SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

ZAVRŠNI RAD br. 1246

VIZUALIZACIJA PROMETA JAVNOG PRIJEVOZA U STVARNOM VREMENU UPORABOM FORMATA GTFS

Luka Miličević

Zagreb, lipanj, 2024.

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

Zagreb, 4. ožujka 2024.

ZAVRŠNI ZADATAK br. 1246

Pristupnik:	**Luka Miličević (0036543289)**
Studij:	Elektrotehnika i informacijska tehnologija i Računarstvo
Modul:	Računarstvo
Mentorica:	izv. prof. dr. sc. Ivana Bosnić

Zadatak: **Vizualizacija prometa javnog prijevoza u stvarnom vremenu uporabom formata GTFS**

Opis zadatka:

Proučiti problematiku pružanja informacija o javnom prijevozu te dostupne norme za razmjenu takvih podataka. Posebnu pažnju posvetiti formatima namijenjenima slanju informacija u stvarnom vremenu. Istražiti normu General Transit Feed Specification (GTFS), a posebno podskup GTFS Realtime (GTFS RT). Za odabrani grad koji pruža usluge javnog prijevoza i daje informacije u formatu GTFS RT potrebno je razviti web-aplikaciju za vizualizaciju trenutnog položaja vozila javnog prijevoza na karti. S obzirom na moguća ograničenja izvora podataka i njihovo slanje u "skoro stvarnom vremenu" (near real-time), potrebno je ostvariti jednostavno predviđanje i animaciju kretanja vozila do sljedeće stanice. Aplikacija treba omogućiti korisniku odabir pojedine prijevozne linije koju želi pratiti i pružiti mu dodatne informacije, poput vremena dolaska vozila na određenu postaju, kašnjenja i slično. S obzirom na način uporabe, aplikacija treba imati responzivni dizajn uz korisničko sučelje prilagođeno i uređajima manjih zaslona.

Rok za predaju rada: 14. lipnja 2024.

Na samom početku želim izraziti svoju duboku zahvalnost svojoj obitelji i prijateljima za neizmjernu podršku i razumijevanje tijekom mog obrazovanja. Posebnu zahvalnost dugujem svojoj mentorici, izv. prof. dr. sc. Ivani Bosnić, čije me stručno vodstvo i savjeti pratilo kroz cijeli proces izrade ovog rada. Vaša podrška bila je ključna za moj akademski napredak i uspjeh.

Sadržaj

1. Uvod

U velikim urbanim gradovima poput Zagreba, javni gradski prijevoz predstavlja neizostavan dio svakodnevnog života. On je temeljni stup mobilnosti, pružajući vitalnu infrastrukturu za povezivanje građana i omogućujući lakše i efikasnije kretanje unutar grada. Velika većina građana svakodnevno koristi javni prijevoz te s obzirom na važnost vremena u užurbanom ritmu gradskog života, dobra organizacija i planiranje putovanja su ključni, jer svi bi htjeli 15 minuta koje potroše na čekanju prijevoza iskoristiti na drukčiji način. Upravo s tim ciljem, ideja završnog rada je stvoriti web aplikaciju, koja pruža stvarno praćenje tramvaja u realnom vremenu, olakšavajući korisnicima efikasno planiranje svojih putovanja, osiguravajući im precizne informacije o dolasku tramvaja i informacijama o pojedinim gradskim linijama prijevoza. Kroz ovaj zadatak, istražit ćemo tehničke izazove s kojima se susrećemo u razvoju takve web aplikacije. Dublje ćemo istražiti ključne elemente koji oblikuju temelje naše aplikacije poput analize API-ja pruženog od strane Zagrebačkog Električnog Tramvaja, proučavanje otvorenih standarda GTFS i GTFS realtime, te detaljniju analizu strukture web-aplikacije, od baze podataka PostgreSQL, poslužiteljskog dijela ostvarenog s node.js i Express.js, do klijentskog dijela implementiranog s React.js i bibliotekama poput Leaflet.js

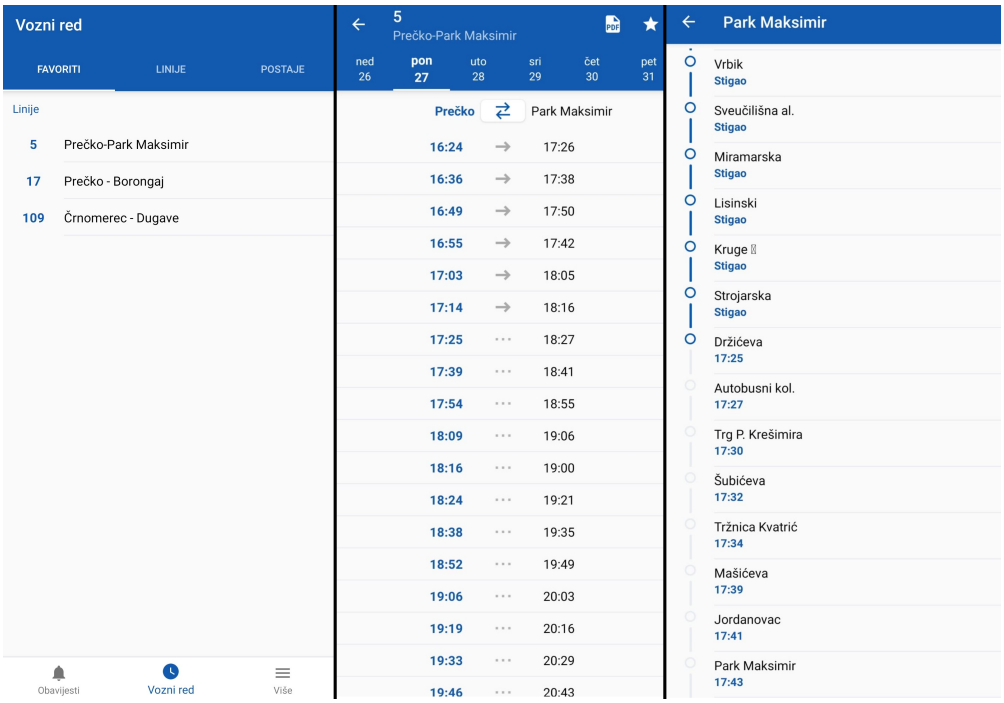
2. Glavni dio

2.1. Zagrebački Električni Tramvaj

Zagrebački električni tramvaj, poznatiji kao ZET, je trgovačko društvo koje je u direktnom vlasništvu Grada Zagreba, te ima ključnu ulogu u organizaciji javnog gradskog prijevoza u Zagrebu. Posluje od 1910. godine i svakodnevno pruža prijevozne usluge za gotovo milijun ljudi pomoću tramvaja, autobusa, uspinjače i specijalnih prijevoza [1]. Nedavno je ZET otvorio svoje podatke javnosti pod Otvorenom dozvolom Republike Hrvatske. Ovi podaci obuhvaćaju GTFS static i GTFS realtime podatke 2.2., koji su trenutno parcijalno implementirani te su dostupni na službenoj stranici ZET-a [2]. Ti podaci će biti temelj aplikacije i kroz njih ćemo detaljnije proći u narednim poglavljima 2.2.3.

Slične aplikacije - ZET info

Jedna popularna aplikacija koja koristi iste ZET-ove podatke je ZET info [3]. Koja primarno koristi GTFS static podatke za prikaz rasporeda tramvaja ZET-a. Ona nije službena ZET-ova aplikacija te dostupna je preko Apple App Store-a i Google play-a. Vrlo je zgodna za imati pri ruci za pregledavanje rasporeda vožnji, ali bi aplikacijom ovog završnog rada htjeli ostvariti i upotrebu stvarnih (realtime) podatatak za tramvaje.



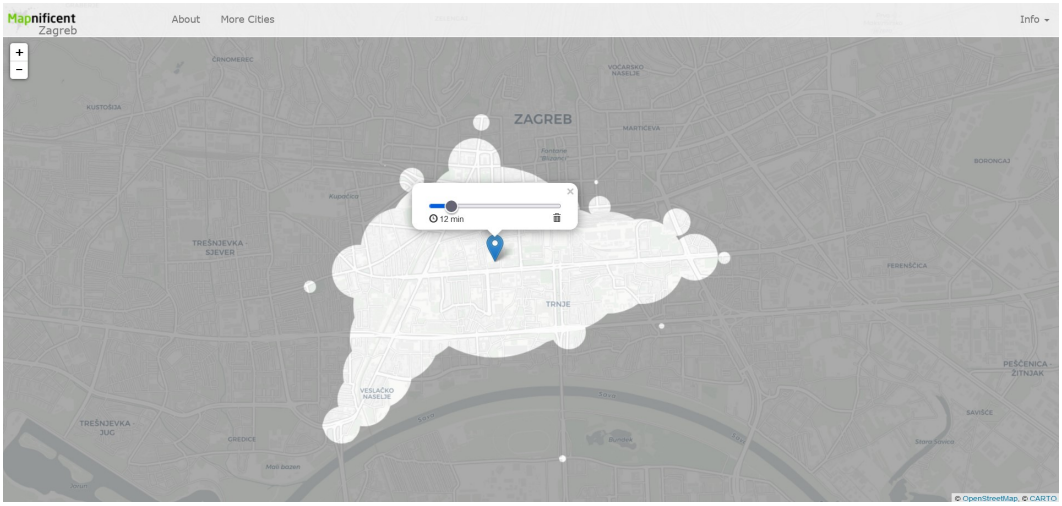
Slika 2.1. ZET info

2.2. General Transit Feed Specification

General Transit Feed Specification, kraće GTFS je otvoreni standardizirani format za rasporede javnog prijevoza i pripadajuće geografske informacije. Široko se koristi diljem svijeta te ga podržavaju više od 10 tisuća agencija javnog prijevoza u preko 100 zemalja. Neke od platformi koje koriste GTFS su Google Maps, Apple Maps, Moovit, OpenStreetMap i druge [4]. GTFS je započeo 2005. godine kao ideja za sporedni projekt Googleovog zaposlenika Chris Harrelsona te je danas *De facto* standard industrije. GTFS se sastoji od dva glavna dijela GTFS schedule (static) i GTFS realtime.

Činjenica da je GTFS otvoreni standard omogućuje da agencije za prijevoz informacije učine dostupnima pomoću bilo kojeg od mnogih alata koji već podržavaju GTFS i korisnicima da preuzmu i obrađuju te informacije sa proizvoljnom aplikacijom koja podržava GTFS ili ako žele razviju svoje programe za to. Otvoreni standardi dovode do stvaranja podataka koji se mogu lako dijeliti i potiču daljnji razvitak.

Jedan zanimljiv primjer male aplikacije koja koristi GTFS podatke da prikaže područja na karti do kojih možete doći javnim prijevozom u određenom vremenu je otvoreni projekt Mapnificent koji je javno dostupan na Github-u [5]. Na slici 2.2. je prikazano dokle je moguće doći javnim prijevozom od zgrade FER-a unutar 12 minuta.



Slika 2.2. Mapnificent Zagreb

2.2.1. GTFS static

GTFS static ili GTFS schedule je kolekcija od barem 6 osnovnih, do 26 CSV (comma-separated values) datoteka s ekstenzijom .txt zapakiranih unutar jedne komprimirane .zip datoteke. Te datoteke sadrže informacije o rutama, rasporedima, stanicama, i ostalim raznim informacijama o prijevozu [6]. Sam format datoteka koji je CSV, pruža otvorenost i jednostavnost rukovanja jer mnogo programskim alata i jezika ima metode za efektivnu obradu CSV što čini implementaciju strojnog čitanja lakoćom. Važno za napomenuti da je način kodiranja datoteka imperativan, obavezuje se korištenje UTF-8 kodiranja (podržan je i UTF-8 BOM).

Na dijagramu 2.3. prikazan je jedan primjer važeće strukture GTFS static modela kojeg prikladno prikazuju datoteke u ER (Entity–relationship) modelu. Zelenom bojom su označene neophodne datoteke, a plavom par dodatnih opcionalnih datoteka te za svaku od njih struktura i međusobne relacije.

Slika 2.3. GTFS-static model [7]

Neophodne datoteke

Moraju biti definirane da bi zadovoljili GTFS standard. One spojene zajedno pomoću pojedinih identifikatora čine cjelovitu sliku mreže javnog prijevoza

- **Agency.txt** - Opće informacije o prijevoznoj agenciji, uključujući njezino ime, web stranicu, i vremensku zonu.
- **Routes.txt** - Definira različite linije javnog prijevoza. Svaka ruta obuhvaća informacije o identifikatorima ruta, tipovima vozila i drugim detaljima.
- **Trips.txt** - Informacije o specifičnim putovanjima na određenim rutama, uključujući vremena polaska i druge povezane podatke.
- **Stop_times.txt** - Informacije o vremenima dolaska i odlaska vozila na svakoj stanici tijekom svakog putovanja.
- **Stops.txt** - Informacije o stanicama, uključujući identifikatore, nazive i geografske koordinate.
- **Calendar.txt** - Informacije o tjednom rasporedu putovanja.

Opcionalne datoteke

Pored neophodnih datoteka korisno je imati i dodatne opcionalne datoteke poput:

- **Shapes.txt** - Informacije o geografskim koordinatama rute i omogućuje lijepo iscrtavanje rute na karti.
- **Transfers.txt** - Informacije o povezivanju više ruta kako bi se olakšala mogućnost presjedanja i kombiniranja prijevoznih linija.
- **Calendar_dates.txt** - Omogućuje definiranje iznimaka u redovnom rasporedu kada usluga radi ili ne radi na pojedinoj liniji zbog smetnji ili radova, ili pak posebnih prijevoza.

2.2.2. GTFS realtime

GTFS realtime ili GTFS-rt je ekstenzija GTFS-a koja je nastala 2011. godine te je također *de facto* industrijski standard za dijeljenje stvarnih podataka o prijevozu. Jer je GTFS otvoreni standard, realtime se razvio s ciljem maksimalne interoperabilnosti i lakoće korištenja da bi olakšali programerima razvoj aplikacija i lakoću dijeljenja informacija o prijevozu između agencija. GTFS-rt sadrži informacije u stvarnom vremenu o lokacijama vozila, predviđenim vremenima dolaska te obavijestima o promjenama ruta i otkazivanjima putem web poslužitelja (putem nekog API) koji koristi protocol buffers (protobuf, poglavlje 2.3.). Podaci o stvarnoj lokaciji stvaraju se neprekidno od strane agencije sustavom za automatsko praćenje vozila dok se vremena dolaska na odredište najčešće izračunavaju pomoću modela strojnog učenja koji analiziraju povijesne podatke o položaju i voznom redu te daje očekivana vremena. Upravo jer se podaci neprestano kreiraju koriste se protocol buffers koji jako brzo i efektivno rade binarnu serijalizaciju podataka u male pakete koji se lako šalju [8].

Glavni "objekt" GTFS realtime feed-a (stream) je *FeedMessage* koji sadrži *FeedHeader* i *FeedEntity*. *FeedHeader* sadrži osnovne informacije o podacima (metapodatke) poput verzije GTFS-a, "*incrementiality*" koji opisuje šalje li se cijeli skup podataka (*dataset*) ili samo razlika od posljednje verzije te vremenska oznaka (*timestamp*) kada su podaci generirani. *FeedEntity* sadrži jedan ili više entiteta koji sadrže najnovije informacije o stanju puta, pozicijama vozila, promjenama na rutama i upozorenjima. Neke od vrsta *FeedEntity*-a su:

- **tripUpdate** - predstavlja informacije jednog putovanja te sadrži objekte *trip* (osnovne informacije o ruti/putu), *stopTimeUpdate*, *vehicle* (informacije o položaju vozila npr. geografske koordinate)
- **stopTimeUpdate** - nalazi se unutar *tripUpdate* i on sadrži informacije o dolascima/odlascima sa stanica te kašnjenja
- **alert** - sadrži upozorenja od nesreća, radova na cesti i njihovi obilazaka, vremenskih upozorenja i drugih raznih iznimaka

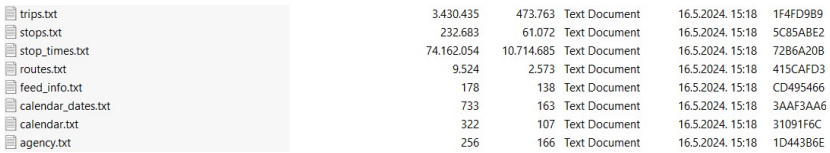
Primjer GTFS-rt podataka se može vidjeti u sljedećem poglavlju na slikama 2.6. i 2.7.

2.2.3. ZET GTFS podaci

Kao što je već rečeno ZET-ovi podaci su dostupni na službenoj stranici svima pod Otvorenom dozvolom Republike Hrvatske [2]. Na slici 2.4. je prikazana ta stranica te na njoj se mogu pronaći linkovi za GTFS static i GTFS realtime podatke.

Prvo ćemo prokomentirati ZET-ove GTFS static podatke, a potom i GTFS realtime podatke. Kada se podaci dohvate sa stranice, dobivenu arhivu treba otvoriti s odgovarajućim alatom za pregled komprimiranih arhiva. Kada se to napravi dobivamo datoteke sa slike 2.5.

Slika 2.4. ZET stranica za GTFS podatke



Slika 2.5. ZET GTFS static podaci

Kao što vidimo tu su sve neophodne datoteke (*trips.txt*, *stops.txt*, *stop_times.txt*, *routes.txt*, *calendar.txt* i *agency.txt*) te par dodatnih datoteka *feed_info.txt* i *calendar_dates.txt* koje nisu pretežito korisne za implementaciju aplikacije. Opcionalna datoteka *feed_info.txt* sadržava opće informacije o GTFS *feed*-u, odnosno informacije poput naziva agencije koja je stvorila *feed*, vremena kada je *feed* generira, verzije GTFS standarda koja se koristi, i slično.

Pobliže ćemo pogledati datoteke *trips.txt*, *stops.txt*, *stop_times.txt* i *routes.txt* jer one sadrže temeljne podatke za rad aplikacije.

trips.txt

route_id	service_id	trip_id	trip_headsign	trip_short_name	direction_id	block_id	shape_id
268	"0_21"	"0_21_26820_268_10001"	"V. Gorica"		**0**	26820	
8	"0_21"	"0_21_801_8_10002"	"Grač. dolje"		1	801	
7	"0_21"	"0_21_701_7_10012"	"Arena Zagreb"		1	701	

Tablica 2.1. Izgled trips.txt

Bitne informacije u *trips.txt* su:

- **route_id** - identifikator rute, odnosno broj linije (npr. Kod 17 Borongaj, to je 17)
- **trip_id** - identifikator putovanja na pojedinoj ruti (svaka ruta ima više tramvaja koji više puta dnevno putuju na toj ruti)
- **trip_short_name** i **direction_id** - ime smjera tramvaja i binarni broj koji određuje smjer (npr. Borongaj)

Također možemo primijetiti da u datoteci nedostaje *shape_id* koji se koristi kao ključ za relaciju *trips.txt* i *shapes.txt* koja kao što je rečeno sadrži detaljne koordinate putanje pojedine rute, s kojom bi se na karti s lakoćom mogao iscrtati put glatkom neisprekidanom krivuljom. Ali u ZET-ovim podacima se ne može pronaći niti *shape_id* niti *shapes.txt* te kao obilazak tom problemu uzete su koordinate svih stanica na pojedinoj ruti koje su spojene ravnim linijom. Izgled te linije nije idealan, ali je rješenje približno. Postoje i određene *routing* komponente, ali i takvo rješenje ima svoje nedostatke, ali o tom više u poglavlju o implementaciji klijentske strane aplikacije 2.4.4.

stops.txt

stop_id	stop_code	stop_name	stop_desc	stop_lat	stop_lon	zone_id	stop_url	location_type	parent_station
"98"		"Črnomerec"		45.815175	15.934464			1	
"100"		"Sveti Duh"		45.813468	15.942297			1	
"106_1"	"1"	"Trg J. Jelačića"		45.812906	15.977312			0	106
"106"		"Trg J. Jelačića"		45.813038	15.976928			1	

Tablica 2.2. Izgled stops.txt

Bitne informacije u *stops.txt* su:

- **stop_id** - identifikator pojedine stanice

- **stop_name** - naziv stanice
- **stop_lat** i **stop_lon** - koordinate stanice

stop_times.txt

trip_id	arrival_time	departure_time	stop_id	stop_sequence	stop_headsign	pickup_type	drop_off_type	shape_dist_traveled
"0_21_26820_268_10001"	"00:30:00"	"00:30:00"	"110_82"	1	"V. Gorica"			
"0_21_26820_268_10001"	"00:32:30"	"00:32:30"	"238_24"	2	"V. Gorica"			
"0_21_26820_268_10001"	"00:33:40"	"00:33:40"	"450_24"	3	"V. Gorica"			
"0_21_26820_268_10001"	"00:35:20"	"00:35:20"	"1085_24"	4	"V. Gorica"			

Tablica 2.3. Izgled stop_times.txt

Bitne informacije u *stop_times.txt* su:

- **trip_id** - identifikator putovanja na pojedinoj ruti
- **arrival_time** i **departure_time** - vrijeme dolaska i odlaska
- **stop_id** - identifikator pojedine stanice
- **stop_sequence** - redni broj stanice na putu

Ovo je najveća datoteka (oko 70MB, ~1000000 linija) koja za svako pojedino putovanje (*trip_id*) sadrži informacije o svim stanicama na putu, očekivanom vremenu dolaska i odlaska sa stanice i redoslijedu stanica. Ona će biti ključna za iscrtavanje rute i predviđanju lokacije tramvaja.

routes.txt

route_id	agency_id	route_short_name	route_long_name	route_desc	route_type	route_url	route_color	route_text_color
1	0	"1"	"Zap.kol. - Borongaj"		0		"ffffff"	"000000"
2	0	"2"	"Črnomerec - Savišće"		0		"ffffff"	"000000"
3	0	"3"	"Ljubljanica -Savišće"		0		"ffffff"	"000000"
4	0	"4"	"Savski most - Dubec"		0		"ffffff"	"000000"

Tablica 2.4. Izgled routes.txt

Ovdje nam je jedina nova bitna informacija **route_long_name** koja nam daje puno ime rute (početno mjesto i krajnje mjesto).
To bi bile najbitnije GTFS static informacije koje su upotrijebljene za izradu aplikacije, koje su od posebne značajnosti za izradu baze podataka koja sprema te odvojene datoteke kao tablice i spaja ih s odgovarajućim ključevima i time omogućuje lako i brzo pretraživanje te formuliranje odgovora 2.4.1.

ZET realtime podaci

Na ZET-ovoj stranici nalazi se link za GTFS realtime podatke 2.4. koji kada se skinu su binarna datoteka u *.proto* formatu. *Protobuf* je opisan u narednom poglavlju 2.3. Ovdje ćemo samo brzo proći kroz konačne podatke kada se obavi deserijalizacija.

Na slici 2.6. se nalazi jedan mali isječak od inače jako velike poruke (obično sadrže oko 500 - 600 entiteta), ali radi jednostavnosti je prikazan samo jedan.

Na početku svake poruke nalazi se *header* te možemo primijetiti da se koristi starija verzija GTFS 1.0. Sada je aktualna verzija 2.0 koja ima bolje definiranu *schemu* s više informacija i pruža veću fleksibilnost. Također piše da se svaki put šalju svi podaci "FULL_DATASET" i na kraju je vremenska oznaka generiranja poruke koja je u UNIX formatu.

Zatim polje *entity* koje sadrži entitete (objekte) koji reprezentiraju informacije o jednom vozilu. Svaki od njih ima jedinstveni *ID* i *tripUpdate*. *TripUpdate* se sastoji od osnovnih informacija o putu na kojemu je vozilo, *ID* tog puta, kada je započeo put, je li put prema rasporedu, broj rute kojoj pripada taj put, vremenska oznaka generiranja tih podataka i *stopTimeUpdate* koji sadrži informacije o zadnjoj posjećenoj stanici njen ID, redni broj i vremenska oznaka.

Jedna stvar koju je vrijedno napomenuti da su ZET-ovi podaci prilično osnovni tjst. sadrže najosnovnije informacije, vjerojatni razlog toga je što su ZET-ovi podaci u fazi razvoja i na stranici je navedeno da je ovo prototip. Na primjer jedna jako velika stvar koja bi bila jako korisna jesu *vehicle* podaci (slika 2.7.) koji sadrže točne geografske lokacije vozila. Trenutačno je aplikacija napravljena tako da na karti prikaže zadnje potvrđene pozicije tramvaja koje su zapravo podaci iz *stopTimeUpdate* odnosno lokacije zadnje posjećenih stanica i markeri koji su predikcije položaja tramvaja temeljene na vremenima iz GTFS static podataka. Implementacija tih vehicle podataka bi izbacilo potrebu za predikcijom i puno olakšala direktno prikazivanje točnih lokacija tramvaja.

```
"header": {
  "gtfsRealtimeVersion": "1.0",
  "incrementality": "FULL_DATASET",
  "timestamp": "1702666249"
},
"entity": [
{
  "id": "XNYGO5S0BU",
  "tripUpdate": {
    "trip": {
      "tripId": "0_1_11504_115_10447",
      "startDate": "20231215",
      "scheduleRelationship": "SCHEDULED
        ",
      "routeId": "115"
    },
    "stopTimeUpdate": [
    {
      "stopSequence": 12,
      "arrival": {
        "delay": 145,
        "time": "1702666205"
      },
      "stopId": "1634_22",
      "scheduleRelationship": "SCHEDULED
        "
    }
    ],
    "timestamp": "1702666189"
  }
}
]
```


Slika 2.6. Dio ZET-ovog GTFS-rt feed-a


```
      "vehicle": {
        "position": {
          "latitude":
             45.8150,
          "longitude":
             15.9819,
          "bearing": 120,
          "odometer":
             15000.5,
          "speed": 30.5
        }
      }
```


Slika 2.7. Primjer vehicle podataka

2.3. Protocol buffers

Protocol Buffers su jezično i platformski neutralni, proširivi mehanizmi za serijalizaciju strukturiranih podataka. Razvijeni od strane Googlea, prvotno su bili namijenjeni internoj uporabi, ali su kasnije postali dostupni pod otvorenom licencom [9]. Glavni cilj protocol buffera je pružiti jednostavnost i visoke performanse, te su posebno dizajnirani kako bi bili manji i brži od XML i JSON formata.

Protobuf koristi .proto datoteke za definiranje strukture podataka koji koriste posebnu sintaksu. Trenutno su podržane verzije proto2 i proto3 u kojima se s ključnom riječi *message* definira struktura podataka poruke. Na slici 2.8. je mali primjer jedne definicije poruke *Person* koja sadrži tri polja za podatke *id, name i email*. Svaka od njih mora imati tip podatka i vrijednost koja služi za identifikaciju polja kod binarne serijalizacije. Neki tipovi podataka koji su podržani su int32, int64, float, double, bool, string, enum i ugrađene poruke.

```
1  syntax = "proto3";
2  message Person{
3    int32 id = 1;
4    string name = 2;
5    string email= 3;
6  }
```

Slika 2.8. Primjer .proto definicije

Kada smo definirali strukturu svoje poruke potrebno je iskoristiti proto compiler (*protoc*) da bi generirali kôd koji se koristi za serijalizaciju i deserializaciju podataka u odabranom programskom jeziku (C++, Java, Python, Go, JavaScript i mnogo drugih). Za primjer sa slike 2.8. i za programski jezik Python, *protoc* compiler generira modul *person_pb2* koji sadrži potrebne metode za kreiranje objekata te strukture, serijalizaciju i deserijalizaciju tih podataka.

```
import person_pb2

person = person_pb2.Person()
person.id = 123
person.name = "John Doe"
person.email = "johndoe@example.com"

# Serijalizacija
serialized_data = person.SerializeToString()

# Deserijalizacija
new_person = person_pb2.Person()
new_person.ParseFromString(serialized_data)
```

Slika 2.9. Python kôd s generiranim proto modulom

Neke prednosti korištenja protocol buffer-a za razliku od tradicionalnijih opcija poput XML-a i JSON-a su:

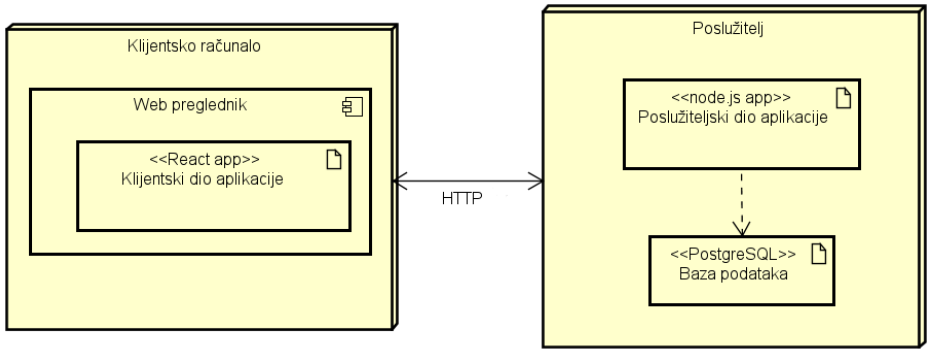
- Protobuf koristi binarni format za serijalizaciju, koji je mnogo kompaktniji i brži za parsiranje.
- Dizajniran s *backward* i *forwards compatibility* na umu
- Međujezična kompatibilnost (*Cross-Language compatibility*)
- Imaju validaciju schema koji striktno čuvaju strukturu podataka

2.4. Arhitektura sustava

Sustav se temelji na modelu Klijent-Poslužitelj. Taj model je danas jedan od najčešće korištenih arhitektura za raspodijeljene sustave. On se sastoji od dva dijela, poslužitelja koji nudi uslugu i klijenta koji traži uslugu. Klijent će biti korisnikovo računalo koje pomoću web preglednika koristi klijentski dio aplikacije (React frontend) te putem HTTP komunikacijskog kanala komunicira s poslužiteljskim dijelom aplikacije (node.js i PostgreSQL).

Klijentski dio aplikacije je izrađen pomoću React.js, popularnog JavaScript okvir za izgradnju korisničkih sučelja, primarno korišten za jednostranične aplikacije (Single-page Application, SPA).

Poslužiteljski dio aplikacije izrađen je pomoću node.js, koji je također JavaScript okvir, ali za izgradnju server-side aplikacija. Aplikacija će također komunicirati i s relacijskom bazom podataka PostgreSQL. Poslužitelj će obrađivati klijentske zahtjeve i pružati odgovarajuće odgovore nazad.



Slika 2.10. Arhitektura sustava

2.4.1. Baza podataka

Baza podataka je izvedena pomoću PostgreSQL-a. PostgreSQL je besplatan i otvoreni, objektno-relacijski sustav za upravljanje bazama podataka. Temelji se na SQL jeziku i nudi brojna proširenja koji olakšavaju pohranu i skaliranje, povećavaju sigurnost te rješavaju probleme paralelnosti pristupa. Nastao je 1986. godine na Kalifornijskom sveučilištu u Berkeleyju. [10]

Baza podataka je korištena zbog velike količine podataka iz ZET-ovih statičnih podataka te bi najoptimalniji način upravljanja i pohranom tih podataka bila baza podataka. Glavne uloge baze podataka u aplikaciji su:

- Spremanje ZET-ovih statičnih podataka kao tablice
- Povezivanje tablica statičnih podataka pomoću ključeva u virtualne tablice (*views*)
- Funkcije za brzi pronalazak traženih podataka prema nekom parametru
- Formiranje odgovora

Ubacivanje podataka iz ZET-ovih datoteka u bazu podataka je automatizirano pomoću python skripte (opisana u narednom poglavlju 2.4.2.). Datoteke koje smo ubacili su *trips.txt, stops.txt, routes.txt i stop_times.txt*. One su ubačene kao najobičnije tablice gdje stupce pojedine tablice definira *header* te datoteke (prisjetimo se one su zapravo CSV datoteke! 2.2.3.).

Spajanje tih tablica je ostvareno virtualnim tablicama tkzv. pogledi (*views*) koji objedinjuju par tablica u jednu i ostavljaju samo navedene stupce. Sadržaj virtualne tablice dinamički se određuje u trenutku obavljanja operacije nad virtualnom tablicom te se ponovo izvode svaki put kada ih se pozove. Time se izbjegava problem "zastarijevanja" podataka.
Neki od pogleda su *stopsjson, tripstops i triptime*. *Stopsjson* vraća polje *key-value* vrijednosti, gdje je ključ *stop_id* i vrijednost objekt s imenom i pozicijom stanice. Rezultat vraća kao JSON objekt.

```
1    SELECT jsonb_object_agg(stop_id::text,
        jsonb_build_object('stop_name', stop_name, '
        stop_lat', stop_lat, 'stop_lon', stop_lon)) AS
        stops
2    FROM stops;
```

Slika 2.11. Kôd pogleda *stopsjson*

Pogled *tripstops* spaja tablice *stop_times* i *stops* pomoću *stop_id* kao ključa. Objedinjuje informacije tih dviju tablica kako bi imali potpune informacije o putu poput redoslijeda stanica, njihovih naziva i lokacija te vremena dolaska i odlaska.

```
SELECT  stop_times.trip_id,
    stops.stop_id,
    stop_times.stop_sequence,
    stops.stop_name,
    stops.stop_lat,
    stops.stop_lon,
    stop_times.arrival_time,
    stop_times.departure_time,
    stop_times.stop_headsign
    FROM stop_times
    JOIN stops USING (stop_id)
    ORDER BY stop_times.stop_sequence;
```

Slika 2.12. Kôd pogleda *tripstops*

Pogled *triptime* filtrira tablicu *tripstops* kako bi izračunali vrijeme potrebno do iduće stanice

```
  SELECT trip_id, stop_sequence,
  COALESCE(lead(arrival_time) OVER () - arrival_time
     , '00:00:00'::interval) AS time_till_next_stop
  FROM tripstops
  ORDER BY stop_sequence;
```

Slika 2.13. Kôd pogleda *triptime*

Možemo direktno pristupati virtualnim tablicama, ali za potrebe aplikacije potrebno je još filtrirati ih prema određenom parametru, zato u bazi podataka imamo i funkcije definirane u jeziku plpgsql. One su:

- *find_route_between_stops(stop_a, stop_b)* - koja prima dva parametra (dvije stanice) i vraća listu ruti koje prolaze kroz te dvije stanice.
- *get_trip_info(trip_id)* - koja prima parametar *trip_id* i vraća list svih stanica puta, nazive i lokacije stanica, vremena dolaska i odlaska te vrijeme potrebno do sljedeće stanice, koristi se za prikaz informacija klikom na marker.

```sql
SELECT jsonb_agg(
  jsonb_build_object(
  'stop_id', stop_id,
  'stop_sequence', stop_sequence,
  'stop_name', stop_name,
  'stop_headsign', stop_headsign,
  'stop_lat', stop_lat,
  'stop_lon', stop_lon,
  'arrival_time', arrival_time,
  'departure_time', departure_time,
  'time_till_next_stop', time_till_next_stop::interval
  )
) INTO trip_info
FROM (
  SELECT *,
  COALESCE(
    LEAD(arrival_time) OVER (ORDER BY stop_sequence) -
       arrival_time, INTERVAL '0 seconds'
  ) AS time_till_next_stop
```

Slika 2.14. Isječak kôda funkcije *get_trip_info(trip_id)*

Navedene funkcije koristi poslužiteljska strana zajedno s parametrom. Primjer korištenja je:

```sql
SELECT find_routes_between_stops('Prisavlje', 'Miramarska')
    AS routes
SELECT get_trip_info('0_21_26820_268_10001') AS trip_info
```

Slika 2.15. Primjer korištenja funkcija

2.4.2. Pomoćna skripte

Za potrebe aplikacije izrađena je jedna pomoćna skripta koju pokreće poslužiteljski dio aplikacije prilikom pokretanja. Ona je zadužena za dohvaćanje najnovijih datoteka sa ZET-ove stranice te je napisana u Pythonu koristeći module *psycopg2, requests i zipfile*.

Pomoću metode *requests.get('https://www.zet.hr/gtfs-scheduled/latest')* sa stranice ZET-a dohvaća najnoviju GTFS static arhivu te potom metodom *extractAll* iz modula *zipfile* raspakirava arhivu i sprema datoteke. Komunikacija s bazom podataka PostgreSQL je ostvarena metodama iz modula *psycopg2*. Prvo je potrebno priložiti vjerodajnice odnosno *username i password* te *host, port i database* (određuju lokaciju baze podataka) za us-

pješno spajanje.

Na slici 2.16. prikazan je isječak kôda pomoću kojeg je ostvareno učitavanje podataka iz datoteka u odgovarajuće tablice. Python funkcije rade u tri koraka, prvo definiramo SQL naredbu zatim šaljemo tu naredbu bazi podataka i na kraju gledamo odgovor je li ispravno izvršena. Funkcija *load_csv_to_postgres(connection, cursor, 'routes', 'routes.txt')* učitava podatke iz *routes.txt* i sprema ih u bazu podataka kao tablica s imenom *routes*, a funkcija `create_route_find_table(connection, cursor)` kreira tablicu *route_find* koju koristi funkcija `find_route_between_stops(stop_a, stop_b)` koja pronalazi rute koje prolaze kroz neke dvije stanice.

```
def load_csv_to_postgres(connection, cursor, table_name,
    csv_file_path):
  try:
    drop_table_sql = f'TRUNCATE {table_name} CASCADE;'
    execute_sql(connection, cursor, drop_table_sql)

    copy_sql = f""" COPY {table_name} FROM '{csv_file_path
       }' WITH
        CSV HEADER DELIMITER as ','"""
    execute_sql(connection, cursor, copy_sql)
    print(f"Loaded {table_name}")
  except Exception as e:
    print(f"Error loading {table_name}: {e}")

def create_route_find_table(connection, cursor):
  create_table_sql = """ CREATE TABLE route_find AS
      SELECT trips.route_id, trips.trip_id, stop_times.
         stop_sequence,
           stop_id, stops.stop_name
      FROM stop_times
      JOIN trips USING (trip_id)
      JOIN stops USING (stop_id); """
  # SQL naredba za brisanje stare tablice i stvaranje
     indeksa
  try:
    # execute naredbe
    execute_sql(connection, cursor, create_table_sql)

    print("route_find table created successfully.")
  except Exception as e:
    print(f"Error creating route_find table: {e}")
```

Slika 2.16. Isječak kôda skripte

2.4.3. Poslužiteljski dio aplikacije - node.js

Poslužiteljski dio aplikacije napisan je u jeziku JavaScript, koristeći Node.js okvir. JavaScript je programski jezik primarno namijenjen korištenju unutar web preglednika, ali uz pomoć Node.js moguće je izvršavanje JavaScript koda na poslužiteljskoj strani, izvan web preglednika, čime se omogućava korištenje jednog jezika za pisanje cjelokupne aplikacije. Najvažniji dodatak Node.js-u je Express.js, minimalistički i fleksibilan web okvir koji olakšava rad s HTTP zahtjevima i izgradnju web poslužitelja. On omogućava jednostavno definiranje ruta, rukovanje HTTP zahtjevima i upravljanje *middleware* funkcijama.

Na slici 2.17. je primjer vrlo jednostavnog poslužitelja koji kada dobije GET zahtjev na ruti *'/api/stops'* provodi upit nad bazom podataka i ovisno o odgovoru vraća uspješni (200 OK s podacima o stanicama) ili neuspješni odgovor (status 500 s porukom greške). Sve ostale dostupne rute aplikacije su napravljene prema istoj shemi.

```
1   const express = require('express');
2   const app = express();
3
4   app.get('/api/stops', async (req, res) => {
5     try {
6       const result = await pool.query(`SELECT DISTINCT
            stop_name FROM stops`);
7       res.json(result.rows);
8     } catch (error) {
9       console.error('[Error] Executing query', error);
10      res.status(500).json({ error: 'An error occurred' });
11    }
12  });
13
14  app.listen(8080, () => {
15    console.log('[Info] Server started on port 8080');
16  });
```

Slika 2.17. Primjer kôda jednostavnog poslužitelja

Kao i kod pomoćne skripte i na poslužitelju je potrebno ostvariti vezu s bazom podataka, to je omogućeno s modulom *pg*. Taj modul omogućava upravljanje vezama prema bazi i izvršavanje upita. Pomoću objekta *pool* definiramo potrebne informacije za spajanje poput *username, host, database, password i port* te onda pozivom metode *query* nad

objektom *pool* možemo izvesti proizvoljne SQL naredbe.

```
const { Pool } = require('pg');
require('dotenv').config();

const pool = new Pool({
  user: process.env.DB_USER,
  host: process.env.DB_HOST,
  database: process.env.DB_NAME,
  password: process.env.DB_PASSWORD,
  port: process.env.DB_PORT,
});

pool.query('SELECT * FROM stops');
```

Slika 2.18. Primjer spajanje s bazom podataka

Poslužitelj osim poslova upravljanja zahtjevima i komunikacijom s bazom podataka radi najbitniji dio aplikacije, a to je dohvaćanje GTFS realtime podataka sa ZET-ove stranice. Taj dio je ostvaren kao funkcija koja se periodički poziva (svakih 15 sekundi) kako bi dohvatila najnoviju *.protobuf* datoteku s realtime podacima, obradila ju i spremila u memoriju za brži pristup. Na slici 2.19. je čitav kod koji je zadužen za dohvaćanje podataka i njegovu deserijalizaciju iz binarnog .protobuf formata u JSON oblik.
Ti podaci se tada spremaju u jedan objekt *data* koji je napravljen tako da sadrži brojeve ruta kao ključeve i svaki tramvaj svrstava u točnu grupu. Također postoji funkcija koja upravlja dodavanjem u taj objekt i funkcija koja periodički briše tramvaje. To je napravljeno jer ZET-ovi podaci nisu konzistentni odnosno neki tramvaji nekada samo nestanu iz podataka makar je u *FeedHeader*-u *.protobuf* poruke navedeno da se svaki put šalju svi podaci (cijeli *dataset*).

```
const GtfsRealtimeBindings = require('gtfs-realtime-bindings');
async function fetchData() {
  try {
    const response = await fetch(feedUrl);
    if (!response.ok) {
      const error = new Error(
      `${response.url}: ${response.status} ${response.
         statusText}`
      );
      error.response = response;
      throw error;
    }
    const buffer = await response.arrayBuffer();
    const feed = GtfsRealtimeBindings.transit_realtime.
       FeedMessage.decode(
    new Uint8Array(buffer)
    );

    feed.entity.forEach((entity) => {
      if (entity) parseEntity(entity);
    });
  } catch (error) {
    console.log(error);
  }
}
```

Slika 2.19. Kôd za dohvaćanje GTFS realtime podataka i deserijalizaciju

2.4.4. Klijentski dio aplikacije - REACT

Klijentski dio također je izrađen u jeziku JavaScript pomoću React.js, popularnog JavaScript okvira koji se primarno koristi za izgradnju jednostraničnih aplikacija (SPA). React.js omogućava razvoj složenih i dinamičnih web aplikacija pomoću komponentnog pristupa, gdje se korisničko sučelje sastoji od manjih, ponovno upotrebljivih komponenti.

Komponente su osnovni gradivni elementi u React-u, a svaka komponenta predstavlja dio korisničkog sučelja. Komponente mogu biti jednostavne, poput gumba, ili složene, poput cijele stranice. Komponente su definirane pomoću JavaScript funkcija ili klasa, te vraćaju JSX (JavaScript XML), koji izgleda slično HTML-u.

Aplikacija se sastoji od par komponenti *App.js, Map.js, MapRouteLine.js, Marker.js, Sidebar.js i TripTimeline.js* od kojih je glavna roditeljska komponenta koja sadrži sve ostale *App.js* (slika 2.20.).

```
client
├── node_modules
├── public      -Kazalo s slikama ikona
├── src
│   ├── App.css     -Glavni stylesheet za aplikaciju
│   ├── App.js      -Glavna React komponenta
│   ├── Map.js      -Komponenta za kartu
│   ├── MapRouteLine.js      -Komponenta za linije ruta na karti
│   ├── Markers.js      -Komponenta za markere na karti
│   ├── Sidebar.js      -Komponenta za izbornik
│   └── TripTimeline.js      -Komponenta za vremensku liniju putovanja
├── package-lock.json
├── package.json
└── README.md
```

Slika 2.20. Osnovna struktura klijentskog kôda

App.js

Kao što je rečeno *App.js* je glavna komponenta koja se nalazi iznad svih ostalih, njezin glavni posao je da objedinjuje komponente i povremeno dohvaća (osvježava) podatke od poslužiteljskog dijela aplikacije uz pomoć *fetch* naredbe. Pseudokôd na slici 2.21. prikazuje način dohvaćanja podataka.

Kôd iskorištava *useEffect() i useState()* hook-ove. Hook-ovi su način na koji React omogućuje dinamičko upravljanje stanjem i ponašanjem funkcionalnih komponenti i njihovim životnim ciklusom. Hook *useEffect()* omogućuje komponentama da definiraju popratne efekte koji se trebaju izvesti nakon što se promjeni neko stanje definirano *useState()* hook-om. U kôdu aplikacija svakih 10 sekundi ili kada se promjeni stanje s odabranim rutama dohvaća nove podatke. Za svaku pojedinu navedenu rutu izvodi se jedan GET zahtjev, to se izvodi asinkrono pomoću obećanja (*Promises*) te se na kraju kada su

svi zahtjevi obrađeni grupira zajedno i šalje *"child"* komponenti *Map* koja prikazuje tramvaje kao markere na karti.

```
useEffect(() => {
  if (route && route.length > 0) {
    const intervalId = setInterval(async () => {
        const allRouteData = await Promise.all(
          route.map(fetchDataForRoute)
        );
        const combinedData = allRouteData.flat();
        setRouteData(combinedData);
    }, 10000);

    return () => clearInterval(intervalId);
  } else {setRouteData(null);}
}, [route, setRouteData]);

const fetchDataForRoute = async (routeValue) => {
  const response = await fetch(
      `/api/route/${routeValue}`
  );
  ...
};
```

Slika 2.21. Pseudokôd dohvaćanja podataka od poslužitelja

Map.js

Komponenta *Map.js* kojoj je glavna uloga da prikazuje kartu ostvarena je pomoću *leaflet.js*-a. *Leaflet.js* je popularna open-source JavaScript biblioteka za interaktivne mape. Omogućava jednostavno dodavanje dinamičkih karti, postavljanje markera, crtanje linija i poligona, interakciju s kartama, dodavanje pop-up prozora i još mnogo toga [14]. Postoji React verzija *leaflet.js* koja objekte tretira kao komponente i s kojom je još jednostavnije raditi u React-u.
Karta koju *leaflet.js* koristi je dohvaćena iz OpenStreetMap projekta, koji je otvorena i slobodna baza geografskih podataka koju održavaju volonteri.

Na slici 2.22. se nalazi isječak koda koji opisuje što sadrži *Map.js* te možemo primijetiti komponente *MapContainer, ZoomControl, TileLayer* koje su standardne komponente iz *leaflet.js* i koje definiraju izgled karte , dok komponenta *Markers* koja se koristi za pos-

tavljanje markera na kartu i komponenta *MapRouteLine* koja se koristi za crtanje linija ruta na karti su definirane za potrebe aplikacije.

```
return (
  <MapContainer
  center={[45.808680463038435, 15.977835680373971]}
  zoom={13}
  zoomControl={false}
  whenReady={(event) => setMapContext(event.target)}
  >
  <ZoomControl position="bottomright" />
  <TileLayer
  attribution='&copy; <a href="https://www.openstreetmap.org/
      copyright">OpenStreetMap</a> contributors'
  url="https://tile.openstreetmap.org/{z}/{x}/{y}.png"
  />
  <Markers
  routeData={routeData}
  tripInfo={tripInfo}
  ....
  />
  <MapRouteLine coordinates={coordinates} />
  <MapClickHandler />
  </MapContainer>
);
```

Slika 2.22. Isječak kôda Map.js

Markers.js

Markers.js komponenta je zadužena za stvaranje marker na karti koji će prikazivati pozicije tramvaja i njihove dotične animacije. Sami markeri na karti su ostvareni uz pomoć *react-leaflet.js* komponenti *Marker i Popup*. Animacije markera koje predstavljaju predikcije položaja tramvaja su bazirane na vremenima iz ZET-ovih rasporeda koji su dio static datoteka (poglavlje 2.2.3.). Osim prikaza obrađuje i klikove na pojedine markere koji tada aktiviraju popratne akcije prikaza dodatnih informacija o putu pomoću *Popup i TripTimeline* komponenti.

MapRouteLine.js

MapRouteLine.js ostvaruje iscrtavanje putanje tramvaja na karti koje je ostvareno dohvaćanjem pozicija stanica tramvaja i spajanjem tih koordinata u jednu *Polyline* liniju. Idealno rješenje ovog problema bi bili podaci iz datoteke *shapes.txt* koji detaljno propisuju putanju rute, ali ta datoteka nedostaje u ZET-ovim static podacima. Još jedan način rješavanja ovog problema je korištenjem *routing* komponente kojoj predajemo koordinate stanica te onda ona provodi pronalaženje puta na karti između svake dvije točke i time stvara glatku putanju na karti. Problem kod tog načina je da se za pronalaženje puta koriste putevi kojim auti mogu voziti, a ne tračnice kojim tramvaji voze što također dovodi do nepreciznosti tjst. neistinitosti nacrtane putanje naspram stvarne, tako da je taj način odbačen.

Sidebar.js

Sidebar.js je glavna i jedina komponenta koja ostvaruje komunikaciju s korisnikom. Korisniku su dostupne padajuće liste ostvarene komponentom *Select* kojima bira rute ili stanice kod dijela za pretraživanje ruta. Također sadrži par tipki koje služe za upravljanjem nad odabranim rutama. Kada se odabere ruta ili pretraže rute za stanicu komponenta mijenja stanje aplikacije što izaziva dohvaćanje novih podataka.

TripTimeline.js

Klikom na marker se izaziva *useEffect()* hook u komponenti *TripTimeline.js* koje aktivira njen prikaz. Ta komponenta je zadužena za prikazivanje rasporeda puta tramvaja, koje je ostvareno kao vremenska crta pomoću *VerticalTimeline* komponente. Raspored puta sadrži u redoslijedu stanice na koje će tramvaj doći, njihovo vrijeme očekivanog dolaska i vremena potrebnog do sljedeće stanice.

2.5. Izgled aplikacije

3. Zaključak

Literatura

[1] Zagrebački Električni Tramvaj, “ZET - O nama”, https://www.zet.hr/o-nama/259, [Stranica posjećena: 16.05.2024.].

[2] ——, “ZET - GTFS podaci”, https://www.zet.hr/odredbe/datoteke-u-gtfs-formatu/669, [Stranica posjećena: 27.05.2024.].

[3] ZET info, https://zet-info.com/.

[4] General Transit Feed Specification, “GTFS - About”, https://gtfs.org/.

[5] Mapnificent, https://github.com/mapnificent/mapnificent.

[6] General Transit Feed Specification, “GTFS - Schedule/Static”, https://gtfs.org/schedule/.

[7] Trafiklab, “Informacije o GTFS schedule modelu i slika modela”, https://www.trafiklab.se/docs/using-trafiklab-data/using-gtfs-files/static-gtfs-files/.

[8] General Transit Feed Specification, “GTFS - Realtime”, https://gtfs.org/realtime/,https://gtfs.org/realtime/reference/.

[9] Protocol buffers, https://protobuf.dev/.

[10] PostgreSQL, https://www.postgresql.org/about/.

[11] Google, “GTFS reference”, https://developers.google.com/transit/gtfs/reference.

[12] Node.js, https://nodejs.org/en.

[13] React.js, https://react.dev/.

[14] Leaflet.js, https://leafletjs.com/.

Sažetak

Vizualizacija prometa javnog prijevoza u stvarnom vremenu uporabom formata GTFS

Luka Miličević

Unesite sažetak na hrvatskom.

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Etiam lobortis facilisis sem. Nullam nec mi et neque pharetra sollicitudin. Praesent imperdiet mi nec ante. Donec ullamcorper, felis non sodales commodo, lectus velit ultrices augue, a dignissim nibh lectus placerat pede. Vivamus nunc nunc, molestie ut, ultricies vel, semper in, velit. Ut porttitor. Praesent in sapien. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Duis fringilla tristique neque. Sed interdum libero ut metus. Pellentesque placerat. Nam rutrum augue a leo. Morbi sed elit sit amet ante lobortis sollicitudin. Praesent blandit blandit mauris. Praesent lectus tellus, aliquet aliquam, luctus a, egestas a, turpis. Mauris lacinia lorem sit amet ipsum. Nunc quis urna dictum turpis accumsan semper.

Ključne riječi: prva ključna riječ; druga ključna riječ; treća ključna riječ

Abstract

Visualization of public transport traffic in real time using the format GTFS

Luka Miličević

Enter the abstract in English.

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like “Huardest gefburn”? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

Keywords: the first keyword; the second keyword; the third keyword

Privitak A: The Code