Multiplatform szoftverfejlesztés

React

React

Hello, World

Hello, World

```
var Greeter = function (p) {
    return React.createElement("h1", null, "Hello, ",p.name,"!");
};
```

- Létrehozunk egy komponenst (Greeter)
  - Ami egy függvény
  - Kap egy p paramétert
    - Amiben egy name tulajdonság van
  - Meghívja a React.createElement függvényt
    - Létrehoz egy h1 objektumot (JS objektum, nem HTML)
    - Hello+p.name+"!" tartalommal
  - Visszaadja a kapott React fát

Hello, World

```
var Greeter = function (p) {
    return React.createElement("h1", null, "Hello, ",p.name,"!");
};
ReactDOM.render(React.createElement(Greeter, { name: "Leo" }),
    document.body);
```

- Meghívjuk a ReactDOM.render függvényt
  - Paraméterek: tartalom és konténer
  - A tartalmat a React.createElement állítja elő
  - A mi komponensünket használva
    - És átadva neki a paramétert

JSX – babeljs

- Egyszerűsített szintaktika
- A fordító átalakítja a kódot az előző formára

```
let Greeter =  function(p) {
    return <h1>Hello, { p.name }!</h1>;
}
ReactDOM.render( <Greeter name="Leo" />, document.body );
```

```
var Greeter = function (p) {
    return React.createElement("h1", null, "Hello, ",p.name,"!");
};
ReactDOM.render(React.createElement(Greeter, { name: "Leo" }),
    document.body);
```

TSX – TypeScript

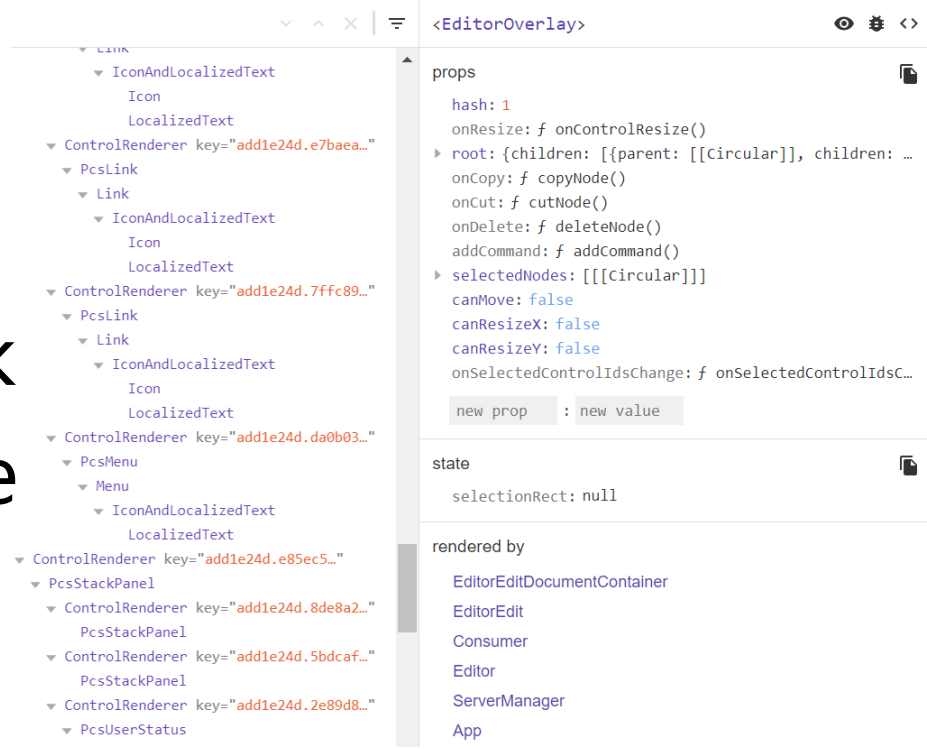
- Típusos JSX

```
let Greeter =  function(p) {
    return <h1>Hello, { p.name }!</h1>;
}
ReactDOM.render( <Greeter name="Leo" />, document.body );
```

```
let Greeter = ( p: {name: string} ) => <h1>Hello, {p.name}!</h1>;
ReactDOM.render( <Greeter name="Leo" />, document.body );
```

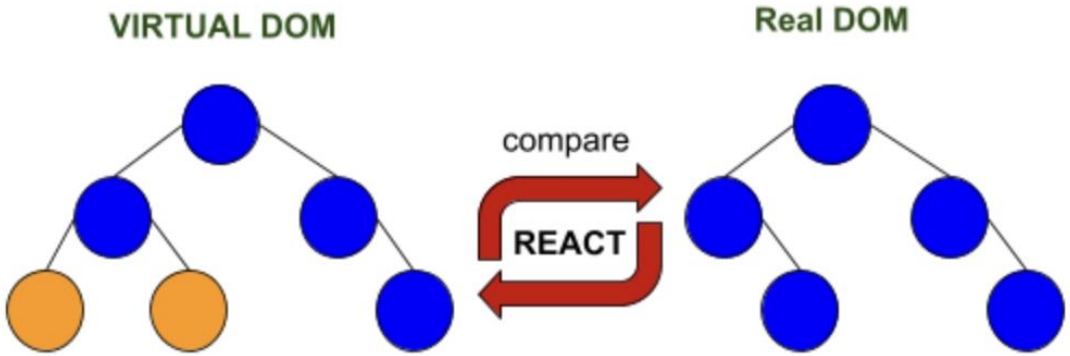
- Kapcsos zárójelek között saját kód
  - Csak kifejezést lehet beírni, ami visszaad valamit

React fa



- A létrehozott elemek nem HTML elemek
- Ugyanúgy hierarchiába vannak rendezve
  - Ez adja majd a HTML fát
- A render hívás végzi el a HTML-re konvertálást
  - Szinkronizációs folyamat, amit csak JS objektumokon végez el
    - Keresi a változásokat (tree reconciliation)
  - Az eredmény egy változás lista
    - Első futáskor az eredmény a teljes fa
    - Utána csak a különbség

React fa



- A különbségből előáll a parancslista
  - Tényleges HTML változtató parancsok
  - Ezt végrehajtja
- Lehetne vizsgálni a tényleges HTML fát is
  - Lassú
- Kézzel beleírni a HTML-be nem lehet
  - Nem észleli
  - Nem írja vissza
  - De ha változik a React fában, akkor felülírja
  - Megbízhatatlan megoldás

React

Összetett komponensek

Osztály komponens

- Osztály, mint komponens (függvény helyett)
  - Egységbe zárás
    - Összegyűjthetjük a komponenshez tartozó függvényeket
  - render függvény kötelező
  - Első típus paraméter a props típusa
  - Lehetnek belső változói

```
class GreeterC extends React.Component<{name: string}>{
    render(){
        return <h1>Hello, {this.props.name}!</h1>;
    }
}
```

Osztály komponens előnyei/hátrányai

- OO mintát jobban kövei, egyszerűbb megérteni
  - Bonyolult komponenseknél jobban látható, hogy mit történik
- Elvileg többet tud – pl. belső változók
  - Ritka az az eset, amikor szükség van erre
  - És akkor is meg lehet oldani máshogy
- Hosszabb kódot eredményez
  - Pár sorral több
- this probléma
  - Mindenhol arrow functiont kell használni, vagy bindolni (később)

Tulajdonságok (props)

- Publikus interfész
  - Elérhető kívülről
  - JSX/TSX támogatja a beállítást
    - Mintha egy sima HTML attribútum lenne
- Nem változtathatjuk belülről – paraméterként viselkedik
- Minden rajzoláskor újra megkapjuk a szülő által adott tulajdonságokat
  - Az előző rajzoláskori tulajdonságok elvesznek
  - Nem alkalmas állapot tárolására

Állapot kezelés (state)

- Belső állapot
  - Ez sok komponensnek nem lesz – állapotmentes
- Megmarad az értéke rajzolások között
  - Ezért alkalmas állapot kezelésre
- Inicializálni kell konstruktor időben

```
state = { name: "" };
```

- Típusa, amit beállítunk kezdő értéknek

Állapot kezelés (state)

```
class Counter extends React.Component<{}, {c: number}>
{
    state = {c: 0};
    inc() { this.setState( { c: this.state.c + 1 } ); }
    render() { return <p>Counter: { this.state.c }</p>; }
}
```

- Állapot kezdeti értéke `{c: 0}`, típusa `{c: number}`
- setState állítja
  - Ez egy rajzolást is kivált
  - Máshogyan nem lehet állítani az állapotot

Állapot kezelés (state) – aszinkron

- setState aszinkron
  - Nem akkor állítja be, amikor meghívjuk
  - Ez optimalizáció miatt van
    - Előbb végigmegy a teljes fán, és csak a végén állít be mindent
  - Számunkra ez nem tűnik fel
    - Kivétel, ha az állapot előző értékét felhasználjuk az új érték állításához

```
inc() {
    this.setState( { c: this.state.c + 1 } ); // nem mindig lesz jó
}
```

Állapot kezelés (state) – aszinkron

- Az aszinkron állapot állítás problémára van megoldás
  - setState tud kezelni függvényt is

```
inc() {
    this.setState( state => {c: state.c + 1} );
}
```

- Csak akkor kell használni, ha az aszinkron működés problémás lehet
  - Nem gyakori

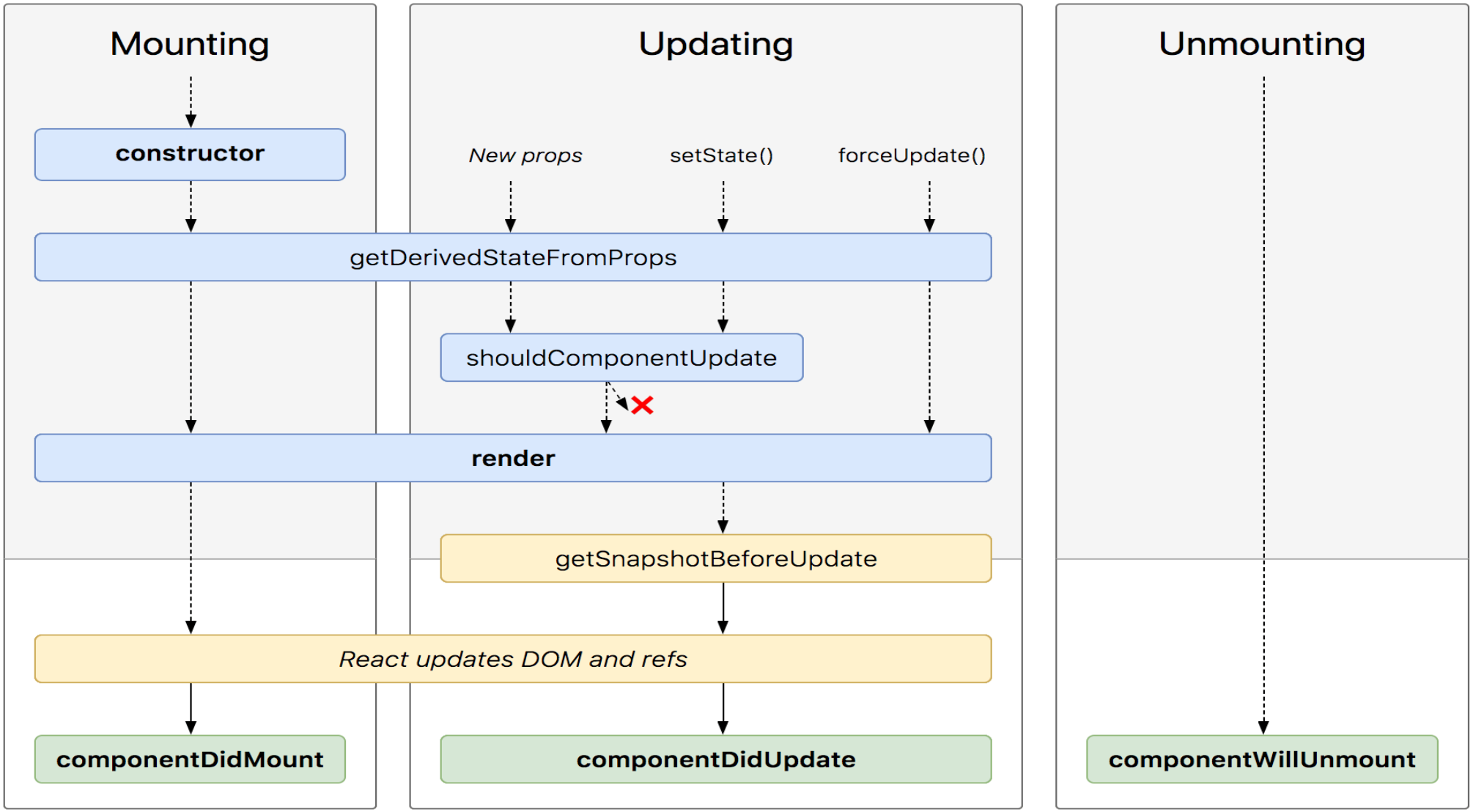
Állapot kezelés (state) – merge

- Állapot állításkor az egyes állapotok külön állíthatók

```
state = { a: 1, b: 2, c: 3 };
inc() { this.setState( { a: 10 } ); }
```

- A kód csak az `a` állapotot állítja át
  - `b` és `c` marad 2 és 3
- Csak a felső szintű állapotokra vonatkozik – shallow

Komponens életciklusa



Életciklus kezelés

- Tipikus beavatkozási helyek, amikor a keretrendszer meghívja a komponensünket
  - constructor – létrehozáskor
  - componentDidMount – első rajzolás után
  - componentDidUpdate – többi rajzolás után, ritkán használt
  - componentWillUnmount – eldobás előtt
- Létrehozhatunk saját tulajdonságokat
  - Nem a props-ban és nem a state-ben
  - Ha nincs köze a rajzoláshoz, vagy kézzel akarjuk kezelni
  - Életciklusa azonos a state-tel

Életciklus kezelés

```
class Counter extends React.Component
{
    private timer: number;
    componentDidMount()
    {
        this.timer = setInterval( () => this.forceUpdate(), 1000 );
    }
    componentWillUnmount()
    {
        clearInterval( this.timer );
    }
    render()
    {
        return <span>{ new Date().toLocaleTimeString() }</span>
    }
}
```

Kompozíció

React megoldása

render

- A felület leírása kóddal történik
  - Hibrid megoldás, nem tisztán deklaratív
  - Egyre népszerűbb, több keretrendszer megy ebben az irányban
- A kódba (render) szinte bármit beírhatunk
- A lényeg, hogy egy fát adjon vissza
  - Ami leírja a felületet
  - Továbbra sem kézzel hozzuk létre a HTML-t
- A teljes szinkronizációt a keretrendszer végzi

Egymásba ágyazás

- A komponensek egy fát alkotnak
  - Statikus gyerekek (fixen beírva)
  - Feltételes gyerekek (if-ben)
  - Generált gyerekek (tömbben)
- Fából több lehet
  - Több konténerbe is tehetünk React fát
  - Egy alkalmazás pici része is lehet React-es
    - Vagy több része egymástól függetlenül
  - Nagyon sok ne legyen
    - Például generálunk egy 1000 cellából álló táblázatot és minden cella egy React fa…

Statikus és feltételes gyerekek

- Statikus elemek esetén nincs kód a renderben
- Feltételes esetben használhatunk
  - && operátor: JS-ben a második tagot adja vissza, ha az első tag igaz
    - Hamis esetben hamis értéket ad vissza, amit React úgy értékel, hogy nincs ott semmi

```
return <header>
  <h1>Hello, Leo!</h1>
  {props.C && <Comp2/>}
</header>;
```

  - ?: operator – feltétel függően csak az egyik

Lista létrehozása

- Tömböt kell visszaadni

```
return <ul>
        { ar.map( (x,idx) => <li key={ar[idx]}><Dot/></li>) }
    </ul>;
```

- Kell key attribútum
  - Ez azonosítja az elemet
  - Innen tudja a React fa szinkronizáció
    - Melyik új elem
    - Melyik törölt
    - Melyik változott

Lista létrehozása – key

- A key egyedi kell legyen a tömbön belül
  - De nem globálisan
- Tipikusan a key egy ID az adatbázisból
- Ha nem tudunk jó key-t adni, akkor használjuk az indexet
  - Ez általában megoldja a problémát
  - Ha az elemeket átrendezzük, akkor lassú
- Új elem létrehozása, aminek még nincs ID-je
  - Adjuk neki olyat, ami amúgy nem lehetséges (pl. -1, vagy "boo")

Több gyökérelem

- A komponens több elemet is visszaadhat
  - Tömbként, vagy <Fragment> virtuális elemmel, vagy röviden <>
- Ez akkor fontos, ha egy wrapper HTML elem (pl. div) elrontaná a formázást/layoutot
  - Ha nem rontja el, akkor betehetünk gyökérnek egy div-et
  - Kerülendő – általában ne tegyünk felesleg plusz elemeket a HTML fába
- Tipikus példa a flexbox
  - Nem lehet plusz elemeket betenni, mert elromlik

Feltételes attribútumok

- Ha feltétel függően akarunk betenni egy HTML attribútumot
  - Hamis értéket adunk neki

```
autoFocus={this.props.autofocus}
```

    - Ez működik érték nélküli attribútumokra
      - Pl. disabled, required, stb.
  - Vagy használjuk a spread operátort

```
let attrs: any = {};
if ( condition )
    attrs.disabled = true;

<input { ...attrs } />
```

class, for, classList

- Az egyes attribútumok elnevezése a JS szintaktikát követi
  - Nem class=“…”, hanem className="„
  - Nem for=“…”, hanem htmlFor=“…”
  - (preact-ben nincs ez a megkötés, ott lehet class-t használni)
- Picit zavaros, mert úgy tűnik, mintha HTML-t írnánk (JSX, TSX miatt)
  - De ez átfordul kódra, ahol a class és for kulcsszavak
- Nincs classList
  - De amúgy is kódból állítjuk elő a class listát
    - Van segédkönyvtár, ha bonyolult: classcat

Komponens gyerekei

- Ha a komponensünkbe beletesznek tartalmat

```
<MyComp>
    <button>Push</button>
</MyComp>
```

- Azt a props.children-en keresztül érjük el
- Bárhogyan felhasználhatjuk segédfüggvényeken keresztül
  - React.Children.map
  - React.Children.count
  - React.Children.toArray
  - …

Öröklés

- Kerülendő (React ajánlás)
  - Az általánosabb komponensből ne származtassunk, hanem a props-on keresztük specializáljuk

Felhasználói bemenet

Esemény kezelés

- HTML elemek eseményeire feliratkozhatunk

```
render(){
    return <p>Counter: { this.state.c }<br />
        <button onClick={ () => this.inc() }>Inc</button></p>;
}
```

- Vagy arrow function, vagy bind
- Hívhatunk bármit
  - Saját függvényt
  - props-ban kapott szülő/külső függvényt
  - Globális függvényt

input és társai

- Felhasználó bemenet kiolvasása
  - Gombnyomásra (pl. elküld gomb)
  - Változásra, például validáláshoz
- Típusok
  - <input type="text"> és társai: textbox
  - <input type="file">: fájlválasztó
  - <textarea>: multiline
  - <select>: combobox
- Nem ide tartozik
  - <input type="button">, <input type="checkbox">, …

state-ben tárolt állapot

```
class TextInput extends React.Component<{}, { value: string }>
{
  state = { value: "" };
  render()
  {
    return <input type="text" value={ this.state.value }
      onChange={ e => this.setState({value: e.target.value})} />
  }
}
```

- setState hívás helyett lehet validálni, stb.

DOM-ban tárolt állapot

- Elsődleges állapot a DOM-ban van
  - El sem tároljuk a state-ben – felesleges
- Amikor szükségünk van rá, kiolvassuk
- Ehhez kell egy referencia
  - Nem triviális, mert a generált objektum elérhetetlen
  - Tudunk referenciát adni objektumokra (ref)

DOM-ban tárolt állapot

```
class TextInputU extends React.Component<{}, {}>
{
    input = React.createRef<HTMLInputElement>();
    push() { alert( this.input.current.value ) }
    render()
    {
        return <div>
            <input type="text" ref={ this.input } />
            <button onClick={ () => this.push() }>Push</button>
        </div>
    }
}
```

Hol legyen az állapot?

- <input type="file"> esetén DOM lehet csak
  - Nem lehet állítani a value-ját
- Kezdeti értéket mindkettő támogatja
  - value a state esetben
  - defaultValue a DOM esetben
  - Azért van különbség, mert a value beállítására a vezérlő írhatatlan lesz a felhasználó számára
    - onChange hívódik így is, ezért működik
    - null-t adva mégis írható lesz

Típusok

- Mit hogyan kell használni

Típus	**Érték**	**Change callback**	**Érték a callbackben**
<input type="text" />	value="string"	onChange	event.target.value
<input type="checkbox" />	checked={boolean}	onChange	event.target.checked
<input type="radio" />	checked={boolean}	onChange	event.target.checked
<textarea />	value="string"	onChange	event.target.value
<select />	value="option value"	onChange	event.target.value

Validáció

- PropTypes modul
- Típus ellenőrzés
  - Ezt TypeScript miatt automatikusan kapjuk
  - Csak a fordítás időben ellenőrizhető adatra
    - Ez szinte minden, ha nem használunk any-t
- Saját ellenőrző
  - Error-t kell visszaadni, ha hibás
- Megkövetelhető, hogy csak 1 gyerek legyen

Hooks

Osztály helyett függvény

Hook – függvény

- Nem kell osztályt írni
- Állapot useState hívással megszerezhető

```
function TextInputHook(props)
{
    let [value, setValue] = React.useState(props.def);
    return <input type="text" value={value}
        onChange={e => setValue(e.target.value)} />
}
```

- Hooks nem tud többet az osztálynál
  - Tömörebb szintaktika

Állapot kezelés

- useState visszaadja az állapotban tárolt értéket és az állapot beállító függvényt
- Állapot már nem egy objektum, hanem lista
  - Egyesével lekérdezhető, sorrend fontos
    - Például a második useState hívás a 2. állapotot adja vissza
    - Nem hagyhatunk ki useState hívást (nem lehet if-ben)

Életciklus kezelés

- Osztálynak voltak életciklus függvényei
  - Fel tudott iratkozni külső eseményekre
- useEffect az életciklus kezelő
  - Függvényt adunk át
  - Meghívja minden rajzolás után
  - Ha visszaadunk egy függvényt, akkor azt meghívja a componentWillUnmount idejében

Életciklus kezelés

```
class Time extends React.Component
{
    private timer: number;
    componentDidMount()
    {
        this.timer = setInterval( () => this.forceUpdate(), 1000 );
    }
    componentWillUnmount()
    {
        clearInterval( this.timer );
    }
    render()
    {
        return <span>{ new Date().toLocaleTimeString() }</span>
    }
}
```

Életciklus kezelés

- Át kell álljunk state-re, csak az vált ki rajzolást

```
function TimeHook()
{
  let [ time, setTime ] = React.useState( "" );
  React.useEffect( () =>
  {
    let timer = setInterval(()=>setTime(new Date().toLocaleTimeString()),1000);
    return () => clearInterval( timer );
  }, [] );
  return <span>{ time }</span>
}
```

- useEffect 2. paramétere a függőség
  - Mikor futtassa a megadott függvényt – itt soha

Sebesség

Alapok

- A render folyamat alapban minden komponenst érint
  - Azokat is, amik nem változtak
  - A rendszer nem tudja, hogy változnak-e, meg kell hívni a render-t, hogy ez kiderüljön
  - Onnan indul, ahol változás történt
    - Tehát csak a részfán megy végig
- Ez a működés optimalizálható
  - A nem változott komponenseket nem kell vizsgálni
  - De valahogyan tudni kell, hogy melyek ezek

PureComponent

- Nem a Component-ből, hanem a PureComponent-ből származtatunk
  - Csak akkor hívódik render, ha a props, vagy state változott
  - Egyéb belső állapota nem lehet (state-en kívül)
- Ha nem ennyire tiszta a rendszer
  - shouldComponentUpdate függvény
  - Meghívódik, hogy kiderüljön, kell-e rendert hívni
  - A PureComponent ezt használja
    - Megnézi, hogy a props és state változott-e

React.memo

- A PureComponent függvény (hook) verziója
  - Csak akkor hívja meg a függvényt, ha a props, vagy state változott
- Különben az eltárolt fát használja – innen a név

```
var comp = React.memo( props =>
{
  // ...
} );
```

Sebesség

- A React fa generálása és hasonlítása gyors
- Komplex felület esetén (10000+ elem) nem gyors
  - Mikor van ilyen sok elem?
    - Táblázatot kell megjeleníteni bonyolult cellákkal, details sorral, stb.
- Teljes komponensek (és gyerekeik) kihagyása jelentősen gyorsít
- Maga a HTML render, amit a böngésző végez, ma már gyors

Kérdések?