INFORMACIJSKI SUSTAV ZA PRAĆENJE STATUSA AUTOMOBILA ZA VRIJEME VOŽNJE

Jurković, Tin

VISOKO UČILIŠTE ALGEBRA

ZAVRŠNI RAD

Informacijski sustav za praćenje statusa automobila za vrijeme vožnje

Tin Jurković

Zagreb, veljača 2020.

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

U Zagrebu, 18.02.2020.

TIN JURKOVIĆ

.

Predgovor

Želim zahvaliti profesoru dr. sc. Leu Mršiću te ostalim profesorima i djelatnicima na Visokom učilištu Algebra koji su mi svojim prenesenim znanjem i vještinama pomogli na akademskom putu. Posebno želim zahvaliti svojoj obitelji, prijateljima i poslovnim kolegama na neizmjernoj potpori tijekom obrazovanja. Njihova potpora mi je puno pomogla prilikom pisanja ovoga rada.

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

Sažetak

Veliki broj automobila, posebno onih starije proizvodnje, nemaju ugrađene napredne sustave za praćenje podataka iz vozila te su vozači ograničeni na svega nekoliko informacija o svome vozilu.

Ovaj rad usmjeren je na izradu aplikacije koja korisnicima vozila omogućava pristup informacijama do kojih je moguće doći obzirom na karakteristike vozila (proizvođač i godina proizvodnje) u jednostavnoj i prikladnoj formi putem mobitela. Podatci koje je u najvećem broju slučajeva moguće prikupiti su npr. uvid u potrošnju goriva, uvid u broj okretaja motora, uvid u temperaturu motora, grafički prikazi informacija kada i u kojem periodu dana je potrošnja goriva bila najveća. Kao dio rada, razvijena je aplikacija kojoj je prethodilo istraživanje na ispitnoj skupini od 10+ ispitanika/vozača kako bi se saznalo koje informacije u vozilu smatraju najpotrebnijima za vrijeme vožnje, koje informacije im trenutno nedostaju a željeli bi ih imati i te što bi im od informacija koje je moguće osigurati obzirom na karakteristike vozila bile korisne. U završnom dijelu rada, osim predstavljanja rezultata i same aplikacije, dokumentirane su sve faze testiranja i prikupljenih podataka/zaključaka a kao smjernice budućim programerima kako o samim karakteristikama korištenja tehnologije/platforme tako i o postupcima u fazi izrade ovakvih rješenja. Predstavljena je važnost ovakvih rješenja za razvoj automobilske industrije te s time povezane industrije razvoja aplikacija s posebnim naglaskom na potencijal ovakvih projekata kao jednostavnih, a efikasnih rješenja koja unapređuju korisničko iskustvo vozača i stupanj korisnosti kroz korištenje dodatnih informacija, a bez velikih investicija u novi automobil.

Cilj ovog rada je podići iskustvo vožnje na vozilima koja nemaju ugrađene napredne sustave za praćenje podataka iz vozila te su vozači ograničeni na svega par takvih informacija, na potpuno novu razinu s uvidom u informacije i podatke koji su karakteristični za više cjenovne klase automobila na tržištu.

Ključne riječi: OBD 2, praćenje statusa automobila, mobilna aplikacija, IOS, MVVM arhitektura koda

Summary

Many cars, especially older ones, do not have advanced systems for monitoring vehicle data, and drivers are limited to just a few information about their vehicle.

This paper focuses on developing an application that allows vehicle users to access information that can be accessed by reference to vehicle characteristics (manufacturer, year of production) in a simple and convenient manner via mobile phone. The data that can be collected in most cases are: fuel consumption, engine speed, engine temperature, graphical information on when and for what period of the day the fuel consumption was highest. As part of the research, an application was developed that was preceded by a survey on a test group of 10+ test subjects/drivers to find out what information they find most needed while driving, what information they currently lack, and what they would like to have information that could be provided given the characteristics of the vehicle was useful. In the final part of the paper, in addition to presenting the results and the application itself, all stages of testing and the collected data/conclusions are documented as guidance to future developers on the characteristics of the use of technology / platform as well as the procedures in the process of developing such solutions. The importance of such solutions for the development of the automotive industry and the related application development industry is presented, with particular emphasis on the potential of such projects as simple yet effective solutions that enhance the driver's experience and degree of utility through the use of additional information without major investment in a new car.

The aim of this research is to elevate the driving experience of vehicles that do not have advanced vehicle data monitoring systems installed, and are limited to a whole new set of information, to a whole new level of insight into the information and data that is characteristic of the higher priced car market.

Keywords: OBD 2, Car Status Tracking, Mobile App, IOS, MVVM Code Architecture

Sadržaj

1. Uvod

Nakon položenog vozačkog ispita, većini osoba prvi automobil je onaj automobil na kojem se zaista uče voziti. Stereotipni opis takvih automobila je da su stari i zakinuti modernijom opremom i informacijama koje su danas dio svakog modernog automobila, kolokvijalno: bitno je da imaju volan, papučicu gasa i papučicu kočnice.

Informacijski sustav za praćenje automobila za vrijeme vožnje je sustav koji omogućava vozačima starih i opremom zakinutih automobila da na mobilnom uređaju dobiju pregled u korisne informacije iz vozila za vrijeme vožnje. Sustav funkcionira na način da se mobilna aplikacija spaja na OBD[1] modul koji je spojen na sustav automobila. Modul direktno komunicira sa senzorima i sustavom automobila te omogućava mobilnoj aplikaciji da direktno pristupi podacima iz automobila. Nakon obrade informacija prikupljenih sa senzora, mobilna aplikacija prikazuje i komunicira podatke vozaču na suptilan način, kako vozač ni u jednom trenutku ne bi bio ometen od kontrole automobilom u prometu. U samim počecima i osmišljavanju prvih koncepata mobilne aplikacije, posebna pažnja je posvećena kreiranju kvalitetnog korisničkog iskustva za vrijeme korištenja aplikacije i kvalitetnog korisničkog sučelja kako bi aplikacija bila adekvatna uvjetima u kojima se koristi.

Mobilna aplikacija uz to što komunicira s modulom, komunicira putem REST API[2]-a sa serverom. Tijekom vožnje mobilna aplikacija šalje podatke o vožnji na server te ih obrađuje i sprema u bazu podataka. Tim informacijama se naknadno može pristupiti putem mobilne aplikacije kao uvid u statistiku vožnje.

[1] http://www.obdclearinghouse.com/oemdb/, Center for Automotive Science and Technology at Weber State University (3.2.2020)
[2] https://www.restapitutorial.com/, Learn REST: A RESTful Tutorial (3.2.2020)

2. Postojeća rješenja

Na tržištu trenutno postoji više rješenja koja omogućavaju pregled informacija iz automobila putem mobilne aplikacije. Karakteristika svih rješenja je da zahtijevaju da se u sklopu sustava nalazi OBD modul koji je priključen u vozilo kako bi mobilna ili web aplikacija mogla pristupiti podacima sa senzora automobila. Na tržištu se izdvajaju rješenja poput TOMTOM Curfer[3] i Automatic[4] koji se danas može koristiti samo na američkom tržištu. Tvrtka TOM TOM je razvila u sklopu Curfer sustava svoj OBD modul i mobilne aplikacije za Android[5] i iOS[6]. Mobilne aplikacije prate putem senzora način vožnje vozača te putem aplikacije uz podatke o vožnji iste obrađuju i prikazuju u kontekstu bodovanja načina vožnje koji su sami razvili. Na taj način formiraju profil vozača koji omogućava dublje razumijevanje vozačevih navika i stila vožnje.

Uz navedene, postoji nekoliko aplikacija koje prate način vožnje putem mobilnih senzora. Dio takvih rješenja podržale su u svojim poslovnim modelima osiguravajuće kuće koja time potiču vozače da boljom i sigurnijom vožnjom sakupljaju bodove putem njihovih aplikacija. Sigurnija vožnja znači više bodova, a više bodova znači veći popust na kupovinu osiguranja.

Svim tim rješenjima zajedničko je da pružaju velik broj informacija o statusu vozila od kojih mnoge za vrijeme vožnje nisu potrebne, međutim ih se može kategorizirati i organizirati na način kojima vozaču (ili tvrtkama poput spomenutih osiguravajućih kuća) pružaju dodanu i konkretnu vrijednost o stilu vožnje.

Rješenje koje je predstavljeno u ovom radu razlikuje se od spomenutih rješenja jer korisniku prikazuje samo one podatke koji su mu potrebni, iste obrađuje i javlja korisniku: na primjer, kada je automobil dovoljno zagrijan za normalnu vožnju, prepoznaje kada se dogodio sudar, prati lokaciju automobila kada korisnik nije u vozilu te pruža jednostavan pristup glazbenom sustavu ili vremenskoj prognozi za vrijeme vožnje.

---

[3] https://www.tomtom.com/en_gb/sat-nav/curfer/products/tomtom-curfer/, TOMTOM Curfer (13.2.2020)
[4] https://www.tomtom.com/products/autostream/, TOMTOM Autostream (8.2.2020)
[5] https://www.android.com/, Android OS (7.2.2020)
[6] https://developer.apple.com/ios/, iOS OS (9.2.2020)

3. Općenito o OBD-II Modul-u

OBD je skraćenica od engleskih riječi „ON BOARD DIAGNOSTIC“ te se taj pojam u automobilskom svijetu odnosi na sustave auto dijagnostike u automobilima. OBD-II je naziv standarda (protokola) za sustave dijagnostike automobila koji se u Europi uvodi od 2000. godine pod nazivom EOBD[7].

Prvi oblici OBD protokola u automobilima se pojavljuju 1980. godine, kada su se počeli po prvi puta na tržištu pojavljivati automobili s ugrađenim računalima koja su realnom vremenu pratila i podešavala ubrizgavanje goriva u glave motora. Dvije godine kasnije General Motors predstavlja Assembly Line Communications Link (ALCL) koji je kasnije preimenovan u naziv Assembly Line Diagnostics Link[8] (ALDL), te taj standard predstavlja prethodnika prvom OBD standardu.

Prvi OBD standard pod nazivom OBD-I pojavljuje se 1987. godine kada se država Kalifornija počinje zalagati da se svi proizvođači automobila pridržavaju nekih osnovnih standarda kako bi se olakšalo komuniciranje vozila s vanjskim svijetom[9]. Kada je OBD-I standard zasnovan nije postojalo pravilo u autoindustriji da se svi proizvođači automobila moraju pridržavati istog standarda, nego je svako koristio svoj. Čak ni konektori za spajanje vozila na računalo nisu bili standardizirani.

Godinu dana nakon zasnivanja OBD-I standarda, 1988. godine, udruga automobilskih inženjera pod nazivom „Society of Automotive Engineers“[10] (SAE) predlaže uvođenje jednog standardnog konektora za spajanje vozila na računalo i jedan protokol. Nakon što je prijedlog usvojen 1994. godine, uvodi se OBD-II standard u Sjedinjenim Američkim Državama, u državi Kaliforniji. Te godine u Kaliforniji sva vozila koja su bila prodana morala su ispuniti OBD-II standard. Na taj način je Kalifornija uspjela ograničiti emisiju štetnih plinova koji automobili ispuštaju u atmosferu. Dvije godine kasnije, 1996. godine, sva vozila prodana u Sjedinjenim Američkim Državama morala su ispunjavati zahtjeve propisane OBD-II standardom.

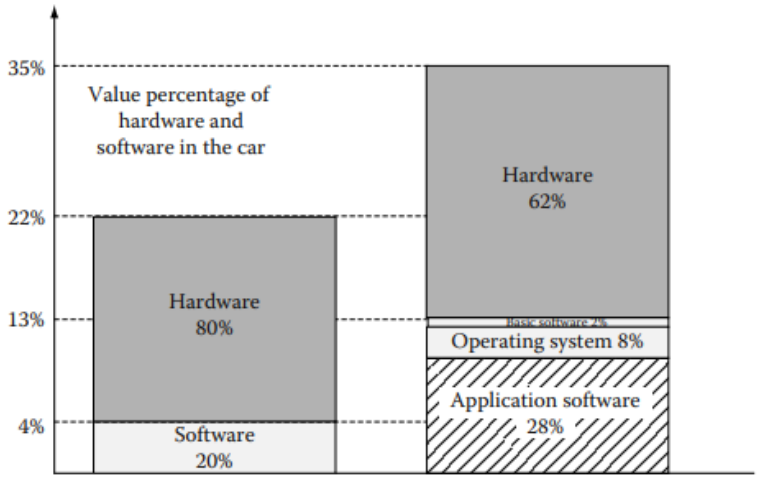
---

[7] https://en.wikipedia.org/wiki/On-board_diagnostics , On-board diagnostics (3.2.2020)
[8] https://en.wikipedia.org/wiki/ALDL , ALDL standard (5.2.2020)
[9] https://ww2.arb.ca.gov/homepage, The California Air Resources Bord (4.2.2020)
[10] https://www.sae.org/ , SAE udruga (5.2.2020)

Europska Unija 2001. godine stvara EOBD standard na temelju američko/kanadske inačice OBD-II standarda u kojem nalaže da sva vozila s benzinskim motorima prodana u Europi moraju ispuniti propise EOBD standarda. Godinu dana kasnije proširuje obvezu i na sva ostala vozila uključujući i ona vozila s dizelskim motorima.

Mnogi djelatnici u autoindustriji tvrde da je uvođenjem OBD-II standarda napokon dovedeno malo reda u do tada pobrkani svijet protokola za dijagnostiku automobila.

Ovim standardom proizvođači automobila obvezuju se omogućiti pristupanje računalu motora koje se koristi za dijagnostiku vozila na način da moraju koristiti standardne procedure i protokole za komunikaciju s računalom i pristup očitanjima, greškama, *freeze frame* podacima i ostalim važnim podacima za kontrolu rada motora.



Slika 3.1: Rast važnosti softvera u automobilu[11]

U moderno doba, automobilska industrija nalazi se pred novim izazovom. Devedeset posto svih inovacija odnosi se na elektroniku, a 80% odnosi se na softver. To znači veliku promjenu u razvoju elektronike. Sve više i više visoko povezanih funkcija mora se razviti do serijske proizvodnje, dok istovremeno ciklusi razvoja postaju sve kraći i kraći. Važnost softvera u automobilskoj industriji vrlo je impresivno prikazana u studiji Mercer Management Consulting i Hypovereinsbank. Prema ovoj studiji, u 2010. godini 38% proizvodnih troškova otišlo je u softver što je prikazano drugim dijagramom slike 3.1. Radi usporedbe, prvi dijagram prikazuje da je u 2000. godini 20% proizvodnih troškova otišlo u razvoj softvera. Za 2020. godinu istraživanja predviđaju da će proizvodni troškovi softvera

[11] Mercer Management Consulting i Hypovereinsbank, studija Automobiltechnologie 2010. München

porasti na čak 55%. S obzirom na ta otkrića, proces razvoja koji uključuje metode razvoja softvera za automobilsku domenu mora se poboljšati.

4. Režimi rada OBD-II Modul-a

Zahtjevi na dizajn i kvalitetu softvera stalno rastu. Ciljevi su uvijek smanjiti vrijeme razvoja softvera i povećati kvalitetu softvera. Još jedan obećavajući, ali ipak izazovan način da se ti ciljevi postignu je ponovna upotreba softvera. Glavni preduvjet za ponovnu upotrebu softvera u automobilskoj domeni je odvajanje hardvera elektroničke upravljačke jedinice (ECU) od ugrađenog softvera koji se na njemu nalazi. Donedavno je percepcija ECU-a proizvođača automobila bila jednaka. Specificirali su i naručili crne kutije od svojih dobavljača. Nakon isporuke uzoraka, testirali su ih i kao crne kutije. Za proizvođača automobila ovaj postupak ima nedostatak što softver mora biti ponovo razvijen za svaki novi projekt ako se promijeni dobavljač. To ne samo da uzrokuje troškove, već i povećava vrijeme razvoja.

Na tržištu postoje uređaji s univerzalnim OBD-II protokolom te se vrlo jednostavno mogu kupiti. U ovom radu se koristi OBD-II uređaj čiji je proizvođač Njemačka tvrtka Auto Pi[12], te se u njemu nalazi Raspberry Pi[13] računalo koje putem LEM327 sučelja komunicira sa upravljačkom jedinicom motora (ECU).

LEM327 je sučelje koje omogućava komunikaciju drugih uređaja sa ECU. S tehničke strane, LEM327 je PIC18F2480 mikrokontroler koji na sebi sadrži program koji naredbu koju je primio pretvara u naredbu propisanu ODB-II protokolom, koju zatim šalje na upravljačku jedinicu motor. Takvi uređaji idealni su za sustave kojima je potreban pristup osnovnim funkcijama čitanja najvažnijih parametara vozila, a nije im potrebno reprogramiranje i adaptacija upravljačke jedinice motora.

Nakon što je OBD-II uređaj spojen na vozilo, uređaj pokreće analizu dostupnih adresa u vozilu. Pošto OBD-II standard pokriva samo pristup podacima iz motora, uređaju su samo zanimljive adrese za komunikaciju od $00 do $17 koje su rezervirane za upravljačku jedinicu motora. Kada je komunikacija između uređaja i ECU-a uspostavljena, uređaj započinje dijagnostičku komunikaciju u jednom od 10 načina rada. Ne podržavaju sva ECU računala nužno sve načine rada, nego to većinom ovisi o starosti vozila. Što je vozilo novije, to je vjerojatnost veća da vozilo podržava više načina rada.

[12] https://www.autopi.io/ , Tvrtka AutoPi (12.2.2020)
[13] https://www.raspberrypi.org/ , Raspberry Pi računalo (6.2.2020)

Svaki način rada je specifičan po svojem zadatku i tipu dijagnostike koji izvršava. U ovom sustavu koristi se prvi način dijagnostike u kojem je bitan pristup trenutnim ulaznim i izlaznim analognim i digitalnim vrijednostima motora, kao i vrijednosti elemenata kojima upravlja ECU. U nastavku su prikazani svi trenutni režimi rada ECU računala.

Tablica 4.1: Tablični prikaz režima rada

Režim rada	**Opis**
Režim rada 1	Analiza parametara motora
Režim rada 2	Detekcija pogrešaka u radu motora
Režim rada 3	Detekcija pogrešaka u upravljačkoj jedinici motora
Režim rada 4	Brisanje podataka o pogrešci
Režim rada 5	Dijagnostika rada lambda sonde
Režim rada 6	Dijagnostika sporednih komponenti
Režim rada 7	Analiza višestrukih pogrešaka
Režim rada 8	Pokretanje testova na dijelovima motora
Režim rada 9	Identifikacija vozila
Režim rada 10	Analiza trajnih pogrešaka u radu

4.1. Režim rada 1: analiza parametara motora

Način rada u kojem dijagnostički uređaj iščitava PID-ove koji opisuju trenutne vrijednosti rada motora. U ovom modu moguće je doći do informacija poput:

- Ulaznih i izlaznih analognih signala poput temperature motora, broja okretaja motora, opterećenju motora.
- Ulaznih i izlaznih digitalnih signala poput je li pedala za kuplung pritisnuta, ili je li pedala za gas pritisnuta ili ne.
- Informacije o elementima kojima upravlja računa motora, poput vremena otvorenosti ubrizgivača za ubrizgivanje goriva u glave motora.
- Informacije o trenutno dostupnosti sustava u vozilu poput postojanje sustava rashlađivanja, postojanje ABS sustava.

4.2. Režim rada 2: detekcija pogrešaka u radu motora

Način rada u kojem dijagnostički uređaj čita vrijednosti *freeze frame* podataka. To su vrijednosti osnovnih podataka motora koji su zabilježeni u trenutku kada je došlo do greške. Vrijednosti se dohvaćaju putem PID-ova, isto kao i u prvom načinu. Pamte se vrijednosti poput temperature motora, postotka opterećenja motora i broja okretaja motora u trenutku kada se dogodila greška.

4.3. Režim rada 3: detekcija pogrešaka u upravljačkoj jedinici motora

Ovaj način dijagnosticira greške koje je upravljačka jedinica motora zabilježila bar tri puta.

4.4. Režim rada 4: brisanje podataka o pogreškama motora

Ovaj način dijagnostike briše greške u upravljačkoj jedinici motora i poništava sve informacije zabilježene uz tu grešku.

4.5. Režim rada 5: dijagnostika rada lambda sonde

U ovom načinu rada uređaj provodi dijagnostiku i provjeru rada lambda sonde. Lambda sonda je senzor za praćenje stanja ispušnih plinova vozila. Provjera se obavlja prije pokretanja vozila, za vrijeme grijanja motora i za vrijeme vožnje.

4.6. Režim rada 6: dijagnostika sporednih komponenti

Ovaj način dijagnosticira i provjerava rad komponenti koje sustav ne nadzire kontinuirano. Normom nije propisano što rezultati ove dijagnostike moraju prikazati te rezultati ovise o proizvođaču automobila.

4.7. Režim rada 7: analiza višestrukih pogrešaka

Vezano na način 3 rada, ovaj način dijagnosticira greške koju su se dogodile manje od 3 puta, ali su se dogodile i mogu utjecati na rad motora.

4.8. Režim rada 8: pokretanje testova na dijelovima motora

Ovaj način rada predviđen je za pokretanje testova na pojedinim dijelovima motora, kao što je to npr. EGR filter. EGR filter služi za dohvat zraka motoru te se u ovom modu može zasebno testirati kako određeni element motora radi i zaključiti rade li svi elementi pojedino.

4.9. Režim rada 9: identifikacija vozila

U ovom načinu dijagnosticiranja prikazuju se osnovne identifikacijske oznake vozila i motora poput:

- VIN – Vehicle Identification Number
- CIN – Calibration Identification Number
- CVN – Calibration Verification Number

4.10. Režim rada 10: analiza trajnih pogrešaka u radu

Ovo je najnoviji način dijagnosticiranja. U ovom načinu rada dohvaćaju se greške, ali ne greške poput onih u načinu 3 ili načinu 7, nego „trajne“ greške. To su greške koje samo vozilo može izbrisati, iako su one obrisane i poništene u modu 4 biti će vidljive sve dok nakon popravka vozila sustav sam ne napravi testiranje na temelju kojeg radi promjenu, poništi i obriše greške.

4.11. Parametarski identifikatori

PID je skraćenica od pojma parametarski identifikator (engl. *Parameter Identifier*). PID parametar svojom zadanom vrijednošću označava točno o kojem senzoru unutar motora se radi.

Svaki senzor koji se nalazi u motoru vozila okarakteriziran je s PID vrijednosti. Na taj način pomoću PID parametara moguće je pristupati i iščitavati vrijednosti senzora. Recimo, senzor brzine motora ima dodijeljen PID 13.

SAE je nakon uvođenja OBD-II standarda do 1996. godine sa standardom pod nazivom J1979 definirala većinu vrijednosti PID parametara za OBD-II standard. Od 1996. godine do danas je definirano 193 PID parametara, recimo, 2007. godine je bilo definirano 137 PID parametara. Razlika u broju PID parametara danas i 2007. godine govori o tome kako automobilska industrija tehnološki napreduje iz desetljeća u desetljeće.

5. Mobilna aplikacija za praćenje statusa automobila za vrijeme vožnje

5.1. Korisničko iskustvo

Na samom početku izrade mobilne aplikacije posebna pažnja je bila usmjerena prema osmišljavanju kvalitetnog korisničkog iskustva korištenja aplikacije. Činjenica je da korisnik ovakav tip aplikacije većinu vremena koristi unutar vozila i za vrijeme vožnje, vrijeme je u kojem se zahtjeva potpuna koncentriranost korisnika na trenutni zadatak, vožnju. Tu se postavilo pitanje kako korisniku na suptilan način, jednostavno i čitljivo prikazati informacije poput:

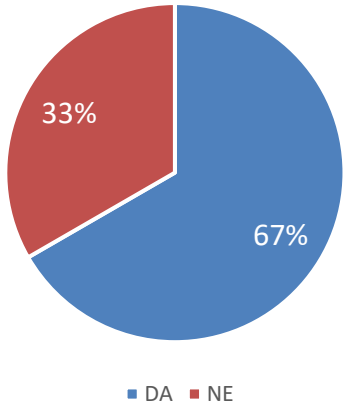
- Informacija direktno pročitanih sa senzora automobila
- Vremenske prognoze
- Glazbenog sustava
- Popisa svih putovanja
- Lokacije vozila
- Statistike vožnje

Prvi korak bio je odabrati informacije koje su korisniku bitne i korisne za vrijeme vožnje i tada osmisliti koncept kako te informacije na jednostavan i pristupačan način prikazati korisniku na mobilnoj aplikaciji, a da uz to jednostavno može pristupiti ostalim uslugama i funkcionalnostima aplikacije. Kao rezultat istraživanja na uzorku od petnaestero ljudi dobivene su informacije koji su podaci najbitniji vozaču za vrijeme vožnje.

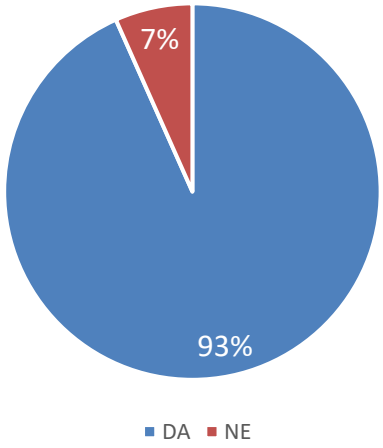
Istraživanje se provodilo u obliku ankete, a ispitanici istraživanja bili su vozači i vozačice, u rasponu od 19 do 33 godina, vlasnici srednjih kategorija automobila između kojih je najstariji automobil bio star 20 godina (2000. godina proizvodnje), a najmlađi 9 godina (2011. godina proizvodnje). Kroz anketu su dobili ponuđena 3 pitanja koja će u nastavku biti analizirana.

Na prva dva ponuđena pitanja kao odgovori ispitanicima su bili ponuđeni odgovori DA i NE. Kao odgovore na zadnje pitanje imali su pravo odabrati četiri od 15 ponuđenih

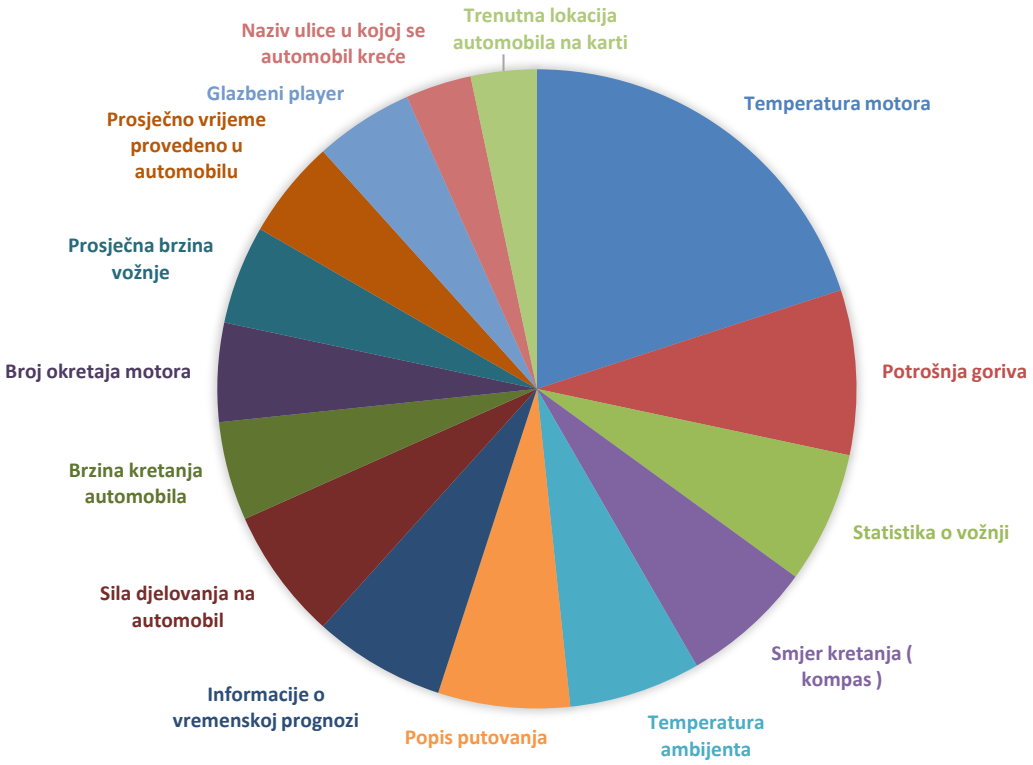
odgovora. Zadnje pitanje je ključno pitanje vezano konkretno za organizaciju informacija i funkcionalnosti na mobilnoj aplikaciji.



Slika 5.11: „Jeste li znali da temperatura motora utječe na kvalitetu vožnje i očuvanost vozila?“



Slika 5.2:„Bi li voljeli znati kada je motor automobila dovoljno zagrijan za normalnu vožnju?“



Slika 5.3: „Koje od navedenih podataka i funkcionalnosti smatrate najrelevantnijima za vrijeme vožnje u automobilu?“

Iz rezultata se može uočiti kako su prva dva pitanja utjecala na odabir relevantnih informacija u zadnjem pitanju. Informacija o temperaturi goriva, na prvi pogled nije toliko relevantna informacija, ali nakon osvještavanja ispitanika kroz prva dva pitanja da je stupanj zagrijanosti motora bitan faktor u očuvanju vozila i kvalitete vožnje, 20% odabira zadnjeg pitanja je bila temperatura motora. Drugim riječima, 12 od 15 ispitanika odgovorilo je da je temperatura relevantna informacija za vrijeme vožnje.

Sljedeći korak nakon istraživanja bio je selekcija informacija i funkcionalnosti. Pri selekciji, informacije i funkcionalnosti su se analizirale silaznim redoslijedom od onih koje su bile najviše puta odabrane pa do prema onima koje su dobile najmanji broj odabira. Informacije i funkcionalnosti grupirane su u tri kategorije u kontekstu važnosti i relevantnosti za vrijeme vožnje:

- Jako bitne – pregledne, nalaze se na glavnom zaslonu.
- Srednje bitne – ne nalaze se na glavnom zaslonu ali moraju biti pristupačne.
- Nebitne – ne nalaze se na glavnom zaslonu aplikacije.

Svrha navedenih kategorija je olakšati osmišljavanje korisničkog sučelja na temelju relevantnosti informacija. Jako bitne funkcionalnosti i informacije se moraju nalaziti na glavnom zaslonu, koji je centralni dio aplikacije na kojem se nalaze sve bitne i relevantne informacije i funkcionalnosti. Srednje bitna kategorija opisuje one informacije i funkcionalnosti koje su bitne korisniku na za vrijeme vožnje ali nije nužno da su konstantno vidljive za vrijeme vožnje. Nebitne funkcionalnosti su one od kojih korisnik ima više koristi dok ne upravlja vozilom. U nastavku se nalazi analiza selektiranja informacija po kategorijama.

Iz istraživanja se može zaključiti kako je primarno bilo ispitanike osvijestiti da je temperatura motora jako bitan podatak u kontekstu upravljanja vozilom. Rezultati iz zadnjeg pitanja ankete pokazuju kako je to bilo uspješno, 12 od 15 ispitanika na zadnjem pitanju odabralo je da je temperatura motora bitan podatak za vrijeme vožnje. Ta funkcionalnost svrstana je u kategoriju jako bitnih informacija.

Sljedeći odgovor po količini odabira je bila funkcionalnost koja prikazuje potrošnju goriva, a kao treći odgovor od strane korisnika je bila funkcionalnost statistika o vožnji. Potrošnja goriva svrstana je u kategoriju jako bitnih funkcionalnosti.

U anketi informacija statistika o vožnji kao ponuđeni odgovor uz sebe je još imala naglašeno da tu funkcionalnost opisuju dvije informacije koje su također bile zastupljene kao zasebni odgovori, a to su prosječna brzina vožnje i prosječno vrijeme provedeno u automobilu. Uz te dvije informacije statistiku vožnje čini i informacija o broju proputovanih kilometara te se ona korisniku prikazuje kao omjer proputovanih kilometara vozila u odnosu na količinu kilometara opsega planeta Zemlje. Zaključak je da statistika o vožnji nije vrsta funkcionalnosti koja treba biti svrstana u kategoriju jako bitna jer se sastoji od 3 informacije za koje je potrebna veća količina prostora na ekranu da bi se prikazala, a one se ne mijenjaju ovisno o trenutnoj vožnji vozača i ne utječu direktno na automobil. Statistika o vožnji svrstana je u kategoriju nebitnih informacija.

Sljedeće informacije u silaznom nizu po količini odabira od strane ispitanika koje su svrstane u kategoriju jako bitnih informacija su informacija o smjeru kretanja vozila i temperatura ambijenta. Nakon toga sljedeći odgovor po zastupljenom broju odabira je funkcionalnost popis putovanja koja korisniku omogućava uvid u sva prijašnja putovanja automobilom. Ta funkcionalnost svrstana je u kategoriju nebitnih funkcionalnosti iz razloga što ona prikazuje veliku količinu informacija koje korisniku nisu bitne za vrijeme vožnje,

nego su bitnije kada korisnik ne upravlja vozilom. Funkcionalnost popis putovanja je svrstana u kategoriju nebitnih funkcionalnosti.

Sljedeći odabir od strane ispitanika je bio funkcionalnost informacije o vremenskoj prognozi. Ta funkcionalnost prikazuje informaciju o trenutnoj prognozi grada u kojem se trenutno nalazite, a isto tako prikazuje vremensku prognozu za naredna 4 dana. Odlučeno je da će ta informacija biti podijeljena u dvije kategorije - jako bitnih i srednje bitnih informacija i funkcionalnosti. U kategoriju jako bitnih svrstana je funkcionalnost o trenutnom vremenu u gradu kojem se korisnik nalazi iz razloga što je korisna vozaču u trenutku vožnje, dok prikaz vremenske prognoze za naredna 4 dana nije toliko bitna vozaču u trenutku vožnje, ali je korisna ako joj se može lako pristupiti za vrijeme vožnje.

Iznenađujuće visok rezultat dobila je informacija djelovanja sile na automobil. Ta informacija u kontekstu g-sile korisniku prikazuje kolika sila djeluje na automobil za vrijeme vožnje. Iz razloga što ju je poprilično lako prikazati uz minimalno oduzimanje prostora na ekranu, svrstana je u kategoriju jako bitnih. Iza sile na automobil po rezultatima slijede informacije o brzini kretanja vozila i broju okretaja motora. U tehnološki naprednom vremenu danas na određenim modelima vozila broj okretaja motora je funkcionalnost koja se dodatno naplaćuje te je ta funkcionalnost svrstana u jako bitne informacije iz razloga što vjerojatno postoje vozila na cesti koja nemaju prikaz te informacije na kontrolnoj ploči. Brzina automobila je informacija koja se usko veže uz funkcionalnost broj okretaja motora pa je ta funkcionalnost isto tako svrstana u kategoriju jako bitne.

Pristup glazbenom sustavu je funkcionalnost koja je korisniku jako korisna za vrijeme vožnje, ali ne treba biti na glavnom zaslonu, nego mora biti lako pristupačna pa je ona svrstana u kategoriju srednje bitnih.

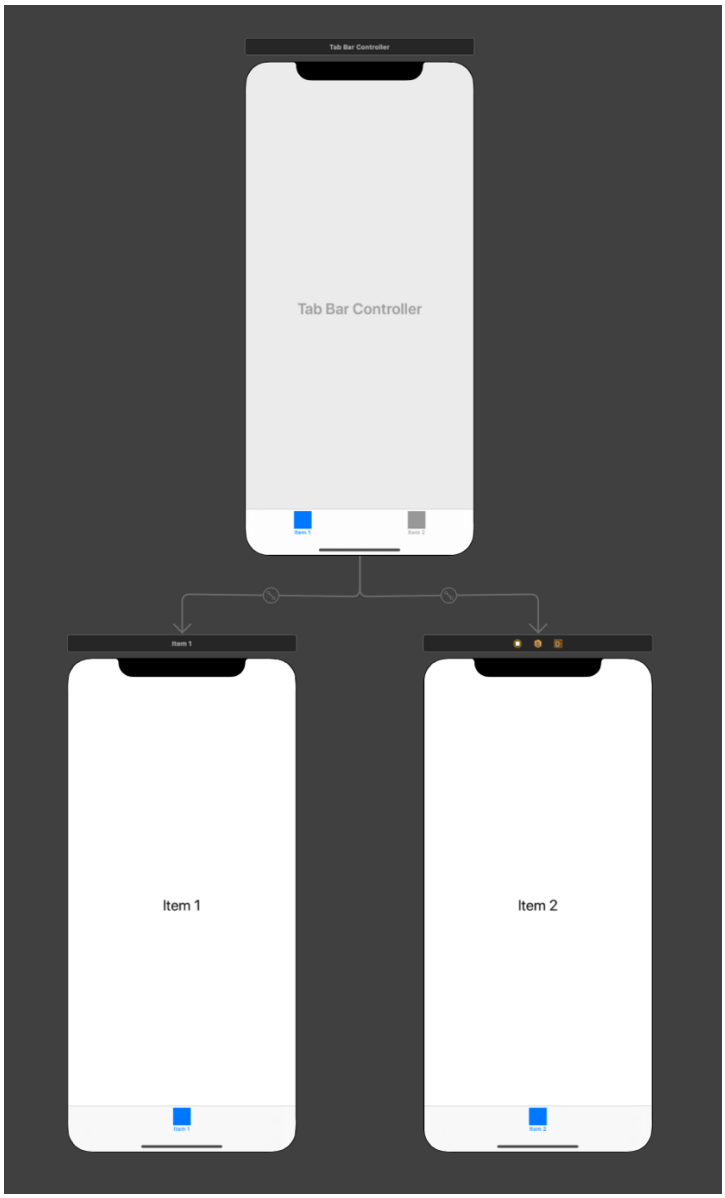
Na kraju su ostali rezultati odgovora za naziv ulice u kojoj se automobil kreće i trenutne lokacije automobila na karti. Naziv ulice u kojoj se automobil kreće svrstan je u jako bitne jer je zaključeno da bi ju bilo praktično prikazati uz funkcionalnost prikaza smjera kretanja vozila. Trenutna lokaciju automobila na karti svrstana je u kategoriju nebitnih informacija iz razloga što je za tu funkcionalnost potrebno korištenje karte, a kartu je po općim pravilima UX-a najbolje prikazivati na zasebnom zaslonu jer je tako najpreglednija i najjednostavnija za korištenje.

Rezultati selekcije funkcionalnosti:

- Jako bitne
  - Temperatura motora
  - Potrošnja goriva
  - Smjer kretanja vozila
  - Temperatura ambijenta vozila
  - Informacije o vremenskoj prognozi – trenutna
  - Sila djelovanja na vozilo
  - Brzina vožnje
  - Broj okretaja motora
  - Naziv ulice kretanja automobila
- Srednje bitne
  - Glazbeni sustav
  - Informacije o vremenskoj prognozi – za sljedeća 4 dana
- Nebitne
  - Statistika o vožnji
  - Trenutna lokacija vozila
  - Popis putovanja

5.2. Korisničko sučelje

Nakon što su informacije razvrstane po kategorijama relevantnosti za vrijeme vožnje, kreiralo se korisničko sučelje. Kako je prije spomenuto, sve funkcionalnosti i informacije koje su svrstane u kategoriju jako bitnih, moraju se prikazivati na glavnom zaslonu aplikacije a one koje su svrstane u kategoriju srednje bitnih moraju biti lako pristupačne za vrijeme vožnje. Nebitne funkcionalnosti i informacije ne trebaju biti vidljive niti lako pristupačne za vrijeme vožnje. S time je odlučeno da će aplikacija biti bazirana na konceptu mobilne aplikacije s karticama. To izgleda na sljedeći način.



Slika 5.4: Prikaz rasporeda zaslona kartične kontrole

Na dnu aplikacije se nalazi kontrola s karticama, svaka kartica u toj kontroli predstavlja zasebne zaslone u aplikaciji te se pritiskom na pojedinu karticu otvara zaslon. U gore navedenoj slici zasloni su označeni oznakama *Item* 1 i *Item* 2. S uzorom na to kreirana je kontrolna kartica u aplikaciji na kojoj svaku karticu predstavlja ikona s obzirom na to što određeni ekran predstavlja i koje funkcionalnosti prikazuje.

Slika 5.5: Tablična kontrola mobilne aplikacije

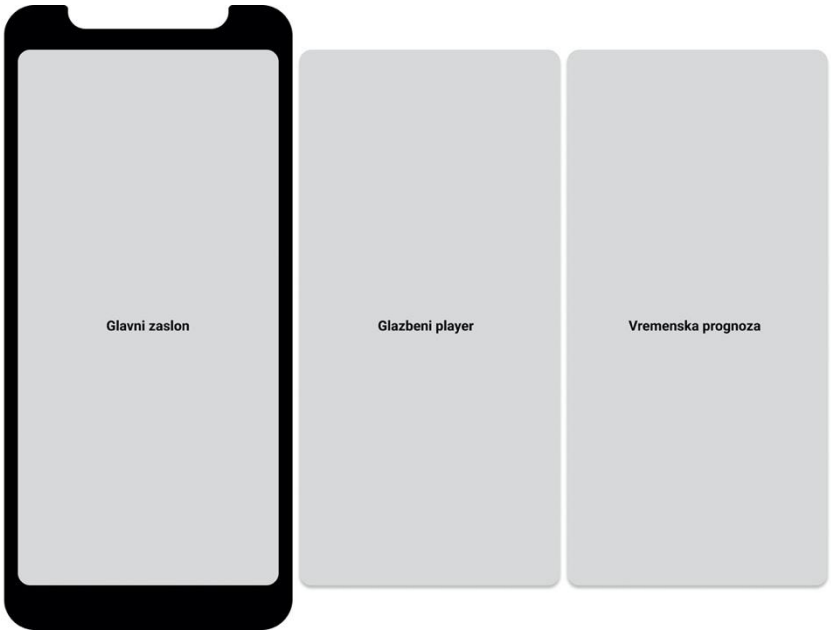
Svaku karticu predstavlja zasebna ikona, a boja ikone označava koju karticu je korisnik trenutno odabrao. Ikona bijele boje označava odabranu karticu, a ikone sivih boja označavaju neodabrane kartice. Na taj način lako je organizirati funkcionalnosti po kategorijama u aplikaciji. Kartice su imenovane na sljedeći način.

Nazivi kartica s lijeva na desno:

- Dashboard – predstavlja glavni zaslon aplikacije.
- Trips – predstavlja zaslon prikaza putovanja.
- Map – predstavlja zaslon s kartom.
- Profile – predstavlja zaslon statistika vožnji i profila korisnika.

Na početnu karticu u kartičnoj kontroli pozicionirane su jako bitne funkcionalnosti pošto one predstavljaju primarne informacije u aplikaciji, a na ostale kartice pozicionirane su funkcionalnosti koje su kategorizirane kao nebitne. Na taj način dobiveni su poseban zaslona za prikaz popisa putovanja vozila i poseban zaslon za kartu na kojem će se prikazivati trenutna lokacija vozila. Zadnja kartica u kontroli predstavlja zaslon na kojem će se prikazivati statistike o vožnji, informacije o vozilu, trenutnoj verziji aplikacije i mogućnost odjave iz aplikacije.

Na početnom zaslonu uz jako bitne funkcionalnosti prikazivat će se i srednje bitne informacije. Početni zaslon sastoji od 3 zaslona unutar jednog. Na početnom zaslonu prikazuju se jako bitne funkcionalnosti, a na ostalim zaslonima se nalaze funkcionalnosti glazbenog *playera* i vremenska prognoza za sljedeća četiri dana. Na taj način lako se može pristupiti srednje bitnim funkcionalnostima na način da jednim potezom prsta korisnik izmjenjuje zaslone. Zasloni su organizirani sljedećim redoslijedom.



Slika 5.6: Nacrt rasporeda zaslona kontrolne ploče

Zasloni se nalaze jedan do drugog u kolekciji zaslona koja se nalazi unutar zaslona koji je predstavljen od strane kartice iz kontrola kartice. Orijentirani su horizontalno što korisniku omogućava da horizontalnim potezima prsta po ekranu izmjenjuje zaslone.

Za glavne boje u aplikaciji odabrane su zeleno-siva, crna i bijela. Posebna nijansa zeleno-sive boje je korisniku ugodna, ne oduzima mu previše pozornosti i dovoljno dobro na sebi daje kontrast na ostale elemente koji se prikazuju crnom i bijelom bojom.

Slika 5.7: Pozadinska boja mobilne aplikacije

Crna boja s 50% prozirnosti na pozadinsku zelenu boju naglašava elemente u kojima se nalaze podaci te su na taj način informacije preglednije i lako uočljivije korisniku na ekranu mobitela. U nastavku se može vidjeti kako izgleda glavni zaslon aplikacije na kojem se prikazuju najbitnije funkcionalnosti i informacije u aplikaciji.

Slika 5.8: Kontrolna ploča aplikacije

Svaka informacija i funkcionalnost kao pozadinu ima crnu boju s prozirnosti od 50% koja umjereno balansira prijelaz iz pozadinske zeleno-sive boje u crnu boju te na taj način svaka informacija i funkcionalnost dobiva svoje pregledno mjesto na ekranu.

Na sličan način se prikazuju podaci o vremenskoj prognozi gdje jedan podatak na zaslonu čini informacija o temperaturi u određenom dijelu dana.

Slika 5.9: Vremenska prognoza

Podaci o vremenskoj prognozi sortirani su po danima te se prikazuju kronološki za svaka tri sata u danu. Informacija o temperaturi je naglašena svijetlo plavom bojom kako bi korisniku bila lakše čitljiva te korisnik bez puno distrakcije može pregledati i zaključiti kako će se temperature kretati kroz trenutan dan i sljedeća 4 dana.

Prema navedenim principima iz prijašnja dva primjera zaslona prikazuju se gotovo sve funkcionalnosti i informacije mobilne aplikacije.

6. Razvoj mobilne aplikacije

U sklopu ovog rada razvijena je mobilna aplikacija za iPhone mobilne uređaje koji koriste operacijski sustav pod nazivom iOS. To je operacijski sustav koji se pokreće na iPhone i iPad uređajima koje proizvodi tvrtka Apple[14]. Uz iPhone i iPad uređaje Apple uz mobilne i tablet uređaje također proizvodi prijenosna računala, stolna računala i pametne satove. Prijenosna računala i stolna računala pokreću operacijske sustave pod nazivom macOS, a pametni satovi rade na operacijskom sustavu pod nazivom watchOS.

Za razvijanje aplikacija za iOS, macOS ili watchOS operacijske sustave potrebno je stolno ili prijenosno računalo s macOS operacijskim sustavom. Aplikacije se razvijaju s Objective-C[15] ili SWIFT[16] programskim jezikom u integriranom razvojnom okruženju pod nazivom XCode[17].

XCode je moguće koristiti samo na macOS operacijskom sustavu te u trenutku pisanja ovog rada ne postoji službeno rješenje od strane Apple koje omogućava korištenje XCode razvojnog okruženja na drugim operacijskim sustavima. Na internetu postoji mnogo prijedloga kako pokrenuti macOS operacijski sustav na računalima koja nisu proizvedena od strane tvrtke Apple, no službene podrške nema.

6.1. Programski jezik

Mobilna aplikacija u ovom radu razvijena je u programskom jeziku Swift. Swift je programski jezik koji je predstavljen 2014. godine na WWDC-u od strane Apple. Nasljednik je dotadašnjeg službenog programskog jezika za razvoj programa i aplikacija za Apple uređaje Objective-C.

Objective-C je objektno orijentirani jezik koji se uz razvoj aplikacija za Apple uređaje u nešto manjem omjeru koristio i za razvoj rješenja na Linux operacijskom sustavu. Nastao je miješanjem programskog jezika Smalltalk[18] i programskog jezika C[19] 1980.

[14] https://en.wikipedia.org/wiki/Apple_Inc., Tvrtka Apple Inc. (13.2.2020)
[15] https://en.wikipedia.org/wiki/Objective-C, Objective-C programski jezik (13.2.2020)
[16] https://developer.apple.com/swift/, Swift programski jezik (13.2.2020)
[17] https://developer.apple.com/xcode/, XCode programsko okruženje (13.2.2020)
[18] https://en.wikipedia.org/wiki/Smalltalk , Smalltalk programski jezik (13.2.2020)
[19] https://en.wikipedia.org/wiki/C_(programming_language), C programski jezik (13.2.2020)

godine. Razvili su ga Brad Cok i Tom Love u tvrtki Stepstone. Do 2014. godine bio je službeni razvojni programski jezik aplikacija za Apple uređaja sve do pojavljivanja Swift programskog jezika. Danas je Objective-C i dalje zastupljen te u manjem omjeru postoje programeri koji i dalje razvijaju aplikacije u tom programskom jeziku.

Swift je objektno orijentirani jezik nastao iz razloga što je Apple želio poboljšati određene dijelove u Objective-C programskog jeziku. Veliki problem s Objective-C je da je sintaksa programskog koda jako nepregledna te je uočavanje grešaka za vrijeme pisanja koda poprilično izazovno. Swift je zadržao neke glavne značajke svog prethodnika, ali s druge strane donio je puno pregledniju i jednostavniju sintaksu, uočavanje grešaka za vrijeme pisanja programskog koda te puno veću brzinu performansi.

Tablica 6.1: Usporedba Objective C i Swift programskog jezika

Objective-C	Swift
NSString *str = @”hello,”; str = [str stringByAppendingString:@” world”];	var str = „hello,“ str += „ world“

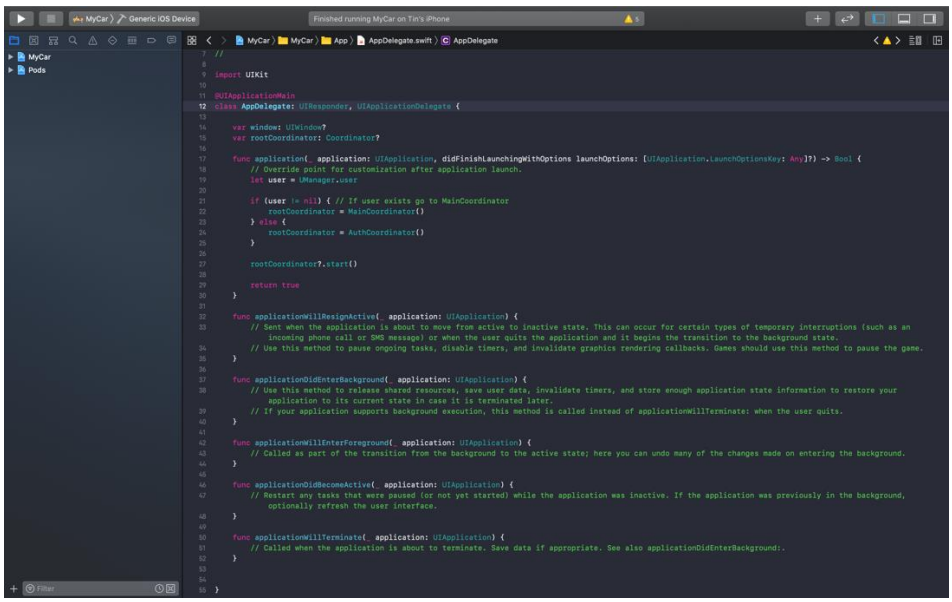
Swift je potpuno kompatibilan s Objective-C te je moguće koristiti oba programska jezika paralelno za vrijeme razvijanja aplikacija.

Mobilna aplikacija u ovom radu razvijena je u najnovijem Swift 5.1 programskom jeziku i XCode 11.3 programskom okruženju.

6.2. XCode razvojno okruženje

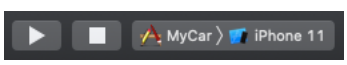
XCode je integrirano razvojno okruženje (IDE) za razvoj aplikacija za Apple uređaje. Kao što je spomenuto u prijašnjem poglavlju, podržava razvoj aplikacija u Objective-C i Swift programskom jeziku te radi s razvojnim okvirima koji se zovu Cocoa i Cocoa Touch. Cocoa je razvojni okvir za izradu korisničkih sučelja za aplikacije i programe na macOS operacijskom sustavu. Cocoa Touch je razvojni okvir za razvoj aplikacija na iOS operacijskom sustavu. Ta dva razvojna okvira slijede MVC model razvoja softvera.

XCode se sastoji od editora izvornog koda, kompajlera, emulatora, razvojnih okvira i još puno elemenata koji olakšavaju razvoj aplikacija i programa. Za razvoj kompletnih aplikacija potreban je samo XCode te ni jedan drugi softver. Postoje mogućnosti razvoja i u nekim drugim okruženjima, ali to se ne preporuča jer Apple ima striktna pravila oko razvoja aplikacija kojih se striktno treba držati kako bi bilo moguće aplikaciju objaviti na App Store-u[20]. Ako jedno od pravila nije zadovoljeno, XCode će upozoriti programere da isprave navedene pogreške kako bi njihova aplikacija mogla biti uspješno objavljena na App Store-u.



Slika 6.1: XCode razvojno okruženje

Za testiranje aplikacije XCode u sebi sadrži iPhone, iPad i iWatch emulatore svih aktualnih modela uređaja koji se mogu pronaći na tržištu. To uvelike pojednostavljuje testiranje aplikacije ukoliko programer ne posjeduje jedan od uređaja. Putem emulatora pomoću dva klika može se pokrenuti testiranje na željenom uređaju. U gornjem desnom kutu IDE-a nalaze se kontrole za odabir uređaja simulatora, pokretanje simulatora i zaustavljanje simulatora.



Slika 6.2: Kontrola za odabir uređaja simulatora, pokretanje i zaustavljanje simulatora

[20] https://www.apple.com/ios/app-store/, App Store – online trgovina Apple aplikacija (18.2.2020)

Pritiskom na naziv uređaja otvara se padajući izbornik s popisom svih uređaja koji se nalaze u IDE-u i onih fizičkih uređaja koji su kabelom spojeni na računalo.

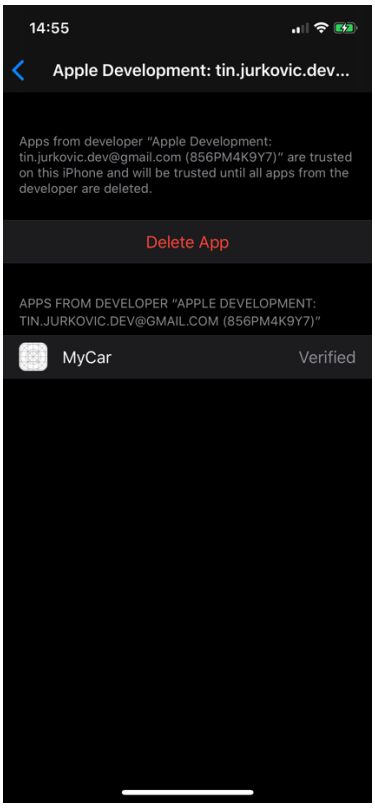
Slika 6.3: Odabir uređaja simulatora

Uređaji su podijeljeni u sekcije *iOS Simulators*, *Build Only Device* i *Device.* Odabirom jednog od uređaja iz sekcije *iOS Simulators* ili *Device* padajući izbornik se zatvara te klikom na gumb za pokretanje simulatora aplikacija se *builda* i pokreće na odabranom uređaju.

Slika 6.4: Prikaz XCode razvojnog okruženja i simulatora

U sekciji *Device* nalaze se fizički uređaji koji su kablom spojeni na računalo. Da bi pokretala aplikacije na fizičkim uređajima, osoba mora biti registrirana kao Apple programer. Na Apple službenim stranicama to je vrlo jednostavan proces. Nakon uspješne registracije svojim računom programer se prijavi unutar IDE-a te je sve spremno za pokretanje aplikacija na fizičkim uređajima. Unutar IDE-a moguće je biti prijavljen i s više

različitih korisničkih računa (npr. poslovni i privatni) te je vrlo lako mijenjati korisničke račune ovisno o projektima. Korisnički račun u sebi sadrži privatne ključeve koji služe za potpisivanje napisanog programskog koda iz sigurnosnih razloga i radi velikog broja verifikacija od strane Apple-a. Prilikom prvog pokretanja aplikacije na fizičkom uređaju dolazi do sigurnosne provjere od strane XCode-a gdje se od programera zahtijeva da u postavkama fizičkog uređaja potvrdi da taj uređaj vjeruje svim aplikacijama koje su razvijene na odabranom korisničkom računu. Na slici se može vidjeti zaslon na kojem je potvrđeno da fizički uređaj vjeruje svim aplikacijama razvijenim putem odabranog korisničkog računa.



Slika 6.5: Zaslon za potvrdu pokretanja aplikacija od navedenog korisnika

Na simulatorima je moguće testirati bazične funkcionalnosti operacijskog sustava i izgled korisničkog sučelja. Naprednija testiranja poput testiranja *push* notifikacija, senzora uređaja (senzora pokreta, senzora ubrzanja, senzora lokacije, itd.), testiranja različitih *framework-a* za korištenje poziva, slanje SMS-ova i ostalih koji su usko vezani za fizičke uređaje, nije moguće testirati na simulatorima, nego samo na fizičkim uređajima. Na internetu se često znaju pronaći pitanja što je bolje koristiti za testiranje između te dvije

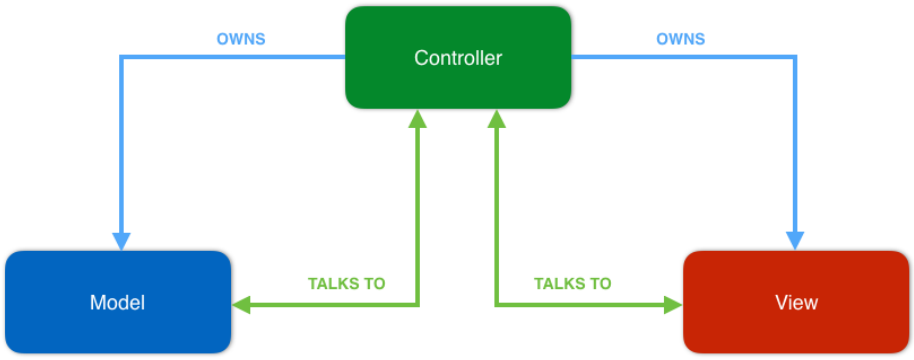
opcije. Točan odgovor na to ne postoji, nego samo ovisi o onome na čemu se trenutno radi u sklopu razvoja mobilne aplikacije. Za testiranje izgleda korisničkog sučelja i testiranje animacija dovoljan je simulator, a ako se žele testirati performanse uređaja dok se podaci dohvaćaju sa servera ili žele testirati geste korisnika, preporuča se testiranje na stvarnom uređaju zbog realnih uvjeta korištenja aplikacije.

6.3. Arhitektura

U ovom poglavlju bit će riječ o arhitekturi programskog koda iPhone aplikacija. Obradit će se kako se u mobilnoj aplikaciji ovog rada organizirao programski kod, koja arhitektura se koristila te kako se upravlja navigacijskom tokom (engl. *Navigation flow)* aplikacije.

6.3.1. Arhitektura programskog koda iPhone aplikacija

Apple je kao načelni model arhitekture programskog koda (engl. *Design Pattern)* postavio arhitekturu *Model-view-controller* (u daljnjem tekstu koristit ćemo skraćenicu MVC). Glavna ideja koja stoji iza MVC arhitekture zasniva se na podjeli aplikacije u tri komponente od kojih svaka ima svoju ulogu u konačnom prikazivanju informacije korisniku. Kao što se može zaključiti iz samog naziva te su komponente - Model, *View* i *Controller.*



Slika 6.6: MVC [21]arhitektura programskog koda

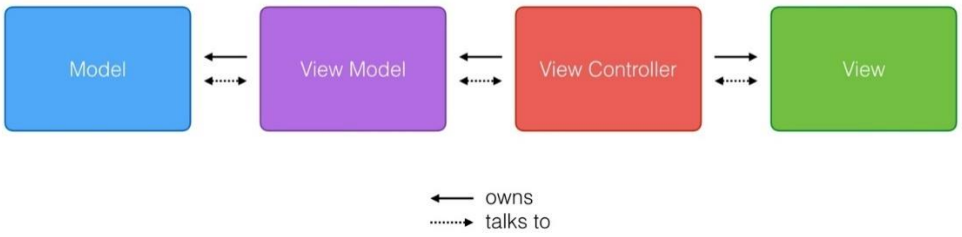
[21] https://www.progressioapps.com/mvvm-design-pattern-swift/, MVVM arhitektura koda (12.2.2020)

Model u MVC arhitekturi opisuje podatke koji su specifični za aplikaciju i definira poslovnu logiku i način manipulacije tih podataka. *Model* može imati jednu ili više poveznica s drugim modelima. U konačnici podaci koji se nalaze u modelima prikazuju se na *View* komponenti na kojoj se odvija uz prikaz informacija i interakcija korisnika. *View* kada zatraži podatke, ne komunicira direktno s *Model* komponentom u kojoj se nalaze podaci, nego direktno komunicira s *Controller-om* putem kojeg zatraži podatke. Tada *Controller* komunicira s modelom od kojih treba podatke te nakon što ih dobije, šalje ih ponovno *View*-u kako bi ih on mogao prikazati. Na ovaj način programski kod je podijeljen u dijelove u kojima s informacije pohranjuju i obrađuju, koji služi za prikaz informacije korisniku i dio koji služi za dohvaćanje i baratanje podacima. Mana MVC arhitekture je ta što kod većih aplikacija programski kod postaje nepregledan te se teže u njemu snaći i teže ga je testirati. Rješenje ovom problemu su mnoge druge arhitekture koje postoje u domeni razvijanja softvera poput *Model-View Model-View* arhitekture, *Model-View-Presenter* ili arhitekture za reaktivne metode razvoja softvera.

6.3.2. Arhitektura programskog koda mobilne aplikacije

Za razvoj mobilne aplikacije ovog rada koriste se principi *Model-View Model- View – Coordinator* arhitekture (u daljnjem tekstu koristit će se skraćenica MVVM+C). Kao što se može zaključiti iz naziva, ova arhitektura se sastoji od komponenata *Model, View Model, View* i *Coordinators*.

MVVM arhitektura se razlikuje u odnosu na MVC arhitekturu po *View Model* komponenti koja se nalazi između *View* komponente i Modela. Njezina je uloga da kada View preko svog kontrolera zatraži podatke, on umjesto direktno s modelom, komunicira s *View Model-om* koji je zadužen da komunicira s potrebnim modelom i obrađuje njegove podatke. Logika obrade podataka iz modela naziva se poslovna logika te će se u daljnjem tekstu koristiti taj izraz. U ovoj arhitekturi komponenta *View* se sastoji od komponente *View* i njezinog *Controllera* koji se ne spominje u nazivu arhitekture, ali se također u njoj nalazi. Cilj ove arhitekture je da se sva poslovna logika koja se u MVC arhitekturi djelomično odvija u Modelima i djelomično u *Controller* komponenti potpuno odvoji u *View Model* komponentu, a da se u *View* komponenti odvija samo logika prikaza informacija i konfiguracije sučelja.

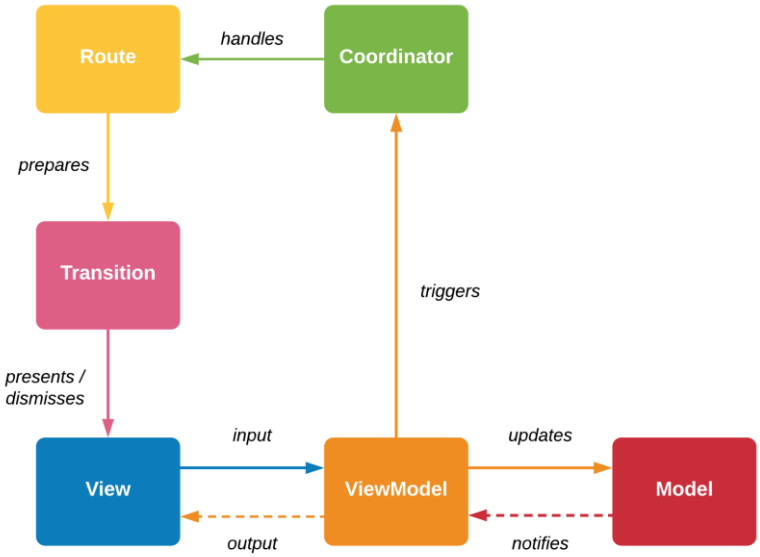


Slika 6.7: MVVM [22]Arhitektura programskog koda

U sklopu MVVM+C arhitekture nalazi se i komponenta *Coordinators* čiji je doslovni prijevod na hrvatski jezik koordinatori. Koordinator kao zasebna komponenta predstavlja arhitekturu navigacije između zaslona kroz aplikaciju. Ideja iza koordinatora je da se metode za tranzicije između zaslona odvoje u posebnu komponentu koja će biti samo zadužena za to. Bez koordinatora navigacija između zaslona poziva se unutar samih kontrolera *View* komponenti gdje u slučaju ako aplikacija ima veći broj zaslona dolazi do problema nepreglednosti metoda za navigaciju i prenošenja podataka između zaslona. S koordinatorima taj problem se rješava na način da se logika za navigaciju postavlja u zasebnu komponentu koja je zadužena za navigaciju između zaslona te prenošenje podataka između istih.

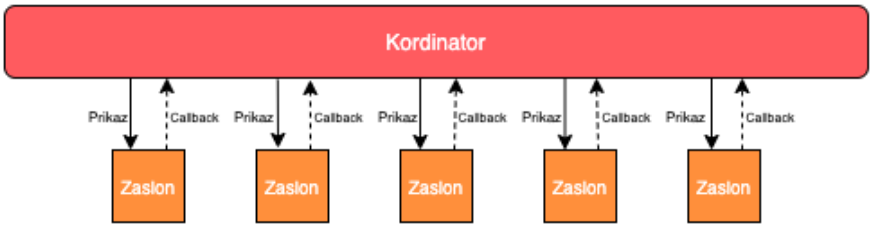
Korištenjem *Coordinators* arhitekture definiraju se koordinatori od kojih će se aplikacija sastojati. Postoje dvije vrste koordinatora, glavni koordinator (engl. *Main Coordinator*) i djeca koordinatora (engl. *Child Coordinators*). Glavni koordinator je vlasnik svih koordinatora djece u aplikaciji te je on zadužen za pozivanje „djece“ koordinatora. Djeca koordinatori definirani su navigacijskim tokom određenje radnje ili funkcionalnosti unutar aplikacije te mogu komunicirati s drugim koordinatorima unutar aplikacije.

[22] https://cocoacasts.com/how-does-mvvm-work, Opis rada MVVM arhitekture (12.2.2020)



Slika 6.8: MVVM+C [23]arhitektura programskog koda

Kao primjer se može navesti početni zaslon Facebook mobilne aplikacije na kojoj se nalaze postovi korisnika. Na tom zaslonu se mogu pregledavati postovi te kada se stisne na određeni post, otvara se novi zaslon s detaljnijim prikazom posta. Unutar tog zaslona ako se stisne na sliku otvara se novi zaslon za prikaz slike. Ovo je primjer navigacijskog toka *child* koordinatora koji se može radi primjera nazvati „Početni Koordinator“ u kojem se nalaze metode za otvaranje zaslona za detaljniji prikaz posta ili u slučaju ako se pritisne na sliku, otvaranja zaslona za prikaz slike.



Slika 6.9: Prikaz *child coordinator* u odnosu na *coordinator*

U gore navedenom dijagramu arhitekture koordinatora zaslon prikazuje koordinator, a zaslon povratno komunicira putem povratnih metoda (engl. *Callback). Callback* je anonimna

[23] https://iosexample.com/reactive-navigation-library-for-ios-based-on-the-coordinator-pattern/ , *Coordinator* arhitektura (12.2.2020)

metoda koja ne vraća ništa, nego šalje podatke i javlja koordinatoru kada treba prikazati sljedeći zaslon. Tada koordinator podatke iz *callbacka* prosljeđuje u metodu koja kreira sljedeći zaslon i vraća ga kako bi ga koordinator mogao prikazati. U nastavku se nalazi primjer programskog koda jednog od koordinatora mobilne aplikacije ovog rada.

```
class TripsCoordinator: Coordinator {
        private var childCoordinators: [Coordinator] = []
        private var rootCoordinator: Coordinator?
        private var navigationController = BaseNavigationController()

        func start() -> UIViewController {
                let vc = self.createTripsViewController()
                navigationController.viewControllers = [vc]
                return navigationController
        }

        func createTripsViewController() -> UIViewController {
                let viewcontroller = TripsViewController.instance()
                let viewmodel = TripsViewModel()

                viewcontroller.viewModel = viewmodel

                viewmodel.onCellTapped = { [weak self] data in
                        guard let self = self else { return }
                        self.navigationController.pushViewController(self.crea
                        teTripsInfoViewController(data: data), animated: true)
                }

                return viewcontroller
        }

        func createTripsInfoViewController(data: TripModel) ->
        UIViewController {
                let viewcontroller = TripsInfoViewController.instance()
                let viewmodel = TripsInfoViewModel(trip: data)

                viewcontroller.viewModel = viewmodel

                return viewcontroller
        }
}
```

Kod 6.1: Primjer koordinator klase

Svaka klasa koordinatora nasljeđuje protokol *Coordinator* unutar kojeg se nalazi `start()` metoda koju svaki koordinator mora zadovoljiti. Uloga te metode je da pokrene koordinator i vrati ga kao `UIViewController` objekt. U razvoju iPhone aplikacija sve

što definira zaslon mobilne aplikacije uključujući i navigacijske kontrole tipa je ili nasljeđuje `UIViewController` klasu.

```
public protocol Coordinator: class {
    @discardableResult
    func start() -> UIViewController
}
```

U koordinatoru se može vidjeti kako je u *View Modelu* definirana anonimna metoda pod nazivom `onCellTapped` koja u sebi sadrži podatak pod nazivom data. Kada *view model* pozove tu metodu, ona se automatski izvršava u koordinatoru te se podatak *data* prosljeđuje u metodu za kreiranje sljedećeg zaslona.

```
class TripsViewModel: BaseViewModel {
    var tripModel: TripModel!
    var onCellTapped: ((TripModel) -> Void)?

    func goOnCellTapped() {
        onCellTapped?(tripModel)
    }
}
```

Kod 6.2: Primjer *view model* klase

U primjeru koda *view model* komponente nalazi se definirana anonimna metoda i metoda koja služi za pozivanje anonimne metode iz *View* kontrolera. Pozivanje tranzicija između zaslona detaljnije će biti obrađeno u narednim poglavljima.

6.4. Kreiranje kartične kontrole

U prijašnjem poglavlju spomenuto je kako će se aplikacija bazirati na kartičnom tipu aplikacije te je objašnjeno što to znači. U ovom poglavlju biti će objašnjeno kako je ta kontrola kreirana i kako funkcionira.

Za kreiranje kartičnog tipa aplikacije potrebna je kontrola pod nazivom *Tab Bar Control*. Ta kontrola je sastavni element UIKit Swift biblioteke za razvoj korisničkih sučelja, korištenja kontroli i elemenata za korisničko sučelje. Po arhitekturi koordinatora ova kontrola se vodi kao glavni koordinator u mobilnoj aplikaciji iz razloga što je glavni navigator kroz aplikaciju. Prije samog pisanja koda za kreiranje *Tab Bar Controller-a*

definirani su *child* koordinatori koje će *Tab Bar Controller* prikazivati pritiskom na pojedinu karticu (engl. *tab*) unutar same kontrole.

Nazivi *child* koordinatora:

- ***HomeCoordinator*** **–** koordinator za prikaz zaslona jako bitnih i srednje bitnih informacija.
- ***TripsCoordinator*** **–** koordinator za prikaz popisa putovanja.
- ***MapCoordinator*** **-** koordinator za prikaz karte na kojoj ce se prikazivati lokacija vozila.
- ***ProfileCoordinator-*** koordinator na kojem se prikazuje statistika o vožnji i informacije o korisniku.

Pritiskom na pojedinu karticu *Tab Bar* kontrola prikazuje koordinator koji kartica predstavlja.

```
 class MainCoordinator: NSObject, Coordinator {

    fileprivate var childCoordinators: [Coordinator] = [HomeCoordinator(),
TripsCoordinator(), MapCoordinator(), ProfileCoordinator()]

    @discardableResult
    func start() -> UIViewController {
        return startMain()
    }

    @discardableResult
    private func startMain() -> UIViewController {
        let tabBarController = createTabBarController()
        tabBarController.showAsRoot()

        if let appDelegate = UIApplication.shared.delegate as? AppDelegate {
            appDelegate.rootCoordinator = self
        }
        return tabBarController
    }
}

// MARK: - Main Tab bar configuration
extension MainCoordinator {
    fileprivate func tabBarItem(for coordinator: Coordinator) ->
UITabBarItem {
```

```
        switch coordinator {
        case is HomeCoordinator:
            return .home
        case is TripsCoordinator:
            return .tours
        case is MapCoordinator:
            return .map
        case is ProfileCoordinator:
            return .settings
        default:
            fatalError("No tab bar set for this coordinator!")
        }
    }

    fileprivate func createTabBarController() -> UITabBarController {
        let tabBarController = UITabBarController()
        let viewControllers = childCoordinators.map { coordinator ->
                                            UIViewController in
            let vc = coordinator.start()
            vc.tabBarItem = tabBarItem(for: coordinator)

            return vc
        }

        tabBarController.tabBar.isTranslucent = true
         tabBarController.tabBar.backgroundColor =
        UIColor.black.withAlphaComponent(0.5)
        tabBarController.delegate = self
        tabBarController.tabBar.backgroundImage = UIImage()
        tabBarController.viewControllers = viewControllers
        return tabBarController
    }
}

extension MainCoordinator: UITabBarControllerDelegate {
    func tabBarController(_ tabBarController: UITabBarController,
shouldSelect viewController: UIViewController) -> Bool {

        if tabBarController.selectedViewController == viewController {
            guard let navigationController = viewController as?
                                UINavigationController else { return true }
            guard let vc = navigationController.viewControllers.first as?
                                HomeViewController else { return true }
            vc.scrollToDashboard()
            return false
        } else {
            return true
        }
    }
}
```

Kod 6.3: `MainCoordinator` koordinator TabBar kontrole

U varijabli `childCoordinators` koja je dostupna samo na razini klase, pohranjeni su svi *child* koordinatori koje će kartična kontrola sadržavati. Protokol metoda `start()` pokreće i vraća rezultat `startMain()` metode tipa `UIViewController`. Unutar `startMain()` metode kreira se kartična kontrola tipa `UITabBarController` u metodi `createTabBarController()` na način da se svaki koordinator koji se nalazi u `childCoordinators` putem protokol metode `start()` pokrene i pohrani u `vc` varijablu na razini petlje. Tada se za taj koordinator kreira kartica koja će se nalaziti unutar kartične kontrole i predstavljati zadani koordinator. Kartica se kreira u `tabBarItem` metodi koja prima varijablu tipa *Coordinator* i vraća tip `UITabBarItem` koji predstavlja kartice unutar kartične kontrole. Metoda pomoću *switch* naredbe grananja na temelju vrste koordinatora kreira karticu s ikonom za zadani koordinator te kao rezultat vraća tu karticu i zadaje ju `tabBarItem` *property-u* koji se nalazi na svakom objektu tipa `UIViewController`. Na kraju kada su svi *child* koordinatori pokrenuti i imaju kreirane kartice za kartični kontroler pohranjuju se u listu `viewControllers` koja se pridodaje na istoimeni *property* koji je dio `UITabBarController` objekta. Na taj način se dodaju koordinatori i njihove kartice koje će kartični kontroler sadržati. U toj metodi također se definiraju pozadinska boja same kontrole i koji protokol kontrola koristi.

Na samom kraju koda vidi se protokol metoda `shouldSelect` koja vraća tip objekta *Bool* koji definira selektira li se na pritisak korisnika kartica ili ne. U njoj je definirano da na pritisak kartice ako se korisnik nalazi na HomeCoordinatoru kontroler ne selektira ponovno karticu, nego pokrene metodu `scrollToDashboard()` o kojoj će biti rečeno više u nastavku ovog rada.

Nakon što je kartični kontroler kreiran, u metodi `start()` zadanom *Coordinator* protokolom na novo kreiranom kontroleru pokreće se metoda `showAsRoot()` koja kartični kontroler na glavni *window* aplikacije postavlja kao korijenski kontroler, što znači da se taj kontroler prikazuje kao glavni kontroler na razini aplikacije.

Metoda `start()` glavnog koordinatora poziva se iz klase pod nazivom *AppDelegate.* To je klasa unutar koje se nalaze protokoli vezani na životni vijek (engl. *life cycle*) mobilne aplikacije na razini operacijskog sustava. Sadrži metode koje se pozivaju kada aplikacija počne raditi u pozadinskom načinu rada, kada se aplikacija ponovno upali iz pozadinskog načina rada, kada se aplikacija ugasi, kada se za vrijeme aplikacije na razini operacijskog sustava dogodi da aplikacija s većim prioritetom rada trenutnu aplikaciju

prebaci u neaktivno stanje (poput aplikacije za pozive kada dođe dolazni poziv), kada se aplikacija vrati u aktivno stanje i onu najbitniju metodu koja se poziva kada je aplikacija uspješno pokrenuta. Ta metoda se zove `application` i prepoznaje se po nazivu parametra protokol metode koja ju poziva `didFinishLaunchingWithOptions`. Unutar te metode putem metode `start()` protokola `Coordinator` pokreće se željeni koordinator nakon paljenja aplikacije koji je u ovom slučaju `MainCoordinator`.

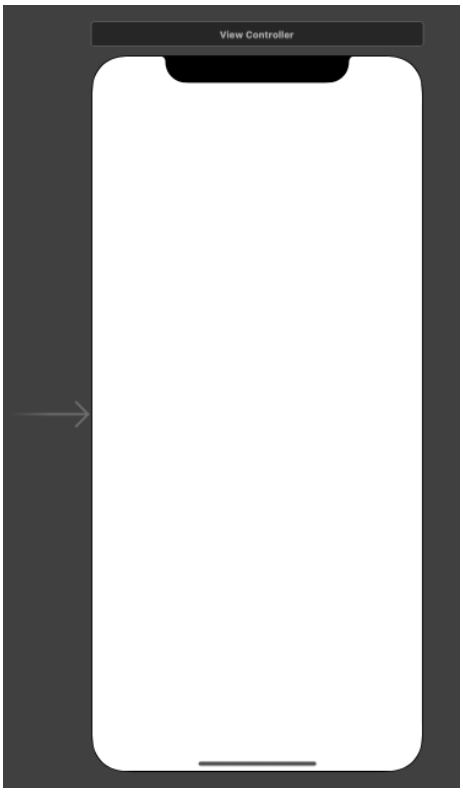
```
func application(_ application: UIApplication, didFinishLaunchingWithOptions launchOptions:
[UIApplication.LaunchOptionsKey: Any]?) -> Bool {
                        let user = UManager.user

                        if (user != nil) { // If user exists go to MainCoordinator
                            rootCoordinator = MainCoordinator()
                        } else {
                            rootCoordinator = AuthCoordinator()
                        }
                        rootCoordinator?.start()
                        return true
                    }
```

Kod 6.4: App delegate `didFinishLaunchingWithOptions` metoda

Prije samog pokretanja glavnog koordinatora provjerava se je li korisnik prijavljen u aplikaciju te u slučaju ako je, pokreće se glavni koordinator. U slučaju da korisnik nije prijavljen pokreće se koordinator za prijavu korisnika te nakon što korisnik uspješno unese podatke za prijavu pokreće se glavni koordinator.

6.5. Kreiranje zaslona

Prilikom razvoja mobilnih aplikacija Swift programskim jezikom u XCode okruženju jedan ili više zaslona (*ViewController*-a) definiraju se pojmom *storyboard.* Zasloni u MVVM+C arhitekturi kreiraju se u trenutku kada *view model* putem *callback*-a komunicira koordinatoru da prikaže željeni zaslon. *Coordinator* arhitektura zadaje dva pravila kojih se programer mora pridržavati kako bi uspješno kreirao i prikazao zaslone. Prvo pravilo je da *storyboard* sadrži samo jedan zaslon koji je zadan kao inicijalni zaslon koji je prepoznatljiv putem strelice s lijeve strane zaslona. Drugo pravilo je da naziv klase *controller-a* zaslona identičan nazivu *storyboard*-a.



Slika 6.10: Prikaz zaslona u *storyboardu*

U koordinatoru su definirane metode koje služe za kreiranje zaslona. Svaki zaslon ima svoju metodu koja može ili ne mora primati podatke. Metoda za kreiranje zaslona prima podatke u slučaju kad se želi neke podatke prenijeti s prijašnjeg zaslona na novi zaslon.

```
func createTripsInfoViewController(data: TripModel) -> UIViewController {
      let viewcontroller = TripsInfoViewController.instance()
      let viewmodel = TripsInfoViewModel(trip: data)
      viewcontroller.viewModel = viewmodel
      return viewcontroller
}
```

Kod 6.5: Metoda u koordinatoru za kreiranje zaslona

Unutar te metode zaslon se kreira pomoću `instance()` metode koja je kreirana kao ekstenzija `UIViewController` klase.

```
extension UIViewController {
    open class func instance() -> Self {
        if let vc = createFromStoryboard(type: self) {
            return vc
```

```
            } else {
                print("WARNING: can't create view controller from
                storybard:\(self)")
                return self.init()
            }
        }

        private class func createFromStoryboard<T:
                UIViewController>(type: T.Type) -> T? {
            let storyboardName = String(describing: type)
            let bundle = Bundle(for: T.self)

             guard bundle.path(forResource: storyboardName,
                    ofType: "storyboardc") != nil else {
             return nil
            }

            let storyboard = UIStoryboard(name: storyboardName,
                bundle: bundle)

            guard let vc =
                storyboard.instantiateInitialViewController()
                else {
                print("no vc in storyboard(hint: check initial
                            vc)")
                return nil
            }
            return vc as? T
        }
    }
```

Kod 6.6: Kreiranje UIViewController zaslona na temelju naziva

Budući da je metoda `instance()` kreirana kao ekstenzija `UIViewController` klase može se pozivati na bilo kojem objektu koji je tog tipa. Unutar nje se poziva metoda `createFromStoryboard` koja kao ulaznu varijablu prima objekt tipa `UIViewController` nad kojim je pozvana metoda `instance()`. U metodi `createFromStoryboard` na temelju ulaznog objekta definira se naziv klase tog objekta. S tim podatkom unutar projekta pronalazi se *storyboard* file željenog zaslona što je moguće iz razloga što su nazivi klase *controller*-a zaslona i *storyboard file*-a identični. Kada je

storyboard željenog zaslona pronađen metodom `instantiateInitialViewController()` stvara se *view controller* objekt zaslona te ga se vraća kao povratni tip metode `createFromStoryboard` i metode `instance()`.

Nakon što je u koordinatoru kreiran *UIViewControler* objekt zaslona, kreira se *view model* zaslona u kojem se odvija sva poslovna logika obrade podataka. U *view model* pomoću ubrizgavanja ovisnosti prosljeđuje se podatak koji je potreban tom zaslonu te se definira da je taj *view model* onaj *view model* koji će taj zaslon koristiti.

Nakon što je zaslon uspješno kreiran, jednom od metoda prikazivanja zaslon se prikazuje na mobilnoj aplikaciji. U ovoj mobilnoj aplikaciji koriste se tri metode prikazivanja zaslona. Prva metoda je putem glavnog koordinatora i kartične kontrole. Unutar *child* koordinatora zasloni se prikazuju uz pomoć *navigation controllera* i pomoću metode `present()` koja se nalazi na svakom objektu tipa `UIViewController`.

Navigation controller je isto kao *TabBar Controller* kontrola za prikaz zaslona koja sadrži listu zaslona za prikazivanje. Onaj zaslon koji je zadnje dodan na listu zaslona automatski se prikazuje putem *navigation controller-a.* Na njemu se također automatski u gornjem djelu zaslona prikazuje *navigation bar* s strelicom za povratak na prijašnji zaslon. Svaki *child* koordinator na razini klase ima svoj *navigation controller* koji unutar tog koordinatora prikazuje zaslone. Pritiskom na strelicu za povratak zaslon koji se trenutno prikazuje uklanja se iz liste zaslona i prikazuje se onaj zaslon koji se nalazio prije trenutno uklonjenog zaslona. U nastavku se nalazi primjer programskog koda dodavanja novog zaslona *navigation controller*-u.

```
self.navigationController.pushViewController(UIViewController()),
            animated: true)
```

Metodom `present()` prikazuje se zaslon preko trenutno prikazanog zaslona te za uklanjanje tako prikazanog zaslona programer mora sam kreirati kontrolu.

6.6. Mrežni ruteri i servisi

Mrežni ruteri i servisi su dio mobilne aplikacije čija je uloga komuniciranje s mrežnim servisima te dohvaćanje podataka sa njih koji su potrebni za rad aplikacije. Kako bi to bilo moguće, potrebno je da je mobilni uređaj na kojem se koristi mobilna aplikacija povezan s internetom. Kako bi se uspješno uspostavila veza između mobilne aplikacije i mrežnog servisa, potrebno je nekoliko informacija kako bi to bilo moguće. Potrebno je definirati pravila za uspostavljanje veze i pripremiti mobilnu aplikaciju na podatke koje može očekivati nakon što je veza uspješno uspostavljena. Informacije i pravila koja su potrebna za uspostavljanje veze definirana su od strane REST API sučelja čija je uloga uspostavljanje veze s mrežnom bazom podataka u kojoj se nalaze podaci, obrađivanje tih podataka i u konačnici vraćanje podataka uređaju s kojeg je uspostavljena konekcija.

REST je skraćenica od engleskog naziva arhitekture za razvoj softvera *Representational state transfer* koja definira ograničenja i pravila za razvoj mrežnih servisa. REST API definira rute, metodu veze i njezine parametre koji su potrebni da bi mrežni servis uspješno vratio željene podatke. Rute, metode i parametri mrežnih servisa definirani su u dokumentacijama koje su izdane od strane programera ili tvrtki. Njihova je svrha dobro opisati svaku rutu, metodu i potrebne parametre kako bi korištenje mrežnog servisa bilo moguće. Primjer mrežne rute servisa za vremensku prognozu https://api.openweathermap.org/data/2.5/weather .

Rute u mobilnoj aplikaciji ovog rada interpretiraju se kroz segmente bazna ruta i njezina putanja. Bazna ruta je ruta koja definira mrežni servis na koji se uređaj spaja, u slučaju prethodnog primjera to je https://api.openweathermap.org/data/2.5. Putanja rute definira točno koji zahtjev REST API treba izvršiti kako bi mobilna aplikacija dobila željene podatke. U prijašnjem primjeru putanju predstavlja „/weather“.

Uz informacije o baznoj ruti i putanjama zahtjeva mrežnog servisa potrebno je znati kojom metodom će se uspostaviti konekcija i koji parametri su potrebni REST API-u da bi izvršio zahtjev na željenoj putanji. Pomoću svih tih informacija moguće je kreirati mrežni zahtjev koji predstavlja cjelinu rute i putanje, metodu spajanja i parametre pomoću kojih se uspostavlja veza s mrežnim servisom.

Uspostavljanje komunikacije s mrežnim servisima u mobilnoj aplikaciji razdvojeno je u dva sloja od kojih svaki ima svoju ulogu:

- **API Ruter**

  Uloga ovog sloja je da kreira mrežne zahtjeve potrebne za komunikaciju s mrežnim servisima.

- **API *Manager***

  Uloga ovog sloja je da sadrži metode za uspostavljanje veze s mrežnim servisima. Svaka metoda poziva API ruter sloj koji vraća mrežne zahtjeve potrebne za uspostavljanje veze.

Sloj za kreiranje zahtjeva definiran je kao enumeracija čiji tipovi su izrazi koji predstavljaju pojedini mrežni zahtjev.

```
public enum WeatherAPIRouter: URLRequestConvertible {
    case now(Double, Double)
    case forecast(Double, Double)

    var path: String {
        switch self {
        case .now:
            return "/weather"
        case .forecast:
            return "/forecast"
        }
    }

    var method: HTTPMethod {
        return .get
    }

    var parameters: [String: String] {
        switch self {
        case .now(let lat, let lon):
            return ["lat": String(lat), "lon": String(lon), "appid":
                      Config.OAuthWeather.appid]
        case .forecast(let lat, let lon):
            return ["lat": String(lat), "lon": String(lon), "appid":
                      Config.OAuthWeather.appid]
        }
    }

    public func asURLRequest() throws -> URLRequest {
        let url = URL(string: Config.OAuthWeather.baseURL)
        let urlRequest = try URLRequest(url:
```

```
                (url?.appendingPathComponent(path))!, method: method)
            let request = try URLEncoding.default.encode(urlRequest, with:
                parameters)
            return request
        }
    }
```

Kod 6.7: Sloj za kreiranje mrežnih zahtjeva servisa za vremensku prognozu

Tipovi enumeracije mogu i ne trebaju primati podatke ovisno o tome očekuje li mrežni zahtjev parametre od korisnika ili ne. U navedenom primjeru programskog koda sloj za kreiranje mrežnih zahtjeva sadrži dva tipa enumeracije *now* i *forecast* od kojih svaki prima po dva ulazna podatka tipa `Double`. Ti podaci u primjeru predstavljaju geografsku širinu i dužinu lokacije za koju se žele dobiti podaci o vremenskoj prognozi.

Enumeracija je definirana protokolom `URLRequestConvertible` koji je dio Alamofire [24]biblioteke koja sadrži metode i protokole za komunikaciju s mrežnim servisima. Taj protokol omogućava kreiranje mrežnog zahtjeva na razini enumeracije metodom `asURLRequest()` koja se poziva svaki puta kada je tip enumeracije definiran. Unutar te metode na temelju bazne rute mrežnog servisa kreira se mrežni zahtjev s putanjom, metodom zahtjeva i parametrima koji su definirani unutar varijabli `path`, `method` i `parameters`. Varijable poprimaju vrijednosti na temelju tipa enumeracija u trenutku kada su pozvane prilikom kreiranja mrežnog zahtjeva. Prilikom kreiranja mrežnog zahtjeva definira se na koji će se način podaci parametara slati na mrežni servis, kao dio rute mrežnog zahtjeva (engl. URL *Encoded*) ili kao JSON podatak (engl. JSON *Encoded*). Na ovaj način sloj za kreiranje mrežnih zahtjeva jednostavan je za pozivanje unutar koda i lako je proširiv na kreiranje više mrežnih zahtjeva. U nastavku se može vidjeti primjer kreiranja mrežnog zahtjeva.

```
let urlRequest = WeatherAPIRouter.now(45.810987, 15.977376)
```

Sloj za uspostavljanje veze s mrežnim servisima definiran je kao klasa koja sadrži statične metode koje ovisno o tipu mrežnog zahtjeva mogu i ne trebaju primati ulazne podatke. Svaka metoda sadrži anonimnu metodu koja se izvršava nakon što je dobiven odgovor od mrežnog servisa. Unutar tih metoda pozivaju se metode koje su dio Alamofire biblioteke pomoću kojeg se otvaraju veze prema mrežnim servisima.

[24] https://github.com/Alamofire/Alamofire, Alamofire biblioteka (18.2.2018.)

```
class WeatherServiceManager {

        public static func now(lat: Double, lon: Double, onComplete: @escaping
(NetworkResponse<WeatherModel>) -> Void) {

    AF.request(WeatherAPIRouter.now(lat, lon)).responseData { (response) in
        let statusCode = response.response?.statusCode ?? 0000

        switch response.result {
                case .success(let data):
                        guard let weather = try? JSONDecoder().decode(WeatherModel.self,
                case
                from: data) else {
                        onComplete(.failure(NetworkError(message: ErrorMessage.parse)))
                                return
                        }

                        onComplete(.success(weather))
                case .failure(_):
                        onComplete(.failure(NetworkError(message: ErrorMessage.failure(code:
                                statusCode))))
                            }
                        }
                    }

        public static func forecast(lat: Double, lon: Double, onComplete: @escaping
                (NetworkResponse<ForecastResponseModel>) -> Void) {

        AF.request(WeatherAPIRouter.forecast(lat, lon)).responseData { (response) in
            let statusCode = response.response?.statusCode ?? 0000

            switch response.result {
                case .success(let data):
                        guard let forecasts = try?
                case
                JSONDecoder().decode(ForecastResponseModel.self, from: data) else {
                                    onComplete(.failure(NetworkError(message:
                                            ErrorMessage.parse)))
                          return
                        }
                        onComplete(.success(forecasts))
                case .failure(_):
                        onComplete(.failure(NetworkError(message: ErrorMessage.failure(code:
                                statusCode))))
                }
            }
        }
}
```

Kod 6.8: Sloj za uspostavljanje veze s mrežnim servisom vremenske prognoze

U metodama na objektu `AF` Alamofire biblioteke poziva se metoda `request()` koja kao ulazni podatak prima mrežni zahtjev kreiran od strane API ruter sloja i anonimnu

metodu `responseData()` unutar koje se provjerava odgovor mrežnog servisa i podataka koji su vraćeni. Naredbom grananja provjerava se uspješnost uspostavljenje veze s mrežnim servisom. Ako je rezultat odgovora mrežnog servisa neuspješan, znači da veza nije uspješno uspostavljena, što znači da je došlo do greške u komunikaciji. Ako je odgovor uspješan, podaci koji su vraćeni kao odgovor s mrežnog servisa u obliku JSON objekta raspakiravaju se i pohranjuju u očekivani model podataka koji je predefiniran na temelju dokumentacije mrežnog servisa.

Nakon što su podaci uspješno pohranjeni, izvršava se anonimna metoda kojom se dobiveni podaci s mrežnog servisa prosljeđuju kao rezultat metode za uspostavljanje veze.

U daljnjim poglavljima biti će objašnjeno kako izgleda pozivanje metoda za uspostavljanje komunikacije s mrežnim servisima.

6.7. Komunikacija s OBD-II modulom

U ovom poglavlju objasnit će se kako mobilna aplikacija komunicira s OBD uređajem i na koji način dohvaća podatke vozila. Model OBD-II uređaja koji se koristi u ovom radu je model prve generacije OBD-II uređaja danske tvrtke AutoPi. Uređaj je baziran na Raspberry Pi računalu s 4G modemom za komunikaciju s internetom, Wi-Fi modemom te Bluetooth-om. Ovaj uređaj je odabran za korištenje u ovom radu iz razloga što uz više mogućnosti komuniciranja s uređajem i s upravljačkom jedinicom motora prati putovanja automobila, bilježi lokaciju vozila putem GPS senzora te je moguće pisati *custom* skripte za razne evente koji se izvršavaju za vrijeme vožnje. Uređaj direktno komunicira sa *cloudom* tvrtke AutoPi kojem se može pristupiti registriranim korisničkim računom s kojim je spojen ID uređaja kako bi se mogao dobiti uvid u podatke s uređaja. Također postoji REST API za dohvaćanje podataka kao što su putovanja vozila i trenutna lokacija vozila.

Uz sve kvalitetne karakteristike, u ovom radu nije iskorišten puni potencijal uređaja iz razloga što uređaj sadrži mnogo nedostataka u svojim performansama što je i očekivano od uređaja prve generacije. Neki od nedostataka su velika količina vremena potrebna za uspostavljanje komunikacije s uređajem i ne optimizirana direktna komunikacija s LEM327 mikrokontrolerom. Pri izradi same mobilne aplikacije veliki izazov je bio uspostaviti direktnu komunikaciju s uređajem i dohvaćati podatke iz vozila u stvarnom vremenu jer nije bilo moguće uspostaviti direktnu komunikaciju putem Bluetooth-a i Wi-Fi modula s

LEM327 mikrokontrolerom. U svrhu realizacije ovog rada odlučeno je da će se direktna komunikacija mobilnog uređaja s OBD-II modulom realizirati putem lokalnog servera koji se nalazi na Raspberry Pi računalu koje komunicira sa LEM327 mikrokontrolerom i dohvaća podatke iz upravljačke jedinice motora. Za to je potrebno da mobilni uređaj bude povezan putem Wi-Fi mreže na sam uređaj OBD-II uređaj.

Na strani mobilne aplikacije komunikacija s lokalnim serverom OBD-II uređaja putem lokalne mreže tretira se kao komunikacija s bilo kojim mrežnim servisom. U prijašnjem poglavlju objašnjeno je kako generalno funkcionira komunikacija mobilne aplikacije s mrežnim servisima te se slojevita organizacija komponenti potrebnih za uspostavljanje veze primijenila i na komunikaciji s OBD-II uređajem.

Bazna ruta za spajanje na lokalni server OBD uređaja u sebi sadrži podatak ID uređaja koji se dohvaća iz podataka dobivenih nakon uspješne prijave korisnika na samom pokretanju aplikacije. Na taj način je uvedena dodatna razina sigurnosti dohvaćanja podataka s lokalnog servera na način da podatke mogu dohvatiti samo oni uređaji koji znaju ID uređaja na koji se spajaju. Bazna ruta za komunikaciju s lokalnim serverom OBD-II uređaja: http://local.autopi.io:9000/dongle/c7498956-18f7-962c-dab6-fd3a30877ce4/execute/.

Za jednostavnije dohvaćanje podataka kreirana je klasa koja sadrži sve metode potrebne za komuniciranje s OBD-II lokalnim serverom i protokoli pomoću kojih se dohvaćeni podaci osvježavaju na zaslonu mobilne aplikacije.

```
class VehicleOBD {
    weak var delegate: VehicleOBDCommunicationDelegate?
    static let data = VehicleOBD()
    private var timer: Timer?

    func start_read_data() {
        timer = Timer.scheduledTimer(timeInterval: 1.0, target: self,
               selector: #selector(read_data), userInfo: nil, repeats: true)
    }

    func stop_read_data() {
        timer?.invalidate()
    }

    @objc private func read_data() {
```

```
            read_SPEED()
            read_RPM()
            read_COOLANT_TEMP()
        }

        func read_SPEED() {
            VehicleDataManager.SPEED { [weak self] response in
                guard let self = self else { return }
                switch response {
                case .success(let vehicleData):
                    let speedValue = vehicleData.value ?? 0
                    self.delegate?.didReadSpeed(speedValue)
                case .failure(let error):
                    print(error.error.message)
                }
            }
        }

        func read_RPM() {
            VehicleDataManager.RPM { [weak self] response in
                guard let self = self else { return }
                switch response {
                case .success(let vehicleData):
                    let rpmValue = vehicleData.value ?? 0
                    self.delegate?.didReadRPM(rpmValue)
                case .failure(let error):
                    print(error.error.message)
                }
            }
        }

        func read_COOLANT_TEMP() {
            VehicleDataManager.COOLANT_TEMP { [weak self] response in
                guard let self = self else { return }
                switch response {
                case .success(let vehicleData):
                    let tempValue = vehicleData.value ?? 0
                    self.delegate?.didReadCoolantTemp(tempValue)
                case .failure(let error):
                    print(error.error.message)
                }
            }
        }
    }
```

Kod 6.9: Klasa koja sadrži metode za komunikaciju s OBD-II uređajem

Klasa `VehicleOBD` inicijalizira se u *view modelu* zaslona na kojem se prikazuju podaci. U konstruktoru *view* modela stvori se objekt tipa `VehicleOBD` te se nakon kreiranja na delegate varijabli (naziv varijable tipa klase protokola, hrv. delegat) tog objekta

zada instanca protokola, koji će biti pozivan iz objekta. Nakon stvaranja objekta pozivanjem metode `start_read_data()` pokreće se dohvaćanje podataka s OBD-II uređaja. U toj metodi nalazi se brojač kojem je zadano da u vremenskom intervalu od 1 sekunde poziva privatnu metodu `read_data()`, unutar koje se pozivaju metode za uspostavljanje komunikacije s lokalnim serverom OBD-II uređajem. U svakoj od tih metoda poziva se statička metoda za uspostavljanje komunikacije i dohvaćanje željenog podatka, klase `VehicleDataManager` koja predstavlja sloj za uspostavljanje komunikacije se lokalnim serverom.

```
class VehicleDataManager {
        static func getSpeed(onComplete: @escaping ((NetworkResponse<CommandValueModel>) ->
                Void)) {
                AF.request(VehicleAPIRouter.speed).responseData { (response) in
                        onComplete(decodeData(response: response))
                    }
                }

        static func getRPM(onComplete: @escaping ((NetworkResponse<CommandValueModel>) ->
                Void)) {
                AF.request(VehicleAPIRouter.rpm).responseData { (response) in
                        onComplete(decodeData(response: response))
                    }
                }

        static func getCoolantTemp(onComplete: @escaping
                ((NetworkResponse<CommandValueModel>) -> Void)) {
                AF.request(VehicleAPIRouter.coolant_temp).responseData { (response) in
                        onComplete(decodeData(response: response))
                    }
                }

        private static func decodeData(response: AFDataResponse<Data>) ->
                NetworkResponse<CommandValueModel> {
        let statusCode = response.response?.statusCode ?? 00

        switch response.result {
                case .success(let data):
                guard let vehicleData = try? JSONDecoder().decode(CommandValueModel.self, from:
        data) else {
                        return .failure(NetworkError(message: ErrorMessage.parse))
                }

                return .success(vehicleData)
                case .failure(_):
                return .failure(NetworkError(message: ErrorMessage.failure(code:
                        statusCode)))
                }
        }
}
```

Kod 6.10: Sloj za uspostavljanje komunikacije s lokalnim serverom OBD-II uređaja

U navedenom primjeru koda sloja za uspostavljanje komunikacije mogu se vidjeti javne statičke metode `getSpeed()`, `getRpm()` i `getCoolantTemp()`. Sloj za komunikaciju sadrži metode za svaki pojedini podatak koji treba biti dohvaćen s lokalnog servera uređaja. Ove metode predstavljaju dohvaćanje podataka o trenutnoj brzini vozila, trenutnom broju okretaja motora i temperaturi rashladne tekućine motora. Pozivanjem svake od tih metoda kreira se mrežni zahtjev za komunikaciju s lokalnim serverom. Sloj za kreiranje mrežnih zahtjeva koji je enumeracijskog tipa, sadrži tipove mrežnih zahtjeva koji uspostavljanjem konekcije s lokalnim serverom uređaja pokreću procese za dohvaćanje podataka sa upravljačke jedinice motora, na temelju komande poslane kao parametar mrežnog zahtijeva.

```
public enum VehicleAPIRouter: URLRequestConvertible {
    case speed
    case rpm
    case coolant_temp

    var path: String {
        return ""
    }

    var method: HTTPMethod {
        return .post
    }

    var parameters: [String: Any] {
        switch self {
        case .speed:
            return ["command": "obd.query", "arg": ["SPEED"], "kwarg": "" ]
        case .rpm:
            return ["command": "obd.query", "arg": ["RPM"], "kwarg": "" ]
        case .coolant_temp:
            return ["command": "obd.query", "arg": ["COOLANT_TEMP"],
                        "kwarg": ""]
        }
    }

    public func asURLRequest() throws -> URLRequest {
        let url = URL(string: Config.Vehicle.localURL)
        var urlRequest = try URLRequest(url:
                (url?.appendingPathComponent(path))!, method: method)
        urlRequest.addValue("application/json", forHTTPHeaderField:
                "Content-Type")
        return try JSONEncoding.default.encode(urlRequest, with: parameters)
    }
}
```

Kod 6.11: Sloj za kreiranje mrežnog zahtjeva za komunikaciju s lokalnim serverom OBD-II uređaja

Varijabla `parameters` koja predstavlja parametre mrežnog zahtjeva, definira vrijednosti na temelju pozivanja tipa ove enumeracijske klase. Parametar tvore 3 podatka od kojih je najbitniji podatak „arg“ koji predstavlja argument podatka koji je potrebno dohvatiti iz upravljačke jedinice motora. Nakon što je mrežni zahtjev kreiran u klasi `VehicleDataManager`, uspostavlja se komunikacija s lokalnim serverom OBD-II uređaja i dohvaćaju se željeni podaci koji su definirani klasom objekta `CommandValueModel`. Nakon što su podaci uspješno pohranjeni u objekt, podaci se anonimnom metodom prosljeđuju kao rezultat uspješno uspostavljene veze s lokalnim serverom. U klasi `VehicleOBD` pozivanjem metoda sloja za uspostavu komunikacije, kao rezultat dobivaju se objekti enumeracijskog tipa `NetworkResponse` koji predstavljaju tip objekta koji je dohvaćen s lokalnog servera i informaciju je li komunikacija uspješno uspostavljena ili je došlo do greške u komunikaciji definiranu kao tip enumeracijske klase.

```
enum NetworkResponse<T: Decodable> {
    case success(T)
    case failure(NetworkError)
}
```

Kod 6.12: NetworkResponse enumeracijska klasa

U rezultatu metoda provjerava se uspješnost uspostavljene konekcije te u slučaju ako je konekcija uspješno uspostavljena dohvaćeni podatci prosljeđuju se putem protokola te se u *view modelu*, u kojem su definirana pravila protokolom, izvršavaju naredbe definirane unutar svake protokol metode.

```
extension MainDashboardCellViewModel:
VehicleOBDCommunicationDelegate {
    func didReadSpeed(_ value: Int) {
        self.onGotSpeed?(value)
    }
    func didReadRPM(_ value: Int) {
        self.onGotRPM?(value)
    }
    func didReadCoolantTemp(_ value: Int) {
        let string = "\(value)°C"
        self.onGotCoolantTemp?(string)
    }
```

```
}

extension MainDashboardCellViewModel: CMMotionDelegate {
    func didUpdateGForce(_ value: Double) {
        let string = String(format: "%.2f", value)
        self.onGotGForce?(string)
    }
}
```

Kod 6.13: Definirane metode protokola unutar view model-a

Unutar svake metode protokola poziva se anonimna funkcija definirana na razini *view model*-a koja predstavlja promjenu određenog podatka na zaslonu aplikacije. U nastavku rada za svaku funkcionalnost mobilne aplikacije biti će opisana logika dohvaćanja i prikaza podataka iz *view model*-a na elementima zaslona aplikacije.

6.8. Funkcionalnosti mobilne aplikacije

U ovom poglavlju opisat će se sve funkcionalnosti mobilne aplikacije. Funkcionalnosti su grupirane svaka na pojedini zaslon mobilne aplikacije te će za svaku funkcionalnost biti opisana logika dohvaćanja i obrade podataka te logika konačnog prikazivanja informacija na zaslon. Logika dohvaćanja i obrade podataka svakog zaslona odvija se *view model*-u zaslona ili kolekcijskih ćelija na kojem se podatak prikazuje.

6.8.1. Kontrolna ploča

Funkcionalnost kontrolne ploče predstavlja prikaz najbitnijih podataka i informacija za vrijeme vožnje automobilom. Prikazuje se kao jedna od tri ćelije elementa kolekcijskog prikaza unutar zaslona predstavljenog s prvom karticom kartične kontrole. Ćelija kontrolne ploče prikazuje se kao početna kartica elementa kolekcija, a s desne strane se nalaze ćelije koje prikazuju srednje funkcionalnosti mobilne aplikacije. Na taj način omogućen je jednostavan pregled srednje bitnih funkcionalnosti i jednostavan povratak na prikaz kontrolne ploče horizontalnim gestama po zaslonu, s lijeva na desno i s desna na lijevo.

Slika 6.11: Ćelija kontrolne ploče

Podaci koji se prikazuju na kontrolnoj ploči:

- Brzina kretanja automobila
- Broj okretaja motora
- Temperatura rashladne tekućine motora
- Potrošnja goriva
- Trenutna ulica po kojoj se automobil kreće
- Trenutna vremenska prognoza za trenutnu lokaciju
- Kompas
- Sila djelovanja na vozilo

Informacije unutar ćelije organizirane su na način da je svaki podatak prikazan unutar svog elementa pogleda (engl. *view*) stilom opisanim u poglavlju Korisničko sučelje. Tip klase koja komunicira s *view-om* i *view modelom* kolekcije ćelije je `UICollectionViewCell`. Njezina uloga je ista kao i uloga `UIViewController` tipa klase svih zaslona i navigacijskih kontrola aplikacije.

```
class MainDashboardCollectionViewCell: UICollectionViewCell {
    var viewModel: MainDashboardCellViewModel!
    var locationManager: CLLocationManager!

    @IBOutlet weak var mainView: UIView!
    @IBOutlet var widgetViews: [UIView]!
```

```
@IBOutlet var stackWidgetViews: [UIView]!
@IBOutlet weak var speedLabel: UILabel!
@IBOutlet weak var kmhLabel: UILabel!
@IBOutlet weak var rpmValueLabel: UILabel!
@IBOutlet weak var rpmLabel: UILabel!
@IBOutlet weak var gForceValueLabel: UILabel!
@IBOutlet weak var gLabel: UILabel!
//Gauges
@IBOutlet weak var speedGaugeView: Gauge!
@IBOutlet weak var rpmGaugeView: Gauge!
@IBOutlet weak var engineTempImageView: UIImageView!
@IBOutlet weak var engineTempLabel: UILabel!
@IBOutlet weak var fuelConsumptionLabel: UILabel!
@IBOutlet weak var navigationImageView: UIImageView!
@IBOutlet weak var addressLabel: UILabel!
@IBOutlet weak var weatherImageView: UIImageView!
@IBOutlet weak var temperatureLabel: UILabel!
@IBOutlet weak var mainTempLabel: UILabel!
@IBOutlet var compassLabels: [UILabel]!
@IBOutlet weak var compassImageView: UIImageView!

override func awakeFromNib() {
    super.awakeFromNib()
    viewModel = MainDashboardCellViewModel()
    configureCallbacks()

    configureElements()
    configureSpeedGauge()
    configureRPMGauge()
}


func configureCallbacks() {

    viewModel.onAdressChanged = { [weak self] address in
        guard let self = self else { return }
        self.addressLabel.text = address
    }

    viewModel.onCompasUpdated = { [weak self] angle in
        guard let self = self else { return }
        UIView.animate(withDuration: 0.5) {
            self.compassImageView.transform =
                  CGAffineTransform(rotationAngle: angle)
        }
    }

    viewModel.onGotGForce = { [weak self] string in
        guard let self = self else { return }
        self.gForceValueLabel.text = string
    }
```

```
        viewModel.onGotWeather = { [weak self] temperature, main, icon in
            guard let self = self else { return }
            self.temperatureLabel.text = temperature
            self.mainTempLabel.text = main
            KingfisherImageLoader.loadWeatherImage(icon: icon, imageView:
               self.weatherImageView)
        }

        viewModel.onGotSpeed = { [weak self] speedValue in
            guard let self = self else { return }
            self.speedLabel.text = String(speedValue)
            self.speedGaugeView.animateRate(0.5, newValue:
               CGFloat(speedValue)) { _ in }
        }

        viewModel.onGotRPM = { [weak self] value in
            guard let self = self else { return }
            self.rpmValueLabel.text = String(value)
            self.rpmGaugeView.animateRate(0.5, newValue: CGFloat(value)) { _
               in }
        }

        viewModel.onGotCoolantTemp = { [weak self] tempString in
            guard let self = self else { return }
            self.engineTempLabel.text = tempString
        }

        viewModel.onGotFuelConsumption = { [weak self] string in
            guard let self = self else { return }
            self.fuelConsumptionLabel.text = string
        }
    }

    func configureElements() {
        mainView.backgroundColor = .clear

        for widgetView in widgetViews {
            widgetView.backgroundColor =
               UIColor.black.withAlphaComponent(0.5)
            widgetView.layer.cornerRadius = 16
        }

        for widgetView in stackWidgetViews {
            widgetView.backgroundColor =
               UIColor.black.withAlphaComponent(0.5)
            widgetView.layer.cornerRadius = 12
        }

        for label in compassLabels {
            label.textColor = .white
```

```
            label.font = Fonts.nunitoSansBold.size(15)
        }

        compassImageView.tintColor = .lightOrange

        speedLabel.font = Fonts.nunitoSansSemiBold.size(52)
        speedLabel.textColor = .white

        kmhLabel.font = Fonts.nunitoSansBold.size(10)
        kmhLabel.textColor = .white

        rpmValueLabel.font = Fonts.nunitoSansSemiBold.size(30)
        rpmValueLabel.textColor = .white

        rpmLabel.font = Fonts.nunitoSansBold.size(10)
        rpmLabel.textColor = .white

        engineTempLabel.font = Fonts.nunitoSansSemiBold.size(48)
        engineTempLabel.textColor = .white

        fuelConsumptionLabel.font = Fonts.nunitoSansSemiBold.size(48)
        fuelConsumptionLabel.textColor = .white

        addressLabel.font = Fonts.nunitoSansSemiBold.size(40)
        addressLabel.textColor = .white

        temperatureLabel.font = Fonts.nunitoSansSemiBold.size(23)
        temperatureLabel.textColor = .white

        mainTempLabel.font = Fonts.nunitoSansRegular.size(17)
        mainTempLabel.textColor = .white

        gForceValueLabel.font = Fonts.nunitoSansBold.size(30)
        gForceValueLabel.textColor = .white

        gLabel.font = Fonts.nunitoSansBold.size(24)
        gLabel.textColor = .caveBlue

        weatherImageView.tintColor = .lightOrange

        navigationImageView.tintColor = .lightOrange
        engineTempImageView.tintColor = .lightOrange
    }

    func configureSpeedGauge() {
        speedGaugeView.colorsArray = [.slowGreen, .caveBlue, .lemonYellow,
               .pinkishDarkRed]
        speedGaugeView.bgColor = .blueGrey
        speedGaugeView.bgAlpha = 0.20
        speedGaugeView.shadowRadius = 1
        speedGaugeView.rotate = 45.0
```

```
            speedGaugeView.maxValue = 180
            speedGaugeView.lineWidth = 22

        }

        func configureRPMGauge() {
            rpmGaugeView.colorsArray = [.lemonYellow, .lightOrange,  .highRed]
            rpmGaugeView.bgColor = .blueGrey
            rpmGaugeView.bgAlpha = 0.20
            rpmGaugeView.shadowRadius = 1
            rpmGaugeView.rotate = 45.0
            rpmGaugeView.maxValue = 8000
            rpmGaugeView.lineWidth = 22
            self.rpmGaugeView.animateRate(0.5, newValue: CGFloat(0)) { _ in }
        }
    }
```

Kod 6.14: Klasa `MainDashboardCollectionViewCell` ćelije kontrolne ploče

U klasi `MainDashboardCollectionViewCell` unutar metode `awakeFromNib()` koja se poziva pri inicijaliziranju ćelije kreira se *view model* objekt, pozivaju se metode za vizualnu konfiguraciju svih elemenata ćelije te se pojedinačno definira prikaz proslijeđenih podataka od strane *view modela* putem anonimnih metoda na elemente ćelije. Pri kreiranju *view model-a* u konstruktoru se inicijalizira objekt klase `VehicleOBD`, koja sadrži metode za dohvaćanje podataka vozila i definira se protokol objekta čija je uloga da svaki novo dohvaćeni podatak proslijedi natrag u `view model`. Nakon što je objekt kreiran, poziva se metoda `start_read_data()` koja pokreće komunikaciju s OBD-II uređajem.

```
class MainDashboardCellViewModel: BaseViewModel {

    var vehicleData: VehicleOBD!
    var cmMotion: CMMotion?
    var weather: WeatherModel?
    var onGotWeather: ((String, String, String) -> Void)?
    var onGotSpeed: ((Int) -> Void)?
    var onGotRPM: ((Int) -> Void)?
    var onGotCoolantTemp: ((String) -> Void)?
    var onGotGForce: ((String) -> Void)?
    var onAdressChanged: ((String) -> Void)?
    var onCompasUpdated: ((CGFloat) -> Void)?

    override init() {
        super.init()

        vehicleData = VehicleOBD()
```

```
    vehicleData.delegate = self
    vehicleData.start_read_data()

    cmMotion = CMMotion()
    cmMotion?.delegate = self
    cmMotion?.start()

    setupCallbacks()
}

func getWeatherIcon() -> String {
    guard let icon = weather?.weather?[0].icon else { return "10d" }

    return icon
}

private func getTemperatureWeather() -> String {
    let temp = getTemperature()
    let tempWeath = "\(Int(temp)) °C"

    return tempWeath
}

private func getTemperature() -> Double {
    guard let calvinTemp = weather?.main?.temp else { return 0.0 }

    let celsiusTemp = calvinTemp - 273.15

    return celsiusTemp.rounded()
}

private func getWeatherMain() -> String {
    guard let name = weather?.weather?[0].main else { return ""}

    return name
}

func getCurrentWeather(location: CLLocation) {
    WeatherServiceManager.now(lat: location.coordinate.latitude, lon:
           location.coordinate.longitude) { [weak self] (response) in
        guard let self = self else { return }
        switch response {
        case .success(let data):
            self.weather = data
            self.onGotWeather?(self.getTemperatureWeather(),
                  self.getWeatherMain(), self.getWeatherIcon())
        case .failure(let error):
            self.onError?(error.error.message)
        }
    }
}
```

```
    func setupCallbacks() {
        CMLocation.shared.onSpeedValueChanged = { [weak self] speedValue in
            guard let self = self else { return }

            self.onGotSpeed?(speedValue)
        }

        CMLocation.shared.onAdressChanged = { [weak self] address in
            guard let self = self else { return }

            self.onAdressChanged?(address)
        }

        CMLocation.shared.onLocationDidUpdated = { [weak self] location in
            guard let self = self else { return }
            self.getCurrentWeather(location: location)
        }

        CMLocation.shared.onHeadingUpdated = { [weak self] heading in
            guard let self = self else { return }
            //To radians
            let angle = (CGFloat(heading) * .pi) / 180
            self.onCompasUpdated?(angle)
        }
    }
}

extension MainDashboardCellViewModel: VehicleOBDCommunicationDelegate {
    func didReadSpeed(_ value: Int) {

        //self.onGotSpeed?(value)
    }

    func didReadRPM(_ value: Int) {
        self.onGotRPM?(value)
    }

    func didReadCoolantTemp(_ value: Int) {
        let string = "\(value)°C"
        self.onGotCoolantTemp?(string)
    }

    func didGotFuelConsumption(_ value: Double) {
        let string = String(format: "%.1f", value)
        self.onGotFuelConsumption?(string)
    }
}

extension MainDashboardCellViewModel: CMMotionDelegate {
    func didUpdateGForce(_ value: Double) {
```

```
            let string = String(format: "%.2f", value)
            self.onGotGForce?(string)
        }
    }
```

Kod 6.15: *View model* ćelije kontrolne ploče

Metode protokola očekuju podatke o brzini kretanja, broju okretaja motora, temperaturi rashladne tekućine motora i podatak o trenutnoj potrošnji goriva. Kada se metoda protokola pozove unutar `VehicleOBD` objekta, u *view model-u* se izvršava metoda protokola u kojoj se obrađuju informacije i izvršavanjem anonimnih metoda prosljeđuju se obrađeni podaci u klasu `MainDashboardCollectionViewCell`, koja nove informacije iz vozila prikazuje na elementima.

Podatak brzine kretanja koji se prikazuje korisniku nije vrijednost dobivena iz OBD-II uređaja u vozilu, nego vrijednost koja se dobiva putem anonimne funkcije statički definiranog objekta klase `CMLocation`. Na taj način korisniku se prikazuju aktualnije vrijednosti kretanja vozila, iz razloga što se podaci iz OBD-II uređaja dohvaćaju u intervalu od jedne sekunde.

6.8.2. Lokacijski servisi

Praćenje lokacije i orijentacije korisnika odvija se u klasi `CMLocation` koja sadrži metode i protokole za očitavanje promjene lokacije korisnika, te vraća podataka na temelju korisnikove nove lokacije. Podaci o lokaciji i orijentaciji temelje se na podacima o geografskoj lokaciji, visini u odnosu na razinu mora i orijentaciji uređaja čiji se podaci dobivaju na temelju podataka prikupljenih iz raznih senzora iPhone uređaja, poput GPS „seznora“, magnetometra, barometra, žiroskopa, Wi-Fi i Bluetooth senzora. Tim podacima pristupa se pomoću Apple službene biblioteke CoreLocation.

Metode klase pozivaju se na statičkoj varijabli u kojoj je kreirana instanca objekta te klase. Na taj način je omogućeno korištenje iste instance tog objekta kroz čitavu klasu, što je elegantno rješenje za ažuriranje svih podataka kroz aplikaciju koji se temelje na lokaciji korisnika poput kompasa, prikaza trenutne ulice kretanja vozila, vremenske prognoze i odnosa lokacije korisnika i vozila na karti.

```
class CMLocation: NSObject, CLLocationManagerDelegate {
    static let shared = CMLocation()
    let locationManager: CLLocationManager
```

```
var lastLocation: CLLocation?
var addresName: String = "" {
    didSet{
        self.onAdressChanged?(addresName)
    }
}

//Callbacks
var onLocationDidUpdated: ((CLLocation) -> Void)?
var onSpeedValueChanged: ((Int) -> Void)?
var onAdressChanged: ((String) -> Void)?
var onHeadingUpdated: ((CLLocationDirection) -> Void)?

override init() {
    locationManager = CLLocationManager()
    locationManager.desiredAccuracy = kCLLocationAccuracyBest
    super.init()
    locationManager.delegate = self
}

func start() {
    locationManager.requestAlwaysAuthorization()

    if CLLocationManager.locationServicesEnabled() {
        locationManager.startUpdatingLocation()
        locationManager.startUpdatingHeading()
    }
}

func stop() {
    locationManager.stopUpdatingLocation()
    locationManager.stopUpdatingHeading()
}

func locationManager(_ manager: CLLocationManager, didUpdateLocations
           locations: [CLLocation]) {
    guard let currentLocation = manager.location else { return }
    onLocationDidUpdated?(currentLocation)

    var speed: CLLocationSpeed = CLLocationSpeed()
    speed = locationManager.location!.speed
    processSpeed(speed: speed)
    guard let mostRecentLocation = locations.last else {
        return
    }

    //Get street and city name based on location
    let geocoder = CLGeocoder()
    geocoder.reverseGeocodeLocation(mostRecentLocation) { (placemarks,
                  error) in
```

```
                guard let placemarks = placemarks, let placemark =
                   placemarks.first else { return }
                   if let city = placemark.locality, let thoroughfare =
                           placemark.thoroughfare {
                    self.addresName =  thoroughfare + ", " + (city as String)
                }
            }
        }

        func locationManager(_ manager: CLLocationManager, didUpdateHeading
                   newHeading: CLHeading) {

            if newHeading.headingAccuracy < 0 {
                return
            }

            // Get the heading (direction)
            let heading: CLLocationDirection = ((newHeading.trueHeading > 0) ?
                newHeading.trueHeading : newHeading.magneticHeading);

            onHeadingUpdated?(heading)
        }

        func locationManager(_ manager: CLLocationManager, didFailWithError
                   error: Error) {
            print(error)
            locationManager.stopUpdatingLocation()
        }

        func processSpeed(speed: CLLocationSpeed) {
            var speedValue: Int = Int(speed * 3.6)
            if speedValue < 0 {
                speedValue = 0
            }
            self.onSpeedValueChanged?(speedValue)
        }
    }
```

Kod 6.16: Klasa za praćenje lokacije korisnika

Praćenje lokacije korisnika pokreće se na samom paljenju aplikacije pozivanjem metode `start()` na statičkoj varijabli `shared`, što je u nastavku prikazano programskim kodom.

```
CMLocation.shared.start()
```

Putem `shared` varijable kroz aplikaciju definiraju se naredbe koje će se izvršavati pozivanjem anonimnih metoda svaki puta kada je očitana nova lokacija. Anonimne metode

definirane su na samom početku klase te svaka anonimna metoda predstavlja podatak koji je potreban za funkcionalnosti mobilne aplikacije. Putem anonimnih metoda vraćaju se podaci o trenutnoj lokaciji korisnika, vrijednosti brzine kretanja, o promjeni adrese ulice kretanja vozila te o promjeni smjera kretanja uređaja. Primjer Kod 6.16.

Protokol `CLLocationManagerDelegate` definiran od strane biblioteke CoreLocation. Sadrži metode putem kojih se prosljeđuju podaci u trenutku novo očitane lokacije i smjera kretanja. Te metode definirane su nazivom `locationManager` te se razlikuju po tipu metode koja je inicirala izvršavanje samog protokola. Naziv metode koja je inicirala izvršavanje protokola nalazi se kao opis varijable objekta koji vraća. U `CMLocation` klasi koriste se dvije vrste protokol metoda, ona koja javlja kada je promijenjena lokacija korisnika pod nazivom `didUpdateLocations` i ona koja javlja kada je promijenjen smjer kretanja korisnika pod nazivom `didUpdateHeading`.

Unutar `didUpdateLocations` metode prije obrade dobivenog podatka poziva se anonimna `onLocationDidUpdated()` metoda kojom se prosljeđuje najnovija vrijednost lokacije korisnika. Nakon toga se iz dobivenog objekta lokacije dohvaća podatak o trenutnoj brzini korisnika na temelju svih senzora uređaja. Podatak o brzini definiran je mjernom jedinicom milja po satu (engl. *miles per hour*) te ga je potrebno preračunati u kilometre po satu te zaokružiti vrijednost kao cijeli broj. Ta obrada podatka o brzini se odvija u metodi `processSpeed()` koja prima vrijednost brzine u prijeđenim miljama po satu. Nakon što su podaci uspješno obrađeni, poziva se anonimna metoda putem koje se obrađen podatak prosljeđuje u *view model* u kojem se poziva anonimna metoda za prikaz podatka na elementu zaslona.

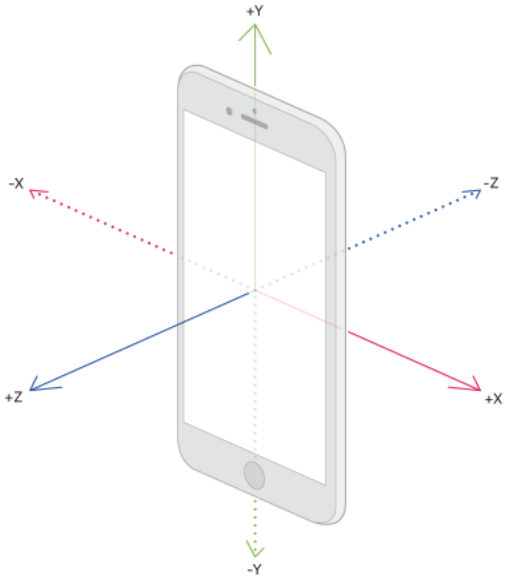
Uz brzinu, na temelju lokacije korisnike dohvaćaju se podaci o ulici i gradu u kojem se korisnik nalazi. Pomoću `CLGeocode` klase koja je dio CoreLocation biblioteke poziva se metoda koja na temelju proslijeđene lokacije uspostavlja vezu s geolokacijskim mrežnim servisom i korisniku dohvaća naziv ulice i grada. Ti podaci se u obliku niza znakova anonimnom metodom prosljeđuju u *view model* kako bi mogli biti prikazani na elementu zaslona.

Unutar *view model-a* na svaku promjenu vrijednosti lokacije putem `onLocationDidUpdated()` metode uspostavlja se veza s mrežnim servisom vremenske prognoze, koji na temelju lokacije korisnika vraća mobilnom uređaju podatke o trenutnoj vremenskoj prognozi.

Metodom `didUpdateHeading` prati se smjer kretanja korisnika na temelju podataka iz senzora. Unutar nje dobiven podatak smjera kretanja se obrađuje te se provjerava vrijednost podatka u odnosu na geografski sjeverni pol. Podatak koji predstavlja smjer orijentacije uređaja je izražen u stupnjevima. U slučaju ako je vrijednost 0 stupnjeva, znači da je uređaj orijentiran prema sjevernom polu, u slučaju ako je vrijednost 90 stupnjeva, uređaj je orijentiran prema istoku, ako je vrijednost 180 stupnjeva, uređaj je orijentiran prema jugu te ako je vrijednost 270 stupnjeva, uređaj je orijentiran prema zapadu. Ako je vrijednost podatka orijentacije u odnosu na sjeverni pol manja od 1, znači da orijentacija uređaja nije definirana. Tada se uzima vrijednost orijentacije uređaja u odnosu na magnetski definirani sjeverni pol. Nakon što je vrijednost orijentacije uređaja definirana, poziva se anonimna metoda kojom se vrijednost orijentacije prosljeđuje u *view model*. Tada se vrijednost iz stupnjeva pretvara u radijane te se ta vrijednost putem anonimne metode prosljeđuje u klasu koja komunicira s *view*-om, unutar koje se animira rotacija slike koja predstavlja „iglu“ kompasa za proslijeđenu vrijednost u radijanima.

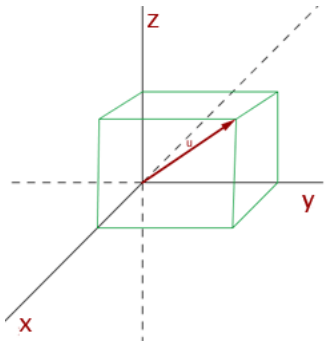
6.8.3. Senzori pokreta

Kako bi se izračunala gravitacijska sila djelovanja na automobil potrebno je prvo izračunati akceleraciju uređaja za vrijeme ubrzavanja ili deceleraciju naglog usporavanja automobila. Pošto se mobilni uređaj iPhone nalazi za vrijeme vožnje u automobilu taj podatak je najbolje dohvatiti iz senzora pokreta samog iPhone uređaja. Akcelerometar je senzor koji se nalazi u iPhone uređaju i mjeri akceleraciju pokreta uređaja u 3D prostornom koordinatnom sustavu.



Slika 6.12: 3D koordinatni sustav iPhone uređaja

Klasa `CMMotion` komunicira direktno s Swift CoreMotion bibliotekom koja na temelju podataka iz senzora akcelerometra, žiroskopa, pedometra, magnetometra i barometra dohvaća podatke koji opisuju kretanje uređaja kroz prostor. U `CMMotion` klasi nalaze se metode i protokoli pomoću kojih se na vrlo jednostavan način može dohvatiti trenutna akceleracija mobilnog uređaja u prostoru. Akceleracija je izražena vrijednostima po x, y i z osi te se zbog toga naziva prostorna akceleracija. Pomoću vrijednosti akceleracije na sve tri osi putem formule jednostavno je izračunati vrijednosti prostornog vektora.



Slika 6.13: Prikaz prostornog vektora u 3D koordinatnom sustavu

Vrijednost vektora prostorne dijagonale koji je na slici 6.13 izražen crvenom bojom, na x, y i z vrijednosti akceleracije je rezultantna sila koja djeluje na uređaj. Vrijednost vektora prostorne dijagonale računa se na sljedeći način.

$$\text{Magnitude} = \sqrt{x^2 + y^2 + z^2}$$

Formula 1. Formula rezultantne sile izražene vektorom prostorne dijagonale

Rezultat koji je u formuli (Formula 1) izražen kao *magnitude* (hrv. veličina) jednak je vrijednosti gravitacijske sile koja djeluje na uređaj za vrijeme vožnje. Mora se naglasiti kako ova funkcionalnost izračunava djelovanje sile na mobilni uređaj. Kako bi se dobila konačna sila djelovanja na automobil, potrebno je vrijednost dobivenu iz formule podijeliti s umnoškom mase automobila i gravitacijske sile.

U *view model*-u klase unutar koje je potrebna gravitacijska sila djelovanja na uređaj kreira se objekt klase `CMMotion` te se nad novo kreiranim objektom poziva metoda `start()` kojom se započinje mjerenje prostorne akceleracije uređaja.

```
class CMMotion: NSObject {
    var delegate: CMMotionDelegate?
    static let shared = CMMotion()
    let motionManager: CMMotionManager
    let motionActivityManager : CMMotionActivityManager
    var lastValue: CMAccelerometerData!
    var isCrashEvenetPresented = false
    var isMovingVehicle = false

    override init() {
        motionManager = CMMotionManager()
        motionActivityManager = CMMotionActivityManager()
    }

    func start() {
        if motionManager.isAccelerometerAvailable {
            motionManager.accelerometerUpdateInterval = 0.1
            motionManager.startAccelerometerUpdates(to: OperationQueue.main)
                        { (data, error) in
                guard let data = data else { return }
                self.calculateForce(data)
            }
        }

        if CMMotionActivityManager.isActivityAvailable() {
            motionActivityManager.startActivityUpdates(to:
                        OperationQueue.main) { (activity) in
                if (activity?.automotive)! {
                    self.isMovingVehicle = true
                }
                if (activity?.cycling)! {
```

```
                    self.isMovingVehicle = false
                }
                if (activity?.running)! {
                    self.isMovingVehicle = false
                }
                if (activity?.walking)! {
                    self.isMovingVehicle = false
                }
                if (activity?.stationary)! {
                    self.isMovingVehicle = false
                }
                if (activity?.unknown)! {
                    self.isMovingVehicle = false
                }
            }
        }
    }

    func stop() {
        motionManager.stopDeviceMotionUpdates()
    }

    func calculateForce(_ value: CMAccelerometerData) {
        let x = value.acceleration.x
        let y = value.acceleration.y
        let z = value.acceleration.z

        let gForce = sqrt(x*x + y*y + z*z)

        if gForce > 2 && !isCrashEvenetPresented && isMovingVehicle{
            delegate?.didCrashOccured(gForce)
            isCrashEvenetPresented = true
        } else {
            isCrashEvenetPresented = false
        }
        delegate?.didUpdateGForce(gForce)
    }
}
```

Kod 6.17: CMMotion klasa za komunikaciju sa senzorom akceleracije

Unutar metode `start()` definira se interval ažuriranja vrijednosti sa senzora. Metodom `startAccelerometerUpdates()` pokreće se ažuriranje vrijednosti sa senzora te se prilikom svake novo dobivene vrijednosti pokreće metoda `calculateForce()`, unutar koje se iščitavaju vrijednosti akceleracije na svakoj od osi te se formulom na temelju dobivenih podataka računa vrijednost gravitacijske sile. Nakon što je gravitacijska sila izračunata, poziva se protokol metoda putem koje se prosljeđuje novo dobivena vrijednost sile u *view model*.

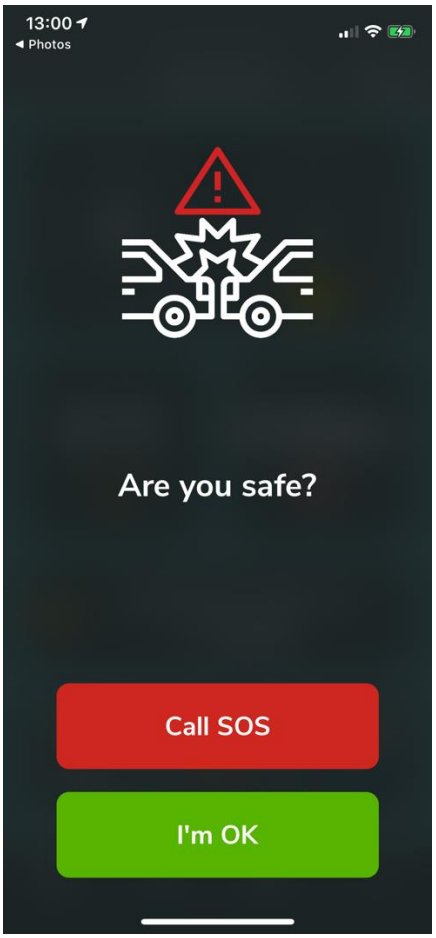
```
protocol CMMotionDelegate: class {
    func didUpdateGForce(_ value: Double)
    func didCrashOccured(_ value: Double)
}
```

Kôd 6.18 `CMMotionDelegate` protokol klase `CMMotion`

6.8.4. Prepoznavanje sudara

Nakon računanja sile koja djeluje na uređaj prilikom vožnje provjerava se je li trenutna gravitacijska sila djelovanja veća od 2, te u slučaju ako je korisnik u vozilu poziva se protokol metoda `didCrashOccured()` unutar koje se prosljeđuje vrijednost sile djelovanja. Radi primjera, vozilu koje se kreće brzinom 100 km/h i naglo udari u drugo vozilo sa djelovanjem sile na zaustavljanje od 2G, potrebno mu je 1.6 sekundi kako bi se zaustavio.

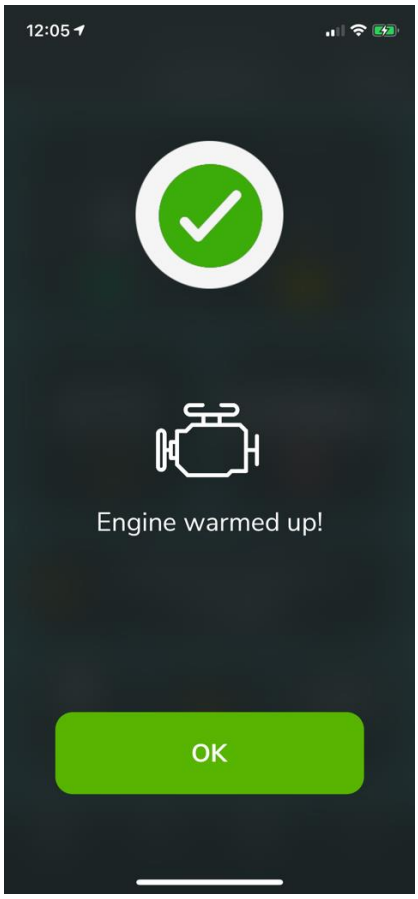
Pozivanjem te protokol metode preko cijelog zaslona prikazuje se zaslon koji nudi mogućnost uspostavljanja telefonskog poziva sa službom hitne pomoći.



Slika 6.14: Zaslon upozorenja u slučaju sudara

6.8.5. Zagrijanost motora

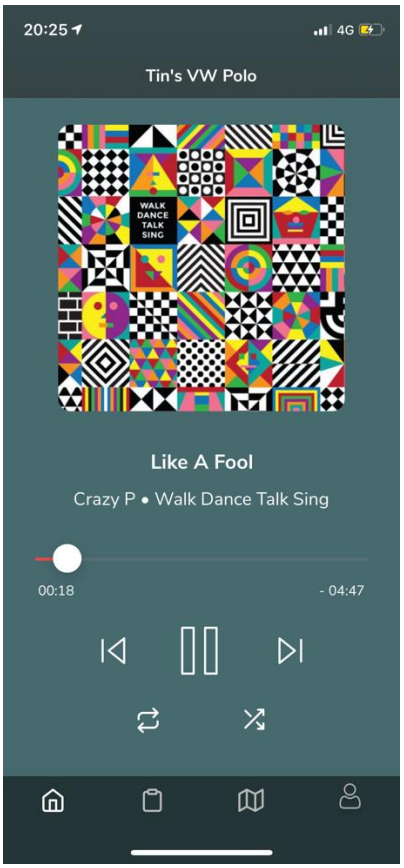
Prilikom dohvaćanja podatka o temperaturi rashladne tekućine motora putem lokalnog servera OBD-II uređaja provjerava se da li je temperatura rashladne tekućine iznad 65º stupnjeva Celzijusa. U trenutku kada temperatura prijeđe tu vrijednost prikazuje se zaslon preko cijelog ekrana, putem kojeg se korisnika obavještava da je temperatura motora dovoljno visoka te da je vozilo spremno za normalnu vožnju. Ekran se korisniku prikazuje 10 sekundi te se nakon toga sam ugasi. Korisnik ima opciju da pritiskom na gumb OK sam ugasi ekran i nastavi normalno koristiti aplikaciju.



Slika 6.15: Zaslon obavijesti o zagrijanosti motora

6.8.6. Glazbeni sustav

Funkcionalnost *media* sustava unutar mobilne aplikacije omogućuje korisniku da kontrolira glazbu koja trenutno svira na mobilnom uređaju bez izlaska iz same aplikacije. Ova funkcionalnost pripada srednje bitnim funkcionalnostima aplikacije te se nalazi u prvoj kolekcijskoj ćeliji s desne strane od kontrolne ploče. Na taj način korisnik za vrijeme vožnje jednim pokretom prsta može pristupiti glazbenom sustavu i vrlo jednostavno kontrolirati glazbu na svom mobilnom uređaju, bez ometanja za vrijeme vožnje.



Slika 6.16: Glazbeni sustav

Pomoću Swift biblioteke pod nazivom MediaPlayer pristupa se *media* centru na razini operacijskog sustava te su dostupne sve akcije potrebne za kontroliranje glazbe, poput puštanja i zaustavljanja glazbe, prebacivanja na prijašnju i sljedeću pjesmu, uključivanje ponavljanja trenutne liste za reprodukciju i miješanje redoslijeda liste za reprodukciju.

Tim akcijama pristupa se putem objekta `MPMusicPlayerController.systemMusicPlayer` kreiranog na razini klase za kontrolu kolekcijske ćelije u kojoj se glazbeni *player* nalazi. Objekt se kreira odabirom tipa sistemskog glazbenog sustava na objektu `MPMusicPlayerController` biblioteke MediaPlayer. U `awakeFromNib()` metodi konfigurira se izgled elemenata ćelije te se definiraju notifikacije, pomoću kojih *media* centar obavještava klasu da je došlo do promjene u reprodukciji glazbe te da treba izvršiti promjenu na elementima zaslona.

```
NotificationCenter.default.addObserver(self,selector:#selector(updateNowPlayingInfo),name:
        NSNotification.Name.MPMusicPlayerControllerNowPlayingItemDidChange,
                object: nil)
```

Naziv notifikacije koja se koristi za praćenje promjena sistemskog *media* centra je `MPMusicPlayerControllerNowPlayingItemDidChange`. Dodavanjem te notifikacije definirano je da se na svaku dolazeću notifikaciju pozove metoda `updateNowPlayingInfo()`, koja poziva metodu `timeFired()`, koja izvršava promjene na elementima zaslona u intervalu od 1/10 sekunde.

```
@objc func timerFired(_ :AnyObject) {
   //Ensure the track exists before pulling the info
        if let currentTrack = MPMusicPlayerController.systemMusicPlayer.nowPlayingItem {

                let trackName = currentTrack.title ?? "Unknown Track"
                let trackArtist = currentTrack.artist ?? "Unknown Artist"
                let trackAlbum = currentTrack.albumTitle ?? "Unknow Album"
                labelTitle.text = "\(trackName)"

                if trackAlbum != "" {
                    artistAlbumLabel.text = "\(trackArtist) • \(trackAlbum)"
                } else {
                     artistAlbumLabel.text = "\(trackArtist)"
                }

                let albumImage = currentTrack.artwork?.image(at: imageAlbum.frame.size)
                if albumImage != nil {
                              imageAlbum.image = albumImage
                } else {
                    imageAlbum.image = #imageLiteral(resourceName: "no-artwork-image")
                }

                let trackDuration = currentTrack.playbackDuration

                let trackElapsed = mp.currentPlaybackTime
                if trackElapsed.isNaN { return }

                labelElapsed.text = formatTimeToString(seconds: Int(trackElapsed))
                let trackRemaining = Int(trackDuration) - Int(trackElapsed)
                let stringRemaining = formatTimeToString(seconds: Int(trackRemaining))
                labelRemaining.text = "- \(stringRemaining)"

                //Maximum value of slider
                sliderTime.maximumValue = Float(trackDuration)

                //Setting current track time value to slidr
                sliderTime.value = Float(trackElapsed)
        }
    }
```

Kod 6.18: Metoda koja osvježava podatke na elementima glazbenog playera

Unutar metode prvo se provjerava postoji li pjesma u sistemskom glazbenom *playeru* koja trenutno svira, na način da se provjerava vrijednost podatka na objektu `MPMusicPlayerController.systemMusicPlayer.nowPlayingItem`.

U slučaju ako podatak ne postoji, ne izvršavaju se naredbe za osvježavanje informacija, a ako postoji, podatak o trenutnoj pjesmi pohranjuje se u varijablu `currentTrack` te se na njemu nalaze sve potrebne informacije o pjesmi poput naziva pjesme, izvođača i naziva albuma, trenutno vrijeme reproduciranja i ukupna vrijednost trajanja pjesme te slika koja predstavlja album pjesme. Informacije o pjesmi se u nastavku metode obrađuju i nakraju unutar same metode prikazuju na elemente glazbenog *playera.*

U nastavku su opisane metode objekta `MPMusicPlayerController.systemMusicPlayer` za kontrolu reproduciranja glazbe:

Pokretanje-zaustavljanje pjesme

Za pokretanje i zaustavljanje pjesme pritiskom na gumb pozivaju se metode `play()` i `pause()` na objektu MP. Provjerava se trenutno stanje sustava te ovisno da li sustav trenutno svira ili ne izvršava se jedna od metoda te se zadaje slika ovisno o akciji koja je izvršena.

```
switch mp.playbackState {
     case .playing:
         mp.pause()
         playBtn.setImage(#imageLiteral(resourceName: "ic-play"), for:
                                .normal)
     case .paused:
         mp.play()
         playBtn.setImage(#imageLiteral(resourceName: "ic-pause"), for:
                                .normal)
     default:
         return
    }
```

Kod 6.19: Pokretanje-zaustavljanje pjesme

Prebacivanje između prethodne i sljedeće pjesme

Pritiskom na gumb za odlazak na sljedeću pjesmu poziva se metode `skipToNextItem()` na objektu `mp`, koja prebacuje sljedeću pjesmu na reprodukcijskoj listi.

Pritiskom na gumb za prethodnu pjesmu, u slučaju ako trenutna pjesma traje manje od 3 sekunde, poziva se metoda `skipToPreviousItem()` koja pušta prethodnu pjesmu. U slučaju ako pjesma traje duže od 3 sekunde, tada se izvršava metoda `skipToBeginning()` koja trenutno pjesmu pušta od samog početka.

```
if mp.currentPlaybackTime < 3 {
      mp.skipToPreviousItem()
} else {
      mp.skipToBeginning()
}
```

Kod 6.20: Prebacivanje na prethodnu pjesmu – vračanje na početak trenutne pjesme

Ponavljanje trenutne liste za reprodukciju

Pritiskom na gumb za ponavljanje trenutne liste reproduciranja provjerava se je li ponavljanje uključeno te se ovisno o stanju zadaje vrijednost koja definira ponavljanje trenutne liste reproduciranja.

```
switch mp.repeatMode {
     case .all:
         mp.repeatMode = .none
         repeatBtn.setImage(#imageLiteral(resourceName: "ic-repeat"), for:
                              .normal) // Normal image version
     case .none:
          mp.repeatMode = .all
          repeatBtn.setImage(#imageLiteral(resourceName: "ic-repeat-
                              selected"), for: .normal) // Selected image
                                                          version
        default:
            return
        }
```

Kod 6.21: Paljenje -gašenje ponavljanja trenutne liste reproduciranja

Miješanje redoslijeda trenutne liste za reprodukciju

Pritiskom na gumb za miješanje redoslijeda pjesama u listi za reproduciranje, provjerava se da li je miješanje liste već uključeno. Ovisno o trenutnoj vrijednosti miješanje se isključuje ili se zadaje tip miješanja.

```
Switch mp.shuffleMode {
     case .songs:
         mp.shuffleMode = .off
         shuffleBtn.setImage(#imageLiteral(resourceName: „ic-shuffle“), for:
               .normal) // Normal image version
     case .off:
```

```
            mp.shuffleMode = .songs
            shuffleBtn.setImage(#imageLiteral(resourceName: „ic-shuffle-
                  selected"), for: .normal) // Selected image version
        default:
            return
        }
```

Kod 6.22: Paljenje -gašenje miješanja liste reproduciranja

6.8.7. Vremenska prognoza za sljedeća 4 dana

Vremenska prognoza za sljedeća četiri dana na trenutnoj lokaciji korisnika je funkcionalnost koja je kategorizirana kao srednje bitna funkcionalnost. U kolekciji uz ćelije kontrolne ploče koja se nalazi u sredini zaslona i ćelije glazbenog *playera* s lijeve strane kontrolne ploče nalazi se ćelija koja prikazuje informacije o vremenskoj prognozi.

Tabličnim prikazom prikazuju se informacije o vremenskoj prognozi za svaka tri sata, za narednih 4 dana. Informacije o vremenskoj prognozi jednog dana grupirane su u sekciju tablice te je svaki naslov sekcije definiran nazivom dan, a za koji se podaci u toj sekciji odnose. Pojedina informacija o vremenskoj prognozi predstavljena je temperaturom, vremenom za koje doba dana se taj podatak odnosi, slikom vremena i opisom vremena.

U `awakeFromNib()` metodi ćelije za prikaz vremenske prognoze definirano je da se na svaku promjenu lokacije, kao rezultat anonimne metode `onLocationDidUpdated` klase `CMLocation`, pozove metoda *view model*-a ćelije pod nazivom `getForecast()` koja kao ulazni podatak prima lokaciju korisnika.

```
func getForecast(location: CLLocation?) {
        if let newLocation: CLLocation = location {
            forecastLocation = newLocation
        }
        guard let forecastLocation = forecastLocation else { return }
        WeatherServiceManager.forecast(lat:
               forecastLocation.coordinate.latitude, lon:
               forecastLocation.coordinate.longitude) { [weak self]
                                                              (response) in
            switch response {
            case .success(let data):
                self?.forecast = data
                self?.sortForecastsByDay()
            case .failure(let error):
                self?.onError?(error.error.message)
            }
        }}
```

Kod 6.23: Metoda za dohvaćanje vremenske prognoze za sljedeća 4 dana

Nakon što je komunikacija s mrežnim servisom uspješno uspostavljena, podaci se spremaju na razinu *view modela* te se metodom `sortForecastsByDay()` dobiveni podaci sortiraju po danima za koje se odnose. Nakon što su uspješno sortirani, poziva se anonimna metoda *view model*-a `onGotForecast()` koja javlja klasi ćelije da prikaže podatke korisniku.

Slika 6.17: Zaslon vremenske prognoze za sljedeća 4 dana

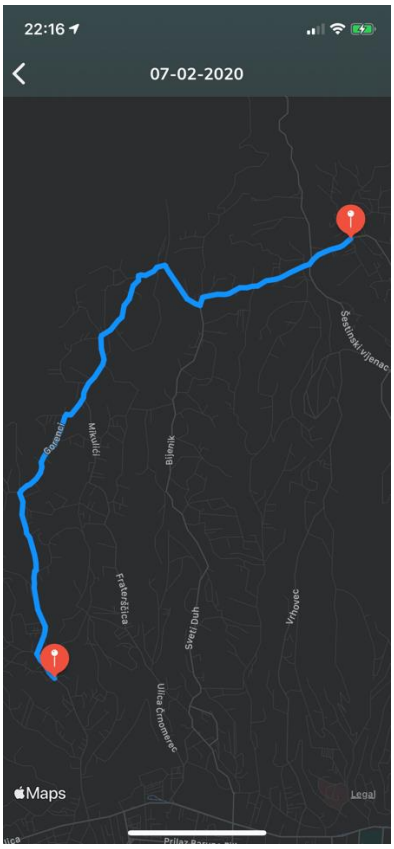
6.8.8. Praćenje putovanja

Mobilna aplikacija korisniku pruža prikaz svih putovanja koje korisnik napravio. Putovanja su zabilježena od strane OBD-II uređaja te su pohranjena na *cloud* tvrtke AutoPi. Putovanjima na *cloudu* pristupa se putem REST API poziv na mrežni servis tvrtke AutoPi. Uređaj na temelju događaja motora definira kada je putovanje započelo i kada je završilo. Prilikom paljenja motora automobila, uređaj očitava da se motor automobila upalio te na temelju tog događaja sprema lokaciju. U trenutku kada korisnik započne vožnju automobilom, uređaj sprema vrijeme početka vožnje. Nakon što se motor automobila ugasi, uređaj sprema lokaciju na kojoj se taj događaj dogodio i računa proteklo vrijeme od

zabilježenog vremena početka vožnje, te se onda informacije o putovanju s uređaja šalje na *cloud* tvrtke AutoPi.

Slika 6.18: Zaslon prikaza popisa putovanja

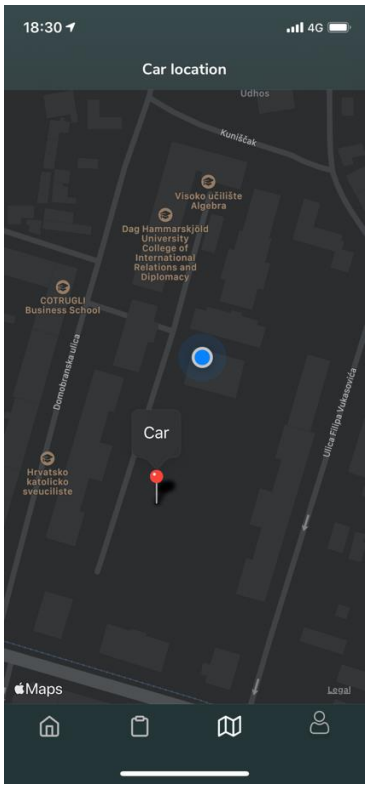
Kako bi se uspješno dohvatili podaci, potreban je *token* za autentifikaciju korisnika na mrežnom servisu tvrtke AutoPi, koji se sprema prilikom prijavljivanja korisnika u aplikaciju. Token se šalje u sklopu *authenticaion header-a* mrežnog zahtjeva. Nakon što mobilna aplikacija dohvati podatke o putovanjima, prikazuje podatke tabličnim prikazom na zaslonu. Prikazuje se početna adresa putovanja, konačna adresa putovanja, duljina i vrijeme putovanja. Dugačkim pritiskom na ćeliju putovanja prikazuje se datum putovanja koji cijelo vrijeme vidljiv u pozadini ćelije. Također pritiskom na ćeliju putovanja otvara se novi zaslon s prikazom karte na kojoj je prikazana početna lokacija i krajnja lokacija putovanja, uz rutu kojom je korisnik putovao.



Slika 6.19: Zaslon prikaza putovanja na karti

6.8.9. Lokacija automobila na karti

Za vrijeme i nakon vožnje OBD-II uređaj komunicira lokaciju kretanja vozila s *cloud*-om te je putem mrežnog servisa moguće dohvatiti tu lokaciju i prikazati ju korisniku na karti, u sklopu mobilne aplikacije. Prednost samog uređaja je to što sadrži bateriju koja napaja uređaj do 45 minuta nakon gašenja automobila te za to vrijeme također šalje svoju lokaciju na *cloud*. Mobilna aplikacija svakih 5 sekundi dohvaća zadnju poznatu lokaciju vozila te ju korisniku prikazuje na karti.



Slika 6.20: Zaslon prikaza lokacije vozila na karti

6.8.10. Statistika vožnje

Funkcionalnost statistike vožnje korisnika definirana je informacijama spremljenim na mrežni servis razvijen u sklopu ovog rada. Mobilna aplikacija komunicira s mrežnim servisom te dohvaća podatke o ukupnom broju kilometara koje korisnik napravio, koji je na mobilnoj aplikaciji prikazan kao postotak proputovanih kilometara, u odnosu na broj kilometara opsega zemlje. Također dohvaća podatke o prosječnoj brzini kretanja vozila te prosječno provedenom vremenu u autu, za vrijeme vožnje. Te informacije se obrađuju na način da za vrijeme vožnje mobilna aplikacija u intervalu od 5 sekundi na mrežni servis ovog rada šalje podatke o brzini te na temelju putovanja korisnika šalje podatke o vremenu provedenom u autu i količini kilometara koju je korisnik prošao automobilom. Te informacije se prikazuju na zaslonu predstavljenim zadnjom karticom kartične kontrole.

Slika 6.21: Zaslon prikaza podataka statistike vožnje

Na tom zaslonu prikazuju se i osnovne informacije o korisniku aplikacije - registracija vozila, informacija kada je OBD-II uređaj zadnji puta komunicirao s *cloud-om*, verzija zakrpe OBD-II uređaja i vremenska zona u kojoj se korisnik nalazi.

7. Mrežna aplikacija

U ovom poglavlju bit će objašnjen razvoj mrežne aplikacije ovog rada. Mobilna aplikacija komunicira s ovim REST API sučeljem za vrijeme vožnje te na njega šalje informacije o brzini kretanja automobila i većim silama koje su djelovale na automobil. Na temelju putovanja zabilježenih putem OBD-II uređaja sprema informacije o duljini putovanja i provedenom vremenu za vrijeme vožnje. Na temelju tih podataka korisnik na mobilnoj aplikaciji ima uvid u statistike o svojoj vožnji

7.1. Tehnologija

Mrežna aplikacija ovog rada razvijena u Node.Js[25] okruženju za pokretanje JavaScript programskog koda. Platforma Node.Js omogućuje pisanje brzih skripti i lak način za izgradnju poslužiteljskih mrežnih aplikacija, pomoću programskog jezika JavaScript. Prednosti ove platforme su velika brzina zahvaljujući V8 JavaScript engine-u. Engine je razvila tvrtka Google[26] u sklopu projekta „Chromium“[27] čiji je zadatak bio razvijanje Google Chrome web preglednika. Također sljedeća prednost je princip rada na samo jednoj niti (engl. *Thread*), u slučaju izvršavanja neke spore I/O operacije, kao što je obrada velikog bloka podataka, program ne čeka, nego kreće obavljati sljedeću funkciju te kada je prijašnja operacija završena poziva se povratna funkcija putem koje se procesuira rezultat. Uz brzinu i korištenje jedne niti prednost je i ta što se može koristiti na svim platformama (macOS, Windows ili Linux).

7.2. DDD metodologija razvoja

Arhitektura koja je korištenja prilikom razvoja mrežne aplikacije je *domain-drive design,* skraćeno DDD arhitektura. Ideja DDD arhitekture je da se razvoj aplikacije bazira na domeni koja predstavlja sferu funkcionalnosti koje su definirane svrhom i logikom rješavanja problema, organizacije ili pojedinca s kojim se suočavaju. Na samom početku razvijanja aplikacije DDD arhitekturom bitno je definirati što je domena te aplikacije kako

[25] https://nodejs.org/en/, Node.JS okruženje
[26] https://en.wikipedia.org/wiki/Google, Tvrtka Google
[27] https://www.chromium.org/, Chromium projekt

bi daljnje planiranje razvoja imalo smisla. Nakon definiranja domene potrebno je grupirati sve informacije i procese koji opisuju problem organizacije i pojedinca. Te informacije i procesi nazivaju se domenskim modelom te se oko njega bazira razvoj aplikacije. Laički rečeno, domenski model predstavljaju sve prikupljene informacije, procesi i znanja koje pridonose rješavanju problema razvojem aplikacije. Domenski model mora biti komuniciran jednostavnim i razumljivim jezikom koji će biti razumljivi tehničkom timu i timu koji predstavlja organizaciju ili pojedinca čiji se problem rješava. Na taj način smanjuje se šansa za krivo komunicirane probleme i potrebna rješenja tih problema.

Na čelu tehničkog tima je osoba koja se naziva domenski ekspert te ona planira i kontrolira razvoj aplikacije. To je osoba koja je potpuno upoznata s domenom i domenskim modelom projekta. Kod razvoja velikih i kompleksnih mrežnih aplikacija mnoge tvrtke angažiraju vanjske konzultante kao domenske eksperte koji planiraju i kontroliraju razvoj aplikacije.

Prakticiranje DDD-a podrazumijeva praćenje 3 načelna principa prilikom razvoja aplikacije:

- Prilikom razvoja aplikacije u fokusu mora biti domena i logika domene.
- Informacije koje predstavljaju domenski model moraju biti kvalitetno definirane.
- Domenski ekspert konstantno mora surađivati s tehničkim timom kako bi domenski model bio u potpunosti implementiran u aplikaciju

U kontekstu razvoja mrežne aplikacije ovog rada domena je bila rješavanje problema oko prikupljanje podataka koji su potrebni za vrijeme vožnje kako bi se u konačnici dobili podaci o statistici vožnje korisnika. Model domene pri razvijanju aplikacije definiran je sljedećim konceptima:

- *Values*
- *Service*
- *Repository*
- *Controller*

Values predstavlja sve objekte podataka potrebnih za rješavanja problema i konačno dobivanje rezultata statistike o vožnji.

Ovim konceptom definirani su podaci:

- `SpeedTracker` – objekt koji definira informaciju brzine vožnje.
- `Travel` – objekt koji opisuje informaciju putovanja kroz informacije o vremenu putovanja i količini puta.
- `ForcePeak` – objekt koji predstavlja informacije većih sila koje su djelovale na vozilo na određenim koordinatama.
- `GeneralData` – objekt koji predstavlja rezultantne informacije statistike vožnje

```
export class GeneralData {
      private averageSpeed: number;
      private averageTimeSpentInCar: number;
      private totalDistanceCovered: number;

      constructor(averageSpeed: number,
      averageTimeSpentInCar: number, totalDistanceCovered:
      number) {
            this.averageSpeed = averageSpeed;
            this.averageTimeSpentInCar =
            averageTimeSpentInCar;
            this.totalDistanceCovered = totalDistanceCovered;
      }

      public getAverageSpeed(): number {
            return this.averageSpeed;
      }

      public getAverageTimeSpentInCar(): number {
            return this.averageTimeSpentInCar;
      }

      public getTotalDistanceCovered(): number {
            return this.totalDistanceCovered;
      }
}
```

Kod 7.1: GeneralData objekt

Service predstavlja protokol metodu koja je potrebna da bi se dobili krajnji rezultat podaci objekta `GeneralData`.

```
export interface GeneralDataService {
    aggregateData(): Promise<GeneralData>;
}
```

Kod 7.2: Protokol GeneralDataService

Repository definira protokol metode koje su potrebne za spremanje i dohvaćanje podataka iz baze podataka mrežne aplikacije.

```
export interface TravelRepository {
    createTravel(distanceCovered: number, durationOfTravel:
                    number): Promise<Travel>;
    getTravels(query: LimitedQuery): Promise<Travel[]>;
}
```

Kod 7.3: Protokol TravelRepository

Controller je dio domenskog modela unutar kojeg se metode protokola koriste za dohvaćanje podataka korisniku i spremanje podataka poslanih od strane korisnika.

```
export class TravelController {
    private repository: TravelRepository;
    private formatter: TravelFormatter;
    private validator: TravelValidator;

    constructor(
        @Inject("TravelRepository") repository: TravelRepository,
        formatter: TravelFormatter,
        validator: TravelValidator
    ) {
        this.repository = repository;
        this.formatter = formatter;
        this.validator = validator;
    }

    public async createTravel(requestPayload: RequestPayload) {
        // Extract data
        let {
            distanceCovered,
            durationOfTravel
```

```
        } = requestPayload.getPayload();

        // Validate input
        const validationResult = await this.validator.validate({
            distanceCovered,
            durationOfTravel
        });

        if (!validationResult.isValid()) {
            return new ValidationErrorResponse("You have validation errors",
                                validationResult.getValidationErrors());
        }

        // Execute action
        const model = await this.repository.createTravel(
            distanceCovered,
            durationOfTravel
        );
        return new JSONResponse(200, this.formatter.format(model));
    }
    public async getTravels(requestPayload: RequestPayload) {
        // Extract data
        const limit = parseInt(requestPayload.getFromQuery("limit",
                        LimitedQuery.DEFAULT_LIMIT), 10);
        const offset = parseInt(requestPayload.getFromQuery("offset",
                        LimitedQuery.DEFAULT_OFFSET), 10);
        // Read from domain
        const results = await this.repository.getTravels(new LimitedQuery(limit, offset));
        return new JSONResponse(200, results.map(this.formatter.format));
    }
}
```

Kod 7.4: TravelController domenski model

7.3. Putanje mrežne aplikacije

Rute za slanje i dohvaćanje podataka definirane su klasi `routes.ts`. Unutar te klase definirane su točne putanje za svaki tip podatka iz domenskog modela. Uz definiranu putanju svakog modela zadani su ruteri koji će biti korišteni na tim putanjama.

```
export default (app: Application) => {
    app.use('/travel', getTravelRouter());
    app.use('/speed-tracker', getSpeedTrackerRouter());
    app.use('/force-peak', getForcePeakRouter());
    app.use('/general-data', getGeneralDataRouter());
}
```

Kod 7.5: Definiranje putanja i rutera za putanje

Zadavanjem rutera na svakoj putanji definira se koje http metode transfera se koriste nad tom putanjom. Na mrežnoj aplikaciji ovog rada koriste se GET i POST metode za dohvaćanje i slanje podataka na svim putanjama osim na putanji /general-data koja koristi samo GET metodu za dohvaćanje statistike vožnje.

Putanje:

- „/travel“ – Putanja za slanje i dohvaćanje podataka vremena i količine prijeđenih kilometara putovanja.
- „/speed-tracker“ – Putanja za slanje i dohvaćanje podatka brzine vožnje
- „/force-peak“ – Putanja za slanje i dohvaćanje podatka djelovanja sile na vozilo na određenoj lokaciji
- „/general-dana“ – Putanja za dohvaćanje podatka statistike vožnje

8. Testiranje

Testiranje mobilne aplikacije primarno se provodilo na vozilu Volkswagen Polo 1.2 TDI godine proizvodnje 2011. Sve funkcionalnosti u potpunosti su testirane na tom vozilu, od strane autora ovog rada. Kako bi se u potpunosti uspješno testirale sve funkcionalnosti mobilne aplikacije OBD-II uređaj je bio povezan na internet mobilnom SIM karticom.

Testiranje je provedeno u dvije faze. U prvoj fazi testirao se dohvat podataka s OBD-II uređaja putem lokalnog servera. U toj fazi testiranja definirano je da dohvat podataka putem lokalnih mrežnih zahtjeva nije najbolje rješenje te se u drugom koraku testiranja ove faze testiralo spajanje mobilne aplikacije putem OBD biblioteka putem Bluetooth tehnologije. Tvrtka Auto.Pi omogućila je korištenje takvog tipa spajanja mjesec dana prije početka izrade ovog rada, s glavnom podrškom za najnoviju seriju OBD-II modula u njihovoj ponudi. Pošto se u ovom radu koristi prva serija modula, cijeli koncept još nije potpuno stabilan te su se pojavili problemi oko prepoznavanja i spajanja na sam modul. U ovoj fazi u svrhu realizacije rada odlučeno je da će se dohvat podataka nastaviti odvijati putem lokalne mreže, a informacija o brzini vozila očitavat će se putem senzora iPhone uređaja. Ostali podaci s kontrolne ploče koji se dohvaćaju s modula poput temperature vozila i potrošnje goriva, ne zahtijevaju kratak interval osvježavanja novim informacijama osim broja okretaja motora. Zaključak je bio da dohvaćanje većine podataka u intervalu od 5 sekundi toliko ne utječe na korisnika, osim s informacijom o RPM-u.

U ovoj fazi definirani su i dostupni senzori vozila na kojem se odvijalo glavno testiranje te se pokazalo kako to vozilo nema senzor koji prati potrošnju goriva. Dohvaćanje tog podatka neovisno o tome implementirano je u rad same mobilne aplikacije. Taj podatak potrebno je samo prikazati na kontrolnoj ploči mobilne aplikacije u slučaju ako se koristi automobil u kojem se taj senzor nalazi.

U sljedećoj fazi testiranja testirale su se kompletne funkcionalnosti mobilne aplikacije i njezino ponašanje za vrijeme vožnje. Također se testirao rad mrežne aplikacije, točnije, slanja i primanja podataka putem Postman[28] okruženja za testiranje komunikacije s mrežnim servisima. Najveći fokus ove faze testiranja je bio očitavanje lokacije korisnika, testiranje

[28] https://www.postman.com/, Postman okruženje za testiranje komunikacije s mrežnim servisima

geocodinga, prikaza poruke kada je motor vozila dovoljno zagrijan i prikaza poruke u slučaju nesreće. Rezultati ove faze testiranja su da lokacijski senzori i očitavanje lokacije korisnika radi poprilično precizno, s odstupanjem od 10 metara od stvarne lokacije. Testiranje funkcionalnosti prepoznavanja prometne nesreće izvršavalo se na način da se naglim pokretima ruke testirala sila djelovanja na uređaj. Nakon testiranja, rad ove funkcionalnosti ograničio se da radi samo u uvjetima kada se korisnik kreće vozilom.

Rezultati ove faze testiranja pokazali su se uspješni jer su potvrdili rad svih funkcionalnosti ove mobilne i mrežne aplikacije.

Zaključak

Dobavljači automobilske elektronike u mogućnosti su usredotočiti svoje razvojne napore na jedinstveni, manji, dio sustava u automobilu. To je u suprotnosti sa stajalištima proizvođača automobila koji je odgovoran za integriranje pojedinih dijelova u cijeli elektronički sustav. Umrežene jedinice s distribuiranim funkcijama zahtijevaju da proizvođači automobila imaju razvojne procese i metode koje omogućuju sigurnu upotrebu softvera na razini sustava. To otvara priliku raznim pružateljima usluga da razviju vlastiti softver koji omogućava razne dodatne informacije i povećava vrijednost automobila na simboličkoj razini.

Tijekom posljednjih 10 godina došlo je do eksponencijalnog porasta broja računalno utemeljenih funkcija ugrađenih u vozila. Razvojni procesi, tehnike i alati promijenili su se kako bi se prilagodila toj evoluciji. Cijeli niz elektroničkih funkcija, kao što su navigacija, prilagodljivo upravljanje, informacije o prometu, kontrola proklizavanja, kontrola stabilizacije i sustavi aktivne sigurnosti implementirani su u današnja vozila. Mnoge od ovih novih funkcija nisu samostalne u smislu da trebaju razmjenjivati informacije - a ponekad s strogim vremenskim ograničenjima - s drugim funkcijama. Na primjer, brzina vozila koju procjenjuju kontroler motora ili senzori okretanja kotača moraju biti poznati kako bi se prilagodio napor upravljača, upravljalo ovjesom ili jednostavno odabrala ispravnu brzinu brisača. Složenost ugrađene arhitekture se neprestano povećava. Danas se do 70 elektroničkih kontrolnih jedinica (ECU-ova) razmjenjuje do 2.500 signala (elementarnih informacije poput brzine vozila) na pet različitih vrsta mreža.

Jedan od glavnih izazova automobilske industrije jest osmisliti metode i alate koji će olakšati integraciju različitih elektroničkih podsustava koji dolaze od raznih dobavljača u globalnu elektroničku arhitekturu vozila.

Ovaj je rad prikazao važnost i primjenjivost prikupljanja i izvođenje informacija iz vozila u kretanju, prikazu tih informacija na dostupan i korisniku jednostavan i pregledan način, kombiniranjem podataka sa senzora mobitela i podataka s raznih senzora iz automobila. Cilj ovog rada, pokazati mogućnosti praćenja podataka iz vozila u kretanju u svrhu bolje informiranosti vozača, predstavljen je kroz niz elemenata od motivacije do izrade aplikacije.

Smatram da ovaj dodatno doprinosi struci i studentima programskog inženjerstva, pokazujući načine izrade i interoperabilnost aplikacije mobilne aplikacije s drugom vrstom hardvera, strojeva ili u ovom slučaju prijevoznog sredstva automobila. Dodatno, za kvalitetniji prikaz, rad sadržava i opis postupaka razvoja i testiranja aplikacije te upravljanje ispravljanjem pogrešaka u radu te testiranja rješenja sa stvarnim korisnicima.

Ovim radom pokazano je da je moguće podići iskustvo vožnje na vozilima koja nemaju ugrađene napredne sustave za praćenje podataka iz vozila te su vozači, inače ograničeni na svega nekoliko takvih informacija, rješenjima predstavljenim u ovom radu dobili potpuno novu razinu uvida u informacije i podatke koji su inače karakteristični za više cjenovne klase automobila na tržištu.

Popis kratica

CIN	*Calibration Identification Numbe*	kalibracijski identifikacijski broj
CVN	*Calibration Verification Number*	kalibracijski verifikacijski broj
EGR	*Exhaust Gas Recirculation*	recirkulacija ispušnih plinova
MVC	*Model View Controller*	model pogled kontroler
MVVM	*Model View Model Controller*	model, pogled modela, kontroler
OBD	*On Board Diagnostic*	standard dijagnostike na ploči
EOBD	*Euro On Board Diagnostic*	europski standard dijagnostike na ploči
PID	*Paramatere Identifier*	parametarski identifikator
URL	*Uniform Resource Locator*	usklađeni lokator sadržaja
VIN	*Vehicle Identification Number*	identifikacijski broj vozila
JSON	*JavaScript Object Notation)*	format za čitljivu razmjenu podataka
RPM	*Revolutions per minute*	broj okretaja u minuti
GPS	*Global Positioning System*	globalni položajni sustav

Popis slika

Popis tablica

Popis formula

Popis kodova

Literatura

1. https://www.kdijagnostika.hr/sve-o-skracenicama-obdii-i-eobd/, OBD i EOBD (3.2.2020)
2. https://www.kdijagnostika.hr/sto-su-obdii-modovi/, OBD-II modovi (9.2.2020)
3. http://www.arb.ca.gov/msprog/obdprog/obdfaq.htm, OBD općenito (4.2.2020)
4. http://www.obd-codes.com/, OBD-II kodovi (3.2.2020)
5. https://worddisk.com/wiki/On-board_diagnostics/, OBD općenito (3.2.2020)
6. https://en.wikipedia.org/wiki/OBD-II_PIDs, OBD-II parametarski identifikatori
7. https://api.autopi.io/, AutoPi API dokumentacija (18.2.2020)
8. Mastering Swift 5: Deep dive into the latest edition of the Swift programming language, 5th edition
9. William Van Hecke: Learning iOS Design: A Hands-On Guide for Programmers and Designers
10. https://developer.apple.com/, Apple razvojna dokumentacija (18.2.2020)
11. Society of Automotive Engineers. J2056/1 Class C Applications Requirements Classifications. In SAE Handbook, Vol. 1,1994.
12. Society of Automotive Engineers. J2056/2 Survey of Known Protocols. In SAE Handbook, Vol 2,1994.
13. Van den Brink, Rob. "Dude, Your Car is Pwnd" (PDF). SANS Institute.
14. Marks, Paul (17 July 2013). "$25 gadget lets hackers seize control of a car". New Scientist. Retrieved 5 November 2013.