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: HTMLInputElement = document.querySelector('input');
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.

React używa własnego systemu zdarzeń. Dlatego nie możesz używać typowych zdarzeń MouseEvents lub podobnych na swoich elementach. Musisz użyć konkretnej wersji typu z React, w przeciwnym razie pojawi się błąd kompilacji.

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

const MyButton = () => (
  <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, void 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());
}
```

Void jest używane tam, gdzie nie ma danych. Na przykład, jeśli funkcja nie zwraca żadnej wartości, możesz określić void jako typ zwracany.

```
function sayHi(): void {
    console.log('Hi!')
}
```

Unknown, void and never

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);
}
```

Undefined i null mają w rzeczywistości swoje typy odpowiednio nazwane niezdefiniowanymi i null.

```
const u: undefined = undefined;
const n: null = null;
```

Domyślnie null i undefined są podtypami wszystkich pozostałych typów. Oznacza to, że możesz przypisać wartość null i niezdefiniowaną do czegoś takiego jak liczba.

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);
// TS wie, że myPet może być Dog 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, control, setValue } = useForm();
  const fieldValue = useWatch({ control, name: “firstName” });
  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>{fieldValue}</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)

Integracja z zod

Zod jako biblioteka do walidacji:

- Zod to deklaratywna biblioteka do walidacji danych, która uprości proces walidacji w aplikacji. Umożliwia definiowanie schematów z użyciem łatwego i czytelnego API.

Zastosowanie w projektach:

- Zod jest powszechnie używany w projektach, w których typowanie i walidacja są kluczowe – może być wykorzystywany z JavaScript oraz TypeScript.

Zachowanie typizacji z TypeScript:

- Zod integruje się z TypeScript, umożliwiając generowanie typów na podstawie zdefiniowanych schematów walidacji. Ułatwia to łapanie błędów w czasie kompilacji.

Integracja z zod

```
import { z } from zod;
import { zodResolver } from '@hookform/resolvers/zod;


const schema = z.object({
        name: z.string().min(1, 'Imię jest wymagane'),
        email: z.string().email('Podaj poprawny email')
});


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

Integracja z zod

```
const MyForm = () => {
  const { register, handleSubmit, formState: { errors } } = useForm({
    resolver: zodResolver(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 zod

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

Integracja z zod

```
export const registerSchema = z
  .object({
    password: z.string()
      .nonempty('Hasło jest wymagane'),
    confirmPassword: z.string()
      .nonempty('Potwierdzenie hasła jest wymagane')
  })
  .superRefine(({ password, confirmPassword} , ctx) => {
    if (password !== confirmPassword) {
      ctx.addIssue({
        code: z.ZodIssueCode.custom,
        message: 'Hasła muszą się zgadzać',
        path: ['confirmPassword'],
      })
    }
  })
```

Zadanie

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

React query

React query

React Query to biblioteka do zarządzania zapytaniami danych w aplikacjach React. Ułatwia pobieranie, buforowanie, synchronizowanie i aktualizowanie danych z serwerów. Zapewnia również mechanizmy do obsługi stanu ładowania, błędów i automatycznego odświeżania danych.

- Automatyczne zarządzanie stanem zapytań, co minimalizuje ręczną obsługę stanów takich jak loading czy error.
- Wbudowane cachowanie danych, co pozwala na efektywne zarządzanie zapytaniami i zmniejszenie liczby żądań HTTP.
- Obsługa automatycznego odświeżania danych, np. po odzyskaniu fokusu przez okno przeglądarki.
- Łatwa integracja z mutacjami danych, co pozwala na bezproblemowe zarządzanie operacjami tworzenia, aktualizacji i usuwania zasobów.
- Optymistyczne aktualizacje, które poprawiają UX, działając natychmiastowo przy przewidywanych wynikach operacji.
- Rozbudowane mechanizmy zarządzania błędami i retry, które zwiększają niezawodność aplikacji.

React query - podstawowe funckje

useQuery: Hook do pobierania danych, który zarządza stanem zapytania i cachowaniem.

useMutation: Hook do zarządzania operacjami zmieniającymi dane, jak dodawanie czy usuwanie.

QueryClient i QueryClientProvider: Narzędzia do konfiguracji globalnego clienta do zarządzania zapytaniami w całej aplikacji.

React query - możliwości

Zarządzanie cachem: Mechanizmy do ustalania czasu ważności danych i strategii ich odświeżania.

Refetching i synchronizacja: Automatyczne odświeżanie danych przy określonych zdarzeniach, takich jak odzyskiwanie fokusu.

Obsługa zależności między zapytaniami: Możliwość konfigurowania zapytań, które zależą od wyników innych zapytań.

Wsparcie dla SSR (Server-Side Rendering): Możliwość prefetchingu danych na serwerze, co jest przydatne w aplikacjach renderowanych po stronie serwera.

Integracja z devtools: Narzędzie do inspekcji zapytań i cache'u, które ułatwia debugowanie i optymalizację.

React query - jak zacząć?

```
npm install @tanstack/react-query
```

```
import React from 'react';
import { QueryClient, QueryClientProvider } from '@tanstack/react-query';

const queryClient = new QueryClient();

function App() {
  return (
    <QueryClientProvider client={queryClient}>
      <YourComponent />
    </QueryClientProvider>
  );
}
```

useQuery

to hook, który służy do pobierania danych. Automatycznie zarządza stanem zapytania i cachowaniem.

```
import { useQuery } from '@tanstack/react-query';
function fetchTodos() {
  return fetch('https://…/todos').then(res =>res.json());
}

function Todos() {
  const { data } = useQuery({
        queryKey: ['todos'],
        queryFn: fetchTodos
});

  return (
    <ul>{data.map(todo => <li key={todo.id}>{todo.title}</li>)}</ul>
  );
}
```

Zadanie

Otwórz aplikację *modul-4-extras/query-app* i dodaj pobieranie danych w komponencie Menu, Admin oraz Details.

Pamiętaj by wykorzystać metody z folderu *services*.

useMutation

jest używany do wykonywania operacji zmieniających dane, takich jak tworzenie, aktualizacja lub usuwanie zasobów. useMutation nie zarządza automatycznie cachowaniem, ale można go używać w połączeniu z innymi funkcjami React Query do aktualizacji danych w cache.

```
import { useMutation, useQueryClient } from '@tanstack/react-query';

function addTodo(newTodo) {
  return fetch('https://jsonplaceholder.typicode.com/todos', {
    method: 'POST',
    headers: {
      'Content-Type': 'application/json',
    },
    body: JSON.stringify(newTodo),
  }).then(res => res.json());
}
```

useMutation

```
function AddTodo() {
  const queryClient = useQueryClient();
  const mutation = useMutation({
        mutationFn: addTodo, {
        onSuccess: () => {
              // Invalidate and refetch todos after a successful mutation
              queryClient.invalidateQueries(['todos']);
            },
  });
  return (
    <button
      onClick={() => { mutation.mutate({ title: 'New Todo', completed: false });}}
    >
      Add Todo
    </button>
  );
}
```

Zadanie

Otwórz aplikację *modul-4-extras/query-app* i dodaj interakcje poprzez przesyłanie danych w komponencie Admin (add-modal oraz edit)

Pamiętaj by wykorzystać metody z folderu *services*.

Zarządzanie cachem

React Query oferuje różne mechanizmy zarządzania cachem, które pozwalają na optymalizację pobierania danych i ich synchronizację.

Automatyczne odświeżanie danych, gdy okno przeglądarki odzyskuje fokus.

```
const { data, refetch } = useQuery({
        queryKey: ['todos'],
        queryFn: fetchTodos,
        refetchOnWindowFocus: true,
});
```

Określenie czasu, przez jaki dane są uznawane za "świeże" lub przechowywane w cache.

```
const { data } = useQuery({
        queryKey: ['todos'],
        queryFn: fetchTodos,
        staleTime: 1000 * 60, // 1 minuta
        gcTime: 1000 * 60 * 5, // 5 minut
});
```

Obsługa błędów

React Query oferuje mechanizmy do łatwego zarządzania błędami w zapytaniach i mutacjach, co pozwala na lepsze zarządzanie stanem aplikacji.

Przykład obsługi błędów w useQuery:

```
import { useQuery } from '@tanstack/react-query';
function TodosWithErrorHandling() {
  const { data, error, isLoading, isError } = useQuery({
        queryKey: ['todos'],
        queryFn: fetchTodos
});

  if (isLoading) return <div>Loading...</div>;
  if (isError) return <div>Error: {error.message}</div>;
  return (
    <ul>{data.map(todo => <li key={todo.id}>{todo.title}</li>)}</ul>
  );
}
```

Obsługa błędów

Przykład obsługi błędów w useMutation:

```
function AddTodoWithErrorHandling() {
  const mutation = useMutation({
    mutationFn: addTodo,
    onError: (error) => { console.error('Error dodawania todo:', error); },
    onSuccess: () => { queryClient.invalidateQueries(['todos']); },
});

return (
  <button onClick={() => {
      mutation.mutate({ title: 'New Todo', completed: false });
    }}
  >
    Add Todo
  </button>
);
```

Optymalizacje i zaawansowane użycie

Używaj unikalnych kluczy do rozróżniania zapytań.

```
const { data } = useQuery({
        queryKey:['todos', userId],
        queryFn:() => fetchTodosByUser(userId)
});
```

Zapytania, które zależą od innych danych.

```
const { data: user } = useQuery({
        queryKey:['todos', userId],
        queryFn: fetchUser
});
const { data: projects } = useQuery({
  queryKey: ['projects', user?.id],
  queryFn: fetchProjectsByUser,
  { enabled: !!user }
});
```

Optymalizacje i zaawansowane użycie

Optymistyczna aktualizacja UI:

```
const mutation = useMutation({
   mutationFn: addTodo,
   onMutate: async (newTodo) => {
       // Anuluj inne zapytania, aby uniknąć konfliktów
        await queryClient.cancelQueries(['todos']);
       // Zapisz aktualny stan cache
        const previousTodos = queryClient.getQueryData(['todos']);
       // Optymistycznie zaktualizuj cache
        queryClient.setQueryData(['todos'], (old) => [ ...old, newTodo]);
       // Zwróć funkcję rollback w przypadku błędu
        return { previousTodos };
       },
  onError: (err, newTodo, context) => {
       queryClient.setQueryData(['todos'], context.previousTodos)
       },
  onSettled: () => { queryClient.invalidateQueries(['todos']); },
});
```

useSuspenseQuery

to funkcja w React Query, która łączy użycie Suspense z zapytaniami.

Wykorzystujemy **useSuspenseQuery** do pobierania danych. Gdy dane są ładowane, React automatycznie obsługuje stan ładowania i przechodzi do komponentu zdefiniowanego w props **fallback** komponentu Suspense.

useSuspenseQuery

```
import { useSuspenseQuery } from '@tanstack/react-query';

function Users() {
  const { data } = useSuspenseQuery(['users'], fetchUsers);
  return (
    <ul>{data.map(user => <li>{user.name}</li>)}</ul>
  );
}

function App() {
  return (
    <QueryClientProvider client={queryClient}>
      <React.Suspense fallback={<div>Loading...</div>}>
        <Users />
      </React.Suspense>
    </QueryClientProvider>
  );
}
```

Zadanie

Wykorzystaj **useSuspenseQuery** w komponencie Menu zamiast klasycznego useQuery. Zmień stan ładowanie na React.Suspense.

Dodaj optymistyczne dodawanie danych po dodaniu nowego elementu w Admin (add modal).

react-i18next

i18n

i18n (Internationalization), czyli proces przygotowywania aplikacji do obsługi różnych języków i lokalizacji bez konieczności zmiany kodu źródłowego

react-i18next to framework internacjonalizacji dla React / React Native, który jest oparty na **i18next**.

Konfiguracja

```
import i18n from 'i18next';
import { initReactI18next } from 'react-i18next';

i18n.use(initReactI18next)
  .init({
    resources: {
      en: { translation: {"welcome": "Welcome to React"} },
      de: { translation: {"welcome": "Willkommen bei React"}}
    },
    lng: "en",
    fallbackLng: "en",
    interpolation: { escapeValue: false }
  });

export default i18n;
```

Konfiguracja

```
import i18n from 'i18next';
import { initReactI18next } from 'react-i18next';
import Backend from 'i18next-http-backend';
import LanguageDetector from 'i18next-browser-languagedetector';

i18n.use(Backend).use(LanguageDetector).use(initReactI18next)
  .init({
    fallbackLng: 'en',
    debug: true,
    backend: { loadPath: '/locales/{{lng}}/translation.json' },
    interpolation: {
      escapeValue: false
    }
  });
```

i18next-http-backend

i18next-http-backend jest pluginem do i18next, który umożliwia ładowanie plików tłumaczeń przez HTTP. Używa się go głównie w aplikacjach frontendowych, gdzie tłumaczenia są przechowywane na serwerze, a ładowane są bezpośrednio do przeglądarki użytkownika w czasie wykonywania aplikacji.

Główne funkcje:

- Ładowanie tłumaczeń z serwera: Umożliwia dynamiczne pobieranie plików tłumaczeń poprzez zapytania HTTP. To jest przydatne, gdy chcesz oddzielić logikę aplikacji od danych tłumaczenia.
- Minimalizacja rozmiaru początkowego pakietu: Tłumaczenia są ładowane na żądanie, co oznacza, że aplikacja nie musi ładować wszystkich języków od razu, co może zmniejszyć rozmiar początkowy pakietu aplikacji.

i18next-browser-languagedetector

i18next-browser-languagedetector jest pluginem do i18next, który automatycznie wykrywa język preferowany przez użytkownika na podstawie informacji dostępnych w przeglądarce.

Główne funkcje:

- Wykrywanie języka z przeglądarki: Rozpoznaje język, analizując różne źródła danych dostępne w przeglądarce, takie jak:
  - Ustawienia językowe przeglądarki (navigator.language).
  - Język strony (tagi HTML, nagłówki HTTP).
  - Parametry GET w URL.
  - Preferencje zapisane w lokalnym storage lub ciasteczkach.
- Automatyczne dopasowanie: Umożliwia stosowanie najlepszych praktyk w zakresie ustalania języka domyślnego dla użytkowników, co zwiększa przyjazność dla użytkownika.

Tłumaczenie

```
import React from 'react';
import { useTranslation } from 'react-i18next';

function MyComponent() {
  const { t } = useTranslation();
  return <h1>{t('welcome')}</h1>;
}
```

Zmienne

```
{
  "greeting": "Hello, {{name}}! Welcome to our website.",
  "items": "You have {{count}} item.",
  "items_plural": "You have {{count}} items."
}

function App() {
  const { t } = useTranslation();
  return (
    <div>
      <p>{t('greeting', { name: "John" })}</p>
      <p>{t('items', { count: 5 })}</p>
    </div>
  );
}
```

Trans

Komponent Trans w react-i18next umożliwia renderowanie złożonych struktur tekstowych z dynamicznymi wartościami i wbudowanym HTML-em, bez konieczności pisania dużej ilości dodatkowego kodu. Jest bardzo przydatny, gdy trzeba wyrenderować elementy w tłumaczonych tekstach, na przykład pogrubione fragmenty, linki lub dynamicznie wstawiane wartości.

```
{
  "welcome": "Welcome to the <strong>{{siteName}}</strong> website!",
  "description": "To get started, <a>click here</a>."
}
```

```
<Trans i18nKey="welcome"
       values={{ siteName: siteName }}
       components={{ a: <a href="https://softiq.pl/" /> }}
/>
```

infoShare ACADEMY

Zmiana języka

```
import React from 'react';
import { useTranslation } from 'react-i18next';

function LanguageSwitcher() {
  const { i18n } = useTranslation();
  const changeLanguage = (lng) => {
    i18n.changeLanguage(lng);
  };
  return (
    <div>
      <button onClick={() => changeLanguage('en')}>English</button>
      <button onClick={() => changeLanguage('de')}>Deutsch</button>
    </div>
  );
}
```

Zadanie

Dodaj tłumaczenie tekstów zgodnie z plikami translate w aplikacji *module-4-extras/i18n-app*.

Pamiętaj o konfiguracji i jej wczytaniu. Wykorzystaj komponent Trans do wygenerowania tagów HTML'a.

Reconciliation

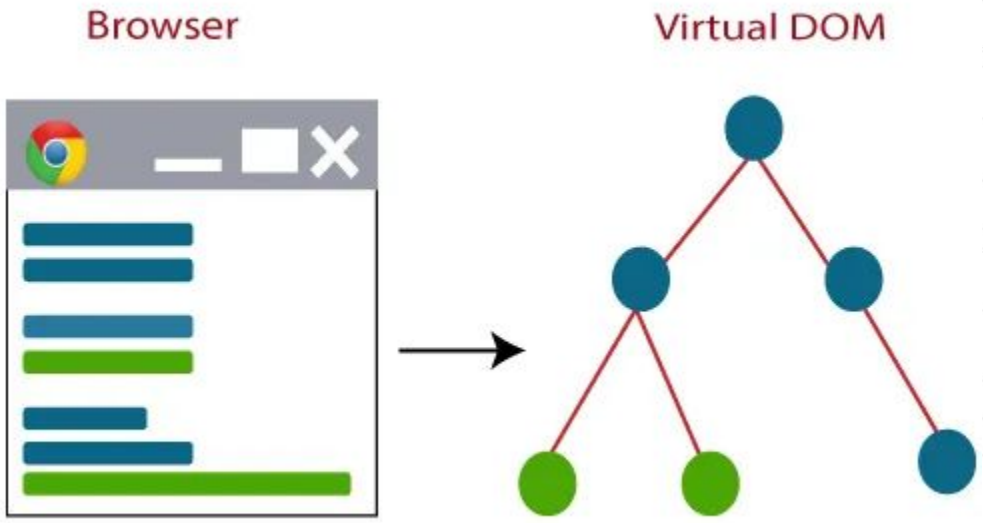
infoShareAcademy.com

Reconciliation

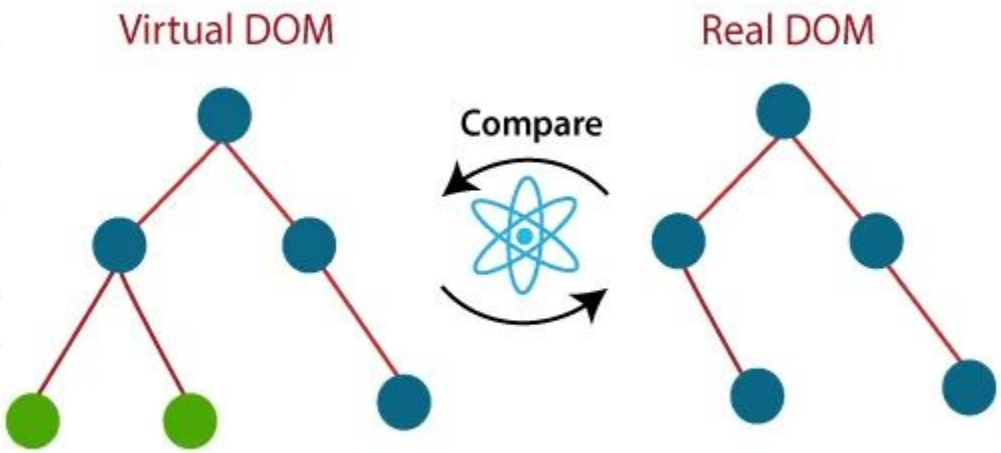
Reconciliation to proces porównywania nowego drzewa elementów React z poprzednim drzewem w celu określenia najefektywniejszego sposobu aktualizacji interfejsu użytkownika. Pozwala to utrzymywać aktualny stan aplikacji przy minimalnych zmianach w DOM, co znacznie poprawia wydajność.

Algorytm porównujący, inaczej diffing algorithm, działa poprzez sprawdzenie różnic między nową a starą wersją drzewa wirtualnego DOM. Operacje takie jak dodanie, usunięcie czy zmiana węzłów są zoptymalizowane w celu zredukowania rzeczywistych modyfikacji w DOM.

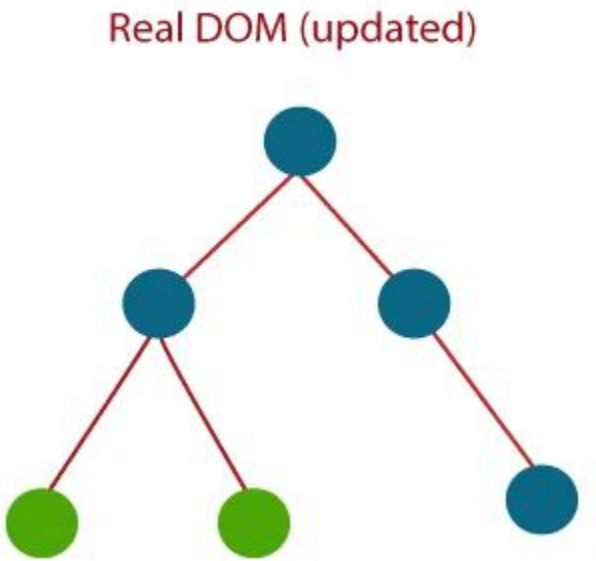
Reconciliation



Reconciliation



Reconciliation



Diffing algorithm

Kiedy stan komponentu się zmienia, React tworzony jest nowy Virtual DOM.
Stary Virtual DOM i nowy Virtual DOM są porównywane w celu znalezienia różnic.

React stosuje kilka kluczowych zasad, które pozwalają na efektywne porównywanie:

- Porównanie typów elementów: Jeśli elementy mają różne typy (np. zamiast), React zakłada, że coś się zmieniło i kasuje stary element, a następnie tworzy nowy.
- Zachowanie struktury: Jeśli elementy są tego samego typu, React porównuje atrybuty i dzieci, aby zidentyfikować, co można zaktualizować.
- Używanie kluczy (keys): Jeśli w drzewie elementów są listy, klucze pomagają Reactowi w identyfikacji elementów. Pozwala to na optymalizację pozostawiania niezmienionych elementów na swoich miejscach, co redukuje ilość operacji na DOM.

Diffing algorithm

Dla każdego elementu w drzewie, który jest tego samego typu, React rekurencyjnie porównuje jego dzieci. W miarę jak drzewo jest przeszukiwane, React może szybko zidentyfikować, które elementy się zmieniły, a które można pozostawić bez zmian.

Po zakończeniu procesu diffing, React generuje listę operacji, które muszą zostać wykonane na rzeczywistym DOM (np. aktualizacja atrybutów, dodanie nowych elementów, usunięcie starych). Te operacje są następnie wykonywane w najefektywniejszy sposób.

Diffing algorithm

Przykład:
Przykładowa struktura Virtual DOM przed i po zmianie:

Stary Virtual DOM:

```
<div>
  <h1>Hello</h1>
  <p>World</p>
</div>
```

Nowy Virtual DOM:

```
<div>
  <h2>Hello, World!</h2>
  <p>React</p>
</div>
```

Typy elementów h1 i h2 różnią się, więc React zastąpi h1 nowym h2.
Zmiana tekstu wewnętrznego: React zaktualizuje zawartość.
p pozostanie, ale tekst wewnętrzny będzie zaktualizowany.
Dzięki temu podejściu, React minimalizuje liczbę operacji na rzeczywistym DOM, co znacznie poprawia wydajność aplikacji.

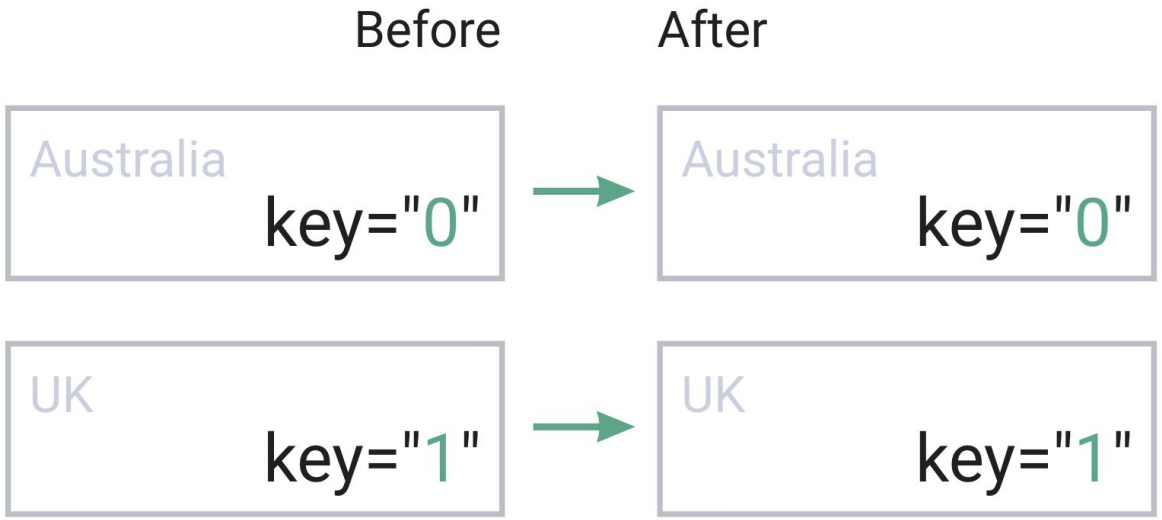
Keys

Klucze są niezbędne dla identyfikacji i śledzenia elementów podczas ich zmian w liście. React używa ich do ustalania, które elementy zostały zmienione, dodane lub usunięte, zapobiegając niepotrzebnym operacjom w DOM.

```
function UserList({ users }) {
  return (
    <ul>
      {users.map(user => (
        <li key={user.id}>{user.name}</li>
      ))}
    </ul>
  );
}
```

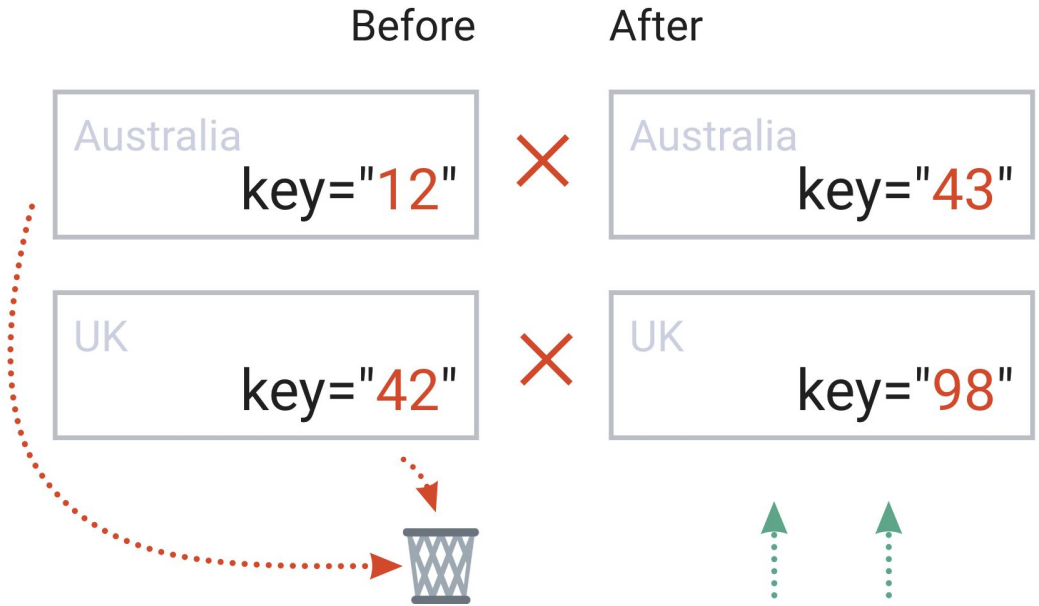
Keys

```
countries.map((country, index) => <Item country={country} key={index} />);
```



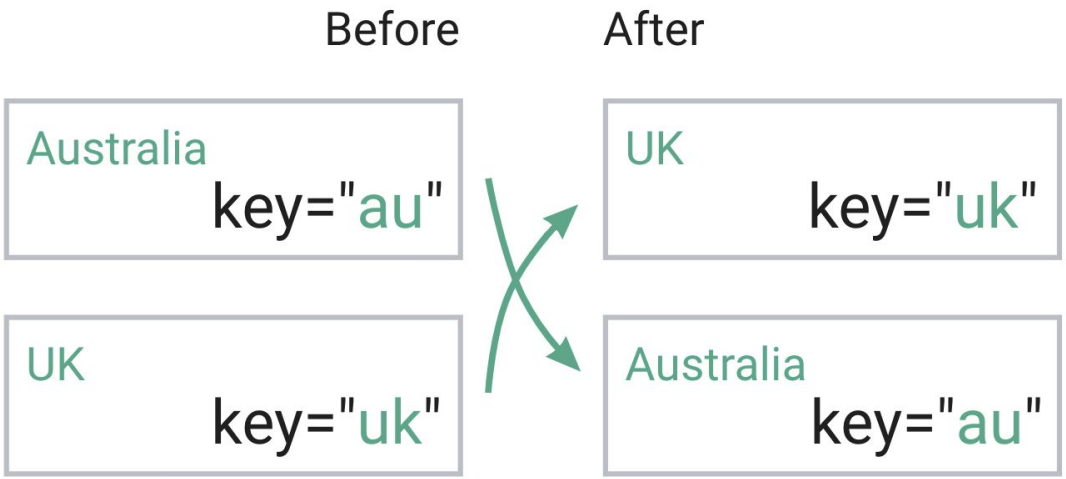
Keys

```
countries.map((country) => <Item country={country} key={Math.random()} />);
```



Keys

```
countries.map((country) => <Item country={country} key={country.id} />);
```



Zadanie

Otwórz aplikację *modul-5-reconciliation/keys-map.*

Sprawdź co się dzieję z aplikacją. Zwróć uwagę, kiedy elementy się przerendorowywują.

Dodatkowo otwórz **Performance** tab w Chrome Devtools i zmień CPU na wolniejszy throttling i zobacz różnice w wydajności obu podejść.

Keys

W React klucz (key) jest często używany w kontekście renderowania list, ale ma również swoje zastosowanie w innych przypadkach, na przykład w komponentach kontrolowanych i dynamicznych.

Użycie klucza jest skutecznym sposobem na wymuszenie pełnego re-renderowania komponentu, ponieważ React identyfikuje komponenty na podstawie ich kluczy. Jeśli zmienisz klucz komponentu, React traktuje go jako nowy komponent i renderuje go od nowa.

Keys

```
const App = () => {
    const [resetKey, setResetKey] = useState(0);

    useEffect(() => {
        setInterval(() => {
            setResetKey(Date.now())
        }, 5000)
    });

  return (
    <WeatherPlugin key={resetKey} />
  );
};
```

Zadanie

Otwórz aplikację *modul-5-reconciliation/key-app*.

Zaimplementuj resetowanie timer tak by faktycznie wszystkie funkcję się zresetowały.

Profiler

Profiler to narzędzie w React Developer Tools, które umożliwia mierzenie czasu renderowania komponentów i identyfikację wąskich gardeł wydajnościowych.

Można w nim również włączyć opcję podświetlania prze renderowania komponentów w ustawieniach w zakładce general.

Zadanie

Otwórz aplikację *modul-5-reconciliation/profiler-app.*

Przeładuj aplikację razem z profilerem by zobaczyć, który element najwolniej się załadował.

Uruchom podświetlanie przeładowywania komponentów i kliknij w przyciski. Postaraj się zoptymalizować re renderowanie.

Optimisations

infoShareAcademy.com

Code splitting

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>
  );
}
```

Code splitting

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

Otwórz aplikację *modul-6-optimisation/lazy-app.*

Dodaj leniwe ładowanie do wszystkich ścieżek (routes) i sprawdź jak zachowuje się doładowywanie kodu w zakładce Network w devtools.

Memo hooks

Memo hooks w React, czyli **useMemo** i **useCallback**, są używane do optymalizacji wydajności komponentów poprzez zapobieganie niepotrzebnym przeliczeniom i re-rendrom.

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

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

useMemo

useMemo to hook React, który pozwala buforować wyniki obliczeń pomiędzy ponownymi renderowaniami.

```
const calculateSum = (items) => {
   return items.reduce((acc, item) => acc + item, 0);
};

const filteredItems = useMemo(() => {
   return items.filter(item => item.toString().includes(filter));
}, [items, filter]);

const sum = useMemo(() => calculateSum(filteredItems), [filteredItems]);
```

useMemo

useMemo to hook React, który pozwala buforować wyniki obliczeń pomiędzy ponownymi renderowaniami.

```
const calculateSum = (items) => {
   return items.reduce((acc, item) => acc + item, 0);
};

const filteredItems = useMemo(() => {
   return items.filter(item => item.toString().includes(filter));
}, [items, filter]);

const sum = useMemo(() => calculateSum(filteredItems), [filteredItems]);
```

Kiedy używać useMemo?

Złożone obliczenia:

```
function complexCalculation(list) {
  // Complex calculation here
  return result;
}

function MyComponent({ list }) {
  const memoizedResult = useMemo(() => {
    return complexCalculation(list)
  }, [list]);

  return <div>{memoizedResult}</div>;
};
```

Kiedy używać useMemo?

Równość referencyjna:
useMemo pomaga zachować te same odniesienia dla obiektów lub tablic podczas przekazywania ich jako prop, unikając niepotrzebnych renderowań.

```
function MyComponent({ values }) {
  const memoizedValues = useMemo(() => values, [values]);
  return <MemoizedChildComponent values={memoizedValues} />;
};

function ChildComponent({ values }) {
  // ...
});

const MemoizedChildComponent = React.memo(ChildComponent)
```

Kiedy używać useMemo?

Filtrowanie lub sortowanie długich list:
Podczas pracy z dużymi zbiorami danych użycie useMemo do filtrowania lub sortowania zapobiega ponownemu wykonywaniu tych operacji podczas każdego renderowania.

```
function MyComponent({ list }) {
  const filteredList = useMemo(() => {
    return list.filter(item => item.active);
  }, [list]);

  return <ListDisplay items={filteredList} />;
};
```

Kiedy nie używać useMemo?

Proste obliczenia:

```
function MyComponent({ value }) {
  const multipliedValue = useMemo(() => {
    return value * 2;
  }, [value]);

  return <div>{multipliedValue}</div>;
};
```

Kiedy nie używać useMemo?

Kiedy używamy prostych typów danych:
Zapamiętywanie prymitywnych wartości, takich jak liczby lub ciągi znaków, jest generalnie przesadą, ponieważ nie powodują niepotrzebnego renderowania.

```
function MyComponent({ value }) {
  // Not recommended for primitive values
  const memoizedValue = useMemo(() => value + 1, [value]);

  return <div>{memoizedValue}</div>;
};
```

Kiedy nie używać useMemo?

Częste zmiany depedencji

Jeśli dane wejściowe często się zmieniają, „useMemo” traci swoją skuteczność, ponieważ często wykonuje przeliczenia.

```
function MyComponent({ frequentlyChangingValue }) {
  // Not recommended for frequently changing dependencies
  const memoizedValue = useMemo(() => {
    return expensiveCalculation(frequentlyChangingValue);
  }, [frequentlyChangingValue]);

  return <div>{memoizedValue}</div>;
};
```

useCallback

useCallback to hook reagujący, który pozwala buforować definicję funkcji pomiędzy ponownymi renderowaniami.

```
const handleSubmit = useCallback((orderDetails) => {
  post('/product/' + productId + '/buy', {
    referrer,
    orderDetails,
  });
}, [productId, referrer]);
```

React.memo

React.memo to wyższy komponent, który umożliwia optymalizację renderowania komponentów funkcjonalnych poprzez zapamiętywanie ich i unikanie zbędnych renderów, jeśli propsy się nie zmieniają.

Zastosowanie: Najczęściej wykorzystywane w komponentach, które renderują się w oparciu o propsy i nie mają swojego własnego stanu.

React.memo

Używamy, gdy komponent renderuje się często, ale jego propsy rzadko się zmieniają.

Kiedy masz złożony komponent, w którym optymalizacja wydajności jest istotna. Gdy propsy komponentu są wartościami prostymi (np. liczby, łańcuchy znaków) albo obiektami, które są niezmienne (immutable).

Jak działa React.memo?

React.memo wykorzystuje płaskie porównanie propsów do wykrywania ich zmian. Jeśli nie wykryje zmian, komponent nie będzie renderowany ponownie.

Możesz również zastosować własną funkcję porównującą, aby zdefiniować logikę porównywania propsów.

React.memo

```
const MyComponent = React.memo(({ name }) => {
  console.log("Rendering:", name);
  return <div>{name}</div>;
});

function App() {
  const [name, setName] = React.useState("Jan");
  const [prop, setProp] = React.useState(1);
  return <div>
      <MyComponent name={name} />
      <button onClick={() => setName("Anna")}>Zmień imię</button>
      <button onClick={() => setProp(prop + 1)}>Rerender</button>
    </div>
}
```

React.memo

Własna funkcja porównująca:

```
const MyComponent = React.memo(({ name }) => {
  console.log("Rendering:", name);
  return <div>{name}</div>;
}, (prevProps, nextProps) => prevProps.name === nextProps.name);
```

React.memo a PureComponent

React.PureComponent automatycznie implementuje shallow comparison dla propsów i stanu, co oznacza, że jeśli propsy lub stan się zmieniają, komponent jest renderowany ponownie.

React.memo działa na komponentach funkcyjnych, podczas gdy **PureComponent** dotyczy komponentów klasowych.

Kiedy nie używać React.memo?

Oto kilka sytuacji, w których nie jest zalecane używanie React.memo w React:

1. Proste komponenty lub elementy

- Jeśli komponent jest prostym komponentem, który renderuje statyczną treść lub nie wymaga dużej logiki renderowania, użycie React.memo może być zbędne. W takich przypadkach overhead związany z porównywaniem propsów może przewyższyć korzyści.

2. Komponenty z dużą ilością stanu

- Gdy komponent ma własny stan lub działa jako kontroler dla mechanizmu zarządzania stanem (np. lokalnie, z użyciem useState), nie ma sensu używać React.memo, ponieważ lokalne zmiany stanu spowodują i tak ponowny render.

Kiedy nie używać React.memo?

3. Często zmieniające się propsy

- Jeśli propsy przekazywane do komponentu często się zmieniają, użycie React.memo może prowadzić do niepotrzebnych porównań wydajnościowych, które i tak nie przyniosą korzyści.

4. Złożoność porównania

- Gdy masz złożone obiekty jako propsy i porównywanie ich przy pomocy shallow comparison nie działa (np. potrzebujesz głębokiego porównania), wtedy React.memo nie zadziała prawidłowo.

Kiedy nie używać React.memo?

5. Nadmiarowe użycie

- Użycie React.memo w każdym komponencie może prowadzić do nadmiernej złożoności w aplikacji. Warto stosować to narzędzie w myśl zasady "później, a nie wcześniej".

6. Brak korzyści w małych projektach

- W małych projektach czy prototypach, gdzie wydajność nie jest dużym problemem, używanie React.memo może być przesadą.

Zadanie

Otwórz aplikację *modul-6-optimisation/memo-app* i wykorzystaj hooki useMemo, useCallback oraz React.memo do optymalizacji rerenderingu w komponentach: *Memo, Callback i UserDataManager*

useRef

to hook w React, który pozwala na przechowywanie mutowalnych obiektów, które nie powodują ponownego renderowania komponentu, gdy ich wartość się zmienia. Używany najczęściej do uzyskiwania dostępu do DOM lub przechowywania wartości, które są stałe przez cykl życia komponentu.

```
function Example() {
  const inputRef = useRef(null);

  const focusInput = () => { inputRef.current && inputRef.current.focus(); };

  return <div>
     <input ref={inputRef} type="text" placeholder="Kliknij przycisk, by skupić" />
     <button onClick={focusInput}>Skup się na polu wejścia</button>
   </div>
}
```

infoShare ACADEMY

useId

to hook wprowadzony w React 18, który generuje unikalne identyfikatory dla komponentów renderowanych w przeglądarkach.
Używany do poprawy dostępności i kodowania komponentów, które wymagają unikalnych atrybutów id, takich jak etykiety i pola formularzy.

```
<label>
 Password:
 <input
  type="password"
  aria-describedby="password-hint"
 />
</label>
<p id="password-hint">
 The password should contain at least 18 characters
</p>
```

infoShare ACADEMY

useLayoutEffect a useEffect

useLayoutEffect to hook w React, który wykonuje kod przed tym, jak przeglądarka pomaluje (wyświetli) zmiany w UI. To daje Ci kontrolę nad zmianami, zanim użytkownicy zobaczą jakiekolwiek aktualizacje.

useEffect to hook uruchamiany po renderowaniu komponentu.
Może powodować opóźnienia w aktualizacji DOM, ponieważ działa asynchronicznie. Idealny do operacji, które nie wpływają na wizualizację, np. przesyłanie danych do API, subskrypcje itp.

useLayoutEffect a useEffect

Jeśli nie musisz manipulować DOM lub robić pomiarów, lepiej używać **useEffect**, ponieważ działa on po renderowaniu i nie wpływa na wydajność. **useEffect** jest asynchroniczny, co oznacza, że nie blokuje UI.

useEffect: użyj go, gdy chcesz mieć efekty uboczne, które NIE wymagają synchronizacji z DOM (np. pobieranie danych).

useLayoutEffect: użyj go, gdy potrzebujesz naprawić coś w DOM i chcesz, aby wszystko wyglądało dobrze dla użytkownika zanim cokolwiek się wyświetli.

```
useLayoutEffect(() => {
  if (someCondition) {
    const element = elementRef.current;
    element.style.backgroundColor = 'blue';
  }
}, [dependency]);
```

Zadanie

Otwórz aplikację *modul-6-optimisation/memo-app* i wykorzystaj hooki useRef do optymalizacji rerenderingu w komponentach:

- *ReactHookForm*: tak by zmiany w formularzu nie wywoływały przerenderowania
- *FetchData*: tak by dane pobierały się dopiero od pierwszej zmiany

State management

infoShareAcademy.com

Centralized State Management

Stan aplikacji jest przechowywany w jednym centralnym miejscu (tzw. store).

Zalety:

- Przewidywalność: Łatwiejsze debugowanie dzięki jednoznaczności stanu.
- Globalny dostęp: Każdy komponent może łatwo uzyskać dostęp do stanu.
- Middleware: Wspiera dodatkowe funkcjonalności, takie jak asynchroniczne operacje.

Wady:

- Złożoność: Może być zbyt rozbudowane dla małych aplikacji.
- Wydajność: Może powodować niepotrzebne renderowanie komponentów.

Distributed State Management

Stan jest rozproszony pomiędzy różnymi komponentami, bez centralnego przechowywania.

Zalety:

- Prostota: Łatwiejsza implementacja w małych projektach.
- Izolacja stanu: Każdy komponent zarządza swoim stanem, co może poprawić wydajność.

Wady:

- Kompleksowość: Trudniej wprowadzić globalne zmiany w stanie.
- Trudności w debugowaniu: Może być trudniej zrozumieć, skąd pochodzi dany stan.

Distributed vs Centralized

Centralized: Gdy aplikacja jest duża, złożona, lub gdy wymagana jest wysoka kontrola nad stanem.
Distributed: Gdy aplikacja jest mniejsza, prostsza, lub gdy stan nie musi być globalnie dostępny.

Distributed vs Centralized

Context API

- Wbudowane narzędzie dostarczane z Reactem.
- Wymaga minimalnej konfiguracji.
- Specjalnie zaprojektowane do statycznych danych, które nie są często odświeżane lub aktualizowane.
- Dodawanie nowych kontekstów wymaga tworzenia od podstaw.
- Debugowanie może być trudne w mocno zagnieżdżonej strukturze komponentów React, nawet z narzędziami deweloperskimi.
- Logika UI i logika zarządzania stanem znajdują się w tym samym komponencie.

Gdy aplikacja jest mniejsza, prostsza, lub gdy stan nie musi być globalnie dostępny.

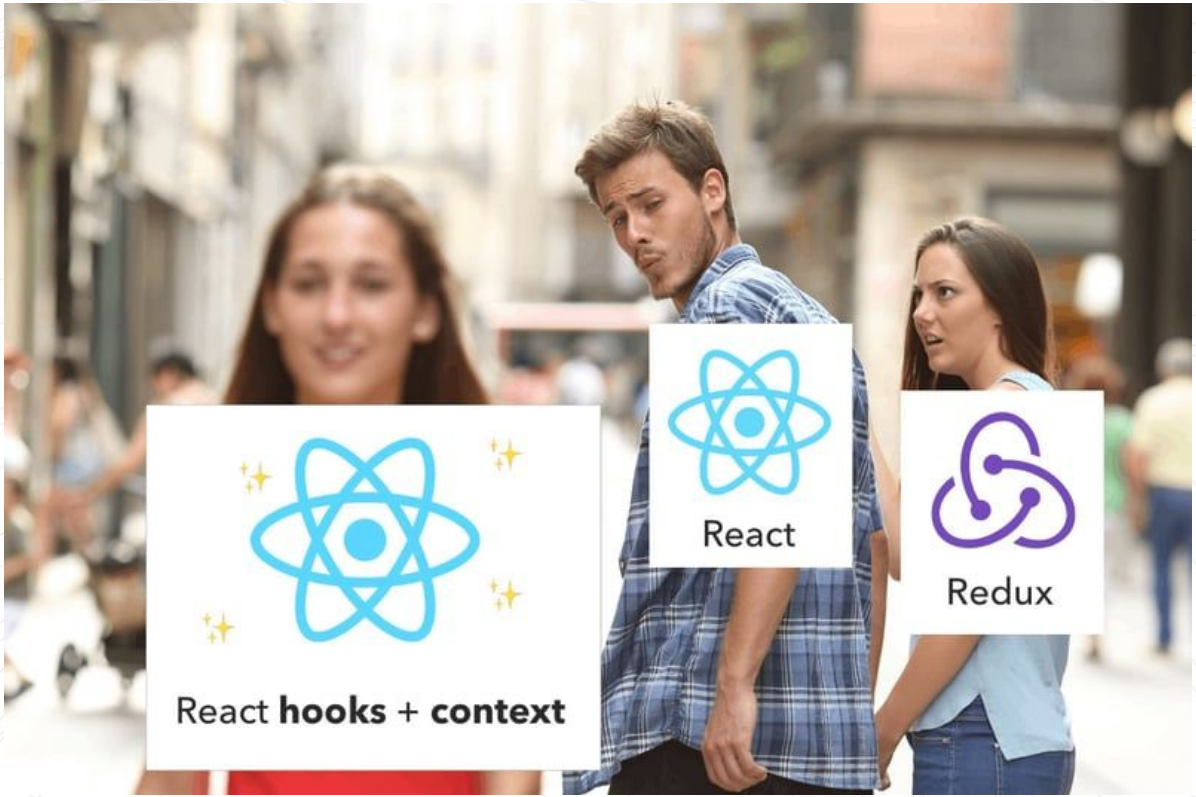
Distributed vs Centralized

Redux

- Wymagana dodatkowa instalacja, co zwiększa rozmiar końcowego pakietu.
- Wymaga obszernej konfiguracji, aby zintegrować ją z aplikacją React.
- Działa doskonale zarówno z danymi statycznymi, jak i dynamicznymi.
- Łatwe do rozszerzenia dzięki prostocie dodawania nowych danych/akcji po początkowej konfiguracji.
- Niezwykle potężne narzędzia deweloperskie Redux ułatwiają debugowanie.
- Lepsza organizacja kodu dzięki oddzieleniu logiki UI od logiki zarządzania stanem.

Gdy aplikacja jest duża, złożona, lub gdy wymagana jest wysoka kontrola nad stanem.

Distributed vs Centralized



useReducer

to hook, który umożliwia zarządzanie stanem w komponentach poprzez reduktory, podobnie jak w Redux. Jest przydatny do skomplikowanego zarządzania stanem, gdy wiele wartości w stanie zależy od siebie.

```
function reducer(state, action) {
  switch (action.type) {
    case 'increment':
      return { count: state.count + 1 };
    case 'decrement':
      return { count: state.count - 1 };
    default:
      throw new Error();
  }
}
```

useReducer

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

function Counter() {
  const [state, dispatch] = useReducer(reducer, { count: 0});

  return (
    <div>
      <p>Licznik: {state.count}</p>
      <button onClick={() => dispatch({ type: 'increment' })}>Zwiększ</button>
      <button onClick={() => dispatch({ type: 'decrement' })}>Zmniejsz</button>
    </div>
  );
}
```

useState

to hook, który umożliwia dodanie stanu lokalnego do komponentów funkcyjnych. Warto pamiętać, że operacja zmiany stanu jest asynchroniczna, czyli funkcja do aktualizacji stanu (setState) nie aktualizuje stanu natychmiastowo. Zamiast tego - zaplanowuje aktualizację stanu, co oznacza, że odczytanie stanu zaraz po jego aktualizacji może zwrócić starą wartość.

```
const handleClick = () => {
  setCount(count + 1);
  console.log(count); // stara wartość
};
```

Zadanie

Otwórz aplikację *modul-7-state/reducer-app* :

- popraw komponent BrokenLocalState tak by działał poprawnie
- uzupełnij ReduxProvider za pomocą useReducer i wykorzystaj go w komponencie counter

Render props

to wzorzec, który pozwala na udostępnienie logiki komponentu w formie funkcji, która zwraca elementy React. Jest to przydatne do dzielenia się kodem i logiką pomiędzy komponentami. Polega on na tym, że komponent przyjmuje funkcję jako prop. Funkcja ta zwraca elementy React i może korzystać z danych przekazanych do niej.

```
<MouseTracker render={({ x, y }) => (
  <h1>
    Pozycja kursora: {x}, {y}
  </h1>
)} />
```

Render props

```
const MouseTracker = ({ render }) => {
  const [position, setPosition] = useState({ x: 0, y: 0 });
  const handleMouseMove = (event) => {
    setPosition({
      x: event.clientX,
      y: event.clientY,
    });
  };

  return (
    <div style={{ height: '100vh' }} onMouseMove={handleMouseMove}>
      {render(position)}
    </div>
  );
};
```

Render (children) props

Ważne jest, aby pamiętać, że tylko dlatego, że wzór nazywa się “render props”, nie musisz używać props o nazwie render, aby użyć tego wzorca. W rzeczywistości każdy element będący funkcją używaną przez komponent do określenia, co ma zostać wyrenderowane, technicznie rzecz biorąc, jest „właściwością renderowania”.

Chociaż powyższe przykłady wykorzystują render props, równie dobrze możemy użyć children

```
<DataProvider>
  {(count) => (
    <h1>Aktualny stan licznika: {count}</h1>
  )}
 </DataProvider>
```

Render (children) props

```
const DataProvider = ({ children }) => {
  const [count, setCount] = useState(0);

  const increment = () => {
    setCount(count + 1);
  };

  return (
    <div>
      <button onClick={increment}>Zwiększ</button>
      {children(count)}
    </div>
  );
};
```

Higher order component

to wzorzec programowania w React, który pozwala na reużywanie logiki komponentów. HOC to funkcja, która przyjmuje komponent jako argument i zwraca nowy komponent.

```
const withLogging = (WrappedComponent) => {
  return (props) => {
    console.log('Renderowanie:', WrappedComponent.name);
    return <WrappedComponent {...props} />;
  };
};
const MyComponent = () => {
  return <h1>Witaj w moim komponencie!</h1>;
};
const MyComponentWithLogging = withLogging(MyComponent);
```

Pub/Sub

to wzorzec architektoniczny, który umożliwia komunikację między komponentami poprzez publikowanie i subskrybowanie zdarzeń, co pozwala na luźne powiązanie.

```
const PubSubContext = createContext();

export const usePubSub = () => useContext(PubSubContext)
```

Pub/Sub

```
export const PubSubProvider = ({ children }) => {
  const [subscribers, setSubscribers] = useState({});
  const publish = (event, data) => {
    if (subscribers[event]) { subscribers[event].forEach(callback => callback(data));}
  };
  const subscribe = (event, callback) => {
    setSubscribers(prev => ({ ...prev,[event]: [...(prev[event] || []), callback] }));
  };

  return (
    <PubSubContext.Provider value={{ publish, subscribe }}>
      {children}
    </PubSubContext.Provider>
  );
};
```

Pub/Sub

```
const Publisher = () => {
  const { publish } = usePubSub();

  const handleClick = () => {
    publish('EVENT_NAME', { message: 'Hello from Publisher!' });
  };

  return <button onClick={handleClick}>Publish Event</button>;
};
```

Pub/Sub

```
const Subscriber = () => {
  const { subscribe } = usePubSub();
  const [message, setMessage] = useState('');

  React.useEffect(() => {
    const onMessageReceived = (data) => {
      setMessage(data.message);
    };
    subscribe('EVENT_NAME', onMessageReceived);

  }, [subscribe]);

  return <div>Received Message: {message}</div>;
};
```

Zadanie

Otwórz aplikację *modul-7-state/patterns-app* :

- zamień komponent FetchData na taki co wykorzystuje wzorzec render props i część odpowiedzialna za pobieranie danych jest reużywalna i poprzez props render i url można pobierać różne dane i renderować inny content
- zamień komponent FetchDataChildren podobnie jak powyżej, ale z wykorzystaniem children
- stwórz withFetchData, gdzie będzie logika pobierania danych i wykorzystaj go w Posts

Redux middlewares

Middleware w Reduxie to funkcje, które rozszerzają możliwości Store'a. Pozwalają na przechwytywanie i modyfikowanie akcji oraz wykonywanie efektów ubocznych (np. asynchronicznych wywołań API).

Popularne middleware: redux-thunk, redux-saga, redux-logger.

Middleware są wywoływane pomiędzy akcjami a reducerami. Mają dostęp do metody dispatch i getState. Sposób działania:

1. Akcja jest wysyłana (dispatched).
2. Middleware przechwytuje akcję.
3. Akcja jest przekazywana dalej lub modyfikowana.
4. Reduktory przetwarzają stan na podstawie przetworzonej akcji.

Redux middlewares

Middleware w Reduxie to funkcje zwracające funkcje.

Podstawowa struktura: store => next => action => {}

```
const loggerMiddlewar = store => next => action => {
        console log( 'Dispatching:' , action );
        let result = next(action);
        console.log( 'Next State:' , store.getState( )) ;
        return result ;
};
```

Redux middlewares

Middleware jest dodawane do Redux Store podczas jego tworzenia za pomocą applyMiddleware. Jednak przy korzystaniu z redux toolkit jest to uproszczone.

```
const store = configureStore({
        reducer: rootReducer,
        middleware: (getDefaultMiddleware) => getDefaultMiddleware().concat(thunk)
});


const enhancers = compose(
        applyMiddleware(thunk)
);
const store = createStore(rootReducer, enhancers);
```

Zadanie

Otwórz aplikację *modul-7-state/middleware-app* i dodaj do niej middleware, który po dokonaniu zakupu w koszyku, doda elementy koszyka do rental office.

Redux saga

Saga to biblioteka, której celem jest tworzenie efektów ubocznych aplikacji.

Jest jak osobny wątek w twojej aplikacji, który jest wyłącznie odpowiedzialny za skutki uboczne. Wykorzystuje funkcję ES6 o nazwie generatory. Dzięki temu asynchroniczny kod wygląda jak standardowy synchroniczny kod JavaScript.

Więcej o generatorach:
https://github.com/gajus/gajus.com-blog/blob/master/posts/the-definitive-guide-to-the-javascript-generators/index.md

Redux saga

Redux Saga działa jako middleware, co oznacza, że ma dostęp do pełnego stanu aplikacji i może również wysyłać akcje. Przechwytuje akcje wysyłane do Redux i uruchamia odpowiednie sagas (funkcje generatorów).

Sagas mogą nasłuchiwać na akcje, wykonywać efekty uboczne (np. wywołania API) i wysyłać nowe akcje do reduktorów.

Podstawowe pojęcia:
Saga: Funkcja generatora, która wykonuje efekty uboczne.
Efekty: Obiekty reprezentujące instrukcje dla middleware, np. call, put, takeEvery.

- call(fn, ...args): Wywołuje funkcję asynchroniczną.
- put(action): Wysyła akcję do reducer'a w Redux.
- takeEvery(actionType, saga): Nasłuchuje na określony typ akcji i uruchamia sagę dla każdej takiej akcji.

Redux saga

```
import { call, put, takeEvery } from 'redux-saga/effects';

function* fetchData(action) {
  try {
    const response = yield call(fetch, `https://api.example.com/data`);
    yield put({ type: 'FETCH_SUCCESS', payload: response });
  } catch (error) {
    yield put({ type: 'FETCH_FAILURE', error });
  }
}
function* watchFetchData() {
  yield takeEvery('FETCH_REQUEST', fetchData); //takeLatest
}
export default watchFetchData;
```

Redux saga

Jeżeli więcej niż 1 saga to ...

```
import { all } from 'redux-saga/effects';

const incrementCounterSaga = function*() { ... }
const decrementCounterSaga = function*() { ... }

export default function* rootSaga() {
    yield all([
        incrementSaga(),
        decrementSaga(),
    ])
}
```

Redux saga

```
import createSagaMiddleware from 'redux-saga';
import { createStore, applyMiddleware } from 'redux';
import rootReducer from './reducers';
import rootSaga from './sagas';

// Tworzenie middlewareconst
sagaMiddleware = createSagaMiddleware();
// Tworzenie store z middleware
const store = configureStore({
        reducer: rootReducer,
        middleware: (getDefaultMiddleware) =>
getDefaultMiddleware().concat(sagaMiddleware)
});

sagaMiddleware.run(rootSaga)
```

Zadanie

Otwórz aplikację *modul-7-state/saga-app*

1. stwórz saga, która będzie nasłuchiwać na akcje zakupu w koszyku, i następnie doda elementy koszyka do rental office
2. stwórz saga, która będzie ponownie pobierać burgery po każdym usunięciu lub dodaniu burgera zamiast rozwiązania z użyciem thunk

Dzięki!

Znajdziecie mnie:
https://www.linkedin.com/in/kamil-richert/
https://github.com/krichert