Aplikacijski okviri za web aplikacije

Milotić, Vinko

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Vinko Milotić

APLIKACIJSKI OKVIRI ZA *WEB*-APLIKACIJE

ZAVRŠNI RAD

Varaždin, 2023.

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Vinko Milotić

OIB: 40531652598

Studij: Informacijski sustavi

APLIKACIJSKI OKVIRI ZA *WEB*-APLIKACIJE

ZAVRŠNI RAD

Mentor:

prof. dr. sc. Dragutin Kermek

Varaždin, rujan 2023.

Vinko Milotić

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 korištene su etički prikladne i prihvatljive metode i tehnike rada.

Autor potvrdio prihvaćanjem odredbi u sustavu FOI-radovi

Sažetak

U teorijskom dijelu rada daje se pregled osobina jezika HTML i CSS te okvira TailwindCSS. U nastavku se opisuje uloga programskog jezika u realizaciji korisničke strane *web*-aplikacije te se prikazuju glavne osobine programskog jezika JavaScript. Slijedi opis uloge programskih okvira na korisničkoj strani uz detaljniji opis ReactJs i Svelte te njihova usporedba. Prijelazom na stranu poslužitelja opisuju se osobine programskih jezika za tu namjenu uz poseban osvrt na NodeJs. Opisom i analizom programskih okvira ExpressJs i NestJS daje se temelj za stvaranje sučelja za programiranje aplikacija (engl. Application Programming Interface, API).

Praktični dio rada sastoji se od detaljnog opisa funkcionalnosti aplikacije uz koji postoje određeni dijagramski prikazi koji služe za jasnije razumijevanje rada aplikacije. Slijedi opis implementacije *web*-aplikacija u dvije verzije s obzirom na programske okvire koji će se koristiti u pojedinoj od njih. Na kraju slijedi opis vlastitih iskustava u razvoju pojedine verzije *web*-aplikacije s obzirom na korištene programske okvire.

Ključne riječi: ReactJs; Svelte; NodeJs; ExpressJs; NestJs; API; HTML; CSS; TailwindCSS

Sadržaj

1. Uvod....1

2. Osnove *web*-aplikacija....2

2.1. HTML i CSS....2

2.1.1. Uvod u HTML....2

2.1.2. Uvod u CSS....4

2.1.2.1. CSS tranzicije i animacije....5

2.2. Uvod u JavaScript....7

2.2.1. Osnove JavaScript varijabli....7

2.2.1. Osnove JavaScript funkcija....8

2.3. TypeScript....9

3. Okviri korisničke strane....11

3.1. Uvod u okvire JavaScripta....11

3.2. ReactJs....11

3.2.1. Virtualni DOM....12

3.2.2. JSX u Reactu....13

3.2.3. Kuke (engl. Hooks)....13

3.2.4. Rekviziti (eng. props)....15

3.3. Svelte....16

3.3.1. Svelte vrsta komponenata....16

3.3.2. Korištenje komponenata....17

3.3.3. Manipuliranje stanjem....18

3.3.4. Rekviziti (eng. props)....19

3.3.5. Svelte spremišta (eng. stores)....19

3.3.6. SvelteKit....22

4. TailwindCSS....23
4.1. Razlika od drugih CSS okvira....23
4.2. Tranzicije i animacije....25
4.3. Prilagodljiv dizajn....25
5. Okviri pozadinske strane....27
5.1. Node.js....27
5.1.1. Asinkron način izvršavanja procesa....27
5.1.2. Node menadžer paketa (eng. Node Packet Manager, npm)....27
5.2. Express.js....27
5.2.1. Komunikacija korisničke i poslužiteljeve strane....28
5.2.2. Parametri zahtjeva....29
5.2.3. Odgovori na zahtjeve....30
5.2.4. CORS....31
5.3. NestJS....31
5.3.1. Nest CLI....32
5.3.2. Dekoratori....32
5.3.3. Ubacivanje ovisnosti (engl. Dependency Injection)....33
5.3.4. Davatelji usluga....34
5.3.5. Moduli....35
5.3.6. Upravljači....36
5.3.7. TypeORM....37
6. Baza podataka....40
6.1. PostgreSQL....40
7. Arhitekture za razvoj aplikacija....43
7.1. Redux....43
7.2. Komponentno bazirana arhitektura....44

8. Praktični dio rada.....46
8.1. Odabir slučaja korištenja.....46
8.2. Rad aplikacija u uljari.....46
8.3. Dijagrami i vizualizacija rada uljare i aplikacija.....48
8.4. Prva aplikacija.....52
8.5. Druga aplikacija.....75
8.6. Dizajn aplikacija.....89
9. Usporedba aplikacija.....93
9.1. Usporedba okvira korisničke strane.....93
9.2. Usporedba okvira poslužiteljeve strane.....94
10. Završetak.....96
11. Popis literature.....97
12. Popis slika.....99

1. Uvod

Razlika između nekadašnjih i današnjih mrežnih aplikacija jako je velika. S vremenom i zahtjevima od kupaca i korisnika mrežnih aplikacija njihov izgled i funkcionalnosti drastično su se promijenili. Također, kako su se mijenjale aplikacije, mijenjali su se programski jezici, aplikacijski okviri i njihovi alati kako bi efikasnije i efektivnije stvarali i održavali mrežne aplikacije.

Hipertekstualni jezik oznaka (HTML), stilski jezik CSS te programski jezik JavaScript postali su standard stvaranja i održavanja mrežnih aplikacija, ali svejedno s vremenom su se i oni mijenjali po zahtjevima i potrebama tržišta. Kako se JavaScript razvijao, tako je došlo i do razvoja JavaScript aplikacijskih okvira za razne svrhe. Primjerice, ReatcJS je aplikacijki okvir koji služi za stvaranje korisničkog dijela aplikacije, dok NestJs služi za stvaranje poslužiteljevog dijela aplikacije. Cilj ovoga rada je proučiti neke od aplikacijskih okvira JavaScripta s poslužiteljeve strane, ali i korisničke te vidjeti kako oni funkcioniraju te je li jedan bolji od drugoga, kako i zašto.

Dakako, u ovome završnom radu govorit će se i o HTML-u i CSS-u koji su osnove i baza stvaranja svih mrežnih aplikacija. Nakon toga proučit će se JavaScript i njegovi aplikacijski okviri s kojima će se (zajedno s HTML-om i CSS-om) napraviti dvije mrežne aplikacije. Te aplikacije bit će uspoređivane s aspekta jednostavnosti programiranja, brzine preuzimanja i stvaranja podataka.

Prva aplikacija bit će napravljena s danas popularnim ReactJS aplikacijskim okvirom s korisničke strane i ExpressJs okvirom s poslužiteljeve strane. Druga aplikacija bit će napravljena s rastućim u popularnosti Svelte aplikacijskim okvirom s korisničke strane i NestJs okvirom s poslužiteljeve strane. Baza podataka bit će napravljena u PostgreSQL-u te će se na nju nakratko osvrnuti kako bi se razumjela struktura podataka kojima će aplikacije manipulirati. Slučaj korištenja za koji će biti napravljene aplikacije je zaprimanje i obrađivanje narudžbi prerađivanja maslina u uljari. Također će ovaj slučaj korištenja biti objašnjen uz pomoć dijagrama.

2. Osnove *web*-aplikacija

Kada bi se *web*-aplikacije i stranice rastavile na glavne dijelove, skoro sve bi se sastojale od tri dijela, a to su: HTML, CSS i JavaScript. HTML bi stvarao strukturu *web*-stranica, tj. on bi prikazivao naslove, gumbe i slično. CSS bi prikazivao dizajn te strukture odnosno on bi dao plavu boju jednom gumbu dok drugom možda zelenu, neki odjeljak bio bi napisan u jednom fontu dok drugi u nekom drugom. JavaScript je dodao funkcionalnosti *web*-aplikacijama i stranicama npr. uz pomoć njega korisnik bi mogao kliknuti na neki gumb i onda se registrirati na neku stranicu.

Kako su potražnja i kompleksnost zahtjeva korisnika rasle, programerima je sve teže bilo (s HTML-om, CSS-om i JavaScriptom) stvarati moderne, interaktivne i lijepo dizajnirane *web*-aplikacije. Time kako su se oni razvijali također su se razvijali i njihovi aplikacijski okviri koji su olakšali programerima praćenje zahtjeva korisnika. Zbog toga se danas uglavnom koriste aplikacijski okviri za rad *web*-aplikacija, ali ako neka osoba želi postati programer *web*-aplikacija, trebala bi imati dobro poznavanje funkcionalnosti (što koja komanda radi) i sintakse (kako se piše) HTML-a, CSS-a i JavaScripta te kako oni međusobno komuniciraju prije negoli krene učiti njihove aplikacijske okvire.

Zbog toga ti programski jezici sami ne bi mogli stvarati moderne, interaktivne i lijepo dizajnirane *web*-aplikacije kao danas te je iznimno bitno da svaka osoba, ako želi postati programer *web*-aplikacija, dobro poznaje sva tri navedena programska jezika, tj. trebalo bi imati dobro poznavanje funkcionalnosti (što koja komanda radi) i sintakse (kako se piše) HTML-a, CSS-a i JavaScripta te kako oni međusobno komuniciraju.

2.1. HTML i CSS

2.1.1. Uvod u HTML

HTML je prezentacijski jezik koji se koristi za kreiranje *web*-aplikacija i *web*-stranica. HTML stoji za Hypertext Markup Language što bi u prijevodu značilo „hipertekstni jezik oznaka“. Razlog zašto nije programski jezik je to što on ne sadrži funkcije logike kao npr. u Javascriptu „a = 3 + 5“ nego samo prikazuje, tj. prezentira elemente na stranicu. On se s

vremenom relativno malo mijenjao u usporedbi s JavaScriptom te je još uvijek jedan od najbitnijih dijelova stvaranja *web*-aplikacija. Karakteriziran je njegovom sintaksom koja u sebi sadrži znakove „<" (više od) i „>" (manje od) npr. „<element>" i „</element>". Prvi znak „<element>" označuje „otvaranje" tog elementa, dok drugi znak „</element>" označuje „zatvaranje" tog elementa te štogod je napisano ili dodano (npr. neki drugi element) između ta dva elementa smatrat će se dijelom tog elementa. Sljedeći primjer (Primjer 1.) prikazuje vizualno kako izgleda jednostavan kod HTML-a.

```
<body>
      <h1>Ovaj tekst se ispisuje na ekran</h1>
</body>
```

Primjer 1. Jednostavan HTML kod

Dakle, u Primjeru 1. dvije su oznake elementa „<body>" koji, kao što njegovo ime govori, prikazuje tijelo *web*-aplikacije. Tekst koji se nalazi između tih oznaka bit će ispisan na ekran. Postoje mnoge oznake u HTML-u kao što su „<h1>" za naslov prve razine, „<header>" za zaglavlje i „<footer>" za podnožje stranice. Navedene oznake su samo neke od raznih oznaka koje HTML nudi. Kako bi se bolje razumjela sintaksa i izgled koda HTML-a, sljedeći primjer (Primjer 2.) bit će malo kompliciraniji te biti će detaljno objašnjen.

```
<!DOCTYPE html>
<html>
<body>

            <h1> Naslov </h1>
            <p> Odjeljak s tekstom </p>
            <button> Klikni me! </button>

      </body>
</html>
```

Primjer 2. Malo kompliciraniji HTML kod

Ovo je dobar jednostavan prikaz (Primjer 2.) kako kod HTML-a izgleda u praksi. Sve je poredano jedno iznad drugoga kako bi se znao redoslijed ispisa elemenata na ekran. To znači da će se u ovome primjeru prvo na ekran ispisati „Naslov", zatim „Odjeljak s tekstom" te onda na kraju bit će gumb koji u sebi sadržava tekst „Klikni me!". Također, primjećuje se da kôd izgleda kao sendvič. U ovome slučaju oznaka „<body>" bila bi kruh, „<h1>" bila bi rajčica, „<p>" bila bi salata, a „<button>" salama. Iako ovaj prikaz zvuči malo duhovito, on može pomoći u razumijevanju HTML-a novim programerima kako bi mogli lakše koncipirati

kod koji žele stvoriti ili u slučaju kada je jako puno linija koda programer može uz pomoć tog gledišta lakše vidjeti koji se element nalazi unutar kojeg drugog elementa.

Kako bi se lakše selektirali elementi u drugim jezicima, elementima HTML-a pridružuju se atributi kao npr.

```
<h1 class="naslov" id="glavni-naslov"> Naslov </h1>
```

Primjer 3. Atributi u HTML-u

U Primjeru 3. element „<h1>“ sadržava atribut „class“ s vrijednosti „naslov“ što znači da pripada klasi „naslov“. Također, taj element sadržava jednoznačni identifikator „id“ s vrijednosti „glavni-naslov“ što znači da kada se u nekom drugom programskom jeziku selektira sve što ima vrijednost „glavni-naslov“, samo taj element će biti selektiran.

2.1.2. Uvod u CSS

Kratica CSS stoji za Cascading Style Sheets što bi u doslovnom prijevodu značilo „kaskadni listovi stila“. Kao i HTML CSS nije programski jezik nego je stilistički jezik. On samo uljepšava elemente HTML-a kako bi *web*-aplikacija ili stranica bila ljepša i pristupačnija. Programer uz pomoć CSS-a može kontrolirati boju teksta, fontove koji se žele koristiti, pozadinske boje ili slike. Kako je u HTML-u sve predstavljeno kao jedno ispod drugoga, programer sa CSS-om također postavlja elemente lijevo ili desno po želji te se tako detaljnije mogu pozicionirati elementi u *web*-aplikaciji [1]. Kako bi CSS znao koji element treba stilirati, on u sebi sadrži selektore. Primjer selektora bilo bi „p“ što selektira sve „<p>“ elemente u HTML kodu. Kada se pridoda deklaracija „font-weight: bold;“ tom selektoru dobije se „p{ font-weight: bold; }“ što znači da će svi „<p>“ elementi u HTML-u imati podebljani tekst [1, str. 231].

CSS selektori mogu sadržavati više deklaracija kako bi se mogli raditi kompleksniji sadržaji na *web*-stranici, primjerice:

```
p {
      color: red;
      text-align: center;
}
```

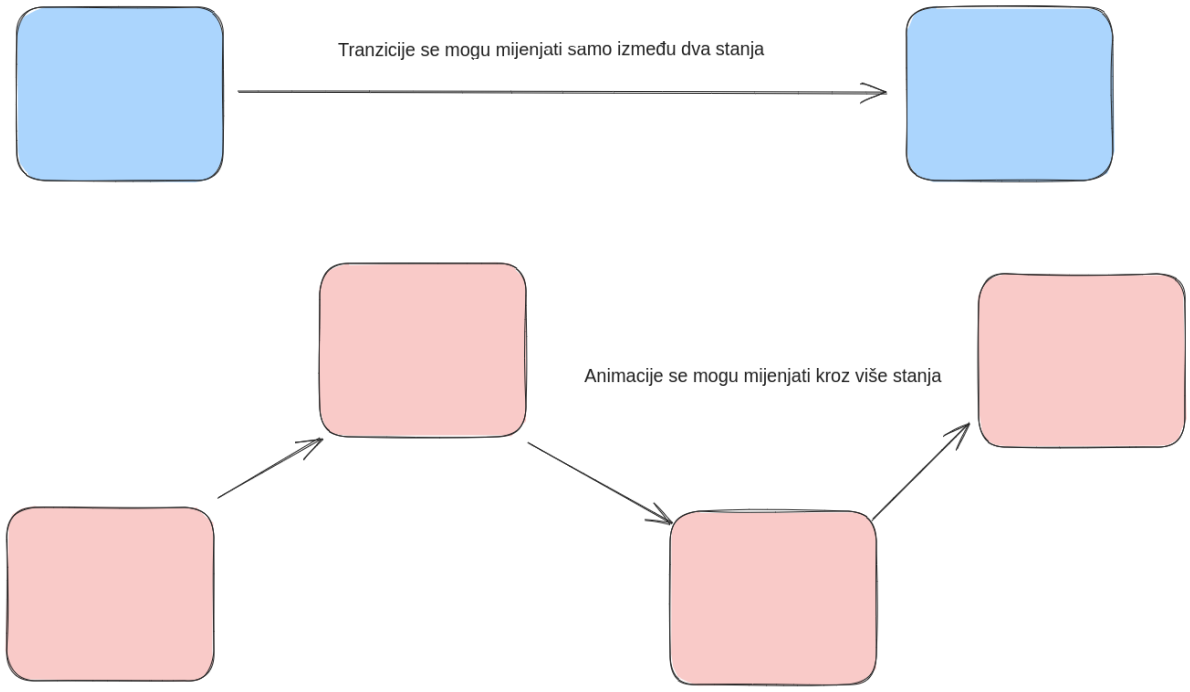
Primjer 4. Jednostavan CSS kod [autorski rad]

Primjer 4. selektira sve „<p>“ elemente u HTML kodu te im daje dvije deklaracije. Prva deklaracija postavlja boju teksta u crveno, dok druga centrira tekst u sredinu elementa relativno na njegove margine.

Na taj jednostavan način „selektiraj, pa opiši“ programeri u počecima mrežnog dizajna i programiranja mogli su raditi mnoge tadašnje moderne dizajne, ali s vremenom kako su se zahtjevi korisnika mijenjali ovakve jednostavne deklaracije nisu bile dovoljne za zadovoljiti potrebe i zahtjeve korisnika. Time su se u CSS dodale tranzicije i animacije.

2.1.2.1. CSS tranzicije i animacije

CSS prijelazi jednostavne su animacije koje se nastaju kao rezultat promjena nekih CSS svojstava. Prijelazi imaju samo dva stanja u njihovoj animaciji, a to su početno i krajnje stanje. Iako su jednostavni oni se najčešće koriste tijekom dizajniranja *web*-stranica i aplikacija. CSS animacije pružaju programeru veću kontrolu nad elementom koji se želi animirati. Animacije mogu se pokretati i samostalno te omogućuju stvaranje više ključnih okvira (eng. keyframes) preko kojih se događa animacija [2, str. 93]. Sljedeća slika (Slika 1.) je vizualni prikaz razlike između tranzicija i animacija.



Slika 1. Razlika tranzicija i animacija [2, str. 94]

Kako bi se napravile tranzicije, programer treba napraviti dvije stvari. Prvo treba odrediti u CSS-u nekim selektorom kojem elementu želi pridodati tranziciju te tom istom elementu treba također dodati neka svojstva (širina, visina, boja…) i koje će se svojstvo mijenjati u tranziciji i koliko dugo bi trajala tranzicija u sekundama ili milisekundama. U drugom koraku programer treba specificirati kada bi se trebala aktivirati tranzicija, na primjer, kada se mišem prijeđe preko elementa. Zadnji korak je postaviti novu vrijednost odabranog svojstva kako bi tranzicija znala do koje mjere bi trebala promijeniti to svojstvo.

```
div {
      width: 100px;
      height: 100px;
      background: red;
      transition: width 2s;
}
div:hover {
      width: 300px;
}
```

Primjer 5. CSS tranzicija

Kod u Primjeru 5. vizualno prikazuje prijašnje objašnjenje stvaranja CSS tranzicija. Postoji „ “ element koji ima neka svojstva (najbitnija za ovo objašnjenje su širina i tranzicija). Širina elementa je 100 piksela, a tranzicija se odnosi na širinu i traje dvije sekunde. Drugi dio ponovno se odnosi na „ “ element, ali uz sebe ima i „:hover“. Dodatni dijelovi u selektorima nakon dvotočke kao što je „:hover“ označuju u kojem trenutku će se tranzicija aktivirati što u ovom slučaju znači da kad se mišem prijeđe preko tog elementa, te kada se to desi, širina tranzicije bi se trebala promijeniti sa sto piksela na tristo piksela u točno dvije sekunde.

Kao što je prije navedeno, CSS animacije daju kompleksnije izglede i animacije od tranzicija, ali to ujedno i znači da je kôd animacija kompleksniji. Animacije u CSS-u rade se uz pomoć pravila ključnih okvira (engl. keyframes). Ključni okviri koraci su kojih animacija mora napraviti po specificiranom redoslijedu redoslijedu. Što je više koraka to je veća kontrola nad animacijom, ali je ujedno i kompleksniji kod [3]. Kada se specificiraju CSS stilovi unutar pravila ključnih okvira, animacija će se u određenim trenucima postupno mijenjati od trenutnog stila do novog stila. Kako bi animacija radila, ona se, kao i tranzicija, mora vezati za neki element. Sljedeći primjer (Primjer 6.) veže animaciju "primjer" na element „ “. Animacija će trajati 4 sekunde i postupno će mijenjati pozadinsku boju elementa „ “ iz "crveno" u "žuto":

```
@keyframes example {
      from {background-color: red;}
      to {background-color: yellow;}
}
div {
      width: 100px;
      height: 100px;
      background-color: red;
      animation-name: example;
      animation-duration: 4s;
}
```

Primjer 6. CSS animacija [autorski rad]

2.2. Uvod u JavaScript

JavaScript je stvoren 1995. u samo deset dana. Kreator JavaScripta je Brendan Eich. Eich je radio u Netscapeu i implementirao JavaScript za njihov web-preglednik, Netscape Navigator. Isprva ideja je bila da se Java koristi za stvaranje glavnih interaktivnih dijelova *web*-stranica te da JavaScript bude samo jezik ljepila za te dijelove *web*-stranica te da napravi HTML malo interaktivnijim. Zbog takve ideje JavaScript je trebao izgledati kao Java. [4, str. 19].

U JavaScriptu varijabla je lokacija prostora računala sa simboličnim imenom koja sadrži neku vrijednost. Njezina vrijednost može se mijenjati (varirati) od čega je i dobila ime „varijabla“. Funkcija u JavaScriptu je skup naredbi uz pomoć kojih se obavljaju neki zadatci kao što su računanje, stvaranje i brisanje. Funkcije imaju neki ulaz i izlaz. Na ulazu mogu se koristiti neki početni parametri dok na izlaz funkcija vraća neku vrijednost[5].

2.2.1. Osnove JavaScript varijabli

Isprva je kôd JavaScripta relativno jednostavan. Ako se želi deklarirati neka varijabla s vrijednosti 5, onda se samo napiše:

```
let a = 5;
```

Primjer 7. Jednostavan JavaScript kod

U primjeru iznad „let“ znači u doslovnom prijevodu „dopusti“. Ta riječ se koristi za deklariranje varijabli što znači da ako se samo napiše (Primjer 8.):

```
let a = 5;

a = 3;
```

Primjer 8. Mijenjanje vrijednosti varijable

vrijednost varijable „a“ će se promijeniti s 5 na 3. Kada se poziva varijabla kao u gornjem primjeru (primjer 8.)(bez riječi “let”), mijenja joj se vrijednost. Deklaracija varijabli s „let“ može se interpretirati kao „dopusti varijabli s imenom 'a' da ima vrijednost od 5''.

2.2.2. Osnove JavaScript funkcija

U JavaScriptu funkcije se deklariraju slično kao i varijable. Za deklarirati funkciju s imenom „zbroji“ napiše se (Primjer 9.):

```
function zbroji(){
}
```

Primjer 9. Jednostavna JavaScript funkcija

Definiranje funkcija u Javascriptu započinje riječju „function“. Poslije te riječi definira se ime funkcije (u ovome primjeru „zbroji“). Primjećuje se da su u Primjeru 9. dvije zagrade (jedna obla „()“ i jedna vitičasta „{}“). Obje zagrade mogu sadržavati varijable, ali ovisno u kojoj je zagradi varijabla deklarirana, imat će različitu funkciju i opseg. Varijable koje su deklarirane unutar oblih zagrada postanu parametri te funkcije dok varijable deklarirane unutar vitičastih zagrada postanu kao i prijašnje objašnjene obične varijable, ali te varijable postanu dostupne samo unutar vitičastih zagrada.

```
let varijabla = 2;
function zbroji(parametar) {

      let varijablaFunkcije = 3;

      let rezultat = parametar + varijablaFunkcije;

      return rezultat;
}

zbroji(varijabla);
```

Primjer 10. Korištenje JavaScript funkcija

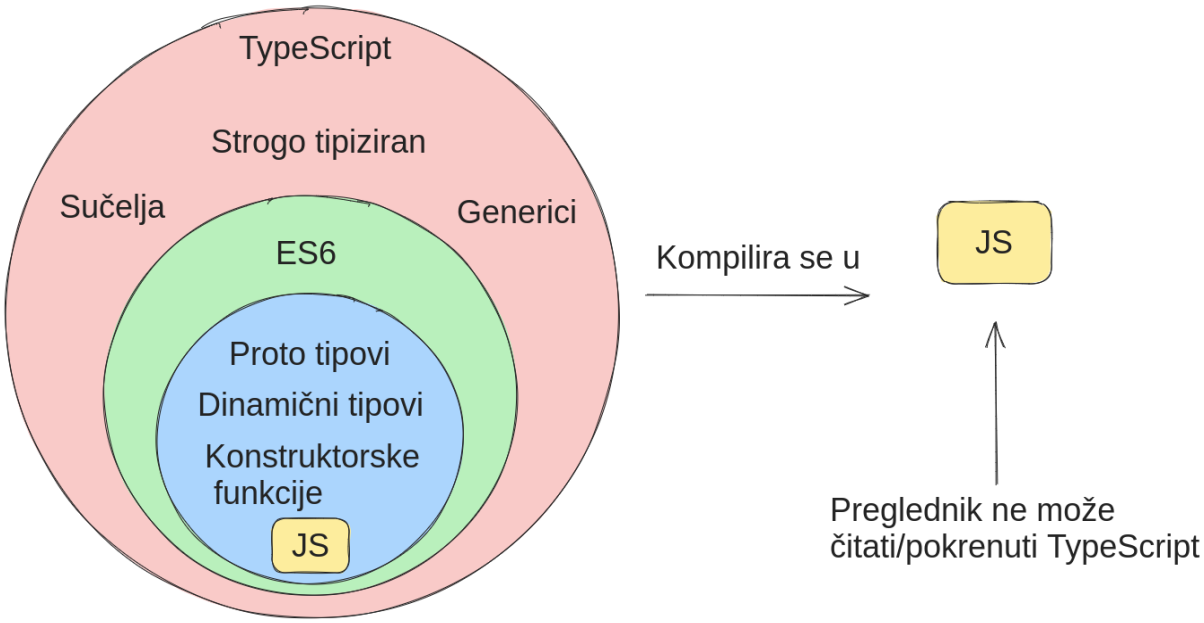
U Primjeru 10. na početku se definira varijabla s imenom „varijabla“ s vrijednosti „2“. Nakon toga deklarira se funkcija imenom „zbroji“ koja prima neki parametar s imenom „parametar“. Unutar funkcije deklarirana je varijabla „varijablaFunkcije“ s vrijednosti „3“. Također, definirana je zadnja varijabla „rezultat“ koja prima vrijednosti od „parametar“ i „varijablaFunkcije“ te ih zbraja i taj rezultat spremi u sebe. Riječ „return“ u funkciji definira izlaz funkcije te što će biti vraćeno na izlazu funkcije (u ovom slučaju prosljeđuje se „rezultat“). Kako bi se funkcija iskoristila, mora se negdje pozvati. Funkcije se pozivaju (kao u gornjem primjeru) tako da se napiše ime funkcije (s oblim zagradama nakon imena) te unutar oblih zagrada specificiraju koji će biti ulazni parametri (ako ih ima, u ovom slučaju varijabla s imenom „varijabla“). Krajnji rezultat ove funkcije trebao bi biti broj 5 zbog toga što funkcija prima parametar „varijabla“ s vrijednosti 2 i zbraja ju s varijablom „varijablaFunkcije“ čija je vrijednost 3. Vrijednost sprema varijabla „rezultat“ i nju funkcija kroz izlaz prosljeđuje van.

2.3. TypeScript

TypeScript je programski jezik kreiran od Microsofta. TypeScript je tipizirani skup JavaScripta koji se kompilira u običan JavaScript. To čini programe napisane u TypeScriptu visoko prenosivima jer se mogu izvoditi na gotovo svakom računalu. Dakle, TypeScript je nadogradnja JavaScripta [6, str. xix].

Jedan od razloga zašto je TypeScript izumljen jest što je s JavaScriptom, iako moguće, teško stvarati velike i skalabilne projekte jer je jezik dinamično tipiziran. Dinamično tipizirani jezici su oni koji tijekom programiranja ne specificiraju tipove podataka npr. u JavaScriptu za deklarirati varijablu znakovnog tipa s vrijednosti “ime” samo se napiše “ let imeVarijable = “ime” ” dok se u statično tipiziranim jezicima treba specificirati kojeg će tipa biti varijabla (broj, znakovi, datum, vrijeme…). S obzirom na to da je TypeScript proširenje JavaScripta, prijašnji primjer može se identično napraviti i u TypeScriptu, ali generalno gledajući, gornji primjer trebao bi izgledati ovako “ let imeVarijable: string = “ime” “. Zbog ovakvog definiranja tipa varijable ako se kasnije želi promijeniti vrijednost iz “string” u “number” (npr. ImeVarijable = 3), to neće biti dozvoljeno te će nastati greška. Zbog takve prirode dinamično tipiziranih jezika ukoliko je neka greška napravljena teže je primijetiti gdje ta greška nastaje jer se ne zna kojeg su tipa varijable dok u statično tipiziranim jezicima eksplicitno je navedeno kojeg su tipa sve varijable. Na poslužiteljevoj strani iznimno je bitno znati kojeg su tipa varijable te zbog toga se sve više aplikacijskih okvira (kao NestJS) bazira na TypeScriptu. Na poslužiteljevoj strani bitno je znati tip svake varijable zbog toga što

poslužiteljeva strana direktno komunicira s bazom podataka u kojoj svaki stupac ima svoj određeni tip podatka. Uz tipove podatka TypeScript nudi sučelja i klase koji mogu apstrahirati funkcije i atribute kako se ne bi ponavljaje iste naredbe za svaki put kada su potrebne.



Slika 2. TypeScript je nadogradnja JavaScripta [https://www.linkedin.com/pulse/typescript-superset-javascript-sunil-sharma]

Uz prijašnji primjer može se primijetiti način deklariranja varijabli u TypeScriptu.

```
let/const/var ime varijable : tip podatka = vrijednost;
```

Primjer 11. Sintaksa deklariranja varijabli u TypeScriptu [autorski rad]

Funkcije u TypeScriptu deklariraju se na sličan način kao i varijable.

```
function ime funckije (parametri) : tip povratnog podatka {logika};
```

Primjer 12. Sintaksa deklariranja funkcija u TypeScriptu [autorski rad]

3. Okviri korisničke strane

3.1. Uvod u okvire JavaScripta

Okviri modeliraju određeno područje te imaju apstraktni dizajn koda domene koji se sastoji od sučelja ili apstraktnih klasa. Apstraktni dizajn stvoren je na taj način kako bi olakšao pregled i stvaranje koda. On definira interakciju između klasa na vrijeme izvođenja aplikacije.

Okvir dolazi s apstraktnim implementacijama koje se mogu koristiti više puta. Te implementacije su zapravo apstraktne klase koje okvirno prikazuju što kod radi, ali zbog apstrakcije sakrivaju ključne radnje koje se dešavaju tijekom vremena izvođenja programa[7, str. 8].

Potpuno je moguće izgraditi snažne *web*-aplikacije bez JavaScript okvira, ali okviri pružaju predložak koji obrađuje zajedničke obrasce programiranja. Svaki put kada se mora izraditi aplikaciju, ne mora se pisati kôd za svaku pojedinu značajku ispočetka. Umjesto toga, može se graditi na postojećem skupu značajki. JavaScript okviri, kao i većina drugih okvira, pružaju neka pravila i smjernice. Koristeći ova pravila i smjernice, svaki programer može složene aplikacije učiniti brže i učinkovitije nego ako bi se odlučio graditi ispočetka. Pravila i smjernice pomažu u oblikovanju i organiziranju *web*-stranice ili *web*-aplikacije. U ovome dijelu fokusirat će se na aplikacijske okvire korisničke strane posebno na ReactJS i Svelte koji su samo jedni od mnogo JavaScript okvira koji se koriste danas.

Grafičko korisničko sučelje *web*-stranice ili aplikacije naziva se korisnička (prednja) strana u razvoju *web*-aplikacija i stranica. Drugim riječima, to je dio aplikacije s kojeg korisnici mogu vidjeti i sudjelovati s njime dok korisnički aplikacijski okviri su programski proizvod/alat/platforma koji služi kao temelj za napredovanje komponenata *web*-rješenja s korisničke strane [10].

3.2. ReactJS

Kada je projekt React 2013. godine bio predstavljen, programeri i korisnici bili su skeptični prema njegovim funkcionalnostima jer su za to vrijeme bile poprilično unikatne, ali s vremenom popularnost mu je drastično brzo narasla te je postao najpopularniji aplikacijski okvir za stvaranje *web*-aplikacija na svijetu.

Tom poveznicom popularizirala se metodologija programiranja nazvana „razvoj vođen komponentama“ (engl. Component Driven Development, CDD). Komponenta u ovome

kontekstu može biti gumb, zaglavlje, podnožje, navigacija, tablica itd. Unutar *web*-aplikacija mnogo je jednakih komponenata tako da ovaj način programiranja može uštedjeti mnogo vremena u stvaranju *web*-aplikacije, ali i u održavanju [8]. Ako dođe zahtjev da se neka komponenta treba promijeniti, treba samo naći gdje je komponenta napravljena, promijeniti štogod je potrebno (npr. promijeniti boju iz plave u zelenu) i time gdjegod se ta komponenta pojavljuje automatski u svim tim mjestima ona je promijenjena. Ovo je samo jedan od mnogo primjera zbog kojih je React popularan danas.

3.2.1. Virtualni DOM

„Model dokumenta (DOM) programsko je sučelje za mrežne (engl. *web*) dokumente. Predstavlja stranicu tako da programi mogu mijenjati strukturu, stil i sadržaj dokumenta. DOM predstavlja dokument kao čvorove i objekte; na taj način programski jezici mogu komunicirati sa stranicom“ [9].

Virtualni DOM (VDOM) programski je koncept koji sadrži idealno predstavljanje korisničkog sučelja te čuva se u memoriji i sinkronizira s DOM-om od strane knjižnice. Taj se postupak naziva usklađivanje (engl. reconciliation). Programer jednostavno može samo specificirati stanje željenog korisničkog sučelja i React će osigurati da DOM reflektira to željeno stanje. React na ovaj način apstrahira mnogo koraka koje bi programer trebao napraviti kako bi stvorio željeno stanje u *web*-aplikaciji. U svijetu Reacta, pojam „virtualni DOM” obično je povezan s React elementima jer su oni objekti koji predstavljaju korisničko sučelje [9].

Budući da je „virtualni DOM” više uzorak nego specifična tehnologija, ljudi ponekad kažu da to znači različite stvari. U svijetu Reacta, pojam „virtualni DOM” obično je povezan s React elementima jer su oni objekti koji predstavljaju korisničko sučelje. React, međutim, koristi i unutarnje objekte zvane „vlakna” (eng. fibers) kako bi se zadržale dodatne informacije o stablu komponente. Oni se također mogu smatrati dijelom virtualne DOM implementacije u Reactu [14].

3.2.2. JSX u Reactu

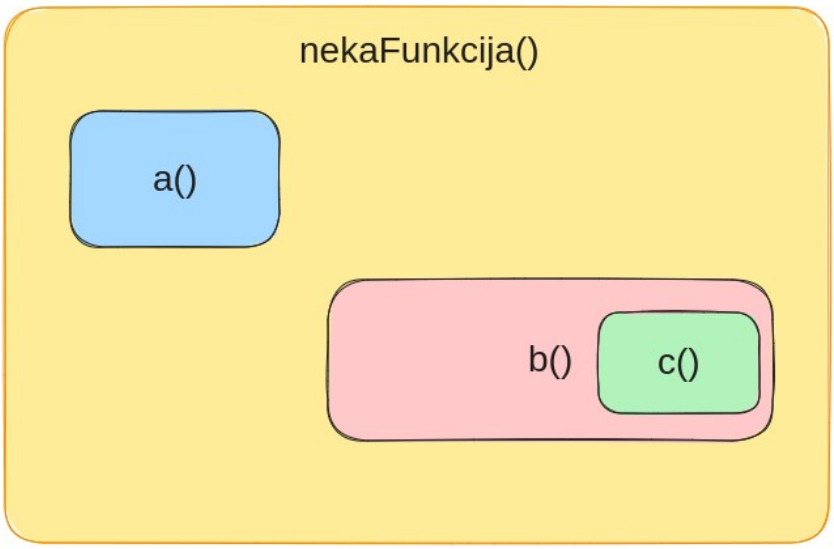
JSX je sintaksno proširenje JavaScripta koje omogućuje pisanje koda sličnog HTML-u unutar JavaScript datoteke. Kao što je navedeno, inače su logika JavaScripta i struktura HTML-a bili odvojeni i držali se u različitim datotekama, a kako je web postao interaktivniji, logika je sve više determinirala sadržaj *web*-aplikacija i stranica. JavaScript je postao vladar HTML-u. Zbog toga u Reactu logika i struktura žive u istom mjestu – komponenti [10] .

```
function App() {
      return (

            <div className="App">
            <h1>Ovo je JSX u React-u</h1>
            </div>
            );
      }
export deault App;
```

Primjer 13. Jednostavan React kod

Primjer 13. je jednostavan JSX kod u Reactu. U JSX-u najbitniji dio koda je funkcija. U Primjeru 13. to bi bila funkcija “function App()” koja s riječi “return” vraća kôd koji liči na HTML. U ovomu slučaju to bi bio naslov prve razine s tekstom „Ovo je JSX u Reactu”.

3.2.3. Kuke (engl. *hooks*)



Slika 5. Funkcija koja koristi funkcije [11]

U programskom jeziku JavaScript funkcije se mogu koristiti kodnom logikom za obavljanje ponavljajućih zadataka. Funkcije su sastavljive. To znači da se može pozvati funkciju u drugoj funkciji i koristiti njen izlaz. U Slici 2. funkcija „nekaFunkcija()“ koristi funkciju „a()“ i „b()“ dok funkcija „b()“ koristi funkciju „c()“.

Isti način vrijedi i za komponente u Reactu. Dakle, i komponente mogu sadržavati/koristiti druge komponente. U Reactu komponente mogu biti bez stanja i sa stanjem. Komponente sa stanjem deklariraju i manipuliraju lokalno stanje u sebi dok komponente bez stanja su zapravo funkcije koje manipuliraju nuspojave. To znači da te funkcije bez stanja uvijek daju jednak izlaz za jednak ulaz. Nuspojave su promjene stanja unutar komponente koje mogu biti očekivane, ali i neočekivane što u teorij nije poželjno za programiranje u Reactu.

Uz pomoći React kuka (engl. hooks) stanje i nuspojave izoliraju se od funkcionalnih komponenata. Ako više komponenata treba koristiti istu kuku, onda ne treba ponovno za svaku kuku pisati kôd, nego se kuka samo pozove unutar tih komponenata [11].

React sadrži 10 predefiniranih kuka. Također programer može stvoriti i svoje kuke. Jedna od tih 10 kuka je kuka “useState”. Ona služi za promatranje, pamćenje i mijenjanje stanja (vrijednosti) neke varijable u kodu. Koristi se na način da se definiraju dvije konstante od kojih jedna sadrži početno (ili trenutno) stanje (tj. vrijednost), dok je druga funkcija uz pomoć koje će se u drugim funkcijama definirati novo stanje varijable. Tim dvjema konstantama bit će pridruženo početno stanje tako da se nakon znaka “=” napiše “useState(‘početna vrijednost’)”. Kako bi se mijenjalo stanje, stvara se funkcija u kojoj se poziva prijašnje definirana funkcija i u nju se stavlja novo stanje. Stanje može se prikazati na ekran preko JSX-a te će svaki puta kada se promijeni stanje odmah biti ažurirano.

```
import React, { useState } from 'react';

function Example() {
      const [count, setCount] = useState(0);
      return (
            <div>
            <p>Vrijednost varijable “count”: {count}</p>
            <button onClick={() => setCount(count + 1)}>
            Klikni me
            </button>
            </div>
```

```
        );
}
```

Primjer 14. Korištenje „useState()“ kuke

Primjer 14. pokazuje korištenje “useState()” kuke. Definira se konstanta “count” koja ima početno stanje od 0 te konstanta “setCount” koja se u “button” elementu koristi za inkrementiranje vrijednosti varijable “count” za 1.

3.2.4. Rekviziti (engl. *props*)

U Reactu često se trebaju koristiti funkcije ili varijable drugih komponenata. Kako bi se pojednostavio odnos između komponenata, stvoreni su rekviziti (engl. *props*). Rekviziti su argumenti koji se prosljeđuju funkciji ili klasi, ali budući da su komponente pretvorene u objekte nalik HTML-u s JSX-om, rekviziti se koriste kao da su HTML atributi. Za razliku od HTML elemenata, uz pomoć rekvizita može se prenositi više različitih tipova podataka kao što su cijeli brojevi, znakovi, polja, objekti i funkcije [12, str. 147].

```
import React from 'react'

export default function Naslov({naslov}) {
        return (
        <div>{naslov}</div>
        )
}
```

Primjer 15. Kod datoteke Naslov.js

```
import Naslov from './components/Naslov';

function App() {

        return (
                <div className="App">
                        <div className='naslov'>
                                <Naslov naslov='Narudzba'/>
                        </div>
                </div>
        );
}

export default App;
```

Primjer 16. Kod datoteke App.js

U Primjeru 15. stvorena je datoteka Naslov.js u kojoj je komponenta Naslov. Ta komponenta prima kao parametar “naslov” što može biti bilo kojeg tipa. Njegova vrijednost ispisuje se kao naslov prve razine. U komponenti App instancira se komponenta Naslov u obliku HTML elementa “<Naslov/>” te se u nju stavlja vrijednost “Narudzba” u atribut “naslov”. Atribut mora imati naziv jednak parametru koji prima komponenta u njenom kodu.

3.3. Svelte

Svelte je malo mlađi aplikacijski okvir korisničke strane. Kreirao ga je Rich Harris 2016. godine te je unutar zadnjih par godina postao jedan od najpopularnijih aplikacijskih okvira od strane programera.

Svelte također kao i React promovira metodologiju *razvoja vođenog komponentama*, ali on ga koristi na sličan, ali ne identičan način. U usporedbi s Reactom, Vueom, Angularom i drugim okvirima, unaprijed se sastavlja aplikacija izgrađena pomoću Sveltea tako da server ne mora služiti cijeli okvir svakom posjetitelju *web*-stranice. Kao rezultat toga sve se osjeća brže i laganije. Kod implementacije, Svelte nestaje i sve što se dobije je JavaScript [13, str. 4].

3.3.1. Svelte vrsta komponenata

Svelte datoteke uglavnom sadrže tri glavna elementa: „<script>“, „<main>“ i „<style>“. Arhitektura bi otprilike izgledala ovako:

```
<script></script>

<main></main>

<style></style>
```

Primjer 17. Svelte arhitektura datoteke

Unutar elementa „<script>“ (Primjer 17.) nalazi se JavaScript logika, unutar „<main>“ HTML kod dok u „<style>“ CSS kod. Ovo je malo različito od Reacta koji u svoje varijable stavlja HTML, ali dodatno je što Svelte ima i CSS dodan u komponenti. Ovo čini Svelte mnogo jednostavnijim za koristiti i razumjeti.

3.3.2. Korištenje komponenata

Kroz sljedeći primjer (Primjer 18.) analizirat će se korištenje komponenata u Svelteu.

```
<div>

	<img src="../multimedija/pas.jpg" alt="Slika psa">

</div>
```

Primjer 18. Komponenta Dog

```
<script>

	import Dog from './Dog.svelte'

</script>
```

Primjer 19. Svelte dodavanje komponenata

U skripti uz pomoć JavaScripta poziva se datoteka „Dog.svelte“. Ta datoteka sprema se kao komponenta pod imenom „Dog“. Tu komponentu može se referencirati nadalje u skripti (ako je potrebno), HTML-u za prikaz te komponente ili u stilu za pozicioniranje ili uljepšavanje komponente. Referenciranje „Dog“ komponente u HTML-u bi izgledalo ovako:

```
<script>
	import Dog from "./Dog.svelte";
</script>

<main>
	<Dog />
</main>
```

Primjer 20. Svelte korištenje komponenata

Dakle, po ovome primjeru (Primjer 20.) vidi se da se komponente u Svelteu mogu referencirati kao da su HTML elementi. Također, vidi se da je komponenta različito napisana od standardnih HTML elemenata u formatu „<ime />“. Ovakav znak „< />“ u HTML-u zove se samozatvarajuća oznaka (engl. *self-closing tag*). Svelte komponente mogu se standardno prikazivati, ali u većini slučajeva to nije potrebno zbog toga što su sva logika, stil i struktura u toj komponenti već napravljeni te nije potrebno dodavati još nešto. Time kôd postaje pregledniji.

3.3.3. Manipuliranje stanjem

Stanje je svaki podatak koji je potreban da bi komponenta prikazala/radila ono što treba raditi/prikazati. Dakle, već samo vrijednost neke varijable može se smatrati stanjem. Primjer 17. vizualno prikazuje definiciju stanja. Stanje deklarirane varijable „count“ je „0“. Stanje se uvijek definira u „<script>“ dijelu datoteke [13].

```
<script>

	let count = 0

</script>
```

Primjer 21. Svelte logika [autorski rad]

Kako bi se kvalitetno manipuliralo stanjem varijable u Primjer 21., trebala bi se napraviti neka funkcija koja mijenja vrijednost varijable. Za primjer funkcija će inkrementirati vrijednost za 1. Također, negdje bi se ta funkcija trebala pozvati kako bi se funkcija iskoristila. U Svelteu funkcije se mogu pozvati direktno u elementu HTML koda i to specificirano u kojem slučaju. Sljedeći primjer (Primjer 22.) prikazuje manipuliranje stanjem s navedenim funkcionalnostima:

```
<script>
	let count = 0
	const incrementCount = () => {
		count++
	}
</script>
<button on:click="{incrementCount}"> {count} </button>
```

Primjer 22. Svelte manipuliranje stanjem [13]

Funkcija „incrementCount“ uzima varijablu „count“ i mijenja joj vrijednost za „+1“. Kreiran je gumb koji poziva „incrementCount“ u trenutku „on:Click“, tj. u trenutku kada korisnik klikne na gumb. Između znakova „<button>“ stavljeno je „{count}“. Time u Svelteu varijabla i njezina vrijednost mogu se staviti direktno u elemente HTML-a [18]. Na ovaj jednostavan način može se manipulirati stanjem u Svelteu.

3.3.4. Rekviziti (engl. *props*)

S obzirom na to da se Svelte također bazira na *razvoju vođenom komponentama* također, kao i React, koristi se rekvizitima. Svelte način korištenja rekvizita skoro je identičan Reactovom načinu. Jedina je razlika što u Svelteu nisu potrebne kuke i kompleksne funkcije za manipuliranje stanjem varijabli. Time je rekvizite relativno lagano koristiti, čitati i razumjeti. Sljedeći primjer (Primjer 23.) uz pomoć dviju komponenata vizualno će prikazati korištenje rekvizita u Svelteu.

```
<script>
	import Answer from './Answer.svelte';
</script>

<Answer answer={42} />
```

Primjer 23. Korištenje rekvizita

```
<script>
	let answer;
</script>

<p>The answer is {answer}</p>
```

Primjer 24. Komponenta „Answer“

U Svelteu, kao i u Reactu, potrebno je prvo napraviti komponentu koju se želi koristiti za rekvizit. Nakon toga u glavnoj komponenti instancira se rekvizit i preko HTML elementa rekvizita pridodaje se vrijednost preko rekvizita.

3.3.5. Svelte spremišta (engl. *stores*)

Ovo poglavlje usredotočuje se na korištenje spremišta za dijeljenje podataka između komponenti, bez obzira na njihov odnos u hijerarhiji komponenti. Spremišta pružaju alternativu korištenju rekvizita ili konteksta. Stanje aplikacije nalazi se izvan bilo koje komponente. Svako spremište sadrži jednu vrijednost, ali ta vrijednost može biti polje ili objekt, koji naravno može sadržavati mnogo vrijednosti.

Postoj više vrsta spremišta: spremišta za zapisivanje, spremišta za čitanje, izvedena spremišta i prilagođena spremišta. Spremišta za zapisivanje dozvoljavaju komponentama da mijenjaju (zapisuju) svoje podatke dok spremišta za čitanje ne dozvoljavaju ikakve promjene.

Izvedena spremišta izračunavaju svoju vrijednost iz drugih spremišta, a prilagođena spremišta mogu raditi sve od prijašnje navedenih spremišta.

Sljedeći primjer (Primjer 25.) služi za pojašnjenje Svelte spremišta. Prvo je kreirana datoteka “stores.js” koja sadrži zapisujuću vrstu spremišta. Spremište se stvara uz pomoć funkcije “writable()” te u nju je prvo stavljena vrijednost 0.

```
<script>
	import { writable } from 'svelte/store';

	export const count = writable(0);
</script>
```

Primjer 25. Datoteka spremišta [13]

Uz pomoć ovog spremišta zadatak je napraviti jednostavan program koji može inkrementirati, dekrementirati i resetirati vrijednost ovoga spremišta. Kako bi to bilo moguće, trebat će se napraviti 3 datoteke s funkcijama i jednu datoteku koja će ukomponirati sve prijašnje datoteke i manipulirati stanjem. Prva od 3 datoteke bit će datoteka “Decrement.svelte” (Primjer 26.) koja će sadržavati funkciju za dekrementiranje vrijednosti varijable “count” iz datoteke “stores.js”. Kako bi korisnik mogao imati interakciju s tom funkcijom, također bit će kreiran gumb koji u trenutku klika ažurira vrijednost varijable “count”.

```
<script>
	import { count } from './stores.js';

	function decrement() {
		count.update((n) => n - 1);
	}
</script>

<button on:click={decrement}> - </button>
```

Primjer 26. Komponenta „Decrement“ [13]

Sljedeća datoteka je datoteka “Increment.svelte” (Primjer 27.) koja će sadržavati jednaku funkciju datoteke “Decrement.svelte”, ali umjesto smanjivanja vrijednosti ova funkcija će povećavati vrijednost.

```
<script>
```

```
	import { count } from './stores.js';

	function increment() {
		count.update((n) => n + 1);
	}
</script>

<button on:click={increment}> + </button>
```

Primjer 27. Komponenta „Increment“ [13]

Zadnja od 3 datoteke će biti datoteka “Reset.svelte” (Primjer 28.) koja će, kao što ime govori, resetirati vrijednost varijable “count” na 0.

```
<script>
	import { count } from './stores.js';

	function reset() {
		count.set(0);
	}
</script>

<button on:click={reset}> reset </button>
```

Primjer 28. Komponenta „Reset“ [13]

Kako bi se sve to spojilo, napravit će se posljednja datoteka “App.svelte”. Ona će sadržavati svoju varijablu “countValue”. Kako bi datoteka “App.svelte” mogla koristiti spremište za sebe, potrebno je uz pomoć funkcije “subscribe()” pretplatiti se na varijablu “count” iz datoteke “stores.js”. U funkciji pretplate specificira se da vrijednost koju to spremište ime treba dodati varijabli “countValue”. Nakon toga “countValue” se koristi za ispis na ekran.

```
<script>
import { count } from './stores.js';
import Incrementer from './Incrementer.svelte';
import Decrementer from './Decrementer.svelte';
import Resetter from './Resetter.svelte';

let countValue;

count.subscribe((value) => {
	countValue = value;
	});
</script>
```

```
<h1>The count is {countValue}</h1>

<Incrementer />
<Decrementer />
<Resetter />
```

Primjer 29. Glavna komponenta [13]

3.3.6. SvelteKit

SvelteKit je Svelteova nadogradnja. On je aplikacijski okvir za aplikacijski okvir što se ukratko kaže metaokvir (engl. *meta-framework*). On koristi sve što i Svelte, ali uz još neke dodane funkcionalnosti primjerice: usmjeravanje, kuke (engl. *hooks*), krajnje točke, radnje obrazaca itd.

Iako Svelte može raditi *web*-aplikacije veoma kvalitetno, inače se preporučuje SvelteKit za rad kompliciranih *web*-aplikacija upravo zbog tih dodanih funkcionalnosti koje programerima olakšavaju programiranje sve kompleksnijih *web*-aplikacija.

SvelteKit zamagljuje liniju između okvira korisničke i pozadinske strane i čini integraciju među njima besprijekornom jer ima kontrolu nad oba, što znači da programer može početi raditi na svom projektu i ne patiti od umora odluka da koji je pristup bolji od nekog drugog. Može koristiti za stvaranje aplikacijskih sučelja za programiranje baš kao što može s Expressom koji će se analizirati u kasnijim poglavljima.

4. TailwindCSS

TailwindCSS je CSS okvir za brzu izgradnju prilagođenih dizajna aplikacija i može olakšati ispravljanje pogrešaka. Za razliku od CSS-a Tailwind se zapisuje direktno u element HTML-a. Time nisu potrebne dodatne datoteke za stvaranje CSS koda, iako Tailwind nudi i taj način korištenja [14, ix-xii].

4.1. Razlika od drugih CSS okvira

Iako Tailwind ima sličnu sintaksu s drugim CSS okvirima, kao što je Bootstrap,

```
<div class="jumbotron text-center">
      <h2>My First Bootstrap Page</h2>
      <p>Some Bootstrap buttons:</p>
      <button class="btn btn-primary">Primary</button>
      <button class="btn btn-success">Success</button>
</div>
```

Primjer 30. Bootstrap kod

njegove naredbe su drugačije od njegovih kompetitora. U Bootstrapu koriste se klase čija imena opisuju kako se trebaju ponašati npr. “button” za gumb i “nav” za navigaciju. U Tailwindu svaki stil dobiva svoju klasu.

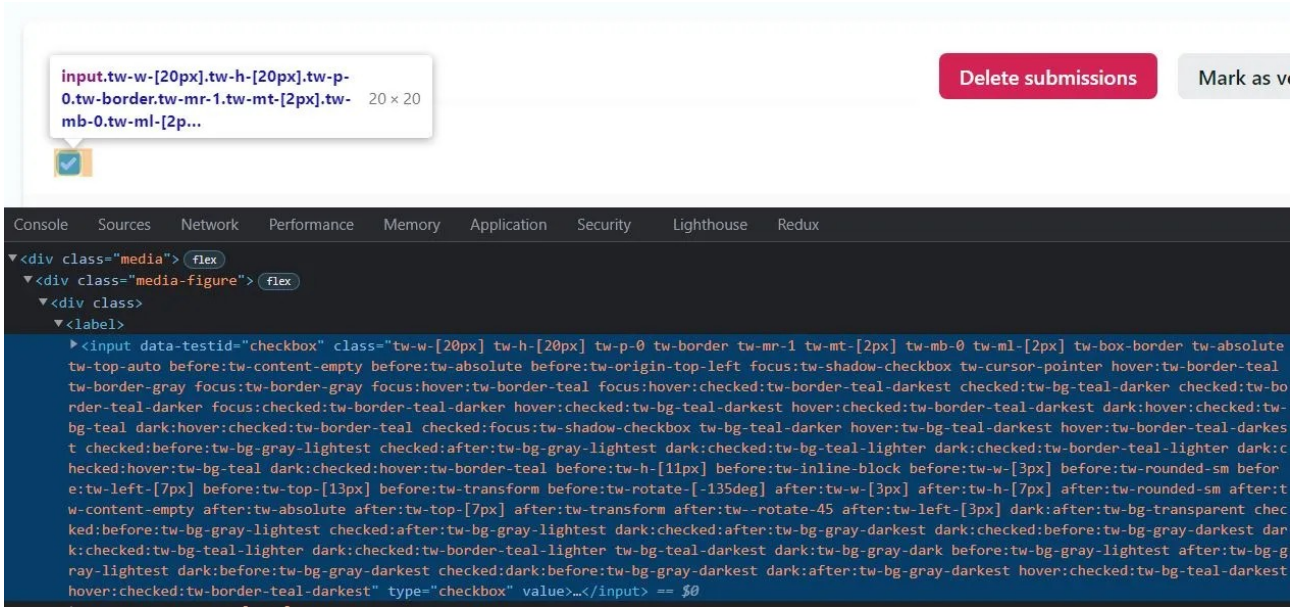
```
<div class="bg-yellow-500"></div>
```

Primjer 31. TailwindCSS kod

Primjer iznad (Primjer 31.) ima klasu koja označuje da je boja pozadine zelena te da joj je snaga boja 500 od mogućih 1000.

Ovaj način stvaranja CSS koda daje Tailwindu mogućnost da istovremeno bude jednostavniji od CSS-a, ali i dovoljno kompleksniji od ostalih CSS okvira. Time programeri mogu stvarati više prilagođene dizajne za razliku od ostalih CSS okvira.

Kako projekt postaje sve kompleksniji, time postaje i Tailwindov kod.



Slika 6. Kompleksan Tailwind kod
(https://www.aleksandrhovhannisyan.com/blog/why-i-dont-like-tailwind-css/)

Zbog toga Tailwind nudi mogućnost stvaranja prilagođenih klasa koje sadrže određene funkcije koje programer definira po potrebi. Uz pomoć tih klasa kôd izgleda pregledniji i jednostavniji za čitati i razumjeti.

```
@tailwind base

@tailwind components;

@tailwind utilities;

.listaKorisnika{
      @apply bg-white rounded overflow-hidden shadow-md;
}
```

Primjer 32. Tailwind prilagođena klasa

Stvaranje klase nije komplicirano. U datoteku style.css definira se ime klase (u ovom slučaju “listaKorisnika”) te nakon toga uz pomoć “@apply” definira se kakav bi trebao biti izgled te klase. Nakon toga klasa se koristi kao i svaka druga Tailwind klasa u HTML elementu. Iz ovoga primjera klasa bi se koristila na sljedeći način:

```
<div class=”listaKorisnika”></div>
```

Primjer 33. Korištenje Tailwind prilagođene klase

4.2. Tranzicije i animacije

Također kao i CSS Tailwind sadržava mogućnost stvaranja tranzicija i animacija. Stvaranje tranzicija relativno je jednostavno. Za napraviti tranziciju potrebno je samo u HTML elementu napisati koja je akcija koja će aktivirati tranziciju te onda što i kako će se promijeniti.

```
<div class=”bg-green-500 hover:bg-yellow-500 transition ease-out duration-500”>Tranzicija</div>
```

Primjer 34. Korištenje Tailwind tranzicije

Primjer 34. mijenja boju pozadine elementa “ ” iz zelene u žutu u trenutku kada se mišem prijeđe iznad elementa. Tranzicija traje 500 milisekundi (pola sekunde) te je tipa “ease-out” što znači da će animacija naglo krenuti, a polako završiti.

```
<div class="animate-bounce rounded p-4 bg-green-400 text-white text-sm"></div>
```

Primjer 35. Korištenje Tailwind „bounce“ animacije

Tailwind nudi samo 4 predefinirane animacije: okretanje (engl. *spin*), pulsiranje (engl. *pulse*), zviždanje (engl. *ping*) i odskakanje (engl. *bounce*). Kako su to najčešće upotrebljavane animacije, uglavnom nije potrebno stvarati nove, ali ako je potrebno, Tailwind dozvoljava stvaranje prilagođenih animacija uz pomoć CSS-ove @keyframes oznake.

4.3. Prilagodljiv dizajn

S obzirom na to da moderne aplikacije trebaju biti napravljene istovremeno za uređaje s različitom veličinom ekrana, time dizajn aplikacije treba biti prilagodljiv, drugim riječima dizajn treba odgovarati za različite veličine ekrana i prilagoditi se njima. Primjerice, na većim ekranima neki gumb trebao bi stajati u sredini ekrana dok na manjim ekranima u desnom gornjem kutu. Kako bi se olakšao proces prilagođavanja dizajna u Tailwindu, lako se može

specificirati kada i gdje se što treba nalaziti. U primjeru ispod (Primjer 36.) može se vidjeti da kada je ekran srednjeg ili većeg tipa, boja teksta elementa je plava, ali ako je ekran manjeg tipa, boja postaje zelena [15].

```
<div class=”text-blue-500 sm:text-green-500 ”>Tekst</div>
```

Primjer 36. Prilagodljiv dizajn

5. Okviri pozadinske strane

5.1. Node.js

Node.js je višeplatformsko okruženje poslužitelja otvorenog koda baziran na JavaScript jeziku te se koristi V8 JavaScript Engineom Google Chromea. Kreirao ga je Ryan Dahl 2009. godine. Pruža događajima upravljano, neblokirajuće (asinkrono) ulazno/izlazno i višeplatformsko okruženje za izgradnju visoko skalabilnih aplikacija na strani poslužitelja. Drugim riječima, Node je okruženje JavaScripta koje dozvoljava korištenje JavaScripta van web preglednika (npr. Chrome, Mozilla...). Time se JavaScript može koristiti za programiranje i poslužiteljevu stranu, umjesto prije samo klijentsku [16, str. 5].

5.1.1. Asinkron način izvršavanja procesa

S obzirom na to da je JavaScript jezik koji se ne može granati u više istovremenih procesnih niti (engl. *threads*), stvoren je Node koji dozvoljava asinkron način rada JavaScripta. Node je napravljen tako da teže zadatke prenese na računalo domaćina te računalo može stvoriti više različitih istovremenih procesa te se na taj način dobiva asinkrona funkcionalnost. Dakle, asinkron način izvršavanja zadataka je taj u kojemu zadaci rade nezavisno jedni o drugima dok sinkron način je taj u kojemu zadaci u redu čekaju jedan drugoga da budu izvršeni kako bi oni mogli doći na red izvršavanja [16, str. 5-9].

5.1.2. Node menadžer paketa (engl. Node Packet Manager, npm)

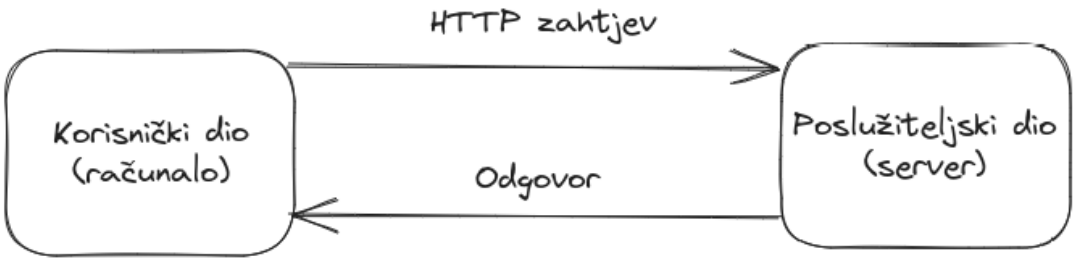
U trenutku instalacije Nodea, također je instaliran npm, paket menadžer za Node.js. On je odgovoran za upravljanje vanjskim paketima (modulima koje su drugi izgradili te koji su dostupni na internetu). Kada je potrebno, uz pomoć npm-a lako se može dodati modul u projekt. Instalacija se vrši preko komandne linije te generalno započinje s “npm install *ime modula*”. Ako je uspješna instalacija, dolazi do potvrde, dok se u suprotnom slučaju prikazuje greška [16].

5.2. Express.js

ExpressJs je aplikacijski okvir za stvaranje poslužiteljske strane *web*-aplikacija i stranica te je stvoren kao nadogradnja na Node.js. Express se nadograđuje na Nodeove značajke kako bi se osigurala jednostavnost korištenja funkcionalnosti koje zadovoljavaju potrebe mrežnog servera [16, str. 85-88].

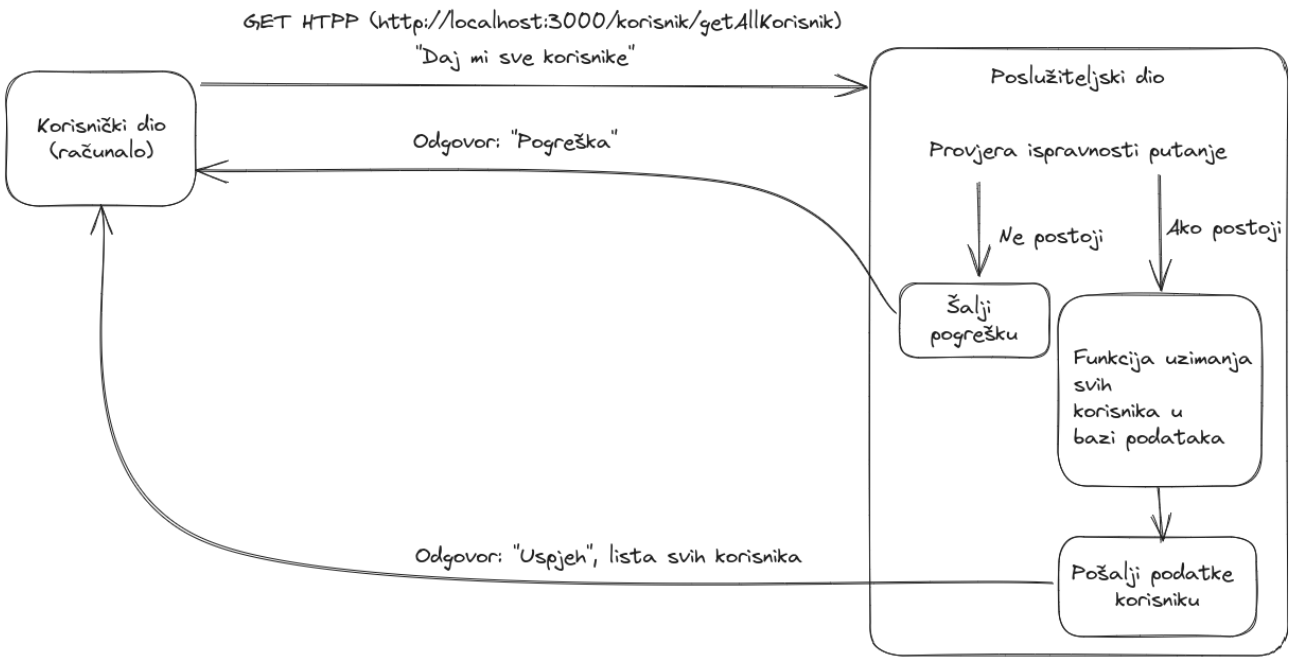
5.2.1. Komunikacija korisničke i poslužiteljeve strane

Web-aplikacije mogu se podijeliti na dva dijela, a to su poslužiteljski (engl. *server*) i korisnički dio (engl. *client*). Korisnički dio je onaj koji korisnik aplikacije može vidjeti npr. gumbi, tekst i animacije, dok je poslužiteljski dio onaj koji korisnik ne može vidjeti npr. algoritmi sortiranja podataka, struktura baze podataka, primanje GET i POST zahtjeva i slično. S obzirom na to da oni međusobno direktno ne mogu komunicirati, na neki način potrebno je uspostaviti vezu komunikacije.



Slika 7. Pojednostavljeni primjer komunikacije korisnika i poslužitelja [autorski rad]

Najčešće se u aplikacijskim okvirima JavaScripta za uspostavu komunikacije koristi Fetch sučelje za programiranje aplikacija. Prije negoli korisnički dio koristi Fetch, API poslužiteljski dio mora imati specificirane putanje za jedinstveno mjesto resursa (engl. Uniform Resource Locator, URL) za određene funkcije npr. “http://localhost:3000/korisnik/getAllKorisnik” putanja za uzimanje svih postojećih korisnika iz baze podataka. Nakon toga korisnički dio uz pomoć Fetch API-ja pristupa toj URL putanji i šalje zahtjeve poslužitelju koji u trenutku pristupa izvrši operaciju za tu specifičnu putanju te šalje korisniku odgovor. Odgovori na zahtjeve mogu biti razni, a najčešći su 200 za OK, 404 za “Not Found” (hrv. nije pronađeno) i 500 za “Internal Server Error” (hrv. unutarnja pogreška poslužitelja). Ako je odgovor pozitivan, također se šalju i potrebni podaci npr. za prošli primjer uz odgovor “200 OK” poslali bi se svi podaci o svim korisnicima u bazi podataka. Sljedeći primjer vizualno će prikazati objašnjen način komunikacije poslužitelja i korisnika.



Slika 8. Kompleksniji prikaz komunikacije korisnika i poslužitelja [autorski rad]

Dakako, gornji prikaz (Slika 8.) još uvijek ima sakrivenih dijelova komunikacije i procesuiranih podataka koji nisu prikazani namjerno kako bi se dobila generalna slika funkcionalnosti komunikacije. U praktičnom dijelu rada ovaj prikaz bit će detaljnije objašnjen.

Na ovaj način ExpressJs (i gotovo svi ostali aplikacijski okviri za poslužitelje) komunicira s korisničkim dijelom aplikacije. U ovome radu uz pomoć Expressa bit će napravljen poslužiteljski dio aplikacije te će komunicirati s korisničkim dijelom koji će biti napravljen u ReactJSu.

5.2.2. Parametri zahtjeva

Često su korisniku potrebni neki specifični podaci iz baze podataka. Tijekom komunikacije korisnik nekako treba poslužitelju moći specificirati određene podatke po kojima bi poslužitelj trebao filtrirati podatke iz baze podatka te bi samo te preostale filtrirane podatke trebao vratiti korisniku. To se radi na način da korisnik u tijelu (engl. *body*) zahtjeva prenosi kojeg tipa bi podaci trebali biti (npr. JavaScript Notacija Objekta (engl. JavaScript Object Notation, JSON)) i koji specifično podaci (npr. Ime mora biti „Marko”, a prezime „Marković”).

Poslužitelj (u ovom slučaju ExpressJs) treba moći pročitati navedene specifikacije iz tijela zahtjeva. U Expressu to se radi tako da se specificira URL putanja preko koje se pristupa za tu funkciju. Nakon toga stvara se funkcija koja ima objekte “request” za zahtjev i

“response” za odgovor. U funkciji se napravi konstanta ili varijabla u koju će se spremiti podaci (ako postoje). U tu konstantu ili varijablu piše se “imeObjektaZahtjeva.body.imeParametara”. Primjer 37. prikazat će dobivanje parametra za ime nekog korisnika preko tijela zahtjeva.

```
app.post('/korisnik/getByName', (req, res) => { const name =
req.body.name })
```

Primjer 37. Pristupanje podacima iz tijela zahtjeva [17, str. 21]

5.2.3. Odgovori na zahtjeve

Da komunikacija između poslužitelja i korisnika bude potpuna, nakon zahtjeva treba biti i odgovor. U Expressu napraviti odgovor, relativno je jednostavno. Za stvaranje odgovora koristi se “send()” metoda u “response” objektu.

```
(req, res) => res.send('Hello World!')
```

Primjer 38. Odgovor na zahtjev

Gornji primjer (Primjer 38.) jednostavan je primjer odgovora na zahtjev. Ovaj odgovor vraća rečenicu “Hello World!”. Također, može se specificirati koji status odgovora će biti npr. 200, 404 ili 500.

```
res.status(404).send('File not found')
```

Primjer 41. Odgovor na zahtjev sa statusom

Kod kompleksnijih odgovora koji sadržavaju neke podatke iz baze podataka najčešće se koristi JSON. JSON je podskup JavaScripta, on sam po sebi nije programski jezik, već zapravo podatkovni oblik za razmjenu podataka[18, str. 37].

```
res.json({ ime: 'Marko', prezime: 'Marković' })
```

Primjer 42. Odgovor na zahtjev s JSON formatom [17]

5.2.4. CORS

Zajedničko korištenje više izvora (engl. Cross-Origin Resource Sharing, CORS), mehanizam koji dozvoljava pristup različitim računalima na neku adresu preko HTTP-a. Kako bi aplikacije mogle raditi, u ovome završnom radu koristit će se CORS. Korištenje CORS-a specificira se na poslužiteljevoj strani. Prvo se preko npm-a instalira paket CORS. Nakon toga se ubacuje u skripti te se onda preko metode “use()” od Expressa krene koristiti CORS [19].

5.3. NestJS

NestJS je aplikacijski okvir za stvaranje poslužiteljske strane *web*-aplikacija i stranica. Nest umjesto JavaScripta koristi TypeScript što je programski jezik koji daje jednostavnost i snagu JavaScripta uz sigurnost tipa (engl. *type safety*) drugih programskih jezika npr. C#. Sigurnost tipa u Nestu je samo dostupno u vrijeme kompiliranja, jer je Nest poslužitelj kompiliran u ExpressJs poslužitelj koji pokreće JavaScript. Međutim, to je još uvijek prednost, budući da omogućuje programerima bolje dizajniranje programa bez pogrešaka prije izvođenja aplikacije [20, str. 8] [21].

Za razliku od Expressa Nest ima forsiran način programiranja poslužiteljske strane. Drugim riječima, za stvaranje dobrih poslužiteljskih dijelova u Nestu potrebno je pratiti Nestovu metodologiju programiranja. ExpressJs daje veliku slobodu programeru koristiti se svakakvim dodatnim alatima na bilo koji način programer želi dok Nest ne dozvoljava taj nivo slobode. Nest se striktno drži tog načina programiranja u kojem su 3 glavna dijela aplikacije koji komuniciraju međusobno (ti dijelovi bit će objašnjeni kasnije) te uz to forsira i korištenje objektno relacijskog preslikavanja (engl. Object-Relational Mapping, ORM).

Iako je isprva to negativna strana Nesta jer ne dozvoljava slobodu programeru, ovo se može smatrati i njegovom prednosti. Razlog tomu jest što taj način programiranja forsira programera da stvara čist kod (engl. *clean code*) koji je kroz povećavanje i skaliranje projekta ključna stvar ka čitljivosti, održavanju i funkcionalnosti koda.

5.3.1. Nest CLI

Sučelje naredbenog retka (engl. Command Line Interface, CLI) alat je preko kojeg korisnik računala može koristiti za direktnu komunikaciju s računalom uz pomoć teksta. Kako bi se olakšalo korištenje Nesta, kreatori Nesta stvorili su prilagođeni CLI za NestJS. Uz pomoć Nest CLI-ja programer može u samo par riječi stvoriti nove datoteke za posebne slučajeve korištenja. Primjerice, programer može stvoriti datoteke modula (moduli će biti objašnjeni kasnije) [20].

```
$ nest generate module Korisnik
```

Primjer 43. Generiranje modula „Korisnik“

Znak “$” označuje početak komandne linije (Primjer 43.). Generalni izgled generiranja sličnih komponenti izgleda:

```
nest generate <shematic> <name>
```

Primjer 44. Shema generiranja datoteka u Nestu [20]

“Schematic” dio Primjer 44. označuje module, servise, dekoratore, upravljače itd.

Nakon izvršenja Primjer 43. s modulom Korisnika nastaje datoteka Korisnik.module.ts u kojoj se nalaze bazni dijelovi svakog modula u Nestu.

```
import { Module } from '@nestjs/common';

@Module({
      controllers: []
      providers: []
})
export class KorisnikModule {}
```

Primjer 45. Kod modula u Nestu

5.3.2. Dekoratori (engl. *decorators*)

„Dekorator je posebna vrsta deklaracije koja se može priložiti deklaraciji klase, metodi, pristupniku, svojstvu ili parametru. [22]” Dekoratori označuju se oznakom “@dekorator” čije ime mora biti jednako s funkcijom koja će biti pozvana za vrijeme izvođenja. Drugim riječima, dekoratori su funkcije koje omotaju (engl. *wrappin*g) druge funkcije (i klase, metode itd.) s dodatnim atributima koji su unikatni tom omotaču. Dakle,

kada se poziva dekorator na neku funkciju, ta funkcija dobiva dodatne mogućnosti od tog dekoratora.

```
function iD(target: Function){
      target.prototype.id = 10
}
@iD
class KlasaPrimjer {
      id: number;
}
```

Primjer 46. Primjer dekoratora [autorski rad]

U primjeru (Primjer 46.) se može primijetiti da postoji obična funkcija koja prima jedan parametar koji je tipa funkcije zbog toga što su klase u TypeScriptu zapravo funkcije. Dekoratorska funkcija traži svojstvo "id" od ciljanog parametra te postavlja mu vrijednost 10.

Ovo nije koncept isključiv za NestJS ili TypeScript. Također, i drugi programski jezici koriste dekoratore npr. Python.

Razlog zašto je tek u ovom poglavlju objašnjen pojam dekoratora jest što se NestJS jako bazira na dekoratore te su oni jedan od ključnih dijelova za razumijevanje načina korištenja Nesta. Postoji mnogo dekoratora u Nestu od kojih su 4 najvažniji: dekoratori modula, upravljača, davatelja usluga i repozitorija. Dakako, moduli, upravljači, servisi i repozitoriji nisu dekoratori sami po sebi; kako bi mogli u Nestu funkcionirati, tako se moraju njihovi dekoratori eksplicitno navesti i koristiti.

5.3.3. Ubacivanje ovisnosti (engl. Dependency Injection)

Ubacivanje ovisnosti je tehnika dobavljanja ovisnog objekta kao što je modul ili komponenta, s ovisnošću poput usluge, čime ga ubrizgava u konstruktora komponente. Time kako je ovisnost ubrizgana u konstruktor komponente, svaki put kada je komponenta pokrenuta, time je i svaka ovisnost u njoj. Na taj način štedi se radna memorija poslužiteljskog dijela zato što se servisi, moduli i komponente koriste samo kad su potrebni. Uz to zadržava se pravilo čistog koda te je kod lakši za koristiti ubuduće [20, str. 13].

```
@Controller('/narudzba')
export class NarudzbaController {
constructor(private readonly narudzbaService: NarudzbaService) {}
```

Primjer 47. Kôd upravljača u Nestu

Gornji primjer (Primjer 47.) je primjer iz koda jedne aplikacije završnog rada. U upravljač narudžbe ubacuje se servis narudžbe u konstruktor upravljača.

5.3.4. Davatelji usluga

Davatelji usluga u Nestu se koriste za kreiranje servisa, tvornica, pomagača itd. Oni se mogu ubaciti u upravljače i druge davatelje usluga po potrebi. Davatelji usluga najčešće se koriste za stvaranje servisa. Servisi služe za stvaranje funkcija koje se mogu koristiti za interakciju s podacima iz baze podataka te s njihovom manipulacijom/filtriranjem [20, str. 22]. Sljedeći primjer (Primjer 48.) prikazuje jednostavni servis u Nestu.

```
@Injectable()

export class KorisnikService {
      constructor(
            @InjectRepository(Korisnik)
            private readonly korisnikRepository: KorisnikRepo,
      ) {}

async getAllKorisnik(): Promise<Korisnik[]> {
      return await this.korisnikRepository.find({
      select: ['korisnik_id', 'username', 'vrsta_korisnika_id'],
      });
}
```

Primjer 48. Davatelj usluga u obliku servisa

Oni asinkrono izvršavaju funkcije u sebi te ako su funkcije uspješne, nastavlja dalje dok ako je neka greška, izbacuje ju van. “<Korisnik[]>” dio tipa označuje da obećanje mora vratiti polje Korisnika iz baze podataka. S riječi “return” vraća se “this.korisnikRepository.find({select: ['korisnik_id', 'username', 'vrsta_korisnika_id'],})” što označuje korištenje repozitorija koji pristupa bazi podataka metodom “find()”. U toj metodi specificirani su stupci iz tablice Korisnik koji su potrebni u odgovoru. Ako tablica ima druge stupce koji nisu specificirani u funkciji, njihovi podaci bit će ignorirani te ne će biti u odgovoru koji će biti poslan korisničkom dijelu aplikacije. Gornji primjer koristi dekorator “@Injectable” što dozvoljava klasi KorisnikService mogućnost ubacivanja u druge servise, module, upravljače i sl. Klasa sadržava konstruktor u kojem je ubačena klasa repozitorija za korisnike (repozitorij će biti objašnjen u kasnijem poglavlju). Nakon toga Korisnik Service ima asinkronu funkciju “getAllKorisnik()” koja je tipa “Promise<Korisnik[]>”. “Promise” dio tipa

označuje da je funkcija tipa obećanja što su u TypeScriptu objekti koji su prikazani u odgovoru od baze podataka.

5.3.5. Moduli

Svaka Nest aplikacija bazira se na modulima od kojih je jedan korijenski modul. Moduli se označuju s “@Module” dekoratorom. Korijenski modul ubacuje u sebe sve ostale module preko “imports” dijela u kodu [20, str. 23].

```
import { Module } from '@nestjs/common';

import { AppController } from './app.controller';
import { AppService } from './app.service';
import { TypeOrmModule } from '@nestjs/typeorm';
import { typeOrmConfig } from 'config/typeorm.config';
import { KorisnikModule } from './korisnik/korisnik.module';
import { VrstaKorisnikaModule } from
'./vrstaKorisnika/vrstaKorisnika.module';
import { NarudzbaModule } from './narudzba/narudzba.module';
import { KupacModule } from './kupac/kupac.module';

@Module({
      imports: [
            TypeOrmModule.forRoot(typeOrmConfig),
            KorisnikModule,
            VrstaKorisnikaModule,
            NarudzbaModule,
            KupacModule,
      ],
      controllers: [AppController],
      providers: [AppService],
})
export class AppModule {}
```

Primjer 49. Korijenski modul u Nestu

U “imports” dijelu uz sve module treba se specificirati i povezivanje na bazu podataka. U ovome završnom radu to se radi uz pomoć TypeOrma koji će kasnije biti objašnjen. U datoteci “TypeOrmConfig” u gornjem primjeru postavljeni su podaci za povezivanje na bazu podataka kako se ne bi vidjeli privatni podaci kao što su lozinka i korisničko ime povezivanja. Modul također ubacuje upravljače i dobavljače usluga. Za korijenski modul upravljači i dobavljači nisu bitni jer se oni koriste kod drugih modula. U drugim modulima bitno je ubaciti

upravljače, dobavljače usluga i entitete koji reprezentiraju tablice u bazi podataka kao u donjem primjeru (Primjer 50.) [21].

```
import { Module } from '@nestjs/common';

import { KorisnikController } from './korisnik.controller';
import { KorisnikService } from './korisnik.service';
import { TypeOrmModule } from '@nestjs/typeorm';
import { KorisnikRepo } from './korisnik.repository';
import { Korisnik } from './korisnik.entity';

@Module({
      imports: [TypeOrmModule.forFeature([Korisnik])],
      controllers: [KorisnikController],
      providers: [KorisnikService, KorisnikRepo],
})
export class KorisnikModule {}
```

Primjer 50. Običan modul u Nestu

5.3.6. Upravljači

Upravljači upravljaju URL putanjama i zahtjevima na te putanje te se označuju "@Controller" dekoratorom. Oni odgovaraju na te putanje s potrebnim podacima klijentskom dijelu aplikacije. Nest će usmjeriti dolazne zahtjeve prema rukovatelju funkcije klasama upravljača ovisno o kojoj se putanji i vrsti zahtjeva radi [20, str. 22].

```
@Controller('/korisnik')

export class KorisnikController {
      constructor(private readonly korisnikService: KorisnikService)
      {}

@Get('/pregledKorisnika')
async getAllKorisnik(@Res() res): Promise<Korisnik[]> {
      const listaKorisnika = await
      this.korisnikService.getAllKorisnik();
      return res.status(200).json(listaKorisnika);
}
```

Primjer 51. Nest upravljač

U primjeru (Primjer 51.) uz dekorator "@Controller" može se vidjeti i dekorator "@Get" koji služi za specifikaciju tipa zahtjeva koji dolazi od klijentskog dijela aplikacije, tj. dekoratorom "@Get" upravljaču predstavlja zadatak da ako je poslan GET zahtjev na njegovu putanju (u ovom primjeru "/korisnik") te na njegovu potputanju

“/korisnik/pregledKorisnika”, treba se izvršiti funkcija pod tim dekoratorom (u gornjem primjeru je funkcija “getAllKorisnik()”).

5.3.7. TypeORM

TypeORM jedan je od raznih alata objektno relacijskih preslikavanja koji se može koristiti na više različitih relacijskih baza podataka. Relacijsko mapirani objekt je alat koji se koristi kako bi se zamaglila razlika između objekta u programu i tablice u bazi podataka. Radi na način da mijenja po potrebi programa stanje između objekta i tablice. Rezultat te promjene je entitet. Drugi naziv za entitet je objekt prijenosa podataka (engl. Data Transfer Object, DTO). Objekt prijenosa podataka zna kako čitati podatke iz baze podataka, ali i zapisivati podatke u nju [20, str. 59-60].

TypeORM bazira se na TypeScript jeziku što je idealno za NestJS s obzirom na to da je i on baziran na njemu.

TypeORM nudi stvaranje repozitorija. Repozitorij u TypeORM-u ima predefinirane funkcije upita prema bazi podataka što značajno olakšava čitljivost i stvaranje koda. Sljedeći primjer (Primjer 52.) prikazuje SQL upit koji se uz pomoć repozitorija zamijeni SQL-om [21].

```
SQL: “SELECT * FROM Korisnik”

TypeORM: “KorisnikRepository.find()”
```

Primjer 52. Zamjena SQL sintakse upita za TypeScript sintaksu

Primjer 52. je jednostavan primjer u kojem se selektiraju svi podaci iz tablice Korisnik. Cijelu tu naredbu TypeORM skratio je samo na metodu “find()”.

Uz pomoć TypeORM-a može se stvoriti entitet tablice iz baze podataka koji se prikazuje kao klasa u kodu NestJS-a.

```
import { Narudzba } from 'src/narudzba/narudzba.entity';

import {
      BaseEntity,
      Column,
      Entity,
      OneToMany,
      PrimaryGeneratedColumn,
} from 'typeorm';

@Entity('Kupac')
export class Kupac extends BaseEntity {
      @PrimaryGeneratedColumn()
      kupac_id: number;
```

```
	@Column({ nullable: false })
	naziv_kupca: string;

	@Column({ nullable: false })
	telefon: string;

	@Column({ nullable: true })
	email: string;

	@Column({ nullable: true })
	adresa: string;

	@OneToMany(() => Narudzba, (Narudzba) => Narudzba.kupac)
	narudzba: Narudzba[];
}
```

Primjer 53. TypeORM entitet

U Primjer 53. također se koriste dekoratori za specificiranje stupaca iz baze podataka. Na samom kraju primjera (Primjer 53.) nalazi se dekorator “@OneToMany” koji označuje relaciju ovog entiteta s drugim entitetom (u ovom slučaju s entitetom “Narudzba”). Dekorator “@OneToMany” govori da ovaj entitet (entitet “Kupac”) više entiteta “Narudzba”, dok entitet “Narudzba” može imati samo jedan entitet “Kupac”. To se zove 1:M veza ili riječima “jedan naprema više, više naprema jedan”.

Entiteti nisu samo za preslikavanje tablica u klase nego služe i za stvaranje objekta prijenosa podataka. DTO može se koristiti kao djelomična preslika entiteta za točan prikaz tipa podataka koji dolaze u tijelu zahtjeva korisnika prema poslužitelju na način da se za svaki slučaj korištenja (registracija korisnika, logiranje korisnika…) kreira poseban DTO koji se može koristiti u funkcijama upravljača i servisa. Time nije uvijek potrebno ponavljati kod za funkcije koje za različite svrhe primaju iste parametre, već se reciklira DTO za svaki slučaj u kojem je potreban. Na taj način kod je uredniji te jednostavniji za održavati s rastućom veličinom projekta u budućnosti.

```
import { IsNotEmpty } from 'class-validator';

export class LogInKorisnikDto {
	@IsNotEmpty({ message: 'Niste unijeli nadimak' })
	username: string;
```

```
	@IsNotEmpty({ message: 'Niste unijeli sifru' })
	password: string;
}
```

Primjer 54. TypeScript DTO

Primjer iznad (Primjer 54.) jednostavan je DTO za logiranje korisnika preko tablice “Korisnik”. Po izgledu veoma je sličan entitetu u prijašnjem primjeru. U Primjer 55. vidjet će se kako se koristi ovaj DTO.

```
@Post('/prijava/logIn')
async KorisnikLogIn(
	@Res() res,
	@Body() logInKorisnikDto: LogInKorisnikDto,
): Promise<Korisnik> {
	const korisnik = await
	this.korisnikService.getKorisnikByUsernamePassword(
		logInKorisnikDto.username,
		logInKorisnikDto.password,
	);
	return res.status(200).json(korisnik);
}
```

Primjer 55. Korištenje DTO-a u upravljaču

6. Baza podataka

Baza podataka je kolekcija podataka koja zadržava te podatke kroz duži period vremena. Ti podaci mogu biti čitani, ažurirani, izbrisani i sl. U početku korištenje baza podataka bilo je iznimno teško. Nisu postojale relacijske baze podataka već baze podataka bazirane na hijerarhiji. Svaki zapis u tim bazama podataka mogao je imati jedan ili više drugih zapisa s kojima je prvi povezan. Otprve nije kompliciranо za koristiti, ali kada nastane veći projekt, teško je tražiti potrebne podatke zato što nisu podijeljeni u sektore (tj. tablice) koji bi dali prikaz programeru što traži kao u relacijskim bazama podataka. Kasnije (u 1970-ima) nastale su relacijske baze podataka koje su drastično olakšale korištenje i manipuliranje podacima unutar istih. Relacijske baze podataka prisutne su i danas.

Za stvaranje i korištenje relacijskim bazama podataka izumljen je jezik isključivo za njih koji se zove strukturirani jezik upita (engl. Structured Query Language, SQL). S vremenom stvarale su se inačice SQL-a npr. MySQL i PostgreSQL [23, str. 3-5].

6.1. PostgreSQL

PostgreSQL jedna je od inačica SQL-a za stvaranje, korištenje i manipuliranje relacijskim bazama podataka. On se također smatra postrelacijskim sistemom baza podataka. Danas je Postgres jedna od najpopularnijih baza podataka [23, str. 3-5].

Jedan od razloga njegove popularnosti jest njegova mogućnost paraleliziranja upita. Paraleliziranje upita znači da se više upita može izvršavati istovremeno. Količina mogućih istovremenih upita uglavnom ovisi o računalu na kojem je baza podataka te koliko jezgri to računalo ima. Zato u teoriji performansa Postgresa raste linearno s brojem jezgri u računalu [23, str. 10].

Korištenje Postgresa relativno je jednostavno. Sljedeći primjeri (Primjer 56.) prikazat će jedne od najbitnijih naredbi u Postgresu.

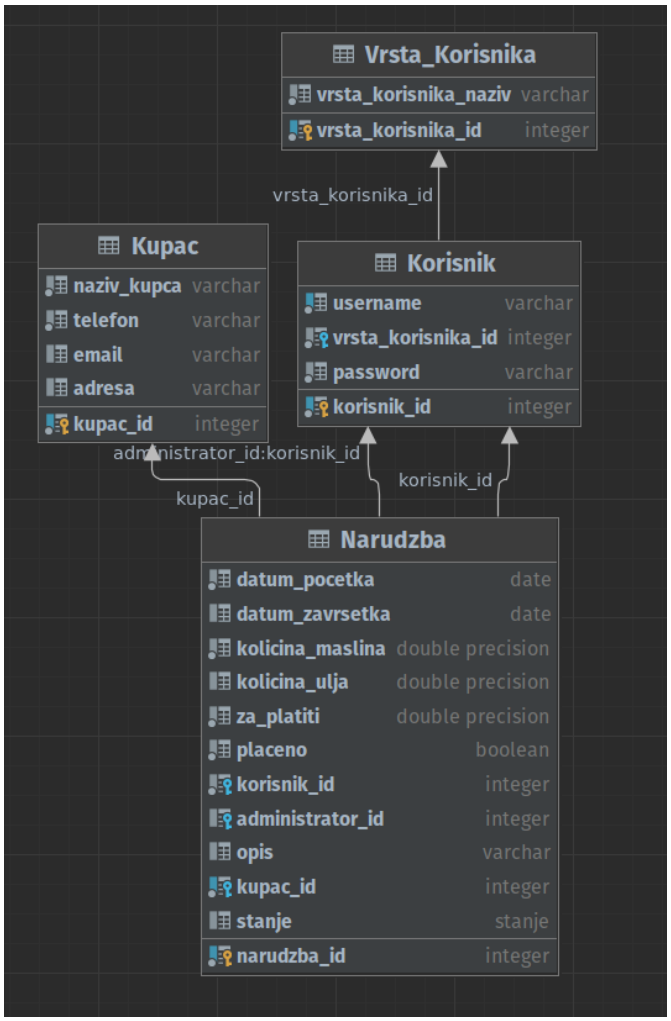
```
CREATE DATABASE BazaPodataka1;
CREATE TABLE Tablica1( sifra integer, ime char);
CREATE USER korisnik1 WITH PASSWORD 'jw8s0F4';
INSERT INTO Tablica1 VALUES ( 1, ‘Marko’ );
```

Primjer 56. PostgreSQL osnovne naredbe [autorski rad]

Prva naredba stvara bazu podataka naziva “BazaPodataka1”. Druga naredba stvara tablicu naziva “Tablica1” te dodaje dva stupca, a to su stupac “sifra” koji je tipa cijelog broja i stupac “ime” koji je tipa znak. Treća naredba stvara korisnika “korisnik1” s lozinkom. Inače, za pristup Postgres bazi podataka potrebni su korisnici kao i u standardnim aplikacijama. Određeni korisnici imaju određena prava za čitanje, pisanje i mijenjanje baze podataka. Zadnja naredba u tablicu “Tablica1” dodaje vrijednosti 1 i “Marko”.

Osoba koja dobro poznaje engleski jezik ne bi trebala imati poteškoće razumjeti naredbe Postgresa.

Raditi velike projekte preko naredbenog retka i ručno upisivati naredbe koje znaju biti jako duge može biti teško i naporno. Zbog toga su napravljeni alati koji dozvoljavaju programerima da preko grafičkog sučelja rade na bazi podataka. Jedan od takvih alata je Datagrip. Datagrip uz navedeno grafičko sučelje također dozvoljava i korištenje naredbenog retka ako ga programer želi koristiti za neke naredbe. Uz to ima i mogućnost prikaza dijagrama baze podataka što vizualno prikazuje sve tablice baze podataka te kako su one povezane međusobno. Sljedeća slika (Slika 10.) je primjer iz baze podataka ovog završnog rada.



Slika 10. ERA dijagram baze podataka

Kako bi se tablice mogle povezati u relacijskim bazama podataka, koriste se vanjski ključevi (engl. *foreign keys*) [23, str. 39]. Vanjski ključevi odnose se na primarni ključ druge tablice. Primjerice, tablica “Narudzba” iz gornjeg primjera sadržava vanjski ključ nazivom “kupac_id” koji se odnosi na primarni ključ tablice “Kupac” čiji je naziv “kupac_id”. Time kada se ispišu svi podaci o nekoj narudžbi također će se ispisati unikatni identifikator (“kupac_id”) kupca koji je naručio narudžbu. Na ovaj jednostavan način efikasno se može pratiti veliki kup podataka tko, kako i zašto je povezan s ostalim dijelovima baze podataka.

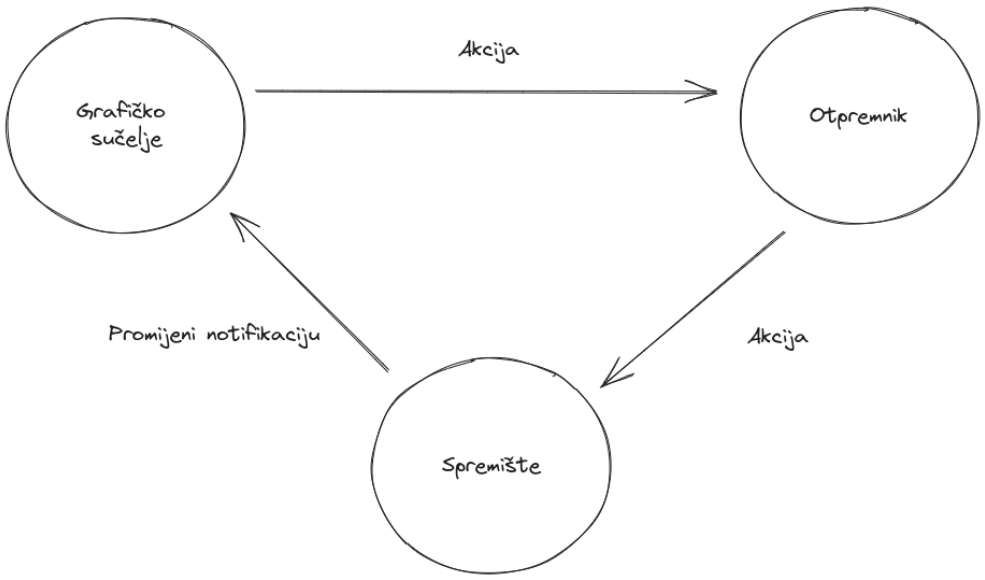
7. Arhitekture za razvoj aplikacija

Pojam arhitekture koda za razvoj aplikacija odnosi se na način strukturiranja koda i datoteka unutar aplikacije. Postoje razni načini slaganja koda koji svaki ima prednosti i nedostatke. Neke vrste arhitekture koriste se samo u posebnim slučajevima ili aplikacijskim okvirima. Primjerice, NestJS nema ime za svoju arhitekturu, ali samo u njemu se koristi specifično takva arhitektura koda. U ovome radu fokus je na dva načina strukturiranja koda, a to su Redux i komponentno bazirana arhitektura.

7.1. Redux

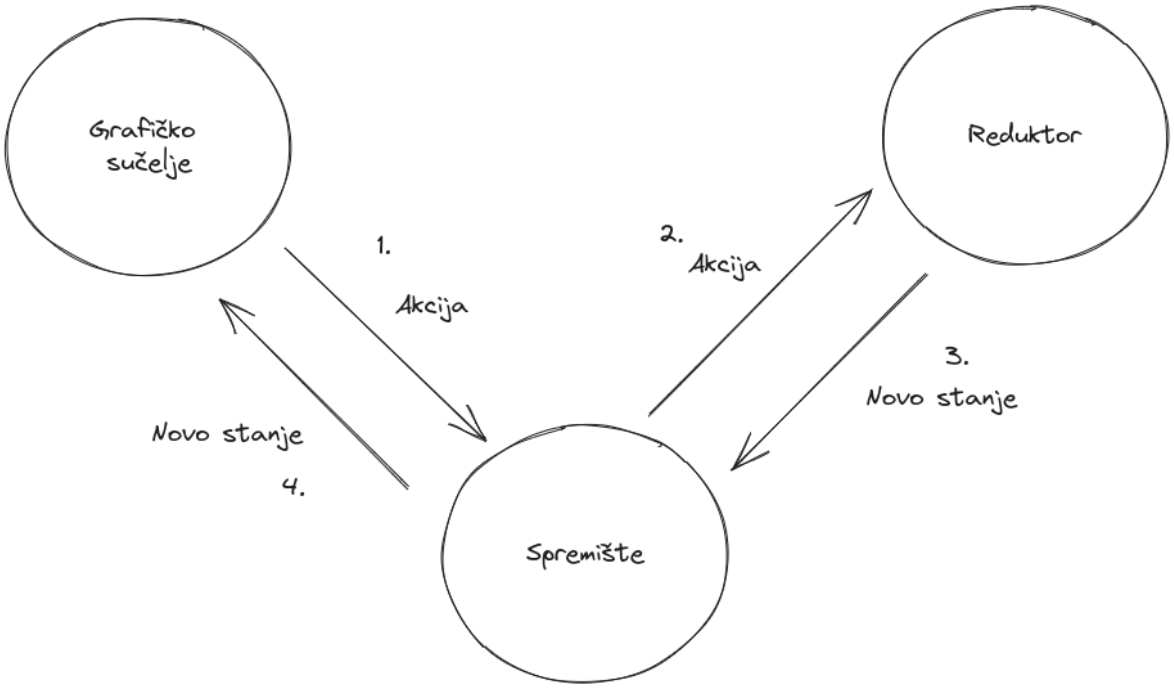
Redux je arhitektura za razvoj aplikacija koja se uglavnom koristi u aplikacijskom okviru ReactJS. Redux je zapravo nadogradnja Flux arhitekture te za razumijevanje Reduxa potrebno je prije razumjeti Flux.

Flux je arhitektura u kojoj se događaji upravljaju jedan po jedan u kružnom toku s 3 glavna aktera: akcija, otpremnik (engl. *dispatcher*) i spremište. Spremište u Reduxu je jednako kao i spremište u Svelteu. Ono u sebi sprema stanje koje se može preko drugih komponenti manipulirati i mijenjati. Akcija je struktura koja opisuje bilo koji događaj u aplikaciji npr. klik miša na gumb. Otpremnik je mjesto u koje se šalju akcije s kojima otpremnik mijenja stanje spremišta [24, str. 6].



Slika 11. Flux graf komunikacije [24, str. 7]

Redux umjesto otpremnika ima reduktore (engl. *reducers*). Za razliku od Fluxa, Redux sadržava samo jedno spremište koje samo po sebi nema logike. Spremište direktno prima akcije što eliminira potrebu za otpremnikom. Zauzvrat, spremište prenosi akcije na funkcije koje mijenjaju stanje. Te funkcije su reduktori.



Slika 12. Redux komunikacija [autorski rad]

Dakle, u Reduxu se prvo preko grafičkog sučelja detektira neki događaj koji aktivira akciju. Akcija se direktno šalje u spremište koje šalje akciju reduktoru. Reduktor mijenja stanje te šalje to stanje spremištu koje ga pročita i ažurira te nakon toga ažurira grafičko sučelje s novim stanjem [24, str. 8].

7.2. Komponentno bazirana arhitektura

Komponentno bazirana arhitektura je način strukturiranja i stvaranja koda u kojem je komponenta glavni akter. Komponenta može biti bilo kakav dio grafičkog sučelja koji korisnik aplikacije vidi. Primjerice, jedna komponenta može biti tablica s podacima svih korisnika ili gumb preko kojeg se prijavljuje na aplikaciju. U ovom načinu strukturiranja svaka komponenta se stavlja u jednu datoteku samo za tu komponentu. Ako se komponenta treba

ponavljati kroz više različitih stranica u aplikaciji, ne treba se ponovno pisati kôd za svaku instancu te komponente nego se samo treba instancirati komponentu u obliku objekta ili HTML elementa.

Za ovu arhitekturu bitna je mogućnost davanja parametara, funkcija i objekta preko komponenata, tj. bitna je mogućnost korištenja rekvizita (engl. *props*) i spremišta (engl. *stores*). Zbog mogućnosti rekvizita kôd nije previše objektno orijentiran, već sadrži i dio funkcionalne orijentacije programiranja što se u današnje doba smatra velikom prednosti.

8. Praktični dio rada

8.1. Odabir slučaja korištenja

Svrha aplikacija završnog rada nije samo prikaz korištenja različitih aplikacijskih okvira, nego i prikaz metodologije i načina razmišljanja programera koji stvara aplikacije za realne slučajeve korištenja. Slučaj korištenja odabran za ovaj završni rad je prerada maslinovog ulja u uljari. Proces prerade maslinovog ulja podrazumijeva zaprimanje maslina kupaca/klijenata, preradu maslina u maslinovo ulje, završavanje narudžbe i informiranje kupca o završenoj narudžbi. Dakako, ovo je pojednostavljen prikaz procesa prerade te se po tom prikazu baziraju aplikacije završnog rada kako ne bi bio dug i kompliciran.

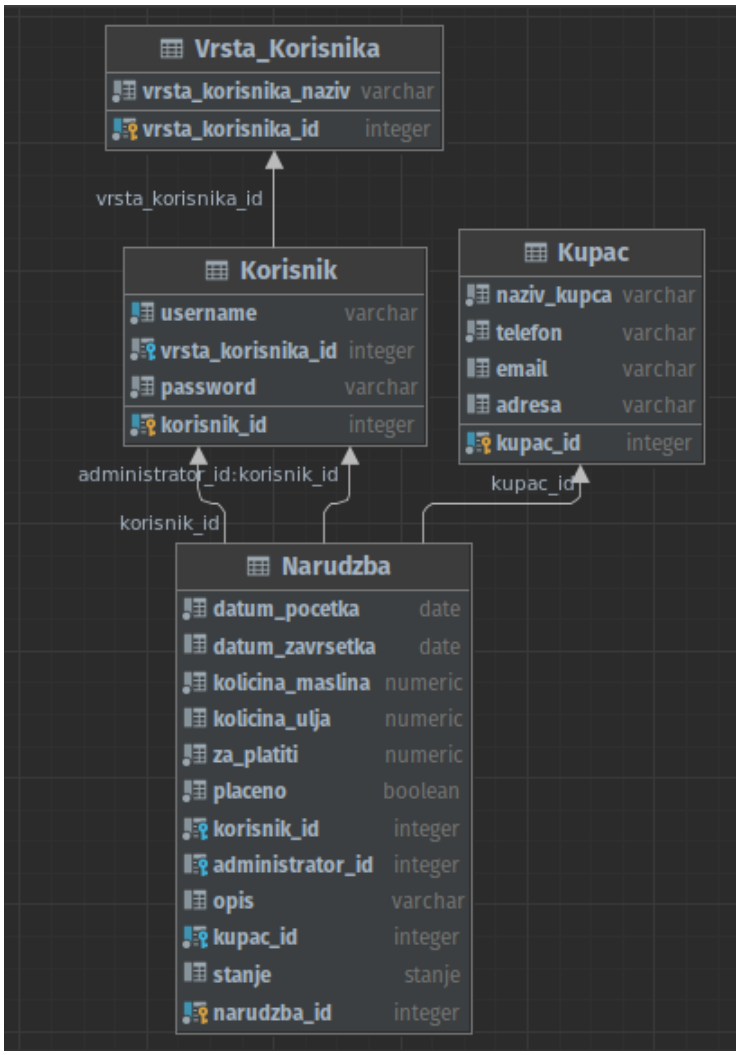
8.2. Rad aplikacija u uljari

Kako bi se efikasno mogli usporediti aplikacijski okviri završnog rada, obje aplikacije moraju biti napravljene za isti slučaj korištenja s minimalnim različitostima. Time su napravljene dvije aplikacije od kojih jedna s okvirima React, Express i Tailwind, dok druga sa Svelte, Nest i također Tailwind zbog toga što je u ovom završnom radu samo jedan aplikacijski okvir za dizajn aplikacija. Aplikacije će zajedno biti opisane s procesom prerade kako bi se dobio bolji uvid u oboje.

Početak prerade maslina u maslinovo ulje započinje s narudžbom kupca za preradu. Aplikacije trebaju u trenutku stvaranja narudžbe zapisati: identifikacijski broj narudžbe, ime i prezime kupca, količinu maslina za preraditi (u kilogramima), koji korisnik aplikacije je zaprimio narudžbu, datum zaprimljene narudžbe, opis narudžbe (ako ga je korisnik napravio), koliko će korisnik trebati platiti na završetku prerade te stanje narudžbe (što je u početku "u čekanju"). Ako u aplikaciji ne postoji kupac, korisnik ga treba zapisati u aplikaciju kako bi uljara tijekom prerade mogla pratiti čije je maslinovo u čekanju, procesu prerade ili završeno. Također, u budućnosti se može pratiti tko je koliko maslina donosio za preradu u uljaru. Nakon upisa narudžbe skladište se masline do trenutka prerade.

Kada se masline postavljaju u proces prerade, korisnik aplikacije treba u aplikaciji specificirati koje masline sa stanjem "u čekanju" mijenjaju stanje u "u procesu". Kada je to napravljeno, administrator aplikacije može pratiti čije masline su stavljene u proizvodnju, u kojoj količini te time može pretpostaviti vrijeme potrebno za preradu u maslinovo ulje i uz pomoć tih informacija davati potrebne odluke ostalim korisnicima (tj. radnicima u uljari).

Kada je proces prerade gotov, administrator postavlja stanje narudžbe u "završeno" i time može pratiti sve završene narudžbe. Također, administrator u trenutku mijenjanja stanja narudžbe treba specificirati datum završetka prerade narudžbe, količinu dobivenog maslinovog ulja i je li plaćena narudžba. Sljedeća slika (Slika 13.) prikazuje kako su svi podaci narudžbi, korisnika i kupca zapisani u bazi podataka.



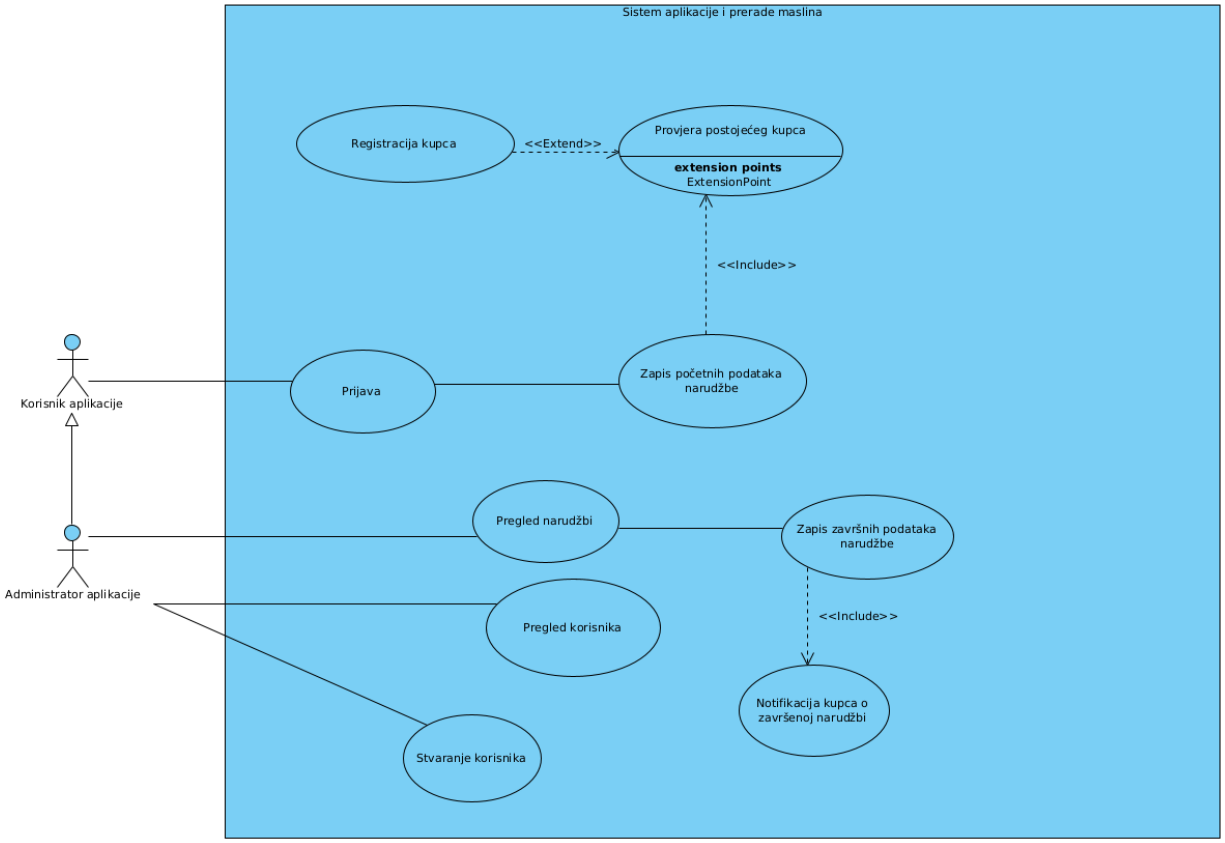
Slika 13. Prikaz tablica baze podataka

Kao što se na slici (Slika 13.) vidi, postoji više vrsta korisnika, a to su korisnik i administrator. Korisnik može u aplikaciju samo unijeti kupca i narudžbu te pregledati narudžbe i kupce. Administrator može raditi sve što i korisnik, ali uz to može unijeti korisnika, završiti narudžbu, pregledati korisnika.

8.3. Dijagrami i vizualizacija rada uljare i aplikacija

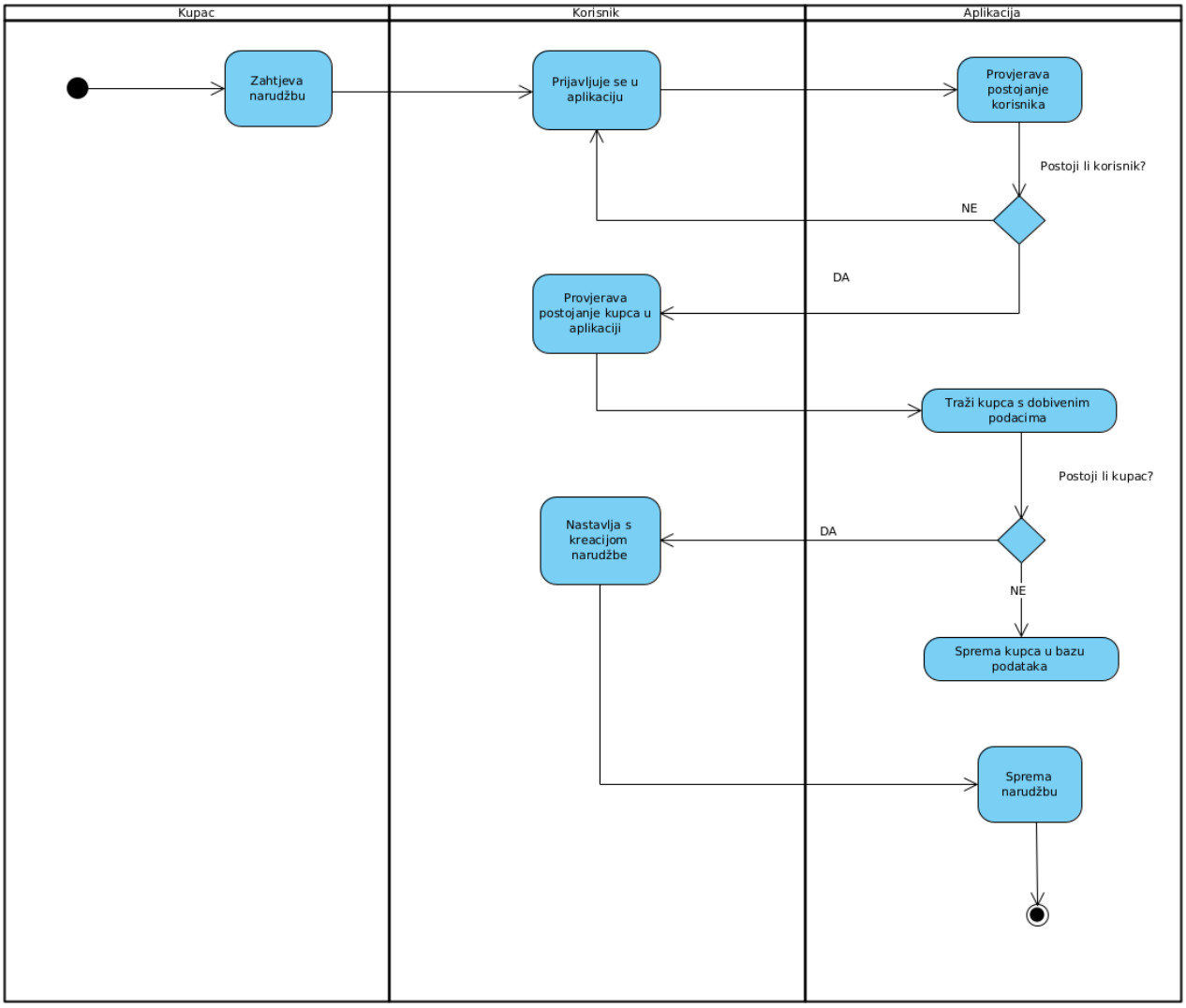
Kako bi se lakše razumjeli procesi rada uljare i aplikacija, u ovome poglavlju bit će prikazani određeni dijagrami koji će na različite načine detaljnije prikazati ne samo rad uljare nego i kako aplikacija u suštini funkcionira.

Sljedeći je osnovni dijagram slučaja korištenja koji pojašnjava prijašnje poglavlje opisa rada uljare i aplikacije. Ovaj dijagram prikazuje samo "što" aplikacija radi, ali ne i "kako" (Slika 14.).



Slika 14. Dijagram slučaja korištenja uljare i aplikacija

Za prikazati "kako" aplikacija radi potreban je dijagram aktivnosti. Za ovaj završni rad napravljena su 3 dijagrama aktivnosti koji obuhvaćaju najbitnije dijelove procesa prerade. Prvi dijagram aktivnosti prikazuje proces zaprimanja narudžbe (Slika 15.).

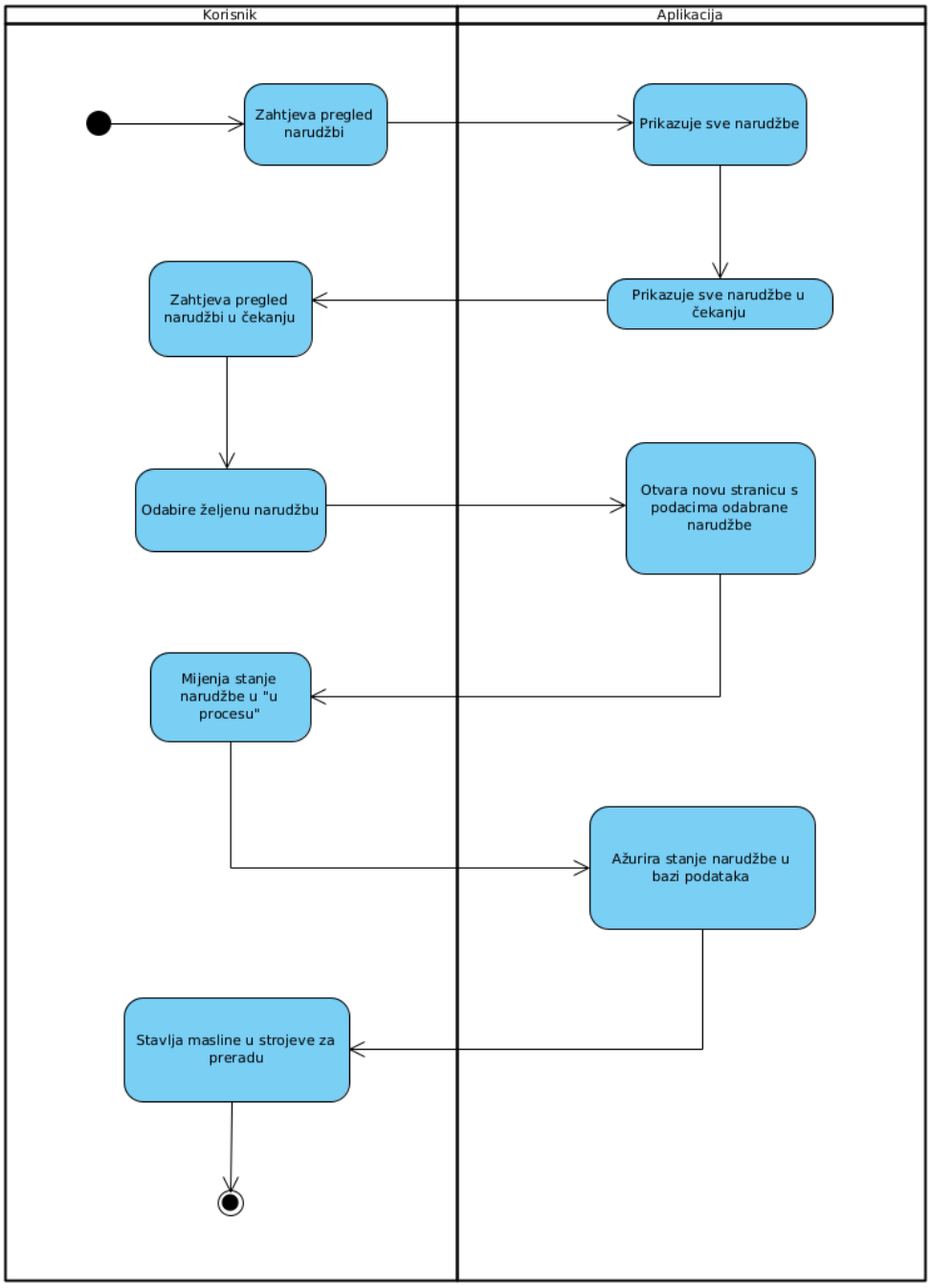


Slika 15. Dijagram aktivnosti prijave i zaprimanja narudžbe

Kupac zahtijeva narudžbu te se nakon toga korisnik prijavljuje u aplikaciju. Ako korisnik ne postoji s danim podacima, korisnik se mora ponovno pokušati prijaviti. Kada se korisnik uspješno prijavi u aplikaciju, prvo provjerava postoji li taj kupac u aplikaciji. Ako kupac ne postoji, korisnik ga mora prvo unijeti prije negoli nastavi s unosom narudžbe. Nakon toga, kada je kupac unesen ako prije nije postojao, korisnik unese sve potrebne podatke i uz pomoć aplikacije sprema ih u bazu podataka. U sljedećim dijagramima dio prijavljivanja korisnika/administratora neće se prikazivati kako se ne bi ponavljalo. Početak budućih dijagrama aktivnosti postavlja se od pretpostavke da je korisnik/administrator već prijavljen.

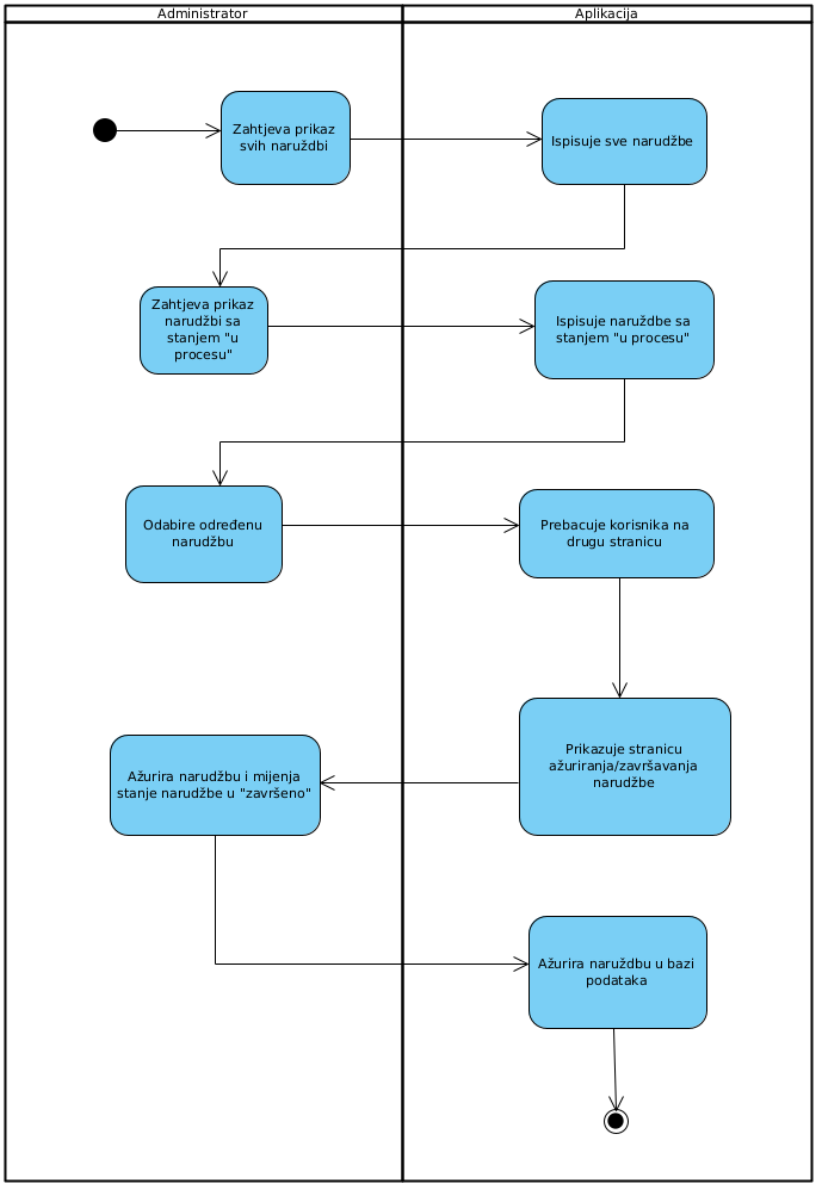
Drugi dijagram (Slika 16.) aktivnosti prikazuje proces postavljanja narudžbe u preradu. Korisnik prvo od aplikacije zahtijeva pregled narudžbi, tj. ulazi u stranicu pregleda narudžbi te se na ulazu automatski šalje zahtjev za prikaz narudžbi. Kako bi lakše pronašao

određenu narudžbu u čekanju, korisnik specificira aplikaciji da mu trebaju samo narudžbe sa stanjem "u čekanju". Aplikacija prikazuje na ekran sve narudžbe sa stanjem "u čekanju". Korisnik odabire željenu narudžbu te joj mijenja stanje u "u procesu". Aplikacija nakon mijenjanja stanja ažurira stanje te narudžbe u bazi podataka. Nakon toga korisnik uzima masline odabrane narudžbe i stavlja ih u stroj za početak prerade.



Slika 16. Dijagram aktivnosti pokretanja narudžbe

Posljednji dijagram (Slika 17.) aktivnosti služi za prikaz završavanja narudžbe. Administrator jedini može završiti narudžbu zbog toga što se on nalazi pored kase uljare kojoj samo on ima pristup.



Slika 17. Dijagram aktivnosti završavanja narudžbe

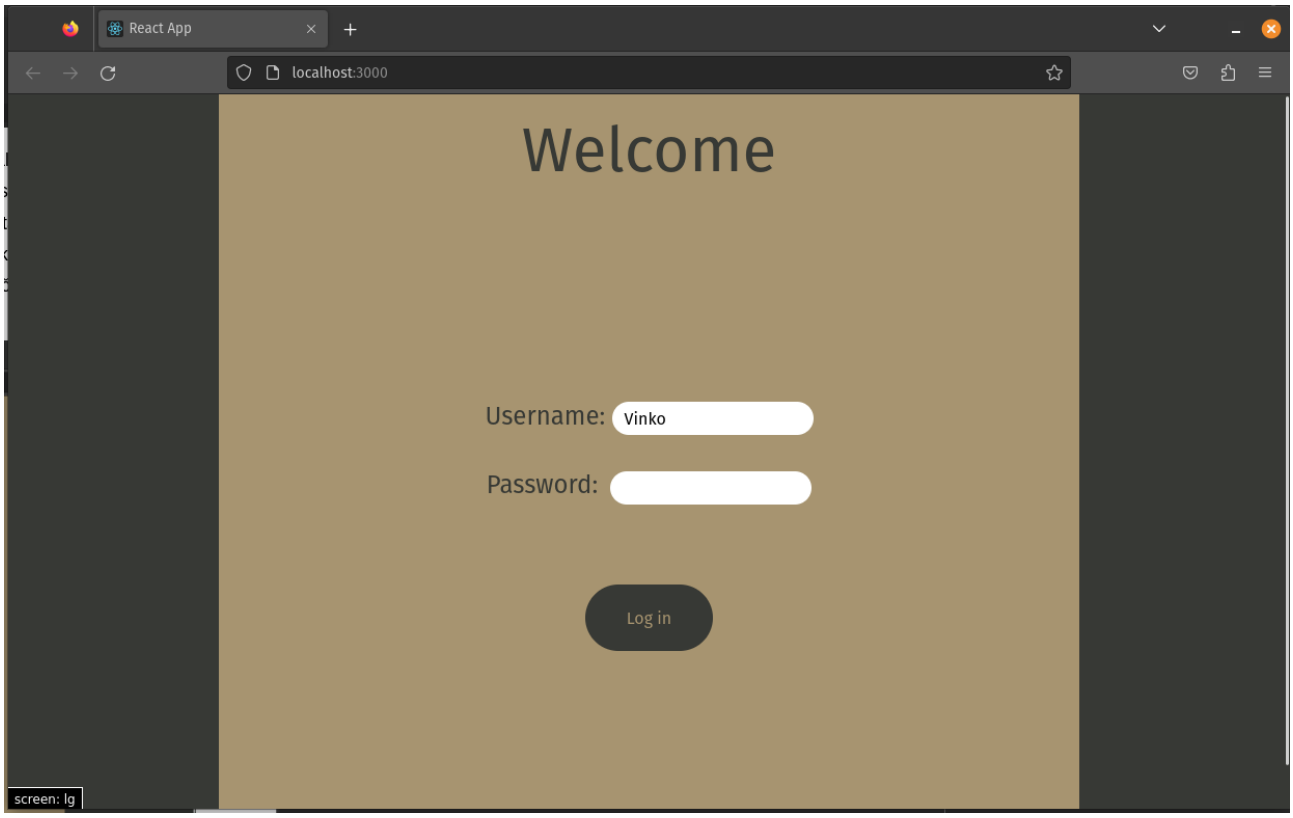
Prvo administrator (kao i korisnik u prošlom dijagramu) zahtijeva od aplikacije prikaz narudžbi. Nakon toga zahtijeva prikaz narudžbi koje su u procesu. S obzirom na to da aplikacija ne može pratiti direktno proces prerade preko strojeva, administrator prvo mora dobiti preko radnika informaciju čija je narudžba gotova te koliko je maslinovog ulja nastalo. U trenutku kada administrator dobije tu informaciju, odabire u aplikaciji narudžbu s danim podacima o završetku od strane radnika. Aplikacija prebacuje administratora na drugu

stranicu u kojoj će unijeti potrebne podatke o narudžbi. Nakon unosa podataka aplikacija sprema nove podatke i ažurira narudžbu u bazi podataka.

8.4. Prva aplikacija

Prva aplikacija napravljena je s aplikacijskim okvirima React.js i Express.js. Uz pomoć React-a napravljena je korisnička strana aplikacije, a s Expressom poslužiteljeva strana aplikacije. Kako ovo poglavlje ne bi bilo dugo i komplicirano, fokus će biti samo na implementaciji najbitnijih dijelova aplikacije, a to su slučajevi korištenja opisani prijašnjim dijagramima aktivnosti. Sporedne funkcionalnosti aplikacije bit će ili ignorirane ili samo spomenute ako je potrebno za opis neke glavne funkcionalnosti aplikacije. Ovo pravilo vrijedit će i za opis druge aplikacije.

Prijava korisnika u aplikaciju prvi je korak za korištenje aplikacije. S obzirom na to da aplikacije nisu postavljene na internet, jedini način pristupa aplikaciji je preko terminala u aplikaciji koja se koristi za programiranje istih aplikacija. Nodeov menadžer paketa koristi se za stvaranje React projekta te zbog toga se i koristi za pokretanje istog. Stvaranje React projekta radi se s naredbom "npx create-react-app ime-aplikacije". Uz pomoć naredbe "npm run start" pokreće se React projekt te prva stranica koja se otvara je stranica prijave (za ovaj projekt, inače, otvara se stranica napravljena od kreatora Reacta) (Slika 18.).



Slika 18. Stranica za prijavu korisnika

Vrijedi napomenuti da se za pokretanje bilo kojeg projekta koji se instalira preko npm-a treba prvo preko terminala postaviti u mapu u kojoj se nalaze sve datoteke projekta npr. Ako je projekt instaliran u mapi "React-client" uz pomoć naredbe "cd ./React-client", postavlja se programer aplikacije u tu mapu. Na slici se ne prikazuje lozinka radi sigurnosti. Sljedeći je kod (Primjer 52.) stranice prijave, ali samo od korisničke strane.

```
import { useNavigate } from "react-router-dom";
import { useDispatch } from "react-redux";
import { logIn } from "../redux/loginStore";
import { logOut } from "../redux/loginStore";

export default function LogIn(){
const navigate = useNavigate();
const routeChange = () =>{
      let path = `/Narudzba/Pregledaj`;
      navigate(path);
};
const dispatch = useDispatch();

const logInFunction = () => {
```

```
            fetch('http://localhost:8080/korisnik/
            getKorisnikByUsername&Password', {
                  method: 'POST',
                  headers: {
                  'Content-Type': 'application/json',
      },
      body: JSON.stringify({
            username: document.getElementById('username').value,
            password: document.getElementById('password').value
      })
}).then(response => response.json())
.then(data => {
      console.log('Success:', data.korisnik_id, data.username,
      data.vrsta_korisnika_id);
      if (data.username !== undefined) {
            dispatch(logIn({korisnik_id_logIn: data.korisnik_id,
            username: data.username, vrsta_korisnika_id:
            data.vrsta_korisnika_id}));
            routeChange();
      }
      else {
            dispatch(logOut());
            alert('Pogresni podaci');
      }
})};
return (
      <div className="flex items-center justify-center h-screen bg-
      primary debug-screens">
            <div className="relative grid h-48 grid-cols-1 px-64 bg-
            secondary py-96 place-content-center rounded-3xl">
            <h1 className="absolute inset-x-0 top-0 flex justify-
            center pt-16 text-6xl text-primary">Welcome</h1>
            <form className="relative grid grid-cols-1 row-span-1
            space-y-8 place-content-center place-items-center h-96">
                  <div>
                        <label for="username" className="text-2xl
                        text-primary">Username: </label>
                        <input className="py-1 rounded-full indent-3"
                        type="text" id="username" name="username"/>
                  </div>
                  <div>
                        <label for="password" className="text-2xl
                        text-primary ">Password:</label>
                        <input className="py-1 ml-3 rounded-full
                        indent-3" type="text" id="password"
                        name="password"/>
                  </div>
                  <button className="absolute bottom-0 px-10 py-5
                  rounded-full bg-primary text-secondary"
```

```
                  id="loginButton" type="button"
                  onClick={logInFunction}>Log in</button>
            </form>
            </div>
      </div>
      );
}
```

Primjer 57. Kod prijave u aplikaciju

U kodu iznad (Primjer 57.) može se primijetiti korištenje kuka (engl. *hooks*) kao što je "useDispatch()" kuka koja dopušta ovoj datoteci projekta koristiti akcije iz Reduxa. Akcije u ovom kodu su "logIn" i "logOut". Kôd ovih akcija je sljedeći (Primjer 58.).

```
import { createSlice } from "@reduxjs/toolkit";

export const logInSlice = createSlice({
      name: 'logIn',
            initialState: {
                  loggedIn: false,
                  korisnik_id_logIn: 0,
                  username: '',
                  password: '',
                  vrsta_korisnika_id: 0,
            },
      reducers: {
      logIn: (state, action) => {
            state.loggedIn = true;
            state.korisnik_id_logIn =
            action.payload.korisnik_id_logIn;
            state.username = action.payload.username;
            state.password = action.payload.password;
            state.vrsta_korisnika_id =
            action.payload.vrsta_korisnika_id;
      },
      checkLogIn: (state, action) => {
            return state.loggedIn;
            },
      logOut: (state, action) => {
            state.loggedIn = false;
            state.korisnik_id_logIn = 0;
            state.username = '';
            state.password = '';
            state.vrsta_korisnika_id = 0;
            },
      }
});
```

```
export const { logIn, checkLogIn, logOut } = logInSlice.actions;

export default logInSlice.reducer;
```

Primjer 58. Redux kod za prijavu korisnika

U kodu iznad (Primjer 58.) koristi se Redux za pamćenje stanja podataka vezanih za korisnika koji se želi prijaviti na aplikaciju. Akcija "logIn" prima parametre "state" i "action". Parametar "state" odnosi se na stanje prijašnje definirano pod imenom "logIn" 9 linija koda iznad akcije "logIn". Parametar "action" sadržava teret (engl. *payload*) što su zapravo samo podaci preneseni negdje u kodu (u ovom slučaju u prvom primjeru koda s kukom "useDispatch()" (primjer broj)). U akciji "logIn" stanje se ažurira s obzirom na prenesene podatke preko "action.payload". Na ovaj način kada se korisnik prijavi njegovi podaci su spremljeni u Redux spremište (engl. *store*). Bitno je napomenuti da Redux spremišta zapamte podatke samo lokalno u *web*-pregledniku, tj. ako se stranica osvježi ili kompletno izađe iz stranice, sve stanje Redux spremišta se briše i postavlja na inicijalno stanje "initialState:" kao u primjeru iznad (Primjer 58.).

Također React kod za prijavu (Primjer 57.) koristi "fetch()" metodu koja se koristi za uspostavljanje komunikacije između korisničkog i poslužiteljevog dijela aplikacije, ali samo na vrijeme koliko je korisničkom dijelu potrebno. Jednom kada se s "fetch()" metodom pozove poslužitelj, komunikacija traje do trenutka dok korisnik ne dobije željene podatke. U trenutku dobivanja podataka komunikacija između korisnika i poslužitelja se obustavlja. U primjeru (Primjer 57.) u metodi "fetch()" postavljen je parametar 'http://localhost:8080/korisnik/getKorisnikByUsername&Password' koji je zapravo jedna od putanja koje ima poslužitelj. Po imenu putanje može se razumjeti da korisnik želi pristupiti korisniku u bazi podataka s određenim nadimkom i lozinkom. Nadimak i lozinka čitaju se iz JSX-a te postavljaju u tijelo zahtjeva. Zahtjev mora biti tipa "POST" kako bi se parametri mogli staviti u tijelo jer ako je zahtjev tipa "GET" korištenje tijela zahtjeva nije moguće. Sljedeći kod (Primjer 59.) poslužitelja u Expressu prikazat će funkcionalnost te putanje.

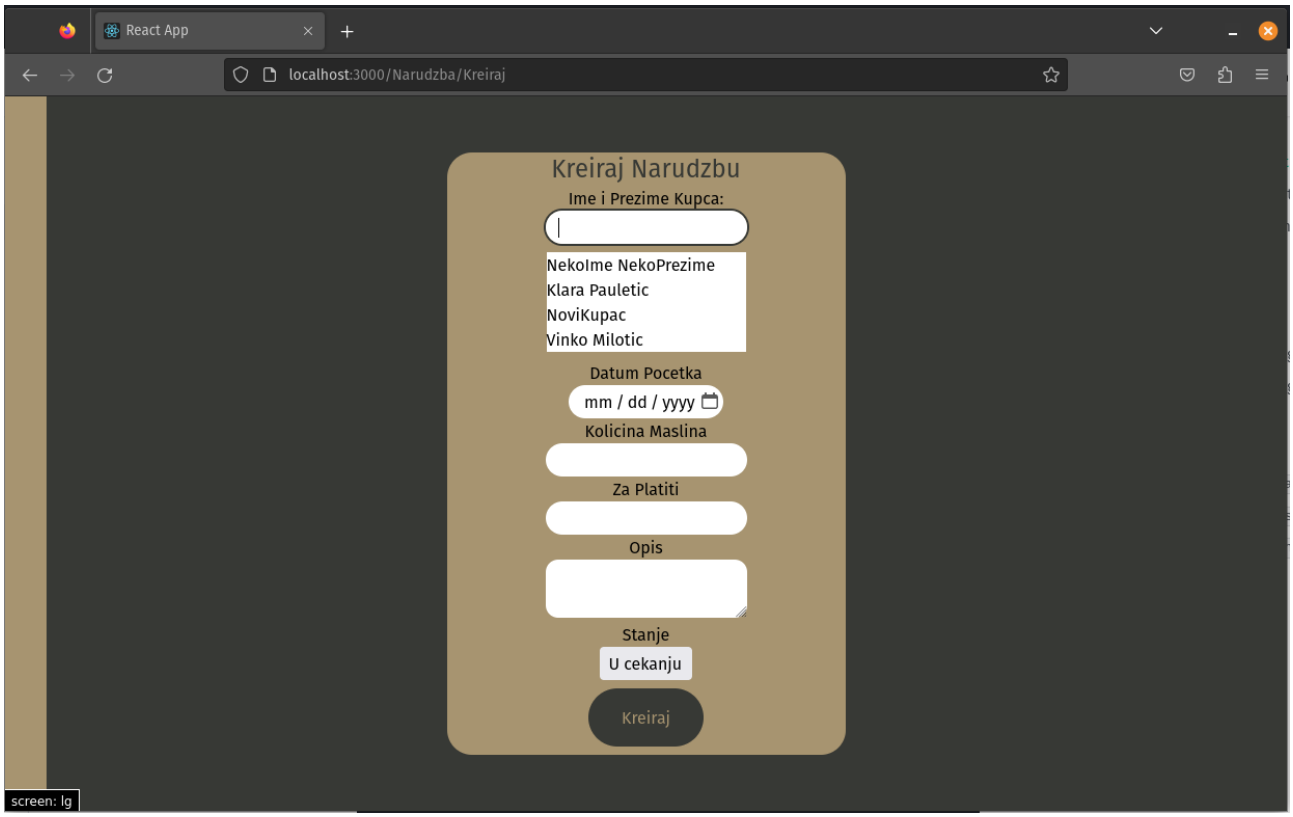
```
let {Client} = require('pg');
const router = require('express').Router();

const database = new Client({
      host: 'postgresql-124060-0.cloudclusters.net',
      user: 'Vinko',
      password: '',
      database: 'BazaPodatakaUljara',
      port: '10109',
```

```
});

database.connect();


router.post('/getKorisnikByUsername&Password', async (req, res) => {
      let queryResult = await database.query('SELECT
      "korisnik_id","username","vrsta_korisnika_id" FROM "Korisnik"
      WHERE "username" = $1 AND "password" = $2', [req.body.username,
      req.body.password]);

      return res.json(queryResult.rows[0]);
});
```

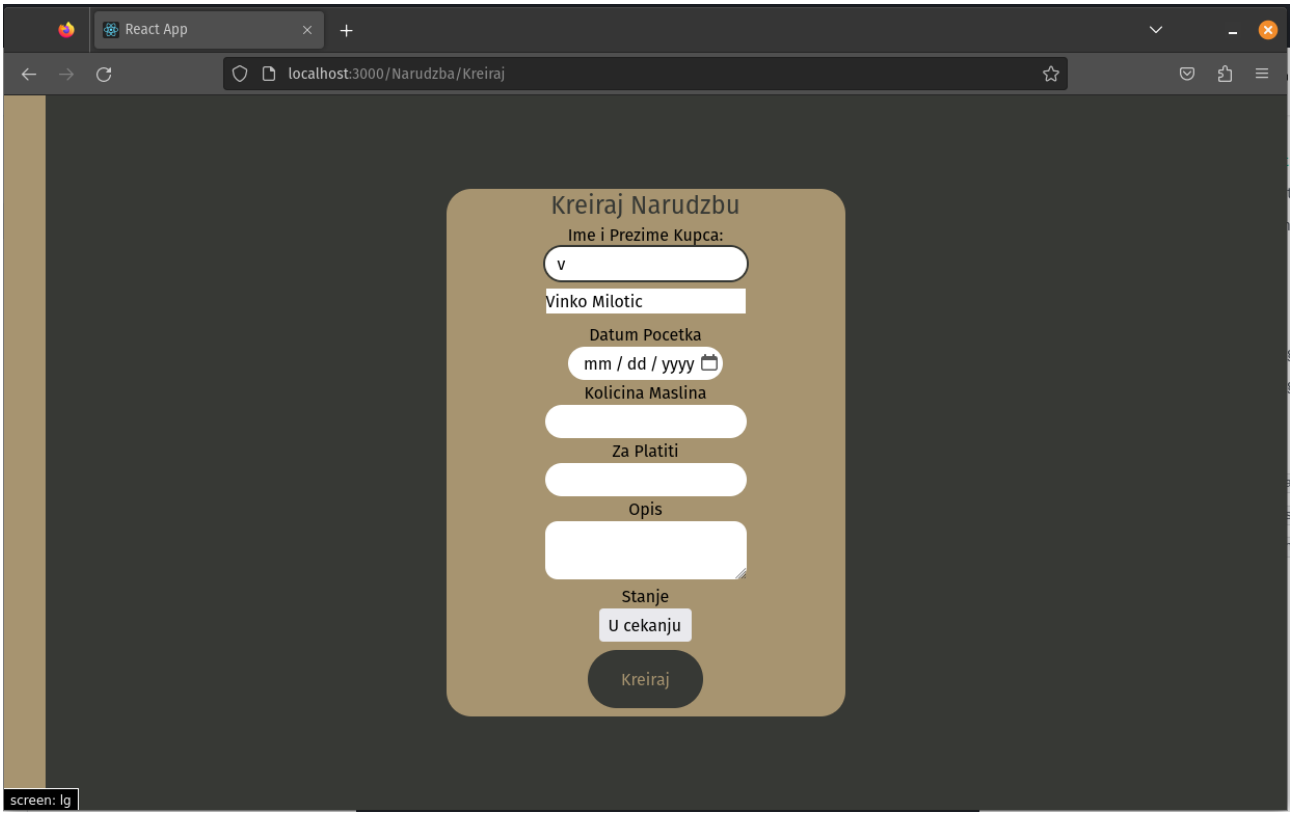
Primjer 59. Kod poslužitelja u Expressu

Kako bi poslužitelj mogao pristupiti bazi podataka, potrebno je instalirati paket "pg". To se radi uz pomoć naredbe "npm install pg". Postoje razni paketi za spajanje na bazu podataka, ali za spajanje na ovu bazu podataka potreban je specifično ovaj jer služi za spajanje na baze podataka stvorene u PostgreSQL jeziku. U kodu je sakrivena lozinka radi sigurnosti aplikacije, ostali podaci su istiniti. Nakon povezivanja na bazu podataka u Expressu specificiraju se tip zahtjeva i putanja na kojima će poslužitelj odraditi potrebnu funkcionalnost koja je u ovom slučaju prijava u aplikaciju. Može se primijetiti u kodu (Primjer 54.) da se upiti pišu u PostreSQL jeziku. Koristi se asinkrona funkcija za cijelu operaciju dohvata korisnika s nadimkom i lozinkom koji su specifirani u korisničkom dijelu (Primjer 59.). Na kraju se vraća odgovor u formatu JSON te se u njemu kao odgovor stavlja prvi red rezultata upita tako da se pozove rezultat "queryResult" i specificira prvi red polja "rows[0]".

Nakon prijave ulazi se u stranicu "Kreiraj Narudzbu" (Slika 19.). Na toj stranici korisnik može upisati potrebne podatke za početak narudžbe. Ispod polja unosa imena i prezimena kupca nalazi se polje s predloženim imenima ovisno o slovima koje je unio korisnik (Slika 20.).



Slika 19. Stranica kreacije narudžbe



Slika 20. Stranica kreacije narudžbe s predloženim imenom

Kada korisnik klikne gumb "Kreiraj", spremaju se dani podaci narudžbe te se stvara nova narudžba u bazi podataka. U korisničkom dijelu (Primjer 60.) za unos narudžbe ponovno se koristi "fetch()" metoda dok u poslužiteljevom dijelu (Primjer 61.) postavlja se zahtjev tipa "POST" i koristi se upit kao u prijavi korisnika (Primjer 59.)

```
export default function KreirajNarudzbu () {
const getKupacIDByNameAndAdresa = () => {
      fetch('http://localhost:8080/kupac/getKupacIDByNameAndAdresa',
      {
      method: 'POST',
      headers: {
            'Content-Type': 'application/json',
      },
      body: JSON.stringify({
            naziv_kupca:
            document.getElementById('naziv_kupca').value,
            adresa: document.getElementById('adresa_search').value
      })
      }).then(response => response.json())
      .then(data => {
      if (data[0].kupac_id === 0) {
            alert("Pogreška u dohvatu kupca!");
      }else {
            setKupacID(data[0].kupac_id);
      }
});
}

const createNarudzba = (e) => {
      e.preventDefault();
      fetch('http://localhost:8080/narudzba/createNarudzba', {
      method: 'POST',
      headers: {
      'Content-Type': 'application/json',
      },
      body: JSON.stringify({
            datum_pocetka:
            document.getElementById('datum_pocetka').value,
            kolicina_maslina:
            document.getElementById('kolicina_maslina').value,
            za_platiti: document.getElementById('za_platiti').value,
            placeno: false,
            korisnik_id: korisnik_id_logIn,
            opis: document.getElementById('opis').value,
            kupac_id: kupac_id,
            stanje: document.getElementById('stanje').value
```

```
      })
      }).then(response => response.json())
      .then(data => {
            console.log('Success:', data);
            routeChangeOnSuccess();
      });
}

return (
<div className="flex justify-center h-screen debug-screens place-
items-center bg-primary">
<NavBar />
<form autoComplete="off" className="relative grid grid-cols-1
rounded-3xl place-items-center w-96 bg-secondary"
onSubmit={createNarudzba}>
      <h1 className="text-2xl text-primary">Kreiraj Narudzbu</h1>
      <label>Ime i Prezime Kupca:</label>
      <input className="py-1 rounded-full indent-3 focus:outline-none
      focus:ring-2 ring-primary" type='text' id='naziv_kupca'
      name='naziv_kupca' onChange={filterKupacNames}/>
      <div className='w-48 mt-2 mb-2 bg-white'>
            {
            listaKupaca.filter(kupac => {
                  const searchItem =
                  document.getElementById('naziv_kupca').value.toLower
                  Case();
                  const imekupca = kupac.naziv_kupca.toLowerCase();
            if (searchItem === "") {
                  return kupac;
            }
            else if
            (kupac.naziv_kupca.toLowerCase().startsWith(searchItem)){
                  return imekupca;
            }
            }).map((kupac) => (
                  <div className="className='indent-2 hover:bg-
                  tertiary hover:text-primary' " key={kupac.kupac_id}
                  onClick={changeNameOnEvent}
                  value={kupac.naziv_kupca}>{kupac.naziv_kupca}</div>
            ))
            }
      </div>
      <label>Unesi adresu koja odgovara kupcu: </label>
      <input className="py-1 rounded-full indent-3 focus:outline-none
      focus:ring-2 ring-primary" type='text' id='adresa_search'
      onChange={filterKupacAdresa}/>
      <div className='w-48 mt-2 mb-2 bg-white'>
      {
      listaSvihAdresa.filter(adresa => {
```

```
            const searchItem =
            document.getElementById('adresa_search').value.toLowerCas
            e();
            const adresaKUpca = adresa.adresa.toLowerCase();
      if (adresaKUpca.toLowerCase().startsWith(searchItem)) {
            return adresa;
      }
      }).map((adresa) => (
      <div className='indent-2 hover:bg-tertiary hover:text-primary'
      key={adresa.adresa} type='button' onClick={changeAdresaOnEvent}
      value={adresa.adresa}>{adresa.adresa}</div>
      ))}
      </div>
      <label>Datum Pocetka</label>
      <input className="py-1 rounded-full indent-3 focus:outline-none
      focus:ring-2 ring-primary" type='date' id='datum_pocetka'
      name='datum_pocetka' />
      <label>Kolicina Maslina</label>
      <input className="py-1 rounded-full indent-3 focus:outline-none
      focus:ring-2 ring-primary" type='text' id='kolicina_maslina'
      name='kolicina_maslina' />
      <label>Za Platiti</label>
      <input className="py-1 rounded-full indent-3 focus:outline-none
      focus:ring-2 ring-primary" type='text' id='za_platiti'
      name='za_platiti' />
      <label>Opis</label>
      <textarea className="py-1 text-center rounded-xl focus:outline-
      none focus:ring-2 ring-primary" type='number' id='opis'
      name='kupac_id' />
      <label className="mt-1">Stanje</label>
      <select className="px-2 py-1 mb-2 overflow-visible rounded
      appearance-none focus:outline-none" type='text' id='stanje'
      name='stanje'>
            <option value='u_cekanju'>U cekanju</option>
            <option value='u_procesu'>U procesu</option>
            <option value='zavrseno'>Zavrseno</option>
      </select>
      <button className="p-4 px-8 mb-2 rounded-full bg-primary text-
      secondary hover:ring-2 hover:ring-primary hover:bg-tertiary
      hover:text-primary" type='submit'>Kreiraj</button>
</form>
</div>
)
```

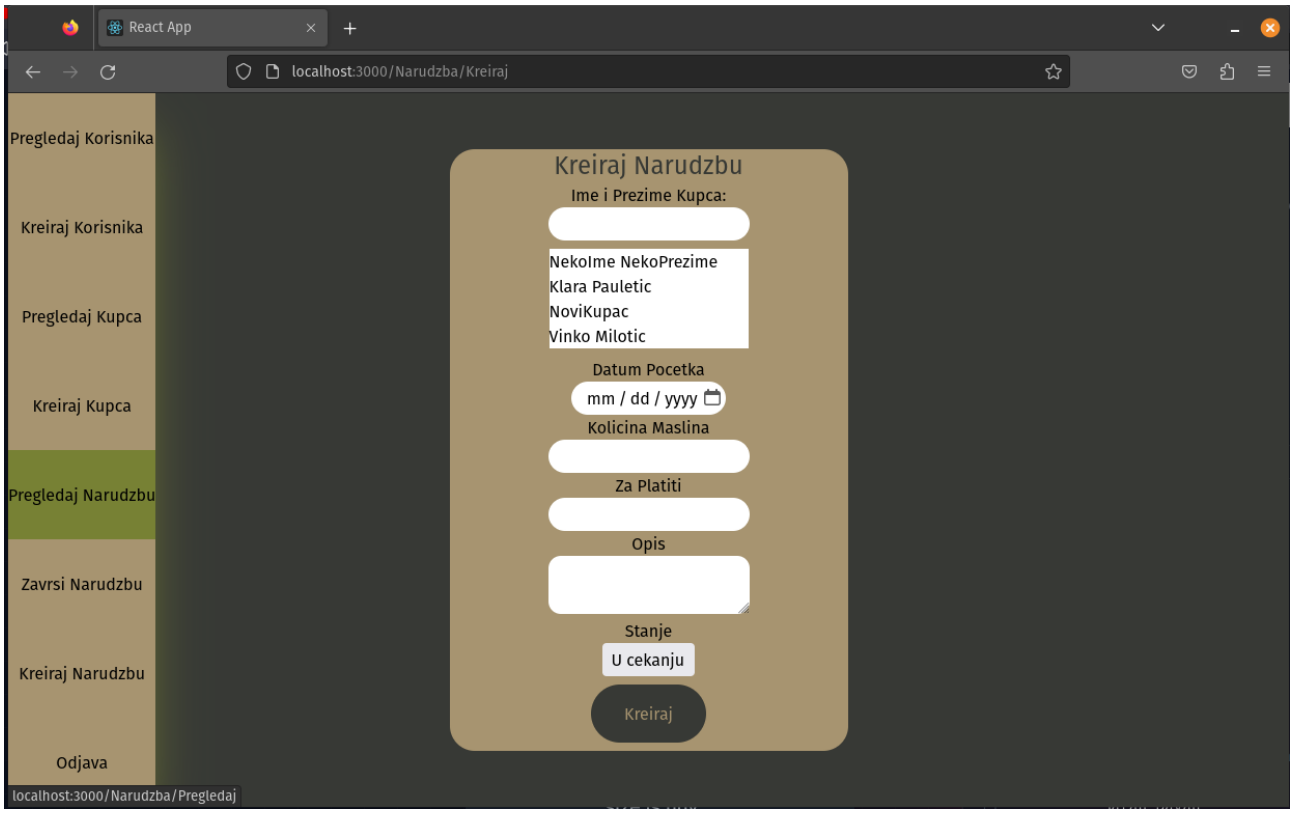
Primjer 60. Kod korisničkog dijela kreacije narudžbe

```
router.post('/createNarudzba', async (req, res) => {
```

```
        try {
              let queryResult = await database.query('INSERT INTO
              "Narudzba"
              ("datum_pocetka","kolicina_maslina","za_platiti","placeno
              ","korisnik_id","opis","kupac_id","stanje") VALUES ($1,
              $2, $3, $4, $5, $6, $7, $8)',
              [req.body.datum_pocetka, req.body.kolicina_maslina,
              req.body.za_platiti, req.body.placeno,
              req.body.korisnik_id, req.body.opis, req.body.kupac_id,
              req.body.stanje]);
        if (!queryResult) {
              throw new Error("Narudzba nije kreirana!");
        }
        else{
              return res.json({ results: "Narudzba uspjesno
              kreirana!" });
        }
        } catch (error) {
              console.log(error);
              return res.json({ results: "Narudzba nije kreirana!" });
        }
});
```

Primjer 61. Kod poslužiteljevog dijela kreacije narudžbe

Kada korisnik treba pokrenuti narudžbu kupca, prvo treba pristupiti stranici pregleda narudžbi. U ovoj aplikaciji pristup drugim stranicama aplikacije vrši se preko navigacijske trake aplikacije (Slika 21).



Slika 21. Navigacijska traka

Kako je navigacijska traka jednaka u svim stranicama, nema smisla ponavljati isti kod na svakoj stranici te je zbog toga napravljena komponenta "NavBar" (kratica za engl. "Navigation Bar", hrv. navigacijska traka) (Primjer 62.).

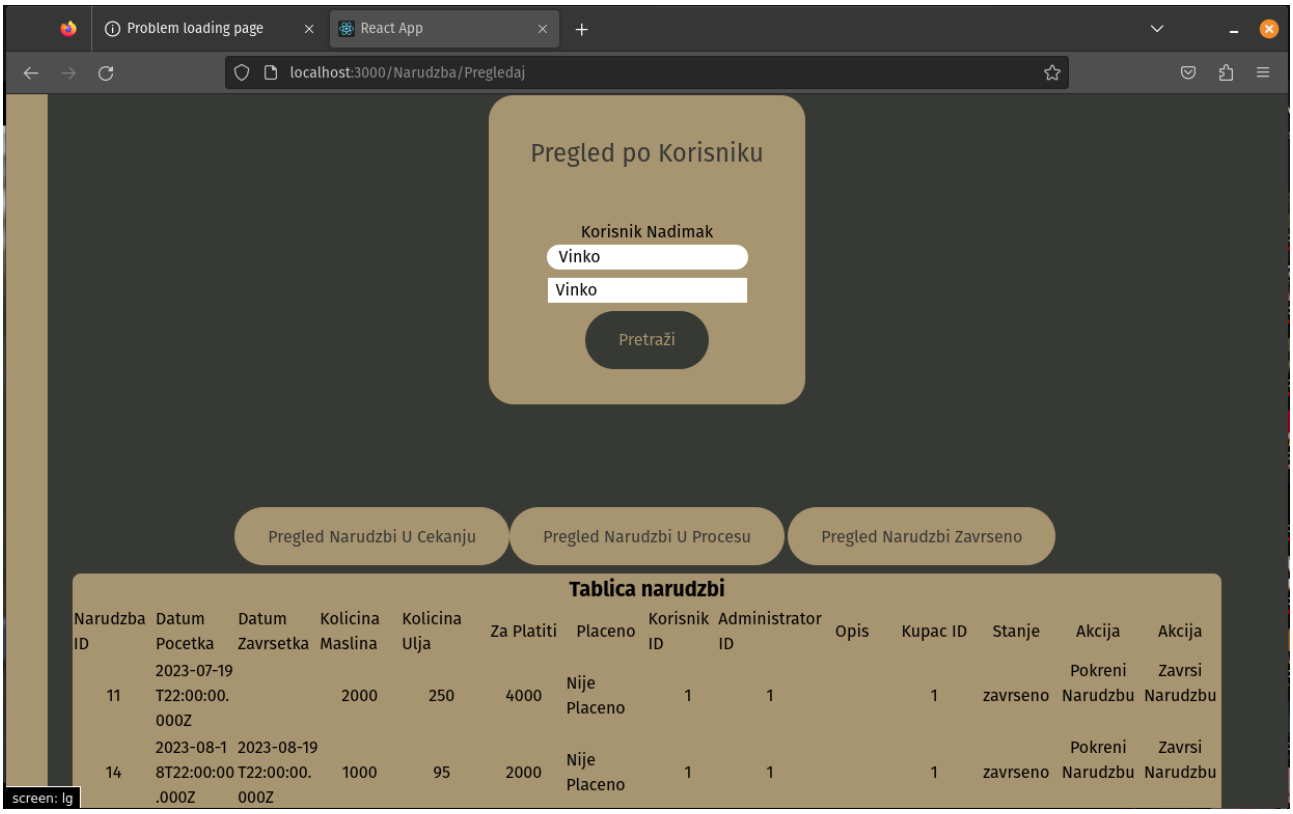
```
import { Link } from 'react-router-dom';

const NavBar = () => {
return(
      <ul className='fixed left-0 grid float-left w-10 h-screen grid-
      cols-1 hover:shadow-2xl hover:shadow-tertiary hover:w-auto
      group place-items-stretch justify-stretch bg-secondary'>
            <li className='flex items-center justify-center invisible
            w-full group-hover:visible hover:bg-tertiary'><Link
            to="/Korisnik/Pregledaj">Pregledaj Korisnika</Link></li>
            <li className='flex items-center justify-center invisible
            w-full group-hover:visible hover:bg-tertiary'><Link
            to="/Korisnik/Kreiraj">Kreiraj Korisnika</Link></li>
            <li className='flex items-center justify-center invisible
            w-full group-hover:visible hover:bg-tertiary'><Link
            to="/Kupac/Pregledaj">Pregledaj Kupca</Link></li>
```

```
                <li className='flex items-center justify-center invisible
                w-full group-hover:visible hover:bg-tertiary'><Link
                to="/Kupac/Kreiraj">Kreiraj Kupca</Link></li>
                <li className='flex items-center justify-center invisible
                w-full group-hover:visible hover:bg-tertiary'><Link
                to="/Narudzba/Pregledaj">Pregledaj Narudzbu</Link></li>
                <li className='flex items-center justify-center invisible
                w-full group-hover:visible hover:bg-tertiary'><Link
                to="/Narudzba/Zavrsi">Zavrsi Narudzbu</Link></li>
                <li className='flex items-center justify-center invisible
                w-full group-hover:visible hover:bg-tertiary'><Link
                to="/Narudzba/Kreiraj">Kreiraj Narudzbu</Link></li>
                <li className='flex items-center justify-center invisible
                w-full group-hover:visible hover:bg-quaternary'><Link
                to="/">Odjava</Link></li>
            </ul>
    )
    }

    export default NavBar;
```

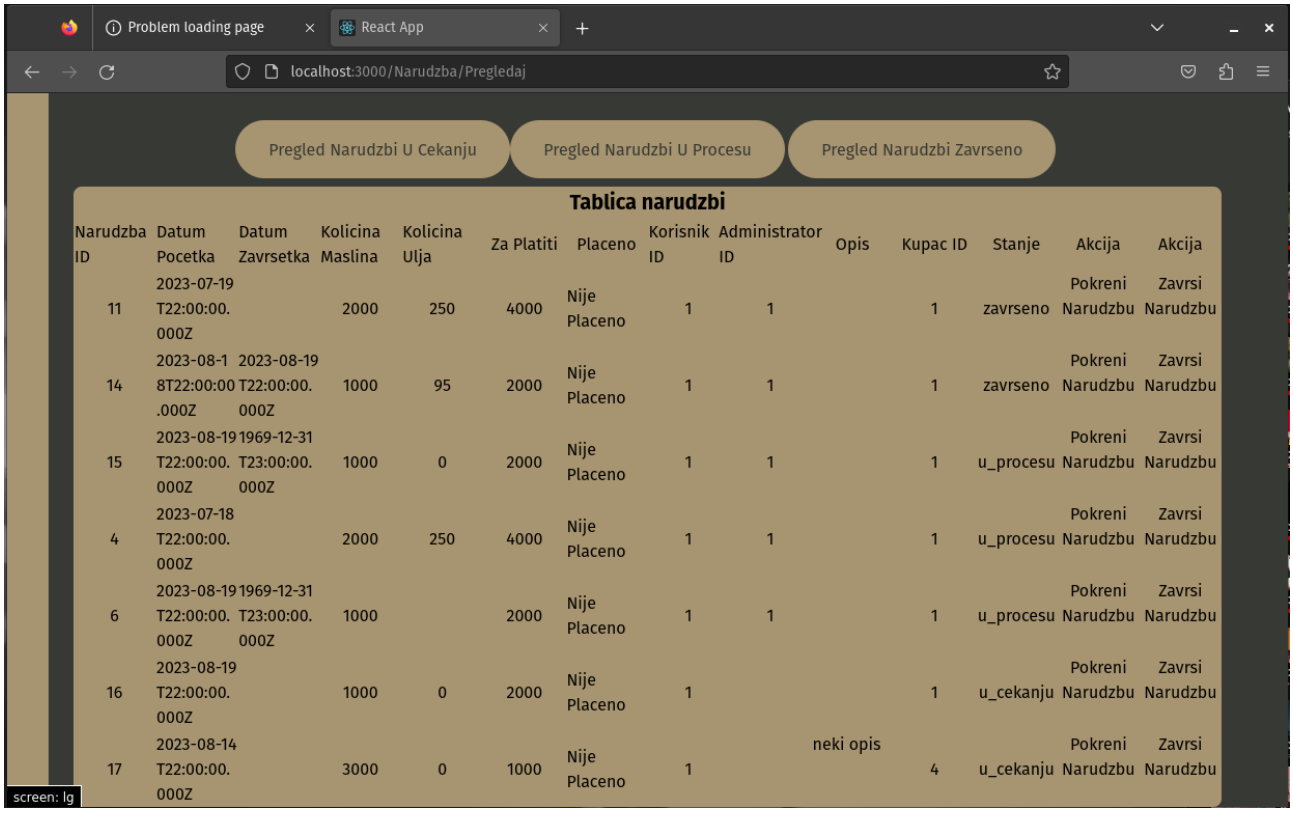
Primjer 62. Kod navigacijske trake

Navigacijska traka originalno je stisnuta na lijevoj strani ekrana te je njen sadržaj sakriven. Ako korisnik mišom prijeđe preko nje, ona se otvara i prikazuje sadržaj uz pomoć kojeg korisnik može pristupiti svim stranicama aplikacije. Detaljniji opis dizajna (boje, tranzicije, pozicioniranje elemenata...) aplikacije bit će pojašnjeni nakon opisa funkcionalnosti obiju aplikacija.

Na odabir "Pregledaj Narudzbu" korisnik prelazi na istoimenu stranicu. Na stranici se nalazi obrazac u koji korisnik može unijeti željeno ime korisnika preko kojeg se filtrira lista narudžbi te prikažu narudžbe napravljene od tog odabranog korisnika (Slika 23.).

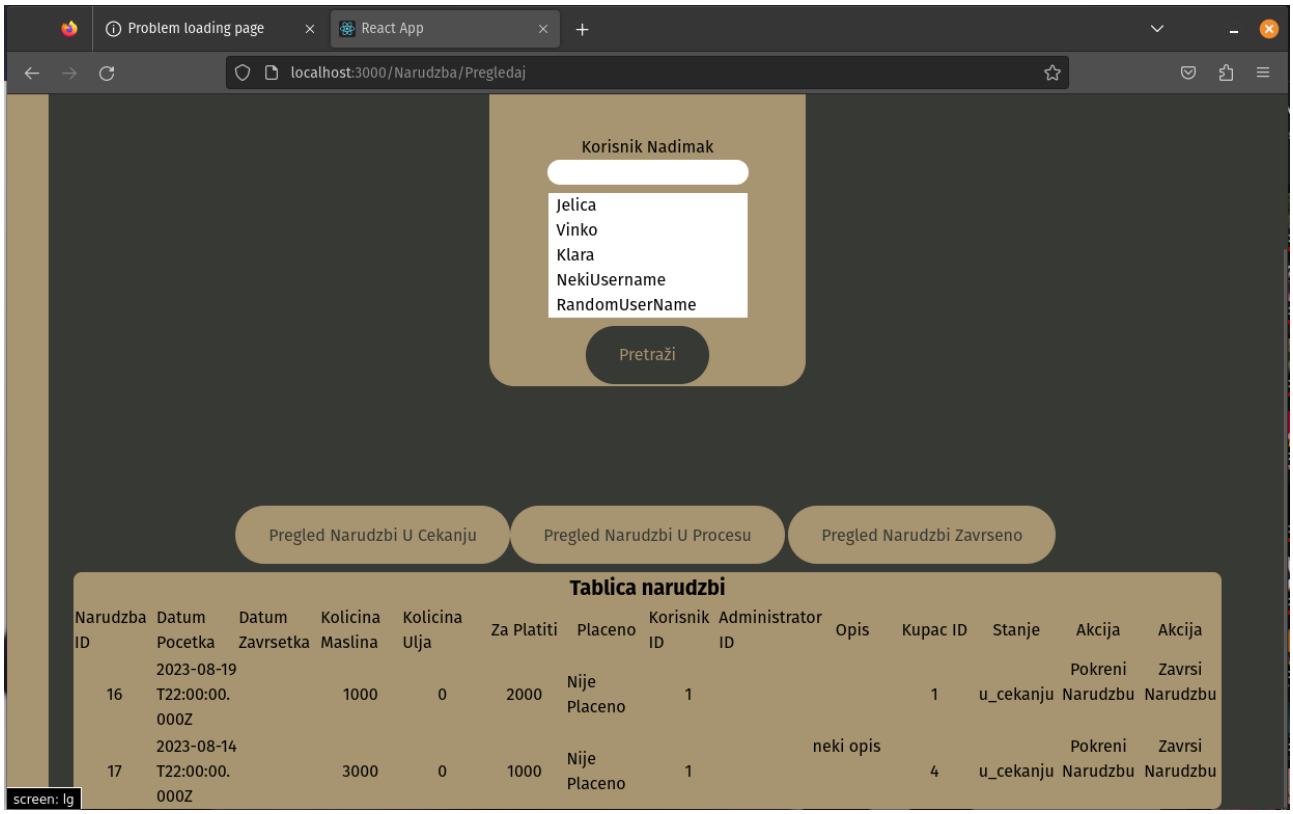


Slika 22. Unos korisnika



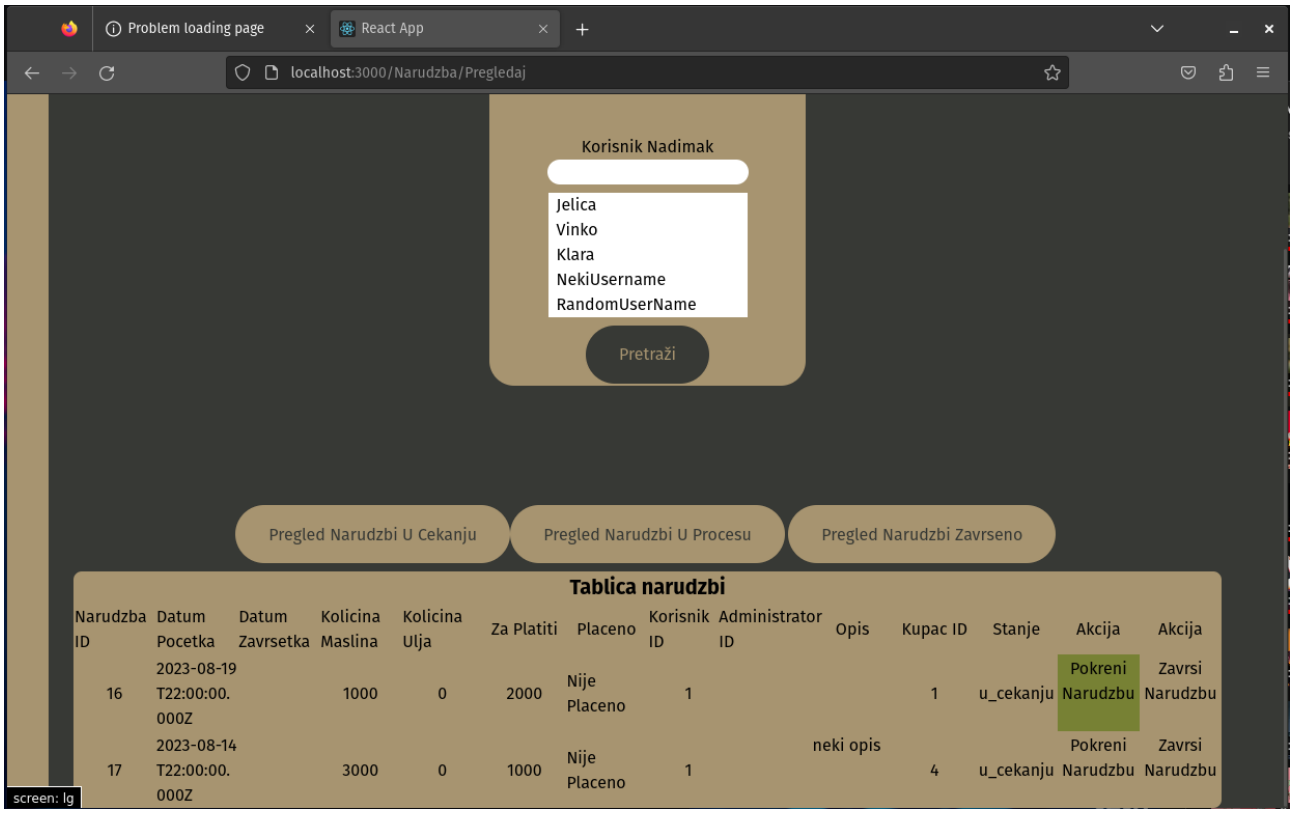
Slika 23. Cijeli rezultat pregleda narudžbi po korisniku

Korisnik također može filtrirati narudžbe po njihovom stanju (u čekanju, u procesu, završeno). Za pokrenuti narudžbu korisnik traži narudžbe u čekanju (Slika 24.).



Slika 24. Rezultat pregleda narudžbi u čekanju

Korisnik odabire jednu od narudžbi i klikne na gumb "pokreni narudžbu" koji odgovara odabranoj narudžbi (Slika 25.).



Slika 25. Korisnik odabire narudžbu

Nakon odabira narudžbe stanje te narudžbe automatski je promijenjeno u stanje "u procesu" (Primjer 63.).

```
router.put('/startNarudzba', async (req, res) => {
      try {
            let queryResult = await database.query('UPDATE "Narudzba"
            SET "stanje"=$1 WHERE "narudzba_id"=$2', ["u_procesu",
            req.body.narudzba_id]);
      if (!queryResult) {
            throw new Error("Narudzba nije zapoceta!");
      }
      else{
            return res.json({ results: "Narudzba uspjesno
            zapoceta!" });
      }
      } catch (error) {
            console.log(error);
            return res.json({ results: "Narudzba nije zapoceta!" });
      }
})
```

Primjer 63. Kod poslužitelja za pokretanje narudžbe

Sljedeći kod (Primjer 64.) prikazuje funkciju koja se pokreće na klik gumba "Pokreni Narudzbu". Funkcija se nalazi u komponenti "NarudzbaTable", ali se pokreće kroz komponentu "NarudzbaRow" koja se nalazi u komponenti "NarudzbaTable".

```
export const NarudzbaTable = ({listaNarudzbi}) => {
const startNarudzba = (event, narudzba) => {
      fetch('http://localhost:8080/narudzba/startNarudzba',{
      method: 'PUT',
      headers: {
            'Content-Type': 'application/json',
      },
      body: JSON.stringify({
            narudzba_id: narudzba.narudzba_id,
            stanje: narudzba.stanje,
      })
      }).then(response => response.json())
      .then(data => {
            console.log('Start Narudzba Response:', data);
      })
}
return (
<table className="grid mx-16 rounded-lg place-content-center bg-
secondary">
      <th className="text-xl">Tablica narudzbi</th>
      <tr className="grid content-center grid-cols-14">
            <td className="flex self-center justify-center">Narudzba
            ID</td>
            <td className="flex self-center justify-center">Datum
            Pocetka</td>
            <td className="flex self-center justify-center">Datum
            Zavrsetka</td>
            <td className="flex self-center justify-center">Kolicina
            Maslina</td>
            <td className="flex self-center justify-center">Kolicina
            Ulja</td>
            <td className="flex self-center justify-center">Za
            Platiti</td>
            <td className="flex self-center justify-
            center">Placeno</td>
            <td className="flex self-center justify-center">Korisnik
            ID</td>
            <td className="flex self-center justify-
            center">Administrator ID</td>
            <td className="flex self-center justify-center">Opis</td>
            <td className="flex self-center justify-center">Kupac
            ID</td>
            <td className="flex self-center
            justify-center">Stanje</td>
```

```
            <td className="flex self-center
            justify-center">Akcija</td>
            <td className="flex self-center
            justify-center">Akcija</td>
      </tr>
      {listaNarudzbi.map((narudzba) => (
            <NarudzbaRow narudzba={narudzba}
            rememberAndCarryNarudzba={rememberAndCarryNarudzba}
            startNarudzba={startNarudzba}/>
            ))}
</table>
      )
}
```

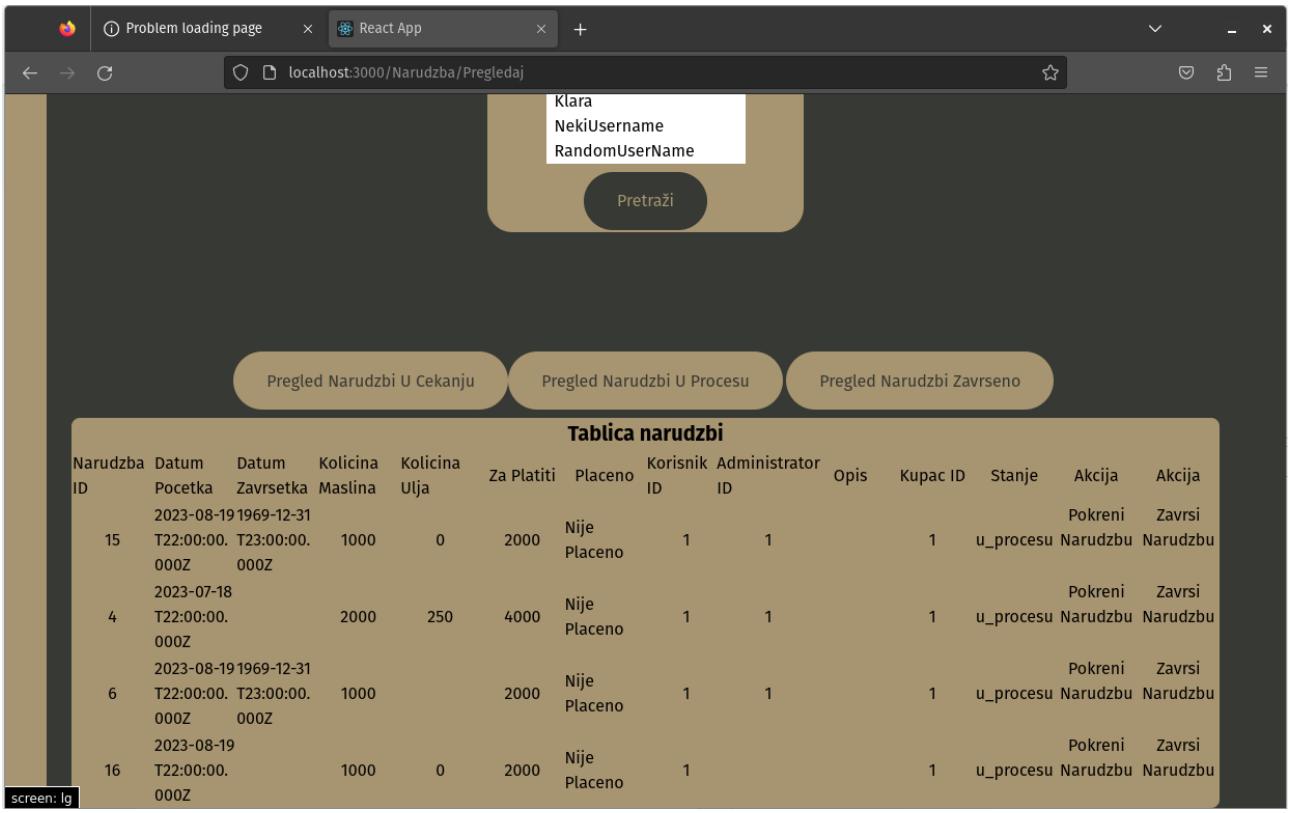
Primjer 64. Kod komponente "NarudzbaTable"

```
export default function NarudzbaRow({narudzba,
rememberAndCarryNarudzba, startNarudzba}) {
const { vrsta_korisnika_id } = useSelector((state) => state.logIn);

console.log("Vrsta Korisnika:", vrsta_korisnika_id);

const [imeKupca, setImeKupca] = useState("");

const getKupacImeByNarudzba = () => {
fetch('http://localhost:8080/kupac/getKupacByID', {
      method: 'POST',
      headers: {
            'Content-Type': 'application/json',
      },
      body: JSON.stringify({
            kupac_id: narudzba.kupac_id
            })
      }).then(response => response.json())
      .then(data => {
            console.log(data[0].naziv_kupca);
            setImeKupca(data[0].naziv_kupca);
            console.log(imeKupca);
      });
}

useEffect(getKupacImeByNarudzba, [narudzba]);
return (
<tr className='grid content-center grid-cols-14'>
      <td className='flex self-center justify-center break-
      words'>{narudzba.narudzba_id}</td>
      <td className='break-words'>{narudzba.datum_pocetka}</td>
      <td className='break-words'>{narudzba.datum_zavrsetka}</td>
```

```
        <td className='flex self-center justify-center break-
        words'>{narudzba.kolicina_maslina}</td>
        <td className='flex self-center justify-center break-
        words'>{narudzba.kolicina_ulja}</td>
        <td className='flex self-center justify-center break-
        words'>{narudzba.za_platiti}</td>
        {narudzba.placeno === true ? <td className='flex self-center
        justify-center break-words'>Placeno</td> : <td className='flex
        self-center justify-center break-words'>Nije Placeno</td>}
        <td className='flex self-center justify-center break-
        words'>{narudzba.korisnik_id}</td>
        <td className='flex self-center justify-center break-
        words'>{narudzba.administrator_id}</td>
        <td className='break-words'>{narudzba.opis}</td>
        <td className='flex self-center justify-center break-words'
        value={narudzba.kupac_id}>{imeKupca}</td>
        <td className='flex self-center justify-center break-
        words'>{narudzba.stanje}</td>
        <td className='hover:bg-tertiary'><button onClick={(event) =>
        startNarudzba(event, narudzba)}>Pokreni Narudzbu</button></td>
        { vrsta_korisnika_id === 1 && <td className='hover:bg-
        tertiary'><button onClick={(event) =>
        rememberAndCarryNarudzba(event, narudzba)}>Zavrsi
        Narudzbu</button></td>}
    </tr>
    )
    }
```

Primjer 65. Kod komponente "NarudzbaRow"

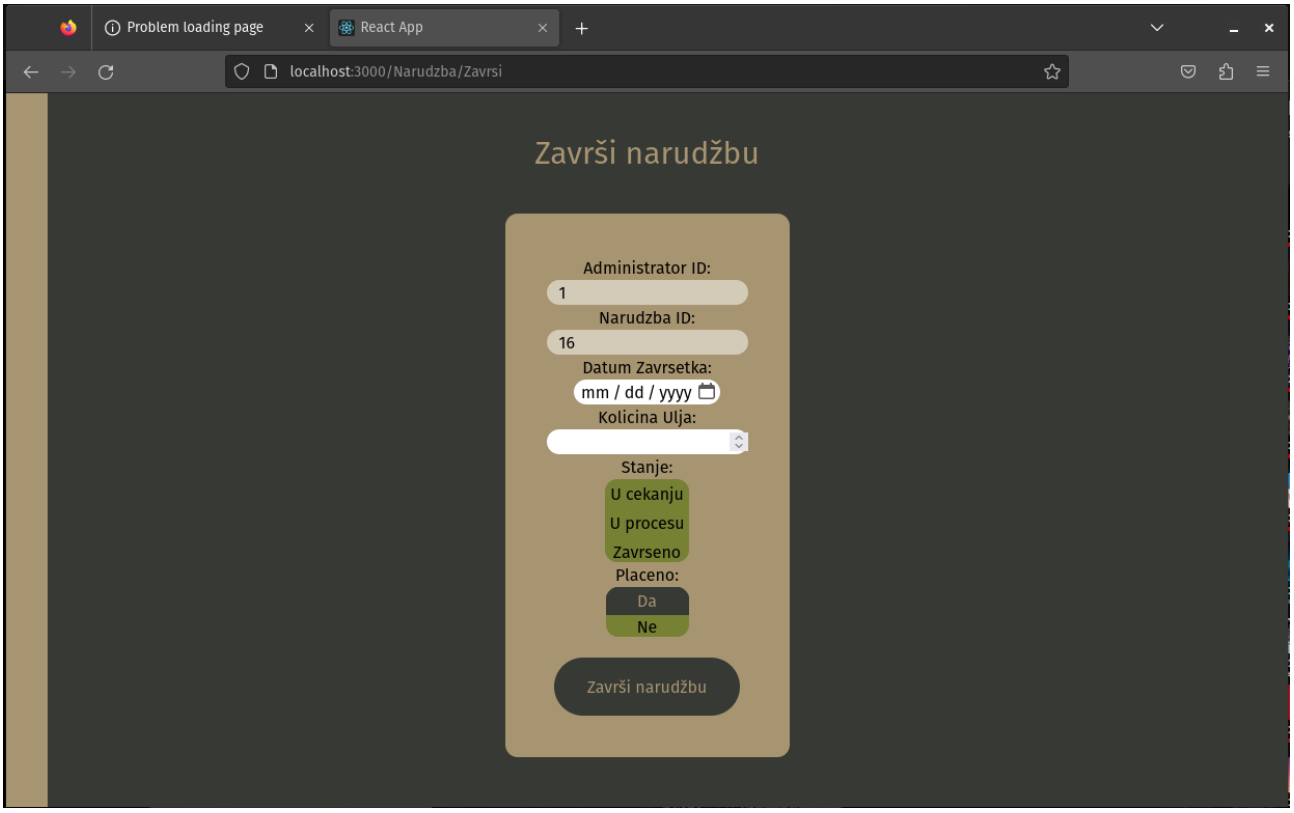
Pokretanje tog gumba radi se uz pomoć rekvizita, specifično rekvizita "startNarudzba". Taj rekvizit je u komponenti "NarudzbaRow", ali se prenosi kroz "NarudzbaTable" prema "NarudzbaRow" s funkcijom "startNarudzba()" koja se pokreće na klik gumba "Pokreni Narudzbu". Na ovaj način u samo par klikova korisnik je pokrenuo narudžbu te se rezultat može vidjeti klikom na gumb "Pregled Narudzbi U Procesu" (Slika 26.).



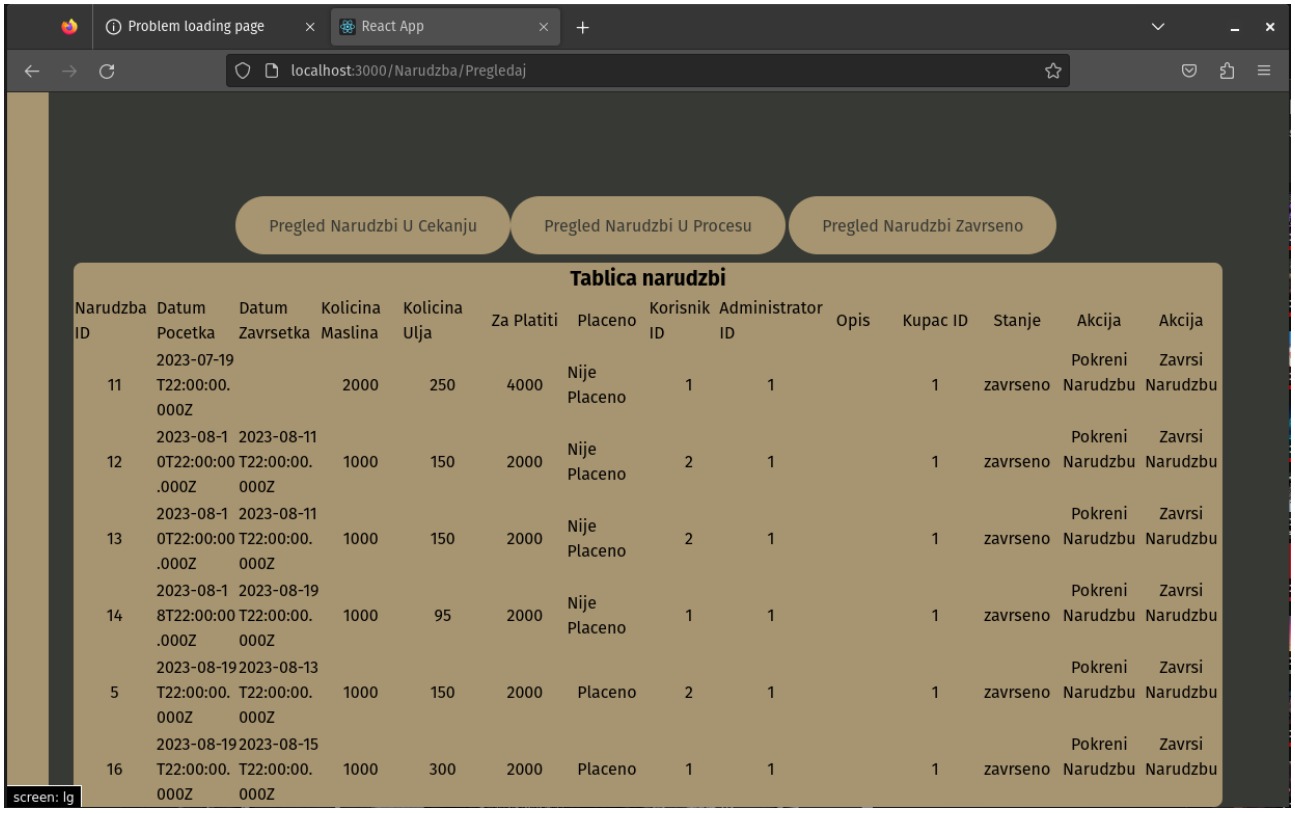
Slika 26. Pregled narudžbi u procesu

Administrator, kada je završen proces prerade, pristupa preko navigacijske stranice pregledu narudžbi te traži određenu narudžbu tako da pregledava listu narudžbi u procesu (Slika 26.).

Zatim odabire gumb "Zavrsi Narudzbu" te se otvara stranica završavanja narudžbe u kojoj administrator unosi potrebne podatke te se klikom na gumb "Završi Narudžbu" svi uneseni podaci spremaju i narudžba je gotova (Slika 27.). Na slici se može vidjeti da su neki podaci već automatski uneseni te postavljeni da korisnik ne može utjecati na njihovu vrijednost, a to su "Administrator ID" i "Narudzba ID". Kada je narudžba gotova, administrator može pogledati sve završene narudžbe na stranici pregleda narudžbi te vidjeti da je narudžba gotova sa svim podacima i stanjem "završeno" (Slika 28.). Zadnja završena narudžba nalazi se na dnu tablice.



Slika 27. Završavanje narudžbe



Slika 28. Pregled završenih narudžbi

Sljedeći je kod koji prikazuje stranicu završavanja narudžbe (Primjer 66.). Kod koristi spremište narudžbe kako bi mogao automatski popuniti dijelove obrasca završavanja narudžbe. Funkcija spremišta nalazi se u stranici pregleda narudžbi te se pokreće na klik gumba "Zavrsi Narudzbu" (Primjer 67.).

```
export default function ZavrsiNarudzbu (){
const { korisnik_id_logIn, vrsta_korisnika_id } = useSelector((state)
=> state.logIn);
const { narudzba_id } = useSelector((state) => state.narudzba);
const endNarudzba = (e) => {
      e.preventDefault();
      fetch("http://localhost:8080/narudzba/endNarudzba",
      { method: "PUT",
      headers: {
            "Content-Type": "application/json",
      },
      body: JSON.stringify({
            narudzba_id: narudzba_id,
            stanje: document.getElementById("stanje").value,
            datum_zavrsetka:
            document.getElementById("datum_zavrsetka").value,
            kolicina_ulja:
            document.getElementById("kolicina_ulja").value,
            placeno: document.getElementById("placeno").value,
            administrator_id: korisnik_id_logIn
      }),
      }).then((response) => response.json())
      .then((response) => {
            console.log(response);
      })
}
return(
<div className="bg-primary">
<NavBar />
<div className="grid grid-cols-1 place-items-center">
<h1 className="mt-10 text-3xl text-secondary">Završi narudžbu</h1>
<form className="grid grid-cols-1 p-10 mt-10 rounded-xl place-
content-center place-items-center bg-secondary"
onSubmit={endNarudzba}>
      <label>Administrator ID: </label>
      <input className="rounded-full indent-3" disabled={true}
      id="administrator_id" name="administrator_id"
      value={korisnik_id_logIn}/>
      <label>Narudzba ID: </label>
      <input className="rounded-full indent-3" disabled={true}
      id="narudzba_id" name="narudzba_id" value={narudzba_id}/>
      <label>Datum Zavrsetka: </label>
      <input className="rounded-full indent-1" type="date"
      id="datum_zavrsetka" name="datum_zavrsetka"/>
      <label>Kolicina Ulja: </label>
```

```
      <input className="rounded-full indent-3" type="number"
      id="kolicina_ulja" name="kolicina_ulja" />
      <label>Stanje: </label>
      <select className="h-20 overflow-visible appearance-none
      focus:outline-none rounded-xl bg-tertiary" multiple id="stanje"
      name="stanje">
            <option className="flex justify-center h-7 rounded-t-xl
            place-items-center checked:bg-primary checked:text-
            secondary checked:border-2 checked:border-primary
            checked:ring-secondary" value="u_cekanju">U
            cekanju</option>
            <option className="flex justify-center h-7 place-items-
            center checked:bg-primary checked:text-secondary
            checked:border-2 checked:border-primary checked:ring-
            secondary" value="u_procesu">U procesu</option>
            <option className="flex justify-center h-7 rounded-b-xl
            place-items-center checked:bg-primary checked:text-
            secondary checked:border-2 checked:border-primary
            checked:ring-secondary"
            value="zavrseno">Zavrseno</option>
      </select>
      <label>Placeno: </label>
      <select className="w-20 h-12 overflow-visible appearance-none
      focus:outline-none rounded-xl bg-tertiary" multiple
      id="placeno" name="placeno">
            <option className="flex justify-center checked:bg-primary
            checked:text-secondary checked:border-2 checked:border-
            primary checked:ring-secondary" value="true"
            selected={true}>Da</option>
            <option className="flex justify-center checked:bg-primary
            checked:text-secondary checked:border-2 checked:border-
            primary checked:ring-secondary" value="false">Ne</option>
      </select>
      <button className="px-8 py-4 mt-5 rounded-full bottom-5
      hover:ring-primary hover:ring-2 bg-primary text-secondary
      hover:bg-tertiary hover:text-primary" type="submit">Završi
      narudžbu</button>
</form>
</div>
</div>
)
}
```

Primjer 66. Kod završavanja narudžbe

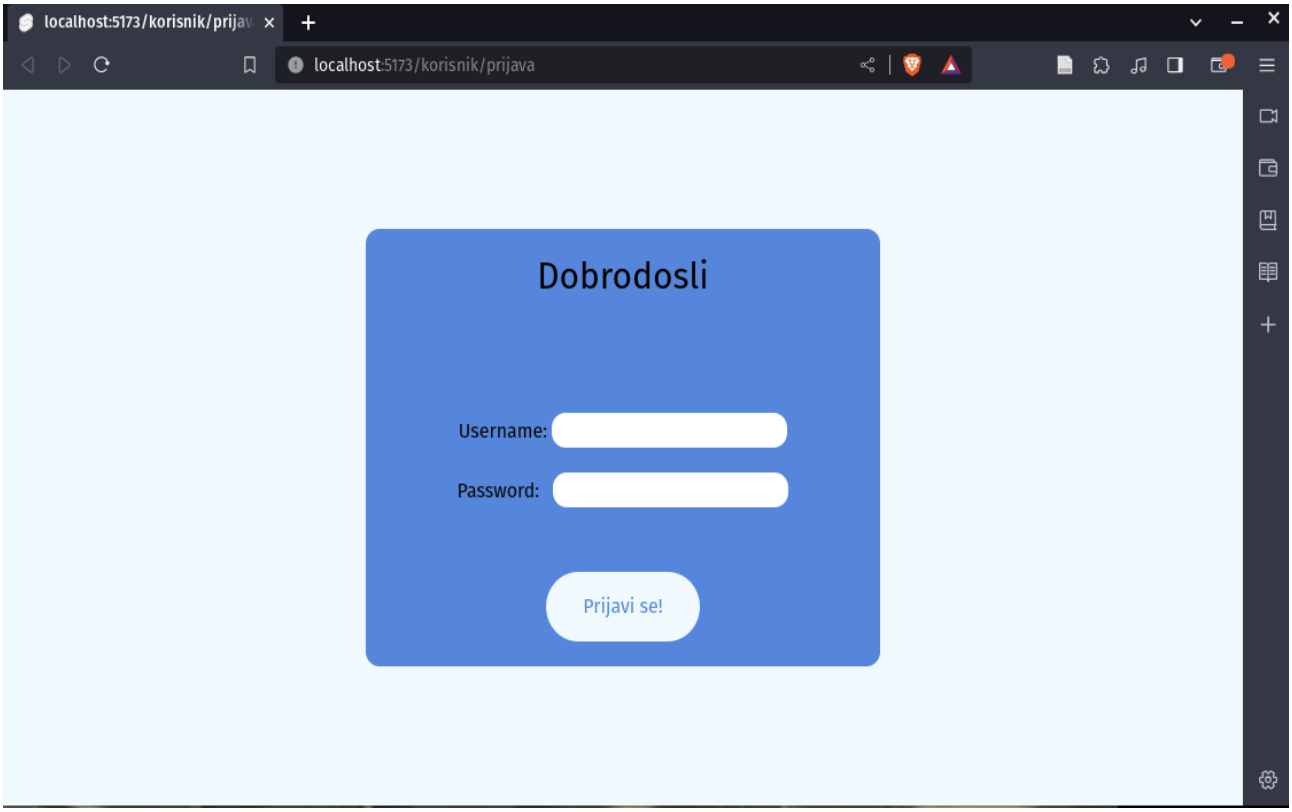
```
async function endNarudzba(narudzba_id, stanje, datum_zavrsetka,
kolicina_ulja, administrator_id, placeno){
      let queryResult = await database.query('UPDATE "Narudzba" SET
      "stanje"=$1, "datum_zavrsetka"=$2, "kolicina_ulja"=$3,
      "administrator_id"=$4, "placeno"=$5 WHERE "narudzba_id"=$6',
```

```
        [stanje, datum_zavrsetka, kolicina_ulja, administrator_id,
        placeno,narudzba_id]);
        return queryResult;
}

router.put('/endNarudzba', async (req, res) => {
        try {
                let queryResult = await endNarudzba(req.body.narudzba_id,
                req.body.stanje, req.body.datum_zavrsetka,
                req.body.kolicina_ulja, req.body.administrator_id,
                req.body.placeno);
        if (!queryResult) {
                throw new Error("Narudzba nije zavrsena!");
        }
        else{
                return res.json({ results: "Narudzba uspjesno
                zavrsena!" });
                }
        } catch (error) {
        console.log(error);
                return res.json({ results: "Narudzba nije zavrsena!" });
        }
})
```

Primjer 67. Kod poslužitelja završavanja narudžbe

8.5. Druga aplikacija

Druga aplikacija napravljena je s aplikacijskim okvirima Svelte i NestJS. Svelte se koristio za stvaranje korisničke strane, a Nest za stvaranje poslužiteljeve strane. Uz Svelte koristio se i SvelteKit, ali minimalno jer je Svelte fokus ovog završnog rada. React i Svelte pokreću mrežne aplikacije isključivo na korisničkoj strani dok SvelteKit na poslužiteljevoj, ali za naknadnu navigaciju kroz aplikaciju SvelteKit pokreće stranice na korisničkoj strani. Zbog toga je na početku *web*-pregledniku lakše pokrenuti aplikaciju s obzirom na to da od SvelteKita dobiva već "pripremljenu" aplikaciju (ili dio aplikacije). la chona



Slika 29. Prijava na Svelte aplikaciji

Pokretanje Svelte/SvelteKit aplikacije vrši se naredbom "npm run dev" dok se pokretanje NestJS poslužitelja vrši naredbom "npm run start:dev". Kao i u prvoj aplikaciji otvara se stranica prijave korisnika na početku pokretanja aplikacije (Slika 29.).

U ovom poglavlju opisat će se proces prerade kao i opis prve aplikacije. Svelte i Nest zajedno će se prikazivati kod opisa funkcija aplikacije.

Sljedeći primjer je primjer koda prijave aplikacije (Primjer 68.). Može se primijetiti da su JavaScript i HTML odvojeni. Svelte također koristi spremišta kao i React, ali za razliku od Reacta ne koristi Redux te je mogućnost korištenja spremišta direktno postavljena u Svelte.

```
async function onSubmit(e){
      const formData = new FormData(e.target);
      const endpoint =
      'http://localhost:3000/korisnik/prijava/logIn';

      const data = {};
      for (let field of formData) {
            const [key, value] = field;
            data[key] = value;
      }
      const response = await fetch(endpoint, {
      method: 'POST',
      headers: {
```

```
            'Content-Type': 'application/json'
        },
        body: JSON.stringify(data)
        })

        const responseJSONData = await response.json();

        prijavljenKorisnikID.set(responseJSONData.korisnik_id);
        prijavljenKorisnikUsername.set(responseJSONData.username);
        vrstaPrijavljenogKorisnika.set(responseJSONData.vrsta_korisnika
        _id);
if (responseJSONData != null) {
        goto('/korisnik/pregledKorisnika');
} else {
        alert('Pogrešno korisničko ime ili lozinka!');
        }
}
</script>

<div class="grid w-screen h-screen bg-primary place-items-center">
<form autocomplete="off" class="relative grid grid-cols-1 p-20 py-32
space-y-5 rounded-xl bg-secondary place-items-center " action=""
on:submit|preventDefault={onSubmit}>
        <h1 class="absolute text-3xl top-5 ">Dobrodosli</h1>
        <div>
            <label for="">Username: </label>
            <input class="h-7 indent-3 rounded-xl" type="text"
            name="username" id="username">
        </div>
        <div>
            <label for="">Password:</label>
            <input class="ml-2 indent-3 h-7 rounded-xl"
            type="password" name="password" id="password">
        </div>
        <button class="absolute px-8 py-4 rounded-full bottom-5
        hover:ring-primary hover:ring-2 bg-primary text-secondary
        hover:bg-tertiary hover:text-primary" type="submit">Prijavi se!
        </button>
</form>
</div>
```

Primjer 68. Svelte kod za prijavu

U primjeru (Primjer 68.) funkcija "onSubmit()" čita podatke iz obrasca na unos obrasca, sortira ih te šalje na putanju za prijavu korisnika koja aktivira Nestovu funkciju prijave (Primjer 69.) (Primjer 70.).

```
import { KorisnikService } from './korisnik.service';
import { Korisnik } from './korisnik.entity';
```

```
@Controller('/korisnik')
export class KorisnikController {
	@Post('/prijava/logIn')
	async KorisnikLogIn(
		@Res() res,
		@Body() logInKorisnikDto: LogInKorisnikDto,
	): Promise<Korisnik> {
	const korisnik = await
		this.korisnikService.getKorisnikByUsernamePassword(
		logInKorisnikDto.username,
		logInKorisnikDto.password,
	);
		return res.status(200).json(korisnik);
	}
}
```

Primjer 69. Nest funkcija u upravljaču za prijavu

```
import { KorisnikRepo } from './korisnik.repository';
import { Korisnik } from './korisnik.entity';

@Injectable()
export class KorisnikService {
	constructor(
	@InjectRepository(Korisnik)
	private readonly korisnikRepository: KorisnikRepo,
	) {}
async getKorisnikByUsernamePassword(
	username: string,
	password: string,
): Promise<Korisnik> {
	return await this.korisnikRepository.findOneBy({
		username: username,
		password: password,
		});
	}
}
```

Primjer 70. Nest funkcija u servisu za prijavu

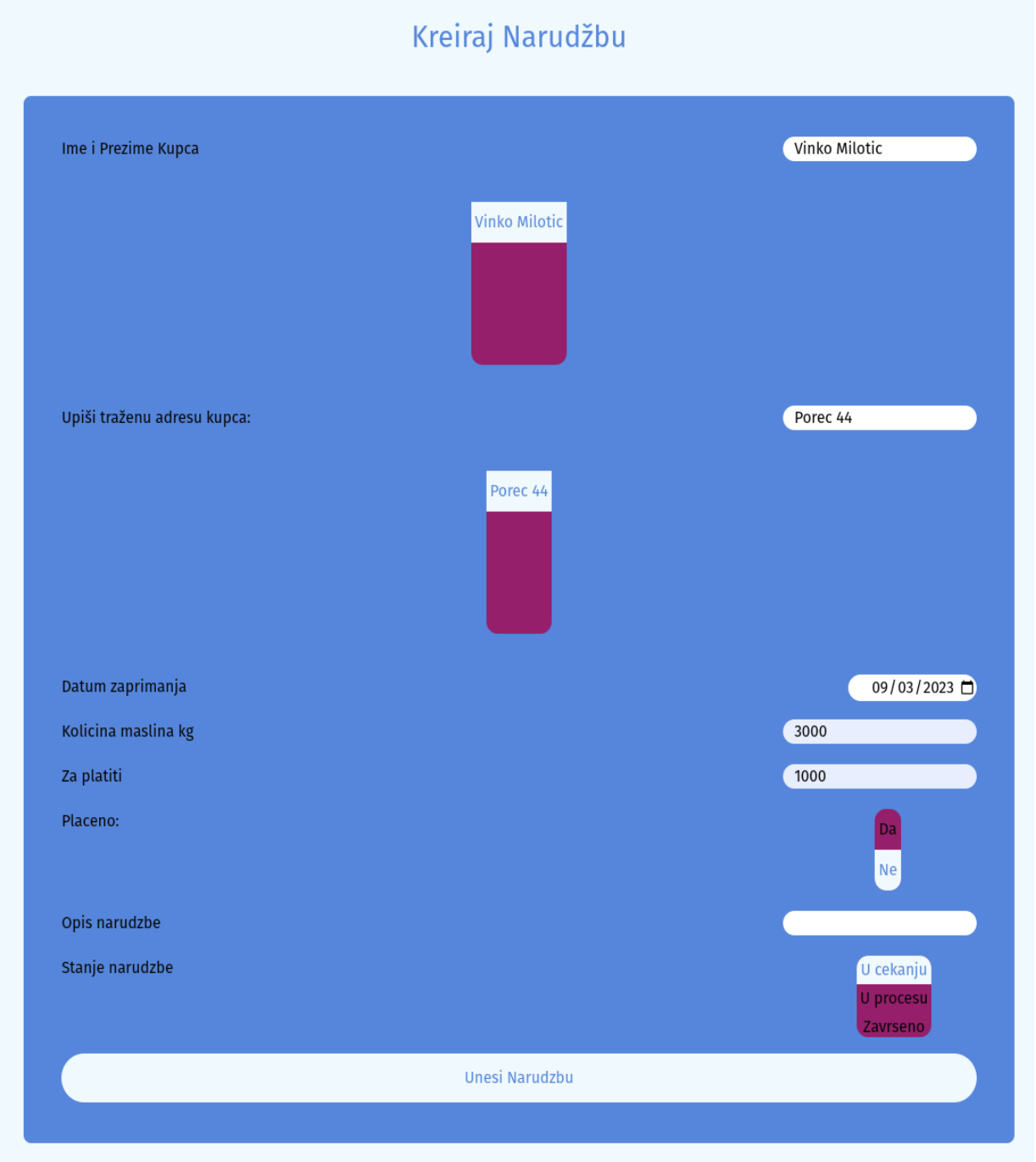
U kodu upravljača i servisa može se primijetiti korištenje ubacivanja ovisnosti. U servis se ubacuje repozitorij korisnika, a u upravljač se ubacuje servis. Repozitorij dozvoljava korištenje predefiniranih funkcija upita prema bazi podataka; time nije potrebno pisati SQL unutar JavaScript-a (u ovom slučaju TypeScripta jer Nest se bazira na TypeScriptu). Sve funkcije su obećanja te je specificirana vrsta podataka očekivana u odgovoru od baze

podataka. U ovom slučaju tip podatka je "Korisnik" koji je entitet u Nestu koji reprezentira tablicu "Korisnik" u bazi podataka (Primjer 71.).

```
import { Narudzba } from 'src/narudzba/narudzba.entity';
import { VrstaKorisnika } from
'src/vrstaKorisnika/vrstaKorisnika.entity';

@Entity('Korisnik')
export class Korisnik extends BaseEntity {
      @PrimaryGeneratedColumn()
      korisnik_id: number;

      @Column({ unique: true, nullable: false })
      username: string;

      @Column({ nullable: false })
      vrsta_korisnika_id: number;

      @ManyToOne(() => VrstaKorisnika)
      @JoinColumn({
      name: 'vrsta_korisnika_id',
      referencedColumnName: 'vrsta_korisnika_id',
      })
      vrstaKorisnika: VrstaKorisnika;

      @Column({ nullable: false })
      password: string;

      @OneToMany(() => Narudzba, (narudzba) => narudzba.korisnik)
      narudzba: Narudzba[];

      @OneToMany(() => Narudzba, (narudzba) =>
      narudzba.administrator)
      narudzbaAdmin: Narudzba[];
}
```

Primjer 71. Entitet korisnika u Nestu

Nakon uspješne prijave korisnik ulazi u aplikaciju. Nakon ulaza odabire stranicu unosa narudžbe kao i u prvoj aplikaciji preko navigacijske trake. Na ulazu u stranicu korisnik vidi obrazac za unos narudžbe (Slika 30.).



Slika 30. Obrazac unosa narudžbe

Svelte nakon unosa narudžbe šalje podatke u JSON obliku Nest poslužitelju koji provjerava ispravnost dobivenih podataka uz pomoć DTO-a za unos narudžbe te ako su podaci točni, spremaju se u bazu podataka (Primjer 72.) (Primjer 73.) (Primjer 74.). Također i u ovoj aplikaciji postoji provjera postojanja kupca. Ako kupac ne postoji, korisnik ga prvo mora registrirati u aplikaciju.

```
function filtrirajKupce() {
```

```
	let nazivKupca =
	document.getElementById('imePrezimeTextBox').value;
	listaFiltriranihKupaca.length = 0;
	listaSvihKupaca.forEach(kupac => {
	if (kupac.naziv_kupca.includes(nazivKupca)) {
		listaFiltriranihKupaca.push(kupac);
		}
	});
}

async function getKupacIDByNameAndAdresa(naziv_kupca, adresa){
await fetch('http://localhost:3000/kupac/byNameAndAdresa',{
	method: 'POST',
	headers: {'Content-Type': 'application/json'},
	body: JSON.stringify({
		naziv_kupca: naziv_kupca,
		adresa: adresa
	})
	}).then(response => response.json())
	.then(data => {
		console.log(data);
		kupacID = 0;
		kupacID = data[0].kupac_id;
		})
}

function changeName(event){
	let nazivKupca = event.target.getAttribute('value');
	document.getElementById('imePrezimeTextBox').value =
		nazivKupca;
	fetch('http://localhost:3000/kupac/adresaByName',{
	method: 'POST',
	headers: {
		'Content-Type': 'application/json'
	},
	body: JSON.stringify({
		naziv_kupca: nazivKupca
		})
	})
	.then(response => response.json())
	.then(data => {
		listaAdresa.length = 0;
		data.forEach(adresa => {
			listaAdresa.push(adresa);
		});
		console.log(listaAdresa);
	})
}

function changeAdresa(event){
```

```
      let adresa = event.target.getAttribute('value');
      document.getElementById('adresaTxtBox').value = adresa;
      let nazivKupca = event.target.getAttribute('value');
      getKupacIDByNameAndAdresa(nazivKupca, adresa);
}

async function unesiNarudzbu(){
      console.log( document.getElementById('datum_pocetka').valueAsDa
      te,
      document.getElementById('kolicina_maslina').valueAsNumber,
      document.getElementById('za_platiti').valueAsNumber,
      document.getElementById('placeno').value,
      idKorisnika,
      document.getElementById('opis').value,
      kupacID,
      document.getElementById('stanje').value);
      await fetch('http://localhost:3000/narudzba/create',{
      method: 'POST',
      headers: {'Content-Type': 'application/json'},
      body: JSON.stringify({
      datum_pocetka:
      document.getElementById('datum_pocetka').valueAsDate,
      kolicina_maslina:
      document.getElementById('kolicina_maslina').valueAsNumber,
      za_platiti:
      document.getElementById('za_platiti').valueAsNumber,
      placeno: document.getElementById('placeno').value,
      korisnik_id: idKorisnika,
      opis: document.getElementById('opis').value,
      kupac_id: kupacID,
      stanje: document.getElementById('stanje').value
})
}).then(response => response.json())
}
<NavBar />
<div class="grid space-y-10 bg-primary place-items-center">
<h1 class="flex justify-center mt-10 text-3xl text-secondary">Kreiraj
Narudžbu</h1>
<form class="grid w-5/12 grid-cols-1 gap-5 p-10 rounded-lg bg-
secondary" action="" on:submit|preventDefault={unesiNarudzbu}>
      <div class="relative w-full">
            <label for="">Ime i Prezime Kupca</label>
            <input class="absolute right-0 rounded-full indent-3"
            type="text" on:input={filtrirajKupce}
            id="imePrezimeTextBox">
      </div>
      <div class="flex justify-center">
      <select class="mt-5 mb-5 overflow-visible appearance-none
      focus:outline-none rounded-xl bg-tertiary" multiple name=""
      id="imePrezimeSelect">
```

```
      {#each listaFiltriranihKupaca as kupac}
            <option class="flex justify-center h-10 place-items-
            center checked:bg-primary checked:text-secondary
            checked:border-2 checked:border-primary
            checked:ring-secondary" value="{kupac.naziv_kupca}"
            on:click={changeName}>{kupac.naziv_kupca}</option>
      {/each}
</select>
</div>
<div class="relative w-full">
      <label for="">Upiši traženu adresu kupca: </label>
      <input class="absolute right-0 rounded-full indent-3"
      type="text" name="adresa" id="adresaTxtBox"
      on:input={filtrirajKupce}>
</div>
<div class="flex justify-center">
<select class="mt-5 mb-5 overflow-visible appearance-none h-46
focus:outline-none rounded-xl bg-tertiary" multiple name=""
id="">
      {#each listaAdresa as adresa}
            <option class="flex justify-center h-10 place-items-
            center checked:bg-primary checked:text-secondary
            checked:border-2 checked:border-primary
            checked:ring-secondary" value="{adresa.adresa}"
            on:click={changeAdresa}>{adresa.adresa}</option>
      {/each}
</select>
</div>
<div class="relative w-full">
      <label for="">Datum zaprimanja</label>
      <input class="absolute right-0 rounded-full indent-3"
      type="Date" id="datum_pocetka">
</div>
<div class="relative w-full">
      <label for="">Kolicina maslina</label>
      <input class="absolute right-0 rounded-full indent-3"
      type="number" id="kolicina_maslina">
      <label for="">kg</label>
</div>
<div class="relative w-full">
      <label for="">Za platiti</label>
      <input id="za_platiti" class="absolute right-0 rounded-
      full indent-3" type="number">
</div>
<div class="relative w-full mb-14">
      <label for="">Placeno: </label>
      <select name="placeno" id="placeno" class="absolute h-20
      overflow-visible appearance-none right-20 focus:outline-
      none rounded-xl bg-tertiary" multiple>
```

```
                  <option class="flex justify-center h-10 rounded-t-xl
                  place-items-center checked:bg-primary checked:text-
                  secondary checked:border-2 checked:border-primary
                  checked:ring-secondary" value="true">Da</option>
                  <option class="flex justify-center h-10 rounded-b-xl
                  place-items-center checked:bg-primary checked:text-
                  secondary checked:border-2 checked:border-primary
                  checked:ring-secondary" value="false">Ne</option>
            </select>
      </div>
      <div class="relative w-full">
            <label for="">Opis narudzbe</label>
            <input id="opis" class="absolute right-0 rounded-full
            indent-3" type="text">
      </div>
      <div class="relative w-full mb-10">
            <label for="">Stanje narudzbe</label>
            <select id="stanje" class="absolute h-20 overflow-visible
            appearance-none right-12 focus:outline-none rounded-xl
            bg-tertiary" multiple>
                  <option class="flex justify-center h-7 rounded-t-xl
                  place-items-center checked:bg-primary checked:text-
                  secondary checked:border-2 checked:border-primary
                  checked:ring-secondary" value={Stanje.U_CEKANJU}>U
                  cekanju</option>
                  <option class="flex justify-center h-7 place-items-
                  center checked:bg-primary checked:text-secondary
                  checked:border-2 checked:border-primary
                  checked:ring-secondary" value={Stanje.U_PROCESU}>U
                  procesu</option>
                  <option class="flex justify-center h-7 rounded-b-xl
                  place-items-center checked:bg-primary checked:text-
                  secondary checked:border-2 checked:border-primary
                  checked:ring-secondary"
                  value={Stanje.ZAVRSENO}>Zavrseno</option>
            </select>
      </div>
      <button class="px-6 py-3 mt-3 rounded-full bottom-5 hover:ring-
      primary hover:ring-2 bg-primary text-secondary hover:bg-
      tertiary hover:text-primary" type="submit">Unesi
      Narudzbu</button>
</form>
</div>
```

Primjer 72. Kod unosa naruždbe i provjere kupca

```
@Post('/create')
async createFirstNarudzba(
```

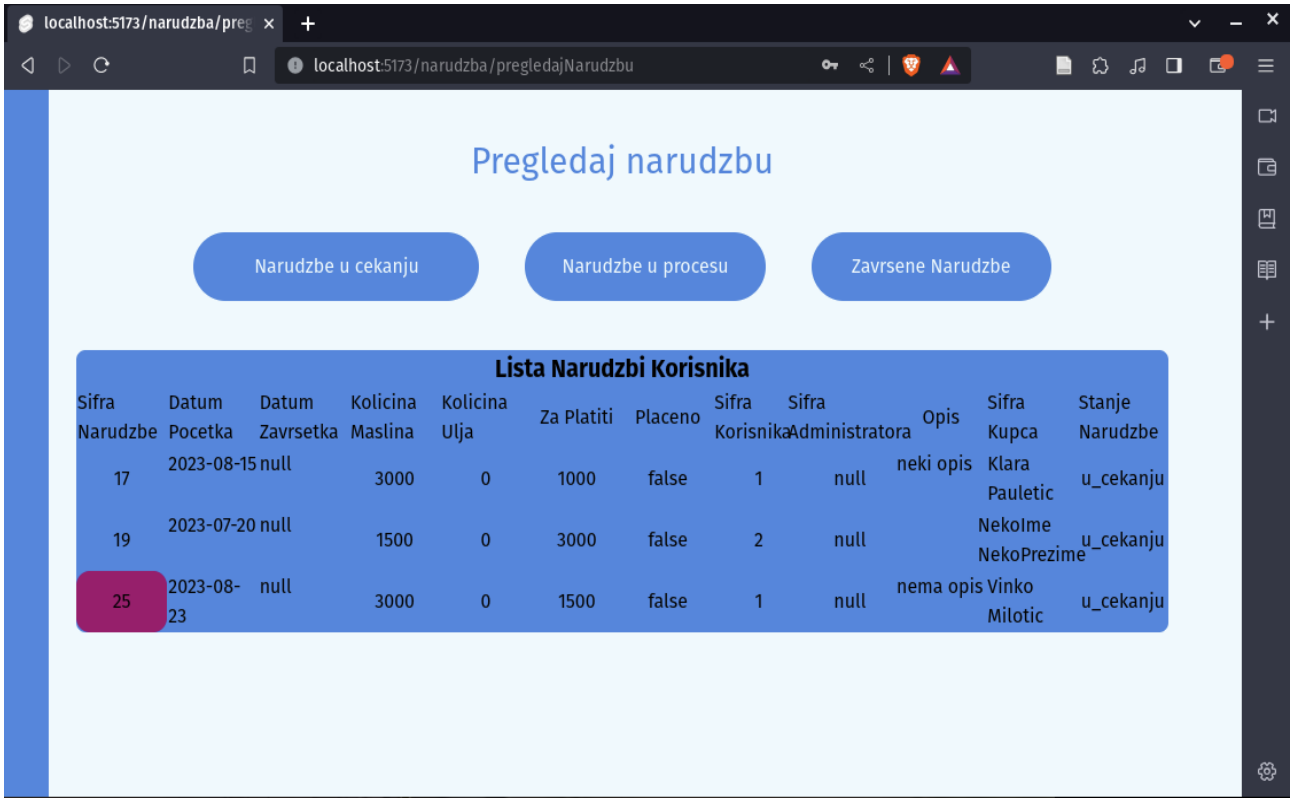
```
      @Res() res,
      @Body() firstNarudzbaData: CreateNarudzbaDto,
) {
      const narudzba = await this.narudzbaService.createNarudzba(
            firstNarudzbaData,
      );
      return res.status(201).json({
            message: 'Narudzba uspješno kreirana',
            narudzba,
      });
}
```

Primjer 73. Nest unos narudžbe u upravljaču

```
export class CreateNarudzbaDto {
      @IsNotEmpty({ message: 'Narudzba mora imati datum pocetka' })
      datum_pocetka: Date;

      @IsNotEmpty({   message:   'Narudzba   mora   imati   kolicinu
      maslina' })
      kolicina_maslina: number;

      @IsNotEmpty({   message:   'Narudzba   mora   imati   cijenu   za
      platiti' })
      za_platiti: number;

      @IsNotEmpty({
      message: 'Narudzba mora imati vrijednost istinitosti placene
      narudzbe',
      })
      placeno: boolean;

      @IsNotEmpty({
      message: 'Narudzba mora imati sifru korisnika koji je zaprimio
      narudzbu',
      })
      korisnik_id: number;

      opis: string;

      @IsNotEmpty({ message: 'Narudzba mora imati sifru kupca' })
      kupac_id: number;

      @IsNotEmpty({ message: 'Narudzba mora imati stanje' })
      stanje: Stanje;
}
```

Primjer 74. Nest DTO za unos narudžbe

Korisnik nakon unosa narudžbe provjerava je li narudžba uspješno napravljena tako da pregledava listu narudžbi u čekanju. Kako bi korisnik pokrenuo narudžbu, treba kliknuti na broj šifre narudžbe koju želi pokrenuti.

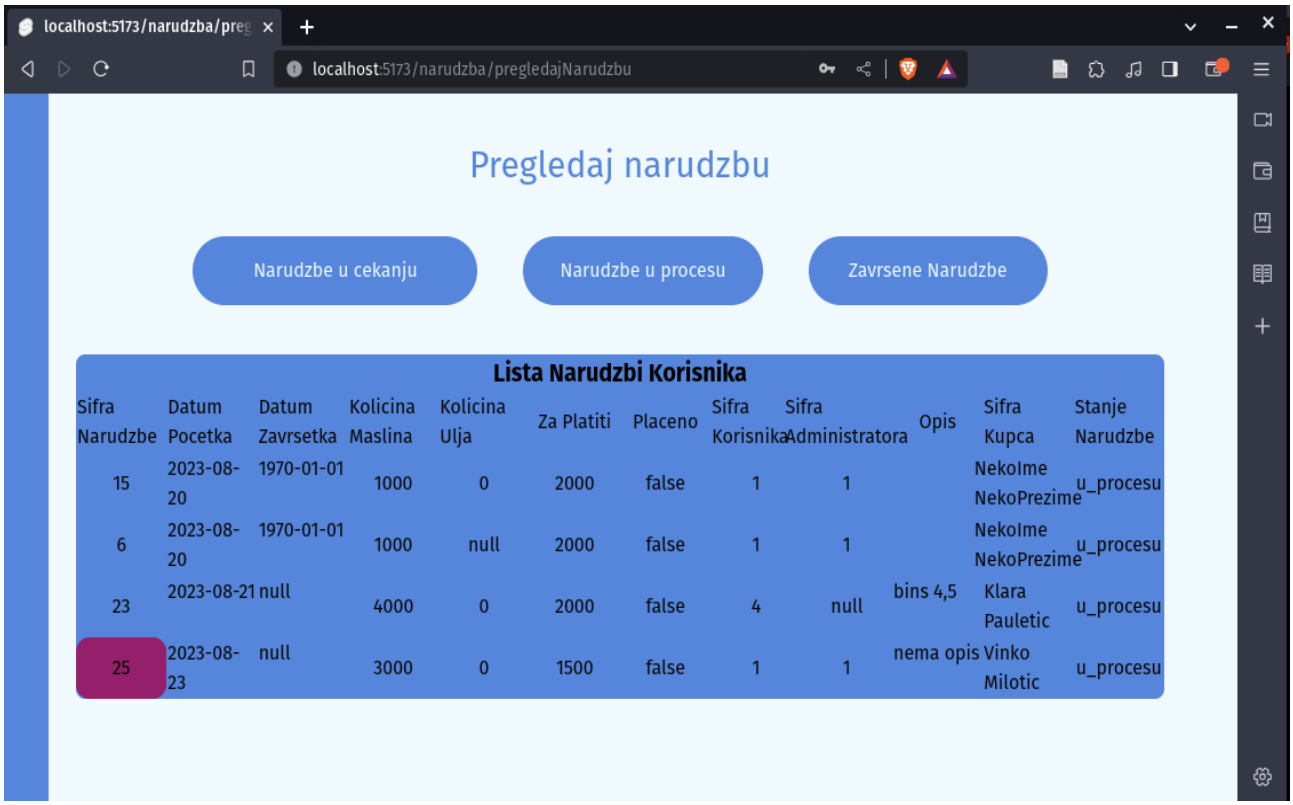


Slika 31. Ažuriranje narudžbe

Nakon odabira aplikacija korisnika prebacuje na stranicu ažuriranja narudžbe u kojoj korisnik unosi potrebne podatke za pokretanje narudžbe.

Slika 32. Obrazac ažuriranja narudžbe

Pokrenuta narudžba može se vidjeti u listi nardužbi sa stanjem "u procesu" (Slika 33.) te kada je gotova na isti se način (preko klika na šifru narudžbe) završava narudžba (Slika 31.). Aplikacija prebacuje korisnika na stranicu ažuriranja u kojoj korisnik unosi potrebne podatke za završetak narudžbe. Svaki zahtjev koji ima podatke u tijelu u Nestu provjerava se i obrađuje preko DTO-a. Ako je zahtjev tipa "GET", znači da zahtjev nema tijelo te DTO u tom slučaju nije potreban (Primjer 75.) (Primjer 76.).



Slika 33. Narudžba u procesu

Slika 34. Završena narudžba

```
import { IsNotEmpty } from 'class-validator';
import { Stanje } from 'src/enum/stanje';


export class UpdateNarudzbaDto {
      @IsNotEmpty({ message: 'Narudzba mora imati sifru' })
      narudzba_id: number;primjer Nest sa DTO

      @IsNotEmpty({ message: 'Narudzba mora imati datum pocetka' })
      datum_pocetka: Date;

      @IsNotEmpty({ message: 'Narudzba mora imati datum zavrsetka' })
      datum_zavrsetka: Date;

      @IsNotEmpty({ message: 'Narudzba mora imati kolicinu
      maslina' })
      kolicina_maslina: number;

      @IsNotEmpty({ message: 'Narudzba mora imati kolicinu ulja' })
      kolicina_ulja: number;

      @IsNotEmpty({ message: 'Narudzba mora imati cijenu za
      platiti' })
      za_platiti: number;

      @IsNotEmpty({
      message: 'Narudzba mora imati vrijednost istinitosti placene
      narudzbe',
      })
      placeno: boolean;

      @IsNotEmpty({
      message: 'Narudzba mora imati sifru korisnika koji je zaprimio
      narudzbu',
      })
      korisnik_id: number;

      @IsNotEmpty({
      message:
      'Narudzba mora imati sifru administratora koji je zavrsio
      narudzbu',
      })
      administrator_id: number;

      opis: string;

      @IsNotEmpty({ message: 'Narudzba mora imati sifru kupca' })
      kupac_id: number;

      @IsNotEmpty({ message: 'Narudzba mora imati stanje' })
```

```
	stanje: Stanje;
}
```

Primjer 75. DTO za ažuriranje narudžbe

```
@Get('/all')
async getAllNarudzba(): Promise<Narudzba[]> {
	return await this.narudzbaService.getAllNarudzba();
}
```

Primjer 76. Funkcija upravljača bez DTO-a

```
@Put('/update')
async updateNarudzba(
	@Res() res,
	@Body() updateNarudzbaData: UpdateNarudzbaDto,
): Promise<Narudzba> {
const narudzba = await this.narudzbaService.updateNarudzba(
	updateNarudzbaData,
	);
return res.status(201).json({
	message: 'Narudzba uspješno ažurirana',
	narudzba,
	});
}
```

Primjer 77. Funkcija ažuriranja narudžbe u upravljaču

8.6. Dizajn aplikacija

Dizajniranje aplikacija bilo je gotovo pa identično, tj. metode i naredbe koje su se koristile u procesu dizajniranja aplikacija bile su skoro iste za obje aplikacije. Jedan od mogućih razloga je što React i Svelte promoviraju arhitekturu koda baziranu na komponente te s obziorm na to da su komponente u objema aplikacijama relativno slične (ako ne i iste u nekim dijelovima aplikacije).

```
<ul class='fixed left-0 grid float-left w-10 h-screen grid-cols-1
hover:w-auto group place-items-stretch justify-stretch bg-secondary'>
	<li class='flex items-center justify-center invisible w-full
	group-hover:visible hover:bg-tertiary'><a
	href="/korisnik/pregledKorisnika">Pregledaj Korisnika</a></li>
	<li class='flex items-center justify-center invisible w-full
	group-hover:visible hover:bg-tertiary'><a
	href="/korisnik/unosKorisnika">Kreiraj Korisnika</a></li>
```

```
      <li class='flex items-center justify-center invisible w-full
      group-hover:visible hover:bg-tertiary'><a
      href="/kupac/pregledKupca">Pregledaj Kupca</a></li>
      <li class='flex items-center justify-center invisible w-full
      group-hover:visible hover:bg-tertiary'><a
      href="/kupac/registracijaKupca">Kreiraj Kupca</a></li>
      <li class='flex items-center justify-center invisible w-full
      group-hover:visible hover:bg-tertiary'><a
      href="/narudzba/pregledajNarudzbu">Pregledaj Narudzbu</a></li>
      <li class='flex items-center justify-center invisible w-full
      group-hover:visible hover:bg-tertiary'><a
      href="/narudzba/azurirajNarudzbu">Azuriraj Narudzbu</a></li>
      <li class='flex items-center justify-center invisible w-full
      group-hover:visible hover:bg-tertiary'><a
      href="/narudzba/unesiNarudzbu">Kreiraj Narudzbu</a></li>
      <li class='flex items-center justify-center invisible w-full
      group-hover:visible hover:bg-quaternary'><a
      href="/korisnik/prijava">Odjava</a></li>
</ul>
```

Primjer 78. Kod navigacijske trake u aplikaciji sa Svelte

```
<ul className='fixed left-0 grid float-left w-10 h-screen grid-cols-1
hover:shadow-2xl hover:shadow-tertiary hover:w-auto group place-
items-stretch justify-stretch bg-secondary'>
      <li className='flex items-center justify-center invisible w-
      full group-hover:visible hover:bg-tertiary'><Link
      to="/Korisnik/Pregledaj">Pregledaj Korisnika</Link></li>
      <li className='flex items-center justify-center invisible w-
      full group-hover:visible hover:bg-tertiary'><Link
      to="/Korisnik/Kreiraj">Kreiraj Korisnika</Link></li>
      <li className='flex items-center justify-center invisible w-
      full group-hover:visible hover:bg-tertiary'><Link
      to="/Kupac/Pregledaj">Pregledaj Kupca</Link></li>
      <li className='flex items-center justify-center invisible w-
      full group-hover:visible hover:bg-tertiary'><Link
      to="/Kupac/Kreiraj">Kreiraj Kupca</Link></li>
      <li className='flex items-center justify-center invisible w-
      full group-hover:visible hover:bg-tertiary'><Link
      to="/Narudzba/Pregledaj">Pregledaj Narudzbu</Link></li>
      <li className='flex items-center justify-center invisible w-
      full group-hover:visible hover:bg-tertiary'><Link
      to="/Narudzba/Zavrsi">Zavrsi Narudzbu</Link></li>
      <li className='flex items-center justify-center invisible w-
      full group-hover:visible hover:bg-tertiary'><Link
      to="/Narudzba/Kreiraj">Kreiraj Narudzbu</Link></li>
```

```
      <li className='flex items-center justify-center invisible w-
      full group-hover:visible hover:bg-quaternary'><Link
      to="/">Odjava</Link></li>
</ul>
```

Primjer 79. Kod navigacijske trake u aplikaciji s React

U primjerima iznad (Primjer 78.) (Primjer 79.) može se primijetiti da za napraviti jednostavni izgled jedne komponente u aplikaciji s TailwindCSS-om kod jako brzo postane "nepotrebno" velik i težak za čitati. Ovi primjeri sadržavaju prilagođenost ekranima samo jedne veličine. Da su primjeri bili prilagođeni za više različitih veličina ekrana, kod bi bio skoro pa nečitljiv. To se smatra najvećom manom TailwindCSS-a. Srećom, kako bi se izbjegao takav kod, Tailwind nudi mogućnost postavljanja ponavljajućih dijelova koda u klase koje se mogu višekratno koristiti po potrebi, ali i ta mogućnost ima negativnu stranu, a to je apstrakcija koda što programeru može znatno otežati dizajniranje ako on nije kreator tih klasa te ne zna što te klase rade u suštini.

S obzirom na to da je Tailwind okvir, on ima neke limitacije u svojim naredbama npr. ako dizajner aplikacije želi postaviti visinu elementa s naredbom "h-13", to neće biti moguće jer "h-13" ne postoji u Tailwindu, ali zato postoje "h-12" i "h-14". Time dizajner aplikacije mora proširiti mogućnosti Tailwinda ručno u postavkama aplikacije ako želi koristiti "h-13". Takav je bio slučaj tijekom dizajniranja tablice narudžbi u aplikaciji s Reactom. U Tailwindu tablice (ili drugi elementi) mogu maksimalno sadržavati 12 stupaca s naredbom "grid-cols-12", ali u slučaju ove aplikacije bilo je potrebno 14 stupaca te zbog toga u Tailwind postavkama aplikacije (datoteka tailwin.config.js) proširena je mogućnost postavljanja broja stupova tablice prikazanih na 14 stupaca (Primjer 80.).

```
/** @type {import('tailwindcss').Config} */

module.exports = {
content: [
      "./src/**/*.{js,jsx,ts,tsx}",
],

theme: {
extend: {
      colors: {
            primary: '#373935',
            secondary: '#A79470',
            tertiary: '#778134',
            quaternary: '#C98260'
      },
gridTemplateColumns: {
```

```
        '14': 'repeat(14, minmax(0, 1fr))',
              }
        },
  },
  plugins: [
        require('tailwindcss-debug-screens'),
        ],
  }
```

Primjer 80. Tailwind konfiguracije s proširenim mogućnostima

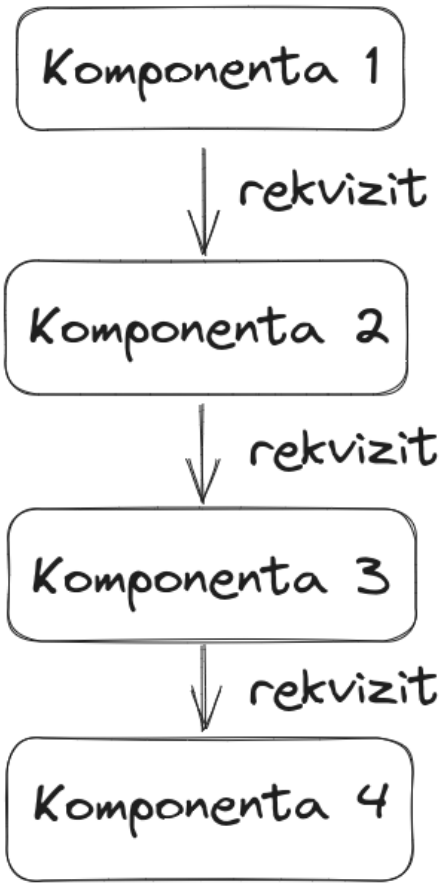
Uz to Tailwind nije sadržavao klase s bojama koje su se koristile u ovim aplikacijama te su zbog toga proširene klase boja s prilagođenim klasama koje sadržavaju boje potrebne za obje aplikacije završnog rada (Primjer 80.).

Iako je do sada Tailwind prikazan negativno, valja napomenuti da je proces dizajniranja s Tailwindom relativno lagan. Najveći faktor za takvo iskustvo jest što su klase u Tailwindu intuitivne te dokumentacija (od kreatora Tailwind okvira) istih je jako lagana za čitati i razumjeti. Također, još jedan faktor bio bi što se klase navode u samom imenu HTML elemenata što tijekom pregledavanja koda može olakšati predočavanje zašto aplikacija izgleda tako.

9. Usporedba Aplikacija

9.1. Usporedba okvira korisničke strane

React i Svelte relativno su slični što se tiče strukturiranja koda s obzirom na to da oba predlažu programeru komponentnu arhitekturu koda. Ključ njihove razlike je upravo to "predlaganje" programeru. React ne "predlaže" programeru način korištenja Reacta, već forsira određeni način korištenja. React ima dobro iskustvo programiranja u dvjema stvarima i to u komponentnom programiranju i rukovanjem stanjem u aplikaciji (većinom zbog Reduxa). Ako programer ima razumijevanje u tim stvarima, neće imati poteškoća s korištenjem Reacta, ali to inherentno postavlja određene granice Reactu, tj. React u nekim slučajevima korištenja neće biti dobar odabir za korištenje kao što su aplikacije za ekstremno velike kompanije u kojima će se morati raditi bušenje rekvizita (engl. *prop drilling*) što može biti jako naporno i teško ne samo za napraviti nego i održavati s obzirom na to da previše komponenti ovisi o toj jednoj komponenti na vrhu (Slika 33.).



Slika 35. Bušenje rekvizita

S druge strane Svelte se može smatrati alatom koji je dobar u svemu, ali najbolji ni u čemu. Sve što je potrebno za napraviti Svelte može raditi dobro, ali ne najbolje. Primjerice, tijekom izgradnje aplikacije jedina napravljena komponenta (u aplikaciji koja se radila sa Svelteom) bila je komponenta navigacijske trake jer kod je bio dovoljno razumljiv za čitanje u svim stranicama da apstrakcija s komponentama uglavnom nije bila potrebna, dok je u aplikaciji koja se radila s Reactom napravljeno 5 komponenata (isključujući komponente stranica). Zbog takvog iskustva Svelte je idealan za projekte malih ili srednjih veličina kao što je ovaj završni rad, ali kao i React, Svelte nije povoljan za izradu velikih projekata. Jedan od razloga bio bi to što je Svelte (kao i React) okvir za izradu jednostraničnih aplikacija (engl. *single-page applications*, SPA) koje mogu imati loše performanse s velikim aplikacijama jer klijentska strana (*web*-preglednik) treba cijelu aplikaciju sama izraditi i kontrolirati. Zbog toga je napravljen SvelteKit (koji se također koristi u ovom završnom radu, ali minimalno) kao nadogradnja na Svelte. SvelteKit radi vizualizaciju sa strane poslužitelja (engl. server-side rendering) što prebacuje jedan dio tereta s klijenta na poslužitelja te time ubrzava aplikaciju. SvelteKit zamagljuje razliku klijenta i poslužitelja što je dobra i ujedno loša stvar u programiranju. Dobra je jer aplikacije postaju brže, ali loša jer programerima postaje teško pratiti u velikim projektima što se izvodi na klijentu i na poslužitelju što može dovesti do nepravilnih opterećenja obiju strana aplikacije.

Unatoč njihovim razlikama Svelte i React su relativno slični te su jedni od najboljih aplikacijskih okvira za programiranje korisničkih strana aplikacija u svijetu. Oboje donose svoje metode programiranja za različite slučajeve korištenja te drastično olakšavaju proces održavanja i stvaranja modernih korisničkih strana aplikacija.

9.2. Usporedba okvira poslužiteljeve strane

Express i Nest slični su po načinu funkcioniranja te međusobno nemaju puno razlika. Oba koriste zadane putanje preko kojih se šalju zahtjevi i odgovori te oba mogu koristiti iste npm pakete namijenjene za Express po potrebi zbog toga što je Nest u suštini baziran na Expressu. Njihova razlika ima najveći naglasak kada projekt sve više raste. Express je namijenjen za projekte manje ili maksimalno srednje veličine. To se može primijetiti po načinu na koji se on koristi. Programer samo treba specificirati putanju i upit za manipulirati s određenom putanjom što je dobro za manje projekte jer oni nemaju puno putanja na koje se šalju zahtjevi. S druge strane, Nest je napravljen za projetke koji imaju velik tok podataka. To se također vidi po načinu korištenja Nesta. U Nestu sve mora biti dobro posloženo i odvojeno u posebne datoteke tako da se samo po potrebi koriste u specifičnim slučajevima korištenja, tj. sve mora biti dobro posloženo kako bi se efikasno koristilo ubacivanje ovisnosti. Za samo

jedan entitet u bazi podataka u Nestu se treba napraviti modul, upravljač, svi potrebni DTO-i, servis i drugi dobavljači te sam entitet sa svim potrebnim relacijama prema drugim entitetima itd. U početku kad se radio ovaj projekt Express je bio puno bolji od Nesta kao okvir za stvaranje poslužiteljske strane upravo zbog njegove jednostavnosti, ali što je bio veći projekt, to se Nest sve više pokazivao kao bolji kandidat. Iako je ovaj projekt relativno mali, Nest se svejedno pokazao kao bolji kandidat zbog jednog razloga, a to je sigurnost tipova (engl. *type safety*). Sigurnost tipova odnosi se na provjeru podataka koji dolaze iz zahtjeva i na provjeru podataka koji se šalju prema korisničkom dijelu aplikacije. Primjerice, kada dođe zahtjev, Nest uz pomoć TypeScripta provjerava jesu li svi dobiveni podaci dobrog tipa, tj. jesu li broj, objekt, znak, datum, vrijeme i slično. To je bitno jer baza podataka za određene podatke može primiti samo jedan tip podataka npr. u ovom projektu primarni ključ za sve tablice može biti samo pozitivni cijeli broj. U Expressu nema nikakvih provjera tipa podataka koji se šalju jer se koristi JavaScript umjesto TypeScripta, a JavaScript uz pomoć njegove riječi "let" može staviti u varijablu bilo kakav tip podatka te ako se ta varijabla promijeni negdje drugdje u kodu, to može dovesti do velikih problema za koje je teško pronaći riješenje.

10. Završetak

U početku stvaranja *web*-aplikacija koristili su se HTML, CSS i JavaScript bez ikakvih aplikacijskih okvira te su se s vremenom i zahtjevima korisnika razvili aplikacijski okviri. JavaScript je *de facto* postao *lingua franca* u svijetu razvijanja *web*-aplikacija te se s vremenom stvorilo mnogo aplikacijskih okvira koji se baziraju na njemu. JavaScript se u početku mogao koristiti samo za stvaranje korisničke strane aplikacije, ali kada je bio stvoren Node.js, počeo se koristiti i za stvaranje poslužiteljeve strane aplikacije. Nakon nekog vremena bio je stvoren TypeScript kako bi se garantirala sigurnost tipova podataka. U Meti (nekad Facebooku) bio je stvoren React za koji je postao najpopularniji JavaScript aplikacijski okvir za stvaranje korisničke strane *web*-aplikacija zbog njegove brzine i dobrog baratanja stanjima aplikacija. Nakon njega bio je stvoren Svelte aplikacijski okvir koji je u samo par godina postao jedan od najpopularnijih aplikacijskih okvira za stvaranje korisničke strane *web*-aplikacija, ali postao je najpopularniji u svijetu od strane programera zbog njegovog dobrog iskustva korištenja. Kako bi se olakšalo stvaranje manjih projekata, bio je napravljen Express koji je svojom jednostavnošću korištenja brzo zamijenio Node (iako je baziran na Nodeu) za stvaranje poslužiteljeve strane aplikacija. Nakon Expressa stvorio se Nest, koji u suštini koristi Express. On je napravljen za stvaranje velikih projekata koji zahtjevaju sigurnost tipa podataka jer Express nije bio povoljan za stvaranje istih. TailwindCSS bio je napravljen kako bi se olakšalo dizajniranje *web*-aplikacija. On je lagano apstrahirao CSS klase kako ne bi previše sakrio od dizajnera što se zbiva u pozadini, ali dovoljno da se može smatrati jednostavnijom verzijom CSS-a. Svi navedeni aplikacijski okviri imaju svoje prednosti i mane te ne postoji jedan aplikacijski okvir koji može obuhvatiti svaki minimalni aspekt programiranja i dizajniranja *web*-aplikacija kao što ne postoji jedan alat za izgradnju svakog minimalnog dijela svake kuće. Na kraju, programer je taj koji treba odlučiti koji će se alat trebati koristiti za njegov projekt.

Popis literature

[1] J. Duckett, *HTML & CSS: design and build websites*. Indianapolis, Indiana: John Wiley & Sons Inc, 2014.

[2] S. Bradley, *CSS animations and transitions for the modern Web*. United States? Peachpit Press, 2015.

[3] '@keyframes - CSS: Cascading Style Sheets | MDN', Feb. 21, 2023. https://developer.mozilla.org/en-US/docs/Web/CSS/@keyframes (accessed May 16, 2023).

[4] A. Rauschmayer, *JavaScript for impatient programmers*. s.l.: s.n., 2019.

[5] 'Functions - JavaScript | MDN', Apr. 23, 2023. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Functions (accessed May 16, 2023).

[6] S. Fenton, *Pro TypeScript*. Berkeley, CA: Apress, 2018. doi: 10.1007/978-1-4842-3249-1.

[7] D. Riehle, *Framework Design: A Role Modeling Approach.*, vol. 20. 2000.

[8] 'Jack Pritchard - Swansea - WhatJackHasMade'. https://whatjackhasmade.co.uk/component-driven-development/ (accessed May 16, 2023).

[9] 'Introduction to the DOM - Web APIs | MDN', Apr. 15, 2023. https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model/Introduction (accessed May 16, 2023).

[10] 'Writing Markup with JSX – React'. https://react.dev/learn/writing-markup-with-jsx (accessed May 16, 2023).

[11] 'React Hooks Fundamentals for Beginners', *freeCodeCamp.org*, Mar. 15, 2022. https://www.freecodecamp.org/news/react-hooks-fundamentals/ (accessed May 16, 2023).

[12] J. Morgan, *How To Code in React.js*.

[13] F. Copes, *The Svelte Handbook*. Accessed: May 16, 2023. [Online]. Available: https://flaviocopes.com/book/svelte/

[14] N. Rappin, *Modern CSS with Tailwind*.

[15] 'Tailwind CSS - Rapidly build modern websites without ever leaving your HTML.', Nov. 15, 2020. https://tailwindcss.com/ (accessed Aug. 05, 2023).

[16] J. Wexler, *Get Programming with Node.js*.

[17] F. Copes, *The Express Handbook*.

[18] B. Smith, *Beginning JSON*. Berkeley, CA: Apress, 2015. doi: 10.1007/978-1-4842-0202-9.

[19] ‘Cross-Origin Resource Sharing (CORS) - HTTP | MDN’, Aug. 10, 2023. https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS (accessed Aug. 16, 2023).

[20] J. Bell, G. Magolan, D. Guijarro, A. de Peretti, and P. Housley, *Nest.js: A Progressive Node.js Framework*. Bleeding Edge Press, 2018.

[21] ‘Documentation | NestJS - A progressive Node.js framework’, *Documentation | NestJS - A progressive Node.js framework*. https://docs.nestjs.com (accessed Aug. 03, 2023).

[22] ‘Handbook - The TypeScript Handbook’. https://www.typescriptlang.org/docs/handbook/intro.html (accessed Aug. 06, 2023).

[23] Pavel Luzanov, Egor Rogov, Igor Levshin (translated by Liudmila Mantrova), *Postgres: The First Experience*. Accessed: Aug. 16, 2023. [Online]. Available: https://postgrespro.com/community/books/introbook

[24] I. Gelman and B. Dinkevich, *The Complete Redux Book*.

Popis slika

Slika 1. Razlika tranzicija i animacija [2, str. 94]

Slika 2. TypeScript je nadogradnja JavaScripta [autorski rad]

Slika 3. Formula deklariranja varijabli u TypeScriptu [autorski rad]

Slika 4. Formula deklariranja funkcija u TypeScriptu [autorski rad]

Slika 5. Funkcija koja koristi funkcije [11]

Slika 6. Kompleksan Tailwind kod (https://www.aleksandrhovhannisyan.com/blog/why-i-dont-like-tailwind-css/)

Slika 7. Pojednostavljeni primjer komunikacije korisnika i poslužitelja [autorski rad]

Slika 8. Kompleksniji prikaz komunikacije korisnika i poslužitelja [autorski rad]

Slika 9. Vizualni prikaz dekoratora [autorski rad]

Slika 10. ERA dijagram baze podataka [autorski rad]

Slika 11. Flux graf komunikacije [24, str. 7]

Slika 12. Redux komunikacija [autorski rad]

Slika 13. Prikaz tablica baze podataka

Slika 14. Dijagram slučaja korištenja uljare i aplikacija

Slika 15. Dijagram aktivnosti prijave i zaprimanja narudžbe

Slika 16. Dijagram aktivnosti pokretanja narudžbe

Slika 17. Dijagram aktivnosti završavanja narudžbe

Slika 18. Stranica za prijavu korisnika

Slika 19. Stranica kreacije narudžbe

Slika 20. Stranica kreacije narudžbe s predloženim imenom

Slika 21. Navigacijska traka

Slika 22. Unos korisnika

Slika 23. Cijeli rezultat pregleda narudžbi po korisniku

Slika 24. Rezultat pregleda narudžbi u čekanju

Slika 25. Korisnik odabire narudžbu

Slika 26. Pregled narudžbi u procesu

Slika 27. Završavanje narudžbe

Slika 28. Pregled završenih narudžbi

Slika 29. Prijava na Svelte aplikaciji

Slika 30. Obrazac unosa narudžbe

Slika 31. Ažuriranje narudžbe

Slika 32. Obrazac ažuriranja narudžbe

Slika 33. Narudžba u procesu

Slika 34. Završena narudžba

Slika 35. Bušenje rekvizita