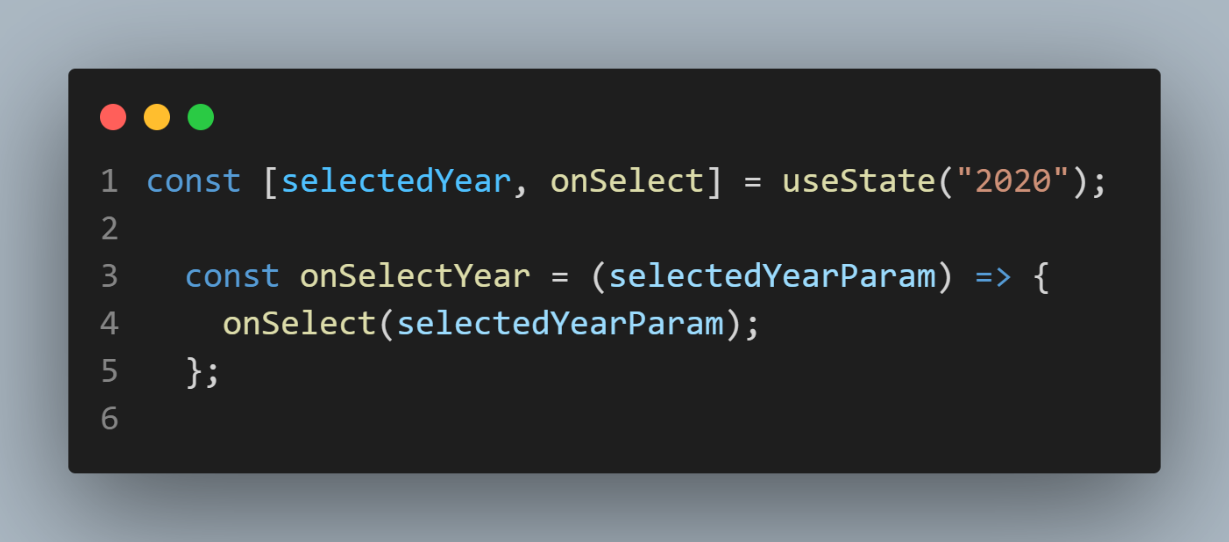
React Udemy

**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.

useEffect:

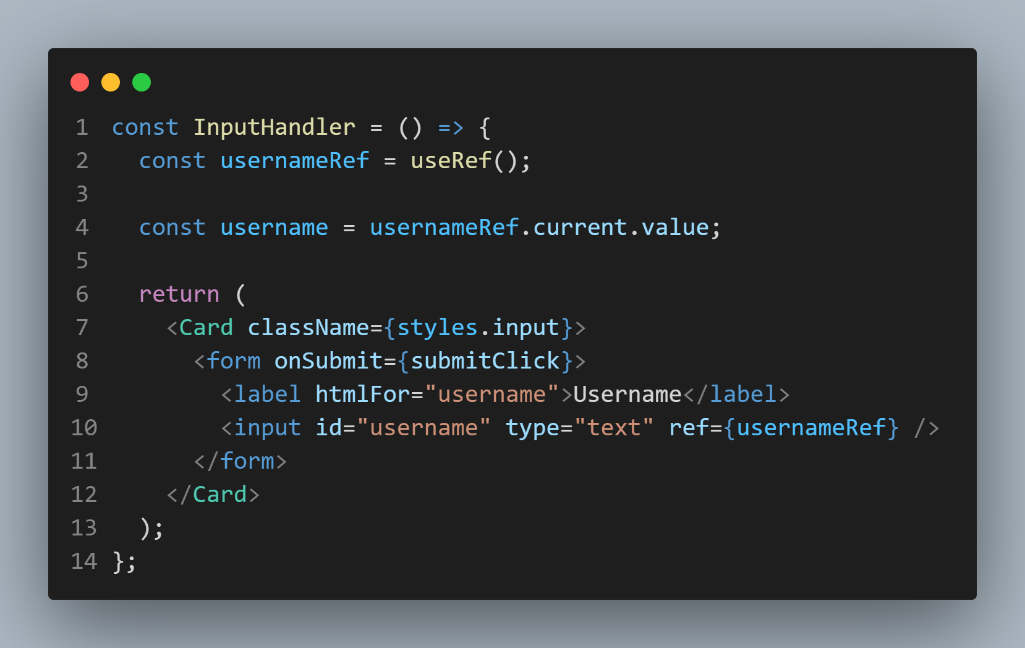
https://hu.reactjs.org/docs/hooks-reference.html#useeffect

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.



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 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. 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 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.

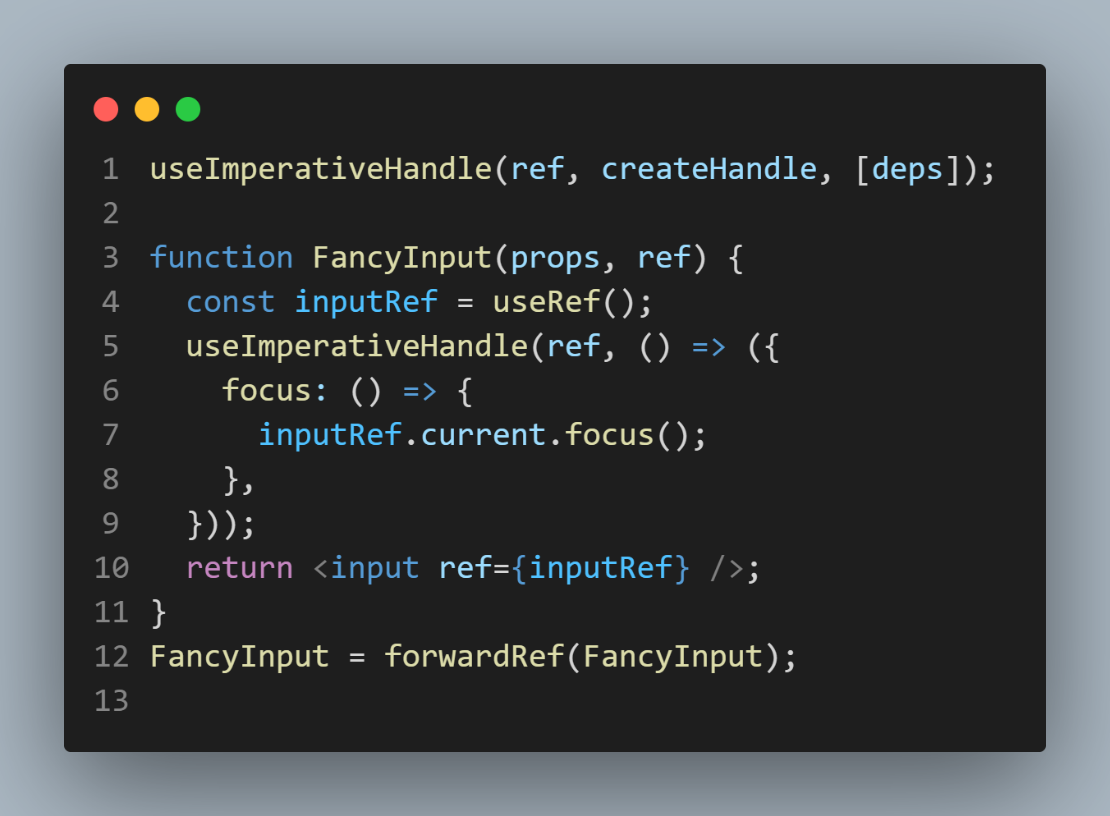
useContext:

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 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.

useImperativeHandler:

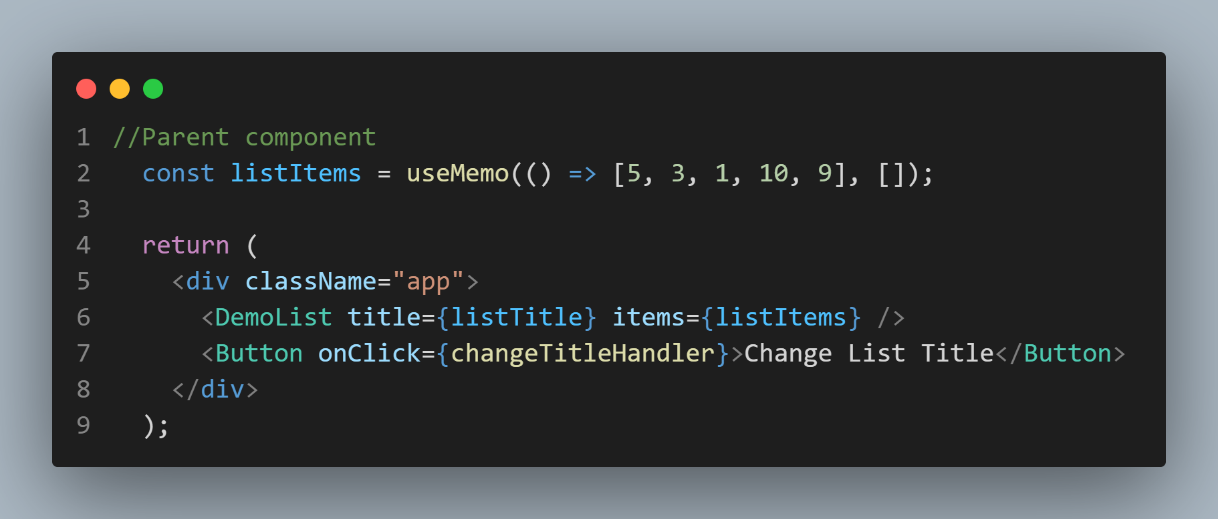
https://hu.reactjs.org/docs/hooks-reference.html#useimperativehandle

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.

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 jegyzetetni 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 dependecy 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.

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 obejktumot 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 obejktumot a gyerek komponensnek, előrre ott is használjuk a useMemo() ****hookot, üres dependency listával.

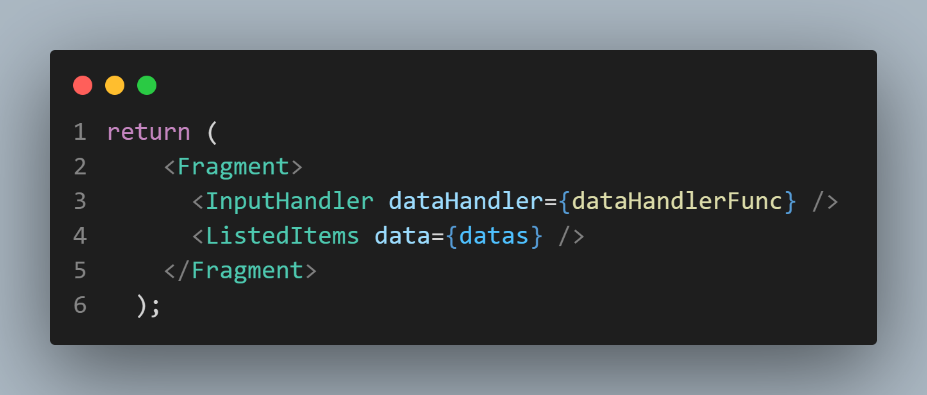
**Eddig megtanult egyéb elemek:**

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.



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.

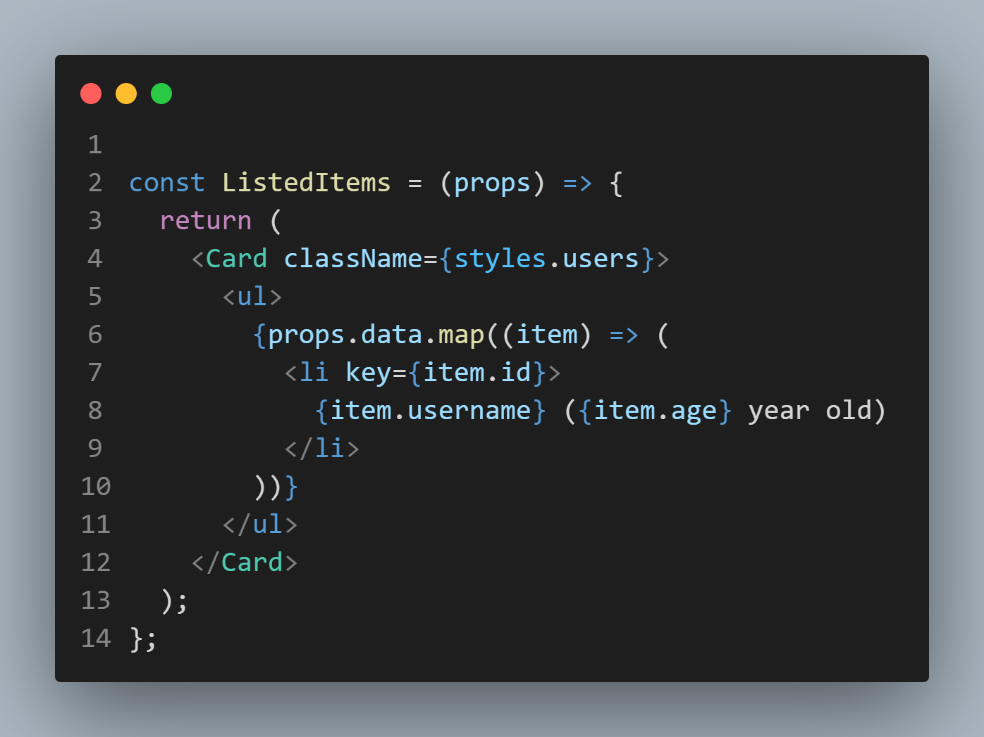
Portal:

 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.

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



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.

Smart CSS:

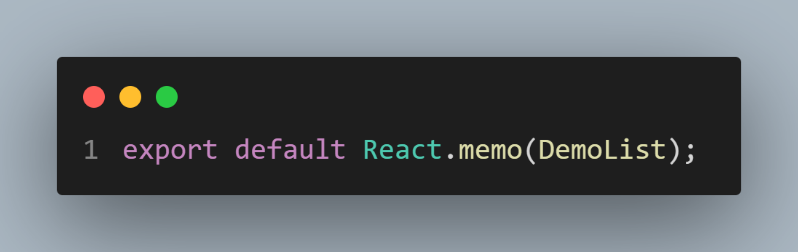
Ha nagy a projectünk, és a css fájlokban el akarjuk kerülni a névütközést, vagy az esetleges classok ütközését, akkor használjunk felokosított CSS fileokat. A CSS fájl elnevézését így készítjül el:

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

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 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 estebne, 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().

**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 Virtaul 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 kmponens újrakiértékelődés 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.