

## 1-130. lessons:

### Eddig megtanult React Hook-ok:

useState:

<https://hu.reactjs.org/docs/hooks-reference.html#usestate>

Egy komponens állapotait a Stat-ekben tudjuk tárolni, mivel, ha simán megfogalmazunk változókat, azok nem updatelődnek a komponens újbóli lefutásakor. Importálnunk kell a 'useState' hookot. Ez a hook két részre bontható. Az egyenlőség bal oldalán a egy tömb van két elemmel. Az első a legutóbbi verziója a változónak/objektumnak. Ezzel tudunk rá hivatkozni. A második elem pedig az a függvény, ami beállítja nekünk ezt a paramétert, ha meghívjuk. A jobb oldalon a useState() belsejében kezdő állapotot is fel tudunk venni.

```
1 const [selectedYear, onSelect] = useState("2020");
2
3 const onSelectYear = (selectedYearParam) => {
4   onSelect(selectedYearParam);
5 };
6
```

useEffect:

<https://hu.reactjs.org/docs/hooks-reference.html#useeffect>

A useEffect-et akkor használjuk, amikor valami mellékhatást akarunk reprezentálni. Két része van ennek, az első a végrehajtandó feladat, a másik pedig egy property tömb. Ebben a tömbben azokat a propertyket vesszük fel, amely elem változását követően a mellékhatást le szeretnénk futtatni. Alapból, ha nincs semmi megadva második paraméternek, akkor ahányszor rerendering történik meghívódik a sideEffect, ha egy üres tömb [] van átadva, akkor csak az ELSŐ komponens renderkor fut le, ha pedig a propertyket is megadjuk, a propertyk változása esetén is újra és újra lefut a side effect. A kurzus során az input kezelésnél használtuk. Az összes gomblenyomást logoltuk egy state-be, mellék hatásként pedig, ha ezek a statek változtak, mindig validáltunk. A kiváltó property maga a state volt, a validáció pedig arra a statere vonatkozott. A validáció eredményt pedig egy újabb stat-be mentettük. A példában a bejelentkezés után a localStorage-be mentett elemet vizsgáltuk, és ez alapján állítottunk be másik elemet. Itt a komponens első rendere során futtatjuk le az effektet.

```

1  useEffect(() => {
2    const storedUserLoggedIn = localStorage.getItem("isLoggedIn");
3
4    if (storedUserLoggedIn === "1") {
5      setIsLoggedIn(true);
6    }
7  }, []);

```

useRef:

<https://hu.reactjs.org/docs/hooks-reference.html#useref>

A Referenciát egy adott komponens elérésére lehet használni. Ennek a komponensnek át kell adnunk ezt a referenciát, majd a myRef.current -> metódussal elérhetjük ezt az elemet a DOM-ban. Ez a referencia ezen túl ehhez az adott komponenshez fog csatlakozni.

```

1  const InputHandler = () => {
2    const usernameRef = useRef();
3
4    const username = usernameRef.current.value;
5
6    return (
7      <Card className={styles.input}>
8        <form onSubmit={submitClick}>
9          <label htmlFor="username">Username</label>
10         <input id="username" type="text" ref={usernameRef} />
11        </form>
12      </Card>
13    );
14  };

```

A példában a kurzort a gombnyomás után az inputmezőre tudjuk irányítani. Pl: egy inputba beírt értéket a myRef.current.value-val érek el. A two way binding helyett is használjuk, ami egy input érték lekérését és az érték visszaadását az input value értékének jelenti.

useReducer:

<https://hu.reactjs.org/docs/hooks-reference.html#usereducer>

A Reducerek használatát akkor vezetjük be, ha már a statek sokasodni kezdenek, és egy adott propertyt már több state is figyel / változtat. A reducer egy szteroidos statenek is megfeleltethető. Sokkal több funkciót tud csinálni, viszont erre nincs

```
1  const emailReducer = (state, action) => {
2    if (action.type === "EMAIL") {
3      return { value: action.val, isValid: action.val.includes("@") };
4    }
5
6    if (action.type === "EMAIL_BLUR") {
7      return { value: state.value, isValid: state.value.includes("@") };
8    }
9    return { value: "", isValid: false };
10 };
11
12 const Login = () => {
13   const [emailState, dispatchEmail] = useReducer(emailReducer, {
14     value: "",
15     isValid: null,
16   });
17
18   const emailChangeHandler = (event) => {
19     dispatchEmail({ type: "EMAIL", val: event.target.value });
20   };
21
22   const validateEmailHandler = () => {
23     dispatchEmail({ type: "EMAIL_BLUR" });
24   };
25
26   return (
27     <Card className={classes.login}>
28       <form onSubmit={handleSubmit}>
29         <Input
30           ref={emailInputRef}
31           id="email"
32           type="email"
33           label="Email"
34           isValid={emailState.isValid}
35           value={emailState.value}
36           onChange={emailChangeHandler}
37           onBlur={validateEmailHandler}
38         />
39       </form>
40     </Card>
41   );
42 };
```

mindig szükségünk. Itt a state szintén tud értékeket, illetve objektumot is hordozni, viszont a felépítése más. A []-en belül a state és egy dispatch függvény foglal helyet, ez a dispatch ha meghívódik, triggerelni fogja a useReducer() első paraméterét, ami egy függvény lesz. A második paramétere egy inicializáló érték, a 3. pedig opcionálisan egy inicializáló függvény tud lenni. Dispatch során egy objektumot adunk át, aminek van egy 'type' propertyje, ezzel tudjuk azonosítani tökéletesen a célunkat. A reducer function a statet tartalmazza és egy actiont. Az action-on keresztül érjük el ezt a type propertyt, amit tudunk ellenőrizni, és ezután eldönteni, hogy mit csináljunk a stattel.

useContext:

<https://hu.reactjs.org/docs/hooks-reference.html#usecontext>

A Context abban segít nekünk, hogy elkerüljük a props chaint, ami a nagyobb projecteknél jelenik meg. Ez azt jelenti, hogy olyan komponenseken is keresztül vezetjük a stateket, propsok segítségével, akik nem is fogják használni. Létre tudunk hozni egy saját file-t, ami a context lesz. Ezt vagy közvetben tudjuk használni, vagyis lesz egy provider azaz szolgáltató, és egy consumer vagyis fogyasztó komponens. A providerben változtatjuk meg az értéket, a consumer komponensben pedig felhasználjuk. Tökéletesen meg tudjuk kerülni a state lifting up mechanizmust. Context fileban a createContext() metódust kell használnunk, ami magába foglalja azt az objektum csomagot, amibe azok a statek helyezkednek el, amiket el akarunk juttatni vagy nagyon fel, vagy nagyon le.

useImperativeHandler:

<https://hu.reactjs.org/docs/hooks-reference.html#useimperativehandle>

Amikor js függvényt akarunk használni, mint pl a focus(), ezeket megtehetjük az alap beépített elemeken, de a saját komponenseinken nem. Ahhoz, hogy egy konvertálást végezzünk, mind a két egymáshoz kapcsolódó komponensben definiálnunk kell egy ref-et. Ahol a beépített elem akarjuk használni a metódust, ott ez a useImperativehandlert használjuk, és ebben a komponensben a React.forwardRef-et is használjuk. Ezzel már nem csak egy props paramétert kapunk, hanem egy ref-et is, amivel tovább tudunk dolgozni. Ritkán használatos, de sokat tud segíteni.

```
1 useImperativeHandle(ref, createHandle, [deps]);
2
3 function FancyInput(props, ref) {
4   const inputRef = useRef();
5   useImperativeHandle(ref, () => ({
6     focus: () => {
7       inputRef.current.focus();
8     },
9   }));
10  return <input ref={inputRef} />;
11 }
12 FancyInput = forwardRef(FancyInput);
13
```

## useCallback:

Amikor a `React.memo()` metódust használjuk, akkor primitív értékek helyett át tudunk adni objektumot is, pl: egy függvény. A js-ben két objektum, ha ugyan úgy is néz ki, a kiértékelés után sosem egyezik meg, mert más memóriacímre fog mutatni az új elem. Így a meghívott `React.memo()` metódusunk azt értékelné, hogy a props megváltozott és újra kiértékelődik. Ennek elkerülése érdekében használjuk a `useCallback()` függvényt, aminek az első paramétere maga a függvény, amit meg akarunk jegyeztetni a függvénnyel, a második paramétere pedig egy dependency lista. Mivel ha ezt a callback-et használjuk, akkor a függvény scope-ban lévő összes adat adott értékét elmenit. Viszont van érték, aminek figyelni kéne az értékét, mert változhat. Ha van ilyen érték akkor azt hozzáadjuk a dependency listához. Ha ezt megcsináljuk, akkor maga a függvény elmentésre kerül, a dependency listában lévő elem / elemek mindig be tudnak frissülni, és így nem értékelődik ki újra az adott komponens, aki propsként egy függvény pointert kap.

```
1 function App() {
2   const [listTitle, setListTitle] = useState("My List");
3   const [isAllowed, setIsAllowed] = useState(false);
4
5   const changeTitleHandler = useCallback(() => {
6     if (isAllowed) {
7       setListTitle("New Title");
8     }
9   }, [isAllowed]);
10
11   return (
12     <div className="app">
13       <Button onClick={changeTitleHandler}>Change List Title</Button>
14     </div>
15   );
16 }
17
18 export default App;
```

## useMemo:

A `useMemo()` metódust akkor használjuk, amikor egy objektumot, pl: egy tömböt akarunk elmenteni. Ez a hook ugyan azt csinálja kb mint a `useCallback()`, csak azt függvényekre használjuk. Itt ugyan úgy két paraméter van. Az első egy arrow function, aminek a visszatérési értéke az objektum, amit menteni akarunk, a másik pedig a dependency list. Ezt a nagyon teljesítmény igényes feladatoknál használjuk, mint pl a `sort()`. Ha nem szükséges, és nem változik a lista, akkor ne futtassa mindig újra ezt a metódust, amikor csak befrissül a szülő komponens. Viszont ehhez két dolog kell. A gyerek komponensben az objektumot fogadó, teljesítmény igényes metódust beletesszük a `useMemo()`-ba, a dependency listához hozzáadjuk az objektumot. Így mivel ez egy referencia érték, ezért mindig újra lefutna a metódus, a memória pointer változás miatt. Viszont erre a megoldás az az, hogy mielőtt

átadjuk az objektumot a gyerek komponensnek, előre ott is használjuk a useMemo() hookot, üres dependency listával.

```
1 //Parent component
2 const listItems = useMemo(() => [5, 3, 1, 10, 9], []);
3
4 return (
5   <div className="app">
6     <DemoList title={listTitle} items={listItems} />
7     <Button onClick={changeTitleHandler}>Change List Title</Button>
8   </div>
9 );
```

```
1 // Child component
2 const { items } = props;
3
4 const sortedList = useMemo(() => {
5   console.log('Items sorted');
6   return items.sort((a, b) => a - b);
7 }, [items]);
8 console.log('DemoList RUNNING');
```

## Eddig megtanult egyéb elemek:

### Props:

A Props arra szolgál, hogy a komponensek között tudjunk adatokat átküldeni. Ha ezt akarjuk használni, akkor a komponens elkészítésekor használnunk kell a 'props' kulcsszót. Miután ezzel megvagyunk a szülő komponensbe, ahol beimportáltuk a gyerek komponens-t és meg is hívjuk, akkor abban az elemben tudjuk az adatokat továbbítani. Meg kell adni egy nevet, amivel hivatkozni tudunk majd rá, és ennek a névvel ellátott elemnek pedig értéket is kell adnunk, ezután majd a gyerek komponensben tudunk rá hivatkozni.

```
1 <ExpensesList items={filteredArray}/>
```

```
1 const ExpensesList = (props) => {
2   return (
3     <ul className="expenses-list">
4       {props.items.map((item) => (
5         <ExpenseItem
6           key={item.id}
7           title={item.title}
8           amount={item.amount}
9           date={item.date}
10          />
11       )}}
12     </ul>
13   );
14 };
```

### Fragment:

A fragmentek előtt tanultuk, hogy mindig kell legyen egy gyökér elem a JSX-es return-nél, mivel így tudja le renderelni az adott komponenst a DOM. Ezt simán lehetett <div>-vel, viszont az is erőforrást igényel, ha sok, úgymond haszontalan html elementet kell renderelni. Ehelyett készítettünk egy wrapping, azaz csomagoló komponens, ami visszatér a {props.children}-nel. Viszont ehelyett tudjuk használni a Fragment-et ami erre van kitalálva. Vagy importáljuk a Fragmentet, vagy React.Fragment-tel hivatkozunk rá.

```
1 return (
2   <Fragment>
3     <InputHandler dataHandler={dataHandlerFunc} />
4     <ListedItems data={datas} />
5   </Fragment>
6 );
```

Portal:

Általában, ha készítünk egy komponenst, akkor az a megfelelő helyre fog a DOM-ban legenerálódni. Viszont vannak olyan esetek, amikor nem akarjuk, hogy külön akarjuk választani az oldal részeit, és ez az új komponenst. A kurzuson a felugró Error ablakkal csináltuk meg ezt. Nem akartuk, hogy az adott oldal tényleges részei között jelenjen meg. Ezért kijelöltünk neki egy helyet, ahová el szeretnénk küldeni, a renderelést követően.

```
1 <body>
2   <noscript>You need to enable JavaScript to run this app.</noscript>
3   <div id="backdrop-root"></div>
4   <div id="overlay-root"></div>
5   <div id="root"></div>
```

Majd a komponensből átirányításra kerül ez az egész elem, és a komponens fában máshol fog elkészülni.

```
1 import React from "react";
2
3 import Card from "./Card";
4 import Button from "./Button";
5 import styles from "./ErrorModal.module.css";
6 import ReactDOM from "react-dom";
7
8 const Backdrop = (props) => {
9   return <div className={styles.backdrop} onClick={props.errorHandler} />;
10 };
11
12 const ModalOverlay = (props) => {
13   return (
14     <Card className={styles.modal}>
15       <header className={styles.header}>
16         <h2>{props.title}</h2>
17       </header>
18       <div className={styles.content}>
19         <p>{props.message}</p>
20       </div>
21       <footer className={styles.actions}>
22         <Button onClick={props.errorHandler}>Okay</Button>
23       </footer>
24     </Card>
25   );
26 };
27
28 const ErrorModal = (props) => {
29   return (
30     <React.Fragment>
31       {ReactDOM.createPortal(
32         <Backdrop errorHandler={props.errorHandler} />,
33         document.getElementById("backdrop-root")
34       )}
35       {ReactDOM.createPortal(
36         <ModalOverlay title={props.title} message={props.message} errorHandler={props.errorHandler} />,
37         document.getElementById("overlay-root")
38       )}
39     </React.Fragment>
40   );
41 };
42
43 export default ErrorModal;
```



Key:

Ha egy komponensből pl: egy map() segítségével kilistázunk adatokat --> így a komponenst sokszorozítjuk, mindig hozzá kell adni a "key" tagot, mivel ez fogja a felsorolt itemeket megkülönböztetni egymástól. Ha ezt nem tesszük ki, akkor a console-on sírás lesz, a "key" tag hiányára támaszkodva.

```
1
2  const ListedItems = (props) => {
3    return (
4      <Card className={styles.users}>
5        <ul>
6          {props.data.map((item) => (
7            <li key={item.id}>
8              {item.username} ({item.age} year old)
9            </li>
10          ))}
11        </ul>
12      </Card>
13    );
14  };
```

Smart CSS:

Ha nagy a projectünk, és a css fájlokban el akarjuk kerülni a névütközést, vagy az esetleges classok ütközését, akkor használjunk felokosított CSS fileokat. A CSS fájl elnevezését így készítjük el:

Így egy egyedi azonosító generálódik minden egyes css fájl tagjai mellé, amit meg tudsz nézni a consoleban. Ezzel elkerülhető az esetleges névütközés.

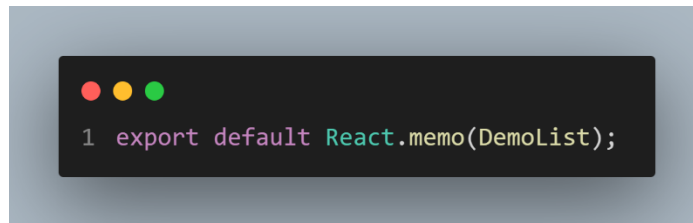
```
1  import React from "react";
2
3  import Card from "../UI/Card";
4  import styles from "./ListedItems.module.css";
5
6  const ListedItems = (props) => {
7    return (
8      <Card className={styles.users}>
9        <ul>
10          {props.data.map((item) => (
11            <li key={item.id}>
12              {item.username} ({item.age} year old)
13            </li>
14          ))}
15        </ul>
16      </Card>
17    );
18  };
19
20  export default ListedItems;
```

## Controlled vs uncontrolled components:

Amikor `useRef()` van használva egy komponensen belül, akkor azt a komponenst uncontrolled-nek tekintjük. Általában Input field-es helyeken használjuk, hogy direct módon elérjünk egy html elemet.

## React.memo():

Amikor egy `state/props/context` érték frissül, akkor a használó komponensek frissülnek, és újra kiértékelődnek. Az adott komponens összes gyerekkomponense is szintén újra kiértékelődik. Abban az esetben, ha nem adunk át props-ot, vagy ha át is adunk a gyereknek, de tudjuk, hogy nem drissül olyan gyakran az a props, mint a szülő komponens, használni tudjuk a `React.memo()` metódust. Ez segít nekünk abban az esetben, ha a props egy primitív érték, és nem változik, akkor ez a gyerekkomponens nem értékelődik ki újra feleslegesen. → csak akkor értékelődik ki újra a gyerek komponens, ha a props változott. Mivel az referencia értékek minden újra kiértékelés során másak lesznek, a `useCallback()` segít függvények esetén, a `useMemo()` pedig objektumok esetén, hogy helyesen működjön a `React.memo()`.

A screenshot of a code editor with a dark background and light-colored text. It shows a single line of code: `1 export default React.memo(DemoList);`. The text is color-coded: `1` is light blue, `export` is light green, `default` is light green, `React.memo` is light green, and `(DemoList);` is light blue. Above the code, there are three small colored circles (red, yellow, green) in a row.

## Jó ha tudod:

### Virtual DOM & real Dom:

A real Dom a böngészőbe beépített mechanizmus, ami lerendereli nekünk az oldal kinézetét a felhasználó számára a reactból kapott komponensek által. A komponensek dolga, hogy értékeket tároljanak/hanszáljanak/továbbítsanak és felépítsenek a component tree-t. A real DOM nem értékelődik ki újra minden egyes komponens újra kiértékelődés esetén, hanem van egy úgynevezett Virtual DOM, ami segít ebben. Ez a Virtual DOM megkapja a komponenseket, elkészíti az új komponensgráfot, megtartja a régit, és összehasonlítja a kettőt, hogy történt-e valami változás a két komponens újakiértékelődés között. Ez csak a HTML! elemekre vonatkozik, az értékekre nem. Ha talál különbséget, csak akkor szól a realDOM-nak hogy értékelődj te is ki újra. Ilyen pl egy `<p>` elem bekerülése a DOM tree-be.