logowania.
3. Razem dodajmy metody z firebase do formularza.
4. Stwórz kontekst z danymi użytkownika i zapisz tam dane o nim.
5. Wykorzystaj kontekst by schować lub pokazać niektóre elementy.





Sign in

Email Address *

Password *

☐ Remember me

SIGN IN

Built with ❤️ by the Material-UI team.



Dzięki

Znajdziecie mnie:

<https://www.linkedin.com/in/kamil-richert/>

<https://github.com/krichert>