JUNIORDev DIGITALNA DALMACIJA

React hooks > useEffect

Sporedni učinci

React komponente su zamišljene kao samostalni elementi koji unutar sebe sadržavaju svu potrebnu logiku za njihovo funkcioniranje. Ipak, postoje slučajevi kada se komponente moraju uskladiti sa nekim vanjskim sustavima - npr. dohvat podataka sa drugog poslužitelja ili slanje analitičkih podataka nakon učitavanja komponente.

U takvim slučajevima koristimo *hook* pod nazivom ***useEffect*** i kao što mu ime govori on nam služi za upravljanje sporednim događajima (*side-effect*). Prije samih detalja o ovome *hook*-u, osvrnuti ćemo se na životni ciklus (*lifecycle*) React komponenti.

Životni ciklus komponenti

Sve React komponente imaju isti životni ciklus:

- komponenta se dodaje (***mount***) na zaslon (prvo renderiranje)
- komponenta se osvježava (***update***) kada primi novi *props* ili se promijeni njeno stanje
- komponenta se uklanja (***unmount***) sa ekrana kada je više nije potrebno prikazivati.

Naredbe koje pišemo unutar tijela komponente se izvršavaju prije renderiranja. Korištenjem *useEffect* možemo komponenti dodati logiku koja će se izvršiti nakon što je komponenta renderirana. U praksi je bolje sve izračune izvšiti prije renderiranja ali to nije uvijek moguće.

useEffect

Korištenje ***useEffect*** *hook*-a se izvodi u tri koraka:

- potrebno je definirati funkciju (***Effect***) koja će se pozivati nakon svakog renderiranja (ako nije drugačije navedeno)
- zatim se definiraju ovisnosti (***dependencies***). Efekt ne moramo pozivati nakon svakog renderiranja već ga možemo uvjetno pozvati ovisno o nekim vrijednostima komponente (npr. varijabli stanja)
- po potrebi dodati funkciju za čišćenje (***cleanup***). Ova funkcija se poziva kada se radi ***unmount*** komponente. Na taj način možemo osigurati da nemamo neke nedovršene akcije ili npr. da nam ne ostaje konekcija prema bazi.

Za početak, ovaj *hook* je također potrebno učitati iz React biblioteke

```
import {useEffect} from "react";
```

Zatim unutar komponente definiramo efekt koji joj želimo dodati:

```
useEffect(callback, [])
```

Pri tome je ***callback*** funkcija tj. efekt kojeg želimo izvršiti, a unutar niza se navode ***dependency*** vrijednosti. Niz možemo i izostaviti i tada se se efekt poziva nakon svakog renderiranja komponente. Najjednostavniji primjer bi bio:

```
useEffect(() => {
  console.log("Komponenta se renderirala")
})
```

Dodavanjem vrijednosti u *dependency* niz određujemo u kojim uvjetima će se pozvati efekt. Možemo vidjeti primjer aplikacije sa dva brojača, pri čemu ćemo samo jedan dodati u *dependency* niz. Promjena prvog brojača će uzrokovati pozivanje efekta, a promjena drugog neće.

App.jsx

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

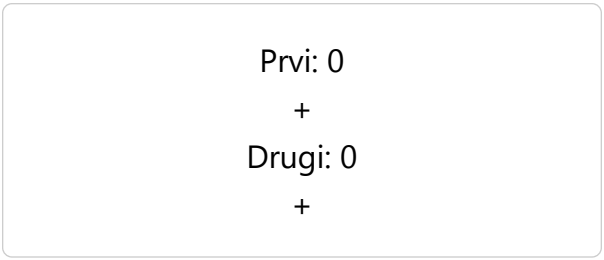
```
function App() {
  const [prvi, postaviPrvi] = useState(0)
  const [drugi, postaviDrugi] = useState(0)

  useEffect(() => {
    console.log("Brojač se uvećao")
  }, [prvi])

  return (
    <div className='main'>
      <p>Prvi: {prvi}</p>
      <button onClick={()=> postaviPrvi(prvi + 1)}>+</button>
      <p>Drugi: {drugi}</p>
      <button onClick={()=> postaviDrugi(drugi + 1)}>+</button>
    </div>
  );
}

export default App;
```

Možemo to pogledati i na interaktivnom primjeru, jedina je razlika što se poruka ispisuje u *Alert* prozoru kako ne bi morali otvarati konzolu:



Primjer učitavanja

Demonstrirati ćemo korištenje ***useEffect*** *hook*-a na primjeru učitavanja podataka sa neke udaljene lokacije. S obzirom da još nismo obradili dio komunikacije sa poslužiteljem, unutar naše aplikacije ćemo definirati pomoćnu funkciju koja simulira čekanje na odgovor od poslužitelja. Napravite novu datoteku "*dohvat.js*" koja izgleda ovako:

dohvat.js

```
const sleep = ms => new Promise(resolve => setTimeout(resolve, ms));
```

```
async function dohvatiKorisnika() {
  await sleep(3000);
  return { ime: "Sara", godine: 21, aktivan: false };
}

export default dohvatiKorisnika;
```

Za sada se nećemo osvrtati na ***Promise*** objekte niti logiku ove funkcije, dovoljno je znati da funkcija čeka 3 sekunde i nakon toga vraća objekt sa podacima o korisniku.

Pogledajmo kako bi izgledala naša React komponenta u kojoj želimo učitati te podatke:

App.jsx

```
import { useState } from "react";
import "./App.css";

import ucitajPodatke from './dohvat'

function App() {
  const [osoba, postaviOsobu] = useState({
    ime: " ...",
    godine: " ..."
  });

  return (
    <div className='main'>
      <p>Ime:{osoba.ime}</p>
      <p>Godine:{osoba.godine}</p>
    </div>
  );
}

export default App;
```

Problem nam je što prilikom postavljanja početnog stanja nemamo stvarne podatke o osobi koju želimo prikazati. Potrebno nam je unutar komponente dodati logiku koja će nakon renderiranja započeti učitavanje podataka te nakon uspješnog učitavanja osvježiti stanje komponente. To je upravo primjer sporednog efekta kojeg rješavamo sa ***useEffect***-om.

App.jsx

```jsx
import { useEffect, useState } from "react";
import "./App.css";

import ucitajPodatke from './dohvat'

function App() {

  const [osoba, postaviOsobu] = useState({
    ime: " ...učitavanje",
    godine: " ...učitavanje"
  });

  useEffect(() => {
    ucitajPodatke().then(rez => {
      postaviOsobu(rez);
    });
  }, []);

  return (
    <div className='main'>
      <p>Ime: {osoba.ime}</p>
      <p>Godine: {osoba.godine}</p>
    </div>
  );
}

export default App;
```

U gornjem primjeru imamo **asinkrono** izvršavanje. Pozivamo funkciju `ucitajPodatke` i tek nakon što smo dobili odgovor od (lažnog) poslužitelja pozivamo funkciju za promjenu stanja i spremamo učitane podatke. Uočite kako je ovaj put *dependency* niz prazan, ali nije izostavljen. Na ovaj način govorimo Reactu da ovaj efekt poziva samo prilikom **prvog renderiranja**.

Cleanup funkcija

Ostalo nam je još vidjeti kako radi *cleanup* funkcija kod *useEffect hook*-a. Svrha ove funkcije je da uklonimo neželjene efekte prije uklanjanja ili osvježavanja komponente. Napraviti ćemo

posebnu komponentu "*Prikaz*" koja će nam služiti za prikaz broja, ali ćemo joj dodati i jedan efekt. Kao što možete vidjeti iz primjera, želimo dodati *eventListener* koji će osluškivati jesmo li pritisnuli neku tipku na tipkovnici i pozvati *alert* prozor sa porukom.

Prikaz.jsx

```
import {useEffect} from "react"

function Prikaz(props) {

  useEffect(() => {
    const pritisak = () => alert("Klik")
    window.addEventListener("keyup", pritisak)
  });

  return (
    <div>
      <p>Broj: {props.broj}</p>
    </div>
  );
}
export default Prikaz
```

Zatim ćemo u glavnom dijelu aplikacije uključiti ovu komponentu:

App.jsx

```jsx
import { useState } from "react";
import "./App.css";

import Prikaz from "./Prikaz"

function App() {
  const [broj, postaviBroj] = useState(0);

  return (
    <div className='main'>
      <Prikaz broj={broj} />
      <button onClick={() => postaviBroj(broj + 1)}>+</button>
    </div>
  );
}

export default App;
```

Već možemo otprilike očekivati što se događa - cijela aplikacija se renderira, uključujući i komponentu `<Prikaz />`. Njeno renderiranje uzrokuje pozivanje *useEffect*-a i na prozor preglednika bi se trebao dodati *eventListener*.

Pritiskom na tipkovnicu već dolazimo do prvog problema - *alert* poruka se pojavljuje dva puta. Ovo je zapravo značajka React-a koja se koristi prilikom razvoja aplikacije (konačna verzija aplikacije nema ovo svojstvo) - komponente će pri pokretanju aplikacije renderirati **dva puta**. Svrha tog postupka je upravo kako bi lakše uočili potencijalne pogreške u našim komponentama. Ako je sva aplikacijska logika ispravno napisana, dvostruko renderiranje ne bi trebalo utjecati na rad aplikacije.

Koji je zapravo problem? Pokušajmo uvećati brojač, što će uzrokovati ponovno iscrtavanje komponente `<Prikaz />` i samim time pozivanje njenog efekta. Ako sada pritisnemo tipku na tipkovnici, *alert* poruka će se ispisati tri puta. Problem je što svakim novim renderiranjem komponente dodajemo novi *eventListener* i oni se samo gomilaju.

Teoretski bi mogli u *useEffect* dodati prazni *dependency* niz tako da se efekt poziva samo jednom ali *eventListener* bi ostao čak i ako se komponenta koja ga je pozvala ukloni sa ekrana

(a to rijetko kada želimo i može dovesti do pogreške u našoj aplikaciji).

Rješenje je dodati već spomenutu *cleanup* funkciju. Ona se definira na način da unutar *effect* funkcije moramo kao povratnu vrijednost poslati drugu funkciju. Ta druga funkcija je upravo *cleanup* funkcija koja je poziva u dva slučaja - ako se radi ***update*** ili ***unmount*** komponente.

Konačni izgled *useEffect*-a unutar komponente bi trebao izgledati ovako:

```
useEffect(() => {
  const pritisak = () => alert("Klik");
  window.addEventListener("keyup", pritisak);

  return () => {
    window.removeEventListener("keyup", pritisak)
  }
});
```

Sada će se prilikom renderiranja komponente dodati *eventListener* ali će se također i prije svakog osvježavanja ili uklanjanja komponente isti taj *eventListener* ukloniti. Rezultat je aplikacija koja ispisuje *alert* poruku samo jednom, neovisno koliko puta osvježili komponentu.

Last updated on March 13, 2023

‹ useRef useContext ›