A REACT egy kiegészítő könyvtár a HTML/CSS/Js-hez a Facebook-tól. Célja, hogy felhasználói interfészt lehessen vele építeni. Előnye, hogy a böngészőben töltődik be, és nem kell a szerver válaszát várni. Ötvözi a HTML-t és JS-t.

Komponensekből épül fel minden weboldal (pl header, sidebar, headline, article). Ezeket a REACT külön kezeli kódblokkokként. Személyre szabott HTML elemekként lehet felfogni.

Szintaxis kiegészítését hívják JSX (JavaScript XML )-nek. Ahhoz hogy használható legyen a böngészőnek, le kell fordítani complier segítségével. A legnépszerűbb complier a Babel nevű.

REACT a kódba írhatunk közvetlenül JS parancsot kapcsos zárójelben így:

{let javaScriptValtozo = 0;}

A kommentelés is hasonlóan történik:

{/\* Ez ki van kommentelve \*/}

:::::::::::::::::::::::::::

DOM manipuláció REACT-tal

:::::::::::::::::::::::::::

Ami a DOM manipulációhoz kell, az ez a parancs:

ReactDOM.render(JSX, document.getElementById('root'))

Div hozzáadása a JSX-el:

Nem kell .createElement(), sem .append(), elég ennyi:

Létrehozzuk:

const JSX = <div></div>;

Ha több soros HTML-t akarok változóba tenni, akkor:

-zárójelbe kell tennem

-egy nyitó és egy záró TAG között kell lennie a HTML kódnak, de akármennyi nested element lehet benne

Különben nem fordítódik le.

Majd beillesztjük:

A ReactDOM a REACT objektuma, ami a DOM manipulációs parancsokat tartalmazza. Elemet így lehet hozzáadni:

ReactDOM.render(valtozobaTettJSXElem, targetNode);

JSX kivételek a HTML kódolásban:

#1. Fontos, hogy mivel JavaScriptet is lehet használni a Reacttal, ezért a félreértés elkerülése érdekében a JavaScriptes parancsszavak ki vannak tiltva a JSX-es HTML kódból.

Ez legtöbbször egyszerűen camelCase-es átírást jelent, mint pl.: a HTML onclick=""-je

onClick="" lesz a JSX-ben. Ahol nem lehetett így megoldani, ott megtoldották a HTML argumentumokat. Pl.:

Osztály elnevezése JSX-ben:

className="osztaly-nev";

A form label-jéhez a <label for=""> itt <label htmlFor="">

#2. Nincsenek self closing TAG-ek a JSX-ben

pl.: <br> -t így kell írni: <br />

#3. Egy TAG-ben lehet létrehozni zárandó TAG-et is így:

<div></div> -et lehet így: <div /> - ekkor nem lehet nestelni bele igaz, de...

/\*

A legrégebbi React-okban még kellett speciális szintaxis, nem lehetett html-t berakni. Akkor kellett a JSX-et tartalmazó file-ba az hogy:

import React from 'react'; - ebből az objektumból hívták le a funkciókat.

React.createElement('div',{id:'tartalmazoDivID'},

React.createElement('h2',{id:'cimsorID'},'String ami a szövege a h2-nek!')

React.createElement(JSXfunkcio, {prop1:'értek'}) )

Ennek HTML eredménye lesz a fordítás után:

<div id="tartalmazoDivID"><h2 id="cimsorID">String ami a szövege a h2-nek!<JSXfunkcio prop1="érték"></JSXfunkcio></h2></div>

Tehát a React.createElement('htmlElem',{attribútum1:'értéke',attribútum2:'értéke'},'szövegtartalom/beágyazott elemek'); - ha nincs property, akkor üres objektumot kell beadni {}, ha pedig nincs szöveg, az simán kihagyható. Ha beágyazunk másik html elemet ebbe, akkor a szöveg helyére kell írni.

Ma is erre a formátumra alakítja át a JSX-et, majd erről a formátumról a böngészős zagyvaságra, csak ez a háttérben történik már.

Ma már nem kell kiírni az importot, ha nincs React.objektum explicite meghívva, ugyanakkor sokan még megteszik, hogy látszódjon, hogy ez egy React file az olvasónak egyből.

\*/

:::::::::::::::::::::::::::::::

Kompenensek a REACT-ban

:::::::::::::::::::::::::::::::

Minden komponensekből áll össze a Reactban. A komponensek személyre szabott HTML elemek (keverve CSS-el és JS-el). Érdemes minden komponenst külön file-ben tárolni, így könnyen megtalaláható, áttekinthető és egyszerűen módosítható lesz.

A file-oknak van egy hierarchiája azért, hogy össze legyenek kötve az index.html-el legvégül.

index.js

========

Ez szedi össze a fő .Css és ReactDOM .js fájlok legmagasabb szintű gyüjtő

fájljait. Meghatározza még a root helyét, ahová renderelve lesz minden is, és megtörténik a renderelés maga a root.render(<App />); - paranccsal.

App.js

========

Ez szedi csokorba a .js fájlokat, ez a React Component tree-jének törzse, innen indul ki minden más komponens.

JSX komponens készítő funkciók elnevezése:

Külön file-ba illik tenni, és az elnevezési konvenciók a következők:

Azonos nevű .js file-ba tesszük mint a tartalmazott funkció, így nagybetűvel kell kezdeni, és camelCase-re kell formázni.

A fileban elkészítjük a JSX funkciót, majd az App.js-be importáljuk, végül beillesztjük az épülő kódba a funkció nevével jelezve a JSX tartalmat így:

function App() {return <JSXFunkcio/>;} - fontos, hogy < />-be tegyük a JSX-et visszaadó funkciót, hogy renderelje.

JSX -funkciót tartalmazó file felépítése:

- - - - - - - - - - - - - - - - - - - - -

Régen minden egyes JSX-file elejére be kellett írni, hogy:

import React from 'react'; - ma már ez meghaladott, az újabb verziók nem kérik.

Létrehozhatók ÍGY:

A) stateless functional component:

const MyFunction = function (){

Szokásos JavaScript funkciót írok ide, ami vagy JSX-et, vagy null-t ad vissza.

}

REACT Funkció Mindig Nagybetűvel Kezdődik!

B)

function MyJSX(){

return (<div><h1>Valami szöveg</h1></div>)

}

Lehet úgy beadni a ReactDOM.render()-be a megnevezett JSX funkciót, mintha self closing HTML lenne, azaz < /> jelek között így:

ReactDOM.render(<MyJSX />,document.getElementById('node\_ahova\_tegye'));

Komponensek személyre szabása:

Ahhoz, hogy a komponenesek dinamikusan módosíthatók legyenek egyes részeiken, megadhatók a konstruáló funkciók argumentumaként a 'props'-ok. A props-ok mindig objektumokként működnek.

function MyJSX(props){

return (<div><h1>{props.headline}</h1><p>{props.paragraph}</p></div>)

} - itt a props helyett bármi lehet természetesen. A Props egy objektum lesz, aminek lehetnek alesetei (itt: headline és paragraph). A props Js kódolással van, ezért {}-el kell kilépni a HTML kódból.

És így lehet renderelni:

ReactDOM.render(<MyJSX headline="Szenzacio!" paragraph="Lorem ipsum" />,document.getElementById('node\_ahova\_tegye'));

Érdekes módon itt a props objektum cimke:tartalom párját a HTML szintaxissa kell beilleszteni cimke="tartalom" formában.

Ez a hardcoded változat, de általában változóban tárolt adatot adunk be a mégnagyobb rugalmasság kedvéért így:

<App cimke1={props.cimke1} cimke2 ={props.cimke2} />

Ugyanezzel a JSX funkcióval nagyobb tömbben is le lehet rendereltetni több elemet így:

var valtozoAmibenJSXVan = (

<div>

<Person name="Sanyi" age="15" />

<Person name="A ló" age="5" />

</div>

); - Itt egy parent elementbe tettünk több JSX elemet. Amikor rendereljük, kikeresi magának a React, és a <DIV>-be ágyazva fogja a DOM-ba illeszteni.

ReactDOM.render(valtozoAmibenJSXVan, document.getElementById('node\_ahova\_tegye') );

Reactban a node-ot hook-nak mondják, valamiért.

Ha kizárólag React-al készül egy weboldal, akkor általában a HTML file-ban csak egy root tartalmazó elem van (<div id="root"></div>), és csak egyszer renderelik le az egész oldalt, amit addig a háttérben a React összerakott.

Amikor létrehozunk egy React-tal egy single page application-t, akkor először beinportálgatjuk a React-os segédkönyvtárakat és a .css- file-t is egy összefoglaló .js file-ba.(index.js). Reactnál lehet .css-t is importálni a Js fileba így:

import './index.css';

Ha ugyanarról a szerverről töltjük be, akkor a beimportált .js file-ok végére nem kell kiírni a .js-t, anélkül lehet beimportálni így:

import App from './App'; - itt az azonos mappában levő (./) App.js-t importáljuk, mégsem írjuk ki a .js-t.

JSX formázása CSS-el:

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Minden komponenst -hasonlóan a JSX- létrehozó funkciókhoz- külön .css fájlban formázzuk. A .css fileokat közvetlenül a formázandó komponenst tartalmazó .js file mellé kell tenni azonos néven, mint a JSX -.js. Végül be kell importálni a .css file-t a komponens .js file-jába így:

import './EztAKomponenstFormazo.css' - itt ki kell írni a .css végződést is!

Fontos emlékezni rá, hogy a JSX HTML-kódjában a class="" -t className=''-nek írjuk!

JS "logika" használata JSX-ben:

Mivel a JavaScript szabadon használható {}-között, ezért bármilyen JS kódot bele lehet tenni a JSX készítő funkciókba közvetlenül is, de úgy szokás, hogy előre elkészítjük a változót, és magába a JSX kódba csak a változó nevét tesszük bele (ami természetesen utal annak tartalmára). Ezzel olvashatóbb marad a JSX kód. Így:

const monthNow = new Date().toLocaleString('en-US', {month: 'long'})

function App(){

return <p>Month: {monthNow}</p>

}

Befogadó komponenes készítése React-ban:

Kellenek olyan komponensek, amik tartalmazó elemek a többi funkcióval is bíró elem számára. Hagyományosan ezeket Card elemeknek hívjuk Ezeket is JSX-ben lehet megcsinálni, de a különbség, hogy a props-a a teljes elemmennyiség lesz, amit a container tartalmaz majd (amik a children elemek). Így:

function Card(props){return <div className="CardOsztaly">{props.children}</div>}

a props.children - parancs a lényeg itt.

A Card komponenseket is szokás külön .js-ben tárolni, és .css -file is jár nekik. Miután ezeket megfelelően elkészítettük, és beimportáltuk az App.js-be, lehet síma HTML elemként belefoglalni a többi elemet így:

<Card><Minden egyéb HTML és JSX></Card> - itt a props.children egyenlő a kezdő- és záróTAG (<Card> és </Card>) közé beágyazott elemekkel.

Osztály megadása Card számára:

-----------------------------

A JSX komponensek csak olyan attribútumokat fogadnak a rendereléskor, amiket előre beléjük írunk. Mivel a Card-ok egyedi készítésű elemek, nem lehet alapértelmezetten a <Card className="osztálynév"> módon használni az osztályokat (se). Ha beillesztjük a React fá-ba és ott adunk meg className=""-et, az csak akkor fog működni, ha a JSX készítő funkcióban, ahol igazi HTML-kódot haszálunk is van className, és oda props.className útján hivatkozzuk be. A .css-hez is kell tehát külön propsot állítani be így:

function Card(props){

return <div className={`osztalyNevAmiAlapACardnak ${props.className}`}>{props.children}</div>}

Itt egy saját .css file-ja van a Card JSX-nek, az a .osztalyNevAmiAlapACardnak osztály, és a classList-hez még bármit hozzá lehet adni magasabb szinten, az lesz a props.className. Backtick-et használunk, mivel a html-be egy stringként adjuk be az osztálynevek szóközzel elválasztott listáját. Ezeket együtt egy küldi majd renderelésre a Card komponens ezzel a kóddal.

CUSTOM User Interface (UI) elemek:

Button:

Készíthető multifunkciós gomb komponens így:

JSX elkészítése (Button.js):

function Button (props){ return (<button type={props.type || 'button'} onClick={props.onClick} }> {props.children} </button>)};

-Itt előkészítjük a gomb komponenst arra, hogy a rendereléskor lehessen beállítani a típust (props.type). Kell beállítani hardcoded alapértelmezettet is, amit akkor használ, ha az első dinamikus változó falsey (mert nem létezik), ez a type-nál a 'button'.

Hasonló a helyzet az onClick-el is. Custom componensnél ez sem használható magától értetődően a rendereléskor, hacsak már a JSX-ben nincs megadva, hogy hogyan kell kezelni az onClick eseményt. Itt is a rendereléskor beadott onClick úgy lesz értelmezhető, ha a JSX-ben ez props-okkal már elő van készítve.

Azért, hogy lehessen belefoglalni akármit itt is a props.children-t használjuk a JSX-ben a <button> TAG-ek között.

Pro tipp: Egyszerre több argumentum tartalmának bevitelére is használhatjuk a props-ot, úgy, hogy spread operatorral küldjük a JSX-kódban így:

function JSXelem(props){ return <input {...props}/> }; Itt a props objektum minden key: value párját így fogja renderelni: key="value";

Error modal komponens;

A hibaüzenetek közléséhez való komponenseket hívjuk így.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

INTERAKTIVITÁS REACT-TAL:

A) HTML megfelelőkkel:

A JSX kódba közvetlenül beírható (ahogy a HTML-be is) az összes on- kezdetű eseménykezelő. Pl. a HTML onclick="JsFunkcio()" a JSX-ben onClick={JsFunkcio}.

Fontos különbség, hogy JSX-ben nem kell a végére (), azaz nem hívjuk meg, azt majd a React intézi.(Ha meghívnánk, akkor már a rendereléskor le lenne futtatva a funkció).

ELNEVEZÉSI KONVENCIÓ:

Az eseménykezelő funkciók nevének vége Handler-el végződik. Pl. buyClickHandler()

State-ek a Reactban

- - - - - - - - - - - - - - - - - - - - -

A Reactos DOM-tartalom megváltoztatására nem lehet szimpla Js-t használni, miután a renderelés lement. A React ugyanis állapotokat (state) tart fenn, és nem engedi megváltoztatni azokat a JS-nek. A state az adott komponens pillanatképeként fogható fel. Ha a state változik arra reagál a React az adott komponens újrarenderelésével.

Ha változtatni akarok a React-al készített webapp-on (a state-en), akkor a React belső beépített funkcióit kell használni.

Ennek menete:

1. Először be kell importálni az adott file-ba a React megfelelő funkcióit így:

import React, {useState} from 'react';

- itt az importált funkció a useState. A React hasonló ("hook") funkciói mind use- kezdetűek.

A useState funkció egy beépített React funkció, ami létrehoz egy "React hook"-ot. Kell neki egy alapérrtelmezett érték (default state value), hogy tudja mi az alapállapota. (Ez lehet props.cimkeNeve is, tehát változó.) Ha useState()-el készített változó van egy JSX -ben, akkor tudja a React, hogy reagánia kell az innen kinduló DOM változtatásokra.

2. Meghívom a useState() funkciót abban a JSX komponenes készítő funkcióban, amiben módosíthatóvá akarom tenni a tartalmat. A useState() meghívásnak a komponenskészítő funkció főrészében kell lennie, nem lehet egy beágyazott funkcióba írni.

A useState() egy két elemű tömböt ad vissza, aminek az első eleme annak a változóak az értéke amit a zárójelek közé (argumentumként) megadtunk, a második pedig egy funkció, amivel ez az előbbi érték változtatható.

Ahhoz tehát, hogy használjuk az értéket át kell mennie egy use- kezdetű funkción, és az onnan visszakapott változót veszi már komolyan a React.

Ahhoz pedig, hogy változtassuk ezt a callbackfunction-t kell használni.

3. Mindkettő elemét a visszakapott tömbnek új változóba kell tenni, amire a legegyszerűbb a dekonstruálás így:

const [valtozo, setValtozo] = useState(props.valtozoNeve);

Ugyan bárminek nevezhető, de konvenció, hogy a változó neve utlajon a tartalmára, és a visszakapott funkció pedig ugyanez legyen csak a set-előtaggal. A useState argumentuma (ami zárójelbe kerül) lehet egy hardcoded alapértelmezett érték, vagy egy üres string (''), mivel a useState mindent stringben tárol. Később úgyis felülírásra kerül, de a zárójel üres nem maradhat!

4. A callback function (itt: setValtozo) segítségével lehet új értéket megadni DOM elemnek az adott JSX / eseménykezelő funkción belül így:

setValtozo('Új szöveg például.');

Fontos! Ekkor újra lerendereli csak azt a JSX elemet, amiben a setValtozo értéke megváltozott. Ezért is érdemes apróbb kompnensekre bontani a nagy egészet, hogy csak kicsi DOM manipulációk történjenek (mert az belassítja a böngészőt).

Fontos #2! Addig nem is változik meg a változó értéke, míg a setValtozo le nem fut az újra rendereléskor. Tehát addig a régi értéke marad a változónak, míg a JSX funkciót le nem futtatták renderelés közben. Ez okozhat félreértéseket! Pl. ha két renderelés közben többször is változatnák az adott state értékét, akkor az csak az utolsóra reagálna, mert amíg nem volt renderelés, addig az előző értéket használja minden egyes setErtek(). Ezért ha már volt értéke a state-nek akkor az aktualizálását azonnal lefutó arrow funkcióval kell végrehajtani így: setErtek(prevErtek =>prevErtek\*2); Ezzel elérhető, hogy renderelés nélkül is több setErtek() segítségével változzon a state tartalma.

Ha több useState hook-ot is használok egy JSX funkcióban, akkor azoknak mindig azonos sorrendben kell lefutniuk a rendereléskor. (A state-be tett változó és a setState funkció amiket visszaad a useState már használható máshogy, de maga a useState() meghívása nem). Ez akkor tud sérülni, ha pl. if() függvénybe tesszük a useState hívását, vagy loopba, stb. Erről kapunk is hibaüzenetet. Mindig a JSX készítő funkció tetejére kell tenni magukban a [stateValtozo, setStateValtozo] = useState (); kifejezéseket.

HA A useState ALAPÉRTELMEZETT ÉRTÉKE EGY FUNKCIÓ, az minden egyes rendereléskor lefut, használva az erőforrásokat. Ha az egy memóriaigényes funkció, akkor lelassítja a gépet/szervert. Azért, hogy csak az első rendereléskor fusson le useState(alapértelmezett funkciója()) megintcsak az arrow funkció kell tök üresen így:

const [valtozo, setValtozo] = useState( ()=> alapErtFunkcio() )

Objektumok használata alapértelmezett state-ként:

[numb,setNumb] = useState({egy: 1, ketto: 2}); - itt a cimkék értékei úgy érhetők el, ahogy az objektumoknál szokás: const egyErteke = numb.egy; const kettoErteke = numb.ketto; Megfontolandó ezért, hogy lehet-e inkább külön state-eket létrehozni az egyes cimkékhez ilyenkor.

Ha objektum értékét akarom felülírni setValtozo()-val:

Fontos, hogy a setValtozo() mindig felülírja a state értékét, nem hozzáír, így arrow functionnal és spread operatorral lehet csak megoldani így:

setValtozo( prevValtozo => { return {...prevValtozo, 'felülírandó cimke': 'új érték' } } ); - itt egy prevValtozo nevű belső változó reprezentálja a state előző értékét, azt transzformáljuk egy arrow funkcióval úgy, hogy egy objektum tartalmazza a prevValtozo összes elemét {...prevValtozo} és azt módosítjuk helyben.

<FORM> használata REACT-tal:

Lépései:

#1. Elkészítjük a <form>-ot html-ben a JSX-en belül, addig minden azonos. Ennek az eseményfigyelőjéhez is bele kell írni egy on-kezdetű sztenderd html eseményfigyelőt (pesze camelCase-ban).

Ha <input type="text">-et használunk, akkor lehet onChange={esemenyKezelo}-t használni így:

<input type="text" onChange={esemenyKezelo}>

Ez minden value változásra (text-nél karakterbeírásra) reagál, de sokoldalú (bármilyen lehetséges változásra reagál az adott input típusnál), így multifunkcionális kompnens készítésre alkalmas.

#2. Megírjuk az eseménykezelő(ke)t az értékek kinyeréséhez: Az eseménykezelő felépítése hasonló egy vanilla JS eventListener callback function-jéhez, azaz kell neki egy event, aminek a target.value értékét kiolvashatjuk. És mivel a 'change' az esemény az onChange-nél, ezért minden billentyűlenyomással aktiválódik ez az eventhandler funkció. pl:

function eventHandler(event){

console.log(event.target.value)} - kiírja konzolba az aktuális tartalmát az input mezőnek.

DE! Csak akkor tudjuk használni is a JS változó értékét DOM manipulációra React-ban, ha alkalmazzuk a useState() -funkciót (Lásd fentebb).

#3. Megírjuk az eseménykezelőt a teljes kérdőív beküldéséhez:

Fontos, hogy alapértelmezetten a tartalmazó szervernek küldi be a böngésző a kérdőívet, és ezt a működést le kell gátolni. Így kell:

function bekuldesHandler(event){ event.preventDefault(); többi funkcióelem...}

Mivel az alapértelmezett működést legátoltuk, beküldéskor nem törlődnek az adatok automatikusan. Erre való a "two-way binding" (kétirányú kötés) nevű eljárás. Itt ez úgy néz ki, hogy az eseménykezelőbe beleírjuk, hogy nullázza le az adott értéket így:

function bekuldesHandler(event){ többi funkcióelem...; setValtozo('')} - ez az egyik kötésvég. A másik pedig az <input> mező alapértelmezett értékének beállítása a változóra így:

<input value={valtozo} ...többi attribútum> -itt tehát amint submitteli a felhaszáló a <form>-ot a bekuldesHandler() beállítja a valtozo értékét ''-ra, és az megjelenik az <input> mezőjében is.

#4. Beágyazzuk az eseménykezelőt a teljes kérdőív beküldéséhez: Ekkor kihasználhatjuk, hogy a böngésző submit gomb (alapértelmezetten a form aljára kerül és, type="submit" argumentummal rendelkezik) megnyomása esetén mit csináljon. Reactban a <FORM> nyitó TAG-jéhez érdemes onSubmit="" argumentumot megadni így:

<form onSubmit={bekuldesHandler}>

Single State / Multiple State

- - - - - - - - - - - - - - - - - - - - -

Egyetlen komponensen belül lehet többször is useState-et használni, ezzel több különböző változót is 'state'-ként tárolva egymástól függetlenül.

Így:

const [valtozo1,setValtozo1] = useState('')

const [valtozo2,setValtozo2] = useState('')

const [valtozo3,setValtozo3] = useState('')

Ekkor mindegyikhez egymástól függetlenül új érték rendelhető a setValtozo1/2/3('Új érték'); paranccsal.

DE összetartozó értéksoroknál lehet egyetlen useState-et is használni, ám ekkor egy key:value párokat tartalmazó objektum lesz az argumentum. Természetesen a tárolt érték is egy objektum lesz.

const [valtozoObjektum, setvaltozoObjektum]=useState({ valtozo1:'', valtozo2:'', valtozo3:''});

ITT FONTOS, hogy ha egy-egy komponens csak egy részét aktualizálja az információnak, akkor a többi értéket meg kell tartani az eredetinek. Ehhez a spread operatort(...) használjuk aktualizáláskor így:

setvaltozoObjektum((korabbiValtozoObjektum)=>{

return {...korabbiValtozoObjektum, valtozo1:'aktualizált érték'}

);

- itt spread operatorral bemásoltuk az eddigi értékeket, de a következő sorban a valtozo1 cimkéhez tartozó értéket felül is írjuk, majd mindezt állítjuk be a valtozoObjektum új értékének.

FONTOS! Az egész bele van téve egy arrow-function-ba, ahol a bemenet az előző objektum változat, a kimenet pedig az aktualizált. Itt a korabbiValtozoObjektum nevű objektum egy belső név, ami az eddigi state-re utal, bármilyen nevet is adjunk neki.

Ez azért kell, hogy nyomatékosítsuk az állapot azonnali frissítését a React-nak, különben megeshetne, hogy az előző, meghaladott állapotot használja a spread operatorral a React. Ezt csak akkor kell használni, ha egy olyan state-et szeretnénk menteni, ami a korábbi state aktualizálását is megkívánja.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

ADATÁRAMLÁS REACT COMPONENT TREE-ben

I.) Fentről (az App.js-től) lefelé a komponensek felé a props-ok adják meg az információ áramlásának a lehetőségét (lásd korábban).

II.) Lentről felfelé (az App.js és azon belül a renderelést végző App() function irányába):

Alapelv: nem lehet kihagyni az adatáramlásból a köztes komponenseket (hasonlóan a fához), az adatokat csak child-ról parent-re lehet felfelé küldeni.

Felfelé küldéshez különleges property-ket kell alkotnunk, amelyek elnevezési konvenciói:

onUserCselekedetAdatai

- Itt is on- al kezdődjön, mint ahogy a beépített eseményfigyelőké, mivel ez is egy funkciót fog kapni értékként.

- A többi rész pedig utaljon a felfelé küldött adatok tartalmára.

- CamelCase természetesen.

Ezeket az info-felfelé-küldő funkciókat a JSX funkción belülre kell helyezni.

Lényege:

Úgy oldható meg, hogy a parent komponens JSX-ében készítünk egy funkciót, amit majd a child JSX-ében hívunk meg, így merítjük ki az adatot eggyel magasabb szintről.

Úgy is tudom elképzelni, mint egy gémeskutat, ahol a kút rúdja a parentJSX, a vödör a merítőfunkció (ami a parent része), a childJSX maga a kút, és a víz az adat. Leküldjük a vödröt (merítőfunkció) a kútba (child), és felhozza az adatot.

Lépései:

#1. A legalsó (User Infó kinyerő, valszeg <input>) komponenst mikor meghívjuk a parentjában, akkor a JSX (a html-nek kinéző) részben egy argumentumot adunk hozzá ami (a fent leírt konvenciók szerint) on- kezdetű, és a "merítő" funkciót fogadja majd.

Kb így:

function ParentJSX(){ return <p> <ChildComponent onMerites={meritoFunkcio} /> </p> }

#2. Készítünk egy funkciót, ami a merítést végzi majd a parent JSX-ében így:

function ParentJSX(){ function meritoFunkcio(childbolVettAdat){

const meritettAdat = {...childbolVettAdat};

többi szokásos JSX elem};

#3. Az infofelvevő child komponensnek is kell bemenetet (props) adni, majd azon belül lefuttatni a parent JSX-ében megírt merítő funkciót, DE azon a néven hivatkozva, amilyen argumentumnévvel elláttam, amikor parent JSX-ben meghívtam a child JSX-et (a példában: <ChildComponent onMerites={meritoFunkcio} />). Tehát a props.on-akármi néven (itt: onMerites(), nem pedig a meritoFunkcio!).

kb. így:

function childJX(props){props.onMerites(kiMeritendoAdat) ;többi szokásos JSX elem}

- Itt a kiMeritendoAdat az az input amit a useState()-(ekk)el gyűjtöttünk a childban.

Az analógiát lezárva: fentről kapja a merítő funkciót (vödröt), a props.-on a lánc, amin keresztül leküldik hozzá a vizesvödröt, amit a gémeskút (parentJSX) felhúz majd a state-el kinyert adattal tele. Ahogy kiemeljük az infót a mélyből, az a "lifting the state up."

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

LIFTING THE STATE UP

Az információ áramlás parent-child viszonylatban lehetséges, sibling-sibling viszonylatban nem. Ha tehát egy sibling-hez akarok információt eljutattni, akkor azt csak a legközelebbi közös (grand)parent-en keresztül lehet. Úgy lehet elképzelni mint egy fát megint csak, ahol két ág között az anyagáramlás csak a legközelebbi elágazáson keresztül lehet, ami néha maga a törzs (a Reactban az App.js).

CONTROLLED COMPONENT VS UNCONTROLLED COMPONENT

:.:.:.:.:.:.:.:.:.:.:.:.:.:.:.:.:.:.:.:.:.:.:.:.

Mind kontrollált komponens az, ahol two-way bindingot használtunk -azaz a childból kimerített adat hatással volt a parent tartalomra, majd az onnan visszaküldött adat a child tartalomra- (példa: az input (child) komponens értékének beállítása a state-be rakott beküldött értékre a parent komponent props-ának megadásával). Szóval ha a ey parent-child komponenspár befolyásolja egymás tartalmát, akkor controlled.

Azért "kontrollált" komponens, mert a tartalmát a React tudja a fenti módon kontrollálni.

PRESENTATIONAL VS STATEFUL COMPONENT

:.:.:.:.:.:.:.:.:.:.:.:.:.:.:.:.:.:.:.:.:.:.:.:.

Másnéven stateless vs stateful, másnéven dumb vs smart compnent.

Minden komponens stateful/smart, ha létrehoz 'state'-et a useState()-el, ami nem az csak presentational/ stateless /dumb.

LISTA KÉSZíTÉS JSX-el (DINAMIKUSAN):

-:-:-:-:-:-:-:-:-:-:-:-:-:-:-:-:-:-:-:-:-

Alapja, hogy a JSX-be beilleszthető egy tömbnyi HTML/JSX kód is, amit automatikusan dekonstuál és beilleszt a DOM-ba a React. Így:

{[<p>Első sor</p>,<p>Második sor</p>]}

-ekkor a DOM-ba bekerül:

Első sor

Második sor

Persze ennek is kell tartalmazó elem.

Ha van már egy tömbnyi adatunk (akár objektumok is lehetnek a tömb elemei), akkor tudunk létrehozni belőle egy sorozat JSX komponenst is így:

adattTomb.map(item=> {return <JSXelem key={item.id} name={item.name} data={item.data}/>})

Itt a JSX csak akkor renderelődik menet közben, ha az adat/tömb amiből nyerjük az infót egy state-ben van rögzítve (state-ban tárolt tömb aktualizálását lásd még korábban).

Key-ek használtata az egyes listaelemek bezonosításához.

++++++++++++++++++++++++++++++++++++

Ha a key property-t kifelejtettük a tömb renderelendő elemeiből, akkor az alábbi hibaüzenet fog megjelenni:

Warning: Each child in a list should have a unique "key" prop.

Ekkor is működik az Reactos oldal, de egyedi key híján minden egyes alkalommal amikor a tömb új state-et kap, az összes eleme újrarenderelődik, ami sok tömbelem esetén már lassuláshoz vezet, illetve bugokat okozhat.

Lehetne egyedi key-t generálni, ha nincs eleve valami ID-ja az objektumban, mégpedig a .map() beépített index-ével, ami a második argumentum alapból.

ÍGy:

adattTomb.map((item,index)=> {return <JSXelem key={index} data={item.data}/>})

- Itt a tartalmazó tömbben betöltött helyét fogja key-nek (egyedi azonosítónak) használni. Csakhogy ha több tömbből vesszük az infót egy istába, akkor több 0. hely lesz, így több key=0 is.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Conditional Content

Azaz feltételhez kötött tartalom.

Bizonyos tartalmak csak akkor jelennek meg, ha a megadott feltételnek megfelelő. (A kurzus példájában az ExpensesFilter.js-ben volt ilyen.)

Alkalmazási módok:

A)TERNARY OPERATORRAL A JSX-ben:

Feltételeket lehet a JSX-be is közvetlenül beilleszteni, de csak ternary operatorral, az if(){}else{}-t nem engedi lefutni a React. Így:

{adatTomb.length == 0? <p>Nincs találat</p> : adatTomb.map(item=> <JSXem data={item}/>) }

- itt, ha üres tömb a bemenet akkor jön 'Nincs találat üzenet', egyébként az adatot felhasználjuk a rendereléskor.

B) EGY TRÜKKEL A JSX-ben:

A JS-ben úgy működik, hogy ha egy feltétel után &&-el csatolunk egy második 'álfeltételt', akkor azt adja vissza, amennyiben az első valós feltétel igaz. Így:

{adatTomb.length == 0 && <p>Nincs találat</p>} - persze itt nem tudunk külön értéket megadni akkorra, ha hamis az első állítás. DE! Lehet egy újabb állításba foglalni a másik opciót ugyanezzel a trükkel így:

{adatTomb.length > 0 && adatTomb.map(item=> <JSXem data={item}/>) }

Ekkor két sorba írjuk a conditional contentet, de jobban olvasható.

C) KIEMELHETŐ A JSX készítő funkció elejére, ahol lehet használni bármilyen JS kódot szabadon, hogy legeneráljuk a JSX kódot, amit a return után változóba téve alkalmazhatunk.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

FORMÁZÁS A REACT-BAN:

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Stílus (CSS -tulajdonságok) beállítása REACT-ban:

¨¨¨¨¨¨¨¨¨¨¨¨¨¨¨¨¨¨¨¨¨¨¨¨¨¨¨¨¨¨¨¨¨¨¨¨¨¨¨¨¨¨¨¨¨¨¨¨¨¨¨

Az a baj a beimportált .css file-okkal, hogy bár csak egy-egy kompnensbe importáljuk be mégis mindenhol alkalmazható lesz a teljes window-ban. Mivel a komponensek szabadon cserélhetők, nehéz lenne követni, hogy éppen benne van-e már az adott css szelektor korábban. Ha igen, akkor nagy kavarodást okozna. Ezért érdemes a következő lehetőségek közül választani:

A) "Inline" formázás:

^^^^^^^^^^^^^^^^^^^^

Ez a legmagasabb szintű formázás, ami mindig üti a .CSS-file-okban leírt szabályokat, ezért érdemes megválogatni, hogy mikor használjuk!

A JSX file-ban a sztenderd (általam alig ismert) HTML formázást lehet alkalmazni, de itt is van egy kis különbség.

A JSX komponens renderelésekor az argumentum listába kell beleírni, hogy style={}, és OBJEKTUMOT kell beadni neki. Tehát dupla{{}}-van, a külső a kiugrás Js-be, a belső az objektum határa.

Így:

<JSXelem style ={ {height:'100%', width:'100%', backgroundColor:'red'} }/> - itt látható, hogy a kötőjeles CSS szabályneveket a Js-hez hasonlóan camelCase-ben kell megadni,(bár a 'background-color':'red' is működött volna).

Feltételesen és dinamikusan is lehet formázni, melynek lépései:.

#1. Kell egy state, ami aktualizálja majd a komponenst miután teljesül a feltétel. Így érdemes:

const [isValid,setIsValid] = useState(true);

#2. Megcsinálom a funkciót, ami dönt az xy kérdésben, és beállítja a statet így:

function xyEventHandler(){feltétel? isValid(true): isValid(false)}; -

#3. <JSXelem style={{color: !isValid ? 'red' : 'black'}} /> - itt ternary operator-t tudunk használni arra, hogy az objektum címkéjét (color:) feltételhez kötve adjuk meg.

B). Classok hozzáadása/elvétele útján:

^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

Lépései:

#1. Létrehozok egy class-t a .css-ben, amelyet akkor teszek rá az adott elemre, ha igaz a feltétel, és egy olyat, ami felülírja az előző class szabályait (arra az esetre, ha hamis lesz a feltétel). Így lehet:

.szabalyok-igaz-esetre{color:red;} - Ez az "igaz" (alapértelmezett) állapot.

.szabalyok-igaz-esetre.felulirva-mert-hamis{color:'black';} - Itt olyan css szelektort csinálunk, ami akkor él csak, ha egyszerre, ugyanazon a TAG-en van rajta az alapértelmezett és a felülíró class is. (MERT NINCS SZÓKÖZ KÖZÖTTÜK!).

Ha van beágyazott elem is, amit külön formáznék, akkor így:

.szabalyok-igaz-esetre.felulirva-mert-hamis h2{color:green;} - itt a beágyazott h2 szövege zöld lesz.

#2. Kell egy state, ami aktualizálja majd a komponenst miután teljesül a feltétel. (Pont mint a A.) pontban) Így érdemes:

const [isValid,setIsValid] = useState(true);

#3. A JSX komponens létrehozó funkciójában beállítunk egy dinamikus className-et így:

<tag className={`szabalyok-igaz-esetre ${!isValid? 'felulirva-mert-hamis': ''}`} /> - itt egyszerűen Js ternary operatorral egy backtick (``)-es stringet teszünk bele, ahol a feltételes részt az isValid state-hez kötjük.

Formázott komponensek külső könyvtár (Styled coponents)

::::::::::::::::::::::::::::::::::::::::::::::::::::::

Hozzá lehet adni egy külső (3rd party library) könyvtárat, a "Styled coponents"-nevűt. (A telepítést lásd az npm megnyitása.txt-ben.) Ez majd egyedi azonosítót generál minden css szelektorhoz, így nincs veszélye a keveredésnek.

1. Először be kell importálni a az adott .js file-ba a library-t így:

import styled from 'styled-components';

Ugyanakkor nem kell .ccs-t importálni!

2. Ha pl. egy gombot akarunk csinálni, akkor:

const Button = styled.button``; - itt csupán meghívjuk a beimportált objektum button metódusát, de nem a szokásos ()-el indítjuk, hanem dupla backtick-el (``) (ezt hívják attacked template literal-nak). Máshol is használható a dupla backtick funkció meghíváshoz a Js-ben, de itt ez a kötelező.

A styled-components tartalmaz mindenféle beépített HTML TAG-re formázást, ami (itt) a styled.htmlTAG``;-szintaxssal meghívható (új instance készíthető).

A backtick-ek közé lehet illeszteni a CSS formázási szabályokat, de nem kell bele szelektor{}-rész, mivel már eleve erre az elemre értelmezendő.

const Button = styled.button`color:red;`;

Ha pszeudo-szelektort, vagy pontosítást (pl TAG nevet) használunk, annak viszont kell {} jelzés, amire a & (ampersand) karakter való így:

const Button = styled.button`color:red; &:hover{color:purple;}`; - itt csak akkor lesz lila a szöveg, ha fölé visszük, egyébként piros.

& h1 {cssSzabály:értéke} - így lehet a nested elemet célozni, ahol érvényes a parent element formázása is, amire az &-vel utalunk.

Dinamikus formázás Styled-components-el:

- - - - - - - -- - - - - -- - - -- - - -

Itt is két lehetőség van:

Az ELSŐ: A korábbiakhoz hasonlóan useState és a felülíró css osztály rátételével lehet a feltételes formázást beállítani (lásd korábban a "Classok hozzáadása/elvétele útján" rész 1-3. pontja), csak egyszerűbb szintaxissal.

Mivel az alapértelmezett formázás már rajta van, csak azt kell eldönteni, hogy felülírandó-e az.

<tag className={!isValid && 'felulirva-mert-hamis'} /> - itt azt a Js trükköt használjuk, hogy ha egy boolean állítás első fele igaz, akkor az && jelek utáni kódot használja. Tehát itt: ha igaz a useState-változós kifejezés, akkor ráteszi a gátlót, ha nem, nem, nem kell az igaz opcióra semmit beállítani, mert az az alapértelmezett.

A MÁSODIK lehetőség az, hogy a props-okon keresztül állítjuk be a styled-components-et.

1. lépése a props létrehozása a State tartalommal:

<JSXkompones feluliroProp={!isValid} />

2. lépésben a props használható a styled-component-en belül azon a helyen, ahol a feltételes formázást foganatosítanám így:

const valtozobaTettTAG = styled.TAG`color: ${props => props.feluliroProp ? 'red' : 'black'};`;

Itt az adott css szabály szövegébe ${} kiugrással teszek egy arrow functiont, ahol a props.feluliroProp értékétől teszem függővé a beillesztett stringrészletet, ternar operatorral.

CSS Modules könyvtár használatával

::::::::::::::::::::::::::::::::::::::::::::::::::::::

Ez egy olyan külső library, ami az npm-be bele van már integrálva, de más coplier-hez hozzá kellene adni. Ez is egyedi szelektorneveket hoz létre a fordítás után, bár jobban értelmezhetőket mint az előző.

Szintén használja a React könyvtárat, így kell az import React from "react";

Ebbe is kell importálni CSS-t, de máshogy kell elnevezni a file-t is így:

Akarmi.module.css - .module.css-re kell végződnie.

És más szintaxissal is importálni így:

import styles from './Button.module.css'

Eze után már úgy értelmezhető a beimportált styles mint egy objektum, és úgy is lehet a formázáskor hivatkozni rá. Pl.:

<button className={styles.cssSzelektor}>

Vagy:

<button className={styles[css-szelektor]}> - csak ez utóbbi működik, ha kötőjel van a class nevében!

A media query-ik is egyszerűek, mert csak beírod a css szabályok közé.

Dinamikus formázás CSS modules-al:

- - - - - - - - - - - - - - - - -

Itt is a korábban látottaknak megfelelő a megoldás, csak ``-ek közé rakva és kiugrásokkal.

<div className={`${styles["form-control"]} ${!isValid && styles.invalid}`>

A media query-ik itt simán a CSS file-ba mennek.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

DEBUGGING A REACTBAN

Hasznos eljárás:

Ha nem működik a renderelés se, akkor biztosan van hibaüzenet a terminálban, vagy a localhost-on, ami irányt mutat.

Meg lehet nézni a console-t is, mert egyes hibatípusokra ott is megjelenhet infó.

Ha lerenderelődik, de nem működik (logikai hiba)- akkor végig kell menni a az érintett komponenstől kiindulva, ahol a hiba van az ez egész láncon akár az App.js-ig, és onnan le is, míg meg nem lesz.

És lehet használni a DevTools-t is a böngészőben (erről lásd később).

Hiba típusok:

Parsing error: - közös parentben kell lennie a JSX-komponens készítő funkció által visszadott html szekvenciának. - tehát a hiba az lehet, hogy nincs egy elembe ágyazva a html kód, vagy nincs bezárva a vége />.

no-undef:

'valtozoNeve' is not defined - nincs olyan nevű változó/funkció, amire hivatkozunk a kódban.- Általában elgépelés van mögötte.

Debugging a böngésző DevTools-ával:

>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

A Vizsgálat -nál a Sources tab-ra kell menni. Ide küld az npm comlier infót a lefordítás előtti kódról is.

Meg lehet nézni a forrást localhost:3000 felhő, static/Js mappa, C:/Users/User..stb- mappában az src mappában minden komponens eredeti fordítás előtti forráskódja.

Amikor a keresett forrásfájl megvan, be lehet szúrni egy breakPoint-ot, úgy, hogy a kód egy adott sorának számára kattintasz, mire az színes lesz. Itt fog megállni a kód futtatása, akkor, mikor legközelebb ez a kód aktiválódna (pl. a user inputja révén). Itt különböző ablakokban megjelennek infók, pl. változók és azok értékei, stb.

A step into next function call -gombra (lefelé, egy pontra mutató nyíl) kattintva lépésenként lehet követni a történéseket. Ha ilyenkor a változók neve fölé viszem a kurzort, azok aktuális tartalmát kiírja nekem.

Böngészőre telepíthető React Developer Tools

>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

Rá kell keresni a google-ban, és automatikusan felajánljaa telepítést.

Amikor kész a console-nál megelenik két új tab, a profiler és a components.

A Components kell most ebből a debughoz.

a Components kiválasztásakor megjelenik a komponens fa és a State-ek is.

===========================================================

HALADÓ JSX ESZKÖZÖK:

===========================================================

WRAPPER elemek:

A <div> suop (div-leves) egy kifejezés arra, amikor a lerenderelt oldal tele van felesleges div-ekkel, csak azért, mert a JSX komponensek-nek kell lennie egy tartalamazó TAG-nek, és ezek ellepik a DOM-ot, zabálva a memóriát és lassítva az appot. Ezt küszöbölik ki a wrapper-módszerek:

A) Wrapper:

Szokás csinálni a hagyományos components mappában egy Helpers mappát is az ilyen elemknek. A Helpers mappában azután a Wrapper.js-be nem is kell importálni a React-ot.

Tartalma csak ennyi:

export default const Wrapper = props =>{return props.children;}

A props.children - ahogy a Card elemnél is írtam, arra való, hogy mindent ami a nyitó és záró TAG közé kerül visszaadjon.

Lényegében tehát ez egy üres JSX, amiben nincs html TAG, de átveri a Reactot.

Használata így:

function divMentesJSX(){ return(<Wrapper> <egyikTAG> <másikTAG> </Wrapper>)};

B) React.Fragment

Természetesen egy ilyen fontos és egyszerű funkció be is van építve a React-ba, ezt hívják fragmentumnak (töredéknek) A Wrapper helyett, amihez előre kell megírni a funkciót lehet használni a <React.Fragment> - TAG-et.

C) Üres <></>-ek

Egyes kiegészítők telepítése esetén lehet üres html TAG-et is használni <></> Wrapper/fragment helyett.

PORTALS:

Mivel a React-ban minden komponensekben van, a HTML elemek nem oda kerülnek, ahol egy simán megírt HTML file-ban kerültek volna. Ez probléma mert:

Nehezebben átlátható a többi fejlesztőnek, ha csapatban dolgozunk.

A screenReader-ek (vakoknak) nem tudják értelmesen felolvasni a tartalmat.

Az overlay elemek (pl.: párbeszédpanel, backdrop=elsötétítés stb.) esetén fokozottan igaz ez, mivel ezek a css formázás miatt a tetejére kerülnek (és elérhetetlenné teszik az alatta levő tartalmat), de a HTML struktúrában el vannak dugva. Ennek kiküszöbölésére való a React portal.

Ez máshová rendereli le a html elemet, mint ahol a React struktúra szerint lenne.

Használata:

#1. Ahhoz, hogy máshová kerüljön a renderelt HTML, kell egy hely az index.html-ben (a public mappában van általában) ahová kerül majd. Létre kell ott hozni a helyét, id-vel beazanosítva. Pl. overlay-eknél rögtön a <body> alá így:

<div id="backdrop-root"></div> - Egy hely a sötétítő hátérnek

<div id="overlay-root"></div> - és egy hely a felugró üzeneteknek.

#2. Meg kell adni a JSX készítő funkcióban, hogy portalt használjon.

Ehhez importáli kell a .js file-ba ReactDOM-ot, ami nem ugyanaz a library, mint a React! (A ReactDOM az "adapter" a böngészőhöz, mert amúgy a React-ban megírt kódot máshol is lehet használni.)

Így:

import ReactDOM from 'react-dom';

Ez után lehet használni a ReactDOM.createPortal()- beépített funkciót, amelyet a JSX funkcióban, a HTML kódból kiugorva kell meghívni. ÍGY:

function JSXkeszito(){return <p>{ReactDOM.createPortal(<htmlTAG />,node)}</p>}; - itt a htmlTag bármi lehet, és argumentomok is, gyakorlatilag rendereléskész JSX kell ide. A node, pedig a szokásos document.getElementById('overlay-root')-módon meghatározott hely. Ugyanúgy fognak működni a JSX elemek, csak máshová lesznek beillesztve a DOM-ba.

REFS

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Hasnonló a useState-hez, ebben is lehet információt tárolni, azonban ennek az információnak az updatelése nem renderelteti újra a komponenst. UseState-el infinite loop-ba lehet kerülni, ha olyan funkciót írunk, ami rendereléskor változtatna egy state-et, mert akkor újra kell renderelni, és az megint változtatná a state-et, stb. Ellenben useRef nem okoz infinite loopot.

Leggyakoribb használata: Gyorsan ki lehet olvasni adatokat pl. input-okból.

A referenciák (hivatkozások)

Ezt is be kell importálni a React mellé így:

import React, {useRef} from "react";

Használni ezt is csak a komponens készítő JSX funkcióban lehet így:

export default function JSXkeszito(){

const valamiRef = useRef(); - itt létrehozom a ref-et

return <htmlTAG id="ez-a-tag" ref={valamiRef} /> }; - itt pedig felhasználom a JSX-ben.

Miután lerendereltem, a valamiRef értéke egy objektum lesz, aminek a felépítése ilyen: {current: TAG#ID} -itt a példában: {current: htmlTAG#ez-a-tag} A ref mindig objektumot ad vissza current: -címkével és a DOM node-ot mindenestül mint érték. A valamiRef.current.barmilyenErtek módon bármilyen node értéket ki lehet olvasni, amire szükségünk van. Az inputnál, pl. a value-t, mint gyakori felhasználás. DE nem szbad vele módosítani adatot, mert azt nem tudná a REact kezelni.

Szintén használható arra, hogy egyes state-ek előző verzióit megőrizzük vele.

const elozoStateErteke = useRef('');

useEffect( ()=>{ elozoStateErteke.current = stateNeve}, [stateNeve] )

-Itt a dependency a [stateNeve], ami ha változik, akkor rendereléskor lefut egy useEffect, ami a kívül létrehozott useRef- által létrehozott objektum 'current' nevű cimkéjéhez rendeli a state tartalmát. Tehát az előző state értéke elérhető lesz így:

elozoStateErteke.current

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

SIDE EFFECTS (useEffect)

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Side Effect minden olyasmi ami nem kapcsolódik közvetlenül a felhasználói felület kirajzolásához, vagy az onnan történő információ kinyeréséhez. Pl. HTTP request-ek és azok feldolgozása. Azért van rájuk szükség, mert a JSX készítő funkciókat minden alkalommal újra lefuttatja a React amikor valami változik, így a hosszabb várakozást igénylő (aszinkron?) funkciók nem lehetnek ezek részei, és nem lehet csak egyszer lefuttatni nékülük egy JSX-et.

Egyszerűbbé teszi az állapotfüggő újrarenderelés kontrollját is.

Mindig a rendrelés után futtatódik le.

useEffect( ()=>{callbackfn} , [dependencies] );

callbackfn- az a funkció, ami lefut, ha a dependencies tömb által tartalmazott 'függelékek' bármelyike megváltozik. Ha azt akarom, hogy csak 1x fusson le, akkor üres tömböt adok meg dependencies-nek, mert az soha nem változik. Így:

useEffect( ()=>{callbackfn} , [] );

/\* kis kitérő:

localStorage - a böngészőbe beépett rendszer. Itt lehet tárolni adatokat miután letöltöttük őket http requesttel pl.

Lehet raktározni a localStorage-ban olyan infót, ami azután is megmarad, hogy a browsert bezárjuk. A Chromnál ez jelenleg 10MB.

Manuálisan is lehet be tölteni infót így:

localStorage.setItem('raktarNeve', 'raktár tartalma string');

Ha nem eleve stringet akarok tárolni, akkor használhatom a JSON.stringify() funkciót így:

localStorage.setItem('raktarNeve', JSON.stringify([1,2,3,{number:'4'}, false]));

Kinyerhető a helyi raktárból így:

localStorage.getItem('raktarNeve') -mivel mindenképpen stringet kellett tárolnunk lehet használni az eredményen JSON.parse() parancsot így:

JSON.parse(localStorage.getItem('raktarNeve'));

\*/

Az egyszer lefutó useEffect()-re jó példa, ha a localStorage-ből akarok kinyerni valamit, feltéve, hogy már mentettem oda, akkor ezzel lehet így:

useEffect( ()=>{

const taroltAdat = JSON.parse(localStorage.getItem('raktarNeve'))

if(tartoltAdat){setAdatom(tartoltAdat)}

, []} - itt az előzetsen tárolt adatot állítom be egy state-nek a setAdatom state updater funkcióval, de csak egyszer, amikor először betöltöm az oldalt ebben a munkamenetben.

A dependency listába általában azokat a state változókat rakjuk, amiknek a tartalmát befolyásolni akarjuk a useEffect-ben. Bele teendők még felhasznált saját készítésű funkciók is elvileg, de nem világos, hogy azok mik, mert a state updateing-ek (setState) nem tartoznak bele... Az biztos, hogy más file-okból importált változók nem jöhetnek bele, csak az adott komponensben készítettek. Tehát azokat a változókat/funkciókat kell használni dependency-ként amik megváltozhatnak a komponens használata során.

Clean UP SideEffect:

Aszinkron funkcióknál éredemes úgy használni a sideEffect-et, hogy időnként kiürítjük a sideEffect tartalmát. Ehhez a return-t használjuk, és visszadunk egy azonnal lefutó arrow function-t így:

useEffect( ()=>{egyéb kód; return ()=>{kód} } , [dependencies] ); - Ez ez első lerendereléskor nem fut le.

Pédául ha minden billentyűlenyomásra küldenénk egy HTTP requestet (mondjuk mert ellenőrizzük, hogy létezik-e már az adott emailcím), az nagy terhelést tenne a szerverre. Ezért megtehetjük, hogy bizonyos időközönként ellenőrzünk csak, és ehhez a JS setTimeout funkciót használjuk:

useEffect(()=>{

const idozitettFunkcio = setTimeout(()=>{ ritkítandó http -kérések kódja )},500);

return ()=>{clearTimeout(idozitettFunkcio);}

},[dependencies])

A return érték arra való, hogy mikor lefut a sideEffect funkció, törölje a rendszeresen lefuttatott (timeout) funkciót.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

useReducer() HOOK

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

A useState-hez hasonlóan state-ek kezelésére való, általában akkor választjuk, ha a state tartalma objektum. Bonyolultabb kód, de összetett helyzetekben hatékonyabb.

[(Talán azért reducer, mert korlátozza a választási lehetőségeket? Vagy az objektum-információját? Nem kell annyi state, ha objektumben tároljuk az állapotokat? )]

Érdemes használni összetartozó statek együttes kezelésére, különösen, ha egy state egy másik state függvénye (pl. az emailcím state-től függ, hogy az emailcímvaliditás state értéke mi.)

#1. LÉPÉS: Szintén be kell importálni:

import React, {useReducer} from 'react';

#2. LÉPÉS: Meghívni a JSX készítő funkcióban kell ezt is a funkció elején így:

const [state, dispatchFn] = useReducer(reducerFunkcio, eredetiÉrték); \*

- itt is két dolgot kapunk vissza ezért dekonstruáljuk:

I.) A state ugyanúgy mint a (useState-ben).

II.) A dispatchFn (dispatch angolul = intézkedés) amit majd meghívunk hogy (megadott paraméterek mentén) updatelje a statet.

A reducerFunkció -az első argumentum-ról lásd később.

az eredetiÉrték az, ami általában egy objektum {eredeti: 'érték'}, de lehet más típus is.

\* Létezik egy harmadik argumentum is az initial function, ami egy olyan funkció, ami...?

#3. LÉPÉS: Reducer funkció definilása:

A reducerFunkcio az, amit a dispatchFn-en keresztül hívhatunk majd meg és a useReducer első argumentumaként használjuk. Ezt nem muszáj a JSX-en belül megírni, ha mégis akkor a useReducer elé tegyük (főleg, ha arrow fukcióval dolgozunk). Ennek az alaképlete a következő: (prevState, action)=> newState -azaz bemenetként megkapja az még éppen aktuális state-et és a transzformáló akciót, és átalakítja őket az új state-té.

Definiálása az előző példát továbbvíve így:

function reducer(state, action){

switch(action.dispatchFnbenMegadottTipus){

case 'tipus1': return {state.eredeti + action.payload.textToConcat}

case 'tipus2': return {state.eredeti -1}

default: return state;

break;}

}

- Itt a reducer funkció segítségével választunk több érték közül, annak a tipusnak a segítségével amit a dispatchFnból kinyerünk majd (lásd később). Ez a reducer funkció fogadja első argumentumként a még aktuális state-et. A második argumentummal érhető el a dispatchFn meghívásakor argumentumként megadott objektum tartalma a cimkeNeve (itt: eredeti) segítségével. (Itt van action.payload -is, erről lásd a dispatchFn-ben).

#4. LÉPÉS: Handler funkciókban használható már a dispatchFn így:

function tipus1Handler(){

dispatchFn( {reducerFnbenMegadottTipus: 'tipus1', payload:{textToConcat : textValtozo} } ) };

- Itt csak az egyik példabeli lehetőséget használtuk, de egy másik handlerben meg a másikat lehetne.

A payload (angolul hasznos teher) az az információ, amihez még hozzá kell férnie a dispatchFn funkciónak, hogy el tudja végezni a feladatát. A payload egy objektum, ami tartalmaz minden változót, ami szükséges.

Jó gyakorlat:

A dispatchFn-nek egy objektumot szokás argumentumként megadni action-ként, ahol a key:value párból a value hagyományosan 'CSUPA\_NAGYBETUS\_STRING'. WDS Kyle tippje, hogy csináljunk egy azonos nevű változót, ami tartalmazza ezt a stringet, így a VSCode kiegészíti nekünk automatikusan, nem fogjuk elgépelni (ami nehezen felderíthető hiba lenne.) itt: (const CSUPA\_NAGYBETUS\_STRING = 'CSUPA\_NAGYBETUS\_STRING')

CONTEXT (useContext hook)

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Mint arról korábban volt szó, sima Js változókban nem lehet adatot tárolni, mert azokkal nem tud a React dolgozni. Sima state-ekben raktározott információt pedig csak props-okon keresztül, parent-child viszonylatban lehet küldeni. Így ha egy grandchild-nak kell küldeni fel a grandparent-nek akkor (fölöslegesen) át kell mennie a parenten is az infónak props-okon keresztül, még ha az nem is használja. Ez nagyon körülményes, ezért találták ki a statek háttértárolójaként a context-et.

Létre lehet hozni külön file-ban így:

XY-context.js:

import React from 'react';

const XYcontext = React.createContext();

export default XYcontext;

- ennyi a file tartalma. Először beimportáljuk a Reactot. Másodszor létrehozzuk a kontextus-t, ami egy objektum benne JSX kódokkal és egyéb adatokkal. Ebbe létrehozáskor bármilyen típusú adatot is beletehetünk, de általában üresen hagyjuk és később töltjük fel (lásd .Provider-nél).

Van minden context-objektumnak két fő funkciója a Provider (szolgáltató) és a Consumer (fogyasztó). Ahová a provider szolgáltat, ott lehet a consumer-nek fogyasztania.

PROVIDEING:

Ahhoz, hogy szolgáltathassunk be kell ágyazni azt a <JSX>-et a <Provider>-be amiben használni akarjuk a kontextus adattartalmát. Ezt úgy érhetjük el, ha a beépített .Provider JSX készítő funkcióját alkalmazzuk a kontextusunknak így:

<XYcontext.Provider value={ {adattartalom: 'bármi lehet'} }> <AholFelhasznalom/> </XYcontext.Provider>

A .Provider value ={} argumentumába megy bele minden amit el akarunk majd érni ott, ahová importáljuk majd. Itt egy objektumot tettünk bele, amit majd szintén elérhetünk a kontextust használó komponensekben. Ha dinamikusan akarom változatni a kontextus tartalmát, akkor egy szimpla useState()-el készített state lehet az objektum adott címkéjével jelzett érték {adattartalom: stateValtozo}.

Innentől az <AholFelhasznalom/> komponens és minden abban lerenderelt child is hozzáfér a kontextus tartalmához. Tehát a providert ott használom, ahol lerenderelem használathoz a komponenst (legmagasabban az App.js-ben).

CONSUMING:

Ezt két módon lehet: .Consumer-el és useContext() hook-al (utóbbi ajánlott).

I. -> "Fogyasztás" .CONSUMER-el:

A .Consumert ott használom, ahol definiálom a felhasználó komponenst (a komponens JSX készítő funkciójában).

Abban a JSX készítő funkcióban, ahol ki szeretném nyerni az infót a kontextusból kell használnom ebbe kell beágyazni a felhasználó komponenseket/TAG-eket. ÍGy:

function JSXkeszito(){

<XYcontext.Consumer>

{ ( ctx )=>{ return (<AholKellAzAdat hozza-feres={ctx.adattartalom} />) } }

</XYcontext.Consumer> };

-Itt a felhasználó komponenst körbevettem a kontextus.Consumer jsx-ével. Ennek első eleme (= child-ja) egy arrow funkció, amibe belemegy a kontextus tartalma (itt: ctx, de ez egy belső név csak) és visszadja mindazon komponensek/TAG-ek html kódját, akinek szolgáltatni akarjuk az adatot. Itt a hozza-feres={ctx.adattartalom} azt mutatja, hogy mostmár lehet belső név.a\_kontextus\_objektum\_címkeneve alapon használni a kontextus tartalmát.

II. -> "Fogyasztás" useContext() HOOK-al:

Most importálni kell a useContext()-et is így:

import React, {useContext} from 'react';

-és mellé természetsen a kontextust tartalmazó .js funkcióját is:

import XYcontext from './XYcontext';

Most a JSX készítő funkció elején (amiben fogyasztani akarom a kontextus tartalmát) meghívom a useContext-et így:

function FelhasznaloJSX(){

const ctx = useContext(XYcontext);

return <consumerHTMLvagyJSX hozza-feres={ctx.adattartalom} />

}

- Itt az előző változathoz hasonlóan létrehozok egy ctx belső nevű konstant, amibe a useContext() segítségével beletöltöm a provider value értékében megadott objektumot (itt: {adattartalom: 'bármi lehet'}). Ezt követően ctx.adattartalom elérési úttal hozzáférek a szolgáltatott infóhoz.

Handler funkciókat is lehet tárolni a provider value objektumában, ekkor csak a korábban definiált funkció nevét kell megadni, de nem szabad meghívni() is. ÍGy:

<XYcontext.Provider value={ {handlerFunction: funkcioNeve} }>

Sokszor a szerkesztést megkönnyíti, ha már a Context készítő JSX-ben a createContext-be beadjuk a változók nevét, és ha a változó egy funkció, akkor pedig egy üres arrow function-t így:

import React from 'react';

const XYcontext = React.createContext(szovegErtek: 'string', booleanErtek: false,

funkcioErtek: ()=>{} );

export default XYcontext;

Ez arra jó, hogy VSCode már az alkalmazás megírásakor fel tudja ajánlani számunkra a tárolt változókat, így kisebb eséllyel zavarodunk bele a fájlok közötti váltogatásba.

Hasznos lehet minden kapcsolódó logikát és state változó definíciót átemelni abba a fájlba, ahol a kontextus provider készül, mert akkor könnyebb nyomon követni a state változókat, és azok manipulációját. Lehet egy külön ContextProvider JSX-et is csinálni és abba beágyazni rendereléskor a teljes összetartozó kompnensgyűjteményt.

A felhasználás korlátai:

- Nem érdemes useContext()-et tenni általános felhasználású (dizájn egységesítő) elemekbe, mint például egy egyedi gomb (custom button). Ezekbe továbbra is props-okon keresztül kell beküldeni az infót és lifting the state up-al kimeríteni onnan.

- Nem alkalmas gyakori state váltásra (másodpercenkénti pl.)

FORWARD REFS

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

forrás:

https://www.youtube.com/watch?v=0YTYqg0ETx8

Forward ref arra való, hogy a useRef() által egy parent komponensben megjelölt referencia Node-ot lepasszolhassunk a child komponensnek.

Ekkor a child komponensben kell a React.forwardRef()-et használni így:

import React from 'react'; - nem kell külön odaírni, de be kell importálni.

function ChildJSX(props, ref){ return <HTMLtag ref={ref}>}; - itt a szokásos propson kívül kap még egy argumentumot, ami a lepasszolt referencia lesz. Vissza pedig olyan JSX kódot ad, amiben meghatározzuk a referencia helyét (ref={ref}), de a useRef-et ehhez a parent komponensben fogjuk szolgáltatni. Végül meg is hívjuk a forwardRefet így:

export default React.forwardRef(ChildJSX); - fontos, hogy ez megtörténjen valamikor, de ha nem az exportálásnál, akkor már előtte a definiálás közben is megtehettem volna így:

const ChildJSX = React.forwardRef( (props, ref)=>{ return <HTMLtag ref={ref}>} );

A parent JSX-ében persze kell használni a useRef-et.

useImperativeHandle()

.:.:.:.:.:.:.:.:.:.:.

WebDevSimplified alapján:

Felül lehet írni a useRef által küldött infót vele akármi más objektumra. Ritkán használják, de ey form-ban input elemiere pl. a .focus() beállítására jól használható.

Használata:

Be kell importálni így:

import React, {useImperativeHandle} from 'react';

A JSX készítő funkcióban meg kell hívni:

function ChildJSX (props, ref){

useImperativeHandle(ref, ()=>{return { uj: 'érték', friss: 'dolog'} }, [dependency] )

}

- itt a 3 dolog amit a useImperativeHandle fogad:

1. a felülírandó referencia,

2. egy arrow funkció, ami visszaadja a ref új értékét, ami egy objektum lesz

3. opcionálisan dependecy-k, ugyanúgy, ahogy a useEffect-nél van. Ha nem adunk meg, akkor minden renderelésnél frissíti csak az értékét.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

REACT HOOK Szabályok:

0. Mindig importáljuk azt a hook-ot, amit használni akarunk

pl.: import React, {useState} from 'react';

1. React hook-ot csak React funkcióban lehet meghívni.

Ami lehet:

a) JSX készítő funkció, VAGY

b) Custom hook-ban (lásd később)

3. Mindig a legmagasabb szinten kell meghívni a hook-okat (JSX készítő) a .js file-ban! Azaz nem szabad beágyazott funkcióban meghívni, vagy block-ot képző kifejezésben (pl. if() statement)

+ 1. useEffect() használatakor minden változót tegyél be dependecy-nek, amit csak tartalmaz az a funkció amibe meghívod a useEffect-et. - Erre fog figyelmeztetni a VSCode piros aláhúzással, és meg is nevezi amit szerinte kihagytunk.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Felsorolásokban meghívott funkciók!

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Valamilyen oknál fogva, ha elsorolásokban akarom ugyanazt a Handler funkciót meghívni akkor a .bind(null, szükségesBemenet) módon kell. Nem világos miért, ugyanis a .bind első paramétere fogadna egy objektumot, ami felülírná a funkció belős this-ét, de erre null-t kap. Lehet, hogy azért, mert így egy másolata jön létre a funkciónak a szükségesBemenet closure-el? Vagy így lehet bemenetet biztosítani egy funkciónak anélkül, hogy lefuttatnánk (amit rendereléskor megtesz a React, ezért amúgy csak a funkció nevét adhatnánk be fn(argumentum) nélkül. Minden esetre így kell:

<JSXListItem onClick={onClikckHandler.bind(null,egyediBemenet)} />