JUNIORDev DIGITALNA DALMACIJA

Komunikacija > Klijent-poslužitelj

Komunikacija s poslužiteljem

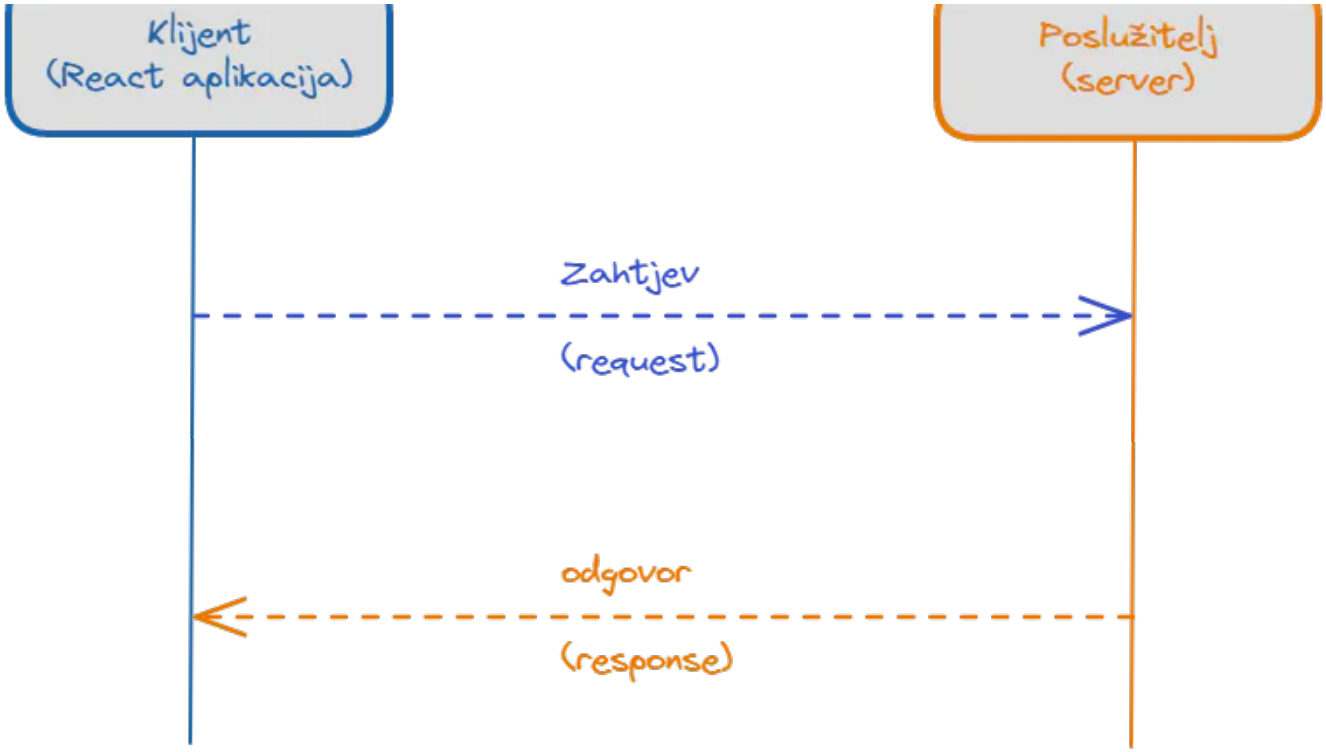
Postoje različite arhitekture i pristupi izradi web aplikacija ali generalno možemo reći da se web aplikacije sastoje od tri glavne komponente:

- Prezentacijski sloj - dio aplikacije koji je vidljiv korisniku i preko kojeg korisnik vrši interakciju sa samom aplikacijom (***Frontend***)
- Poslovni sloj - dio aplikacije koji sadrži implementaciju ključnih funkcionalnosti sustava i enkapsulira svu poslovnu logiku aplikacije (***Backend***)
- Podatkovni sloj - dio aplikacije koji pruža pristup podacima koji se koriste u sklopu aplikacije (***Baza podataka***)

React aplikacije spadaju u dio prezentacijskog sloja tj. korisničkog (*Frontend*) dijela aplikacije. Prilikom razvoja React aplikacije glavni fokus nam je omogućiti korisniku što ugodnije i funkcionalnije okruženje za rad sa našom aplikacijom. Iako sva tri sloja aplikacije možemo razvijati odvojeno, potrebno je implementirati način komunikacije između tih slojeva. Zbog toga ćemo u idućim primjerima fokusirati na komunikaciju između korisničkog (*Frontend*) i poslužiteljskog (*Backend*) sloja aplikacije.

Klijent - poslužitelj arhitektura

Najjednostavniji način komunikacije sa poslužiteljem (serverom) je tzv. "*klijent-poslužitelj*" ili "*klijent-server*" arhitektura. Naravno krećemo od pretpostavke da su klijent i poslužitelj povezani u mrežu. Komunikacija se odvija na način da klijent pošalje zahtjev (*request*) koristeći neki od web protokola (npr. HTTP ili HTTPS). Jednom kada zahtjev stigne do poslužitelja, on će ga obraditi i poslati odgovor (*response*) natrag klijentu. Taj odgovor u pravilu sadrži tražene podatke koje onda klijent pri primitku odgovora može prikazati korisniku na sučelju.



Ovo je naravno prilično pojednostavljen prikaz ali za početak dovoljan za ilustrirati način komunikacije između klijenta i poslužitelja. Kasnije ćemo se upoznati i sa različitim vrstama zahtjeva i načinima komunikacije.

Asinkrono izvršavanje

Za pravilnu implementaciju klijent-poslužitelj komunikacije potrebno nam je dobro poznavanje još jednog koncepta - asinkronog izvršavanja kôda.

Većina naredbi koje pišemo u JS-u su sinkrone i izvršavaju se slijedno - jedna po jedna. Problem sa klijent-poslužitelj komunikacijom (iz perspektive klijentske aplikacije) je što nakon slanja zahtjeva, klijent više nema kontrolu nad izvršavanjem - tj. ne može točno znati kada (i hoće li) poslužitelj obraditi njegov zahtjev i poslati odgovor. U većini slučajeva su nam za nastavak izvršavanja aplikacije potrebni podaci koje smo zatražili od poslužitelja. Iz tog razloga je potrebno "pauzirati" izvršavanje logike na klijentu sve dok ne dobijemo traženi odgovor.

Promise objekt

Moderni JavaScript za upravljanje asinkronim izvršavanjem naredbi koristi ugrađeni ***Promise*** objekt. *Promise* objekt (u prijevodu doslovno "*obećanje*") je zamjena za vrijednost koja nam nije nužno poznata u trenutku stvaranja objekta. Možemo reći da *Promise* objekt predstavlja eventualan rezultat neke asinkrone operacije (bio on pozitivan ili negativan).

Na ovaj način smo omogućili asinkronim funkcijama vraćanje vrijednosti slično kao što to rade i sinkrone. Jedina je razlika što konačni rezultat ne dobijemo odmah nego imamo *obećanje (Promise)* tog konačnog rezultata koji će se pojaviti kada se asinkrona funkcija izvrši do kraja.

Promise objekti se mogu nalaziti u jednom od tri stanja:

- ***pending*** (čekanje) - početno stanje svih *Promise* objekata, označava da se još uvijek čeka rezultat asinkrone operacije
- ***fulfilled*** (ispunjeno) - operacija je uspješno izvršena i dostupan nam je konačni rezultat
- ***rejected*** (odbijeno) - operacija je bila neuspješna, umjesto rezultata imamo poruku o pogrešci (*error*)

Asinkrone funkcije u JS-u vraćaju *Promise* objekt kako bi mogli upravljati izvršavanjem naše aplikacije. Komunikacija sa poslužiteljem je upravo primjer asinkrone operacije. U trenutku kada pošaljemo zahtjev na poslužitelj tj. pozovemo funkciju za slanje zahtjeva, nemamo odmah na raspolaganju dostupan konačni odgovor, ali dobijemo *Promise* objekt čije onda stanje možemo provjeravati.

Svaki *Promise* objekt sadrži tzv. *executor* funkciju koja bi trebala izvršiti određeni zadatak. Ta funkcija također prima dva argumenta (automatski od strane JS-a) koji predstavljaju reference za metode pomoću kojih signaliziramo ispunjenje (*resolve*) ili odbijanje (*reject*) zadatka.

Stvaranje Promise objekta

Pogledajmo primjer jednostavnog Promise objekta koji nakon jedne sekunde izvršava zadatak i šalje signal da je obećanje ispunjeno.

```
let obecanje = new Promise((resolve, reject) => {
  setTimeout(() => resolve("Gotovo"), 1000)
```

```
})
```

Na isti način smo mogli i odbiti obećanje, samo koristimo drugu referencu:

```
let obecanje = new Promise((resolve, reject) => {
  setTimeout(() => reject("Pogreška"), 1000)
})
```

Dohvat rezultata

Prilikom komunikacije sa poslužiteljem nećemo pisati vlastite *Promise* objekte već ćemo koristiti asinkrone funkcije koje nam vraćaju već definirane *Promise* objekte. Zbog toga nam je trenutno važnije vidjeti kako znamo je li *Promise* objekt ispunjen i na koji način ćemo dohvatiti njegovu vrijednost.

To se radi korištenjem metode `.then()` koju pozivamo nad samim *Promise* objektom. Korištenjem gornjeg primjera to bi izgledalo ovako:

```
let obecanje = new Promise((resolve, reject) => {
  setTimeout(() => resolve("Gotovo"), 1000)
});

obecanje.then(rezultat => {
  console.log(rezultat)
});
```

Metodi `.then()` kao argument moramo poslati funkciju koja će se pozvati nakon izvršavanja obećanja i koja kao argument prima konačnu vrijednost asinkrone operacije. U ovom trenutku smo sigurni da je asinkrona operacija uspješno izvršena te da imamo konačnu vrijednost koju sada možemo upotrijebiti u našoj aplikaciji. U ovom primjeru je samo ispisujemo na konzolu kako bi vidjelo da se radi o istoj vrijednosti koju smo poslali kao argument `resolve()` metodi (poruku "Gotovo").

Upravljanje pogreškama

Izvršavanje asinkrone operacije ne mora uvijek biti uspješno. Tada ćemo naravno imati i drugačiju logiku aplikacije jer nismo dobili traženi podatak. *Callback* funkcija unutar `.then()` metode će se pozvati samo u slučaju uspješno izvršene operacije. Za upravljanje odbijenih obećanja koristimo `.catch()` metodu koju ulančavamo na isti *Promise* objekt i to nakon `.then()` metode. Promijeniti ćemo prethodni primjer tako da se *Promise* odbije i dodati ćemo logiku upravljanja pogreškom.

```
let obecanje = new Promise((resolve, reject) => {
  setTimeout(() => reject("Greška!"), 1000)
})

obecanje
  .then(rezultat => {
    console.log(rezultat)
  })
  .catch(greska => {
    console.log("Razlog pogreške: " + greska)
  })
```

Callback funkcije su u gornjim primjerima pisane u punim oblicima zbog boljeg razumijevanja. S obzirom da ovdje imamo samo jedan argument i jedan izraz možemo koristiti skraćeni oblik (u oba slučaja):

```
let obecanje = new Promise((resolve, reject) => {
  setTimeout(() => reject("Greška!"), 1000)
});

obecanje.then(rez => console.log(rez)).catch(err => console.log(err))
```

Postoji i metoda `.finally()` koja će se uvijek izvršiti pri završetku asinkrone operacije, neovisno je li ono bilo uspješno (*resolve*) ili ne (*reject*).

Ulančavanje

Jedno od važnih značajki `.then()` metode je da i ona (tj. njena *callback* funkcija) može

generirati *Promise* objekt. To u praksi znači da pri završetku jedne asinkrone operacije možemo pozvati drugu asinkronu operaciju te zatim čekati na njeno izvršavanje.

```
obecanje
  .then(rez => {
    console.log(rez)
    return new Promise((resolve, reject) => {
      setTimeout(() => resolve("Drugi!"), 1000)
    })
  })
  .then(rez => {
    console.log(rez)
  })
```

Ovaj primjer izgleda složeno pa ćemo ga prikazati na jednostavniji način, koristeći izmišljene asinkrone funkcije:

```
posaljiZahtjev()
  .then(rez => {
    obradiRezultat(rez)
  })
  .then(podaci => {
    console.log(podaci)
  })
```

U pojednostavljenom primjeru možemo vidjeti poziv funkcije *posaljiZahtjev* koja u jednom trenutku dobije traženi odgovor. Taj odgovor šaljemo drugoj funkciji pod imenom *obradiRezultat* koja je također asinkrona. Kada i ona odradi svoj zadatak i pošalje nam konačni rezultat, tek ćemo ga onda ispisati na konzoli.

Kod slučajeva gdje imamo ulančavanje, upravljanje pogreškama postaje malo složenije ali za sada se nećemo osvrtati na to. S obzirom da smo upoznali osnove izvršavanja asinkronih funkcija vrijeme je da se upoznamo sa načinima slanja zahtjeva iz naše React aplikacije prema poslužitelju.

Last updated on March 28, 2023

Stilizirane komponente

Slanje zahtjeva