Razvoj aplikacija za infotainment sustave u vozilima

Gajari, Filip

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

Sveučilišni diplomski studij Računarstva, smjer Programsko inženjerstvo

RAZVOJ APLIKACIJA ZA INFOTAINMENT SUSTAVE U VOZILIMA

Diplomski rad

Filip Gajari

Mentor: Doc.dr.sc. Josip Balen (FERIT),

Sumentor: Dean Rumen (GlobalLogic)

Osijek, 2021.

SADRŽAJ

1. UVOD

U današnje vrijeme Android operacijski sustav se može naći u različitim uređajima, kao što su mobiteli, televizori, tableti itd. Međutim, Google nije stao samo na tome. Zajedno sa Intelom, te uz pomoć Volva i Audia, odlučio je stupiti u auto industriju i razviti Android Automotive koji je predstavljen u ožujku 2017. godine. Android Automotive predstavlja varijaciju Android operacijskog sustava koja se samostalno, bez ikakvih dodatnih uređaja, pokreće na informacijsko-zabavnom sustavu vozila. On ne predstavlja posebni razvoj Androida, već se nalazi u istom spremištu (engl. *depository*) kao i sam Android operacijski sustav koji se pokreće na mobitelima te koristi istu bazu programskih kodova.

Sam sustav nudi jako puno mogućnosti programerima u razvijanju samostalnih aplikacija, zbog čega dolazi i s velikom bazom programskih kodova što olakšava sam razvoj novih aplikacija. Osim rukovanja informacijsko-zabavnim zadacima kao što su slanje poruka, reprodukcija glazbe i slično, također rukuje važnim funkcijama automobila, kao što je upravljanje klimom, kamerama, ventilacijom itd. Također, važno je spomenuti da je otvoren te da podržava aplikacije koje su izrađene za Android kao i za Android Auto. Prvi automobil predstavljen sa Android Automotiveom je Polestar 2 čija proizvodnja je počela u ožujku 2020. godine, ali očekuje se da će se ovaj operacijski sustav naći i u Renaultu te u automobilima General Motorsa koji u planu ima predstaviti svoj prvi automobil 2021.

U poglavlju 2 ovog rada opisana su aktualna znanstvena i praktična dostignuća u području rada. U poglavlju 3 bavimo se arhitekturom, sigurnošću i dopuštenjima Android Automotive operacijskog sustava te je napravljena usporedba s Android operacijskim sustavom i Android Autom. U poglavlju 4 napravljena je usporedba s ostalim sličnim sustavima kao što su iDrive informacijsko-zabavni sustav, Mercedes-Benz User Experience informacijsko-zabavni sustav i Audi Multi Media Interface informacijsko-zabavni sustav, dok je u poglavlju 5 opisana aplikacija izrađena kao dio praktičnog dijela rada.

2. AKTUALNA ZNANSTVENA I PRAKTIČNA DOSTIGNUĆA U PODRUČJU RADA

Za Android Automotive operacijski sustav, kao i za svaki novi sustav koji će u budućnosti predstavljati mjerilo kvalitete informacijsko-zabavnog sustava u automobilima, zahtjeva vrijeme da ga veliki automobilski proizvođači prihvate i implementiraju u same automobile. U ovom trenutku, samo Polestar 2 koristiti Android Automotive operacijski sustav, ali je pitanje vremena kada će i ostali proizvođači početi ugrađivati isti. Iako je ovo veliki korak za automobilske proizvođače, jer bi time odustali od vlastitog sustava, kupci diktiraju smjer u kojem se tržište razvija. Danas, većina ljudi koristi opciju spajanja vlastitog mobitela na informacijsko-zabavni sustav automobila čime šalju poruku samim proizvođačima da žele koristiti operacijske sustave koje koriste i na mobilnim uređajima.

2.1. GENIVI ALLIANCE ORGANIZACIJA

GENIVI Alliance predstavlja organizaciju koja je osnovana 2009. godine s ciljem integracije Linuxa i softverskog pristupa s otvorenim kodom u automobile. Prvi zadatak organizacije, koji je i uspješno proveden, bio je integracija same platforme u informacijsko-zabavne sustave vozila, dok im je sljedeći zadatak pomoći proizvođačima u sljedećim aktivnostima[1]:

1. Integraciji više operacijskih sustava u središnji zaslon te u povezani digitalni kokpit
2. Poboljšati povezanost vozila s oblakom
3. Promovirati zajednički model podataka o vozilu i standardni katalog usluga

Kao što je spomenuto, jedna od najvažnijih stvari je povezanost sa proizvođačima te pružanje OEM-ima (engl. *Original Equipment Manufacturer*) i njihovim dobavljačima moderne i jednostavnije metode za proizvodnju automobilskih softvera. Jedan od jako važnih GENIVI projekata koji je u tijeku je Android Automotive SIG. Ovaj projekt je orijentiran na integraciju softvera i podržava proizvođače automobila koji su se okrenuli Androidu. Sam cilj ovog projekta je poboljšati Android Automotive operacijski sustav s gledišta sigurnosti, pristupa informacijama, podrške za više zaslona, upravljanja zvukom, ažuriranja sustava kako bi bio u mogućnosti podržati nove verzije Androida operacijskog sustava, itd [2]. Proizvođači automobilske opreme sve više prihvaćaju Android Automotive operacijski sustav čime šalju poruku Googleu da žele siguran

sustav koji će bez greške surađivati s kompletnim automobilom, a između ostalog i s opremom koju oni proizvede.

2.2. OPEN AUTOMOTIVE ALLIANCE (OAA)

Google je, 2014. godine, odlučio steći dominaciju ispred Applea u korištenju mobilnih uređaja na informacijsko-zabavnim sustavima u automobilima. Kao što je 2007. to učinio na tržištu mobilnih uređaja stvaranjem udruženja pod nazivom Open Handset Alliance, Google se odlučio na sličan potez stvaranjem udruženja proizvođača automobila pod nazivom Open Automotive Alliance [3]. Samo udruženje predstavlja grupu proizvođača koji su u to vrijeme bili zainteresirani za implementiranje mogućnosti korištenja Android mobilnih uređaja na vlastitim informacijsko-zabavnim sustavima. Svaki proizvođač automobila želi u svojim automobila imati tehnologije koje su u trendu te koje osiguravaju veću sigurnost vozačima. Danas, svi novi automobili imaju tu mogućnost.

U to vrijeme neki informacijsko-zabavni sustavi koristili su operacijski sustav baziran na Linuxu, neki na BlackBerryju, a neki proizvođači su surađivali sa Microsoftom [4]. Takva raznolikost sustava otežavala je posao kompanijama koje su razvijale aplikacije za automobile jer su morali prilagoditi aplikacije svakom brandu automobila posebno. Mogućnost korištenja mobilnog uređaja, definitivno, je olakšala njihov posao. Budući da je Android operacijski sustav baziran na Linuxu, osnutak GENIVI Alliancea, 2009., išao je u korist Googlea te mu je bio veliki znak u kojem će se smjeru kretati uspješnost osnivanja OOA, te na kraju i sama uspješnost probitka na tržište. Neki od proizvođača automobila koji su dio OAA udruženja su: Audi, General Motors, Honda, Hyundai, Alfa Romeo, Fiat, Ford, Opel, Škoda, Volvo, Volkswagen, Mercedes itd [5].

2.3. SLIČNA APLIKACIJSKA RJEŠENJA

Kao što je spomenuto u podpoglavlju 1.1. u svrhu praktičnog dijela diplomskog rada napravljana je aplikacija koja prikazuje osnovne informacije automobila. Pošto je Android Automotive novi operacijski sustav na tržištu, slične aplikacije, napravljene posebno za njega, gotovo je nemoguće pronaći. Jedna od takvih je Project black. Project black je prva njemačka automobilska platforma koja omogućava implementaciju i testiranje povezanih usluga pomoću Android Automotive operacijskog sustava i to na brz i učinkovit način.

Sama platforma ima integriranu mogućnost korištenja navigacije, čitanja važnih informacija automobila (senzora automobila), analizu podataka, reprodukciju glazbe itd [6]. Iako na tržištu nema sličnih aplikacija za Android Automotive operacijski sustav, mogu se naći mobilne aplikacije koje se putem Android Auta mogu koristiti na zaslonu automobila, ali imaju jako malo mogućnosti. Neke od takvih aplikacija su:

1. GPS Speedometer and Odometer [7]
2. DigiHUD Speedometer [8]
3. Drag Racer Car Performance [9]
4. SpeedView [10]
5. Ulysse Speedometer [11]

Kao što se može vidjeti iz naziva, sve te aplikacije su namijenjene za prikazivanje brzine. Ono što je posebno kod Android Automotive operacijskog sustava je to što omogućuje pristup karakterističnim senzorima automobila, što Android operacijski sustav ne omogućuje. Sve te aplikacije mogu se koristit preko Android Auotmotive operacijskog sustava na zaslonu automobila, ali nemaju mogućnost pristupa senzorima za gorivo, ručnu konačnicu, vanjsku temperaturu itd.

3. ANDROID AUTOMOTIVE OPERACIJSKI SUSTAV

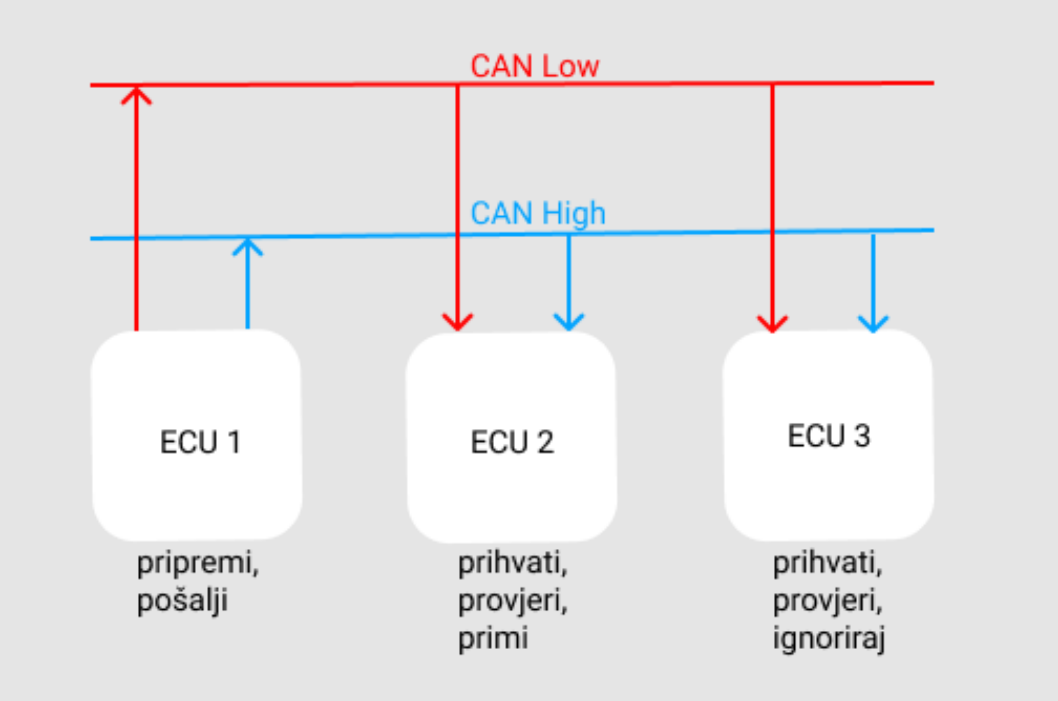
Automobili predstavljaju sustav koji se sastoji od mnoštva podsustava. Ti podsustavi su međusobno povezani putem različitih topologija sabirnice. Osim međusobne povezanosti, podsustavi su također, putem različitih topologija, povezani s informacijsko-zabavnim sustavom u vozilu. Određeni tip sabirnice i protokola se obično razlikuje od proizvođača do proizvođača, ali se čak razlikuje i između različitih modela vozila istog proizvođača. Primjeri sabirnica mogu biti Controller Area Network (CAN), sabirnica lokalne interkonektne mreže (LIN), MOST, Ethernet i TCP/IP mreža poput BroadR-Reacha [12].

Kada govorimo o Android Automotive operacijskom sustavu, govorimo o sloju apstrakcije hardware-a Android Automotive (HAL) koji predstavlja sučelje za razvoj Android Automotive implementacija. Sistemski integratori imaju mogućnost implementacije HAL modula u vozilu povezivanjem HAL sučelja specifičnih za funkciju platforme (npr. HVAC) s mrežnim sučeljima specifičnim za tehnologiju (npr. CAN sabirnica).

Uobičajene implementacije obično uključuju namjensku mikrokontrolersku jedinicu (MCU) koja pokreće vlasnički operativni sustav u stvarnom vremenu (RTOS) u svrhu pristupa CAN sabirnici, te se serijskom vezom može povezati s Android Automotive operacijskom sustavom, odnosno procesorom na kojem se nalazi sam sustav [12].

3.1. CAN SABIRNICA

CAN sabirnica (engl. *Controller Area Network*) predstavlja „živčani sustav“ vozila koja omogućuje komunikaciju među elektroničkim upravljačkim jedinicama vozila kao što su audio sustav i sustav zračnih jastuka [13]. Elektroničke upravljačke jedinice mogu poslati podatke (kao što su senzorski podaci), preko CAN sabirnice odnosno njezinih žica (CAN Low i CAN High), drugoj jedinici koja može odlučiti želi li primiti ili odbiti podatke. Na slici 3.1. može se vidjeti komunikacija CAN sabirnicom.



Slika 3.1. Komunikacija CAN sabirnicom

3.2. ARHITEKTURA ANDROID AUTOMOTIVE SUSTAVA

Arhitektura Android Automotive operacijskog sustava sastoji se od tri sloja modula. Prvi sloj je aplikacijski sloj gdje se aplikacije mogu podijeliti u tri skupine: aplikacije koje su sastavni dio operacijskog sustava, OEM aplikacije i aplikacija treće strane (engl. *third-party app*) [14]. Drugi sloj predstavlja sloj AOSP-a (engl. *Android Open Source Project*) ili SDK sloj (engl. Software development kit). Svaka aplikacija komunicira sa specifičnom *CarManager* komponentom (API) dok svaki *CarManager* modul komunicira sa *CarServicom*. Postoje različite instance *CarService* poput *CarSensorService* koji omogućuje aplikacijama treće strane pristup podacima senzora. *CarService* komponenta može se smatrati sigurnosnim *middlewareom* gdje je definirana kontrola pristupa (engl. *permission*) [14]. Treći sloj predstavlja sloj apstrakcije hardvera vozila (engl. *Vehicle Hardware Abstraction Layer*) koji mapira svojstva vozila na CAN signale definirane u DBC datoteci. VHAL ima mogućnost interakcije sa kontrolnom jedinicom automobila preko CAN sabirnice, etherneta ili nečeg drugoga. Na slici 3.2. možemo vidjeti arhitekturu HAL-a i Andorid Auotmotive sustava [15].

Slika 3.2. Arhitektura HAL-a i Android Automotive operacijskog sustava

3.3. SIGURNOST

Zaštita podataka kod Android Automotive operacijskog sustava vrši se na sljedeće načine [16]:

1. Aplikacije – sustav provjera ima li aplikacija dopuštenje za komunikaciju s podsustavima automobila
2. Dobro definirani API-ji – generički API-ji ne prihvaćaju proizvoljne binarne velike objekte podataka (engl. *blob*) (API-ji moraju biti dobro definirani)
3. Auto servis (engl. *CarService*) - ažuriranja su dopuštena samo putem OTA (engl. *Over-the-air programming*) ili USB-a, uz šifriranje cijelog diska i bootanje
4. *Vehicle* HAL - potvrđuje da su dopuštene određene poruke.

Kako bi se pristupilo većini svojstava vozila, potrebno je dopuštenje. Aplikacija korisnika traži za dopuštenje koje on treba odobriti. To je vidljivo prilikom pristupa određenim senzorima kao što su ABS, kontrola trakcije, kilometraža, odnosno onome što može na bilo koji način kompromitirati sigurnost vozača.

3.4. DOPUŠTENJA

Android dopuštenja mogu se podijeliti u četiri skupine [17]:

1. *NORMAL* – dopuštenja koja imaju minimalan rizik na korisnikovu privatnost te ih sustav automatski odobrava bez potrebe za korisnikovim odobrenjem.
2. *DANGEROUS* – definiraju se u slučaju pristupa korisnikovim privatnim informacijama te zahtijevaju korisnikovo odobrenje.
3. *SIGNATURE* – dopuštenja koja se odobravaju prilikom instalacije aplikacije, ali samo ako aplikacija ima isti certifikat kao i aplikacija koja definira te dozvole.
4. *SIGNATURE|PRIVILEGED* – dopuštenja koja su odobrena unaprijed instaliranim ili kriptografski potpisanim aplikacijama

Aplikacije treće strane koriste opasne ili normalne dopuštenja, a OEM aplikacije *signature* dopuštenja. Tablice 3.1., 3.2, 3.3. i 3.4 prikazuju sva dopuštenja koja su dio Android Automotive operacijskog sustava [14].

Tablica 3.1. Normalna dopuštenja Android Automotive operacijskog sustava

Razina zaštite	Ime dopuštenja
Normalna	READ_CAR_DISPLAY_UNITS, CONTROL_CAR_DISPLAY_UNITS, CAR_ENERGY_PORTS, CAR_INFO, CAR_EXTERIOR_ENVIRONMENT, CAR_POWERTRAIN

Tablica 3.2. Opasna dopuštenja Android Automotive operacijskog sustava

Razina zaštite	Ime dopuštenja
Normalna	CAR_SPEED, CAR_ENERGY

Tablica 3.3. Potpis (engl. *signature*) dopuštenja Android Automotive operacijskog sustava

Razina zaštite	Ime dopuštenja
Potpis	BIND_VMS_CLIENT, BIND_PROJECTION_SERVICE, BIND_INSTRUMENT_CLUSTER_RENDERER_SERVICE, BIND_CAR_INPUT_SERVICE

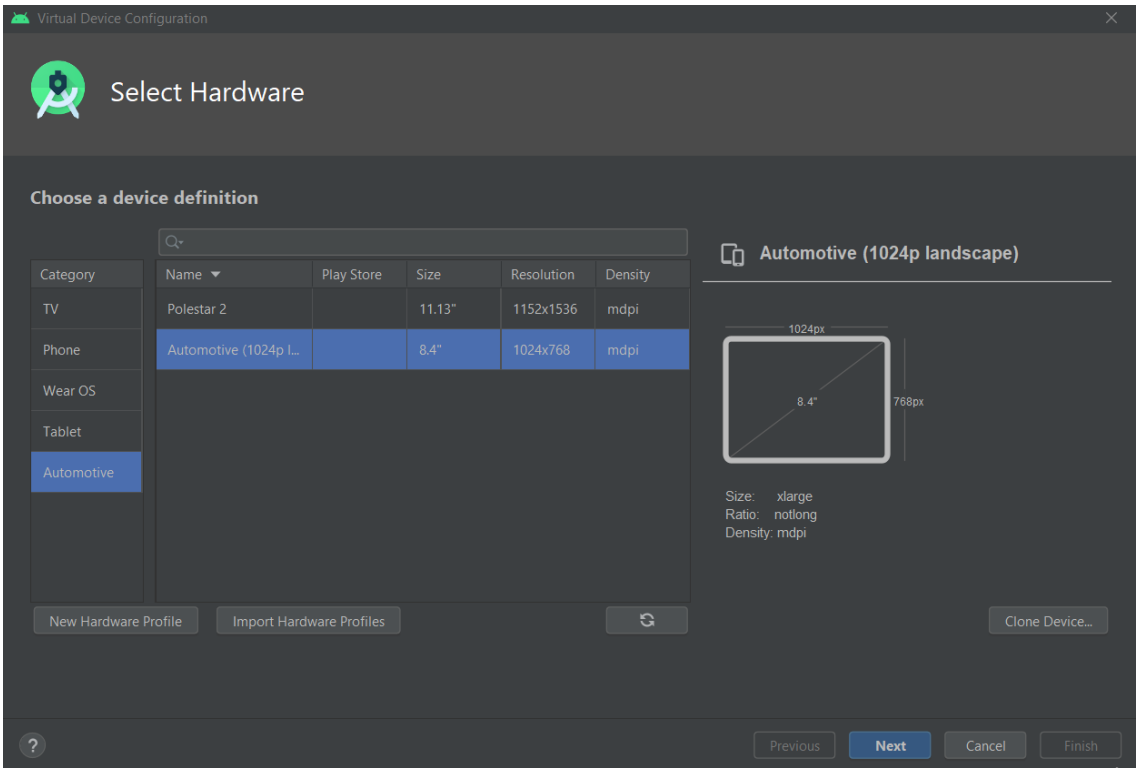
Tablica 3.4. Potpis | privilegirana dopuštenja Android Automotive operacijskog sustava

Razina zaštite	Ime dopuštenja
Potpis | privilegiran	CAR_MOCK_VEHICLE_HAL, READ_CAR_STEERING, CAR_IDENTIFICATION, CAR_MILEAGE, CAR_TIRES, CAR_ENGINE_DETAILED, CAR_DYNAMICS_STATE, CAR_VENDOR_EXTENSION, CAR_PROJECTION, ACCESS_CAR_PROJECTION_STATUS, CONTROL_CAR_SEATS, CONTROL_CAR_MIRRORS, CONTROL_CAR_WINDOWS, CONTROL_CAR_DOORS, CONTROL_CAR_CLIMATE, CAR_EXTERIOR_LIGHTS, CONTROL_CAR_EXTERIOR_LIGHTS, READ_CAR_INTERIOR_LIGHTS, CONTROL_CAR_INTERIOR_LIGHTS, CAR_POWER, CAR_NAVIGATION_MANAGER, CAR_DIAGNOSTICS, CLEAR_CAR_DIAGNOSTICS

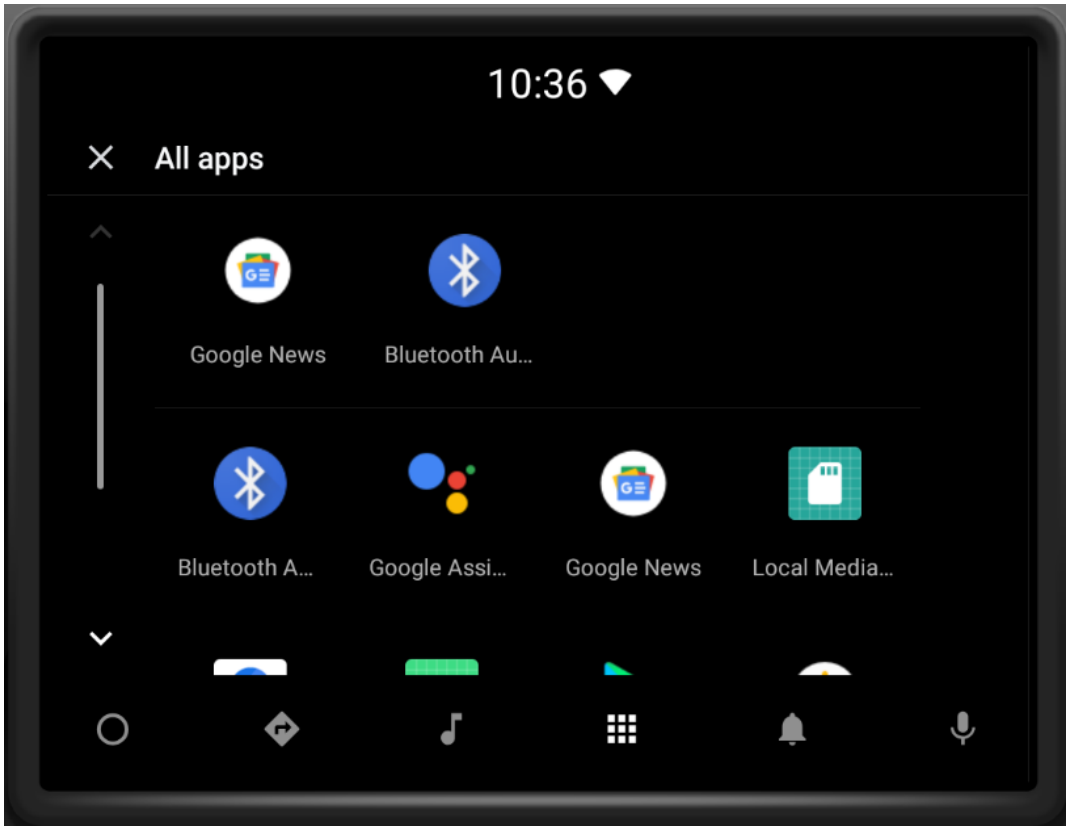
Kao što se može vidjeti u tablici, većina dopuštenja su potpisana ili privilegirana što znači da ih jedino izvorne (engl. *native*) aplikacije mogu koristit. Aplikacije treće strane upotrebljavaju normalna ili opasna dopuštenja. Opasna dopuštenja, *CAR_SPEED* i *CAR_ENERGY* su jedina u ovom trenutku koja zahtijevaju eksplicitni korisnikov pristanak. Sva dopuštenja mogu se naći u *android.car.permission* paketu [18].

3.5. EMULATOR

Razvojem samog Android Automotive operacijski sustava bilo je potrebno razviti i emulator koji će u potpunosti biti u mogućnosti simulirati informacijsko-zabavni sustav vozila, ali i programerima olakšati razvoj aplikacija. Emulator sadrži Google Play Store što znači da programeri mogu testirati aplikaciju, uključujući instaliranje i skidanje same aplikacije bez potrebe za automobilom koji sadrži Android Automotive operacijski sustav [19]. Također, nove verzije Android Studija koje podržavaju Android Automotive operacijski sustav imaju mogućnost korištenja dvije vrste virtualnih uređaja, što se može vidjeti na slici 3.3., dok se samo sučelje emulatora može vidjeti na slici 3.4.



Slika 3.3. Virtualni uređaji Android Automotive operacijskog sustava



Slika 3.4. Izgled sučelja emulatora

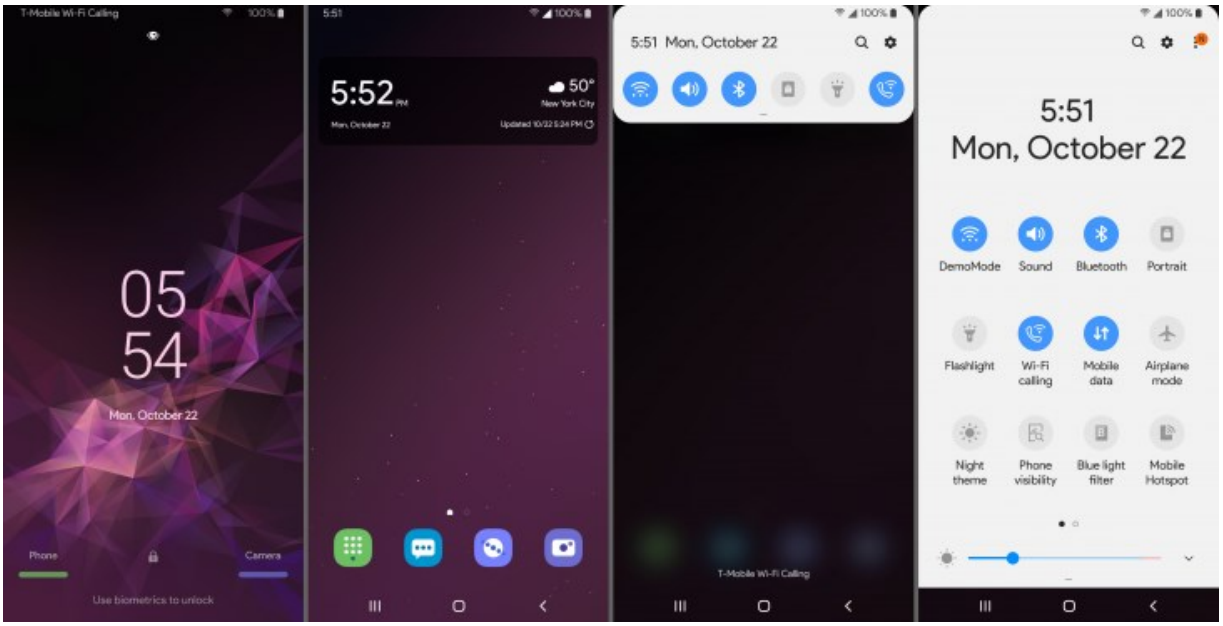
3.6. USPOREDBA ANDROID AUTOMOTIVE OPERACIJSKOG SUSTAVA I ANDROID OPERACIJSKOG SUSTAVA

Kao što je spomenuto u uvodu, Android Automotive operacijski sustav je Android operacijski sustav prilagođen za automobile. Nalazi se u istom spremniku i koristi istu bazu programskih kodova [12], što znači da programeri razvijaju aplikacije na jednak način kao i za mobilne uređaje uz male preinake. Također, koristi isti sigurnosni model, infrastrukturu, razvojne alate koji su također prilagođeni samom operacijskom sustavu. Na slikama 3.5. [20] i 3.6. [21] mogu se vidjeti razlike u sučeljima.

Glavna razlika između Android Automotive operacijskog sustava i Android operacijskog sustava je ta što je kod Android Automotive operacijskog sustava dodana podrška za različite značajke automobila kao što je pristup senzorima karakterističnima za automobil. Osim aplikacija koje su razvijene za Android Automotive operacijski sustav, on podržava i aplikacije razvijene za Android. To predstavlja programiram veliku prednost, zbog smanjenja posla. Jedina razlika u

programskom kodu aplikacije su dijelovi karakteristični za Android Automotive operacijski sustav, dok je većina programskog koda ista, odnosno zajednička za oba operacijska sustava.

Slika 3.5. Izgled sučelja Android Automotive operacijskog sustava



Slika 3.6. Izgled sučelja Android operacijskog sustav na mobitelima

3.7. USPOREDBA ANDROID AUTOMOTIVE I ANDROID OPERACIJSKOG SUSTAVA

Android Automotive operacijski sustav nije Android Auto. Najveća razlika je u tome, što je Android Automotive samostalni operacijski sustav koji se pokreće na sustavu vozila i ne ovisi o mobilnom uređaju, dok se kod Android Auto aplikacije vrte na mobilnom uređaju te se repliciraju na sustavu automobila [12]. Također, da bi se Android Auto mogao koristi, potrebno je mobilni uređaj povezati preko USB konekcije sa samim informacijsko-zabavnim sustavom. Međutim, moderni automobili dolaze i sa opcijom bežičnog spajanja mobilnog uređaja u svrhu korištenja Android Auta.

Većina današnjih automobila ima mogućnost korištenja Android Auta koji je zapravo odlična alternativa Android Automotivu. Najviše je popularan zbog mogućnosti korištenja Google mapa koje dolaze kao standard kod Android Automotiva. Na slikama 3.5. i 3.7. [22] mogu se vidjeti razlike u sučeljima Android Automotiva i Android Auta.

Slika 3.7. Izgled sučelja Android Autoa

4. USPOREDBA ANDROID AUTOMOTIVEA S OSTALIM SLIČNIM SUSTAVIMA

4.1. iDRIVE INFORMACIJSKO-ZABAVNI SUSTAV

iDrive je predstavlja informacijsko-zabavni sustav u BMW automobilima. Predstavljen je 2001. godine u modelu E65 serija 7. Sastoji se od LCD zaslona koji se nalazi na nadzornoj ploči vozila te kontrolnog gumba koji se nalazi na središnjoj konzoli i služi za upravljene kompletnim sustavom.

iDrive kao i Android Automotive operacijski sustav nudi pristup informacijama vozilima, namještanje klime, namještanje sjedala, odnosno nudi pristup sekundarnim sustavima u vozilu. Razne aplikacije kao što su Google mape ili Spotify nisu ugrađene u iDrive kao što je slučaj u Android Automotivu operacijski sustav, međutim nudi mogućnost integracije mobitelom putem BMW aplikacija. Drugim riječima, nudi mogućnost povezivanja mobilnog uređaja na informacijsko-zabavni sustav te omogućuje uvid u poruke ili informacije na društvenim mrežama, pristup Spotifyu, Pandora Radiou, itd [23].

BMW-ov informacijsko-zabavni sustav sadrži kontrolu pokreta (engl. *gesture control*). Ova značajka radi na način da pomoću 3D kamera snima korisnikove geste rukama. Pomoću ove značajke vozač može kontrolirati telefonske pozive, funkcije radia, navigaciju, pomoćne kamere za vožnju, itd. Ovim se postiže napredak u sigurnosti upravljanja vozilom jer vozač nije usredotočen na informacijsko-zabavni sustava vozila već na samu vožnju [24]. Android Automotive operacijski sustava ne sadrži ovu mogućnost, odnosno puni potencijal samog operacijskog sustava s gledište upravljanja pokretima ruku nije još iskorišten [25].

Kao i Android Automotive operacijski sustav, iDrive nudi aplikacije razvijene za sam sustav kao što su prognoza vremena, vijesti, email te BMW Connected. Također, BMW-ov osobni sustav nudi mogućnost prepoznavanja glasa te upravljanja sustavom pomoću glasa, isto kao i Android Automotive operacijski sustav pomoću Google Assistant koji je, ipak, superiorniji. Kao i većina modernih sustava nudi mogućnost korištenja Android Auta. Na slici 4.1. [26] i 4.2. [26] možemo vidjeti izgleda sučelja iDrivea.

Slika 4.1. Izgled osnovanog sučelja iDrive informacijsko-zabavnog sustava

Slika 4.2. Izgled iDrive navigacija

4.2. MERCEDES-BENZ USER EXPERIENCE INFORMACIJSKO-ZABAVNI SUSTAV

MBUX (engl. *Mercedes-Benz User Experience*) je informacijsko-zabavni sustav koji je razvio Mercedes. Mnogi smatraju da je to jedan, ako ne i najbolji sustav koji se danas može pronaći u automobilu. Jedinstvena značajka ovog sustava je korištenje umjetne inteligencija koja sustavu omogućava učenje. Također, može se individualizirati i prilagoditi korisniku [27]. Sustav se sastoji od dva ekrana:

- digitalnog sučelja ispred vozača
- ekrana koji se nalazi na nadzornoj ploči.

Sustav nudi namještanje ugođaja u automobilu, kontrolirajući ambijentalna svjetla, može zapamtiti omiljene pjesme te pamti put do radnog mjesta i najčešće birane brojeve. Kao Android Automotive operacijski sustav, MBUX ima ugrađen sustav glasovnog upravljanja te se aktivira na rečenicu „Hej, Mercedes“ [29]. Također, kao i BMW-ov informacijsko-zabavni sustav iDrive, MBUX nudi mogućnost korištenja Android Auta. Slike 4.3. [29] i 4.4. [28] prikazuju izgled sučelja MBUX-a.

Slika 4.3. Sučelje MBUX informacijsko-zabavnog sustava

Slika 4.4. Izgled MBUX navigacije

4.3. AUDI MULTI MEDIA INTERFACE INFORMACIJSKO-ZABAVNI SUSTAV

Audi Muti Media Interface (engl. *Audi MMI*) predstavlja informacijsko-zabavni sustav koji je razvije od strane Audia. Kao i Android Automotive operacijski sustav ima mogućnost kontrole audio sustava, komunikacije, navigacije itd. Sastoji se od digitalnog sučelja ispred vozača te ekrana na nadzornoj ploči.

MMI navigacijski i informacijsko-zabavni sustav nudi prikaz slika Google Eartha što je najbliže onomu što nudi Android Automotive operacijski sustav, te informacije o vremenu leta, cijenama goriva i naplati parkinga. Također, dostupne su i razne aplikacije koje su razvijene posebno za sam sustav te prognoza, vijesti i prometne informacije ili aktualne slike putem Google Eartha i Google Street Viewa ili Audi music streama [30].

Kao i ostali sustavi, nudi mogućnost upravljanja glasom koji olakšava telefoniranje i promjenu radio stanice, međutim nije toliko dobar kao ostali sustavi, pogotovo Google Assistant. Kao što je spomenuto u uvodu, Audi je pomogao u razvoju Android Automotive operacijskog sustava te se očekuje da će i Audijevi automobili u budućnosti biti opremljeni samim operacijskim sustavom.

Slike 4.5. [31] i 4.6. [32] prikazuju izgled sučelje, odnosno digitalnog kokpita, Audi MMI-a informacijsko-zabavnog sustava.

Slika 4.5. Izgled Audi MMI sučelje

Slika 4.6. Izgled Google Eartha (navigacija)

5. OPIS PRAKTIČNOG DJELA RADA

Praktični dio ovog rada predstavlja aplikacija za prikaz osnovnih informacija vozila te navigaciju. Osnovne informacije vozila dohvaćaju se preko senzora koji su karakteristični za Android Automotive studio, a što se tiče navigacije, ona je implementirana koristeći Mapbox API.

Kako bi uopće mogli pristupiti senzorima i ostalim karakteristikama auta, potrebno je kreirati objekt klase *Car*. To se vrši pomoću metode *createCar* koja se nalazi unutar klase *Car*, odnosno *Car* API-ja, što se može vidjeti u programskom kodu 5.1. Ta metoda povezuje *Car* objekt sinkrono sa *Car* servisom te kao parametre prima *context* i *ServiceConnection* objekt koji sadrži dvije metode, *onServiceConnected* i *onServiceDisconnected* [8]. Unutar metode *onServiceConnected* potrebo je kreirati instancu *CarSensorManager* koja nam omogućava pristup metodi *registerListener* koja kao parametar prima slušatelj (engl. *listener*) koji nam služi za slušanje promjena senzora.

```
car = Car.createCar(this, object : ServiceConnection {
    override fun onServiceDisconnected(name: ComponentName?) {
        TODO("Not yet implemented")
    }

    override fun onServiceConnected(name: ComponentName?, service: IBinder?) {
        TODO("Not yet implemented")
    }
})
```

Programski kod 5.1. Kreiranje *Car* objekta

Kako bi se pristupilo određenim senzorima, korisnik mora odobriti njihovu uporabu, dok se nekim senzorima može pristupiti bez korisnikove dozvole. Nažalost, određenim senzorima nije bilo moguće pristupiti jer ubrajaju u kategoriju *signature* ili *signature/privileged* što se može vidjeti u podpoglavlju 3.4.

5.1 SENSOR_TYPE_CAR_SPEED

SENSOR_TYPE_CAR_SPEED predstavlja senzor koji se koristi kada želimo pristupiti podatku vezanom za brzinu vozila. Kako bi pristupili brzini vozila, potrebno je registrirati slušatelj, u ovom slučaju pod nazivom *onSensorListener*, pomoću metode *registerListener* koji je zadužen za praćenje promjena vezanih za brzinu te se pokreće svaki put kad dođe do same promjene u vrijednosti senzora brzine. Osim slušatelja, *registerListener* prima i tip senzora, u ovom slučaju

SENSOR_TYPE_CAR_SPEED te *rate* koji određuje brzinu čitanja senzora (Programski kod 5.2). Također, potrebno je implemenirati *carSensorListener* tipa *CarSensorManager.onSensorChangeListener* (Programski kod 5.3.).

```
CoroutineScope(Dispatchers.IO).launch {
    carSensorManager.registerListener(
        carSensorListener,
        CarSensorManager.SENSOR_TYPE_CAR_SPEED,
        CarSensorManager.SENSOR_RATE_NORMAL
    )
```

Programski kod 5.2. Registrirani slušatelj sa senzorom *SENSOR_TYPE_CAR_SPEED*

```
private val carSensorListener = object : CarSensorManager.OnSensorChangedListener {
    override fun onSensorChanged(p0: CarSensorEvent?) {
        TODO("Not yet implemented")
    }
}
```

Programski kod 5.3. Implementacja *carSensorListener* sljušatelja

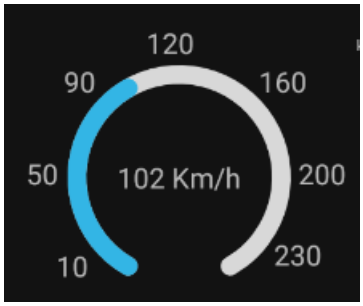
Na slici se može vidjeti da slušatelj ima ugrađenu metodu *onSensorChanged* koja se pokreće kada dođe do promjene senzora. Ta metoda prima *CarSensorEvent* kao parametar. *CarSensorEvent* sadrži informacije o tipu senzora, ali i o njegovo trenutnoj vrijednosti. U programskom kodu 5.4 može se vidjeti implementacija funkcije *onSesnorChanged*.

Pomoću svojstva *sensorType* potrebno je provjeriti radi li se o odgovarajućem senzoru, u ovom slučaju o *SENSOR_TYPE_CAR_SPEED*. Nakon toga, pomoću svojstva *floatValues* pristupa se samoj vrijednosti senzora. Na kraju se ispisuje vrijednost brzine na ekranu.

```
if (carEvent?.sensorType == CarSensorManager.SENSOR_TYPE_CAR_SPEED) {

    speed.text = (carEvent.floatValues[0] * 3.6f).toInt()
        .toString() + " Km/h"
    saSpeed.progress = ((carEvent.floatValues[0] * 3.6f) / 2.5).toInt()
    y = (carEvent.floatValues[0] * 3.6f).toInt()
}
```

Programski kod 5.4. Pristup i ispis informacija vezanih za *SENSOR_TYPE_CAR_SPEED*

Također, kako bi vizualno ljepše izgledao prikaz promjene brzine, korišten je i SeekArc API za implementaciju kružne trake napretka (engl. *circle progress bar*). U programskom kodu 5.4. može se vidjeti metoda pod nazivom *progress* (*setProgress*) koja služi za postavljanje trake za napredak, ovisno o vrijednosti brzine, što je vizualno prikazano plavom bojom. Vizualni prikaz brzine može se vidjeti na slici 5.1.

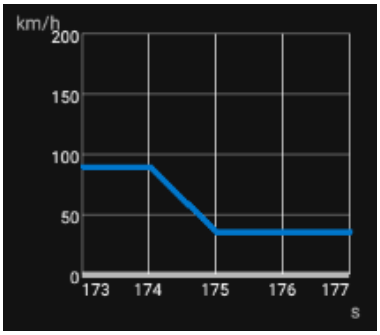


Slika 5.1. Vizualni prikaz brzine

Osim što je brzina auta prikazana trakom za napredak i tekstom unutar te trake, napravljen je i linearni graf koji pokazuje promjenu brzine u vremenu, što se može vidjeti u programskom kodu 5.5. Svake sekunde graf se ažurira, odnosno dodaje novu točku. To je postignuto pomoću *Runnable* sučelje i *Handler* klase. Graf je kreiran pomoću *GraphView* biblioteke te se može vidjeti na slici 5.9.

```
fun startRunnable() {
    runnable = object : Runnable {
        override fun run() {
            seriesSpeed.appendData(DataPoint(x.toDouble(), y.toDouble()), false, 5)
            x += 1
            graphSpeed.addSeries(seriesSpeed)
            handler.postDelayed(this, 1000)
        }
    }
    handler.post(runnable)
}
```

Programski kod 5.5. Implementacija *Runnable* objekta



Slika 5.9. Prikaz grafa promjene brzine

5.2 SENSOR_TYPE_ENV_OUTSIDE_TEMPERATURE

SENSOR_TYPE_ENV_OUTSIDE_TEMPERATURE predstavlja senzor koji prikazuje informacije vezane za vanjsku temperaturu. Kao i kod ostalih senzora, prije svega, potrebno je registrirati

slušatelja, ali ovaj put funkciji *registerListener* prosljeđuje se *SENSOR_TYPE_ENV_OUTSIDE_TEMPERATURE*. U programskom kodu 5.6. može se vidjeti dio koda koji obavlja taj posao.

```
carSensorManager.registerListener(
    carSensorListener,
    CarSensorManager.SENSOR_TYPE_ENV_OUTSIDE_TEMPERATURE,
    CarSensorManager.SENSOR_RATE_NORMAL
)
```

Programski kod 5.6. Registrirani slušatelj sa senzorom SENSOR_TYPE_ENV_OUTSIDE_TEMPERATURE

Kao što se može vidjeti u programskom kodu 5.6, sve izgleda isto kao i kod SENSOR_TYPE_CAR_SPEED, samo što je proslijeđen drugačiji senzor. Prilikom promjene vrijednosti temperature, pokreće se *carSensorListener* sa svojom funkcijom *onSensorChanged*. Unutar te funkcije provjerava se tip senzora (programski kod 5.7.). Vizualni prikaz temperature, odnosno ispis temperature može se vidjeti na slici 5.2.

```
if (carEvent?.sensorType == CarSensorManager.SENSOR_TYPE_ENV_OUTSIDE_TEMPERATURE) {
    tvTemp.text = carEvent.floatValues[0].toString() + "°C"
}
```

Programski kod 5.7. Pristup i ispis informacija vezanih za *SENSOR_TYPE_ENV_OUTSIDE_TEMPERATURE*

Slika 5.2. Vizualni prikaz temperature

5.3. SENSOR_TYPE_PARKING_BRAKE

SENSOR_TYPE_PRKING_BRAKE predstavlja senzor koji prikazuje informacije vezane za položaj ručne kočnice, dignuta ili spuštena. Kao i kod ostalih senzora, prije svega, potrebno je registrirati slušatelj, ali ovaj put funkciji *resgistreListener* prosljeđuje se *SENSOR_TYPE_PARKING_BRAKE*. U programskom kodu 5.8. može se vidjeti dio koda koji obavlja taj posao.

```
carSensorManager.registerListener(
    carSensorListener,
    CarSensorManager.SENSOR_TYPE_PARKING_BRAKE,
    CarSensorManager.SENSOR_RATE_NORMAL
)
```

Programski kod 5.8. Registrirani slušatelj sa senzorom *SENSOR_TYPE_PARKING_BRAKE*

Kao što se može vidjeti na slici, sve izgleda isto kao i kod ostalih senzora, samo što je proslijeđen drugačiji senzor. Prilikom promjene vrijednosti, odnosno položaja ručne kočnice, pokreće se *carSensorListener* sa svojom funkcijom *onSensorChanged*. Unutar te funkcije provjerava se tip senzora - ako je on *SENSOR_TYPE_PARKING_BRAKE* izvrši se programski kod 5.9. Sam prikaz vrijednosti senzora, odnosno položaja ručne kočnice, nije tekstualni već je predočen malom slikom koja predstavlja lampicu u automobilu.

```
if (carEvent?.sensorType == CarSensorManager.SENSOR_TYPE_PARKING_BRAKE) {
    if (carEvent.intValues[0] == 1) {
        ivParkingBrake.setColorFilter(getColor(this@MainActivity, R.color.red))
    } else {
        ivParkingBrake.setColorFilter(getColor(this@MainActivity, R.color.white))
    }
}
```

Programski kod 5.9. Pristup i ispis informacija vezanih za *SENSOR_TYPE_PARKING_BRAKE*

Na slici se može vidjeti upotreba metode *setColorFilter*. Ova metoda mijenja boju slike ovisno o tome je li ručna kočnica podignuta ili spuštena. Ako je podignuta, vrijednost *intValues* je 1 te se boja slike postavlja u crvenu boju, ako je ta vrijednost 0 slika ostaje bijela. Na slikama 5.3. i 5.4. možemo vidjeti kako to izgleda.

Slika 5.3. Ručna kočnica povučena

Slika 5.4. Ručna kočnica spuštena

5.4. SENSOR_TYPE_IGNITION_STATE

SENSOR_TYPE_IGNITION_STATE predstavlja senzor koji prikazuje informacije vezane za rad motora, odnosno radi li motor ili ne. Kao i kod ostalih senzora, prije svega, potrebno je registrirati slušatelj, ali ovaj put se funkciji *registerListener* prosljeđuje *SENSOR_TYPE_IGNITION_STATE* (programski kod 5.10.).

```
carSensorManager.registerListener(
    carSensorListener,
    CarSensorManager.SENSOR_TYPE_IGNITION_STATE,
    CarSensorManager.SENSOR_RATE_NORMAL
)
```

Programski kod 5.10. Registrirani slušatelj sa senzorom *SENSOR_TYPE_IGNITION_STATE*

Kao što se može vidjeti na primjeru programskog koda 5.10., sve izgleda isto kao i kod ostalih senzora, samo što je proslijeđen drugačiji senzor. Prilikom promjene vrijednosti, odnosno položaja ručne kočnice, pokreće se *carSensorListener* sa svojom funkcijom *onSensorChanged*. Unutar te funkcije provjerava se tip senzora - ako je on *SENSOR_TYPE_IGNITION_STATE* izvrši se programski kod 5.11. Sam prikaz vrijednosti senzora, odnosno radi motor automobila ili ne, nije tekstualni već je predočen malom slikom koja predstavlja lampicu u automobilu.

```
if (carEvent?.sensorType == CarSensorManager.SENSOR_TYPE_IGNITION_STATE) {
    if (carEvent.intValues[0] == 1) {
        ivIgnition.setColorFilter(getColor(this@MainActivity, R.color.red))
    } else {
        ivIgnition.setColorFilter(getColor(this@MainActivity, R.color.white))
    }
}
```

Programski kod 5.11. Pristup i ispis informacija vezanih za *SENSOR_TYPE_IGNITION_STATE*

Kao i kod *SENSOR_TYPE_PARKING_BRAKE*, koristi se *setColorFilter* za promjenu boje slike. U ovom slučaju bijela boja ukazuje na uključenost motora, a crvena na isključenost. Na slikama 5.5. i 5.6. može se vidjeti kako to vizualno izgleda.

Slika 5.5. Motor uključen

Slika 5.6. Motor uključen

5.5. SENSOR_TYPE_GEAR

SENSOR_TYPE_IGNITION_STATE predstavlja senzor koji prikazuje informacije vezane za položaj mjenjača. Kao i kod ostalih senzora, prije svega, potrebno je registrirati slušatelj, ali ovaj put funkciji *registerListener* prosljeđuje se *SENSOR_TYPE_TYPE_GEAR* (Programski kod 5.12.).

```
carSensorManager.registerListener(
    carSensorListener,
    CarSensorManager.SENSOR_TYPE_GEAR,
    CarSensorManager.SENSOR_RATE_NORMAL
)
```

Programski kod 5.12. Registracija slušatelja za SENSOR_TYPE_GAR

Prilikom promjene vrijednosti, odnosno položaja mjenjača, pokreće se *carSensorListener* sa svojom funkcijom *onSensorChanged*. Unutar te funkcije provjerava se tip senzora - ako je on SENSOR_TYPE_GEAR izvrši se programski kod 5.13. Položaj mjenjača predočen je prirodnim brojevima 1, 2, 4, 8 te ovisno o vrijednosti koju sadrži *intValues*, postavlja se boja teksta, odnosno slova koja označavaju položaj mjenjača što se može vidjeti na slikama 5.7., 5.8., 5.9., 5.10.

```
if (carEvent?.sensorType == CarSensorManager.SENSOR_TYPE_GEAR) {
    tvNeutral.setTextColor(Color.WHITE)
    tvReverse.setTextColor(Color.WHITE)
    tvPark.setTextColor(Color.WHITE)
    tvDrive.setTextColor(Color.WHITE)

    when (carEvent.intValues[0]) {
        1 -> tvNeutral.setTextColor(Color.RED)
        2 -> tvReverse.setTextColor(Color.RED)
        4 -> tvPark.setTextColor(Color.RED)
        8 -> tvDrive.setTextColor(Color.RED)
    }
}
```

Slika 5.13. Pristup i ispis informacija vezanih za SENSOR_TYPE_GEAR

Slika 5.7. Položaj mjenjača u PARK

Slika 5.8. Položaj mjenjača u REVERSE

Slika 5.9. Položaj mjenjača u NEUTRAL

Slika 5.10. Položaj mjenjača u DRIVE

5.6. SENSOR_TYPE_FUEL_LEVEL

SENSOR_TYPE_FUEL_LEVEL predstavlja senzor koji prikazuje informacije vezane za nivo goriva u spremniku. Kao i kod ostalih senzora, prije svega, potrebno je registrirati slušatelj, ali

ovaj put funkciji *registerListener* prosljeđuje se *SENSOR_TYPE_FUEL_LEVEL* (Programski kod 5.14.).

```
carSensorManager.registerListener(
    carSensorListener,
    CarSensorManager.SENSOR_TYPE_FUEL_LEVEL,
    CarSensorManager.SENSOR_RATE_NORMAL
)
```

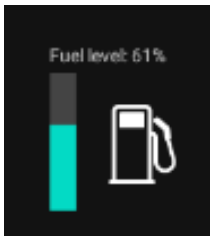
Slika 5.14. Pristup i ispis informacija vezanih za *SENSOR_TYPE_FUEL_LEVEL*

Prilikom promjene vrijednosti nivoa goriva u spremniku, pokreće se *carSensorListener* sa svojom funkcijom *onSensorChanged*. Unutar te funkcije provjerava se tip senzora - ako je on *SENSOR_TYPE_FULE_LEVEL* izvrši se programski kod 5.15. Također, u programskom kodu 5.15. može se vidjeti korištenje trake za napredak *pbBatteryLevel* kojoj je cilj predočiti koliko je još ostalo goriva u spremniku automobila. Metoda *ofInt* koja je dio klase *ObjectAnimator* služi za „popunjavanje“ trake do određene vrijednosti, u ovom slučaju do vrijednosti *currentProgress*. Ako je vrijednost unutar *floatValues* veća od 65 litara, onda je *curretnProgress* po unaprijed zadanoj vrijednosti 65 litara. Također, ako je u spremniku goriva ostalo manje od šesnaestine goriva, tada se pali lampica.

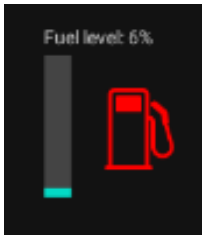
```
if (carEvent?.sensorType == CarSensorManager.SENSOR_TYPE_FUEL_LEVEL) {
    if (carEvent.floatValues[0] <= 65000) {
        currentProgress =  carEvent.floatValues[0].toInt()

        ObjectAnimator.ofInt(pbBatteryLevel, "progress", currentProgress)
            .setDuration(2000)
            .start()

        fuelLevel = (carEvent.floatValues[0] / 65000) * 100
        tvFuelLevel.text = "Fuel level: ${(fuelLevel).toInt()}%"
        if (carEvent.floatValues[0] <= 4062) {
            ivFuel.setColorFilter(getColor(this@MainActivity, R.color.red))
        } else {
            ivFuel.setColorFilter(getColor(this@MainActivity, R.color.white))
        }
    } else {

        ObjectAnimator.ofInt(pbBatteryLevel, "progress", 65000)
            .setDuration(2000)
            .start()
    }
```

Programski kod 5.15. Pristup i ispis informacija vezanih za SENSOR_TYPE_FUEL_LEVEL

Na slikama 5.11. i 5.12. može se vidjeti vizualni prikaz informacija vezanih za *SENSOR_TYPE_FUEL_LEVEL*.



Slika 5.11. Razina goriva iznad 4 litre



Slika 5.12. Razina goriva ispod 4 litre

5.7. NAVIGACIJA

U svrhu izrade navigacije, korišten je Navigation SDK koji omogućuje korištenje ruta, glasovnih uputa, preciznu detekciju trenutne lokacije itd. Osim Navigation SDK korišten je i Maps SDK koji omogućuje prikazivanje mape, odabir izgleda mape te dodavanje izvora i slojeva na mapu. Prvi korak izrade navigacije je postaviti mapu. Funkcijom *getInstance*, koja je dio *Mapbox* klase, dohvaćamo instancu mape te primamo *context* i privatni token koji je potrebno kreirati. *Mapbox* klasa predstavlja ulaznu točku za inicijalizaciju Mapbox SDK [33]. Nakon toga, pomoću metode *getMapAsync*, koja je dio klase *MapView*, potrebno je postaviti *callback* koji će se aktivirati kada je instanca *MapboxMap* spremna za upotrebu, što se može vidjeti na programskom kodu 5.16. *MapView* klasa se koristiti za prikaz podataka na karti i za manipulaciju sadržaja mape te za centriranje karte na zadanu koordinatu i za određivanje veličine područja koje se želi prikazati [24].

```
Mapbox.getInstance(this, getString(R.string.mapbox_access_token))
setContentView(R.layout.activity_main)
mapView = findViewById(R.id.mapView)
mapView.onCreate(savedInstanceState)
mapView.getMapAsync(this)
```

Programski kod 5.16. Upotreba *MapView* klase (postavljanje mape)

Callback metoda, koja se poziva funkcijom *getMapAsync*, je *onMapReady* (Programski kod 5.17.). Ova metoda dio je interfacea *OnMapReadyCallbac*. Unutar te metode potrebno je inicijalizirati

objekt klase *MapboxMap*. Ova klasa omogućuje interakciju sa Mapbox SDK-a te otkriva ulaznu točku za sve metode koje su povezane s *MapViewom*. Objekt te klase se ne može izravno instancirati, nego se mora dobiti iz metode *getMapAsync* na *MapViewu* koji je dodan u aplikaciji [19].

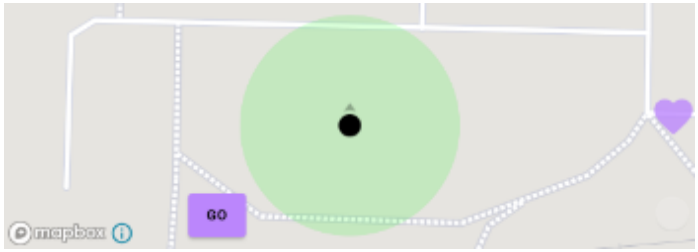
```
 @SuppressLint("MissingPermission")
override fun onMapReady(mapboxMap: MapboxMap) {
    mapBox = mapboxMap
    mapBox.setStyle(Style.MAPBOX_STREETS) { style ->
        findCurrentLocation(style)
```

Programski kod 5.17. *onMapReady* metoda

U programskom kodu 5.17. može se vidjeti metoda *setStyle*. Ona služi za postavljanje odnosno učitavanje stila kojeg smo proslijedili kao parametar. Nakon toga, poziva se funkcija *findCurrentLocation* koja omogućuje prikaz trenutne lokacije korisnika. Ona prima *Style* objekt koji je zapravo *callback* metode *setStyle*. *Style* objekt poziva se kada je mapa uspješno učitana. Pomoću *LocationComponent* locira se trenutna korisnikova lokacija, te se postavljaju određene opcije kao što su *isLocationComponentEnabled* - „točkica“ koja prikazuje korisnikovu lokaciju na mapi ili *camerMode* koji određuje kako će aplikacija pratiti korisnikovu lokaciju (programski kod 5.18.). Vizualni prikaz korisnikove lokacije može se vidjeti na slici 5.13.

```
 locationComponent = mapBox.locationComponent.apply {
    activateLocationComponent(locationComponentActivationOptions)
    isLocationComponentEnabled = true
    cameraMode = CameraMode.TRACKING
    renderMode = RenderMode.COMPASS
    Log.d("location1", this.lastKnownLocation?.latitude.toString())
}
```

Programski kod 5.18. Pronalazak korisnikove trenutne lokacije



Slika 5.13. Korisnikova lokacija

Nakon što je implementirana mogućnost prikaza trenutne korisnikove lokacije, potrebno je implementirati način na koji će korisnik moći odrediti destinaciju na koju se želi odvesti. To je

riješeno pomoću *onMapLongClick* metode koja je dio *OnMapLongClickListenea* sučelja. Međutim, kako bi se korisniku vizualno prikazala ruta i destinacija, potrebno je unutar metode *setStyle* (Programski kod 5.17.) dodati izvore, slojeve i slike koje će se prikazati na mapi, a dodaju se preko *Style* objekta koji je *callback*.

Prije svega, potrebno je pomoću metode *addSource* dodati 2 nova *GeoJsonSource* izvora kojima se prosljeđuje *id*. Prvi izvor predstavlja *click* odnosno korisnikovo odredište, a drugi liniju koja iscrtava rutu do odredišta. Također, pomoću metode *addImage* dodajemo sliku koja će biti korištena u samom stilu mape, u ovom slučaju crveni „pin“ (Programski kod 5.19.).

```
style.addSource(GeoJsonSource("CLICK_SOURCE"))
style.addSource(GeoJsonSource("ROUTE_LINE_SOURCE_ID", GeoJsonOptions().withLineMetrics(true)))
style.addImage("ICON_ID", BitmapUtils.getBitmapFromDrawable(ContextCompat.getDrawable(this,
R.drawable.mapbox_marker_icon_default))!!)
```

Programski kod 5.19. Dodavanje izvora na mapu

Ove izvore koristit ćemo za kreiranje slojeva koji će se prikazivati kao „pin“ odredišta, odnosno linija do odredišta. Linija predstavlja donji sloj i dodaje se kao *LineLayer* pomoću metode *addLayerBelow*. *LineLayer* metoda unutar *addLayerBelow* metode prima *id* izvora *ROUTE_LINE_SOURCE_ID* i *id* sloja *ROUTE_LAYER_ID* te se pomoću metode *withProperties* postavljaju određena svojstva (Programski kod 5.20.).

```
style.addLayerBelow(
    LineLayer("ROUTE_LAYER_ID", "ROUTE_LINE_SOURCE_ID")
        .withProperties(
            lineCap(LINE_CAP_ROUND),
            lineJoin(LINE_JOIN_ROUND),
            lineWidth(6f),
            lineGradient(
                interpolate(
                    linear(),
                    lineProgress(),
                    stop(1.0f, color(Color.RED)),
                    stop(5.0f, color(Color.BLUE)),
                    stop(10.0f, color(Color.GREEN))
                )
            )
        ),
    "mapbox-location-shadow-layer"
)
```

Programski kod 5.20. Dodavanje sloja linije pomoću *addLayerBelow* metodom

Također, „pin“ odredišta potrebno je kreirati kao sloj i to iznad sloja linije pomoću metode *addLayerAbove*. Njega se dodaje iznad *LineLayera* sa *ROUTE_LAYER_ID*. Sam „pin“ se dodaje

metodom *SymboLayer* te se pomoću metode *withProperties* dodaje slika koju smo dodali. Programski kod 5.21. može se vidjeti ispod.

```
style.addLayerAbove(
    SymbolLayer("CLICK_LAYER", "CLICK_SOURCE")
        .withProperties(
            iconImage("ICON_ID"),
            textField("Destination")
        ),
    "ROUTE_LAYER_ID"
)
```

Programski kod 5.21. Dodavanje sloja odredišta (pin) pomoću metode *addLayerAbove*

Kada smo kreirali i dodali sve slojeve na mapu, potrebno ih je pokazati na mapi. To se vrši kada korisnik izvrši događaj dugog klika na mapu te se poziva *callback* metoda *onMapLongClick* koja je dio *addOnMapLongClickListenera*. Unutar te metode potrebno je dohvatiti stil pomoću metode *getStyle*. Jedino na taj način možemo promijeniti vizualni izgled mape i prikazati slojeve koje smo dodali odnosno kreirali [24].

Unutar metode *getStyle* koja također ima *Style* objekt kao *callback*, koristi se metoda *getSourceAs<GeoJsonSource>* koja omogućuje dohvaćanje određenog izvora, u ovom slučaju izvora koji predstavlja „pin“ odredište. Nakon toga, metodom *setGeoJson* prikazuje se „pin“ odredišta na lokaciji (odnosno na mapi) koja je zabilježena prilikom klika na mapu. Obje metode su dio *Style* klase (Programski kod 5.22.).

```
override fun onMapLongClick(point: LatLng): Boolean {
    mapBox.getStyle {style ->
        val clickPointSource = style.getSourceAs<GeoJsonSource>("CLICK_SOURCE")
        clickPointSource?.setGeoJson(fromLngLat(point.longitude, point.latitude))
    }
```

Programski kod 5.22. Prikazivanje pina odredišta na mapi

Prije nego što prikažemo sloj linije koji predstavlja rutu, potrebno je kreirati *MapboxNavigation* objekt koji nam omogućuje upotrebu metode *requestRoutes*. Ova metoda služi za „*zatraživanje*“ rute te je dio klase *MapboxNavigation* koja predstavlja ulaznu točku za interakciju s Mapbox Navigation SDK. U programskom kodu 5.23. može se vidjeti način kreiranja *MapboxNavigation* objekt sa određenim metodama koje će biti objašnjene poslije.

```
mapboxNavigation = MapboxNavigationProvider.create(navigationOptions).apply {
    registerRouteProgressObserver(routeProgressObserver)
    registerBannerInstructionsObserver(bannerInstructionsObserver)
    registerTripSessionStateObserver(tripSessionStateObserver)
    attachFasterRouteObserver(fasterRouteObserver)
    registerOffRouteObserver(offRouteObserver)
    registerArrivalObserver(arrivalObserver)
}
```

Programski kod 5.23. Kreiranje *MapboxNavigation* objekt

Metoda *requestRoutes* prima dva parametra, *RouteOptions* objekt i *RoutesRequestCallback*. *RouteOptions* objekt kreira se pomoću *RouteOptions.builder* koji šalje zahtjev za dohvaćanje rute sa Mapbox Direction API-ja. Prilikom kreiranja zahtjeva, postoje tri metode koje se moraju koristiti, a to su *applyDeafultParams*, *accesToken* i *coordinates*. U *coordinates* se prosljeđuje lokacija korisnika i destinacija odredišta, koordinate za koje šaljemo zahtjev za dohvaćanje rute. Nakon što je kreiran zahtjev, potrebno je proslijediti i *RoutesRequestCallback* (Programski kod 5.24.).

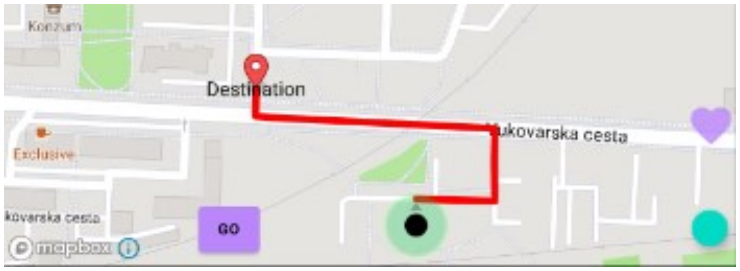
```
destinationPoint = fromLngLat(point.longitude, point.latitude)

locationComponent.lastKnownLocation?.let { originLocation ->
    originPoint = fromLngLat( originLocation.longitude, originLocation.latitude)

    mapboxNavigation?.requestRoutes(
        RouteOptions.builder().applyDefaultParams()
            .accessToken(getString(R.string.mapbox_access_token))
            .coordinates(originPoint, null, destinationPoint)
            .geometries(RouteUrl.GEOMETRY_POLYLINE6)
            .profile(RouteUrl.PROFILE_DRIVING)
            .alternatives(true)
            .voiceInstructions(true)
            .bannerInstructions(true)
            .steps(true)
            .voiceUnits(DirectionsCriteria.METRIC)
            .continueStraight(true)
            .build(),
        routesReqCallback
    )
```

Programski kod 5.24. Zahtjev za rutu i kreiranje zahtjeva

RoutesRequestCallback je slušatelj koji se poziva svaki put kada je novi skup ruta dostupan. Sadrži metodu *onRoutesReady* koja kao parametar ima listu ruta koje su dobivene sa Mapbox Direction API-a. Ruta pod indeksom 0 predstavlja primarnu rutu i ona se pokazuje na mapi. Unutar ove funkcije prikazuje se sloj linije na mapi koji predstavlja rutu (programski kod 5.25.). Na slici 5.14 može se vidjeti kako to izgleda na mapi.

```
private val routesReqCallback = object : RoutesRequestCallback {

    override fun onRoutesReady(routes: List<DirectionsRoute>) {
        Log.d("routes", routes[0].toString())
        if (routes.isNotEmpty()) {
            mapBox.getStyle {
                val clickPointSource =
it.getSourceAs<GeoJsonSource>("ROUTE_LINE_SOURCE_ID")
                val routeLineString = LineString.fromPolyline(
                    routes[0].geometry()!!,
                    6
                )
                clickPointSource?.setGeoJson(routeLineString)
            }
        }
        route_retrieval_progress_spinner.visibility = View.INVISIBLE
        mapBox.moveCamera(CameraUpdateFactory.zoomTo(15.0))
    }
```

Programski kod 5.25. Prikazivanje rute na mapi pomoć linije



Slika 5.14. Prikaz odredišta i rute

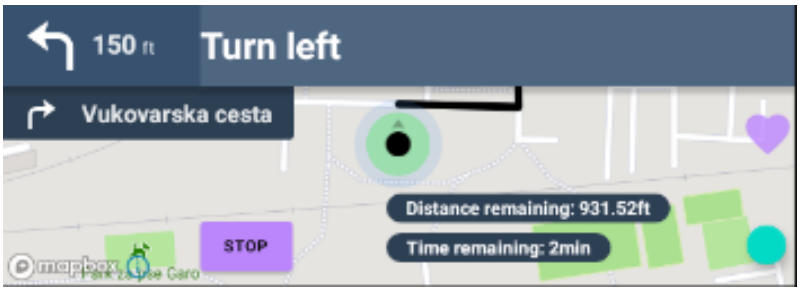
Kako bi navigacija bila potpuna, potrebno je dodati još par *listenra*. Najvažniji od njih su *BannerInstructionsObserver* i *RouteProgressObserver*. *BannerInstructionObserver* (Programski kod 5.26.) pokazuje korake do odredišta, odnosno kojim putem se treba kretati, a *RouteProgressObserver* (Programski kod 5.27.) služi za prikaz udaljenosti i vremena koji su ostali do odredišta. Vizualni prikaz može vidjeti na slici 5.15.

```
private val bannerInstructionsObserver = object: BannerInstructionsObserver {
    override fun onNewBannerInstructions(bannerInstructions: BannerInstructions) {
        Log.d("banner", bannerInstructions.toString())
        instructionView.updateBannerInstructionsWith(bannerInstructions)
        instructionView.toggleGuidanceView(bannerInstructions)
        instructionView.retrieveAlertView()
        instructionView.requestFocusFromTouch()

        if (bannerInstructions != null && imageProvider != null) {
            imageProvider.renderGuidanceView(bannerInstructions, guidanceViewImageProvider)
        }
    }
}
```

Programski kod 5.26. Kreranje *BannerInstructionsObserver* slušatelja

```
private val routeProgressObserver = object : RouteProgressObserver {
    override fun onRouteProgressChanged(routeProgress: RouteProgress) {
        instructionView.updateDistanceWith(routeProgress)
        wNVDurationRemaining.updateVisibility(true)
        wNVDistanceRemaining.updateVisibility(true)

        val durationRemaining: String = if(routeProgress.durationRemaining < 3600)

(TimeUnit.SECONDS.toMinutes(routeProgress.durationRemaining.toLong()).toInt()).toStri
ng() + "min"
        else

(TimeUnit.SECONDS.toHours(routeProgress.durationRemaining.toLong()).toInt()).toString
() + "h"

        val distanceRemaining: String = if(routeProgress.distanceRemaining < 1000)
            String.format("%.2f", routeProgress.distanceRemaining) + "ft"
        else
            String.format("%.2f", routeProgress.distanceRemaining / 5280) + "mi"

        wNVDurationRemaining.updateWayNameText("Time remaining: " +
durationRemaining)
        wNVDistanceRemaining.updateWayNameText("Distance remaining: " +
distanceRemaining.toString())
        Log.d("progess", routeProgress.distanceRemaining.toString())
    }
}
```

Programski kod 5.27. Kreiranje *RouteProgressObserver* slušatelja



Slika 5.15. Vizualni prikaz navigacije

Navigacija se pokreće na gumb, čime se poziva funkcija *startSession* koja pokreće aktivno stanje navigacije (Programski kod 5.28.). Također, osim *BannerInstructionObserver* mogu se registrirati još neki slušatelji kao što su *TipSessionStateObserver* koji prati je li navigacija aktivna ili ne, *FasterRouteObserever* koji traži brže rute, *OffRouteObserevr* koji detektira ako je korisnik „sišao“ s rute te *ArrivalObserver* koji obavještava korisnika da je stigao na odredište.

```
@SuppressLint("MissingPermission")
private fun setOnButtonClick() {
    buttonStartNav.setOnClickListener {
        updateCameraOnNavigationStateChange(true)
        mapboxNavigation!!.startTripSession()
        buttonStartNav.visibility = View.GONE
        buttonEndNav.visibility = View.VISIBLE

        val cameraPosition = CameraPosition.Builder()
            .target(LatLng(originPoint.latitude(), originPoint.longitude()))
            .zoom(16.5)
            .build()
        mapBox.moveCamera(CameraUpdateFactory.newCameraPosition(cameraPosition))
    }

    buttonEndNav.setOnClickListener {
        instructionView.visibility = View.GONE
        mapboxNavigation!!.stopTripSession()
        buttonEndNav.visibility = View.GONE
        buttonStartNav.visibility = View.VISIBLE
    }
}
```

Programski kod 5.28. Kod za pokretanje i prekid navigacije

Jedna od važnijih funkcija vezanih za navigaciju je dodavanje omiljenih odredišta, što će kasnije biti detaljnije objašnjeno. Omiljena odredišta se spremaju u lokalnu bazu podataka koristeći *Room*. Kako bi korisnikova nova omiljena destinacija bila odmah vidljiva, nakon pohrane u bazu podataka, koristi se *LiveData* u sklopu MVVM arhitekture (engl. *Model-View-ViewModel*). *LiveData* je klasa nositelja podataka promatrača koja je svjesna životnog ciklus, što znači da ažurira samo promatrače (engl. *Observer*) koji su u aktivnom stanju životnog ciklusa (*STARTED* ili *RESUMED*). Inače, kada se uređaj rotira, aktivnost (engl. *Activity*) se ponovno kreira, *onCreate* se ponovno poziva i postavljaju se UI komponente. Koristeći *LiveData* se to neće dogoditi jer *ViewModel* nije *lifecycle* klasa aktivnosti, tako da nema svoj vlastiti životni vijek, što znači da će svaki put kada se aktivnost stvori, odmah primiti trenutne podatke iz *ViewModela*. Pošto se *LiveData* može promatrati unutar određenog životnog ciklusa, znači da se *Observer* može dodati u paru sa *LifecycleOwner*, te će *Observer* biti obavješten o promjenama podataka jedino ako je *LifecycleOwner* u aktivnom stanju [34]. Kao što je već spomenuto, kao lokalna baza podataka korišten je *Room*. Prije svega potrebno je kreirati klasu koja će predstavljati tablicu u bazi podataka (Programski kod 5.29.).

```
 @Entity(tableName = "detination_entity")
data class DestinationEntity(
    @ColumnInfo(name = "destination_name")
    var destinationName: String,
    @ColumnInfo(name = "destination_lng")
    var destinationLng: Double,
    @ColumnInfo(name = "destination_lat")
    var destinationLat: Double
) {
    @PrimaryKey(autoGenerate = true)
    var id: Int? = null
}
```

Programski kod 5.29. Klasa koja predstavlja tablicu u bazi podataka

Kako bi se moglo pristupiti podacima iz tablice, potrebno je kreirati DAO (engl. *Data Access Object*) koji sadrži određene metode za upis podataka u tablicu, brisanje podataka, vraćanje podataka iz tablice itd. DAO se može vidjeti u programskom kodu 5.30.

```
@Dao
interface DestinationDao {

    @Insert(onConflict = OnConflictStrategy.REPLACE)
    suspend fun upsert(destination: DestinationEntity)

    @Delete
    suspend fun delete(destination: DestinationEntity)

    @Query("SELECT * FROM detination_entity")
    fun getAllDestinations(): LiveData<List<DestinationEntity>>

    @Query("SELECT destination_lat FROM detination_entity WHERE destination_name =
:name")
    suspend fun getDestinationLat(name: String): Double

    @Query("SELECT destination_lng FROM detination_entity WHERE destination_name =
:name")
    suspend fun getDestinationLng(name: String): Double
}
```

Programski kod 5.30. Kreiranje DAO objekta

Kao što se može vidjeti u programskom kodu 5.30., funkcije *upsert* i *delete* su *suspend* funkcije jer SQL (engl. *Structured Query Language*) ne dopušta pisanje ili brisanje na glavnoj niti, stoga se koriste korutine (engl. *corutines*). Nakon što je kreiran DAO, potrebno je kreirati samu bazu podataka. Programski kod 5.31 može se vidjeti ispod.

```
@Database(
    entities = [DestinationEntity::class],
    version = 1
)
abstract class DestinationDatabase: RoomDatabase() {

    abstract fun getDestinationDao(): DestinationDao

    companion object {
        @Volatile
        private var instance: DestinationDatabase? = null
        private val LOCK = Any()

        operator  fun invoke(context: Context) = instance
            ?: synchronized(LOCK) {
            instance
                ?: createDatabse(
                    context
                )
                    .also { instance = it }
        }

        private fun createDatabse(context: Context) =
            Room.databaseBuilder(context.applicationContext,
                DestinationDatabase::class.java, "DestinationDB.db").build()
    }
}
```

Programski kod 5.31. Implementacija baze podataka

Unutar *companion object* potrebno je kreirati instancu *DestinationDatabase* koja mora biti *singleton* kako u memoriji ne bi postajalo više instanci iste baze. Kako bi se to postiglo, iznad varijable *instance*, što se može vidjeti u programskom kodu 5.31., stavlja se *@Volotile* što osigurava da je pisanje u ovu varijablu odmah vidljivo drugim nitima kako ne bi postojalo više instanci. Funkcijom *invoke* vraća se instanca baze ako postoji, a ako ne postoji, kreiramo je funkcijom *createDatabase* unutar *synchronized* bloka. *Synchronized* blok osigurava da više niti, u isto vrijeme, ne može postaviti instancu za vrijeme kada se izvršava kod unutar tog bloka.

Pošto je riječ o MVVM arhitekturi, potrebno je imati repozitorij koji će dohvaćati podatke iz modela pristupajući DAO objektu preko funkcije *getDestinationDao* iz *DestinationDatabase* klase (Programski kod 5.31.). Repozitorij se može vidjeti u programskom kodu 5.32.

```
 class DestinationRepository(
    private val db: DestinationDatabase
) {
    suspend fun upsert(destination: DestinationEntity) =
db.getDestinationDao().upsert(destination)

    suspend fun delete(destination: DestinationEntity) =
db.getDestinationDao().delete(destination)

    fun getAllDestinations() = db.getDestinationDao().getAllDestinations()

    suspend fun getDestinationLat(name: String) =
db.getDestinationDao().getDestinationLat(name)

    suspend fun getDestinationLng(name: String) =
db.getDestinationDao().getDestinationLng(name)
}
```

Programski kod 5.32. Implementacija repozitorija

ViewModel ne pristupa podacima direktno već preko repozitorija. Funkcije koje su odgovorne za pisanje i brisanje izvršavaju se unutar „*CorutineScope.lunch*“ bloka. Programski kod 5.33. prikazuje implementaciju *ViewModela*. *DestinationViewModel*, koji predstavlja *ViewModel*, potrebno je kreirati preko *DestinationViewModelFactory* što se može vidjeti u programskom kodu 5.34., a postupak kreiranja može se vidjeti u programskom kodu 5.35.

```
class DestinationViewModel(
    private val repository: DestinationRepository
): ViewModel() {

    fun upsert(destination: DestinationEntity) =
CoroutineScope(Dispatchers.Main).launch {
        repository.upsert(destination)
    }

    fun delete(destination: DestinationEntity) =
CoroutineScope(Dispatchers.Main).launch {
        repository.delete(destination)
    }

    fun getAllDestinations() = repository.getAllDestinations()

    fun getDestinationLat(name: String) = runBlocking {
        repository.getDestinationLat(name)
    }

    fun getDestinationLng(name: String) = runBlocking {
        repository.getDestinationLng(name)
    }
}
```

Programski kod 5.33. Implementacija *DestinationViewModel* (*ViewModel*) klase

```
@Suppress("UNCHECKED_CAST")
class DestinationViewModelFactory(
    private val repository: DestinationRepository
): ViewModelProvider.NewInstanceFactory() {

    override fun <T : ViewModel?> create(modelClass: Class<T>): T {
        return DestinationViewModel(repository) as T
    }
}
```

Programski kod 5.34. Implemetacija *DestinationViewModelFactory* klase

```
val database = DestinationDatabase(this)
val repository = DestinationRepository(database)
val factory = DestinationViewModelFactory(repository)
val viewModel = ViewModelProviders.of(this,
factory).get(DestinationViewModel::class.java)
```

Programski kod 5.35. Postupak kreiranja DestinationViewModela

Nakon što je kreirana baza i kreiran *DestinationViewModel*, moguće je izvršiti pohranu i prikaz omiljenih odredišta. To se vrši na pritisak plavog gumba ispod gumba u obliku srca (slika 5.15.). Pritiskom na taj gumb iskače skočni izbornik (slika 5.16.) koji omogućuje upis imena nove omiljene destinacije te pritiskom na gumb *Save* sprema se destinacija u bazu podataka (Programski kod 5.45.).

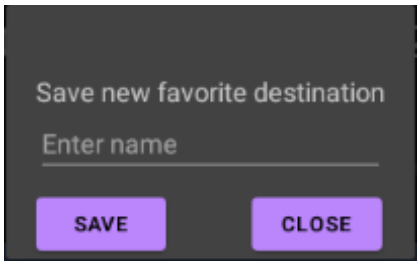
```
popupMenu.setOnMenuItemClickListener {menuItem ->
    val lat = viewModel.getDestinationLat(menuItem.title.toString())
    val lng = viewModel.getDestinationLng(menuItem.title.toString())
    val latlng = LatLng(lat, lng)
    Log.d("latlng", "Book")
    onMapLongClick(latlng)
}
```

Programski kod 5.36. Pohranjivanje destinacije u bazu podataka

Nakon što je pohranjena nova destinacija, *Observer* sluša promjene i poziva se kada do njih dođe te vraća listu destinacija (Programski kod 5.46.). Vizualni prikaz omiljenih odredišta može vidjeti na slici 5.17.

```
val popupMenu = PopupMenu(this, fabOpenDialog)
viewModel.getAllDestinations().observe(this, Observer { Destinations ->
    popupMenu.menu.clear()
    for(destination in Destinations) {
        popupMenu.menu.add(Menu.NONE, destination.id!!, destination.id!!,
destination.destinationName)
    }
})
```

Programski kod 5.37. Pristup podacima u bazi podataka



Slika 5.16. Skočni izbornik za pohranu omiljenog odredišta

Slika 5.17. Popis omiljenih odredišta

6. ZAKLJUČAK

Auto industrija iz dana u dan sve više i više raste i postaje najmoćnija industrija na svijetu. Automobili nisu više prijevozna sredstva koja čovjeku omogućuju jedino prijevoz od jedne točke do druge, već postaju autonomni roboti koji mogu prepoznavati glas, voziti samostalno te reagirati u slučaju opasnosti.

Mobitel je postao neizostavan u čovjekovom životu. On više nije dodatak, već predmet bez kojeg ljudi ne mogu živjeti. Google je to prepoznao i otišao korak dalje. Razvio je Android Automotive sustav, sustav koji će u budućnosti postati standardni informacijsko-zabavni sustav u automobilima, te njime nadogradio postojeći Android Auto. Android Automotive predstavlja sustav koji može upravljati senzorima automobila, prepoznaje glas, te nudi korisnicima aplikacije poput Google mapa i Spotify, koje u današnje vrijeme postaju neizostavne u svakodnevnoj upotrebi.

Svrha ovog rada bila je upoznati se s Android Automotive operacijskim sustavom, njegovim mogućnostima te prednostima koje nudi naspram ostalih informacijsko-zabavnih sustava. Android Automotive je jednostavan operacijski sustav za korištenje i nudi mogućnosti jednake onima na mobilnom uređaju. Ipak, najvažnije je to što korisniku nudi veliku razinu sigurnosti. Kako bi se što bolje upoznalo s Android Automotive operacijskim sustavom, napravljan je aplikacija.

Cilj aplikacije bio je upoznati se s Android Automotive operacijskim sustavom i što bolje razumjeti njegove mogućnosti. Aplikacija prikazuje trenutna stanja senzora automobila te navigaciju kojom korisnik može jednostavno rukovati. Kada bi uspoređivali Android Automotive i Andorid operacijski sustav, možemo uočiti povezanost ova dva operacijska sustava te primjenjivost istog programskog koda prilikom izrade aplikacije. Što se tiče razlika, one su minimalne, ali su značajne prilikom izrada aplikacija za Android Automotive operacijski sustav, kao naprimjer pristup senzorima automobila. Također, prilikom izrade navigacije, za koju je korišten Navigation SDK, mogla se vidjeti sličnost između ova dva operacijska sustava, što znači da se isti programski kod može koristiti prilikom izrade aplikacije za oba, ali i za Android Auto od kojeg je Android Automotive operacijski sustav superiorniji.

LITERATURA

[1] GENIVI Alliance, About GENIVI [online], 2020., dostupno na: https://www.genivi.org/about-genivi [prosinac, 2020.]

[2] GENIVI Alliance, Android Automotive SIG [online], 2020., dostupno na: https://at.projects.genivi.org/wiki/display/DIRO/Android+Automotive+SIG [prosinac, 2020.]

[3] N. Lomas, Google Announces Open Automotive Alliance To Drive Andorid Into Connected Cars [online], 2020., dostupno na: https://techcrunch.com/2014/01/06/oaa/?guccounter=1&guce_referrer=aHR0cHM6Ly93d3cuZ29vZ2xlLmNvbS8&guce_referrer_sig=AQAAADVI4Np76A0WKX4RwHvGVjyecpYbJIMg2Ex6eJh88fXHbATbdmiyaqoX_E-o2lKt6eYhFJnM6MYMtN-naH1ijlc1n8xnfJnBQiaOezLpF-qR68GelJnBaeg-BZHioezbFlO4Pf0DHXaXQZn6DJtzh6OPBlrdSex229Qs2Ld69RZ2%C5%BE, [prosinac, 2020].

[4] Open Automotive Alliance, Inroducing the open automotive alliance [online], 2014., dostupno na: https://www.openautoalliance.net/#about [prosinac, 2020.]

[5] C. Threewitt, How the Open Automotive Alliance Works [online], howstuffworks, 2014., dostupno na: https://auto.howstuffworks.com/open-automotive-alliance.htm [prosinac, 2020.]

[6] Project black, P3, 2020., https://www.p3-ds.com/portfolio/project-black/ [prosinac, 2020]

[7] COOL NIKS, GPS Seedometer and Odometer (Mileage Tracker), Google Play, San Francisco (Kalifornija), 2018.,dostupno na: https://play.google.com/store/apps/details?id=com.coolniks.niksgps&hl=en_US&gl=US [prosinac, 2020.]

[8] J. Moss, DigiHUD Speedometer [online], Google Play, Blackfordby (UK), 2016, dpstupno na: https://play.google.com/store/apps/details?id=org.mrchops.android.digihud&hl=en&gl=US

[9] M. Miłejko, Drag Racer Car Performance [online], Google Play, Warszawa (Poljska), 2019., dostupno na: https://play.google.com/store/apps/details?id=com.milejko.dragracer&hl=hr&gl=US

[10] Code Sector, SpeedView [online], Google Play, Sydney (Australija), 2010., dostupno na: https://play.google.com/store/apps/details?id=com.codesector.speedview.free&hl=hr&gl=US

[11] Binarytoys, Ulysse Speedometer [online], Google Play, Buetzberg, Switzerland, 2011., dostupno na: https://play.google.com/store/apps/details?id=com.binarytoys.speedometer&hl=en_US&gl=US

[12] Automotive [online], Android Open Source Project, 2020., https://source.android.com/devices/automotive [prosinac, 2020.]

[13] Nepoznati autor, CAN Bus Explained – A Simple Intro: What is CAN bus? [online], CSS Electronics, Aabyhoej (Danska), 2020., dostupno na: https://www.csselectronics.com/screen/page/simple-intro-to-can-bus [prosinac, 2020.]

[14] M. Pese i K. Shine, Security Analysis of Android Automotive [online – znanstveni rad], Georgia Institute of Technology, SAD, 2020., dostupno na: https://rtcl.eecs.umich.edu/rtclweb/assets/publications/2020/mert-sae-2020.pdf [prosinac, 2020.]

[15] Android Automotive (AOSP): What is Android Automotive (AOSP)? [online], CONJURE, UK, 2020., dostupno na: https://www.conjure.co.uk/services/android-automotive/ [prosinac, 2020.]

[16] A. Agarwal i K. Jaysukh Padia, Moving towards a complete android automotive OS for new age cars [online], Kalifornija (SAD), 2020., dostupno na: https://www.pathpartnertech.com/moving-towards-a-complete-android-automotive-os-for-new-age-cars/ [prosinac, 2020.]

[17] Permissions on Android [online], Andorid Developers, Kalifornija (SAD), 2020, dostupno na: https://developer.android.com/guide/topics/permissions/overview [prosinac, 2020.]

[18] AndroidManifest.xml [online], Android Google Source, 2015, dostupno na: https://android.googlesource.com/platform/packages/services/Car/+/master/service/AndroidManifest.xml [prosinac, 2020.]

[19] I. Bonifacici, Google's new emulator makes Android Automotive development easier: Developers no longer need a car to create apps for the infotainment system [online], engadget, 2019., dostupno na: https://www.engadget.com/2019-10-21-google-android-automotive-emulator.html [prosinac, 2020.]

[20] Nepoznati autor, We get a first glimpse of the Android Automtoive OS start apps [online], NewsBeezer, SAD, 2020., dostupno na: https://newsbeezer.com/aus/we-get-a-first-glimpse-of-the-android-automotive-os-start-apps/ [prosinac, 2020.]

[21] G. Gao, Another Galaxy S9 Android Pie leak shows more of the UI [online], Pinterest, 2018., dostupno na: https://www.pinterest.com/pin/5418462037678471/ [prosinac, 2020.]

[22] N. Šabić, Android Auto stiže u BiH, Hrvatsku, Makedoniju, Srbiju i još u 30-ak zemalja svijeta [online], Bajtbox, BiH, 2020., dostupno na: https://www.bajtbox.com/android-auto-stize-u-bih-hrvatsku-makedoniju-srbiju-i-jos-u-30-ak-zemalja-svijeta/ [prosinac, 2020.]

[23] Nepoznati autor, BMW Apps Retrofit [online], BimmerTech, dostupno na: https://www.bimmer-tech.net/category-bmw-apps-retrofit [prosinac, 2020.]

[24] U. I Nduanya, C. V Oleka, E. Z. Orji, DESIGN AND IMPLEMENTATION OF ARDUINO-BASED GESTURECONTROLLED SYSTEM WITH ACCELEROMETER [online - znanstveni rad], Department of Computer Engineering, Enugu State University of Science and Technology (ESUT), Enugu State, Nigeria, 2019., dostupno na: https://www.ijrter.com/papers/volume-4/issue-7/design-and-implementation-of-arduino-based-gesture-controlled-system-with-accelerometer.pdf

[25] B. Hermann, Exposing a Gesture Sensor in Android Automotive OS [online], Android Automotive OS book, 2020., dostupno na: https://www.androidautomotivebook.com/the-next-big-thing-exposing-a-gesture-sensor-in-android-automotive-embedded-os/ [prosinac, 2020.]

[26] T. Fraser, BMW iDrive OS 7.0 review [online], WhichCar, Australija, 2020., dostupno na: https://www.whichcar.com.au/car-reviews/bmw-idrive-review [prosinac, 2020.]

[27] Mercedes-Benz: Mercedes-Benz User Experience [online], Mercedes-Benz, Hrvatska, 2020., dostupno na: https://www.mercedes-benz.hr/osobna-vozila/mercedes-benz-vozila/mbux.html [prosinac, 2020.]

[28] MBUX – MERCEDES-BENZ USER EXPERIENCE [online], Gašparić Grupa, Hrvatska, dostupno na: https://www.auto.hr/novosti/mbux-mercedes-benz-user-experience/1826 [prosinac, 2020.]

[29] D. Fung, BUX will 'quickly proliferate' throughout the Mercedes-Benz range: The new MBUX infotainment system will be offered in quick order in other Mercedes-Benz models [online], CarAdvice, Australija, 2018., dostupno na: https://www.caradvice.com.au/619662/mbux-will-quickly-proliferate-throughout-the-mercedes-benz-range/ [prosinac, 2020.]

[30] Audi connect [online], Audi Hrvatska, Hrvatska, 2020, dostupno na: https://www.audi.hr/audi-iskustvo-audi-svijet/audi-connect/audi-connect-usluge/unos-cilja-preko-myaudi-ili-google-maps [prosinac, 2020.]

[31] G. King, What is Audi MMI? Is it worth it? [online], Carwow, London (UK), 2020., dostupno na: https://www.carwow.co.uk/guides/glossary/what-is-audi-mmi-0557 [prosinac, 2020.]

[32] Nepoznati autor, Guide: Enable Google Earth/Traffic on Canada spec Audi [online – forum] ,2017 ,dostupno na: https://www.audiworld.com/forums/a4-b9-platform-discussion-212/guide-enable-google-earth-traffic-canada-spec-audi-2912779/

[33] Maps SDK for Android (dokumentacija) [online], Washington, D.C (SAD), 2020., dostupno na: https://docs.mapbox.com/android/maps/guides/ [prosinac, 2020.]

[34] P. Lackner, WHAT IS MVVM [online], 2019., dostupno na: https://www.youtube.com/watch?v=cfcyYO7osnk&list=PLQkwcJG4YTCT0RouHZ6sQlE4JE6VyD2zO [prosinac, 2020.]

SAŽETAK

Android Automotive sustav predstavlja varijaciju Android operacijskog sustava koja se samostalno, bez ikakvih dodatnih uređaja, vrti na informacijsko-zabavnim sustavima u vozilima. Sam sustav nudi jako puno mogućnosti programerima u razvijanju samostalnih aplikacija, zbog čega dolazi i sa velikom bazom programskog koda što olakšava sam razvoj novih aplikacija. Zadatak ovog rada bio je upoznati se s Android Automotive operacijskim sustavom, njegovom arhitekturom, sa sigurnosti zaštite podataka te dopuštenjima korištenja određenih senzora automobila. Osim toga, napravljana je usporedba s Android operacijskim sustavom na mobilnim uređajima, Android Autom te s informacijsko-zabavnim sustavima proizvođača automobila BMW, Audi i Mercedes. U svrhu ovog rada napravljena je aplikacija za Android Automotitve operacijski sustav koja prikazuje trenutna stanja senzora automobila te navigaciju kojom korisnik može jednostavno rukovati.

ABSTRACT

DEVELOPMENT OF APLICATIONS FOR INFOTAINMENT SYSTEMS IN VEHICLES

Android Automotive is a variation of the Android operating system that runs independently, without any additional devices, on in-vehicle infotainment systems. The system itself offers a lot of possibilities for developers in the development of standalone applications, which is why it comes with a large code base, which facilitates the development of new applications. Task of this paper was to get acquainted with the Android Automotive operating system, its architecture, data protection security and permissions to use certain car sensors. In addition, a comparison was made with the Android operating system on mobile devices, Android Auto and infotainment systems of car manufacturers such as BMW, Audi and Mercedes. For the purpose of this work, an application for the Android Automotive operating system was created, which shows the current states of the car's sensors and navigation that the user can easily operate.

ŽIVOTOPIS

Filip Gajari rođen je 14.04.1996 godine u Zagrebu. Osnovnu školi završio je u Križevcima, a opću gimnaziju „Fran Galović“ u Koprivnici. Školske godine 2015./2016. maturirao je, te je upisao Fakultet elektrotehnike, računarstva i informacijskih tehnologija u Osijeku, smjer Računarstvo. 2018. godine završava Preddiplomski studij Računarstva te upisuje diplomski studij Računarstva, smjer Programsko inženjerstvo. Osim toga, završio je 4. razreda glazbene škole „Albert Štriga“ u Križevcima, smjer violina.

Filip Gajari