

Kurs

React

Tomasz Bąba

2025

Wymagania

- zainstalowany node.js w wersji 22.X.X i npm 10.X.X
- zainstalowany edytor (zalecany VS Code)

Założenia

- wiedza z zakresu HTML5,
- wiedza z zakresu stylowania - CSS3 (scss),
- wiedza z zakresu JavaScript,
- wiedza na temat asynchroniczności w JS

Czego się nauczyć

- tworzenia projektów od podstaw z **Vite**
- pracy z **JSX** - rozszerzenie składni JS **przypominający** HTML
- tworzenia widoków z podziałem na komponenty
- routingu po stronie klienta
- obsługi formularzy
- **react Hooks**
- podstaw typowania z wykorzystaniem **TypeScript**
- zarządzanie stanem z **context API**
- różnego podejścia do stylowania aplikacji
- komunikacji z API

Podsumowanie

- aplikacja złożona z trzech widoków: listy użytkowników (Users), widoku danych użytkownika (SingleUser), formularza dodawania użytkownika (Form/AddUser),
- **UserContext**
 - przechowuje użytkowników
- **Users**
 - pobiera listę użytkowników
 - wyświetla listę
 - daje możliwość usuwania użytkownika,
 - Wybór (kliknięcie) użytkownika przenosi nas widok szczegółowych danych użytkownika (ustawia odpowiedni url),
- **User** pobiera dane z (dwa warianty):
 - pobiera użytkownika z kontekst (czy zawsze zadziała?)
 - pobiera użytkownika z API

a potem wyświetla pełną informację o użytkowniku. **Z url pobiera id użytkownika**
- **AddUser/Form**
 - dodaje nowego użytkownika do bazy i przekierowuje do **UserList**

JavaScript - implementacja specyfikacji EcmaScript

- przełomową wersją języka JavaScript była wersja oznaczona jako ES6 (ES2015)
- jest wspierana (zaimplementowana) we wszystkich popularnych przeglądarkach caniuse.com/es6
- współcześnie programiści piszą kod, wykorzystując najnowsze funkcjonalności JavaScript, a dzięki transpilerom zachowuje się kompatybilność wsteczną ze starszymi wersjami przeglądarek.
- Najnowszą, ustandaryzowaną wersją jest **ES15** (więcej o nowościach [w3schools](https://www.w3schools.com/js/default.asp))
- Aby móc korzystać z najnowszych funkcjonalności a przy tym zapewnić działanie na starszych wersjach przeglądarek, używa się narzędzi, które transpilują (przekształcają) kod źródłowy do innej, starszej wersji ES

Przydatne narzędzia frontendowca

- **NodeJS** - środowisko uruchomieniowe kodu javascript, interpreter
- **npm** / yarn - package manager, taskrunner - instalacja zależności i odpalanie skryptów
- **npmx** - narzędzie do odpalania paczek z rejestru npm bez instalacji
- TypeScript / **Babel** - transpiler - przekształca źródłowy kod do równoważnego kodu w innej wersji lub języka
- TypeScript - język oparty na JS dodający do niego statyczne typowanie
- Webpack / Rollup / Parcel / esbuild - bundler - składa kod źródłowy w paczki, podczas kompilacji może wykonać dodatkowe zadania np, minifikacja, autoprefix css
- **ESLint** / **TypeScript-ESLint** - narzędzie sprawdzające jakość kodu i informujące o potencjalnych błędach
- **Prettier** - narzędzie do automatycznego formatowania plików

Ciekawostka: **Droga współczesnego frontendowca**

- [developer roadmap](#)
- [frontend roadmap](#)
- [react roadmap](#)

Czym jest aplikacja webowa

Aplikacja webowa to aplikacja uruchamiana w przeglądarce. Główną cechą aplikacji jest interaktywność czyli umożliwienie wykonywania akcji przez użytkownika.

Strony internetowe mają charakter informacyjny czyli nie umożliwiają interakcji z użytkownikiem.

Elementami interaktywnymi są np. formularze, przyciski

Przygotowanie aplikacji z VITE

VITE - narzędzie do wygenerowania szablonu aplikacji z podstawową konfiguracją

```
# npm 6.x
npm create vite@latest szkolenie --template react-ts

# npm 7+, extra double-dash is needed:
npm create vite@latest szkolenie -- --template react-ts

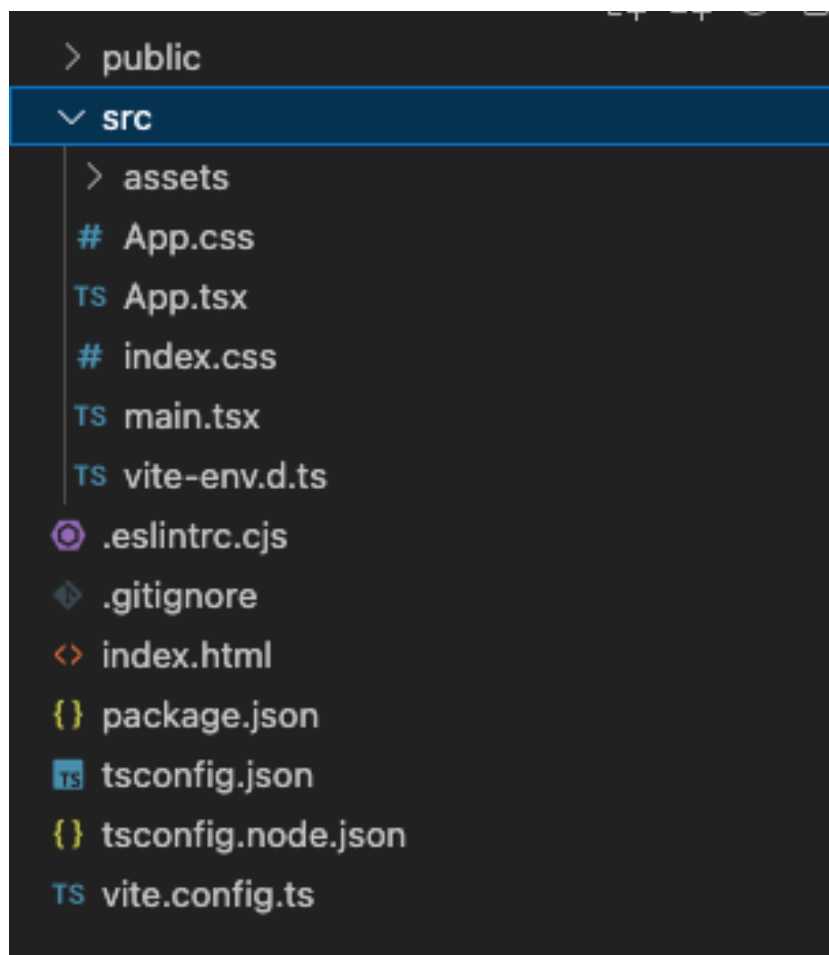
cd szkolenie
npm install
npm run dev
```

Klikamy w link, który pojawi się w konsoli (<http://localhost:5173/>)

VITE instaluje zależności, tworzy szablon aplikacji i konfigurację:

- vite - paczka zawierająca konfigurację i skrypty potrzebne do tworzenia aplikacji, m. in. lokalny serwer, transpilacja, budowanie aplikacji
- Esbuild
- JSX
- Babel
- TypeScript
- CSS Modules oraz klasyczną obsługę CSS
- prosty szablon aplikacji z React, React-DOM i przykładowym komponentem
- ESLint

Struktura plików



ZADANIE

Utworzenie projektu z VITE

```
# npm 6.x
```

```
npm create vite@latest szkolenie --template react-ts
```

```
# npm 7+, extra double-dash is needed:
```

```
npm create vite@latest szkolenie -- --template react-ts
```

```
cd szkolenie
```

```
npm install
```

```
npm run dev
```

Package.json

```
{
  "name": "react-course-v5",
  "private": true,
  "version": "0.0.0",
  "type": "module",
  "scripts": {
    "dev": "vite",
    "build": "tsc && vite build",
    "lint": "eslint . --ext ts,tsx --report-unused-disable-directives\n--max-warnings 0",
    "preview": "vite preview"
  }
}
```

```
"dependencies": {
  "react": "^19.1.0",
  "react-dom": "^19.1.0"
},
"devDependencies": {
  "@eslint/js": "^9.25.0",
  "@types/react": "^19.1.2",
  "@types/react-dom": "^19.1.2",
  "@vitejs/plugin-react": "^4.4.1",
  "eslint": "^9.25.0",
  "eslint-plugin-react-hooks": "^5.2.0",
  "eslint-plugin-react-refresh": "^0.4.19",
  "globals": "^16.0.0",
  "typescript": "~5.8.3",
  "typescript-eslint": "^8.30.1",
  "vite": "^6.3.5"}
```

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <link rel="icon" type="image/svg+xml" href="/vite.svg" />
    <meta name="viewport" content="width=device-width,
initial-scale=1.0" />
    <title>Vite + React + TS</title>
  </head>
  <body>
    <div id="root"></div>
    <script type="module" src="/src/main.tsx"></script>
  </body>
</html>
```

Czym jest React

- biblioteka do tworzenia interfejsów użytkownika i interakcji z nim,
- jest deklaratywna, czyli nie opisujemy, jak ma się zmieniać UI, tylko definiuje, jak ma wyglądać UI w danym momencie,
- wykorzystuje mechanizm **Virtual DOM** (wirtualna reprezentacja prawdziwego drzewa DOM przechowywana w pamięci), dzięki czemu do minimum zmniejsza ilość interakcji z drzewem DOM a co za tym idzie zmiany są bardzo szybkie,
- nie wprowadza żadnych funkcjonalności związanych z komunikacją z API, routingiem czy obsługi API przeglądarki. Takie możliwości są dodawane poprzez zewnętrzne biblioteki (np axios, react-router) czy poprzez pisanie własnych rozwiązań w JS.
- building block reacta to tzw **komponent**
- z pomocą komponentów buduje się duże i skomplikowane aplikacje

Do poprawnego działania niezbędny jest 'most' pomiędzy Reactem a drzewem DOM. Narzędziem, które synchronizuje Virtual DOM z prawdziwym DOM nazywa się ReactDOM.


```
/** main.tsx */  
  
import { StrictMode } from 'react'  
import { createRoot } from 'react-dom/client'  
import './index.css'  
import App from './App.tsx'  
  
createRoot(document.getElementById('root')!).render(  
  <StrictMode>  
    <App />  
  </StrictMode>,  
)
```

- punktem wejściowym aplikacji jest plik (main.tsx, index.tsx), który tworzy root'a a następnie wywołuje na nim metodę .render() z paczki react-dom,
- root to istniejący element z drzewa DOM,
- render przyjmuje w argumencie **element Reactowy** do wyrenderowania,
- przekazany komponent staje się "nadrzędnym" komponentem, w którym możemy dowolnie budować całą aplikację - logika, markup, kompozycje,

JSX - JavaScript XML

- Język podobny do html pozwalający na opisywanie za pomocą “znaczników html” wyglądu strony w kodzie JavaScript.
- JSX jest w czasie kompilacji tłumaczony na funkcje Reactowe (‘prawdziwy’ kod JS), które tworzą **elementy reactowe**. (obiekty JS)
- tagi pisane z małej litery:
 - **<div></div>**, **** odpowiadają tagom html,
- tagi pisane z **wielkiej litery**:
 - **<Header>**, **<App>** to **komponenty**, jest to wymóg niezbędny do poprawnego działania aplikacji.
- **każdy** tag musi posiadać tag zamykający
- możemy zagnieżdżać dowolną ilość tagów i komponentów
- pozwala na łączenie logiki (JS) z markupem (HTML) i stylami (CSS) w jednym pliku

```
<div className="app">
  <Header className="app-header">
    <img
      src={logo}
      className="app-logo"
      alt="logo" />
    <p>
      Edit src/App.tsx and save to reload.
    </p>
    <a
      className="app-link"
      href="https://react.dev/"
      target="_blank"
    >
      Learn React
    </a>
  </Header>
</div>
```

JSX (TSX) - JavaScript XML

- JSX pozwala na osadzanie **wyrażeń** JavaScript razem ze znacznikami html. Wyrażenia wpisuje się pomiędzy klamry **{ }**.
- wyrażenia to nie tylko operacje arytmetyczne ale także odwołanie się do zmiennej
- w JSX atrybuty znaczników zapisujemy w notacji CamelCase (czasem dodawane są suffixy lub prefixy).
np. **class** w JSX to **className**,
click to **onClick** itp.
- najczęściej używane:
 - className,
 - htmlFor,
 - value,
 - onClick,
 - onChange,
 - onFocus

Więcej na:

[Wyrażenia JS](#)

[React props](#)

```
<Header className="app-header">
  <p>
    {a+b}
    {a === 2 ? "true" : "false"}
  </p>

  <input
    type="text"
    onChange={ (e) => {
      console.log("Wpisano", e.target.value)
    }}
  />
</Header>
```

Komponent

- zwykła funkcja JavaScript
- nazwa komponentu (funkcji) **musi** zaczynać się od wielkiej litery
- zwyczajowo nazwy plików zawierające komponenty, nazywa się tak samo jak sam główny komponent
- komponent zwraca kod **jsx**:
 - [elementy reactowe](#)
 - tablice i fragmenty*
 - portale*
 - string, number,
 - boolean, null, undefined
- komponent musi zwracać tylko jeden element - element nadrzędny (**node**)
- element nadrzędny (node) może posiadać dowolną ilość dzieci (zagnieżdżeń)
- z założenia komponenty są reużywalne
- aby nie tworzyć 'pustych' elementów (np. <div>, który nie jest potrzebny w komponencie a jedynie grupuje pozostałe elementy) można wykorzystać tzw.

React Fragment

```
/** App.tsx */  
export default function App() {  
  return <div>  
    Hello World!  
  </div>  
}
```

Hello World!

```
export default function App() {  
  return <div className="greeting">  
    Hello World!  
  </div>  
}
```

Zostanie przekształcone na:

```
export default function App() {  
  return React.createElement(  
    'div',  
    {className: 'greeting'},  
    'Hello world'  
  )  
}
```


Komponent się wyrenderuje czy zgłosił błąd?

```
export function Container() {  
  return <div> Hello </div>  
    <div className="container">World!</div>  
}  
  
export function Container() {  
  return (  
    <div>Hello</div>  
    <div className="container"> World! </div>  
  )  
}
```

```
export function Container() {  
  return <>  
    <div>Hello</div>  
    <div className="container">World!</div>  
  </>  
}
```

```
export function Container() {  
  return <React.Fragment>  
    <div>Hello</div>  
    <div className="container">World!</div>  
  </React.Fragment>  
}
```

ZADANIE

Pierwszy komponent (1)

- usuń zawartość pliku App.tsx
- napisz komponent, który renderuje 'Hello world'

Tworzenie kompozycji

składanie komponentów w jedną aplikację.

- kompozycję tworzymy poprzez zagnieżdżanie elementów reactowych
- zaleca się tworzenie wielu małych komponentów, które są odpowiedzialne za jedną rzecz, niż jeden duży komponent, który robi wiele rzeczy,
- komponent jest standardową funkcją więc jeden plik może zawierać definicje wielu komponentów (czego zwykle się unika)
- lepszym pomysłem jest tworzenie każdego komponentu w osobnych plikach
- JavaScript **modules** pozwala na używanie kodu JS (nie tylko komponentów) między różnymi plikami
- aby pozwolić na **używanie** kodu **w innym pliku** musi być on wyeksportowany
- aby **użyć kod z innego pliku** musimy go zaimportować
- jeśli plik posiada jeden komponent/funkcję to używamy **default export**, jeśli więcej to **named exports**
- elementy javascript importujemy bez podania rozszerzenia, style importujemy podając pełną nazwę pliku (IDE zwykle importuje pliki automatycznie z odpowiednią składnią),

```
function Header() {  
  return <h1>Pierwszy komponent</h1>;  
}  
  
function Profile() {  
  return ;  
}  
  
function App() {  
  return <section>  
    <Header />  
    Hello  
    <div>world!</div>  
    <Profile />  
    <Profile />  
  </section>;  
}
```

Pierwszy komponent

Hello
world!



```
/** Header.tsx */
```

```
export default function Header() {  
  return <h1>Pierwszy komponent</h1>;  
}
```

```
/** Profile.tsx */
```

```
export default function Profile() {  
  return ;  
}
```

```
import Header from "./Header";
import Profile from "./Profile";

export default function App() {
  return <section>
    <Header />
    Hello
    <div>world!</div>
    <Profile />
    <Profile />
  </section>;
}
```


Named Export

```
/** Header.tsx */  
export function Header() {  
  return <h1>Pierwszy komponent</h1>;  
}  
  
/** App.tsx */  
import { Header } from "../Header";
```

Props

- props - zmienne przekazane do komponentu
 - składnia taka jak przy dodawaniu atrybutu do tagu, nadajemy dowolne nazwy
- do każdego komponentu można przekazać props
- każdy atrybut/zmienna przekazana do komponentu jest jego propsem* (bez key)
- props bez podanej wartości zawsze otrzymuje wartość **true**
- props można przekazać z wykorzystaniem "spread operator" (...attrs)
- wszystkie elementy (tagi oraz komponenty) zagnieżdżone w:
 - tagu html - zostaną wyrenderowane automatycznie
 - komponencie - zostaną przekazane do komponentu jako props **children** i nie są automatycznie renderowane
- wszystkie propsy są "składane" do jednego obiektu w komponencie, do którego są przekazywane
- props są tylko do odczytu
- render jest wywołaniem funkcji, czyli komponentu, przez React, który zwraca gotowy markup wykorzystując aktualne dane jakie posiada

```
import User from "../User";
import Container from "../Container";
export default function App() {
  const isActive = false
  const lastName = "Kowalski"
  return <div className="app">
    <Container className="app-container">
      <User
        age={100}
        firstName="Jan"
        lastName={lastName}
        user={{firstName: "Jan", lastName: "Kowalski"}}
        isActive
      />
    </Container>
  </div>
}
```

```
// Container.tsx
```

```
export default function Container({ className, children }) {  
  return <div className={className}>{children}</div>;  
}
```

```
// User.tsx
```

```
export default function User({firstName, lastName, user, className,  
isActive}) {  
  return <div className={className}>{firstName}</div>  
}
```

```
export default function User(props) {  
  return <div className={props.className}>{props.firstName}</div>  
}
```

ZADANIE

Props (2)

Utworzenie komponentu

- **App.tsx** - przekazuje propsy **firstName** i **lastName**
- **User.tsx** - wyświetla dane z propsów

TypeScript

- superset ze statycznym typowaniem kodu JS, który jest transpilowany do JS
- podstawowe typy:
 - string
 - number
 - boolean
 - Array<T> / T[]
 - object
 - any*
 - void
 - null i undefined
 - unknown
- składowe TS:
 - Type aliases / interface
 - Class,
 - Generic types

Składnia i typowanie

```
interface IUser {  
    firstName: string | undefined;  
    lastName: string | undefined;  
    address: Address | undefined;  
    getFullAddress: () => string;  
    permissions: Permission[];  
}  
  
type Address = {  
    street: string;  
    houseNo: number;  
    country?: Country;  
    city: string;  
}  
  
type Country = "poland" | "germany" | "united_kingdom" | "france";
```

```
export class User implements IUser {
  public firstName: string | undefined
  public lastName: string | undefined = undefined;
  public address: IAddress | undefined = undefined;
  public permissions: Permission[] = [];

  constructor() {
    this.firstName = undefined;
  }

  public getFullAddress(): string {
    if (this.address) {
      const { street, houseNo, country, city } = this.address;
      return `${street} ${houseNo} ${city} ${country}`;
    }
    return "";
  }
}
```



```
export default function User({
  firstName,
  lastName
}: {firstName: string, lastName: string}) {

  return (
    <div>
      My name is: {firstName} {lastName}
    </div>
  )
}
```

```
interface IProps {  
  firstName: string;  
  lastName: string;  
}  
  
function User({ firstName, lastName }: IProps) {  
  return (  
    <div>  
      My name is: {firstName} {lastName}  
    </div>  
  )  
}
```

```
interface IProps {  
  firstName: string;  
  lastName: string;  
}  
  
export default function User(props: IProps) {  
  return (  
    <div>  
      My name is: {props.firstName} {props.lastName}  
    </div>  
  );  
}
```

ZADANIE

TS (3)

- dodaj props 'age' do komponentu User
- dodaj typy do props w User

Interaktywność

- aby “złapać” event użytkownika trzeba przekazać funkcję jako props do elementu
- funkcja musi być **przekazana** a nie wywołana
- funkcje mogą być przekazane do komponentów jak każdy inny props
- funkcja w argumencie otrzymuje obiekt event

```
export default function Button() {  
  function handleClick() {  
    alert("You clicked me!");  
  }  
  return <>  
    <button onClick={handleClick}>Kliknij!</button>  
    <button  
      onClick={function handleClick(e) {  
        alert("You clicked me!");  
      }}>Kliknij</button>  
    <button onClick={(e) => { alert("You clicked me!"); }}>  
      Kliknij  
    </button>  
  </>  
}
```

ZADANIE

Eventy (4)

- w komponencie User dodaj `<button>` który wywoła alert z `firstName`

Zmiana wartości zmiennych jako reakcja na działanie użytkownika

```
export default function Button() {  
  let index = 1;  
  
  function handleClick() {  
    index = index + 1;  
  }  
  
  return <>  
    wartość index: {index}  
    <button onClick={handleClick}>+ 1</button>;  
  </>;  
}
```


Aktualizacja UI po wykonaniu akcji

- komponent to zwykła funkcja - renderowanie komponentu to wywołanie funkcji
- render komponentu inicjalizuje za każdym razem na nowo swoje lokalne zmienne
- zmiana wartości zmiennej lokalnej nie wywoła re-renderu komponentu

Potrzebujemy dwóch rzeczy jakie muszą się wydarzyć:

- zachowanie (zapamiętanie) nowej wartości zmiennej
- wywołanie re-renderu

State

- stan - "pamięć" komponentu - przechowuje dane w komponencie, które **nie zmieniają** się między re-renderami
- zarówno stan i propsy służą do sterowania wyglądem strony
- każdy komponent może posiadać swój wewnętrzny/lokalny stan
- zmiana stanu powoduje re-render komponentu i **wszystkich** jego dzieci (rerender \neq aktualizacja UI)

useState

- podstawowy hook do przechowywania stanu
- zwraca wartość stanu oraz funkcję (**setter function**),
- zwraca krotkę (tuple) dzięki temu mamy możliwość dowolnie nazwać stan i funkcję

```
- const [index, setIndex] = useState(0);
```

- setter - wywołanie aktualizuje stan a następnie re-renderuje komponent
- zapis pomiędzy [oraz] nazywa się destrukcją
- argument jaki przyjmuje useState to jego wartość początkowa
- stan może przechowywać każdy typ danych - od wartości prymitywnych do złożonych struktur danych
- setter podmienia cały poprzedni stan
- komponent może posiadać wiele useState
- stan jest przypisany do danego komponentu - wyrenderowanie dwóch tych samych komponentów utworzy dwa osobne niezależne zestawy stanów

Hooks

- specjalne funkcje, które można użyć tylko w komponentach lub innych hookach
- nazwa zawsze zaczyna się od 'use'
- są to dodatkowe funkcjonalności Reacta
- należy je używać tylko w 'zakresie' (scope) komponentu - nie można* w zagnieżdżeniach czyli w instrukcjach warunkowych (if), pętlach czy innych funkcjach
- ich działanie jest uzależnione od kolejności wywołania w komponencie
- można tworzyć własne hooki

Cykl życia komponentu

- Każdy komponent przechodzi przez ten sam cykl
- montowanie - komponenty jest wyświetlony na ekranie
- aktualizacja - komponent zostaje zaktualizowany kiedy otrzyma nowe propsy lub stan
- odmontowanie - komponent zostaje "usunięty z ekranu", oraz jego stan jest czyszczony

Przykład użycia **useState**:

```
const [firstName, setFirstName] = useState("John");

const [counter, setCounter] = useState(() => Math.random()*100)

const [user, setUser] = useState({ firstName: "Jan", lastName:
"Kowalski" });

const [users, setUsers] = useState([{ firstName: "John", id: 1 }]);
```

ZADANIE

useState (5)

- wartości zmiennych przekazywane do User
przenieść do useState w App - każdy osobno

useState - aktualizacja stanu

```
const [firstName, setFirstName] = useState("John");  
setFirstName("Jane");  
  
const [counter, setCounter] = useState(() => Math.random() * 100);  
setCounter(2);
```

useState w połączeniu z akcjami

```
export default function App() {  
  const [name, setName] = useState('');  
  const [counter, setCounter] = useState(() => Math.floor(  
Math.random()*100))  
  
  return <div>  
    {name} {counter}  
    <input onChange={ (e) => {setName(e.target.value)}} />  
    <button onClick={ () => {setCounter(counter + 1)}}>  
      Click me!  
    </button>  
  </div>;  
}
```


ZADANIE

useState + event (6.1)

- dodaj w komponencie App dwa inputy - dla firstname i lastName,
- onChange event ustawia odpowiednio stan

Kilka słów o aktualizacji stanu

```
export default function Counter() {  
  const [index, setIndex] = useState(0);  
  const localIndex = 0;  
  return (<>  
    {index} {localIndex}  
    <button  
      onClick={() => {  
        localIndex + 3;  
        setIndex(index + 1);  
        setIndex(index + 1);  
        setIndex(index + 1);  
      }}>+3</button>  
    </>);  
}
```

```
<button
  onClick={() => {
    setIndex(index + 1);
    setIndex(index + 1);
    setIndex(index + 1);
  }}>+3
</button>
```

```
<button
  onClick={() => {
    setIndex(0 + 1);
    setIndex(0 + 1);
    setIndex(0 + 1);
  }}>+3
</button>
```

```
export default function Counter() {  
  const [index, setIndex] = useState(0);  
  
  return (  
    <>  
      {index}  
      <button  
        onClick={() => {  
          setIndex((n) => n + 1);  
          setIndex((n) => n + 1);  
          setIndex((n) => n + 1);  
        }}>+3</button>  
    </>);  
}
```

Kilka słów o aktualizacji stanu

- zmiana stanu wywoła re-render
- przed re-renderem stan jest aktualizowany i useState otrzymuje najnowszy zestaw danych
- każdy render komponentu tworzy 'nowy' zestaw zmiennych i funkcji lokalnych
- aktualizować stan można poprzez podanie do settera nowej wartości lub tzw. **updater function**
- updater function zawsze otrzymuje najnowszą wartość stanu w argumencie
- updater function **zwraca nową wartość stanu**
- React optymalizuje ilość re-renderów - czeka aż funkcja zakończy swoje działanie a następnie wywoła re-render. Jest to tzw **batching** ([dev mode troubleshooting](#))
- batching nie ma zastosowania jeśli użytkownik kilka razy kliknie ten sam przycisk - jest to wielokrotne wywołanie tej samej funkcji
- więcej o działaniu setterów możecie poczytać [w dokumentacji](#)
- **mutowanie** stanu nie wywoła re-renderu - bez użycia settera, react nie wie że stan się zmienił

```
const [user, setUser] = useState({
  firstName: "Jan",
  lastName: "Kowalski"
});

/** Która linijka poprawnie zaktualizuje stan - zmieni firstName i
    pozostawi niezmiennione lastName */
/* 1 */ user.firstName = "Adam";

/* 2 */ setUser({firstName: "Adam"});

/* 3 */ setUser({ ...user, firstName: "Adam" });
```

Kilka zasad lepszego zarządzania stanem bez pomyłek

- grupuj powiązany stan - jeśli zawsze aktualizujesz dwa stany w tych samych akcjach, połącz je w jeden stan
- unikaj sprzeczności - dwa stany się wzajemnie wykluczają
- usuwaj niepotrzebny stan - jeśli można informacje wywnioskować na podstawie innego stanu lub propsów - nie twórz osobnego stanu
- nie duplikuj stanu
- unikaj mocno zagnieżdżonego stanu - złożone struktury mogą być ciężkie do zarządzania

```
const [index, setIndex] = useState(1);  
setIndex((prev) => prev + 2);
```

```
const [user, setUser] = useState({  
  firstName: "Jan",  
  lastName: "Kowalski"  
});  
setUser({ ...user, firstName: "Adam" });
```

```
const [users, setUsers] = useState([{ firstName: "John", id: 1 }]);  
setUsers([...users, { firstName: "Jane", id: 2 }]);
```


ZADANIE

setState + event 2 (6.3)

- z trzech useState zrób jeden, który przechowuje obiekt usera z takimi samymi danymi
- zmienić działanie eventów onChange aby odpowiednio aktualizowały stan

Renderowanie list

- kolekcje danych w React renderuje się najczęściej za pomocą metody **.map()**
- każdy element w liście musi posiadać props **key** z unikalnym identyfikatorem w obrębie swojego rodzeństwa
- key **nie** jest dostępny jako prop w komponencie dziecku
- w przypadku list, react porównuje elementy po wartości key i na tej podstawie decyduje czy wykonać aktualizację widoku - optymalizacja
- key pomaga zidentyfikować elementy które się zmieniły, zostały dodane lub usunięte
- najlepiej jako wartość key używać ID elementu (jeśli dostępne), użycie index nie jest dobrym pomysłem

```
import { useState } from "react";
import User from "../User";

export default function App() {
  const [user, setUser] = useState({
    firstName: "Jan",
    lastName: "Kowalski",
    age: 23,
  });

  const [users, setUsers] = useState([
    { firstName: "Jan", lastName: "Kowalski", age: 23, id: 1 },
    { firstName: "Anna", lastName: "Nowak", age: 33, id: 2 },
    { firstName: "Piotr", lastName: "Kowalski", age: 43, id: 3 },
  ]);
```

```
return (<>
  <input
    onChange={ (e) => {
      setUser({ ...user, firstName: e.target.value });
    }}
    name="firstName"
    placeholder="Imię"
  />
  <input
    onChange={ (e) => {
      setUser({ ...user, lastName: e.target.value });
    }}
    name="lastName"
    placeholder="Nazwisko"
  />
```

```
<User
  firstName={user.firstName}
  lastName={user.lastName}
  age={user.age}
/>

<div>Lista</div>
{users.map((user) => (
  <div key={user.id}>
    <User
      firstName={user.firstName}
      lastName={user.lastName}
      age={user.age}
    />
  </div>
))}</>);}
```

ZADANIE

renderowanie list (7)

- do komponentu App dodaj stan z listą użytkowników, uzupełnij listę przykładowymi danymi
- każdy użytkownik musi posiadać id
- wyrenderuj listę użytkowników w komponencie App

Renderowanie warunkowe

- służy do renderowania części komponentu w zależności od stanu aplikacji
- składnia taka sama jak warunki w JS
- możemy nie renderować "niepełnego" komponentu jeśli brakuje danych - zwrócić **null** lub **undefined**

```
export default function User({
  firstName,
  lastName,
  age,
}: IProps): React.ReactElement {
  if (!firstName && !lastName) {
    return <>Brak danych</>;
  }

  return (
    <>
      {firstName ? firstName : "Dodaj imię"}
      {lastName && <strong>{lastName}</strong>},
      wiek {age} {age && age > 18 && <i>pełnoletni</i>}
    </>
  );
}
```


ZADANIE

renderowanie warunkowe (8)

- jeśli firstName lub lastName są puste, User nic nie renderuje
- do App dodaj input do zmiany pola 'age'

Formularze

- podstawowy element umożliwiający interakcję z użytkownikiem
- w HTML elementy formularza posiadają i zarządzają swoim stanem
- w React można stworzyć formularz i pojedyncze inputki jako:
 - **Uncontrolled Components** - źródłem danych (single source of truth) jest DOM, nie zarządzamy stanem pól z poziomu React
 - **Controlled Components** - źródłem danych jest React i zarządzamy stanem poprzez przekazywanie do pól formularza odpowiednich propsów - **value** oraz pobieranie wartości poprzez np. event **onChange**

```
<input
  onChange={ (e) => {
    setUser({ ...user, firstName: e.target.value });
  }}
  name="firstName"
  placeholder="Imię"
/>
```

```
<input
  onChange={ (e) => {
    setUser({ ...user, lastName: e.target.value });
  }}
  name="lastName"
  placeholder="Nazwisko"
/>
```

```
export default function Form() {  
  const [user, setUser] = useState({ firstName: "", lastName: "" });  
  
  function handleFirstName(firstName: string) {  
    setUser((prev) => ({ ...prev, firstName}));  
  }  
  
  function handleLastName(lastName: string) {  
    setUser((prev) => ({ ...prev, lastName }));  
  }  
  
  return (  
    <form>  
      <input value={user.firstName}  
        onChange={ (e) => handleFirstName(e.target.value) } />  
      <input value={user.lastName}  
        onChange={ (e) => handleLastName(e.target.value) } />  
    </form>)  
}
```

```
export default function Form() {  
  const [user, setUser] = useState({ firstName: "", lastName: "" });  
  
  function handleChange(e: ChangeEvent<HTMLInputElement>) {  
    setUser((prev) => ({  
      ...prev,  
      [e.target.name]: e.target.value,  
    }));  
  }  
  
  return <form>  
    <input name="firstName" value={user.firstName}  
onChange={handleChange} />  
    <input name="lastName" value={user.lastName}  
onChange={handleChange} />  
    </form>; }
```

```
export default function Form() {
  const [user, setUser] = useState({ firstName: "", lastName: "" });
  function handleSubmit(e: FormEvent<HTMLFormElement>) {
    e.preventDefault();
    const firstName = e.target.elements.firstName.value;
    const lastName = e.target.elements.lastName.value;
    setUser({ firstName, lastName })
  }
  return (
    <form onSubmit={handleSubmit}>
      <input type="text" name="firstName" placeholder="Imię*" />
      <input type="text" name="lastName" placeholder="Nazwisko*" />
      <button type="submit">Dodaj użytkownika</button>
    </form>);
}

{/* Zamiast submit na button można użyć onClick jednak tracimy
możliwość 'wysłania' formularza za pomocą kliknięcia 'enter' */}
```

ZADANIE

Form (9)

- nowy komponent Form, w nim formularz z inputkami (firstName, lastName, age) i przyciskiem typu 'submit' - przenieść z App
- onSubmit tylko wywołuje metodę `e.preventDefault()`

Zarządzanie listami

- w stanie można przechowywać tablice
- mutowanie tablicy np przez `.push()` czy `.splice()` nie wywoła re-renderu
- aby poprawnie zaktualizować stan należy stworzyć nową wersję tablicy a następnie ustawić nowy stan
- aby dodać nowy element do tablicy można użyć spread operator lub `concat`

```
- setUsers([...users, newUser])  
- setUsers(users.concat(newUser))
```

- ta sama zasada dotyczy się usuwania czy modyfikacji elementów tablicy.
 - do usunięcia elementu idealnie sprawdzi się metoda `.filter()`
 - do modyfikacji elementu można wykorzystać metodę `.map()`

ZADANIE

zarządzanie listami (10)

- stworzenie w App funkcji, która dodaje nowy element do listy 'users'
- Form przez props otrzymuje funkcję z App
- Form na onSubmit wywołuje funkcję z props

UseRef

- służy do przechowywania wartości
- jest to **mutowalny** obiekt, którego wartość pobieramy z pola 'current'
- wartość z useRef **nie zmienia się** przy re-renderze
- zmiana wartości **nie** powoduje re-renderu komponentu
- może **przechowywać referencję** do obiektu DOM aby mieć do niego bezpośredni dostęp - nie korzystamy bezpośrednio z Document API (getElementBy*)

```
export default function Component() {  
  let nonRefValue = 10;  
  const refValue = useRef(10);  
  const [state, setState] = useState(true);  
  
  function changeValues() {  
    nonRefValue += 10;  
    refValue.current += 10;  
    setState(!state);  
  }  
  
  return <div>  
    Non ref: {nonRefValue}  
    Ref: {refValue.current}  
    <button onClick={changeValues}>+10</button>  
  </div>;}
```

```
export default function Form() {  
  const formRef = useRef<HTMLFormElement>(null);  
  
  function handleSubmit(e: React.FormEvent<HTMLFormElement>) {  
    e.preventDefault();  
    formRef.current?.reset(); // reset form  
    formRef.current?.firstName.focus(); // focus on firstName input  
  }  
  
  return <form onSubmit={handleSubmit} ref={formRef}>  
    <input onChange={handlechange} name="firstName" />  
    <input onChange={handlechange} name="lastName" />  
    <input onChange={handlechange} name="age" />  
    <button type="submit">Dodaj</button>  
  </form>;  
}
```

ZADANIE

useRef (11)

- zresetuj formularz po wywołaniu onSubmit w komponencie Form
- ustaw kursor (focus) na polu firstName po onSubmit
- zablokuj przycisk submit jeśli wszystkie pola nie są uzupełnione*

Dodawanie zewnętrznych paczek

Instalacje bibliotek wykonamy np. z pomocą npm. Aby zainstalować paczkę, należy w katalogu głównym projektu (tam gdzie plik **package.json**) wykonać polecenie:

```
npm install <nazwa-biblioteki>
```

lub

```
npm install --save-dev <nazwa-biblioteki>
```

np. do wykonania routingu (zarejestrowania “wirtualnych” podstron i przechodzenia między nimi) wykorzystamy bibliotekę **react-router-dom**, do REST API **axios** itp.

Routing pozwala nam przechodzić między poszczególnymi częściami aplikacji bez przeładowania strony jednocześnie zmieniając adresy URL w pasku przeglądarki. Aby zainstalować **typy TS** dla danej biblioteki najczęściej wystarczy wykonać:

```
npm install @types/<nazwa-biblioteki>
```

Stylowanie - global CSS

- zewnętrzne pliki .css,
- importowane osobno w poszczególnych komponentach lub importowany jeden globalny plik w głównym pliku aplikacji,

Polecam pisanie styli w składni scss - potrzebny jest do tego preprocesor Sass

```
npm install --save-dev sass
```

Sass - kilka podstawowych funkcjonalności:

- zagnieżdżanie instrukcji,
- zmienne,
- mixins (funkcje),
- import plików

Globalne style w Vite:

- import pliku ze stylami wstrzyknie kod css to tagu <style> podczas pracy na serwerze deweloperskim - aby lepiej współpracować z HMR
- budowanie aplikacji utworzy nowy plik .css, który zostanie podlinkowany w tagu <link>
- wszystkie style są dostępne globalnie, nieważne w którym pliku są zaimportowane


```
import "../user.scss";

export default function User({ user }: IProps) {
  const { firstName, lastName, age } = user;

  return (
    <div className="user grid grid-col-3">
      <span>{firstName ? firstName : "- -"}</span>
      <span>{lastName} && <strong>{lastName}</strong></span>
      <span>
        {age} {age && age > 18 && <i>pełnoletni</i>}
      </span>
    </div>
  );
}
```

```
// scss
.grid {
  display: grid;
  &-col-3 {
    grid-template-columns: repeat(3, 1fr);
    grid-gap: 1rem;
  }
}

// css
.grid {
  display: grid;
}

.grid-col-3 {
  grid-template-columns: repeat(3, 1fr);
  grid-gap: 1rem;
}
```

CSS modules

- plik musi mieć nazwę: [nazwa pliku].**module**.[s]css
- Importujemy moduł do zmiennej
- zapewniają nam unikalność nazw klas pomiędzy modułami
- można mieszać globalne style z CSS modules

```
import style from "../addUserForm.module.scss";

export default function AddUserForm({ addUser }: IProps) {
  return (
    <div className={style.addUser}>
      <form onSubmit={handleSubmit} ref={formRef}>
        <input
          className={style.addUser__input}
          onChange={handleChange}
          name="firstName"
        />
        <input
          className={style.addUser__input}
          onChange={handleChange}
          name="lastName"
        />
      </form>
    </div>
  );
}
```

```
<input
  onChange={handleChange}
  name="age"
  className={style.addUser__input}
/>
<button
  type="submit"
  className="btn"
  disabled={! (user.firstName && user.lastName && user.age) }
>
  Dodaj
</button>
</form>
</div>
);
}
```

```
// addUserForm.module.scss
```

```
.addUser {  
  margin: 0 auto;  
  display: flex;  
  width: 500px;  
  justify-content: center;
```

```
  &__input {  
    margin: 5px 0;  
    padding: 5px 10px;  
    display: block;  
    width: 200px;  
  }
```

```
}
```

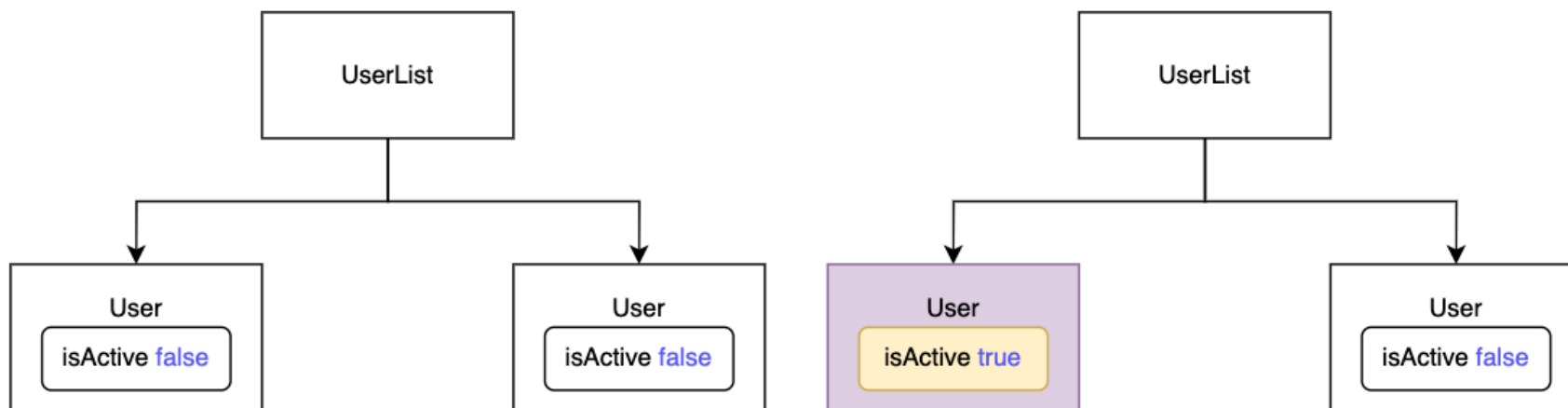
ZADANIE

Global scss + scss modules (12)

- dodanie stylu do formularza oraz listy userów

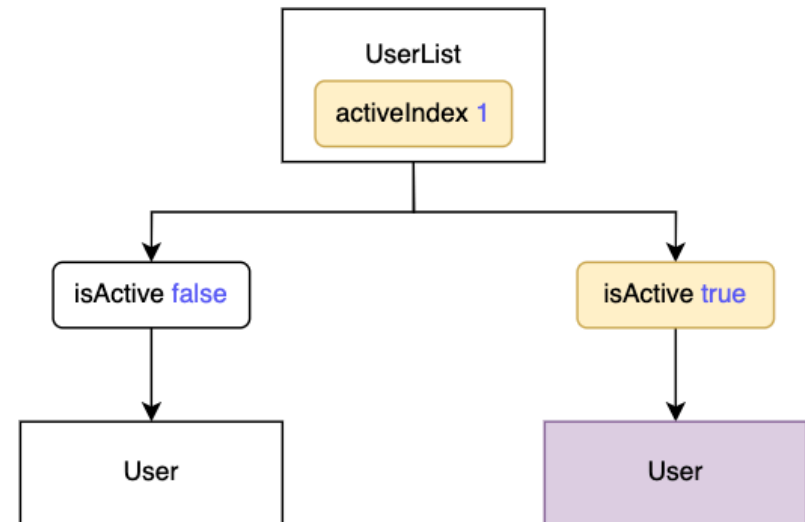
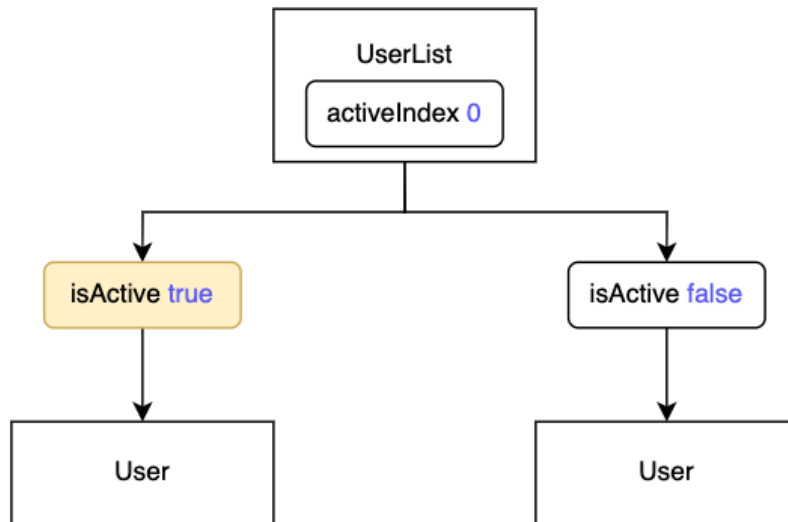
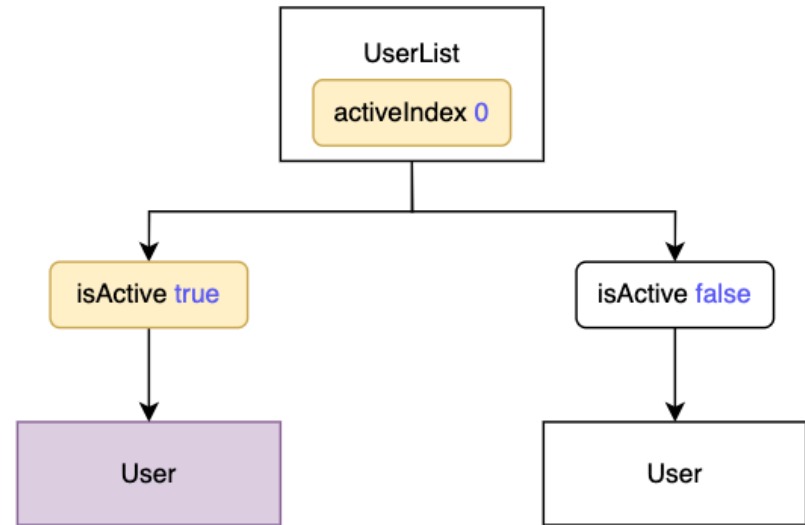
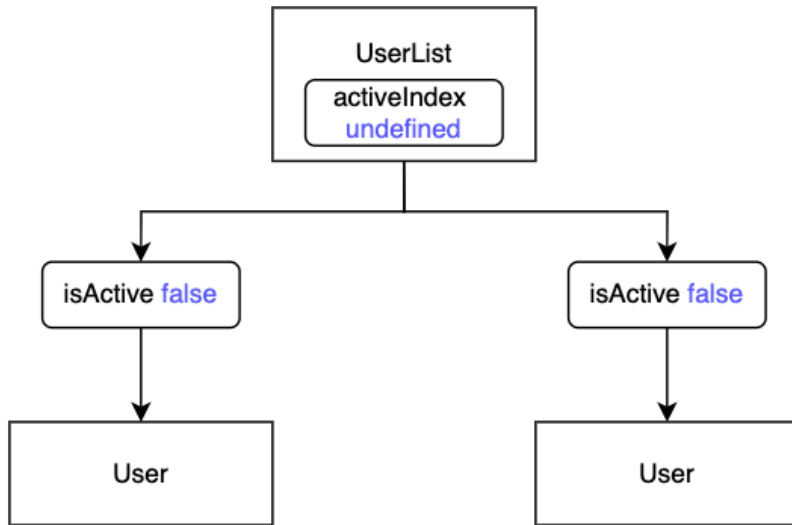
Dzielenie stanu między komponentami

- stan jest przypisany do danego komponentu - wyrenderowanie dwóch tych samych komponentów utworzy dwa osobne **niezależne** zestawy stanów



Dzielenie stanu między komponentami

- jeśli chcemy aby kilka komponentów zależało od tego samego stanu, trzeba ten stan przenieść "w górę" (*lifting state up*) drzewa do najbliższego wspólnego rodzica
- stan (i setter jeśli potrzeba) przekazać przez props do komponentów zależnych
- [react.dev - sharing state](https://react.dev/learn/sharing-state-between-components)



ZADANIE

lifting state up (13)

- zrób refactor User aby część danych pokazywała się po kliknięciu w niego
- utwórz komponent UserList który renderuje listę User
- UserList przechowuje informację o tym który User wyświetla szczegółowe dane

Edycja i usuwanie użytkownika z listy

```
function addUser(newUser: IUser) {  
    const userWithId = { ...newUser, id: id++ };  
    setUsers([...users, userWithId]);  
}  
  
function editUser(id: number, editedUser: IUser) {  
    const editedUsers = users.map((user) =>  
        user.id === id ? { ...user, ...editedUser } : user  
    );  
    setUsers(editedUsers);  
}  
  
function deleteUser(id: number) {  
    setUsers(users.filter((user) => user.id !== id));  
}
```

useReducer

- alternatywa dla useState, zwykle używany gdy przechowujemy złożone struktury danych i wykonujemy wiele operacji na stanie,
- pozwala na wyciągnięcie logiki poza komponent (funkcja **reducer**),

```
(state, action) => newState;
```
- useReducer przyjmuje funkcję reducera oraz stan początkowy,
- zwraca **stan** oraz funkcję **dispatch**, poprzez którą wywołujemy reducer, który aktualizuje stan
- reducer zwraca nowy stan
- useReducer zwraca stałą referencję do metody dispatch

```
const [state, dispatch] = useReducer(userReducer, initialState);

function addUser(newUser: IUser) {
  dispatch({ type: "ADD_USER", payload: newUser });
}

function editUser(id: number, editedUser: IUser) {
  dispatch({ type: "EDIT_USER", payload: { ...editedUser, id } });
}

function deleteUser(id: number) {
  dispatch({ type: "DELETE_USER", payload: { id } });
}

type ACTION =
  | { type: "ADD_USER"; payload: IUser }
  | { type: "EDIT_USER"; payload: IUserWithID }
  | { type: "DELETE_USER"; payload: { id: number } };
```

```
export default function userReducer(state: IUserWithID[], action:
ACTION) {
  switch (action.type) {
    case "ADD_USER":
      return [...state, { ...action.payload, id: id++ }];
    case "EDIT_USER":
      return state.map((user) =>
        user.id === action.payload.id ? action.payload : user
      );
    case "DELETE_USER":
      return state.filter((user) => user.id !== action.payload.id);
    default:
      throw new Error();
  }
}
```

useReducer vs useState

- ilość kodu - mały stan vs złożone struktury danych
- ilość akcji użytkownika
- debugowanie
- testowanie

ZADANIE

useReducer (14)

- w App podmień useState na useReducer
- dodaj usuwanie użytkownika

Custom Hook

- wydzielenie logiki do osobnych funkcji, które można użyć w wielu komponentach
- nazwa funkcji **musi** zaczynać się od **use**
- można w nich używać innych hooków, również innych customowych
- pozwalają dzielić logikę ale nie stan
- może być użyty wielokrotnie w tym samym komponencie
- są uruchamiane przy każdym renderze
- zwraca to co jest potrzebne do poprawnego działania komponentu

```
export function useCollapse(): {  
  activeIndex: number | undefined;  
  setActive: (index: number | undefined) => void;  
} {  
  const [activeIndex, setActiveIndex] = useState(undefined);  
  function setActive(index: number | undefined) {  
    if (index === activeIndex) {  
      setActiveIndex(undefined);  
    } else {  
      setActiveIndex(index);  
    }  
  }  
  return {  
    activeIndex,  
    setActive,  
  };  
}
```

```
function useInput() {  
  const [value, setValue] = useState("");  
  
  function onChange(e) {  
    setValue(e.target.value);  
  }  
  
  return { value, onChange };  
};
```

```
// Component.tsx
function Component() {
  const usernameInput = useInput();
  const passwordInput = useInput();

  return (
    <form>
      <input type="text" placeholder="username" {...usernameInput} />
      <input type="password" placeholder="password"
        {...passwordInput} />
    </form>
  );
};
```

ZADANIE

custom hook (15)

- wyciągnij logikę do pokazywania szczegółów użytkownika do custom hook

Routing

npm i react-router

- client side routing - SPA z podziałem na "podstrony"
- osobne URL,
- parametryzacja w URL
- historia przeglądania,
- zagnieżdżanie ścieżek

```
import { RouterProvider, createBrowserRouter, Link, Outlet } from
"react-router";
import Home from "../Home";
import App from "../App";

export default function Router() {
  return <RouterProvider router={router} />;
}
```



```
const router = createBrowserRouter([
  {
    path: "/",
    element: <Nav />,
    children: [
      {
        index: true,
        element: <Home />,
      },
      {
        path: "users",
        element: <App />,
      },
    ],
  },
]);
```

```
function Nav() {  
  return (  
    <>  
    <nav>  
      <ul>  
        <li>  
          <Link to="/">Home</Link>  
        </li>  
        <li>  
          <Link to="/users">Users</Link>  
        </li>  
      </ul>  
    </nav>  
    <Outlet />  
  </>  
);}
```

Parametry w url

dynamiczne elementy w url

- używamy do wyświetlania komponentu uzależnionego od jakiejś zmiennej

```
import {RouterProvider, createBrowserRouter, Link, Outlet,
useParams} from "react-router";

export default function Router() {
  return <RouterProvider router={router} />;
}
```

```
const router = createBrowserRouter([
  {
    path: "/",
    element: <Nav />,
    children: [
      {
        path: "/param-test-name/:name",
        element: <ParamTest />,
      },
      {
        path: "/param-test-id/:id",
        element: <ParamTest />,
      },
    ],
  },
]);
```

```
function Nav() {  
  return <>  
    <nav><ul>  
      <li>  
        <Link to="/param-test-name/React">Param - React</Link>  
      </li>  
      <li>  
        <Link to="/param-test-name/Vue">Param - Vue</Link>  
      </li>  
      <li>  
        <Link to="/param-test-id/123432">Param - 123432</Link>  
      </li>  
    </ul></nav>  
    <Outlet />  
  </>  
}
```

```
function ParamTest(): React.ReactElement {  
  const { name, id } = useParams();  
  return (  
    <div>  
      <div>Nazwa: {name}</div>  
      <div>Id: {id}</div>  
    </div>  
  );  
}
```

Przekierowanie bez komponentu Link

```
function ParamTest(): React.ReactElement {  
  const navigate = useNavigate();  
  const { name, id } = useParams();  
  
  return (  
    <div>  
      <div>Nazwa: {name}</div>  
      <div>Id: {id}</div>  
      <button onClick={() => navigate("/")}>Home</button>  
    </div>  
  );  
}
```

ZADANIE

Routing (16)

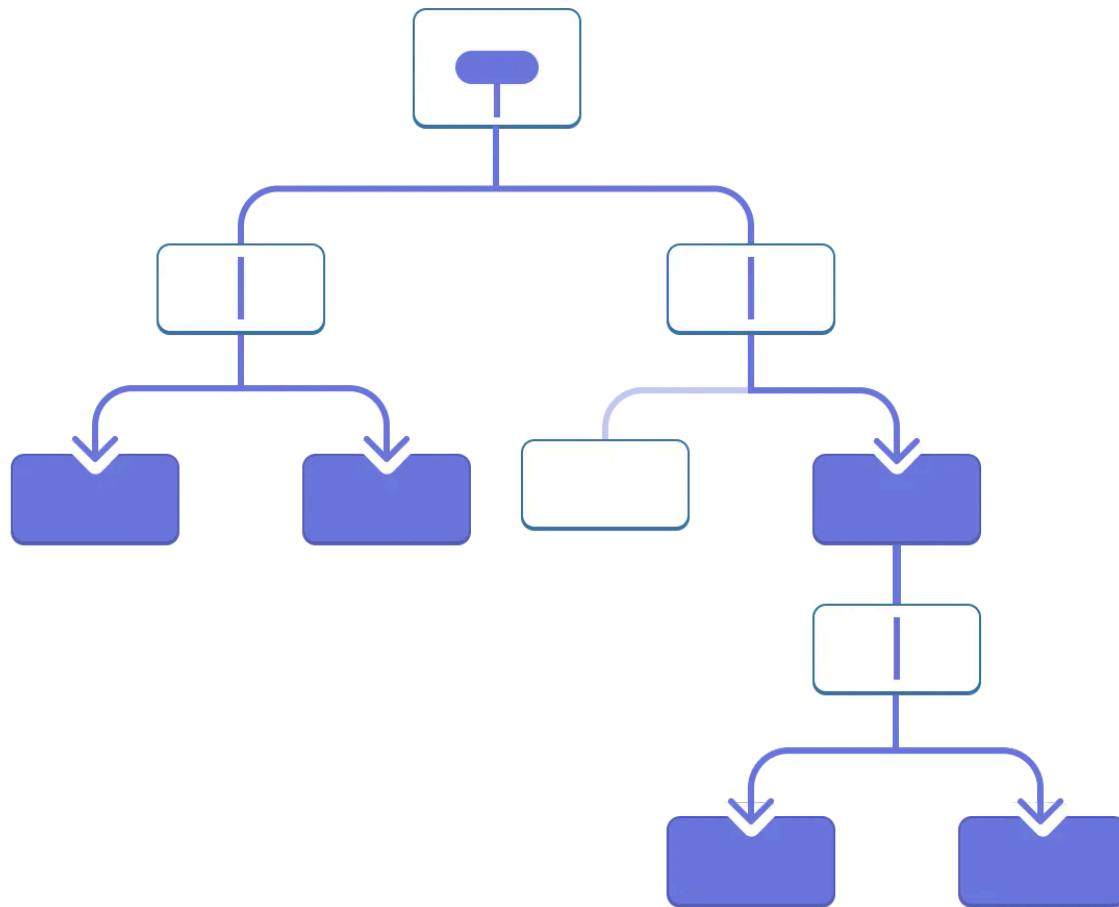
- dodanie routingu, ścieżki:
 - / - Home (nowy komponent - renderuje Hello)
 - /users - aktualny komponent App

Zarządzanie stanem

Rozwiązania pozwalające na łatwe przekazywanie stanu między komponentami na różnych poziomach zagnieżdżenia (bez przekazywania props na każdym poziomie - **prop drilling**) lub na tym samym poziomie (routing) oraz do zapewnienia aby UI posiadał zawsze aktualny stan

- **Context** - mechanizm wbudowany w React
- Redux Toolkit
- MobX
- zustand
- TanStack Query

Prop drilling



źródło: react.dev

```
import { useContext, createContext, useState } from 'react';

const UserContext = createContext(undefined);

const initialState = { firstName: "Jan", lastName: "Kowalski" };

export function UserProvider({ children }: React.ReactElement) {
  const [user, setUser] = useState(initialState);

  return (
    <UserContext.Provider value={{ user }}>
      {children}
    </UserContext.Provider>
  )
}

export const useUserContext = () => useContext(UserContext);
```

```
const router = createBrowserRouter([
  {
    {...}
    element: (
      <UserContextProvider>
        <Nav />
      </UserContextProvider>
    ),
    {...}
  ]]);
```

```
function App() {  
  /*** 1 */  
  const { user } = useContext(UserContext);  
  /*** 2 */  
  const { user } = useUserContext();  
  /*** 3 */  
  const { user } = use(UserContext);  
  return <>  
    <h2>User</h2>  
    <p>  
      Hello {user.firstName} {user.lastName}  
    </p>  
  </>  
}
```

Przez kontekst możemy przekazywać dowolną ilość i rodzaj danych (prymitywy, obiekty, funkcje) jednak dobrą praktyką jest aby jak najmniej danych było przekazywane ze względu na ryzyko dużej ilości niepotrzebnych re-renderów.

```
export function UserProvider({ children }: React.ReactElement) {
  const [user, setUser] = useState(initialState);

  const setNewUser = ({firstName, lastName}) => {
    setUser({ firstName, lastName });
  }

  return (
    <UserContext.Provider value={{ user, setNewUser }}>
      {children}
    </UserContext.Provider>
  )
}
```

```
function App() {  
  const { user, setNewUser } = useUserContext();  
  
  return <>  
    <h2>User</h2>  
    <p>  
      Hello {user.firstName} {user.lastName}  
    </p>  
    <AddNewUserForm onSubmit={setNewUser} />  
  </>  
}
```

ZADANIE

Context (17)

- stwórz `AppContext` razem z `Providerem`
- przenieś stan z `App` (`useReducer`) do `providera`
- pobierz stan i akcję z `useContext` a nie przez `props`

ZADANIE

Context + routing(18)

- dodaj nową zakładkę (route) która będzie renderować formularz dodawania użytkownika
- w Nav dodaj Link do nowego routa
- usuń z App formularz
- przekieruj do listy userów po submit

Stylowanie aplikacji - Styled Components

```
npm install --save styled-components
```

```
npm install --save @types/styled-components
```

- definiuje komponent razem ze stylami,
- brak nazw klas a więc brak błędów związanych z nadpisywaniem styli,
- łatwiejszy refactor i zarządzanie stylami - nie trzeba szukać wystąpień klas i zastanawiać się jak to wpłynie na pozostałe części aplikacji
- dynamiczne stylowanie za pomocą propsów
- automatyczne dodawanie vendor prefix
- przekazuje wszystkie propsy/atributy do komponentu/tagu
- style są dodawane do head strony

```
<nav>
  <ul className="nav">
    <li className="nav__item">
      <Link to="/add" className="nav__link">
        Dodaj
      </Link>
    </li>
    <li className="nav__item">
      <Link to="/users" className="nav__link">
        Użytkownicy
      </Link>
    </li>
  </ul>
</nav>;
```

```
import styled from "styled-components";
export const Navbar = styled.nav`
  width: 100%;
  display: block;
`;
export const Nav = styled.ul`
  padding: 0;
  display: flex;
  justify-content: flex-start;
  align-items: center;
  list-style-type: none;
`;
export const NavItem = styled.li<{ home?: boolean }>`
  margin: 0 10px;
  font-weight: ${ (props) => (props.home ? 700 : 400) };
`;
```

```
import { Link } from "react-router-dom";
import styled from "styled-components";

export const NavLink = styled(Link) `
  padding: 5px 10px;
  text-decoration: none;
  color: #222;

  &:hover {
    color: #777;
  }
`;
```

```
<Navbar>
  <Nav>
    <NavItem><NavLink to="/about">About</NavLink></NavItem>
    <NavItem home><NavLink to="/users">Users</NavLink></NavItem>
  </Nav>
</Navbar>
```

```
<nav class="sc-AxjAm eHKixY">
  <ul class="sc-AxirZ fdKNwi">
    <li class="sc-AxiKw LWpGx"><a class="sc-AxhCb kJtxMV"
href="/about">About</a></li>
    <li class="sc-AxiKw LWpGx"><a class="sc-AxhCb kJtxMV"
href="/users">Users</a></li>
  </ul>
</nav>
```

ZADANIE

Styled components

- **ostylowanie widoków w styled-components**

Komunikacja HTTP

Komunikacja z API poprzez HTTP jest operacją asynchroniczną. Aby móc działać na danych otrzymanych z serwera, musimy poczekać na jego odpowiedź. Można wykorzystać:

- Callback function,
- Promise,
- Async/await - składnia podobna do synchronicznego kodu

Callback, Promise

- Callback to funkcja przekazywana jako argument do innej (najczęściej) asynchronicznej funkcji,
- Promise to obiekt będący obietnicą otrzymania wyniku w przyszłości po wykonaniu asynchronicznej akcji:

```
const promise1 = new Promise((resolve, reject) => {  
  resolve('foo');  
});  
  
promise1.then((value) => {  
  console.log(value); // foo  
}).catch((error) => {  
  console.error(error);  
});  
  
console.log(promise1); // [object Promise]
```

Async/await

- async function - funkcja asynchroniczna
- swoją składnią przypominają bardziej kod pisany synchronicznie
- często używany z blokiem **try .. catch** aby wyłapywać błędy

```
function resolveAfter2Seconds() {  
  return new Promise(resolve => {  
    setTimeout(() => { resolve('resolved'); }, 2000);  
  });  
}  
  
async function asyncCall() {  
  try {  
    const result = await resolveAfter2Seconds();  
    console.log(result); // 'resolved'  
  } catch (error) {}  
}  
  
asyncCall();
```

Metody HTTP

- GET - służy do pobierania danych,
- POST - służy do tworzenia danych,
- PUT - służy do modyfikowania danych,
- DELETE - służą do usuwania danych

REST

Rest jest koncepcją według której połączenie metody HTTP z adresem url daje wiedzę o tym jaką akcję chce wykonać strona wykonująca zapytanie:

- | | |
|--------------------|---------------------------------------|
| - GET /users | - pobierz listę użytkowników |
| - GET /users/4 | - pobierz użytkownika o id 4 |
| - POST /users | - utwórz nowego użytkownika |
| - PUT /users/20 | - zmodyfikuj dane użytkownika o id 20 |
| - DELETE /users/19 | - usuń użytkownika o id 19 |

JSON SERVER

- pozwala uruchomić prosty serwer deweloperski, który będzie serwował kolekcje i zasoby określone w pliku json.
- zarówno edycje pliku mają wpływ na odpowiedzi serwera, jak i zapytania do serwera modyfikują plik (ich stany są zsynchronizowane).
- dostarcza w pełni funkcjonalne REST API (możemy wykonywać operacje na "sztucznej" (mock) bazie danych (plik .json))

Instalacja

```
npm install --save-dev json-server
```

uruchomienie

```
npx json-server db.json
```

```
// db.json
```

```
{  
  "users": [  
    {  
      "id": 1,  
      "firstName": "John",  
      "lastName": "Doe"  
    },  
    {  
      "id": 2,  
      "firstName": "Jean",  
      "lastName": "Dean"  
    }  
  ]  
}
```

<http://localhost:3000/users> - sprawdźcie wynik w przeglądarce

ZADANIE

JSON server (19)

- utworzenie mockowej bazy z json-server
- wypełnienie db.json przykładowymi danymi (lista userów)

FETCH / AXIOS / KY?

- **Fetch** - następca XMLHttpRequest bazująca na obietnicach,
- **Axios** i **Ky** to biblioteki ułatwiające pracę:
 - prostsze api,
 - skróty metod (.post(), .get()),
 - (lepsz?) obsługa błędów

```
npm install axios
```

Fetch i Axios

```
fetch(url, options)
```

```
axios(url, options)
```

GET:

```
fetch('http://localhost:3001/users')  
  .then((response: Response) => {  
    return response.json();  
  }).then(body => console.log(body);  
  ).catch(err => console.error(err))
```

```
axios('http://localhost:3001/users')  
  .then((response) => {  
    console.log(response.data);  
  }).catch(err => console.error(err))
```


POST:

```
const data = { firstName: "John", lastName: "Doe",  
email: "test3@cmu.edu", age: "28"}  
  
fetch('http://localhost:3001/users', {  
  method: 'POST',  
  body: JSON.stringify(data),  
  headers: {  
    'Content-Type': 'application/json'  
  })  
}).then((res) => {  
  return res.json();  
}).then((respo) => {  
  console.log(respo);  
})
```

```
axios.post('http://localhost:3001/users', data)
  .then((response) => {
    console.log(response.data);
  }).catch(err => console.error(err))
```

Options:

Fetch:

```
method: 'POST', // *GET, POST, PUT, DELETE, etc.
mode: 'cors', // no-cors, *cors, same-origin
cache: 'no-cache', // *default, no-cache, reload, force-cache,
only-if-cached
credentials: 'same-origin', // include, *same-origin, omit
headers: { 'Content-Type': 'application/json' // 'Content-Type':
'application/x-www-form-urlencoded', },
redirect: 'follow', // manual, *follow, error
referrerPolicy: 'no-referrer', // no-referrer,
*no-referrer-when-downgrade, origin, origin-when-cross-origin,
same-origin, strict-origin, strict-origin-when-cross-origin, unsafe-url
body: JSON.stringify(data) // body data type must match "Content-Type"
header
```

Axios:

https://axios-http.com/docs/req_config

```
// api.ts  
  
import axios from "axios";  
  
export default axios.create({  
  baseURL: "http://localhost:3000",  
});
```

```
const body = { firstName: "John", lastName: "Smith" }  
  
try {  
  const response = await api.post("/users", body)  
  console.log(response.data);  
} catch (error) {  
  console.error(error);  
}  
  
api.get('/users', {params: {id: 1}})
```

ZADANIE

Axios (20)

- zainstaluj axios
- (poza komponentem) - pobierz użytkowników, wyloguj wynik w konsoli
(<http://localhost:3000/users>)

UseEffect

Zgodnie z dokumentacją:

Hook efektów pozwala na przeprowadzanie efektów ubocznych w komponentach spowodowanych samym renderowaniem komponentu

- to znaczy, że odpowiedzialnością **useEffect** jest wykonanie operacji, które **nie są efektem działania użytkownika**
- jest wywoływany **po każdym renderze (*)**
- może zwracać funkcję, która zostanie wywołana przed odmontowaniem komponentu / przed kolejnym wywołaniem tego samego useEffect
- jeden komponent może mieć wiele useEffect

```
useEffect(() => {  
  ///  
})
```

```
useEffect(() => {  
  api.get('/users').then((response) => {  
    setUsers(response.data);  
  }).catch(err => console.error(err))  
  
  return () => {  
    //wywołaj przed odmontowaniem  
  };  
})
```

*Optymalizacja wywołań useEffect

Aby nie wywoływać useEffect przy każdym renderze, można wykorzystać opcjonalny drugi argument - tablicę zależności:

wywołanie tylko raz po pierwszym renderze

```
useEffect(() => {  
  api.get('/users').then((response) => {  
    setUsers(response.data)  
  }).catch(err => console.error(err));  
}, [])
```

wywołanie po zmianie props.userId

```
useEffect(() => {  
  api.get(`/users/${props.userId}`).then((response) => {  
    setUser(response.data)  
  }).catch(err => console.error(err));  
}, [props.userId])
```


useEffect

- useEffect **nie** zostanie wywołany jeśli wszystkie elementy podane w tablicy zależności (**dependencies**) są takie same jak w poprzednim renderze
- pusta tablica zależności spowoduje wywołanie efektu raz po “zamontowaniu” komponentu - po pierwszym renderze
- jeśli useEffekt korzysta ze zmiennych, wszystkie takie zmienne dla poprawnego działania muszą być dodane do tablicy zależności

Kiedy NIE używać useEffect

- do transformacji, kalkulacji danych, które są potrzebne do renderowania np. filtrowanie listy
- do wywoływania funkcji, które są wynikiem działania użytkownika

ZADANIE

useEffect (21)

- pobierz użytkowników z json server a następnie dodaj ich do stanu
- submit form dodaje usera do bazy,
- nowy route, który wyświetla profil
- kliknięcie w użytkownika na liście otwiera /users/:id

useCallback, useMemo

- służą do optymalizacji wydajności (**false positive**)
- pierwszy zwraca 'zapamiętaną' funkcję, drugi 'zapamiętaną wartość'
- każda optymalizacja niesie za sobą koszt jaki trzeba ponieść
- **useCallback** powinniśmy używać gdy chcemy mieć tę samą referencję do funkcji po każdym renderze - optymalizacja ilości re-renderów
- **useMemo** gdy musimy wykonać kosztowne obliczenia

Podsumowanie

- aplikacja złożona z trzech widoków: listy użytkowników (Users), widoku danych użytkownika (SingleUser), formularza dodawania użytkownika (Form/AddUser),
- **UserContext**
 - przechowuje użytkowników
- **Users**
 - pobiera listę użytkowników
 - wyświetla listę
 - daje możliwość usuwania użytkownika,
 - Wybór (kliknięcie) użytkownika przenosi nas widok szczegółowych danych użytkownika (ustawia odpowiedni url),
- **User** pobiera dane z (dwa warianty):
 - pobiera użytkownika z kontekst (czy zawsze zadziała?)
 - pobiera użytkownika z API

a potem wyświetla pełną informację o użytkowniku. **Z url pobiera id użytkownika**
- **AddUser/Form**
 - dodaje nowego użytkownika do bazy i przekierowuje do **UserList**

Pytania?

Dziękuję za uwagę

Przydatne linki:

- <https://reactjs.org/docs/glossary.html>