React Udemy

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 tőmb 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.

```
import React, { useState } from "react";
```

```
const [selectedYear, onSelect] = useState("2020");

const onSelectYear = (selectedYearParam) => {
  onSelect(selectedYearParam);
};
```

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 bejelntkezés után a localstorage-be mentett elemet vizsgáltuk, és ez alapján állítottunk be másik elemet. Itt a komponens első rendere során futtatjuk le az effektet.

```
const [isLoggedIn, setIsLoggedIn] = useState(false);

useEffect(() => {
  const storedUserLoggedIn = localStorage.getItem("isLoggedIn");

  if (storedUserLoggedIn === "1") {
    setIsLoggedIn(true);
  }
}, []);
```

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 -> metóódussal

```
function TextInputWithFocusButton() {
  const inputEl = useRef(null);
  const onButtonClick = () => {
    // `current` points to the mounted text input element
    inputEl.current.focus();
  };
  return (
    <>
      <input ref={inputEl} type="text" />
      <button onClick={onButtonClick}>Focus the input</button>
    </>
  );
}
```

elérhetjük ezt az elemet a DOM-ban. Ez a referencia ezen túl ehhez az adott komponenshez fog csatlakozni. A példában a kurzort a gombnyomás után az inputmezőre tudjuk irányítani.

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 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.

```
const initialState = {count: 0};

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

function Counter() {
  const [state, dispatch] = useReducer(reducer, initialState);
  return (
    <>
      Count: {state.count}
      <button onClick={() => dispatch({type: 'decrement'})}>-</button>
      <button onClick={() => dispatch({type: 'increment'})}>+</button>
    </>
  );
}
```

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 filet, ami a context lesz. Ezt vagy direktben 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.

```
const themes = {
  light: {
    foreground: "#000000",
    background: "#eeeeee"
  },
  dark: {
    foreground: "#ffffff",
    background: "#222222"
  }
};

const ThemeContext = React.createContext(themes.light);

function App() {
  return (
    <ThemeContext.Provider value={themes.dark}>
      <Toolbar />
    </ThemeContext.Provider>
  );
}

function Toolbar(props) {
  return (
    <div>
      <ThemedButton />
    </div>
  );
}

function ThemedButton() {
  const theme = useContext(ThemeContext);
  return (
    <button style={{ background: theme.background, color: theme.foreground }}>
      I am styled by theme context!
    </button>
  );
}
```

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 elemen 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.

```
useImperativeHandle(ref, createHandle, [deps])
```

```
function FancyInput(props, ref) {
  const inputRef = useRef();
  useImperativeHandle(ref, () => ({
    focus: () => {
      inputRef.current.focus();
    }
  }));
  return <input ref={inputRef} ... />;
}
FancyInput = forwardRef(FancyInput);
```

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.

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

```
const ExpensesList = (props) => {
  return (
    <ul className="expenses-list">
      {props.items.map((item) => (
        <ExpenseItem
          key={item.id}
          title={item.title}
          amount={item.amount}
          date={item.date}
        />
      ))}
    </ul>
  );
};
```

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 -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.

```
return (
  <Fragment>
    <InputHandler dataHandler={dataHandlerFunc} />
    <ListedItems data={datas} />
  </Fragment>
);
```

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.

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

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

```
<React.Fragment>
  {ReactDOM.createPortal(
    <Backdrop errorHandler={props.errorHandler} />,
    document.getElementById("backdrop-root")
  )}
  {ReactDOM.createPortal(
    <ModalOverlay title={props.title} message={props
    document.getElementById("overlay-root")
  )}
</React.Fragment>
```

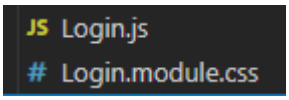
Key:

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

```
{props.items.map(goal => (
  <CourseGoalItem
    key={goal.id}
    id={goal.id}
    onDelete={props.onDeleteItem}
  >
    {goal.text}
  </CourseGoalItem>
))}
```

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 elnevézését így készítjül el:




```
import classes from "./Input.module.css";
```

```
<div
  className={classes.control}
>
```

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