JUNIORDev DIGITALNA DALMACIJA

CRUD operacije > Slanje podataka

Slanje podataka

Nakon što smo implementirali dohvat podataka sa poslužitelja, možemo krenuti sa implementacijom slanja novih podataka. Prije samog slanja potrebne su nam komponente za unos.

Forma za unos

Za početak ćemo napraviti novu komponentu koja će predstavljati formu za unos podataka nove rezervacije. Krenimo od početnog izgleda kojeg ćemo nadograđivati korak po korak. Komponenta će nam sigurno imati varijablu stanja u kojoj ćemo spremati sve podatke i pomoćnu funkciju koju ćemo koristiti kada budemo htjeli poslati podatke na poslužitelj. Podatke ćemo spremati u jedan objekt zbog jednostavnosti - možemo imati i odvojeno stanje za svaki podatak ali ovako će nam biti jednostavnije poslati cijeli objekt na poslužitelj.

components/UnosForma.jsx

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

function UnosForma() {
  const [formaPodaci, postaviPodatke] = useState({
    ime: "",
    prezime: "",
    dob: "",
    pocetak: "",
    kraj: "",
    klasa: "",
  });

  const saljiPodatke = event => {
    event.preventDefault();
    console.log(formaPodaci);
  };
```

```
  return (
    <form onSubmit={saljiPodatke}>
      /* Ostatak elemenata za unos */
      <button type='submit'>Nova rezervacija</button>
    </form>
  );
}

export default UnosForma;
```

Trenutno nam funkcija `saljiPodatke()` nema nikakvu funkcionalnost osim zaustavljanja klasične *submit* akcije i ispisivanja podataka na konzolu. Ovaj dio ćemo implementirati posljednji, kada se uvjerimo da dobro dohvaćamo podatke sa polja za unos.

Unos podataka

Krenimo sada sa implementacijom elemenata za unos unutar `<form>` elementa. Na prethodnim lekcijama smo se već upoznali sa načinom unosa podataka pa nećemo previše ulaziti u detalje. Cilj nam je dodati elemente za unos kako bi mogli unijeti podatak o novoj rezervaciji. Polja za unos teksta su najjednostavnija za implementirati pa ćemo krenuti sa njima.

Unos teksta

S obzirom da imamo varijablu stanja za spremanje podataka očito je da ćemo koristiti pristup sa kontroliranim komponentama. Zbog toga `<input>` elementu definiramo svojstva ***value*** i ***onChange***. Vrijednost elementa vezujemo uz odgovarajuću vrijednost iz varijable stanje, a za promjenu ulaza ćemo napisati pomoćnu funkciju.

```
return (
  <form onSubmit={saljiPodatke}>
    <div>
      <label>
        Ime:
        <input
          type='text'
          name='ime'
```

```
            value={formaPodaci.ime}
            onChange={promjenaUlaza}
            required
          />
        </label>
      </div>
      <button type='submit'>Nova rezervacija</button>
    </form>
  );
```

Cilj nam je napisati jednu funkciju koja može upravljati svim promjenama ulaza, kako ne bi morali za svaki element pisati posebnu pomoćnu funkciju. Pogledajmo koje podatke funkcija prima kada se promjeni vrijednost nekog *input* elementa:

```
function promjenaUlaza(event) {
  console.log(event.target);
}
```

Kroz svojstvo *event.target* možemo pristupiti HTML elementu koji je detektirao promjenu. Ako postavimo da svaki element za unos ima atribut ***name*** koji odgovara svojstvu u našoj varijabli stanja, onda možemo napisati pomoćnu funkciju koja će nam pokrivati promjene u više elemenata.

```
function promjenaUlaza(event) {
  const { name, value } = event.target;
  postaviPodatke({ ...formaPodaci, [name]: value });
}
```

Na isti način možemo dodati elemente za unos prezimena i godina (nisu u prikazu podataka ali jesu u "bazi") pri čemu samo moramo voditi računa o imenima atributa.

```
  <div>
    <label>
      Prezime:
      <input type='text' name='prezime' value={formaPodaci.prezime}
        onChange={promjenaUlaza} required />
    </label>
  </div>
```

```
<div>
  <label>
    Godina:
    <input type='number' name='dob' value={formaPodaci.dob}
      onChange={promjenaUlaza} required/>
  </label>
</div>
```

Za sva tri polja za unos možemo iskoristiti funkciju `promjenaUlaza()`. Prije nastavka provjerite rade li ispravno sva tri polja za unos - iskoristite opciju slanja podataka koja ih trenutno ispisuje na konzolu.

Odabir iz liste

Idući korak nam je implementirati dva elementa za unos u obliku liste gradova - jedan za odabir polazišta, a drugi za odabir odredišta putovanja. Kako bi smanjili mogućnost pogreške, umjesto unosa teksta korisnik će na raspolaganju imati popis gradova. Aplikacija će imati još jednu dodatnu funkcionalnost - umjesto da je popis gradova fiksan, dozvoliti ćemo mogućnost promjene, te sa poslužitelja dohvaćati niz gradova. Popis gradova je dostupan na lokalnom poslužitelju na adresi `/gradovi/`. Gradove ćemo dohvatiti na početku komponente, ponovno uz pomoć *useEffect hook*-a.

UnosForma.jsx

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

function UnosForma() {
  const [gradovi, postaviGradove] = useState([]);
  // Ostali useState

  useEffect(() => {
    axios
      .get("http://localhost:3001/gradovi")
      .then(rez => postaviGradove(rez.data))
      .catch(err => console.log(err.message));
  }, []);

  // Ostatak komponente
}
```

Kada smo ispravno dohvatili gradove i spremili ih u stanje komponente `gradovi`, iskoristiti ćemo taj niz gradova kako bi napravili `<select>` element sa točno tim vrijednostima.

```
<div>
  <label>
    Početak putovanja:
    <select
      name='pocetak'
      value={formaPodaci.pocetak}
      onChange={promjenaUlaza}
      required
    >
      <option value=''>--Odaberi grad--</option>
      {gradovi.map(grad => (
        <option key={grad} value={grad}>
          {grad}
        </option>
      ))}
    </select>
  </label>
</div>
```

Atributi uz `<select>` oznaku su jednaki kao i kod unosa teksta - moramo imati `value` i `onChange` atribute kako bi mogli upravljati stanjem komponente, te odgovarajući `name` atribut koji koristi funkciji `promjenaUlaza()` za dohvat odgovarajuće vrijednosti.

Unutar `<select>` oznake potrebno je ugnijezditi `<option>` elemente kako bi odredili koje vrijednosti korisnik može izabrati iz popisa. Umjesto fiksnog navođenja opcija, iskoristiti ćemo mogućnost Reacta i pomoću `map()` metode dinamički generirati `<option>` elemente - po jedan za svaki element iz niza `gradovi` kojeg smo u prethodnom koraku napunili sa gradovima dobivenim od strane poslužitelja.

Drugi element za odabir grada je identičan, samo moramo promijeniti vrijednosti `name` i `value` atributa na `name='kraj'` i `value={formaPodaci.kraj}` (i naravno opis uz labelu) kako bi nam se drugi odabir spremao u odgovarajuće svojstvo stanja komponente.

Odabir - *radio* element

Preostaje nam još odabir klase - ekonomska (E) ili poslovna (B - *business*). Za to nam je najbolja opcija `<radio>` element. Ponovno ćemo dohvatiti vrijednosti klasa za poslužitelja i spremiti ih u varijablu stanja.

UnosForma.jsx

```
function UnosForma() {
  const [klase, postaviKlase] = useState([])
  // Ostatak stanja

  useEffect(() => {
    axios
      .get("http://localhost:3001/gradovi")
      .then(rez => postaviGradove(rez.data))
      .catch(err => console.log(err.message));
    axios
      .get("http://localhost:3001/klase")
      .then(rez => postaviKlase(rez.data))
      .catch(err => console.log(err.message));
  }, []);
```

```
    // Ostatak komponente
```

U ovom primjeru smo učitavanje oba skupa podataka stavili pod isti *useEffect hook*. S obzirom da se radi samo u učitavanju prilikom prvog renderiranja ovo nam je prihvatljivo. U slučaju kada imate akcije koje su neovisne jedna o drugoj, a mogu imati svoje *dependency* vrijednosti, bolje ih je napisati u odvojenim *useEffect hook*-ovima.

Kada imamo više zahtjeva koje želimo istovremeno obraditi možemo koristiti i `Promise.all([])` metodu koja prima niz asinkronih operacija (u našem slučaju zahtjva) te poziva `then()` tek kada se sve operacije unutar niza izvrše. U tom slučaju bi *useEffect hook* izgledao ovako:

```
useEffect(() => {
  Promise.all([
    axios.get("http://localhost:3001/gradovi"),
    axios.get("http://localhost:3001/klase"),
  ])
    .then(([rezGradovi, rezKlase]) => {
      postaviGradove(rezGradovi.data);
      postaviKlase(rezKlase.data);
    })
    .catch(err => console.log(err.message));
}, []);
```

Nakon što smo spremili klase (pogledajte u konzoli ili u datoteci kako izgledaju podaci) možemo generirati `<radio>` elemente.

```
<div>
  <label>
    Klasa:
    {klase.map(klasa => (
      <label key={klasa.oznaka}>
        <input
          type='radio'
          name='klasa'
          value={klasa.oznaka}
          checked={formaPodaci.klasa === klasa.oznaka}
          onChange={promjenaUlaza}
          required
```

```
          />{" "}
          {klasa.ime}
        </label>
      ))}
    </label>
  </div>
```

Kod korištenja `<radio>` elemenata moramo voditi računa i o atributu `checked` koji mora biti postavljen na `true/false` vrijednost te zbog toga radimo logičku usporedbu odgovara li oznaka pojedinog elementa vrijednosti koja je spremljena u stanju komponente.

Konačno imamo funkcionalnu formu za unos podataka (nismo se osvrtali na oblikovanje ali to je u ovom kontekstu manje važno). Struktura komponente se malo "napuhala" što je možda znak da bi je mogli podijeliti u manje komponente - npr. posebna komponenta za svaki `<input>` element ali ni to nam trenutno nije u fokusu. Umjesto toga, fokusirati ćemo se na ono zbog čega smo i počeli raditi formu - slanje prikupljenih podataka na poslužitelj.

Slanje podataka

Do sada smo se upoznali samo sa GET zahtjevom koji nam služi za dohvat podataka. Kada želimo slati podatke na poslužitelj koristimo drugu vrstu zahtjeva, konkretno POST zahtjev. POST zahtjev se razlikuje od GET zahtjeva po tome što uz sami zahtjev moramo priložiti i podatke koje želimo poslati.

Podaci koje šaljemo se spremaju u **tijelo** (***body***) samog zahtjeva, naravno prije nego se pošalje. Ovisno o načinu na koji je poslužitelj implementiran, ponekad je potrebno u zaglavlje (***header***) zahtjeva dodati informaciju u kojem formatu su priloženi podaci. Na taj način poslužitelj zna kako ih pravilno obraditi.

POST zahtjev u Axios-u

Slanje POST zahtjeve u Axios-u je također prilično jednostavno. Koristi se ugrađena `post()` metoda koja prima dva parametra - adresu (URL) na koju šaljemo zahtjev te podatak koji želimo poslati.

Obrada odgovora je identična kao i kod GET zahtjeva, koristimo `then()` metodu kako bi definirali što će se dogoditi nakon što dobijemo odgovor od poslužitelja. Sama struktura odgovora ovisi o implementaciji poslužitelja - u većini slučajeva ćete dobiti statusni kôd 200 (OK) ili 201 (*Created*). Također, neki poslužitelji vraćaju podatak u obliku u kojem je spremljen na poslužitelj i koji se može se razlikovati od podatka kojeg smo mi poslali. Najčešći primjer je dodavanje ID atributa prilikom spremanja u bazu ili dodavanje nekog dodatnog meta-podatka od strane baze/poslužitelja.

Prije slanja podataka pogledajmo kako su strukturirani podaci u "bazi":

```
{
  "id": 1,
  "osoba": {
    "ime": "Ivana",
    "prezime": "Ivić",
    "dob": 24
  },
  "karta": {
    "pocetak": "Split",
    "kraj": "Rim",
    "klasa": "E"
  }
}
```

Očito je da nam struktura objekta u varijabli stanja ne odgovara u potpunosti pa ćemo je prije slanja na poslužitelj transformirati u odgovarajući oblik. Napisati ćemo pomoćnu funkciju kako bi nam funkcija za slanje bila preglednija.

```
function obradiPodatke(objekt){
  return {
    "osoba" : {
      "ime" : objekt.ime,
      "prezime": objekt.prezime,
      "dob": Number(objekt.dob)
    },
    "karta":{
      "pocetak": objekt.pocetak,
      "kraj": objekt.kraj,
      "klasa": objekt.klasa
```

```
    }
  }
}
```

Sada možemo dovršiti funkcionalnost funkcije za slanje podataka na poslužitelj:

```
const saljiPodatke = event => {
  event.preventDefault();
  console.log(formaPodaci);

  const zaSlanje = obradiPodatke(formaPodaci)

  axios.post('http://localhost:3001/rezervacije', zaSlanje)
  .then(rez => console.log(rez))
};
```

Axios automatski prepoznaje o kojem tipu podataka se radi pa ne moramo postavljati vrijednosti zaglavlja, ali u slučaju kada je to potrebno napraviti, dovoljno je poslati prilikom poziva `post()` metode poslati i treći parametar koji će sadržavati postavke (konfiguracijski objekt) zahtjeva, uključujući zaglavlja. U našem slučaju bi to izgledalo ovako:

```
axios.post("http://localhost:3001/rezervacije", zaSlanje, {
  headers: {
    'content-type': "application/json"
  }
})
.then(rez => console.log(rez))
```

Za sve ostale postavke zahtjeva pogledajte službenu Axios dokumentaciju.

Ako pogledamo odgovor koji smo dobili od lokalnog poslužitelja nakon slanja, vidjeti ćemo da smo dobili status 201 kao odgovor te da u *data* svojstvu imamo spremljeni objekt (zajedno sa dodanim ID svojstvom). Jedino što nam trenutno nedostaje je osvježavanje prikaza podataka.

Osvježavanje prikaza

Za implementaciju osvježavanja prikaza moramo prvo analizirati strukturu naše aplikacije.

Podaci o rezervacijama se nalaze u glavnom dijelu aplikacije, konkretno `<App />` komponenti. Logika slanja zahtjeva i primanja odgovora je implementirana u `<UnosForma />` komponenti koja je *child* komponenta od `<App />`.

Ovakva situacija nije ništa neuobičajeno (niti nepoznato jer smo već imali takve primjere). Iz glavnog dijela aplikacije ćemo *child* komponenti poslati referencu na funkciju za promjenu varijable stanja:

App.jsx

```
function App() {
  const [rezervacije, postaviRezervacije] = useState([]);

  useEffect(() => {
    axios
      .get("http://localhost:3001/rezervacije/")
      .then(res => postaviRezervacije(res.data));
  }, []);

  return (
    <div className='App'>
      <h2>Popis rezervacija</h2>
      <Tablica rezervacije={rezervacije} />
      <h2>Nova rezervacija</h2>
      <UnosForma dodaj={postaviRezervacije} />
    </div>
  );
}

export default App;
```

Zatim ćemo u `<UnosForma />` komponenti iskoristiti dobiveno svojstvo i nakon što primimo odgovor od poslužitelja sa novim objektom, postaviti ćemo novu vrijednost stanja roditeljske komponente (ne zaboravite dodati *props* u potpis funkcije komponente).

```
  const saljiPodatke = event => {
    event.preventDefault();
    console.log(formaPodaci);

    const zaSlanje = obradiPodatke(formaPodaci)
```

```
      axios.post('http://localhost:3001/rezervacije', zaSlanje)
      .then(rez => {
        props.dodaj(stanje => [...stanje, rez.data])
      })
    };
```

Uočite da smo prilikom postavljanja nove vrijednosti stanja koristili *updater* funkciju jer ona ima pristup trenutnoj vrijednosti varijable stanja. Alternativa bi bila uz funkciju za promjenu stanja slati i cijelo stanje kao odvojeni *props* atribut ali kao što vidimo, za tim nema potrebe.

Sada nam se nakon promjene stanja glavna komponenta automatski osvježava i prikazuje novi podatak na sučelju.

Napomena: Pripazite na činjenicu da nakon slanja novog podatka i dobivanja odgovora nismo slali novi GET zahtjev na poslužitelj već smo u varijablu stanja dodali novi podatak. Tehnički je moguće da se od trenutka prvog učitavanja do slanja podatka (i dobivanja odgovora) promijenilo stanje u bazi podataka. Naravno, to ovisi o konkretnoj aplikaciji.

U ovom slučaju ne predstavlja problem jer nitko drugi nema mogućnost dodavanja rezervacija ali bi u stvarnoj primjeni vjerojatno trebalo implementirati neku dodatnu logiku - najjednostavnije bi bilo u funkciji za slanje podataka nakon dobivenog odgovora poslati GET zahtjev i onda njegov odgovor iskoristiti za postavljanje novog stanja. Također bi se mogao prilagoditi i poslužitelj (što u ovom slučaju ne možemo napraviti) na način da nakon svakog spremanja novog podatka šalje cijeli niz podataka kao odgovor, a ne samo spremljeni podatak. Dio za slanje zahtjeva bi u tom slučaju izgledao ovako:

```
axios.post("http://localhost:3001/rezervacije", zaSlanje)
  .then(rez => {
    axios.get("http://localhost:3001/rezervacije")
      .then(rez => props.dodaj(rez.data));
});
```

Zadatak

Prije nego nastavimo sa ostalim vrstama zahtjeva, imamo jedan logički problem u aplikaciji.

Korištenjem ***required*** atributa smo pokrili osnovni validacijski uvjet da nijedno polje ne može biti prazno ali postoji još jedan očiti način kako korisnik može unijeti logički pogrešne podatke. Korisnika ništa ne sprječava da unese isti grad kao početnu i krajnju točku putovanja - nadogradite funkciju za slanje podataka da se u tom slučaju umjesto slanja podataka na poslužitelj korisniku ispiše poruka da gradovi moraju biti različiti.

‹ Dohvat podataka

Brisanje podataka ›