Potpora starih i novih oblika mobilnosti osoba informacijsko komunikacijskom tehnologijom

Pivk, Luka

SVEUČILIŠTE JOSIPA JURJA STROSSMAYERA U OSIJEKU
FAKULTET ELEKTROTEHNIKE, RAČUNARSTVA I INFORMACIJSKIH TEHNOLOGIJA OSIJEK

Sveučilišni diplomski studij Računarstvo

POTPORA STARIH I NOVIH OBLIKA MOBILNOSTI OSOBA INFORMACIJSKO KOMUNIKACIJSKOM TEHNOLOGIJOM

Diplomski rad

Luka Pivk

Osijek, 2024.

FERIT
FAKULTET ELEKTROTEHNIKE, RAČUNARSTVA I INFORMACIJSKIH TEHNOLOGIJA **OSIJEK**

Obrazac D1: Obrazac za ocjenu diplomskog rada na sveučilišnom diplomskom studiju

Ocjena diplomskog rada na sveučilišnom diplomskom studiju

Ime i prezime pristupnika:	Luka Pivk
Studij, smjer:	Sveučilišni diplomski studij Računarstvo
Mat. br. pristupnika, god.	D-1151R, 13.10.2020.
JMBAG:	0165075374
Mentor:	prof. dr. sc. Dominika Crnjac Milić
Sumentor:	prof. dr. sc. Krešimir Grgić
Sumentor iz tvrtke:	
Predsjednik Povjerenstva:	izv. prof. dr. sc. Zdravko Krpić
Član Povjerenstva 1:	prof. dr. sc. Dominika Crnjac Milić
Član Povjerenstva 2:	doc. dr. sc. Bruno Zorić
Naslov diplomskog rada:	Potpora starih i novih oblika mobilnosti osoba informacijsko komunikacijskom tehnologijom
Znanstvena grana diplomskog rada:	**Programsko inženjerstvo (zn. polje računarstvo)**
Zadatak diplomskog rada:	Sve veća potreba za kretanjem te brzi način života generirali su potrebu za osmišljavanjem novih oblika prijevoza osoba. Urbane sredine suočavaju se s nizom problema, kao što su prometna zagušenja te onečišćenje okoliša uzrokovano prometom. Globalizacija je pridonijela mobilnosti kao bitnom čimbeniku ljudskog života. Kretanje po različitim gradovima svijeta koje zahtijeva brzo snalaženje sve više je prisutno. Isto tako racionaliziranje troška prijevoza može implicirati znatne uštede za sudionike u prometu. Radom je potrebno analizirati tehnički
Datum ocjene pismenog dijela diplomskog rada od strane mentora:	22.09.2024.
Ocjena pismenog dijela diplomskog rada od strane mentora:	Izvrstan (5)
Datum obrane diplomskog rada:	25.09.2024
Ocjena usmenog dijela diplomskog rada (obrane):	Izvrstan (5)
Ukupna ocjena diplomskog rada:	Izvrstan (5)
Datum potvrde mentora o predaji konačne verzije diplomskog rada čime je pristupnik završio sveučilišni diplomski studij:	01.10.2024.

FERIT
FAKULTET ELEKTROTEHNIKE, RAČUNARSTVA I INFORMACIJSKIH TEHNOLOGIJA **OSIJEK**

IZJAVA O IZVORNOSTI RADA

Osijek, 01.10.2024.

Ime i prezime Pristupnika:	Luka Pivk
Studij:	Sveučilišni diplomski studij Računarstvo
Mat. br. Pristupnika, godina upisa:	D-1151R, 13.10.2020.
Turnitin podudaranje [%]:	5

Ovom izjavom izjavljujem da je rad pod nazivom**: Potpora starih i novih oblika mobilnosti osoba informacijsko komunikacijskom tehnologijom**

izrađen pod vodstvom mentora prof. dr. sc. Dominika Crnjac Milić

i sumentora prof. dr. sc. Krešimir Grgić

moj vlastiti rad i prema mom najboljem znanju ne sadrži prethodno objavljene ili neobjavljene pisane materijale drugih osoba, osim onih koji su izričito priznati navođenjem literature i drugih izvora informacija.
Izjavljujem da je intelektualni sadržaj navedenog rada proizvod mog vlastitog rada, osim u onom dijelu za koji mi je bila potrebna pomoć mentora, sumentora i drugih osoba, a što je izričito navedeno u radu.

Potpis pristupnika:

SADRŽAJ

1. UVOD

Prema procjeni UN-a [1] 2007. godine dogodio se povijesni trenutak za ljudsku civilizaciju – broj ljudi koji žive u urbanim sredinama po prvi puta u povijesti postao je veći od broja ljudi koji žive u ruralnim sredinama. Od tada se trend urbanizacije nastavio i nema naznaka posustati pa sada više od 4.5 milijarde ljudi živi u gradovima [2]. Takav nagli porast gradskog stanovništva stvorio je gradovima niz problema koji su najčešće vezani uz promet: zagušenost i preopterećenja na određenim prometnim linijama u određeno doba dana, prevelika buka i onečišćenje okoliša samo su neki od njih. Dodatno, početkom dvadesetih godina 21. stoljeća, globalna financijska i energetska kriza izazvana pandemijom virusa COVID-19 te ruskom invazijom na Ukrajinu uzrokovala je značajna poskupljenja svih vrsta energenata pa je time pretjerana ovisnost o prometu, posebno o osobnim vozilima pogonjenim na naftne derivate, počela predstavljati i ogromne novčane troškove za svakog pojedinca [3].

U mnogim europskim gradovima, investicije u javni prijevoz i biciklističke staze uspjele su privući građane da odbace osobne automobile i počnu sve više koristiti bicikle, romobile, vlakove, tramvaje i autobuse, no njihov utjecaj nije dovoljno velik da bi činio značajnu razliku. Oko 80% svih prijeđenih kilometara stanovnika Europske unije 2021. godine svejedno je otpao na automobilski prijevoz [4]. Stoga, jasno je da su Europi potrebne dodatne investicije u promet te nova, moderna rješenja.

Ovome radu cilj je analizirati tehnički, ekonomski i pravni aspekt postojećih rješenja za individualni prijevoz u Republici Hrvatskoj i svijetu te predložiti poboljšanja vezana za sustave informiranja osoba o najbržoj mogućnosti dolaska na željenu destinaciju, fokusirajući se na vozila pogonjena obnovljivim izvorima energije ili javni prijevoz. Mobilnom aplikacijom danom ovim radom korisnik bi doprinosio smanjenju ranije nabrojanih negativnih utjecaja urbanizacije na promet, a istovremeno bi putovao na željene destinacije u zadovoljavajućem vremenu. Također, razne tvrtke, čije je područje djelovanja u prometu, dobile bi platformu za oglašavanje i pružanje vlastitih usluga. Pri izradi rješenja spomenute su značajke postojećih aplikacija slične upotrebe. Zatim, detaljno su objašnjene tehnologije i principi korišteni pri njegovoj izradi u programskom jeziku JavaScriptu koristeći razvojne okvire poput Node.js-a, NestJS-a, Reacta i React Nativea, kao i ideje osnovnih računalnih paradigmi i tehnologija poput REST arhitekture te SQL baza podataka. Nadalje, opisano je kako su navedene tehnologije primijenjene na zadatak rada i na koji način su doprinijele rješavanju problema. Na kraju, ukratko je opisano korisničko sučelje te kako bi se korisnici trebali služiti danim rješenjem.

2. POSTOJEĆA RJEŠENJA I IZAZOVI NJIHOVE IMPLEMENTACIJE

Iako masovna urbanizacija za vrijeme industrijskih revolucija na hrvatske gradove nije imala utjecaj kao na London, Pariz ili Berlin, a kamoli gradove Sjeverne Amerike i Azije, početkom 90-ih godina 20. stoljeća i padom socijalizma u istočnoj Europi, njeni efekti počeli su se jasno osjetiti i u najvećim središtima Republike Hrvatske [5]. Naime, u političkom sustavu Socijalističke Federativne Republike Jugoslavije (SFRJ) postojale su institucije za urbano planiranje koje su zapošljavale eksperte iz raznih područja djelatnosti čije je mišljenje bilo cijenjeno i nerijetko – jedino ispravno. Ekonomski aspekti i volja građana nisu imali veliki utjecaj pri donošenju odluka o urbanom planiranju. To je značilo da su se gradovi širili sporo i planski, često nauštrb bržeg demografskog i gospodarskog razvoja. Raspadom SFRJ i uvođenjem kapitalističkog sustava u Hrvatskoj, važna urbana središta doživjela su brzi demografski rast i priljev kapitala, no ti procesi nisu bili praćeni odgovarajućim urbanističkim planiranjem, što je dovelo do niza infrastrukturnih problema poput nekontrolirane izgradnje stambenih objekata, stvarajući probleme u mnogim aspektima gradskog života, pa tako i u prometu [5]. Kao i u ostatku svijeta, potreba građana za prometnom infrastrukturom uslijed rasta korištenja osobnih automobila i stvarni kapaciteti postojećih prometnih sustava postajali su neusklađeni. Velik broj novih prigradskih naselja velikih gradova nije bio povezan adekvatnim cestovnim mrežama, dok se broj vozila na cestama znatno povećavao [6]. Takav ubrzani razvoj doveo je do zagušenja u prometu, osobito u centralnim dijelovima gradova, gdje su prometne gužve svakodnevna pojava sve do danas. Osim gubljenja vremena u gužvama, urbanizacijom nastali problemi poput nedostatka parkinga, velike buke, onečišćenja okoliša te rast cijena naftnih derivata, također značajno doprinose smanjenju kvalitete života građana.

Diljem svijeta ovi problemi su prisutni već desetljećima, međutim vlastima država i gradova je s pravne i ekonomske strane gotovo nemoguće spriječiti negativne utjecaje nagle urbanizacije. Zbog brzog rasta stanovništva i infrastrukture, vlasti često nemaju dovoljno vremena ili resursa da unaprijed isplaniraju adekvatnu prometnu infrastrukturu kako bi udovoljile potrebama stanovništva. Razvoj održivih urbanih sredina zahtijeva balansiranje ekonomskih, društvenih i okolišnih ciljeva te koordiniranje različitih sektora, uključujući gradnju, transport, ekologiju i gospodarstvo, što može značajno otežati i usporiti provedbu pravnih mjera. Pri tome, mjere koje bi ograničile rast broja automobila i stambenih prostora mogu naići na žestok otpor zbog ekonomskih interesa, posebno privatnih tvrtki u transportnom i građevinskom sektoru. Osim toga, ljudi su skloni koristiti osobne automobile zbog udobnosti, a navike teško mijenjaju čak i kada postoje alternative poput kvalitetnog javnog prijevoza ili biciklističkih staza. Navedeni čimbenici

čine prevenciju negativnih učinaka urbanizacije iznimno izazovnom te zahtijevaju dugoročno planiranje i značajne financijske i organizacijske resurse [7].

Jedna od rijetkih uspješnih mjera, koja je, također, naišla na žestok otpor privatnog automobilskog sektora, stroga je regulacija Europske unije za smanjenje emisija ugljičnog dioksida (CO2) iz automobila [8]. Ona postavlja ciljeve za smanjenje emisija CO2 iz novih automobila i lakih komercijalnih vozila. Od 2025. do 2029. cilj je smanjiti emisije novih automobila za 15%, od 2030. do 2035. za 55%, a do 2050. godine svi novi automobili trebali bi imati 0 grama CO2 emisije po prijeđenom kilometru, što znači potpuno izbacivanje vozila s unutarnjim izgaranjem iz prodaje. Dodatno, u mnogim državama postoji mehanizam novčanih poticaja za kupnju električnih vozila ili vozila s niskim emisijama CO2, kao i poticaji za prodavače koji ispune ciljeve prodaje takvih vozila, dok proizvođači koji premaše svoje emisijske ciljeve plaćaju novčane kazne.

2.1. Javne investicije

U Zagrebu, glavnome gradu i gospodarskom centru Hrvatske, zbog broja stanovnika, ranije spomenuti problemi su najizraženiji. Iako u značajno manjem obujmu, slična situacija dogodila se i Osijeku, najvećem gradu istočne Hrvatske koji je, također, prošao tranziciju u suvremeno urbano središte i time se susreo s problemom modernizacije javnog prijevoza, nedostatkom parkirnih mjesta i prometnim gužvama na pristupnim točkama grada. Velik dio problema u prometu Osijek je riješio 2015. godine završetkom gradnje brze ceste oko grada, tj. Osječke obilaznice, a odgovore na pitanja kako nastavlja unaprjeđenje prometne infrastrukture, u sklopu ovog rada, potraženi su u upravi Grada Osijeka. Postavljena pitanja i dobiveni odgovori u polustrukturiranom intervjuu s dogradonačelnikom Grada Osijeka dr. sc. Draganom Vulinom preneseni su u cijelosti u nastavku. Cilj intervjua bio je dobiti bolji uvid u funkcioniranje javnih ustanova Republike Hrvatske i njihove perspektive o tehničkim, ekonomskim i pravnim aspektima rješenja za probleme urbanizacije i mobilnosti.

- **Koliko je Republika Hrvatska otvorena prema novim oblicima mobilnosti i jesu li im hrvatski zakoni o prometu prilagođeni?**

„Hrvatska je pokazala značajan interes za nove oblike prometa, uključujući električna vozila, biciklističku infrastrukturu i različite oblike mikromobilnosti poput električnih romobila. Hrvatski zakoni ponekad znaju biti ne usklađeni obzirom na brzinu uvođenja i stupanj razvitka novih oblika prometa, ali su nadležna tijela Republike Hrvatske otvorena za razgovore i izmjene zakonskih regulativa radi poboljšanja i sigurnosti novih oblika prometa. Vidljiv je pozitivan trend prema prihvaćanju i integraciji novih oblika mobilnosti u svakodnevni život.“

- **Koje projekte je Grad Osijek proveo proteklih godina u svrhu poboljšanja prometa?**

„Grad Osijek je u proteklih nekoliko godina proveo nekoliko značajnih projekata usmjerenih na poboljšanje prometa, s naglaskom na javni prijevoz te promociju električnih vozila, bicikala i romobila. Trenutno najveća investicija je projekt Modernizacija tramvajske infrastrukture Grada Osijeka kojom će se modernizirati 9.5 kilometara tramvajske pruge i kontaktnog vodiča, tri ispravljačke stanice te 23 nova stajališta po najvećim standardima čime će postati najmodernija pruga u ovom dijelu Europe.

Grad Osijek kontinuirano radi na unapređenju autobusnog prijevoza, uključujući obnovu voznog parka s novim, ekološki prihvatljivijim autobusima. Kroz Europske fondove Gradski prijevoz putnika Osijek (GPP Osijek) je nabavio 13 autobusa najvišeg EURO 6 standarda. Nadalje, GPP Osijek je kroz Nacionalni plan oporavka i otpornosti potpisao Ugovor o dodjeli bespovratnih sredstava za nabavku deset novih niskopodnih tramvaja te se očekuje do kraja tekuće godine potpisivanje ugovora za još deset tramvaja čime će GPP Osijek modernizirati cjelokupnu tramvajsku flotu najnovijim niskopodnim tramvajima.

Projektom E-mobilnost grada Osijeka uspostavio se sustav dijeljenog korištenja bicikala kao nadopune javnog prijevoza kroz izgradnju infrastrukture i nabavu bicikala i opreme [9]. Opremljeno je ukupno 25 samposlužnih stanica raspoređenih na prostoru Grada Osijeka. Uspoređujući s prosječnim vrijednostima koje određuju veličinu sustava, s tim brojem samposlužnih stanica sustav javnih bicikala u Osijeku ima više od dvije stanice na 10 000 stanovnika – čime se svrstao među gradove prosječnih vrijednosti na razini Europske unije prema istraživanju OBIS-a (engl. *Optimising Bike Sharing in European Cities*) [9]. U sklopu projekta nabavljeno je 175 bicikala od čega ih je 50 na električni pogon.“

- **Koji su bili najveći izazovi tih projekata?**

„Najveći izazovi su bili složenost i dugotrajnost procesa Javne nabave, administrativni izazovi vezani uz dozvole i suglasnosti te koordinacija između institucija.“

- **Koji su budući planovi grada Osijeka u vezi daljnjeg razvoja održivih oblika mobilnosti?**

„Trenutni budući planovi su proširenje i optimizacija lokacija vezanih uz Sustav dijeljenih bicikala, dodatno proširenje tramvajskih linija i dodatna elektrifikacija autobusnog voznog parka.“

- **Postoje li projekti koji su već u fazi planiranja ili realizacije?**

„Trenutno je u fazi izrada projekta nabavke novih 17 električnih autobusa najnovije generacije s popratnom infrastrukturom. Nabavlja se dodatnih 10 novih tramvaja. Radi se na proširenju postojećih tramvajskih linija koje će spajati Novu Remizu za koju već postoji Idejno rješenje. Grad Osijek zajedno sa GPP-om Osijek, također, razmatra uvođenje *Park & Ride* sustava te *car-sharing* opcije.

Park & Ride sustav dozvoljava građanima da parkiraju auto na određenoj lokaciji, a zatim koriste javni prijevoz za transport do konačnog odredišta čime se nastoji smanjiti upotreba osobnih automobila u gradskim centrima.

Car-sharing je model korištenja vozila po principu iznajmljivanja. Umjesto posjedovanja vozila, korisnik iznajmljuje vozilo na određeno vrijeme na samoposlužnim lokacijama, ali za razliku od *rent-a-car* sustava najam je moguć na znatno kraće vrijeme i bez iscrpne popratne papirologije."

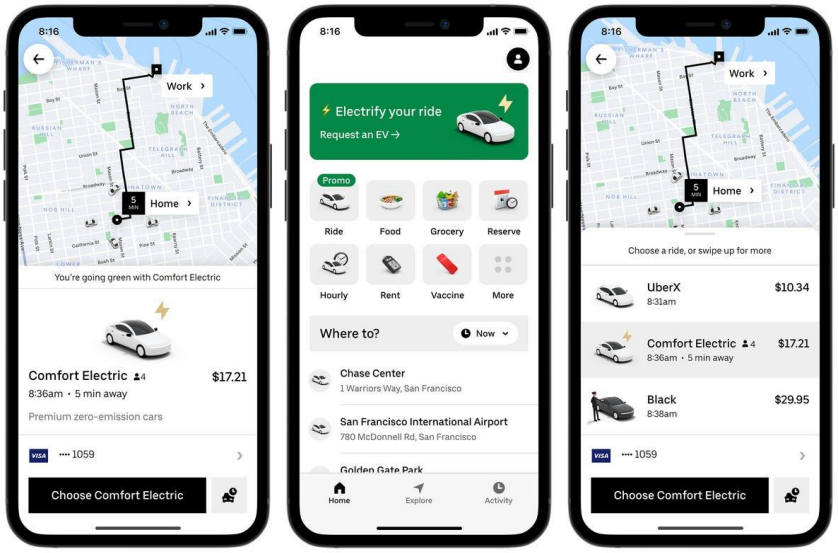
Na pozitivnom primjeru grada Osijeka, jasno je da u Hrvatskoj postoji niz instrumenata koje javne institucije mogu koristiti za poboljšanje gradskog prometa, istovremeno fokusirajući se na njegove održive oblike. Međutim, iz odgovora se, također, može zaključiti da njihovi pravni aspekti poput zakona o Javnoj nabavi i ostalih birokratskih procesa u nekoj mjeri mogu usporiti investicije, čiji se ekonomski resursi većinom fokusiraju na moderniziranje postojeće infrastrukture, a ne nužno na digitalna rješenja.

2.2. Privatne investicije

Osim javnih investicija gradskih i državnih institucija, i privatne investicije mogu biti instrument za provođenje promjena u prometu. Primjeri takvih investicija u svijetu, uključujući i Hrvatsku, dva su digitalna globalna velikana – Uber i Bolt. To su primarno mobilne aplikacije koje su značajno utjecale na poboljšanje gradskog prometa u Hrvatskoj, osobito u najvećim gradovima poput Zagreba i Osijeka, ali i drugih. Njihov ulazak na hrvatsko tržište donio je niz promjena u načinu na koji se građani kreću gradom.

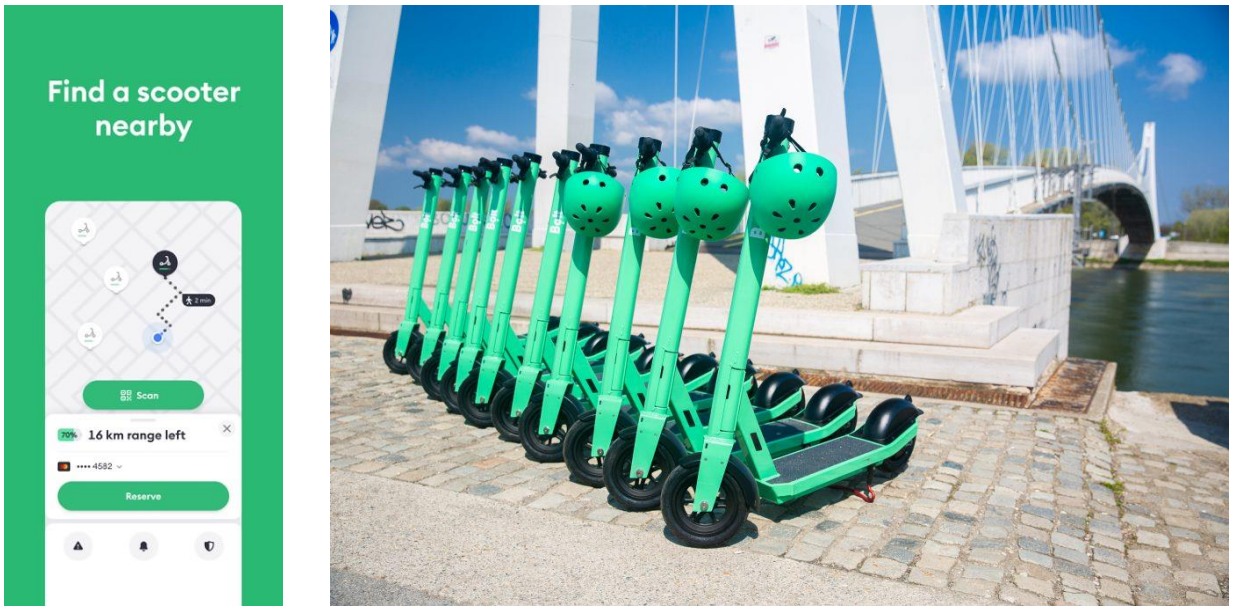
Uber je u Hrvatsku stigao 2015. godine, prvo u Zagreb, a kasnije i u drugim većim gradovima, nudeći alternativu tradicionalnim taksi službama. Jedna od glavnih prednosti Ubera je pristupačnost i fleksibilnost [10]. Korisnici putem mobilne aplikacije mogu brzo i jednostavno naručiti vožnju u bilo koje vrijeme i s bilo koje lokacije, što je posebno korisno u velikim gradovima, gdje javni prijevoz nije uvijek pouzdan, osobito tijekom noćnih sati. Cijene vožnje putem Ubera često su konkurentnije u odnosu na tradicionalne taksi službe, što je direktno privuklo

veliki broj korisnika. Osim toga, Uber Green usluga, koja koristi električna vozila, dodatno je doprinijela smanjenju emisija štetnih plinova. Na slici 2.1. izgled je glavnog sučelja Uber mobilne aplikacije na kojem se pruža usluga prijevoza, odnosno odabira destinacije na karti i potvrda vožnje.



Sl. 2.1. Glavno sučelje Uber mobilne aplikacije [10]

Bolt, poznat kao bivši Taxify, ušao je na hrvatsko tržište 2017. godine, ubrzo nakon Ubera. U početku, nudio je gotovo identičnu uslugu kao i Uber, međutim, u posljednjih nekoliko godina istaknuo se na tržištu razvojem mikromobilnosti. Osim standardnih vožnji, Bolt je uveo usluge dijeljenja električnih romobila, što je dodatno poboljšalo mogućnosti kratkih putovanja u gradovima [11]. Jedan od gradova obuhvaćenih tim projektom je i Osijek. Korištenjem električnog romobila smanjuje se ovisnost o automobilima za kratke rute, što pomaže u smanjenju prometnih gužvi te pri pronalaženju parkirnog mjesta, koji su česti problemi u urbanim sredinama. Ekološki aspekt također igra značajnu ulogu. Električni romobili su vozila na baterije, što znači da ne ispuštaju štetne emisije i time doprinose smanjenju zagađenja zraka u gradovima. Ova vrsta mobilnosti usklađena je s globalnim trendovima koji teže prema održivijim oblicima urbanog transporta. Na slici 2.2. s lijeva je prikaz sučelja za vožnju električnog romobila u Bolt mobilnoj aplikaciji, a s desna verzija romobila koji su dostupni u Osijeku.



Sl. 2.2. Sučelje Bolt mobilne aplikacije i osječki Bolt električni romobili [11]

Zbog načina kojim su demonstrirale kako moderne tehnologije mogu doprinijeti boljem, održivijem i raznovrsnijem gradskom prometu, ove dvije mobilne aplikacije poslužile su ovome radu kao inspiracija za razvoj proširenog softverskog rješenja koje objedinjuje usluge nalik njihovima s uslugama javnog gradskog prijevoza.

Slične ideje o višestrukom objedinjavanju raznih oblika prometa koje bi uključivale i javni prijevoz, zbog svoje kompleksnosti u praksi još uvijek nisu u potpunosti realizirane. Dosada su u svijetu viđeni samo pokušaji takvih sustava i aplikacija, a Europa je najbliža lansiranju jednog takvog projekta u slobodnu upotrebu. Naime, riječ je o IP4MaaS (engl. *Interoperable Profile for Mobility as a Service*), inicijativi u kojoj sudjeluje nekolicina država i privatnih tvrtki, usmjerenoj na standardizaciju i unaprjeđenje interoperabilnosti različitih sustava unutar koncepta „*Mobility as a Service*“ (MaaS) [12]. MaaS je ideja koja integrira različite vrste prijevoza u jedinstveni digitalni servis te time omogućava korisnicima planiranje, rezerviranje i plaćanje putovanja kroz jednistvenu platformu, odnosno aplikaciju. IP4MaaS projekt je u završnim fazama, a glavne aktivnosti službeno su završene u lipnju 2023. godine. Projekt je uključivao šest demonstracijskih lokacija, među kojima je bio i Osijek, gdje su testirani različiti aspekti multimodalnog putovanja poput alata za planiranje, praćenje i dijeljenje putovanja integrirani u jedinstvenu aplikaciju pod nazivom Suputnik (engl. *Travel Companion*). Rezultati testova su se u trenutku pisanja rada analizirali kako bi se procijenili zadovoljstvo korisnika i potencijal za širu primjenu diljem Europe.

Iako su Uber, Bolt i IP4MaaS projekt predvodnici inovacija koji vlastitim uslugama drastično poboljšavaju promet u velikim gradovima, također su indikator velikog problema tog sektora. Uber je globalni gigant s godišnjim prihodima od 38 milijardi dolara i preko 30 000 zaposlenika [10], Boltovi godišnji prihodi iznose preko 1.7 milijarde eura [11], a IP4MaaS je projekt u koji su uključeni desetci globalnih kompanija te države Europske unije. Iz ovih podataka jasno je da, ako već nemate globalno poslovno djelovanje s prihodima u milijardama eura ili dolara, natjecati se s njima u tržišnoj utakmici je gotovo nemoguće. Dodatno, Uber i Bolt na korak su da potpuno monopoliziraju tržište na kojem djeluju. Vrlo je teško pronaći veliki grad u najrazvijenijim zemljama svijeta u kojem Uber ne djeluje i u kojemu njegova aplikacija nije najbolja opcija za taksi usluge. Naime, Uber djeluje u čak 10 500 gradova diljem svijeta [10], a slična situacija je i s uslugom mikromobilnosti putem električnih romobila koje nudi Bolt, koja je trenutno dostupna u preko 100 europskih gradova s preko 130 000 tisuća vozila [11]. Time ove dvije kompanije, osim ogromne financijske moći, imaju i monopol nad inovacijama u dijelovima prometnog sektora u kojem djeluju lako preuzimajući i implementirajući usluge financijski značajno slabije konkurencije. Ovisnost o nekoliko privatnih kompanija je situacija koju bi svako područje djelatnosti trebalo nastojati izbjeći i veliki je ekonomski problem za bilo koju potencijalnu privatnu investiciju u promet i mobilnost.

3. TEHNOLOGIJE I ALATI POTREBNI ZA IZRADU APLIKACIJA

U suvremenom svijetu razvoja softvera, na klijentskoj strani aplikacija, React predstavlja najpopularniju JavaScript biblioteku za kreiranje interaktivnih korisničkih sučelja. React Native proširuje njenu mogućnost na mobilne platforme, omogućavajući razvoj aplikacija za iOS i Android operacijske sustave korištenjem istih principa i tehnika. S druge strane JavaScript ekosustava, NestJS je jedno od najpopularnijih okruženja za izgradnju poslužiteljskih aplikacija temeljeno na Node.js razvojnom okviru. Ova tri alata, zajedno s tehnologijama koje unutar sebe obuhvaćaju, te alternativnim klijentskim bibliotekama poput Angulara, formiraju snažan skup tehnologija koji programerima omogućava da od vrha do dna izgrade kompleksne, visoko interaktivne web i mobilne aplikacije u istom programskom jeziku. Jasno je, stoga, da je za njihovo korištenje neophodno duboko razumijevanje JavaScripta.

Uz navedeno, poznavanje TypeScripta, nadogradnje za JavaScript koji jeziku dodaje statičke tipove podataka, također je postalo programerski standard jer gotovo sve JavaScript biblioteke i razvojna okruženja intenzivno koriste TypeScript za poboljšanje razvojnog iskustva, sigurnosti i pouzdanosti aplikacija.

Za uspješan razvoj aplikacija koristeći biblioteke React i React Native, potrebno je temeljno razumijevanje HTML-a i CSS-a, osnovnih alata za izradu strukturiranih i stiliziranih korisničkih sučelja.

Nadalje, za rad s NestJS-om, važno je razumijevanje koncepta razvoja poslužiteljskih aplikacija - REST arhitekture, HTTP-a, asinkronog programiranja u Node.js razvojnom okruženju, kao i znanje o upravljanju bazama podataka.

Ovo poglavlje pruža pregled svake od navedenih tehnologija objašnjavajući kako se svaka od njih može ili mora koristiti. Razumijevanje njihovih osnova omogućuje čitatelju da bolje shvati kako se razvijaju moderne aplikacije s JavaScriptom duž cijelog računalnog stoga te kako se integriraju klijentske i poslužiteljska strana aplikacije pri stvaranju kohezivnih i funkcionalnih digitalnih rješenja.

3.1. Representational State Transfer

Aplikacijsko programsko sučelje ili API (engl. *Application Programming Interface*) je set pravila i protokola za korištenje neke softverske aplikacije. Ono definira metode i formate prijenosnih podataka koje aplikacija može koristiti kako bi komunicirala s drugim programima,

aplikacijama ili hardverskim komponentama s vlastitim softverom. Ukratko, API je ugovor između dva softvera koji definira njihovu interakciju [13].

API-ji rješavaju nekolicinu važnih problema pri interakciji dva softvera:

- Sakrivaju unutarnju kompleksnost sustava izlažući samo set funkcionalnosti koje druga strana može koristiti bez da zna detalje implementacije.
- Omogućava raznorodnim softverima da komuniciraju međusobno, neovisno o programskom jeziku kojima su izrađeni ili platformi na kojoj se pokreću.
- Dozvoljavaju programerima da kompleksne sustave rastave na manje, upravljive komponente, što olakšava razvoj, testiranje i održavanje softvera.
- Omogućavaju programerima da izgrade aplikacije koje mogu lako evoluirati s vremenom kroz dodavanje ili mijenjanje funkcionalnosti bez utjecaja na korisničko iskustvo.
- Pružaju dodatni sloj sigurnosti kroz autentikaciju i autorizaciju zahtjeva, štiteći time podatke i funkcionalnosti aplikacije od neovlaštenog pristupa.

Web API-ji su skupovi pravila i protokola koji omogućuju interakciju između različitih softverskih aplikacija putem Interneta. Takvi API-ji dostupni su bilo kakvoj klijentskoj aplikaciji - web, desktop ili mobilnoj - koja podržava HTTP (engl. *HyperText Transfer Protocol*). Klijentska strana aplikacije šalje zahtjev serveru koji taj zahtjev obrađuje i zatim vraća odgovor. Podaci se zahtjevom i odgovorom obično prenose u formatima kao što su JSON (engl. *JavaScript Object Notation*) ili XML (engl. *eXtensible Markup* Language).

HTTP je temeljni protokol za prijenos podataka putem Interneta, a njegove najvažnije značajke prema [14], navedene su u nastavku:

- Svaki zahtjev koji klijent šalje poslužitelju je neovisan i mora sadržavati sve informacije potrebne za obradu zahtjeva jer poslužitelj ne pohranjuje podatke između dva zahtjeva.
- Veza između klijenta i poslužitelja otvara se na početku svakog HTTP zahtjeva i zatvara nakon što je odgovor na zahtjev poslan.
- Podržava različite metode među kojima su najpoznatije *GET* (za dohvaćanje podataka), *POST* (za slanje/spremanje podataka), *PUT* (za ažuriranje podataka) i *DELETE* (za brisanje podataka).

- Podržava različite vrste formata prijenosnih podataka. Osim JSON-a i XML-a, prijenosni podaci mogu biti datoteke, tokovi podataka i ostali.
- Svaki odgovor na zahtjev klijenta sadrži statusni kod prema kojemu klijent može dobiti više informacija o uspješnosti izvršavanja zahtjeva.

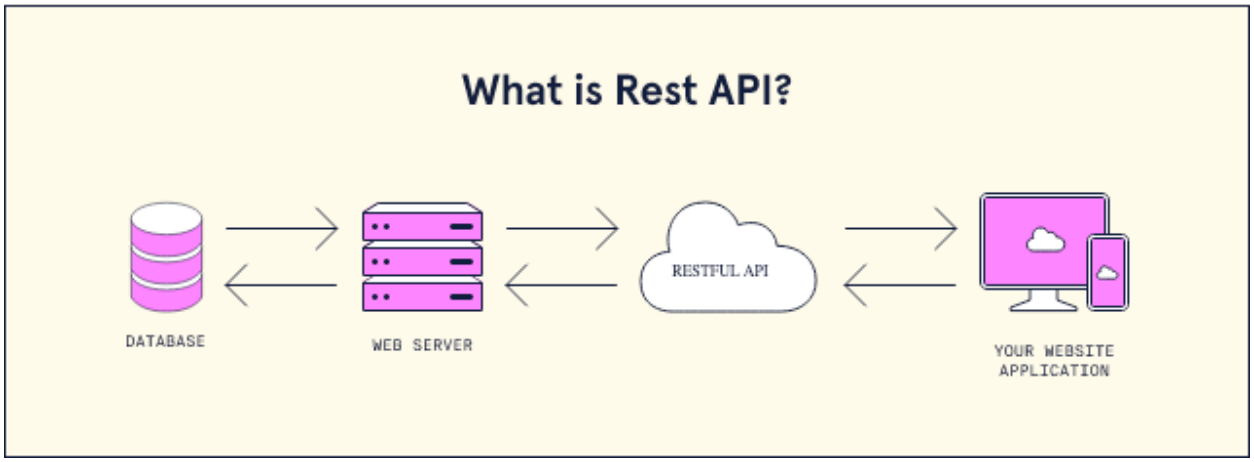
Najpoznatija implementacija Web API-ja je REST (engl. *Representational State Transfer*). REST je softverska arhitektura koja definira skup pravila za dizajniranje API-ja, s ciljem olakšavanja komunikacije unutar složenih mrežnih sustava poput Interneta. Omogućava izgradnju skalabilnih, visoko učinkovitih i pouzdanih web servisa. Također, REST arhitektura je jednostavna za implementaciju na različitim platformama, što je čini najpopularnijim rješenjem za razvoj modernih API-ja.

API-ji razvijeni prateći principe REST arhitekture poznati su kao REST API-ji, dok se web servisi koji ih implementiraju nazivaju RESTful web servisima. Termini REST API i RESTful API često koriste izmjenično jer se oba odnose na API-je koji su dizajnirani u skladu s ovim arhitektonskim stilom.

Kako REST API-ji koriste HTTP za prijenos podataka, njihove glavne značajke odgovaraju glavnim značajkama HTTP-a. Međutim, REST arhitektura implementira i neke dodatne, ali jednako važne principe opisane u nastavku:

- Uniformno sučelje je ključni element dizajna svakog RESTful web servisa koji zahtijeva standardiziranu komunikaciju između klijenta i poslužitelja. Poslužitelj šalje podatke u standardnom formatu, omogućavajući klijentima da identificiraju, modificiraju ili brišu resurse koristeći jedinstvene identifikatore i prateći putanje krajnjih točaka.
- RESTful web servisi mogu biti organizirani u slojevite arhitekture, gdje klijenti komuniciraju s posredničkim slojevima ili poslužiteljima, ne znajući za složenost računalne mreže iza njih, što omogućava veću skalabilnost i sigurnost sustava.
- RESTful web servisi podržavaju spremanje podataka u priručnu (engl. *cache*) memoriju na klijentskim slojevima aplikacija kako bi se smanjio broj nepotrebnih poziva prema poslužitelju.
- Poslužitelji mogu privremeno proširiti ili prilagoditi funkcionalnost klijenta slanjem izvršnog koda na klijentov zahtjev omogućavajući dinamičnije korisničko iskustvo.

Na slici 3.1. prikazana je jednostavna vizualizacija uloge RESTful API-ja koje se nalazi između poslužitelja i klijentskih aplikacija te omogućuje njihovu komunikaciju.



Sl. 3.1. Vizualizacija uloge RESTful API-ja [15]

REST API-ju može se pristupiti iz bilo kojeg sučelja koje podržava HTTP protokol, uključujući web pretraživače, mobilne aplikacije, desktop aplikacije i druge poslužiteljske aplikacije, što omogućava široku kompatibilnost i lakoću integracije RESTful web servisa u različite platforme i tehnologije.

3.2. JavaScript

JavaScript je programski jezik koji u 2024. godini dominira softverskim svijetom [16]. Iako je najpoznatiji po ključnoj ulozi u izgradnji korisničkih sučelja za web preglednike, s vremenom se proširio na gotovo sve aspekte razvoja softvera. JavaScript se danas koristi i za razvoj poslužiteljskih aplikacija pomoću platformi kao što je Node.js, mobilnih aplikacija putem razvojnih okvira kao što su React Native i Ionic, desktop aplikacija koristeći Electron, te čak i u područjima kao što su razvoj igara. Njegova svestranost, podržana bogatim ekosustavom biblioteka i alata, čini ga jednim od najvažnijih programskih jezika današnjice.

Prema [17], JavaScript se odlikuje brojnim ključnim značajkama koje pridonose njegovoj širokoj uporabi i svestranosti, a neke od njih su opisane u nastavku:

- Kao interpretirani jezik, JavaScript kod se izvršava na jednoj niti, liniju po liniju, što uklanja potrebu za prethodnom kompilacijom i time pojednostavljuje razvojni proces te omogućuje brze testove i promjene koda.
- Varijable u JavaScriptu ne zahtijevaju deklariranje tipa te mogu sadržavati bilo koju vrstu podataka u različitim trenucima. Takva fleksibilnost omogućuje programerima da pišu

sažetiji i prilagodljiviji kod, no istovremeno zahtijeva pažljivo rukovanje kako bi se izbjegle pogreške vezane uz tipove podataka.

- JavaScript podržava različite programske paradigme poput proceduralnog programiranja, objektno-orijentiranog programiranja, funkcionalnog programiranja i programiranja vođenog događajima.
- Iako je evoluirao u jezik s mnogo različitih upotreba i dostupnih paradigmi, JavaScript je jezik inherentno vođen događajima, što ga čini posebno pogodnim za izradu interaktivnih web aplikacija, gdje korisničko iskustvo dinamično reagira na unose (poput klikova i pritisaka tipki).
- Izvrsno upravlja asinkronim programiranjem putem mehanizama kao što su povratni pozivi (engl. *callbacks*), obećanja (engl. *promises*) i te modernom *async/await* sintaksom, čime osigurava da aplikacije ostanu brze i učinkovite za krajnje korisnike.

- **Varijable**

Varijable se u JavaScriptu deklariraju koristeći ključne riječi *var, let* ili *const*, a podržani su različiti tipovi podataka kao što su brojevi, stringovi, boolean podaci, objekti, nizovi, funkcije, *null* i *undefined* vrijednosti. Varijable ne zahtijevaju deklariranje tipa podatka te se podatak jednog tipa može prepisati drugim.

- **Funkcije**

Funkcije u JavaScriptu se tretiraju kao objekti, što znači da se mogu dodijeliti varijablama, proslijediti kao argumenti drugim funkcijama te biti vraćene kao vrijednosti iz drugih funkcija. Takva fleksibilnost omogućuje implementaciju ranije spomenutih, raznovrsnih programskih paradigmi te mogućnost ponovne uporabe koda.

- **Objekti**

JavaScript objekti su fundamentalni elementi jezika koji omogućuju organizaciju i manipulaciju podacima putem kolekcija parova ključ-vrijednost. Svaki objekt sastoji se od svojstava (engl. *properties*), gdje svaki par ključa i vrijednosti definira jedno svojstvo objekta. Ključ može biti bilo koji string, dok vrijednost može biti bilo koji JavaScript tip podatka, uključujući druge objekte, nizove ili funkcije. Na slici 3.2. prikazan je primjer takvog objekta koji sadrži podatke o korisniku.

Linija ***Kod***

```
1:      const user = {
2:         firstname: 'John',
3:         lastname: 'Smith',
4:         address: {
5:            street: '6th Street',
6:            number: 128,
7:            city: 'New York',
8:         },
9:         hobbies: ['reading', 'traveling'],
10:        greet: function(){ console.log('Hello!'); },
11:     };
```

Sl. 3.2. Primjer JavaScript objekta

- **Nizovi**

JavaScript nizovi su strukture za organiziranje uređenih lista vrijednosti, indeksirani počevši od 0. Moguće je dodavati, uklanjati i mijenjati elemente te iterirati kroz niz korištenjem različitih metoda kao što su *push(), pop(), shift(), unshift(), slice(), splice(), forEach(), map(), filter()* i *reduce().* Nizovi također podržavaju višedimenzionalne strukture za organiziranje kompleksnih podataka.

- **Kontrolne strukture**

Kontrolne strukture u JavaScriptu vrlo su slične strukturama u ostalim programskim jezicima. Blokovi *if-else* i *switch*, te *for, do-while* i *while* petlje pružaju standardne mehanizme za upravljanje tijekom izvršavanja programa, bilo da je riječ o donošenju odluka na temelju uvjeta ili iteriranju kroz podatke.

- **Rukovanje pogreškama**

Pogreškama se u JavaScriptu rukuje pomoću *try-catch* blokova, koji osiguravaju da se neočekivani problemi mogu jednostavno obraditi bez uzrokovanja pada cijelog programa. Time poboljšavaju robusnost i pouzdanost JavaScript aplikacija. Na slici 3.3. prikazan je primjer *try-catch* bloka koji će, ako dođe do pogreške pri izvršenju metode, ispisati pogrešku na ekran umjesto srušiti program.

Linija ***Kod***

```
1:      try {
2:         const result = riskyFunction();
3:      } catch (error) {
4:         console.log(error);
5:      };
```

Sl. 3.3. Primjer *try-catch* bloka

S obzirom na činjenicu da je JavaScript iznimno decentraliziran programski jezik, odnosno da njegova implementacija i podrška ne ovise o jednoj organizaciji ili entitetu, već o širokoj zajednici developera i organizacija, te da svaki web pretraživač implementira vlastite pristupe pri pokretanju JavaScript koda, njegova službena dokumentacija ne postoji. Međutim, zajednica JavaScript developera pod okriljem organizacije [18] ECMA International periodično objavljuje standardiziranu specifikaciju ECMAScript, koja definira sintaksu, semantiku i ključne značajke jezika. Takav decentraliziran pristup upravljanja otvorenim kodom omogućava JavaScriptu da bude dinamičan i prilagodljiv jezik, koji se kontinuirano razvija i prilagođava različitim potrebama i tehnološkim promjenama.

3.3. TypeScript

JavaScript je počeo kao jednostavan skriptni jezik za web preglednike, ali je s vremenom postao ključan za razvoj interaktivnih sustava. Optimizacije preglednika i pojavljivanje alata poput Angulara i Reacta omogućili su razvoj složenih aplikacija s tisućama linija koda. Danas se JavaScript koristi ne samo za web, već i za poslužiteljske, desktop i mobilne aplikacije, postajući univerzalan alat u programskom razvoju [19].

Svaki programski jezik karakteriziraju specifične osobitosti, a skromne početne premise na kojima je zasnovan JavaScript kasnije su rezultirale mnoštvom takvih značajki koje su znale stvarati probleme.

Na primjeru JavaScript koda sa slike 3.4., ističe se jedna velika pogreška – pri korištenju *console.log()* metode, ime atributa *lastname* objekta *user* nije ispravno napisano. Većina programskih jezika, odnosno većina programa u kojima bi se pisao kod za, primjerice, C# ili Python, istakla bi takvu grešku i prije pokretanja koda, a neki bi ju otkrili tijekom kompilacije, odnosno prije nego što se ijedna linija koda izvrši. JavaScript ju, pak, prepoznaje tak kada se ta linija koda izvrši, što nužno znači potpuni ispad programa zbog pogreške u kodu. Pri pisanju

manjih programa takve greške se lako pronalaze, no u kontekstu aplikacija od tisuća linija koda, slična konstantna iznenađenja su ozbiljan problem.

Linija ***Kod***

```
1:      const user = {
2:         firstname: 'John',
3:         lastname: 'Smith'
4:      };
5:
6:      console.log(user.lstname);
```

Sl. 3.4. Primjer česte greške u JavaScriptu

Određivanje što je greška, a što nije, na temelju vrsta vrijednosti podataka na kojima se provode operacije, poznato je kao statička provjera tipova. TypeScript je dodatak JavaScriptu, odnosno statički provjeravač tipova podataka (engl. *static type-checker*) koji traži greške u kodu prije njegova izvršenja [19]. U ranijem primjeru TypeScript bi programeru naglasio grešku već pri pisanju koda te spriječio kompilaciju. Ukratko, TypeScript je skup pravila za upotrebu različitih tipova podataka, a neka od njegovih najvažnijih pravila objašnjena su u nastavku.

- **Primitivi**

JavaScript ima tri vrlo često korištena primitivna tipa podatka – broj, string i boolean – prikazana na primjeru 3.5. Svaki od njih ima odgovarajući tip u TypeScriptu. Također, ima dvije primitivne vrijednosti koje se koriste za signaliziranje odsutne ili neinicijalizirane vrijednosti: *null* i *undefined*.

Linija ***Kod***

```
1:      const firstname: string = 'John';
2:      const age: number = 32;
3:      const isActive: boolean = true;
```

Sl. 3.5. TypeScript primitivi

- **Nizovi**

Pri navođenju tipova nizova koristi se sintaksa sa slike 3.6., a ona funkcionira za bilo koji tip podatka.

Linija ***Kod***

```
1:      const names: string[] = ['John', 'Mark', 'Luke'];
2:      const years: number = [2011, 2020];
```

Sl. 3.6. TypeScript nizovi

- **Neodređeni tip**

TypeScript ima poseban tip – *any* – koji se koristi kad god se želi izbjeći da određena vrijednost ne uzrokuje greške pri provjeri tipa. Kada je vrijednost tipa *any*, može se pristupiti bilo kojim njezinim svojstvima (koja su isto tipa *any*), pozvati je kao funkciju, dodijeliti joj vrijednosti bilo kojeg tipa, ili napraviti bilo što drugo, što je sintaktički dopušteno. Međutim, korištenje ovog tipa preporuča se izbjegavati jer zapravo poništava svrhu TypeScripta.

- **Funkcije**

Funkcije su primarni način obrade podataka u JavaScriptu. TypeScript omogućuje navođenje tipova ulaznih parametara i izlaznih vrijednosti funkcija. Na slici 3.7. prikazana je sintaksa dodjeljivanja tipa ulaznih parametara i povratne vrijednosti funkcije.

Linija ***Kod***

```
1:      function add(a: number, b: number): number {
2:         return a + b;
3:      }
```

Sl. 3.7. Primjer dodavanja tipova funkciji u TypeScriptu

- **Objekti**

Osim primitivnih tipova, najčešća vrsta tipa u JavaScriptu je tip objekta. To se odnosi na bilo koju JavaScript vrijednost sa svojstvima, što uključuje gotovo sve tipove podataka. Za definiciju tipa objekta, jednostavno se trebaju nabrojati njegovi atributi i odgovarajući tipovi. Primjer definicije objekta za podatke o korisniku koji se prosljeđuje kao parametar funkciji prikazan je na slici 3.8..

Linija ***Kod***

```
1:      function editUser(user: { firstname: string; age: string; … }){ … }
```

Sl. 3.8. Primjer definicije objekta unutar parametara funkcije

- **Unija**

TypeScript omogućuje izgradnju novih tipova podataka iz postojećih, koristeći veliki broj operatora. Jedan od njih je operator unije. Unija tipova je tip koji je formiran od dva ili više drugih tipova i predstavlja vrijednosti koje mogu biti bilo koji od tih tipova. Svaki od tih tipova nazivaju se članovima unije.

- **Alias tipa**

Pri pisanju TypeScript koda, često se događa da se želi koristiti isti tip više puta i referirati se na njega jednim imenom. Alias tipa je upravo to – ime za bilo koji tip. Na primjeru sa slike 3.9. primjer je aliasa tipa za korisnika.

Linija Kod

```
1:     type User = {
2:       firstname: string;
3:       lastname: string;
4:       age: number;
5:     };
6:
7:     const user: User = { firstname: 'John', lastname: 'Smith', age: 32 };
```

Sl. 3.9. Primjer aliasa tipa za objekte koji predstavljaju korisnika

- **Sučelja**

Deklaracija sučelja je još jedan način da se imenuje tip objekta. Korištenje aliasa tipa i deklaracije sučelja je gotovo jednako, osim činjenice da se deklaracija sučelja može proširiti (engl. *extend*).

- **Enumi**

Enumi (engl. *enums*) su jedna od rijetkih značajki koje nisu direktno proširenje tipova JavaScripta. Oni omogućavaju programeru da definira skup imenovanih konstanti. Korištenje enuma može olakšati dokumentiranje namjere ili stvaranje skupa različitih slučajeva. TypeScript pruža i numeričke i na stringovima bazirane enume. Na slici 3.10. prikazan je primjer enuma koji predstavlja slučajeve stanja semafora.

Linija Kod

```
1:     enum TrafficLight {
2:        RED,
3:        YELLOW,
4:        GREEN
5:     };
6:
7:     console.log(TrafficLight.RED); // 'RED'
```

Sl. 3.10. Primjer enuma za semafor

Osim nabrojanih osnovnih značajki, TypeScript nudi i gomilu drugih koncepata provjere tipova podataka koji se mogu poistovjetiti s popularnim, strogo tipiziranim jezicima, a neki od njih su: generički tipovi, dekorateri, *async/await* tipovi, *tuple* tipovi, prostor imena (engl. *namespace*) i ostali.

TypeScript poboljšava čitljivost i održivost koda, olakšava refaktoriranje i pruža bolju podršku u alatima za autokompletiranje i navigaciju po kodu. Sve spomenuto doprinosi bržem razvoju aplikacija, smanjenju broja grešaka i lakšem upravljanju složenim projektima.

3.4. Node.js

Node.js, ili samo Node, je multiplatformsko okruženje otvorenog koda za izvršavanje JavaScript koda izvan web preglednika. Razvio ga je 2009. Ryan Dahl s ciljem izgradnje skalabilnih poslužiteljskih aplikacija. Za izvršavanje koda Node koristi V8 JavaScript „motor“ (engl. *engine*), isti koji koristi Google Chrome [20].

Jedna od ključnih karakteristika Nodea je njegova sposobnost da pokreće aplikacije unutar jednog procesa, bez potrebe za stvaranjem novih niti za svaki pojedinačni zahtjev. Ovo je moguće zahvaljujući asinkronim neblokirajućim operacijama koje su ugrađeni u biblioteku.

Kada Node izvršava ulazno-izlaznu operaciju, poput učitavanja podataka s poslužitelja, pristupa bazi podataka ili datotečnom sustavu, umjesto da blokira nit i troši procesorske cikluse čekajući, Node nastavlja s operacijom kada odgovor stigne. Ovo Nodeu omogućava upravljanje tisućama istovremenih veza s jednim poslužiteljem bez da razmišlja o konkurentnošću niti, što može biti veliki izvor grešaka. Većina biblioteka u Node ekosustavu razvijena je koristeći takve neblokirajuće paradigme [20].

Velika prednost Nodea leži i u tome što milijuni developera klijentskih aplikacija, koji pišu JavaScript za web preglednike, mogu pisati i poslužiteljski kod bez potrebe za učenjem potpuno

novog jezika. Također, u Nodeu novi ECMAScript standardi mogu se koristiti bez problema jer programeri ne moraju čekati da svi korisnici ažuriraju svoje preglednike – oni sami odlučuju koju verziju ECMAScripta koristiti promjenom verzije Nodea.

Takva fleksibilnost, zajedno s jednonitnom arhitekturom, asinkronim operacijama i bogatim ekosustavom, čini Node izuzetno moćnim alatom za razvoj brzih, skalabilnih i efikasnih aplikacija, od web i mobilnih aplikacija do mikroservisa i alata za automatizaciju. U nastavku poglavlja objašnjene su neke od spomenutih značajki [20].

- **Asinkrone i događajima vođene operacije**

Asinkrone i događajima vođene ulazno-izlazne operacije su srž Nodea koji mu omogućava da se ističe u performansama i efikasnosti. U ovom modelu, operacije poput čitanja datoteka ili upita prema bazi podataka pokreću se bez blokiranja izvršavanja ostatka koda. To znači da se aplikacija može nastaviti s drugim zadacima dok čeka da se operacija završi. Kada operacija bude gotova, rezultat se obrađuje kroz prethodno definirane funkcije, poznate kao povratni pozivi, ili korištenjem modernijih pristupa poput obećanja te *async/await* sintakse. Ovaj pristup omogućava Node aplikacijama da efikasno upravljaju velikim brojem istovremenih veza, čineći ih izuzetno skalabilnima i brzima.

Događajima vođeni dio Nodea odnosi se na način na koji Node koristi događaje i njihove slušatelje za obradu asinkronih rezultata, omogućavajući aplikaciji da bude u stalnom stanju čekanja, spremna za reakciju na početak ili završetak ulazno-izlaznih operacija. Ovaj model povećava propusnost i smanjuje potrošnju resursa.

- **Sustav datoteka**

Manipulacija datotekama u Nodeu omogućena je kroz modul *fs* (engl. *file system*), koji pruža API za rad s datotekama i direktorijima na poslužitelju. Ovaj modul omogućava čitanje, pisanje, brisanje datoteka, kao i kreiranje i brisanje direktorija, promjenu dozvola datoteka i direktorija te mnoge druge operacije. Node podržava sinkroni i asinkroni pristup manipulaciji datotekama, omogućavajući programerima da biraju pristup koji najbolje odgovara njihovim potrebama.

Na slici 3.11. prikazan je primjer preporučenog asinkronog korištenja *fs* modula za čitanje tekstualne datoteke s poslužitelja.

Linija Kod

```
1:     async function readFile() {
2:       try {
3:         const data = await fs.readFile('file.txt');
4:         console.log(data);
5:       } catch (err) {
6:         console.log(err);
7:       }
8:     }
```

Sl. 3.11. Primjer korištenja *fs* modula unutar *async*/*await* sintakse

Sinkroni pristup, s druge strane, blokira izvršavanje daljnjeg koda dok se operacija nad datotekom ne završi. Iako je ovaj pristup jednostavniji za razumijevanje i korištenje, može pogoršati performanse aplikacije, posebno u scenarijima s velikim brojem operacija.

Node također omogućava rad s tokovima podataka (engl. *data streams*), koji su posebno korisni za obradu velikih skupova podataka bez potrebe za učitavanjem cijelog skupa u memoriju.

- **Node upravitelj paketa**

NPM, skraćenica za Node upravitelj paketa (engl. *Node Package Manager*), predstavlja temeljni alat ekosustava Node.js-a koji služi dvostrukoj svrsi: kao centralizirani repozitorij za dijeljenje i distribuciju biblioteka i paketa te kao linija naredbi za upravljanje tim bibliotekama i paketima unutar JavaScript projekata.

Kroz NPM, programeri mogu lako pristupiti i koristiti tisuće paketa koje su drugi razvili, što omogućava brži razvoj aplikacija uz manje ponavljanja koda. Kada programer želi dodati funkcionalnost u svoj projekt, može koristiti NPM da pretraži i instalira pakete koji zadovoljavaju te potrebe, čime se izbjegava potreba za ručnim pisanjem kompleksnog koda. Instalacija paketa putem NPM-a automatski rješava ovisnosti paketa, osiguravajući da sve potrebne biblioteke budu prisutne za pravilan rad instaliranog paketa.

Osim instalacije, NPM omogućava i verzioniranje paketa, što znači da programeri mogu odrediti specifične verzije paketa koje žele koristiti, čime se osigurava kompatibilnost i stabilnost aplikacije.

Svaki Node projekt obično sadrži *package.json* datoteku koja služi kao manifest projekta, navodeći sve pakete o kojima projekt ovisi. Ova datoteka, također, omogućava jednostavno dijeljenje projekata s drugima, jer drugi programeri mogu instalirati sve potrebne ovisnosti jednostavnom NPM naredbom *npm install*.

NPM također omogućava upravljanje Node projektima kroz definiranje skripti unutar *package.json* datoteke. Ove skripte mogu obuhvatiti različite zadatke, uključujući pokretanje poslužitelja, testiranje, ili bilo koju drugu automatizaciju potrebnu za razvoj ili produkciju aplikacije.

3.5. NestJS

U posljednjih nekoliko godina, zahvaljujući Nodeu, JavaScript je postao "lingua franca" za aplikacije i na klijentskoj i na poslužiteljskoj strani weba. To je omogućilo ogroman rast razvojnih okvira poput Reacta, Angulara i Vuea, stvorenih za pisanje brzih i proširivih aplikacija za klijentsku stranu. Slična revolucija dogodila se i na poslužiteljskoj strani rastom popularnosti pomoćnih biblioteka i alata za Node poput Express.js-a, no sve do pojave razvojnog okvira NestJS, nijedan od njih nije uspio konkretno riješiti najveći problem poslužiteljskih aplikacija - arhitekturu.

NestJS je prva, i danas najpopularnija, nadogradnja za Node koja nudi gotovo rješenje za arhitekturu poslužiteljskih aplikacija te pri tome omogućava programerima da stvaraju visoko skalabilne, modularne i lako održive aplikacije [21]. Arhitektura NestJS je snažno inspirirana Angularom, no kako je bazirana na objektno-orijentiranim načelima, vrlo je slična i arhitekturama razvojnih okvira za izradu poslužiteljskih aplikacija u drugim programskim jezicima – poput one .NET okvira za C#, te Spring Boot okvira za Javu. Ovaj pristup dozvoljava programerima da koriste poznate koncepte i obrasce dizajna koji su primjenjivi na bilo koje objektno-orijentirano okruženje, što olakšava razvoj složenih aplikacija i promiče dobre prakse programiranja.

Pojavom NestJS-a, JavaScript programeri dobili su mogućnost razvijati poslužiteljske aplikacije bez da sami smišljaju i implementiraju arhitekture aplikacija. U nastavku poglavlja dan je pregled najbitnijih značajki prema [21], koje omogućuju spomenuto.

- **Dekorateri**

Dekorateri u NestJS-u su posebne funkcije koje se koriste za dodavanje metapodataka klasama, metodama, svojstvima ili parametrima. Oni su ključni dio sintakse TypeScripta, koji NestJS intenzivno koristi za dodavanje funkcionalnosti i strukture aplikacijama. Dekorateri omogućavaju programerima da na deklarativan način specificiraju ponašanje dijelova svoje aplikacije, kao što su, u nastavku objašnjeni, kontroleri, servisi, moduli i ostalo. Na slici 3.12. prikazan je primjer korištenja dekoratera za klasu tipa kontrolera koji upravlja zahtjevima i odgovorima vezanih za korisnike (engl. *Users*).

Linija Kod

```
1:     @Controller('users')
2:     export class UsersController { … }
```

Sl. 3.12. Primjer korištenja *@Controller()* dekoratera

- **Upravljači zahtjevima**

Upravljači zahtjevima (engl. *controllers*), ili skraćeno kontroleri, u NestJS-u su klase koje upravljaju dolaznim zahtjevima klijenta, pozivaju servise koji se bave obrađivanjem tih zahtjeva, tj. poslovnom logikom aplikacije, te vraćaju odgovore klijentu.

U NestJS-u, kontroleri se definiraju dekoraterom *@Controller()*, koji može primiti putanju kao argument. Ova putanja određuje osnovni URL na koji kontroler reagira. Unutar kontrolera, metode se zatim mapiraju na specifične HTTP zahtjeve korištenjem dekoratera poput *@Get(), @Post(), @Put(), @Delete()*, itd. Pri tome, svaka od tih metoda izvršava poslovnu logiku putem pomoćnih servisa i zatim vraćati odgovore.

Kontroleri često trebaju pristup detaljima klijentskog zahtjeva. NestJS omogućava pristup objektu zahtjeva korištenjem dekoratera *@Req()* za injektiranje tog objekta u argumente metode kontrolera. Objekt zahtjeva predstavlja HTTP zahtjev i sadrži parametre zahtjeva, HTTP zaglavlja i tijelo zahtjeva. U većini slučajeva nije potrebno ručno dohvaćati cijeli objekt zahtjeva jer NestJS nudi i dekoratere poput *@Param(), @Body()* ili *@Query()* koji mogu iz zahtjeva elegantno uzeti samo ono što je metodi kontrolera potrebno. Slično vrijedi i za objekt odgovora koji se može koristiti putem *@Res()* dekoratera. Slika 3.13. prikazuje primjer kontrolera koji sadrži *GET* krajnju točku za dohvaćanje jednog korisnika pomoću njegovog id-ja.

Linija Kod

```
1:     @Controller('users')
2:     export class UsersController {
3:        @Get('/:id')
4:        getUser(@Req() req, @Res() res, @Param('id') id: string){ … }
5:     }
```

Sl. 3.13. Primjer definiranja krajnje točke unutar kontrolera

- **Davatelji usluga**

U NestJS-u, davatelj usluga (engl. *provider*) je bilo koja klasa koja može pružiti određenu funkcionalnost ili vrijednost drugim dijelovima aplikacije te time omogućiti modularnost i ponovnu upotrebu koda.

Davatelji usluga u NestJS-u mogu biti:

- Servisi: Klase koje sadrže poslovnu logiku ili operacije vezane za bazu podataka.
- Repozitoriji: Klase koje upravljaju pristupom bazi podataka, često koristeći ORM alate, objašnjene u nastavku rada.
- Tvornice: Funkcije ili klase koje dinamički stvaraju instance drugih klasa ili vrijednosti.
- Pomoćnici: Klase ili funkcije koje pružaju specifične, ali uske funkcionalnosti, kao što su pomoć pri radu s brojevima, tekstom ili datumima, enkripcijom i sl.

Davatelji usluga se deklariraju unutar modula, što omogućava NestJS aplikaciji da upravlja njihovim životnim ciklusom i poziva ih samo u dijelovima aplikacije gdje su potrebni. Zahvaljujući ovom principu, komponente aplikacije mogu lako pristupiti zajedničkim funkcionalnostima i resursima bez potrebe za čvrstim vezama, a rezultat toga je čist i lako održiv, proširiv kod. Na slici 3.14. prikazan je primjer servisa koji koristi repozitorij za manipulaciju podataka u bazi podataka vezanih za korisnike.

Linija Kod

```
1:    @Injectable()
2:    export class UsersService {
3:       constructor(
4:          @InjectRepository(User) private userRepository: Repository<User>,
5:       ){}
6:
7:       public async findAll(): Promise<User[]> {
8:          return this.userRepository.find();
9:       }
10:   }
```

Sl. 3.14. Primjer davatelja usluga - servisa i repozitorija

- **Moduli**

U NestJS-u, moduli su klase koje koriste dekorater *@Module()* za organizaciju koda u kompaktne, održive dijelove. Moduli su ključni za strukturiranje aplikacija u NestJS-u jer omogućavaju grupiranje srodnih komponenti, kao što su kontroleri, davatelji usluga (servisi, repozitoriji itd.), i drugi moduli, na način koji promiče modularnost, ponovnu upotrebljivost te lakše testiranje.

Svaki modul u NestJS-u može sadržavati:

1. Kontrolere
2. Davatelje usluga
3. Uvoz modula - lista modula koja omogućava trenutnom modulu da koristi davatelje usluga iz uvezenih modula.
4. Izvoz modula - lista davatelja usluga trenutnog modula koji su dostupni za uvoz drugim modulima.

Na slici 3.15. primjer je definicije modula za dio poslužitelja koji se bavi korisnicima.

Linija Kod

```
1:     @Module({
2:       imports: [TypeOrmModule.forFeature([UserEntity]), AppConfigModule],
3:       controllers: [UsersController],
4:       providers: [UsersService],
5:       exports: [UsersService],
6:     })
7:     export class UsersModule {}
```

Sl. 3.15. Primjer definicije modula

- **Posrednički sloj**

Posrednički sloj (engl. *middleware*) u NestJS-u je niz funkcija koji se izvršava prije metoda kontrolera unutar ciklusa zahtjev/odgovor. Funkcije tog sloja mogu pristupiti objektu zahtjeva i odgovora, kao i *next* funkciji, koja prosljeđuje kontrolu sljedećoj funkciji u nizu. Posrednički sloj se koristi za različite zadatke kao što su bilježenje zahtjeva i odgovora, validacija zahtjeva, autentikacija i autorizacija, postavljanje zaglavlja odgovora, i slično.

U NestJS-u, posrednički sloj se definira kao klasa ili funkcija. Kao klasa mora implementirati *NestMiddleware* sučelje, koje zahtijeva definiranje use metode. Ova metoda prima *req* (zahtjev), *res* (odgovor), i *next* (funkcija za prelazak na sljedeću funkciju ili kontroler) argumente.

- **Cijevi**

U NestJS-u, cijevi (engl. *pipes*) su klase koje implementiraju *PipeTransform* sučelje i mogu se primijeniti na parametre krajnjih točaka, parametre metoda, ili globalno na cijelu aplikaciju.

Cijevi imaju dva slučaja upotrebe:

1. Transformacija: Promjena formata ili tipa ulaznih podataka. Na primjer, pretvaranje stringa koji predstavlja broj u stvarni numerički tip.
2. Validacija: Provjera da li su ulazni podaci ispravni. Ako podaci nisu valjani, cijev može baciti iznimku koja prekida izvršavanje metode i vraća grešku klijentu.

Slika 3.16. prikazuje primjer korištenja cijevi *ParseIntPipe* koja transformira id korisnika iz stringa u cijeli broj.

Linija Kod

```
1:     @Controller('users')
2:     export class UsersController {
3:        @Get('/:id')
4:        public async getUser(@Param('id', ParseIntPipe) id: number){ … }
5:     }
```

Sl. 3.16. Primjer korištenja cijevi

- **Čuvari**

U NestJS-u, čuvari (engl. *guards*) su klase koje implementiraju *CanActivate* sučelje te se koriste za autorizaciju i kontrolu pristupa, omogućavajući ili onemogućavajući izvršavanje određenih zahtjeva na temelju poslovnih pravila ili uvjeta sigurnosti. Čuvari se izvršavaju nakon posredničkog sloja, ali prije cijevi.

Čuvari se mogu primijeniti na razini metoda unutar kontrolera ili globalno na razini cijele aplikacije. Kada se čuvar primijeni, on analizira dolazni zahtjev i, na temelju logike implementirane unutar metode *canActivate*, odlučuje hoće li nastaviti s obradom zahtjeva ili ga prekinuti.

Na slici 3.17. prikazano je korištenje čuvara koji provjerava je li korisnik koji pristupa krajnjoj točki autoriziran prije nego što se zahtjev obradi. Ako korisnik nije autoriziran, krajnja točka automatski šalje grešku klijentu.

Linija Kod

```
1:     @Controller('users')
2:     export class UsersController {
3:        @Get('/:id')
4:        @UseGuards(AuthGuard)
5:        public async getUser(@Param('id', ParseIntPipe) id: number){ … }
9:     }
```

Sl. 3.17. Primjer korištenja čuvara

- **Konfiguracija okruženja**

Svaka aplikacija može se pokrenuti u različitim okruženjima te ovisno o okruženju aplikacija može imati drugačiju konfiguraciju. Primjerice, ako je riječ o lokalnom razvojnom okruženju, aplikacija se vjerojatno spaja na bazu podataka na lokalnom operacijskom sustavu računala pa, stoga, mora imati definirane podatke za spajanje u tom specifičnom okruženju.

NestJS pristupa konfiguraciji okruženja kroz konfiguracijski modul (engl. *ConfigModule*) i servis (engl. *ConfigService*) čime omogućava aplikacijama da čitaju konfiguracijske postavke iz različitih izvora, kao što su varijable okruženja obično pohranjene u *.env* datotekama, na organiziran i siguran način. Proces konfiguracije ostvaruje se u dva koraka:

1. Konfiguracijski modul koristi se za uvoz i registraciju konfiguracijskog servisa unutar NestJS aplikacije. Može se konfigurirati da automatski učita varijable okruženja iz *.env* datoteka koristeći *ConfigModule.forRoot()* metodu.
2. Konfiguracijski servis koristi se za dohvaćanje konfiguracijskih vrijednosti unutar aplikacije. Može se uključiti u bilo koju komponentu aplikacije (npr., servise, kontrolere) gdje su potrebne konfiguracijske postavke. Servis pruža razne metode, no metoda *get()* za jednostavno dohvaćanje vrijednosti konfiguracije po imenu varijable okruženja koristi se najčešće.

3.6. Relacijske baze podataka

Relacijske baze podataka su sustavi koji strukturiraju podatke u međusobno povezane tablice, omogućavajući efikasnu organizaciju i lako pristupanje informacijama putem SQL-a. Takva organizacija podataka ne samo da olakšava njihovo razumijevanje i obradu, već i osigurava visoku razinu integriteta podataka [22].

SQL (engl. *Structured Query Language*) je standardizirani jezik za upravljanje i manipulaciju podacima u relacijskim bazama podataka. Razvijen je 1970-ih u IBM-u, a kasnije standardiziran pri ANSI institutu (engl. *American National Standards Institute*) te ISO organizaciji (engl. *International Organization for Standardization*). SQL omogućava korisnicima da izvršavaju širok spektar operacija nad bazama podataka, uključujući kreiranje struktura baza podataka, upravljanje podacima, upravljanje transakcijama, kontrolu pristupa podacima i ostalo. Osnovne komponente SQL-a prema [22] opisane su u nastavku:

- Jezik definicije - DDL (engl. *Data Definition Language*): SQL koji se koristi za definiranje, mijenjanje i brisanje struktura baza podataka. Uključuje naredbe kao što su *CREATE* za stvaranje novih tablica ili baza podataka, *ALTER* za mijenjanje strukture postojećih tablica, i *DROP* za brisanje tablica ili baza podataka.
- Jezik manipulacije - DML (engl. *Data Manipulation Language*): Omogućava manipulaciju podacima unutar tablica. To uključuje naredbe kao što su *INSERT* za dodavanje novih redaka u tablicu, *UPDATE* za mijenjanje postojećih podataka, i *DELETE* za brisanje redaka iz tablice.
- Jezik upita - DQL (engl. *Data Query Language*): Koristi se za dohvaćanje podataka iz baza podataka. Najčešće korištena DQL naredba je *SELECT*, koja omogućava korisnicima da dohvate podatke iz jedne ili više tablica.
- Jezik upravljanja - DCL (engl. *Data Control Language*): Uključuje naredbe za kontrolu pristupa podacima i bazi podataka. To uključuje *GRANT* za dodjelu prava pristupa i *REVOKE* za uklanjanje prava pristupa.
- Jezik transakcije - TCL (engl. *Transaction Control Language*): Koristi se za upravljanje transakcijama u bazi podataka. Naredbe kao što su *COMMIT* za potvrdu transakcije, *ROLLBACK* za povratak na stanje prije transakcije, i *SAVEPOINT* za postavljanje točaka za povratak unutar transakcije, omogućavaju kontrolu nad izvršavanjem transakcija.

Najvažnije značajke SQL-a su:

- Deklarativnost – SQL omogućava korisnicima da specificiraju što žele dobiti kao rezultat, bez potrebe da detaljno opisuju kako do tog rezultata doći.
- Standardizacija – Iako postoji mnogo dijalekata SQL-a, osnovni SQL je standardiziran, što omogućava prenosivost upita između različitih sustava za upravljanje bazama podataka.
- Svestranost – SQL se može koristiti za širok spektar operacija, od jednostavnih upita do složenih analiza podataka i manipulacija.

U kontekstu ovoga rada, za stvaranje i upravljanje relacijskom bazom podataka korišten je MySQL, softverska platforma otvorenog koda koja implementira SQL. Ispred alternativa poput PostgreSQL ili Oracle relacijskih sustava te nerelacijskih sustava poput MongoDB i Redis baza podataka, MySQL je izabran jer je besplatan te nudi dobar balans između performansi i jednostavnosti korištenja što je prikladno za obujmom malene projekte.

3.7. TypeORM

Prema [23], objektno-relacijsko mapiranje ili ORM (engl. *Object-Relational Mapping*) je tehnika programiranja koja omogućava omogućava programerima da manipuliraju podacima iz relacijskih baza podataka koristeći objekte u željenom programskom jeziku. ORM biblioteke skrivaju složenost SQL upita i omogućavaju rad s bazom podataka na višoj razini apstrakcije čime pojednostavnjuje razvoj i održavanje koda.

TypeORM je jedna od najpoznatijih JavaScript ORM biblioteka, specifično dizajnirana za rad u TypeScript okruženjima i nudi podršku za većinu relacijskih baza podataka. TypeORM omogućava definiranje i manipulaciju tablicama baze podataka koristeći dekoratere i klase, što omogućava vrlo intuitivan i deklarativan pristup modeliranju podataka te se po tom pristupu savršeno slaže s ranije objašnjenim NestJS razvojnim okvirom.

Za korištenje TypeORM-a unutar NestJS-a potrebno je putem NPM-a instalirati odgovarajući paket te konfigurirati ga unutar korijenskog modula NestJS aplikacije (slika 3.18.), a zatim je moguće proizvoljno definirati entitete i manipulirati podacima u željenoj relacijskoj bazi podataka pomoću repozitorija.

Linija Kod

```
1:     @Module({
2:       imports: [
3:         TypeOrmModule.forRootAsync({
4:          imports: [AppConfigModule],
5:          inject: [AppConfigService],
6:          useFactory: (cfg: AppConfigService) => {
7:            return {
8:              type: 'mysql',
9:              host: cfg.get<string>(ENVIRONMENT_VARIABLES.DB_HOST),
10:             database: cfg.get<string>(ENVIRONMENT_VARIABLES.DB_NAME),
11:             port: cfg.get<string>(ENVIRONMENT_VARIABLES.DB_PORT),
12:             …
13:             entites: [UserEntity, …],
14:           };
15:         },
16:      ],
17:      …
18:    })
19:    export class AppModule {}
```

Sl. 3.18. Konfiguriranje TypeORM-a za korištenje u NestJS aplikaciji

3.8. HTML

HTML (engl. *Hyper Text Markup Language*) je standardizirani jezik koji se koristi za kreiranje i strukturiranje sadržaja na Internetu. Ovaj jezik omogućava web dizajnerima i programerima da definiraju kako web preglednici prikazuju tekst, slike i druge multimedijske elemente. HTML se sastoji od niza oznaka koje okružuju različite dijelove sadržaja i daju upute pregledniku o tome kako ih prikazati. Dok preglednici interpretiraju oznake za prikaz sadržaja, one same nisu vidljive korisnicima [24].

Primjerice, osnovna struktura HTML dokumenta uključuje:

- **<*html*>** oznaku koja označava početak i kraj dokumenta,
- **<*head*>** oznaku unutar koje se nalazi prostor za metapodatke i poveznice,
- **<*body*>** oznaku koja sadrži vidljivi sadržaj stranice.

Unutar <*body*> oznake HTML dokumenta može se nalaziti raznovrstan sadržaj koji je namijenjen prikazu korisnicima. Ovo uključuje tekst, slike, videozapise, tablice, liste, forme i druge elemente koji čine strukturu i sadržaj web stranice. Neki od tih elemenata su:

- Tekstualni elementi:
  - <*h1*> do <*h6*>: Oznake za naslove i podnaslove, gdje <*h1*> predstavlja najvažniji naslov, a <*h6*> najmanje važan.
  - <*p*>: Oznaka za paragraf, koristi se za grupiranje i prikaz tekstualnog sadržaja.
- Multimedijski elementi:
  - <*img*>: Oznaka za dodavanje slika na web stranicu.
  - <*video*> i <*audio*>: Oznake za dodavanje video i audio sadržaja.
  - <*canvas*>: Oznaka koja omogućava dinamičko crtanje putem JavaScripta.
- Strukturni elementi:
  - <*div*>: Oznaka za grupiranje ili stiliziranje većih dijelova sadržaja.
  - <*span*>: Oznaka za grupiranje ili stiliziranje manjih dijelova teksta ili sadržaja unutar drugih elemenata.
- Liste:
  - <*ul*>: Neuređena lista - bez numeracije stavki.
  - <*ol*>: Uređena lista - prikazuje stavke s numeracijom.
  - <*li*>: Oznaka za pojedinačnu stavku unutar liste.

- Linkovi i navigacija:
    - <*a*>: Oznaka za poveznicu, tj. komuniciranje s drugim stranicama ili resursima.
    - <*nav*>: Semantička oznaka koja se koristi za označavanje navigacijskih linkova.
- Forme:
    - <*form*>: Oznaka koja definira formu za unos podataka.
    - <*input*>, <*textarea*>, <*button*>, <*select*>: Elementi koji se koriste unutar formi za unos teksta, odabir opcija, potvrdu forme i ostalo.
- Tablice:
    - <*table*>: Oznaka za kreiranje tablica.
    - <*tr*>, <*td*>, <*th*>: Oznake za definiranje redova, ćelija i zaglavlja unutar tablica.

Nabrojane oznake samo su dio duge liste oznaka koje se mogu naći unutar <*body*> oznake HTML dokumenta. Njihovim korištenjem web dizajneri i programeri mogu kreirati bogate i vizualno privlačne web stranice. Kada se HTML kombinira s JavaScriptom, mogućnosti za interakciju i dinamičnost web stranica znatno se proširuju jer JavaScript omogućava manipulaciju HTML elemenata u realnom vremenu. Primjerice, JavaScript može reagirati na korisničke događaje poput klikova miša ili tipkanja na tipkovnici, mijenjati sadržaj na stranici bez potrebe za ponovnim učitavanjem, te dinamički ažurirati stilove i prikaz elemenata. Sinergija između HTML-a i JavaScripta temelj je modernog web razvoja.

3.9. CSS

CSS (engl. *Cascading Style Sheets*) je alat za definiranje vizualnog izgleda web stranica koji omogućava web dizajnerima i programerima da precizno kontroliraju stilove elemenata na web stranici, uključujući boje, fontove, raspored, razmake i mnoge druge aspekte dizajna. Obično se koristi u kombinaciji s HTML-om, gdje HTML služi za stvaranje strukture sadržaja web stranice, a CSS za definiranje kako taj sadržaj izgleda [24].

CSS pravila se mogu dodati izravno unutar HTML dokumenta <*style*> oznakom unutar <*head*> oznaka, uključiti u dokument kao vanjska datoteka, ili primijeniti unutar oznaka za pojedinačne elemente.

CSS sintaksa se sastoji od selektora (koji određuje na koje HTML oznake se pravilo primjenjuje) i deklaracija (koje definiraju kako su ti elementi stilizirani). Deklaracije se sastoje od svojstava i njihovih vrijednosti, odvojenih dvotočkom.

Primjerice, pravilo *p { color: 'blue'; font-size: 16px }* primjenjuje se na sve <*p*> elemente unutar HTML dokumenta, postavljajući boju teksta na plavu i veličinu fonta na 16 piksela.

Dodatno, CSS podržava napredne funkcije poput pseudo-klasa i pseudo-elemenata, koji omogućavaju detaljno stiliziranje specifičnih stanja HTML elemenata, dok se sustavi CSS pravila poput Flexbox i Grid sustava omogućavaju stvaranje složenih sučelja osjetljivih na širinu i visinu ekrana zaslona na kojemu korisnik pregledava web stranicu.

Upotrebom CSS-a, programeri imaju mogućnost uspostaviti jedinstven i privlačan vizualni identitet web stranica, čime se znatno unapređuje iskustvo korisnika. CSS, stoga, predstavlja temelj suvremenog web dizajna.

3.10. React

React je JavaScript biblioteka otvorenog koda za izgradnju korisničkih sučelja. Stvorio ju je Facebook (sada Meta) i održava je zajedno s širokom zajednicom programera. Ova biblioteka omogućuje programerima da stvaraju kompleksna interaktivna korisnička sučelja iz malih, izoliranih, ponovo upotrebivih dijelova koda kojeg nazivaju komponentama. Zbog svoje jednostavnosti, performansi, skalabilnosti te sposobnosti da efikasno kombinira nekoliko ključnih funkcionalnosti pri prikazivanju sadržaja stranice, React je stekao ogromnu popularnost. U nastavku, objašnjene su neke od tih funkcionalnosti [25].

- **JSX**

JSX je sintaksna ekstenzija za JavaScript koju React preporučuje za pisanje HTML-a unutar JavaScripta i koristi se za opisivanje strukture korisničkih sučelja. JSX omogućuje pisanje koda koji kombinira logiku s dizajnom aplikacije unutar istih komponenti. Na slici 3.19. je jednostavan primjer korištenja JSX-a unutar React funkcionalne komponente.

Linija Kod

```
1:     function Greeting(){
2:        const name = 'John';
3:
4:        return <h1>Hello, {name}!</h1>;
5:     }
```

Sl. 3.19. Primjer korištenja JSX-a

- **Funkcionalne komponente**

Komponente u Reactu omogućavaju razdvajanje koda na modularne cjeline čime olakšavaju upravljanje i ponovnu upotrebu koda u razvoju aplikacija. Funkcionalne komponente su

jednostavne JavaScript funkcije koje prihvaćaju parametre (engl. *props*), a kao povratnu vrijednost vraćaju React elemente koji grade strukturu aplikacije (slika. 3.20.). React kuke (engl. *hooks*) objašnjenih u nastavku poglavlja, kontroliraju stanje, životni ciklus i ostale mogućnosti funkcionalnih komponenti koje nudi React.

Linija Kod

```
1:     function Greeting(props: { name: string; }){
2:        return <h1>Hello, {props.name}!</h1>;
3:     }
```

Sl. 3.20. Primjer jednostavne React funkcionalne komponente

- **React Hooks**

Prije kuka, funkcionalne komponente bile su iznimno ograničene jer nisu mogle koristiti lokalno ili globalno stanje aplikacije, ili pristupiti životnom ciklusu komponente. Te funkcionalnosti bile su rezervirane samo za klasne komponente. Kuke su funkcionalnim komponentama riješile taj problem, pružajući jednostavniji i moćniji način za izgradnju kompleksnih komponenti i time u modernim React aplikacijama potpuno eliminirali potrebu za klasnim komponentama.

- *useState()*: Osnovna je kuka koja omogućava dodavanje stanja u funkcionalne komponente. Svaki poziv *useState()* kuke vraća par: trenutnu vrijednost stanja i funkciju koja ažurira to stanje. Ova kuka najčešće omogućava komponentama ažuriranje trenutnog stanja nakon interakcija poput klika gumba (slika. 3.21.).

Linija Kod

```
1:     function Counter(){
2:        const [count, setCount] = useState<number>(0);
3:
4:        return (
5:          <div>
6:           <h1>Counter: {count}</h1>
7:           <button onClick={() => setCount(count + 1)}>Increment</button>
8:          </div>
9:        );
10:    }
```

Sl. 3.21. Primjer upotrebe *useState()* kuke

- *useEffect()*: Simulira ponašanje metoda životnog ciklusa klasnih komponenti koje su se izvodile u trenutcima kada bi se komponenta prvi puta prikazala na ekranu ili kada

bi se u njoj promijenilo lokalno stanje (slika 3.22.). Omogućuje komponenti da reagira na događaje svaki puta kada se neki od parametara s liste osjetljivosti promjeni.

Linija Kod

```
1:     function TestComponent(){
2:       useEffect(() => {
3:         console.log('Component has mounted!');
4:       }, []);
5:
6:       return <> … </>;
7:     }
```

Sl. 3.22. Primjer *useEffect()* kuke

- *useRef()*: Omogućuje pohranjivanje i pristupanje podacima unutar komponente bez utjecaja na proces prikazivanja te komponente. Drugim riječima, promjene vrijednosti spremljene pomoću *useRef()* kuke neće pokrenuti osvježavanje komponente.
- *useMemo()* i *useCallback()*: Optimiziraju performanse komponenti sprečavajući nepotrebno osvježavanje, odnosno govore React prevoditelju da dio koda unutar navedenih kuka ne mora ponovno obraditi. *useMemo()* se koristi za pamćenje povratnih vrijednosti proizašlih iz skupih izračuna, a *useCallback()* za pamćenje definicija funkcija.

- **Kompozicija komponenti**

Kompozicija u Reactu omogućuje stvaranje novih komponenti integracijom već postojećih, izbjegavajući potrebu za nasljeđivanjem. Komponente mogu uključivati druge komponente kao svoje djecu ili ih proslijediti kao parametre čime omogućavaju laku i fleksibilnu ponovnu upotrebu koda. Nasuprot tomu, nasljeđivanje podrazumijeva kreiranje novih komponenti proširenjem funkcionalnosti onih već postojećih, što može rezultirati kompleksnim i teško održivim hijerarhijama koje React nastoji izbjeći. Na slici 3.23. prikazan je jednostavan primjer kompozicije komponenti.

Linija Kod

```
1:     function Header() { return <header>Header</header>; }
2:
3:     function MainContent() { return <main>Main Content</main>; }
4:
5:     function App() {
6:           return (
7:             <div>
8:               <Header />
9:               <MainContent />
10:            </div>
11:          );
12:    }
```

Sl. 3.23. Primjer kompozicije React komponenti

- **Jednosmjerni protok podataka**

Koncept jednosmjernog protoka podataka u Reactu osigurava da se informacije unutar aplikacije prenose linearno, od roditeljskih komponenata prema dječjim čime se podrazumijeva da stanje aplikacije i logika upravljanja tim stanjem uglavnom nalaze na višim razinama hijerarhije komponenata. Kada roditeljska komponenta posjeduje određeno stanje, ona to stanje može jednostavno proslijediti svojim dječjim komponentama putem parametara (engl. *props*). Ovaj pristup savršeno se slaže s konceptom kompozicije komponenti te pojednostavljuje strukturu i upravljanje aplikacijom olakšavajući praćenje promjena u podacima i otklanjanje potencijalnih grešaka.

- **React biblioteke**

React biblioteke su vanjski paketi ili moduli razvijeni od strane React zajednice ili trećih strana, koji se mogu integrirati u React aplikacije kako bi se proširile njihove funkcionalnosti ili olakšao razvoj. Ove biblioteke obično rješavaju specifične probleme ili nude gotove komponente koje se mogu ponovno koristiti u različitim projektima. U nastavku objašnjeno je nekoliko kategorija takvih biblioteka.

  - Biblioteke koje upravljaju stanjem aplikacije: Ove biblioteke pružaju alate i mehanizme za efikasno upravljanje i dijeljenje stanja unutar aplikacije. Omogućavaju centralizirano ili decentralizirano upravljanje stanjem, olakšavajući komunikaciju i dijeljenje podataka između komponenata. Primjeri takvih biblioteka su: Redux, Zustand i Jotai.

- Biblioteke za asinkronu komunikaciju i upravljanje podacima: Biblioteke u ovoj kategoriji fokusirane su na upravljanje asinkronim operacijama poput dohvaćanja podataka s poslužitelja, spremanja podataka u predmemoriju i sinkronizacije stanja. Ukratko, olakšavaju rad s udaljenim podacima, odnosno integraciju s vanjskim API-jima. Najpoznatiji primjeri ovih kategorije su: TanStack (React) Query i React SWR.
- Biblioteke gotovih komponenti: Ove biblioteke nude predizrađene komponente i dizajnerske sustave koji se mogu koristiti za brz razvoj korisničkih sučelja. Uključuju komponente poput gumba, komponente za unos podataka, tablice, navigacijske elemente, komponente za prikazivanje teksta i mnoge druge, često prateći određene dizajnerske smjernice poput pažljivo izrađenih sustava boja, debljina linija i širine razmaka te veličina fonta i ikona. Predstavnici ovih biblioteka su: MaterialUI, ChakraUI, React Bootstrap te AntDesign.

Osim prethodno navedenih kategorija, postoji još nekoliko važnih kategorija React biblioteka koje se često koriste u razvoju klijentskih aplikacija poput onih za: navigaciju, forme, animacije, lokalizaciju, testiranje te optimizaciju performansi.

Svaka od ovih kategorija ima za cilj pojednostavniti i ubrzati razvoj aplikacija, omogućavajući programerima da se usredotoče na specifične poslovne aspekte aplikacije umjesto na osnovnu implementaciju ili ponovno smišljanje već dostupnih rješenja.

- **React razvojni okviri**

Osim tradicionalne upotrebe u izradi korisničkih sučelja na webu, React se također koristi u razvojnom okviru pod nazivom Next.js. Next je postao popularan među React programerima zbog svoje implementacije "prikazivanja" na poslužiteljskoj strani (engl. *server-side rendering* - SSR). SSR omogućava aplikacijama da generiraju HTML na poslužitelju prije nego što se stranica pošalje korisnikovom pregledniku. Ova značajka ima ključnu ulogu u poboljšanju performansi aplikacije, smanjenju vremena učitavanja stranica i optimizaciji za tražilice (engl. *search engine optimization* - SEO), jer tražilice lakše indeksiraju sadržaj koji je već prethodno "prikazan" na poslužitelju.

Osim Nexta, u posljednjih nekoliko godina pojavili su se još desetci razvojnih okvira koja pokušavaju, svaki na svoj način, proširiti osnovne funkcionalnosti Reacta. Neki od njih su Gatsby, Remix, i Astro te su razvijeni kako bi riješili specifične potrebe. Takva raznolikost omogućava programerima da odaberu alat koji najbolje odgovara ciljevima njihovog projekta, čineći ekosustav

Reacta najpopularnijom, najbogatijom i najfleksibilnijom tehnologijom za razvoj klijentskih strana aplikacija.

3.11. React Native

React Native je JavaScript razvojni okvir za pisanje mobilnih aplikacija koje se izvorno prikazuju na iOS i Android operacijskim sustavima. Temelji se na prethodno objašnjenom Reactu, ali umjesto prikazivanja aplikacija na web preglednicima, React Native cilja na mobilne platforme. Drugim riječima, web programeri upoznati s Reactom, putem React Nativea mogu pisati nativne mobilne aplikacije kroz "udobnost" JavaScript biblioteke koju već poznaju [26].

Slično kao React za web, developeri React Native pišu koristeći JSX, no u pozadini React Native prevoditelj poziva izvorne API-je za prikazivanje mobilnih komponenata u Objective-C-u (za iOS) ili Javi (za Android). Time aplikacije koriste stvarno mobilno korisničko sučelje, a ne *webview*. Uz to, React Native implementira JavaScript API-je za hardverske komponente mobilnih uređaja poput kamere, bluetooth uređaja, WiFi ili lokacijskih usluga i ostalih.

- **Nativne komponente**

U komponentama Reacta za web prikazuju se HTML elementi. U React Nativeu, svi ti elementi zamjenjuju se React komponentama prilagođenim za iOS i Android. Najosnovnija je univerzalna komponenta *View*, jednostavan i fleksibilan element za grupiranje sadržaja analogan HTML-ovom *div* elementu. Na iOS-u komponenta *View* prikazuje *UIView* jezika Objective-C, dok na Androidu prikazuje u Javin *View*.

Osim komponente *View*, React Native nudi i druge komponente prilagođene mobilnim platformama. Na primjer, komponenta *Text* se koristi za prikazivanje teksta, slično kao što bi se u web razvoju koristio element *p*. Komponenta *Image* omogućava prikazivanje slika, a *Button* se koristi za stvaranje interaktivnih gumba. Sve ove komponente prilagođene su da iskoriste izvorne mogućnosti mobilnih platformi.

U ranim danima React Nativea, jedan od izazova bio je nedostatak apstrakcije za neke komponente koje su specifične za iOS ili Android. Na primjer, komponenta *DatePickerIOS* bila je dostupna samo za iOS, pružajući korisničko sučelje za odabir datuma, dok je za Android postojala izgledom slična, ali logički različita komponenta *DatePickerAndroid*. Ova razlika zahtijevala je od programera da pišu uvjetni kod kako bi se osiguralo da se prava komponenta koristi na odgovarajućoj platformi, što je kompliciralo razvoj i održavanje koda.

Tijekom vremena, zajednica i razvojni tim React Nativea radili su na poboljšanju takvih problema kroz uvođenje univerzalnih komponenti koje rade na oba sustava, kao i bolje apstrakcije za postojeće platformski specifične komponente. Danas, zbog sve veće popularnosti React Nativea, situacija kao s komponentom za odabir datuma gotovo uopće nema. Gotovo da ne postoji značajka u programiranju mobilnih aplikacija koju zajednica nije prilagodila i apstrahirala za React Native.

- **Stiliziranje nativnih komponenti**

Stiliziranje u React Nativeu temelji se na pristupu koji je prilagođen specifičnostima mobilnih aplikacija, razlikujući se od tradicionalnog pristupa stiliziranju sadržaja na webu s Reactom i CSS-om. Ovdje, stilovi se definiraju korištenjem JavaScript objekata, što omogućava integraciju stilova direktno s logikom aplikacije.

React Native nudi *StyleSheet* API, koji služi kao apstrakcija za definiranje i manipulaciju stilovima (slika 3.24.). Korištenje *StyleSheet.create()* metode ne samo da čini kod čišćim i organiziranijim, već i poboljšava performanse aplikacije tako što stilove obrađuje i provjerava unaprijed. Osim toga, *StyleSheet* omogućava referenciranje stilova po identifikatoru, slično kao i CSS.

Linija Kod

```
1:     function App(){
2:        return (
5:          <View style={styles.container}>
6:           <Text style={styles.title}>Hello, React Native!</Text>
8:          </View>
9:        );
10:    }
11:
12:    const styles = StyleSheet.create({
13:       container: {
14:          alignItems: 'center',
15:       },
16:       title: {
17:          fontSize: 24,
18:       },
19:    });
```

Sl. 3.24. Primjer korištenja StyleSheet API-ja u React Nativeu

Stilovi, također, mogu biti direktno dodijeljeni komponentama kroz *style* parametar. Ovaj pristup se ne preporučuje zbog smanjenja čitljivosti i održivosti koda, no ponekad su nužni jer su korisni za dinamičko stiliziranje koje ovisi o stanju komponente.

Uspoređujući s Reactom za web, gdje se stilovi obično definiraju korištenjem vanjskih CSS datoteka, React Native pristup stiliziranju nudi jedinstvenu integraciju stilova s JavaScriptom, što olakšava razvoj i omogućava veću fleksibilnost u dizajniranju korisničkog sučelja mobilnih aplikacija.

- **API-ji domaćina**

Najveća razlika između Reacta za web i React Nativea leži u načinu na koji se pristupa API-jima domaćina, odnosno mobilnih uređaja. Na webu postoje razlike između arhitektura web preglednika, ali svi najpopularniji moderni web preglednici podržavaju zajedničku osnovu dijeljenih značajki te, stoga, pri razvijanju React aplikacije obično nije potrebno paziti na značajke pojedinog web preglednika. Kod React Nativea, pak, specifični API-ji iOS i Android uređaja igraju mnogo veću ulogu u stvaranju korisničkog iskustva jer uključuju širok opseg različitih funkcionalnosti, od pohrane podataka, lokacijskih usluga, sve do direktnog pristupa hardveru poput kamere, koje mogu varirati od uređaja do uređaja.

React Native olakšava pristup ovim API-jima putem jednostavnih JavaScript sučelja, omogućujući programerima da se usredotoče na stvaranje vlastite poslovne logike umjesto na složenost komunikacije s API-jima uređaja. Osim toga, programerska zajednica oko React Nativea je kroz godine omogućila podršku za veliku većinu iOS i Android API-ja te vrlo brzo reagira na njihova ažuriranja prilagođavajući nove značajke React Nativeu.

- **React Native biblioteke**

Vanjske biblioteke u React Nativeu igraju sličnu ulogu kao i kod Reacta za web, ali su specifično prilagođene za razvoj mobilnih aplikacija. S obzirom na činjenicu da su sve biblioteke React ekosustava napisane u JavaScriptu, mnoge od njih mogu se koristiti unutar oba razvojna okvira, osim onih koje ima specifične uloge. Primjerice, navigacija kroz web i mobilne aplikacije temeljena je na potpuno drugačijim principima pa analogno tome postoje različite biblioteke za navigaciju u Reactu za web i React Nativeu.

4. PROGRAMSKO RJEŠENJE APLIKACIJA

Ovo poglavlje objašnjava najvažnije dijelove razvoja sveobuhvatnog rješenja zadatka rada koje uključuje poslužiteljsku aplikaciju razvijenu u NestJS-u i dvije klijentske aplikacije: React Native mobilnu aplikaciju za krajnje korisnike i React web aplikaciju za tvrtke.

NestJS server aplikacija implementira RESTful arhitekturu i funkcionalnosti kao što su registracija i prijava korisnika i administratora, te osigurava autentikaciju i autorizaciju pri pristupanju ostalim krajnjim točkama koje manipuliraju vozilima, vožnjama, javnim prijevozom i vijestima.

S druge strane, React Native mobilna aplikacija omogućava korisnicima pregled vijesti o tvrtkama i njihovim uslugama, pristup karti grada s označenim vozilima i javnim prijevozom dostupnim za korištenje te pregled podataka o prethodnim vožnjama i vijestima. Ova aplikacija predstavlja ključno korisničko sučelje koje povezuje krajnje korisnike s uslugama koje pružaju tvrtke.

Za administratore javnog prijevoza i tvrtke koje nude usluge, razvijena je React web aplikacija koja služi kao platforma za upravljanje sadržajem koji se prikazuje korisnicima, uključujući upravljanje linijama javnog prijevoza, vozilima i objavljivanjem vijesti.

Kroz kombinaciju ranije objašnjenih tehnologija i pristupa razvoju JavaScript aplikacija, predstavljeno rješenje demonstrira kako moderni softverski alati i prakse mogu biti iskorišteni za poboljšanje pristupa i upravljanja uslugama u dinamičnom i zahtjevnom okruženju kao što je gradski promet.

4.1. Dijeljeni NPM paket

Kako bi se poboljšala kohezija te smanjilo dupliciranje koda unutar tri različite aplikacije ručno je stvoren jednostavan NPM paket kojeg sve tri aplikacije mogu koristiti. Paket sadrži definicije TypeScript tipova entiteta korištenih u realizaciji projekta, definicije objekata za prijenos podataka (engl. *Data Transfer Object* ili DTO) i implementacije pomoćnih funkcija za rad s brojevima, tekstom, koordinatama i ostalim strukturama.

4.1.1. Proces stvaranja paketa

1. Inicijalizacija i konfiguracija:

Projekt se inicijalizira kao NPM paket i konfigurira za upotrebu s TypeScriptom. Zatim se instaliraju bilo koje pomoćne JavaScript biblioteke poput biblioteke Lodash koja omogućava mnoštvo metoda za manipulaciju JavaScript objekata i nizova.

2. Strukturiranje i implementacija projekta:

Projekt se organizira u direktorije koji odražavaju njegov sadržaj: tipove, DTO objekte i pomoćne funkcije, te se unutar odgovarajućih direktorija definiraju navedene strukture.

3. Kompilacija:

TypeScript kod se prevodi u JavaScript naredbom *npm build* i sprema se u odvojeni *dist* direktorij kojemu pristupaju poslužiteljska i klijentske aplikacije.

4.1.2. Definiranje tipova, DTO objekata i pomoćnih funkcija

Definiranjem tipova kao što su korisnik (engl. *User*), tvrtka (engl. *Company*), vozilo (engl. *Vehicle*), vožnja (engl. *Ride*), stanica (engl. *Station*), linija (engl. *Line*) i vijest (engl. *News*) unutar zajedničkog paketa (slika 4.1.), osigurava se da sve aplikacije koriste iste definicije - poslužitelj te tipove koristi za definiranje tablica u bazi podataka, a klijenti za prikazivanje podataka u korisničkom sučelju. To eliminira mogućnost neslaganja tipova duž cijelog projekta, odnosno olakšava razvoj i održavanje koda.

Linija Kod

```
1:     export interface Vehicle {
2:        id: number;
3:        code: string;
4:        type: VehicleType;
5:        active: boolean;
6:        location: Coordinate;
7:        createdAt: Date;
8:        …
9:     }
```

Sl. 4.1. Primjer definiranja i izvoza tipa vozila

DTO objekti važni su za prijenos podataka između klijentskih aplikacija i poslužitelja. Svaki put kad klijent šalje zahtjev za stvaranjem ili uređivanjem, primjerice, vozila, mora poslati poslužitelju željene informacije o tom vozilu u tijelu (engl. *body*) zahtjeva. Kako bi klijent znao što točno u zahtjevu treba poslati, a poslužitelj što u zahtjevu prima, a nakon obrade vratiti kao odgovor, tijela zahtjeva i odgovora definirana su DTO objektima te njihovim centraliziranjem olakšava se njihova ponovna upotreba i osigurava da su sve aplikacije usklađene s očekivanim strukturama podataka. Na slici 4.2. prikazan je primjer definicije tipa DTO objekta koji se koristi pri zahtjevu za ocjenjivanje vožnje.

Linija Kod

```
1:     export interface RateRideDto {
2:       rideId: number;
3:       rating: number;
4:     }
```

Sl. 4.2. Definicija tipa DTO objekta za ocjenjivanje vožnje

Pomoćne funkcije za transformaciju koordinata, obradu brojeva, teksta i sličnih struktura mogu biti korisne u različitim dijelovima aplikacija. Umjesto da se iste funkcije implementiraju više puta, one se mogu održavati u zajedničkom paketu i dijeliti među aplikacijama, što smanjuje duplikaciju koda i olakšava ažuriranja. Na slici 4.3. prikazan je primjer takve pomoćne funkcije koja, u ovom slučaju, transformira objekt tipa *Coordinate* u format kojeg koristi SQL baza podataka.

Linija Kod

```
1:     export interface Coordinate {
2:       latitude: number;
3:       longitude: number;
4:     }
5:
6:     export const convertCoordinateToSqlPoint(coordinate: Coordinate){
7:       return `POINT(${coordinate.latitude} ${coordinate.longitude}`;
8:     }
```

Sl. 4.3. Definicija sučelja Coordinate i implementacije metode za transformaciju objekta tipa Coordinate

4.2. Najvažniji dijelovi NestJS poslužiteljske aplikacije

NestJS poslužiteljska aplikacija središte je rješenja ovog rada u kojem se obrađuju resursi pohranjeni u SQL bazi podataka. Za upravljanjem tim resursima potrebno je konfigurirati povezivanje aplikacije s bazom podataka i njeno pokretanje, definirati SQL tablice te implementirati davatelje usluga i kontrolere, a kako bi resursi bili zaštićeni od neovlaštene upotrebe – autentikaciju te autorizaciju. Navedeni koraci detaljno su opisani u nastavku ovog poglavlja.

4.2.1. Konfiguracija i pokretanje aplikacije

Osim konfiguracijskog modula i konfiguracijskog servisa koji se koriste kroz cijelu aplikaciju, postoje još dvije važne datoteke u kojima se odvija proces konfiguracije NestJS aplikacije, a to su datoteka *main.ts* te korijenski modul aplikacije, odnosno datoteka *app.module.ts.*

Datoteka *main.ts* sadrži funkciju *bootstrap()* koja pokreće aplikaciju (slika 4.4.). Proces započinje stvaranjem instance NestJS aplikacije koristeći korijenski modul (engl. *AppModule*).

Nakon toga, dohvaća se instanca konfiguracijskog servisa (engl. *AppConfigService*) koja omogućava pristup varijablama okruženja. Ove vrijednosti se koriste za postavljanje prefiksa za sve rute krajnjih točaka u aplikaciji.

Na kraju, aplikacija se postavlja tako da osluškuje na portu koji se dobije iz varijabli okruženja, ili se koristi uobičajeni *port* 3000 ako vrijednost nije postavljena i zatim se pokreće.

Linija Kod

```
1:     async function bootstrap(){
2:        const app = await NestFactory.create(AppModule);
3:
4:        const cfg = app.get(AppConfigService);
5:
6:        const apiPrefix = cfg.get(ENVIRONMENT_VARIABLES.API_PREFIX);
7:        app.setGlobalPrefix(apiPrefix);
8:
9:        const port = cfg.get(ENVIRONMENT_VARIBALES.PORT, 3000);
10:       await app.listen(port);
12:    }
```

Sl. 4.4. Implementacija metode bootstrap koja pokreće NestJS poslužiteljsku aplikaciju

Ranije spomenuti korijenski modul u NestJS aplikacijama služi kao glavni modul koji orkestrira strukturom aplikacije. Kroz njega se uvozi konfiguracijski modul, definira pristup TypeORM modula MySQL bazi podataka, registriraju entiteti, odnosno tablice koje baza treba stvoriti te se uvoze svi ostali moduli korišteni u aplikaciji (slika 4.5.).

Linija Kod

```
1:     @Module({
2:       imports: [
3:         AppConfigModule,
4:         TypeOrmModule.forRootAsync({
5:          …
6:          useFactory: (cfg: AppConfigService) => {
7:            return {
8:              …
9:              entities: [UserEntity, VehicleEntity, RideEntity, …],
10:           };
11:         },
12:        }),
13:        UsersModule,
14:        AuthModule,
15:        VehiclesModule,
16:        RidesModule,
17:        …
18:      ],
19:    })
20:    export class AppModule {}
```

Sl. 4.5. Korijenski modul NestJS aplikacije

Nakon što su obavljeni navedeni koraci i implementirani konfiguracijski i korijenski moduli, moguće je pokrenuti aplikaciju NPM naredbom *npm start* kojom je omogućeno dinamičko osvježavanje, odnosno ponovo pokretanje nakon svake promjene u kodu.

4.2.2. Definiranje SQL tablica

Sljedeći korak pri implementaciji poslužiteljske aplikacije je definirati entitete, odnosno logičke sudionike aplikacije poput korisnika, tvrtki, vožnji i ostalih, pomoću TypeORM-a koji omogućuje mapiranje TypeScript klasa u definicije SQL tablica. Tablice implementira MySQL relacijska baza podataka. Na slici 4.6. prikazan je primjer definicije klase korisnika, odnosno entiteta *UserEntity* čiji atributi postaju stupci u MySQL tablici.

Linija Kod

```
1:     @Entity({ name: 'user' })
2:     export class UserEntity implements UserWithLoginDetails {
3:       @PrimaryGeneratedColumn()
4:       id: number;
5:
6:       @Column()
7:       username: string;
8:
9:       @Column({ select: false })
10:      password: string;
11:
12:      @Column()
13:      name: string;
14:
15:      @Column()
16:      email: string;
17:
18:      @CreateDateColumn()
19:      createdAt: Date;
20:      …
21:    }
```

Sl. 4.6. Primjer definicije klase korisnika, odnosno entiteta *UserEntity*

Dekorater *@Entity()* označava klasu kao entitet, odnosno tablicu u bazi podataka. Opcija *{ name: 'user' }* specificira ime tablice.

@PrimaryGeneratedColumn() dekorater označava primarni ključ koji se automatski generira. U ovom slučaju, id korisnika je primarni ključ.

Dekorater *@Column()* se koristi za definiranje običnog stupca u tablici. Ako se koristi bez opcija, podrazumijeva se da je tip stupca određen tipom svojstva u TypeScriptu. Na primjer, *username: string* postaje stupac tipa *VARCHAR*.

Opcija *{ select: false }* koristi se za stupce koji ne bi trebali biti automatski uključeni u rezultate *SELECT* upita. Na primjer, lozinka (engl. *password*) je osjetljiv podatak kojeg se ne želi zabunom vratiti klijentu kao odgovor na bilo koji zahtjev.

@CreateDateColumn() dekorater automatski bilježi datum i vrijeme stvaranja retka u tablici kao stupac *createdAt*. Analogno tome, dostupni su *@UpdateDateColumn()* i *@DeleteDateColumn()* koji mogu bilježiti ažuriranje i brisanje retka iz tablice.

TypeORM nudi mnoštvo drugih dekoratera i opcija pri definiranju tablica u bazi podataka. Neki od najbitnijih tiču se relacija između entiteta, a primjer takvog dekoratera prikazan je na primjeru sa slike 4.7..

Linija Kod

```
1:     @Entity({ name: 'ride' })
2:     export class RideEntity {
3:       …
3:       @Column()
4:       userId: string;
5:
6:       @ManyToOne(() => UserEntity, (user) => user.id)
7:       @JoinColumn({ name: 'userId' })
8:       user: Relation<UserEntity>;
10:    }
```

Sl. 4.7. Definiranje relacije @ManyToOne() i @JoinColumn() dekoraterima

U entitetu vožnje (engl. *RideEntity*), relacija s entitetom korisnika (engl. *UserEntity*) definirana je korištenjem *@ManyToOne()* dekoratera. U praksi, to znači da jedan korisnik može imati više vožnji, ali svaka vožnja pripada samo jednom korisniku.

@JoinColumn({ name: 'userId' }) dekorater s opcijom specificira stupac iz tablice vožnji koji se koristi za povezivanje dva entiteta. U ovom slučaju, stupac *userId* služi kao vanjski ključ koji upućuje na id stupac korisnika.

Nakon implementacije relacije, pri dohvaćanju vožnje, moguće je dohvatiti korisnika koji je "odvozio" tu vožnju. Alternativno, pri dohvaćanju korisnika moguće je dohvatiti sve vožnje koje je taj korisnik "odvozio".

4.2.3. Implementacija davatelja usluga i kontrolera

Kako je ranije objašnjeno, davatelji usluga u NestJS-u su bilo koje klase koje mogu pružiti određenu funkcionalnost ili vrijednost drugim dijelovima aplikacije, a najčešće su to servisi i repozitoriji. U nastavku poglavlja objašnjen je njihov odnos te kako pružaju funkcionalnost kontrolerima.

Repozitorij je klasa koja upravlja pristupom nekom TypeORM entitetu, odnosno tablici u bazi podataka te u TypeScriptu implementira metode za manipulacijom redaka u traženoj tablici. Njenu instancu servis koristit za implementaciju poslovne logike. Na primjeru servisa za upravljanje vozilima (engl. *VehiclesService*) sa slike 4.8. prikazano je kako klasa servisa u vlastitom

konstruktoru inicijalizira repozitorij vozila (engl. *vehiclesRepository*) te ga postavlja kao privatnu varijablu koju može koristiti samo klasa *VehiclesService*.

U nastavku primjera, prikazana je implementacije metode koja koristi repozitorij vozila za dohvaćanje vozila po njegovom id-ju te dodatno dohvaća relaciju vozila s njegovom tvrtkom, uključujući podatke o tvrtki u rezultat upita.

Zatim je prikazana implementacija metode za stvaranje novog vozila na temelju predanih joj informacija koja kao odgovor vraća podatke o upravo stvorenom vozilu te metoda za brisanje vozila po id-ju.

Na isti način moguće je manipulirati bilo kojim drugim entitetom, u bilo kojem drugom servisu, a TypeORM pritom nudi pregršt raznovrsnih i naprednih opcija pri pisanju upita kojima se na lakši način putem TypeScripta može replicirati SQL.

Linija Kod

```
1:     @Injectable()
2:     export class VehiclesService {
3:       constructor(
4:         @InjectRepository(VehicleEntity)
           private readonly vehiclesRepository: Repository<Vehicle>,
5:       ){}
6:
7:       public async findOneById(id: number): Promise<Vehicle> {
8:         const query = this.vehiclesRepository
9:          .createQueryBuilder('vehicle')
10:         .andWhere('vehicle.id = :id', { id })
11:         .andWhere('vehicle.deletedAt IS NULL')
12:         .withDeleted()
13:         .leftJoinAndSelect('vehicle.company', 'company');
14:
15:        const vehicle = await query.getOne();
16:
17:        return vehicle;
18:      }
19:
20:      public async create(type: VehicleType, company: Company) {
21:        const { id: companyId, headquarters } = company;
22:        const location = convertCoordinateToSqlPoint(headquarters);
23:
24:        const vehicle = this.vehiclesRepository.create({
25:         type,
26:         location,
27:         createdAt: new Date(),
28:         companyId,
29:        });
30:        await this.vehiclesRepository.save(vehicle);
31:
32:        return this.findOneById(vehicle.id);
33:      }
34:
35:      public asnyc delete(id: number) {
36:        await this.vehiclesRepository.softDelete({ id });
37:      }
38:    }
```

Sl. 4.8. Servis za upravljanje vozilima

Tako implementiran servis moguće je iskoristiti u odgovarajućem kontroleru. Na sljedećem primjeru sa slike 4.9., prikazano je kako kontroler vozila (engl. *VehiclesController*) može koristiti klasu *VehiclesService*, s ranijeg objašnjenog primjera.

Linija Kod

```
1:     @Controller('vehicles')
2:     export class VehiclesController {
3:        constructor(private vehiclesService: VehiclesService) {}
4:
4:        …
3:        @Put('/:id/enable')
4:        public async enable (@Param('id', ParseIntPipe) id: number){
5:           return this.vehiclesService.enable(id);
6:        }
8:     }
```

Sl. 4.9. Primjer definiranja krajnje točke unutar kontrolera

Dekorater *@Controller('vehicles')* označava klasu kao kontroler koji upravlja putanjama vezanim za vozila. Argument *'vehicles'* koji se prosljeđuje dekorateru postavlja osnovnu putanju za sve putanje unutar tog kontrolera. Dakle, svim krajnjim točkama bit dodaje se prefiks *'/vehicles'*.

Zatim, kontroler u konstruktoru inicijalizira *vehiclesService* varijablu što omogućava metodama krajnjih točaka da koriste servis. Na kraju, implementira metode kojima dodaje dekorater *@Get(), @Post(), @Put()* ili *@Delete()* analogno HTTP metodi koja se koristi za danu krajnju točku.

Na primjeru je, također, prikazana *@Put()* krajnja točka kojoj je zadatak omogućiti vozilo dostupnim za vožnju te pri tome koristi dekoratere za izvlačenje parametara putanje koristeći cijev za transformaciju stringa u broj.

Svi drugi kontroleri i njihove krajnje točke, te servisi koje kontroleri koriste implementirani su na isti način.

4.2.4. Autentikacija i autorizacija

Za potpunu implementaciju NestJS poslužiteljske aplikacije spremnu za korištenje s klijentskim aplikacijama, potrebno je učiniti krajnje točke zaštićenima od neovlaštenog korištenja. Trenutno, navedenim krajnjim točkama može pristupiti bilo koji klijent bez ikakve provjere identiteta. To može dovesti do raznih problema poput dohvaćanja, uređivanja ili brisanja tuđih podataka te umetanja prevelikog broja redaka u bazu podataka što eventualno dovodi do pada performansi ili potpunog ispada NestJS aplikacije.

NestJS, stoga, nudi elegantne i jednostavne načine za određivanje identiteta klijenta koji pristupa krajnjim točkama poslužiteljske aplikacije, odnosno autentikaciju i autorizaciju.

Autentikacija i autorizacija su dva ključna koncepta u sigurnosti informacijskih sustava koji se često koriste zajedno kako bi se osigurao siguran pristup resursima. Autentikacija je proces provjere identiteta korisnika. Autorizacija je proces koji slijedi autentikaciju i određuje što autenticirani korisnik smije raditi [21].

NestJS predlaže korištenje Passport biblioteke koja podržava širok spektar strategija autentikacije, uključujući lokalnu autentikaciju (korisničko ime i lozinka), JSON Web Tokene (JWT), OAuth autentikaciju i mnoge druge.

Prvi korak je pri implementaciji autentikacije Passport bibliotekom je odabrati odgovarajuću strategiju. U ovoj aplikaciji odabrana je JWT strategija jer je JWT format posebno dizajniran da bude kompaktan, što omogućava lako slanje informacija kroz putanje, *POST* zahtjeve, ili unutar HTTP zaglavlja, te da bude samodostatan, noseći sve potrebne informacije o korisniku. Token se sastoji od tri dijela: zaglavlja (engl. *header*), tereta (engl. *payload*), i potpisa (engl. *signature*). Zaglavlje sadrži informacije o tipu tokena i algoritmu kriptiranja. Teret nosi tvrdnje koje se odnose na korisnika i dodatne metapodatke, kao što su korisničko ime, uloga, i vrijeme isteka tokena. Potpis osigurava integritet i autentičnost tokena, sprječavajući njegovu manipulaciju.

Na slici 4.10. prikazana je klasa *JwtStrategy* koja proširuje Passport klasu i implementira metodu za validaciju (engl. *validate*) koja iz predanog joj tokena izvlači podatke o korisniku te ukoliko korisnik zaista postoji, vraća ga kao povratnu vrijednost.

Drugi korak je registrirati ovu klasu u neki od modula. NestJS predlaže stvaranje zasebnog autentikacijskog modula (engl. *AuthModule*) umjesto dodavanje konfiguracije unutar korijenskog modula. Pritom, potrebno je uvesti gotovi *JwtModule* s njegovom konfiguracijom u *AuthModule*, zatim registrirati *JwtStrategy* klasu te uz nju registrirati klasu čuvara (engl. *JwtAuthGuard*) koji se primijenjuje globalno, tj. na cijelu aplikaciju (slika 4.11.). Čuvar koristi strategiju za zaštitu svih krajnjih točaka u aplikaciji.

Linija Kod

```
1:     @Injectable()
2:     export class JwtStrategy extends PassportStrategy('jwt') {
3:       constructor(
4:         private cfg: AppConfigService,
5:         private usersService: UsersService
6:       ){ … }
7:
8:       public asnyc validate(payload: JwtPayload): Promise<User | null> {
9:         const user = await this.usersService.findOneById(payload.id);
10:
10:        if(user && user.enabled){ return user; }
11:
12:        return null;
13:      }
14:    }
```

Sl. 4.10. *JwtStrategy* klasa

Linija Kod

```
1:     @Module({
2:       imports: [
3:         AppConfigModule,
4:         JwtModule.registerAsync({
5:          imports: [AppConfigModule],
6:          useFactory: (cfg: AppConfigService) => {
7:            return {
9:              secret: cfg.get<string>(ENVIRONMENT_VARIABLES.JWT_SECRET),
10:           };
11:         }, …
12:        }),
13:        PassportModule,
14:        UsersModule,
18:      ],
16:      controllers: [AuthController],
17:      providers: [
18:        AuthService,
19:        JwtStrategy,
20:        { provide: APP_GUARD, useClass: JwtAuthGuard },
21:      ],
19:    })
20:    export class AuthModule {}
```

Sl. 4.11. Konfiguracija autentikacijskog modula

Nakon realizacije ova dva koraka, ako klijent želi pristupiti nekoj od krajnjih točaka poslužitelja, u zahtjevu mora poslati token, koji može dobiti samo uspješnom prijavom (engl. *login*) u aplikaciju. Prijava se, standardno, provodi slanjem korisničkog imena i lozinke koje je klijent definirao pri registraciji.

Međutim, nisu sve krajnje točke namijenjene samo ovlaštenim korisnicima. Za javne krajnje točke, kao one za inicijalnu prijavu, NestJS nudi dekorater *@Public()* koji se postavlja iznad metoda unutar kontrolera, a on poništava ulogu globalnog čuvara.

Za autorizaciju, NestJS nudi korištenje kontrole pristupa resursima na bazi uloga. Svakom korisniku potrebno je dodijeliti ulogu - u slučaju ove aplikacije to su uloge običnog korisnika i administratora, odnosno tvrtke. Zatim, potrebno je implementirati čuvar uloga (engl. *RolesGuard*) prikazanog na slici 4.12..

Linija Kod

```
1:     @Injectable()
2:     export class RolesGuard extends CanActivate {
3:        constructor(private reflector: Reflector) {}
4:
5:        canActivate(context: ExecutionContext): boolean {
9:          const req = context.switchToHttp().getRequest();
10:
10:         const required roles = this.reflector.getAllAndOverride(
11:          ROLES_ANNOTATION_KEY,
12:          [context.getHandler(), context.getClass()]
13:         );
14:
15:         if(!requiredRoles) { return true; }
16:
17:         const user: User = req.user;
18:
19:         return requiredRoles.some((role) => user.role_name === role);
20:       }
21:    }
```

Sl. 4.12. Čuvar uloga

Pri pristupanju nekoj od krajnjih točaka aplikacije, ova klasa iz zahtjeva kojeg je klijent poslao dohvaća korisnika i provjerava ima li korisnik dozvoljenu ulogu za pristup krajnjoj točki. Definiranje koje uloge mogu pristupiti kojoj krajnjoj točki izvodi se dodavanjem dekoratera *@Roles()* iznad metode krajnje točke (slika 4.13.).

Linija Kod

```
1:     @Controller('vehicles')
2:     export class VehiclesController {
3:       constructor(private vehiclesService: VehiclesService) {}
4:
4:       …
5:       @Put('/:id/enable')
6:       @Roles(UserRole.ADMIN)
7:       public async enable(@Param('id', ParseIntPipe) id: number){
8:         return this.vehiclesService.enable(id);
9:       }
11:    }
```

Sl. 4.13. Upotreba @Roles() dekoratera

Dodavanjem autentikacije i autorizacije, poslužiteljska NestJS aplikacija postaje sigurna i spremna za korištenje.

4.3. Najvažniji dijelovi React web aplikacije za tvrtke

Uzimajući u obzir činjenicu da je literatura na temu osnovnog postavljanja React web aplikacija široko dostupna [25] te da je ono u posljednje vrijeme postalo izuzetno jednostavno zbog razvojnih alata poput Vitea, koji nude predefinirane postavke i jednom naredbom podižu sve pomoćne biblioteke s najmodernijim konfiguracijama [27], rad se u ovome poglavlju fokusira na implementaciju specifičnih rješenja za glavne probleme React aplikacije - dohvaćanje podataka s poslužitelja, upravljanje globalnim stanjem aplikacije te stvaranje korisničkog sučelja. Rješenja na sva tri navedena problema objašnjena su na primjerima iz klijentske web aplikacije za administratore javnog prijevoza i tvrtke koje nude usluge, dane ovim radom.

4.3.1. Komunikacija s poslužiteljem

Većina osnovnih razvojnih okvira za klijentsku stranu weba nema jasno definiran način dohvaćanja ili ažuriranja podataka s poslužitelja. Zbog toga programeri često stvaraju vlastite načine za dohvaćanje podataka miješajući u njih biblioteke koje upravljaju stanjem klijentske aplikacije, primjerice Redux. Takve biblioteke nisu idealne za rad s asinkronim, odnosno poslužiteljskim stanjem, jer je poslužiteljsko stanje:

- pohranjeno na udaljenoj lokaciji koju klijent ne može kontrolirati,
- zahtijeva asinkrone API-je za dohvaćanje i ažuriranje,
- može se promijeniti bez znanja klijenta,
- može postati zastarjelo.

Ubroje li se u to izazovi poput predučitavanja podataka, lijenog učitavanja (engl. *lazy-loading*), kontroliranja višestrukih zahtjeva istovremeno te optimizacije performansi, biblioteke za rad s klijentskim stanjem i njihova neadekvatna rješenja postaju potpuno pogrešan odabir za dohvaćanje podataka s poslužitelja.

TanStack (ili React) Query, biblioteka korištena u ovom radu, pomaže u rješavanju ovih izazova te time smanjuje količinu koda u aplikaciji, čini aplikaciju lakšom za održavanje te poboljšava performanse aplikacije. Na primjerima u nastavku pokazano je kako to radi [28].

Kod na slici 4.14. predstavlja primjer React komponente koja koristi glavnu značajku biblioteke, odnosno *useQuery()* kuku za slanje upita (*GET* zahtjeva). Zahtjev od poslužitelja traži podatke o romobilima. Tijekom učitavanja podataka, komponenta prikazuje poruku *"Loading scooters..."*. Nakon uspješnog dohvaćanja, prikazuje podatke o romobilima u obliku liste.

Linija Kod

```
1:     export const ScootersList = () => {
2:       …
3:       const { data, isLoading } = useQuery({
4:         queryKey['scooters'],
5:         queryFn: async () => axiosClient.get<Scooter[]>(
6:           '/vehicles/scooters'
7:         );
8:       });
9:
10:      if(isLoading){ return <p>Loading scooters…</p> }
11
12:      return (
13:        …
14:        <ul>
15:          { _.map(scooters, (scooter) => (
16:            <li>{scooter.code}</li>
17:          )}
18:        </ul>
19:      );
20:    }
```

Sl. 4.14. Dohvaćanje podataka o romobilima s poslužitelja

Osim podataka o repozitoriju, kako je prikazano, *useQuery()* kuka daje komponenti pristup raznim informacijama o upućenom zahtjevu - je li dohvaćanje u tijeku, je li dohvaćanje uspješno ili se dogodila pogreška, metode za ručno, ponovno slanje zahtjeva te desetke drugih opcija – sve kroz nekoliko linija koda. Prije pojave ove biblioteke, za implementaciju samo djelića funkcionalnosti koje ona nudi, programeri su morali napisati stotine linija koda koji je često imao loše performanse [28].

Na sličan način radi i *useMutation()* kuka koju biblioteka predlaže za izvršavanje operacije slanja, uređivanja ili brisanja podataka na poslužitelju, odnosno *POST*, *PUT* i *DELETE* zahtjeve. Ono što ga razlikuje od *useQuery()* jest potreba da se pri slanju zahtjeva pošalju dodatni podaci koje poslužitelj obrađuje te zbog toga vraća *mutate()* funkciju. Nju ostatak komponente koristi kada želi inicirati zahtjev predajući joj podatke kroz parametre (slika 4.15.).

Linija Kod

```
1:     export const ScootersTable = () => {
2:       …
3:       const { mutate, isPending } = useMutation({
4:         queryKey['delete-scooter'],
5:         queryFn: async ({ id }: { id: number }) => {
6:           await axiosClient.delete<void>(
7:             '/vehicles/scooters/{id}'
8:           );
9:       });
10:
11:      return (
12:        …
13:        <div>
14:          { _.map(scooters, (scooter) => (
15:            <div> … </div> // scooter information
16:            <button onPress={() => mutate({ id: scooter.id })}>
17:              { isPending ? 'Deleting …' : 'DELETE' }
18:            </button>
19:          )}
20:        </div>
21:      );
20:    }
```

Sl. 4.15. Primjer korištenja *useMutation()* kuke

Osim ove dvije kuke koje su u aplikaciji korištene za sve zahtjeve prema poslužitelju, TanStack Query nudi mnoštvo drugih načina za upravljanje poslužiteljskim stanjem. Neke od njih su *useQueryClient()* za globalno upravljanje zahtjevima, *useInfiniteQuery()* za beskonačno učitavanje podataka te *useIsFetching()* i *useIsMutating()* za praćenje aktivnih zahtjeva. Također,

omogućuje predučitavanje, otkazivanje zahtjeva, provjeru pogrešnih podataka i nudi razvojne alate za pregled stanja te otklanjanje grešaka unutar aplikacije.

4.3.2. Rukovanje globalnim stanjem aplikacije

Rukovanje globalnim stanjem u React aplikacijama može biti izazovno zbog povećanja njegove složenosti rastom obujma aplikacije. Komponente koje često trebaju pristupati i ažurirati zajedničke podatke mogu prouzročiti probleme s razumijevanjem i održivosti koda. Zatim, konstantno prosljeđivanje podataka kroz nekoliko razina hijerarhije komponenti (engl. *prop drilling*) može dodatno zakomplicirati situaciju. Zbog navedenih problema sinkronizacija stanja između različitih dijelova aplikacije postaje teška – promjene u jednom dijelu moraju biti reflektirane u svim dijelovima koji koriste isto stanje. Sukladno tome, performanse aplikacije mogu patiti zbog nepotrebnih ponovnih osvježavanja komponenti koje ovise o ažuriranom stanju. Srećom, React zajednica nudi pregršt biblioteka koje rješavaju navedene probleme

Zustand je biblioteka za upravljanje globalnim stanjem u React aplikacijama koja se fokusira na jednostavnost i performanse. Omogućuje centralizirano stanje dostupno cijeloj aplikaciji, eliminirajući potrebu za prosljeđivanjem podataka kroz hijerarhiju komponenti. API za definiranje i korištenje globalnog stanja je intuitivan jer komponente koriste samo jedan *useStore()* kuku za pristup stanju, što smanjuje složenost koda. Zustand, također, omogućuje korištenje selektora za precizno odabiranje dijelova stanja koje komponente trebaju, smanjujući nepotrebna osvježavanja komponenti. Također, ova biblioteka lako se integrira s rješenjima za asinkrono dohvaćanje podataka poput ranije spomenutog TanStack Queryja. Na slici 4.16. prikazan je primjer korištenja Zustand biblioteke unutar aplikacije [29].

Kod dan u primjeru definira Zustand centralizirano stanje za upravljanje autentikacijom u React aplikaciji. *AuthState* sučelje opisuje strukturu stanja koja uključuje *accessToken*, *refreshToken* i *company* atribute. Atribut *company* predstavlja tvrtku odnosno administratora koji se prijavljuje u aplikaciju, a tokeni predstavljaju vrijednosti koje administrator koristi za vlastitu identifikaciju pri slanju zahtjeva prema poslužitelju.

AuthStore sučelje proširuje *AuthState* s metodama za postavljanje i resetiranje stanja. Kuka *useAuthStore()* stvara stanje s inicijalnim vrijednostima te pruža definirane metode za ažuriranje i resetiranje stanja koje se mogu koristiti u drugim komponentama. Korištenjem navedene kuke, bilo koja komponenta u aplikaciji može dohvaćati i ažurirati autentikacijsko stanje bez da se brine o performansama aplikacije.

Linija Kod

```
1:     export interface AuthState {
2:       accessToken: string | null;
3:       refreshToken: string | null;
4:       company: Company | null;
5:     }
6:
7:     export interface AuthStore extends AuthState {
8:       setAuthState: (state: AuthState) => void;
9:       resetAuthState: () => void;
10:
11:      setAccessToken: (accessToken: string) => void;
12:      setRefreshToken: (refreshToken: string) => void;
13:      setCompany: (company: Company) => void;
14:    }
15:
16:    export const useAuthStore = create<AuthStore>((set) => ({
17:      accessToken: null,
18:      refreshToken: null,
19:      company: null,
20:
21:      setAuthState: (state) => { set(state); },
22:      resetAuthState: () => {
23:        set({ accessToken: null, refreshToken: null, company: null });
24:      },
25:
26:      setAccessToken: (accessToken) => set({ accessToken }),
27:      setRefreshToken: (refreshToken) => set({ refreshToken }),
28:      setCompany: (company) => set({ company }),
29:    }));
```

Sl. 4.16. Zustand implementacija za autentikacijsko stanje React aplikacije

4.3.3. Stvaranje korisničkog sučelja

Implementacijom komunikacije s poslužiteljem te rukovanja globalnim stanjem, React aplikaciji omogućeno je da napravi posljednji korak - stvori vizualno sučelje koje korisniku daje pregled podataka i elemente za interakciju s tim podacima.

Međutim, umjesto korištenja čistog HTML-a i CSS-a, aplikacija se oslanja na jednu od biblioteka koja nudi gotove React komponente. Takve biblioteke su zbirke unaprijed izgrađenih, ponovno upotrebljivih komponenata i mogu sadržavati sve od osnovnih elemenata korisničkog sučelja poput gumba i polja za unos podataka do složenih funkcionalnosti poput kompleksnih tablica,

odabira datuma, modalnih prozora te navigacije [30]. Popularni primjeri su Material UI, ChakraUI, AntDesign i React Bootstrap.

One u pozadini koriste sve ranije objašnjene principe dizajna: HTML elementi se koriste za strukturiranje sadržaja komponenata, izvedenice CSS-a se koristi za njihovo stiliziranje, dok React značajke poput kuka omogućuju komponentama interaktivnost i kontrolu vlastitog stanja.

Međutim, kako su na njima godinama radili ogromni timovi od više stotina ili tisuća ljudi, riječ je o izuzetno kompleksnim zbirkama komponenti s vlastitim sustavima boja, ikona, margina, responzivnosti, prilagodbe i mnogih drugih značajki koje bi običnom programeru bile potpuno neizvedive za napisati od nule, čak i za vrlo jednostavnu komponentu poput gumba. Ove biblioteke, stoga, štede vrijeme malim timovima koji se žele fokusirati na poslovnu logiku aplikacije, a istovremeno pružaju moderno dizajnirane, pristupačne i brze komponente koje je po potrebi te aplikacije moguće prilagoditi.

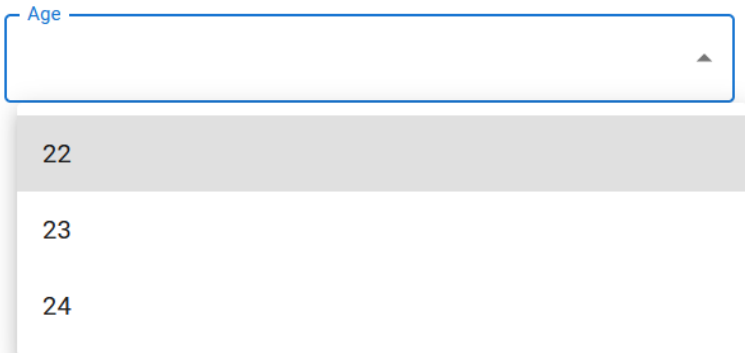
U ovome radu korištena je biblioteka Material UI koja implementira ranije objašnjene principe, pritom koristeći popularni jezik dizajna - Googleov Material Design. Na slici 4.17. prikazan je primjer korištenja Material UI biblioteke.

Ova komponenta implementira padajući izbornik i koristi se unutar raznih formi za unos podataka diljem aplikacije [30]. Ono što Material UI nudi pri izgradnji takve komponente su “okvir” *FormControl*, dva elementa koja on obuhvaća - labelu (engl. *InputLabel*) i padajući meni (engl. *Select*) s listom opcija (engl. *MenuItem*). Jednostavnim prosljeđivanjem parametara i mapiranjem opcija na odgovarajućem mjestu, dobiva se rezultat sa slike 4.18..

Linija Kod

```
1:     export const SelectInput = ({ name, label, options }) => {
2:       return (
3:         <Field name={name}>
4:          {({ field, form }) => {
5:            const handleChange = (event) => { form.handleChange(…); };
6:
7:            return (
8:              <FormControl fullWidth variant="outlined">
9:                <InputLabel>{label}</InputLabel>
10:               <Select value={field.value} onChange={handleChange}>
11:                 {_.map(options, ({value, label}, i) => {
12:                   <MenuItem key={i} value={value}>
13:                    {label}
14:                   </MenuItem>
15:                 ))}
16:               </Select>
17:             </FormControl>
18:           );
19:         })
20:       </Field>
21:     );
22:   };
```

Sl. 4.17. Implementacija komponente padajućeg izbornika



Sl. 4.18. Izgled komponente padajućeg izbornika

Osim Material UI biblioteke, za izgradnju korisničkog sučelja u aplikaciji korištene su i tzv. "bezglave" (engl. *headless*) biblioteke. Takve biblioteke su setovi funkcionalnosti koje ne propisuju ili ne implementiraju korisničko sučelje izravno, već pružaju logiku i upravljanje stanjem bez predodređenog prikaza. Ovo omogućava programerima da koriste bilo koju biblioteku komponenti ili vlastite komponente za vizualizaciju, dok *headless* biblioteka upravlja logikom i

podacima u pozadini. Primjeri takvih biblioteka u ovoj aplikaciji su Formik za upravljanje formama i TanStack Table za upravljanje tablicama.

Formik nudi opcije za validaciju i slanje podataka, upravljanje greškama te rukovanje promjenama i podnošenjem formi bez potrebe za ručnom implementacijom nabrojanih funkcionalnosti. Primjer njegova korištenja, također je vidljiv na slici 4.17.. Ugradnjom *SelectInput* komponente u bilo koju Formik formu, komponenta dobija pristup cijeloj formi i njenoj *handleChange()* metodi koja ažurira vrijednost polja u Formikovom stanju forme kada se dogodi promjena, odnosno korisnik odabere opciju u padajućem izborniku.

React Table nudi funkcionalnosti za upravljanje tablicama podataka kao što su sortiranje, filtriranje, grupiranje, paginacija te upravljanje stanjem redaka i ćelija, na sličan način kao i Formik, dok se za sam vizualni dizajn tablica brinu Material UI komponente poput onih za zaglavlje (engl. *TableHead*) i tijelo tablice (engl. *TableBody*) te za retke (engl. *TableRow*) i ćelije (engl. *TableCell*).

4.4. Najvažniji dijelovi React Native mobilne aplikacije za korisnike

U ovom poglavlju fokus je na ključnim aspektima razvoja mobilnih aplikacija koristeći React Native, s posebnim naglaskom na one dijelove aplikacije koji se razlikuju od razvoja web aplikacija u Reactu. Koncepti i tehnologije poput komunikacije s poslužiteljem pomoću TanStack Query biblioteke, upravljanja globalnim stanjem pomoću Zustand biblioteke, te stvaranja sučelja koristeći zbirke komponenata kao što je MaterialUI, već su detaljno objašnjeni u prethodnom poglavlju i primjenjivi su u kontekstu React Native okruženja te ih, stoga, nije potrebno ponovo objašnjavati.

Međutim, postoji nekoliko specifičnosti React Nativea koje zahtijevaju dodatnu pažnju, a to su navigacija unutar mobilne aplikacije, korištenje biblioteke za prikazivanje karte područja na kojem se nalazi korisnik te pristup hardverskim API-jima mobilnih uređaja, odnosno kameri.

4.4.1. Navigacija

Implementacija navigacije u React Nativeu jedan je ključnih koraka pri razvoju aplikacija jer omogućava korisnicima da se kreću između različitih ekrana i dijelova aplikacije. Za razliku od web aplikacija razvijenih u Reactu, gdje se navigacija često oslanja na poveznice i preglednikov API za povijest, React Native aplikacije zahtijevaju drugačiji pristup navigaciji zbog njihovog okruženja, odnosno nedostatka web preglednika.

U React Native aplikacijama, navigacija se postiže korištenjem JavaScript biblioteka koje simuliraju iskustvo navigacije prisutno u izvornim mobilnim aplikacijama. Ove biblioteke omogućuju stvaranje stoga navigacije, navigacije u obliku kartica (engl. *tabs*), navigacije u obliku ladice (engl. *drawer*) i drugih oblika navigacije koje korisnici očekuju u mobilnim aplikacijama.

U ovome projektu korištene su dvije takve biblioteke u kombinaciji - React Navigation Native biblioteka te React Navigation Drawer biblioteka.

Prva pruža zajedničku infrastrukturu za različite tipove navigacije te prema [31] omogućuje:

- Stvaranje navigacijskog kontejnera (engl. *NavigationContainer*) koji služi kao korijen navigacijskog stanja.
- Definiranje navigacijskih stogova (engl. *stack*) kroz koje se korisnik može kretati, uključujući mogućnost povratka na prethodni ekran.

Druga dodaje podršku za navigaciju u obliku ladice (engl. *drawer*). Ladica daje korisnicima opciju da povlačenjem ili pritiskom na gumb otvore bočni izbornik s navigacijskim opcijama. Prema [31], ova biblioteka omogućuje:

- Stvaranje komponente ladice unutar navigacijske strukture koja korisnicima daje pristup različitim dijelovima aplikacije kroz bočni izbornik.
- Prilagodbu izgleda sadržaja ladice, animacija otvaranja i zatvaranja, i ostalih vizualnih elemenata.

Na primjeru sa slike 4.19. prikazana je suradnja ove dvije biblioteke kroz *RootNavigator* komponentu koja služi kao korijenski navigacijski upravitelj u React Native aplikaciji. Koristi se za određivanje početnog ekrana na temelju trenutnog autentikacijskog stanja korisnika. Ovisno o tome je li korisnik prijavljen ili ne, prikazuje različite navigacijske tokove.

Linija Kod

```
1:     export const RootNavigator = () => {
2:       const { user } = useAuthStore();
3:
4:       return (
5:         <>
6:           { user === null ? (
7:             <Stack.Navigator screenOptions={{ … }}>
8:               <Stack.Screen name=”Auth” component={AuthNavigator} … />
9:             </Stack.Navigator>
10:          ) : (
11:            <DrawerNavigator />
12:          )}
13:        </>
14:      );
15:    };
```

Sl. 4.19. Korijenski navigacijski upravitelj React Native aplikacije

Ako korisnik nije prijavljen prikazuje se *AuthNavigator*, odnosno skup ekrana za autentikaciju (prijavu i registraciju) korisnika. Ako je korisnik prijavljen, prikazuje se *DrawerNavigator*, odnosno skup ekrana kojima prijavljeni korisnik može pristupiti i kojima korisnik upravlja putem ladice. Na slici 4.20. je implementacija *DrawerNavigator* komponente čija su djeca: početni ekran, ekran za korištenje usluge vožnje vozila, ekran za pregled povijesti vožnja, ekran za pregled vijesti, te ekran za upravljanje korisničkim profilom.

Linija Kod

```
1:     export const DrawerNavigator = () => {
2:       …
3:       return (
4:         <Drawer.Navigator screenOptions={{ … }}>
6:           <Drawer.Screen name=”Home” component={HomeScreen} … />
7:           <Drawer.Screen name=”Drive” component={DriveScreen} … />
8:           <Drawer.Screen name=”Rides” component={RidesScreen} … />
9:           <Drawer.Screen name=”News” component={NewsScreen} … />
10:          <Drawer.Screen name=”Profile” component={ProfileScreen} … />
11:        </Drawer.Navigator>
12:      );
13:    }
```

Sl. 4.20. *DrawerNavigator* komponenta za upravljanje navigacijom putem ladice

Osim samih komponenti navigacije, ove biblioteke nude dodatne kuke i metode za upravljanje navigacijom. Primjerice, kuke *useNavigation()*, *useRoute*() i *useFocusEffect()* omogućuju pristup navigacijskim metodama i navigacijskom stanju iz bilo koje komponente unutar React Native

aplikacije, dok metode kao što su *navigate(), goBack(), reset()* daju programeru gotove načine za implementaciju korisniku intuitivnog navigacijskog toga. Na slici 4.21. primjer je korištenja *useNavigation()* koja nudi metodu za ručno otvaranje ladice za navigaciju na klik bilo kojeg gumba.

Linija Kod

```
1:     export const FloatingMenu = ({ … }) => {
2:       …
3:       const navigation = useNavigation();
4:
5:       return (
6:         <View>
7:           …
8:           <IconButton
9:             icon={'drawer'}
10:            onPress={() => navigation.dispatch(DrawerActions.open())}
11:          />
12:        </View>
13:      );
14:    }
```

Sl. 4.21. Primjer upotrebe *useNavigation()* kuke

4.4.2. Karte

S obzirom na to da je cilj ove React Native aplikacije pružiti korisnicima lakše kretanje u prometu, jedan od najvažnijih koraka je implementacija pregleda i korištenja usluga koje se nude na području mjesta u kojem se nalazi korisnik. Za implementaciju takve značajke korištena je React Native Maps biblioteka.

Prema [32], ova biblioteka omogućava integraciju karata u mobilne aplikacije na jednostavan i fleksibilan način, a neke od njenih brojnih značajki objašnjene su u nastavku:

- Nudi mogućnost odabira pružatelja usluga karte te prilagodbu izgleda karte, što pomaže pri usklađivanju dizajna karte s ukupnim dizajnom aplikacije.
- Omogućava postavljanje oznaka na karti za prikaz specifičnih lokacija, odnosno točaka interesa koje su u slučaju ove aplikacije lokacije vozila i stanica za javni prijevoz.
- Nudi sustav za crtanje proizvoljnih geometrijskih likova i linija na karti, što je korisno za prikazivanje područja ili u ovom slučaju, linija javnog prijevoza.
- Pruža potpunu kontrolu nad gestama za upravljanje kartom uključujući pomicanje, zumiranje i rotaciju karte.

Na slici 4.22. prikazano je pojednostavljeno korištenje ove biblioteke unutar aplikacije. Pri inicijalizaciji komponente za prikaz karte (engl. *MapView*) potrebno joj je proslijediti ime davatelja usluge, inicijalnu regiju koju prikaz učitava te stil karte.

Linija Kod

```
1:     export const DriveScreen = ({ … }) => {
2:       …
3:       return (
4:         <MapView
5:           provider={PROVIDER_GOOGLE}
6:           initialRegion={INITIAL_REGION}
7:           style={styles.mapView}>
8:           {_.map(vehicles, (vehicle) => (
9:             <VehicleMarker
10:              key={vehicle.id}
11:              vehicle={vehicle}
12:              onPress={handleVehicleMarkerPress}
13:            />
14:          ))}
15:          {currentLine && (
16:            <MapViewDirections
17:              origin={_.first(currentLine.stations).location}
18:              destination={_.last(currentLine.stations).location}
19:              waypoints={_.slice(currentLine.stations, 1, -1)}
20:              style={styles.directions}
21:            />
22:          ))}
23:        </MapView>
24:      );
25:    }
```

Sl. 4.22. Korištenje komponenti React Native Maps biblioteke

Za prikaz oznaka vozila i stanica za javni prijevoz na karti, ručno su stvorene komponente koje nadograđuju komponentu *Marker* te su im predane metode koje reagiraju na dodir korisnika.

Dodatno, kada korisnik odabere stanicu koja ga zanima, dobiva prikaz linija koje prolaze kroz odabranu stanicu. Ako, zatim, odabere neku od linija, na karti mu se prikazuje cijela ruta kojom ta linija prolazi u boji te linije. Za to je korištena komponenta *MapViewDirections* kojoj je potrebno proslijediti početak linije (prvu stanicu, engl. *origin*), kraj linije (zadnju stanicu, engl. *destination*), ostale stanice unutar rute (engl. *waypoints*) te boju i debljinu linije.

4.4.3. Kamera

Jedna od usluga koju tvrtke putem aplikacije mogu nuditi korisnicima je vožnja električnih romobila i bicikala po uzoru na, u radu ranije spomenutu, Bolt aplikaciju.

Kada korisnik odabere romobil na karti te je u njegovoj blizini, ako želi aktivirati romobil za početak vožnje, potrebno je unijeti njegovu šifru. Tu značajku moguće je implementirati jednostavnim elementom za unos teksta gdje korisnik tipkovnicom unosi traženu šifru. Međutim, za to postoji još elegantnije rješenje koje se oslanja na hardver današnjih pametnih telefona, odnosno kameru. Umjesto ručnog upisivanja šifre, korisnik može iskoristiti kameru za skeniranje koda koji automatski aktivira romobil.

Tip koda koji je korišten u ovoj aplikaciji je QR kod (engl. *Quick Response* - brzi odgovor). To je dvodimenzionalni kod koji se u kontekstu mobilnih aplikacija često koriste za brzo dijeljenje hiperveza, autentikaciju, plaćanje usluga ili jednostavno dijeljenje bilo kojeg oblika podataka.

React Native i njegova zajednica nude postojeća rješenja za korištenje kamere i očitavanje QR kodova u obliku React Native Vision Camera biblioteke [33]. Na slici 4.23. prikazano je rukovanje kukama i metodama koje nudi ta biblioteka.

Za inicijalizaciju kamere koristi se kuka *useCameraDevice()* s argumentom koji određuje koristi li se prednja ili stražnja kamera uređaja, a vraća instancu kamere. Pri prvom korištenju kamere na uređaju, korisnik mora dati dozvolu aplikaciji za korištenje kamere. To mu omogućavaju *useState()* i *useEffect()*. Konačno, za očitavanje QR koda, biblioteka nudi *useCodeScanner()* kuku koja prima argumente za tip koda i funkciju koja se izvršava kada kamera automatski očita kod. Ako je sve uspješno inicijalizirano, potrebno je samo proslijediti dobivene instance komponenti *Camera* koja unutar korisničkog sučelja prikazuje ono što vidi kamera.

Linija Kod

```
1:     export const StartRideModal = ({ … }) => {
2:       …
3:       const [hasPermission, setHasPermission] = useState(false);
4:       const device = useCameraDevice('back');
5:
6:       useEffect(() => {
7:         (async () => {
8:           const status = await Camera.requestCameraPermission();
9:           setHasPermission(status === 'granted');
10:        })();
11:      }, []);
12:
13:      const codeScanner = useCodeScanner({
14:        codeTypes: ['qr'],
15:        onCodeScanned: (code) => { … },
16:      })
13:
14:      return (
15:        <Modal …>
16:          …
17:          {device && hasPermission && (
18:            <Camera device={device} isActive codeScanner={codeScanner}/>
19:          )}
21:         </Modal>
22:      );
23:    };
```

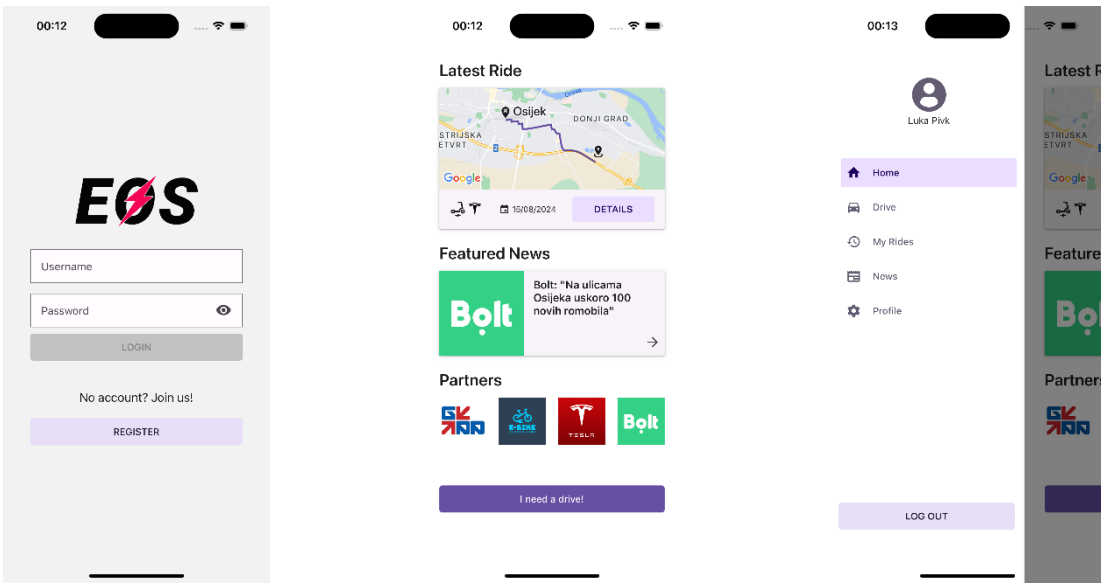
Sl. 4.23. Inicijalizacija komponente kamere u React Native aplikaciji

5. KORISNIČKO SUČELJE I PRIMJENA KLIJENTSKIH APLIKACIJA

U ovom poglavlju opisani su dizajn i primjena korisničkih sučelja dviju klijentskih aplikacija razvijenih u okviru ovog rada. Mobilna aplikacija, izgrađena korištenjem React Native tehnologije, namijenjena je korisnicima koji žele pristupiti uslugama tvrtki koje nude električne automobile, bicikle i romobile, kao i javni prijevoz te pregledavati vijesti. Ova aplikacija omogućava korisnicima intuitivan i učinkovit način upravljanja svojim transportnim potrebama i pristupom aktualnim informacijama. Web aplikacija, izrađena u Reactu, služi kao administratorski alat za tvrtke, omogućujući im upravljanje vozilima, javnim prijevozom i vijestima. Kroz vizualni pregled ovih aplikacija, objašnjeni su glavni elementi korisničkog sučelja te njihova namjena.

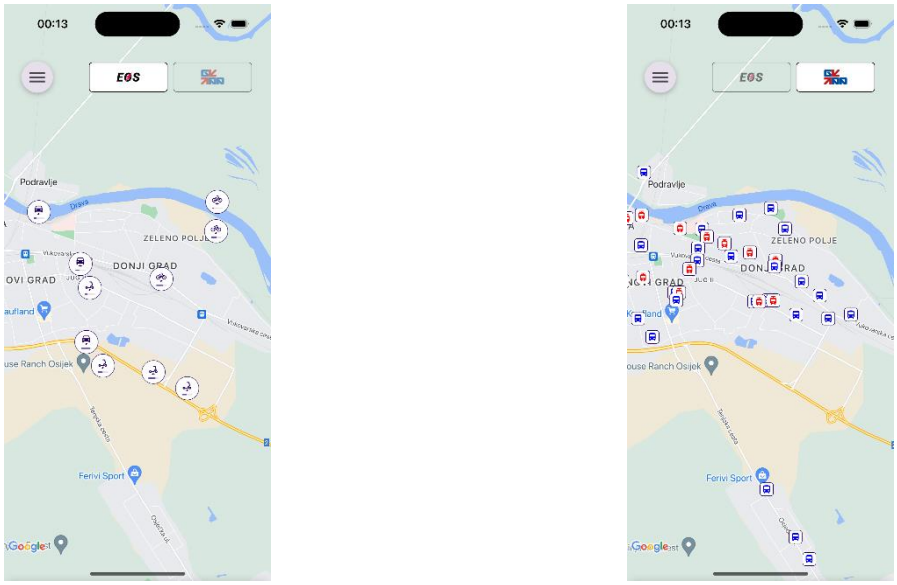
5.1. Aplikacija za korisnike

Na slici 5.1. prikaz je početnog sučelja kojeg korisnik vidi pri korištenju mobilne aplikacije. Prvi ekran s lijeva sadrži standardnu formu za prijavu korisnika u aplikaciju korisničkim imenom i lozinkom. Ekran u sredini početni je ekran za prijavljenog korisnika gdje se prikazuju osnovne informacije poput posljednje korisnikove vožnje, istaknute vijesti i sl. Ekran s desna prikazuje izgled ladice za navigaciju kojom korisnik može pristupiti ostalim dijelovima aplikacije te se odjaviti iz aplikacije.



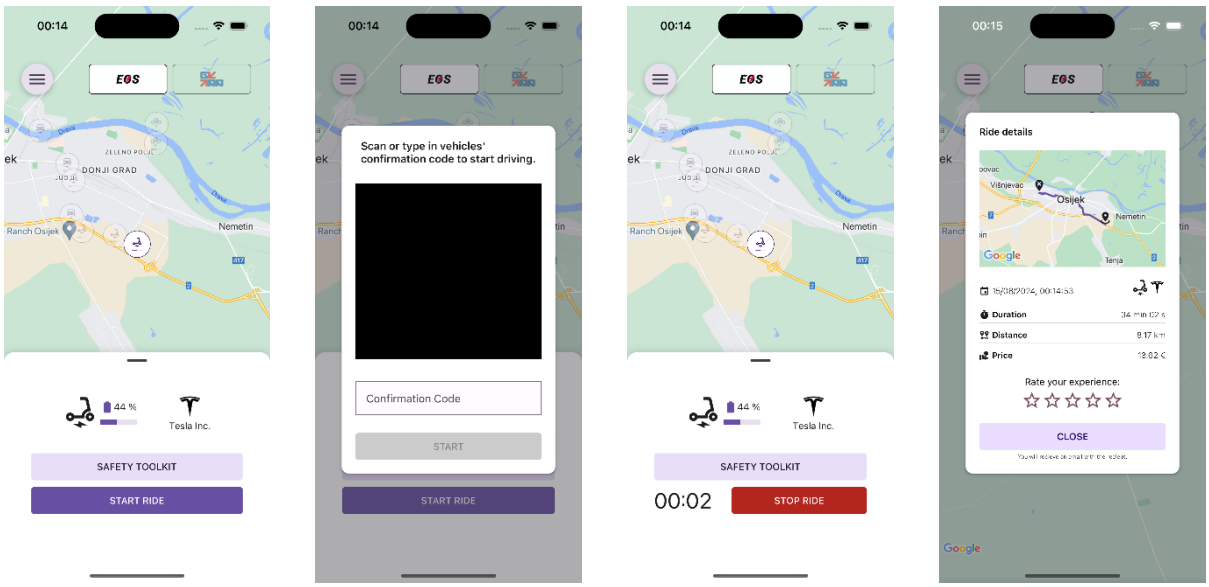
Sl. 5.1. Sučelje za prijavu, početni ekran i ladica za navigaciju

Slika 5.2. prikazuje ekran aplikacije na kojemu se nudi glavna značajka aplikacije – korištenje usluga koje nude privatne tvrtke poput vožnje električnih automobila, bicikala ili romobila te pregled informacija o javnom prijevozu. Na karti grada, u kojem se korisnik nalazi, prikazane su točke interesa, odnosno vozila (ekran s lijeva) ili stanice javnog prijevoza (ekran s desna).



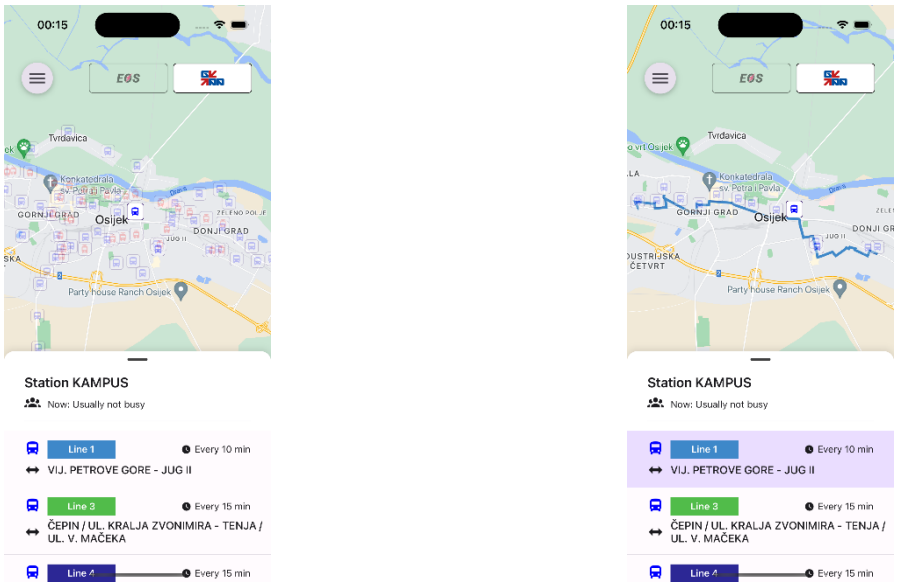
Sl. 5.2. Glavni ekran aplikacije s kartom transportnih usluga

Da bi korisnik započeo vožnju, primjerice električnim romobilom, prvo je potrebno dodirnuti željeno vozilo na karti, nakon čega se otvara meni s donje strane ekrana koji prikazuje dodatne informacije o vozilu. Zatim, dodirnuti gumb *Start Ride* te skenirati QR kod vozila ili ručno upisati kod u predviđeno mjesto čime vožnja započinje. Gumbom *Stop Ride* korisnik završava vožnju i na ekranu dobiva modal s informacijama o cijeni vožnje gdje, također, može ocijeniti uslugu. Slika 5.3. prikazuje izgled sučelja za navedeno.



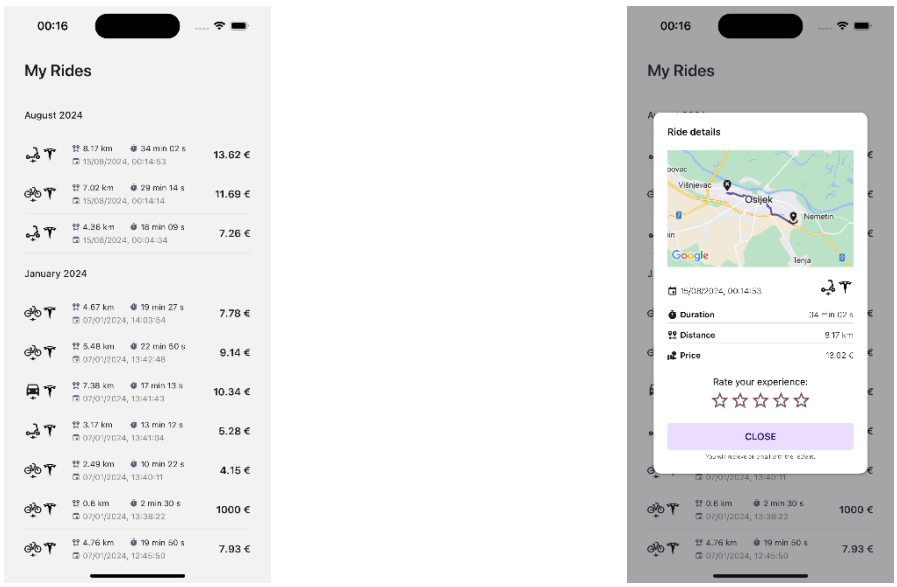
Sl. 5.3. Sučelje za početak i kraj vožnje

Na slici 5.4. prikazano je alternativno korištenje istog ekrana. Dodirom na stanicu javnog prijevoza korisnik u meniju na donjoj strani ekrana dobiva informacije o svim linijama koje prolaze kroz odabranu stanicu. Zatim, odabirom neke od linija s liste, korisniku se na karti označava cijela ruta linije u odgovarajućoj boji.



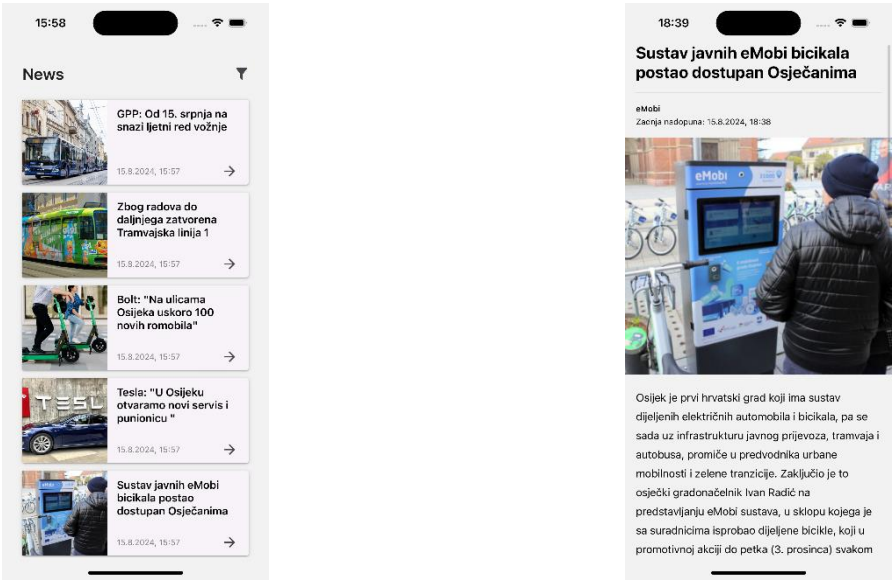
Sl. 5.4. Pregled dostupnih linija za odabranu autobusnu ili tramvajsku stanicu

Za pregled povijesti vožnji korisniku je dan poseban ekran (slika 5.5.) na kojemu se nalazi lista vožnji s osnovnim informacija. Za više informacija o pojedinoj vožnji potrebno je dodirnuti jednu od vožnji s liste, nakon čega se otvara modal koji prikazuje detalje kao i u trenutku stopiranja vožnje, uključujući prikaz vožnje na karti.



Sl. 5.5. Sučelje za pregled povijesti vožnji

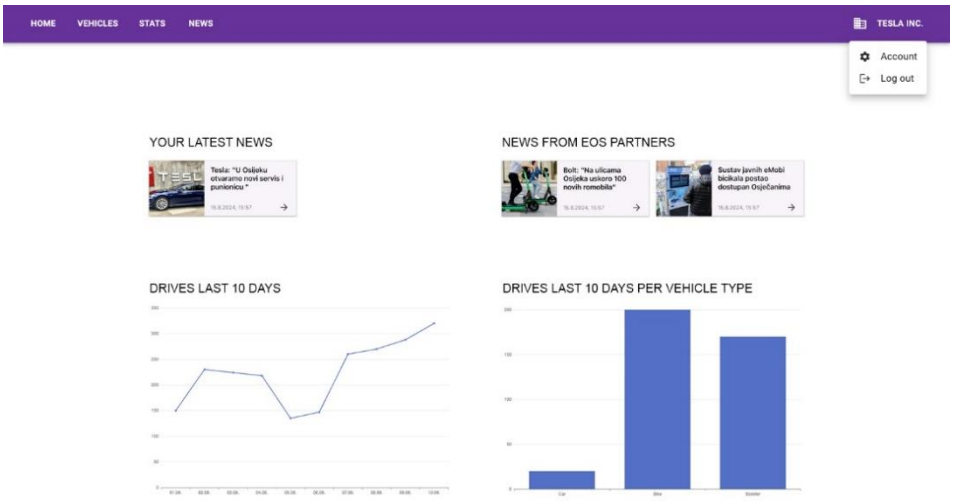
Dodatno, kako bi korisnik bio u toku s najnovijim informacija o uslugama aplikacije i njenih partnera, na posebnom sučelju može pregledavati vijesti koje objavljuju tvrtke. Na slici 5.6. prikaz je tog sučelja – s lijeva je ekran na kojemu je lista vijesti, a s desna ekran za pregled pojedinačne vijesti kojemu korisnik pristupa dodirom na neku vijest s liste.



Sl. 5.6. Sučelje za pregled vijesti

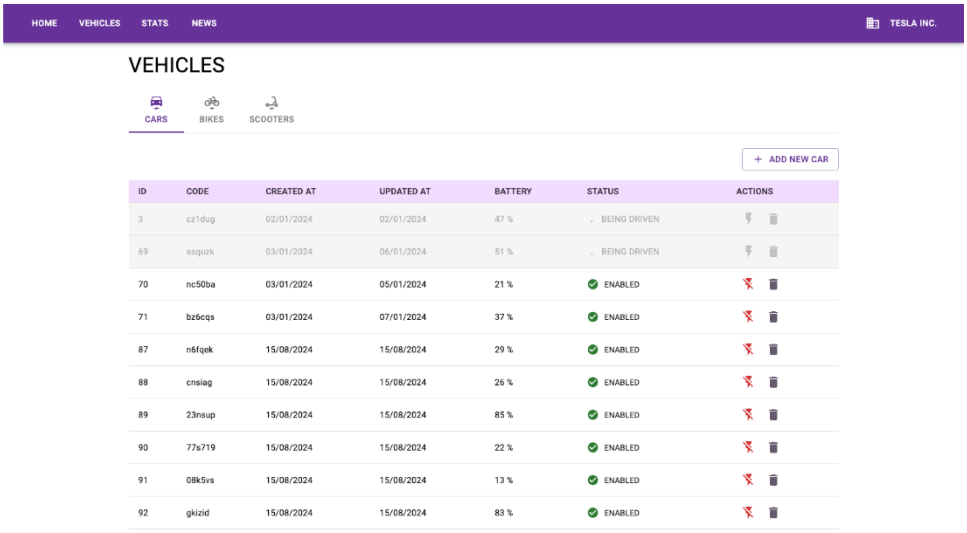
5.2. Aplikacija za tvrtke

Tvrtke za upravljanje vlastitim uslugama koriste web aplikaciju, a za prijavu koriste korisničko ime i lozinku, jednako kao i korisnici. Nakon prijave aplikacija učitava početni ekran (slika 5.7.) gdje administrator može vidjeti navigacijsku traku, vlastite i tuđe aktualne vijesti te vizualni prikaz podataka o korisničkoj upotrebi tvrtkinih usluga u posljednjih deset dana.



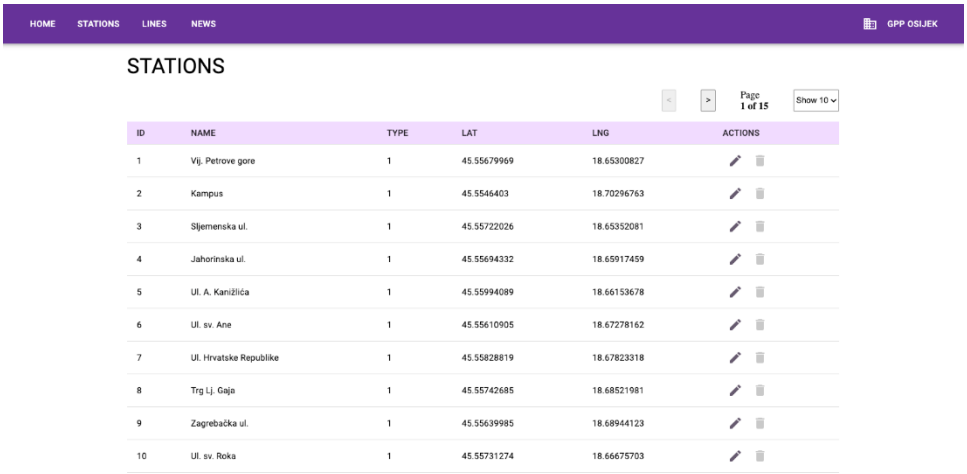
Sl. 5.7. Početna stranica

Vozilima administrator može upravljati sučeljem sa slike 5.8., kojemu pristupa navigacijskom trakom, a tamo može pregledavati informacije o aktivnim vozilima, dodavati nova vozila, deaktivirati vozila ili potpuno obrisati vozila iz sustava.

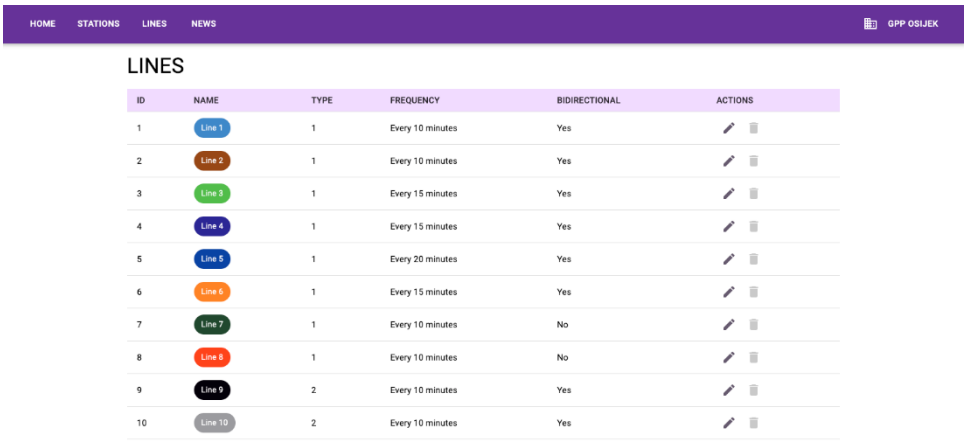


Sl. 5.8. Sučelje za upravljanje vozilima

Slično sučelje korišteno je i za tvrtku koja se bavi javnim prijevozom. Na slici 5.9. tablica je autobusnih i tramvajskih stanica gdje administrator može uređivati njihove detalje ili ih maknuti iz sustava, dok je na slici 5.10. tablica autobusnih i tramvajskih linija. Ostale tvrtke ne mogu pristupiti ovome sučelju.

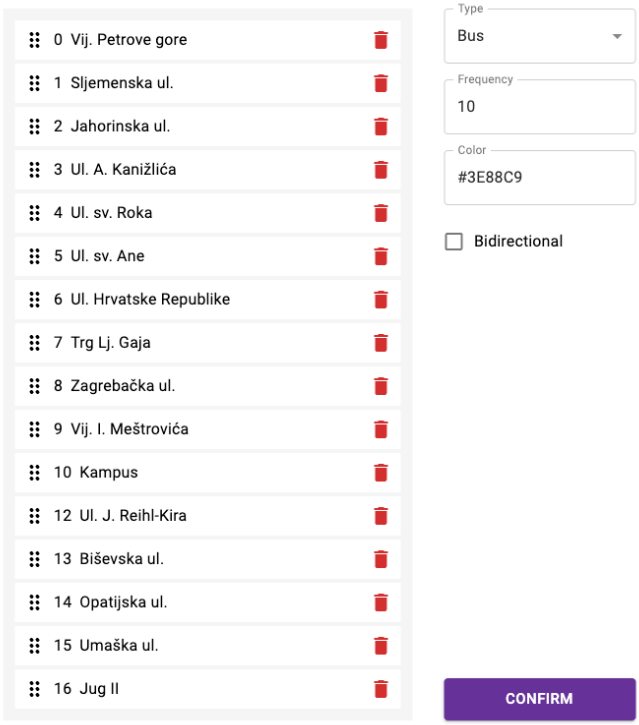


Sl. 5.9. Sučelje za upravljanje autobusnim i tramvajskim stanicama



Sl. 5.10. Tablica autobusnih i tramvajskih linija

Za uređivanje linija administratoru javnog prijevoza je dano posebno sučelje (slika 5.11.) – modal kojim ima mogućnost mijenjati redoslijed stanica unutar linije te obrisati ili dodati stanicu, kao i promijeniti druge detalje linije.



Sl. 5.11. Modal za uređivanje detalja autobusne ili tramvajske linije

Slično sučelje, odnosno kombinacija tablica i modala, korištena su i za upravljanje vijestima, a njemu mogu pristupiti sve tvrtke u sustavu.

6. ZAKLJUČAK

Cilj ovog diplomskog rada bio je izraditi softversko rješenje koje podupire stare i nove oblike mobilnosti, fokusirajući se na obnovljive izvore energije i javni prijevoz, kao odgovor na procese urbanizacije i njihove štetne utjecaje na promet u gradovima. Radi boljeg upoznavanja s problemima koje je urbanizacija stvorila prometu, u uvodu su analizirani njeni utjecaji i prilagodbe javnih institucija te privatnih tvrtki na izazove povećane gužve, zagađenja, i potrebe za održivim načinima prijevoza u gradskim sredinama. U sklopu toga, proveden je polustrukturirani intervjuu s dogradonačelnikom Grada Osijeka dr. sc. Draganom Vulinom, kako bi se dobio bolji uvod u funkcioniranje javnih ustanova Republike Hrvatske, a zatim su analizirane globalno popularne aplikacije Uber i Bolt, koje su ovome radu služile kao referentna točka.

Za izradu rješenja korištene su najsuvremenije tehnologije temeljene na programskom jeziku JavaScript – NestJS, React i React Native – te univerzalne programerske paradigme poput REST API-ja i jezika SQL za upravljanje bazom podataka. Prvo su objašnjene njihove glavne značajke i načini korištenja, a zatim i njihova primjena na zadatak rada. Rezultat rada je softversko rješenje koje uključuje: NestJS poslužiteljsku aplikaciju koja upravlja MySQL bazom podataka, React Native klijentsku mobilnu aplikaciju za iOS i Android operacijske sustave, za korisnike usluga vožnje električnih automobila, bicikala, romobila i javnog prijevoza, te React klijentsku web aplikaciju koja služi tvrtkama za upravljanje uslugama nuđenim korisnicima mobilne aplikacije. U završnom dijelu rada opisano je korisničko sučelje klijentskih aplikacije te njihova upotreba.

Dodatno, na rješenje rada moguće je nadograditi funkcionalnosti poput implementacije plaćanja usluga unutar mobilne aplikacije, dublje statističke analize korištenja usluga za tvrtke, i mnoge druge. Međutim, važno je napomenuti kako ovo softversko rješenje ne može stajati samo, već zahtjeva kompleksnu hardversku podršku. Uređaji spomenuti kroz rad, odnosno električni automobili, bicikli i romobili te vozila javnog prijevoza gradova, morali bi biti prilagođeni kako bi komunicirali s danim aplikacijama. Time bi se rješenje rada razvilo u zaokružen proizvod spreman za upotrebu.

LITERATURA

[1] Ujedinjeni narodi, *World Urbanization Prospects, The 2007 Revision*, 2008.

[2] Ritchie, H., Roser, M., *Urbanization, Our Word in Data*, dostupno na: https://ourworldindata.org/urbanization, [posjećeno 1.9.2024.]

[3] Europsko vijeće, *Rast cijena energije u 2021.*, 2022., dostupno na: https://www.consilium.europa.eu/hr/infographics/energy-prices-2021/, [posjećeno 1.9.2024.]

[4] Eurostat, *EU people on the move: changes in a decade*, 2023., dostupno na: https://ec.europa.eu/eurostat/web/products-eurostat-news, [posjećeno 1.9.2024.]

[5] Zlatar, J., *Zagreb, Cities*, Elsevier, vol. 49, str. 144-155, 2014.,

[6] Brčić, D., Šimunović, Lj., Slavuj, M., *Upravljanje prijevoznom potražnjom u gradovima*, Priručnik, Fakultet prometnih znanosti Zagreb, 2016.

[7] Nieuwenhuijsen, M.J., *Urban and transport planning, environmental exposures and health-new concepts, methods and tools to improve health in cities*, Environmental Health, 2016., dostupno na: https://ehjournal.biomedcentral.com, [posjećeno 10.9.2024.]

[8] Europska komisija, *CO_2 emission performance standards for cars and vans*, dostupno na: https://climate.ec.europa.eu, [posjećeno 10.9.2024.]

[9] Grad Osijek, *E-mobilnost grada Osijeka*, 2019., dostupno na: https://www.osijek.hr/e-mobilnost-grada-osijeka/, [posjećeno 16.9.2024.]

[10] Web stranica tvrtke Uber, dostupno na: https://www.uber.com, [posjećeno 1.9.2024.]

[11] Web stranica tvrtke Bolt, dostupno na: https://bolt.eu/hr-hr/, [posjećeno 1.9.2024.]

[12] Web stranica IP4MaaS projekta, dostupno na: https://www.ip4maas.eu, [posjećeno 1.9.2024.]

[13] Fielding, R. T., *Architectural Styles and the Design of Network-based Software Architectures*, 2000.

[14] Gourley, D., Totty, B., *HTTP: The Definitive Guide*, 2002.

[15] Codecademy, *What is REST?*, dostupno na: https://www.codecademy.com/article/what-is-rest, [posjećeno 14.9.2024.]

[16] Statista, *Most used programming languages among developers worldwide as of 2024*, 2024., dostupno na: https://www.statista.com/, [posjećeno 12.8.2024.]

[17] JavaScript, Mozilla Foundation, dostupno na: https://developer.mozilla.org/en-US/docs/Web/JavaScript, [posjećeno 12.8.2024.]

[18] Web stranica organizacije ECMA International, dostupno na: https://ecma-international.org/, [posjećeno 12.8.2024.]

[19] TypeScript dokumentacija, Microsoft, dostupno na: https://www.typescriptlang.org/docs/, [posjećeno 12.8.2024.]

[20] Web stranica razvojnog okvira Node.js, dostupno na: https://nodejs.org/en, [posjećeno 12.8.2024.]

[21] Web stranica razvojnog okvira NestJS, dostupno na: https://nestjs.com/, [posjećeno 12.8.2024.]

[22] Widom, J., Garcia-Molina H., Ullman J., *Database Systems: The Complete Book*, 2001.

[23] TypeORM dokumentacija, dostupno na: https://typeorm.io/, [posjećeno 12.8.2024.]

[24] Duckett, J., *HTML and CSS: Design and Build Websites*, 2011.

[25] Web stranica razvojnog okvira React, Meta, dostupno na: https://react.dev/, [posjećeno 12.8.2024.]

[26] Web stranica razvojnog okvira React Native, Meta, dostupno na: https://reactnative.dev/, [posjećeno 12.8.2024.]

[27] Web stranica alata i razvojnog poslužitelja Vite, dostupno na: https://vitejs.dev/, [posjećeno 10.9.2024.]

[28] Web stranica biblioteke TanStack Query, dostupno na: https://tanstack.com/query/latest, [posjećeno 14.8.2024.]

[29] Web stranica biblioteke Zustand, dostupno na: https://zustand.docs.pmnd.rs, [posjećeno 14.8.2024.]

[30] Web stranica biblioteke MaterialUI, Google, dostupno na: https://mui.com/, [posjećeno 14.8.2024.]

[31] Web stranica biblioteke React Navigation, dostupno na: https://reactnavigation.org/, [posjećeno 14.8.2024.]

[32] GitHub repozitorij biblioteke React Native Maps, dostupno na: https://github.com/react-native-maps/react-native-maps, [posjećeno 15.8.2024.]

[33] GitHub repozitorij biblioteke React Native Vision Camera, dostupno na: https://github.com/mrousavy/react-native-vision-camera, [posjećeno 15.8.2024.]

SAŽETAK

Nagla urbanizacija diljem svijeta stvorila je gradovima niz problema vezanih uz promet poput zagušenosti na određenim prometnim linijama u određeno doba dana, prevelike buke, onečišćenja okoliša te prevelikih troškova energenata. Tehnički, ekonomski i pravni aspekti urbanizacije i prometa u Republici Hrvatskoj i svijetu čine rješenja rijetkima i skupima. Kao rezultat rada i prijedlog za poboljšanje sustava mobilnosti izrađeno je softversko rješenje u obliku poslužiteljske aplikacije te dvije klijentske aplikacije, fokusirajući se na javni prijevoz i vozila pogonjena obnovljivim izvorima energije. Temelj sve tri aplikacije je programski jezik JavaScript te razvojni okviri i biblioteke njegovog okruženja kao što su NestJS, React i React Native. NestJS poslužiteljska aplikacija središte je sustava i služi za upravljanje resursima koji su pohranjeni u SQL bazu podataka. React klijentska web aplikacija alat je za administratore javnog prijevoza i tvrtke koje nude usluge mobilnosti, dok React Native mobilna aplikacija nudi korisnicima sučelje za lakše, efikasnije i održivije kretanje po urbanim sredinama.

Ključne riječi: JavaScript, mobilnost, NestJS, React, React Native

ABSTRACT

Support for Old and New Forms of Personal Mobility through Information and Communication Technology

Rapid urbanization worldwide has caused many traffic-related issues in cities, such as congestion on certain traffic routes at specific times of the day, excessive noise, environmental pollution, and high costs of energy sources. The technical, economic, and legal aspects of urbanization and traffic in the Republic of Croatia and globally result in scarce and costly solutions. As a result of this thesis and as a proposal for improving the mobility system, a software solution has been developed in the form of a server application and two client applications, focusing on public transport and vehicles powered by renewable energy sources. The foundation of all three applications is the JavaScript programming language and the frameworks and libraries from its environment, such as NestJS, React, and React Native. The NestJS server application is the central hub for managing resources stored in an SQL database. The React client web application is a tool for public transport administrators and companies offering mobility services, while the React Native mobile application provides users with an interface for simpler, more efficient, and sustainable traveling within urban environments.

Keywords: JavaScript, mobility, NestJS, React, React Native

ŽIVOTOPIS

Luka Pivk rođen je 10.10.1998. u Osijeku. Nakon pohađanja Osnovne škole J. A. Ćolnića u Đakovu, upisuje opću gimnaziju A. G. Matoša. 2017. godine završava gimnaziju te upisuje prijediplomski studij Računarstva na Fakultetu elektrotehnike, računarstva i informacijskih tehnologija u Osijeku. Završetkom prijediplomskog studija i stjecanjem akademskog naziva sveučilišnog prvostupnika računarstva, upisuje diplomski studij Računarstvo, modul Informacijske i podatkovne znanosti, na istom fakultetu. Tijekom studija počinje se profesionalno baviti razvojem softvera radeći u nekoliko domaćih i stranih poduzeća, fokusirajući se na programski jezik JavaScript i brojne razvojne okvire njegova okruženja.

PRILOZI

GitHub repozitorij aplikacija danim u radu, dostupno na: https://github.com/lpivk-stilico/eos