JUNIORDev DIGITALNA DALMACIJA



# Kontekst

Tok podataka u Reactu bi trebao biti jednosmjeran - od roditeljskog elementa (komponente) prema njegovoj djeci (*child* elementima). Prosljeđivanje se vrši kroz svojstva (*props*) koja se šalju ugniježđenim komponentama prilikom njihovog instanciranja. Već smo prethodno spominjali pristup "podizanja stanja" (*lifting the state*) kojeg se poželjno držati prilikom određivanja u koju komponentu ćemo spremiti određeno stanje.

Ponekad nam se može dogoditi da se komponenta kojoj želimo poslati podatak nalazi "duboko" u DOM strukturi tj. u stablu komponenti - *child* komponenta može biti nekoliko razina ispod roditelja koji sadrži potrebni podatak. U takvim situacijama željeni podatak bi se trebao propagirati razinu po razinu sve dok ne dođe do tražene komponente. To u praksi znači da komponentama koje se nalaze između šaljemo podatke koji im zapravo ne trebaju i koje samo prosljeđuju dalje svojoj djeci. Problem sa ovim pristupom je što vrlo brzo postaje teško održavati takvu aplikaciju i teško je pratiti logiku tj. tok podataka.

Navedeni problem možemo riješiti na dva načina. Prvi je uvođenjem centralnog spremnika (*store*) na razini cijele aplikacije u kojem bi onda bila spremljena sva stanja koja se koriste u više od jedne komponente. Ovaj pristup najčešće zahtjeve korištenje dodatne biblioteke kao što je ***Redux*** i veće promjene na aplikaciji. S obzirom da se radi o složenijim konceptima, za sada ćemo taj pristup preskočiti.

Kod manjih aplikacija i jednostavnijih slučajeva možemo koristiti i ***useContext*** *hook* koji nam dozvoljava da nad nekim skupom komponenti definiramo globalni kontekst koji će biti zajednički za sve pripadajuće komponente. Definiranje konteksta se odvija u tri koraka:

- stvaranje (*creating*) konteksta
- pružanje (*providing*) konteksta
- korištenje (*consuming*) konteksta

Korištenje konteksta unutar aplikacije ćemo demonstrirati na primjeru oblikovanja. Većina današnjih aplikacija (kao i ova stranica) podržava svijetlu i tamnu temu tj. izgled sučelja. Promjenom teme većina komponenti bi trebala promijeniti izgled, što znači da im je potreban podatak koja je trenutna tema. Bez korištenja konteksta svim komponentama u stablu bi trebali slati podatak koja je aktualna tema. Kontekst nam dozvoljava da sve odabrane komponente imaju pristup istom podatku, bez da ga stalno prosljeđujemo.

# Stvaranje konteksta

Za stvaranje konteksta koristimo funkciju `createContext()` koja je uključena u React. Ovo se najčešće radi u odvojeno datoteci (zbog preglednosti i lakšeg održavanja aplikacije). Kontekst možete zamisliti kao posebnu komponentu koju stvaramo sa tom funkcijom i postavljamo početnu vrijednost konteksta.

kontekst.js

```
import {createContext} from 'react'

const TemaContext = createContext("light") // ili "dark"

export default TemaContext
```

U ovom primjeru smo stvorili kontekst pod nazivom `TemaContext` i postavili mu početnu vrijednost na "*light*" (ona se može kasnije i mijenjati).

# Pružanje konteksta

Nakon što smo stvorili kontekst, potrebno je odrediti njegov doseg - tj. koje će sve komponente imati pristup tom kontekstu. To radimo na način da roditeljski element "omotamo" sa našim kontekstom. Time smo tom elementu i svim njegovim ugniježđenim (*child*) elementima omogućili pristup kontekstu.

App.jsx

```
import TemaContext from "./kontekst";

function App() {
  return (
    <div>
      <Naslov />
      <TemaContext.Provider value="dark">
        <GlavnaKomponenta />
      </TemaContext.Provider>
      <Footer />
    </div>
  );
}
```

U ovom primjeru naša aplikacija se sastoji od tri glavne cjeline - naslova, glavnog dijela aplikacije i podnožja (*footer*). Naslov i *footer* nam imaju uvijek isti izgled pa njima nije potreban pristup kontekstu. Glavni dio aplikacije smo "omotali" sa `<TemaContext.Provider>` komponentom što znači da komponenta `<GlavnaKomponenta />` i svi njeni *child* elementi imaju pristup vrijednosti zadanog konteksta. Iako smo prilikom stvaranja konteksta zadali početnu vrijednost, ovdje je pomoću svojstva ***value*** mijenjamo.

# Korištenje konteksta

Preostalo nam je još vidjeti kako dohvaćamo vrijednost konteksta. Za to postoje dva načina. Prvi (preporučeni) je korištenje ***useContext*** *hook*-a pomoću kojeg možemo "konzumirati" željeni kontekst i dohvatiti njegovu vrijednost.

Tipka.jsx

```
import { useContext } from 'react';
import TemaContext from "./kontekst";

function Tipka(props) {
  const tema = useContext(TemaContext)
  return(
    <>
```

```
      <button className={tema} onClick={() => props.klik}>{props.natpis}</button>
    </>
  )
}
```

Kao što vidimo, u samoj komponenti moramo učitati *useContext hook*, kao i naš kontekst čiju vrijednost dohvaćamo. Zatim je u komponenti dovoljno pozvati taj *hook*, poslati mu željeni kontekst kao argument i spremiti dohvaćenu vrijednost koju zatim možemo koristiti unutar naše komponente. U ovom slučaju, ovisno o vrijednosti konteksta elementu dodjeljujemo različito svojstvo klase i na taj način možemo primijeniti različite stilove oblikovanja.

Drugi način je bez *hook*-a, uz korištenje `Consumer` komponente koja se nalazi u svakom stvorenom kontekstu. Unutar te komponente možete pristupiti vrijednosti konteksta i iskoristiti ga kako bi vratili izgled komponente. Gornji primjer napisan na taj način bi izgledao ovako:

Tipka.jsx

```
import TemaContext from "./kontekst";

function Tipka(props) {
  return(
    <TemaContext.Consumer>
      { tema => <button className={tema} onClick={() => props.klik}>{props.natpis}</button>
    </TemaContext.Consumer>
  )
}
```

Krajnji rezultat je isti u oba slučaja, samo što je prvi način možda malo pregledniji i jednostavniji. Naravno, bilo bi dobro implementirati funkcionalnost promjene vrijednosti konteksta. To također nije složeno, dovoljno nam je vratiti se u korak pružanja konteksta te umjesto fiksne vrijednosti koristiti varijablu stanja te dodati mogućnost njene promjene.

App.jsx

```
function App() {
  const [tema, postaviTemu] = useState("light")

  function promjenaTeme(){
    postaviTemu(tema == "light" ? "dark" : "light")
```

```
    }

    return (
      <div>
        <Naslov />
        <TemaContext.Provider value={tema}>
          <GlavnaKomponenta />
        </TemaContext.Provider>
        <button onClick={promjenaTeme}>Light/Dark</button>
        <Footer />
      </div>
    );
  }
```

Promjenom vrijednosti konteksta, sve komponente koje koriste taj kontekst će automatski dobiti novu vrijednost. Osim već navedenih, još jedna prednost ovog pristupa je što korak korištenja konteksta možemo ponavljati više puta i vrlo je jednostavno naknadno dodati komponenti pristup vrijednosti konteksta, dovoljno je iskoristiti *useContext hook* i željeni kontekst.

Last updated on March 13, 2023

‹ useEffect

Unos podataka ›