SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

DIPLOMSKI RAD br. 1790

SUSTAV ZA AUTOMATSKO PRAĆENJE I LOCIRANJE KUĆNIH LJUBIMACA

Bernard Radman

Zagreb, veljača 2019

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA
ODBOR ZA DIPLOMSKI RAD PROFILA

Zagreb, 5. listopada 2018.

DIPLOMSKI ZADATAK br. 1790

Pristupnik: **Bernard Radman (0036473424)**
Studij: Računarstvo
Profil: Računalno inženjerstvo

Zadatak: **Sustav za automatsko praćenje i lociranje kućnih ljubimaca**

Opis zadatka:

U okviru diplomskog rada potrebno je prvo istražiti komercijalne sustave koji se koriste za praćenje kućnih ljubimaca (pasa i mačaka), te obavještavanje vlasnika ako životinja izađe izvan dopuštene zone kretanja. Nadalje, potrebno je istražiti mogućnosti izgradnje takvog sustava modifikacijom i programskim proširenjem standardnih mobilnih platformi (pametnih mobilnih telefona) u svrhu korištenja svih ključnih funkcionalnosti koje takve platforme osiguravaju. To se prvenstveno odnosi na geolociranje bilo korištenjem mobilne mreže i baznih stanica, odnosno korištenjem ugrađenog GPS prijamnika. Istražiti mogućnost daljinskog upravljanja ovog mobilnog uređaja za praćenje, prvenstveno radi praćenja statusa napunjenosti baterije i upravljanja potrošnjom energije. Osmisliti arhitekturu takvog sustava, te predložiti poslužiteljsko i klijentsko programsko rješenje korištenjem suvremenih web okolina i alata.

Zadatak uručen pristupniku: 12. listopada 2018.
Rok za predaju rada: 8. veljače 2019.

Mentor:
Prof. dr. sc. Davor Petrinović

Djelovođa:
Prof. dr. sc. Danko Basch

Predsjednik odbora za diplomski rad profila:
Prof. dr. sc. Mario Kovač

Sadržaj

1 Uvod

Suvremeni način života i ubrzani razvoj tehnologije omogućio je ljudima lagodan život. No u doba kada tehnologija strelovito napreduje, jedna stvar je ostala ista: naša ljubav prema kućnim ljubimcima. Stoga je danas na tržištu moguće pronaći veliki broj raznih uređaja koji omogućavaju lociranje i lakši pronalazak životinja ukoliko se one izgube kako bismo bezbrižno mogli uživati u šetnjama po prirodi ili pak gradskom parku. Neki od takvih uređaja bazirani su na detekciji i slanju GPS lokacije te omogućavaju veliku preciznost i gotovo sa sigurnošću garantiraju pronalazak izgubljenoga ljubimca.

Predviđa se da će u 2019. godini broj korisnika mobilnih uređaja doseći 5 milijardi, od čega su otprilike 50% korisnici pametnih telefona [1]. Ako pretpostavimo da se pametni telefoni zamjenjuju novijim uređajima u prosjeku svake dvije godine i u obzir uzmemo činjenicu da je u 2017. godini prodano oko 1.54 milijarde pametnih telefona [2], dolazimo do zaključka da je isto tako veliki broj starih uređaja koji se prestanu koristiti i odbace. Naravno da su razlozi za zamjenu mobitela brojni, od toga da je uređaj razbijen, prestao raditi zbog pada u vodu i slično, no veliki broj uređaja koji se odbacuju je i dalje funkcionalan ali jednostavno više nije dorastao potrebama korisnika.

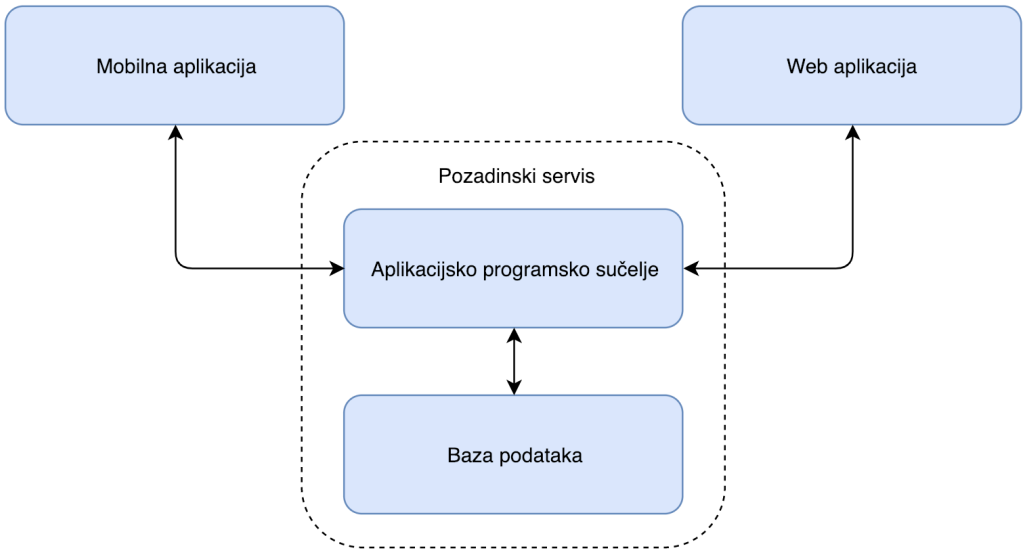
Kako bi se podario novi život starim pametnim telefonima, potrebno ih je iskoristiti za neke druge, manje zahtjevne svrhe. S tom idejom i željom za skrb prema kućnim ljubimcima pisan je i ovaj diplomski rad. U sklopu rada biti će opisan način na koji je moguće razviti sustav za praćenje lokacije kućnih ljubimaca temeljen na sustavu Android, te će se razvijeni sustav usporediti s već postojećim uređajima na tržištu.

2 Opis razvijenoga sustava

U svrhu ovog diplomskog rada izrađene su tri aplikacije koje zajedno služe za kontrolu lokacije kućnih ljubimaca. Navedene aplikacije su:

- aplikacija za mobilne uređaje temeljene na sustavu Android koja služi za slanje lokacije i stanja baterije
- pozadinski servis za primanje podataka, obradu i spremanje u bazu podataka
- web aplikacije za prikaz podataka

U nastavku su detaljno objašnjene sve tri aplikacije, a na slici broj 1 vidljiva je skica arhitekture sustava.



Slika 1. Skica arhitekture sustava

2.1 Pozadinski servis

Pozadinski servis (engl. *backend service*) je temeljna aplikacija ovog sustava. Uloga ove aplikacije je primanje podataka o lokaciji i stanju baterije od mobilne aplikacije i spremanje istih u bazu podataka, ali ujedno služi i za slanje podataka prema korisničkoj aplikaciji na zahtjev. Za izgradnju servisa korišten je *Node.js*, poznat još kao i *Node*. *Node* je poslužiteljsko okruženje otvorenog koda (engl. *open source server environment*), kojeg pokreću asinkroni JavaScript događaji, a dizajniran je za izgradnju skalabilnih mrežnih aplikacija [3]. Više istovremenih zahtjeva *Node* može paralelno obrađivati, a ukoliko nema novih zahtjeva za obradu, *Node* odlazi u stanje mirovanja (engl. *sleep*). Kao proširenje na *Node* korišten je *Express*, minimalistički i fleksibilni programski okvir (engl. *framework*) koji pruža robustan skup značajki kako za web, tako i za mobilne aplikacije [4]. *Express* je korišten za izradu aplikacijskog programskog sučelja (engl. *application programming interface, API*) kojem pristupaju mobilna i korisnička web aplikacija. Pomoću *Expressa* se deklariraju rute koje označavaju kako krajnje točke aplikacije, odnosno jedinstveni identifikatori resursa (engl. u*niform resource identifier*, *URI*), odgovaraju na zahtjeve klijenata [5]. Generalni oblik ruta je:

app.METHOD(PATH, HANDLER),

gdje je:

- *app* – instanca *Expressa*
- *METHOD* – HTTP zahtjev, napisan malim slovima
- *PATH* – putanja do resursa
- *HANDLER* – funkcija koja se izvede kada se ruta podudara s traženom

Korištenjem modula za *Node* zvanog „*express-generator*" generiran je kostur aplikacije [6]. Zatim su na kostur dodane sve potrebne rute. Kako bi se detaljnije objasnile pojedine rute korištene u izrađenom servisu, najprije je potrebno objasniti korištene modele i bazu podataka. Za spremanje podataka korišten je *MongoDB* kao baza podataka. *MongoDB* je dokumentno orijentirani program za baze podataka koji koristi dokumente u *JSON* formatu zapisa za spremanje [7].

MongoDB je program sam za sebe, te kako bi se ostvarila komunikacija između servisa i baze podataka, potreban je još jedan alat. Upravo tome služi *Mongoose*, alat za objektno-relacijsko mapiranje (engl. *object-relational mapping*, *ORM*) za *Node*. *Mongoose* nudi jednostavna rješenja bazirana na shemi za modeliranje podataka aplikacije, uključujući ugrađenu pretvorbu tipova, validaciju, izgradnju upita, i još mnogo toga [8]. U nastavku je prikazan isječak koda sheme koja se koristi kao model za spremanje podataka lokacije i stanja baterije ovog sustava.

```
const mongoose = require("mongoose");
const Schema = mongoose.Schema;
// Create schema and model
const RecordSchema = new Schema({
  timestamp: { type: String, required: true },
  batteryPercentage: { type: Number, required: true },
  latitude: { type: Number, required: true },
  longitude: { type: Number, required: true },
  isGPSLocation: { type: Boolean, required: true },
  comment: String
});
const Record = mongoose.model("record", RecordSchema);
module.exports = Record;
```

Isječak koda 1. RecordShema

Konstanta *RecordSchema* je instanca *Mongoose* sheme koja opisuje koje će sve atribute imati pojedini zapis u bazi podataka:

- *timestamp* – datum i vrijeme kada je zapis kreiran, zapisano u ISO formatu
- *batteryPercentage* – vrijednost postotka baterije u trenutku slanja zapisa
- *latitude* – geografska širina
- *longitude* – geografska visina
- *isGPSLocation* – *boolean* vrijednost je li primljeni podatak veće, odnosno GPS preciznosti (*true* ako jest, *false* ako nije)
- *comment* – dodatno polje koje omogućuje naknadni zapis komentara

Nakon što se definira shema, pomoću nje se gradi model *Record* koji se dalje koristi u aplikaciji. Na sličan način definira se shema i model koji se koristi za *push* obavijesti kojima se od mobilne aplikacije zahtijeva lokacija veće preciznosti, odnosno ona dobivena pomoću GPS prijamnika.

```
const mongoose = require("mongoose");
const Schema = mongoose.Schema;
// Create schema and model
const PushTokenSchema = new Schema({
  token: { type: String, required: true },
  username: { type: String, required: true }
});
const PushToken = mongoose.model("pushToken", PushTokenSchema);
module.exports = PushToken;
```

Isječak koda 2. PushTokenShema

Ova shema sadrži samo dva atributa:

- *token* – token koji se automatski generira pri prvom pokretanju aplikacije te pošalje pozadinskom servisu koji ga registrira
- *username* – korisničko ime koje je pridjeljeno tokenu

Kada smo definirali model zapisa, možemo ga iskoristiti u rutama *Expressa* za obradu korisničkih zahtjeva. Najjednostavniji primjer rute je zahtjev za dohvaćanjem svih zapisa iz baze podataka. Sljedeći isječak koda upravo je ta ruta.

```
/* GET all */
router.get("/", (req, res, next) => {
  Record.find({})
    .sort({ timestamp: -1 })
    .then(doc => res.json(doc))
    .catch(err => res.status(500).json(err));
});
```

Isječak koda 3. Obrada GET zahtjeva na rutu "/"

Ova ruta i njezina pripadajuća funkcija pozivaju se na HTTP GET zahtjev na sami korijen (engl. *root*) aplikacije, odnosno na putanju „/”. Sve što ova ruta odrađuje jest da pomoću unaprijed definiranog modela dohvaća sve zapise, poredane od najnovijeg prema najstarijem i prosljeđuje te podatke korisničkoj aplikaciji koja ih je zatražila. Isječak koda 4 prikazuje drugu osnovnu rutu, onu za stvaranje novih zapisa podataka u bazi.

```
/* POST */
router.post("/", (req, res, next) => {
  if (Object.keys(req.body).length === 0) {
    return res.status(400).send("Request body is missing");
  }
  /* Add received time as timestamp if one is not included in request */
  let record = new Record({
    ...req.body,
    timestamp: req.body.timestamp || new Date().toISOString()
  });
  record
    .save()
    .then(doc => {
      if (!doc || doc.length === 0 || record.isNew) {
        return res.status(501).send(doc);
      }
      res.status(201).send(doc);
    })
    .catch(err => {
      console.error("error", err);
      res.status(500).json(err);
    });
});
```

Isječak koda 4. Obrada POST zahtjeva na rutu "/"

U njoj se obrađuju PUSH HTTP zahtjevi koji dolaze na *root* rutu. U njoj se najprije provjerava postoji li tijelo zahtjeva, zatim se doda atribut o datumu i vremenu prihvata zahtjeva ukoliko se isti već ne nalazi u njemu. Nakon toga se zapis spremi u bazu podataka, i ukoliko je sve prošlo u redu, korisniku se pošalje HTTP zahtjev sa statusnim kodom 201 i isti taj podatak. Inače, pošalje se zahtjev sa kodom 500 koji označava da je došlo do pogreške u sustavu.

Budući da mobilna aplikacija razlikuje slanje podataka s manjom preciznosti lokacije od one s većom preciznosti, ona je zadužena za postavljanje zastavice radi li se o GPS lokaciji ili ne, a u oba slučaja se koristi ista ruta na pozadinskom servisu za spremanje novih zapisa. No postoji razlika u dohvatu podataka. Za dohvaćanje podataka s GPS lokacijom koristi se ruta prikaza na isječku koda 5.

```
/* GET only GPS locations */
router.get("/GPS", (req, res, next) => {
  Record.find({ isGPSLocation: true })
    .sort({ timestamp: -1 })
    .then(doc => res.json(doc))
    .catch(err => res.status(500).json(err));
});
```

Isječak koda 5. Obrada GET zahtjeva na rutu "/GPS"

Ova ruta je vrlo slična onoj za dohvat svih podataka, razlika je u samo u tome što se filtriraju i vraćaju samo oni podatci čija je zastavica *isGPSLocation* postavljena u *true*, logičnu jedinicu. Budući da svaki zapis u *MongoDB* bazi podataka pri upisu automatski dobiva atribut *_id*, pomoću njega je moguće pronaći pojedine zapise i promijeniti im određene vrijednosti ili čak kompletno ih obrisati. U svrhu ostvarivanja pravog *CRUD* (engl. *Create, Read, Update, Delete*) sustava, izgrađene su još dvije rute koje omogućuju rad s pojedinim zapisima. To su ruta za brisanje cijelog zapisa i ruta za izmjenu podataka zapisa. Prikazane su na isječcima koda 6 i 7.

```
/* DELETE by _id */
router.delete("/:id", (req, res, next) => {
  Record.findByIdAndRemove(req.params.id, (error, result) => {
    if (error) {
      res.status(400).json(error);
    } else {
      res.json(result);
    }
  });
});
```

Isječak koda 6. Obrada DELETE zahtjeva na rutu "/:id"

```
/* UPDATE by _id */
router.put("/:id", (req, res, next) => {
  Record.findByIdAndUpdate(
    req.params.id,
    { ...req.body },
    { new: true },
    (error, result) => {
      if (error) {
        res.status(500).json(error);
      } else {
        res.json(result);
      }
    }
  );
});
```

Isječak koda 7. Obrada PUT zahtjeva na rutu "/:id"

Preostale se još samo ruta za registraciju *push* tokena i ruta čijim se pozivom zahtjeva slanje *push* obavijesti prema mobilnoj aplikaciji. Ruta za registraciju *push* tokena prikazana je na isječku koda 8.

```
/* POST create push token for push notifications */
router.post("/push-token", (req, res, next) => {
  if (Object.keys(req.body).length === 0) {
    return res.status(400).send("Request body is missing");
  }

  PushToken.findOneAndUpdate(
    { ...req.body },
    { ...req.body },
    { upsert: true },
    (error, result) => {
      if (error) {
        res.status(500).json(error);
      } else {
        res.json(result);
      }
    }
  );
});
```

Isječak koda 8. Obrada PUSH zahtjeva na rutu "/push-token"

HTTP POST zahtjevom na putanju „*/push-token*” šalje se token u tijelu zahtjeva. Uz pomoć metode *findOneAndUpdate* nad modelom *PushToken*, ostvaruje se ažuriranje tokena ali ujedno i stvaranje novog zapisa ako on već ne postoji tako što se za opcije postavi zastavica *upsert*. Ruta za zahtjev GPS lokacije malo je složenija. Njena struktura prikazana je na isječku koda 9.

```
/* POST send push notification to android  */
router.get("/requestGPSLocation", async (req, res, next) => {
  let messages = [];
  await PushToken.find({})
    .select({ token: 1, _id: 0 })
    .then(items =>
      items.map(item => {
        if (!Expo.isExpoPushToken(item.token)) {
          console.error(
            `Push token ${item.token} is not a valid Expo push token`
          );
        } else {
          messages.push({
            to: item.token,
            sound: "default",
            body: "GPS location requested",
            data: { GPSLocation: true }
          });
        }
      })
    );
  expo.sendPushNotificationsAsync(messages);
  res.status(200).send("Requested");
});
```

Isječak koda 9. Obrada GET zahtjeva na rutu "/requestGPSLocation"

Najprije se iz baze podataka dohvaćaju svi *push* tokeni, zatim se za svaki dohvaćeni token provjerava njegova ispravnost pomoću modula *Expo*, o kojem će biti riječ poslije, te ukoliko je token ispravan dodaje se u red poruka. Zatim se prema svim ispravnim tokenima šalje *push* obavijest, opet uz pomoć *Expo* modula, a korisničkoj aplikaciji se vrati poruka o uspjehu poslanih obavijesti. Ovo je asinkrona radnja, te nakon što se zatraži GPS lokacija, korisnik ne može znati kada će se uistinu zatraženi podatak vratiti prema servisu, već može nakon nekog kratkog vremena

ponovno zatražiti sve podatke i nadati se da je podatak s GPS lokacijom već pristigao. Ukoliko nije, preostaje mu sačekati, te probati ponovno nakon nekog vremena. U trenutnoj izvedbi servisa ne postoji nikakva identifikacija i mogućnost za više korisnika, već se sve temelji na jednom korisniku i jednoj mobilnoj aplikaciji koja generira token. Proširenje sustava na više korisnika je ostavljeno za daljnji rad, te se vrlo lako može ostvariti kao dodatni sloj u aplikaciji.

2.2 Korisnička web aplikacija

Korisnička aplikacija zamišljena je kao aplikacija pomoću koje korisnik ima uvid u sve zapise u bazi podataka, kao i mogućnost uređivanja i brisanja pojedinih zapisa. Osim toga, korisniku je omogućeno i slanje zahtjeva za GPS lokacijom. Izrađena je JavaScript aplikacija uz pomoć *Reacta*. *React* je JavaScript biblioteka za izgradnju korisničkih sučelja razvijena i održavana od strane Facebooka [9]. *React* se koristi za izradu web aplikacija na jednoj stranici čiji se sadržaj dinamički mijenja u interakciji s korisnikom, umjesto da se svaki put dohvaća nova stranica. Osim toga, uz pomoć *Reacta* je moguće izrađivati i mobilne aplikacije. Za izradu kompleksnijih *React* aplikacija potrebne su dodatne biblioteke za upravljanje stanjem, usmjeravanje i interakciju s API-jem. Za izradu kostura aplikacije iskorišten je alat *Create React App*. To je službeno podržani alat za kreiranje *React* aplikacija koji nudi postavljanje radne okoline bez nepotrebnih dodatnih konfiguracija [10]. Uz pretpostavku da je na računalu već instaliran *Node*, potrebna je svega jedna naredba u konzoli kako bi se postavila radna okolina spreman za rad:

npx create-react-app my-app,

gdje je *my-app* ime za željenu aplikaciju.

U generiranoj aplikaciji ne postoji biblioteka za vizualno ljepši prikaz elemenata, te se uobičajeno koriste druge biblioteke od kojih je najpopularnija *Bootstrap*. U izradi ovog rada korištena je biblioteka *Black Dashboard React* koja u pozadini koristi upravo *Bootstrap 4* [11]. *Black Dashboard React* sadrži samo gotove elemente za prikaz te je na programeru da razvije svu logiku iza njih.

Osnovna komponenta razvijene aplikacije nalazi se u *App.js* datoteci u korijenskom direktoriju. U njoj se nalaze komponente za iniciranje stanja (engl. *state*), postavke za usmjeravanje i komponenta zadužena za prikaz obavijesti. Za sve rute definira se raspored sadržan u *layouts/Admin.jsx* datoteci, a ruta „/" preusmjerava se na „/dashboard" čime se prikazuje početni zaslon. Isječak koda 10 prikazuje isječak iz *App.js* datoteke.

```
class App extends Component {
  render() {
    const { store } = this.props;
    return (
      <Provider store={store}>
        <BrowserRouter>
          <Switch>
            <Redirect exact from="/" to="/dashboard" />
            <Route path="/" render={props => <AdminLayout {...props} />} />
          </Switch>
        </BrowserRouter>
        <ToastContainer autoClose={5000} hideProgressBar={true} />
      </Provider>
    );
  }
}
```

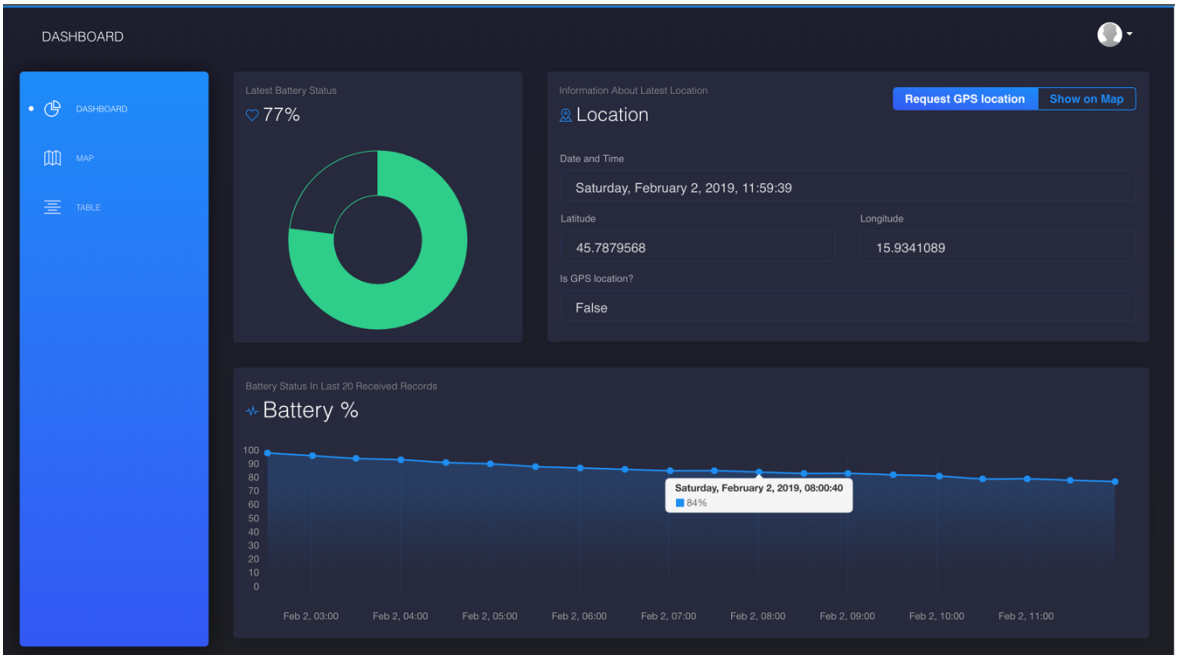
Isječak koda 10. Komponenta App.js

Isječak koda broj 11 sadrži dio koda iz datoteke *layouts/Admin.jsx* u kojem se vidi da su svim zaslonima zajednička bočna traka za navigaciju i gornja administratorska traka, te da se samo mijenja sadržaj unutar elementa razreda „*main-panel*" uz pomoć komponente *Switch* koja dinamički mijenja prikazani sadržaj u ovisnost o ruti koja se zahtjeva.

```
render() {
  return (
    <Fragment>
      <div className="wrapper">
        <Sidebar
          {...this.props}
          routes={routes}
          bgColor={this.state.backgroundColor}
          toggleSidebar={this.toggleSidebar}
        />
        <div
          className="main-panel"
          ref="mainPanel"
          data={this.state.backgroundColor}
        >
          <AdminNavbar
            {...this.props}
            brandText={this.getBrandText(this.props.location.pathname)}
            toggleSidebar={this.toggleSidebar}
            sidebarOpened={this.state.sidebarOpened}
          />
          <Switch>{this.getRoutes(routes)}</Switch>
        </div>
      </div>
    </Fragment>
  );
}
```

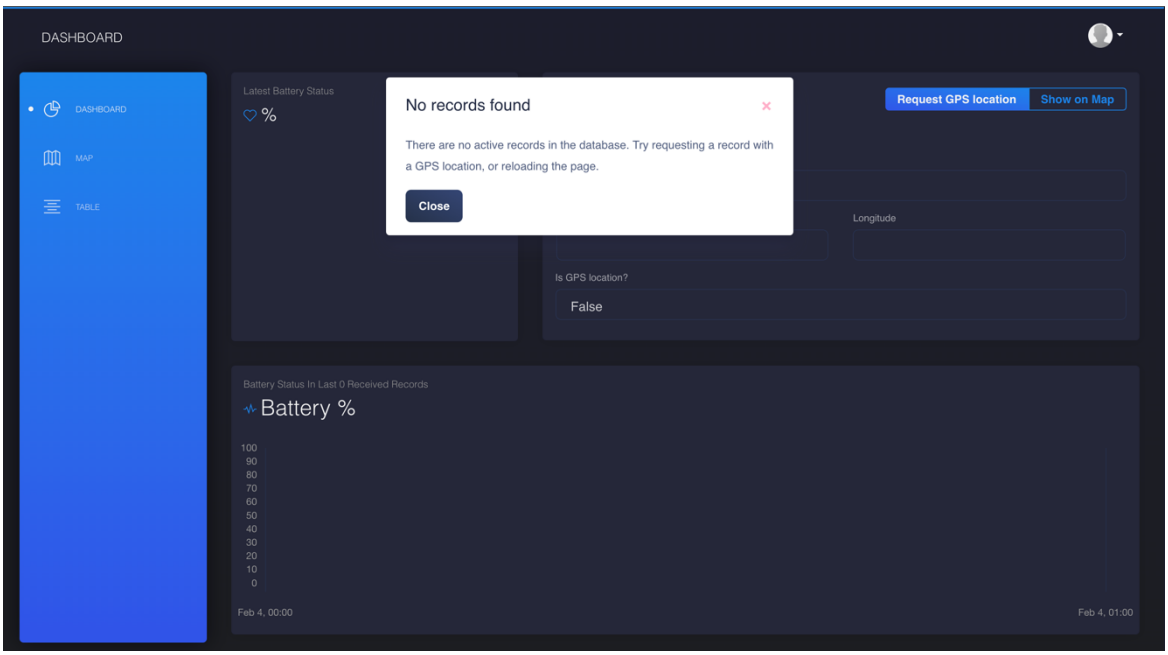
Isječak koda 11. Render metoda komponente Admin.js

Razvijena korisnička aplikacija sastoji se od tri zaslona (engl. *screen*). Prvi od njih, koji je ujedno i ulazna točka aplikacije, je kontrolna ploča na kojem je vidljiv graf stanja baterije kroz zadnjih 20 primljenih zapisa, kružni graf za prikaz zadnjeg poznatog stanja baterije, kao i detalji o zadnjoj poznatoj lokaciji. Osim toga, tu se nalaze i gumbi za zahtjev GPS lokacije i gumb za prikaz zadnje lokacije na karti. Na slici broj 2 se nalazi opisani zaslon.



Slika 2. Zaslon kontrolne ploče

Ukoliko u bazi podataka ne postoji niti jedan zapis za prikaz, korisniku se prikaže modalni prozor (engl. *modal window*) s obavijesti da nema zapisa. Primjer toga je vidljiv na slici 3.



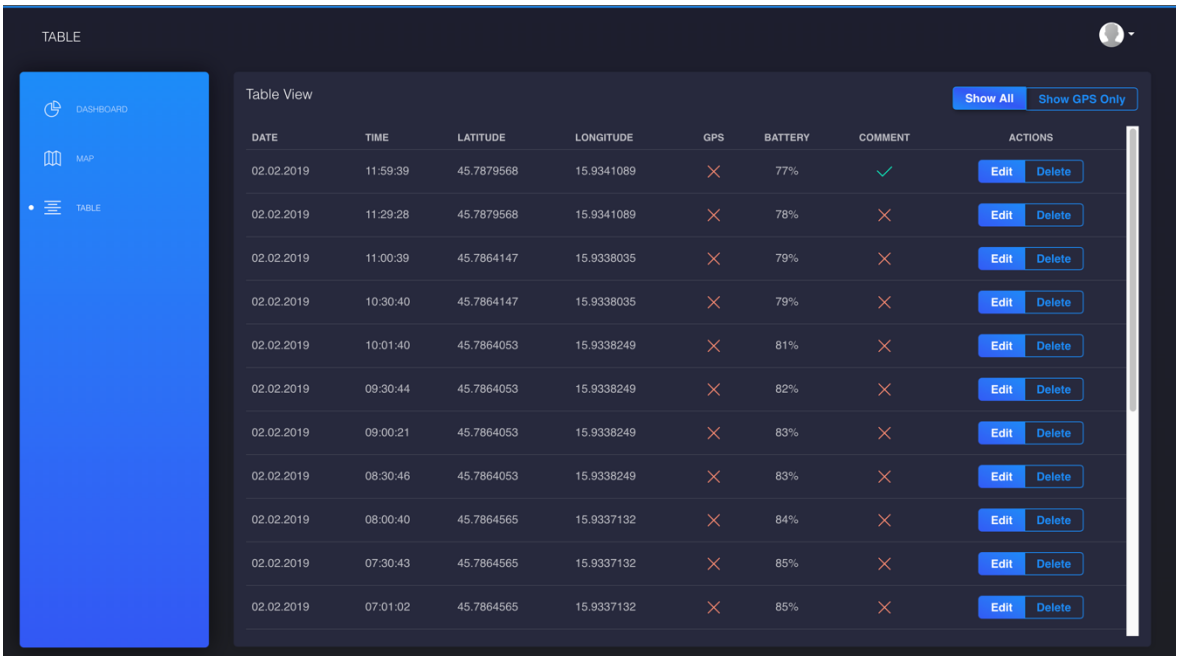
Slika 3. Zaslon kontrolne ploče s modalnim prozorom upozorenja

Isječak koda 12 prikazuje *render* metodu (metodu koja se poziva za prikaz elemenata u internetskom pregledniku) gore prikazanog zaslona. Vidljiva je jasna modularnost koda, odnosno razdvajanje svih elemenata koji odrađuju zasebnu ulogu u posebne komponente, koje se onda samo pozivaju unutar *render* metode glavne komponente. Tako se pri pozivu *BatteryPercentageGraph* komponente, istoj predaje prvi zapis iz liste kao rekvizit (engl. *prop*), a ona je zadužena da primljene podatke prikaže na ispravan način. Budući da smo odvojili ispis ove komponente u zaseban dokument, moguće je bilo gdje drugdje u aplikaciji pozvati isti tu komponentu i predati joj zapis za koji će ona iscrtati kružni graf stanja baterije. Upravo time se postiže modularnost i lakša ponovna upotreba istih komponenti.

```
render() {
  const { records, isFetching, gpsIsFetching } = this.props;
  if (isFetching && records.length === 0) {
    return <LoadingComponent />;
  } else {
    return (
      <div className="content">
        <Row>
          <Col lg="4">
            <BatteryPercentageGraph record={records[0]} />
          </Col>
          <Col lg="8">
            <LocationInformation
              record={records[0]}
              onRequestGPSLocation={this.handleRequestGPSLocation}
              onShowOnMap={this.handleShowOnMap}
              gpsRequestDisabled={gpsIsFetching}
            />
          </Col>
        </Row>
        <Row>
          <Col xs="12">
            <BatteryLineGraph records={records} />
          </Col>
        </Row>
        <NoRecordsModal
          isOpen={this.state.noRecordsModalActive}
          onToggle={this.handleCloseModal}
        />
      </div>
    );
  }
}
```

Isječak koda 12. Render metoda *dashboard* zaslona

Dobar primjer ponovne upotrebe komponente vidljiv je na zaslonu za prikaz zapisa u obliku tablice, vidljivom na slici 4. Isječak koda 13 prikazuje dio *render* metode te komponente. Unutar *tbody* elementa se za svaki zapis poziva komponenta *TableRow* kojoj se predaje pojedini zapis, a koja je zadužena za ispis podataka o pojedinom zapisu u tabličnom obliku. Osim toga, svaka *TableRow* komponenta sadrži i mogućnost uređivanja i brisanja pojedinog zapisa. Takve „pametne" komponente koje ne služe samo za ispis podataka, nego ujedno i odrađuju nekakvu logiku i imaju mogućnost poziva akcija, nazivaju se kontejneri (engl. *container*).

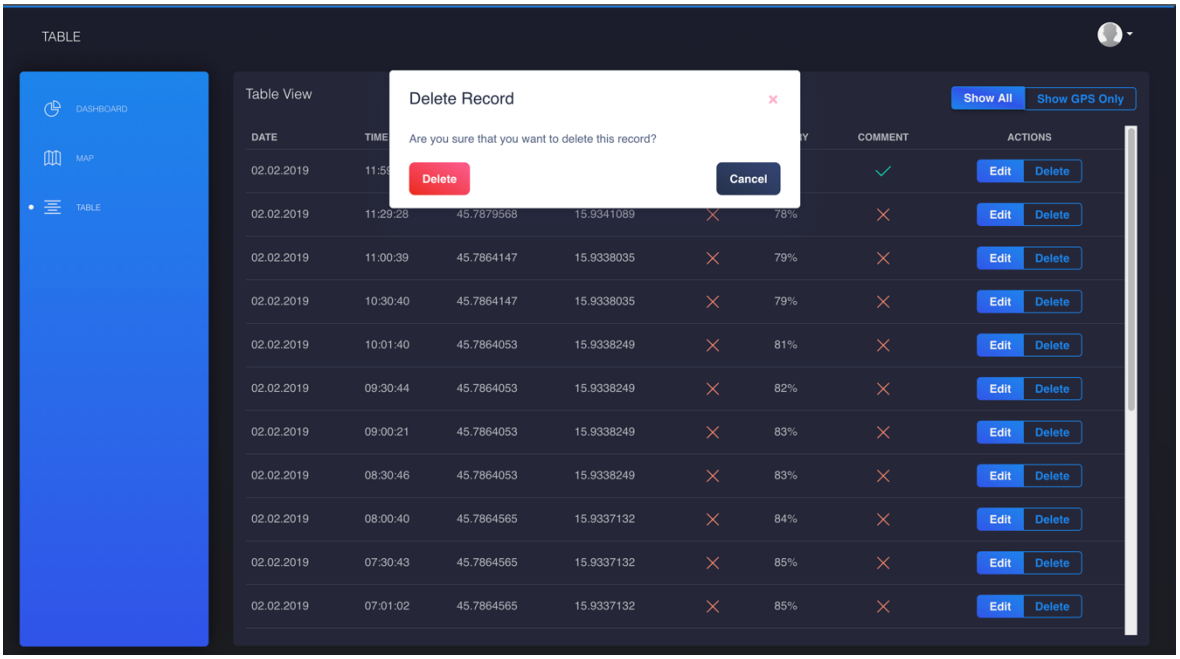


Slika 4. Zaslon za prikaz zapisa u tabličnom obliku

```
<tbody>
  {records.map(record => (
    <TableRow key={record._id} record={record} />
  ))}
</tbody>
```

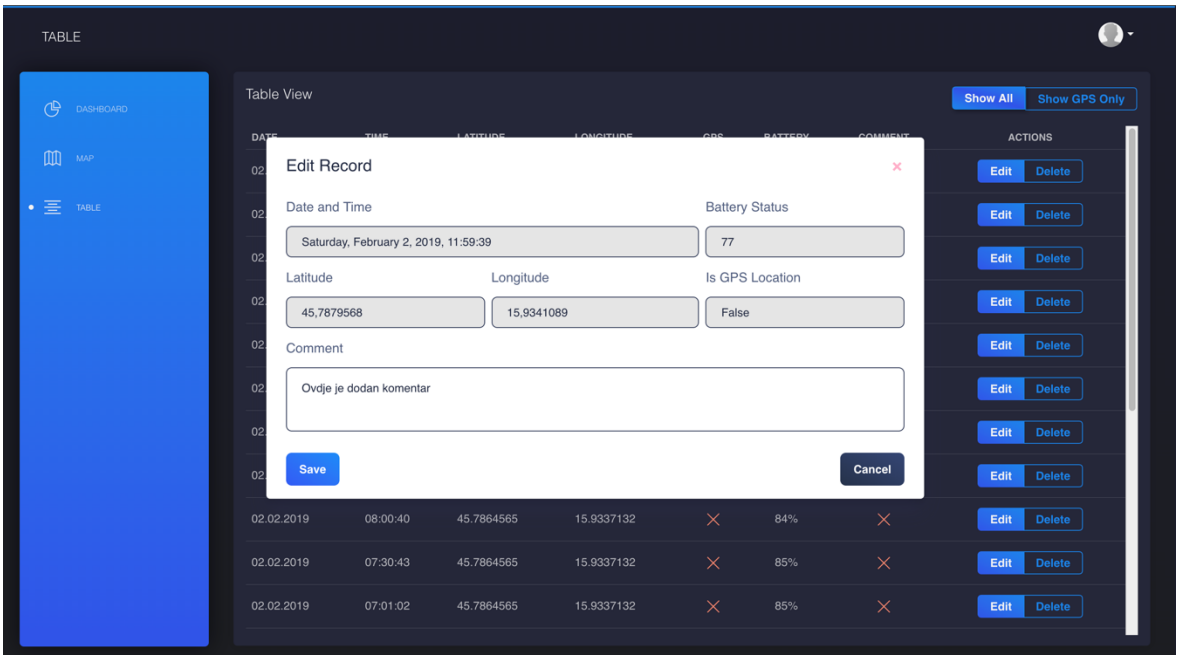
Isječak koda 13. *tbody* element komponente za ispis u tabličnom obliku

Na slici 5 se vidi upozorenje u obliku modala u kojem se od korisnika traži da potvrdi brisanje zapisa, kako bi se onemogućilo slučajno brisanje, jer zapis nije moguće vratiti nakon brisanja.



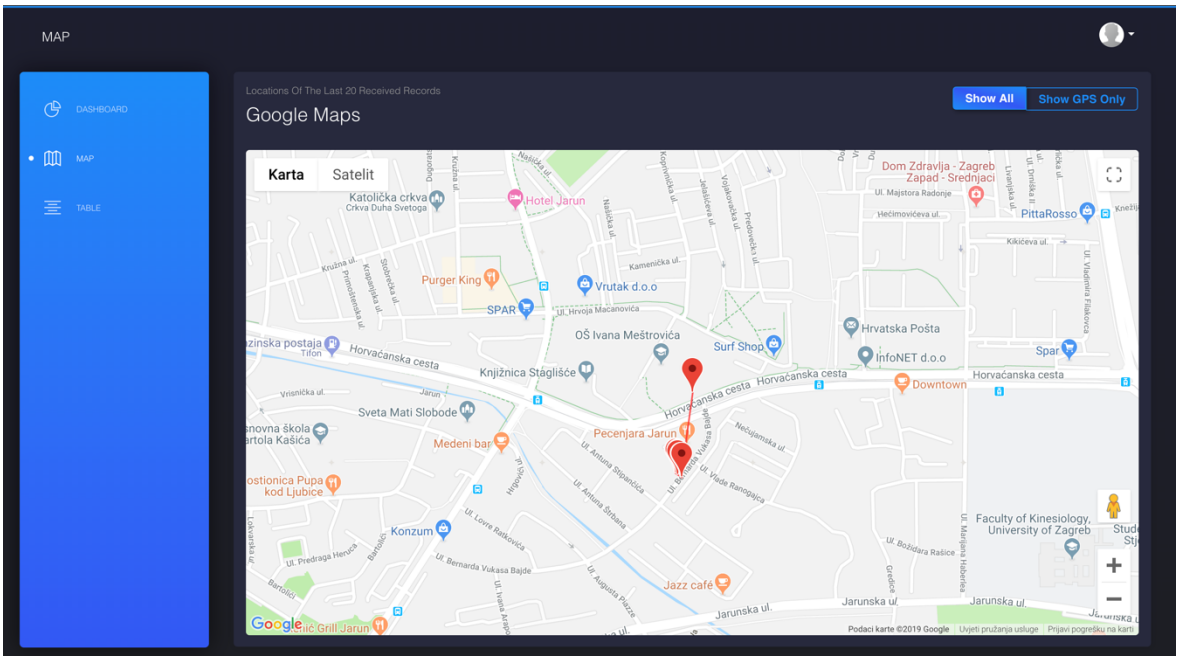
Slika 5. Modal za potvrdu brisanja zapisa

Slika broj 6 prikazuje modal za uređivanje pojedinog zapisa. Prikažu se svi podatci zapisa, no moguće je samo unijeti ili izmijeniti komentar zapisa kako bi se očuvao integritet podataka.



Slika 6. Modal za uređivanje zapisa

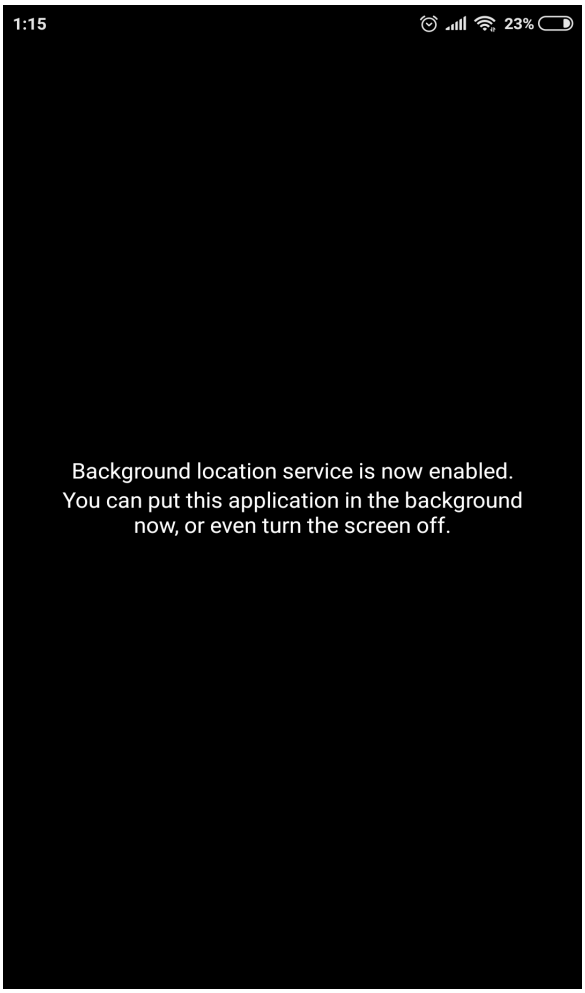
Posljednji zaslon u korisničkoj aplikaciji je zaslon s integriranom *Google Maps* kartom za prikaz lokacija zapisa na karti. Na zaslonu se prikazuje samo zadnjih 20 lokacija kako ne bi došlo do prenatrpanosti lokacija na karti. Omogućeno je filtriranje zapisa kako bi se prikazali samo oni s GPS lokacijom. Na slici 7 je vidljiv zaslon s kartom. Oznake označavaju zapise i njihove lokacije te je prijelazom miša preko pojedinog moguće vidjeti detalje, odnosno datum i vrijeme nastanka zapisa. Osim toga, na karti je iscrtana linija koja redom spaja oznake kako bi se dobila linija kretanja. Trenutno je jedini način da se usporedi starost zapisa ručnim uspoređivanjem vremena, no u daljnjem razvoju sustava u planu je vizualno naglasiti starost zapisa. Na primjer, gradacijom boje označavati zapise od starijih prema mlađima.



Slika 7. Zaslon za prikaz karte

2.3 Mobilna aplikacija

Mobilna aplikacija osmišljena je kao servis koji radi u pozadini čak i kada je zaslon uređaja ugašen. Razlog tomu je da bi se što manje trošila baterija, to jest, za vrijeme rada aplikacija ne prikazuje ništa bitno na zaslonu te je zaslon moguće ugasiti. Osim toga većina današnjih uređaja gasi zaslon nakon nekog vremena ako nema interakcije s korisnikom, no i dalje želimo slati status o stanju baterije i lokaciji na pozadinski servis. Kada je aplikacija upaljena i trenutno u fokusu, prikazuje se samo poruka o tome da je pozadinski servis za lokaciju upaljen te da korisnik može postaviti aplikaciju u pozadinu ili čak ugasiti zaslon, što se može vidjeti na slici 8.



Slika 8. Zaslon pri radu mobilne aplikacije

Mobilna aplikacija rađena je uz pomoć *Expoa*, seta alata, biblioteka i servisa koji omogućuju izradu izvornih (engl. *native*) mobilnih aplikacija za iOS i Android uređaje u JavaScript jeziku [12]. Za razvoj aplikacija s *Expoom* potrebna su dva alata: lokalni razvojni alat na računalu i klijentska aplikacija za mobilni uređaj pomoću koje se pokreće aplikacija koja se razvija. *Expo CLI* je alat za rad na računalu. Uz sučelje naredbenog retka (engl. *command-line interface*, *CLI*), dolazi i grafičko korisničko sučelje *Expo Developer Tools*, koje se pali u web pregledniku. *Expo Client* je klijentska aplikacija za mobilne uređaje, simulatore, i emulatore, koja omogućuje pregledavanje aplikacija dok ih razvijamo. Kada se pokrene *Expo CLI*, generira se QR kod, a skeniranjem istoga pomoću *Expo Client* aplikacije na tom uređaju se otvara aplikacija koju razvijamo. Sve spremljene promjene u projektu automatski se razvijaju i vidljive su nakon nekoliko trenutaka na uređaju. To omogućuje brz razvoj mobilnih aplikacija. Uz to što je razvoj aplikacija vrlo brz, sav kod koji se piše u *Expou* je u JavaScriptu i *React Nativu*, a *Expo* sav napisani kod pretvara u izvorne kodove za Android i iOS, te time nije potrebno pisati dvije različite aplikacije za različite platforme. Pisanje aplikacija u *React Nativu* je jako slično web aplikacijama pisanim u *Reactu*.

Ulazna datoteka razvijene mobilne aplikacije je *App.js* koja se nalazi u korijenskom direktoriju, a dio njenog sadržaja je prikazan na isječcima koda 14, 15 i 16.

```
import { LOCATION_TASK } from "./constants";
import {
  startLocationUpdatesAsync,
  registerForPushNotificationsAsync,
  sendGPSLocationAsync,
  sendLocationAsync
} from "./utils";
TaskManager.defineTask(LOCATION_TASK, ({ data, error }) => {
  if (error) {
    // Error occurred - check `error.message` for more details.
    console.log("ERROR: ", error);
    return;
  }
  if (data) {
    sendLocationAsync(data);
  }
});
```

Isječak koda 14. Poziv *defineTask* metode u App.js datoteci

Najprije u globalnom opsegu aplikacije deklarira zadatak uz pomoć naredbe *TaskManager.defineTask()*. *TaskManager* je jedan od *Expo* API-ja koji omogućuje upravljanje zadatcima, posebno onima koji se izvode dok je aplikacija u pozadini [13]. Pomoću naredbe *defineTask* se definira zadatak tako da mu se pridjeli ime i funkcija koja se obavi na njegov poziv. Zadatak je potrebno definirati u globalnom opsegu a ne u metodama životnog ciklusa *Reacta* zbog toga što u slučaju zadataka koji se obavljaju u pozadini, želimo pokrenuti aplikaciju, izvršiti zadatak, te zatim ponovno postaviti aplikaciju u pozadinu, bez da se išta prikaže na zaslonu uređaja. Nakon definicije zadatka počinje glavna komponenta. U njoj se u metodi životnog ciklusa *componentDidMount* koja se izvrši nakon što se aplikacija uspješno montira, pozovu funkcije *startLocationUpdatesAsync* i *registerForPushNotificationsAsync*, te osim toga i doda *handler* na *push* obavijesti, čime se na sve *push* obavijesti pozove metoda *_handleNotification*. Ukoliko obavijest koja se primi sadrži zastavicu *GPSLocation* pozove se metoda *sendGPSLocationAsync*. Nakon toga slijedi *render* metoda koja samo ispisuje već spomenutu poruku da je aplikaciju moguće staviti u pozadinu ili ugasiti zaslon.

```
export default class App extends React.Component {
  componentDidMount() {
    startLocationUpdatesAsync();
    registerForPushNotificationsAsync();
    this._notificationSubscription = Notifications.addListener(
      this._handleNotification
    );
  }
  _handleNotification = notification => {
    if (notification.data && notification.data.GPSLocation) {
      sendGPSLocationAsync();
    }
  };
  render() { ... }
}
```

Isječak koda 15. Komponenta App

```
render() {
  return (
    <View style={styles.container}>
      <View style={styles.box}>
        <Text style={styles.text}>
          Background location service is now enabled.
        </Text>
        <Text style={styles.text}>
          You can put this application in the background now, or even turn the
          screen off.
        </Text>
      </View>
    </View>
  );
}
```

Isječak koda 16. Render metoda komponente App

startLocationUpdatesAsync je metoda čiji je kod prikazan u isječku 17. Njena zadaća je pitati korisnika za dopuštenje kako bi aplikacija smjela upotrijebiti lokaciju uređaja, te je to dovoljno učiniti samo jednom. Ukoliko je dopuštenje već jednom dozvoljeno, Android sustav to pamti, te aplikacija ne traži odziv korisnika pri sljedećem paljenju već nastavlja dalje. Pozove se metoda *startLocationUpdatesAsync* iz *Location* API-ja, te se registrira zadatak za primanje lokacije s umjerenom točnošću, isti onaj definiran s *defineTask*, u određenom vremenskom intervalu.

```
import { Location, Permissions } from "expo";
import { LOCATION_TASK, TIME_INTERVAL } from "../constants";
export const startLocationUpdatesAsync = async () => {
  const { status } = await Permissions.askAsync(Permissions.LOCATION);
  if (status === "granted") {
    await Location.startLocationUpdatesAsync(LOCATION_TASK, {
      accuracy: Location.Accuracy.Balanced,
      timeInterval: TIME_INTERVAL
    });
  } else {
    throw new Error("Location permission not granted");
  }
};
```

Isječak koda 17. Metoda *startLocationUpdatesAsync*

TIME_INTERVAL je konstanta definirana u *constants.js* datoteci, a predstavlja broj milisekundi intervala nakon kojeg će se ciklički dohvaćati lokacija, te je postavljena na 1800000 što iznosi 30 minuta. Zadatak se vjerojatno neće izvesti točno svakih 30 minuta jer tu igraju ulogu i ostali zadatci operacijskog sustava, ali je dovoljno blizu. Kada se izvrši zadatak dohvata lokacije i pozove metoda *sendLocationAsync*, čiji je isječak vidljiv na isječku koda 18, dohvati se i stanje baterije uređaja, te se sve zajedno pošalje na razvijeni pozadinski servis. Vidljivo je da je zastavica *isGPSLocation* postavljena u logičku nulu, a to je zbog toga jer smo pri definiranju zadatka rekli da želimo umjerenu točnost što zapravo znači da će dobivena lokacija biti unutar radijusa 100 metara od stvarne lokacije.

```
import { Location, Permissions } from "expo";
import axios from "axios";
import { API_URL } from "../constants";
import { getBatteryStatusAsync } from "./getBatteryStatusAsync";
export const sendLocationAsync = async data => {
  const { locations } = data;
  const batteryPercentage = await getBatteryStatusAsync();
  axios
    .post(`${API_URL}/`, {
      ...locations[0].coords,
      isGPSLocation: false,
      batteryPercentage: batteryPercentage
    })
    .then(doc => console.log("Location sent"))
    .catch(err => console.error("Error while sending data: ", err));
};
```

Isječak koda 18. Metoda *sendLocationAsync*

Pri pozivu metode za registriranje za *push* obavijesti, generira se token koji jedinstveno identificira uređaj, što je prikazano na isječku koda 19. Trenutno se prema servisu šalje kodirano korisničko ime „Demo”, no u daljenjem radu na aplikaciji slati će se korisničko ime s kojim se korisnik prijavio u aplikaciju.

```
import { Permissions, Notifications } from "expo";
import axios from "axios";
import { API_URL } from "../constants";
export const registerForPushNotificationsAsync = async () => {
  const { status } = await Permissions.askAsync(Permissions.NOTIFICATIONS);
  if (status === "granted") {
    // Get the token that uniquely identifies this device
    let token = await Notifications.getExpoPushTokenAsync();
    return axios.post(`${API_URL}/push-token`, {
      token: token,
      username: "Demo"
    });
  } else {
    throw new Error("Notifications permission not granted");
  }
};
```

Isječak koda 19. *Metoda registerForPushNotificationsAsync*

Jednom kada se primi *push* obavijest, pozove se metoda *sendGPSLocationAsync* prikazana na isječku broj 20. U njoj se dohvaća trenutna lokacija s najvećom preciznošću i stanje baterije, te se sve zajedno šalje prema *backend* servisu uz postavljenu zastavicu *isGPSLocation*.

```
import { Location, Permissions } from "expo";
import axios from "axios";
import { API_URL } from "../constants";
import { getBatteryStatusAsync } from "./getBatteryStatusAsync";
export const sendGPSLocationAsync = async () => {
  const { status } = await Permissions.askAsync(Permissions.LOCATION);
  if (status === "granted") {
    const location = await Location.getCurrentPositionAsync({
      accuracy: Location.Accuracy.Highest
    });
    const batteryPercentage = await getBatteryStatusAsync();
    axios
      .post(`${API_URL}/`, {
        ...location.coords,
        isGPSLocation: true,
        batteryPercentage: batteryPercentage
      })
      .then(doc => console.log("GPS location sent"))
      .catch(err => console.error("Error while sending data: ", err));
  } else {
    throw new Error("Location permission not granted");}};
```

Isječak koda 20. *Metoda sendGPSLocationAsync*

3 Upute za pokretanje sustava

Kao preduvjet za pokretanje razvijenoga sustava potreban je *Node* i *npm* (engl. *Node.js package manager*). Upute za instalaciju *Nodea* moguće je pronaći na službenim stranicama a instalacije je potpuno besplatna [14]. Osim toga, za ispravan rad sustava je potrebna *MongoDB* baza podataka. *Mongo* je potrebno preuzeti sa službenih stranica [15], gdje je ujedno moguće pronaći upute za instalaciju za pojedini operacijski sustav [16]. Nakon instalacije *Nodea* i *MongoDBa*, potrebno je pokrenuti bazu podataka kako bi se razvijeni sustav mogao na nju spojiti. Zatim se najprije pokreće pozadinski servis jer je on osnova sustava. Pozicioniramo se u korijenski direktorij aplikacije, te pokrenemo naredbu

npm install

Tom naredbom se preuzimaju i instaliraju svi moduli potrebni za rad aplikacije, poput *mongoosea*. Nakon toga naredbom

npm run devstart

se pokreće server u *development* načinu rada koji prema zadanim postavkama sluša na portu 5000, ukoliko se postavljanjem varijable sustava *PORT* ne postavi drugačije. Nakon uspješnog pokretanja *backend* servisa, možemo pokrenuti korisničku web aplikaciju. U drugoj konzoli se na sličan način pozicioniramo u korijenski direktorij te aplikacije, te pokrenemo naredbu za instalaciju modula. Naredbom

npm run devstart

pokrenemo server koji poslužuje aplikaciju na portu 3000. Ako upalimo internetski pretraživač i postavimo se na adresu

http://localhost:3000/

možemo vidjeti da aplikacija radi, a sve spremljene izmjene u kodu aplikacije biti će automatski vidljive u pregledniku. Ukoliko želimo pokrenuti produkcijski način rada, potrebno je obje naredbe za pokretanje zamijeniti sa sljedećom

npm start.

Za pokretanje mobilne aplikacije u *development* načinu rada, potreban je *Expo CLI* i *Expo* aplikacija za mobilne uređaje. *Expo CLI* je moguće instalirati naredbom

npm install -g expo-cli

dok je mobilnu aplikaciju moguće preuzeti u službenim trgovinama, Play Store za Android uređaje i App Store za iOS uređaje. Postupak je ponovno vrlo sličan: pozicioniramo se u korijenski direktorij projekta, pokrenemo naredbu

npm install

kako bismo instalirali sve potrebne module, te zatim pokrenemo naredbu za pokretanje lokalnog razvojnog poslužitelja:

npm start.

Nakon toga na zaslonu se generira QR kao što je vidljivo na slici 9, a nakon što ga skeniramo sa *Expo* aplikacijom na mobilnom uređaju, na tom uređaju se počne prikazivati aplikacija koju razvijamo.

Slika 9. Ispis konzole pri radu s *Expo* alatom

Expo alat sadrži dosta stvari potrebnih za razvoj, i za veliki broj aplikacija moguće je sve željene ciljeve ostvariti čistim JavaScript kodom. No ukoliko su nam potrebne neke stvari koje nedostaju, poput dohvata informacija o stanju uređaja ili nekih drugih prilagođenih modula, tada je moguće napraviti *eject* aplikacije kako bi se dobili novi direktoriji *ios* i *android* koji sadrže izvorne (engl. *native)* projekte koji se mogu otvoriti i izgraditi s Xcodeom i Android Studiom. Takvi projekti i dalje ovise o *Expou*, ali sada se koristi Objective-C i Java biblioteka *ExpoKit* koja omogućava korištenje *Expo* platforme kao dio nativnog projetka [17].

Ako se radi u čistom JavaScriptu sa platformom *Expo*, bez da se projekt „izbaci" (engl. *eject*), tada je za stvaranje samostalnih binarnih *.apk* ili *.ipa* datoteka potrebno konfigurirati *app.json* dokument unutar korijenskog direktorija, te zatim pokrenuti naredbe

expo build:android ili *expo build:ios*.

Expo će za nas odraditi sve potrebno, te će se u konzoli ispisati URL s kojeg je moguće preuzeti gotove datoteke. No, ako se pak projekt *ejecta*, tada je potrebno ručno unutar Android Studija i Xcodea pokrenuti izgradnju aplikacija. Jednom kada imamo gotove binarne datoteke, moguće ih je prenijeti na mobilni uređaj i instalirati kao zasebne aplikacije, uz uvjet da je na uređaju dozvoljena instalacija aplikacija s nepoznatih izvora. Jednom kada smo dovoljno testirali i zadovoljni smo svojom aplikacijom, možemo ju objaviti na Appleov App Storeu ili Google Playu.

4 Mjerenja potrošnje baterije mobilnog uređaja

U svrhu mjerenja utjecaja rada mobilne aplikacije na potrošnju baterije korišten je Xiaomi Redmi Node 4 koji sadrži Li-Po bateriju kapaciteta 4100 mAh. Uređaj je napunjen do 100% te ostavljen 12 sati bez interakcije. Nakon isteka vremena zabilježen je status baterije, te je uređaj ponovno napunjen do 100%. Uređaj je zatim ponovno ostavljen 12 sati bez interakcije, ali ovaj put s uključenom razvijenom mobilnom aplikacijom. Postupak je zatim ponovljen ali za manji interval slanja podataka, ovaj put uz period od 15 minuta. Mjerenja su ponovljena tri puta kako bi se dobila srednja vrijednost potrošnje baterije. U nastavku je prikazana tablica s izmjerenim statusima baterije za svako mjerenje. Iz tablice je vidljivo da rad aplikacije u pozadini ne stvara puno dodatne potrošnje baterije. To dakako ovisi o intervalu u kojem se podatci šalju prema API-ju, ali također i od drugim servisima i aplikacijama koje se vrte u pozadini uređaja, poput provjere za nadogradnje aplikacija i aplikacija za *chat*. Za izvedbu točnijeg mjerenja i utvrđivanja utjecaja na potrošnju baterije potrebno je ponoviti navedena mjerenja više puta, uz korištenja različitih intervala slanja kako bi se dobila ovisnost potrošnje o intervalu. No, isto tako bilo bi dobro provesti mjerenja i na više različitih uređaja kako bi se ustanovilo postoji li određena ovisnost o vrsti i kapacitetu baterije. Nažalost u okviru ovoga rada to nije bilo moguće ostvariti zbog nedostatka opreme.

Tablica 1. Stanje baterije nakon 12 sati radi

	Bez aplikacije	Interval 30 minuta	Interval 15 minuta
1. mjerenje	85%	77%	71%
2. mjerenje	82%	76%	69%
3. mjerenje	83%	77%	70%

Za vrijeme pisanja ovog rada u najnovijoj verziji *Expoa*, verziji v32.0.0 korištenoj u ovom radu, prisutna je greška u kodu kod rada pozadinskih zadataka na Android uređajima, gdje se deklarirani zadatci ne bi obavljali periodički. Stoga je u svrhu mjerenja potrošnje baterije bilo potrebno napraviti modifikaciju programa na način da je pozadinski servis periodički slao *push* obavijest s zahtjevom za slanje lokacije. Ove *push* obavijesti mobilna je aplikacija obrađivala na sličan način kao i zahtjeve za GPS lokacijom, ali je koristila optimalnu preciznost kao parametar dohvata lokacije umjesto najbolje moguće preciznosti. Razvijeno zaobilaženje problema ne bi trebalo imati prevelikoga utjecaja na potrošnju baterije u odnosu na inicijalni sustav jer se u oba slučaja dohvaća lokacija i ista šalje prema API-ju, a razlika je u tome što je dodana obrada *push* obavijesti, te bi upravo zbog toga prvobitni oblik aplikacije trebao trošiti čak i manji postotak baterije od modificiranoga.

5 Usporedba s drugim proizvodima na tržištu

Na tržištu je danas moguće pronaći mnogo uređaja za praćenje kućnih ljubimaca, no njihove cijene se kreću od 50 do 200 američkih dolara te uz to mnogi iziskuju mjesečnu pretplatu [18]. Neki od popularnijih uređaja nude mogućnosti praćenja lokacije ljubimca u živo te nude mogućnost praćenja aktivnosti. Jedan takav uređaj je Whistle 3 prikazan na slici 10, no namijenjen je samo za američko tržište. Whistle 3 nudi trajanje baterije do čak 10 dana, praćenje aktivnosti i lokacije kućnoga ljubimca na pripadajućoj mobilnoj aplikaciji te je ujedno i vodo-nepropustan. Whistle 3 se može kupi na Amazonu za nešto manje od $100, što otprilike iznosi 650 kuna, no zbog korištenja mobilne mreže zahtijeva mjesečnu pretplatu u iznosu od $10 mjesečno.

Slika 10. Whistle 3 [19]

Još jedan od popularnih uređaja je Tractive prikazan na slici 11. Uređaj se može kupiti na Amazonu za $52 no također zahtjeva mjesečnu pretplatu. Tractive je moguće koristiti i u Hrvatskoj, no za to je potrebno kupiti *premium* pretplatu na godišnjoj ili dvogodišnjoj razini čija je cijena dodatnih €4.99, odnosno €4.16. Tractive također dolazi uz pripadajuću mobilnu i web aplikaciju, a baterija uređaja izdrži do 3 dana.

Slika 11. Tractive [20]

Oba navedena primjera uređaja za lociranje koriste SIM karticu i mobilne podatke za slanje podataka, te je zbog toga potrebna mjesečna pretplata. Mobilna aplikacija razvijena u ovom radu također zahtijeva mobilnu mrežu ili povezanost na *WiFi* kako bi slala podatke prema pozadinskom servisu. No za razliku od navedenih primjera, razvijena aplikacija ne nudi mogućnost praćenja aktivnosti ljubimaca. Osim toga, potrošnja baterije i to koliko često će se uređaj na kojem je aplikacija morati puniti, ovisi ponajprije o stanju i kapacitetu baterije tog mobilnoga uređaja, te će to vrijeme biti drugačije za svaki uređaj koji se koristi za praćenje. No dodatnim radom na sustavu moguće je ostvariti funkcionalnosti poput „virtualnih ograda”, gdje bi se vlasniku kućnoga ljubimca poslala obavijest čim bi ljubimac izašao iz unaprijed definiranoga područja.

6 Zaključak

U sklopu ovoga rada izgrađen je sustava za kontrolu lokacije kućnih ljubimaca modifikacijom i programskim proširenjem Android operacijskog sustava za pametne telefone. Ostvareni sustav je prenosiv i moguće ga je instalirati na većini današnjih uređaja. Korištenje izgrađene aplikacije ne stvara preveliki trošak na potrošnju baterije uređaja, te je omogućen rad u pozadini. Uz mobilnu aplikaciju koja šalje lokaciju uređaja i status baterije izgrađen je i prateći pozadinski servis za primanje tih podataka. Osim toga, izgrađena je i korisnička aplikacija u kojoj je moguće pregledati sve dobivene podatke.

Zbog *buga* u verziji alata korištenoj za izgradnju mobilne aplikacije, njen pravilni rad nije u potpunosti ostvaren, no očekivano je da će taj kvar biti otklonjen u sljedećoj verziji čime će biti omogućen prvobitni oblik aplikacije. No unatoč tome dobiveni rezultati na potrošnju baterije su prihvatljivi.

Ostvareni sustav može se mjeriti s ostalim uređajima za kontrolu lokacije kućnih ljubimaca koje je moguće naći na tržištu jer ostvaruje približno istu funkcionalnost za nižu cijenu i omogućuje iskorištavanje starih pametnih telefona koji se više ne koriste u uobičajene svrhe.

Izgrađeni sustav nije u potpunosti spreman za masovno korištenje te je na njemu potreban dodatni rad. To ponajprije uključuje omogućavanje registracije i prijave više korisnika gdje bi svaki korisnik imao pristup vlastitim podacima. Osim toga, jedno od mogućih proširenja je i mogućnost kontrole više uređaja za nadzor lokacije po pojedinom korisniku, gdje bi se podatci s različitih izvora mogli odvojiti i posebno prikazati. Sustav ostvaren u ovom radu ostvaruje dovoljnu minimalnu funkcionalnost za nastavak rada na njemu.

7 Literatura

[1] *Number of mobile phone users worldwide from 2015 to 2020 (in billions)*, https://www.statista.com/statistics/274774/forecast-of-mobile-phone-users-worldwide/, 3.2.2019.

[2] *Number of smartphones sold to end users worldwide from 2007 to 2017 (in million units)*, https://www.statista.com/statistics/263437/global-smartphone-sales-to-end-users-since-2007/, 3.2.2019.

[3] *About Node.js*, https://nodejs.org/en/about/, 5.1.2019.

[4] *Express - Fast, unopinionated, minimalist web framework for Node.js*, https://expressjs.com/, 5.1.2019.

[5] *Routing*, https://expressjs.com/en/guide/routing.html, 5.1.2019.

[6] *Express application generator*, https://expressjs.com/en/starter/generator.html, 5.1.2019.

[7] *What is MongoDB?*, https://www.mongodb.com/what-is-mongodb, 5.1.2019.

[8] *mongoose - elegant mongodb object modeling for node.js*, https://mongoosejs.com/, 6.1.2019.

[9] *React - A JavaScript library for building user interfaces*, https://reactjs.org/, 10.1.2019.

[10] *Getting Started – Create React App*, https://facebook.github.io/create-react-app/docs/getting-started, 10.1.2019.

[11] *Black Dashboard React - FREE BOOTSTRAP 4 (REACTSTRAP) ADMIN TEMPLATE*, https://www.creative-tim.com/product/black-dashboard-react, 11.1.2019.

[12] *Quick Start - Expo Documentation*, https://docs.expo.io/versions/v32.0.0/, 15.1.2019.
[13] *TaskManager - Expo Documentation*, https://docs.expo.io/versions/v32.0.0/sdk/task-manager/, 15.1.2019.

[14] *Downloads*, https://nodejs.org/en/download/, 1.2.2019.

[15] *MongoDB Download Center*, https://www.mongodb.com/download-center/community, 1.2.2019.

[16] *Install MongoDB Community Edition,* https://docs.mongodb.com/v3.2/administration/install-community/, 1.2.2019.

[17] *Ejecting to ExpoKit*, https://docs.expo.io/versions/v32.0.0/expokit/eject/, 1.2.2019.

[18] Anna Burleson, *Best Pet Tracking Devices of 2019*, https://www.toptenreviews.com/electronics/family/best-pet-tracking-devices/, 2.2.2019.

[19] *Whistle 3 - GPS Pet Tracker & Activity Monitor,* https://shop.whistle.com/products/whistle-3-gps-pet-tracker-activity-monitor, 2.2.2019.

[20] *GPS Tracker for Dogs*, https://tractive.com/en/pd/gps-tracker, 2.2.2019.

8 Naslov, sažetak i ključne riječi

Sustav za automatsko praćenje i lociranje kućnih ljubimaca

Tema ovoga rada je ostvarivanje sustava za automatsko praćenje i lociranje kućnih ljubimaca. Osmišljene su i izgrađene tri aplikacije u tu svrhu: mobilna aplikacija za slanje podataka o lokaciji, pozadinski servis za primanje i spremanje podataka, te korisnička aplikacija za prikaz i uređivanje podataka. U radu su detaljno opisane sve tri aplikacije uz komentiranje bitnih dijelova koda, nakon čega su navedene upute za pokretanje i daljnji rad na sustavu. Osim toga, analizirana su provedena mjerenja na potrošnju baterije pri radu mobilne aplikacije na stvarnom uređaju. Za kraj je provedena usporedba ostvarenoga sustava s nekim od sličnih uređaja dostupnih na tržištu.

praćenje lokacije, GPS, Android, kućni ljubimci

9 Title, summary and keywords

Automatic Tracking and Positioning System for Pets

The subject of this thesis is to create a system for automatic tracking and positioning for pets. Three applications have been designed for this purpose: a mobile application for sending location data, a background service for receiving and saving data, and a user application for displaying and editing data. In this paper, all three applications are described in detail with comments regarding the essential parts of the code, followed by instructions for starting and continuing the work on the system. In addition, battery measurements were performed and analysed on the actual mobile device. At the end, designed system is compared with some of the similar devices available on the market.

location tracking, GPS, Android, pets