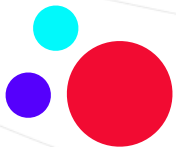


React

CSS Modules oraz CSS in JS

infoShare Academy



HELLO

Jakub Wojtach

Senior **full stack** developer





Zacznijmy ten dzień z przytupem!

Adam

Ewa

Azja

Afryka

500+

Tanie kredyty

V8

Elektryk



Agenda

- **Podstawy** teorii
- **Stylizowanie** aplikacji z użyciem CSS Modules
- **Wykorzystanie** wybranej biblioteki CSS in JS
- **Zestawienia** rozwiązań stylowania aplikacji



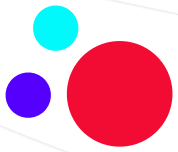
Współpraca

- Zadajemy pytania w dowolnym momencie – kanał **merytoryka**
- Krótkie przerwy (**5 min**) co godzinę
- Długa przerwa (**20 min**) po ostatnim bloku



- W dniu dzisiejszym zobrazujemy sobie ograniczenia globalnych stylów CSS
- Porozmawiamy również o tym w jaki sposób stylować aplikację z użyciem CSS Modules
- Następnie przejdziemy sobie do stylowania aplikacji z użyciem wybranej biblioteki CSS in JS na przykładzie Styled Components





- Skupimy się na poznaniu technik stylowania w React i skupimy się na dwóch głównych technikach:
 - CSS Modules
 - CSS-in-JS z użyciem **Styled components**
- Obie z tych technik stosowane są w różnych projektach i warto znać różnice między nimi, ale żadne z rozwiązań nie ma jednoznacznej "przewagi" nad drugim, choć najpopularniejszym podejściem jest raczej wykorzystanie biblioteki Styled components **aktualnie**.
- Pracując w ekosystemie Reacta należy znać **obie** techniki, aby sztucznie się nie ograniczać.

- Najbardziej "wbudowany" sposób stylowania w React to po prostu wykorzystanie standardowych plików .css. Dotychczas ten sposób już wykorzystywaliśmy, bo CRA wygenerował nam pliki np. index.css który był importowany i wykorzystywany w pliku index.js.
- Pokażemy to na przykładzie komponentu UsersListItem gdzie w folderze dodamy odpowiednik plik CSS UsersListItem.css który będzie wyglądał następująco:

```
.user {  
  background-color: pink;  
}
```

- A następnie w UsersListItem importujemy plik CSS i wykorzystujemy tą klasę:

```
import React from 'react'
import './UsersListItem.css'

const UsersListItem = userData => (
  <li className="user">
    <div>{userData.avatar}</div>
    <div>
      <p>{userData.firstName}</p>
      <p>{userData.lastName}</p>
    </div>
  </li>
)
```

Przypomnienie - className odpowiada atrybutowi class z HTML.

- Każdy z elementów listy (pojedynczy użytkownik) ma kolor tła ustawiony na różowy.



CSS – ograniczenia

- Taki sposób stylowania skaluje się nie najlepiej, zwłaszcza, że mamy inne sposoby w React które w połączeniu z tym ekosystemem dają nam nieco większe możliwości.
- Tego sposobu się najczęściej nie wykorzystuje, chyba, że mamy do czynienia z prostym projektem. Mimo wszystko, warto wykorzystać możliwość wykorzystania modułów CSS w React za pomocą CRA sprawia, że ta metoda się nie broni.
- W przypadku aplikacji produkcyjnych często natrafiamy na sytuacje, w których często używane nazwy klas są nadpisywane przez biblioteki doinstalowane do naszego projektu, co prowadzi do niepożądanych efektów i niszczenia naszej pracy, na którą klient poświęcił przecież swoje pieniądze!
- Na ratunek przychodzą nam wtedy nowe rozwiązania, a jednym z nich są **CSS Modules**



CSS Modules in 2 minutes

learn while you poop

- Popularnym problemem, z którym często spotykamy się przy projektowaniu HTML i CSS jest kolizja nazw klas – kilku programistów może wybrać tę samą nazwę klasy dla różnych elementów i po dodaniu swoich arkuszy stylów do projektu ich właściwości zaczną ze sobą oddziaływać – łączyć się lub nadpisywać. W celu uniknięcia tego typu sytuacji stosuje się różne techniki takie jak tworzenie przestrzeni nazw albo metodologie np. BEM.
- W świecie JS problem ten rozwiązano nieco inaczej – konieczność dbania o to, by klasy były unikalne przesunięta została z programisty na **bundler**. Przy użyciu **css-loader** z **włączoną opcją** *css-modules*, bundler zmieni nasze nazwy klas na **pseudo-losowe**, zapewniając że szansa nadpisywania się klas kilku niezależnych elementów jest znikoma.

- W celu skorzystania z css-modules musimy zmodyfikować nieco proces, w który importujemy nasz plik CSS. Na skutek tej zmiany otrzymamy obiekt, który zawierać będzie zdefiniowane przez nas klasy jako klucze, zaś ich wartość zawierać będzie pseudo-losową nazwę klasy, wygenerowaną dla konkretnego przypadku użycia.
- Zakładając, że w naszym projekcie istnieje plik style.css o zawartości:

```
.button {  
  color: white;  
  background: blue;  
  padding: 10px;  
}
```

- Możemy użyć css-loader i css-modules w następując sposób:



CSS Modules

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

import Styles from "./style.css";
console.log(Styles);

/**
 * {
 *   button: 'RMStbBE9w'
 * }
 */

ReactDOM.render(
  <button className={Styles.button}>Kliknij mnie!</button>,
  document.getElementById('root')
);
```

- W tym wypadku webpack zaimportuje nasz plik CSS, lecz przed umieszczeniem go w dokumencie przetworzy znajdujące się w nim selektory CSS zastępując klasy wg. podanego w konfiguracji wzorca.
- W przypadku naszego projektu, który wygenerowaliśmy z użyciem vite – obsługa CSS Modules działa od razu!
- Jedyne o czym musimy pamiętać, to aby nadawać plikom ze stylami rozszerzenie **.module.css**

Zadanie

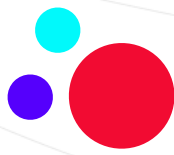


CSS Modules – classNames

- Aby tworzyć złożone klasy, łączyć je ze sobą i aplikować je warunkowo używamy atrybutu **className**
- Pomocną paczką, ułatwiającą ten proces jest paczka **classNames**
- Możemy to również zrobić za pomocą templateLiterals, jednak jest to często mniej czytelne i bardziej karkołomne

```
const classNames = classNames(styles.sideMenu,  
  { [styles.active]: this.props.menuOpen }  
);
```

Zadanie



CSS Modules – :global

- Kwerenda **:global** przełącza stylowanie w danym selektorze na globalny scope
- Co nam to da? CSS modules nie będzie modularyzował kodu CSS w danym selektorze i nie doda losowego ciągu znaków do nazwy klasy, co sprawi, że ta klasa wykorzystana gdziekolwiek będzie przekazywała style zapisane wewnątrz każdemu elementowi z tą klasą.

```
:global(.myclass) {  
  background-color: red;  
}
```

- Można to wykorzystać również do stylów użytych z zewnątrz, z zewnętrznej biblioteki, ale będącej częścią klasy, którą dodajemy do naszego elementu (np. Style z bootstrapa itd.)



CSS Modules – composing

- css-modules pozwalają na composing (łączenie) kilku klas razem aby stworzyć nowe klasy za pomocą polecenia **compose**.
- Działa to na podobnej zasadzie jak przy tworzeniu klas w innych językach, pozwalając na tworzenie bardziej złożonych obiektów zbudowanych z uprzednio zdefiniowanych stylów lub innych klas

```
.classA {  
  background-color: green;  
  color: white;  
}
```

```
.classB {  
  composes: classA;  
  color: blue;  
}
```

Zadanie



- A co jeśli zamiast importować plik ze stylami, wrzucić by je tak bezpośrednio do kodu JavaScript zachowując przy tym <style> tag zamiast in-line styling? Nie ma problemu! Podejście takie nazywane jest **css-in-js**.
- Do jego użycia potrzebujemy dodatkowej biblioteki, która nam przetłumaczy nasz kod css-in-js do postaci wynikowej.
- Takich bibliotek jest cała masa, ja skupię się na bardzo popularnej **styled-components**
- Cała magia css-in-js polega na wykorzystaniu funkcjonalności języka JS 'Template literals', którą możemy cieszyć się od wersji ES6.



Styled components

- Styled components to biblioteka, która korzysta ze stringów interpolowanych (template literals) wprowadzonych w wersji JavaScript ES6
- Pozwala ona na bardzo proste stylowanie komponentów React poprzez dodanie stylów bezpośrednio do komponentu.
- Daje możliwość budowania niestandardowych komponentów przy użyciu CSS
- Jest modularny, elastyczny, pozwala na przekazywanie dodatkowych parametrów do samego komponentu
- Dzięki niemu możemy rozszerzać style, zagnieżdżać selektory, korzystać z polimorfizmu, pisać animacje i wiele innych..



Styled components – template literals

- Jak już wielokrotnie zostało wspomniane – główna składowa styled components to template literals. To właśnie dzięki nim tworzymy style dla **styled components**

```
export const Container = styled.div`  
  display: flex;  
  flex-direction: row;  
  justify-content: center;  
  align-items: center;  
`;  
;
```

```
export const Button = styled.button`  
  width: 40px;  
  height: 40px;  
`;  
;
```




Styled components – template literals

- Jak widzicie jest to prosty div, który będzie pilnował ułożenia naszych elementów wewnątrz siebie.
- Aby wykorzystać komponent wewnątrz Reacta należy go zaimportować:

```
import { Container, Button } from './App.styles';
```

- Dzięki takiemu wykorzystaniu możemy potem wykorzystać dany element wewnątrz naszej aplikacji

```
const App = () => {  
  return (  
    <Container>  
      <Button>Click me!</Button>  
    </Container>  
  );  
}
```


Zadanie



Styled components – props

- Styled components pozwala nam na przekazanie własności, które następnie możemy wykorzystać wewnątrz naszego komponentu po stronie deklaracji **styled components**
- Stosujemy to często do modyfikacji aktywnego stanu komponentu, motywu schematu, korzystania z ogólnego theme (o czym później) i... wiele innych!



Styled components – props

```
// Create an Input component that'll render an <input> tag with some styles
const Input = styled.input`
  padding: 0.5em;
  margin: 0.5em;
  color: ${props => props.inputColor || "palevioletred"};
  background: papayawhip;
  border: none;
  border-radius: 3px;
`;
```

```
// Render a styled text input with the standard input color, and one with a custom
input color
render(
  <div>
    <Input defaultValue="@probablyup" type="text" />
    <Input defaultValue="@geelen" type="text" inputColor="rebeccapurple" />
  </div>
);
```



Styled components – rozszerzanie styli

- Podobnie jak w przypadku css modules, w styled components mamy możliwość rozszerzania styli konkretnego komponentu o wszystko poza jedną drobną zmianą
- Aby tego dokonać należy opakować uprzednio utworzony komponent w znacznik styled – w ten sposób nastąpi dziedziczenie wszystkich jego parametrów, a ewentualnie nadpisanie konkretnych wartości dokona się w definicji poniżej



Styled components – rozszerzanie styli

// The Button from the last section without the interpolations

```
const Button = styled.button`  
  color: palevioletred;  
  font-size: 1em;  
  margin: 1em;  
  padding: 0.25em 1em;  
  border: 2px solid palevioletred;  
  border-radius: 3px;  
`;  
;
```

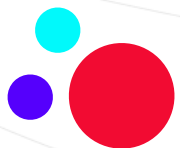
Normal Button

Tomato Button

// A new component based on Button, but with some override styles

```
const TomatoButton = styled(Button)`  
  color: tomato;  
  border-color: tomato;  
`;  
;
```

Zadanie



Styled components – zagnieżdżanie selektorów

- Z racji na to, że Styled Components, pod spodem, używa preprocesora stylis, który pozwala na wykorzystywanie zapisu, który jest znany osobom, które wcześniej korzystały z scss.
- Dzięki temu możliwe jest korzystanie z wielu mechanizmów.
- Przykładowe:
 - **&** – pojedynczy ampersand odwołujący się do wszystkich instancji danego komponentu, używane do nadpisywania
 - **&&** – podwójny ampersand odwołuje się do tej konkretnej instancji danego komponentu, użyteczne gdy piszemy style warunkowe, a nie chcemy, by wszystkie instancje zostały przezeń nadpisane
 - I wiele innych...

Zadanie



Styled components – .attrs

- API **SC** pozwala nam na ustawianie atrybutów elementu DOM z poziomu definicji komponentu
- Odwołujemy się do niego za pomocą atrybutu .attrs, a sama funkcja jako parametr przyjmuje obiekt, za pomocą którego możemy ustawić atrybuty, które dany element DOM może przyjąć
- Dzięki tej funkcji możemy np. Stworzyć button, który w domyśle będzie typu submit, co pozwala na pomijanie tego parametru w późniejszym użyciu



STYLED COMPONENTS

PASSED PROPS &
ADDING ATTRIBUTES

07

Zadanie



Styled components – .as

- Wykorzystywany w momencie, gdy chcemy zachować całe stylowanie, jakie przypisaliśmy do danego elementu, ale wyrenderować go w drzewie jako inny element DOM
- Przykładem może być pięknie ostylowany przycisk, który chcemy wyrenderować jako link – wszystkie style, jakie dodaliśmy do przycisku będą tutaj dodane



Styled components – transient props

- Wykorzystywane w momencie, gdy życzeniem programisty jest ukrycie propsów przekazanych do komponentu w drzewie DOM
- Dzięki temu nie otrzymamy dziwnych atrybutów w komponentach jako kod wynikowy, a utrzymamy całą funkcjonalność przekazywanych propsów
- Aby wykorzystać ten mechanizm dodany props należy poprzedzić znakiem dollar, czyli \$



Styled components – transient props

```
const Comp = styled.div`  
  color: ${props =>  
    props.$draggable || 'black'};  
`;  
  
render(  
  <Comp $draggable="red" draggable="true">  
    Drag me!  
  </Comp>  
) ;
```




Styled components – animacje

- Animacje CSS używane z normalnym słowem kluczowym **@keyframes** nie są przypisane do pojedynczego komponentu, ale aby uniknąć nadpisywania przez tę samą nazwę należy jej unikać, gdy możemy korzystać z mechanizmu **SC**
- Aby tego dokonać **SC** udostępnia helper **keyframes**, który wygeneruje unikatową instancję, którą możemy wykorzystywać na poziomie naszej aplikacji
- Co ciekawe – keyframes nie są wspierane w React-native, do tego służy osobne api **ReactNative.Animated**
-



STYLED COMPONENTS

ANIMATIONS

08

Zadanie



Styled components – CSS

- Jest to helper, który pozwala na generowanie kodu CSS za pomocą template literals
- Pozwala na wykorzystywanie pełnych możliwości **SC** wewnątrz, takich jak interpolacja funkcji z zewnątrz
- W moim przypadku wykorzystywałem to do bardziej złożonych stylów warunkowych dla danego komponentu



Styled components – CSS

```
import styled, { css } from 'styled-components'
```

```
const complexMixin = css`  
  color: ${props => (props.whiteColor ? 'white' : 'black')};  
`
```

```
const StyledComp = styled.div`  
  /* This is an example of a nested interpolation */  
  ${props => (props.complex ? complexMixin : 'color: blue;')};  
`
```

Zadanie



Styled components – createGlobalStyle

- Funkcja helper, pozwalająca na wygenerowanie specjalnej instancji **StyledComponents**, której zadaniem jest zarządzanie globalnymi stylami aplikacji
- W normalnym przypadku scope SC ograniczony jest tylko i wyłącznie do swojej instancji i lokalnego CSS, co oznacza izolację od innych komponentów
- W przypadku tej funkcji ta limitacja zostaje usunięta i dzięki temu możemy nadać globalne style, które normalnie dodalibyśmy gdzieś w pliku reset lub innym tego typu
- <https://styled-components.com/docs/api#createglobalstyle>



Styled components – Theming

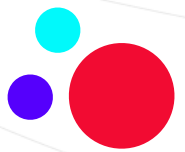
- Jeden z moich ulubionych mechanizmów API **SC**
- Wykorzystywany do tworzenia motywów dla naszej strony, za jego pomocą możemy np zadeklarować kolory dla całej aplikacji, globalny font, letter spacing itd.
- Pod spodem korzysta z Context API, które pozwala na udostępnienie danych na wybranym poziomie
- Wykorzystanie theme jest bardzo proste, a samo dodanie motywu odbywa się za pośrednictwem komponentu **ThemeProvider**
- W późniejszym etapie każdy komponent poniżej tego providera otrzymuje dostęp do obiektu theme, w którym będą przechowywane atrybuty, które ustawiliśmy w obiekcie



STYLED COMPONENTS

THEMING

09



Styled components – przykład praktyczny Cyclops app





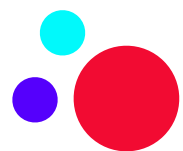
Styled components – wsparcie IDE

- W naszym przypadku – bardzo polecam wtyczkę <https://marketplace.visualstudio.com/items?itemName=styled-components.vscode-styled-components>
- Pozwala ona na kolorowanie składni, odpowiednie podpowiedzi kodu i wiele innych



CSS in JS – inne rozwiązania

- **Emotion** – <https://github.com/jsjoeio/styled-components-vs-emotion>
- **Stitches** – w linkach dla chętnych
- **Linaria** – zero runtime, czyli CSS eksportowany do plików CSS podczas buildu, mechanizmy SASS, dynamiczne dropsy, sourcemapsy, logika JS



Zestawienie rozwiązań

- **Zalety**

- Automatycznie wydzielany krytyczny CSS
- Automatyczne generowanie unikalnych nazw klas
- Łatwy sposób dodawania dynamicznych stylów
- Automatyczny vendor prefixing
- Pełnoprawne API umożliwiające tworzenie zarządzalnych i skalowalnych stylów aplikacji
- Podejście modularne, komponentowe



CSS in JS – wady i zalety

• Wady

- Kod css zaczyna wyglądać bardziej niż JavaScript, co może być problematyczne, zwłaszcza dla niedoświadczonego programisty
- W przypadku małej bazy kodu napisanej w CSS przepisanie jej na SC jest dość proste, w przypadku dużego projektu jest to już proces długi i skomplikowany
- Nie jest rozwiązaniem, które powinno być stosowane wszędzie, gdyż wprowadza dodatkową warstwę komplikacji
- Overhead JS w runtime wynikający z użycia biblioteki CSS in JS
- **Problemy wydajnościowe w React 18 dla Concurrent Mode**
- Problemy z wydajnością w przypadku SSR

- **Zalety**

- Nadpisywanie stylów w tych samych klasach używanych w różnych plikach nie występuje w przypadku tego mechanizmu – rozwiązuje to dynamiczne generowanie nazw klas
- CSS Modules pozwala na wysłanie tych samych klas do wielu komponentów
- Jedną z głównych zalet to fakt, że możemy za ich pomocą edytować dowolny kod CSS bez strachu przed tym, jak ten kod wpłynie na inne podstrony
- Pomimo tego, jak złożony jest kod źródłowy samego projektu, rozwiązanie jest relatywnie proste w wykorzystaniu i zrozumiałe dla innych programistów



CSS Modules – wady i zalety

- **Wady**

- Style muszą być zaaplikowane jako obiekt, dowolnie w notacji kropki lub tablicowej, gdy używamy ich w projekcie
- W porównaniu do Styled Components, które omawialiśmy przy CSS in JS, CSS modules nie akceptują żadnych propsów
- Jeśli chcemy pracować ze stylami globalnymi CSS modules nie będą dobrym rozwiązaniem

- https://www.youtube.com/watch?v=Sgcfiow4fVQ&ab_channel=RichardOliverBray – konfiguracja CSS modules w projekcie opartym o Site
- <https://dhanrajsp.me/snippets/css-module-going-global-with-class-selectors> – local i global w CSS Modules
- https://www.youtube.com/watch?v=EsSi4cER48E&ab_channel=LevelUpTuts – czym są CSS in JS, jak działają, jak działa Styled Components
- https://www.youtube.com/watch?v=QZvP5O-BU60&ab_channel=ForThoseWhoCode – stitches crash course

Dziękuję za uwagę

Jakub Wojtach