

**Šolski center Novo mesto**

**Srednja elektro šola in tehniška gimnazija**

**Šegova ulica 112**

**8000 Novo mesto**

**Seminarska naloga pri predmetu računalništvo na splošni maturi**

**Aplikacije in informacijski sistemi**

**Informacijski sistem podjetja**

Avtor: Erik Radovičevič

Mentor: dr. Albert Zorko

Novo mesto, april 2023



## **Povzetek**

Seminarska naloga je dokumentacija za izdelek pri izpitu iz računalništva na splošni maturi. Obsega opis orodij in postopkov za izdelavo aplikacije, modeliranje, samo zgradbo aplikacije in testiranje. Uporabljena orodja so na kratko opisana z vidika, ki bi bil čim bolj razumljiv nekomu, ki se z njimi še ni seznanil, saj sem tudi jaz z opisom spoznal orodja, ki sem jih nato uporabil pri izdelavi. Pri modeliranju so izdelani in razloženi trije diagrami, ki opisujejo podatkovno bazo in zgradbo aplikacije. V opisu aplikacije je predstavljena zgradba iz vidika glavnih funkcij, pri čemer sem navedel primere programske kode iz aplikacije le v tolikšni meri, da sem lahko razložil koncepte, ki sem jih uporabil. Pri testiranju, kjer sem preizkusil delovanje programa, sem odkril tudi nekaj napak in program popravil, da je deloval tako, kot želimo. Na koncu naloge je zahvala vsem, ki so pripomogli k izdelavi te seminarske naloge.

### **Ključne besede:**

- Spletna aplikacija
- React.js
- Node.js
- Podatkovna baza MySQL
- Diagram
- Odjemalec in strežnik



# KAZALO VSEBINE

<b>1</b>	<b>UVOD .....</b>	<b>1</b>
<b>2</b>	<b>ORODJA IN POSTOPKI .....</b>	<b>2</b>
2.1	SPLETNO PROGRAMIRANJE .....	2
2.2	HTML .....	2
2.3	CSS .....	3
2.4	JAVASCRIPT .....	5
2.5	REACT.JS .....	5
2.5.1	<i>Prednosti</i> .....	6
2.5.2	<i>Delovanje</i> .....	6
2.5.3	<i>Fleksibilnost</i> .....	7
2.6	NODE.JS .....	7
2.6.1	<i>Delovanje</i> .....	7
2.6.2	<i>Razširjenost</i> .....	7
2.7	EXPRESS.JS .....	8
2.8	AXIOS .....	8
2.9	MYSQL .....	9
2.9.1	<i>SQL</i> .....	9
2.9.2	<i>Prednosti</i> .....	9
2.10	ASINHRONO PROGRAMIRANJE .....	9
<b>3</b>	<b>MODELIRANJE .....</b>	<b>12</b>
3.1	ER DIAGRAM .....	12
3.2	DIAGRAM TOKA PODATKOV .....	14
3.3	DIAGRAM ZGRADBE PROGRAMA .....	17
<b>4</b>	<b>APLIKACIJA .....</b>	<b>20</b>
4.1	STREŽNIŠKA STRAN .....	20
4.1.1	<i>Express funkcije za HTTP poizvedbe</i> .....	21
4.1.2	<i>Axios HTTP poizvedbe</i> .....	24
4.2	STRAN ODJEMALCA .....	27
4.2.1	<i>JSX</i> .....	27
4.2.2	<i>Kavlji</i> .....	29
4.2.3	<i>Usmerjanje in navigacija</i> .....	42
4.2.4	<i>Prenašanje lastnosti</i> .....	47
4.2.5	<i>Dodajanje stilov s CSS</i> .....	49
4.2.6	<i>Priprava za produkcijo in lokalno oddajanje</i> .....	50
4.3	PODATKOVNA BAZA .....	51
<b>5</b>	<b>TESTIRANJE .....</b>	<b>55</b>
5.1	SPLETNA TRGOVINA .....	55
5.2	SISTEM UPORABNIKOV .....	59
<b>6</b>	<b>ZAKLJUČEK .....</b>	<b>63</b>
<b>7</b>	<b>ZAHVALA .....</b>	<b>64</b>
<b>8</b>	<b>VIRI IN LITERATURA .....</b>	<b>65</b>

## KAZALO SLIK

Slika 1: Poenostavljena sestava aplikacije (20).....	8
Slika 2: Prikaz delovanja obljub v JavaScriptu (21).....	10
Slika 3: Diagram konteksta za nakup .....	15
Slika 4: Diagram nivoja 0 za nakup.....	16
Slika 5: Diagram nivoja 1 za nakup.....	16
Slika 6: Diagram strežniškega dela .....	17
Slika 7: Povezava "uvaža" .....	18
Slika 8: Povezava "posreduje" .....	19
Slika 9: Pregled pridobljenega izdelka iz podatkovne baze v orodju za razvijalce brskalnika	25
Slika 10: Slika izdelka v spletni trgovini.....	26
Slika 11: Prikaz navigacijske vrstice trgovine na vrhu.....	35
Slika 12: Prikaz navigacijske vrstice trgovine ob premikanju navzdol.....	35
Slika 13: Prikaz uporabe useRef pri iskanju izdelkov .....	38
Slika 14: Polje za izdelek v spletni trgovini .....	38
Slika 15: Komponenta Error ob napačnem URL.....	44
Slika 16: Link za prijavo ali odjavo.....	45
Slika 17: Posnetek zaslona prikaza podrobnosti izdelka .....	55
Slika 18: Pogled nakupovalne košarice .....	56
Slika 19: Iskanje izdelkov po imenu .....	57
Slika 20: Spletna trgovina ob napaki strežnika .....	58
Slika 21: Pregled uporabnikov .....	60
Slika 22: Dodajanje izdelkov.....	61
Slika 23: Varnostno opozorilo pred izvedbo SQL ukaza .....	62

## KAZALO IZSEKOV PROGRAMA

Izsek 1: Enostaven HTML dokument, uporabljen v aplikaciji.....	3
Izsek 2: Povezovanje HTML dokumenta s CSS datoteko (5).....	4
Izsek 3: Primeri CSS stilskih določb .....	4
Izsek 4: Primer uporabe obljub.....	10
Izsek 5: Primer asinhronne funkcije z async/await .....	11
Izsek 6: Kreiranje express aplikacije .....	20
Izsek 7: Dodajanje funkcionalnosti express aplikaciji .....	21
Izsek 8: Funkcija za express GET zahtevo .....	22
Izsek 9: Funkcija za express DELETE zahtevo.....	22
Izsek 10: Funkcija za express POST zahtevo .....	23
Izsek 11: Axios GET zahteva .....	24
Izsek 12: Axios POST zahteva .....	24
Izsek 13: Vnosno polje za slike .....	25
Izsek 14: Pridobivanje slik izdelkov.....	26
Izsek 15: Primer programske komponente .....	28
Izsek 16: Uporaba useState.....	29
Izsek 17: Nastavljanje nove vrednosti objekta .....	30
Izsek 18: Spread operator na tabeli.....	31

Izsek 19: Nakupovalni kontekst .....	32
Izsek 20: Uporaba ponudnika konteksta.....	32
Izsek 21: Pridobivanje konteksta z useContext .....	33
Izsek 22: Uporaba useRef za predhodni odmik strani (100) .....	34
Izsek 23: Element za samodejni premik pogleda strani .....	36
Izsek 24: Uporaba useRef na elementih JSX .....	37
Izsek 25: Uporaba funkcije preventDefault() pri obrazcu .....	38
Izsek 26: Programska koda za gumb iz ProduktC.jsx .....	38
Izsek 27: Uporaba useEffect za pridobivanje vloge uporabnika .....	40
Izsek 28: Uporaba useCallback v VsebinaTrgovineC.jsx .....	41
Izsek 29: Prikaz poti React Routerja v App.js.....	43
Izsek 30: Uporaba Link v navigacijski vrstici .....	45
Izsek 31: Uporaba useLocation v Avtentikacija.....	46
Izsek 32: Uporaba useNavigate v IzbrisProfilaC.jsx.....	47
Izsek 33: Uporaba zunanjih povezav .....	47
Izsek 34: Drugi način sprejemanja lastnosti .....	48
Izsek 35: Podajanje lastnosti nadrejene komponente .....	48
Izsek 36: Prvi način sprejemanja lastnosti.....	48
Izsek 37: Stilska določila za elemente z atributom className='poljeProdukta' .....	49
Izsek 38: Element <div> z atributom className='poljeProdukta' .....	49
Izsek 39: Ukaz za statično oddajanje aplikacije .....	50
Izsek 40: Batch datoteka za lokalni zagon aplikacije .....	50
Izsek 41: Ustvarjanje baze podatkov .....	51
Izsek 42: SQL stavki za ustvarjanje tabele Stranke_in_zaposleni .....	52
Izsek 43: SQL stavka za dodajanje administratorja podatkovne baze.....	53
Izsek 44: Povezava .env s programskimi datotekami.....	53
Izsek 45: Uporaba spremenljivk okolja za povezavo s podatkovno bazo .....	54
Izsek 46: Spremenljivke okolja v .env.....	54
Izsek 47: SQL stavek za izbris testnih podatkov .....	62

## KAZALO PRIKAZOV

Prikaz 1: Opredeljene entitete s primarnimi ključi.....	13
Prikaz 2: Opredeljene entitete z atributi .....	13
Prikaz 3: Relacijska shema .....	14





# 1 Uvod

Seminarska naloga je namenjena dokumentiranju aplikacije, ki je zaključni izdelek za predmet računalništvo pri splošni maturi. Namen je opisati orodja in metode, uporabljene pri izdelavi aplikacije in načrt same izdelave. Načrt obsega fazo načrtovanja, izvedbe in testiranja. Pri vseh fazah so opisani najpomembnejši postopki. Med izdelavo programa in seminarske naloge sem načrtoval program s pomočjo spletnega orodja *diagrams.net*, ga nato implementiral v razvojnih okoljih *Visual Studio Code* in *MySQL Workbench* in testiral s pomočjo vgrajenega orodja za razvijalce v brskalnikih Chrome in Firefox. Cilj seminarske naloge je, da se v njej predstavi najpomembnejše gradnike programa na čim bolj razumljiv način.

## 2 Orodja in postopki

### 2.1 Spletno programiranje

Spletno programiranje je dandanes nepogrešljivo, saj se s spletnimi aplikacijami srečujemo vsakodnevno. Je zelo razširjeno in pomembno za hiter razvoj svetovnega medmrežja, ki postaja vse bolj prepleteno in kompleksno. Kaj pa sploh spada pod pojem spletno programiranje? Po definiciji iz Techopedie je spletno programiranje pisanje, označevanje in kodiranje, vpleteno v izdelavo spletnih vsebin in programov za odjemalce in strežnike. Najpogosteje s tem mislimo na programske jezike XML, HTML, JavaScript, Perl 5 in PHP. (1) Programiranje spletnih aplikacij obsega znanje na področjih programiranja uporabniškega vmesnika, logike delovanja programa in izdelave podatkovne baze.

Pri izdelavi spletne aplikacije razdelimo programsko kodo na kodo odjemalca ali klienta in strežnika. Koda klienta je namenjena dostopanju do podatkov s strežnika in sprejemanju vnesenih podatkov uporabnika. Poleg tega skrbi za dinamičen uporabniški vmesnik, ki vsebuje ustrezne varnostne mehanizme za preprečevanje napak. Strežniška programska koda je povezana s pridobivanjem podatkov iz podatkovnih baz, varnostnimi mehanizmi in določa zmogljivost programa. Za strežniško stran se pogosto uporabljajo jeziki PHP, Java, ASP.NET, JavaScript in drugi. (1)

Nezahtevne aplikacije lahko podatke shranjujejo v datotekah na samem sistemu izvajanja, vendar se pri današnjih potrebah po povezovanju in prepletanju informacij iz mnogo različnih virov izkaže, da je ta način zelo neprimeren za shranjevanje večjih količin podatkov. V ta namen uporabljamo podatkovne baze. Z njimi shranjujemo medsebojno povezane podatke in do njih preprosto dostopamo prek sistema za upravljanje s podatkovno bazo (SUPB). Ta zagotavlja varnost in zanesljivost pri ravnanju s podatki. (2)

### 2.2 HTML

HTML ali angleško *Hypertext Markup Language* je format za pisanje dokumentov spletnih strani. Poleg besedila vsebuje značke, s katerimi besedilo in drugo vsebino dodajamo v strukture za prikaz. Poznamo veliko različnih vrst značk, na primer za hiperpovezave, odstavke, naslove itd. Značke so sestavljene iz znakov "<" in ">", znotraj katerih je ime. Ime določa strukturo besedila ali drugih strukturnih elementov, vsebovanih med odpirajočo značko <p> in zapirajočo značko </p>. Odpirajoča značka ima lahko dodane attribute, ki so zapisani v obliki

`<a href="vrednost" name="hiperpovezava">`, in ki že določeni strukturi elementa dodajo podatke. Te lahko kasneje uporabimo pri povezovanju struktur in programiranju uporabniškega vmesnika. Nekatere značke nimajo vsebine, ki bi jo prikazovali. Te ne potrebujejo zapirajoče značke, poznamo pa tudi samozapirajoče značke, kot je na primer `<br/>` za skok v novo vrsto. Čeprav znajo brskalniki danes popraviti marsikatero napako v formatu HTML, na primer dodati značko za zapiranje odstavka, je dobra praksa, da pišemo dokumente dosledno in berljivo. (3)

HTML dokumenti se začnejo z značko `<!doctype html>`, ki brskalniku pove, naj interpretira podatke kot moderen HTML zapis. Podatki dokumenta se nahajajo v `<html>`, ki je običajno razdeljen na glavo `<head>` in telo `<body>`. V glavi dokumenta so podatki o dokumentu samem, imenovani tudi metapodatki `<meta/>`, in povezave z drugimi dokumenti `<link/>` ter naslov `<title>`, v telesu pa je vsebina spletne strani. Na Izsek 1 lahko vidimo zgradbo dokumenta *index.html*. (3)

*Izsek 1: Enostaven HTML dokument, uporabljen v aplikaciji*

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8" />
    <meta name="viewport"
      content="width=device-width, initial-scale=1" />
    <meta name="theme-color" content="#000000" />
    <meta name="description"
      content="Web site created using create-react-app"
    />
    <link rel="manifest" href="%PUBLIC_URL%/manifest.json" />
    <title>Aplikacija Matura</title>
  </head>
  <body>
    <div id="root"></div>
  </body>
</html>
```

## 2.3 CSS

CSS (*Cascading Style Sheets*) je standard za formatiranje in stilsko urejanje elementov, napisanih v označevalnem jeziku, kot na primer HTML. Ustvarili so ga na W3C konzorciju leta 1996 z namenom oblikovanja elementov na HTML straneh. (4) Za vzdrževanje in posodabljanje standarda je odgovoren W3C konzorcij. CSS omogoča dodajanje ustreznega oblikovanja vsebini spletnih strani in s tem pripomore k berljivosti in izgledu. CSS lahko

pišemo na tri različne načine, ki se uporabljajo glede na to, koliko oblikovanja potrebujemo in kje. (5)

Najpogosteje se CSS uporablja v posebnih datotekah s končnico *.css*, v katerih določamo stil za posamezne elemente oziroma njihove ID-je, razrede (*class*) ali HTML značke. Te datoteke nato povežemo s HTML dokumentom, v katerem želimo uporabiti stilsko oblikovanje, in sicer z značko, ki ga lahko vidimo na Izsek 2 in se nahaja v glavi HTML dokumenta.

*Izsek 2: Povezovanje HTML dokumenta s CSS datoteko (5)*

```
<link rel="stylesheet" type="text/css" href="css_oblikovanje.css">
```

Ta način je bolj fleksibilen in lažji za vzdrževanje ter pohitri oblikovanje z ne redundantnim zapisovanjem stila za več elementov, ki si delijo iste značilnosti. (4) Mogoča pa je tudi uporaba CSS v posebnem predelu samega HTML dokumenta, obdanim z značko `<style></style>`. To imenujemo notranje oblikovanje (*internal style level*) in ima prednost pred zunanjim oblikovanjem iz drugih CSS datotek. Če želimo dodajati stil le nekaterim elementom ali dodati le majhen popravek k tistemu iz zunanjih datotek, lahko uporabimo tudi t. i. vrstični CSS (*inline style*), ki ga zapišemo kot atribut *style* k HTML znački.

CSS ima preprosto sintakso, sestavljeno iz CSS selektorja in zavitih oklepajev `»{ }«`, ki predstavljajo deklarativni blok, v katerem so dodani stilski atributi za ta selektor, ločeni s podpičjem. Selektor določa, kateri elementi bodo uporabljali stilske lastnosti. Določimo lahko elemente s posebnim ID-jem, in sicer z dodajanjem `»#«` pred ime ID-ja, prav tako lahko oblikujemo več elementov hkrati. To storimo z določanjem enakih vrednosti atributa *class* v HTML dokumentu in nato v CSS pred to vrednost selektorja dodamo `».«`. CSS omogoča tudi pisanje stila za posamezne HTML značke. (5) Poglejmo si primere za določanje oblikovanja po ID, razredu in znački na Izsek 3.

*Izsek 3: Primeri CSS stilskih določb*

```
/* določanje za elemente z atributom id = "abcd" */
#abcd {
    background-color: white;
}

/* določanje za elemente z atributom class = "razred" */
.razred {
    font-size: 20px;
}

/* določanje za značko <p> */
p {
    border: 1px solid black;
}
```

## 2.4 JavaScript

JavaScript je odprt objektni programski jezik za ustvarjanje spletnih aplikacij na več platformah. Razvili so ga pri Netscape v sodelovanju s Sun Microsystems in Brendanom Eichom. Predstavljen je bil leta 1995, z namenom dodajanja programov do tedaj statičnim spletnim stranem na brskalniku Netscape Navigator. (3)

JavaScript kot visokonivojski programski jezik omogoča programiranje kompleksnih spletnih strani, ki povezujejo objekte in vire na odjemalcih in strežnikih. (6) Z njim lahko spletnim stranem dodamo elemente in funkcionalnosti, ki so nujni za uporabnost spletnih strani. (7) JavaScript so kmalu po predstavitvi prevzeli vsi večji grafični spletni brskalniki, saj je omogočil izgradnjo modernih spletnih aplikacij. Prednost je bila ta, da je lahko uporabnik neposredno komuniciral s spletno aplikacijo, ne da bi bilo pri tem potrebno ponovno naložiti spletno stran pri vsakem dejanju.

Poimenovanje je podobno programskemu jeziku Java, katere uporaba je v času nastanka JavaScripta hitro naraščala. JavaScript je bilo odlično ime, s katerim so pridobili pozornost širše množice razvijalcev. Vendar se je to izkazalo za napako, ljudje so namreč pomotoma mešali dva popolnoma različna programska jezika, zato so JavaScript poimenovali tudi ECMAScript, po mednarodni Ecma organizaciji, ki je standardizirala jezik. (3)

Jezik je bil oblikovan zelo »liberalno«, saj dovoljuje zelo različno sintakso, kar je po eni strani prednost predvsem za začetnike pri programiranju, pa tudi na splošno je mogoče implementirati veliko programskih rešitev na lažji način, kot bi to bilo mogoče z bolj togimi programskimi jeziki. Po drugi strani pa je velika slabost težje odkrivanje in odpravljanje napak.

Najbolj razširjena različica JavaScripta, verzija 3, je bila prisotna med vzponom od leta 2000 do 2010. Medtem, ko je jezik uspešno pridobival vedno več programerjev, so delali na večji posodobitvi, ki pa so jo opustili zaradi težkega prehoda, ki bi veliko razvijalcem povzročal težave, in raje izdali verzijo z manj spremembami. Vendar so izboljšave vpeljali leta 2015 s šesti izdajo ECMAScripta. Od tedaj jezik prejema vsakoletne manjše posodobitve, ki pripomorejo k lažjemu sprejemanju novosti. (3)

## 2.5 React.js

React.js je odprtokodna knjižnica za JavaScript, ki jo je razvil Facebook za izdelavo uporabniških vmesnikov. (8) (9) (10) Uporablja se za hitro in učinkovito izdelavo interaktivnih uporabniških vmesnikov in spletnih aplikacij, pri čemer je potrebno napisati manj programske

kode, kot bi jo potrebovali z običajnim *vanilla* JavaScriptom. Koncept Reacta je razvijanje komponent za večkratno uporabo, ki se med seboj povežejo in sestavljajo celoten uporabniški vmesnik. Uporablja se na odjemalčevi strani aplikacije, kjer stremimo k čim hitrejšemu in optimalnemu upodabljanju informacij. To pa je z Reactom mogoče storiti na razvijalcu prijazen način, z razbitjem kode na gradnike in poenostavitvijo kompleksnega uporabniškega vmesnika. (11)

### 2.5.1 Prednosti

Slabost spletnih strani, zgrajenih le z JavaScriptom, je interaktivnost z uporabnikom. Ob vsaki spremembi namreč odjemalec pošlje novo poizvedbo na strežnik, od koder se pridobi nova spletna stran. Poleg tega mora razvijalec strukturirati program tako, da manipulira z DOM (*Document Object Model*) elementi tako, da ugotovi, kaj se je na strani spremenilo, in temu primerno posodobi DOM, kar zahteva ponovno nalaganje. DOM je model objektov na spletni strani, ki omogoča, da programsko spreminjamo strukturo, stil in vsebino dokumenta. Vsak element na spletni strani je vozlišče in več vozlišč je lahko del objekta. Ker je dokument predstavljen kot objektno orientiran model, ga lahko spreminjamo s programskim jezikom, kot je JavaScript. (12) Za spletne strani z majhno potrebo po interaktivnosti in dostopanju do strežniških podatkov je tradicionalen način povsem sprejemljiv, vendar se pri kompleksnejših aplikacijah in spletnih straneh, ki potrebujejo veliko komunikacije s strežniki, lahko pojavijo težave. Večkratno ponovno nalaganje spletnih strani pomeni počasno aplikacijo in slabo uporabniško izkušnjo. React rešuje problem nepotrebnega ponovnega nalaganja celotnih spletnih strani. Omogoča izgradnjo aplikacij z le eno spletno stranjo, pri katerih se ob zagonu aplikacije naloži le en HTML dokument. Če se spremeni le del strani, se ne nalaga ponovno celotna stran, ampak le določena komponenta na njej. Temu rečemo tudi usmerjanje na strani odjemalca (angl. *client-side routing*), saj lahko novo »spletno stran« predstavimo le z novo komponento in ni potrebno ponovno pridobivanje nove spletne strani s strežnika. Zato so aplikacije, ki implementirajo React, hitreje in bolj dinamične. (11)

### 2.5.2 Delovanje

React se za učinkovit prikaz komponent spletnih strani poslužuje metode manipulacije DOM s pomočjo navideznega DOM (angl. *Virtual DOM*). (13) Slednji je kopija originalnega DOM in se ponovno naloži ob spremembi na spletni strani. React nato primerja navidezni DOM z originalnim in ugotovi optimalen način spremembe originalnega DOM, ne da bi s tem povzročil ponovno nalaganje spletne strani. To omogoča ustvarjanje zelo dinamičnih in odzivnih spletnih aplikacij. (11)

### 2.5.3 Fleksibilnost

React omogoča, da ga uporabimo ravno toliko, kot ga potrebujemo. Dodajamo ga lahko že obstoječim spletnim stranem, ga v nove vključimo le deloma ali postopoma, ali pa z njim zgradimo popolnoma celoten uporabniški vmesnik. Prav tako ne uveljavlja posebnih konvencij za programsko kodo, saj je to prepuščeno razvijalcem. (11)

## 2.6 Node.js

Node.js je odprtokodno izvajalno okolje programskega jezika JavaScript in knjižnica za izvajanje spletnih aplikacij zunaj odjemalčevega brskalnika. Deluje na več platformah in se uporablja za programiranje strežniške kode z jezikom JavaScript ter tako omogoča izdelavo odjemalčeve in strežniške kode le z uporabo JavaScripta. (14) Razvil ga je Ryan Dahl, leta 2009. Zgrajen je na podlagi *Google Chrome V8 engine*, napisanem v programskem jeziku C++, ki prevaja kodo po sistemu JIT (angl. *just-in-time*), kar pomeni, da se JavaScript sproti prevaja v bitno kodo za izvajanje. (14) Node.js je razširjen zaradi svojega modela, ki temelji na asinhronem modelu, ki temelji na dogodkih. (15) Node.js vsebuje tudi upravitelja paketov npm (*node package manager*) z vmesnikom ukazne vrstice. Z npm-jem je mogoče uporabiti register JavaScripta, ki je največja shramba za ta jezik na svetu in ponuja številne pakete s programskimi knjižnicami.

### 2.6.1 Delovanje

Node.js ob zagonu aplikacije zažene proces in ne ustvarja novih niti za vsako zahtevo, kar je sicer pogosta praksa pri tradicionalnih programih na strežniški strani. Tako ne pride do težav pri vzporednih povezavah. Node.js okolje deluje asinhrono in temelji na dogodkih, zato je pisanje kode zanj drugačno od tradicionalnega. Node izvaja zanko dogodkov, ki obdeluje dohodne zahteve po vrstnem redu le teh v čakalni vrsti. Zanka obravnava majhne zahteve eno za drugo, ne da bi čakala na odgovore. Tako različnim nitim ni potrebno čakati na odzive pred nadaljevanjem programa in vhodno izhodne operacije niso blokirane. Zato je Node.js zelo uporabno okolje za aplikacije z velikimi obremenitvami vzporednih procesov. Node pa se zaradi svojih lastnosti ne obnese dobro pri aplikacijah, ki so procesorsko intenzivne in potrebujejo izvajanje zahtevnih operacij na strežniku (14)

### 2.6.2 Razširjenost

»Node.js je uporabljen pri vsaj 30 milijonih spletnih straneh.« Ta podatek je zelo zgovoren glede popularnosti izvajalnega okolja. Številne prednosti ga uvrščajo med najbolj uporabljane

izvajalne platforme, uporabljajo ga tudi ene najbolj obiskanih spletnih strani, predvsem zaradi krajšega časa zagona, večje zanesljivosti in hitrosti delovanja. (16)

## 2.7 Express.js

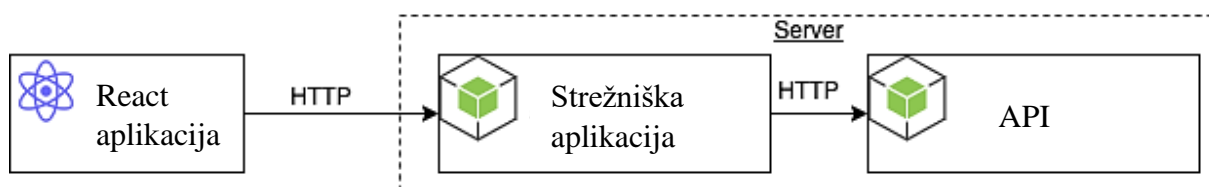
Express.js je strežniško ogrodje, ki temelji na HTTP modulu Node.js in komponentah *Connect*. Te komponente so vmesna programska oprema, ki razvijalcem omogočajo visoko prilagodljivost njihovim potrebam. Z njimi se olajša in zmanjša pisanje ponavljajoče kode za HTTP zahteve. (17) Ponuja nabor orodij za ustvarjanje HTTP poizvedb, usmerjanje in vmesno programsko opremo za izdelavo spletnih aplikacij. (18)

## 2.8 Axios

Je HTTP odjemalec za Node.js in brskalnik, ki temelji na obljubah (angl. *promises*). Lahko deluje v brskalniku in vozliščih z isto kodno osnovo. Na strežniški strani uporablja izvorni http modul Node.js, na odjemalčevi strani pa XMLHttpRequest<sup>1</sup>. (19)

Axios je knjižnica, ki ni nujno potrebna za pošiljanje klicev na API ali aplikacijski programski vmesnik (angl. *Application Programming Interface*), saj to lahko storimo z že vgrajenimi objekti v JavaScriptu, kot so Fetch API in XMLHttpRequest. Čeprav Axios vrača obljube in deluje podobno kot Fetch API, ponuja dodatne funkcionalnosti. Lahko deluje tako v brskalniku, kot tudi v strežniških aplikacijah in je kompatibilen z ogrodji JavaScripta, med drugimi tudi z Reactom. (20)

Pri aplikaciji sem uporabil odjemalca Axios, ki se je prek HTTP zahteve z URL povezal na funkcije ogrodja Express.js na strežniški strani, ki vsebuje REST API-je<sup>2</sup>. Delovanje aplikacije na poenostavljen način prikazuje Slika 1.



Slika 1: Poenostavljena sestava aplikacije (20)

<sup>1</sup> Objekti, ki se uporabljajo za pridobivanje podatkov iz strežnika, ne da bi za to potrebovali ponovno nalaganje celotne spletne strani (104)

<sup>2</sup> API-ji, ki so oblikovani po načelih REST oz. reprezentativnega prenosa stanja (105)



## 2.9 MySQL

MySQL je odprtokoden sistem za upravljanje s podatkovnimi bazami ali SUPB (angl. *RDBMS*), ki so ga razvili v švedskem podjetju MySQL AB leta 1994. Od leta 2008 pa je bil polni lastnik podjetje Sun Microsystems, katerega je leta 2010 prevzel Oracle. MySQL je programska oprema, razvita s programskima jezikoma C in C++, za ustvarjanje in upravljanje podatkovnih baz, ki temeljijo na relacijskem podatkovnem modelu, kar pomeni shranjevanje podatkov v obliki tabel. MySQL deluje na modelu odjemalec – strežnik, pri čemer je odjemalec povpraševalec po podatkih iz baze, strežnik pa je SUPB sam, ki dostavi odjemalcu želene podatke. Komunikacija med odjemalcem in strežnikom poteka preko jezika SQL (angl. *Structured Query Language*). (21)

### 2.9.1 SQL

SQL je povpraševalni jezik za dostopanje do baz podatkov preko sistema za upravljanje s podatkovno bazo. Razvil ga je Ted Codd v začetku 70. let, kmalu je nadomestil takratne jezike in postajal vse bolj uporabljan. SQL je postopkovni programski jezik, ki strežniku pove, kaj naj stori s podatki, ne pa kako naj to stori. Osnovne operacije, ki jih lahko izvajamo so poizvedovanje po podatkih, definiranje shem in relacijskih povezav, urejanje podatkov v bazi in nadzor nad dostopom do podatkov. Te operacije ponujajo SUPB-ji in SQL omogoča uporabniku, da strežniku poda svoje zahteve. (21)

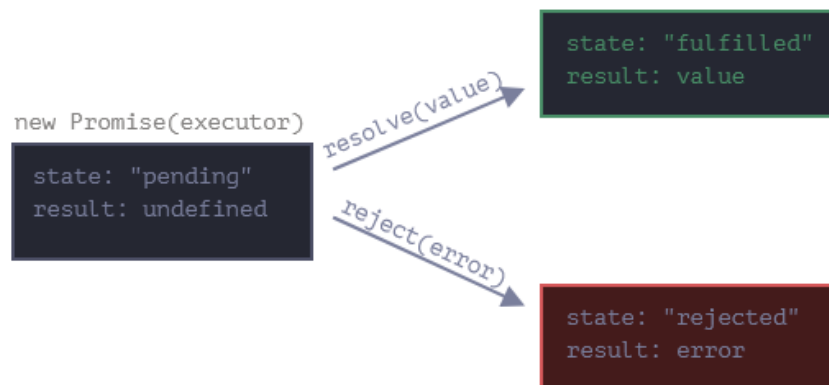
### 2.9.2 Prednosti

MySQL je na tržišču eden vodilnih sistemov za upravljanje s podatkovnimi bazami. Tako visoko ga uvrščajo njegova visoka zmogljivost, zanesljivost, varnost in standardiziranost. Je sistem, ki je zelo uveljavljen v industriji in brezplačno ponuja zelo dobro zmogljivost in dokumentacijo. (21)

## 2.10 Asinhrono programiranje

Asinhrono programiranje je način programiranja, pri katerem začnemo določena opravila in ne čakamo, da se končajo, ampak nadaljujemo z izvajanjem programa. Je nasprotje sinhronemu programiranju, pri katerem se ukazi izvajajo strogo po zaporedju in se v primeru zahtevne operacije zaustavi ter čaka na izvedbo. Asinhrono programiranje je zasnovano za prilagajanje zamiku med klicem funkcije in vrnitvijo vrednosti. Tako lahko uporabnik nadaljuje z uporabo aplikacije medtem ko se v ozadju izvaja opravilo, na primer uporabnik odda zahtevo za zapis podatkov v podatkovno bazo in nadaljuje z uporabo aplikacije na drugi strani. Tako lahko

začnemo več procesov, ki se izvajajo istočasno in ko se končajo nadaljujemo z operacijami, ki so potrebovale podatke. (22) Asinhrono programiranje temelji na obljubah, ki so posebni



Slika 2: Prikaz delovanja obljub v JavaScriptu (21)

JavaScript objekti, ki povezujejo kodo, ki zahteva operacijo s kodo, ki jo izvaja. Obljuba je na voljo, ko se dana operacija izvede. Če se je operacija izvedla uspešno, pokliče *callback* funkcijo `resolve()`, v nasprotnem primeru pa *callback* funkcijo `reject()`.

Za funkcijo, ki je uporabnik podatkov iz obljube lahko zapišemo dejanja po razrešitvi obljube. To storimo z metodo `then()`, ki sprejme kot prvi argument funkcijo ali drug dogodek, ki se zgodi ob uspešni razrešitvi obljube, drugi argument pa je namenjen izvedbi ob neuspešni razrešitvi. Mogoča je tudi uporaba metode `catch()`, če želimo definirati dogodke le po neuspešni razrešitvi obljube. Poznamo tudi metodo `finally()`, ki se izvede ob kakršnikoli razrešitvi obljube. (23) Oglejmo si primer uporabe na Izsek 4.

Na Izsek 4 je funkcija `pridobiProdukte()`, ki vrača obljubo, ki jo vrne metoda `axios.get()`. Ker funkcija vrača obljubo, ob klicu funkcije uporabimo `then()`, `catch()` in `finally()`. Metodi `then()` se izvedeta, če je obljuba uspešno pridobila rezultat. Prvi vrne vrednost lastnosti `data` pridobljenega rezultata, ki je v spremenljivki `response`. Kar vrne

Izsek 4: Primer uporabe obljub

```

function pridobiPodatke(uporabniskoIme){
  return axios.get(`http://localhost:3000/api`, { params: {
    uporabnisko_ime: uporabniskoIme }, });
}
pridobiPodatke("uporabnisko_ime1")
  .then((response) => {console.log("uspesno"); return
response.data})
  .then((data) => { console.log(data)})
  .catch((error) => { console.log("napaka")})
  .finally(() => { console.log("koncano")});
  
```

predhodni `then()`, lahko sprejmemo v drugem `then()`, v tem primeru smo vrednost sprejeli s spremenljivko `data`. Metoda `catch()` se izvede ob neuspešnem pridobivanju rezultata, `finally()` pa se izvede v vsakem primeru.

Namesto obljub in metod lahko uporabljamo tudi enostavnejšo sintakso, ki se imenuje *async/await*. *Async* je oznaka za asinhrono funkcijo, kar pomeni, da vrača obljubo. Če iz funkcije ne vračamo obljube sami, se samodejno vrne uspešno razrešena obljuba, z vrednostjo, ki smo jo podali za vračanje. *Async* torej sam poskrbi, da funkcija vrača obljubo. Rezervirana beseda *await* pa sporoči JavaScriptu, naj počaka na razreševanje obljube in vrne rezultat. Uporabljati jo smemo le znotraj *async* funkcij, saj *await* prekine izvajanje funkcije, dokler obljuba ni razrešena. Z njim torej ustavimo le funkcijo, medtem ko program teče dalje. Ko je obljuba razrešena, se nadaljuje izvajanje funkcije. Če se obljuba razreši uspešno, potem *await Promise* vrne rezultat, če pa neuspešno, pa program javi napako. Zato moramo *await* stavke vedno obdati s `try...catch` blokom. *Catch* se izvede v primeru neuspešno razrešene obljube. (23) Poglejmo si primer na Izsek 5.

Izsek 5: Primer asinhrono funkcije z *async/await*

```
async function pridobiPodatke(uporabniskoIme){
  let profilnaSlika = null;
  try{
    profilnaSlika = await axios.get(`http://localhost:3000/api`,
      { params: { uporabnisko_ime: uporabniskoIme }, })
  } catch (error) {
    alert(error);
  }
  return profilnaSlika;
}
```

Funkcija *pridobiPodatke()* je asinhrona in vrača obljubo. V njej ustvarimo spremenljivko *profilnaSlika*, ki bo shranjevala sliko, pridobljeno iz podatkovne baze. Znotraj *try...catch* bloka bomo spremenljivki priredili vrednost, ki jo bo vrnila funkcija *axios.get()*. *Await* poskrbi, da se izvajanje funkcije zaustavi, dokler ne pridobimo podatkov, ki razrešijo obljubo. *Await* nato ob uspešni pridobitvi podatkov iz obljube vrne rezultat, saj se je ta uspešno razrešila, v nasprotnem primeru pa javi napako, ki jo ujame *catch* blok. Funkcija na koncu vrne pridobljen podatek ali začetno vrednost spremenljivke, če je prišlo do napake.

### 3 Modeliranje

Prvi koraki pri razvoju aplikacije so načrtovanje strukture in modeliranje. Zgradba je lahko zelo zapletena, zato je nujno, da jo dokumentiramo in s tem omogočimo hitrejše zaznavanje in odpravljanje napak ter morebitno kasnejšo nadgradnjo. Za načrtovanje sistema sem uporabil tehniko gradnje programa od zgoraj navzdol. To pomeni, da sem aplikacijo razdelil na probleme s funkcionalno dekompozicijo in jih reševal sproti. Tekom gradnje aplikacije sem dodajal in ugotavljal probleme ter jih reševal dokler nisem dobil končne rešitve problema – celotno aplikacijo. (24) Pri izdelavi aplikacije sem upošteval tudi programirni postopek, t. j. sistematični postopek izdelave programa. Najprej sem opredelil cilje, torej funkcionalnosti, ki jih mora aplikacija dosegati. Zatem sem določil lastnosti teh funkcionalnosti in izbral metode in tehnike za načrtovanje in izdelavo. Načrtovanje sem tudi podrobneje opisal v tem poglavju. Ko sem naredil načrt, sem se lotil kodiranja in sprotnega popravljanja napak. Na koncu pa sem še izvedel testiranje programa. Ker aplikacija ni namenjena uporabi v produkciji, sem izpustil korake uporabe med uporabniki in vzdrževanja. (2)

#### 3.1 ER diagram

ER diagram ali diagram razmerij entitet (angl. *Entity Relationship Diagram*) je diagram, ki se uporablja za grafični prikaz razmerij med objekti ali entitetami v informacijskem sistemu. Uporablja se za modeliranje relacijskih podatkovnih baz, kar služi za pregled informacijskih potreb organizacije in kot načrt za iskanje napak ali nadgradnjo sistema. (25) ER je mešanica mrežnega in relacijskega podatkovnega modela, ki sta bila razvita predhodno, vendar sta imela pomanjkljivosti, ki jih ER odpravlja. Osnova je matematična definicija množice in relacije. Glavni gradniki so entiteta, atribut, relacija ali povezava, tip entitete. Entiteto sestavlja tip entitete in atributi, pri čemer posamezen atribut priredi tipu entitete množico pripadajočih vrednosti atributa. Entitete predstavimo s pravokotnikom, v katerem so atributi in njihove lastnosti. Med seboj so entitete povezane z relacijami glede na števnost. (2)

Za aplikacijo, ki hrani podatke o uporabnikih, naročilih in ostalih informacijah podjetja, potrebujemo dobro strukturirano podatkovno bazo, zato ustvarimo ER model po standardnem postopku. Najprej opredelimo entitete in določimo primarne ključe. Ker vemo, da bomo imeli v aplikaciji sistem za uporabniške račune, bomo potrebovali entiteto, ki jo poimenujmo *Uporabniki*, za primarni ključ bomo nastavili samo uporabniško ime, saj se morajo ta razlikovati. O strankah in zaposlenih bomo shranjevali podrobnejše informacije, za to bomo naredili entiteto *Stranke\_in\_zaposleni*, s primarnim ključem *ID*. Stranka bo lahko v spletni

trgovini naročala izdelke. Tako bomo ustvarili entitete *Narocila*, *Izdelki\_pri\_narocilu* in *Izdelki*. Entiteta *Narocila* ima primarni ključ *ID\_narocila*, *Izdelki* pa *ID\_izdelka*. Entiteti *Izdelki\_pri\_narocilu* nismo določili primarnega ključa, saj bo le povezovala posamezne izdelke z naročilom. Potrebujemo le še entiteto za račune, s primarnim ključem *ID\_racuna*. Opredelitev entitet vidimo na Prikaz 1.

**UPORABNIKI** (uporabnisko\_ime\*)  
**STRANKE\_IN\_ZAPOSLENI** (ID\*)  
**NAROCILA** (ID\_narocila\*)  
**IZDELKI\_PRI\_NAROCILU** ()  
**IZDELKI** (ID\_izdelka\*)  
**RACUNI** (ID\_racuna\*)

*Prikaz 1: Opredeljene entitete s primarnimi ključi*

Naslednji korak je opredelitev atributov entitetam. Entiteta *Uporabnik* bo shranjevala uporabniško ime, geslo, vlogo in logično vrednost omogočen. Za *Stranke\_in\_zaposleni* določimo attribute *ID*, uporabniško ime, elektronski naslov, ime, priimek, ulica in hišna številka, kraj, poštna številka, telefonska številka, podjetje, oddelek in plača. Atribut uporabniško ime je tuji ključ, ki se navezuje na uporabniško ime iz entitete *Uporabniki*. Entiteta *Narocila* ima attribute *ID\_narocila*, datum, *ID\_stranke*, logično vrednost opravljeno, ime stranke, priimek stranke, naslov dostave in poština. *ID\_stranke* je tuji ključ, ki se navezuje z atributom *ID* iz *Stranke\_in\_zaposleni*. Entiteta *Izdelki* ima *ID\_izdelka*, ime, kategorijo, ceno za kos, kosov na voljo, kratek opis, informacije, popust in sliko. Entiteta *Racuni* ima attribute *ID\_racuna*, *ID\_narocila*, kupca, znesek za plačilo in datum izdaje. Tukaj je tuji ključ *ID\_narocila*, ki se navezuje na *ID\_narocila* iz *Narocila*. Ostala je le še entiteta *Izdelki\_pri\_narocilu*, ki ima dva tuja ključa: *ID\_narocila* in *ID\_izdelka*, ter atributa količina in cena. Razporeditev atributov lahko vidimo na Prikaz 2.

**UPORABNIKI** (uporabnisko\_ime\*, geslo, vloga, omogocen)  
**STRANKE\_IN\_ZAPOSLENI** (ID\*, uporabnisko\_ime, elektronski\_naslov, ime, priimek, ulica\_in\_hisna\_stevilka, kraj, postna\_stevilka, telefonska\_stevilka, podjetje, oddelek, placa)  
**NAROCILA** (ID\_narocila\*, datum, ID\_stranke, opravljeno, imeStranke, priimekStranke, naslovDostave, postnina)  
**IZDELKI\_PRI\_NAROCILU** (ID\_narocila, ID\_izdelka, kolicina, cena)  
**IZDELKI** (ID\_izdelka\*, ime, kategorija, cena\_za\_kos, kosov\_na\_voljo, kratek\_opis, informacije, popust, slika)  
**RACUNI** (ID\_racuna\*, ID\_narocila, kupec, za\_placilo, datumIzdaje)

*Prikaz 2: Opredeljene entitete z atributi*

Ko imamo opredeljene entitete z atributi, ustvarimo še relacijsko shemo, ki jo nato grafično predstavimo z ER diagramom. Relacijsko shemo vidimo na Prikaz 3.

**UPORABNIKI** (uporabnisko\_ime: A(150), geslo: A(150), vloga: N, omogocen: N)  
**STRANKE\_IN\_ZAPOSLENI** (ID: N, uporabnisko\_ime: A(150) → Uporabniki, elektronski\_naslov: A(160), ime: A(120), priimek: A(130), ulica\_in\_hisna\_stevilka°: A(160), kraj°: A(130), postna\_stevilka°: N, telefonska\_stevilka°: A(20), podjetje°: A(255), oddelek°: A(100), placa°: N)  
**NAROCILA** (ID\_narocila: N, datum°: D, ID\_stranke°: N → Stranke in zaposleni, opravljeno: N, imeStranke°: A(50), priimekStranke°: A(80), naslovDostave°: A(80), postnina°: N)  
**IZDELKI\_PRI\_NAROCILU** (ID\_narocila: N → Narocila, ID\_izdelka: N → Izdelki, kolicina: N, cena: N)  
**IZDELKI** (ID\_izdelka: N, ime: A(40), kategorija: A(20), cena\_za\_kos: N, kosov\_na\_voljo: N, kratek\_opis°: A(40), informacije°: A(65535), popust°: N, slika°: A(4294967295))  
**RACUNI** (ID\_racuna: N, ID\_narocila: N → Narocila, kupec: A(150), za\_placilo: N, datumIzdaje°: D)

Prikaz 3: Relacijska shema

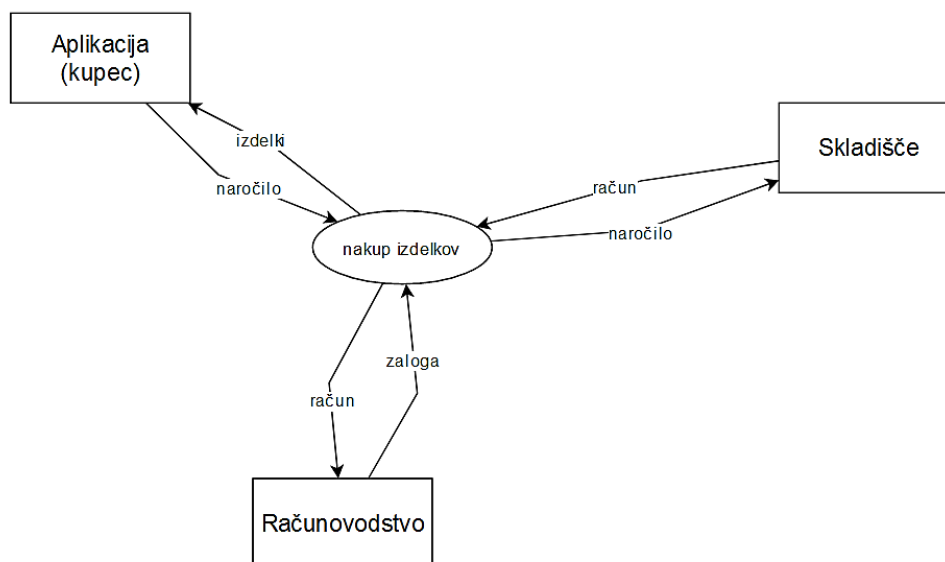
Iz relacijske sheme izdelamo ER diagram podatkovne baze, ki vsebuje natančne opise entitet in povezave. ER diagram si lahko ogledamo v Prilogi 1. Iz diagrama je razvidno, da je med *Uporabniki* in *Stranke\_in\_zaposleni* enostavna relacija, kar pomeni, da ima vsak zapis tipa *Uporabniki* natanko en zapis tipa *Stranke\_in\_zaposleni* in obratno. Prav tako je enostavna relacija vzpostavljena med entitetama *Narocila* in *Racuni*. Med relacijama *Stranke\_in\_zaposleni* in *Narocila* pa je relacija kompleksna, in sicer ena proti mnogo. Ena vrstica tipa *Stranke\_in\_zaposleni* ima lahko več zapisov tipa *Narocila* in ena vrstica tipa *Narocila* ima natanko en zapis tipa *Stranke\_in\_zaposleni*. Enak tip relacije velja tudi med entitetama *Narocila* in *Izdelki\_pri\_narocilu* ter *Izdelki* in *Izdelki\_pri\_narocilu*.

## 3.2 Diagram toka podatkov

Diagram toka podatkov ali DTP (angl. *Data Flow Diagram*) je diagram, s katerim predstavljamo pretok podatkov v informacijskem sistemu. Z njim opišemo procese, ki so vpleteni v prenašanje podatkov po sistemu od vnosa podatkov do shranjevanja. (26) V modeliranju se je pojavil leta 1978 z DeMarcom in postal danes najbolj razširjena tehnika modeliranja postopkov. Med ustvarjanjem diagrama izvedemo sistemsko analizo in nato za

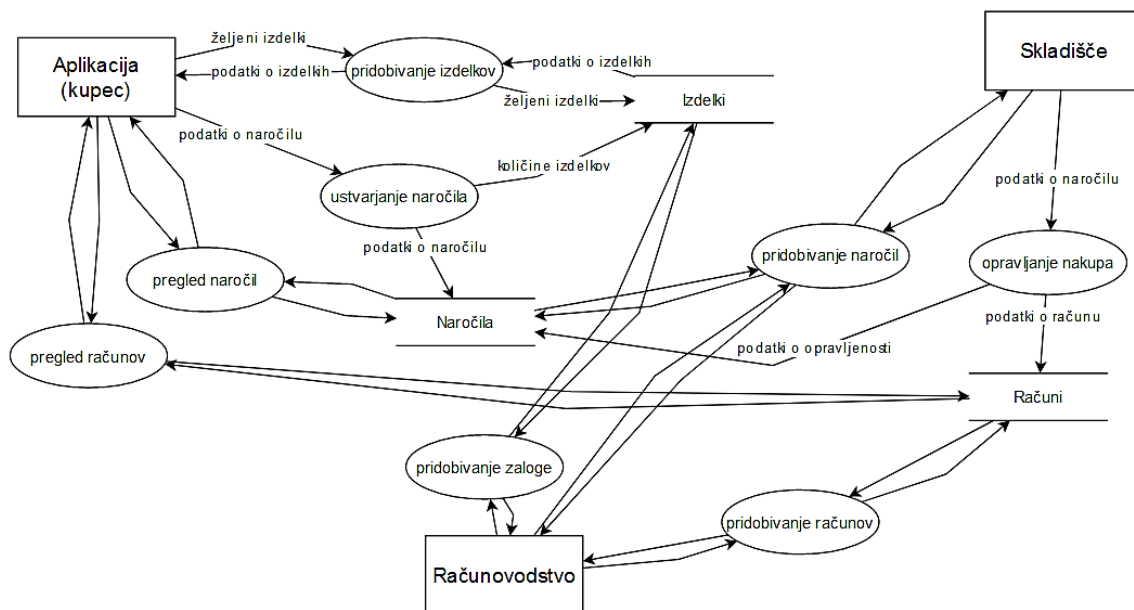
nerazstavljene atomarne procese zapišemo s psevdokodo. Osnovni elementi DTP so postopek, proces ali aktivnost, predstavljeni z elipso, zunanja entiteta, predstavljena s pravokotnikom, zbirka podatkov, predstavljena z nepopolnim pravokotnikom in tok podatkov, predstavljen s puščico. (2) Preden pa začnemo oblikovati DTP, pa moramo poznati še nekatera pravila, in sicer entiteta ne more prenesti podatkov drugi entiteti brez vmesnega procesa. Enako velja za prenos podatkov med entiteto in podatkovno bazo in med dvema podatkovnima zbirkama. Prav tako moramo paziti, da ima proces vedno vhodne in izhodne podatke. (26)

Pri naši aplikaciji je zelo smiselno ustvariti diagram toka podatkov za nakupovanje v spletni trgovini. V našem primeru imamo tri zunanje entitete: Aplikacija (kupec), Skladišče in Računovodstvo. Na začetku naredimo diagram konteksta, ki vsebuje le zunanje entitete in vse procese vsebovane v osrednjem procesu, kot je prikazano na Slika 3.



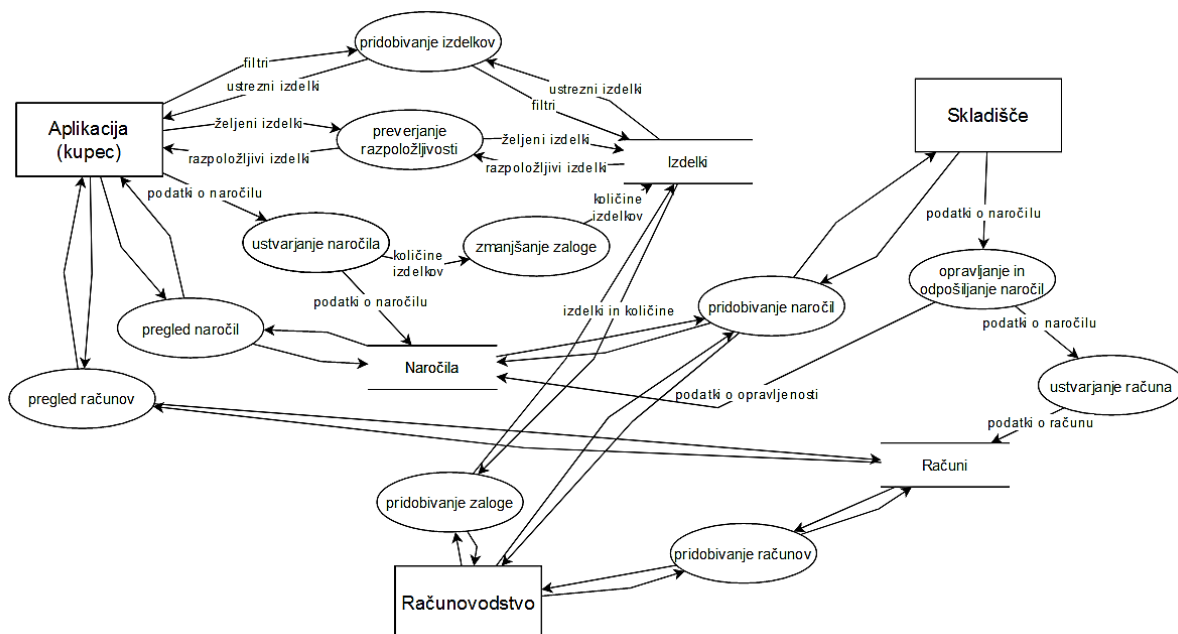
Slika 3: Diagram konteksta za nakup

Ko imamo diagram konteksta, lahko ustvarimo diagram nivoja 0, ki poleg entitet vsebuje še vse glavne podprocesse in podatkovne zbirke. Glavni proces *nakup izdelkov* se razdeli na več podprocesov, in sicer na *pridobivanje izdelkov*, *ustvarjanje naročila*, *pridobivanje naročil*, *opravljanje nakupa*, *pridobivanje računov* in *zaloga* ter *pregled naročil* in računov. Ti procesi prenašajo podatke med entitetami in podatkovnimi zbirkami *Izdelki*, *Narocilo* in *Racuni*, kot vidimo na Slika 4.



Slika 4: Diagram nivoja 0 za nakup

Zdaj pa je potrebno le še razdeliti nekatere procese na podprocesse, kar nam da diagram nivoja 1, ki bo tudi zaključni diagram toka podatkov za nakup. Diagram nivoja 1 s podprocesi je prikazan na Slika 5 in v Prilogi 2.



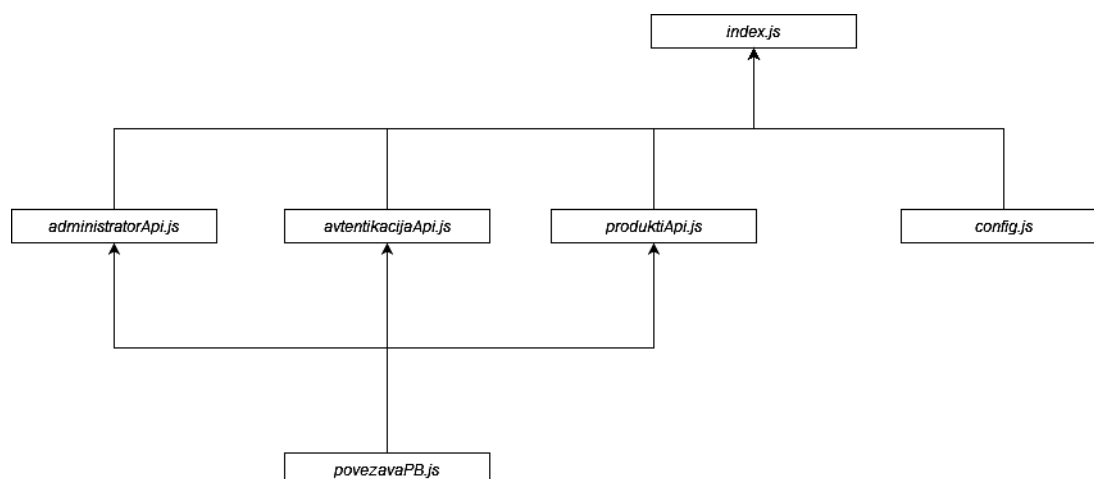
Slika 5: Diagram nivoja 1 za nakup



### 3.3 Diagram zgradbe programa

Da bi dokumentiral programske komponente za lažje razumevanje in odpravljanje napak, sem tekom izgradnje programa izdeloval diagram s komponentami in povezavami med njimi. Aplikacija je zgrajena na osnovi Node.js in njegovih knjižnic ter z JavaScriptom. Za t. i. *back-end* oziroma strežniški del sem uporabil module *Node*, knjižnico *Express.js* pa za izgradnjo vmesnikov uporabniškega programa oziroma API-jev. Za odjemalčevo stran ali *front-end* pa sem uporabil knjižnico *React.js*, s katero se gradi uporabniški vmesnik. Zaradi specifičnosti knjižnice *React* in enostavnosti strežniškega dela sem izdelal model programa, ki ni implementiran s posebnim standardom, saj sem uporabniški vmesnik gradil s pomočjo novejšega pristopa pri *React-u*, t. j. s funkcijami, namesto z razredi. Zato sem ustvaril diagram, ki je nekakšen prirejen razredni UML diagram, saj predstavlja funkcije.

Oglejmo si zgradbo strežniškega dela programa na enostavnem diagramu iz Slika 6.



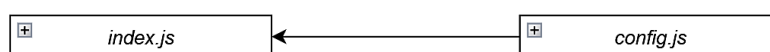
Slika 6: Diagram strežniškega dela

Iz diagrama lahko razberemo strukturo datotek na strežniški strani. Najvišje je datoteka *index.js*, v kateri ustvarimo *Express* aplikacijo in zanjo definiramo vrata (angl. *port*), dodamo mehanizme za dostop do virov in podatkov, razčlenjevanje formatov podatkov, nalaganje datotek in če želimo statično oddajati aplikacijo še dodatno korensko mapo datotek. V datoteki *config.js* imamo shranjene globalne spremenljivke v objektu *global.config*. Datoteke *administratorApi.js*, *avtentikacijaApi.js* in *produktiApi.js* vsebujejo funkcije API, ki dostopajo do podatkovne baze. Podatki za dostop do same baze pa se nahajajo v datoteki *povezavaPB.js*. Puščice prikazujejo uvoze datotek, in sicer v smeri puščic. Tako vidimo, da so v *index.js* uvožene datoteke *administratorApi.js*, *avtentikacijaApi.js*, *produktiApi.js* in *config.js*. V prve

tri pa je uvožena datoteka *povezavaPB.js*. Uvožena datoteka pomeni, da lahko v naši datoteki uporabljamo vse spremenljivke in funkcije iz uvožene datoteke.

Diagram za odjemalcev del predstavlja vse datoteke s funkcijami uporabniškega vmesnika in lastnosti, ki so definirane v njih. Vsak pravokotnik predstavlja datoteko, v primeru, da je v datoteki funkcija, pa so v pravokotniku dodana imena stanj in spremenljivk, ki so v tej funkciji definirana. Najvišje je datoteka *index.html*, ki je začetni HTML dokument za uporabniški vmesnik spletne aplikacije. Ta dokument se najprej naloži v brskalniku, nato pa se izvede še datoteka *index.js*, ki je začetna datoteka za logično upodabljanje uporabniškega vmesnika. V njej definiramo korenski element iz datoteke *index.html* z id-jem »root«, v katerem se bo prikazoval uporabniški vmesnik na podlagi koncepta knjižnice React.js, virtualnega DOM. (27) Prav tako določimo datoteko *App.js*, ki bo prva podrejena komponenta in vsebuje usmerjanje na strani odjemalca in vse ostale datoteke s komponentami. (28)

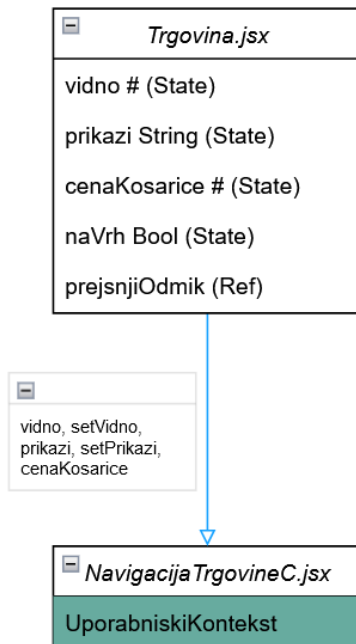
Na datoteko *index.js* je vezana tudi *config.js*, in sicer s črno polno puščico, kar pomeni, da datoteko *config.js* uvozimo v *index.js* in lahko znotraj le te uporabljamo vsebimo uvožene datoteke. V tem primeru uvozimo objekt z globalnimi spremenljivkami iz *config.js*. Podobno povezave s črno puščico v isti namen uporabimo za stilske datoteke CSS, ki jih uvažamo in tako pridobimo dostop do stilov v njih. Poglejmo primer povezave s polno črno puščico na Slika 7, kjer *index.js* uvažava *config.js*.



Slika 7: Povezava "uvažava"

Povezave z modro prazno puščico povezujejo starševske datoteke s podrejenimi datotekami, ki jih vsebujejo. V nasprotni smeri puščice lahko opazujemo uvoze podrejenih datotek v nadrejene, v pravi smeri puščice pa je ponazorjena pot za podajanje lastnosti oziroma stanj iz višje komponente v nižjo ali podrejeno komponento. S tem se podatki prenašajo med komponentami in tako se ob spremembi stanj ponovno naložijo komponente vse do tiste, v kateri je stanje definirano. Na Slika 8 lahko vidimo primer povezave »posreduje« s stanji in metodami za spreminjanje le teh, ki jih iz datoteke komponente oz. funkcije *Trgovina.jsx*

posredujemo datoteki komponente *NavigacijaTrgovineC.jsx*. V tej datoteki omogočimo dostop do posredovanih stanj in metod.



Slika 8: Povezava "posreduje"

Diagram zgradbe programa vsebuje še barvno simboliko, in sicer HTML datoteke so rdeče in CSS datoteke modre barve. Prav tako sta posebej obarvana oba uporabljena konteksta, vsak s svojo barvo, s katero so pobarvane tudi datoteke, v katerih se kontekst uporabi. Barve kontekstov in definicije stanj komponent v datotekah nam omogočajo preprost pogled ponovnega nalaganja komponent ob spremembah stanja. Diagram zgradbe programa si lahko v celoti ogledamo na povezavi iz Priloga 11. Delovanje komponent in kontekstov bomo razložili v nadaljevanju.

## 4 Aplikacija

Spletne aplikacije sestavljata dve strani, imenovani odjemalčeva in strežniška. Obe delujeta v medsebojni komunikaciji, da omogočata delovanje enotne aplikacije. Odjemalčeva stran (angl. *front-end* ali *client-side*) skrbi za prikaz podatkov in prejemanje uporabnikovih vnosov. (29) Implementirana je v programskem jeziku JavaScript s knjižnico React.js za gradnjo uporabniškega vmesnika. Ta knjižnica omogoča lažje oblikovanje vmesnika z zapisom posameznih strani v aplikaciji v obliki komponent, ki v sebi hranijo stanja spremenljivk in JSX prikaza podatkov ter z določanjem stilov prek CSS datotek. Strežniška stran (angl. *back-end* ali *server-side*) je odgovorna za organizacijo in hranjenje podatkov ter dostop do strežniških podatkov, na primer iz podatkovne baze. (29) Ustvarjena je na okolju Node.js z jezikom JavaScript, na katerem delujeta knjižnici vmesne programske opreme Express.js in Axios, s pomočjo katerih lahko dostopamo do podatkov iz baze. Express omogoča dostop do podatkovne baze, Axios pa HTTP poizvedbe za dostop do funkcij Expressa. Strežniški podatki se pridobijo s poizvedbami na podatkovno bazo, kar omogoča sistem za upravljanje podatkovnih baz MySQL.

Izdelava aplikacije je potekala v razvojnem okolju Visual Studio Code, medtem ko sem podatkovno bazo ustvaril v razvojnem okolju MySQL Workbench.

### 4.1 Strežniška stran

Strežniška stran aplikacije (*back-end*) je del aplikacije, ki se nahaja v mapi *server*. Za strežnik, ki sprejema zahteve iz strani odjemalca, bomo uporabili ogrodje Express.js z okoljem Node.js. Express.js prevzame upravljanje z zahtevami, HTTP metodami in oddajanje datotek, česar Node.js kot izvajalno okolje ne zna. Express.js je torej namenjen razvoju API-jev za spletne aplikacije v povezavi z Node.js, ki upravlja s strežniki in potmi. (30)

Poglejmo si, kako delujeta Node in Express skupaj v naši aplikaciji. Celotna koda strežniške strani je na voljo na povezavi v Priloga 10. Potem ko smo namestili modul *express* z ukazom *npm i express*, datoteki *index.js* mape *server*, od koder bomo zagnali strežnik, uvozimo modul *express* in ustvarimo *express* aplikacijo. (31)

*Izsek 6: Kreiranje express aplikacije*

```
import express from 'express';  
const app = express();
```

Objekt *app* sedaj predstavlja Express aplikacijo. Na njem lahko kličemo metode za usmerjanje HTTP zahtev, konfiguracijo vmesne programske opreme, upodabljanje HTML pogledov, registracijo mehanizma predloge in spreminjanja nastavitev aplikacije za nadzor obnašanja. (32) Da bo strežnik dostopen, moramo na objektu *app* klicati metodo `listen()` in ji podati vrata, na katerih posluša zahteve. Vrata pridobimo iz spremenljivk okolja, ki jih bomo razložili v nadaljevanju. Na *app* kličemo še metodo `use()`, ki omogoči uporabo drugih modulov v *express* aplikaciji. Omogočili smo modul *cors*, ki omogoča dostop do virov spletne strani iz različnih domen, *express.json()*, ki razčleni telo (angl. *body*) HTTP zahteve v formatu JSON in *express.urlencoded()*, ki razčleni dohodne zahteve z URL kodiranimi podatki. Dodali smo še modul *fileUpload* za nalaganje datotek, kar potrebujemo pri nalaganju slik. (33) Poleg tega smo dodali poti za naš API in jih povezali z datotekami. Preostanek datoteke *index.js* lahko vidimo na Izsek 7.

*Izsek 7: Dodajanje funkcionalnosti express aplikaciji*

```
app.listen(process.env.PORT, () => {
  console.log(`Express strežnik se izvaja na vratih ${process.env.PORT}`);
});
app.use(cors());
app.use(express.json());
app.use(express.urlencoded({ extended: true }));
app.use(fileUpload());

app.use('/api/produkti', produktiApi);
app.use('/api/avtentikacija', avtentikacijaApi);
app.use('/api/administrator', administratorApi);
```

#### 4.1.1 Express funkcije za HTTP poizvedbe

Z *express* aplikacijo lahko s funkcijami HTTP zahtev, kot so GET, POST, DELETE, povežemo odjemalca s podatkovno bazo. Poizvedbe smo razdelili na tri celote, ki so v datotekah *administratorApi.js*, *avtentikacijaApi.js* in *produktiApi.js*. *AdministratorApi.js* je namenjena zahtevam za administrativne zadeve, *avtentikacijaApi.js* zahtevam za avtentikacijo uporabnikov in *produktiApi.js* zahtevam za upravljanje s produkti v trgovini. V te datoteke smo uvozili *pool*, ki je povezava do podatkovne baze in naredili objekt *router*, ki omogoča, da organiziramo poti iz trenutne datoteke s potmi, ki so vodile do te datoteke in tako dobimo celoten URL za dostop do funkcij API. Poglejmo zahtevo GET iz *administratorApi.js*, ki omogoča vračanje izdelkov glede na določen kriterij, na Izsek 8.

*Izsek 8: Funkcija za express GET zahtevo*

```
import express from 'express';
const router = express.Router();
import pool from '../povezavaPB.js';

router.get('/izdelki', async (req, res) => {
  const kriterij = req.query.iskalniKriterij;
  const niz = req.query.iskalniNiz;

  try { let odziv = await pool.query(`select * from Izdelki where
    ${kriterij} = ?`, [niz]);
    res.status(200).send(odziv[0]);
  } catch (napaka) { console.log(napaka);
    res.status(400).send(`error`);
  } });
```

V metodo, ki se izvede ob poslani zahtevi na URL `/api/administrator/izdelki`, dobimo objekt *req*, ki vsebuje lastnost *query*, ki je prav tako objekt. (34) Vsebuje lastnosti parametrov URL-ja, od katerih uporabimo *iskalniKriterij* in *iskalniNiz*. Tako pridobimo podatke, ki jih uporabimo v SQL stavku. Znotraj `try..catch` bloka izvedemo poizvedbo na bazo podatkov prek objekta *pool* in metode *.query()*. Prvi argument metode *query()* je SQL stavek, v katerega namesto vrednosti vpišemo »?» in vrednosti po vrstnem redu dodamo v tabelo, ki je drugi argument metode *query()*. Ta mehanizem je namenjen preprečevanju tehnike vdora *SQL injection*, ki izkoristi slabo integracijo podatkovne baze in pomanjkanje preverjanja uporabniškega vnosa. (35) Kot lahko vidimo na Izsek 8, za uporabo kriterija nismo uporabili varnega načina. Tega nismo storili, saj ne določamo vrednosti v stavku SQL, ampak sam stavek. To smo sicer storili na varen način, saj je v odjemalčevem delu aplikacije uporabnikov vnos preverjen pred pošiljanjem zahteve. Če se pridobivanje podatkov iz baze izvede brez napak, objektu *res* dodelimo lastnost *status* 200, ki pomeni uspešno HTTP zahtevo, in pošljemo rezultat. Če pride do napake, jo izpišemo in pošljemo njeno sporočilo. (36)

Poglejmo si še metodi za zahteve POST in DELETE, ki smo ju uporabili v aplikaciji, na Izsek 9 in Izsek 10.

*Izsek 9: Funkcija za express DELETE zahtevo*

```
router.delete('/izbrisi', async (req, res) => {
  const uporabnisko_ime = req.query.uporabnisko_ime;
  try { await pool.query(`delete from Uporabniki where uporabnisko_ime =
    ?`, [uporabnisko_ime]);
    res.status(200).send('operacija uspešna');
  } catch (napaka) {
    console.log(napaka);
    res.status(400).send(`error`); } });
```

Zahteva DELETE je zelo podobna zahtevi GET, medtem ko je zahteva POST iz Izsek 10 nekoliko drugačna.

*Izsek 10: Funkcija za express POST zahtevo*

```
router.post('/dodajIzdelek', async (req, res) => {
  const ime = req.body.ime;
  const kategorija = req.body.kategorija;
  const cena_za_kos = parseFloat(req.body.cena_za_kos).toFixed(2);
  const kosov_na_voljo = parseInt(req.body.kosov_na_voljo);
  const kratek_opis = req.body.kratek_opis === 'null' ? null :
req.body.kratek_opis;
  const informacije = req.body.informacije === 'null' ? null :
req.body.informacije;
  const popust = parseInt(req.body.popust);
  let slika = null;
  if (req.files !== null && req.files.slika !== null &&
req.files.slika.data !== null) {
    slika = req.files.slika.data;
  }
  try { await pool.query(`insert into Izdelki values
    (default,?,?,?,?,?,?,?)`, [ime, kategorija, cena_za_kos,
    kosov_na_voljo, kratek_opis, informacije, popust, slika]);
    res.status(200).send('uspešna operacija');
  } catch (napaka) {
    console.log(napaka);
    res.status(400).send(`error`);
  }
});
```

Pri POST zahtevi uporabimo lastnost *body* pri objektu *req*, saj nam prej omenjeni modul *express.json()* razčleni telo HTTP zahteve, ki jo uporabimo. (37) Prav tako uporabimo modul *fileUpload()*, ki nam datoteke iz zahteve shrani v objekt *req.files*, pod imenom, ki ga določimo, ko naredimo zahtevo, kar bomo videli v nadaljevanju. (38) V podatkovno bazo vnesemo vse podatke o izdelku in ob uspešni poizvedbi na podatkovno bazo vrnemo status 200 in sporočilo *'uspešna operacija'*.

Kot smo videli v primerih API funkcij za sprejemanje HTTP zahtev in pošiljanje poizvedb na podatkovno bazo, vsaki funkciji definiramo URL pot. Ker smo API razdelili na več datotek, smo uporabili *express.Router()*. Ta omogoča dodajanje poti do datoteke vozlišču, ki se v tej datoteki nahaja. Ker imamo pri prejšnjem primeru s POST funkcijo pot *'/dodajIzdelek'*, je dejanska pot do vozlišča *'/administratorApi/dodajIzdelek'*. (39)

#### 4.1.2 Axios HTTP poizvedbe

S knjižnico Axios iz odjemalčeve strani dostopamo do funkcij za HTTP zahteve Expressa (*REST endpoints*), prek URL poti na strežniku Express. V naši aplikaciji smo uporabili poizvedbe `axios.get()`, `axios.delete()` in `axios.post()`. V funkcije podamo kot prvi argument URL do vozlišča na strežniku in kot drugi argument parametre. (40) Poglejmo si primer Axios GET zahteve za funkcijo iz Izsek 8 na Izsek 11.

Izsek 11: Axios GET zahteva

```
const pridobiInfoOsebah = async () => {
  try {
    let odziv = await
    axios.get(`http://localhost:${global.config.port}/api/administrator/oseb
e`, { params: { iskalniKriterij: 1, iskalniNiz: 1 } });
    setTabela(odziv.data);
  } catch (napaka) {
    console.log(`Prišlo je do napake: ${napaka}`);
  }
};
```

Parametre podamo kot lastnost *params*, ki je objekt z našimi vrednostmi, ki jih podajamo v objekt *req*. Enako storimo tudi z zahtevo DELETE, le da uporabimo funkcijo `axios.delete()`. Poglejmo še primer zahteve POST, v katero dodamo tudi sliko.

Izsek 12: Axios POST zahteva

```
const podatki = new FormData();
podatki.append('slika', datoteka);
podatki.append('ID_izdelka', predmet.ID_izdelka);
try { await
  axios.post(`http://localhost:${global.config.port}/api/administrator/nal
oziSlika`, podatki, {
    headers: {
      'Content-Type': 'multipart/form-data',
    },
  });
} catch (napaka) {
  console.log(napaka);
}
```

Za pošiljanje slike prek HTTP zahteve moramo uporabiti *FormData*. (41) Vanj vnesemo vse podatke za pošiljanje in glavi HTTP zahteve dodelimo lastnost *'Content-Type': 'multipart/form-data'*. (42) Stanje *datoteka*, v katerem je shranjena slika, dobimo iz vnosnega polja za datoteke, ime, ki ga podamo, je pomembno za dostop v funkciji *expressa*, kot smo predhodno omenili. Oglejmo si kodo vnosnega polja za slike na Izsek 13.



## Izsek 13: Vnosno polje za slike

```

<input style={{ minWidth: '300px' }}
      type='file'
      encType='multipart/form-data'
      name='slika'
      accept='image/gif, image/jpeg, image/png'
      onChange={(e) => {
        setDatoteka(e.target.files[0]);
      }} />

```

Tip vnosnega polja je *files*, ker vnašamo datoteke, prav tako dodamo atribut *accept*, ki poskrbi, da se kot možni vnosi prikažejo le datoteke določenega tipa. Stanje *datoteka*, ki ga oddamo z zahtevo na strežnik, nastavimo z dodajanjem datoteke v to vnosno polje, saj se sproži funkcija *setDatoteka()*, z argumentom, ki je prva datoteka iz vnosnega polja.

Poglejmo si še HTTP zahtevo za pridobivanje slike izdelka, da jo lahko prikažemo v spletni trgovini. V komponenti *VsebinaTrgovineC.jsx* se sproži funkcija *pridobiProdukte()*, če je stanje *niProduktov* enako *true*. Tedaj se najprej ustvari HTTP Axios GET zahteva, ki vrne 6 izdelkov iz baze podatkov. Poglejmo primer pridobljenega izdelka iz baze na Slika 9.

```

▼ (6) [{...}, {...}, {...}, {...}, {...}, {...}] ⓘ
  ▼ 0:
    ID_izdelka: 47
    cena_za_kos: 159.99
    ime: "LCD monitor DELL S2721HN"
    informacije: null
    kategorija: "monitorji"
    kosov_na_voljo: 3
    kratek_opis: null
    popust: 0
    ▼ slika:
      ► data: (39794) [82, 73, 70, 70, 106, 155, 0, 0, 87, 69, 66, 80, 86, 80, 56, 32, 94, 155, 0,
        type: "Buffer"

```

Slika 9: Pregled pridobljenega izdelka iz podatkovne baze v orodju za razvijalce brskalnika

Izdelek ima pod lastnostjo *slika* shranjen objekt tipa *Buffer*. Tega objekta ne moremo neposredno prikazati kot sliko, zato ob vsaki zahtevi po izdelkih izvedemo še eno zahtevo za pridobitev slik izdelkov. Poglejmo programsko kodo na Izsek 14.

## Izsek 14: Pridobivanje slik izdelkov

```

const VsebinaTrgovine = ({ prikazi, setPrikazi, setCenaKosarice,
setVidno, niProduktov, setNiProduktov }) => {
  const pridobiProdukte = useCallback(async () => {
    try {
      //...pridobimo produkte v tabelo odziv
      //...uredimo nekatera stanja glede nalaganja
      // dodamo vsakemu izdelku kolicino v kosarici in sliko
      odziv.forEach(async (element) => {
        let rezultat = await axios.get(
`http://localhost:${global.config.port}/api/administrator/pridobiSliko`,
        {
          method: 'get',
          responseType: 'blob',
          params: { ID_izdelka: element.ID_izdelka, },
        }
      );
      element.kolicina = 0;
      if (rezultat.data.size === 0) {
        element.slika = null;
      } else {
        element.slika = URL.createObjectURL(rezultat.data);
      }
    });
    setPrikazaniProdukti([...prikazaniProdukti, ...odziv]);
  } catch (napaka) {
    // ...
  }
}, [prikazaniProdukti, setNiProduktov]);
}

```

Za vsak element iz prejšnjega odziva oziroma vsak izdelek bomo naredili HTTP GET zahtevo. Dodamo ji objekt z lastnostmi, ki določijo, kakšnega tipa naj bo vrnjena vrednost. To določa lastnost *responseType*, ki jo nastavimo na *blob* (*Binary large Object*). Če je velikost datoteke 0, je slika elementa *null*, saj je ni v bazi podatkov, sicer pa iz objekta *blob* ustvarimo URL, prek katerega dostopamo do objekta, z metodo *URL.createObjectURL()*. (43) Sedaj lahko prikažemo sliko na spletni strani tako, da kličemo lastnost *slika* našega izdelka.



monitorji

Apple Studio Display

Slika 10: Slika izdelka v spletni trgovini

## 4.2 Stran odjemalca

Stran odjemalca je del aplikacije, ki je namenjen uporabniškemu vmesniku in interakciji uporabnika s strežniško stranjo. Nahaja se v mapi *client*. S funkcijami iz knjižnice Axios lahko na odjemalčevi strani dostopamo do funkcij vmesne programske opreme Express.js, s katero dostopamo do podatkovne baze. Odjemalčeva stran je sestavljena iz datotek HTML, CSS in JSX. React aplikacijo za *front-end* razvijamo s pomočjo okolja Node.js, saj je razvoj poenostavljen zaradi vsebovanega upravljalca paketov *npm* in upravitelja odvisnosti modulov oziroma uvozov in izvozov datotek. (44) Na koncu razvoja uporabimo ukaz `npm run build`, ki iz datotek v mapi ustvari mapo *build* z optimizirano produkcijsko različico aplikacije. Oddajanje aplikacije na lokalnem strežniku bomo razložili na koncu poglavja.

Stran odjemalca zajema uporabniški vmesnik, ki je napisan v jeziku JavaScript, s knjižnico React.js. React zahteva poznavanje določenih konceptov, ki jih bomo opisali v nadaljevanju na primerih iz programske kode v aplikaciji. Celotna programska koda odjemalčeve strani je na voljo v mapi GitHub repozitorija, do katerega povezava je dostopna v Prilogi 8.

### 4.2.1 JSX

JSX ali *JavaScript XML* je način zapisa HTML v Reactu, ki zelo olajša kreiranje elementov v JavaScriptu in dodajanje le teh v DOM, saj nam ni potrebno ustvarjati referenc iz datotek JavaScript na datoteko HTML z metodami. (45) Je torej sintaksna razširitev za JavaScript, ki omogoča pisanje oznak, ki so podobne HTML oznakam, znotraj JavaScript datotek. To prinaša bolj berljivo in jedrnato programsko kodo. JSX se je pojavil zaradi povečevanja kompleksnosti programske logike in manjšega števila dejanskih oznak elementov spletnih strani. Zato React združuje programsko logiko in označbe elementov v skupni kodi znotraj komponent. React sicer ni nujno uporabljen s formatom JSX, ker pa temelji na JavaScriptu, je React namenjen prikazovanju dinamičnih informacij v obliki komponent. Komponente so programske funkcije,

ki vračajo oznake, da jih React lahko prikaže v brskalniku. (46) Poglejmo si primer komponente oz. funkcije iz aplikacije na Izsek 15.

Izsek 15: Primer programske komponente

```
const Error = () => {
  return (
    <div className='error'>
      <div className='napaka'>
        <Warning size={30} style={{ marginRight: '10px'
      </div>
      Napaka, stran ni bila najdena
    </div>
    <div>
      <Link to='/'>Domov</Link>
    </div>
  </div>
  );
};
```

Imamo funkcijo *Error*, ki vrača element `<div>`, ki ga React prikaže v brskalniku. Torej so lahko JSX elementi vrednosti spremenljivk. (47) Na tak način so zapisani vsi deli uporabniškega vmesnika aplikacije.

Pri uporabi JSX se moramo držati osnovnih sintaktičnih pravil. Komponenta lahko vrača le en element. Če jih moramo vrniti več, jih je potrebno umestiti v skupen element, po navadi v `<div></div>` ali pa fragment `<></>`. Pri zapisu JSX je pomembno tudi, da so vse oznake vedno zaprte bodisi z zapirajočo značko ali pa je že začetna značka samozapirajoča. Potrebno je paziti tudi na poimenovanja, ki se morajo skladati s pravili poimenovanja za JavaScript, saj se JSX s pomočjo orodja *babel* pretvori v JavaScript. Tako ne smemo uporabljati vezajev in rezerviranih besed, običajno uporabljamo tudi zapisovanje »camelCase«. Pravila JavaScripta onemogočajo popolno enakost s HTML zapisom, kar lahko vidimo tudi iz primera na Izsek 15, ko moramo za dodajanje stilov *div*-u uporabiti atribut *className* in ne za HTML običajni *class* atribut. (46) Prav tako se držimo pravila, da oznake komponent pišemo z veliko začetnico, HTML elemente pa z malo. (47)

Tesna povezanost JavaScripta in oznak nam omogoča dodajanje logike v označbe elementov. V JSX lahko to storimo z uporabo zavrtih oklepajev, v katere zapišemo ime spremenljivke. Tako se vsebina znotraj oznake spreminja ob spremembah vrednosti spremenljivk. Tako pa lahko med označbe dodajamo tudi objekte, in sicer z dvojnimi zavrtimi oklepaji. To je lahko zelo uporabno, na primer pri dodajanju medvrstičnega CSS, kar lahko vidimo na izseku iz programa Izsek 15, pri elementu *Warning*. (48)

#### 4.2.2 Kavli

React je zgrajen na konceptu komponent, ki so izolirane funkcije, ki jih lahko ponovno uporabimo. Poznamo razredne in funkcijske komponente. React kavli (angl. *React Hooks*) so preproste funkcije JavaScripta, ki jih uporabljamo za izolacijo delov za večkratno uporabo od funkcijskih komponent oz. spremenljivk. (49) (50) Pred uvedbo kavljev je bilo potrebno vsa stanja spremenljivk shranjevati v razrednih komponentah, kavli pa to odpravljajo in omogočajo programiranje uporabniškega vmesnika le z uporabo funkcij. Za svoj uporabniški vmesnik sem uporabil le funkcijske komponente s kavli in ne razrednih komponent. Funkcijska komponenta ima lahko v sebi definirana stanja, ali pa ne. Logika stanj (angl. *stateful logic*) so stanja spremenljivk, ki jih s pomočjo kavljev izoliramo od komponente. Tako jih lahko uporabimo na več mestih v programu in ne obstajajo le znotraj funkcije. (51) Kavli v Reactu torej omogočajo shranjevanje stanja komponent, ki jih sicer v funkcijah ni mogoče shranjevati. Poglejmo si vse kavle, ki sem jih uporabil v programu.

##### 4.2.2.1 useState

Kavlj *useState* je najbolj osnoven, saj omogoča shranjevanje stanj. Definiramo jih z imenom spremenljivke in funkcijo za posodabljanje. Funkcija se po konvenciji poimenuje kot *setImeSpremenljivke* in se skupaj z imenom spremenljivke zapiše v destrukuriranje tabele, ki mu priredimo začetno vrednost s funkcijo *useState(vrednost)*, kar lahko vidimo na Izsek 16. (51)

*Izsek 16: Uporaba useState*

```
const Trgovina = ({ Ref }) => {
  const [vidno, setVidno] = useState(0);
  const [prikazi, setPrikazi] = useState('nakupovanje');
  const [cenaKosarice, setCenaKosarice] = useState(0);
  const [naVrh, setNaVrh] = useState(false);
  ...
}
```

Vrednosti vseh spremenljivk *useState* moramo vedno spreminjati s funkcijo za posodabljanje, saj le ta sproži ponovno nalaganje komponente in tako spreminjanje prikazanih vrednosti. Pri uporabi vseh kavljev moramo biti pozorni na to, da se po klicu v funkciji ne izvedejo takoj, ampak gredo v čakalno vrsto po zaporedju klicanj. Zatem ko se komponenta prikaže do konca oziroma se izvede stavek *return*, se najprej izvede funkcija kavla, ki je bil klican prvi, nato tistega, ki je bil klican drugi itd. (52) (53)

Posodabljanje stanj, ki so tabele ali objekti, v Reactu ne sme biti neposredno, ampak prek funkcije *set* z ustvarjanjem novega objekta ali tabele. Stanja, definirana z *useState* obravnavamo kot nespremenljiva (angl. *immutable*), saj ob neposredni spremembi lastnosti objekta ali elementa tabele React ne zazna, da je prišlo do spremembe in komponenta se ne naloži ponovno. Poglejmo primer iz programske kode, kjer objekt, ki je shranjen v stanje, spremenimo z ustvarjanjem novega z operatorjem *spread*. Nov objekt je argument funkcije *set*, ki nastavi novo vrednost stanja. (54)

*Izsek 17: Nastavljanje nove vrednosti objekta*

```
const IzbrisProfila = ({ props }) => {
  ...
  const [sporocilo, setSporocilo] = useState({ sporociloGeslo: '' });

  const preveriObstojece = async () => {
    try {
      let odziv = await
      axios.get(`http://localhost:${global.config.port}/api/avtentikacija/`, {
        params: {
          uporabnisko_ime: uporabnik.uporabnisko_ime,
          geslo: geslo,
        },
      });
      if (odziv.data) {
        setPonovljenoGeslo(false);
        setSporocilo({ ...sporocilo, sporociloGeslo: `` });
      } else {
        setSporocilo({ ...sporocilo, sporociloGeslo: `Nepravilno geslo` });
      }
    } catch (napaka) {
      console.log(napaka);
      setSporocilo({ ...sporocilo, sporociloNapaka: `${napaka.message}
      (${napaka.name})` });
      setNapaka(true);
    }
  };
}
```

Na Izsek 17 smo stanju *sporocilo*, ki je objekt v metodi *preveriObstojece()* priredili nov objekt. Nov objekt smo naredili z uporabo operatorja *...* (*spread*), ki naredi plitvo (angl. *shallow*) kopijo objekta. To pomeni, da kopira lastnosti le do globine enega nivoja, kar pomeni, da je potrebno ob uporabi gnezdene lastnosti *spread* operator uporabiti večkrat. S *spread* operatorjem dobimo kopijo trenutnega stanja, vendar lahko objektu tudi dodamo lastnost ali mu spremenimo obstoječo, kar smo tudi storili, ko smo nastavili lastnosti *sporociloGeslo* novo

vrednost. (54) Podobno spreminjamo tudi stanja, ki so tabele, le da uporabimo namesto zavitih oglete oklepaje. Poglejmo primer na Izsek 18.

*Izsek 18: Spread operator na tabeli*

```
setKategorijeF([...kategorijeF, e.target.value]);
```

Izsek 18 je del kode komponente *NakupovanjeC.jsx*, kjer uporabljamo filtre za filtriranje prikazanih izdelkov. Če označimo potrditveno polje določene kategorije, ki je še ni v tabeli filtrov *kategorijeF*, potem njeno vrednost pridobimo z *e.target.value*, ki vrne vrednost elementa *target*, na katerem se zgodi dogodek *e*. Ker je naš element potrditveno polje z dogodkom *e*, ki se zgodi ob spremembi, dobimo vrednost *value* tega polja in jo dodamo v tabelo. (55) Glede na to tabelo se ustvari poizvedba na podatkovno bazo po filtriranih podatkih.

#### 4.2.2.2 useContext

Kavelj *useContext* je namenjen enostavnejšemu dostopanju komponent do stanj in funkcij. Z njim ustvarimo kontekst, denimo nakupovanje, in vanj shranimo stanja, ki so pomembna za ta del aplikacije. Ko to storimo, moramo ustvariti še ponudnika (angl. *Provider*) tega konteksta, znotraj katerega bo mogoč dostop do vsebovanih stanj. Vse komponente znotraj ponudnika imajo tako dostop do vrednosti konteksta, ne da bi bilo potrebno spremenljivke podajati iz funkcije v funkcijo po drevesni strukturi navzdol. (56) Konteksti so večinoma namenjeni dostopu v razpršenih komponentah na večjih globinah v drevesu komponent, kjer bi bilo potrebno prenašati spremenljivke čez mnogo funkcij. Uporaba konteksta je sicer priporočena le v nekaterih primerih, saj ima tudi negativno plat. Ob spreminjanju vrednosti konteksta se namreč ponovno naložijo prav vsi potomci ponudnika konteksta, kar lahko zelo upočasni aplikacijo. (57)

V svoji aplikaciji sem se odločil za uporabo nakupovalnega in uporabniškega konteksta. Nakupovalni kontekst shranjuje stanje *kosarica*, ki je tabela za hranjenje izdelkov v košarici. Poglejmo izsek programa iz datoteke *NakupovalniKontekst.js*.

*Izsek 19: Nakupovalni kontekst*

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

export const NakupovalniKontekst = createContext(null);

export const NakupovalniKontekstProvider = ({ children }) => {
  const [kosarica, setKosarica] = useState([]);

  const contextValue = { kosarica, setKosarica };

  return <NakupovalniKontekst.Provider
    value={contextValue}>{children}</NakupovalniKontekst.Provider>;
};
```

Na Izsek 19 lahko vidimo, da z uporabo funkcije `createContext()`, v katero podamo privzeto vrednost *null*, ustvarimo instanco konteksta. (58) Privzeta vrednost bo uporabljena v primeru, da iz konteksta skušamo pridobiti vrednosti, ki jih v njem ni. Definiramo še funkcijo `NakupovalniKontekstProvider()`. V njej določimo stanje in ustvarimo objekt *contextValue*, ki bo na voljo komponentam. Funkcija vrne ponudnika vrednosti z atributom *value*, s stanji na voljo in znotraj ponudnika bodo njegovi potomci: `{children}`. (59) Sedaj pa si oglejmo uporabo ponudnika v datoteki *Trgovina.jsx*, in sicer na Izsek 20.

*Izsek 20: Uporaba ponudnika konteksta*

```
import { NakupovalniKontekstProvider } from
  '../.. /contexts/NakupovalniKontekst';

<NakupovalniKontekstProvider>
  <div className='trgovina'>
    <NavigacijaTrgovine vidno={vidno} ... />
    <VsebinaTrgovine setVidno={setVidno} ... />
    ...
  </div>
</NakupovalniKontekstProvider>
```

Ponudnik konteksta vsem potomcem, ki jih vsebuje, omogoča dostop do atributa *values*. Oglejmo si še, kako se dostopa do vrednosti konteksta v komponenti *VsebinaTrgovine* z ukazom *useContext*. (58)



Uporabniški kontekst je sestavljen na isti način, le da ima na voljo druga stanja in dodatno določa uporabniško ime uporabnika s kavljem, ki ga bomo opisali v nadaljevanju.

#### 4.2.2.3 useRef

*useRef* je kavelj, ki vrne objekt z lastnostjo *current*. Ta objekt je spremenljiv, za razliko od stanj *useState*. (51) Omogoča sklicevanje na vrednost, ki ob spreminjanju ne sproži ponovnega nalaganja komponente. To se zgodi, ker React ne ve, če se spremeni navaden objekt JavaScript. Dodamo mu lahko začetno vrednost s funkcijo *useRef(zacetnaVrednost)*, s čimer določimo, kakšna je začetna vrednost lastnosti *current*. (60) Poglejmo primer uporabe *useRef* v funkciji *Trgovina* na Izsek 22.

*Izsek 21: Pridobivanje konteksta z useContext*

```
import { useContext } from 'react';
import { NakupovalniKontekst } from
'../../contexts/NakupovalniKontekst';

const VsebinaTrgovine = ({ prikazi, setPrikazi, setCenaKosarice,
setVidno }) => {
  const { kosarica } = useContext(NakupovalniKontekst);
  ...
}
```

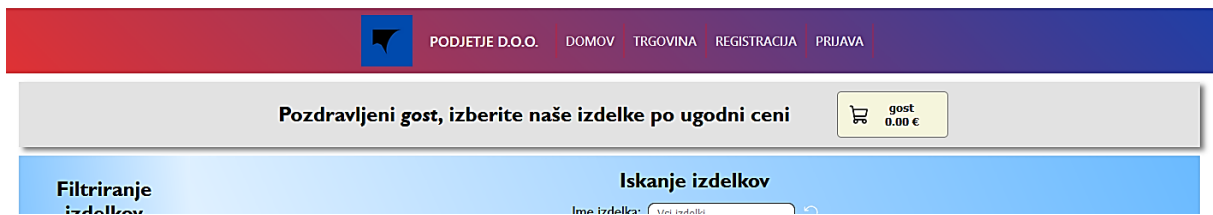
Izsek 22: Uporaba *useRef* za predhodni odmik strani (100)

```
import { useEffect, useState, useRef, useMemo } from 'react';
...
const Trgovina = ({ Ref }) => {
  ...
  const prejsnjiOdmik = useRef(0);

  useEffect(() => {
    if (prikazi === 'nakupovanje') {
      const pomikanje = () => {
        if (naVrh) { setNaVrh(false); }
        let odmikOdVrha = window.pageYOffset;
        if (odmikOdVrha < 235) { setVidno(0); }
        } else if (naVrh) { setVidno(0); }
        } else {
          if (prejsnjiOdmik.current < odmikOdVrha &&
(vidno === 2 || vidno === 0)) { setVidno(2);
          } else { setVidno(0); }
        }
        prejsnjiOdmik.current = odmikOdVrha;
      };
      window.addEventListener('scroll', pomikanje);
      return () => {
        window.removeEventListener('scroll', pomikanje);
      };
    }
  });
  ...
  useEffect(() => {
    if (naVrh) {
      Ref.current.scrollToView({ behaviour: 'smooth' });
    }
  }, [naVrh, Ref]);
}
```

Uporabili smo začetno vrednost 0, kajti komponenta trgovine bo na začetku imela predhodno stanje odmika 0. Nato se izvede vsebina kavlja *useEffect*, ki ga bomo razložili v nadaljevanju. Ta se izvede po vsakem prikazu vrnjenega JSX zapisa in če je vrednost *prikazi* enaka 'nakupovanje', torej če smo na strani za nakupovanje izdelkov, bomo na objekt *window* dodali poslušalca dogodkov (angl. *event listener*). Objekt *window* je JavaScript objekt, ki predstavlja odprto okno v brskalniku. Na okno naše aplikacije tako dodamo poslušalca dogodkov z metodo *.addEventListener(dogodek, funkcija)*. Argument *dogodek* je nujen in predstavlja ime dogodka iz *HTML DOM Event Object* objekta. *Funkcija* je prav tako nujna in se izvede, ko se zgodi dogodek. (61) V primeru na Izsek 22 smo na dogodek *scroll* oz. premikanje gor-dol dodali predhodno definirano funkcijo *pomikanje()*. Slednja se izvede ob

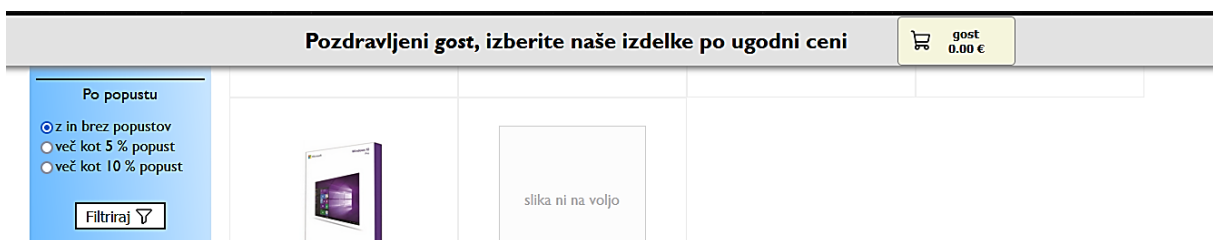
premiku strani in v njej najprej preverimo, ali smo določili vrednost spremenljivke *naVrh* in jo v tem primeru ponastavimo. *naVrh* namreč predstavlja gumb za samodejni premik na vrh strani. Nato nastavimo odmik od vrha z lastnostjo objekta *window*, ki hrani vrednost za odmik v smeri



Slika 11: Prikaz navigacijske vrstice trgovine na vrhu

y. Zatem preverjamo velikost odmika in nastavimo spremenljivko vidno na 0 če želimo prikaz navigacijske vrstice na vrhu, kot kaže Slika 11.

Ob premiku navzdol, pa želimo, da bo navigacijska vrstica vidna nad ostalimi elementi. Da ugotovimo premik navzdol, primerjamo, ali je *prejsnjiOdmik.current* manjši od zdajšnjega odmika. Prikaz navigacijske vrstice trgovine ob pomikanju navzdol lahko vidimo na Slika 12.



Slika 12: Prikaz navigacijske vrstice trgovine ob premikanju navzdol

Na koncu izseka Izsek 22 imamo še kavelj *useEffect*, s katerim ob spremembi spremenljivk *naVrh* ali *Ref* in ob pogoju, da je *naVrh true*, premaknemo pogled strani na element *Ref*. Element *Ref* je komponenta *Trgovina.jsx* pridobila prek podane lastnosti iz komponente *App.js*. *Ref* tako predstavlja element `<div>` iz *App.js*, kot vidimo na Izsek 23.

*Izsek 23: Element za samodejni premik pogleda strani*

```
function App(){
  const zgoraj = useRef('zgoraj');
  return(
    <Router>
      <UporabniskiKontekstProvider>
        <div className='vsebina' ref={zgoraj}>
          <NavigacijskaVrstica />
          <div>
            <Routes>
              ...
              <Route path='/trgovina'
                element={<Trgovina Ref={zgoraj} />}
              />
              ...
            </Routes>
          </div>
          <Noga />
        </div>
      </UporabniskiKontekstProvider>
    </Router>
  );
}
```

Spremenljivka *zgoraj* je referenca na element `<div>` z atributom `ref={zgoraj}`. Spremenljivko prenesemo na komponento `<Trgovina>`, z atributom *Ref*. Zato lahko trgovini uporabimo metodo brskalnikovega API `scrollIntoView()`, ki premakne stran tako, da je na vrhu element, na katerem smo klicali metodo. Dodamo ji lahko še parametre za prikaz na vrhu ali dnu in obnašanje, ki smo ga uporabili na Izsek 22. (62) (63)

Za ta primer smo uporabili *useRef*, ker spreminjanje vrednosti prejšnjega odmika ne sme povzročiti ponovnega nalaganja. V lastnost *current* pa lahko shranimo tudi JSX element. To storimo z določanjem kavlja *useRef* in nato JSX elementu dodamo atribut *ref*. Poglejmo si primer v komponenti *NakupovanjeC.jsx* na Izsek 24.

Izsek 24: Uporaba *useRef* na elementih *JSX*

```

import { useState, useEffect, useRef } from 'react';

const Nakupovanje = ({ props }) => {
  const iskalnoPolje = useRef({});
  ...
  <input ref={iskalnoPolje} style={{ width: 'auto' }}
    className='tekstovnoPolje' placeholder='Vsi izdelki'
    onChange={(e) => {
      e.preventDefault();
      setIskalniNiz(e.target.value);
      setPosodobiIskalnik(true);
    }}
    onFocus={(e) => {
      setFokus1(true);
    }}></input>
  ...
  <div title={predlog.ime} key={predlog.ime} className='predlog'
    onClick={(e) => {
      e.preventDefault();
      e.stopPropagation();
      iskalnoPolje.current.focus();
      setFokus1(true);
      setIskaniIzdelek(predlog.ime);
      setIskalniNiz('');
      iskalnoPolje.current.value = predlog.ime;
    }}
    onFocus={(e) => {
      iskalnoPolje.current.focus();
      setFokus1(true);
    }}>
    {predlog.ime}
  </div>
}

```

Za iskalno polje ustvarimo spremenljivko z uporabo kavlja *useRef* in poimenujemo *iskalnoPolje*. Nato JSX elementu `<input>` dodamo atribut *ref*, z vrednostjo, ki bo spremenljivka *iskalnoPolje*. Zdaj lahko manipuliramo z JSX elementom tako, da na *iskalnoPolje.current* kličemo funkcije brskalnikovega API. V zgornjem primeru smo uporabili metodo `focus()`, ki doda fokus na element. Lahko mu tudi določimo vrednost za prikaz, tako da mu določimo lastnost *value*: `iskalnoPolje.current.value = predlog.ime`, ali pa pridobimo njegovo vrednost z `iskalnoPolje.current.value`. (59) Z določanjem vrednosti smo omogočili, da v funkciji *Nakupovanje* po pridobitvi možnih imen izdelkov glede na napis v iskalnem polju ob kliku na predlog zapišemo besedilo predloga tudi v *iskalnoPolje*. Dogajanje prikazuje Slika 13.



Slika 13: Prikaz uporabe useRef pri iskanju izdelkov

Poleg tega lahko na Izsek 24 zasledimo tudi funkciji `preventDefault()` in `stopPropagation()`. Ti funkciji sta klicani na objektu *e*, ki predstavlja dogodek ob kliku ali fokusiranju elementa. Funkcija `preventDefault()` preprečuje ponovno nalaganje celotne spletne strani s strani brskalnika ob sprožitvi dogodka. Zelo uporaben je na primer tudi če želimo oddati obrazec z gumbom *submit*, a s tem ne želimo povzročiti ponovnega nalaganja strani. (64) To sem uporabil pri obrazcu v komponenti *RegistracijaC.jsx*, ko ne želimo ponovnega nalaganja strani, ker bomo z React-om spremenili komponente na strani. Uporabo lahko vidimo na Izsek 25.

Izsek 25: Uporaba funkcije `preventDefault()` pri obrazcu

```
<form
  ref={obrazec}
  onSubmit={(e) => {
    e.preventDefault();
    poslji(e); }}>
```

Funkcija `stopPropagation()` pa se uporablja tam, kjer si ne želimo nadaljnega razširjanja dogodka na druge komponente. Poglejmo si primer prikaza izdelka, na katerem je gumb za dodajanje v košarico. Ob kliku na gumb se izdelek doda v košarico, ob kliku na celotno polje izdelka pa se prikaže podrobna stran za pregled informacij izdelka oziroma komponente *PodrobnostiC.jsx*. Ker ob kliku na dodajanje v košarico ne želimo odpreti podrobnosti izdelka, bomo uporabili funkcijo `stopPropagation()` na dogodku, ki ga sprožimo s pritiskom na gumb. (65)

Izsek 26: Programska koda za gumb iz *ProduktC.jsx*

```
<button className='dodajVKosarico'
  onClick={(e) => { e.preventDefault();
    e.stopPropagation();
    taProdukt.kolicina++;
    setKosarica([...kosarica, taProdukt]);
  }}> Dodaj v košarico </button>
```



Slika 14: Polje za izdelek v spletni trgovini

Pri kavlju *useRef* moramo paziti, da objekta ne spreminjamo ali beremo med prikazovanjem komponente, saj morajo biti funkcije v React-u čiste, kar pomeni, da morajo za enak vhod vedno vrniti enak izhod. Torej ne morejo zaradi vrednosti objekta *useRef* vračati drugačnih vrednosti.

#### 4.2.2.4 *useEffect*

Je kavelj, s katerim upravljamo s stranskimi učinki, kot so klici API, naročnine, časovniki, mutacije in drugo. (51) Namenjen je sinhronizaciji komponente z zunanjim sistemom. *useEffect* uporabljamo z dvema parametroma, in sicer s parametrom *setup* in opcijskim parametrom *dependencies*. *Setup* je funkcija, ki vsebuje celotno logiko, ki jo želimo izvajati, *dependencies* pa je seznam spreminjajočih vrednosti, ki bodo ob spremembi sprožile ponovno izvajanje *setupa* in so sklicevane znotraj njegove kode. To so lahko dedovane lastnosti, stanja in vse spremenljivke ali funkcije iz telesa komponente. V *setup* lahko dodamo tudi funkcijo *cleanup*, ki se ob spremembi izvede pred ponovnim izvajanjem. Ko se komponenta doda na DOM, React prvič izvede funkcijo *setup useEffect*-a in jo zatem izvaja po vsakem ponovnem nalaganju, če se pri tem spremeni ena od spremenljivk v *dependencies*. Vedno se izvede potem ko funkcija komponente že vrne JSX in se DOM že posodobi. Ob spremembi *dependencies* bo React najprej izvedel *cleanup* funkcijo s starimi vrednostmi, nato pa *setup* z novimi vrednostmi. *Cleanup* funkcija se izvede tudi, ko se komponenta odstrani iz DOM. (66) Primer *cleanup* funkcije lahko vidimo na Izsek 22, kjer odstrani poslušalca dogodkov pred dodajanjem novega v ponovni izvedbi *setup* funkcije. Poglejmo še primer uporabe *useEffect*-a brez *cleanup* funkcije v programski kodi komponente *Profil* na Izsek 27.

Izsek 27: Uporaba *useEffect* za pridobivanje vloge uporabnika

```

import { useContext, useEffect, useState } from 'react';
import { UporabniskiKontekst } from '../contexts/UporabniskiKontekst';
import KrožnoNalaganje from '@mui/material/CircularProgress';
import Skatla from '@mui/material/Box';

const Profil = () => {
  const { uporabnik } = useContext(UporabniskiKontekst);
  const [vloga, setVloga] = useState(null);
  ...
  useEffect(() => {
    const pridobiVlogo = async () => {
      try { let odziv = await
axios.get(`http://localhost:${global.config.port}/api/avtentikacija/vlog
a`, { params: { uporabnisko_ime: uporabnik.uporabnisko_ime, }, });
        setVloga(parseInt(odziv.data));
      } catch (napaka) {
        console.log(napaka);
      }
    };
    pridobiVlogo();
  }, [uporabnik.uporabnisko_ime]);

  if (vloga === null) {
    return (<
      <label>Nalaganje profila ...</label>
      <Skatla sx={{display: 'flex'}} className='nalaganje'>
        <KrožnoNalaganje color='inherit' />
      </Skatla>
    </>
  );
  }
  ...
}

```

V komponenti *Profil* najprej iz konteksta pridobimo uporabnikove podatke in naredimo potrebna stanja z *useState*. Za določanje možnih aktivnosti uporabnika moramo pridobiti njegovo vlogo, ki je zapisana v podatkovni bazi. Ta podatek bomo shranili v stanje *vloga*. Ker je podatkovna baza zunanji sistem, od katerega moramo pridobiti informacijo, je potrebna sinhronizacija s sistemom. Zato uporabimo *useEffect*, v katerem s pomočjo knjižnice *Axios* kličemo funkcijo vmesne programske opreme strežniškega dela, ki pridobi podatke iz baze. Ker lahko pridobivanje vloge traja dlje časa, uporabimo asinhrono funkcijo *pridobiVlogo()*. Znotraj *try...catch* bloka počakamo na razrešitev obljube, ki jo vrača *axios.get()*. S to funkcijo kličemo funkcijo API, na vozlišču */api/avtentikacija/vloga*, in dodamo parameter *uporabnisko\_ime* v objekt *params*. Ko pridobimo informacijo o vlogi, uporabimo



funkcijo *setVloga()*, ki nastavi stanje vloga na pridobljen podatek in povzroči ponovno nalaganje komponente. Preden smo pridobili vlogo, je bila ta *null*, zato je komponenta prikazovala le animacijo za nalaganje profila. Za animacijo nalaganja smo uporabili knjižnico *Reacta*, *Material UI*, iz katere smo uvozili elementa *CircularProgress* in *Box*. Element *CircularProgress* je vrteča krožnica, ki pove uporabniku, da vsebina še ni naložena. Uporabljen Hook *useEffect* je odvisen od spremenljivke *uporabnik.uporabnisko\_ime*, kar pomeni, da se izvede le, ko se spremeni vrednost le te.

#### 4.2.2.5 useCallback

*useCallback* je *React Hook*, katerega namen je ohranjanje istega izvoda funkcije med večkratnim prikazovanjem komponente. Uporabljamo ga, ko funkcij ne izvedemo ob vsakem ponovnem nalaganju komponente, ampak le ob klicu funkcije. Dokler bodo vrednosti *dependencies* enake, se bo ob klicu vračal in izvajal isti izvod funkcije. (67) Poglejmo primer uporabe iz Izsek 28.

*Izsek 28: Uporaba useCallback v VsebinaTrgovineC.jsx*

```
import { useCallback, ... } from 'react';
const VsebinaTrgovine = ({ ... }) => {
  const pridobiProdukte = useCallback(async () => {
    try { let odziv = await
    axios.get(`http://localhost:${global.config.port}/api/produkti/`, {
      params: { steviloIzdelkov: 6, brezPodvajanja:
    prikazaniProdukti.map((a) => a.ID_izdelka),
      },
    });
    ...
  } catch (napaka) { console.log(napaka); setNapaka(true);
    setPrikazaniProdukti([]);
  }
  }, [prikazaniProdukti, setNiProduktov]);

  useEffect(() => {
    if (niProduktov) {
      if (stNalaganj.current === 0) {
        setPrikazaniProdukti([]);
        stNalaganj.current++;
        pridobiProdukte();
      }
    }
  }, [kosarica, setCenaKosarice, niProduktov, pridobiProdukte, stNalaganj,
    napaka, setNiProduktov]);
```

Iz Izsek 28 lahko razberemo, da se funkcija `pridobiProdukte()` ne izvede z vsakim ponovnim nalaganjem komponente, zato je obdana z *useCallback*. Njene *dependencies* so stanja *prikazaniProdukti* in *setNiProduktov*, kar pomeni, da če se spremeni tabela prikazanih produktov, se bo ustvarila nova instanca funkcije `pridobiProdukte()`, *setNiProduktov* pa je že sama po sebi funkcija. `pridobiProdukte()` kličemo znotraj *useEffect*, a le pod pogojem, da je vrednost *niProduktov true*, torej ko nimamo produktov za prikaz. Z *useCallback* smo tako omogočili, da funkcijo kličemo le ob določenem pogoju.

#### 4.2.3 Usmerjanje in navigacija

React Router je knjižnica JavaScripta, ki omogoča, da v Reactu ustvarimo kompleksne aplikacije odjemalčeve strani. Z njim definiramo poti URL povezave, na katerih bodo prikazane določene komponente. Z *React Router* se poenostavi dodajanje novih funkcionalnosti in izboljša berljivost in struktura programske kode. Tako lahko ustvarimo aplikacijo z eno spletno stranjo, saj se komponente na njej menjajo glede na poti (*routes*).

Običajno spletno aplikacijo gradimo z več stranmi, ki se nanašajo na različne vsebine. Med njimi se uporabnik lahko pomika, zato moramo te strani nekako povezati med seboj. To storimo z definiranjem poti vsaki komponenti, ki predstavlja svojo spletno stran. Med stranmi lahko preklapljam s povezavami, ki preklopijo URL na željeno pot za ogled komponente. Za uporabo React Routerja, ga naložimo z ukazom *npm i react-router-dom*. Zatem ga uvozimo v datoteko *App.js*, kjer bomo ustvarili vse poti za usmerjanje in od koder bomo dosegli vse komponente v aplikaciji. (68) Poglejmo si datoteko *App.js* in v njej določene poti za *Router* na Izsek 29.

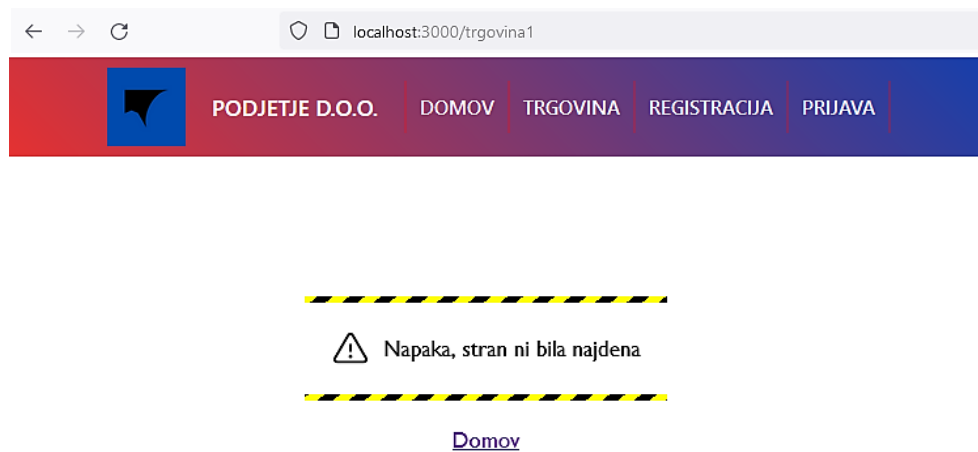
Izsek 29: Prikaz poti React Routerja v App.js

```

...
import { BrowserRouter, Routes, Route } from 'react-router-dom';
...
function App() {
  const zgoraj = useRef('zgoraj');
  return (
    <BrowserRouter>
      <UporabniskiKontekstProvider>
        <div className='vsebina' ref={zgoraj}>
          <NavigacijskaVrstica />
          <div>
            <Routes>
              <Route path='/' element={<Domov />} />
              <Route path='/oNas' element={<ONas />} />
              <Route path='/avtentikacija' element={<Avtentikacija/>} />
              <Route path='/trgovina' element={<Trgovina Ref={zgoraj}
/>} />
              <Route path='/*' element={<Error />} />
            </Routes>
          </div>
          <Noga />
        </div>
      </UporabniskiKontekstProvider>
    </BrowserRouter>
  );
}
export default App;

```

Glavna komponenta za navigacijo v aplikaciji je *BrowserRouter* komponenta. Ta shrani trenutno lokacijo v naslovno vrstico brskalnika z URL-ji in krmari s pomočjo vgrajenega brskalnikovega sklada zgodovine. (69) Znotraj *BrowserRouterja* je komponenta *Routes*, ki ob vsaki spremembi lokacije pregleda vse svoje potomce in najde ujemajočo pot. (70) Potomci komponente *Routes* pa so komponente *Route*, ki povezujejo URL poti s komponentami za prikaz. Več *Route*-ov lahko gnezdimo in kompleksna sestava aplikacije postane enostavnejša. *Route* ima atribut *path*, v katerega podamo URL pot, kot smo to storili na Izsek 29. Na primer za URL `'/trgovina'` bo *BrowserRouter* vrnil atribut *element*, ki je komponenta `<Trgovina>`. Za oblikovanje poti imamo možne tudi dinamične segmente z uporabo `>><<` in neobvezne segmente z uporabo `>>?<<`, ki jih sicer v aplikaciji nismo uporabili. (71) Smo pa uporabili segment *star*, ki poveže vse poti, ki imajo namesto zvezde v sebi katerekoli znake. Ta segment smo uporabili za prikaz strani z napako, če vnesemo napačen URL. Poglejmo primer iz Slika 15, na katerem smo vnesli v URL pot `'/trgovina1'`, ki ji nismo posebej definirali komponente, razen s segmentom `"/*`, ki prikaže za vse ostale poti komponento *Error*. (72)



Slika 15: Komponenta Error ob napačnem URL

Vsi elementi, ki so na Izsek 29 zunaj *Routes*, bodo prisotni na vsaki strani, ker jih ne preklapljam. Tako bosta *NavigacijskaVrstica* in *Noga* na vsaki strani, ne glede na URL pot. (68)

*React Router Link* je komponenta, ki nadomešča oznako `<a>` pri navigaciji med stranmi aplikacije. Prednost komponente *Link* je ta, da namesto ustvarjanja povezav strežniške strani, ki jih ustvarja `<a>`, raje uporabimo usmerjanje na strani odjemalca. Tako se ob kliku na `<Link>` v aplikaciji na odjemalčevi strani preveri pot in naloži komponento, ki ji ustreza. Tak način je hitrejši in porablja manj virov, ker ne nalagamo spletne strani za vsak klik na povezavo. (73) Poglejmo si primer uporabe komponent *Link* iz komponente *NavigacijskaVrsticaC.jsx* na Izsek 30.

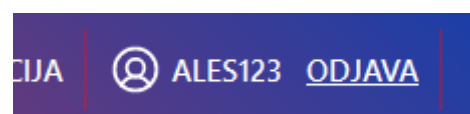
Izsek 30: Uporaba *Link* v navigacijski vrstici

```
import { Link } from 'react-router-dom';
...
const NavigacijskaVrstica = () => {
  ...
  const { uporabnik, jeAvtenticiran, setJeAvtenticiran } =
    useContext(UporabniskiKontekst);
  return (
    ...
    <Link to="/" className='linki'> Domov </Link>
    ...
    <Link to='/avtentikacija'
      state={{ prikazAvtentikacija: 'prijava',
        sporocilo: jeAvtenticiran ? sporocilo.sporocilo1 : '', }}
      className='linki'>
      {jeAvtenticiran ? ( <
        <UserCircle size={28} style={{ marginRight: '4px' }} />
        <label style={{ marginRight: '12px' }}>
          {uporabnik.uporabnisko_ime} </label>
        <label onClick={e => {
          e.preventDefault();
          e.stopPropagation();
          setJeAvtenticiran(false); }}
          style={{ textDecoration: 'underline' }}> Odjava </label>
        </> ) : ( 'Prijava' ) }
      </Link>
    );
  };
};
```

Na zgornjem izseku programa lahko vidimo dva primera uporabe komponente *Link*. Prvi primer je enostaven, saj klik nanj vodi na pot, določeno z atributom *to*, torej s klikom nanj pridemo na komponento *Domov*, kar lahko razberemo iz Izsek 29. V drugi *Link* komponenti pa smo za pot določili *'/avtentikacija'*, poleg tega pa tudi atribut *state*, saj želimo s prehodom na drugo komponento prenesti določene informacije s sabo. V *state* vnesemo objekt z dvema lastnostma, *prikazAvtentikacija* in *sporocilo*. Objekt se bo shranil ob kliku na povezavo *Link* in bo dostopen v komponenti, kamor se premaknemo. Znotraj *Link*-a je dodan še pogojni operator, ki v primeru, da je uporabnik že *avtentificiran*, prikaže uporabniško ime in *Link* za odjavo, sicer pa le *Link* za prijavo, kar lahko vidimo na Slika 16.



Neprijavljen uporabnik - gost



Prijavljen uporabnik – ales123

Slika 16: *Link* za prijavo ali odjavo

Poglejmo, kako lahko dostopamo do podatkov, ki smo jih v obliki objekta atributu *state* predali *Link*-u iz Izsek 30. Po kliku na *Link* za prijavo ali odjavo se zaradi atributa to premaknemo na komponento *Avtentikacija*, kar je vidno iz Izsek 29 in Izsek 30. Poglejmo si programsko kodo komponente *Avtentikacija*.

Izsek 31: Uporaba *useLocation* v *Avtentikacija*

```
import { useLocation } from 'react-router-dom';
...
const Avtentikacija = () => {
  const location = useLocation();
  const { jeAvtenticiran } = useContext(UporabniskiKontekst);

  return (
    <div>
      {jeAvtenticiran ? (
        <Profil />
      ) : location.state.prikazAvtentikacija === 'prijava' ? (
        <Prijava />
      ) : location.state.prikazAvtentikacija === 'registracija' ? (
        <Registracija />
      ) : ( <Error /> )}
    </div>
  );
};
export default Avtentikacija;
```

V funkciji *Avtentikacija()* smo z uporabili kavelj *useLocation()*. *useLocation()* vrne objekt *location*, ki vsebuje vse informacije o trenutnem URL. Z vsako spremembo URL bo vrnjen nov objekt *location*. Ta objekt ima lastnosti *hash*, *pathname*, *search*, *state* in *key*. V navigacijski vrstici na Izsek 30 smo nastavili atribut *state* na objekt, ki ga sedaj lahko pridobimo z *location.state*, kar smo tudi storili na Izsek 31. (74) (75) S tem smo pridobili vrednost lastnosti *prikazAvtentikacija*, da prikažemo komponento *Prijava*.

V aplikaciji pa se lahko pojavi potreba po navigaciji na drugo komponento tudi po končanem opravilu, ki ne zahteva klika na gumb ali povezavo. Tedaj uporabimo kavelj *useNavigate*, s katerim lahko preidemo na katerokoli URL pot. Mogoče je tudi prehajanje na prejšnjo in naslednjo stran. Poglejmo primer uporabe *useNavigate()* v aplikaciji na Izsek 32.

Izsek 32: Uporaba *useNavigate* v *IzbrisProfilaC.jsx*

```
import { useNavigate } from 'react-router-dom';
...
const IzbrisProfila = ({ props }) => {
  const navigate = useNavigate();
  ...
  navigate('/', { state: {
    sporočilo: `račun ${uporabnik.uporabnisko_ime} izbrisan`
  }
});
  ...
}
```

Podobno kot s komponento *Link*, lahko z *navigate* prehajamo na druge URL in za prehod shranimo podatke v lastnosti *state*, kot smo to storili na zgornjem primeru. Do podatkov nato dostopamo z *useLocation()*. (76)

Ker se z *Link* komponento in *navigate* lahko premikamo le med lokalnimi potmi, moramo za odpiranje zunanjih povezav uporabiti metodo brskalnikovega API za objekt *window*, *open()*. To smo storili v nogi strani, kjer smo dodali povezave na družabna omrežja podjetja. Primer povezav za odpiranje zunanjih povezav je podan na Izsek 33.

Izsek 33: Uporaba zunanjih povezav

```
const Noga = () => {
  ...
  <button onClick={(e) => {
    window.open('https://www.facebook.com', '_blank');
  }}>
    <FacebookLogo size={25} />
  </button>
  ...
}
```

#### 4.2.4 Prenašanje lastnosti

Med funkcijami komponent Reacta pogosto potrebujemo prenašanje lastnosti (angl. *props* oz. *properties*) iz nadrejene komponente v podrejeno. To storimo tako, da podatke podamo kot vrednosti atributov podrejene komponente. Poglejmo primer prenašanja lastnosti iz komponente *PrikazProduktovC.jsx* v komponento *ProduktC.jsx* na Izsek 34.

*Izsek 34: Podajanje lastnosti nadrejene komponente*

```

const PrikazProduktov = ({ setFokus1, props, stVsehProduktov, filtriraj,
  filtri, kategorijeF }) => {
  ...
  {props.prikazaniProdukti.map((produkt) => {
    return (
      <Produkt
        key={produkt.ID_izdelka}
        setVidno={props.setVidno}
        setPrikazi={props.setPrikazi}
        taProdukt={produkt}
        setIzbranProdukt={props.setIzbranProdukt}
        setIzKosarice={props.setIzKosarice}
      />
    );
  })} }

```

V komponenti *PrikazProduktov.jsx* imamo tabelo vseh produktov, ki jih bomo prikazali, zato s funkcijo `map()` ustvarimo novo tabelo, ki bo imela iste elemente, vendar v telesu *callback* funkcije vračamo komponento `<Produkt>`, ki ji dodamo attribute z vrednostmi stanj iz zdajšnje komponente `<PrikazProduktov>`. S tem omogočimo dostop do teh vrednosti v podrejeni komponenti, kar lahko vidimo na Izsek 35 in Izsek 36.

*Izsek 35: Prvi način sprejemanja lastnosti*

```

const Produkt = ({ setPrikazi, taProdukt, setIzbranProdukt,
  setIzKosarice, setVidno }) => {
  ...
  <div className='poljeProdukta' onClick={(e) => {
    e.preventDefault();
    setIzbranProdukt(taProdukt);
    setPrikazi('produkt');
    setVidno(0);
    setIzKosarice(false);
  }}>
  ... }

```

*Izsek 36: Drugi način sprejemanja lastnosti*

```

const Produkt = (props) => {
  ...
  <div className='poljeProdukta' onClick={(e) => {
    e.preventDefault();
    props.setIzbranProdukt(props.taProdukt);
    props.setPrikazi('produkt');
    props.setVidno(0);
    props.setIzKosarice(false);
  }}>
  ... }

```



V obeh primerih smo prenesli lastnosti iz nadrejene v podrejeno komponento, pri čemer je razlika le v klicu lastnosti in metod. Pri prenašanju lastnosti se torej ustvari objekt *props*, v katerem so kot lastnosti shranjeni podatki nadrejene komponente. (77) Do teh lahko dostopamo tako, da v oklepajih destrukuriramo objekt in pridobimo iz njega posamezne lastnosti, kot smo storili na Izsek 35, ali pa v funkcijo vnesemo sam objekt *props* in nato kličemo vsako lastnost posebej. (78)

#### 4.2.5 Dodajanje stilov s CSS

Kot smo že omenili, lahko s CSS datotekami dodajamo stilske lastnosti HTML in JSX značkam. Za namen dodajanja stilov sem v aplikaciji komponentam, katere sem želel oblikovati, dodal atribut *className*. Nato sem se na ime skliceval v datotekah *.css*. Povezavo lahko razberemo iz izsekov Izsek 38 in Izsek 37.

*Izsek 38: Element <div> z atributom className='poljeProdukta'*

```
const Produkt = (props) => {  
  ...  
  return (  
    <div className='poljeProdukta'  
      onClick={(e) => {  
        e.preventDefault();  
        props.setIzbranProdukt(props.taProdukt);  
        props.setPrikazi('produkt');  
        props.setVidno(0);  
        props.setIzKosarice(false);  
      }}>
```

*Izsek 37: Stilska določila za elemente z atributom className='poljeProdukta'*

```
.poljeProdukta {  
  display: inline-block;  
  justify-content: center;  
  align-items: center;  
  border: 1px solid #f0f0f0;  
  margin: 0px;  
  padding: 30px;  
  width: 190px;  
  height: 475px;  
  background-color: white;  
  text-overflow: ellipsis;  
}
```

Iz zgornjih izsekov vidimo, kako se poveže element s stilsko določbo z uporabo razredov. Za določanje stila razredu, uporabimo CSS selektor, sestavljen iz pike in imena razreda. Ta stil lahko uporabimo na več elementih, ki jim je določen isti razred. (2)

#### 4.2.6 Priprava za produkcijo in lokalno oddajanje

Ko ustvarimo različico za produkcijo z ukazom `npm run build` v mapi *client*, jo lahko oddajamo na strežniku kot statično spletno aplikacijo. Uporabili bomo paket *npm*, imenovan *serve*, ki omogoča prav to. Naložimo modul z ukazom `npm i -g serve` in aplikacijo oddajamo z ukazom v ukazni vrstici iz Izsek 39. (79)

*Izsek 39: Ukaz za statično oddajanje aplikacije*

```
serve -s build -l 3010
```

Ukaz vsebuje mapo, ki jo oddaja in vrata (*port*), na katerih jo oddaja. (80) Tako zaženemo odjemalčevo stran.

Za zaganjanje celotne aplikacije moramo uporabiti tudi ukaz `node index.js`, ki zažene strežniško stran aplikacije iz ukazne vrstice. Nato odpremo brskalnik na URL-ju: <http://localhost:3010/>, kjer lahko vidimo aplikacijo.

Za hitrejši zagon aplikacije lahko uporabimo *batch* datoteke, ki jih uporabljamo za avtomatizacijo ponavljajočih procesov. (81) Za zagon našega strežnika za oddajanje spletne aplikacije odjemalca in aplikacije strežniške strani bomo torej ustvarili datoteko *AplikacijaMatura.bat*. Poglejmo si njeno vsebino na Izsek 40.

*Izsek 40: Batch datoteka za lokalni zagon aplikacije*

```
@echo off
cd "C:\VisualStudioCode\AplikacijaMatura\server\"
start node index.js
cd "C:\VisualStudioCode\AplikacijaMatura\client"
start serve -s build -l 3010
start firefox http://localhost:3010
exit /B
```

Ukaz `@echo off` poskrbi, da ukaz te datoteke ni prikazan v ukaznem pozivu ob izvedbi. Sledi ukaz za pomik v mapo *server*, kjer z ukazom `start node index.js` zaženemo strežniško stran aplikacije. Nato se pomaknemo še v mapo *client* in zaženemo strežnik za oddajanje vsebine mape *build*, kjer se nahaja odjemalčeva aplikacija. Zatem zaženemo še brskalnik na lokalnem URL-ju in končamo *batch* skript z ukazom `exit /B`. (82) Tako smo zagnali aplikacijo in jo lahko začnemo uporabljati v brskalniku.

### 4.3 Podatkovna baza

Za ustvarjanje podatkovne baze sem uporabil sistem za upravljanje s podatkovno bazo MySQL. Za delo s podatkovno bazo potrebujemo strežnik MySQL in orodje za komunikacijo z njim. Za našo podatkovno bazo smo uporabili *MySQL Workbench*, ki ponuja dostop do podatkovne baze in urejevalnik za razvoj SQL stavkov.

Za začetek razvoja podatkovne baze v orodju *MySQL Workbench* ustvarimo povezavo z bazo. To storimo prek okna, v katerega vpišemo ime povezave, uporabniško ime, geslo in dodatne nastavitve, ali pa za lokalno uporabo uporabimo že nastavljene. Konfiguriramo lahko tudi upravljanje s strežnikom in nato testiramo ter potrdimo povezavo. (83)

Ko smo nastavili povezavo s podatkovno bazo, lahko s stavki SQL ustvarimo novo shemo, tabele in relacije med njimi. Ker imamo že modelirano strukturo podatkovne baze, je potrebno le vse ustvariti v jeziku SQL. V Priloga 11 si na povezavi do *GitHub* repozitorija lahko ogledamo SQL stavke za kreiranje tabel.

Kot lahko vidimo na ER modelu iz Priloga 1, moramo ustvariti 6 tabel. Najprej ustvarimo podatkovno bazo in se povežemo nanjo, z ukazoma iz Izsek 41. Nato sledi ustvarjanje tabel. Tabela *Uporabniki* bo shranjevala uporabniška imena, šifrirana gesla, številke vlog in logično vrednost, ki pove, ali je uporabnik omogočen. Ustvarimo še tabelo *Stranke\_in\_zaposleni*, ki shranjuje podrobne podatke oseb. Ti podatki so *ID*, uporabniško ime, elektronski naslov, ime, priimek, ulica in hišna številka, kraj, poštna številka, telefonska številka, podjetje, oddelek in plača. Primarni ključ je *ID*, tuji ključ pa je *uporabnisko\_ime*, ki se povezuje s tabelo *Uporabniki*, kjer je *uporabnisko\_ime* primarni ključ. Povezava med tabelama omogoča kaskadno brisanje in posodabljanje. Poglejmo SQL stavek za ustvarjanje tabele *Stranke\_in\_zaposleni* na Izsek 42.

*Izsek 41: Ustvarjanje baze podatkov*

```
CREATE DATABASE bazaMatura;  
USE bazaMatura;
```

Izsek 42: SQL stavki za ustvarjanje tabele *Stranke\_in\_zaposleni*

```
CREATE TABLE IF NOT EXISTS Stranke_in_zaposleni (
    ID INT PRIMARY KEY AUTO_INCREMENT,
    uporabnisko_ime VARCHAR(150) DEFAULT NULL,
    elektronski_naslov VARCHAR(160) UNIQUE NOT NULL,
    ime VARCHAR(120) NOT NULL,
    priimek VARCHAR(130) NOT NULL,
    ulica_in_hisna_stevilka VARCHAR(160) DEFAULT NULL,
    kraj VARCHAR(130) DEFAULT NULL,
    postna_stevilka INT DEFAULT NULL CHECK (postna_stevilka < 10000
        AND postna_stevilka > 0),
    telefonska_stevilka VARCHAR(20) DEFAULT NULL,
    podjetje VARCHAR(255) DEFAULT NULL,
    oddelek VARCHAR(100) DEFAULT NULL,
    placa DOUBLE DEFAULT 0.00,
    CONSTRAINT fk_uime FOREIGN KEY (uporabnisko_ime)
        REFERENCES Uporabniki (uporabnisko_ime)
        ON DELETE CASCADE ON UPDATE CASCADE
);
```

Ustvarimo še tabelo *Narocila*, ki shranjuje *ID\_narocila*, datum, *ID\_stranke*, logično vrednost za opravljenost naročila, ime in priimek stranke, naslov za dostavo in ceno poštnine. *ID\_narocila* je primarni, *ID\_stranke* pa tuji ključ, ki se navezuje na *ID* iz tabele *Stranke\_in\_zaposleni*. Vsak račun bo dobil tudi svoj račun, zato ustvarimo tudi tabelo *Racuni* s primarnim ključem *ID\_racuna* in tujim ključem *ID\_narocila*. Poleg tega imamo še kupca, znesek za plačilo in datum izdaje. Ker ima naročilo enega ali več izdelkov, ustvarimo še tabelo *Izdelki*. V njej hranimo *ID\_izdelka*, ime, kategorijo, ceno za kos, število kosov na voljo, kratek opis izdelka, celotne informacije, odstotek popusta in sliko. Potrebno je le še povezati tabeli *Narocila* in *Izdelki* tako, da bo lahko pri enem naročilu več izdelkov. Zato ustvarimo še zadnjo tabelo *Izdelki\_pri\_narocilu*, ki nima primarnega ključa, saj služi le povezavi med tabelama. Ima pa zato dva tuja ključa, in sicer *ID\_narocila*, ki se navezuje na tabelo naročil in *ID\_izdelka*, ki se navezuje na tabelo izdelkov. Tako lahko shranimo več ID-jev izdelka z istim ID-je naročila.

Ko smo naredili vse tabele, moramo vnesti le še podatke za administratorja baze. Ta bo lahko prek aplikacije dostopal do nje in dodajal ostale uporabnike in izdelke. Uporabimo dva ukaza, ki ju lahko vidimo na Izsek 43.

*Izsek 43: SQL stavka za dodajanje administratorja podatkovne baze*

```
INSERT INTO Uporabniki
VALUES ('admin', '8451029959740982', 0, default);

INSERT INTO Stranke_in_zaposleni
(uporabnisko_ime, elektronski_naslov, ime, priimek)
VALUES ('admin', 'admin', 'admin', 'admin');
```

V tabelo *Uporabniki* dodamo uporabnika z uporabniškim imenom *admin* in geslom *admin*, le da je geslo šifrirano s funkcijo `cyrb53Hash()`, ki pretvori niz znakov v 53-bitno šifro. (84) (85) Dodelimo mu vlogo 0, saj bo administrator in pustimo, da je profil omogočen. V *Stranke\_in\_zaposleni* dodamo administratorju zahtevana polja z vrednostmi *admin*. S tem smo vzpostavili podatkovno bazo, ki je kompatibilna z aplikacijo in jo lahko takoj pričnemo uporabljati.

Ker bomo bazo in aplikacijo uporabljali na lokalnem računalniku, bomo v datoteki *.env* v mapi *server* dodali spremenljivke okolja (angl. *environment variables*). Spremenljivke okolja so spremenljivke, ki so na voljo programu med izvajanjem in jih je mogoče spreminjati glede na okolje, kjer se program izvaja. Uporabljamo jih iz razlogov ločevanja spremenljivk od glavnega programa in uporabo, ko jih potrebujemo, zaradi lažjega prehoda na drugo množico podatkov za konfiguracijo, ki je sicer skoraj nemogoč in zaradi zavarovanja občutljivih podatkov, kot so ključi, podrobnosti o prijavi, podatkovni bazi itd. (86)

Ker strežnik in odjemalec tečeta vsak v svojem procesu, naredimo *.env* le v mapi strežnika *server*. V mapi *server* torej ustvarimo datoteko *.env*, od koder bo aplikacija pridobila spremenljivke okolja med izvajanjem. Za ta korak potrebujemo modul *dotenv*, ki naloži vsebino datoteke *.env* v program. Za inicializacijo paketa dodamo v datoteki iz mape *server*: *index.js* in *povezavaPB.js*, v katerih bomo spremenljivke uporabili, vrstice iz Izsek 44.

*Izsek 44: Povezava .env s programskimi datotekami*

```
import dotenv from 'dotenv';
dotenv.config();
```

Poglejmo si datoteko *.env* na Izsek 46.

*Izsek 46: Spremenljivke okolja v .env*

```
# NODE APP ENVIRONMENT VARIABLES
# NODE_ENV=development
PORT=3005

# DATABASE CONNECTION ENVIRONMENT VARIABLES
PB_HOST=localhost
PB_PORT=3306
PB_UPORABNIK=root
PB_GESLO=rootsql
PB_IME=bazaMatura
```

V *.env* imamo spremenljivke za shranjevanje vrat (angl. *port*), na katerih bo oddajal strežnik *express*. Prav tako imamo shranjene spremenljivke za dostop do podatkovne baze, in sicer ime gostitelja, vrata za dostop, ime in geslo uporabnika in ime podatkovne baze. Do teh spremenljivk v programu v mapi *server* dostopamo prek objekta *process.env*, kar lahko vidimo na Izsek 45.

*Izsek 45: Uporaba spremenljivk okolja za povezavo s podatkovno bazo*

```
const pool = createPool({
  host: process.env.PB_HOST,
  port: process.env.PB_PORT,
  user: process.env.PB_UPORABNIK,
  password: process.env.PB_GESLO,
  database: process.env.PB_IME,
  connectionLimit: 10,
  dateStrings: true,
});
```

Na Izsek 45 smo ustvarili povezavo s podatkovno bazo, ki nam omogoča predhodno naložen modul *mysql2*. (87) Vanj moramo vnesti podatke, specifične za našo podatkovno bazo, ki jih pridobimo iz *.env*. Če bi tako želeli spremeniti podatkovno bazo, bi to zlahka storili s spreminjanjem vrednosti spremenljivk okolja v *.env*.

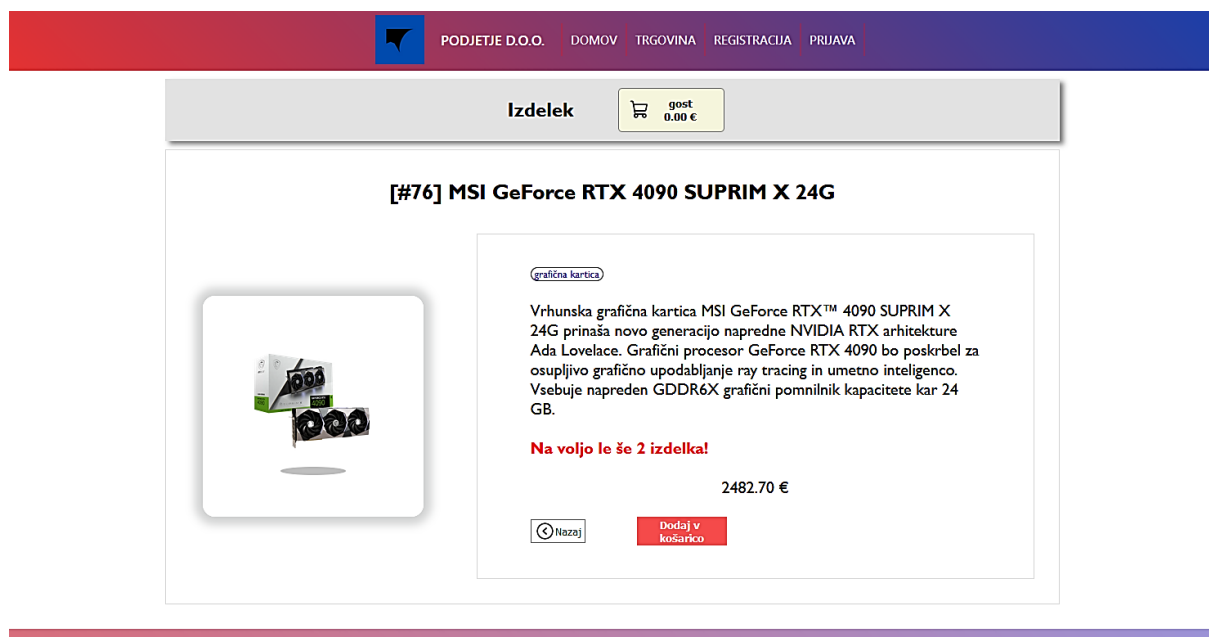
## 5 Testiranje

Preizkušanje ali testiranje je ključna faza pri razvoju aplikacije, saj lahko šele s testiranjem potrdimo funkcionalnost aplikacije v večini primerov. S testiranjem v program vnašamo podatke, ki so pričakovani in tudi take, ki niso pričakovani, se pa lahko zgodijo pri uporabi. S testiranjem ugotovimo, na katere scenarije se program ne odzove pravilno in s tem odkrijemo napake v programu. Testiranje aplikacije sem izvajal že med samim programiranjem, saj sem vsako novo funkcionalnost preveril sproti, kar mi je omogočal način izdelave programa od zgoraj navzdol. Zadnji preizkus pa bom izvedel in ga opisal v tem poglavju.

### 5.1 Spletna trgovina

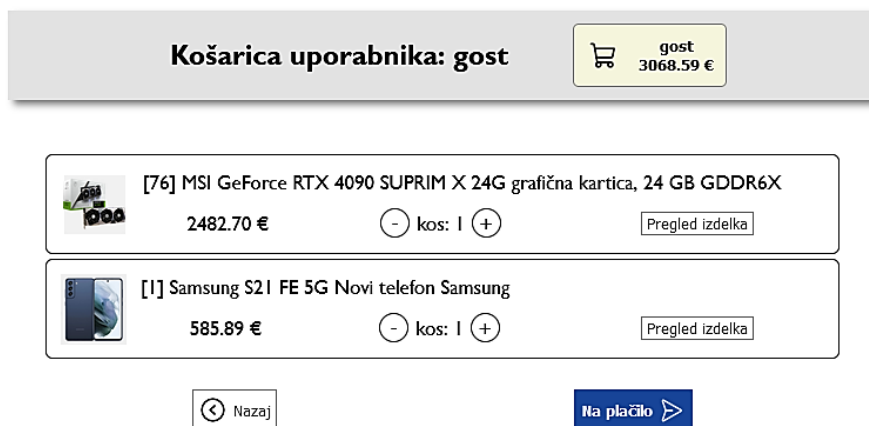
Najprej se lotimo testiranja spletne trgovine. Spletna trgovina nam ponuja precej možnih dejanj. Lahko kliknemo na izdelek in si ogledamo njegove podrobnosti, ali pa ga dodamo v košarico. Sprva je na strani prikazanih 6 izdelkov, lahko pa jih prikažemo več z gumbom *Prikaži več* na dnu strani. Mogoča je tudi uporaba filtrov na levi strani, in sicer filtriranja po kategoriji izdelka, ceni, ali popustu. Izdelke lahko tudi iščemo po imenu z iskalno vrstico na vrhu. Navigacijska vrstica trgovine nam omogoča odpiranje košarice, in sicer iz dveh možnih položajev, ko smo na vrhu strani, ali pa nismo na vrhu in se pomikamo navzdol. V slednjem primeru imamo še možnost klika na gumb, ki nas iz spodnjega dela strani prestavi na vrh. (Priloga 3)

Preizkusimo prikaz produkta ob kliku nanj. Odpre se stran s podrobnostmi produkta, kot lahko vidimo na Slika 17.



Slika 17: Posnetek zaslona prikaza podrobnosti izdelka

Ob kliku na dodajanje v košarico se izdelek uspešno doda in cena košarice se spremeni. Z gumbom nazaj se lahko premaknemo nazaj na spletno trgovino. Sprva pa je med preizkušanjem prišlo do neskončnega nalaganja izdelkov, kar sem popravil z izbrisom programske kode v *KosaricaC.jsx*, ki nastavi *niProduktov* na *true*. Dodajmo še en izdelek z gumbom prek trgovine in pogledjmo v košarico.



Slika 18: Pogled nakupovalne košarice

V košarico sta se uspešno dodala oba izdelka in cena se posodobi. Preizkusimo še dodajanje in odstranjevanje kosov in ugotovimo, da deluje pravilno, saj se dodajanje zaustavi ob največji količini, ki je na voljo, in se samodejno odstrani iz košarice, ko je količina 0. Iz košarice je mogoč ponoven pregled podrobnosti izdelka, zato preverimo tudi to možnost. Ugotovimo, da smo prišli na enako stran kot prej iz trgovine, le da je gumb za dodajanje onemogočen in se ob kliku nazaj vrnemo v košarico. Na plačilo se bomo vrnili kasneje, sedaj pa pojdimo nazaj na trgovino. Preizkusimo gumb *Prikaži več*, ki pravilno doda 6 izdelkov že prikazanim, pri čemer pravilno izpisuje število prikazanih izdelkov in izgine ob prikazu vseh. Gumb za pomik na vrh nas prav tako pravilno pomakne na vrh strani.

Poglejmo sedaj navigacijsko vrstico trgovine. Ko smo na vrhu strani, se prikaže statično, pod glavno navigacijsko vrstico aplikacije. Ob premiku navzdol za določen odmik pa se pojavi navigacijska vrstica, ki na vrhu strani ostane z nami, čeprav se pomikamo navzdol. Ko se prikaže druga navigacijska vrstica, se prikaže tudi gumb za skok na vrh. Ko pa se premaknemo navzgor, pa se navigacijska vrstica skrije. Prikaz deluje pravilno, preverimo še odpiranje košarice prek navigacijske vrstice, ko se ne nahajamo na vrhu strani. Prav tako pridemo v košarico, ki deluje pravilno.

Osredotočimo se še na filtriranje. Denimo, da izberemo le filtre nekaterih kategorij in filtriramo. Prikažejo se pravilni izdelki, če označimo kategorije in vsi izdelki, če ne označimo nobene.



Poizkusimo še filtriranje po popustu, kjer se pojavi enaka težava kot prej: neskončno nalaganje izdelkov. Rešimo jo prav tako kot prvič, z odstranjevanjem metode v *NakupovanjeC.jsx*, ki nastavi *niProduktov* na *true*. Potem ko smo preizkusili še kombinacijo obeh filtrov, preizkusimo še filtriranje po ceni. Filtriranje po ceni sem preizkusil z vnosom običajnih cen, negativnih števil in črk. V vseh primerih sem dobil pravilno množico izdelkov. Pri običajnih cenah se filter upošteval cene izdelkov s popustom, pri ostalih dveh vnosih pa je izbrisal nepravilne znake in filtriral brez cene. Preizkusil sem še nekaj kombinacij vseh filtrov, ki so delovali po pričakovanju. V trgovini imamo še dodatno funkcijo, in sicer iskanje izdelkov po imenu. Po vnosu znakov v iskalno vrstico, se izpiše 6 predlogov izdelkov, ki so na voljo iz podatkovne baze in se ujemajo z iskalnim nizom. Ob kliku na predlog, se v iskalno poje vnese ime zelenega izdelka in ob kliku na povečevalno steklo se izvede poizvedba, ki nam vrne izdelek s tem imenom. Zaradi delovanja stavkov SQL se sicer pri predlogih vrnejo tudi izdelki, ki namesto črke *s* vsebujejo *š*, kar lahko opazimo na Slika 19.

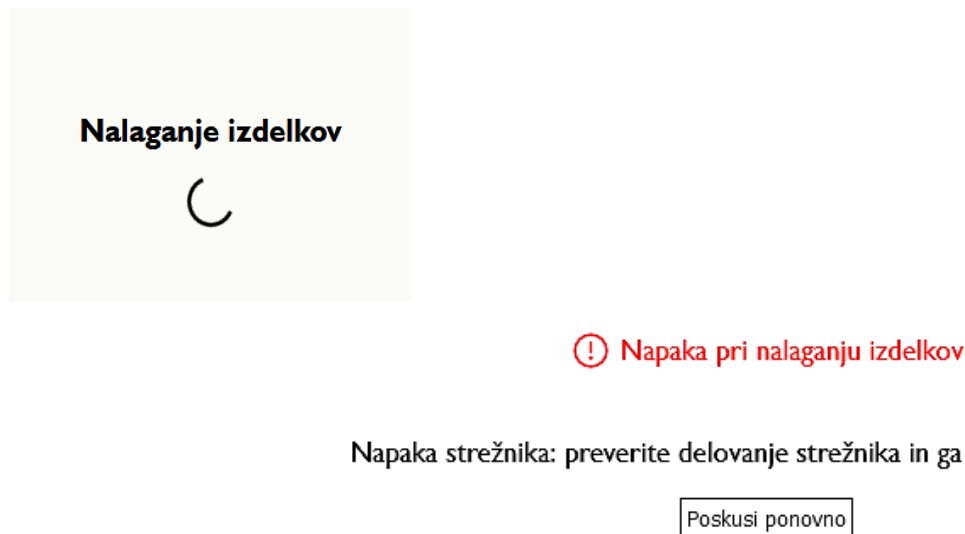


Slika 19: Iskanje izdelkov po imenu

Če izberemo na primer prvi predlog in kliknemo na gumb išči, se nam v predelu za prikaz izdelkov prikaže le ta element. Preizkusil sem nekaj iskanj in delovanje iskanja se je odrezalo po pričakovanjih. Poleg tega imamo ob praznem polju za iskanje izdelkov možnost osveževanja izdelkov z gumbom ob iskalnem polju. Če kliknemo nanj, se izvede ponovno pridobivanje izdelkov iz podatkovne baze. Preizkusimo še osveževanje v primeru, da uporabimo določene filtre. Osveževanje povzroči ponastavitev vseh filtrov in vrne nove izdelke v naključnem vrstnem redu.

S tem smo preizkusili vse funkcionalnosti spletne trgovine z izjemo naročanja. Preizkusimo še delovanje trgovine v primeru, da strežnik za dostop do podatkovne baze ne deluje. V ukaznem pozivu smo ustavili proces strežnika Express.js in ponovno naložili spletno aplikacijo v načinu

razvoja. Pri pogledu spletne trgovine nas najprej pričaka animacija za nalaganje, ki se po nekaj sekundah spremeni v obvestilo o napaki. Poglejmo ju na Slika 20.



Slika 20: Spletna trgovina ob napaki strežnika

Po določenem času se nalaganje preneha in če do takrat ne prejmemo informacij o izdelkih, se prikaže stran z napako. Ker bo aplikacija delovala lokalno, je na obvestilu napake navodilo, da naj ponovno zaženemo strežnik. Z gumbom *Poskusi ponovno* lahko spet poskusimo naložiti izdelke. Preizkusimo napako na strežniku takoj po zagonu aplikacije in med osveževanjem izdelkov. V prvem primeru se trgovina obnaša po pričakovanjih, saj se po ponovnem zagonu strežnika in pritisku na gumb *Poskusi ponovno* izdelki uspešno naložijo. Enako velja tudi za izpad strežnika in nato osveževanje izdelkov.

Poglejmo si še delovanje sistema naročanja po dodajanju izdelkov v košarico. Ko se iz košarice z gumbom premaknemo na plačilo, se nam prikaže stran s podatki o nakupu, ki jo lahko vidimo tudi v Priloga 4. V ta obrazec vnesemo podatke o kupcu in prejemniku, če to nista isti osebi, in naslov za dostavo. Izberemo lahko tudi način dostave, pri katerem se cena prišteje k računu. Na koncu je še pregled vsebine nakupovalne košarice in izbira načina plačila. V vnosna polja lahko vnašamo poljubne znake, ostala vnosa pa sta s klikom na možno izbiro, zato je tukaj testiranje nepotrebno. Kljub temu pa lahko pregledamo pravilnost podatkov v košarici in pošiljanje naročila po kliku na gumb *Oddaj naročilo*. Pri ustvarjanju naročila imamo dva možna načina. Naročamo lahko kot gost, ali pa kot uporabnik, ki ima narejen račun. Ustvarjanje računa bomo testirali v nadaljevanju, potrebujemo pa primerjavo pri naročanju. V primeru, da se prijavimo kot uporabnik, se naši podatki ob prihodu na stran *Podatki o nakupu* samodejno izpolnijo. Prav

tako pridobimo možnost načina plačila s kartico. Ključna prednost pa je ta, da si lahko kasneje v svojem profilu ogledamo vsa naročila in tudi račune za opravljene nakupe, kar bomo testirali v nadaljevanju. Zaenkrat bomo preizkusili le, ali se naročilo dejansko ustvari in zapiše v podatkovno bazo. Sprva sem preizkusil oddajo nakupa kot gost. Ob oddaji naročila se nam prikaže obvestilo, da je bilo naročilo oddano. Ker smo kupovali kot gost, ne moramo spletno pregledati naročila, račun naj bi v teoriji prišel skupaj z dostavljenim paketom. Preizkusimo še naročanje z uporabo uporabniškega profila. V tem primeru so se na strani za naročanje podatki o kupcu samodejno izpolnili na pravilne vrednosti profila uporabnika, prav tako je bila na voljo možnost nakupa s kartico in pregleda nakupa iz profila, kar bomo še testirali v nadaljevanju. V podatkovni bazi sem preveril, ali so bila naročila shranjena in ugotovil, da so bila pravilno vnesena in da se je temu primerno zmanjšala zaloga izdelkov iz tabele *Izdelki*.

Med preizkušanjem izpolnjevanja in oddaje naročanja sem v programu odkril napako, in sicer v komponenti *KosaricaC.jsx*. Iz prehoda iz košarice na podatke o naročilu, je bila cena košarice različna in mnogo manjša od pravilne ter negativna. Izkazalo se je, da se je v programski kodi pred znak za odštevanje prikradel tudi znak za množenje. Po odstranitvi je bila cena košarice pravilna.

Preizkusimo še obnašanje programa pri napaki streznika, če se ta zgodi medtem ko smo izdelke že dodali v košarico a še nismo oddali naročila. Najprej s prijavljenim uporabnikom. Ob prehodu iz košarice na plačilo, se nam podatki o kupcu ne izpolnijo samodejno. Če jih izpolnimo in skušamo oddati naročilo, nas po nekaj sekundah pričaka sporočilo, da je bila potrditev naročila neuspešna in lahko se vrnemo v košarico. Enako deluje tudi v primeru nakupa kot gost.



Vzemimo še primer, ko kupec doda izdelke v košarico, a je njihova zaloga majhna. Zgodi se lahko, da se pred oddajo naročila zaloga izprazni. Poustvarimo tako situacijo in pogledajmo, kaj se zgodi. V primeru, da gremo iz košarice v komponento *Podatki o nakupu*, se izdelek odstrani iz košarice in prikaže se obvestilo, da je bil izdelek odstranjen, ker ga ni več na zalogi. Enako se zgodi tudi, če je zaloga izdelkov pošla med našim izpolnjevanjem informacij o naročilu, zato ne pride do nedovoljenih stanj.








## 5.2 Sistem uporabnikov

Aplikacija ima implementiran sistem za registracijo in prijavo uporabnikov. Uporabnik se registrira na komponenti *Registracija*, ki jo lahko vidimo v Prilogi 5. Testirajmo vnos v polja

obrazca in podatkovno bazo tako, da dodamo nekaj uporabnikov in poskušamo v polja vpisati nedovoljene vrednosti. Pri vnašanju uporabnikov nisem naletel na nobeno težavo. Poskusimo se prijaviti v enega izmed registriranih profilov. Na komponenti *Prijava* vnesemo uporabniško ime in geslo. Preizkusimo napačen vnos, vnos, če strežnik ni dosegljiv in pravilen vnos. Ob napačnem vnosu dobimo obvestilo, da so podatki napačni, če pa se prijavimo, vendar strežnik ne deluje, nam program napiše, da gre za napako, in naj poskusimo kasneje. Če pa vnesemo pravilne podatke, se nam odpre stran z informacijami o uporabniku, ki si jo lahko ogledamo v Priloga 6. Uporabnik lahko uredi nekatere osebne podatke, si spremeni geslo, izbriše račun in se odjavi. Poleg tega ima na voljo določen nabor funkcionalnosti, ki mu lahko omogočajo pregled uporabnikov v sistemu, pregled oseb, dodajanje uporabnikov ali izdelkov, pregled vseh izdelkov, pregled naročil in računov ter upravljanje s podatkovno bazo. Te funkcije so na voljo administratorjem, medtem ko so funkcije pregleda naročil, računov, oseb ter pregled v domeni zaposlenih. Računovodje imajo enaka pooblastila kot zaposleni, le da lahko še dodajajo izdelke v bazo. Stranka ima možnost le pregleda naročil in računov, in sicer le svojih. Preizkusil sem vse funkcionalnosti profila stranke, pri čemer nisem odkril napak. Preglejmo še vse funkcionalnosti administratorja. Omogočen mu je pregled uporabnikov, pri čemer lahko spreminja njihovo vlogo in onemogoči profil. Vlogo spremeni z vpisom številke, profil pa omogoči s klikom na povezavo, kot lahko vidimo iz Slika 21.

### Pregled uporabnikov

 Nazaj
Vsi 

Uporabniško ime	Geslo	Vloga	Omogočen	Spremeni
admin	8451029959740982	0 (admin)	 Omogočen	/
ales123	3090675575880903	1 (zaposleni)	 Omogočen	 Onemogoči
andrej123	8365844368405956	3 (racunovodja)	 Omogočen	 Onemogoči
boris123	7953066060766799	2 (stranka)	 Omogočen	 Onemogoči

Slika 21: Pregled uporabnikov

Uporabnike lahko tudi filtrira glede na vlogo. Pri vsaki tabeli si lahko ogledamo podrobnosti zapisa (vrstice) s klikom nanj. Tako se nam v zbrani obliki prikažejo podrobnosti. Preglejmo še *Pregled oseb*, kjer so zapisane podrobnejše informacije o osebah. Zapisne lahko iščemo in s klikom na vrstico odpremo podroben pogled. Osebi lahko spremenimo oddelek in plačo, kar je v preizkušenih primerih vselej delovalo po pričakovanjih. Dodajanje uporabnikov je komponenta, ki je precej podobna obrazcu za registracijo, le da tu lahko administrator dodeli

vlogo uporabniku, ki ni nujno le stranka, prav tako mu lahko dodeli plačo in oddelek. Pri vnosu nekaj uporabnikov, se je izkazalo, da sistem za vnos deluje in opozori ob napaki na strežniku. Podoben obrazec imamo tudi za vnašanje izdelkov, lahko ga vidimo na Slika 22.

### Dodajanje izdelkov

Naložite sliko:  No file selected.

Ime izdelka

Kategorija izdelka

Cena za kos  decimalna pika

Kosov na voljo

Kratek opis

Informacije  Podrobnosti

Popust

Slika 22: Dodajanje izdelkov

Pri izdelku lahko vnašamo poleg osnovnih podatkov tudi sliko. Naložimo jo prek nalagalnika slik, ki odpre *raziskovalca* in prikaže le slike formata, ki je dovoljen. Ob uspešnem shranjevanju izdelka v podatkovno bazo se izpiše sporočilo o uspešnem zapisu, v nasprotnem primeru pa se izpiše sporočilo o napaki na strežniku. Vnesene izdelke lahko pogledamo v pregledu izdelkov. Ob kliku na izdelek, se nam prikažejo podrobnosti, ki jih lahko spreminjamo. V Prilogi 7 si lahko ogledamo pogled podrobnosti izdelka. Testiral sem spremembo vseh lastnosti izdelkov in dodajanje slik.

Prav tako kot izdelke, lahko pregledujemo tudi naročila in račune. Za pregled teh je mogoče uporabiti filtre. V podrobnostih pa lahko zaposleni naročila označijo kot opravljena in s tem generirajo račun, ki se shrani v tabelo *Racuni*. Naročilo sicer vsebuje tudi pregled vseh izdelkov, ki so bili naročeni. Ob testiranju teh prav tako nisem zaznal težav.

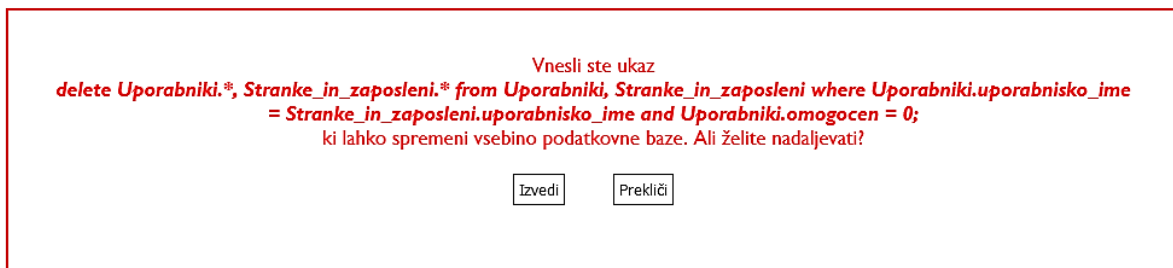
Za konec nam ostane le še upravljanje s podatkovno bazo, ki je dostopno s strani administratorja. Predstavlja le polje za vnos SQL stavkov in prikaže vrnjene vrednosti v tabeli. Najprej preverimo delovanje ukazov DQL, t. j. nekaj stavkov *select*. Preverimo še stavke DML,

to so *insert*, *update* in *delete*. Nato preverimo še delovanje ukazov DDL, ki spremenijo stanje ali metapodatke baze. Uporabimo ukaz *delete* iz Izsek 47, ki bo izbrisal testne podatke iz baz *Uporabniki* ter *Stranke\_in\_zaposleni*, če smo prej vse profile za izbris onemogočili. Vnesemo ukaz v vnosno polje in kliknemo *Izvedi*.

*Izsek 47: SQL stavek za izbris testnih podatkov*

```
DELETE Uporabniki.*, Stranke_in_zaposleni.*  
FROM Uporabniki, Stranke_in_zaposleni  
WHERE  
Uporabniki.uporabnisko_ime = Stranke_in_zaposleni.uporabnisko_ime  
AND Uporabniki.omogocen = 0;
```

Tedaj se nam prikaže varnostno opozorilo pred izvedbo nepopravljivih sprememb v bazi podatkov, ki ga lahko vidimo na Slika 23. Za izvajanje kliknemo *Izvedi* in ukaz se izvede.



*Slika 23: Varnostno opozorilo pred izvedbo SQL ukaza*

S preizkušanjem smo testirali glavne funkcionalnosti programa in zagotovili, da ta deluje tako, kot mora.

## 6 Zaključek

Z izdelavo te seminarske naloge sem zajel velik delež obravnavane snovi pri predmetu računalništvo na tehniški gimnaziji in tudi precej stvari, ki sem se jih na novo naučil. Dosegli smo cilj, ki je izdelana delujoča aplikacija, ki ponuja precej stvari za preizkušanje. S seminarsko nalogo smo dokumentirali program in postopke izdelave, kar je predvideno. Seveda je možnosti za izboljšavo še veliko, denimo večja podatkovna baza, ki bi ponujala več možnosti shranjevanja podatkov ali pa bolj oblikovan uporabniški vmesnik. Kljub temu mislim, da je seminarska naloga dosegla cilj spoznavanja novih tehnologij in uporabo že naučenih. Nedvomno sem spoznal, kakšna je struktura aplikacij, kar mi bo najverjetneje v nadaljevanju študija zelo pomagalo.

## **7 Zahvala**

Ob napisanem končnem izdelku za splošno maturo se zahvaljujem vsem profesorjem računalništva, ki so mi v vseh štirih letih izobraževanja podajali zanje, ki ga bom nedvomno v svoji karieri lahko unovčil in na teh temeljih gradil znanje. Posebna zahvala gre profesorju dr. Albertu Zorku, ki nas je na zaključni izpit splošne mature iz računalništva pripravljajl vsa štiri leta. Prav tako gre zahvala tudi profesorjem Gregorju Medetu, Simonu Vovku, Miletu Božiću in Tomažu Ferbežarju ter Srednji elektro šoli in tehniški gimnaziji Šolskega centra Novo mesto. Zahvaljujem se tudi svojemu očetu in mami za pomoč pri oblikovanju koncepta spletne trgovine ter za tisk in vezavo naloge.



## 8 Viri in literatura

1. Web Programming. [Elektronski] Techopedia, 13. maj 2020. [Navedeno: 24. januar 2023.] <https://www.techopedia.com/definition/23898/web-programming>.
2. Računalništvo - podatkovne baze. Zorko, Albert. Novo mesto : s.n., 2023.
3. Haverbeke, Marijn. *Eloquent JavaScript 3rd edition*. San Francisco : No Starch Press, 2019. 978-1-59327-950-9.
4. B., Artūras. What Is CSS and How Does It Work? *Hostinger tutorials*. [Elektronski] Hostinger, 4. januar 2023. [Navedeno: 30. januar 2023.] <https://www.hostinger.com/tutorials/what-is-css>.
5. McKenzie, Cameron. CSS (cascading style sheets). *UI patterns*. [Elektronski] The Server Side, oktober 2021. [Navedeno: 30. januar 2023.] <https://www.theserverside.com/definition/cascading-style-sheet-CSS>.
6. Netscape company press relations. 28 industry-leading companies to endorse JavaScript as a complement to Java for easy online application development. *Netscape and Sun announce JavaScript, the open, cross-platform object scripting language for enterprise networks and the internet*. [Elektronski] Netscape, 16. september 2007. [Navedeno: 24. januar 2023.] <https://web.archive.org/web/20070916144913/https://wp.netscape.com/newsref/pr/newsrelease67.html>.
7. MDN contributors. What is JavaScript? *MDN Web Docs*. [Elektronski] Mozilla, 14. september 2022. [Navedeno: 24. januar 2023.] [https://developer.mozilla.org/en-US/docs/Learn/JavaScript/First\\_steps/What\\_is\\_JavaScript](https://developer.mozilla.org/en-US/docs/Learn/JavaScript/First_steps/What_is_JavaScript).
8. Meta Platforms, Inc. Docs. *React*. [Elektronski] Facebook Open Source, 2023. [Navedeno: 29. januar 2023.] <https://reactjs.org/>.
9. —. Learn React. *React Docs Beta*. [Elektronski] Facebook Open Source, 2023. [Navedeno: 29. januar 2023.] <https://beta.reactjs.org/learn>.
10. —. Api reference. *React Docs Beta*. [Elektronski] Facebook Open Source, 2023. [Navedeno: 29. januar 2023.] <https://beta.reactjs.org/reference/react>.
11. Herbert, David. What is React.js? (Uses, Examples, & More). [Elektronski] HubSpot, 27. junij 2022. [Navedeno: 29. januar 2023.] <https://blog.hubspot.com/website/react-js>.
12. MDN Web Docs. Introduction to the DOM. *References*. [Elektronski] Mozilla Corporation. [Navedeno: 9. april 2023.] [https://developer.mozilla.org/en-US/docs/Web/API/Document\\_Object\\_Model/Introduction](https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model/Introduction).
13. Facebook Open Source. Virtual DOM and Internals. *React docs*. [Elektronski] Facebook Open Source. [Navedeno: 17. april 2023.] <https://legacy.reactjs.org/docs/faq-internals.html#what-is-the-virtual-dom>.
14. Sheldon, Robert in Denman, James. Node.js (Node). *WhatIs.com*. [Elektronski] TechTarget, november 2022. [Navedeno: 29. januar 2023.]

[https://www.techtarget.com/whatis/definition/Nodejs#:~:text=js%20\(Node\)%3F-,Node.,to%20learn%20an%20additional%20language..](https://www.techtarget.com/whatis/definition/Nodejs#:~:text=js%20(Node)%3F-,Node.,to%20learn%20an%20additional%20language..)

15. Sufiyan, Taha. What is Node.js: A Comprehensive Guide. *SimpliLearn*. [Elektronski] 24. november 2022. [Navedeno: 29. januar 2023.]

<https://www.simplilearn.com/tutorials/nodejs-tutorial/what-is-nodejs>.

16. Krstic, Branko. 64 Node JS Stats that Prove Its Awesomeness in 2022. *Web tribunal*. [Elektronski] 6. april 2022. [Navedeno: 29. januar 2023.]

<https://webtribunal.net/blog/node-js-stats/#:~:text=1.,at%20least%2030%20million%20websites..>

17. Mardan, Azat. *Express.js Guide: The Comprehensive Book on Express.js*. s.l. : Leanpub, 2014. 9781494269272.

18. Kinsta. What Is Express.js? Everything You Should Know. [Elektronski] Kinsta, 26. oktober 2022. [Navedeno: 17. april 2023.] <https://kinsta.com/knowledgebase/what-is-express-js/>.

19. MIT. *Axios*. [Elektronski] [Navedeno: 29. januar 2023.] <https://axios-http.com/docs/intro>.

20. Das, Pratik. Complete Guide to Axios HTTP Client. [Elektronski] Reflectoring.io, 20. maj 2022. [Navedeno: 17. april 2023.] <https://reflectoring.io/tutorial-guide-axios/>.

21. B., Richard. What is MySQL: MySQL Explained For Beginners. *Hostinger tutorials*. [Elektronski] Hostinger, 14. december 2022. [Navedeno: 30. januar 2023.] <https://www.hostinger.com/tutorials/what-is-mysql>.

22. Johnson, Jonathan. Asynchronous Programming: A Beginner's Guide. [Elektronski] BMC blogs, 9. oktober 2020. [Navedeno: 14. april 2023.] <https://www.bmc.com/blogs/asynchronous-programming/>.

23. Kantor, Ilya. Promises, async/await. [Elektronski] JavaScript.info. [Navedeno: 14. april 2023.] <https://javascript.info/async>.

24. Kumar Panigrahi, Kiran. Difference between Bottom-Up Model and Top-Down Model. [Elektronski] Tutorialspoint, 20. februar 2023. [Navedeno: 17. april 2023.] <https://www.tutorialspoint.com/difference-between-bottom-up-model-and-top-down-model>.

25. Biscobing, Jacqueline. Entity Relationship Diagram (ERD) . [Elektronski] Tech Target. [Navedeno: 10. april 2023.] <https://www.techtarget.com/searchdatamanagement/definition/entity-relationship-diagram-ERD>.

26. Visual Paradigm. What is Data Flow Diagram? [Elektronski] Visual Paradigm. [Navedeno: 10. april 2023.] <https://www.visual-paradigm.com/guide/data-flow-diagram/what-is-data-flow-diagram/>.

27. Shamim, Uzair. An Introduction To Rendering In React. [Elektronski] Medium, 30. december 2017. [Navedeno: 11. april 2023.] <https://medium.com/information-and-technology/an-introduction-to-react-rendering-9c24a96b838b>.

28. Imoh, Ifeoma. What Is React Router & What Is React Location. [Elektronski] Progress Telerik, 9. februar 2022. [Navedeno: 11. april 2023.] <https://www.telerik.com/blogs/what-is-react-router-what-is-react-location>.
29. Ferguson, Nicole. What's the Difference Between Frontend and Backend Web Development? [Elektronski] CF Blog, 13. marec 2023. [Navedeno: 12. april 2023.] <https://careerfoundry.com/en/blog/web-development/whats-the-difference-between-frontend-and-backend/>.
30. Sharma, Anubhav. What Is Express JS In Node JS? [Elektronski] Simplilearn, 10. april 2023. [Navedeno: 16. april 2023.] <https://www.simplilearn.com/tutorials/nodejs-tutorial/what-is-express-js>.
31. OpenJS Foundation. Installing. [Elektronski] Express.js. [Navedeno: 16. april 2023.] <https://expressjs.com/en/starter/installing.html>.
32. MDN contributors. [Elektronski] Mdn web docs, 24. februar 2023. [Navedeno: 16. april 2023.] [https://developer.mozilla.org/en-US/docs/Learn/Server-side/Express\\_Nodejs/Introduction](https://developer.mozilla.org/en-US/docs/Learn/Server-side/Express_Nodejs/Introduction).
33. OpenJS Foundation. express(). 4.x API. [Elektronski] Express.js. [Navedeno: 16. april 2023.] <https://expressjs.com/en/api.html>.
34. Calaça, Luiz. You need to know about req and res objects to build an API in Node.js using Express.js. [Elektronski] Dev.to, 12. junij 2022. [Navedeno: 16. april 2023.] <https://dev.to/luizcalaca/you-need-to-know-about-req-and-res-objects-to-build-an-api-in-nodejs-using-expressjs-4f3i>.
35. StackHawk. Node.js SQL Injection Guide: Examples and Prevention. [Elektronski] StackHawk, 6. september 2021. [Navedeno: 11. april 2023.] <https://www.stackhawk.com/blog/node-js-sql-injection-guide-examples-and-prevention/>.
36. Mdn contributors. HTTP response status codes. *Mdn web docs*. [Elektronski] Mozilla, 10. april 2023. [Navedeno: 12. april 2023.] [https://developer.mozilla.org/en-US/docs/Web/HTTP/Status#client\\_error\\_responses](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status#client_error_responses).
37. Shahid. Handle GET and POST Request in Express. [Elektronski] CodeforGeek, 21. april 2021. [Navedeno: 11. april 2023.] <https://codeforgeek.com/handle-get-post-request-express-4/>.
38. Sebastian, Nathan. ExpressJS - How to use express-fileupload to handle uploading files. [Elektronski] 2. julij 2021. [Navedeno: 16. april 2023.] [https://sebastian.com/express-fileupload/?utm\\_content=cmp-true](https://sebastian.com/express-fileupload/?utm_content=cmp-true).
39. OpenJS Foundation. Routing. [Elektronski] Express.js. [Navedeno: 16. april 2023.] <https://expressjs.com/en/guide/routing.html>.
40. ZetCode. Axios tutorial. [Elektronski] ZetCode, 10. januar 2023. [Navedeno: 11. april 2023.] <https://zetcode.com/javascript/axios/>.
41. Asaolu, Elijah. Axios Multipart Form Data - Sending File Through a Form with JavaScript. [Elektronski] StackAbuse, 1. februar 2022. [Navedeno: 11. april 2023.]

<https://stackabuse.com/axios-multipart-form-data-sending-file-through-a-form-with-javascript/>.

42. Axios contributors. Multipart Bodies. [Elektronski] Axios. [Navedeno: 16. april 2023.] <https://axios-http.com/docs/multipart>.

43. MDN contributors. URL: createObjectURL() static method. [Elektronski] Mdn web docs, 12. april 2023. [Navedeno: 17. april 2023.] <https://developer.mozilla.org/en-US/docs/Web/API/URL/createObjectURL>.

44. Buna, Samer. Why React Developers LOVE Node. [Elektronski] jsComplete, september 2020. [Navedeno: 13. april 2023.] <https://jscomplete.com/learn/why-node-for-react>.

45. W3 Schools. React JSX. [Elektronski] W3 Schools. [Navedeno: 13. april 2023.] [https://www.w3schools.com/REACT/react\\_jsx.asp](https://www.w3schools.com/REACT/react_jsx.asp).

46. Writing Markup with JSX. *React Dev*. [Elektronski] Meta Open Source. [Navedeno: 13. april 2023.] <https://react.dev/learn/writing-markup-with-jsx>.

47. Imran, Danyal. Everything React — All about JSX. [Elektronski] Medium, 24. februar 2019. [Navedeno: 11. april 2023.] [https://medium.com/@danyal\\_imran/everything-react-all-about-jsx-4a5123ac8606](https://medium.com/@danyal_imran/everything-react-all-about-jsx-4a5123ac8606).

48. Meta Open Source. JavaScript in JSX with Curly Braces. [Elektronski] Meta Open Source. [Navedeno: 13. april 2023.] <https://react.dev/learn/javascript-in-jsx-with-curly-braces>.

49. PedroTech. *React Hooks Course - All React Hooks Explained*. [videoposnetek] s.l. : YouTube, 2021.

50. Fireship. *SI*. [videoposnetek] s.l. : YouTube, 2021.

51. Adhikary, Tapas. React Hooks Fundamentals for Beginners. [Elektronski] Free Code Camp, 15. marec 2022. [Navedeno: 13. april 2023.] <https://www.freecodecamp.org/news/react-hooks-fundamentals/>.

52. Meta Open Source. useState. [Elektronski] Meta Open Source. [Navedeno: 13. april 2023.] <https://react.dev/reference/react/useState>.

53. Web Dev Cody. *Why You Need to Understand Re-rendering in React and useState Hook*. [videoposnetek] s.l. : YouTube, 2021.

54. Meta Open Source. Updating Objects in State. *Recat dev*. [Elektronski] Meta Open Source. [Navedeno: 15. april 2023.] <https://react.dev/learn/updating-objects-in-state>.

55. MDN contributors. Event: target property. [Elektronski] Mdn web docs, 7. april 2023. [Navedeno: 15. april 2023.] <https://developer.mozilla.org/en-US/docs/Web/API/Event/target>.

56. Devtrium. How to use React Context like a pro. [Elektronski] Devtrium, 13. september 2021. [Navedeno: 11. april 2023.] <https://devtrium.com/posts/how-use-react-context-pro>.

57. Meta Open Source. useContext. [Elektronski] Meta Open Source. [Navedeno: 13. april 2023.] <https://react.dev/reference/react/useContext>.
58. Pavlutin, Dmitri. A Guide to React Context and useContext() Hook. [Elektronski] 2. februar 2023. [Navedeno: 11. april 2023.] <https://dmitripavlutin.com/react-context-and-usecontext/>.
59. Hadzhiev, Borislav. How to get the Value of an Input field in React. [Elektronski] [Navedeno: 11. april 2023.] <https://bobbyhadz.com/blog/react-get-input-value>.
60. Meta Open Source. useRef. *React dev*. [Elektronski] Meta Open Source. [Navedeno: 14. april 2023.] <https://react.dev/reference/react/useRef>.
61. W3 Schools. Window addEventListener(). *JS Reference*. [Elektronski] W3 Schools. [Navedeno: 14. april 2023.] [https://www.w3schools.com/jsref/met\\_win\\_addeventlistener.asp](https://www.w3schools.com/jsref/met_win_addeventlistener.asp).
62. MDN contributors. Element: scrollIntoView() method. [Elektronski] Mdn web docs, 7. april 2023. [Navedeno: 15. april 2023.] <https://developer.mozilla.org/en-US/docs/Web/API/Element/scrollIntoView>.
63. Stonys, Vincas. How to Scroll to Element in React. [Elektronski] Code Frontend, 10. september 2022. [Navedeno: 15. april 2023.] <https://codefrontend.com/scroll-to-element-in-react/>.
64. Wieruch, Robin. What is preventDefault() in React. [Elektronski] 28. maj 2019. [Navedeno: 11. april 2023.] <https://www.robinwieruch.de/react-preventdefault/>.
65. MDN contributors. Event: stopPropagation() method. [Elektronski] Mozilla MDN web docs, 7. april 2023. [Navedeno: 14. april 2023.] <https://developer.mozilla.org/en-US/docs/Web/API/Event/stopPropagation>.
66. Meta Open Source. useEffect. *React dev*. [Elektronski] Meta Open Source. [Navedeno: 14. april 2023.] <https://react.dev/reference/react/useEffect>.
67. Pavlutin, Dmitri. Your Guide to React.useCallback(). [Elektronski] 28. januar 2023. [Navedeno: 15. april 2023.] <https://dmitripavlutin.com/react-usecallback/>.
68. Simplilearn. Why Should You Use a Router in React.js? *Software development*. [Elektronski] Simplilearn, 20. februar 2023. [Navedeno: 15. april 2023.] <https://www.simplilearn.com/tutorials/reactjs-tutorial/routing-in-reactjs>.
69. Remix Software, Inc. <BrowserRouter>. *ReactRouter*. [Elektronski] Remix Software, Inc. [Navedeno: 15. april 2023.] <https://reactrouter.com/en/main/router-components/browser-router>.
70. —. <Routes>. *React Router*. [Elektronski] Remix Software, Inc. [Navedeno: 15. april 2023.] <https://reactrouter.com/en/main/components/routes>.
71. Mittal, Aman. A guide to using React Router v6 in React apps. [Elektronski] LogRocket, 7. avgust 2020. [Navedeno: 11. april 2023.] <https://blog.logrocket.com/react-router-v6/>.

72. Remix Software, Inc. Route. *React Router*. [Elektronski] Remix Software, Inc. [Navedeno: 15. april 2023.] <https://reactrouter.com/en/main/route/route>.
73. Macoveiciuc, Andreea. How to handle navigation in your app with React Router Link. [Elektronski] New line, 29. maj 2020. [Navedeno: 15. april 2023.] <https://www.newline.co/@andreeamaco/how-to-handle-navigation-in-your-app-with-react-router-link--088f82d3>.
74. Napoleon. React Router useLocation hook – Tutorial and Examples. [Elektronski] Kinda Code, 3. marec 2023. [Navedeno: 15. april 2023.] [https://www.kindacode.com/article/react-router-uselocation-hook-tutorial-and-examples/?utm\\_content=cmp-true](https://www.kindacode.com/article/react-router-uselocation-hook-tutorial-and-examples/?utm_content=cmp-true).
75. Nasreen. How to use the useLocation hook in React. [Elektronski] Educative. [Navedeno: 11. april 2023.] <https://www.educative.io/answers/how-to-use-the-uselocation-hook-in-react>.
76. Mubashar, Saleh. useNavigate tutorial React JS. [Elektronski] Dev, 28. oktober 2021. [Navedeno: 15. april 2023.] <https://dev.to/salehmubashar/usenavigate-tutorial-react-js-aop>.
77. Sebastian, Nathan. How passing props to component works in React. [Elektronski] LogRocket, 17. december 2020. [Navedeno: 15. april 2023.] <https://blog.logrocket.com/the-beginners-guide-to-mastering-react-props-3f6f01fd7099/>.
78. MDN contributors. Destructuring assignment. [Elektronski] Mdn web docs, 5. april 2023. [Navedeno: 15. april 2023.] [https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Destructuring\\_assignment](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Destructuring_assignment).
79. Bado, Bernard. How to Run React JS Build Locally. [Elektronski] Upbeat Code, 30. november 2021. [Navedeno: 13. april 2023.] <https://www.upbeatcode.com/react/how-to-run-react-js-build-locally/>.
80. Deployment. *Create React App*. [Elektronski] Facebook, 1. december 2022. [Navedeno: 13. april 2023.] <https://create-react-app.dev/docs/deployment/>.
81. Buckley, Ian. How to Create a Batch (BAT) File in Windows: 5 Simple Steps. *Windows*. [Elektronski] Make use of, 19. julij 2022. [Navedeno: 17. april 2023.] <https://www.makeuseof.com/tag/write-simple-batch-bat-file/>.
82. Sieber, Tina. How to Launch Multiple Programs With One Shortcut in Windows 10. *Windows*. [Elektronski] Make use of, 27. julij 2021. [Navedeno: 17. april 2023.] <https://www.makeuseof.com/tag/launch-multiple-programs-single-shortcut-using-batch-file/>.
83. Peterson, Richard. MySQL Workbench Tutorial: What is, How to Install & Use. [Elektronski] Guru 99, 8. april 2023. [Navedeno: 15. april 2023.] <https://www.guru99.com/introduction-to-mysql-workbench.html>.
84. bryc. code. *GitHub repository*. [Elektronski] GitHub. [Navedeno: 16. april 2023.] <https://github.com/bryc/code/blob/master/jshash/experimental/cyrb53.js>.

85. —. **Generate a Hash from string in Javascript.** *Questions.* [Elektronski] StackOverflow, 4. september 2018. [Navedeno: 16. april 2023.] <https://stackoverflow.com/questions/7616461/generate-a-hash-from-string-in-javascript>.
86. Kinsta. **Environment Variables: What They Are and How To Use Them.** [Elektronski] Kinsta, 28. marec 2023. [Navedeno: 16. april 2023.] <https://kinsta.com/knowledgebase/what-is-an-environment-variable/>.
87. MySQL examples in Node.js. [Elektronski] Evertpot, 9. januar 2019. [Navedeno: 12. april 2023.] <https://evertpot.com/executing-a-mysql-query-in-nodejs/>.
88. Kyrnin, Jennifer. **Understanding the Index.html Page on a Website.** [Elektronski] ThoughtCo., 20. november 2020. [Navedeno: 25. januar 2023.] <https://www.thoughtco.com/index-html-page-3466505>.
89. StrongLoop, IBM, expressjs.com contributors. **5.x API.** *Express.* [Elektronski] Express.js, 2017. [Navedeno: 29. januar 2023.] <https://expressjs.com/en/5x/api.html>.
90. Dietrich, James. **setTimeout in React Components Using Hooks.** [Elektronski] Upmostly. [Navedeno: 11. april 2023.] <https://upmostly.com/tutorials/settimeout-in-react-components-using-hooks>.
91. Coding Tech. **Common React Mistakes: useEffect, useCallback and useMemo Hooks.** [videoposnetek] s.l. : YouTube, 2021.
92. Code Step By Step. **React js tutorial for beginners - useMemo vs useEffect / difference in useEffect and useMemo.** [videoposnetek] s.l. : YouTube, 2021.
93. PedroTech. **React Shopping Cart Ecommerce Beginner Website - Build & Deploy A React Beginner Project.** [videoposnetek] s.l. : YouTube, 2022.
94. Singh, Ashutosh. **Understanding Axios GET requests.** [Elektronski] LogRocket, 3. marec 2022. [Navedeno: 11. april 2023.] <https://blog.logrocket.com/understanding-axios-get-requests/>.
95. Parthibakumar, Murugesan. **What is .env ? How to Set up and run a .env file in Node?** [Elektronski] Codementor, 20. januar 2022. [Navedeno: 11. april 2023.] <https://www.codementor.io/@parthibakumarmurugesan/what-is-env-how-to-set-up-and-run-a-env-file-in-node-1pnyxw9yxj>.
96. Kundel, Dominik. **Working With Environment Variables in Node.js.** [Elektronski] Twilio, 10. februar 2022. [Navedeno: 11. april 2023.] <https://www.twilio.com/blog/working-with-environment-variables-in-node-js-html>.
97. Web Dev Simplified. **Learn Express Middleware In 14 Minutes.** [videoposnetek] s.l. : YouTube, 2020.
98. Anson the Developer. **ExpressJS Tutorial #8 - Connecting to MySQL Database.** [videoposnetek] s.l. : YouTube, 2020.
99. Simon, Kevin. **How to Upload Image in MySQL using Node.js and React.js.** [Elektronski] 2022. [Navedeno: 11. april 2023.] <https://morioh.com/p/d6bd1ff174c8>.

100. Orlov, Maxim. Send a File With Axios in Node.js. [Elektronski] [Navedeno: 11. april 2023.] <https://maximorlov.com/send-a-file-with-axios-in-nodejs/>.
101. Mdn web docs. Using FormData Objects. [Elektronski] Mozilla. [Navedeno: 11. april 2023.] [https://developer.mozilla.org/en-US/docs/Web/API/FormData/Using\\_FormData\\_Objects](https://developer.mozilla.org/en-US/docs/Web/API/FormData/Using_FormData_Objects).
102. Fakiolas, Marios. Handle Blobs requests with Axios the right way. [Elektronski] Medium, 12. maj 2018. [Navedeno: 11. april 2023.] <https://medium.com/@fakiolino/handle-blobs-requests-with-axios-the-right-way-bb905bdb1c04>.
103. Mulani, Safa. SQL SELECT statement with COUNT() function. [Elektronski] Digital Ocean, 3. avgust 2022. [Navedeno: 12. april 2023.] <https://www.digitalocean.com/community/tutorials/sql-select-statement-with-count>.
104. MySQL Tutorial. MySQL Tutorial. [Elektronski] MySQL Tutorial. [Navedeno: 12. april 2023.] <https://www.mysqltutorial.org/>.
105. Amit Thinks. *How to install MySQL 8.0.22 Server and Workbench latest version on Windows 10*. [videoposnetek] s.l. : YouTube, 2020.
106. Characters that are valid for user IDs and passwords. *IBM Business Automation Workflow*. [Elektronski] IBM, 7. maj 2021. [Navedeno: 12. april 2023.] <https://www.ibm.com/docs/en/baw/19.x?topic=security-characters-that-are-valid-user-ids-passwords>.
107. Dobra Somesan, Aleksandru. `iskalnoPolje.current.value` . [Elektronski] Codesandbox. [Navedeno: 15. april 2023.] <https://codesandbox.io/s/react-header-to-hide-on-scroll-ild1v?from-embed=&file=/src/index.js>.
108. PedroTech. *UseContext Hook Tutorial In ReactJS / Global States*. [videoposnetek] s.l. : YouTube, 2020.
109. MDN contributors. XMLHttpRequest. [Elektronski] Mdn web docs, 18. februar 2023. [Navedeno: 17. april 2023.] <https://developer.mozilla.org/en-US/docs/Web/API/XMLHttpRequest>.
110. IBM. What is a REST API? . [Elektronski] IBM. [Navedeno: 17. april 2023.] <https://www.ibm.com/topics/rest-apis>.



## STVARNO KAZALO

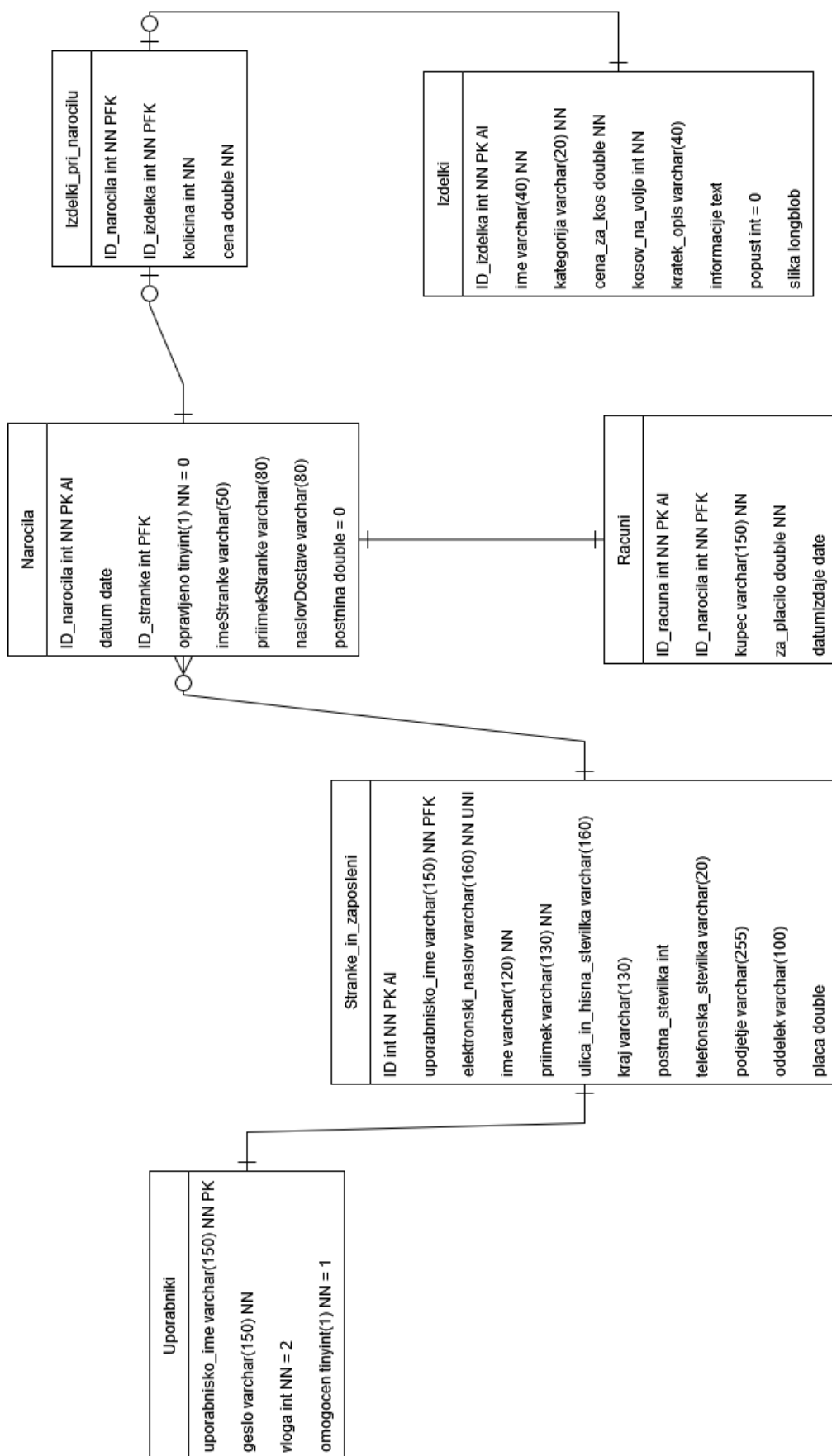
<i>.env</i> .....	7, 55, 56, 73
administratorja.....	7, 54, 62, 63
analizo .....	14
animacija .....	43
API8, 17, 20, 21, 23, 36, 37, 40, 42, 49, 67, 69, 70, 71, 73, 74	
aplikacij .....	2, 5, 6, 7, 8, 65
Aplikacija .....	5, 15, 17, 20, 61
aplikacije 3, 6, 7, 1, 2, 6, 7, 8, 9, 12, 18, 20, 21, 22, 27, 28, 31, 34, 44, 45, 46, 52, 54, 57, 58, 60	
argument.....	10, 22, 24, 30
asinhrono .....	7, 11, 42
<i>async</i> .....	6, 11, 68
<i>Async</i> .....	11
atribut ..4, 12, 25, 28, 36, 37, 45, 47, 48, 51	
avtentikacija .....	21
<i>await</i> .....	6, 11, 68
Axios 5, 6, 8, 20, 24, 25, 27, 42, 68, 69, 70, 73, 74	
<i>babel</i> .....	28
<i>back-end</i> .....	17, 20
<i>batch</i> .....	52, 72
brskalnik .....	8, 52
<i>callback</i> .....	10, 50
<i>Chrome</i> .....	1, 7
<i>client</i> .....	6, 20, 27, 52, 69
CSS.....	5, 6, 3, 4, 18, 19, 20, 27, 28, 51, 67
CSS selektor .....	51
<i>current</i> .....	33, 35, 36, 37, 74
datoteka .....	7, 17, 18, 24, 25
DDL.....	64
DELETE.....	6, 21, 22, 23, 24
destruktuiramo .....	51
destruktuiranje.....	29
diagram konteksta .....	15
diagram nivoja 0.....	15
diagram nivoja 1 .....	16
Diagram toka podatkov .....	5, 14
Diagram zgradbe programa .....	5, 17, 19
DML .....	63
dogodek .....	10, 31, 34, 39
DOM.....	6, 18, 27, 34, 40, 67
DQL.....	63
DTP .....	14
ECMAScript.....	5
elektronski naslov.....	13, 53
element.....	6, 18, 26, 28, 31, 35, 36, 37, 45, 51, 59, 71
entiteta.....	12, 13, 15
ER diagram .....	5, 12, 14
Express.js ....	5, 8, 17, 20, 27, 59, 68, 69, 73
Facebook.....	5, 67, 72
filtre.....	31, 58, 59, 63
Filtriranje .....	59
<i>front-end</i> .....	17, 20, 27
funkcija .....	10, 11, 18, 25, 29, 34, 40, 44
funkcionalno dekompozicijo .....	12
geslo .....	13, 53, 55, 56, 62
GET.....	6, 21, 23, 24, 25, 26, 69, 73
GitHub .....	27, 53, 72
glavi .....	3, 4, 24
gumb .....	7, 35, 40, 48, 57, 58, 59, 60
HTML ....	5, 6, 2, 3, 4, 6, 18, 19, 21, 27, 28, 34, 51
HTTP ...	5, 8, 20, 21, 22, 23, 24, 25, 26, 68, 69
ID .....	4, 12, 13, 53, 54
informacije .....	12, 13, 47, 48, 54, 62
informacijskem sistemu .....	12, 14
izvozov .....	27
JavaScript... 5, 2, 5, 6, 7, 10, 20, 27, 28, 33, 34, 67, 68, 69, 70, 72	
jezik.....	5, 7, 9
JSON.....	21
JSX.. 5, 7, 20, 27, 28, 34, 36, 37, 40, 51, 70	
kavlji .....	29
Kavlji .....	29
knjižnica.....	5, 7, 8, 20, 44
komponenta.... 6, 18, 29, 30, 34, 35, 40, 43, 45, 46, 62	
konceptov .....	27
kontekst.....	7, 19, 31, 32, 33
konteksta .....	6, 7, 19, 31, 32, 42
kopijo .....	30
košarico .....	40, 57, 58, 60, 61
lastnosti .... 5, 7, 4, 7, 10, 12, 18, 22, 30, 33, 35, 40, 48, 49, 51, 63	
Logika stanj .....	29
mapo .....	17, 27, 52
metoda.....	10
model .....	6, 12, 17, 68
Modeliranje.....	5, 12
modul .....	8, 20, 21, 23, 52, 55, 56
MySQL ....	3, 5, 1, 9, 20, 53, 68, 72, 73, 74

način .....	7, 1, 2, 4, 5, 6, 8, 9, 22, 27, 28, 33, 46, 57, 60
načrtovanje .....	12
nalaganja.....	6, 19, 33, 36, 39, 43, 58
nalaganje datotek.....	17, 21
napak .....	3, 2, 5, 12, 17, 22, 62
napake.....	11, 22, 57, 60
navigacija .....	5, 44
nespremenljiva .....	30
Node ...	3, 5, 7, 8, 17, 20, 27, 67, 68, 69, 70, 73, 74
Node.js	3, 5, 7, 8, 17, 20, 27, 67, 68, 69, 73, 74
notranje oblikovanje .....	4
npm.....	7, 20, 27, 44, 52
objekt	18, 21, 22, 23, 24, 25, 26, 30, 32, 33, 34, 42, 47, 48, 49, 51
Objekt.....	21, 34, 47
obljubah.....	8, 10
obljube .....	8, 10, 11, 42
odjemalca ..	5, 2, 6, 8, 18, 20, 21, 27, 46, 52
odjemalec – strežnik.....	9
odmik.....	7, 35, 58
ogrodje.....	8, 20
okno .....	34
okolje.....	7, 20, 55
operacije .....	7, 9
paket .....	52
platforme .....	8
podatek .....	7, 11, 42
podatke ...	2, 3, 9, 10, 12, 15, 22, 23, 24, 42, 49, 53, 54, 56, 57, 60, 61, 62, 64
Podatkovna baza.....	3, 5, 53
podatkovne baze ..	6, 7, 2, 11, 14, 17, 20, 21, 22, 27, 53, 56, 59, 67
podatkovnih baz .....	2, 9, 12, 20
podprocese.....	15, 16
poizvedb .....	8, 23
poizvedbe .....	5, 20, 21, 24
pomik.....	52, 58
ponovno nalaganje.....	6, 8, 29, 39, 43
<i>pool</i> .....	21, 22
posodabljanje.....	3, 29, 53
POST .....	6, 21, 22, 23, 24, 69
postopek .....	12, 15
pot.....	18, 23, 44, 45, 46, 47, 48
poti.....	7, 21, 23, 24, 44, 45
potomci.....	31, 32
povezave.....	3, 14, 18, 44, 49, 53
pravil.....	28
Preizkušanje.....	57
premikanje .....	34
prenos.....	15
prikaz ...	2, 6, 12, 20, 35, 36, 37, 44, 45, 57, 59
primarne ključe .....	12
primarni ključ.....	12, 53
Privzeta vrednost .....	32
proces .....	7, 15, 59
produkcijo .....	5, 52
produksijsko različico.....	27
programiranje.....	5, 2, 5, 7, 9, 29
<i>props</i> .....	49, 51, 72
razred .....	51
razvijalcem.....	5, 7, 8
React	3, 5, 7, 5, 6, 7, 17, 18, 20, 27, 28, 29, 30, 33, 39, 40, 43, 44, 46, 67, 68, 69, 70, 71, 72, 73
React.js.....	3, 5, 17, 18, 20, 27, 67, 71, 73
Reacta .....	6, 43, 49
Reactom .....	6, 8
referenca .....	36
relacija.....	12, 14
relacijsko shemo .....	14
rezultat .....	10, 11, 22
<i>router</i> .....	21, 44, 69, 71, 72
segment .....	45
<i>server</i> .....	20, 52, 55, 56
sinhronizaciji.....	40
sintakso .....	4, 5, 11
sistem .....	1, 9, 12, 20, 42, 53, 61, 63
Sistem uporabnikov .....	5, 61
sklada .....	45
slika.....	24, 25, 26
sliko.....	11, 13, 24, 25, 26, 54, 63, 65
Spletna trgovina .....	5, 6, 57
spremenljivk okolja .....	7, 21, 56
spremenljivke.....	11, 17, 28, 29, 31, 35, 40, 43, 55, 56, 65
Spremenljivke okolja.....	7, 55
SQL.....	5, 6, 7, 9, 22, 53, 59, 63, 69, 74
<i>SQL injection</i> .....	22
stanj.....	18, 19, 29, 30, 31, 33, 50, 61
stanja ...	8, 19, 20, 29, 30, 31, 33, 40, 42, 44
stanje .....	18, 25, 30, 32, 34, 42, 64
<i>state</i> .....	47, 48, 49, 70
statično .....	7, 17, 52, 58
<i>status</i> .....	22, 23, 69
stil .....	4, 6, 51

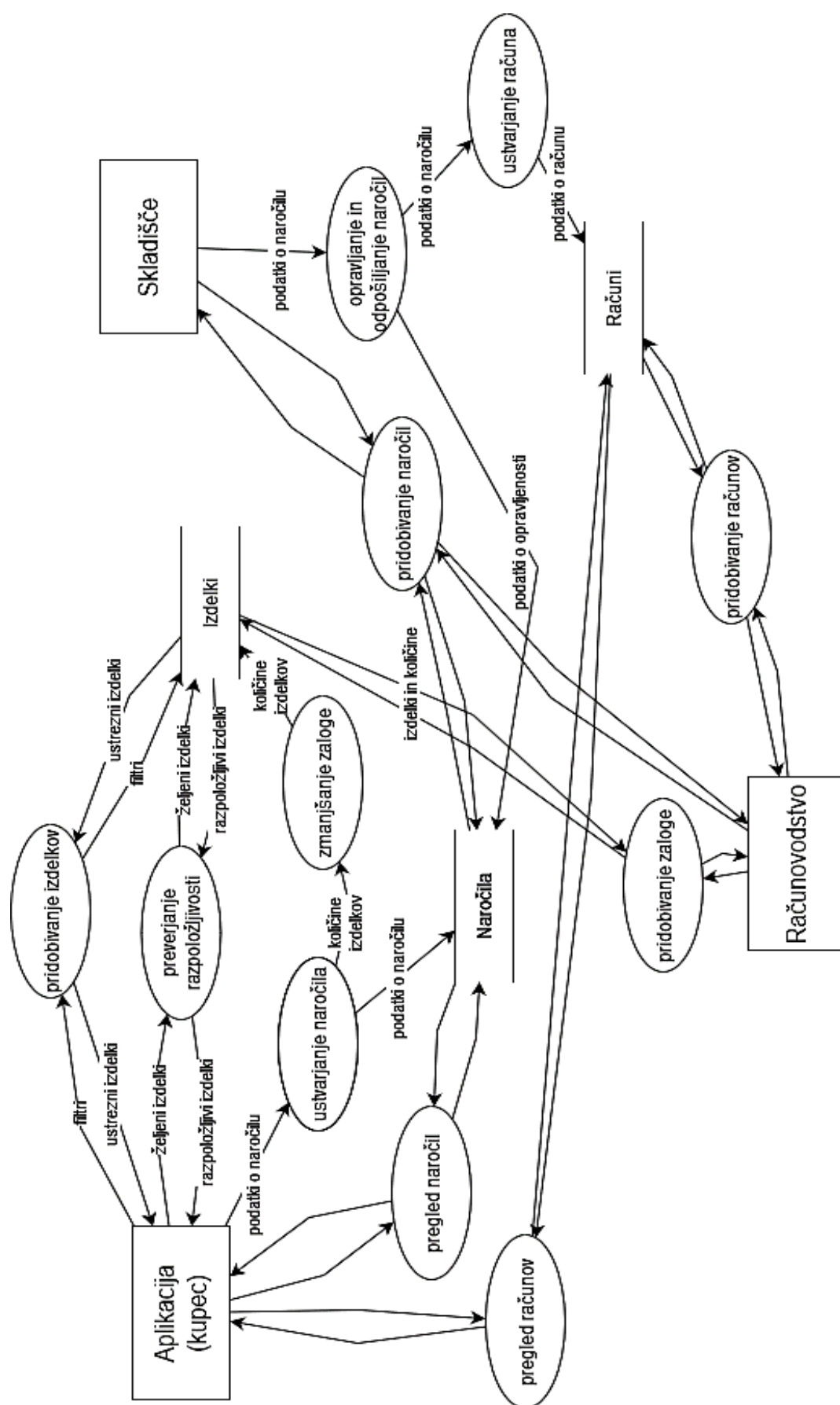
stran ....5, 2, 5, 6, 17, 20, 27, 36, 40, 44, 48,  
 52, 57, 58, 60, 62  
 strežnik ...3, 6, 9, 20, 21, 25, 52, 53, 55, 56,  
 59, 60, 62  
 strežniški del.....17  
 strukture.....2, 12  
 strukturo .....2, 6, 17, 53  
*submit* .....39  
 SUPB .....2, 9  
 šifrirano .....55  
 tabel .....9, 53  
 testiranje .....3, 12, 57, 60  
 Testiranje .....5, 57  
 tuji ključ.....13, 53, 54  
 učinki .....40  
 ukazi .....9  
 uporabniški vmesnik .....17, 18  
 uporabniško ime .....12, 13, 33, 47, 53, 62  
 upravljanje .....2, 9, 20, 21, 53, 62, 63  
 urejevalnik .....53  
 URL 6, 8, 21, 22, 23, 24, 26, 44, 45, 46, 48,  
 49, 52, 70  
 useCallback .....7, 43, 44, 71, 73  
 useContext .....7, 31, 32, 71  
 useEffect.....7, 34, 35, 40, 42, 44, 71, 73  
 useLocation() .....48, 49

useNavigate() .....48  
 useRef .....6, 7, 33, 36, 37, 40, 71  
 useState .....6, 29, 30, 33, 42, 70  
 usmerjanje .....6, 8, 18, 21, 44, 46  
 uvoze .....17, 18  
 uvozov .....27  
*vanilla* .....6  
 virov .....2, 17, 21, 46  
 vlogo .....13, 42, 55, 62  
 vmesna programska oprema .....8  
 vmesnik .....2, 6, 7, 8, 17, 18, 27, 29, 65  
 vnosno polje.....25, 64  
 vozlišče .....6  
 vrata .....17, 21, 52, 56  
 vrednost 3, 4, 10, 11, 13, 26, 29, 30, 31, 32,  
 33, 34, 37, 43, 44, 48, 53, 54  
 vrednosti 6, 4, 9, 12, 22, 28, 29, 31, 32, 36,  
 37, 40, 43, 49, 50, 56, 61, 62, 63  
 vrstica.....14, 35, 57, 58  
 vrstični CSS .....4  
 zagon.....7, 52  
 zahteve .....7, 8, 9, 20, 21, 22, 23, 24  
 Zgradba .....12  
 značka .....2, 28  
 zunanjim oblikovanjem .....4

## Priloga 1: ER diagram



## Priloga 2: DTP nivoja 1




Priloga 3: Posnetek zaslona spletne trgovine



Priloga 4: Posnetek zaslona komponente za podatke o nakupu

Podatki o nakupu

 gost  
2698.47 €

Podatki o kupcu:

Ime:


Priimek:


Naslov:

Naslov za dostavo:

Naslov za dostavo:

Način dostave:

☒ Pošta Slovenije  + 0.00 €

☐ Hitra pošta  + 3.00 €


Pregled košarice


Produkt	Količina	Cena/kos
SAMSUNG GALAXY TAB S6 Lite 2022	3	449.90 €
Samsung S21 FE 5G, Novi telefon Samsung	2	585.89 €
DELL P2422H monitor, 60,45 cm, Novi monitor Dell	1	176.99 €


Stroški dostave: 0 €

Za plačilo: **2698.47 €**

Način plačila:

☐ Po prevzemu 

 Nazaj

Oddaj naročilo 

## Priloga 5: Registracija uporabnikov

### Registracija

Pozdravljeni, registrirajte se in pridobite možnost hitrejšega spletnega naročanja.

Že imate račun? [Prijava](#)

Uporabniško ime:

Geslo:

Ponovite geslo:

E-naslov:

Ime:

Priimek:

Ulica in hišna št.:

Poštna št. in kraj:


Telefonska številka:


Podjetje:


[Potrdi](#)




## Profil: erik123

 Pregled naročil

 Pregled računov

Uredi 

Spremeni geslo 

uporabnisko\_ime:

erik123

elektronski\_naslov:

erik123@gmail.com

ime:

Erik

priimek:

Radovičević

ulica\_in\_hisna\_stevilka:

Ulica 5

kraj:

Novo mesto


postna\_stevilka:


8000


podjetje:


geslo:

●●●●●●●●●●●●●●●●

Shrani spremembe 


Ponastavi 

Izbriši račun 

Odjava 

## Priloga 7: Podrobnosti izdelka

✕ Izbriši iz PB

ID_izdelka:	78	
ime:	<input type="text" value="Sony PlayStation 5 igralna konzola"/>	<input type="button" value="Potrdi"/>
kategorija:	<input type="text" value="igralne konzole"/>	<input type="button" value="Potrdi"/>
cena_za_kos:	<input type="text" value="545 €"/>	<input type="button" value="Potrdi"/>
kosov_na_voljo:	<input type="text" value="1"/>	<input type="button" value="Potrdi"/>
kratek_opis:	<input type="text"/>	<input type="button" value="Potrdi"/>
informacije:	<div> Igralna konzola  PlayStation 5 (C šasijska)  prinaša osupljivo  zmogljivost za igranje  videoiger. Opremljena je z  visoko zmogljivim </div>	<input type="button" value="Potrdi"/>
popust:	<input type="text" value="0 %"/>	<input type="button" value="Potrdi"/>
slika:	<div>  </div> <div> <input type="button" value="Browse..."/> No file selected. <input type="button" value="Naloži novo sliko"/> </div>	

Priloga 9: Povezava do GitHub repozitorija *AplikacijaMatura*

<https://github.com/rade02/AplikacijaMatura>

Priloga 8: Povezava do GitHub mape *client*

<https://github.com/rade02/AplikacijaMatura/tree/main/client>

Priloga 10: Povezava do GitHub mape *server*

<https://github.com/rade02/AplikacijaMatura/tree/main/server>

Priloga 11: Povezava do GitHub mape *docs*

<https://github.com/rade02/AplikacijaMatura/tree/main/docs>