SUSTAV ZA NAPLAĆIVANJE JAVNOG PRIJEVOZA

Mandić, Igor

VISOKO UČILIŠTE ALGEBRA

ZAVRŠNI RAD

SUSTAV ZA NAPLAĆIVANJE JAVNOG PRIJEVOZA

Igor Mandić

Zagreb, kolovoz 2018.

„Pod punom odgovornošću pismeno potvrđujem da je ovo moj autorski rad čiji niti jedan dio nije nastao kopiranjem ili plagiranjem tuđeg sadržaja. Prilikom izrade rada koristio sam tuđe materijale navedene u popisu literature, ali nisam kopirao niti jedan njihov dio, osim citata za koje sam naveo autora i izvor te ih jasno označio znakovima navodnika. U slučaju da se u bilo kojem trenutku dokaže suprotno, spreman sam snositi sve posljedice uključivo i poništenje javne isprave stečene dijelom i na temelju ovoga rada“.

U Zagrebu, 08.08.2018.

Predgovor

Zahvaljujem svom mentoru, dipl.ing. Aleksanderu Radovanu, na iskazanom povjerenju i svesrdnoj pomoći tijekom izrade završnog rada.

Zahvaljujem i mag.ing.comp. Denisu Vajaku, Karlu Crnekoviću i Darianu Jugu na pomoći i savjetima tijekom izrade završnog rada i testiranju sustava.

Od srca zahvaljujem svojoj obitelji na pruženoj potpori tijekom studija.

Prilikom uvezivanja rada, umjesto ove stranice ne zaboravite umetnuti original potvrde o prihvaćanju teme završnog rada kojeg ste preuzeli u studentskoj referadi

Sažetak

Ideja razvoja sustava nastala je iz želje da se ljudima putem tehnologije olakša i pojednostavi svakodnevni život. Giganti poput Applea i Googlea uvode ljude u novu tehnološku dob u kojoj im se pruža prilika da dosta trivijalnih zadataka koje neprestano odrađuju automatiziraju, olakšaju ili u potpunosti uklone, kako bi imali više vremena koncentrirati se na stvari koje su uistinu bitne.

Abstract

The driving force behind the idea for this system comes from a desire to help and empower people through technology by making their everyday life easier and more painless. Technological giants the likes of Apple and Google lead us into a new technological era in which we have the opportunity to make our daily lives easier by automating and making our tasks and routines easier or outright removing some of the unnecessary tasks.

Sadržaj

1. Uvod 1
1.1. ZET i kupovina karata za javni prijevoz 1
1.2. Usporedba 2
2. Conductor 3
2.1. GTFS 4
2.2. Prepoznavanje vozila i postaja 6
2.2.1. Eddystone 10
2.3. Conductor - Backend 13
2.3.1. Korištenje 19
3. „Conductor– Mobile“ 22
Zaključak 28
Popis kratica 29
Popis slika 30
Popis kôdova 31
Literatura 32

1. Uvod

Cilj ovog završnog rada je analizirati problematiku plaćanja javnog prijevoza, analizirati postojeći sustav, te osmisliti rješenje koje bi uklonilo nedostatke istog upotrebom svakodnevno dostupne tehnologije. Završni rad je fokusiran na javni prijevoz grada Zagreba, no sustav opisan u radu lako je primjenjiv i u drugim gradovima, pošto je za potpunu funkcionalnost sustava potrebna minimalna integracija s postojećim sustavom upravljanja javnog prijevoza. Trenutni sustav plaćanja zasniva se na kupnji papirnatih karata koje su dostupne isključivo na odabranim prodajnim mjestima diljem Zagreba ili kod vozača tramvaja i autobusa, te kupovinom povlaštenih ili pretplatnih karata na prodajnim mjestima ZET-a.

U drugom poglavlju će biti opisan rad sustava, tehnologije koje su bile korištene u izradi, te način na koji je moguće putem *API-a* lako integrirati sustav Conductor s postojećim sustavima za upravljanje javnim prijevozom.

Treće poglavlje posvećeno je opisu radu aplikacije te promjenama koje su potrebne na samim vozilima kako bi se omogućilo korištenje sustava.

1.1. ZET i kupovina karata za javni prijevoz

ZET-ove prijevozne karte dostupne su u dva oblika, u obliku papirnatog listića koji je moguće kupiti kod vozača i na odabranim prodajnim mjestima (kiosci Tiska, iNovina, tobacco shopovi na autobusnim i tramvajskim postajama i sl.), te u obliku RFID kartice koju je moguće kupiti isključivo na ZET-ovim prodajnim mjestima kojih po cijelom Zagrebu postoji malen broj.

Papirnati listići su u uporabi kao pojedinačne karte koje moraju biti poništene prilikom ulaska u vozilo. ZET-ov tarifni sustav sadrži više vrsta pojedinačnih karata s vremenskim ograničenjima od 30 ili 90 minuta ili cjelodnevna karta s kojom je moguće neograničeno putovati sve dok karta vrijedi.

RFID kartice koriste se kao vrijednosne karte koje je potrebno nadopuniti na odabranim prodajnim mjestima te vrijednost iste umanjuje se prilikom kupnje karata u vozilu. Druga vrsta RFID karte je pokaz, mjesečna karta koju je moguće kupiti na prodajnim mjestima ZET-a, koji korisnik mora ovjeriti na uređaju u vozilu.

1.2. Usporedba

Najviše nezadovoljstva građana Zagreba s postojećim sustavom proizlazi iz visoke i nelogične cijene prijevoznih karata, te nepotrebne komplikacije u nabavi i plaćanju istih.

Sustav Conductor taj problem rješava vrlo jednostavno, na način da omogući plaćanje javnog prijevoza putem mobilnog telefona. Sustav je kreiran i ostvaren tako da će pri ulasku korisnika u vozilo ZET-a, mobilni uređaj prepoznati u kojem se vozilu korisnik nalazi, te u izborniku u aplikaciji ponuditi popis sljedećih postaja. U izborniku korisnik odabere postaju do koje putuje, te plati samo za prijeđenu udaljenost, što je puno pošteniji i pristupačniji model naplate javnog prijevoza.

Jedna od glavnih prednosti je ta da se vozača vozila ne ometa tijekom vožnje, uzevši u obzir da u svim vozilima postoji pisano upozorenje o neometanju vozača tijekom vožnje.

Na sličan način aplikacija lako prepoznaje na kojoj se postaji korisnik nalazi i prikaže vozni red za sve tramvajske ili autobusne linije koje se zaustavljaju na toj postaji.

Ograničenje postojećeg sustava su zasloni koji nerijetko imaju kvar zbog kojeg su slova i brojevi neispravno prikazani; npr. broj 8 izgleda kao broj 6 ili 9 što zbunjuje korisnike javnog prijevoza, odnosno zasloni često u potpunosti ne rade. Isto tako trenutni zasloni imaju mogućnost prikazati samo sljedeća četiri stajanja, dok Conductor može prikazati raspored stajanja za cijeli dan.

2. Conductor

Conductor je zasnovan na softveru otvorenog koda Odoo u obliku modula, u kojem se model podataka temelji na GTFS standardu. Glavna prednost Odooa je u ubrzavanju razvoja sustava, olakšavanju održavanja te vrlo lakoj mogućnosti proširivanja s dodatnim funkcionalnostima, dok istovremeno minimalizira problematiku izrade korisničkog sučelja, validacije unosa podataka, te pristupnih kontrola.

Korištenjem GTFS-a kao temelja za model podataka o voznom redu javnog prijevoza omogućena je vrlo lagana integracija s postojećim informacijskim sustavom koji ZET koristi za vođenje voznih redova svojih prijevoznih linija. Nakon nedavne modernizacije ZET-ovog informacijskog sustava svi podaci o voznim redovima su javno dostupni u GTFS formatu.

Za uspješan rad sustava potrebna je sinkronizacija između sustava ZET-a i Conductora kako bi u aplikaciji bio točno prikazan popis nadolazećih postaja za određenu prijevoznu liniju i cjelodnevni raspored stajanja linija na određenoj postaji. Ta sinkronizacija je vrlo lako ostvariva putem *API* metoda koje su korištene za slanje novih podataka o planiranim vremenima stajanja vozila na svakoj postaji ili čak putem WebSocketa kroz koji se konstantno šalju promjene istih u sustav Conductor.

Kupljena karta se plaća kreditnom ili debitnom karticom putem *payment gateway* servisa poput Stripea ili Braintreea.

Cijeli sustav je osmišljen tako da komplimentira postojeći, što u praksi znači da ako putnik nije u mogućnosti platiti prijevoznu kartu putem mobilnog uređaja, još uvijek ima mogućnost kupiti kartu na uobičajen način.

Kao dodatnu funkcionalnost Conductor nudi i plaćanje parkinga u jednom ili više gradova. Danas većina gradova uz naplaćivanje javnog parkinga putem automata za naplatu parkinga nudi i mogućnost plaćanja putem SMS naplatnog sustava. Conductor se oslanja na taj sustav, jer u pozadini uz korisnikovo dopuštenje odradi potrebnu komunikaciju s istim, u svrhu prijave i plaćanja istog.

2.1. GTFS

GTFS (engl. General Transit Feed Specification) je de-facto standard za pohranu i prijenos podataka o rasporedu javnog prijevoza.

Standard je započet kao projekt u Googleu 2005. godine u potrazi za rješenjem za prikaz informacija o javnom prijevozu u Google Mapsu. Standard je u početku imao naziv *Google Transit Feed Specification*, no nakon šire primjene standard je bio preimenovan u *General Transit Feed Specification.*[1]

Struktura je vrlo jednostavna i sastoji se od između 6 i 13 *CSV* datoteka koje imaju *.txt* ekstenziju. Te datoteke su zatim komprimirane i pohranjene u *.zip* datoteci, a znakovi su kodirani s *UTF-8* standardom kodiranja. [2]

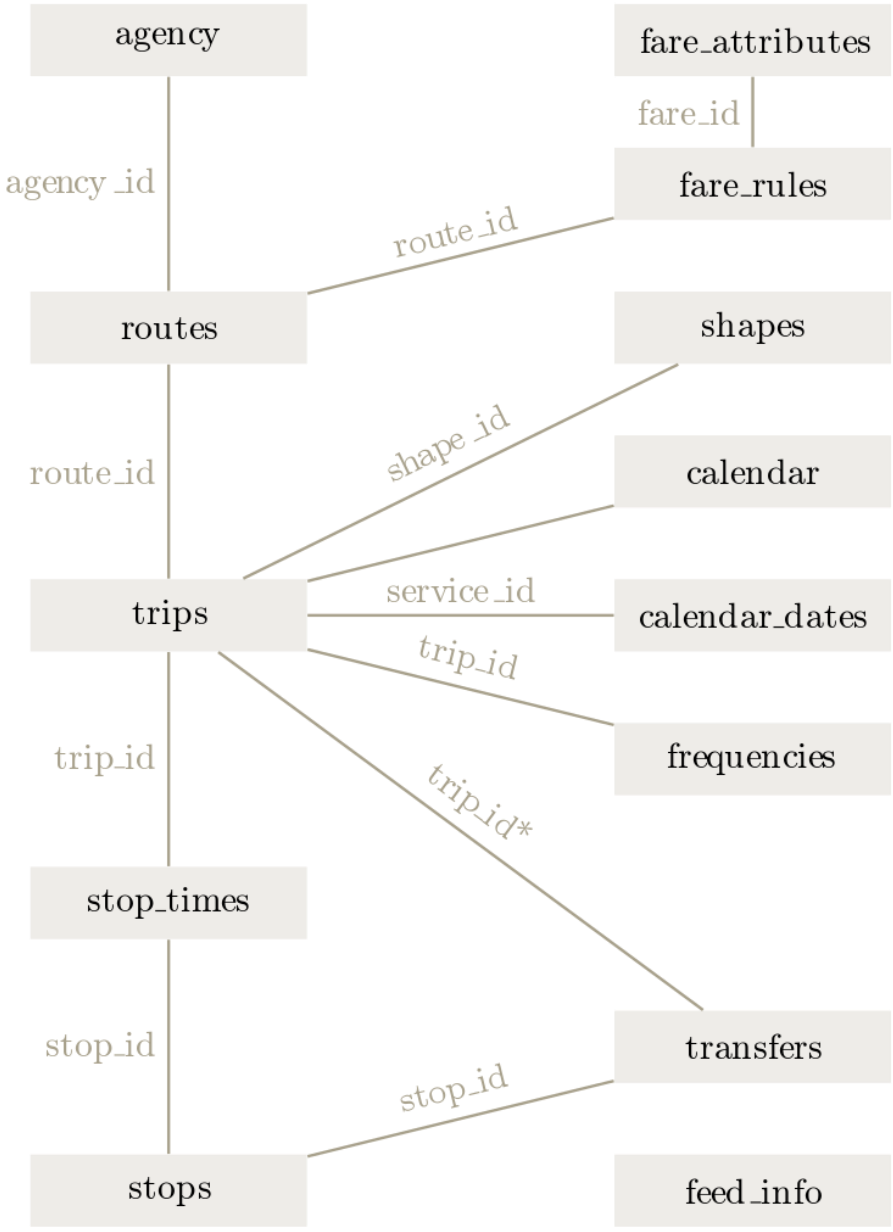
Tablice pohranjene u tim *CSV* datotekama su u međusobnim relacijama, te zajedno opisuju raspored djelovanja tranzitnog sustava.

Obavezne tablice su:

- *agency* - sadrži informacije o tranzitnoj agenciji, odnosno organu ili organizaciji koja je zadužena za upravljanje tranzitnog sustava
- *routes* - sadrži informacije o pojedinačnim linijama
- *trips* - sadrži putovanja vozila i služi kao relacija između *routes* i *stop_times*
- *stop_times* – vrijeme stajanja i kretanja vozila te lokacija istog
- *stops* - sadrži popis svih postaja
- *calendar* - sadrži obrasce ponavljajućih usluga

Neobavezne tablice su:

- *calendar_dates* - iznimke za svaki *service_id* koji se nalazi u *calendar.txt* datoteci
- *fare_attributes* - informacije o cijenama tranzita
- *shapes* - pravila kako prikazati tranzitne rute
- *frequencies* - vrijeme između dva putovanja određene linije
- *transfers* - pravila kako raditi poveznicu između dvije točke prestupa
- *feed_info* - dodatne informacije o toku



Slika 1. Relacije među tablicama [3]

2.2. Prepoznavanje vozila i postaja

Za prepoznavanje vozila se koriste takozvani *Bluetooth Beaconi*. To su hardverski uređaji koji koriste *BLE* (engl. *Bluetooth Low Energy)* standard u kojem je definiran način na koji se periodično odašilje jedinstven *UUID* identifikator.

BLE je tehnologija za bežične osobne mreže (*engl. wireless personal area network*) dizajnirana s ciljem primjene u poljima zdravstva i fitnessa, sigurnosti te kućne zabave. Za razliku od prethodnih standarda, *BLE* je dizajniran s ciljem korištenja osjetno manje količine električne energije tijekom rada bez značajne promjene u dometu signala. Gotovo svi mobilni operacijski sustavi (Android i iOS), te Windows 8 i Windows 10, macOS i Linux podržavaju taj novi standard [4]. Standard je idealan za primjenu u *Bluetooth Beacon* tehnologiji jer omogućuje dugoročni rad odašiljača koristeći bateriju duže vrijeme, čiji vijek trajanja može biti od jedne do dvije godine.

UUID (*engl. Universally Unique Identifier*) je 128-bitni broj koji je korišten za identifikaciju informacija u računalnim sustavima. Drugo ime za isti je *Globally Unique Identifier*. UUID generiran na standardni način je za praktične svrhe jedinstven bez da ovisi o centralnom registru kako bi se koordiniralo generiranje istog. Unatoč tome, vjerojatnost da su dva UUID-a jedinstvena nije 0, no ona je još uvijek dovoljno mala da je zanemariva. Za 50% vjerojatnosti kolizije dvaju UUID identifikatora, moralo bi se generirati $2.71 * 10^{18}$ identifikatora, odnosno moralo bi se 85 godina neprestano generirati jedna milijarda UUID-a u jednoj sekundi.[5]

Na tržištu su trenutno najpopularnija dva standarda za *bluetooth beacone*, Googleov *Eddystone* i Appleov *iBeacon*. Conductor za svoje potrebe koristi *Eddystone* standard zbog *Eddystone EID* formata u svrhu sprječavanja problema podvaljivanja u kojem zlonamjerni korisnik može kopirati *UUID* određenog *bluetooth beacona* kako bi ometao rad sustava.

Beaconi su raspoređeni po vozilu te je njihova jačina odašiljanja signala kalibrirana tako da svojim dometom pokriju cijelu dužinu i širinu vozila uz minimalno preklapanje dometa signala dvaju vozila koja stoje jedno pored drugog. Budući da je moguće u bilo kojem trenutku detektirati signal od više različitih *beacona*, aplikacija uzima u obzir onaj najbliži.

```
private BeaconInfo getNearestBeacon() {

        BeaconInfo closestBeacon = null;
        BleSignal closestSignal = null;

        for (String key: spottedVehicleBeacons.keySet()) {

                BeaconInfo currentBeacon = spottedVehicleBeacons.get(key);

                BleSignal currentSignal = beaconTelemetry.get(key);

                if (closestBeacon == null || closestSignal == null) {

                        closestBeacon = currentBeacon;

                        closestSignal = currentSignal;

                        continue;

                }

                if (currentSignal.getRssi() < closestSignal.getRssi()) {

                        closestBeacon = currentBeacon;

                        closestSignal = currentSignal;

                }

        }

        return closestBeacon;

}
```

Kôd 1. Metoda za odabir najbližeg *beacona*

Tijekom rada aplikacije podaci za svaki *beacon* koji su detektirani spremljeni su u dvije `HashMap` varijable. Jedna je korištena za telemetriju, a druga za objekte koji sadrže podatke asocirane uz svaki *beacon.* UUID identifikator koji je asociran uz svaki *beacon* je korišten kao ključ u obje `HashMap` varijable za lakšu iteraciju, kao što je prikazano u metodi u kôdu 1.

Algoritam u prethodnom isječku kôda je osnovan na *linear search* algoritmu u kojem kôd iterativno prolazi kroz UUID identifikatore svih do sada detektiranih *beacona*, te koristeći taj identifikator dohvaća podatke o telemetriji za svaki.

Varijable *closestSignal* i *currentSignal* su korištene za usporedbu pri pronalaženju najbližeg *beacona* temeljem jačine signala. Prva varijabla sadrži podatke o signalu do sada najbližeg *beacona,* dok druga sadrži iste podatke za trenutnu iteraciju. Ako je signal za *beacon* u trenutnoj iteraciji jači od prethodno određenog, uzima se kao do sada najjači, te se sprema u varijablu *closestSignal*, a podaci o *beaconu* koji je smatran najbližim su spremljeni u varijablu *closestBeacon.*

Taj postupak dohvaćanja podataka se ponavlja za sve identifikatore, te metoda na kraju vrati podatke asocirane uz *beacon* koji je najbliži korisnikovom uređaju.

Metoda u sljedećem isječku je zadužena za dohvaćanje podataka o postaji ili vozilu kojem je *beacon* dodijeljen, te prosljeđivanju iste u ostale dijelove kôda aplikacije.

```
public void notifyBeaconListeners(String beaconUuid) {
    Call<BeaconInfo> beaconInfoCall = beaconsApi.getBeacon(beaconUuid.replace("-", ""));
    beaconInfoCall.enqueue(new Callback<BeaconInfo>() {
        @Override
        public void onResponse(Call<BeaconInfo> call, Response<BeaconInfo> response) {
            if (response.isSuccessful()) {
                if (response.body().getType() == BeaconInfo.TypeEnum.VEHICLE) {
                    spottedVehicleBeacons.put(String.valueOf(response.body().getId()),
                        response.body());
                }
                if (response.body().getType() == BeaconInfo.TypeEnum.STATION) {
                    spottedStationBeacons.put(String.valueOf(response.body().getId()),
                        response.body());
                }
                for (IBeaconSpotter beaconSpotter : beaconSpotterList) {
                    beaconSpotter.onBeaconSpotted(spottedStationBeacons,
                        spottedVehicleBeacons, beaconSpotTimes, beaconTelemetry);
                }
            } else {
                Log.e(TAG, "onFailure: Failed to get beacon info: " + response.raw());
            }
        }
        @Override
        public void onFailure(Call<BeaconInfo> call, Throwable t) {
            Log.e(TAG, "onFailure: Failed to get beacon info: " + t.getMessage());
        }
    });
}
```

Kôd 2. Dohvaćanje podatka asociranog s *beaconom* i prosljeđivanje istog

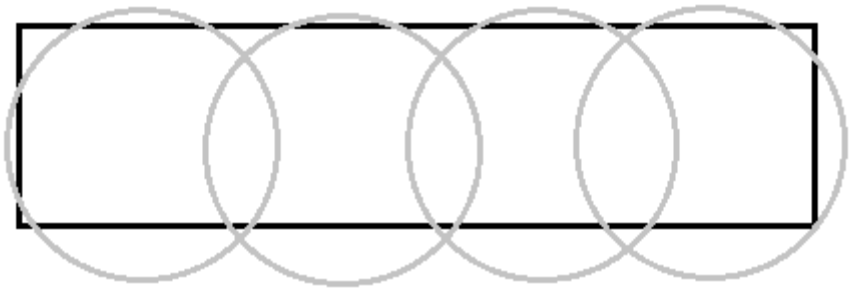
Metoda `notifyBeaconListeners` je pozvana svaki put kad aplikacija prepozna *beacon* i pročita njemu asociran identifikator koji je zatim normaliziran u niz slova i brojeva bez znaka za minus, te u tom obliku proslijeđen na API server s pozivom metode `enqueue`.

Odgovor od servera je zaprimljen u `Callback<BeaconInfo>` objektu (jer je metoda asinkrona) u kojem se radi provjera vrste *beacona* u metodi `onResponse` koja je pozvana kada je zaprimljen odgovor od servera. U toj metodi se nalazi kôd koji gleda vrstu *beacona* (moguće vrste: „vehicle“ ili „station“) te sukladno tome su podaci spremljeni u namijenjene varijable. Razlika između te dvije vrste je u podacima koji su asocirani. *Vehicle beacon* sadrži podatke o nadolazećim stanicama, dok *Station beacon* sadrži podatke o stanici poput imena stanice i stajanja vozila.

Varijabla s imenom `spottedStationBeacons` je korištena za pohranu podataka o *beaconima* koji su dodijeljeni postajama, a `spottedVehicleBeacons` za pohranu podataka o *beaconima* koji su dodijeljeni vozilima.

Nakon sortiranja podataka, kôd u for petlji iterira kroz popis objekata koji čekaju obavijesti te istima proslijedi obje varijable.

Bilo koja moguća greška koja može nastati tijekom dohvaćanja podataka sa servera je obrađena u metodi `onFailure` ili u `onResponse`. Prva metoda je pozvana ako komunikacija sa serverom uopće nije moguća zbog greške u mreži, a u drugoj metodi se gleda da li je server vratio HTTP kod u rasponu od 200 do 300. Sve greške koje se dogode su bilježene u Androidov log do kojeg se dostupa s alatom *Logcat* koji je dostupan u *Android Studiu*.



Slika 2. Primjer postavljanja *beacona* te konfiguracije dometa

Slika 2 ilustrira domet pojedinačnih *beacona* u vozilu te njihove lokacije. *Beaconi* nisu centrirani u samom vozilu, već su pomaknuti bliže vratima koja se nalaze na desnoj strani vozila u odnosu na smjer kretanja, jer se rijetko, ako ikada, dogodi situacija u kojoj dva vozila putuju u istom smjeru jedno pored drugog. Čak i da jesu dva vozila u bilo kojem obliku jedno pored dugog, aplikacija još uvijek uzima u obzir samo najbliži *beacon* te zbog toga bi *beaconi* koji se nalaze u drugom vozilu bili ignorirani.

2.2.1. Eddystone

Eddystone je *Bluetooth Low Energy* profil koji je dizajnirao Google i javno objavio 2015. godine pod licencom Apache 2.0. U standardu su definirani četiri tipa *Eddystone* poruka tj. *Eddystone Frame*. [6]

Nazivi su:

- *Eddystone-UID*: *frame* koji u sebi sadrži unikatan identifikator koji je sastavljen od *Namespace* komponente veličine 10 bajtova, te *Instance* komponente veličine 6 bajtova [7].
- *Eddystone-URL*: sadrži u sebi URL (engl. *Uniform Resource Locator*) do web stranice. Te poruke su prikazane u obliku notifikacije na uređajima s operacijskim sustavom Android 4.4. ili novijim. Za detekciju poruka i prikaza notifikacija na iOS uređaju korisnik mora imati instaliran Google Chrome s ispravnim dozvolama [7].

- *Eddystone-TLM*: specijaliziran *frame* koji sadrži telemetriju za pojedinačni *bluetooth beacon*. U njemu su zapisane informacije o razini baterije, podaci sa senzora ili ostali podaci koji su administratorima bitni. Ovaj *frame* mora biti uparen s nekim drugim *frameom* kako bi *bluetooth beacon* imao korisnu primjenu [7].
- *Eddystone-EID*: djeluje vrlo slično *Eddystone-UID* okviru, no razlika je u tome da umjesto odašiljanja statičkog identifikatora, emitira identifikator koji se periodično mijenja. Taj identifikator je onda moguće pretvoriti u konkretnu informaciju - najčešće kroz Google Proximity Beacon API koji je dio Google Beacon platforme. Prilikom konfiguracije *bluetooth beacon-a* za emitiranje *Eddystone-EID* okvir, kriptografski ključ koji je korišten za generiranje rotirajućih ključeva je pohranjen na samom uređaju i na servisu koji je korišten za pretvorbu ključa [7].

Za korištenje *Google Beacon* platforme Google je još razvio i *Nearby API* u obliku programskih knjižnica za Android i iOS platforme. One olakšavaju posao programera na način da apstrahiraju i sakriju detalje prepoznavanja *bluetooth beacon*-a, komunikaciju s *Google Beacon* platformom i dohvaćanje asociranih podataka za iste.

Google je paralelno s *Eddystone* standardom upogonio svoju *Google Beacon* platformu koja služi kao centralni registar svih *bluetooth beacona*, u kojem je moguće asocirati podatak u obliku *Attachmenta* uz svaki *Beacon*.

Isječak koda u nastavku pokazuje kako aplikacija koristi *Nearby* API. Objekt `MessageListener` sadrži *callback* metode koje API poziva kada je novi *beacon* detektiran, kada uređaj izađe van dometa *beacona*, te metodu kojom uređaj prepoznaje promjenu snage signala. Svaki put kad je *beacon* detektiran, identifikator istog je proslijeđen u metodu za dohvaćanje podataka i obavještavanje, koja je opisana u prethodnom isječku.

```
private void nearbySubscribe() {
    mMessageListener = new MessageListener() {
        @Override
        public void onFound(Message message) {
            super.onFound(message);
            notifyBeaconListeners(new String(message.getContent()));
            beaconSpotTimes.put(new String(message.getContent()),
                org.joda.time.LocalDateTime.now());
        }
        @Override
        public void onLost(Message message) {
            super.onLost(message);
        }
        @Override
        public void onBleSignalChanged(Message message, BleSignal bleSignal) {
            super.onBleSignalChanged(message, bleSignal);
            beaconTelemetry.put(String.valueOf(message.getContent()), bleSignal);
        }
    };

    try {
        SubscribeOptions options = new SubscribeOptions.Builder()
            .setStrategy(Strategy.BLE_ONLY)
            .build();
        Nearby.Messages.subscribe(mGoogleApiClient, mMessageListener, options);
    }
    catch(Exception ex) {
        Log.e("BT", "Error:" + ex.toString());
    }
}
```

Kôd 3. Detektiranje *beacona* i dohvaćanje UUID identifikatora za isti

Svaki puta kada je novi *beacon* detektiran, metoda `onFound` je pozvana, u njoj su onda pozvane metode `nofityBeaconListeners` i `beaconSpotTimes.put`. Prva metoda je zadužena za obavještavanje i prosljeđivanje podataka o *beaconu* kada je prepoznat, dok druga metoda sprema vrijeme kada je *beacon* bio prepoznat.

Metoda `onLost` je pozvana svaki puta kada uređaj izgubi kontakt s *beaconom*, odnosno kada prestane primati poruke od *beacona*.

Metoda `onBleSignalChanged` je pozvana svaki puta kada sustav detektira promjenu u jačini signala pojedinačnog *beacona*. Vrijednost koja se gleda je RSSI (*engl. Received signal strength indication*), što je mjerilo za snagu zaprimljenog radijskog signala. U toj metodi svaki puta kada je pozvana, kôd spremi novu RSSI vrijednost pozivom metode `beaconTelemetry.put`.

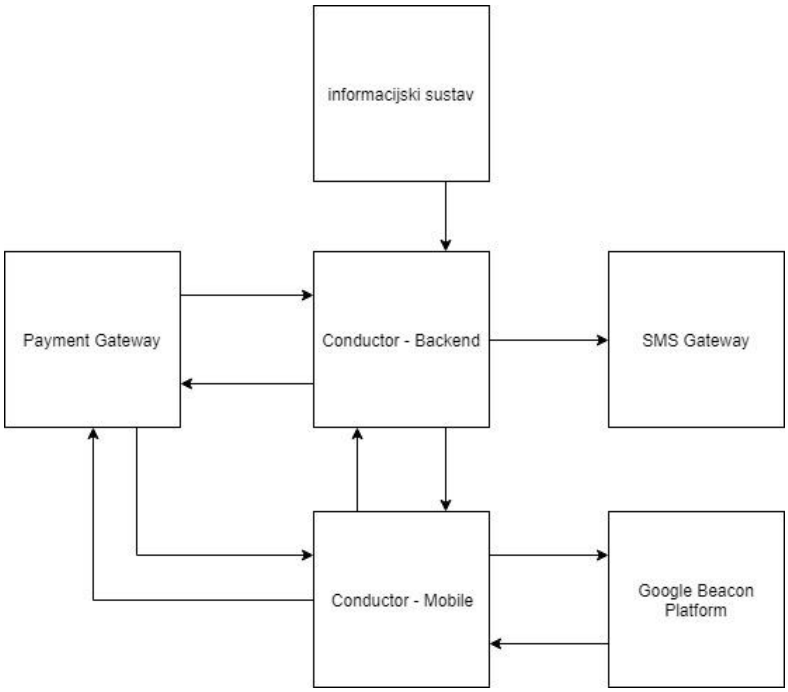
Kôd u `try` bloku je zadužen za registriranje `MessageListener` objekta sa sustavom. Tu može nastati fatalna greška u sustavu zbog toga što aplikaciji sustav nije dozvolio korištenje određenih API-a potrebnih za funkcionalnost ili sam uređaj nema hardver potreban za korištenje BLE tehnologije.

2.3. Conductor - Backend

Conductor je podijeljen u dvije logičke jedinice nazvane „Conductor – Backend“ te „Conductor – Mobile“.

„Conductor – Backend“ je zadužen za:

- pohranu GTFS podataka iz kojih su vidljive informacije za svako vozilo i postaju
- prikaz informacija o rasporedu vožnji vozila
- izračun cijene prijevozne karte temeljem broja proputovanih stajališta
- komunikaciju s *payment gateway* servisom
- komunikaciju s mobilnom aplikacijom za dohvat informacija o voznom redu te kupnju prijevoznih karata
- komunikaciju s informacijskim sustavom operatera javnog prijevoza u svrhu sinkronizacije
- komunikacija s *SMS Gatewayom* za slanje autorizacijskih kodova tijekom prijave i registracije u aplikaciju



Slika 3. Shema sustava i komunikacije s vanjskim servisima

Komunikacija između „Conductor – Backend“ te „Conductor – Mobile“ odvija se putem *REST API-a* koji je definiran uz pomoć Swagger 2.0 standarda koji je vrlo sličan *WSDL*-u. Kako bi se osigurala maksimalna sigurnost rada sustava Conductor, komunikacija se odvija samo putem HTTPS protokola.

Interakcija, komunikacija i međusobna povezanost prethodno navedenih vanjskih servisa, „Conductor – Backenda“ i „Conductor – Mobilea“ je prikazana u Slici 3.

„Conductor – Backend“ i *Payment Gateway* međusobno izmjenjuju podatke potrebne za terećenje korisnikove kartice u obliku jednokratnog tokena i količine novaca koju treba naplatiti.

„Conductor – Mobile“ i „Conductor – Backend“ međusobno razmjenjuju podatke o vozilima, stanicama, identifikatore asocirane s *beaconima,* jednokratne identifikatore za terećenje kartice, te informacije o kupljenim kartama. „Conductor – Mobile“ jednokratni identifikator dobije od Payment Gateway servisa, dok za pridobivanje asociranih identifikatora koristi Google Beacon Platformu.

Za uspješno slanje SMS poruka s autentikacijskim kôdom sustav „Conductor – Backend“ šalje tijelo poruke i telefonski broj primatelja *SMS Gatewayu.*

U svrhu prikazivanja ispravnih podataka informacijski sustav operatera javnog prijevoza šalje podatke o voznom redu u sustav „Conductor – Backend“.

```
/beacon/{beacon_uuid}:
  get:
    tags:
    - "beacons"
    summary: "Get beacon information"
    description: ""
    operationId: "get_beacon"
    produces:
    - "application/json"
    parameters:
    - name: "beacon_uuid"
      in: "path"
      description: "Beacon ID to search by"
      required: true
      type: "string"
    responses:
      200:
        description: "Found beacon"
        schema:
          $ref: "#/definitions/BeaconInfo"
      404:
        description: "No beacon found for given uuid"
    x-swagger-router-controller: "conductor_api.controllers.beacons_controller"
```

Kôd 4. Primjer Swagger definicije REST metode za dohvaćanje podataka o *beaconu*

Za svaki REST *endpoint* je u Swaggeru definirano u kojem obliku su podaci vraćeni (JSON ili XML) kao parametar za `produces`, koje ulazne parametre prima (zadan pod *parameters*), u kojem dijelu su oni dostavljeni (u samoj putanji do *endpointa* ili u *bodyu* poruke poslane serveru), te koji su odgovori servera mogući (navedeni u `responses`). Polje `operationId` definira metodu koja je zadužena za obrađivanje *requesta.* Primjer toga je prikazan u prethodnom isječku kôda.

Za uspješno terećenje korisnikove kartice prilikom plaćanja, sustav mora imati mogućnost komunikacije s *payment gateway* servisom u svrhu potvrđivanja transakcije. Komunikacija se odvija putem REST API-a koristeći klijentsku biblioteku *(engl. client library)* pomoću koje servis apstrahira komunikaciju između njihovog servera te „Conductor – Backend“ sustava.

U aplikaciju „Conductor – Mobile“ korisnik prilikom prvog plaćanja prijevozne karte unese kartične podatke nakon čega su ti podaci pohranjeni na *payment gateway* servisu i njima se pristupa pomoću jedinstvene šifre koju servis generira. Na taj način nije potreban ponovni unos kartice za sva naknadna plaćanja.

```
@requires_auth
def get_braintree():
    gateway = braintree.BraintreeGateway(
        braintree.Configuration(
            braintree.Environment.Sandbox,
            merchant_id=config["merchant_id"],
            public_key=config["public_key"],
            private_key=config["private_key"]
        )
    )
    odoo_api = OdooApi()
    user_id = odoo_api.get_user(connexion.request.authorization.username)
    customer_id = odoo_api.get_user_info_fromid(user_id)['card_token']
    if (customer_id == False):
        odoo_api.set_card_token(user_id, gateway.customer.create().customer.id)
    client_token = gateway.client_token.generate({"customer_id": customer_id})
    return UserBraintree(stripe_key=client_token)
```

Kôd 5. Kreiranje identifikatora za spremanje podataka o kartici na *payment gatewayu*

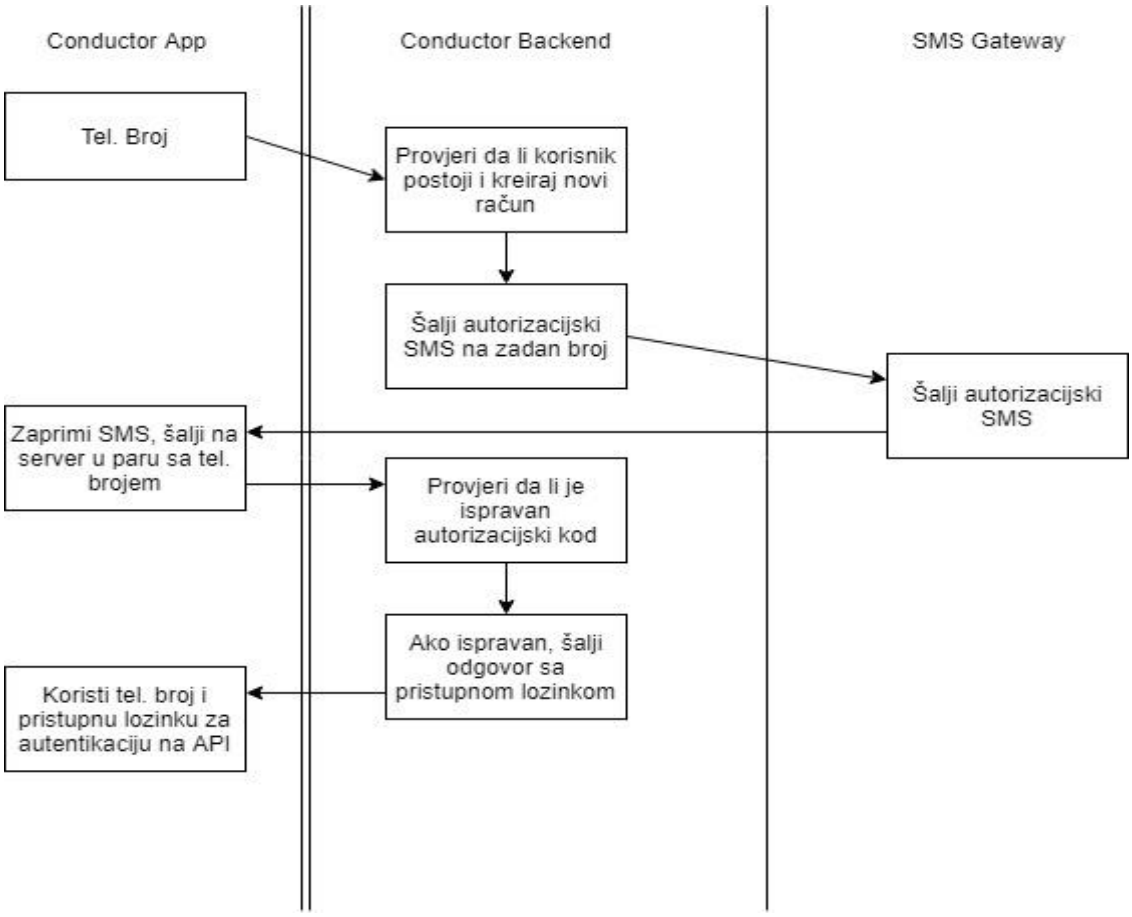
Metoda u isječku kôda 5 je osmišljena tako da u svakom pozivu vrati jedinstvenu šifru za plaćanje. Za lakšu interoperabilnost s Braintree sustavom potrebno je za svakog korisnika zatražiti unikatan identifikator od sustava, te ga pohraniti u bazi podataka. On je dohvaćen ili kreiran u varijablu `customer_id`. Identifikator je korišten za dohvaćanje jedinstvene šifre od servisa koja se vraća aplikaciji svaki put.

Za olakšanje komunikacije sa sustavom Odoo i korištenje XML-RPC API-a kreirana je `OdooApi` klasa koja sadrži sve metode potrebne u komunikaciji s Odooom. U ovom slučaju bile su korištene metode `get_user` , `get_user_info_fromid` i `set_card_token`. Prva metoda vrati `id` korisnika za dano korisničko ime korišteno u HTTP Basic autentikaciji, druga metoda dohvaća sve podatke koji su pohranjeni za danog korisnika, a zadnja metoda sprema Braintree identifikator u bazu podataka.

Budući da ne postoji unificiran sustav za upravljanje rasporedom javnog prijevoza, način komunikacije nije unaprijed definiran. Iz tog razloga postoji mogućnost izrade potrebnih API metoda za svakog operatera posebno. Alternativa API-u je korištenje *WebSocket* tehnologije za neposredno slanje promjena voznog reda u sustav „Conductor – Backend“.

Alternativa korištenju API-a ili *WebSocketa* je periodični uvoz podataka o voznom redu u „Conductor – Backend“ sustav ukoliko operater javnog prijevoza nema mogućnost implementacije komunikacije putem API-a ili *WebSocketa*.

Sve API metode koje nudi sustav „Conductor – Backend“ izuzevši one koje se koriste u Autorizacijskom toku zahtijevaju autorizaciju. U tu svrhu se koristi standard HTTP Basic u kojem se umjesto korisničkog imena unese telefonski broj, a za lozinku je korištena lozinka koju server kreira tijekom autorizacijskog toka. On je prikazan na sljedećoj slici iz koje je vidljivo što se događa u svakom koraku.



Slika 4. Autorizacijski tok

Ovaj način autorizacije je vrlo sličan onima koji su korišteni u popularnim aplikacijama poput WhatsAppa ili Vibera. S ovim pristupom korisnik ne mora pamtiti lozinke niti korisničko ime jer su oni pohranjeni u aplikaciji. Jedina odgovornost koju korisnik u ovom slučaju ima je zaštititi telefon od krađe.

```
def create(self, vals):
    auth_key = ''.join(random.choice('0123456789') for _ in range(6))
    vals['verify_code'] = auth_key
    returnval = super(User, self).create(vals)

    self.send_twilio_sms(vals['phone_number'],
       self.gen_twilio_msg(auth_key))

    return returnval
```

Kôd 6. Generiranje i slanje jednokratne autentikacijske šifre

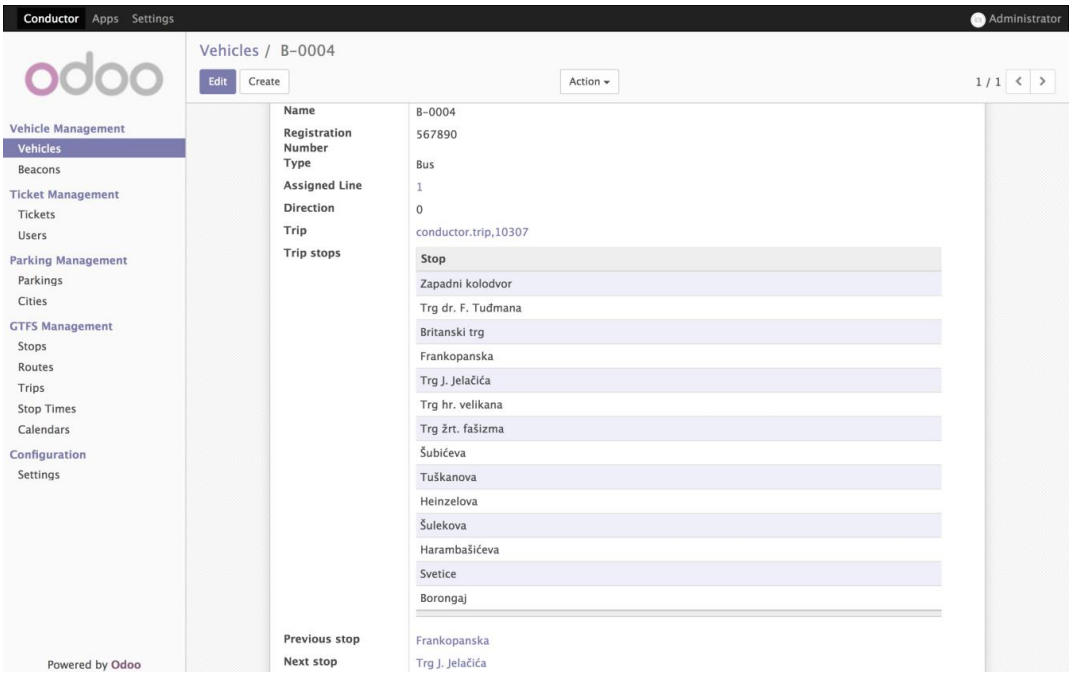
Za slanje jednokratne autentikacijske šifre je korišten servis Twilio za koji postoji biblioteka pisana u programskom jeziku Python korištena u metodi `send_twilio_sms`. Šifra za autentikaciju je generirana na prvoj liniji te je pohranjena u bazi podataka za kasnije korištenje u provjeri. Nakon pohrane se generira i šalje jednostavna poruka koja sadrži tu šifru.

Isječak kôda 6 prikazuje kako je šifra za autentikaciju generirana i pohranjena u varijablu `auth_key` na prvoj liniji koristeći *list comprehension,* koji kreira listu sa 6 nasumično izabranih brojeva pretvorenih u niz. Šifra je pohranjena u bazi podataka za kasnije korištenje u provjeri, nakon čega je korištena u generiranju i slanju jednostavne poruka koja sadrži istu.

2.3.1. Korištenje

Svaki podatak koji je korišten u radu sustava moguće je promijeniti i ažurirati kroz sučelje za upravljanje. Ono je podijeljeno u logičke kategorije prema podacima kojima raspolaže. Primjerice, pod kategorijom *Vehicle Management* nalaze se sučelja *Vehicles* te *Beacons*.

Primarna uloga *bluetooth beacona* je detekcija i identifikacija vozila. Sučelje za upravljanje istima nalazi se u kategoriji sa sučeljem za upravljanje vozilima. Iako su *bluetooth beaconi* korišteni za detekciju i identifikaciju stajališta i vozila, njihovo sučelje za upravljanje nema zasebnu kategoriju. Sličan primjer takvog grupiranja je kategorija *Ticket Management* u kojoj se nalaze sučelja za upravljanje kupljenim kartama te korisnicima.



Slika 5. Sučelje za upravljanje vozilima

Sučelje „Conductor – Backenda“ slijedi stalan oblik u kojem se izbornik nalazi na lijevoj stani, a podaci na desnoj. Primarni način prikaza svih zapisa je u popisu u kojem je moguće izabrati individualni zapis te otvoriti formu za uređivanje istog.

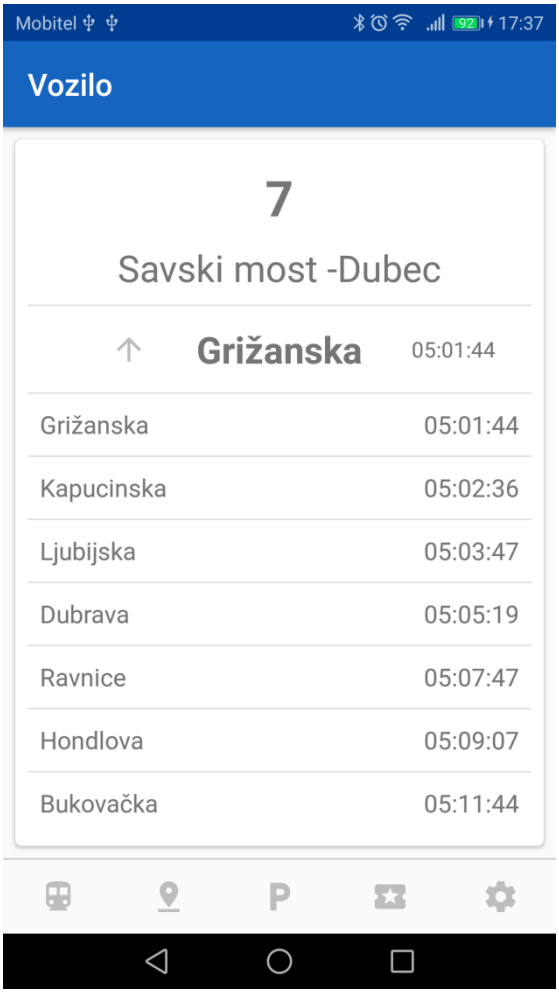
Prikaz forme slijedi oblik koji je prikazan na Slici 5.

- Vehicle Management
  - *Vehicles* - Popis svih vozila kojima raspolaže upravitelj javnog prijevoza. Ovdje se svakom vozilu dodjeli *trip* koji je zakazan po rasporedu. Primjerice, vozilo ima zakazan put s identifikatorom „100500“ koji je na liniji 257 u smjeru s oznakom 1. Ta oznaka služi za određivanje redoslijeda postaja na liniji, odnosno smjer u kojem vozilo putuje linijom.

        - *Beacons* – Popis svih *bluetooth beacona* koji postoje u sustavu. Putem ovog sučelja upravlja se njihovim identifikatorima te dodjeljuju postaje i vozila.
- Ticket Management
    - *Tickets* - Popis svih kupljenih karata. Prema potrebi moguće je ručno kreirati prijevoznu kartu. Sučelje više postoji u svrhu otkrivanja i rješavanja mogućih problema, primjerice pogrešnog izračuna cijene karte.
    - Users - Popis svih registriranih korisnika. Najviše poteškoća se očekuje pri registraciji i prijavi u aplikaciju. Zbog toga je predviđena mogućnost ručnog kreiranja korisničkog računa te ovjere putem telefona, ukoliko korisnik nije u mogućnosti putem SMS-a zaprimiti kôd za ovjeru.
- Parking Management
    - *Cities* - Zbog načina na koji su podaci strukturirani u sustavu Conductor, moguće je jednu instancu sustava primijeniti za upravljanje parkinga u više gradova.
    - *Parkings* - Upravljanje svim parkirnim zonama koje se vode kroz sustav Conductor.
- *GTFS Management* - Komunikacija i interoperabilnost sustava Conductor s vanjskim sustavima za upravljanje rasporedom javnog prijevoza u pravilu teče automatski. Međutim, u nekim nepredvidljivim okolnostima tijekom rada sustava bit će potrebna ručna intervencija za ispravljanje podataka. Imena sučelja su u direktnoj korelaciji s imenima tablica definiranima u GTFS formatu.
    - *Stops*
    - *Routes*
    - *Trips*
    - *Stop Times*
    - *Calendars*
- *Configuration* – Bilo koji konfiguracijski parametri potrebni za rad sustava. Trenutno sadrži samo ključeve za ovjeru na vanjskim servisima poput *payment gatewaya*, koji su navedeni u prethodnim poglavljima.

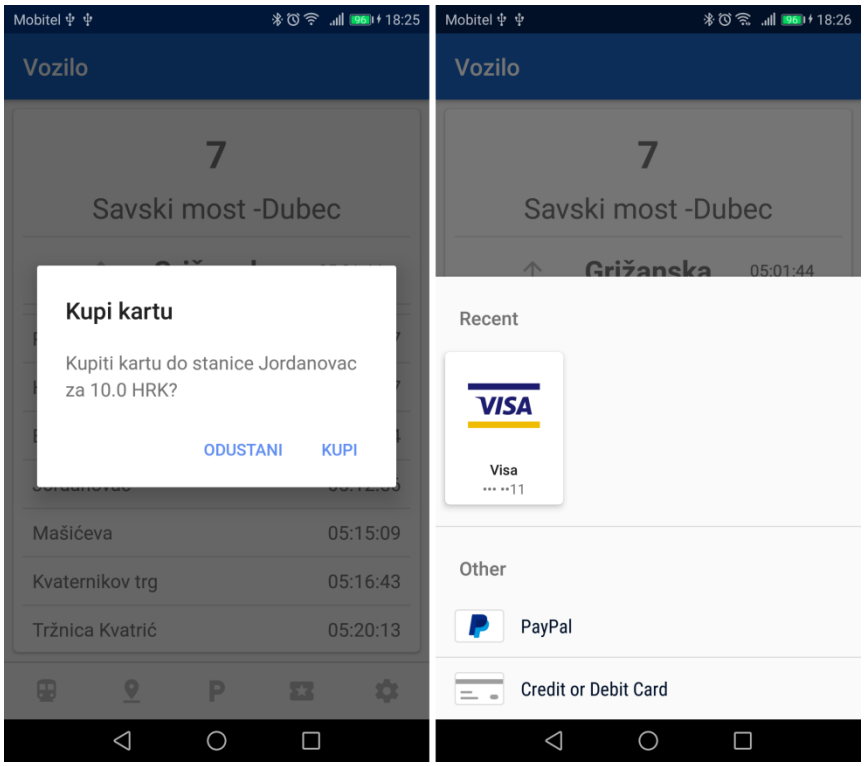
3. „Conductor– Mobile“

Sučelje aplikacije je osmišljeno i dizajnirano u svrhu što lakšeg korištenja.



Slika 6. Popis nadolazećih postaja za vozilo

Nakon učitavanja podataka o prijevoznoj liniji za vozilo, korisnik mora samo izabrati postaju do koje želi putovati i potvrditi plaćanje. Slika 6 prikazuje sučelje za prikaz nadolazećih postaja, te vremena stajanja.



Slika 7. Potvrđivanje kupnje i plaćanje prijevozne karte

Aplikacija podatke s kartice sprema u *payment gatewayu*. BrainTree (*payment gateway* korišten u aplikaciji) u tu svrhu koristi takozvani *client token*, koji je za svakog korisnika na „Conductor – Backendu“ kreiran prilikom prvog plaćanja. Taj *client token* je zatim korišten za svako plaćanje.

Sučelje za plaćanje prijevoznih karata je prikazano na slici 7, na lijevoj strani slike je prikazan dijalog u kojem korisnik prvo potvrdi odabir postaje, a na desnoj strani je prikazan izbornik za biranje kartice koja će biti korištena za terećenje.

```
private void selectPaymentMethod(String clientToken) {
    DropInRequest dropInRequest = new DropInRequest().clientToken(clientToken);
    startActivityForResult(dropInRequest.getIntent(getContext()), PAYPAL_DROPIN_PICKER);
}
```

Kôd 7. Početak BrainTree *payment flowa*

Braintree omogućava vrlo laku integraciju s Android sustavom koristeći `Activity` objekt operacijskog sustava. Jedino što zahtijeva je jednokratnu šifru generiranu na serveru za korisnika u obliku parametra, koja je onda dalje korištena za dohvaćanje spremljenih podataka kartice, te terećenje iste. Nakon odabira kartice i uspješnog plaćanja aplikacija zove metodu `finishTicketPurchase`.

```
public void finishTicketPurchase(DropInResult result) {
    showSpinner();
    String paymentMethodNonce = result.getPaymentMethodNonce().getNonce();
    TicketConfirmBuyBody ticketConfirmBuyBody = new TicketConfirmBuyBody()
            .ticketId(ticketPurchase.getId())
            .paymentMethodNonce(paymentMethodNonce);
    ticketsApi.postConfirmBuyTicket(ticketConfirmBuyBody)
        .enqueue(new Callback<GenericResponse>() {
        @Override
        public void onResponse(Call<GenericResponse> call,
                               Response<GenericResponse> response) {
            hideSpinner();
            if (response.isSuccessful()) {
                String msg = String.format(
                                "Karta kupljena za %s HRK",
                                ticketPurchase.getPrice())
                showToast(true, msg);
            } else {
                showToast(true, "Greška pri kupnji");
            }
            ticketPurchase = null;
        }

        @Override
        public void onFailure(Call<GenericResponse> call, Throwable t) {
            hideSpinner();
            Log.e(TAG, "sendTokenPurchaseNonce.onFailure: " + t.getMessage());
            showToast(true, "Greška pri kupnji");
            ticketPurchase = null;
        }
    });
}
```

Kôd 8. Slanje jednokratnog *payment tokena* na API

Opisana metoda je zadužena za terećenje kartice korisnika za naplatu prijevozne karte. U prvom dijelu kôda kreira se *request* objekt koji će biti poslan na server, te sadrži potrebne parametre za REST metodu koju zove (identifikator prijevozne karte te jednokratna šifra za terećenje korisnikove kartice).

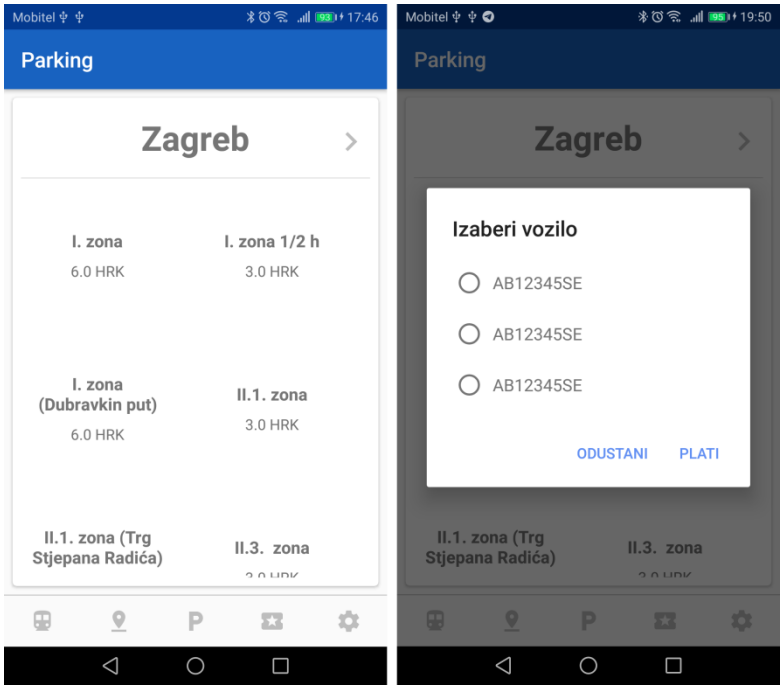
Objekt `Callback<GenericResponse>` zadan kao parametar `enqueue` metode je zadužen za zaprimanje odgovora servera te obavještavanje korisnika o uspješnosti plaćanja. Za to je potrebno u `onResponse` metodi provjeriti je li poziv uspješan (HTTP statusni kôd koji označava uspješnost procesiranja zahtjeva) pozivom metode `response.isSuccessful()`. Isto tako nužno je definirati ponašanje aplikacije ukoliko poziv nije uspješan metodom `onFailure`.

Korisnik vrlo jednostavno može vidjeti informacije o nadolazećim vozilima za bilo koju postaju na kojoj se nalazi.

Na slici ispod je prikazan popis svih stajanja za iduća 24 sata.

Slika 8. Informacije o liniji za stajalište

Plaćanje parkinga je jednako jednostavno. Korisnik izabere grad i zonu za koju želi platiti parking. U dijalogu izabere registarsku oznaku vozila i potvrdi plaćanje. Korisničko sučelje za te dvije radnje je prikazano na idućoj slici. Na lijevoj polovici je prikazan izbornik parkirnih zona, a na desnoj je prikazan izbornik registracijskih oznaka.



Slika 9. Plaćanje parkinga

Nakon pritiska na gumb „plati“ aplikacija u pozadini pošalje SMS koji sadrži registracijsku oznaku vozila na odgovarajući telefonski broj za odabranu parkirnu zonu.

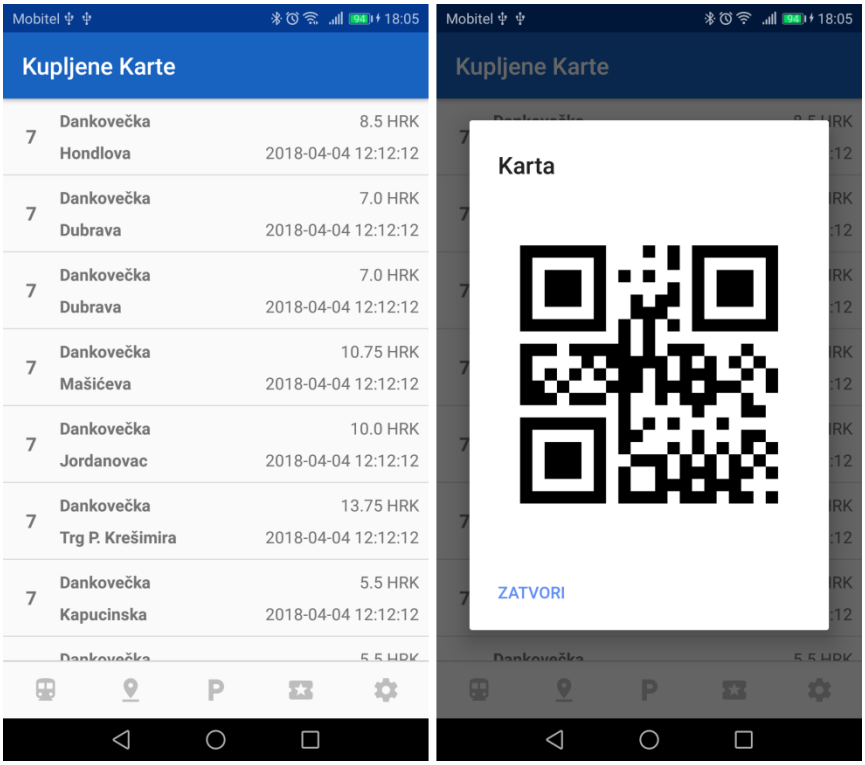
```
private void sendParkingSMS(String licencePlate, String parkingPhone) {
    SmsManager smsManager = SmsManager.getDefault();
    String normalised = licencePlate.replaceAll("[^a-zA-Z0-9]", "").toUpperCase();

    smsManager.sendTextMessage(parkingPhone, null, normalised, null, null);

    showToast(true, "Parking for " + licencePlate + " bought.");
}
```

Kôd 9. Slanje SMS-a

Kôd za slanje SMS-a je vrlo jednostavan jer se operacijski sustav pobrine za sve. Potrebno je samo zadati broj telefona na koji će se poruka slati te njen sadržaj. Sustav za naplatu parkinga u Zagrebu očekuje registracijsku oznaku bez ikakvih dodatnih simbola osim slova i brojeva za što se brine 3. linija kôda u kojoj je korišten *regex* za uklanjanje svih znakova koji nisu brojevi ili slova. Nakon normalizacije se rezultat koristi kao sadržaj poruke.

U aplikaciji je moguće vidjeti sve karte koje je korisnik ikada kupio, te za svaku prikazati QR kôd ukoliko je potrebna kontrola kupljene karte. U QR kôdu je zapisan jedinstven identifikator s kojim se pristupa do svih podataka o prijevoznoj karti.

U slici 10 na lijevoj strani je predstavljen prikaz popisa svih kupljenih karata te osnovne informacije o istima, a na desnoj strani je prikazano kako izgleda dijalog s QR kôdom korišten prilikom kontrole prijevozne karte.



Slika 10. Pregled kupljenih prijevoznih karata

Zaključak

Sustav opisan u završnom radu je samo prototip kojim se demonstrira jedan od mogućih načina nadogradnje postojećeg sustava naplate javnog prijevoza. On je izrađen s tehnologijama i metodama koje su danas standardne u informatičkoj industriji za slična rješenja, što omogućava lakši i brži razvoj sustava koji bi bio korišten u budućnosti te stabilniji rad istog.

Jedno od mogućih ograničenja sustava je nepristupačnost korisnicima javnog prijevoza koji nemaju pametne telefone i starijim osobama koje nisu upoznate s novijim tehnologijama. Mogući način za rješavanje tog problema je da novi sustav ne zamijeni stari u potpunosti nego ga komplimentira.

Prikazan koncept sustava je moguće vrlo lako nadograditi i unaprijediti prije puštanja u pogon, te prilagoditi za specifične potrebe svakog operatera javnog prijevoza.

Vizija ovog projekta je da će jednog dana u svim većim gradovima diljem Europe ovakvi sustavi biti dostupni i međusobno umreženi. Na taj način bi korisnik na svom uređaju mogao imati instaliranu i podešenu jednu aplikaciju koja bi radila svugdje, neovisno o gradu i državi.

Popis kratica

API	*Application Programming Interface*
FTP	*File Transfer Protocol*
GTFS	*General Transit Feed Specification*
HTTP	*Hypertext Transfer Protocol*
HTTPS	*Hypertext Transfer Protocol Secure*
JSON	*JavaScript Object Notation*
REST	*Representational State Transfer*
RSSI	*Received signal strength indication*
UUID	*Universally Unique Identifier*
WSDL	*Web Services Description Language*
XML	*Extensible Markup Language*
ZET	*Zagrebački Električni Tramvaj*

Popis slika

Slika 1. Relacije među tablicama 5

Slika 2. Primjer postavljanja *beacona* te konfiguracije dometa 10

Slika 3. Shema sustava i komunikacije s vanjskim servisima 14

Slika 4. Autorizacijski tok 18

Slika 5. Sučelje za upravljanje vozilima 20

Slika 6. Popis nadolazećih postaja za vozilo 22

Slika 7. Potvrđivanje kupnje i plaćanje prijevozne karte 23

Slika 8. Informacije o liniji za stajalište 25

Slika 9. Plaćanje parkinga 26

Slika 10. Pregled kupljenih prijevoznih karata 27

Popis kôdova

Kôd 1. Metoda za odabir najbližeg *beacona* 7

Kôd 2. Dohvaćanje podatka asociranog s *beaconom* i prosljeđivanje istog..... 8

Kôd 3. Detektiranje *beacona* i dohvaćanje UUID identifikatora za isti..... 12

Kôd 4. Primjer Swagger definicije REST metode za dohvaćanje podataka o *beaconu*..... 15

Kôd 5. Kreiranje identifikatora za spremanje podataka o kartici na *payment gatewayu* 16

Kôd 6. Generiranje i slanje jednokratne autentikacijske šifre..... 18

Kôd 7. Početak BrainTree *payment flowa*..... 23

Kôd 8. Slanje jednokratnog *payment tokena* na API..... 24

Kôd 9. Slanje SMS-a..... 26

Literatura

1. General Transit Feed Specification: https://en.wikipedia.org/wiki/General_Transit_Feed_Specification 24.05.2018.
2. GTFS Static Overview: https://developers.google.com/transit/gtfs/ 24.05.2018.
3. Dijagram GTFS klasa https://upload.wikimedia.org/wikipedia/commons/2/28/GTFS_class_diagram.svg 24.05.2018
4. Bluetooth low energy beacon: https://en.wikipedia.org/wiki/Bluetooth_low_energy 28.05.2018.
5. Universally Unique Identifier: https://en.wikipedia.org/wiki/Universally_unique_identifier 25.05.2018
6. Eddystone (Google): https://en.wikipedia.org/wiki/Eddystone_(Google) 29.05.2018.
7. Eddystone Format: https://developers.google.com/beacons/eddystone 29.05.2018.