infoShare ACADEMY

REACT ADVANCED

infoShare Academy

infoShareAcademy.com

HELLO

Kamil Richert

Senior Software Engineer w Atlassian

infoShare ACADEMY


Stylowanie


info Share
ACADEMY

Stylowanie

CSS w React działają tak samo jak w przypadku zwykłego HTML'a z drobną różnicą w postaci dodawania styli. 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 styli.

App.css

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

App.js

```
import './App.css';

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

Stylowanie

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

```
<div style={{color: ‘red’, backgroundColor: ‘white’ }}>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.

Zadanie

Otwórz folder *modul-1-styling/broken-app* i spróbuj naprawić aplikacje (a dokładniej jej stylowanie) tak by zaczęła wyglądać normalnie.

CSS Modules

infoShareAcademy.com

CSS Modules

Moduły CSS umożliwiają określenie zakresu CSS poprzez automatyczne utworzenie unikalnej nazwy klasy w formacie

[nazwa pliku]_[nazwa klasy]_[hash].

Oznacza to, że rozwiązuje problem z nadpisywanie nazw klas przez inne komponenty i możesz używać tej samej nazwy klasy dla każdego komponentu.

Create React App obsługuje moduły CSS przy użyciu konwencji nazewnictwa plików [nazwa].module.css.
https://create-react-app.dev/docs/adding-a-css-modules-stylesheet/

App.modules.css

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

App.js

```
import styles from ‘./App.module.css’;

function App() {
        return <div className={styles.root}>Tekst</div>
}
```

Zadanie

Otwórz folder *modul-1-styling/modules-app* i popraw aplikacje (a dokładniej jej stylowanie) z użyciem CSS Modules.

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;
`
```

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

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

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

Otwórz folder *modul-1-styling/styled-app* i popraw aplikacje (a dokładniej jej stylowanie) z użyciem Styled components.


Theming


infoShare
ACADEMY

Theming

Motyw określa kolor elementów, odcień tła, poziom cienia, czy odpowiednią przezroczystość czcionki itp.

Motywy pozwalają nadać aplikacji spójny ton. Pozwala dostosować wszystkie aspekty projektu.

Aby zapewnić większą spójność między aplikacjami, do wyboru są jasne i ciemne typy motywów. Domyślnie komponenty używają motywu jasnego.

Theming

Wykonanie motywu w aplikacji Reactowych można wykonać na parę sposobów:

- używając CSS variables
- poprzez React context
- za pomocą styled-components

CSS variables

```
import React, { useState } from "react";
import { Button } from "./ui-components";

function App() {
  const [theme, updateTheme] = useState("light-theme");

  const toggleTheme = () => {...};

  return (
    <div id="app" className={theme}>
      <Button onClick={() => toggleTheme()} />
    </div>
  );
}
```

CSS variables

```
.light-theme {
  --foreground: "#000000";
  --background: "#eeeeee";
  --primary: "#0092e3";
  --font: "Nunito";
}

.dark-theme {
  --foreground: "#ffffff";
  --background: "#222222";
  --primary: "#0017e3";
  --font: "Nunito";
}
```

```
.button {
  background: var(--background);
  color: var(--primary);
}
```

```
import React from "react";

function Button() {
  return (
    <button className="button">
      I am styled by theme CSS variables!
    </button>
  );
}
```

infoShare ACADEMY

React context

```
const theme = {
  light: {
    foreground: "#000000",
    background: "#eeeeee",
    primary: "#0092e3",
    font: "Nunito",
  },
  dark: {
    foreground: "#ffffff",
    background: "#222222",
    primary: "#0017e3",
    font: "Nunito",
  },
};
```

```
import { createContext } from "react";
import { theme } from './theme'

const ThemeContext = createContext(
        { theme: theme.light }
);
```

React context

```
function App() {
  return (
    <ThemeContext.Provider value={themes.dark}>
      <Button />
    </ThemeContext.Provider>
  );
}

import { useContext } from "react";

function Button() {
  const theme = useContext(ThemeContext);
  // mamy dostęp do motywu
}
```

Styled components

styled-components ma pełną obsługę motywów poprzez eksport komponentu <ThemeProvider>. Ten komponent udostępnia motyw wszystkim komponentom React znajdującym się pod nim za pośrednictwem context API. W drzewie renderowania wszystkie komponenty stylizowane będą miały dostęp do dostarczonego motywu.

Styled components

```
const theme = {
  main: "mediumseagreen"
};

render(
  <ThemeProvider theme={theme}>
    <Button>Themed</Button>
  </ThemeProvider>
);
```

```
const Button = styled.button`
  color: ${props => props.theme.main};
  border: 2px solid ${props =>
props.theme.main};
`;

Button.defaultProps = {
  theme: {
    main: "#BF4F74"
  }
}
```

Styled components

Function themes - możesz także przekazać funkcję dla wartości motywu. Ta funkcja otrzyma motyw nadrzędny, czyli od innego <ThemeProvider> znajdującego się wyżej w drzewie.

withTheme

Jeśli kiedykolwiek będziesz musiał użyć bieżącego motywu poza komponentami stylizowanymi (np. wewnątrz dużych komponentów), możesz użyć komponentu wyższego rzędu withTheme.

The theme prop

Motyw można także przekazać do komponentu za pomocą prop'a.

Jest to przydatne do obejścia brakującego ThemeProvider lub jego zastąpienia.

```
<Button theme={{ main: "darkorange" }}>Overridden</Button>
```

infoShare ACADEMY

Styled components

useContext

Możesz także użyć useContext, aby uzyskać dostęp do bieżącego motywu poza stylizowanymi komponentami podczas pracy z React Hooks.

useTheme

Możesz także użyć useTheme, aby uzyskać dostęp do bieżącego motywu poza stylizowanymi komponentami podczas pracy z React Hooks.

więcej tutaj: https://styled-components.com/docs/advanced#theming

Zadanie

Otwórz folder *modul-1-styling/theme-app* i dodaj motyw za pomocą wybranego z podanych sposobów.

Typescript

infoShareAcademy.com

Typescript

Parę słów na początek przypomnienia co to jest TypeScript?

- Nadstawka na JavaScript, która dodaje statyczne typowanie.
- Stworzony przez Microsoft, wydany w 2012 roku.

Dlaczego warto używać TypeScript?

- Poprawa jakości kodu poprzez wykrywanie błędów w czasie kompilacji.
- Lepsza czytelność i zrozumiałość kodu dzięki typom.
- Rozbudowane wsparcie dla narzędzi (IDE).

Native DOM and React typing

TypeScript dostarcza wbudowane typy dla elementów DOM, takie jak HTMLElement, HTMLInputElement, Event, MouseEvent itp.

Użycie właściwego typu pozwala na dostęp do specyficznych właściwości i metod elementów.

```
const button: HTMLButtonElement = document.getElementById('myButton');
button.addEventListener('click', (event: MouseEvent) => {
  console.log(event.clientX, event.clientY);
});

const inputElement = document.querySelector('input') as HTMLInputElement;
inputElement.addEventListener('input', (event: Event) => {
  const target = event.target as HTMLInputElement;
  console.log(target.value);
});
```

Native DOM and React typing

TypeScript udostępnia również typy dla zdarzeń w React, takie jak React.MouseEvent, React.ChangeEvent, React.FormEvent, itp.

```
const handleClick = (event: React.MouseEvent<HTMLButtonElement>) => {
  console.log('Button clicked:', event.currentTarget);
};

const MyButton: React.FC = () => (
  <button onClick={handleClick}>Click Me!</button>
);
```

Component, props and state types

React udostępnia typy takie jak React.FC (dla komponentów funkcyjnych) oraz React.Component (dla komponentów klasowych).

Dzięki tym typom zyskujemy automatyczne typowanie dla propsów i dzieci komponentów.

Component, props and state types

Definiowanie interfejsów dla propsów pozwala na lepsze zarządzanie typami danych przekazywanych do komponentów. Można to zrobić za pomocą Reactowych typów lub typując bezpośrednio obiekt.

```
interface ButtonProps {
  label: string;
  onClick: () => void;
}

const Button: React.FC<ButtonProps> = ({ label, onClick }) => (
  <button onClick={onClick}>{label}</button>
);

const Button = ({ label, onClick }: ButtonProps) => (
  <button onClick={onClick}>{label}</button>
);
```

Component, props and state types

Możemy też otypować hooki w Reactcie:

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

const Counter: React.FC = () => {
  const [count, setCount] = useState<number>(0);

  return (
    <div>
      <p>Count: {count}</p>
      <button onClick={() => setCount(count +
1)}>Increment</button>
    </div>
  );
};
```

Zadanie

Dodaj pełne typowanie w aplikacji *modul-2-typescript/types-app*

Type vs interface

Główne różnice między type a interface to:

- type pozwala na tworzenie typów niestandardowych, takich jak np. unie, aliasów, literałów, a interface pozwala na definiowanie tylko obiektów lub klas.
- interface umożliwia dziedziczenie za pomocą słowa kluczowego extends, podczas gdy type nie pozwala na dziedziczenie.
- type można użyć wraz z innymi typami, takimi jak union czy intersection, podczas gdy interface nie można.
- interface po ponownej deklaracji jest scalany z poprzednią deklaracją

Type vs interface

```ts
type User = {
  name: string;
  age: number;
};

type UserID = string | number;

type Admin = User & {
  permissions: string[];
};
```

```ts
interface IUser {
  name: string;
  age: number;
}

interface IAdmin extends IUser {
  permissions: string[];
}
```

Generic types

Używane do tworzenia komponentów i funkcji, które mogą działać z różnymi typami.

```
function identity<T>(arg: T): T {
    return arg;
}

const stringIdentity = identity<string>("Hello");

const numberIdentity = identity<number>(42);
```

Zadanie

Wykonaj zadanie w *modul-2-typescript/zadanie-1*

Unknown and never

Unknown to bezpieczniejszy odpowiednik any, wymaga sprawdzenia typu przed użyciem.

```
let value: unknown;
value = "Hello";
if (typeof value === "string") {
  console.log(value.toUpperCase());
}
```

Never to typ używany dla wartości, które nigdy nie występują. Użyteczny w funkcjach, które zawsze rzucają błędy lub nigdy nie zwracają.

```
function throwError(message: string): never {
    throw new Error(message);
}
```

Type guards

Techniki używane do sprawdzania typów w czasie wykonywania. Pomagają TypeScriptowi lepiej zrozumieć typy zmiennych.

Z użyciem operatora typeof:

```
function isString(value: any): value is string {
  return typeof value === 'string';
}

let someValue: any = "Hello";
if (isString(someValue)) {
  console.log(someValue.toUpperCase());
  // TypeScript wie, że someValue to string
}
```

Type guards

Z użyciem operatora instanceof:

```
class Animal {
  makeSound() {
    console.log("...");
  }
}

class Dog extends Animal {
  bark() {
    console.log("Woof!");
  }
}
```

```
function makeAnimalSound(animal: Animal) {
        animal.makeSound();
        if (animal instanceof Dog) {
            animal.bark();
        }
}

const myPet: Animal = new Dog();
makeAnimalSound(myPet);
// TypeScript wie, że myPet może być Dogiem w bloku if
```

Type guards

Użycie niestandardowych type guards:

```
interface Fish {
  swim: () => void;
}

interface Bird {
  fly: () => void;
}
```

```
function isFish(pet: Fish | Bird): pet is Fish {
  return (pet as Fish).swim !== undefined;
}

function move(pet: Fish | Bird) {
  if (isFish(pet)) {
    pet.swim();
  } else {
    pet.fly();
  }
}
```

Słowo kluczowe as

```
interface Pokeball {
    name: string;
    chanceRate: number;
    rarity: PokeballRarity;
}

let ultraball: Pokeball = {};

let greatball: Pokeball = {} as Pokeball;
```

```
let ultraball: Pokeball

Type '{}' is missing the following properties from type
'Pokeball': name, chanceRate, rarity (2739)

Peek Problem (Alt+F8)    No quick fixes available
```

Słowo kluczowe as pozwoli nam powiedzieć TypeScriptowi, że dany obiekt na na pewno będize danego typu nawet jeśli teraz tak nie wygląda

Zadanie

Wykonaj zadanie w *modul-2-typescript/zadanie-2*

Formularze

infoShareAcademy.com

React hook form

Podstawowe API:

- useForm: Hook do zarządzania formularzem.
- register: Funkcja do rejestrowania pól formularza.
- handleSubmit: Funkcja do obsługi submita formularza.

```
const { register, handleSubmit } = useForm();
const onSubmit = data => console.log(data);

<form onSubmit={handleSubmit(onSubmit)}>
  <input {...register('firstName')} placeholder="First Name" />
  <button type="submit">Submit</button>
</form>
```

Zadanie

Wykorzystaj *react-hook-form* w aplikacji *modul-3-forms/basic-form-app*

Predefiniowane wartości i reset

useForm zwraca metodę do resetowania formularza oraz można do niego przekazać jako argument predefiniowane wartości

```
function PredefinedForm() {
  const { register, handleSubmit, reset } = useForm({
    defaultValues: {
      firstName: "John",
      lastName: "Doe"
    }
  });
```

Walidacja

Wbudowana walidacja: required, minLength, maxLength oraz accessibility.

```
<input
  id="name"
  aria-invalid={errors.name ? "true" : "false"}
  {...register("name", { required: true, maxLength: 30 })}
/>
```

Walidacja

```
const { register, handleSubmit, formState: { errors } } = useForm()
const onSubmit = (data) => console.log(data)

return (
  <form onSubmit={handleSubmit(onSubmit)}>
    <input id="name" {...register("name", { required: true, maxLength: 30 })}/>

    {errors.name && errors.name.type === "required" && (
      <span role="alert">This is required</span>
    )}
    {errors.name && errors.name.type === "maxLength" && (
      <span role="alert">Max length exceeded</span>
    )}
    <input type="submit" />
  </form>
)
}
```

Zadanie

W aplikacji *modul-3-forms/basic-form-app*:

Dodaj podstawową walidację do formularza i wyświetl odpowiednią notyfikacje w przypadku błędu.

- name i surname powinien być wymagany i pozwalać jedynie na litery
- comment powinien mieć maksymalnie 50 znaków

Dodaj przycisk do resetowania formularza.

Dynamiczne pola

```
import { useForm, useFieldArray } from 'react-hook-form';

function DynamicForm() {
  const { register, control, handleSubmit } = useForm();
  const { fields, append, remove } = useFieldArray({ control, name: "users"  });
  const onSubmit = data => console.log(data);

  return (
    <form onSubmit={handleSubmit(onSubmit)}>
      {fields.map((field, index) => <input {...register(`users.${index}.name`)} />)}
      <button type="button" onClick={() => append({ name: "" })}>
        Add User
      </button>
      <input type="submit" />
    </form>
  );
}
```

Reagowanie na zmiany i wymuszone zmiany

```
function AdvancedForm() {
  const { register, handleSubmit, watch, setValue } = useForm();
  const watchAllFields = watch();
  const onSubmit = data => console.log(data);
  return (
    <form onSubmit={handleSubmit(onSubmit)}>
      <input {...register("firstName")} placeholder="First Name" />
      <input {...register("lastName")} placeholder="Last Name" />
      <button type="button" onClick={() => setValue("firstName", "Alice")}>
        Set First Name to Alice
      </button>
      <input type="submit" />
      <pre>{JSON.stringify(watchAllFields, null, 2)}</pre>
    </form>
  );
}
```

Integracja z zewnętrzną paczka

```
import { useForm, Controller } from 'react-hook-form';
import TextField from '@mui/material/TextField';

function MaterialUIForm() {
  const { control, handleSubmit } = useForm();

  return (
     <Controller
       name="firstName"
       control={control}
       render={({ field }) => (
         <TextField
           {...field}
         />
       )
     />
}
```

Zadanie

Stwórz wieloetapowy formularz w *modul-3-forms/dynamic-form*, który przyjmuje wartości:

- nazwa użytkownika
- wiek użytkownika
- rok urodzenia (powinien się sam ustawić po wprowadzeniu wieku)
- zainteresowania użytkownika (możliwość dodania więcej niż 1 pola)

Każda nowy input powinien się pojawiać dopiero po wypełnieniu poprzedniego.

Integracja z yup

```
import * as yup from 'yup';
import { yupResolver } from '@hookform/resolvers/yup';


const schema = yup.object().shape({
        firstName: yup.string().required('Imię jest wymagane'),
        email: yup.string()
                        .email('Nieprawidłowy email')
                        .required('Email jest wymagany'),
});

const { register } = useForm({ resolver: yupResolver(schema)  });
```

Integracja z yup

```
const MyForm = () => {
  const { register, handleSubmit, formState: { errors } } = useForm({
    resolver: yupResolver(schema)
  });
  return (
    <form onSubmit={...}>
      <div>
        <input {...register('firstName')} />
        {errors.firstName && <p>{errors.firstName.message}</p>}
      </div>
      <div>
        <input {...register('email')} />
          {errors.email && <p>{errors.email.message}</p>}
        </div>
      </form>
    );
}
```

Integracja z yup

```
const schema = yup.object().shape({
  password: yup.string()
    .min(8,'Hasło musi mieć co najmniej 8 znaków')
        .matches(/[a-z]/, 'Musi zawierać co najmniej jedną małą literę')
        .matches(/[A-Z]/, 'Musi zawierać co najmniej jedną dużą literę')
        .matches(/[0-9]/, 'Musi zawierać co najmniej jedną cyfrę')
        .required('Hasło jest wymagane'),
    });

const schema = yup.object().shape({
  password: yup.string()
    .min(8, 'Hasło musi mieć co najmniej 8 znaków')
    .required('Hasło jest wymagane'),
  confirmPassword: yup.string()
    .oneOf([yup.ref('password'), null], 'Hasła muszą się zgadzać')
    .required('Potwierdzenie hasła jest wymagane'),
});
```

Zadanie

Wykonaj taki sam podstawowy formularz jak w basic app, ale z użyciem walidacji przy pomocy **yup** oraz bardziej zaawansowane formularze w *modul-3-forms/yup-form-apps*