Razvoj web aplikacija u realnom vremenu na primjeru naručivanja u restoranu

Bel, Filip

Undergraduate thesis / Završni rad

2020

Degree Grantor / Ustanova koja je dodijelila akademski / stručni stupanj: **University of Zagreb, Faculty of Organization and Informatics / Sveučilište u Zagrebu, Fakultet organizacije i informatike**

Permanent link / Trajna poveznica: https://urn.nsk.hr/urn:nbn:hr:211:876795

Rights / Prava: Attribution-NonCommercial-NoDerivs 3.0 Unported/Imenovanje-Nekomercijalno-Bez prerada 3.0

Download date / Datum preuzimanja: **2024-03-29**

SVEUČILIŠTE U ZAGREBU
FAKULTET ORGANIZACIJE I INFORMATIKE
VARAŽDIN

Repository / Repozitorij:

Faculty of Organization and Informatics - Digital Repository

zir.nsk.hr

dabar
DIGITALNI AKADEMSKI ARHIVI I REPOZITORIJI

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Filip Bel

Razvoj web aplikacija u realnom vremenu na primjeru naručivanja u restoranu

ZAVRŠNI RAD

Varaždin, 2020.

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Filip Bel

Matični broj: 45896/17–R

Studij: Informacijski sustavi

Razvoj web aplikacija u realnom vremenu na primjeru naručivanja u restoranu

ZAVRŠNI RAD

Mentor:

Prof. dr. sc. Danijel Radošević

Varaždin, srpanj 2020.

Filip Bel

Izjava o izvornosti

Izjavljujem da je moj završni rad izvorni rezultat mojeg rada te da se u izradi istoga nisam koristio drugim izvorima osim onima koji su u njemu navedeni. Za izradu rada su korištene etički prikladne i prihvatljive metode i tehnike rada.

Autor potvrdio prihvaćanjem odredbi u sustavu FOI-radovi

Sažetak

Tema ovog rada je razvoj web aplikacija u realnom vremenu na primjeru naručivanja u restoranu. Uvod ću započeti opisom aspekata modernog weba koji imaju značaj za razvoj web aplikacija te su relevantni temi komunikacije u realnom vremenu na webu. Potom ću objasniti koncept komunikacije u realnom vremenu na webu te ću se osvrnuti na nekoliko tehnika implementacije pomoću HTTP protokola, a potom na implementaciju pomoću WebSocket protokola. Princip rada svake od tehnika biti će objašnjen te demonstriran implementacijom koja je relevantna temi praktičnog dijela ovog rada – razvoju web aplikacije koja omogućuje naručivanje u restoranu. Nakon obrade teorije i analize tehnika za implementaciju komunikacije u realnom vremenu na webu slijedi praktični dio rada. Odabrat ću najprikladniju tehniku za izradu aplikacije koja korisnicima restorana omogućuje izradu narudžbe koja će se konobaru dinamički prikazati u realnom vremenu. Na kraju rada slijedi zaključak i komentari na obrađenu temu.

Ključne riječi: web aplikacija, realno vrijeme, AJAX, polling, Server-Sent Events, WebSocket protokol

Sadržaj

1. Uvod 1
2. Metode i tehnike rada 2
2.1. JavaScript 2
2.2. Node.js 2
2.3. Express 3
2.4. React 3
2.5. Bootstrap 4
2.6. MySQL 4
2.7. Socket.io 4
3. Moderni web 5
3.1. Dinamičke web stranice 5
3.2. Razlika web stranice i web aplikacije 7
3.3. AJAX 8
3.3.1. Implementacija 9
3.3.2. Primjena 11
4. Komunikacija u realnom vremenu na webu 12
5. Načini implementacije 14
5.1. HTTP tehnike 15
5.1.1. Polling 16
5.1.2. Long Polling 19
5.1.3. Server-Sent Events (SSE) 21
5.2. WebSocket protokol 24
6. Implementacija aplikacije 28
6.1. Opis aplikacijske domene 28
6.2. ERA model 29

6.3. Razvoj na strani poslužitelja 31
6.3.1. Aplikacijsko programsko sučelje (API) 31
6.3.2. WebSocket poslužitelj 36
6.4. Razvoj na strani korisnika 44
6.4.1. Neregistrirani korisnik 44
6.4.2. Registrirani korisnik 45
6.4.3. Konobar 46
6.4.4. Provođenje narudžbe u realnom vremenu 50
7. Zaključak 59
8. Popis literature 60
9. Popis slika 62
10. Popis tablica 64

1. Uvod

Web (eng. *World Wide Web*) postoji već 30 godina te se kroz to vrijeme znatno promijenio. Niz poboljšanja na webu promijenila su način na koji ga danas koristimo. Nekad je pristupanje web stranici značilo čitanje stranice koja je primarno sadržavala tekst te možda pokoje slike, no sada je to potpuno drukčije. Danas je izgled web stranice obogaćen CSS stilskim jezikom, pomoću JavaScript jezika omogućena je interakcija korisnika i web stranice, uvedena je dinamičnost korištenjem jezika na strani poslužitelja, podaci se dinamički prikazuju u realnom vremenu kako i nastaju itd.

Komunikacija u realnom vremenu na webu predstavlja primanje podataka na strani klijenta čim oni nastanu na poslužitelju, s minimalnim mogućim zakašnjenjem. Koncept web aplikacija u realnom vremenu postoji već dulje vrijeme, no u proteklom je desetljeću posebno privukao pažnju. Komunikacija u realnom vremenu na webu je dvosmjerna komunikacija između klijenta i poslužitelja koja može biti potpuno ili polu dvosmjerna. U počecima razvoja ideje komunikacije u realnom vremenu na webu, to je bilo moguće ostvariti jedino tehnologijom koja je tada bila dostupna – HTTP protokol. Pošto HTTP protokol ne omogućuje dvosmjernu komunikaciju, da bi se ona implementirala bilo je potrebno osmisliti neke tehnike koje će zaobići ovo ograničenje. Iako je navedeno ograničenje uspješno zaobiđeno nizom tehnika, pri čemu veliku ulogu ima AJAX tehnologija, i dalje zbog arhitekture HTTP protokola nije moguće ostvariti potpuno dvosmjernu komunikaciju, no tu ulogu preuzela je pojava WebSocket protokola koji je dizajniran isključivo za tu svrhu [1].

U ovom ću radu prvo krenuti od nekih aspekata modernog weba te ću pojasniti koncept i implementaciju komunikacije u realnom vremenu na webu. Pritom ću se osvrnuti na tradicionalne metode implementacije pomoću HTTP protokola, a potom na modernije rješenje - WebSocket protokol. Nakon pregleda dostupnih načina implementacije, odabrat ću jedan od njih koji ću iskoristiti u praktičnom dijelu rada za izradu web aplikacije pomoću koje će korisnici restorana moći izrađivati narudžbe koje će konobari primati u realnom vremenu. Što se tiče implementacija primjera, na strani poslužitelja koristit će se Node.js radna okolina te JavaScript na strani klijenta. Strana poslužitelja praktičnog dijela rada također će biti implementirana koristeći Node.js, dok ću za stranu klijenta koristiti HTML, CSS, Bootstrap Framework, JavaScript i React, o čemu će više riječi biti kasnije.

2. Metode i tehnike rada

Za izradu implementacija u teorijskom dijelu te za razvoj same aplikacije u praktičnom dijelu rada, osim HTML-a i CSS-a, korišteno je niz web tehnologija koje će u nastavku biti ukratko opisane. U teorijskom dijelu je pri izradi implementacija komunikacije u realnom vremenu na webu korišten JavaScript skriptni jezik na strani klijenta te također na strani poslužitelja koristeći radnu okolinu Node.js te Express razvojni okvir. U praktičnom dijelu rada se za razvoj aplikacije koristi JavaScript, Node.js radna okolina, Express razvojni okvir, React biblioteka, Bootstrap razvojni okvir, MySQL sustav za upravljanje bazom podataka te Socket.io biblioteka za implementaciju komunikacije putem WebSocket protokola.

2.1. JavaScript

JavaScript je skriptni objektno orijentirani programski jezik čija sintaksa nalikuje C++ jeziku koji se koristi za razvoj dinamičkih i interaktivnih web stranica. Sve što se dinamički prikazuje na web stranici bez potrebe za osvježavanjem od strane korisnika ostvareno je pomoću JavaScripta koji se pokreče u web pregledniku klijenta.

JavaScript skriptu u web pregledniku izvršava program koji se naziva JavaScript stroj (eng. *JavaScript engine*). Svaki web preglednik koji ima mogućnost izvršavanja JavaScript skripte ima ugrađen JavaScript stroj. Kao jedan od najpoznatijih se ističe Chrome V8 koji je ugrađen u preglednik Google Chrome te se također koristi u Node.js radnoj okolini [2]. Osim za razvoj web aplikacija na strani klijenta, JavaScript je u zadnje vrijeme sve zastupljeniji jezik za programiranje na strani poslužitelja (Node.js) te za razvoj mobilnih aplikacija (React Native).

2.2. Node.js

Node.js je JavaScript radna okolina otvorenog koda koja služi za pokretanje JavaScript skripte izvan web preglednika što omogućava korištenje JavaScripta na strani poslužitelja. Node.js tako može generirati dinamički sadržaj web stranice, čitati i manipulirati datotekama datotečnog sustava, spajati se na bazu podataka i još mnogo toga. Node.js ima arhitekturu temeljenu na događajima (eng. *event-driven architecture*), što će biti vidljivo prilikom implementacije WebSocket poslužitelja, koja omogućuje asinkrono procesuiranje ulaznih i izlaznih podataka (eng. *Asynchronuous I/O*). U sklopu instalacije Node.js radne okoline uključen je upravitelj paketa npm (Node Package Manager) koji služi za laku i brzu instalaciju biblioteka koje želimo uključiti u Node.js skriptu [3].

2.3. Express

Express je razvojni okvir za Node.js radnu okolinu koji je dizajniran za izgradnju web aplikacija i aplikacijskih programskih sučelja (API). Omogućuje stvaranje krajnjih točaka (eng. *endpoints*) API-ja ili web aplikacije koju izrađujemo – npr. „/user/login“, definiranje dozvoljenih metoda slanja zahtjeva prema krajnjim točkama (npr. GET, POST) te razvijanje logike za obradu zaprimljenog zahtjeva i formiranje te slanje odgovora na primljeni zahtjev [4]. Slijedi primjer krajnje točke „/user/login“ koja će prilikom dolaska GET zahtjeva klijentu vratiti odgovor „Please login“ u obliku teksta.

```
app.get('/user/login', (req, res) => {
  res.send('Please login')
})
```

2.4. React

React je deklarativna, učinkovita i fleksibilna JavaScript biblioteka otvorenog koda za izgradnju dinamičkih i interaktivnih korisničkih sučelja. Razvijena je 2013. godine od strane Facebooka te omogućuje izgradnju kompleksnih korisničkih sučelja sastavljanjem malih izoliranih komada koda nazvanih komponentama [5]. Zahvaljujući React biblioteci te ostalim bibliotekama i razvojnim okvirima za izradu dinamičkog i interaktivnog korisničkog sučelja, sve su popularnije tzv. *single-page* aplikacije (SPA) čija glavna značajka jest dinamička promjena sadržaja trenutne stranice bez potrebe za potpunim osvježavanjem.

Ključ u dinamičkoj promjeni sadržaja web stranice je manipulacija DOM-om. Problem u tome jest taj što manipulacija DOM-om može biti spora ukoliko se stranica sastoji od jako velike količine elemenata, što je danas slučaj za single-page aplikacije, zato što je potrebno duže vremena da se pronađe traženi element koji se želi ažurirat. Taj problem React rješava korištenjem virtualnog DOM-a. Naime, virtualni DOM je JavaScript objekt koji je kopija stvarnog DOM-a stranice te su operacije za manipulacijom tog objekta brze u usporedbi s manipulacijom stvarnog DOM-a. Na taj način ukoliko je potrebno ažurirati npr. 50 elemenata u DOM-u, prvo se oni pronađu i ažuriraju u virtualnom DOM-u te nakon toga React uspoređuje virtualni DOM i stvarni DOM te na temelju primijećenih promjena ažurira stvarni DOM. Na taj način React pruža odlične performanse, što je i jedna od ključnih značajki ove biblioteke [6].

Za izradu prezentacijskog dijela React komponenata koristi se JSX (JavaScript Syntax Extension) jezik koji sintaksom jako nalikuje na HTML/XML. JSX nam omogućuje stvaranje

vlastitih oznaka koje enkapsuliraju funkcionalnost ostalih komponenti (Navbar i Footer u primjeru koji slijedi) te korištenje JavaScript koda unutar JSX elemenata [7]. Slijedi primjer jedne komponente koja uključuje komponentu Navbar i Footer te ispisuje naslov i trenutno vrijeme (timestamp).

```
const Component = () => {
    return (
        <div>
            <Navbar />
            <h1>Title</h1>
            <p>{Date.now()}</p>
            <Footer />
        </div>
     );
}
```

2.5. Bootstrap

Bootstrap je jedan od najpopularnijih razvojnih okvira koji se koristi za razvoj responzivnih web stranica. Sadrži CSS i JavaScript predloške dizajna za tipografiju, forme, gumbe i ostalih elemenata korisničkog sučelja. Zahvaljujući jednostavnošću korištenja omogućuje brz razvoj responzivnih web aplikacija modernog dizajna [8].

2.6. MySQL

MySQL je jedan od najpopularnijih sustava za upravljanje relacijskim bazama podataka koji koristi strukturni upitni jezik (SQL) za dodavanje novih i manipulaciju postojećih podataka.

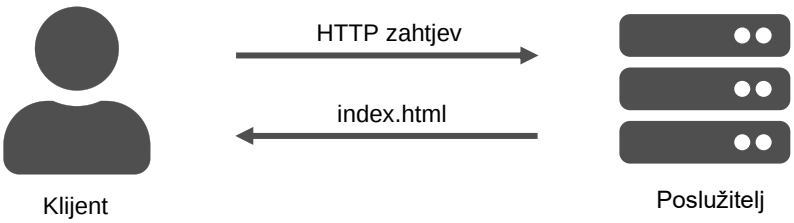
2.7. Socket.io

Socket.io je JavaScript biblioteka koja omogućuje razvoj web aplikacija s komunikacijom u realnom vremenu pomoću WebSocket protokola. Iako za razvoj WebSocket poslužitelja nisu potrebne dodatne biblioteke, zahvaljujući mnoštvo praktičnih značajki i dodatnih mogućnosti, ova biblioteka to uvelike olakšava. Neke od mogućnosti u razvoju WebSocket poslužitelja koje ova biblioteka pruža su kreiranje soba (eng. *rooms*) na koje se korisnici mogu spojiti, definiranje vlastitih događaja odnosno tipova poruka koje će se razmjenjivati, emitiranje poruka prema više klijenata odjednom, pohrana podataka povezanih sa svakim klijentom, asinkrono procesuiranje ulaznih i izlaznih podataka itd. [9]

3. Moderni web

3.1. Dinamičke web stranice

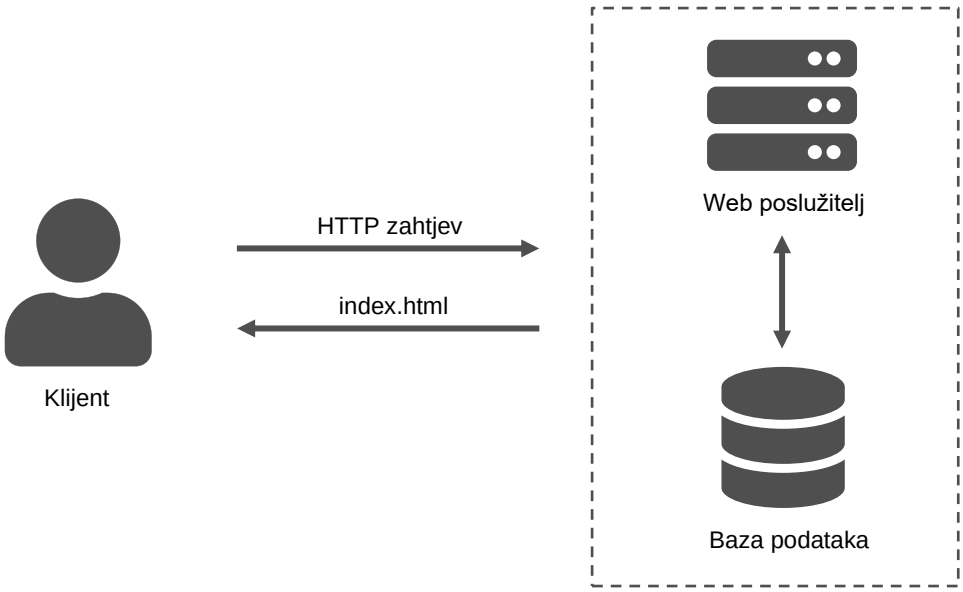
S obzirom na dinamičnost, web stranice mogu biti statičke i dinamičke. Statičke web stranice su one stranice koje na svaki korisnikov zahtjev pružaju jednak sadržaj. Na primjer ukoliko želimo napraviti statičku web stranicu koja će korisniku prikazivati jelovnik restorana, u datoteku jelovnik.html ćemo pomoću HTML-a fiksno napisati kod kojim ćemo korisniku prikazati jelovnik. Svaki puta kada korisnik pristupi toj web stranici, poslužitelj će pročitati datoteku jelovnik.html te će njen sadržaj poslati korisniku (slika 1). Tako dugo dok se datoteka jelovnik.html ne promijeni na poslužitelju, korisnik će primati isti sadržaj, odnosno jelovnik. No, što ako je jelovnik potrebno mijenjati svaki tjedan? Da bi se korisniku prikazao novi jelovnik, tada bismo koristeći ovaj princip svaki tjedan trebali mijenjati sadržaj datoteke jelovnik.html.



Slika 1: Prikaz dohvaćanja statične web stranice

S druge strane, dinamičke web stranice su one koje nisu pohranjene u formatu koji web preglednik može direktno prikazati korisniku, već je prilikom dolaska zahtjeva od strane klijenta potrebno izvesti dodatnu operaciju koja će generirati sadržaj u formatu koji web preglednik može prikazati korisniku. Navedena dodatna operacija određena je skriptnim ili programskim jezikom – npr. PHP, Python, Java, JavaScript (Node.js) [10]. Proces pristupanju dinamičkoj web stranici je slijedeći: korisnik poslužitelju šalje zahtjev nakon čega poslužitelj interpretira npr. Node.js skriptu koja generira HTML sadržaj te se taj sadržaj kao odgovor šalje klijentu koji se zatim prikazuje u njegovom web pregledniku. Vratimo se prethodnom primjeru – želimo napraviti dinamičku web stranicu koja će prikazivati jelovnik restorana koji se mijenja na tjednoj bazi. Ovim pristupom možemo u web stranicu ukomponirati bazu podataka gdje ćemo

spremati jelovnik, a Node.js skripta će se spajati na bazu podataka, pročitati ovotjedni jelovnik te generirati sadržaj (HTML datoteku) koji će se kao odgovor poslati klijentu (slika 2).



Slika 2: Prikaz dohvaćanja dinamične web stranice

HTTP je izvorno bio dizajniran za posluživanje statičkih međusobno povezanih multimedijskih dokumenata, odnosno hiperteksta, no danas ćemo rijetko naići na web stranicu koja je statička, a razlog tome je niz prednosti dinamičkih web stranica.

Jedna od glavnih prednosti dinamičkih stranica je jednostavno ažuriranje sadržaja. Npr. ukoliko želimo napraviti blog, primjena koncepta dinamičke web stranice omogućuje nam dinamičko čitanje svih objava iz baze podataka u koju ćemo ih spremati. Na taj način možemo objave unositi putem forme na web stranici te će se one korisniku prikazivati iz baze podataka. Nadalje, jedan od danas bitnijih razloga korištenja dinamičkih web stranica jest prikaz sadržaja s obzirom na korisničku ulogu – npr. ako želimo da se običnom korisniku prikažu sve objave na blogu, a administratoru želimo dati mogućnost dodavanja novih objava te uređivanje postojećih. Još jedna prednost, koja znatno utječe na korisničko iskustvo, jest interaktivnost. Naime, korisnici mogu s dinamičkim web stranicama imati interakciju koja znatno utječe na korisničko iskustvo – npr. filteri pretraživanja mogu sadržaj koji će biti prikazan prilagoditi potrebama korisnika [11].

3.2. Razlika web stranice i web aplikacije

Danas se često spominje pojam web aplikacije, no što taj pojam predstavlja te kako se razlikuje od web stranice? S obzirom na to da je većina današnjih web stranica dinamička, pojmovi web stranice i web aplikacije se često poistovjećuju, no razlika zapravo postoji.

Glavna razlika ovih pojmova jest ta što je web stranica informativnog tipa, dok je web aplikacija dizajnirana s ciljem interakcije s krajnjim korisnikom. Uzmimo za primjer web stranicu nekog restorana. Ako ta web stranica sadrži samo podatke o npr. nazivu restorana, adresi, jelovniku, broju telefona te ne omogućuje neku pretjeranu interakciju s korisnikom, tada je to web stranica informativnog karaktera. S druge strane, ako bi ta web stranica imala mogućnosti registracije korisnika te potom provođenje rezervacije stola, naručivanje jela, slanje upita, ocjenjivanje jela itd., tada bismo tu web stranicu mogli kategorizirati kao web aplikaciju. Danas su web aplikacije sve zastupljenije, a razlog tome su brojne prednosti istih [12].

U usporedbi s web stranicama, web aplikacije pružaju bolje korisničko iskustvo pošto su interakcija i dinamičnost na višoj razini. Korisnici ne samo da čitaju sadržaj nego i manipuliraju njime – dodaju ga, brišu i izmjenjuju. Uzmimo za primjer web aplikaciju Facebook - korisnici sami pišu objave, oni odlučuju o tome koji će sadržaj aplikacije biti, a programeri određuju mogućnosti i ograničenja korisnika u stvaranju sadržaja.

Razvoj web aplikacija može se podijeliti na dvije cjeline – programiranje na strani poslužitelja (eng. *backend development*) i programiranje na strani klijenta (eng. *frontend development*). Programiranje na strani klijenta određuje kako će aplikacija izgledati što uključuje razvoj korisničkog sučelja primarno korištenjem jezika HTML, CSS i Javascript te nekih razvojnih okvira kao što su React ili Vue. Programiranje na strani poslužitelja određuje kako će aplikacija funkcionirati (npr. kako će se spajati na bazu podataka i unositi podatke), odnosno sadrži svu logiku aplikacije koja vizualno nije vidljiva krajnjem korisniku. Logika aplikacije programira se u jezicima na strani poslužitelja kao što su PHP, Ruby, Python, JavaScript (Node.js).

3.3. AJAX

AJAX (eng. *Asynchronous JavaScript And XML*) je pojam čiji naziv je 2005. godine predstavio Jesse James Garrett te označava skup tehnologija koje web aplikaciji omogućuju asinkronu razmjenu podataka s poslužiteljem na način da u pozadini bez osvježavanja cijele stranice šalje zahtjeve poslužitelju, prima tražene podatke u odgovoru poslužitelja te na temelju primljenih podataka ažurira svoj sadržaj. Kako bi to bilo moguće, AJAX se sastoji od slijedećih tehnologija [13]:

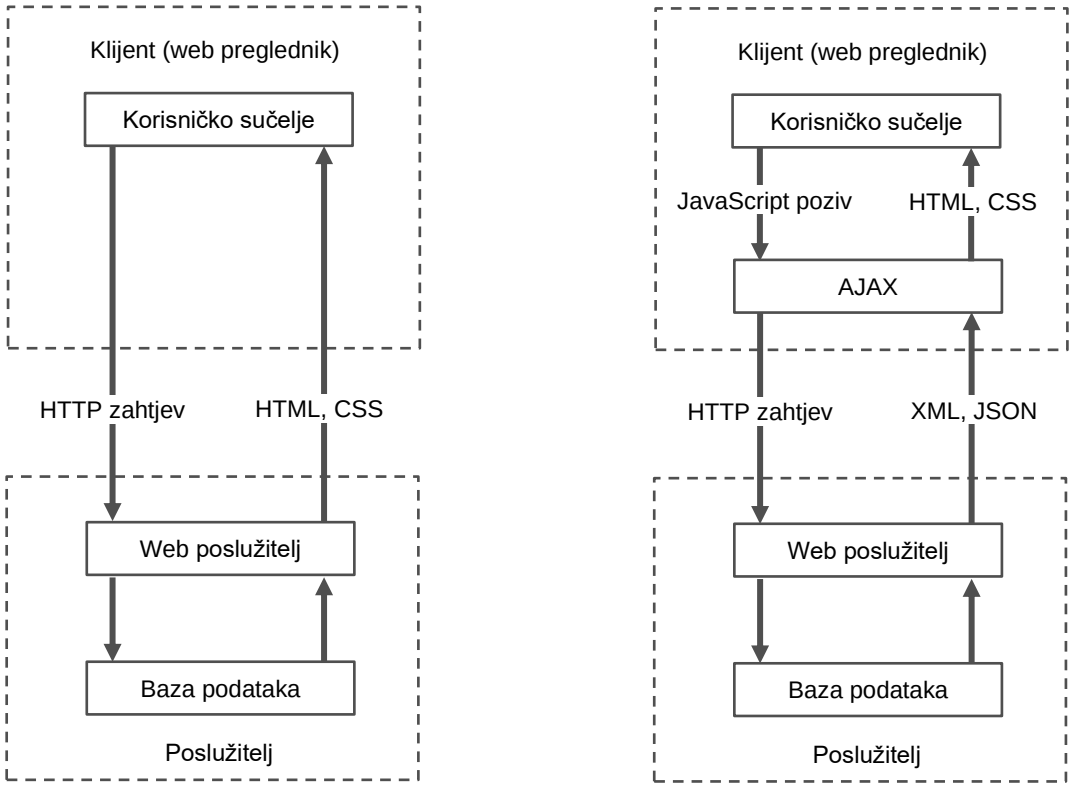
- HTML (XHTML) i CSS za prikaz i oblikovanje sadržaja,
- DOM (Document Object Model) za dinamički prikaz i interakciju sa sadržajem,
- XML ili JSON za prijenos podataka između klijenta i poslužitelja
- XMLHttpRequest objekt za asinkronu komunikaciju
- JavaScript koji objedinjuje sve navedene tehnologije

Ipak, glavni aspekt AJAX tehnologije je XMLHttpRequest objekt koji služi za asinkrono komuniciranje s poslužiteljem koristeći HTTP protokol. Iako XML stoji u nazivu ovog skupa tehnologija, format prijenosa podataka nije ograničen samo na XML format. Naime, danas se za tu svrhu većinom koristi JSON format, no može se i običan tekst.

Princip korištenja ovog skupa tehnologija je slijedeći: u nekom trenutku, ili nakon nekog događaja (npr. kada korisnik klikne na gumb), htjeti ćemo dohvatiti podatke s poslužitelja - npr. konobar će htjeti dohvatiti nove narudžbe koje su poslali korisnici. Pritiskom na gumb poziva se JavaScript funkcija koja asinkrono šalje HTTP zahtjev poslužitelju na što on vraća odgovor u JSON formatu koji sadrži narudžbe koje su korisnici napravili. Odgovor se pomoću JavaScripta pročita te se dinamički (bez osvježavanja cijele stranice) ažurira DOM web aplikacije na temelju primljenih podataka o narudžbama – dodaju se HTML elementi koji mogu biti oblikovani pomoću CSS-a. Primjećujemo da smo ovim primjerom pomoću JavaScripta objedinili sve spomenute tehnologije (slika 3).

AJAX koncept postao je prepoznatljiv kada ga je Google uveo u svoje aplikacije Google Mail i Google Maps 2004. godine koje su dobri primjeri primjene ovih tehnologija [14]. Prilikom učitavanja Google Maps aplikacije prikazuje se ograničeno područje na karti koje trenutno korisnik vidi, no kako se korisnik pomiče po karti tako njegov preglednik asinkrono šalje zahtjeve prema poslužitelju za dohvaćanje podataka o području na karti do kojeg se

pomaknuo. Kada stigne odgovor od poslužitelja s podacima od tom području, JavaScript u korisnikovom pregledniku ažurira DOM aplikacije te se tako prikazuje novo područje na karti.



Slika 3: Prikaz rada tradicionalne web aplikacije (lijevo) i web aplikacije korištenjem AJAX tehnologija (desno) [13]

3.3.1. Implementacija

Uzmimo slijedeći primjer – želimo napraviti web aplikaciju kojoj ćemo konobarima restorana klikom na gumb omogućiti dohvaćanje narudžbi koje su korisnici napravili. Nakon što konobar klikne na gumb, narudžbe se dinamički dohvaćaju i prikazuju bez osvježavanja stranice.

U praksi bi rješenje ovog primjera izgledalo ovako – nakon što konobar klikne na gumb, pomoću AJAX-a se poslužitelju šalje zahtjev za dohvaćanje narudžbi. Poslužitelj zatim dohvaća narudžbe iz baze podataka te ih u odgovoru vraća klijentu. Zbog jednostavnosti demonstracije u ovom primjeru, izostavit ćemo bazu podataka te će se zahtjev slati prema JSON datoteci u kojoj se nalaze podaci o narudžbama. Slijedi sadržaj navedene datoteke:

```
[
  {
    "id_narudzbe": 1,
    "datum_vrijeme": "2020-04-15T04:30:37.000Z",
    "stol": 1,
    "iznos": 125
  },
  {
    "id_narudzbe": 2,
    "datum_vrijeme": "2020-04-15T04:36:20.000Z",
    "stol": 2,
    "iznos": 170
  }
]
```

Postoji nekoliko načina implementacije AJAX-a. Tradicionalan način implementacije je pomoću XMLHttpRequest objekta za koji nisu potrebne dodatne biblioteke. Slijedi implementacija i objašnjenje funkcije koja se poziva klikom na gumb za dohvaćanje narudžbi:

```
1   const getOrders = () => {
2     const xhr = new XMLHttpRequest();
3     xhr.open("GET", "./orders.json", true);
4
5     xhr.onreadystatechange = () => {
6       if (xhr.readyState === XMLHttpRequest.DONE) {
7         const orders = JSON.parse(xhr.responseText);
8         ispisiNarudzbe(orders);
9       }
10    };
11    xhr.send(null);
12  };
```

U liniji 2 napravljen je XMLHttpRequest objekt te je u idućoj liniji pozvana funkcija open koja redom prima slijedeće argumente - naziv metode kojom se šalje zahtjev, URL mjesta na koji se zahtjev šalje te opcija šalje li se zahtjev asinkrono (true) ili sinkrono (false). Svaki XMLHttpRequest ima svoje stanje (eng. *state*) u kojem se nalazi. Prije nego što je zahtjev poslan, XMLHttpRequest objekt nalazi se u stanju UNSENT. Nakon što je zahtjev poslan te je u tijeku preuzimanje tijela odgovora poslužitelja, stanje se mijenja u LOADING. Nakon što je primljen odgovor poslužitelja, XMLHttpRequest prelazi u stanje DONE [15]. U liniji 5 napravljen je osluškivač onreadystatechange događaja koji se poziva svaki puta kada se stanje XMLHttpRequest objekta promijeni. Pošto želimo prikazati primljene narudžbe nakon što zaprimimo odgovor od poslužitelja, zanima nas samo provjera kada je objekt u stanju DONE, što provjeravamo u liniji 6. Nakon što je odgovor primljen, isti se pretvara u JSON format (linija

7) te se u idućoj narudžbi poziva funkcija za ispis dohvaćenih narudžbi. U liniji 11 šaljemo pripremljeni zahtjev.

Ovo je bila implementacija na tradicionalan način, kako je to prvotno bilo i zamišljeno. Iako se i danas koristi ovaj način implementacije, sve su više prisutne modernije implementacije AJAX-a kao npr. funkcija fetch() ili razne biblioteke. Jedna od najpoznatijih JavaScript biblioteka jQuery ima ugrađenu funkciju za slanje asinkronih zahtjeva koja implementaciju AJAX-a čini još jednostavnijom:

```
1  $.getJSON("./orders.json").then((orders) => ispisiNarudzbe(orders));
```

3.3.2. Primjena

AJAX je danas osnova web aplikacija te su zahvaljujući tome one interaktivne i dinamične. Jedna od široko korištenih primjena ovih tehnologija je validacija formi. Često se susrećemo s web aplikacijama kod kojih prilikom registracije odmah nakon upisa e-mail adrese dobijemo povratnu informaciju o tome je li ona već iskorištena za neki drugi korisnički račun, a ne tek kada pritisnemo na gumb za registriranje nakon čega se stranica osvježi da bi nas obavijestila o tome da je unesena e-mail adresa već zauzeta. Način na koji to funkcionira jest da nakon unosa e-mail adrese u polje, naš preglednik asinkrono šalje zahtjev prema poslužitelju te tako provjerava je li unesena adresa već u upotrebi. Još jedan primjer s kojim se svakodnevno susrećemo je automatska ispuna unosa (eng. *autocomplete*) koju koristi Google tražilica. Naime, kada započnemo s unosom nekog pojma u tražilicu, asinkrono se šalje zahtjev za pretraživanje pojmova na poslužitelj te se prima odgovor koji sadrži neke prijedloge pretrage koji se zatim korisniku dinamički prikažu. Mnoge web aplikacije kao što su Facebook ili Twitter omogućuju pregled obavijesti u realnom ili skoro realnom vremenu što je moguće implementirati korištenjem AJAX tehnologija o čemu će biti više riječi kasnije [16].

4. Komunikacija u realnom vremenu na webu

Osim već spomenute dinamičnosti i interaktivnosti web aplikacija, svakodnevno se susrećemo s još jednom tehnologijom modernog weba koja u posljednje vrijeme privlači sve više pažnje – komunikacijom u realnom vremenu.

Pojam realnog vremena odnosi se na vremensku prirodu između pojave događaja i trenutka kada smo mi svjesni tog događaja. Mnogi događaji s kojima se svakodnevno susrećemo tako zahtijevaju nekakvu reakciju u realnom vremenu – bio taj događaj pritisak automobilske kočnice ili pritisak na sklop za paljenje svjetla, očekujemo da će se instantno pokrenuti neka akcija koje ćemo biti svjesni – auto će se početi zaustavljati, odnosno upalit će se žarulja. Realno vrijeme na webu funkcionira na isti princip – klijenti žele biti svjesni novih događaja na poslužitelju (dostupnosti novih podataka, npr. nove narudžbe) čim oni nastanu na poslužitelju, s minimalnim mogućim zakašnjenjem. Jednostavno rečeno, komunikacija u realnom vremenu na webu je pružanje ažuriranih podataka od strane poslužitelja odmah čim su oni dostupni, bez potrebe klijenta za ponovnim osvježavanjem stranice [17].

Komunikacija u realnom vremenu može činiti glavni koncept same aplikacije – npr. aplikacije za brzu razmjenu poruka (eng. *chat*), glasovna komunikacija putem interneta (npr. Skype) itd. S druge strane, značajka komunikacije u realnom vremenu neke aplikacije može biti samo njen dio koji ju nadopunjuje i povećava korisničko iskustvo korištenja te aplikacije. Značajke aplikacija koje spadaju u drugu skupinu su npr. obavijesti i popis novih objava na Facebooku koji se korisniku pojavljuju dinamički u realnom vremenu kako i nastaju, bez potrebe za osvježavanjem stranice.

Prema smjeru odvijanja, komunikacija u realnom vremenu spada u dvosmjerne komunikacije (eng. *duplex*) pošto obje strane (klijent i poslužitelj) mogu međusobno slati poruke. Nadalje, dvosmjernu komunikaciju možemo podijeliti na polu-dvosmjernu (eng. *half duplex*) i potpuno dvosmjernu (eng. *full duplex*). Polu-dvosmjerna komunikacija omogućuje komunikaciju u oba smjera, ali ne istodobno nego se u jednom trenutku podaci mogu slati samo u jednom smjeru. Primjer polu-dvosmjerne komunikacije je komunikacija putem Walkie-Talkie uređaja u kojoj osoba može govoriti (slati podatke) tek kada druga osoba završi s govorom. S druge strane, potpuno dvosmjerna komunikacija je komunikacija koja omogućuje istodobnu razmjenu informacija u oba smjera (npr. razgovor uživo s drugom osobom) [18].

Komunikacija u realnom vremenu ima veliku važnost jer u ovom dobu širokog prihvaćanja pametnih telefona korisnici od usluga za razmjenu informacija očekuju odgovor na njihove zahtjeve u realnom vremenu – žele biti obaviješteni o novostima u trenutku kada one nastanu, poruke žele primiti istog trenutka kada su poslane od strane drugih korisnika itd.

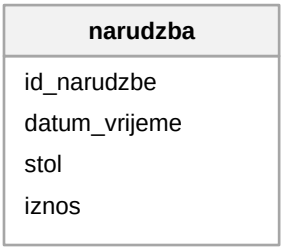
Iako nam je web dao mogućnost razmjene podataka s drugim korisnicima, može proći dosta vremena prije nego što korisnik zaprimi podatke koji su mu poslani. Uzmimo za primjer forume koji su u početku weba bili popularni za razmjenu podataka. Kada bi netko napisao novu objavu na forumu, istu će drugi korisnik vidjeti tek kada ponovno osvježi stranicu – to može biti odmah čim je objava napravljena, no može biti i dosta kasnije, ali nije garantirano da će nova objava stići do njega u isto vrijeme kada će biti i napravljena.

Komunikacija u realnom vremenu nam omogućuje brzu razmjenu podataka – objavu odnosno poruku primatelj prima u realnom vremenu kada je ona i nastala. Osim brže razmjene podataka, poboljšanje korisničkog iskustva je također jedna od prednosti web aplikacija u realnom vremenu. Intuitivnije je i praktičnije za korisnika dinamičko prikazivanje novih podataka (npr. novih narudžbi u restoranu), bez potrebe za osvježivanjem web aplikacije.

5. Načini implementacije

Komunikacija u realnom vremenu funkcionira na način da se klijentu šalju podaci vremenski kako se i pojavljuju odnosno ažuriraju na serveru. Problem je taj što HTTP protokol izvorno nije bio namijenjen ovom načinu komunikacije – poslužitelj ne može sam slati podatke prema klijentu nego je klijent taj koji mora inicijalizirati komunikaciju, a poruke poslužitelja su odgovori na klijentove upite. Kako bi se zadovoljila rastuća potreba za aplikacijama ovog tipa, korišteno je nekoliko tehnika. Sve do nedavno korištenje HTTP protokola na netradicionalan način je bio uobičajen način postizanja komunikacije u realnom vremenu na webu. Pojavom WebSocket protokola korištenje HTTP protokola za komunikaciju u realnom vremenu postala je stara, no još uvijek znatno korištena tehnika. Slijedi objašnjenje te implementacija nekoliko HTTP načina implementacije komunikacije u realnom vremenu na webu te pregled modernog rješenja problema potpuno dvosmjerne komunikacije – WebSocket protokola. [17]

Za demonstraciju implementacije tehnika koje će biti spomenute osvrnut ćemo se na ranije spomenut primjer dohvaćanja novih narudžbi, no ovog ćemo puta scenariju dodati bazu podataka. Imat ćemo bazu podataka koja će se zbog jednostavnost sastojati samo od jedne relacije u koju se spremaju narudžbe klijenata nekog restorana (slika 4). Svaka narudžba ima svoj jedinstven broj (*id_narudzbe*), datum i vrijeme (*datum_vrijeme*), oznaku stola na kojem je napravljena (*stol*) te ukupan iznos (*iznos*).



Slika 4: Relacija „narudzbe“ u koju se spremaju narudžbe

Pretpostavimo da već neki dio aplikacije upisuje narudžbe u ovu relaciju kada ih klijenti provedu. Zadatak nam je pomoću navedenih tehnika izraditi web aplikaciju koja će dohvaćati i prikazivati narudžbe u realnom vremenu kako one nastaju. Istražit ćemo i analizirati način rada te prednosti i nedostatke svake tehnike. Strana poslužitelja će biti implementirana pomoću Node.js radne okoline koja omogućuje pokretanje JavaScript koda izvan web preglednika, odnosno na poslužitelju. Strana klijenta biti će implementirana kao web aplikacija koristeći JavaScript.

5.1. HTTP tehnike

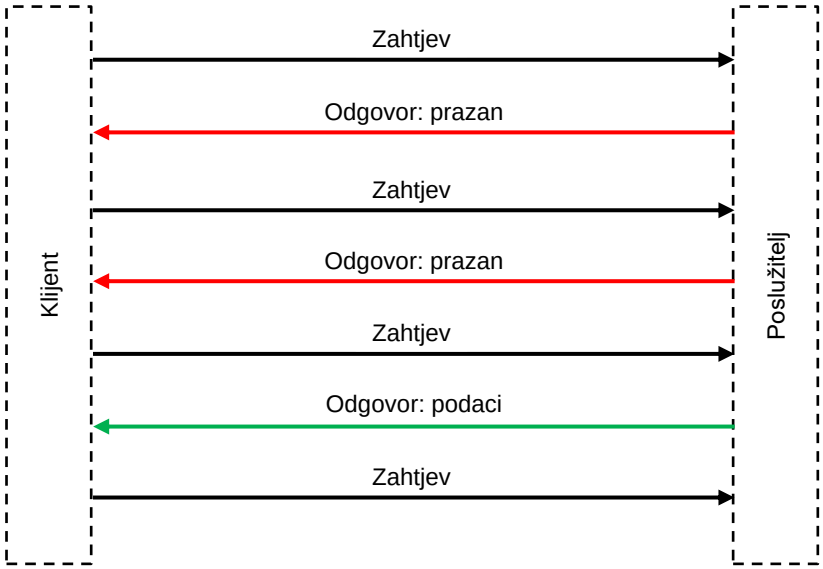
HTTP (eng. *Hypertext Transfer Protocol*) je protokol koji služi za razmjenu podataka na webu između klijenta i poslužitelja koji funkcionira na principu para zahtjev-odgovor. Klijent (web preglednik) tako šalje poslužitelju zahtjev za dohvaćanje sadržaja web stranice te on u odgovoru na taj zahtjev vraća traženi sadržaj. Svaki zahtjev sadrži naziv metode kojom se zahtjev šalje, verziju HTTP protokola koji se koristi te zaglavlje koje uključuje podatke kao što su izvorišna adresa, tip podataka koji se očekuje u odgovoru itd. Odgovor osim podataka koji se prenose također sadrži neke dodatne informacije - verziju HTTP protokola, statusni kod i zaglavlje. Stoga je izgledno da se osim samih podataka HTTP protokolom prenose i neki drugi podaci koji, ukoliko se šalju kratke poruke, mogu činiti velik udio ukupnog prometa razmjene podataka [19].

HTTP se ne temelji na stalno otvorenoj vezi između klijenta i poslužitelja za vrijeme razmjene podataka. Klijenti šalju zahtjeve poslužiteljima koji ih potom obrađuju te šalju odgovor. Nakon što klijent primi odgovor, veza između klijenta i poslužitelja se prekida. Za slanje svakog idućeg zahtjeva, odnosno za svaku iduću razmjenu poruke, potrebno je uspostaviti novu vezu. S obzirom da se kod svake razmjene poruke otvara nova veza pri čemu se svakim zahtjevom klijenta i odgovorom poslužitelja osim samih podataka šalju i zaglavlja, ovo predstavlja glavni nedostatak komunikacije u realnom vremenu putem HTTP protokola pošto zaglavlja stvaraju dodatan teret ukupnim podacima koji se razmjenjuju zbog čega je i latencija u razmjeni podataka veća [19].

Podaci HTTP protokolom teku jednosmjerno od strane poslužitelja prema klijentu čemu prethodi zahtjev klijenta za podacima. S obzirom da HTTP ne omogućuje dvosmjernu komunikaciju, implementacija komunikacije u realnom vremenu koristeći HTTP protokol temelji se na tehnikama koje ovu prepreku zaobilaze često koristeći AJAX tehnologije (polling i long polling tehnike). Slijedi opis i implementacija često korištenih tehnika komunikacije u realnom vremenu koristeći HTTP protokol [17].

5.1.1. Polling

Polling je način implementacije komunikacije u realnom vremenu koristeći HTTP protokol koji funkcionira na način da klijent, odnosno web stranica, periodički u intervalima dohvaća podatke s poslužitelja tako da mu šalje HTTP zahtjeve na koje potom dobiva odgovor u obliku traženih podataka. Odgovor poslužitelja također može biti prazan ukoliko u trenutku slanja zahtjeva od strane klijenta ne raspolaže novim podacima od prošlog odgovora. Ukoliko novi podaci postoje, oni su vraćeni klijentu kao odgovor na poslan zahtjev (slika 5). Ovaj se način, ukoliko klijent šalje zahtjeve dovoljno često, može percipirati kao komunikacija u realnom vremenu. Komunikacija npr. u chat aplikaciji se ne smatra komunikacijom u realnom vremenu ako se nove poruke dohvaćaju svakih nekoliko minuta, no ako taj interval smanjimo na jednu sekundu tada već možemo reći da aplikacija pruža komunikaciju u realnom vremenu, iako postoje načini učinkovitog smanjenja tog intervala što ćemo saznati u nastavku [20].



Slika 5: Prikaz komunikacije klijenta i poslužitelja tehnikom polling

Ova tehnika je poprilično jednostavna te se implementira pomoću AJAX-a pošto se podaci s poslužitelja periodički dohvaćaju u pozadini, a prilikom zaprimanja istih oni se korisniku prikazuju na način da se ažurira DOM same aplikacije.

Slijedi implementacija polling tehnike na ranije objašnjenom scenariju. Poslužitelj će prilikom dolaska zahtjeva od klijenta provjeriti ima li novih narudžbi u bazi podataka. Ukoliko nove narudžbe postoje, podaci o istima će se vratiti klijentu dok će u protivnom klijent primiti prazan odgovor. S druge strane, klijent će periodički u intervalima od dvije sekunde poslužitelju slati zahtjev za novim narudžbama. Slijedi kod krajnje točke „/orders“ na poslužitelju:

```
1  app.get("/orders", (req, res) => {
2
3    let lastId = parseInt(req.query.lastId);
4
5    db.query("SELECT MAX(id_narudzbe) AS lastId FROM narudzba",
6     (err, result) => {
7      if (err) throw err;
8      if (lastId == -1)
9        return res.json({ lastId: result[0].lastId, result: [] });
10
11      db.query("SELECT * FROM narudzba WHERE id_narudzbe > ?", lastId,
12     (err, result) => {
13        if (err) throw err;
14        if (result.length > 0)
15         res.json({ lastId: lastId + result.length, result });
16        else
17          res.end();
18      });
19    });
20  });
```

Svaki zahtjev na ovu točku mora sadržavati parametar „lastId“. Kao početnu vrijednost ovog parametra klijent šalje -1. Nakon što poslužitelj primi zahtjev s vrijednošću parametra „lastId“ -1, u bazi se provjerava identifikator (id_narudzbe) zadnje spremljene narudžbe koji se kao odgovor vraća klijentu (linija 9). Klijent zatim pri slanju svakog idućeg zahtjeva poslužitelju šalje taj identifikator kao „lastId“ parametar na što poslužitelj odgovara podacima svih narudžbi koje su napravljene nakon narudžbe čiji identifikator je jednak „lastId“ parametru (linija 15) u slijedećem JSON formatu:

```
{
  "lastId": 45,
  "result": [
    {
      "id_narudzbe": 45,
      "datum_vrijeme": "2020-04-15T04:30:37.000Z",
      "stol": 2,
      "iznos": 100
    }
  ]
}
```

Ukoliko nema novih narudžbi nakon one koju je klijent zadnju primio, klijent će od poslužitelja primiti prazan odgovor (linija 17) te će poslati upit ponovo nakon dvije sekunde.

Slijedi pojašnjenje implementacije na strani klijenta. Funkcija u liniji 3-10 ponavlja se svakih dvije sekunde (linija 12) te se u njoj poziva jQuery funkcija koja šalje AJAX zahtjev prema poslužitelju pri čemu uključuje „lastId“ parametar čija je početna vrijednost -1 (linija 4). Ukoliko poslužitelj odgovori s podacima o novim narudžbama, iz odgovora se uzima identifikator posljednje narudžbe te se sprema u varijablu „lastId“ (linija 6) koja će biti uključena kao parametar u slanju idućeg zahtjeva prema poslužitelju. U liniji 7 se primljeni podaci prosljeđuju funkciji prikaziNarudzbe koja korisniku ispisuje primljene narudžbe.

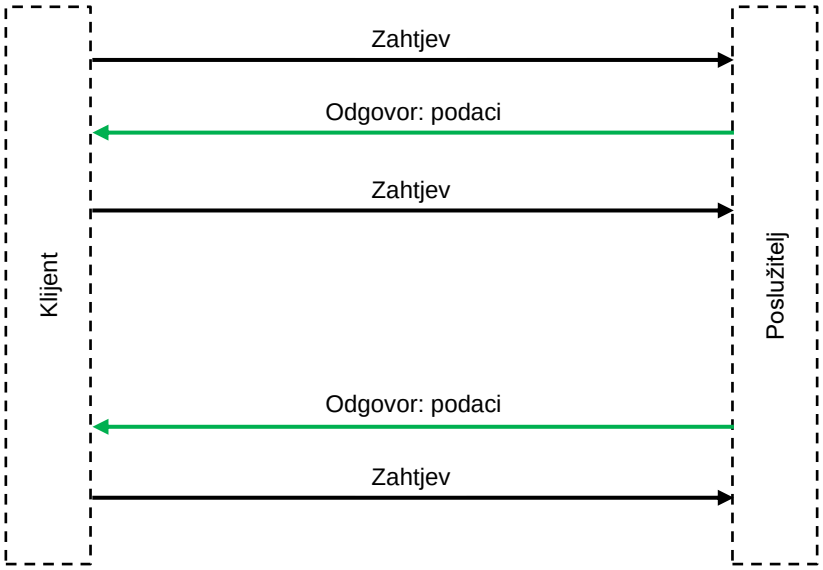
```
1   let lastId = -1;
2
3   const polling = () => {
4      $.get(`/orders?lastId=${lastId}`, (response) => {
5         if (response) {
6            lastId = response.lastId;
7            prikaziNarudzbe(response);
8         }
9       });
10  };
11
12  setInterval(polling, 2000);
```

Nedostatak ovog načina jest to što je izuzetno neefikasan. Prvi problem je taj što će mnoštvo poslanih zahtjeva od strane klijenta primiti prazan odgovor, odnosno u vrijeme slanja zahtjeva na poslužitelju neće postojati novi podaci pa će tako klijent primiti prazan odgovor što znači da smo potrošili mrežne i procesorske resurse, a nismo dobili nikakve korisne podatke. Drugi problem jest taj što potrošnja resursa (mrežni promet i procesorsko vrijeme) direktno ovisi o prihvatljivom kašnjenju u isporuci podataka s poslužitelja. Npr. ukoliko je prihvatljivo kašnjenje nisko (nekoliko sekundi), tada frekvencija slanja zahtjeva za novim podacima mora biti veća što znači da će se u određenom intervalu poslati više zahtjeva pa će prema tome potrošnja resursa biti veća [1].

Prednost ove tehnike je jednostavnost implementacije i podržanost od strane svih modernih web preglednika [1].

5.1.2. Long Polling

U odnosu na tradicionalan polling, long polling je tehnika koja nastoji minimizirati kašnjenje u isporuci novih podataka s poslužitelja te potrošnju mrežnih i procesorskih resursa. Promjena koja omogućuje ovu prednost u odnosu na „običan“ polling jest ta što poslužitelj na zaprimljen zahtjev ne odgovara odmah, potencijalno s praznim odgovorom, već tek kada na raspolaganju ima podatke koje klijentu može poslati kao odgovor. Nakon što poslužitelj pošalje odgovor s novim podacima, klijent podatke zaprima te odmah šalje novi zahtjev. Poslužitelj ponovo čeka s odgovorom na novi zahtjev tako dugo dok ne bude imao nove podatke koje klijent očekuje u odgovoru na poslani upit (slika 6) [20].



Slika 6: Prikaz komunikacije klijenta i poslužitelja tehnikom long polling

S obzirom da veza između klijenta i poslužitelja ostaje otvorena sve dok poslužitelj ne pošalje odgovor, moguće je da će veza ostati otvorena duže vrijeme ako poslužitelj neće imati novih podataka koje kao odgovor može poslati klijentu. U tom slučaju je moguće da zahtjev istekne (eng. *timeout*) pa je bitno na strani klijenta provjeravati je li zahtjev istekao te ako je, potrebno je poslati novi zahtjev poslužitelju.

Implementacija long pollinga također je realizirana pomoću AJAX tehnologija te se temelji na „običnom“ pollingu uz neke promjene. Slijedi prikaz i objašnjenje koda na strani poslužitelja.

```
1   app.get("/orders", (req, res) => {
2     let lastId = parseInt(req.query.lastId);
3
4     db.query("SELECT MAX(id_narudzbe) AS lastId FROM narudzba",
5       (err,  result) => {
6        if (err) throw err;
7        if (lastId == -1)
8          return res.json({ lastId: result[0].lastId, result: [] });
9        const dohvacanje = setInterval(() => {
10         db.query("SELECT * FROM narudzba WHERE id_narudzbe > ?", lastId,
11         (err, result) => {
12           if (err) throw err;
13           if (result.length > 0) {
14             res.json({ lastId: lastId + result.length, result });
15             clearInterval(dohvacanje);
16           }
17         });
18       }, 2000);
19    });
20  });
```

Što se tiče strane poslužitelja, jedina promjena u odnosu na prethodnu tehniku jest ta što je dohvaćanje novih narudžbi implementirano na način da se one provjeravaju slanjem upita bazi podataka u intervalima od dvije sekunde. Ukoliko postoje nove narudžbe šalje se odgovor klijentu, dok se u protivnom provjera novih narudžbi ponavlja nakon dvije sekunde. Implementacija na strani klijenta funkcionira na način da se u liniji 4 šalje zahtjev prema poslužitelju te nakon što se primi odgovor, prikazuju se narudžbe (linija 6) i funkcija se ponovo poziva, odnosno ponovo se šalje zahtjev (linija 7).

```
1   let lastId = -1;
2
3   const longPolling = () => {
4      $.get(`/orders2?lastId=${lastId}`, (response) => {
5        lastId = response.lastId;
6        prikaziNarudzbe(response);
7        longPolling();
8      }).fail((req, status, mes) => {
9        if (status === "timeout") longPolling();
10     });
11   };
12
13  longPolling();
```

Iako optimalnija od prijašnje metode, s željom da se ova tehnika još više unaprijedi možemo uvidjeti neke njene mane. Problem koji je i dalje prisutan jest ponavljajuće slanje HTTP zaglavlja sa svakim zahtjevom od strane klijenta i sa svakim odgovorom od strane poslužitelja. Iako zaglavlja ne sadrže velik teret, ako razmjenjujemo kratke poruke ona doista mogu činiti velik postotak ukupnog prometa razmjene podataka između klijenta i poslužitelja [1].

5.1.3. Server-Sent Events (SSE)

Server-Sent Events (SSE) je standardizirana HTML5 tehnologija koja omogućuje web preglednicima (klijentima) primanje podataka od poslužitelja u realnom vremenu bez potrebe za konstantnim slanjem zahtjeva za podacima prema poslužitelju. Glavna razlika u odnosu na prethodne tehnike jest ta što polling i long polling prilikom svakog zahtjeva za novim podacima stvaraju novu vezu, dok se korištenjem Server-Sent Events tehnologije za svaku razmjenu poruke koristi ista, odnosno samo jedna veza. Upravo zbog toga je ova tehnika efikasnija jer se time rješava problem stalne razmjene zaglavlja HTTP zahtjeva i odgovora koja, ukoliko se šalju kratke poruke, mogu činiti velik udio ukupnog prometa razmjene podataka.

Ova metoda funkcionira na način da klijent šalje poslužitelju HTTP zahtjev za otvaranje veze na koji poslužitelj odgovara statusnim kodom 200 nakon čega je veza uspostavljena (slika 7). Kako se za razmjenu poruka koristi samo jedna veza, ona za vrijeme slanja podataka mora biti trajno otvorena, odnosno mora se definirati da se nakon odgovora poslužitelja ona ne zatvara. To je postignuto na način da u inicijalnom odgovoru poslužitelja u HTTP zaglavlje dodaje polje „Connection: keep-alive“ čime se definira da veza ostaje trajno otvorena (eng. *HTTP persistent connection*) [21].

Slika 7: Prikaz komunikacije klijenta i poslužitelja tehnikom Server-Sent Events

Osim navedenog polja, poslužitelj također dodaje polje „Content-Type: text/event-stream“ čime klijentu daje do znanja da se radi o SSE porukama koje se razmijenjuju. Nakon što je veza uspostavljena, poruke između klijenta i poslužitelja se razmjenjuju u slijedećem formatu:

```
id: 1
event: narudzba
data: [{"id_narudzbe":6, "stol":1, "iznos":100}]
```

Ukoliko se u bilo kojem trenutku prekine veza između klijenta i poslužitelja, postoji mehanizam koji osigurava da će klijent primiti sve poruke koje je poslužitelj poslao za to vrijeme kada veza nije bila uspostavljena. To je moguće zahvaljujući opcionalnom polju „id“ koji predstavlja jedinstven identifikator poruke. Kada klijent primi poruku s poljem „id“, sprema njenu vrijednost te se prilikom ponovnog spajanja na poslužitelj (ukoliko dođe do prekida) ona šalje poslužitelju kako bi znao koju poruku je klijent zadnju primio. Na taj način klijent ne propušta poruke koje poslužitelj šalje za vrijeme kada je veza prekinuta.

Klijent se, kada uspostavi vezu s poslužiteljem, može pretplatiti na sve poruke odnosno događaje koje će poslužitelj slati ili samo na određene. Ukoliko poslužitelj u poruci navede polje „event“, tada poruku šalje samo klijentu koji je pretplaćen na taj navedeni događaj.

Polje „data“ je obavezno te sadrži same podatke koji se prenose bilo to u obliku čistog teksta, JSON-a ili nekog drugog formata.

Ono po čemu je Server-Sent Events tehnologija specifična jest to što polling metode funkcioniraju na način da klijent konstantno dohvaća (eng. *pull*) podatke s poslužitelja, dok u ovoj metodi glavnu ulogu u razmjeni podataka ima poslužitelj jer on odašilje (eng. *push*) podatke prema klijentu koji zahtjev šalje samo inicijalno prilikom otvaranja veze. Slijedi implementacija na strani poslužitelja:

```
1  res.writeHead(200, {
2    "Content-Type": "text/event-stream",
3    "Cache-Control": "no-cache",
4    "Connection": "keep-alive",
5  });
6  res.connection.setTimeout(0);
7  res.write("\n");
```

U ovom dijelu poslužitelj nakon primitka zahtjeva za otvaranje veze klijentu šalje odgovor pri čemu uključuje prije spomenuta polja zaglavlja. Nakon što je veza uspostavljena, preostali dio implementacije dosta nalikuje na implementaciji prethodne tehnike.

```
8  db.query("SELECT MAX(id_narudzbe) AS lastId FROM narudzba",
9  (err, result) => {
10    if (err) throw err;
11    lastId = result[0].lastId;
12  });
13
14  const dohvacanje = setInterval(() => {
15    db.query("SELECT * FROM narudzba WHERE id_narudzbe > ?", lastId,
16     (err, result) => {
17      if (err) throw err;
18      if (result.length > 0) {
19        res.write(`id: ${messageId}\n`);
20        res.write(`data: ${JSON.stringify(result)}\n\n`);
21        messageId += 1;
22        lastId += result.length;
23      }
24    });
25  }, 2000);
```

U linijama 8-12 provjerava se zadnja zabilježena narudžba u bazi podataka te se sprema njen identifikator. Nakon toga se u intervalima od dvije sekunde provjeravaju narudžbe koje su pristigle nakon narudžbe čiji je identifikator spremljen. Ukoliko postoje nove narudžbe, šalje se odgovor klijentu u spomenutom formatu koji uključuje polje „id“ i „data“. Implementacija na strani klijenta je poprilično jednostavna:

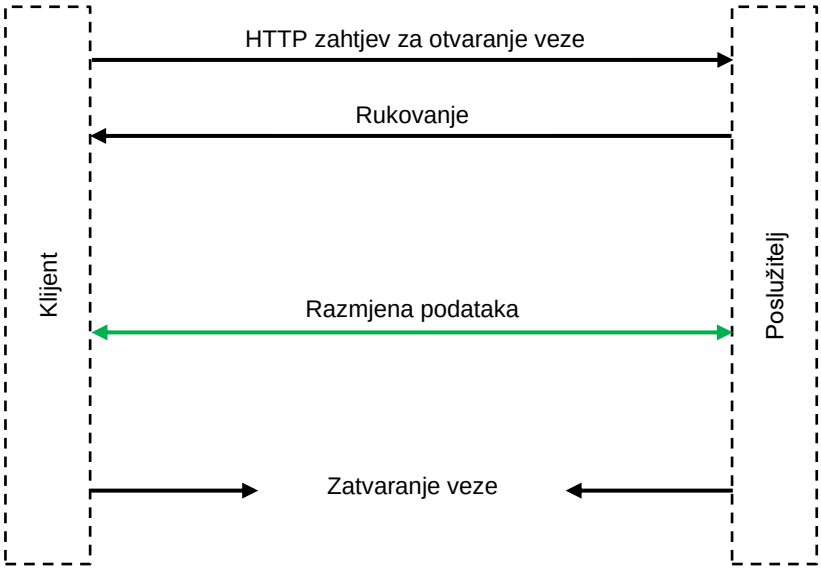
```
1  const source = new EventSource("/orders");
2  source.onmessage = (event) => {
3     ispis(event.data);
4  };
```

Potrebno je stvoriti EventSource objekt te navesti adresu poslužitelja na koji se spajamo. Kako bismo mogli čitati poruke koje poslužitelj šalje, potrebno je kreirati osluškivač onmessage događaja koji pri dolasku poruke poziva funkciju za ispis podataka o pristigloj narudžbi (linija 3).

Kao i svaka od metoda, Server-Sent Events ima svoje prednosti i nedostatke. Pošto se za razmjenu poruka koristi samo jedna veza koja za čitavo vrijeme razmjene podataka ostaje trajno otvorena, riješen je problem ponavljajućeg slanja HTTP zaglavlja pa je prema tome i latencija razmjene poruka smanjena. Nadalje, ugrađen je mehanizam ponovnog spajanja na poslužitelj ukoliko dođe do prekida veze te se pomoću polja „id“ automatski dohvaćaju poruke koje klijent nije primio zbog prekida veze. S druge strane, ograničenje ove metode jest to što je komunikacija jednosmjerna pa tako klijent ne može slati poruke ili parametre poslužitelju nakon što je veza uspostavljena, već samo prilikom slanja zahtjeva za uspostavu veze. Još jedan nedostatak je što ovu metodu ne podržavaju neki stariji web preglednici [22].

5.2. WebSocket protokol

Iako se Server-Sent Events tehnologija čini kao odličan izbor, i dalje postoje neke mane koje su za implementaciju komunikacije u realnom vremenu pomoću HTTP protokola neizbježne. Upravo zbog toga 2011. godine predstavljen je WebSocket protokol kao evolucija u komunikaciji u realnom vremenu na webu. WebSocket protokol funkcionira na princip korištenja jedne TCP veze za potpuno dvosmjernu komunikaciju između klijenta i poslužitelja. WebSocket protokol ne koristi http:// ili https:// URL shemu kao HTTP protokol, već koristi ws:// odnosno wss:// za uspostavu sigurne WebSocket veze [1].



Slika 8: Prikaz komunikacije klijenta i poslužitelja putem WebSocket

Komunikacija započinje uspostavom veze na način da klijent inicijalno šalje HTTP zahtjev prema WebSocket poslužitelju (slika 8) koji sadrži polja zaglavlja „Upgrade: websocket“ i „Connection: Upgrade“ što znači da trenutno uspostavljenu HTTP vezu želi nadograditi na WebSocket protokol. Slijedi primjer HTTP zahtjeva za uspostavu veze [23]:

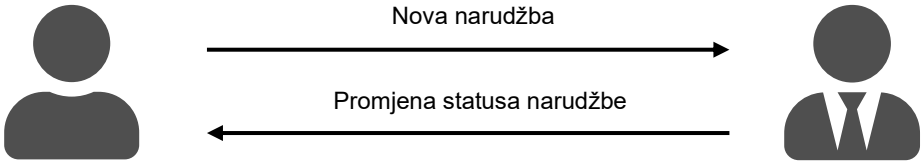
```
GET ws://localhost:8080/ HTTP/1.1
Host: 127.0.0.1:5500
Connection: Upgrade
Pragma: no-cache
Cache-Control: no-cache
Upgrade: websocket
Sec-WebSocket-Key: SY+0710EzPSP6kso/mK/aw==
Sec-WebSocket-Version: 13
```

Poslužitelj na zaprimljeni zahtjev odgovara statusnim kodom 101 koji označava promjenu protokola na onaj koji je naveden u zahtjevu u polju „Upgrade“, a to je WebSocket protokol:

```
HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: D+eU4vV2Jj/ZTkbgwOhrEKbWIb0=
```

Ovim odgovorom završeno je rukovanje te je uspostavljena veza kojom može započeti potpuno dvosmjerna razmjena podataka između klijenta i poslužitelja.

Za ovaj primjer implementacije želimo napraviti aplikaciju u kojoj postoje dvije vrste korisnika – administratori odnosno konobari restorana i korisnici. Korisnici izrađuju narudžbe koje konobari primaju nakon čega ih označavaju kao u pripremi. Nakon što konobar narudžbu označi kao u pripremi, o tome se obavještava korisnika (slika 9).



Slika 9: Prikaz komunikacije između konobara i korisnika

Primjećujemo da je ovo obostrana komunikacija – konobar u isto vrijeme može primiti narudžbu i označiti neku drugu kao u pripremi zbog čega je za ovaj scenarij pogodna implementacija pomoću WebSocket protokola. Slijedi implementacija na strani poslužitelja:

```
1  const wss = new WebSocket.Server({ port: 8080 });
2
3  let admins = [];
4  let orders = [];
5
6  wss.on("connection", (ws, req) => {
7    if (req.url === "/restaurant") admins.push(ws);
8
9    ws.on("message", (message) => {
10      const data = JSON.parse(message);
11
12      if (data.type === "newOrder") {
13        orders[data.order.order_id] = ws;
14        saveOrder(data.order);
15        admins.forEach((s) => s.send(JSON.stringify(data.order)));
16      }
```

```
17
18      if (data.type === "orderUpdate") {
19        const order = {
20          order_id: data.order_id,
21          status: "In preparation",
22        };
23        updateOrder(order);
24        orders[data.order_id].send(JSON.stringify(order));
25      }
26    });
27  });
```

Zamišljeno je da se konobari na WebSocket poslužitelj spajaju putem ws://localhost:8080/restaurant, a korisnici putem ws://localhost:8080 što će biti jednostavan način razlikovanja ove dvije vrste korisnika. U prvoj liniji stvaramo novi WebSocket Server objekt kojemu prosljeđujemo objekt koji sadrži port na kojem će se poslužitelj pokretati (8080). Prilikom spajanja klijenta na poslužitelj u liniji 7 provjeravamo je li klijent administrator (konobar) ili je običan korisnik te ukoliko je administrator spremamo njegovu vezu (odnosno *socket*) u polje „admins“.

Konobari i korisnici razmjenjivati će dva tipa poruka – „newOrder“ koju korisnik šalje pri stvaranju nove narudžbe i „orderUpdate“ koju konobar šalje korisniku pri promjeni statusa narudžbe u pripremi. Ukoliko je zaprimljena poruka o narudžbi koja izgleda ovako:

```
type: "newOrder",
order: {
   order_id: 1,
   datum_vrijeme: Thu Apr 30 2020 12:57:19 GMT+0200
   stol: 1,
   iznos: 100,
}
```

u polje „orders“ sprema se identifikator narudžbe („order_id“) zajedno s vezom (eng. *socket*) korisnika koji ju je napravio kako bismo kasnije znali kojoj vezi (odnosno kojem klijentu koji je spojen na poslužitelj) moramo poslati poruku o promjeni statusa njegove narudžbe. Nakon toga se u liniji 14 narudžba sprema u bazu podataka te se u liniji 15 svakom konobaru (koje smo spremili u polje „admins“) šalje poruka o pristigloj narudžbi. Drugi tip poruke je „orderUpdate“ koji izgleda ovako:

```
type: "orderUpdate",
order_id: 1
```

Ovu poruku prema poslužitelju šalje konobar pri čemu se u liniji 23 ažurira status narudžbe u bazi podataka te se u idućoj liniji putem pristigle vrijednosti „order_id“ polja dohvaća veza korisnika koji je tu narudžbu napravio i šalje mu se poruka o promjeni statusa njegove

narudžbe. Implementacija na strani klijenta je poprilično kratka i jednostavna. Slijedi implementacija za korisnika koji šalje narudžbu:

```
1    const socket = new WebSocket("ws://localhost:8080");
2
3    const makeOrder = () => {
4      socket.send(JSON.stringify(getOrderData()));
5    };
6
7    socket.onmessage = (event) => {
8      const data = JSON.parse(event.data);
9      updateStatus(data);
10   };
```

U prvoj liniji stvoren je novi WebSocket objekt i kao parametar proslijeđena mu je URL adresa poslužitelja na koji se odmah spaja. U liniji 3 napravljena je funkcija koja se poziva kada korisnik klikne na gumb za slanje narudžbe. Podaci o narudžbi se dohvaćaju pozivom funkcije „getOrderData“ te se oni šalju poslužitelju u obliku „newOrder“ poruke. U linijama 7-10 dodan je rukovatelj događaja koji se poziva pri primanju poruke od poslužitelja. U pitanju je „updateOrder“ poruka nakon čega se u liniji 9 poziva funkcija koja korisniku prikazuje obavijest o promijeni statusa njegove narudžbe. Slijedi implementacija administratora, odnosno konobara:

```
1    const socket = new WebSocket("ws://localhost:8080/restaurant");
2
3    const updateOrder = () => {
4      socket.send(JSON.stringify(getOrderData()));
5    };
6
7    socket.onmessage = (event) => {
8      const data = JSON.parse(event.data);
9      addOrder(data);
10   };
```

Implementacija na strani konobara razlikuje se od one za korisnika po tome što se u liniji 4 šalje „updateOrder“ poruka nakon što je kliknut gumb za promjenu statusa narudžbe, a u liniji 7-10 prima se „newOrder“ poruka te se poziva funkcija „addOrder“ koja konobaru prikazuje podatke o pristigloj narudžbi.

Može se zaključiti da je implementacija sa strane klijenta poprilično jednostavna no strana poslužitelja može biti dosta kompleksna, pogotovo za veće aplikacije koje razmjenjuju više vrsta poruka i uključuju više logike.

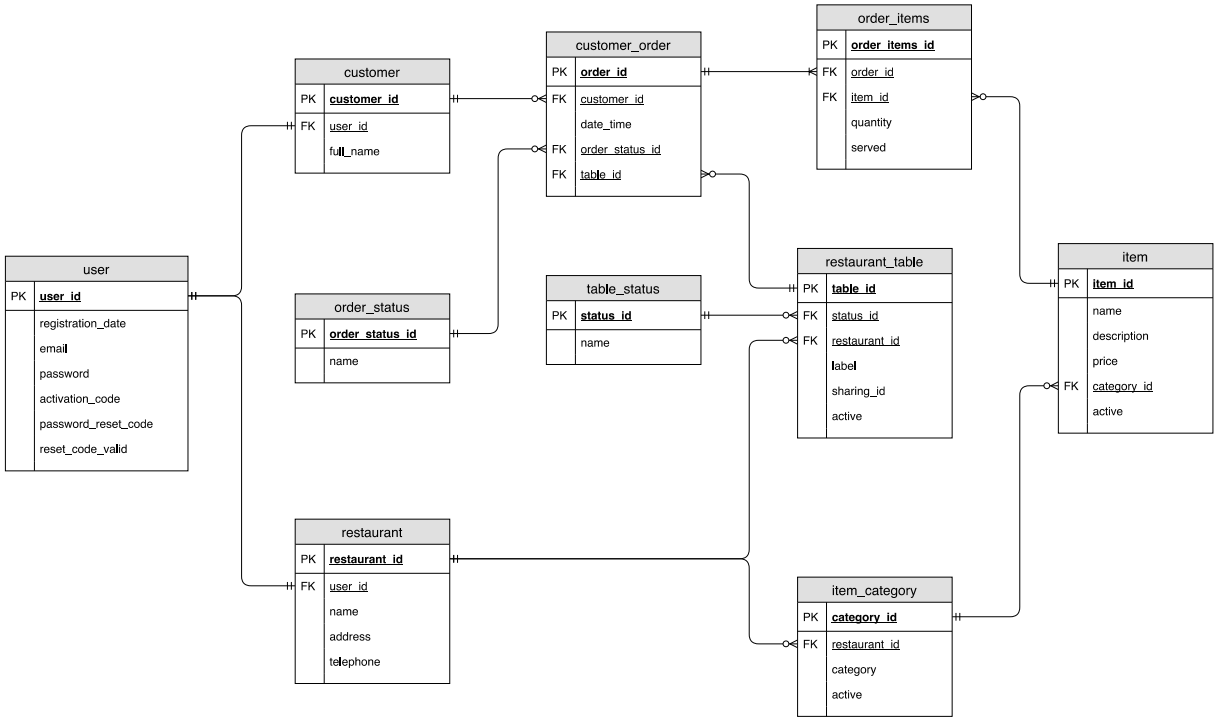
6. Implementacija aplikacije

6.1. Opis aplikacijske domene

U praktičnom dijelu ovog rada izradit ću web aplikaciju u realnom vremenu za provođenje i upravljanje narudžbama u restoranu. Postojat će tri tipa korisnika aplikacije – gost (ne registrirani korisnik), registrirani korisnik i konobar restorana. Gost će imati mogućnost izrade narudžbe kada dođe u restoran, a registrirani korisnik će osim izrade narudžbe imati mogućnost pregleda svih svojih prošlih narudžbi. Konobar restorana će imati mogućnost uređivanja jelovnika, dodavanja i uređivanja stolova restorana, pregled svih prošlih narudžbi te zaprimanje narudžbi korisnika u realnom vremenu.

Slučaj korištenja ove aplikacije biti će slijedeći: konobar se u restoranu prijavljuje u aplikaciju te mu se prikazuje popis svih stolova restorana koje je ranije dodao u aplikaciju te njihov status (slobodan, zauzet). Na istom se zaslonu u realnom vremenu dinamički prikazuju pristigle narudžbe koje stvaraju gosti ili registrirani korisnici u restoranu. Svaki stol restorana ima jedinstveni identifikator pomoću kojeg je stvorena poveznica koju kada korisnik posjeti, otvara mu se mogućnost izrade nove narudžbe za taj stol. Zbog praktičnog razloga aplikacija će za svaki stol generirati QR kod koji će sadržavati navedenu poveznicu kojeg je potom moguće ispisati na papir i staviti na stol kako bi gosti u restoranu mogli jednostavno mobitelom skenirati kod koji bi ih odmah preusmjerio na dio aplikacije za izradu nove narudžbe za taj stol. Nakon što korisnik mobitelom skenira kod ili u preglednik unese poveznicu za stol za koji je sjeo, prikazuje mu se jelovnik restorana te zatim može odabrati stavke koje želi naručiti. Nakon što napravi narudžbu, konobaru se ista odmah prikazuje na zaslonu koju, nakon što na stol odnese naručene stavke, označava kao „poslužena“ nakon čega se korisniku omogućuje dodavanje novih stavaka u narudžbu (npr. ukoliko za početak želi naručiti samo piće, a kasnije i jelo). Korisnik tako u narudžbu može dodati nove stavke te se ona opet prikazuje konobaru. Korisnik u bilo kojem trenutku putem aplikacije može zatražiti račun o čemu se obavještava konobara. Nakon dostavljanja računa na stol i naplate, konobar označava narudžbu kao plaćenu te se status stola mijenja u „slobodan“ i korisniku se prikazuje poruka o uspješno završenoj narudžbi.

6.2. ERA model



Slika 10: ERA model aplikacije

Na slici 10 prikazan je ERA model baze podataka u koju se spremaju podaci aplikacije – korisnici, restorani, narudžbe itd. U relaciji „user“ spremaju se podaci o registriranim korisnicima. Za svaku vrstu registriranog korisnika (klijent i restoran) postoje zasebne relacije u koje se spremaju podaci specifični toj vrsti korisnika – relacije „customer“ i „restaurant“.

Restoran odnosno konobar može dodavati kategorije koje se spremaju u relaciju „item_category“ kojima se pridružuju stavke menija restorana (relacija „item“). Jednom restoranu tako može pripadati više kategorija, dok je jedna kategorija vezana samo za jedan restoran. U kreirane kategorije konobar može dodavati stavke (relacija „item“) za koje se unose naziv, opis i cijena. Konobar također može dodavati stolove (relacija „restaurant_table“). Jedan restoran tako može imati više stolova, dok jedan stol pripada samo jednom restoranu. Svaki stol vezan je za relaciju „table_status“ koja određuje njegov status – „Empty“ ukoliko trenutno nema klijenata za tim stolom ili „Occupied“ ukoliko je stol zauzet, odnosno trenutno postoji aktivna narudžba za taj stol. Svaki stol ima svoj naziv (atribut „label“) – npr. T1, T2, T3, zatim

atribut „sharing_id“ u koji se sprema jedinstvena oznaka stola pomoću koje se stvara poveznica koja kada je klijent otvori, omogućuje mu se izrada narudžbe za taj stol.

Relacije „restaurant_table“, „item_category“ i „item“ posjeduju atribut „active“ koji je tipa boolean te kao početnu vrijednost ima true. Ukoliko se neki slog iz navedenih relacija želi obrisati, tada umjesto da se on obriše, njegov atribut „active“ mijenja vrijednost u false. Tako brisanjem neke stavke, ona se neće doslovno obrisati već će njen atribut „active“ biti postavljen na false. Na taj se način očuvaju podaci te stavke pošto je ona još uvijek vezana za neke račune. Na isti princip je implementirana npr. promjena cijena stavaka – „stara“ stavka se „briše“ na način da atribut „active“ poprima vrijednost false te se dodaje nova stavka koja je kopija stare samo s promijenjenom cijenom. Na taj način osigurano je da, ukoliko se promijeni cijena neke stavke, na dosadašnjim računima će i dalje biti ista cijena te stavke, odnosno ona se neće promijeniti.

U relaciji „customer_order“ spremaju se narudžbe koje su vezane za klijenta (relacija „customer“) i za stol restorana na kojem su napravljene (relacija „restaurant_table“). Svaka narudžba također je vezana za relaciju „order_status“ koja određuje njezin status. U početku kada je narudžba napravljena ona ima status „Placed“. Nakon što je narudžba poslužena, status se mijenja u „Served“, a nakon što klijent zatraži račun status prelazi u „Receipt requested“. Plaćanjem računa narudžba prelazi u finalno stanje „Paid“ te se oslobađa stol na kojem je narudžba napravljena – status stola prelazi iz „Occupied“ u „Empty“. Jedna narudžba sadrži više stavki (relacija „item“), dok se jedna stavka može nalaziti u više narudžbi. S obzirom da je ovo veza više na više, realizirana je pomoćnom relacijom „order_items“ u koju se također sprema količina stavke na računu te se bilježi je li stavka poslužena ili nije (atribut „served“). Podatak o tome je li stavka poslužena bilježi se zbog toga što nakon izrade narudžbe, korisnik kasnije može dodati nove stavke u tu narudžbu te će zahvaljujući atributu „served“ konobar vidjeti koje su to nove stavke koje je korisnik dodao, a koje su već one koje je poslužio.

6.3. Razvoj na strani poslužitelja

Poslužitelj je razvijen u Node.js radnoj okolini koristeći JavaScript te se strukturno može podijeliti na dva ključna dijela – REST API (aplikacijsko programsko sučelje) i WebSocket poslužitelj.

Aplikacijsko programsko sučelje razvijeno je pomoću Express razvojnog okvira te služi kao komunikacija između aplikacije na strani korisnika (eng. *front-end*) i aplikacije na strani poslužitelja (eng. *back-end*). Aplikacijsko programsko sučelje definira krajnje točke poslužitelja (eng. *endpoints*) na koje dio na strani korisnika šalje HTTP zahtjeve. Pri primitku zahtjeva poslužitelj se spaja na MySQL bazu podataka te vrši upite – npr. dohvaća narudžbe i vraća ih klijentu u JSON formatu.

WebSocket poslužitelj služi za ostvarenje potpuno dvosmjerne komunikacije u realnom vremenu između dva tipa korisnika - klijenata i restorana (konobara). WebSocket poslužitelj osluškuje nekoliko vrsta događaja odnosno poruka, obrađuje ih te potom prosljeđuje. Sva logika aplikacije koja nije vidljiva krajnjem korisniku na taj je način implementirana na strani poslužitelja.

6.3.1. Aplikacijsko programsko sučelje (API)

Glavna svrha poslužiteljskog dijela aplikacije jest implementacija poslovne logike aplikacije, kao npr. vršenje upita na bazu podataka, koja zbog sigurnosnih razloga ne bi smjela biti izložena na strani korisnika. Način i pravila interakcije poslužiteljskog i korisničkog dijela aplikacije određeni su aplikacijskim programskim sučeljem (API) koje sadrži funkcije koje korisnički dio poziva slanjem HTTP zahtjeva na krajnje točke (eng. *endpoints*) API-ja.

Autentifikacija i autorizacija korisnika realizirana je pomoću JSON Web Token (JWT) standarda koji predstavlja kompaktan način sigurnog potvrđivanja korisnikovog identiteta u HTTP komunikaciji [24]. JSON Web Token je zapravo šifriran niz znakova koji sadrži digitalno potpisane podatke o korisniku. Za implementaciju ovog načina autentifikacije korištena je „jsonwebtoken“ biblioteka dostupna putem npm upravitelja paketa. Slijedi tablica u kojoj su prikazane krajnje točke čija funkcija je autentifikacija korisnika.

Tablica 1: Prikaz krajnjih točaka čija funkcija je autentifikacija korisnika

Metoda	Krajnja točka	Tijelo zahtjeva	Opis
POST	/login	`{ email: 'user@mail.com', password: '123456' }`	Prijava korisnika, tj. generiranje JWT-a
POST	/verify	-	Provjera validnosti JWT-a
POST	/logout	-	Odjava, tj. brisanje JWT-a iz zahtjeva

Prilikom uspješne prijave korisnika putem krajnje točke „/login“ generira se JWT koji sadrži podatke korisnika – jedinstveni identifikator (id), tip korisnika (klijent ili restoran) te naziv klijenta odnosno restorana. JWT se šifrira pomoću zadane šifre te se sadržaj (eng. *payload*) koji se prenosi digitalno potpisuje. Slijedi primjer šifriranog JWT-a:

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpZCI6NCwidHlwZSI6InJlc3
RhdXJhbnQiLCJuYW1lIjoiUmVzdGF1cmFudCBUZXN0IiwiaWF0IjoxNTkxNjIwMTUwfQ
.cGDiK2hQ0Ozcm9zOKdrc-Hp3F73EQP3gwqthppx0KM8
```

Ovaj JWT sadrži slijedeće podatke o korisniku u JSON formatu:

```
{
  "id": 4,
  "type": "restaurant",
  "name": "Restaurant Test"
}
```

JWT se nakon uspješne prijave u šifriranom obliku sprema u HttpOnly kolačić (eng. *HttpOnly cookie*) te se prilaže uz svaki HTTP zahtjev koji korisnik šalje prema API-ju. Pri dolasku zahtjeva na npr. krajnju točku za pregled svih prijašnjih narudžbi, poslužitelj provjerava je li u zahtjevu prisutan kolačić s JWT-om te ukoliko je, provjerava njegovu ispravnost odnosno digitalni potpis. Slijedi dio koda krajnje točke „/login“ koji služi za generiranje JWT-a:

```
1   const token = jwt.sign({ id, type, name }, process.env.JWT_SECRET);
2
3   res.cookie("Authorization", `Bearer ${token}`, {
4     httpOnly: true,
5   });
6
7   res.status(200).json({
8     success: true,
9     user: { id, type, name },
10  });
```

U prvoj se liniji potpisuje odnosno generira JWT u koji se pohranjuju podaci o korisniku koji su dohvaćeni iz baze podataka – jedinstven identifikator, tip korisnika te ime korisnika. JWT se šifrira s šifrom koja je funkciji sign proslijeđena u drugom argumentu. Potom se generirani JWT sprema u HttpOnly kolačić (linije 3-5) te se u liniji 7-10 korisniku vraća odgovor o uspješnoj prijavi, koji uključuje njegove podatke.

Krajnja točka „/verify“ služi za validaciju priloženog JWT-a. Ova krajnja točka se na korisničkoj strani poziva pri pristupanju rutama/putanjama kojima mogu pristupiti samo prijavljeni korisnici. Slijedi kod navedene krajnje točke:

```
1   const verify = (req, res) => {
2     const token = req.cookies.Authorization.split(" ")[1];
3
4     if (!token)
5       return res.status(401).json({
6         success: false,
7         message: "No token"
8       });
9
10     jwt.verify(token, process.env.JWT_SECRET, (err, user) => {
11      if (err)
12        return res.status(401).json({
13          success: false,
14          message: "Invalid token"
15        });
16      return res.status(200).json({ success: true, data: user });
17    });
18  };
```

U liniji 2 se dohvaća kolačić iz zahtjeva s nazivom „Authorization“ u kojemu se očekuje JWT. Zatim se u linijama 4-8 provjerava je li JWT priložen te ako nije vraća se status 401 i poruka o neuspjehu. Ukoliko je JWT priložen, u liniji 10 se provjerava njegova validnost. Ukoliko JWT nije ispravan, poslužitelj vraća statusni kod 401 i poruku o neuspjehu, dok u suprotnom vraća statusni kod 200 te podatke o korisniku.

Krajnja točka „/logout“ vrši odjavu korisnika na način da briše „Authorization“ kolačić iz zahtjeva (linija 2) te nakon toga vraća poruku o uspješnoj odjavi:

```
1  const logout = (req, res) => {
2    res.clearCookie("Authorization");
3    res.status(200).json({ success: true });
4. };
```

U tablici 2 prikazane su krajnje točke za upravljanje podacima o stolovima restorana.

Tablica 2: Krajnje točke za upravljanje podacima o stolovima restorana

Metoda	Krajnja točka	Tijelo zahtjeva	Opis
GET	/:restaurantId/tables	-	Dohvaćanje svih stolova određenog restorana
POST	/:restaurantId/tables	`{ label: 'T1' }`	Dodavanje novog stola
DELETE	/:restaurantId/tables/:tableId	-	Brisanje određenog stola
GET	/:restaurantId/tables/:tableId	-	Dohvaćanje podataka o određenom stolu
PUT	/:restaurantId/tables/:tableId	`{ label: 'T2' }`	Ažuriranje podataka o određenom stolu

Kada korisnik prijavljen kao konobar u aplikaciji doda novi stol, korisnička strana aplikacije šalje zahtjev prema krajnjoj točki „/:restaurantId/tables“ na poslužiteljskoj strani s priloženim JWT-om prijavljenog korisnika. Ukoliko je JWT validan i korisnik ima dopuštenje za dodavanje stola, funkcija koja se izvršava pri primitku zahtjeva na toj krajnjoj točki izvršava upit u bazu podataka za dodavanje novog zapisa. Na ovaj način funkcioniraju sve radnje s zapisima u bazi podataka – dodavanje novih zapisa te brisanje i izmjena postojećih. Slijedi kod kojim su definirane krajnje točke iz tablice 2:

```
1   const express = require("express");
2   const auth = require("../middleware/auth");
3   const router = express.Router();
4   const restaurant = require("../controller/restaurantController");
5
6   router.get("/:id/tables", auth, restaurant.getTables);
7   router.post("/:id/tables", auth, restaurant.addTables);
8   router.delete("/:id/tables/:tableId", auth, restaurant.deleteTable);
9   router.get("/:id/tables/:tableId", auth, restaurant.getTable);
10  router.put("/:id/tables/:tableId", auth, restaurant.editTable);
```

Svaka krajnja točka definirana je funkcijom router.[metoda]. Tako je npr. za krajnju točku za dohvaćanje svih stolova restorana u liniji 6 definirano da će se pri primitku GET zahtjeva na putanju „/:id/tables“, gdje je „:id“ parametar koji određuje jedinstven identifikator restorana, pozvati funkcija „auth“, a nakon toga funkcija „getTables“. Funkcija „auth“ služi za validaciju JWT-a te se naziva tzv. *middleware* funkcija zato što u njoj ne završava HTTP ciklus zahtjev-

odgovor, već ona obavlja zadatak (validaciju JWT-a) koji prethodi glavnoj funkciji te krajnje točke – getTables. Slijedi kod funkcije getTables:

```
1   const getTables = (req, res) => {
2     const restaurantId = req.params.id;
3       conn.query(
4         "SELECT table_id, label, name AS status, sharing_id
5          FROM restaurant_table t
6          JOIN table_status s ON t.status_id = s.status_id
7          WHERE restaurant_id = ? AND active = 1 ORDER BY table_id",
8         [restaurantId],
9         (err, results) => {
10          res.status(200).json({ success: true, data: results });
11        }
12      );
13  };
```

Ova funkcija u liniji 2 u varijablu sprema parametar „id“ koji predstavlja jedinstven identifikator restorana. Zatim u idućim linijama izvršava SQL upit u bazi te rezultate vraća klijentu kao odgovor (linija 10).

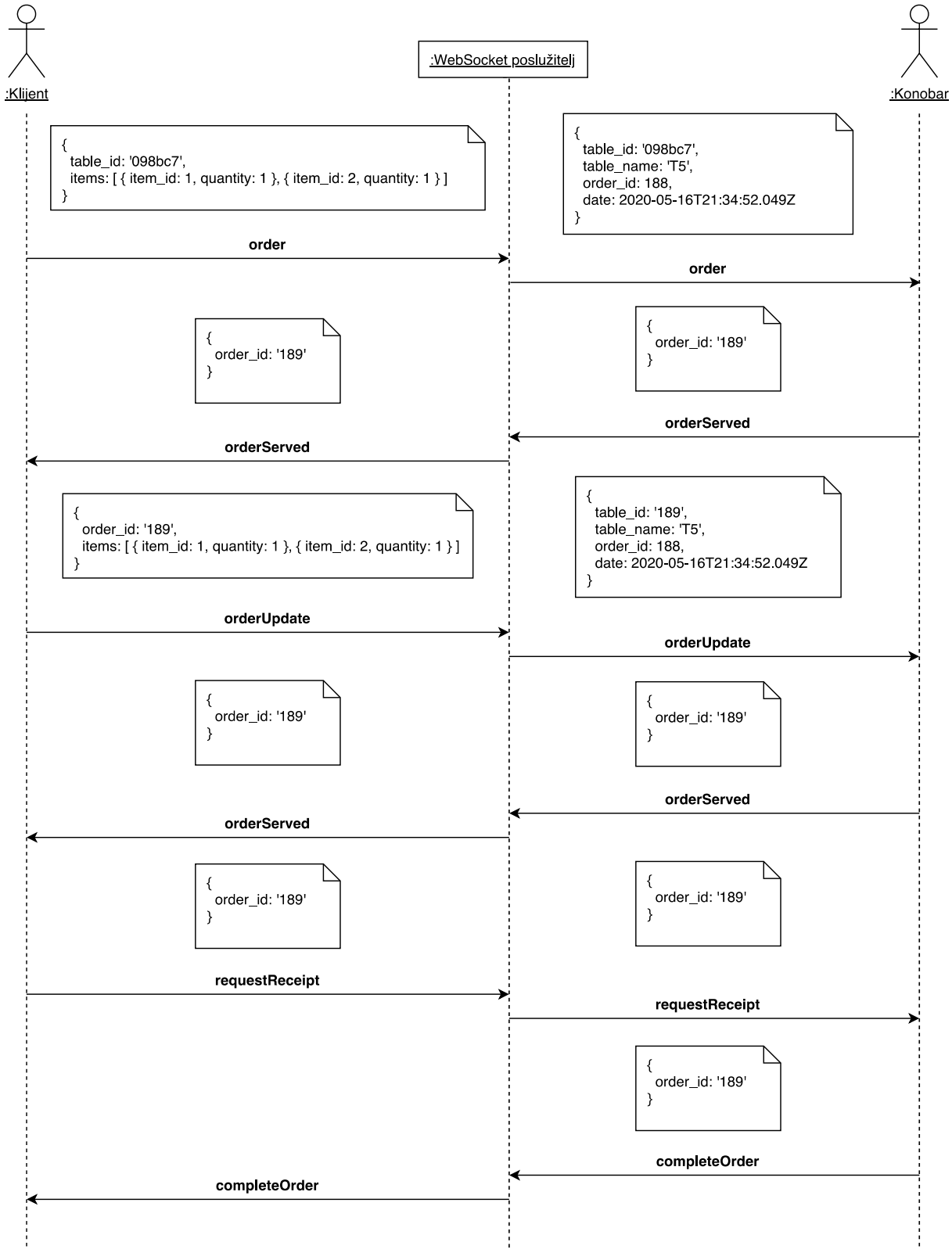
6.3.2. WebSocket poslužitelj

Komunikacija u realnom vremenu će biti implementirana koristeći WebSocket protokol iz nekoliko razloga. Prvi je taj što od spomenutih načina implementacija u teorijskom dijelu ovog rada, WebSocket protokol jedini omogućuje potpuno dvosmjernu komunikaciju. Dvosmjerna komunikacija je u ovom primjeru važna iz razloga što postoji scenarij u kojemu će konobar u isto vrijeme zaprimiti novu narudžbu (primit će poruku od poslužitelja) i označit će neku narudžbu kao plaćenu (poslat će poruku poslužitelju). Idući razlog jest efikasnost. Naime, s obzirom da WebSocket veza ostaje trajno otvorena za čitavo vrijeme prijenosa podataka, slanjem svake poruke se ne razmjenjuju HTTP zaglavlja što čini ovu tehniku efikasnijom od načina implementacije koji koriste HTTP protokol.

Na slici 11 prikazan je dijagram slijeda prema UML-u koji prikazuje slijed razmjene poruka između klijenta i konobara te njihov sadržaj. Poruke se ne razmjenjuju direktno između ova dva učesnika, već posredstvom WebSocket poslužitelja na koji učesnici šalju poruke. Nakon primitka poruke, poslužitelj ju obrađuje te na temelju primljenih podataka izvršava npr. unos narudžbe u bazu podataka, ažuriranje narudžbe u bazi podataka itd.

Iz dijagrama možemo vidjeti koje tipove poruka mora osluškivati svaki učesnik. Klijent tako mora osluškivati poruke *orderServed* i *completeOrder*, dok konobar osluškuje poruke *order*, *orderUpdate* i *requestReceipt*. WebSocket poslužitelj, koji će biti razvijen pomoću Socket.io biblioteke, osluškuje sve tipove poruka.

Slijedi opis slijeda razmjene poruka između klijenta i konobara. Klijent nakon što otvori aplikaciju ima mogućnost odabira stavaka iz menija restorana koje želi naručiti. Nakon odabira stavaka i njihove količine, pritiskom na gumb za slanje narudžbe se šalje poruka tipa *order* na poslužitelj koja sadrži jedinstven identifikator stola na kojem je napravljena narudžba te popis svih stavaka koje je korisnik dodao u narudžbu. Pri primitku poruke tipa *order*, poslužitelj u bazu podataka unosi novu narudžbu te konobaru šalje jedinstven identifikator stola, naziv stola te jedinstven identifikator i datum upravo kreirane narudžbe. Nakon toga, konobar može narudžbu označiti kao posluženu pri čemu se poslužitelju šalje poruka tipa *orderServed* koja sadrži jedinstven identifikator narudžbe. Pri primitku poruke poslužitelj na temelju primljenog identifikatora narudžbe istoj u bazi podataka mijenja status u poslužena. Nakon toga poslužitelj porukom *orderServed* obavještava klijenta o promjeni statusa njegove narudžbe nakon čega mu se dozvoljava dodavanje novih stavaka u narudžbu. Korisnik sada može ili zatražiti račun i završiti narudžbu ili može dodati nove stavke u narudžbu.



Slika 11: Dijagram slijeda razmjene poruka putem WebSocket protokola

Ukoliko se odluči za dodavanje novih stavaka, odabire stavke i pritiskom na gumb za narudžbu poslužitelju se porukom *orderUpdate* šalje identifikator narudžbe u koju korisnik želi dodati stavke (order_id) te stavke koje je korisnik odabrao. Pri primitku poruke, poslužitelj u bazu podataka unosi nove stavke te konobaru šalje poruku tipa *orderUpdate* koja je sadržajem jednaka poruci *order*. Nakon toga konobar opet nakon što posluži klijenta označava narudžbu kao posluženu (*orderServed* poruka) o čemu poslužitelj obavještava klijenta. Na kraju klijent može klikom na gumb od konobara zatražiti račun pri čemu se poslužitelju šalje poruka tipa *requestReceipt* koja sadrži identifikator narudžbe. Poslužitelj potom ažurira status narudžbe u bazi podataka te konobara porukom *requestReceipt* obavještava o zatraženom računu. Nakon što donese klijentu račun kojeg on potom plaća, konobar narudžbu označava kao završenu pri čemu se poslužitelju šalje poruka tipa *completeOrder* koja sadrži identifikator narudžbe. Kada poslužitelj primi poruku tog tipa, označava narudžbu u bazi podataka kao završenu, mijenja status stola u „Empty“ te klijenta porukom *completeOrder* obavještava o završenoj narudžbi.

Nakon razrade načina i pravila komunikacije između klijenta i konobara, možemo krenuti na pregled same implementacije. Iz dijagrama (slika 11) se vidi da poslužitelj mora osluškivati slijedeće događaje odnosno tipove poruka: *order, orderServed, orderUpdate, requestReceipt, orderComplete*. Implementacija je strukturirana tako da su u jednoj datoteci tj. modulu postavljeni svi potrebni osluškivači događaja. Slijedi dio koda tog modula:

```
1  // Kada se korisnik spoji na poslužitelj
2  io.on("connect", (socket) => {
3
4    // Provjeri tip korisnika koji se spojio
5    onConnect(socket);
6
7    // Klijent je napravio novu narudžbu
8    socket.on("order", (order) => onOrder(socket, order));
9
10   // Konobar je označio narudžbu kao posluženu
11   socket.on("orderServed", (order) => onOrderServed(order));
12
13   // Klijent je dodao nove stavke u narudžbu
14   socket.on("orderUpdate", (order) => onOrderUpdate(order));
15
16   // Klijent je zatražio račun
17   socket.on("requestReceipt", (order) => onRequestReceipt(order));
18
19   // Konobar je označio narudžbu kao plaćenu
20   socket.on("orderComplete", (order) => onOrderComplete(socket, order));
21
22   // Korisnik se odspojio
23   socket.on("disconnect", () => onDisconnect(socket));
24
25 });
```

Ranije je spomenuto da Node.js radna okolina ima arhitekturu temeljenu na događajima (eng. *event-driven architecture*). Upravo to je vidljivo iz gore priloženog koda. Naime, za registraciju osluškivača događaja, odnosno poruka, koristi se funkcija socket.on koja kao prvi argument prima naziv događaja (poruke) koji osluškujemo, a kao drugi argument prima tzv. *callback* funkciju koja će se izvršiti pri pojavi navedenog događaja. Tako su u linijama 7-23 registrirani osluškivači za svaki događaj odnosno tip poruka koji će se razmjenjivati između klijenta i konobara (slika 11).

Prije objašnjenja logike obrade primljenih poruka, bitno je za napomenuti da su definirana dva globalna polja – connected_restaurants i customer_orders. U prvo polje spremaju se konekcije (eng. *sockets*) konobara koji su trenutno spojeni na poslužitelj kako bi se pri primitku nove narudžbe znalo kojoj konekciji je potrebno poslati poruku *order*. Konekcije se u ovo polje spremaju pod indeksom koji je jednak identifikatoru restorana (id) prema kojemu se pri dolasku narudžbe pronalazi konekcija konobara. U polje customer_orders spremaju se konekcije klijenata koji su napravili narudžbe kako bi se na temelju identifikatora narudžbe (order_id), što je ujedno indeks spremljenih konekcija u polju, znalo kojoj konekciji je potrebno poslati poruke *orderServed* i *completeOrder*. U liniji 5 se pri spajanju korisnika na poslužitelj poziva funkcija onConnect koja služi za autentifikaciju. Slijedi kod i objašnjenje te funkcije:

```
1  const onConnect = (socket) => {
2
3    // Ukoliko JWT nije priložen, spojio se gost
4    if (!socket.handshake.headers.cookie ||
5        !cookie.parse(socket.handshake.headers.cookie).Authorization)
6        return;
7
8    // Ukoliko se spojio registrirani korisnik, dohvati kolačić s JWT-om
9    const authorization_cookie =
10         cookie.parse(socket.handshake.headers.cookie).Authorization;
11
12   // Dohvati JWT iz kolačića pošto je on u formatu 'Bearer <JWT>'
13   const token = authorization_cookie.split(" ")[1];
14
15   // Validacija JWT-a
16   jwt.verify(token, process.env.JWT_SECRET, (err, user) => {
17
18     // Ukoliko JWT nije ispravan, odspoji klijenta
19     if (err) socket.disconnect();
20
21     // Ukoliko se spojio restoran/konobar, spremi njegovu konekciju
22     if (user.type === "restaurant") {
23
24       // Spremi konekciju konobara u polje
25       connected_restaurants[user.id] = socket;
26     }
27   });
28
29 };
```

Nakon što je korisnik prošao korak autentifikacije, može započeti razmjena poruka između njega i poslužitelja. Slijedi kod funkcije onOrder koja se poziva prilikom dolaska poruke tipa *order* od strane klijenta. U ovoj funkciji se narudžba unosi u bazu podataka te se iz polja connected_restaurants dohvaća konekcija konobara restorana za koji je narudžba napravljena kako bi se njemu poslala poruka tipa *order* (slika 11). Funkcija prima parametre socket – konekcija klijenta koji je napravio narudžbu te order – primljeni podaci o narudžbi.

```
1  const onOrder = (socket, order) => {
2
3     // Dohvati podatke o tom stolu (npr. kojem restoranu pripada)
4     conn.query("SELECT * FROM restaurant_table WHERE sharing_id = ?",
5     order.table_id, (err, results) => {
6       if (err) throw err;
7         const { restaurant_id, table_id, label } = results[0];
8
9         // Dohvati konekciju konobara tog restorana
10        const restaurantSocket = connected_restaurants[restaurant_id];
11
12        // Unesi narudžbu u bazu podataka
13        conn.query("INSERT INTO customer_order
14        VALUES (DEFAULT, ?, NOW(), 1, ?)",
15        [order.customer_id, table_id], (err, result) => {
16          if (err) throw err;
17
18          // Pošalji konobaru poruku 'order'
19          const orderMessage = { ...order, table_name: label,
20                          order_id: result.insertId, date: new Date() };
21
22          if (restaurantSocket)
23              restaurantSocket.emit("order", orderMessage);
24
25          // Spremi konekciju klijenta pod indeksom broja narudžbe
26          customer_orders[result.insertId] = socket;
27
28          // Dodaj stavke narudžbe u bazu podataka
29          order.items.forEach((item) => {
30            conn.query("INSERT INTO order_items
31            VALUES (DEFAULT, ?, ?, ?, DEFAULT)",
32            [result.insertId, item.item_id, item.quantity]);
33          });
34
35        });
36
37        // Promijeni status stola u 'Occupied'
38        conn.query("UPDATE restaurant_table SET status_id=2
39                    WHERE table_id=?", table_id);
40     });
41
42  };
```

Nakon što konobar zaprimi poruku tipa order te posluži klijenta stavkama iz narudžbe, narudžbu označava kao posluženu pri čemu poslužitelj prima *orderServed* poruku koju zatim prosljeđuje klijentu (slika 11). Slijedi kod funkcije onOrderServed koja se izvršava pri

zaprimanju poruke *orderServed* od strane konobara. Funkcija kao parametar prima podatke o narudžbi od kojih uzimamo samo order_id.

```
const onOrderServed = ({ order_id }) => {

  // Ažuriraj status narudžbe u bazi podataka
  conn.query("UPDATE customer_order SET order_status_id = 2
              WHERE order_id 6 = ?", order_id);

  // Označi stavke narudžbe kao poslužene
  conn.query("UPDATE order_items SET served = 1
              WHERE order_id = ?", order_id);

  // Pošalji klijentu poruku 'orderServed'
  customer_orders[order_id].emit("orderServed", { order_id });

};
```

Nakon što je narudžba poslužena, korisnik ju može nadopuniti dodavanjem novih stavaka pri čemu se poslužitelju šalje poruka *orderUpdate* (slika 11) koju obrađuje funkcija onOrderUpdate koja će biti prikazana u nastavku. Funkcija kao parametar prima podatke o narudžbi od kojih izdvaja podatke table_id, order_id i items.

```
const onOrderUpdate = ({ table_id, order_id, items }) => {

   // Provjeri kojem restoranu pripada stol ove narudžbe
   conn.query("SELECT * FROM restaurant_table WHERE sharing_id = ?",
   table_id, (err, results) => {
     if (err) throw err;

     const { restaurant_id, label } = results[0];

     // Dohvati spojenog konobara tog restorana
     const restaurantSocket = connected_restaurants[restaurant_id];

     // Ažuriraj status narudžbe
     conn.query("UPDATE customer_order SET order_status_id = 1
     WHERE order_id = ?", order_id, (err, results) => {
       if (err) throw err;

       // Dodaj nove stavke u narudžbu
       items.forEach((item) => {
         conn.query("INSERT INTO order_items
                     VALUES (DEFAULT, ?, ?, ?, DEFAULT)",
                     [order_id, item.item_id, item.quantity]);
       });

       // Pošalji konobaru poruku 'orderUpdate'
       const orderMessage = { items, table_name: label,
                              order_id, table_id, date: new Date() };
       if (restaurantSocket)
         restaurantSocket.emit("orderUpdate", orderMessage);
     });
   });
};
```

Kada klijent želi zatražiti račun od konobara, klikom na gumb šalje se poslužitelju poruka *requestReceipt* čija funkcija za obradu slijedu u nastavku. Funkcija kao parametar prima podatke o narudžbi od kojih uzima samo identifikator narudžbe (order_id).

```
1  const onRequestReceipt = ({ order_id }) => {
2
3    // Provjeri kojem restoranu pripada stol ove narudžbe
4    conn.query("SELECT restaurant_id FROM customer_order o
5                JOIN restaurant_table t ON o.table_id = t.table_id
6                WHERE order_id = ?", order_id,
7      (err, results) => {
8        if (err) throw err;
9
10       const { restaurant_id } = results[0];
11
12       // Dohvati konekciju konobara tog restorana
13       const restaurantSocket = connected_restaurants[restaurant_id];
14
15       // Ažuriraj status narudžbe
16       conn.query("UPDATE customer_order SET order_status_id = 3
17       WHERE order_id = ?", order_id, (err, results) => {
18         if (err) throw err;
19
20         // Pošalji konobaru poruku 'requestReceipt'
21         if (restaurantSocket)
22           restaurantSocket.emit("requestReceipt", { order_id });
23       });
24     }
25   );
26
27 };
```

Nakon zaprimanja poruke o zatraženom računu, konobar klijentu odnosi račun kojeg on plaća te potom konobar označava narudžbu kao završenu tj. plaćenu. Klikom na gumb za označavanje narudžbe kao plaćene, poslužitelju se šalje poruka tipa *completeOrder* koja se prosljeđuje klijentu koji je napravio narudžbu (slika 11). U nastavku slijedi kod i objašnjenje funkcije onOrderComplete koja obrađuje poruku *completeOrder* na strani poslužitelja. Funkcija kao parametar prima identifikator narudžbe (order_id).

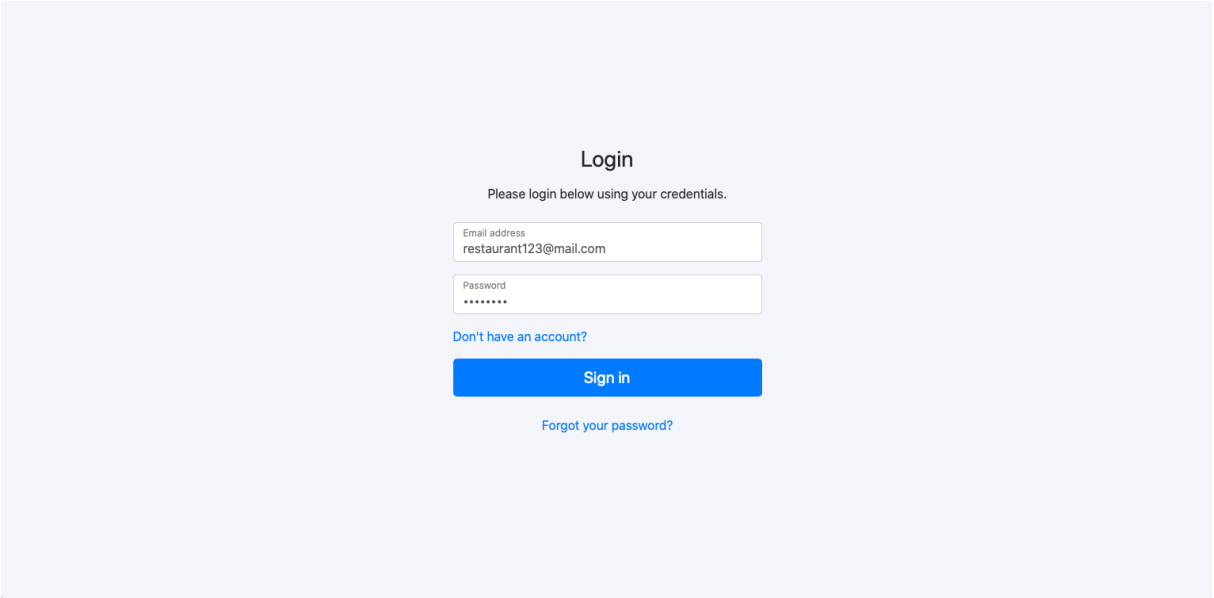
```
1  const onOrderComplete = ({ order_id }) => {
2
3    // Ažuriraj status narudžbe u 'Paid'
4    conn.query("UPDATE customer_order SET order_status_id = 4
5                WHERE order_id = ?", order_id, (err) => {
6      if (err) throw err;
7
8      // Dohvati podatke o stolu narudžbe
9      conn.query("SELECT t.restaurant_id, t.table_id FROM customer_order o
10                 JOIN restaurant_table t ON o.table_id = t.table_id
11                 WHERE order_id = ?", order_id, (err, results) => {
12         if (err) throw err;
13
14         const { table_id } = results[0];
15
16         // Ažuriraj status stola u 'Empty'
17         conn.query("UPDATE restaurant_table SET status_id = 1
18                     WHERE table_id = ?", table_id, (err) => {
19          if (err) throw err;
20
21          // Pošalji klijentu poruku 'orderComplete'
22          customer_orders[order_id].emit("orderComplete");
23        });
24      }
25    );
26  });
27
28 };
```

6.4. Razvoj na strani korisnika

Aplikacija je na strani korisnika razvijena koristeći React biblioteku za izgradnju dinamičkog i interaktivnog korisničkog sučelja. Od ostalih značajnijih biblioteka koje su korištene pri razvoju korisničkog sučelja bitno je istaknuti Axios biblioteku koja je korištena za slanje AJAX zahtjeva prema poslužiteljskoj strani (API-ju). Za upravljanje stanjem React komponenata (eng. *state management*) korištena je Redux biblioteka. Ikone koje se koriste su iz Font Awesome skupa ikona. U nastavku će biti prikazane funkcionalnosti i korisničko sučelje za svakog tipa korisnika.

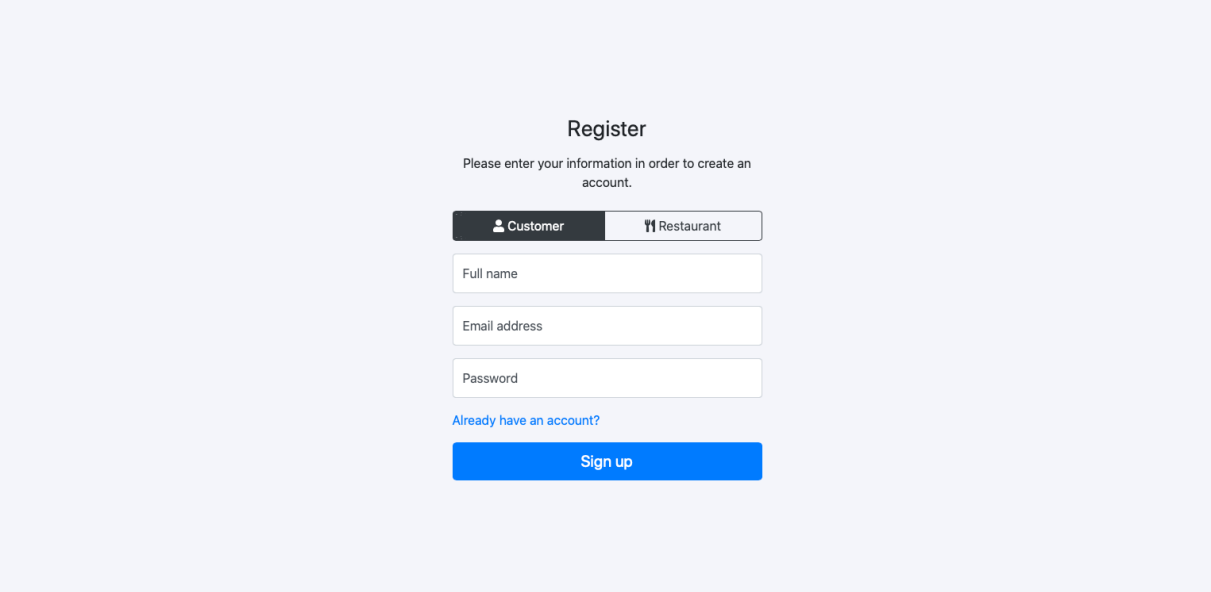
6.4.1. Neregistrirani korisnik

Neregistrirani korisnik ima mogućnosti registracije, prijave te izrade nove narudžbe. Na slici 12 prikazana je forma za prijavu pomoću koje se korisnik unosom email adrese i lozinke prijavljuje u aplikaciju.



Slika 12: Zaslon prijave u aplikaciju

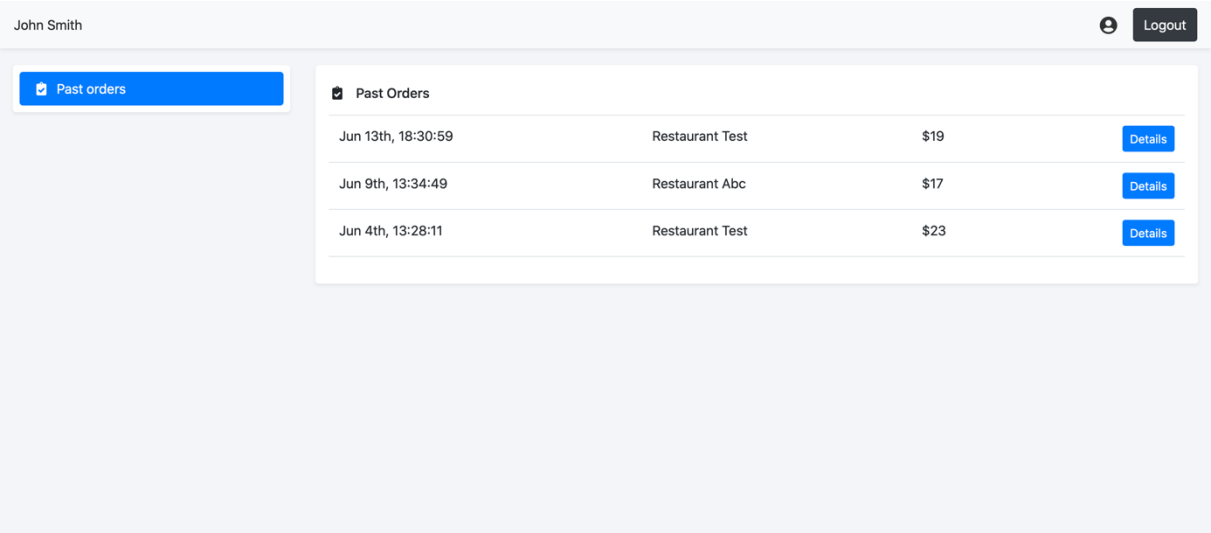
Na slici 13 prikazana je forma za registraciju korisnika. Pri registraciji korisnik odabire želi li se registrirati kao restoran, te će tako moći napraviti meni za svoj restoran i primati narudžbe, ili kao klijent koji će moći stvarati narudžbe (isto kao neregistrirani korisnik) te vidjeti svoje prošle narudžbe.



Slika 13: Forma za registraciju

6.4.2. Registrirani korisnik

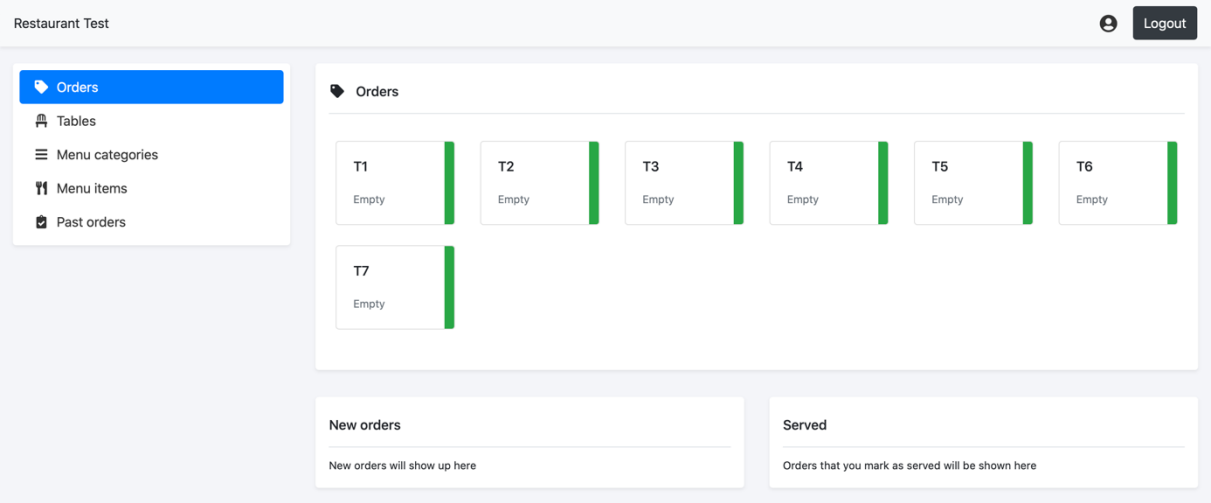
Registrirani korisnik ima mogućnost izrade narudžbe u nekom od restorana isto kao neregistrirani korisnik, no dodatna mogućnost koju ima kada se prijavi u aplikaciju jest pregled svih dosadašnjih narudžbi i stavaka tih narudžbi (slika 14).



Slika 14: Pregled svih dosadašnjih narudžbi (registrirani korisnik)

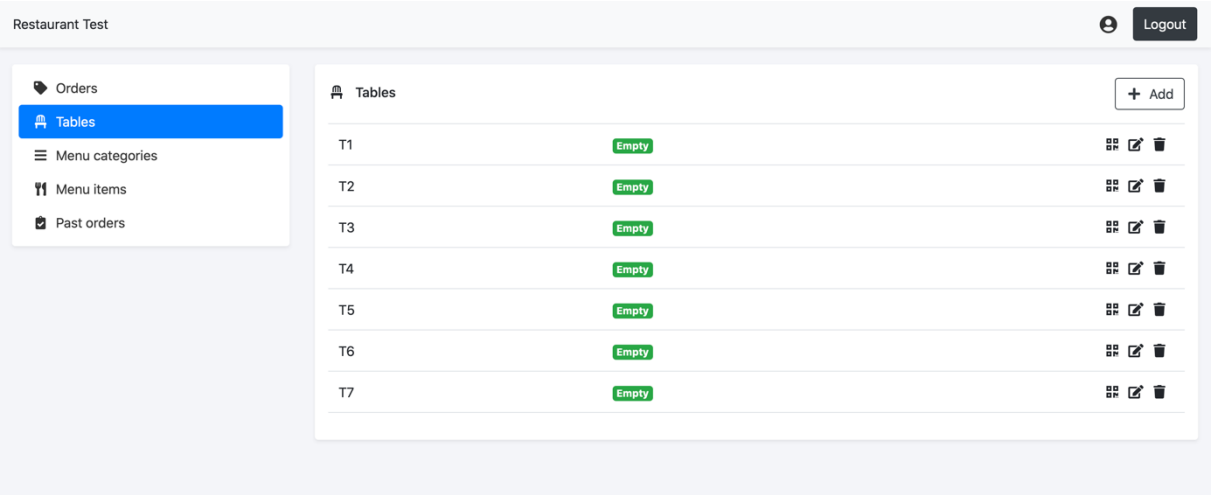
6.4.3. Konobar

Korisnik se u aplikaciju može registrirati kao konobar (slika 13) pri čemu dobiva mogućnosti dodavanja stolova restorana, kreiranja menija te kategorija menija, dodavanja stavaka na meni, pregleda prošlih narudžbi te upravljanja narudžbama u realnom vremenu kako one nastaju, što će biti objašnjeno u idućem poglavlju.

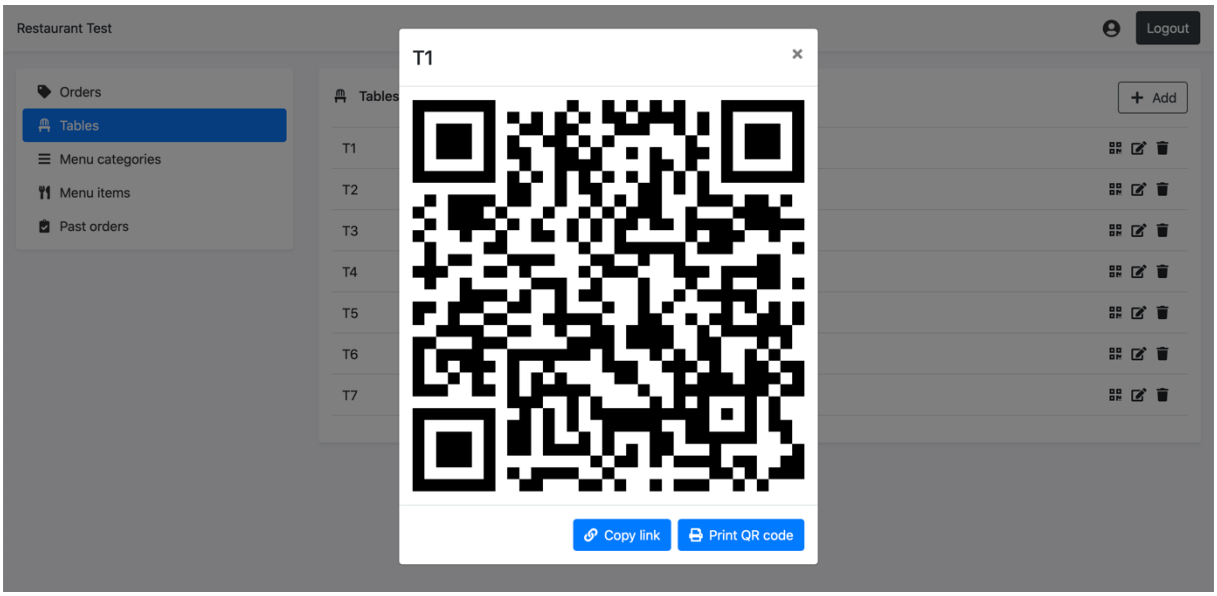


Slika 15: Prikaz narudžbi i statusa stolova u realnom vremenu

Na slici 15 prikazan je zaslon za upravljanje narudžbama i stanjem stolova u realnom vremenu, o čemu će više riječi biti kasnije, dok je na slici 16 prikazan zaslon za pregled svih stolova restorana s mogućnošću dodavanja novih te uređivanja postojećih stolova.

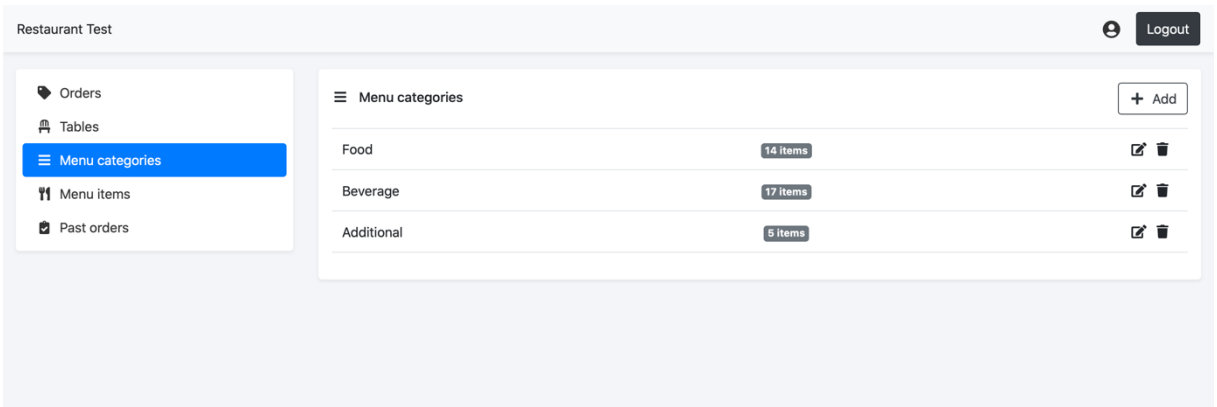


Slika 16: Prikaz svih stolova restorana s mogućnošću dodavanja novih te uređivanja postojećih



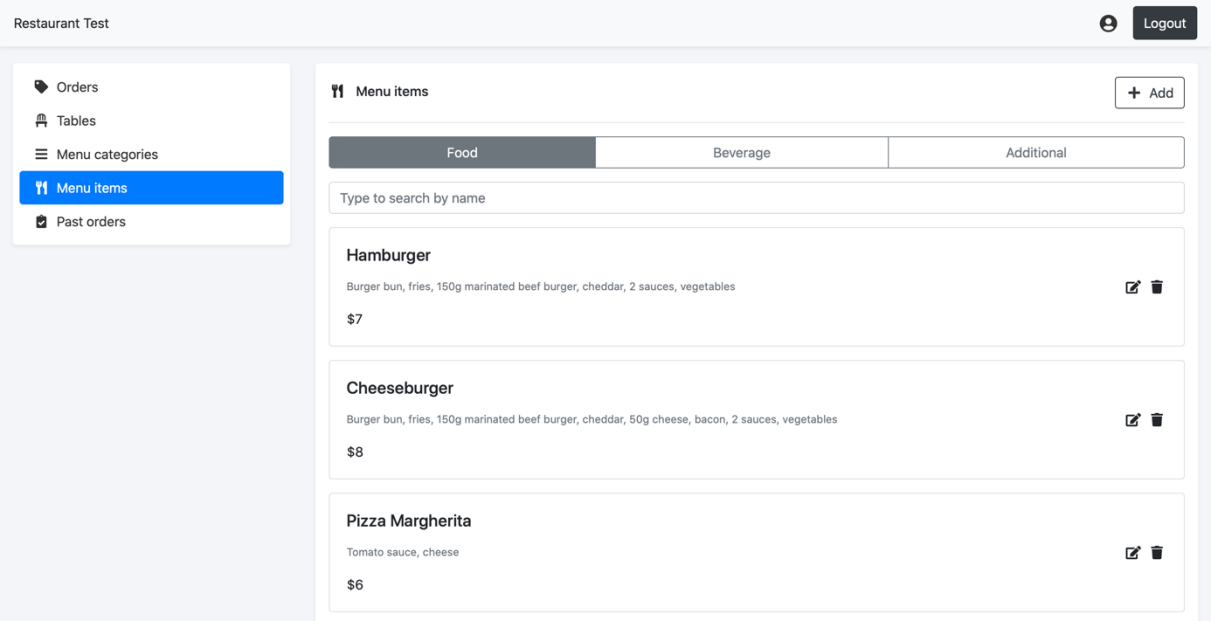
Slika 17: Prikaz modala za dijeljenje poveznice i ispis QR koda odabranog stola

Svaki stol restorana ima svoj jedinstveni identifikator pomoću kojeg se generira poveznica koju kada korisnik otvori, omogućuje mu se izrada narudžbe za taj stol. Aplikacija osim poveznice za svaki stol generira QR kod koji vodi na tu poveznicu za stvaranje narudžbe. Zamišljeno je da konobar ispiše QR kod na papir i stavi ga na stol te će tako klijent kada sjedne za stol moći mobitelom skenirati QR kod koji će mu odmah otvoriti aplikaciju gdje će moći napraviti narudžbu za taj stol. Nakon što korisnik napravi narudžbu, konobar će u aplikaciji dobiti obavijest da je na tom stolu napravljena nova narudžba te će tako znati na koji stol mora odnijeti naručene stavke. Za generiranje i prikaz QR koda korištena je „react-qr-svg“ biblioteka koja pruža komponentu koja generira i prikazuje QR kod.



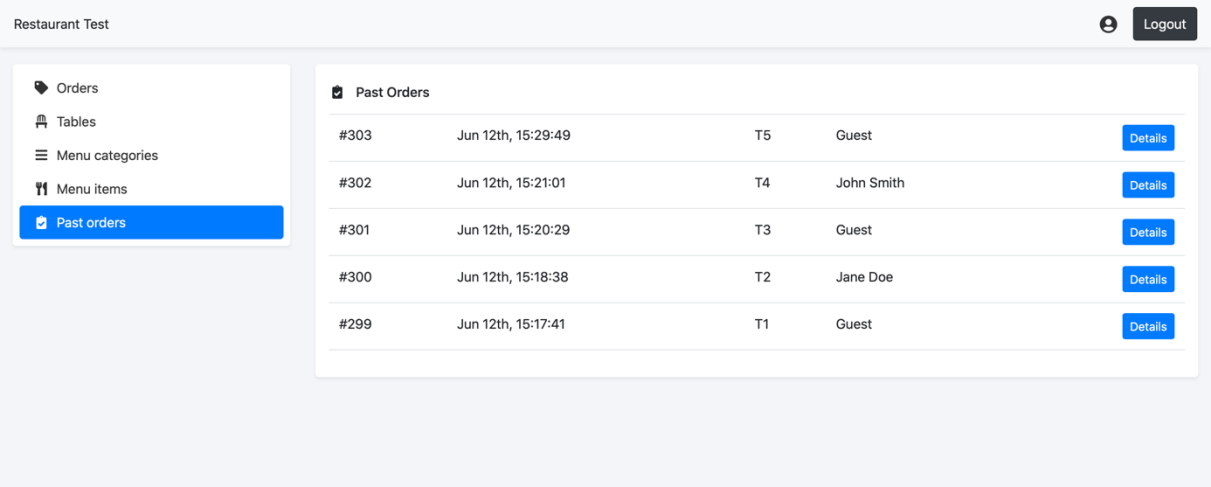
Slika 18: Prikaz svih kategorija menija s mogućnošću dodavanja novih i uređivanja postojećih kategorija

Konobar može dodavati kategorije menija (slika 18) u koje može dodavati stavke (slika 19). Stavke je moguće pretraživati prema nazivu i kategoriji.



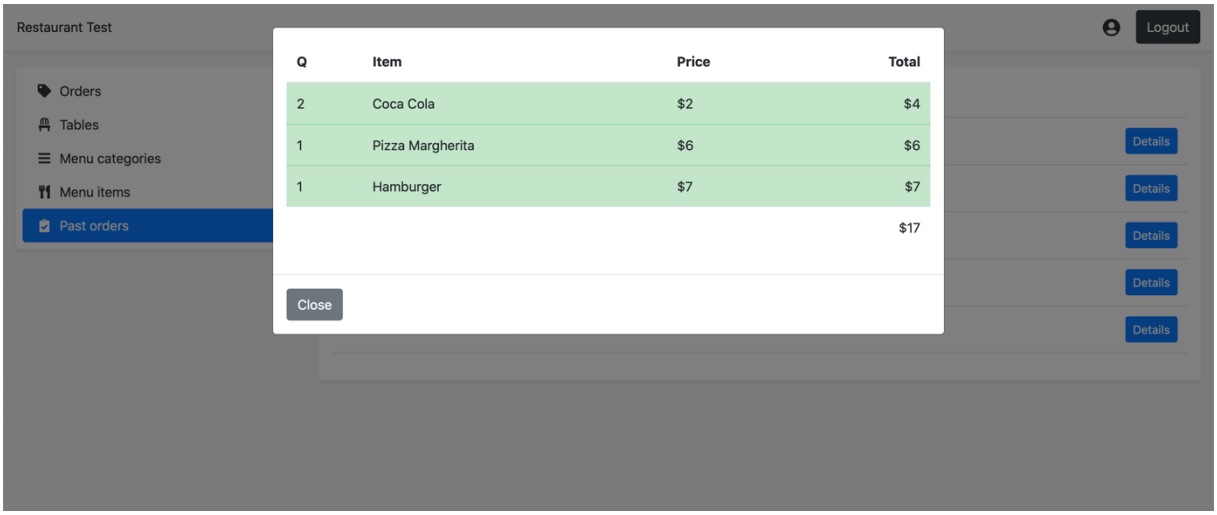
Slika 19: Prikaz svih stavaka menija s mogućnošću dodavanja novih te uređivanja postojećih stavaka

Konobar također može vidjeti sve dosadašnje narudžbe restorana što je prikazano na slici 20. Za svaku je narudžbu prikazan identifikator, datum i vrijeme, stol te naziv klijenta.



Slika 20: Prikaz svih prošlih narudžbi s mogućnošću pregleda stavaka i detalja odabrane narudžbe

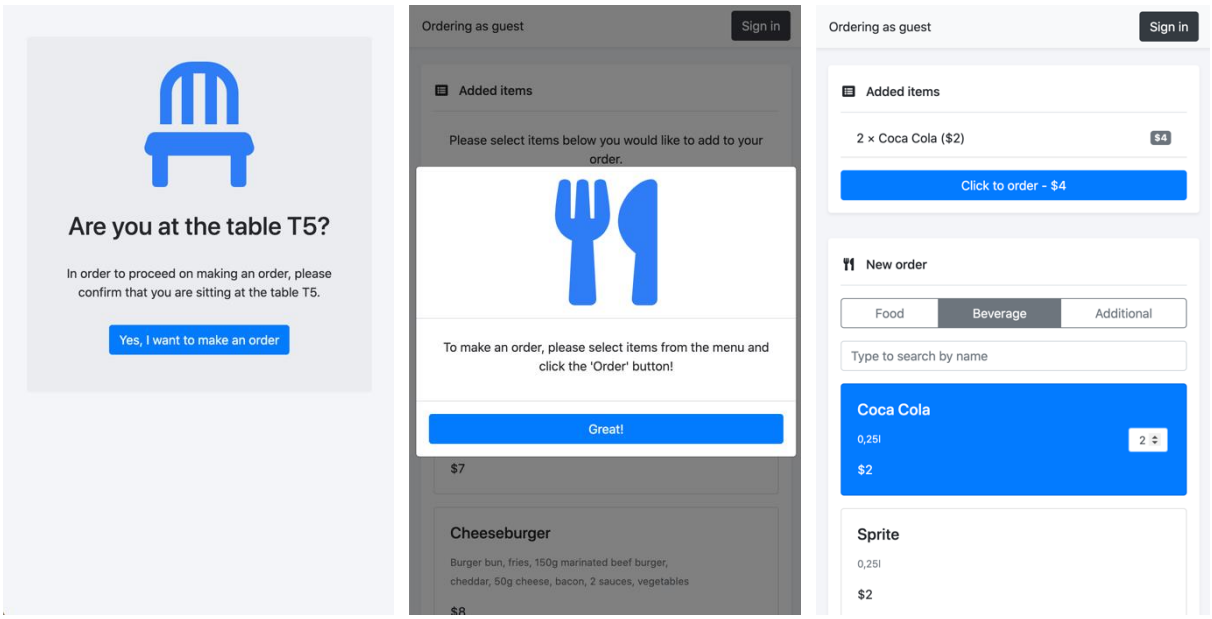
Klikom na gumb „Details“, moguće je vidjeti stavke narudžbe kao što je to prikazano na slici 21.



Slika 21: Prikaz stavaka odabrane narudžbe

6.4.4. Provođenje narudžbe u realnom vremenu

Kada korisnik dođe u restoran i skeniranjem QR koda na stolu otvori aplikaciju, prvo potvrđuje nalazi li se na stolu čiju poveznicu je otvorio skeniranjem koda. Nakon potvrde prikazuju mu se upute za izradu nove narudžbe nakon čega može odabrati stavke i količinu stavaka iz menija restorana koje želi naručiti (slika 22).

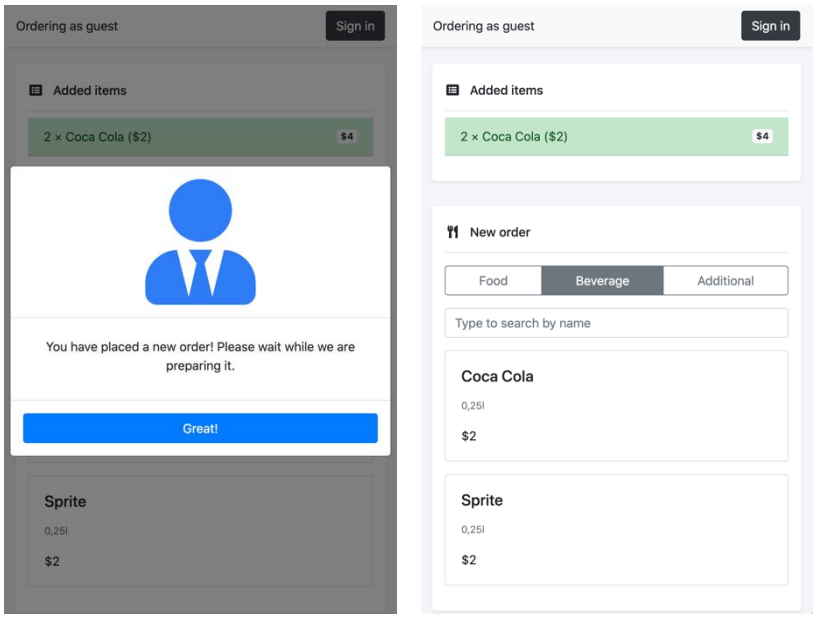


Slika 22: Stvaranje nove narudžbe na strani klijenta

Pritiskom na gumb za naručivanje, poziva se slijedeći kod koji koristeći Socket.io biblioteku na korisničkoj strani šalje poruku tipa *order* na poslužitelj (slika 11).

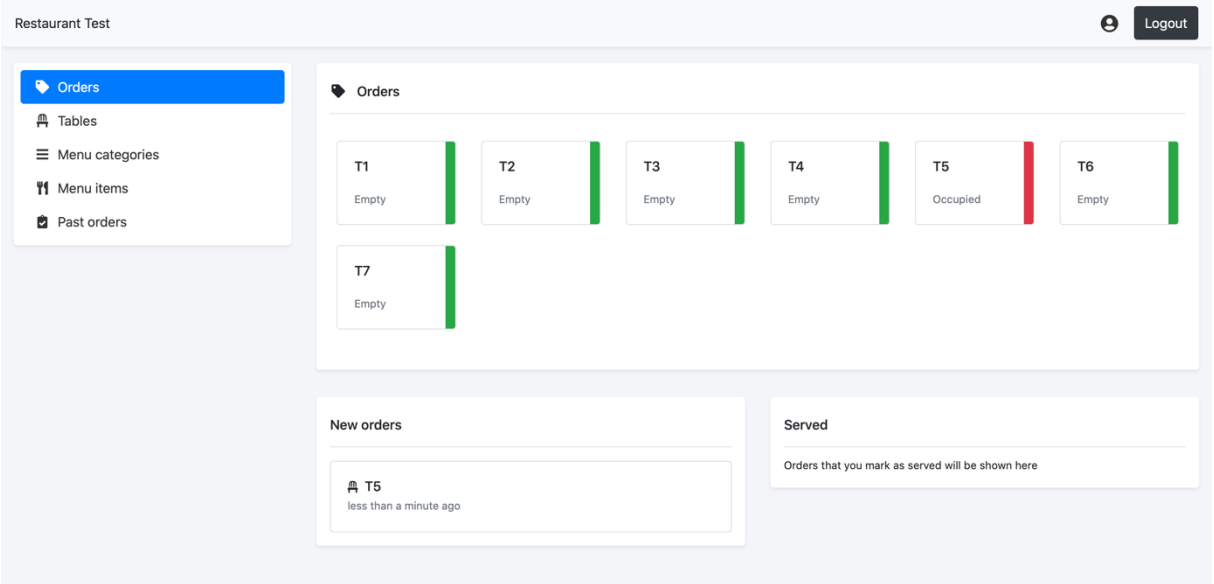
```
1   // Pripremi sadržaj poruke
2   const orderMessage = {
3     table_id: tableID,
4     customer_id: user.user_id,
5     items: addedItems,
6   };
7
8   // Pošalji poruku tipa 'order' poslužitelju
9   socket.emit("order", orderMessage);
10
11  // Promijeni status narudžbe na strani klijenta
12  setOrderStatus(ORDER.PLACED);
```

U liniji 12 mijenja se status narudžbe na strani klijenta. Promjenom statusa narudžbe prikazuje se modal o uspješno napravljenoj narudžbi te se prilagođava korisničko sučelje trenutnim statusom narudžbe (slika 23).



Slika 23: Uspješno stvaranje nove narudžbe na strani klijenta

Nakon što je korisnik napravio narudžbu, čeka konobara da mu je posluži (slika 23) te će nakon toga imati mogućnost dodavanja novih stavaka u narudžbu i traženje računa. Konobaru se napravljena narudžba prikazuje u sekciji „New orders“ te se ažurira prikaz stolova s promijenjenim statusom stola na kojem je narudžba napravljena (slika 24).



Slika 24: Prikaz napravljene narudžbe na strani konobara

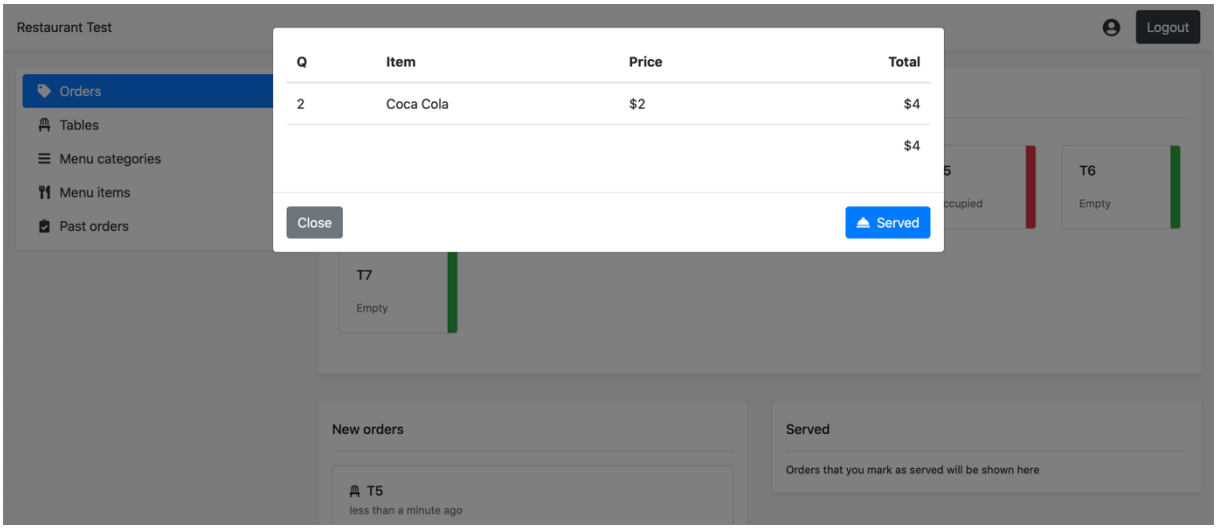
Jedan od tipova poruka koji strana konobara osluškuje je upravo poruka *order* koju konobar prima nakon što klijent napravi novu narudžbu (slika 11). Slijedi dio koda koji je zaslužan za obradu primljene poruke tipa *order* na strani konobara:

```
1  // Osluškivač poruke tipa 'order'
2  socket.on("order", (order) => {
3
4    // Prikaži stol kao zauzet
5    setOccupied(order);
6
7    // Prikaži novu narudžbu u sekciji „New orders"
8    setOrders((old) => [...old, order]);
9  });
```

Pri dolasku poruke o novoj narudžbi, u liniji 5 poziva se funkcija koja mijenja status stola na korisničkom sučelju iz „Empty" u „Occupied" (slika 24). U liniji 8 mijenja se stanje (eng. *state*) trenutne komponente koje je ranije u kodu definirano slijedećim kodom kao polje narudžbi:

```
1  const [orders, setOrders] = useState([]);
```

Navedeno stanje mijenja se na način da se narudžbama koje se već nalaze u polju dodaje nova narudžba koja je pristigla tom porukom.

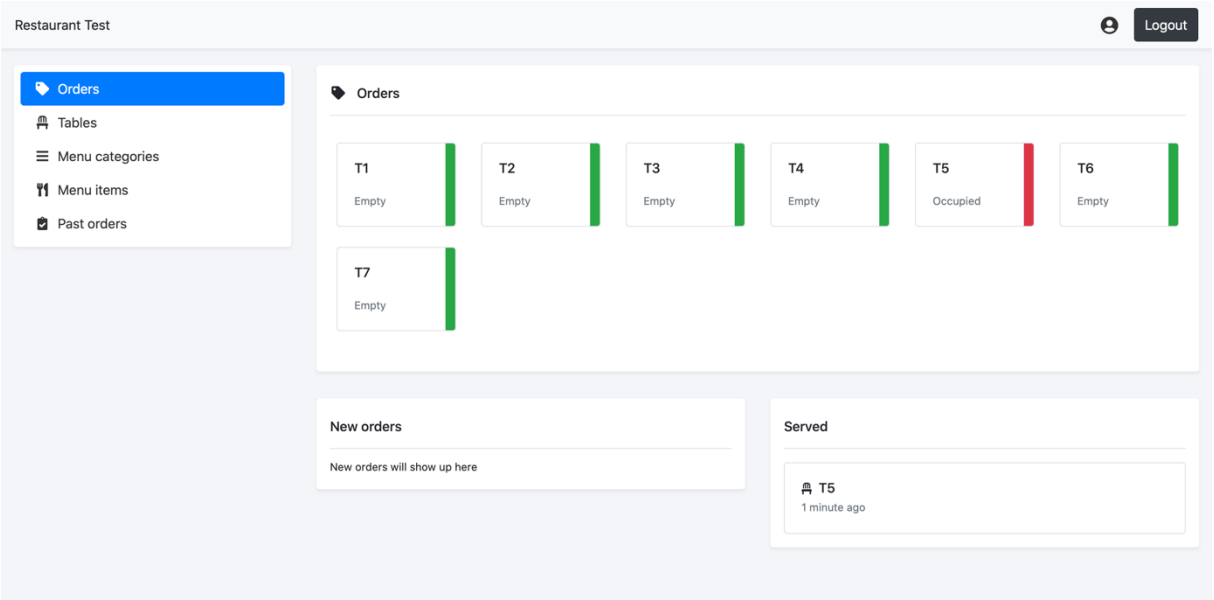


Slika 25: Prikaz stavaka narudžbe s mogućnošću označavanja narudžbe kao poslužena

Klikom na pristiglu narudžbu u sekciji „New orders", konobar može vidjeti naručene stavke (slika 25) te nakon što ih posluži klijentu, klikom na gumb „Served" narudžbu označava kao poslužena te se prema poslužitelju šalje poruka tipa *orderServed* (slika 11). Slijedi dio koda iz

funkcije koja se izvršava klikom na gumb „Served“ koji šalje *orderServed* poruku na poslužitelj koja sadrži identifikator narudžbe koja je označena kao poslužena:

```
1   socket.emit("orderServed", { order_id });
```



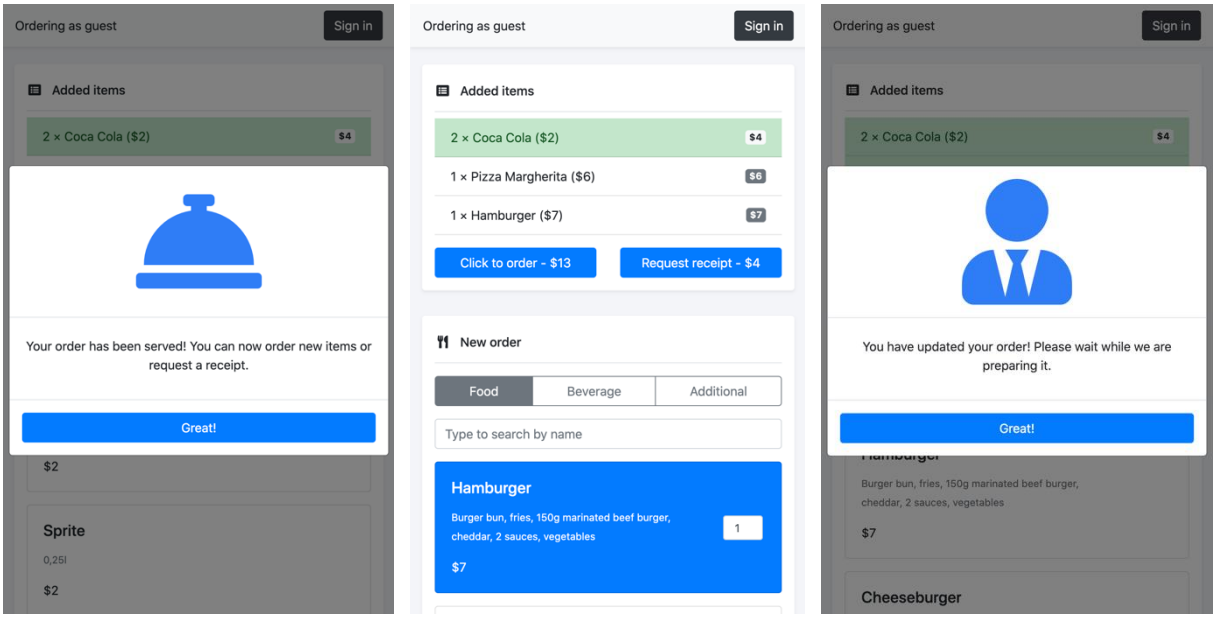
Slika 26: Prikaz zaslona za pregled narudžbi nakon što je narudžba označena kao poslužena

Nakon što je narudžba označena kao poslužena, ona se konobaru prebacuje u sekciju „Served“ (slika 26), a korisniku se daje mogućnost dodavanja novih stavaka u narudžbu ili traženje računa (slika 27). Da bi korisnik znao kada je njegova narudžba označena kao poslužena, korisnička strana aplikacije sluša *orderServed* poruke, što je prikazano kodom koji slijedi:

```
1  // Osluškivač poruke tipa 'orderServed'
2  socket.on("orderServed", ({ order_id }) => {
3
4    // Promijeni status narudžbe na strani klijenta
5    setOrderStatus(ORDER.SERVED);
6
7    // Zabilježi identifikator narudžbe
8    setOrderID(order_id);
9  });
```

U liniji 5 opet se mijenja status narudžbe na temelju kojeg se korisničko sučelje prilagođava statusu – omogućuje se odabir stavaka iz menija te se prikazuju gumbi za potvrdu narudžbe i

za traženje računa (slika 27). Stavke koje su već poslužene su označene zelenom bojom, dok su one novo dodane označene bijelom bojom.



Slika 27: Nakon što je narudžba poslužena, korisnik ima mogućnost dodavanja novih stavaka ili traženje računa

Ukoliko korisnik odluči dodati nove stavke u narudžbu, nakon što ih odabere i klikne na gumb za narudžbu, prema poslužitelju se šalje poruka tipa *orderUpdate* (slika 11). Slijedi dio koda koji šalje orderUpdate poruku prema poslužitelju nakon što korisnik potvrdi narudžbu:

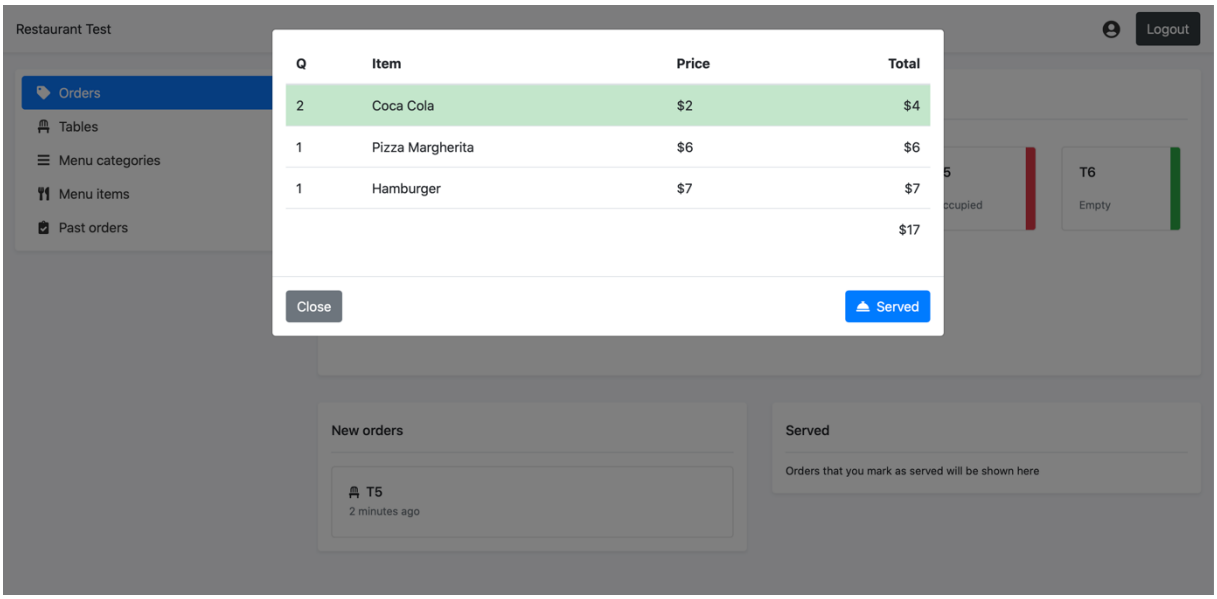
```
1   // Pripremi sadržaj poruke
2   const orderMessage = {
3     order_id: orderID,
4     table_id: tableID,
5     customer_id: user.user_id,
6     items: addedItems,
7   };
8
9   // Pošalji poruku tipa 'orderUpdate' poslužitelju
10  socket.emit("orderUpdate", orderMessage);
11
12  // Promijeni status narudžbe na strani klijenta
13  setOrderStatus(ORDER.PLACED)
```

Princip slanja poruke tipa *orderUpdate* veoma je sličan slanju poruke *order*. Razlika je u tome što se poruci *orderUpdate* prilaže također i identifikator narudžbe (order_id) kako bi poslužitelj znao kojoj narudžbi u bazi podataka treba dodati nove stavke. Konobar poruku *orderUpdate*

zaprima od poslužitelja, a logika obrade zaprimljene poruke je slična onoj za obradu poruke tipa *order*:

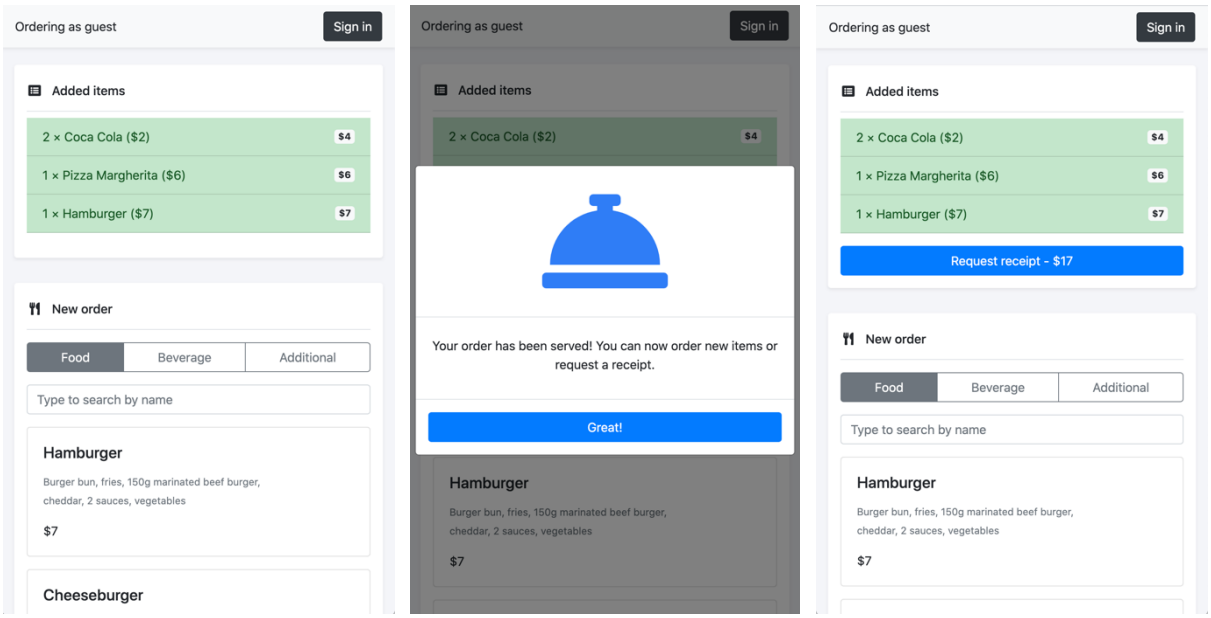
```
1  // Osluškivač poruke tipa 'orderUpdate'
2  socket.on("orderUpdate", (order) => {
3
4    // Prikaži ažuriranu narudžbu u sekciji „New orders“
5    setOrders((old) => [...old, order]);
6  });
```

Narudžba se konobaru opet prikazuje u sekciji „New orders“ te se klikom na nju prikazuju stavke te narudžbe. Zelenom bojom označene su stavke koje su već ranije poslužene, a bijelom bojom su označene nove stavke narudžbe (slika 28).



Slika 28: Prikaz novo dodanih stavaka narudžbe

Klikom na gumb „Served“, poslužitelju se ponovo na isti način šalje poruka *orderServed* koja se zatim prosljeđuje klijentu kojemu se prikazuje poruka o tome da je njegova narudžba poslužena. Korisnik ponovo ima opciju dodavanja novih stavaka u narudžbu ili traženja računa. Ukoliko ne odabere nove stavke, korisniku je ponuđena samo opcija traženja računa (slika 29).

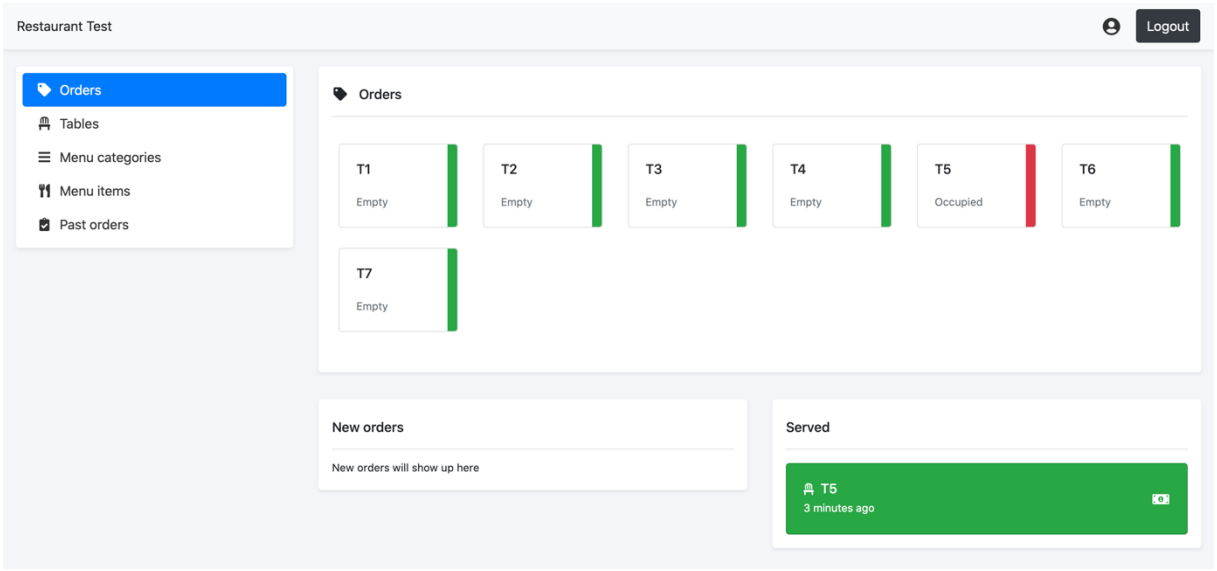


Slika 29: Nakon posluživanja njegove narudžbe, korisnik ima mogućnost traženja računa

Ukoliko korisnik želi zatražiti račun i završiti narudžbu, to može učiniti klikom na gumb „Request receipt“ (slika 29) te se tada poslužitelju šalje *requestReceipt* poruka koja se proslijeđuje konobaru (slika 11). Klikom na gumb „Request receipt“ izvršava se slijedeći kod:

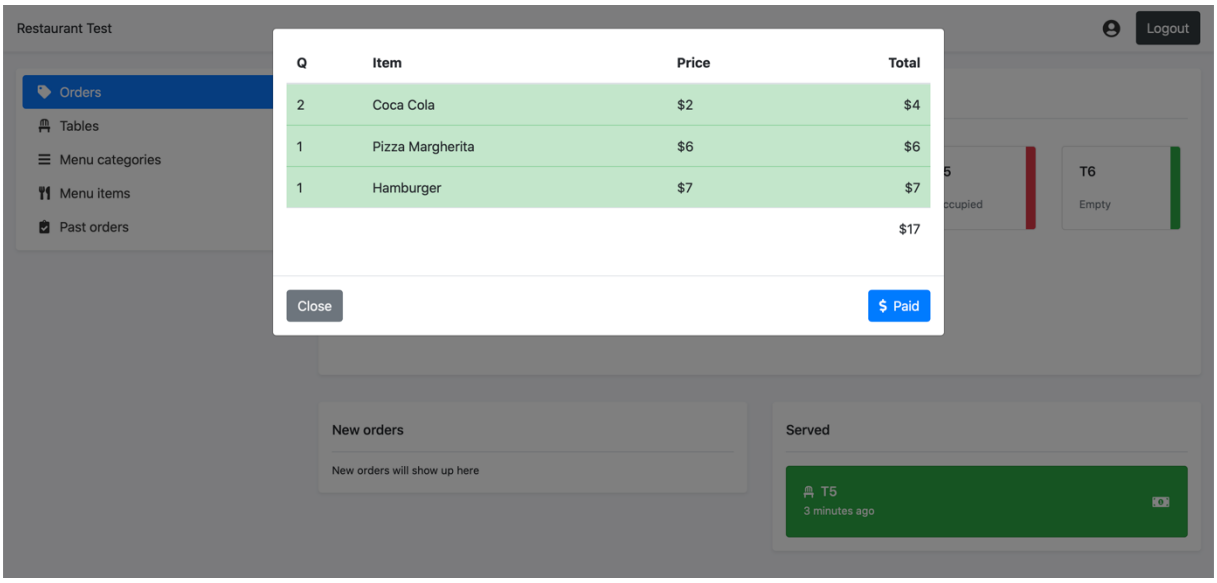
```
1  // Pošalji poslužitelju 'requestReceipt' poruku
2  socket.emit("requestReceipt", { order_id: orderID });
3
4  // Promijeni status narudžbe na strani klijenta
5  setOrderStatus(ORDER.RECEIPT_REQUESTED);
```

Aplikacija na strani konobara osluškuje poruke tipa *requestReceipt* te pri primitku poruke ove vrste narudžbu za koju je zatražen račun označava zelenom bojom (slika 30).



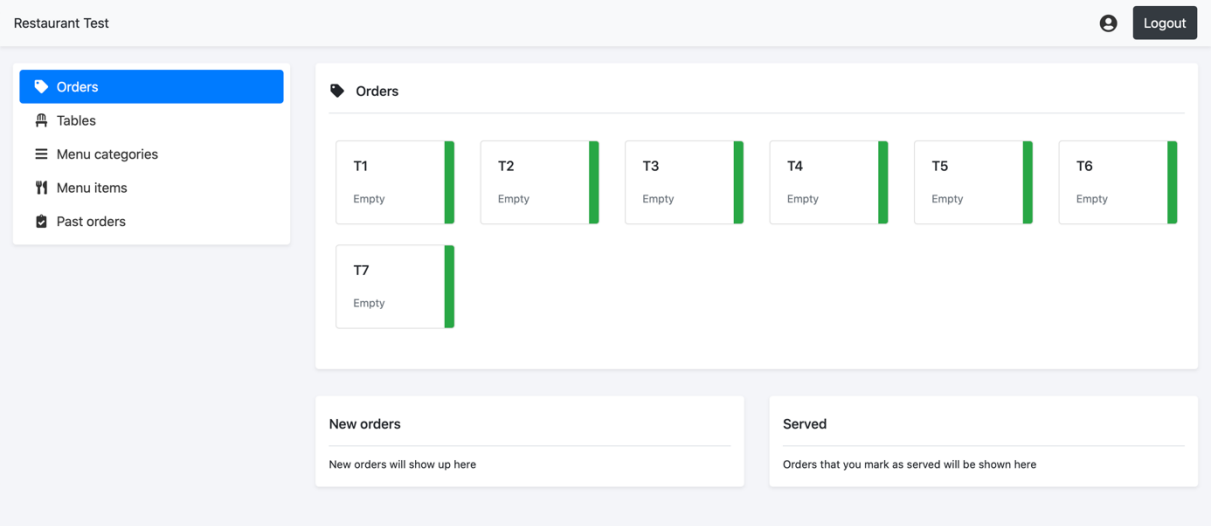
Slika 30: Prikaz narudžbe za koju je zatražen račun na strani konobara

Klikom na narudžbu za koju je zatražen račun prikazuju se sve stavke računa te se konobaru daje mogućnost označavanja računa kao plaćenog (slika 31).



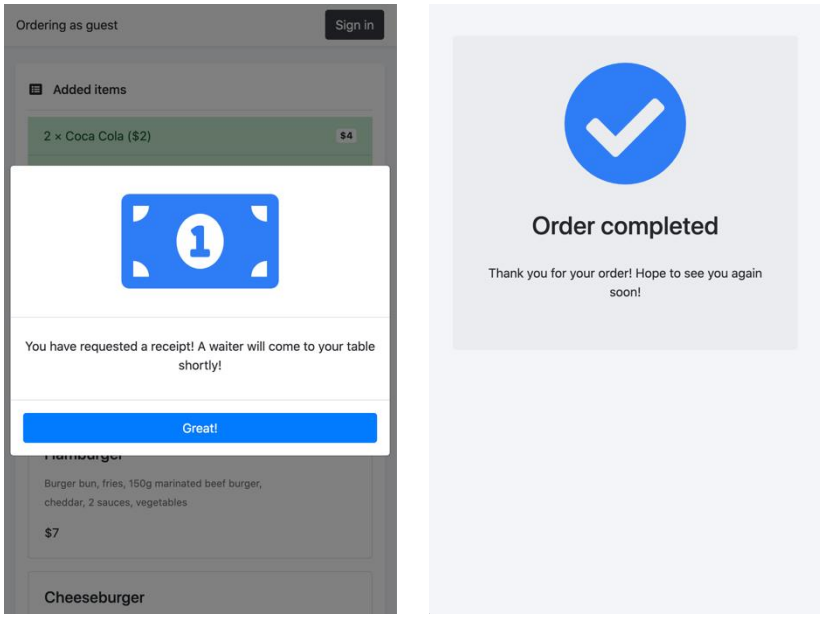
Slika 31: Prikaz stavaka računa s mogućnošću označavanja računa kao plaćenog

Klikom na gumb „Paid“, na poslužitelj se šalje poruka tipa *orderComplete* (slika 11) koja sadrži identifikator narudžbe. Status stola na kojem je ta narudžba napravljena mijenja se iz „Occupied“ u „Empty“ te se na strani konobara ažurira prikaz stolova (slika 32).



Slika 32: Prikaz sučelja konobara nakon završene narudžbe i ažuriranja prikaza stolova

Aplikacija na strani klijenta osluškuje poruku tipa *orderComplete* te se pri dolasku poruke tog tipa klijentu prikazuje poruka o uspješnoj narudžbi (slika 33).



Slika 33: Prikaz sučelja nakon završene narudžbe na strani klijenta

7. Zaključak

Cilj ovog rada bio je objasniti pojam komunikacije u realnom vremenu te istražiti načine na koje se komunikacija u realnom vremenu može implementirati na webu. Na početku rada objašnjeni su neki aspekti modernog weba koji imaju značaj za razvoj web aplikacija te su relevantni temi komunikacije u realnom vremenu na webu. Objašnjeno je i demonstrirano nekoliko načina implementacije komunikacije u realnom vremenu na webu koji se mogu podijeliti na implementacije pomoću HTTP protokola te na implementaciju pomoću WebSocket protokola.

Polling i long polling su poprilično jednostavne HTTP tehnike koje se implementiraju pomoću AJAX-a, no imaju svoje nedostatke. Glavna mana HTTP načina implementacija jest ponavljajuće slanje HTTP zaglavlja koja čine dodatan teret zbog kojeg je vrijeme prijenosa podataka duže. Unatoč nedostacima, ova se metoda danas često koristi zahvaljujući jednostavnošću implementacije i podržanosti svih preglednika, iako zbog svoje neefikasnosti u usporedbi s drugim tehnikama nije preporučana. Server-Sent Events tehnika je efikasan način razmjene podataka u realnom vremenu na webu pomoću HTTP protokola pošto se za cijelo vrijeme razmjene podataka koristi samo jedna veza. Iako je ograničenje ove tehnike komunikacija u samo jednom smjeru, postoji mnogo slučajeva u kojima se ona koristi.

WebSocket protokol je moderno rješenje implementacije komunikacije u realnom vremenu na webu koje omogućuje potpuno dvosmjernu komunikaciju. Upravo iz tog je razloga ova tehnika odabrana za razvoj aplikacije u praktičnom dijelu rada. Implementacijom aplikacije otkrio sam široke mogućnosti koje pruža Socket.io biblioteka, a i sam WebSocket protokol u usporedbi s HTTP metodama implementacije komunikacije u realnom vremenu na webu. React biblioteka uvelike mi je svojim deklarativnim pristupom omogućila brz razvoj interaktivnog korisničkog sučelja, dok je na strani poslužitelja Node.js radna okolina svojom proširivošću zaslužna za implementaciju poslovne logike aplikacije koristeći biblioteke kao što su Express i Socket.io.

Iako sam već bio upoznat s širokom upotrebom komunikacije u realnom vremenu na webu, izradom ovog rada naučio sam mnogo o važnosti iste te o mogućnostima koje ona pruža. Smatram da je prije same izrade aplikacije korisno proučiti one najbitnije i najviše korištene načine implementacije jer svaki ima svoje specifičnosti po kojima će možda za potrebe aplikacije koja se razvija biti prikladniji od drugog. Obradom ovog malog dijela široke teme modernog weba naučio sam mnogo o razvoju aplikacija s komunikacijom u realnom vremenu što će mi uvelike pomoći u budućem razvoju web aplikacija.

8. Popis literature

[1] K. Johannessen, „Real Time Web Applications: Comparing frameworks and transport mechanisms“ [Diplomski rad]. University of Oslo, Norveška, 2014, Dostupno na: https://www.duo.uio.no/handle/10852/42049 [Pristupljeno: 25-ožu-2020]

[2] „An Introduction to JavaScript“, 2020. [Na internetu]. Dostupno na: https://javascript.info/intro [Pristupljeno: 25-tra-2020]

[3] P. Patel, „What exactly is Node.js?“, 2018. [Na internetu]. Dostupno na: https://www.freecodecamp.org/news/what-exactly-is-node-js-ae36e97449f5/ [Pristupljeno: 25-svi-2020]

[4] „Express“ (bez dat.) [Na internetu]. Dostupno na: https://expressjs.com [Pristupljeno: 26-svi-2020]

[5] „React“ (bez dat.) [Na internetu]. Dostupno na: https://reactjs.org [Pristupljeno: 26-svi-2020]

[6] „Understanding the Virtual DOM“, 2018. [Na internetu]. Dostupno na: https://bitsofco.de/understanding-the-virtual-dom/ [Pristupljeno: 27-svi-2020]

[7] „JSX and the Virtual DOM“, 2020. [Na internetu]. Dostupno na: https://www.newline.co/fullstack-react/p/jsx-and-the-virtual-dom/ [Pristupljeno: 27-svi-2020]

[8] T. Rascia, “What is Bootstrap and How Do I Use It?”, 2015. [Na internetu]. Dostupno na: https://www.taniarascia.com/what-is-bootstrap-and-how-do-i-use-it/ [Pristupljeno: 27-svi-2020]

[9] “What Socket.IO is” (bez dat.) [Na internetu]. Dostupno na: https://socket.io/docs/ [Pristupljeno: 28-svi-2020]

[10] A. Staff, “Static vs Dynamic Website: What Is the Difference?”, 2019. [Na internetu]. Dostupno na: https://wpamelia.com/static-vs-dynamic-website/ [Pristupljeno: 2-tra-2020]

[11] P. Jain, “Static Vs Dynamic Website: Advantages And Disadvantages”, 2017. [Na internetu]. Dostupno na: https://www.weblinkindia.net/blog/static-vs-dynamic-website-advantages-disadvantages [Pristupljeno: 4-tra-2020]

[12] R. Gibb, “What is a Web Application?”, 2016. [Na internetu]. Dostupno na: https://blog.stackpath.com/web-application/ [Pristupljeno: 5-tra-2020]

[13] J. J. Garrett, „Ajax: A New Approach to Web Applications“, 2005. [Na internetu]. Dostupno na: https://courses.cs.washington.edu/courses/cse490h/07sp/readings/ajax_adaptive_path.pdf [Pristupljeno: 25-ožu-2020]

[14] A. Swartz, „A Brief History of Ajax“, 2005. [Na internetu]. Dostupno na: http://www.aaronsw.com/weblog/ajaxhistory [Pristupljeno: 26-ožu-2020]

[15] „XMLHttpRequest.readyState“, 2019. [Na internetu]. Dostupno na: https://developer.mozilla.org/en-US/docs/Web/API/XMLHttpRequest/readyState [Pristupljeno: 27-ožu-2020]

[16] “How AJAX Works: 10 Practical Uses For AJAX”, 2009. [Na internetu]. Dostupno na: https://www.jotform.com/blog/how-ajax-works/ [Pristupljeno: 26-ožu-2020]

[17] J. Lengstorf, P Leggetter, *Realtime Web Apps: With HTML5 WebSocket, PHP, and jQuery*. New York: Apress, 2013.

[18] R. Selmer, „What Is Real Time Communications?“, 2017. [Na internetu]. Dostupno na: https://www.vonage.com/resources/articles/real-time-communications/ [Pristupljeno: 30-ožu-2020]

[19] „An overview of HTTP“, 2019. [Na internetu]. Dostupno na: https://developer.mozilla.org/en-US/docs/Web/HTTP/Overview [Pristupljeno: 10-tra-2020]

[20] S. Basu, „Real-Time Communication Techniques“, 2019. [Na internetu]. Dostupno na: https://www.telerik.com/blogs/real-time-communication-techniques [Pristupljeno: 10-tra-2020]

[21] „Server-Sent Events (SSE): A conceptual deep dive“ Dostupno na: https://www.ably.io/concepts/server-sent-events [Pristupljeno: 16-tra-2020]

[22] N. Babel, „What is Server-Sent Events?“, 2017. [Na internetu]. Dostupno na: https://apifriends.com/api-streaming/server-sent-events/ [Pristupljeno: 16-tra-2020]

[23] K. Sookocheff, „How do WebSockets Work?“, 2019. [Na internetu]. Dostupno na: https://sookocheff.com/post/networking/how-do-websockets-work/ [Pristupljeno: 19-tra-2020]

[24] Carnet, “Sigurnost HTTP REST API-ja”, 2020. [Na internetu]. Dostupno na: https://www.cert.hr/wp-content/uploads/2020/03/Sigurnost_HTTP_API-ja.pdf [Pristupljeno: 17-tra-2020]

[25] “Introduction to JSON Web Tokens” (bez dat.) [Na internetu]. Dostupno na: https://jwt.io/introduction/ [Pristupljeno: 25-svi-2020]

[26] “About Node.js” (bez dat.) [Na internetu]. Dostupno na: https://nodejs.org/en/about/ [Pristupljeno: 25-svi-2020]

[27] “What are WebSockets?” (bez dat.) [Na internetu]. Dostupno na: https://pusher.com/websockets [Pristupljeno: 20-tra-2020]

[28] J. Butz, “Server-Sent Events With Node”, 2018. [Na internetu]. Dostupno na: https://jasonbutz.info/2018/08/server-sent-events-with-node/ [Pristupljeno: 17-tra-2020]

[29] S. Prickett, “A Look at Server-Sent Events”, 2019. [Na internetu]. Dostupno na: https://medium.com/conectric-networks/a-look-at-server-sent-events-54a77f8d6ff7 [Pristupljeno: 16-tra-2020]

[30] “The WebSocket API (WebSockets)” (bez dat.) [Na internetu]. Dostupno na: https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API [Pristupljeno: 11-svi-2020]

9. Popis slika

Slika 1: Prikaz dohvaćanja statične web stranice 5

Slika 2: Prikaz dohvaćanja dinamične web stranice 6

Slika 3: Prikaz rada tradicionalne web aplikacije (lijevo) i web aplikacije korištenjem AJAX tehnologija (desno) [13] 9

Slika 4: Relacija „narudzbe“ u koju se spremaju narudžbe 14

Slika 5: Prikaz komunikacije klijenta i poslužitelja tehnikom polling 16

Slika 6: Prikaz komunikacije klijenta i poslužitelja tehnikom long polling 19

Slika 7: Prikaz komunikacije klijenta i poslužitelja tehnikom Server-Sent Events 21

Slika 8: Prikaz komunikacije klijenta i poslužitelja putem WebSocket 24

Slika 9: Prikaz komunikacije između konobara i korisnika 25

Slika 10: ERA model aplikacije 29

Slika 11: Dijagram slijeda razmjene poruka putem WebSocket protokola 37

Slika 12: Zaslon prijave u aplikaciju 44

Slika 13: Forma za registraciju 45

Slika 14: Pregled svih dosadašnjih narudžbi (registrirani korisnik) 45

Slika 15: Prikaz narudžbi i statusa stolova u realnom vremenu 46

Slika 16: Prikaz svih stolova restorana s mogućnošću dodavanja novih te uređivanja postojećih 46

Slika 17: Prikaz modala za dijeljenje poveznice i ispis QR koda odabranog stola 47

Slika 18: Prikaz svih kategorija menija s mogućnošću dodavanja novih i uređivanja postojećih kategorija 47

Slika 19: Prikaz svih stavaka menija s mogućnošću dodavanja novih te uređivanja postojećih stavaka 48

Slika 20: Prikaz svih prošlih narudžbi s mogućnošću pregleda stavaka i detalja odabrane narudžbe 48

Slika 21: Prikaz stavaka odabrane narudžbe 49

Slika 22: Stvaranje nove narudžbe na strani klijenta 50

Slika 23: Uspješno stvaranje nove narudžbe na strani klijenta.....51

Slika 24: Prikaz napravljene narudžbe na strani konobara 51

Slika 25: Prikaz stavaka narudžbe s mogućnošću označavanja narudžbe kao poslužena....52

Slika 26: Prikaz zaslona za pregled narudžbi nakon što je narudžba označena kao poslužena 53

Slika 27: Nakon što je narudžba poslužena, korisnik ima mogućnost dodavanja novih stavaka ili traženje računa 54

Slika 28: Prikaz novo dodanih stavaka narudžbe.....55

Slika 29: Nakon posluživanja njegove narudžbe, korisnik ima mogućnost traženja računa...56

Slika 30: Prikaz narudžbe za koju je zatražen račun na strani konobara 57

Slika 31: Prikaz stavaka računa s mogućnošću označavanja računa kao plaćenog 57

Slika 32: Prikaz sučelja konobara nakon završene narudžbe i ažuriranja prikaza stolova.....58

Slika 33: Prikaz sučelja nakon završene narudžbe na strani klijenta.....58

10. Popis tablica

Tablica 1: Prikaz krajnjih točaka čija funkcija je autentifikacija korisnika 32

Tablica 2: Krajnje točke za upravljanje podacima o stolovima restorana.....34