REACT

Biblioteka, framework, technologia

Zajmuje się wyświetlaniem interface, zarzadza stanem

Kompomnent, to jest co widzimy na stronie, osobne to sa np. menu, header, przyciski. Sa od siebie niezalezne

Kompozycja komponentu jest kluczowa

Każdy eklement wyświetlony to jest obiekt, DOM to jest lacznik.

Virtual Dom, to jest reprezentcja interfejcu w pamięci synchroznizowana z prawdzimym domem.

Do tej pory uzywalismy kodu imperatywnego (mowi krok pom kroku komputerowi co robic) instrukcje

Kod deklaratywny, definiujemy koczowy wygld interface i jk się ma zachowywać. Reszta jest robiona pod mska reacta.

FRAMEWORK, struktura aplikacji, dostarcza gotowe rozwiązania, definiuje strukturę aplikacji do podążania. IOS odwrócona kontrola

Przykłady

Angular.io (wspolczesny)

Angularjs.org

Embers.com

Meteor.com

Biblioteki: reactjs, org vuejs.org polimer-project.org backbonejs.org

SPA silgle page aplication, nie potrzebuje przeladowania, ma tylko jeden plik html, działa w obrebie jednej stronie działa w przeglądarce

TWORZENNIE APLIKACJI W REACK

Nowy folder otwieramy w VSC i terminal node -v, npm -v, npx create-react-app clicker --template typescript (clicker to nazwa najszej aplikacji)

O

Kod piszemy w src

W widoku VSC mamy mieć otwarty tylko folder naszej aplikacji, przez open foilder

W filderze public, html możemy dodac czcionke, ytul, ale body nie ruszamy, usuwamy header z app.tsc

I 2 i 3 import,

<https://github.com/ziarenn/ZDFRONpol17-Clicker>

tworzymy pod src nowe foldery i pliki

webpack mieli pliki i wysyla na serwer, skleja w calosc

mamy komponent:

const Display = () => {

return <div>

    <p>123 to jest display</p>

</div>

}

export default Display

importujemy do app.tsx

import Display from "./components/Display/Displey";

function App() {

  return (

    <div className="App">

      <h1>ZDFRONpol17 clicker/react</h1>

      <span>123</span>

      <button> click </button>

      <Display />

    </div>

  );

Rafc mozmey zaicjalizowac nowy komponent

W {} wstawiamy waroci JS np. <p>{x}</p>

Żeby cos wsadzis do innego, a apptscx:

const response = {

    firstName: "john",

    lastName: "smith",

    age: 30,

<Display dataToPassDown={response}/>

A w display wpisujemy

DESTRUKTURYZACJA

Obiekt const person ={

firstName:”john”,

lastName:”N”,

age: 30,

married: true,

};

Const{ firstName, lastName, age} = person;

Console.log(firstName, lastName, age);

// PRZEKAZYWANIE DANYCH Z KOMPONENTU RODZICA DO KOMPONENTU DZIECKA - PROPS // 1. Wpisanie propsa w tag komponentu (wyświetlnie). // 2. Stworzenie odpowiedniego interfejsu w komponencie który odbiera dane. Pamiętaj że nawet jeżeli przekazujesz tylko 1 prop, jest on obwijany obiektem. // 3. Odebranie propsów przez parametr w komponencie dziecku. Od razu destrukturyzuj parametr. // 4. Korzystaj z danych.// PRZEKAZYWANIE DANYCH Z KOMPONENTU RODZICA DO KOMPONENTU DZIECKA - PROPS // 1. Wpisanie propsa w tag komponentu (wyświetlnie). // 2. Stworzenie odpowiedniego interfejsu w komponencie który odbiera dane. Pamiętaj że nawet jeżeli przekazujesz tylko 1 prop, jest on obwijany obiektem. // 3. Odebranie propsów przez parametr w komponencie dziecku. Od razu destrukturyzuj parametr. // 4. Korzystaj z danych.

// 1. Przekaż obiekt response do komponentu Form.tsx propsem. Nie zapomnij o odebraniu propsów, destrukturyzacji i intefejsie // 2. Wyświetl dane w tabeli (po lewej w tabeli imie nazwisko wiek, po prawej dane)

W Table:

|  |
| --- |
| import React from "react"; |
|  |  |
|  | interface Person { |
|  | firstName: string; |
|  | lastName: string; |
|  | age: number; |
|  | } |
|  |  |
|  | interface TableProps { |
|  | response: Person; |
|  | } |
|  |  |
|  | const Table = ({ response }: TableProps) => { |
|  | return ( |
|  | <table> |
|  | <thead> |
|  | <tr> |
|  | <th>Imie</th> |
|  | <th>{response.firstName}</th> |
|  | </tr> |
|  | </thead> |
|  | <tbody> |
|  | <tr> |
|  | <th>Nazwisko</th> |
|  | <th>{response.lastName}</th> |
|  | </tr> |
|  | </tbody> |
|  | <tfoot> |
|  | <tr> |
|  | <th>Wiek</th> |
|  | <th>{response.age}</th> |
|  | </tr> |
|  | </tfoot> |
|  | </table> |
|  | ); |
|  | }; |
|  |  |
|  | export default Table; |
|  | // imie | xyz |
|  | // nazwisko | xyz |
|  | // wiek | xyz |

A w App zmieniamy tylko jena linie:

<Table response={response} />

STANY

Stan komponentu jest informacja co się dzieje z komponentem, ozna to zapisac jako rozne typy danych np. strig, liczba. Np. najechany, kliknięty, odjechany, odklikniety.

Hook

To funkcja e react lub od osob trzecich, ma dostep do wewnetrnych mechanizmow reacta

Zasady

1.

Mogą być używane w komponentach funkcyjnych (funktion)

2. musza być wywoływane

Tylko na top scopie komponentach funkcyjnych

TOP SCOPE to jest gora opisu komponentu, musza być zawsze w pierwszej linii komponentu

3. nazwy hookow zaczynają się od „use” np. useSytate, useRef, UseEffect, useForm.

useState sluzy do przechowywania wartości, których znmiana ma spowodować przeladowanie się komponentu.

Stan pusty => request => response do stanu => przeladowanie komponentu => wyświetlanie responsu.

useState zwraca liste z 2 elementai: zmienna stanowa (stan) i funkcje aktualizacyjna zinna stanowa (stan).

Zmienna stanowa moze byc dowolna, w nazwewnictwie funkcji aktualizacyjnej zmienna stanowa jest konwencja set + nazwa zmiennej stanowej

const [count, setCount] = useState(0)

nie robimy: coint = 5, count++

console.log(“component sie przeladowal”);

cost increaseCount = () => {

setCount(count + 1);

console.log(count);

jest dedykowana funkcja

setCount(5)

// 1. Stwórz komponent DisplayRandom. // 2. W komponencie DisplayRandom stwórz stan randomNumber, wartość początkowa: 0 // 3. Stwórz funkcję updateRandomNumber, w tej funkcji przy użyciu innej odpowiedniej funkcji aktulizuj stan randomNumber na losową liczbę. // 4. Stan randomNumber wyświetlaj w paragrafie w komponencie DisplayRandom. // 5. Stwórz przycisk z textem Update, na ten przycisk nadaj onClick, do onClicka podaj funkcję updateRandomNumber. //

import { useState } from "react";

const DisplayRandom = () => {

  const [randomNumber, setRandomNumber] = useState(0);

  const updateRandomNumber = () => {

    const random = Math.random();

    setRandomNumber(random);

  };

  return (

    <>

      <p>{randomNumber}</p>

      <button onClick={updateRandomNumber}>Update</button>

    </>

  );

};

export default DisplayRandom;

6. Sam komponent DisplayRandom wyświetl w App.tsx

import DisplayRandom from "./DisplayRandom/DisplayRandom";

<DisplayRandom />

Teraz przenocimy to do app tsx

Funkcja akualizacyjna, przyjmuje strong, zwraca nic:

(x: string) => void

// 1. Zdefiniowanie stanu // app

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

  const [randomNumber, (stan) setRandomNumber(funkcja aktualizacyjna stan)] = useState(0)

2. Przekazanie funkcji aktulizującej stan do komponentu niżej

  <p>{randomNumber}</p> 9wyswietlamy stan

// 3. Wywołanie tam tej funkcji

<DisplayRandom setRandomNuber={setRandomNumber}/>

// 4. Zmienna stanowa w komponencie wyższym się aktulizuje // 5. Przeładowanie komponentu // 6. Wyświetlenie nowej wartości

import { useState } from "react";

interface DisplayRandomProps {

  setRandomNumber: (value: number) => void  //nazwa value nie ma znaczenia, piszemy potem typ, bo chcemy liczbe

}

const DisplayRandom = ({setRandomNumber} :  DisplayRandomProps) => {

  const updateRandomNumber = () => {

    const random = Math.random();

    setRandomNumber(random); //wywoluje funkcje

}

  return (

    <>

      <button onClick={updateRandomNumber}>Update</button>

    </>

  );

};

export default DisplayRandom;

W app piszemy <DisplayRandom setRandomNumber={setRandomNumber} />

<p>{randomNumber}</p>

Random

|  |
| --- |
| import { useState } from "react"; |
|  |  |
|  | // interface x { |
|  | // setName: (value: string) => void |
|  | // } |
|  |  |
|  | interface DisplayRandomProps { |
|  | setRandomNumber: (value: number) => void; |
|  | } |
|  |  |
|  | const DisplayRandom = ({ setRandomNumber }: DisplayRandomProps) => { |
|  | const updateRandomNumber = () => { |
|  | const random = Math.random(); |
|  | setRandomNumber(random); |
|  | }; |
|  |  |
|  | return ( |
|  | <> |
|  | <button onClick={updateRandomNumber}>Update</button> |
|  | </> |
|  | ); |
|  | }; |
|  |  |
|  | export default DisplayRandom; |

Wazne, na gorze w obu zakładkach:

import React, { useState } from "react";

w app

// 1. Przekazanie stanu count i funkcji aktulizującej stan count do Manipulator.tsx // 2. W Manipulatorze stwórz 2 funkcje: incrementCount i decrementCount. W incrementCount wywołuj setCount w taki sposób żeby stan count zwiększał się o jeden (NIE COUNT++), korzystaj ze stanu count który również jest dostępny w tym komponencie. W decrementCount rób odwrotność, zmniejszaj stan count o 1. // 3. W Manipulatorze na przycisk "+" nałóż onClick, do onClicka podaj incrementCount, do onClicka na przycisku "-" podaj decrementCount. // 4. Z App.tsx do CountDisplay przekaż stan count. Wyświetlaj go w spanie w którym teraz mamy wyświetlone 0.

W app

|  |
| --- |
| const [count, setCount] = useState(0); |
|  | const [randomNumber, setRandomNumber] = useState(0); |

|  |
| --- |
| return ( |
|  | <div className="App"> |
|  | <Heading /> |
|  | <CountDisplay count={count} /> |
|  | <Manipulator count={count} setCount={setCount} /> |
|  | <DisplayRandom setRandomNumber={setRandomNumber} /> |
|  | <p>{randomNumber}</p> |
|  | </div> |

MANIPULATOR

|  |
| --- |
| import React from "react"; |
|  |  |
|  | interface ManipulatorProps { |
|  | count: number; |
|  | setCount: (value: number) => void; |
|  | } |
|  |  |
|  | const Manipulator = ({ count, setCount }: ManipulatorProps) => { |
|  | const incrementCount = () => { |
|  | setCount(count + 1); |
|  | }; |
|  |  |
|  | const decrementCount = () => { |
|  | setCount(count - 1); |
|  | }; |
|  |  |
|  | return ( |
|  | <div> |
|  | <h5>Zwiększ lub zmniejsz</h5> |
|  | <button onClick={incrementCount}>+</button> |
|  | <button onClick={decrementCount}>-</button> |
|  | </div> |
|  | ); |
|  | }; |
|  |  |
|  | export default Manipulator; |

COUNT DISPLAY

|  |
| --- |
| import React from "react"; |
|  |  |
|  | interface CountDisplayProps { |
|  | count: number; |
|  | } |
|  |  |
|  | const CountDisplay = ({ count }: CountDisplayProps) => { |
|  | return ( |
|  | <div> |
|  | <span>{count}</span> |
|  | </div> |
|  | ); |
|  | }; |
|  |  |
|  | export default CountDisplay; |

APLIKACJA NEWS

Sciagamy

npx create-react-app sdanews --template typescript

// 1. Koncepcja komponentu // 2. Props // 3. Interface do propsów // 4. useState // 5. State lifting

<https://mui.com/>

<https://github.com/ziarenn/ZDFRONpol17-SDANEWS>

git init

git clone x

przed npm install cd do repo

npm install

npm start