TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond
Arvutiteaduse instituut

Vootele Verrev, 142683IAPB

OBD-II NÄITUDEL PÕHINEV INFOSÜSTEEM JA MOBIILIRAKENDUS

bakalaureusetöö

Juhendaja: Roger Kerse
MSc

Tallinn 2017

Autorideklaratsioon

Kinnitan, et olen koostanud antud lõputöö iseseisvalt ning seda ei ole kellegi teise poolt varem kaitsmisele esitatud. Kõik töö koostamisel kasutatud teiste autorite tööd, olulised seisukohad, kirjandusallikatest ja mujalt pärinevad andmed on töös viidatud.

Autor: Vootele Verrev

11.04.2017

Annotatsioon

Sõidukite sisepõlemismootorid on oma ehituselt ja talituselt keerulised. Komponentide kuludes suureneb oht mootori töös ohtlike vigade tekkimiseks. Probleemi tekkepõhjuse diagnoosimine osutub sageli keeruliseks, mistõttu on hakatud selle protsessi lihtsustamiseks kasutama auto juhtajult informatsiooni kogumist. Killustatuse vältimiseks on loodud standardne liides *OBD-II*, mille abil on võimalik kindlaid protokolle kasutades juhtajult veakoode ja erinevate andurite väärtusi lugeda. Selleks otstarbeks tuleb kasutada erilist tarkvara, mis on enamasti mõeldud reaalajaks informatsiooni ammutamiseks.

Antud lõputöö raames valmib infosüsteem ja mobiilirakendus, mille abil talletatakse pikaajaliselt infot (kellaaeg, sõiduki asukoht) reiside ja mootori andurite väärtuste kohta. Selle otstarbeks on esiteks automatiseerida sõidupäeviku pidamist, salvestades sõitudel auto asukoha ja kellaaja ning teiseks pakkuda võimalikult palju infot mootori töö kohta, et vigade tekkimisel andurite väärtustest terviklikku pilti manada. Lahenduse abil saab mootori tööd pikaajaliselt jälgida, ja hõlpsasti leida selliseid reise, mille puhul salvestatud andurite väärtused väljusid kasutaja või masinõppe poolt seatud piiridest. Lisaks kujutatakse andurite kaupa graafikutel nende väärtuste muutumine ajas koos piirjoontega, kui need on seatud.

Lõputöö on kirjutatud eest keeles ning sisaldab teksti 31 leheküljel, 5 peatükki, 13 joonist, 2 tabelit.

Abstract

Trip infosystem and app based on *OBD-II* logging

The internal combustion engines of vehicles are complex in their function as well as construction. As the various components experience wear, the possibility of a serious failure increases. In such an event, diagnosing the cause of the problem can prove to be a difficult and time consuming task. To make this process easier, information such as error codes and values of sensors is collected from the engine control unit itself. In order to avoid fragmentation, a port standard *OBD-II* was implemented. Using this physical interface, various protocols are employed to read data off the ECU. In order to do this in practice, special software must be used, which are mostly meant for real-time data collection.

The aim of this thesis, is to create an infosystem and mobile app which persists information about trips (time and location) for a long period of time as well as the values of the different engine sensors. The system will help automate the process of keeping tabs on the trips made with a vehicle by saving the time and location, and provide as much information about the operation of the engine, so that when it encounters a problem, the user is provided with the entire state of the engine. The project will help observe the operation of an engine in the long term and help filter out the trips during which the different sensor values rose above or fell below the thresholds set either by the user itself or automatically by machine learning. In addition, the sensor values are visualized on charts along with the threshold values for the sensor.

The thesis is in Estonian and contains 31 pages of text, 5 chapters, 13 figures, 2 tables.

Lühendite ja mõistete sõnastik

SPA	*Single page application*, üheleheküljeline veebirakendus
SEO	*Search engine optimization*, otsingumootori relevantsuse järjekorras olevate tulemuste nimistus veebilehe kõrgemale tõstmine
OBD	*On-board diagnostics,* sõidukite diagnostikapordi spetsifikatsioon lubamaks mootori andurite väärtusi lugeda
REST	*Representational state transfer*, skeem mis määrab, kuidas üle võrgu ressursside poole pöörduda
HTTP	*Hypertext transfer protocol*, võrguprotokoll mis on mõeldud hüperteksti (tekst + *html tag*id) edastamiseks
MPA	*Mutiple page application*, mitmeleheküljeline veebirakendus
CLI	*Command line interface*, graafilise kasutajaliideseta programm, mida käivitatakse käsureal
JWT	*JSON Web token*, autentimise ja autoriseerimise jaoks väljatöötatud tehnoloogia, mis määrab ära kuidas *token*eid koostada ja kontrollida
MVC	*Model-view-controller*, mudel, mis jagab tarkvara kolmeks eraldiseisvaks osaks – mudeliks, vaateks ja kontrolleriks
URL	*Uniform resource identifier,* unikaalne sõne, mille abil on võimalik ressursi poole üle veebi pöörduda
SQL	*Structured query language*, keel, milles moodustatud päringuid andmebaasisüsteem käivitab ja vastavalt päringule mingi hulga andmeid tagastab
CSRF	*Cross-site request forgery,* rünne, kus kasutaja sessioon kaaperdatakse ja saadud sessiooni id abil esinetakse ohvrina
Overhead	Tarkvaras tehtav lisatöö, ilma milleta rakendus oleks sama funktsionaalsusega ning töötaks veidi kiiremini, kuid oleks arendaja vaatenurgast halvema arhitektuuriga
Mono runtime	*Xamarin*i poolt rakendusega kaasa pandav tarkvara, mis võimaldab *Microsoft*i *.NET* rakendusi mitmel platvormil käivitada
ECU	*Engine control unit,* mootori juhtaju

Sisukord

Jooniste loetelu

Tabelite loetelu

1 Sissejuhatus

Sõidukid, ning eriti sisepõlemismootori jõul liikuvad autod pakuvad autorile oma ehituse ja talituse tõttu huvi. Kuna inseneriaspektist vaadatuna on nad võrreldes näiteks elektriautodega keerukamad, koosnedes paljudest koos töötavatest komponentidest, on mitmeid erinevaid põhjuseid, mis mootori töös probleeme võivad esile kutsuda. Tänapäeval on võimalik suhteliselt suurte summade eest lasta remonditöökodadel mootoridiagnostikat teha, kuid tänu infotehnoloogia kiirele arengule on saanud võimalikuks ka kasutajal endal tuvastada lihtsamate probleemide tekkepõhjuseid. Selleks on hakatud müüma odavaid universaalse ühendusviisiga skännereid, mis juhtajult mitmesugust infot koguda võimaldavad. Nimetatud seadmed on ise suhteliselt primitiivsed ja vajavad kõrgema abstraktsioonitasemega tarkvara, mis neile käsklusi jagaks, et oodatud funktsionaalsust pakkuda.

Loodud on selliseid tarkvaralahendusi mis reaalajas nimetatud väärtusi kuvavad. Lisaks on tehtud ka infot talletavaid rakendusi, mis, olles enamasti töölauaprogrammid, vajavad lisatarkvara allalaadimist, mis ei pruugi kõigi operatsioonisüsteemidega ühilduda ja mille kasutajaliides ei ole kuigi kasutajasõbralik ja/või on ajale jalgu jäänud. Enamasti on sellised lahendused ka tasulised.

Käesoleva lõputöö eesmärk on luua uus infosüsteem, mis suudab kasutajapõhiselt sõidukite erinevate andurite väärtusi ja infot reisi kohta (distants, kestvus, geograafiline asukoht) logida, pikaajaliselt talletada ja mugaval viisil veebirakenduses kuvada. Kogutud andmed aitavad sõidupäevikut pidada, kuna salvestada on võimalik sõiduki asukoht koos kellaaja ja kuupäevaga ning lihtsustada mootoriprobleemide tekkimise korral diagnostikat, pakkudes võimalikult palju informatsiooni mootori andurite väärtuste kohta. Kasutaja saab seada andurite kaupa neile piirväärtusi, millest väljumise korral konkreetne reis, mille kestel piiridest väljumine toimus, eraldi nimekirjas kajastuks. Lisaks on piirjooned näha andurite graafikutel, mille abil on kiiresti võimalik leida hetked, mil väärtus piiridest väljus. Graafikupunkti klikkimisel on võimalik näha

ka muude andurite väärtusi sel hetkel. Kogu infosüsteem peab olema võimalikult lihtne ja nõudma vaid klientrakenduse installeerimist telefoni.

Töö annab lisaks ka lühiülevaate juba olemasolevatest sarnastest lahendustest.

2 Loodav süsteem ja konkurendid

Loodaval süsteemil on kaks eesmärki: esiteks sõidukitega tehtavate reiside marsruutide ning aegade salvestamine ja taasesitamine ning eksportimine sõidupäeviku tarbeks ja teiseks sõiduki mootoris vigade tuvastamine ja andurite väärtuste kõrvalekallete tekkimisel nende avastamine veebiliidese abil. Kasutajal on võimalik valida kolme logimisrežiimi vahel, igaüks neist talletab erineval hulgal infot:

1. Asukoha logimine – salvestatakse sõiduki asukoht ja aeg, mida veebiliideses kaardil kuvatakse.

2. Mootori andurite väärtuste ja vigade logimine – salvestatakse esinenud veakoode ja kasutaja poolt valitud mootori andurite väärtusi (kõik võimalikud andurid on toodud lisa 1 all), ning mida veebiliideses graafikutena visualiseeritakse.

3. Asukoha ja mootori andurite väärtuste ning veakoodide logimine – kombinatsioon kahest eelnevast, mis lisab kaardivaate igale asukohapunktile ka kõikide valitud andurite väärtused, kusjuures säilib võimalus vaadata ka lihtsalt andurite graafikuid.

Kasutaja saab mugaval viisil pilves hoida oma tehtud reise sõidupäeviku tarbeks ning neid ka mugavalt kalendrivaates ülevaatlikult näha või uurida mingi probleemi tekkepõhjust, jälgides mootori andurite väärtuste muutumist ajas. Mingi väärtuse normist kõrvalekalle viitab enamasti kindlale probleemihulgale, mis aitab diagnostikal otsinguruumi vähendada. Selle protsessi hõlbustamiseks on kasutajal võimalik määrata andurite kaupa piirväärtusi (kas miinimumväärtus, maksimumväärtus või mõlemad). Kui sõidu kestel piirväärtust ületatakse, on seda konkreetse anduri väärtuste graafikul näha. Graafikul punkti klikkimisel näidatakse ka teiste andurite väärtusi sel ajahetkel. Samuti kuvatakse eraldi nimekiri kõikidest sellistest reisidest, mille vältel piirväärtusi ületati.

Loodav süsteem kombineerib kahte funktsionaalsust ja lubab kasutajal erineval hulgal infot salvestada ning anduritele piirväärtusi seada, mistõttu ei ole identset tarkvara

loodud. Ometi on programmeeritud mitmeid *Android*i rakendusi, mis reaalajas üle *OBD-II* infot koguvad ja reaalajas kuvavad. Nendeks on näiteks *OBD Car Doctor Pro* ja *DashCommand*. Kasutajaliides asub mobiiltelefonis, mille väiksel ekraanil on suhteliselt ebamugav kaarte, graafikuid ja muud sarnast vaadelda. Teadaolevalt ei sea ükski olemasolev konkureeriv tarkvara automaatselt masinõppe abil selliselt piirväärtusi, nagu loodav projekt seda teeb.

Üks tuntumaid diagnostikarakendusi on Torque, millest on olemas nii tasuline kui ka tasuta versioon. [1] Antud rakendus näitab reaalajas andurite väärtusi ja veakoode ning on loodud ka veebiliides andmete visualiseerimiseks. Kasutajaliidese poolest on tunda, et tegemist on vanema rakendusega. Samuti ei ole võimalik kasutajal vältida mootori andurite väärtuste logimist näiteks sõidupäeviku tarbeks. Lisaks on tegemist vaid *Android*il töötava rakendusega. Infosüsteemi loomisel on arvestatud tulevikus võimaliku *iOS* rakenduse loomisega, kuna server ja klientrakendused on teineteisest võimalikult sõltumatuna realiseeritud, sest suhtlus komponentide vahel toimub vaid *REST* päringute abil. [2] Selleks tuleks kasutusele võtta *WiFi*l töötav *OBD-II ELM327* luger, kuid enamus klientrakenduse loogikast oleks ülekantav ka *iOS*ile.

Eriliseks loodava rakenduse teeb ka asjaolu, et kasutajal on võimalik määrata igale andurile piirväärtusi, mis aitab kasutajale väärtuse piiridest väljumisel lihtsa vaevaga kõrvalekalde nähtavaks teha.

3 Infosüsteemi ülesehitus

Loodud infosüsteem koosneb kolmest osast – kahest klientrakendusest ning ühest serverist. Omavahel suhtlevad nimetatud alamsüsteemid ainult *REST* päringute abil. Nii *Android*i kui ka veebipõhine klient saab ja annab oma info ühisesse kesksesse serverisse. Kogu süsteemi struktuur on üles ehitatud konkreetseid vajadusi silmas pidades – näiteks kahe kliendi olemasolu on seetõttu vaja, et juhtajuga suhtlemiseks valitud seade kasutab *bluetooth*i ja enamik nutitelefone seda ühendusviisi ka toetab. Just telefon on kõige mugavam viis infot koguda, ometi ei ole hiljem kuigi mugav väikeselt mobiiliekraanilt graafikuid ja kaarti vaadata. Selleks otstarbeks on loodud veebipõhine klientrakendus mis töötab kõigis modernsetes veebibrauserites ilma lisatarkvara installeerimata.

Telefonirakendus on kirjutatud vaid *Android*i platvormile, kuna alternatiivsed operatsioonisüsteemid osutusid problemaatiliseks - *iOS* ei toeta turvakaalutlustel Apple poolt kinnitamata seadmete (sealhulgas *OBD-II* skännerite) suhtlust üle *bluetooth*i ja *Windows Phone* on populaarsust niivõrd kaotanud, et sellele keskenduda ei ole kuigi mõistlik. Ometi uuris autor tehnoloogiate kasuks otsustamise ajal erinevaid hetkel aktuaalseid võimalusi.

Kuna infosüsteem on kasutajapõhine siis muuhulgas pandi rõhku ka turvalisusele. Selle eesmärgi täitmiseks kasutati *JSON Web Token*id (*JWT*). Sellest tuleb lähemalt juttu serveri alapeatükis.

Ülevaatlikult toimib tervik järgnevalt:

1. Kasutaja kasutab mobillirakendust, et reisi ja/või mootori näite logida.

2. Server võtab kogu mobiilikliendi töötamise ajal vastu *REST* päringuid ja talletab saadud info andmebaasi.

3. Kasutaja saab veebirakenduse abil eelnevalt kogutud infot vaadata viisil, kus rakendus saadab ja võtab vastu *REST* päringute abil andmebaasi salvestatud teavet. *Android*i rakendus jookseb telefonis, server ja veebirakendus asuvad eraldi masinas ja olenevalt päringust server kas kuulab päringuid *REST endpoin*tidele või tagastab veebirakenduse failid brauseris kuvamiseks.

4 *Android*i klientrakenduse loomine

Mobiilirakenduse loomiseks on mitu erinevat võimalust, mistõttu tuli projekti realiseerimiseks valida konkreetne tehnoloogia mis süsteemi eripärasid arvesse võttes kõige paremini sobiks. Teadliku valiku langetamiseks viis autor end kurssi hetkel aktuaalsete võimalustega ja tegi kogutud info põhjal otsuse.

4.1 Mobiilirakenduste ehitamise võimalused

Erinevatest konkureerivate operatsioonisüsteemide olemasulust johtuvalt tuleb mobiilirakenduse loomisel kasutada erinevaid teeke ja programmeerimiskeeli. See eeldab mitmele süsteemile keskendunud inimeselt mitme tehnoloogia tundmist ja palju aega. Selle kitsaskoha kõrvaldamisega on viimasel ajal hakatud aktiivselt tegelema. On loodud võimalusi, mis aitaks suuri koodijuppe platvormide vahel jagada, kuid ka nendel lahendustel on negatiivseid külgi. Hetkel on levinud kolm eri iseloomuga võimalust, millest tuleb põgusalt juttu. Lisaks nendele variantidele luuakse vahel rakenduse asemel hoopis mobiilis hästi töötav veebileht, aga praeguse tehnoloogia juures ei ole võimalik mugaval viisil telefoni riistvaraga, näiteks *bluetooth*i ja *GPS*iga, suhelda. Antud lahendus ei sobi kuigi hästi loodavasse süsteemi, kuna suhelda tuleb *bluetooth*i seadmega. Seetõttu selline võimalus pikema vaatluse alla ei tule.

4.1.1 *Native* rakendus

Native rakendus on selline tarkvara, mis kompileeritakse otse vastavale kujule, mida seadme riistvaral käivitada saab. [3]

Native rakenduse ehitamisel luuakse igale platvormile eraldi rakendus omaette alamprojektina kasutades selleks konkreetsele operatsioonisüsteemile omaseid teeke. Nii oleks esmase plaani kohaselt loodud kolm eraldi rakendust – *Androidile*, W*indows Phonele* ja Apple *iOS*ile. Sääraselt toimides oleks vastavate alamprojektide loomisel ette

teada, mis operatsioonile konkreetne tarkvara lõpuks jõuab. Sellisel juhul on koodi arhitektuuri mugavam hallata.

Kõnealune lähenemine annab võimaluse võtta rakenduse disainimisel arvesse platvormispetsiifilisi tavasid. Kõige rohkem paistavad need välja just kasutajaliidesest – näiteks on nii *Android*s kui ka *Windows Phone*s kombeks eelmisesse vaatesse naasmiseks kasutada telefonil selleks otstarbeks eraldi olemasolevat tagasi-nuppu, *iOS*i puhul on see nupp tavaliselt, rakenduse vasak ülemises nurgas. [4]

Kirjeldatud lahendus on kolmest vaatluse all olevast kõige aja- ja töömahukam. Korraga tuleb hallata kolme projekti koos oma eripärade ja arendusmetoodikatega. Näiteks on erinevused juba kasutatavates programmeerimiskeeltes rääkimata kasutada olevatest teekidest. Sellise võimaluse valikul on aga plussiks lihtsam platvormispetsiifiliste eripärade haldamine ja kergema vaevaga saab jälgida, et rakendus järgiks operatsioonisüsteemi loojate soovitusi kasutajaliidese disainil. Sageli on sellised rakendused ka suurema töökiirusega ja tunduvad kasutajale kergemini kasutatavad, sest paljud rakendused millega kasutaja juba tuttav on järgivad samu disainipõhimõtteid.

4.1.2 *Semi-native* rakendus

Semi-native rakenduse puhul jagatakse mobiilirakendus kaheks osaks: loogika- ja kasutajaliidese osa. Loogikaosa kasutavad kõik platvormid üht moodi. Sellesse ossa satub tavaliselt rakenduse loogika – tehtavad arvutused, serveriga suhtlemine, andmete töötlemine ja hoiustamine. Kasutajaliidese osa üritatakse hoida võimalikult väiksena, et suuremat osa koodist platvormide vahel jagada. Siia alla käib kõik mis puudutab rakenduse välimust. See osa luuakse igale platvormile eraldi. Oluline on märkida, et kood kompileeritakse igale operatsioonisüsteemile eraldi ja saadud baitkood erineb platvormiti.

Sellise lahenduse tugevaks küljeks on sarnaselt *native*le platvormispetsiifiliste disainijuhtnööride tihe järgimine. Lisaks on see tavaliselt suurema töökiirusega, kui hübriidlahendus. Kirjutatav koodihulk on ka väiksem, sest mingit osa programmist jagatakse kõigi platvormide vahel kusjuures kasutusel on tavaliselt vaid üks keel mitme erineva asemel. Kuna selline toimimisviis on võrdlemisi uus ja kasutuses on veel üpris palju vanemaid seadmeid mida valitud tehnoloogia ei toeta, siis võib jääda püütav

turuosa väiksemaks kui loodeti. Nõrgemateks külgedeks on *native* rakendustega võrreldes pisut madalam töökiirus mis tuleneb kõrgema abstraktsioonitasemega seonduvatest lisakihtidest (*overhead*), ja näiteks *Xamarini* puhul lisatarkvara pakendamine lõpptootesse (m*ono runtime*), mis teeb rakenduse tavaliselt mälumahu poolest suuremaks. [5]

Sellist võimalust saab rakendada kasutades näiteks *X*amarini ja *React* n*ative* tehnoloogiaid. Mõlemaga on võimalik ka kogu kood ühiseks teha (kaasa arvatud kasutajaliidese osa), kuid olenevalt konkreetsest juhu asjaoludest ei pruugi see võimalik olla. [6]

4.1.3 *Hybrid* rakendus

Eelnevate metoodikate puhul jõudis lõpuks telefoni kompileeritud rakendus, seevastu hübriidlahenduse puhul on märgatav selge ideoloogiline vahe – kogu loodud rakendus on lähtekoodi mõttes 100% ulatuses sama olenemata platvormist ja seda ei kompileerita konkreetse operatsioonisüsteemi jaoks sobivaks jooksutatavaks programmiks. Selle asemel tekitatakse rakenduse käivitamisel komponent nimega *WebView*, mis on sisuliselt tervet ekraani kattev veebibrauser. Kogu rakendus jookseb loodud komponendis ja käitub nagu tavaline veebileht, ometi ei samasta kasutaja seda harjumuspärase veebilehega. [7]

Sellise lahenduse puhul on selgeks plussiks vaid ühe koodibaasi haldamine – kui mõlema alternatiivi korral on vähemalt mingi osa koodist dubleeritud, siis hübriidrakenduse puhul see nii ei ole. See ei tee arendust mitte ainult kiiremaks vaid ka lihtsamini jälgitavaks, kuna koodis ei ole erisusi mis aeglustaks selle lugemist ja mõistmist. Ometi leiduvad nii mõnedki nõrgad kohad: 1. Selline rakendus on oluliselt aeglasem kui otse riistvaral käivitatav baitkood, eriti suureks probleemiks võib see kujuneda rohkete animatsioonide ja keerukate arvutuste kasutamisel. 2. Rakendus näeb identne välja kõigis operatsioonisüsteemides, mistõttu ei saa järgida kasutajaliidese loomise soovituslikke jooni. 3. Riistvarakomponentidega suhtlus on keeruline, aeglane, piiratud või isegi võimatu kuna rakendus jookseb sisuliselt brauseris millel pole ligipääsu operatsioonisüsteemi madalamatele kihtidele. [8]

Eelnevast nähtub, et selline lahendusviis sobiks hästi just lihtsamate kasutajaliidestega rakendustele mis ei pea riistvaraga kuigi palju suhtlema.

4.2 *Android*i klientrakenduse tööpõhimõte

Antud projekti skoobis püsides (*bluetooth*il põhinev *OBD-II* lugeja) sobis kõige paremini *Android*i rakenduse loomine. Ometi andis uurimistöö kasulikke teadmisi tulevikus kirjutatavate rakenduse loomiseks. Kuna rakendus pidi töötama vaid ühel platvormil, siis osutus valituks *native* lahendusviis, sest see pakkus kõige paremat töökiirust ning teiste valikute tugevad küljed ei oleks saanud esile tulla, sest koodibaase oli ainult üks ja duplikeeritavat lähtekoodi ei tekkinud.

Mobiilirakenduse loomiseks kasutati *Android Studio*t mis teeb lihtsamaks probleemide põhjuste tuvastamise, kasutajaliide disainimise ja koodi kirjutamise. Samuti on võimalus koodi *Android*i emulaatoris käivitada. Alternatiivina kasutas autor selleks otstarbeks telefoni, kuna emulaator on aeglasem nii käivitamisel kui ka töötamisel. Lisaks saab füüsilise seadme kasutamisel parema ettekujutuse rakenduse toimimisest lõppkasutaja vaatepunktist.

4.2.1 Kasutatud teegid

Kahel tarkvaral on sageli ühiseid jooni, isegi kui nad ei kuulu samasse valdkonda ja lahendavad täiesti erinevaid probleeme. Levinud ühisteks joonteks on näiteks *HTTP* päringute tegemine, objektide serialiseerimine ja deserialiseerimine ning andmebaasiga suhtlemine. Igas projektis ei ole mõistlik ühiseid osi uuesti ja uuesti programmeerida. Sellest tähelepanekust tulenevalt on hakatud valmistama erilisi programmijuppe mis sääraseid toiminguid teevad kusjuures nende liides on võimalikult lihtne ja üldotstarbeline, et neid mistahes rakenduses kasutada saaks. Kirjeldatud programmijuppe nimetatakse teekideks. [9]

Järgnevalt tuleb juttu mõnest teegist, mida *Android*i klientrakendus üldlevinud funktsionaalsuse tarbeks kasutas.

4.2.1.1 Volley

*Android*i klientrakendus peab loodava serveriga suhtlema üle *HTTP* kasutades selleks *REST* päringuid. Selleks otstarbeks valis autor teegi nimega *Volley*. Tegemist on Google poolt loodud tarkvaraga mis suudab erinevaid *HTTP* päringuid saata. [10] Traditsiooniliselt on *Android*i puhul kasutatud *HttpClient*it, kuid seda edaspidi enam ei arendata ja tulevikus see eemaldatakse A*ndroid*ist. [11] *Volley* on tõestatult kiirem kui teised sarnased võimalused ja kasutab ohtralt päringute saatmisel nende vahemällu salvestamist. [12]

4.2.1.2 GSON

Andmeid kogutakse rakenduses lihtsama töödeldavuse huvides Java klassidena, mis sisaldavad vaid *getter*eid ja *setter*eid. Selliste objektide serverisse saatmisel üle *HTTP* konverteeritakse nad erilise formaadiga teksti kujule. Seda protsessi nimetatakse objekti serialiseerimiseks. [13]

Projektis teisendati kõik objektid *JSON*i kujule kuna tegemist on kõige levinuma formaadiga ja võrreldes näiteks XMLiga on selle süntaks minimaalsem ja kiiremini mõistetav. Seetõttu on saadetavad sõnumid väiksemad, mis aitab töökiirusele kaasa. Nimetatud põhjuste tõttu meeldib autorile just *JSON* kõige rohkem. Kõnealuse protsessi tarbeks langes valituks jällegi Google poolt arendatud teek *GSON*.

Antud teek meeldis autorile oma lihtsa API tõttu. Samuti oli selle töökiirus väikeste objektide serialiseerimisel parem kui näiteks laialtlevinud konkurendi Jacksoni puhul [14].

4.2.1.3 *OBD-II* Java API

Selleks, et *bluetooth*i abil sõiduki mootori juhtajult andurite väärtusi ja veakoode lugeda kasutas autor teeki nimega *obd-java-api* [15]. Tegemist oli ainsa sellise funktsionaalsusega tarkvaraga ja seetõttu osutus see valikuks. Antud teegi põhitöö seisnes 16ndkujul olevate *AT* käskude (*ELM327* spetsifikatsiooni käsud) [16] ja juhtaju tagastatud väärtuste tõlkimises inimloetavale kujule Java objektide väljades.

4.2.2 Arhitektuur

4.2.2.1 *Android*i tarkvara ja juhtaju suhtlus

Arvuteid on järk-järguliselt aina rohkem sõidukites kasutama hakatud, mistõttu on saanud võimalikuks nende kasutamine diagnostika otstarbel. Algselt ei olnud sellise võimaluse pakkumine tootjatele kohustuslik ning kui otsustati selline võimalus lisada, siis viis kuidas juhtajuga suhelda võis tootjate kaupa erineda. Killustatuse vältimiseks loodi nõndanimetatud *OBD-II* (On-board diagnostics) spetsifikatsioon, mis määras kindlaks diagnostikapordi füüsilise liidese omadused. Kõnealune spetsifikatsioon muutus tootjatele kohustuslikuks aastal 1996 USAs ning peagi ka Euroopas. [17]

Kuivõrd *OBD-II* määras kindlaks füüsilise liidese omadused, ei kirjutanud see spetsifikatsioon ette mis protokolli diagnostikaks kasutama peaks. Nii tekkis nimekiri erinevatest protokollidest, mis *OBD-II*te kasutas. Skänneerimistööriista tootja kanda jäi mitme protokolli tundmine.

Selleks, et võimalikult paljude sõidukitega ühilduda, kasutas autor töö tegemisel ELM327-tüüpi lugejat. Tegemist on ELM Electronicsi poolt loodud mikrokontrolleriga, mis pakub ühise liidese enamikule levinud *OBD-II* protokollidele. [18]

4.2.2.2 Klientrakenduse tööpõhimõte

Rakenduse avamisel peab kasutaja end emaili ja parooliga autentima. Sisselogimiskatsel saadetakse *HTTP* päring serverile, mis kontrollib andmete õigsust ja tagastab kas veateate või *JSON* web *token*i. Sõltuvalt vastusest kuvatakse kas veateade või avatakse järgmine vaade jättes meelde saadud *token*i. Järgmine vaade (põhivaade) sisaldab vaid üht nuppu, mis reisi alustab. Sellele vajutamisel saab kasutaja valida kolme erineva režiimi vahel: 1. reisi logimine (ainult asukoht ja aeg) 2. mootori näitude logimine (veakoodid ja valitud andurite väärtused) 3. reisi ja mootori näitude logimine (nii asukoht kui ka valitud andurite näidud ja veakoodid). Kui valitakse režiim, mis sisaldab mootori näitude logimist, kuvatakse nimekiri võimalikest anduritest, mille väärtusi logida. Reisi alustamisel kuvatakse kahest leheküljest koosnev vaade, esimene neist kuvab hetkel käimasoleva reisi üldandmeid (distants, kestvus, kogutud andmekomplektide arv) ja reisi lõpetamise nupu ning sõrmega vasakule tõmmates (swipe) viimased kogutud andmed.

Rakenduses on kasutusel olenevalt kasutaja valitud režiimist kuni kaks lõime. Esimene neist on põhilõim, mis uuendab kasutajaliidest ning teine küsib pidevalt ELM327 seadmelt mootori andurite hetkväärtusi. GPS asukoha uuendamise tarbeks on eriline *listener* klass.

Selleks, et rakenduse erinevaid osi võimalikult iseseisvana hoida on kasutusel *Android*i *broadcast*id. Tegemist on sõnumitega, mida saadetakse välja ning millega saab andmeid kaasa panna. Neid sõnumeid saab mistahes komponendis vastu võtta. Selleks peab soovitud kohas klass end *broadcast*ide kuulajaks registreerima, intent filtriga määrama milliseid teateid see vastu soovib võtta ning määrama *BroadcastReceiver*i. See vastuvõtja võib olla klass ise või selle anonüümne alamklass. [19] Järgneb näide broadcasti saatmisest (joonis 1) ja vastuvõtmisest (joonis 2).

```
//Tektitatakse intent koos stringiga (defineeritud konstandid klassis
BroadcastMessages), mis identifitseerib mis tüüpi
//broadcastiga tegemist on
Intent intent = new Intent(BroadcastMessages.LOCATION_UPDATE);

intent.putExtra(BroadcastExtras.LOCATION, location);
intent.putExtra(BroadcastExtras.DISTANCE_TRAVELLED, distanceTravel-
led);
//Intentile lisatakse juurde andmeid, mille hiljem saab välja võtta
//need objektid peavad olema sellise klassi instantsid, mis pärib
klassi Parcelable

LocalBroadcastManager.getInstance(context).sendBroadcast(intent);
//Broadcasti välja saatmine
```

Joonis 1. *Broadcast*i saatmine *Android*is

```
IntentFilter intentFilter = new IntentFilter();
intentFilter.addAction(BroadcastMessages.ENGINE_DATA_UPDATE);
//Intentfilteri abil registreeritakse BroadcastReceiver või sellest
pärinev klass kuulama addActioniga antud sõnumitüüpe (stringid, mis on
defineeritud klassis BroadcastMessages)

...
//Registreeritakse LocalBroadCastManageri abil vastuvõtja
(BroadcastReceiver), mis defineeritakse anonüümsena teise klassi
funktsiooni sees. See võiks olla ka eraldiseisev klass.
LocalBroadcastManager.getInstance(view.getContext()).registerReceiver(
new BroadcastReceiver() {
    @Override
    public void onReceive(Context context, Intent intent) {
      //Seda funktsiooni kutsutakse, kui uus broadcast kohale jõuab
        switch (intent.getAction()) {
      //Kuna kuulata võib mitmeid eri broadcaste, tuleb kontrollida,
mis tüüpi sõnum kohale jõudis
            case BroadcastMessages.ENGINE_DATA_UPDATE:
                if (intent.hasExtra(BroadcastExtras.ENGINE_DATA)) {
```

```
onEngineDataUpdate(intent.getParcelableExtra(BroadcastExtras.ENGINE_DA
TA));
//intent.getParcelableExtra abil võetakse välja broadcastiga kaasa
pandud andmed.
      }
                break;
        }
    }
}, intentFilter);
```

Joonis 2. *Broadcast*i vastuvõtmine *Android*is

Rakenduses on kasutusel kaks eraldiseisvat komponenti, mis erineval ajal kogutavad andmekomplektid serverisse saatmisvalmis saavad: 1. LocationListener – hoiab ja kogub infot hetkeasukoha ja läbitud distantsi kohta 2. ObdReaderCommunicator – hoiab ja kogub mootori andurite ja veakoodide kohta infot. Mõlemad komponendid saadavad andmekomplekti valmissaamisel välja teated mis seda infot sisaldavad.

Rakenduse keskseks punktiks on DataAssembler, mis kuulab nii asukohateateid kui ka mootori andurite väärtuste andmekomplekti uuenduste teateid. Kui mõlemad teated on laekunud, siis koostab see klass serverisse saatmiseks sobiva andmekomplekti, mis siis *JSON*i kujule serialiseeritakse ja ServerCommunicator klassile edastatakse. Viimaks paneb see klass andmetele juurde hetkel kasutuseloleva logimise hetkel kohalikult talletatud *JWT*, et kasutajat identifitseerida ja saadab kogutud info serverisse.

Kõik kasutatavad *HTTP* päringud on asünkroonsed mille vastused saadakse *callback*ist. Selline lähenemine jätab rakenduse päringu tegemise ajal kasutaja käskluseid vastuvõtvaks.

Mobiilirakenduse kuvatõmmised on leitavad lisa 5 all.

5 Veebipõhise klientrakenduse loomine

Veebirakenduste loomisel on palju erinevaid metoodikaid ja tehnoloogiaid mille seast valida. Üks olulisemaid on valikuid seisneb serveriga suhtlemise viisis – kas server peaks tagastama ühe lehe ühe päringu kohta või rakenduse mis sisaldab vajalikke lehti. Selleks, et antud fundamentaalne valik teha, uuris autor mõlema võimaluse eeliseid ja puudusi.

5.1 Mitme- ja üheleheküljeline veebirakendus

Tegemist on kahe ideoloogiliselt vastandliku lähenemisviisiga mida tuntakse kui: 1. mitmeleheküljeline rakendus (multiple-page application - *MPA*) 2. üheleheküljeline rakendus (single-page application – *SPA*). Viimane neist on hakanud just hiljuti populaarsust koguma asendamaks vanemat mitmelehelist arhitektuuri. Järgnevalt on põgusalt neid võrreldud leidmaks kahest parema lahenduse lõputöös rakendamise tarbeks.

Mitmeleheküljeline veebirakendus on selline rakendus, kus iga vaade (lehekülg) saadakse serverist uue GET päringu vastusena. See tähendab, et linkide abil navigeerimisel laetakse serverist uus *html* dokument, mida kasutaja tajub lehe värskendamisena (refresh). Sellega kaasneb tuntav paus mis on esimeseks nõrgaks küljeks selle lähenemisviisi puhul. Teise miinusena võib välja tuua asjaolu, et kui mitmel lehel kasutatakse samu ressursse siis küsitakse igal uue lehe laadimisel neid uuesti. Sellega kaasneb ajakulu isegi siis, kui ressurss mida päritakse on vahemälus (cache). Viimaseks nõrgaks küljeks on klientrakenduse ja serveri tihe sidestatus, ehk server muudab *html* mallide sisu enne lõpliku lehe tagastamist kliendile. Selline korraldus tähendab, et tugeva sõltuvuse tõttu klientrakenduse muutmise korral tuleb ka serverit muuta mis teeb arendustegevuse aeglasemaks. [20]

Kõige olulisem tugevus on seotud otsingumootorite optimiseerimisega (search engine optimization – *SEO*). See tähendab, et mitmelehelist rakendust on lihtsam otsingumootori päringu vastusena relevantsuse mõttes kõrgemale tõsta, kuna otsingumootori tarkvara suudab selliseid veebilehti paremini läbi sõeluda. Kuna ühelehelised rakendused on hakanud kanda kinnitama siis sellele probleemile on üha rohkem keskendatud ja peagi see kaob sootuks. [21]

Üheleheküljeline veebirakendus on selline rakendus, mille lõppkasutaja saab tervikuna oma masinasse ühe *HTTP* päringu vastusena. Serveri poolt tagastatakse vaid üks terviklik *html* fail milleks on rakenduse alguspunkt index.html. Vajalikud skriptid ja ressursid küsitakse serverist vaid ühel korral ning lehtede vahetamise loogika eest vastutab klient ehk server. Selline lähenemisviis on suhteliselt uus.

Üheks olulisimaks tugevuseks *SPA*de puhul on navigeerimisel puuduv paus rakenduse töös. See tähendab, et kasutajakogemus sarnaneb rohkem tavapärasele arvutiprogrammile mitte traditsioonilisele veebirakendusele. Töökiirust aitab veel tõsta fakt, et vajaminevaid ressursse (skriptid, pildid, kujundusfailid jms) päritakse serverist vaid korra. Plussiks on kindlasti ka asjaolu, et klientrakenduse ja serveri vaheline sõltuvus on väike, sest ainsaks kokkupuutevormiks on *REST* päringud andmete küsimiseks ja saatmiseks, ning server ise *html*i mallides väärtusi ei muuda. Kõik see toimub brauseris. Seetõttu on lihtne luua mitu klienti ühele serverile või olemasoleva klientrakenduse funktsionaalsust muuta või täiendada. [22]

Eelnevalt mainitud mitmeheküljelise veebirakenduse tugevus *SEO* vallas ongi peamiseks *SPA* nõrkuseks. Ajalooliselt ei ole suutnud otsingumootorid kuigi hästi selliseid rakendusi läbi sõeluda, kuid see probleem leevendub pidevalt ja eeldatavasti peatselt ka kaob. [23] Lisaks on tavaliselt rakenduse esmane laadimine pisut aeglasem kui traditsioonilise veebilehe puhul. See tuleneb rakenduse avamisel keerukate raamistike ja kokkupakitud suurte skriptide ning stiilifailide korraga laadimisest. Samuti vajavad sellised rakendused tööks *JavaScript*i lubamist kasutaja brauseris, mis tähendab seda, et mõned kasutajad ei pruugi soovitud teabele ligi pääseda. [24]

Mõlemal lahendusviisil on häid ja halbu külgi, kuid tulevikku vaadates on *SPA* nõrkused pigem lahendatava iseloomuga. Seetõttu valis töö autor veebirakenduse loomiseks just selle lähenemisviisi.

5.2 *Front-end JavaScript*i raamistikud ja teegid

Ühelehelise veebirakenduse tüüpi tarkvara loomiseks on välja töötatud mitmeid *front end* raamistikke ja teeke mis aitavad arendajal kiiremini kvaliteetset tarkvara toota. Säärase tehnoloogia valiku tegemine jääbki viimaseks oluliseks otsuseks veebilehe arendamisel.

Lähema vaatluse alla võeti *React* ja *Angular2*, kuna populaarsuse mõttes on need konkurentidega võrreldes oluliselt laiemalt levinud. Sellega kaasneb parem kasutajatugi ja suurem hulk erinevaid lisakomponente (näiteks kalendrikomponendid, vormi valideerimiskomponendid jms) mis lõpptoote töökindlamaks teevad ning arenduskiirust tõstavad [25].

5.2.1 *React* ja *Angular2*

React, nagu ka *Angular2* on avatud lähtekoodiga kasutajaliideste tarkvara. Sarnased on nad veel ka seetõttu, et mõlemad jagavad rakenduse loogilisteks tükkideks mida nimetatakse komponentideks. Enamasti kasutatakse *React*i koos programmeerimiskeelega *JSX*, mis on sarnaselt *Typescript*ile *JavaScript*i edasiarendus [26]. Antud keeles segatakse *html*i ja *JavaScript*i süntaksit. Samuti on kasutusel loogelised sulud, mille vahele kirjutatud muutuja nimed või tehted asendatakse enne lehe kuvamist vastavate väärtustega. Komponente väljendatakse klassidena, mis võivad olla olekulised, see tähendab, et klassil on atribuut props, mille näol on tegemist *JSON* objektiga. Kõik muutujad, mille olekut mälus hoitakse on selle objekti võtmete nimekirjas esindatud. See erineb *Angular2*st, kuna seal sellist vahemuutujat ei ole – muutjaid mille väärtusi meeles hoitakse on otse klassiväljas. [27]

Oluline erinevus kahe tehnoloogia vahel on asjaolu, et kui *Angular2* on terve raamistik, mis määrab ära kogu rakenduse struktuuri siis *React* on teek, mida kasutatakse tavaliselt koos teiste tehnoloogiatega ning mis tegeleb *MVC* (model-view-controller) mudelis vaid vaate kuvamise ja manipuleerimisega.

Eelnevast nähtub, et mõlemal tehnoloogial on nii ühiseid jooni kui ka põhimõttelisi erinevusi.

5.2.2 Tehnoloogia valik

Autorile meeldis *Angular2* loodava rakenduse arhitektuuri tõttu – viis kuidas loogiliselt seotud komponendid, stiilifailid, teenused ja *html*i mallid ühte terviklikku kausta kogutakse on intuitiivne. Rakenduse skaleerudes aitab selline struktureerimine „puhtust hoida" ning arenduse ajal koodifailide vahel navigeerimine on tõhus, kuna arendaja teab juba ette kust mida otsida. Lisaks on projekti alustamine pisut kiirem kui *React*i puhul, sest *Angular2* on erinevalt *React*ist kõik-ühes raamistik mis sisaldab kõike vajalikku, et veebirakendus luua. Seetõttu valis autor töö tegemiseks *Angular2*e. Oluline on aga märkida, et mõlemad variandid on head ning sellest tulenevalt on nad ka praktikas laialdast kasutust leidnud.

5.3 Arhitektuur

Kuna *Angular2* on suhteliselt suur raamistik mille algse konfiguratsiooni tegmemine ei ole triviaalne, siis on loodud ametlik programm, mis algse projekti struktuuri vastavalt soovituslikule viisile valmis teeb. Kõnealuse programmi nimi on *angular-cli*, millest võib välja lugeda, et tegemist on käsurea programmiga. Selle abil on võimalik genereerida mugaval viisil muuhulgas ka komponente ja teenuseid. Rakendust ehitades kasutatakse *webpack*i, mis pakib kõik skriptid ja muud vajalikud failid kokku üheks failiks mille siis server kasutajale tagastab. Antud töö raames just seda tarkvara rakendati. [28]

Projekt on jagatud vaadeteks ning iga vaate failid asuvad eraldi kaustas mille nimi ühtib vaate nimega. Igas sellises kaustas on *html* fail, mis moodustab vaate struktuuri ja sisu (vormid, päised, jalused jms), *SCSS* (*SASS*) fail, mis määrab ära kujunduse ning mis kompileeritakse *css* failiks ning viimaks *ts* (*Typescript*) fail, mis sisaldab endas komponenti, mis eelnimetatud *html* malli sisu vastavalt tingimustele muudab. Lisaks väljatoodud kohustuslikele failidele võib esineda ka selliseid teenuse ts faile, mis on vaid selle vaatega seotud.

Kujunduse tegemiseks on kasutusel *SASS* ehk syntactically awesome style sheets. See on tehnoloogia, mis lisab *css*ile palju kasulikke lisandeid, näiteks muutujad, tehted ja

reeglid. Kuna brauserid neid faile kasutada ei oska, teisendatakse need lõpuks *css* failideks.

Angular2 ametlikul veebilehel soovitatakse õppematerjalis programmeerida *Typescript*i keeles, kuna see toetab arendajat koodi kirjutamise hetkel tüüpide kontrolliga [29]. Analoogselt *SASS*iga ei oska brauserid *Typescript*i käivitada mistõttu transpileeritakse see enne käivitumist *JavaScript*iks. *Typescript* lisab tavalisele *JavaScript*ile, nagu nimigi seda ütleb, muutujatüübid. See aitab programmeerimisel tehtavaid vigu vähendada. Soovitust järgides ning uue tehnoloogia õppimise huvides valis autor klientrakenduse programmeerimiskeeleks *Typescript*i.

5.4 Tööpõhimõte

Kõige esimene sisselaetav *html* fail *index.html* sisaldab endas <*app-root*> *tag*i. Angular2s vastab igale komponendile üks *tag*, mida *html*is kasutada saab. Lõpliku lehe koostamisel asendab raamistik kõik selle *tag*i esinemised vastava komponendi *html*iga. See tähendab, et kõige esimene komponent mida näidatakse on see, mille *tag* vastab *app-root*ile. Antud projektis on selleks *app.component*. Selle juurkomponendi *html* koosneb omakorda kahest *tag*ist. Esimene neist vastab menüükomponendile, kusjuures olenevalt hetkel aktiivsest vaatest on see üldine või reisiga seotud. Teise näol on tegemist erilise raamistiku poolt pakutava *tag*iga <*router-outlet*>, mis ei vasta otseselt ühelegi komponendile, vaid mille raamistik asendab töö käigus vastavalt aktiivsele *route'*ile sobiva komponendiga.

Komponendi klassil on erinevaid elutsükliga seotud funktsioone, mida raamistik kindlatel tingimustel välja kutsub. Kuna loodavad komponendid laiendavad raamistiku baasklassi, siis on võimalik elutsüklifunktsioone üle kirjutada. Nendeks on näiteks konstruktor, *ngOnInit* ja *ngOnDestroy*. [30]

Komponendid, millel on andmevajadused kasutavad selleks konstruktorisse antud teenuseid mis siis klassiväljas salvestatakse. Klassiväljale salvestamise protsess toimub automaatselt ja seda nimetatakse sõltuvuse sisestamiseks (dependency injection). Kui komponent asub moodulis, mille *provider*ite hulgas antud teenust pole ning pole ühtegi

vanemkomponenti millel see olemas oleks, tuleb see komponendi enda *provider*ite listi lisada. [31]

Vaadete vahel navigeerimiseks on kasutusel *Angular2*e juurde kuuluv router. Selle konfiguratsioon on lihtne, koosnedes vaid ühest *Typescript* failist, milles on defineeritud list *JSON* objektidest, kus on määratud kaks või kolm võti-väärtus paari: *path*, mis määrab millise *route'*ile aktiveerimise puhul komponenti kasutatakse, *component*, mis määrab vastava komponendi ning valikulisena *canActivate,* mida kasutatakse kasutaja autentimist nõudvate vaadete puhul. Viimase väärtuseks seatakse teenus, mis implementeerib tõeväärtust tagastavat canActivate meetodit. Käesoleva projekti puhul kontrollib kõnealune funktsioon kas serverilt saadud ning salvestatud *token* kehtib ja on korrektne. Järgneb näide kirjeldatud *route*ide defineerimisest (joonis 3):

```
import { LoginComponent } from './login/login.component';
import { RegisterComponent } from './register/register.component';
import { Login } from './login/login.component';
...
const routes: Routes = [
  {
    path: 'thresholds',
    component: SetThresholdsComponent,
    canActivate: [AuthGuardService]
  },
  {
    path: 'thresholdtrips',
    component: ThresholdTripsComponent,
    canActivate: [AuthGuardService]
  },
  {
    path: 'login',
    component: LoginComponent
  },
...
```

Joonis 3. *Route*ide defineerimine *Angular2*s

Tüüpilise *Angular2*e komponendi näide on leitav lisa 1 all ning komponendi ja *html* malli sidumise näide lisa 3 all. Nimistu kõigist klientrakenduse teenustest ja komponentidest koos lühikirjeldusega on leitav lisa 2 all.

Angular2 osutus veebirakenduse loomiseks sobivaks valikuks, sest valminud lahendus oli lähtekoodi mõttes hea ülesehitusega. Erinevad vaated ja nende juurde kuuluvad serveriga suhtlevad teenused asusid ühes alamkaustas ja uute vaadete lisamine oli lihtne ja kiire, sest ei olnud vaja rakenduse teisi osi muuta. Lisaks serveris sissetulevate

andmete valideerimisele, oli vajalik ka vormide puhul kasutajat vääralt täidetud väljadest informeerida. Selleks otstarbeks sobis hästi *Angular2*e validatsioonimoodul. Rakenduse struktuuri algne ülesseadmine oli kiire, sest *Angular2* sisaldas alustamiseks kõike vajalikku, erinevalt näiteks *React*ist.

Kuvatõmmised veebirakendusest on esitatud lisa 4 all.

6 Serveri loomine

Selleks, et kogutud andmeid säilitada ja erinevatel viisidel visualiseerida on kasutusel keskne rakenduse server ning andmebaas. Kõnealune server tegeleb korraga kahe ülesande täitmisega: 1. tagastab kasutajale klientrakenduse failid, ehk on veebiserver 2. pakub nii veebipõhisele kui ka *Android*i klientrakendusele *REST endpoint*e. Kuna olemas on väga palju erinevaid teeke, raamistikke ja tehnoloogiaid sääraste serverite loomiseks, tuleb järgnevalt sellistest võimalustest lähemalt juttu. See aitab valida probleemi lahendamiseks sobiva tehnoloogia.

6.1 Programmeerimiskeel

On olemas sellist tüüpi servereid, mis tagastavad staatilisi või dünaamilisi lehekülgi (näiteks Apache ja nginx), kuid edasises arutluses mõeldakse veebiserveri all sellist rakendust, mis *REST* päringutele vastuseid annab.

Veebipõhiste klientrakenduste puhul on kasutatavate programmeerimiskeelte hulk piiratud brauseri poolt toetatavate keelte hulgaga, milleks on tänapäeval sisuliselt vaid *JavaScript*. Kuna arendajal on võimalik valida mis masinas serverrakendus tööle hakkab (oluline on vaid see, mis kujul see päringutele vastab), on võimalike valikute hulk oluliselt suurem. Laia valiku korral ei ole alati lihtne sellist fundamentaalset otsust langetada. [32]

Kirjeldatud veebiserveri ülesandeks on kuulata klientide päringuid ning nendele vastaspoole oodatud kujul vastata. Sõltuvalt päringu tüübist, selle parameetritest ning kehaosast, server kas salvestab uut infot andmebaasi või kogub sealt juba olemasolevaid andmeid mida siis enne kliendile tagastamist kindlale kujule teisendab. Siit nähtub, et serveri jaoks on oluline andmebaasiga suhtlus ja mitmesugune tekstitöötlus. Valitav keel võiks selliste ülesannetega võimalikult mugavalt toimida.

Serverirakendusi kirjutatakse sageli sellistes keeltes, nagu *Java*, *C#*, *Python* ja *JavaScript* (*Node.js* platvormil). Kõik loetletud programmeerimiskeeled sobivad eeltoodud probleemide mugavaks lahendamiseks, kuid staatiliselt kirjutatavad *Java* ja *C#* nõuavad sageli rohkem koodi kui skriptimiskeeled, et sama funktsionaalsust pakkuda. Tänu Google poolt loodud Chrome V8 *JavaScript*i mootorile on võimalik *JavaScript*i masinkoodiks kompileerida, mis tõstab oluliselt programmi töökiirust [33]. See tähendab, et suurt töökiirust nõudvaid servereid saab nüüd kirjutada *JavaScript*is. Selline lähenemine tõstab arenduskiirust, kuna loodav koodi hulk on väiksem, kui näiteks *Java* puhul.

Kuna klientide ja serveri vaheline suhtlus toimub vaid *JSON* sõnumitega, on *JavaScript*il põhineval serveril lisaks väiksemale koodihulgale ka eelis, et server mitte ainult ei tagasta *JSON* objekte vaid hoiab andmeid ka sisemiselt sellel kujul. See tähendab, et erinevalt näiteks *Java*st või *C#*st, ei ole enne päringule vastamist objekte *JSON*iks serialiseerida ning sissetulnud päringuga saadetud objekte deserialiseerida. [34]

Autor valis serveri programmeerimiskeeleks *JavaScript*i kasutades *Node.js* platvormi, sest siis kasutaks nii veebipõhine klientrakendus kui ka server sarnast programmeerimiskeelt. Samuti meeldis talle *JavaScript* süntaksi poolest rohkem kui konkureeriv Python.

6.2 Raamistik

Tarkvara luues on ilmnenud, et olenemata otstarbest, lahendavad veebiserverid suurel hulgal sarnaseid ülesandeid. Veebiserverite programmeerimiseks sobivate keelte jaoks on välja kujunenud mitmeid raamistikke, mis pakkudes levinud probleemidele valmislahendusi tarkvaraarendusprotsessi kiirendavad.

Node tehnoloogial põhinevate serverite arendusel on kõige populaarsemaks raamistikuks *Express*. Kõnealuse tehnoloogia oluliseks osaks on tarkvaraarendusmuster nimega *middleware* (vahevara). Tegemist on funktsioonidega, mis käivitatakse igal päringul ning mille argumendina annab raamistik kaasa päringu objekti, vastuse objekti ja viite järgmisele vahevarale (*next*). Funktsioon töötleb päringut ja/või vastust või teeb

muid toiminguid (näiteks salvestab andmeid andmebaasi) ning kutsub siis välja *next* viite abil järgmise vahevara. [35]

Kirjeldatud mustri tõttu on *Express*i ökosüsteem väga rikkalik – loodud on väga mitmesugust funktsionaalsust implementeerivaid vahevarasid. See on üks põhjustest, miks antud raamistikku nii laialdaselt kasutatakse. Raamistiku vahavarade abil lihtsustatakse sageli näiteks päringu kehaosa valideerimist ja parsimist ning päringute logimist faili.

*JavaScript*i serveris kasutamise üks eelistest on väike koodihulk. Minimaalse funktsionaalsusega serveri lähtekood on kõigest viis rida pikk. Ametlik näide minimaalsest serverist näeb välja selline (joonis 4):

```
var express = require('express')
var app = express()
app.get('/', function (req, res) {
      res.send('hello world')
})
```

Joonis 4. Minimaalne expressi server

See server vastab juur-*URL*ile tehtud GET tüüpi päringule sõnega „hello world”.

Serveri loomisel osutus *Express* heaks valikuks, sest võttes arvesse funktsionaalsuse mahtu jäi koodi hulk väikseks. *Boilerplate* koodi (paljudele programmidele ühine osa koodist, mis funktsionaalsust ei lisa, aga on programmi tööks vajalik) oli vähe. *JSON*-formaadis andmeid talletava andmebaasiga suhtlus nõudis samuti vähe koodi, kuna rakenduse objekte ei tulnud teisendada enne andmebaasi lisamist või pärast sealt pärimist. Andmete saatmine ja vastuvõtmine toimus samuti *JSON*-formaadis, mis tähendab, et üheski rakenduse osas teisendamisi ei tehtud ja sisuliselt kogu kood lisas serverile funktsionaalsust.

6.3 Serveri arhitektuur

Valminud infosüsteem rakendab klient-server arhitektuuri, kusjuures kasutusel on kaks erinevat klientrakendust. Selleks, et mõlemad kliendid võimalikult suurt osa serveri lähtekoodist jagaksid, on ainukeseks nendevaheliseks sidepidamisviisiks *REST* päringud. Server ise kunagi klienti ei muuda, nagu näiteks PHP puhul seda tehakse.

Rakenduse alguspunktiks on fail app.js, mis sisaldab endas hulga moodulite importimisi ning kasutatava vahevara registreerimiskäske (app.use). Samuti seadistatakse teeke ja määratakse fail, mis sisaldab kõiki *REST*i *endpoint*e. Viimaks luuakse ühendus andmebaasiga. Rakenduse konfiguratsiooni hoitakse eraldi *JSON* failis mis siis algfailis sisse laetakse.

Kui kõik käsud app.js failis on täidetud, jääb server päringute ootele. *Route*'id, millele neid oodatakse on defineeritud failis routes.js. Selle kuju on järgmine (joonis 5):

```
let express = require('express');
let router = express.Router();
let userController = require('./controllers/user/userController');
let tripsController = require('./controllers/trips/tripsController');

...

router.route('/user/login').post(userController.login);
router.route('/user/register').post(userController.register);
router.route('/user/activate/:id').get(userController.activate);
router.route('/user/accountSettings').put(userController.upda-
teAccountSettings);
router.route('/trip/:id').delete(tripsController.deleteTrip);

...

module.exports = router;
```

Joonis 5. *Route*'ide defineerimine

Lähtekoodist nähtub, et *route* defineeritakse määrates express.Router() objektil mis tüüpi päringut sellele oodatakse (get, put, post, delete jms) ja argumendina antakse kaasa funktsioon, mida raamistik käivitab, kui vastav päring sisse tuleb. Kõik sellised funktsioonid on defineeritud kontrolleri objektis. Nimetatud kontrollerid on otstarbe järgi jagatud eraldi kaustadesse – näiteks userController asub kaustas controllers/user ning tripsController vastavalt controllers/trips. Lisaks hoitakse nendes kaustades

kontrolleri funktsioone, iga üks eraldi failis. Kontrollerid on järgneva struktuuriga (joonis 6):

```
module.exports = {
    login: require('./login'),
    register: require('./register'),
    activate: require('./activate'),
    getTotalStatistics: require('./getTotalStatistics'),

    ...

};
```

Joonis 6. Kontrollerite defineerimine

Tegemist on väga lihtsate failidega, mis koosnevad vaid eksporditavast *JSON* objektist, kus võtmeteks on funktsiooni nimi ja väärtusteks käsk require, mis asendatakse töö käigus selle argumendina kaasa antud failist eksporditud funktsiooniga. Funktsioonid ongi ahelas viimased, nendes toimub andmete andmebaasist küsimine ja sinna salvestamine ning muud infosüsteemi tööks vajalikud toimingud.

Selline rakenduse tükeldamine aitab arendajal rakenduse funktsionaalsuse juurdelisamisel või vana muutmisel kogu tarkvara paremini hoomata ning hallata.

6.4 Andmebaasisüsteem

Andmebaasisüsteeme on kahte tüüpi: relatsioonilisi ning mitterelatsioonilisi. Praktiliselt kõik relatsioonilised andmebaasisüsteemid ootavad kasutajalt päringute tegemiseks *SQL*i kasutamist. Andmed jagatakse tabelitesse (relatsioonidesse) kus veerud määravad mis infot olemi kohta hoitakse (näiteks inimese pikkus, kaal, kinganumber jms) ja read määravad ühe olemi. Igal real on oma unikaalne võti, mille abil saab tabelist kindlaid ridu pärida või teiste tabelitega liita. Sellist tüüpi andebaasisüsteemid on sisemiselt keeruka struktuuriga ja töö jagamine masinate vahel ei ole triviaalne (horisontaalne skaleerimine). Mitterelatsioonilisi andmebaasisüsteeme nimetatakse kirjanduses sageli ka *NoSQL* andmebaasisüsteemideks, demostreerides relatsiooniliste andmebaasisüsteemidega ideloogilist vastandumist. Selliste süsteemide korral ei jagata andmeid relatsioonidesse, vaid hoitakse sõltuvalt implementatsioonist erinevalt. Levinud on andmete paigutamine dokumentidesse, mida hoitakse kollektsioonides. Dokumendid on sisuliselt võtme-väärtuse paarid ja ühes kollektsioonis võib võtmete

hulk ja sisu dokumendite kaupa erineda. *NoSQL* andmebaasisüsteemides tehtavad päringud on sageli kiiremad ja süsteemi ennast on lihtsam horisontaalselt skaleerida. Nõrkuseks alternatiivi ees on, et säilitatavad andmed ei pruugi olla korrektsed, kuna nende sisestamisel puudub SQL-tüüpi süsteemidele omane sisendandmete valideerimine. [36]

NoSQL-tüüpi andmebaasisüsteemid sisendandmete tüüpi reeglina ei valideeri. See tähendab, et kõrge töökindluse tagamiseks peaks serveris enne andmete salvestamist neid täiendavalt valideerima. Kuna jällegi on tegemist laialtlevinud probleemiga, on loodud hulk teeke, mis seda tööd lihtsustavad.

Autor otsustas valida infosüsteemi loomiseks mitterelatsioonilise andmebaasisüsteemi *MongoDB*. Otsustamisel võeti arvesse kõrget töökiirust ja mugavat integreerumist *Express*i serveri ja andmebaasi vahel (*MongoDB*, nagu ka veebiserver, hoiab andmeid *JSON*-tüüpi dokumentides, mistõttu objektide konverteerimist ei toimu). Samuti on tegemist ühe populaarseima mitterelatsioonilise andmebaasisüsteemiga mis tähendab, et vajadusel on kasutajatoe leidmine oluliselt lihtsam kui mõne vähekasutatud tehnoloogia puhul.

Eelkirjeldatud sisendite valideerimise tarbeks on projektis kasutusel Mongoose. Selleks otstarbeks kasutab Mongoose *JSON*-kujul *schema*sid, milles määratakse ära dokumendi võtmed ja nendele vastavate väärtuste andmetüübid. Järgnevalt on esitatud näide sellisest *schema*st (joonis 7):

```
...

let TripSchema = new Schema({
    user: {
        type: mongoose.Schema.Types.ObjectId,
        ref: 'User'
    },
    locationName: String,
    duration: Number,
    distance: Number,
    startingTimestamp: Date,
    endingTimestamp: Date,
    loggingType: String,
    routePoints: [
        {
            type: mongoose.Schema.Types.ObjectId,
            ref: 'RoutePoint'
        }
    ],
    errorCodes: [
        {
            type: mongoose.Schema.Types.ObjectId,
            ref: 'ErrorCode'
        }
    ]
});

...

module.exports = mongoose.model('Trip', TripSchema);
```

Joonis 7. Mongoose schema

Mongoose'is on kõigil *schema*del defineeritud funktsioon save ja find, mis vastavalt salvestavad objekti dokumendina *MongoDB* andmebaasi või leiavad kõik dokumendid, mis päringule vastavad. Mõlema funktsiooni kutsumisel antakse argumendina *callback*-funktsioon, mis käivitub, kui päring ära tehakse. Nende funktsioonide argumendina annab Mongoose sisse lisaks tulemusele ka veaobjekti, mis sisaldab endas infot päringu läbikukkumise põhjuse kohta.

Tüüpiline objekti salvestamine toimub järgnevalt (joonis 8):

```
...

let registrationHash = new RegistrationHash({
      hash: randomstring.generate({length: 64, charset:
'alphabetic'}),
      user: user
    }); //objekti loomine
    registrationHash.save((err) => { //objekti salvestamine
        if (err) {
            res.json({success: false});
        } else {
      sendConfirmationEmailAndSaveUser(res, user, registrationHash.-
hash); //edukal salvestamisel töö jätkamine
        }
    });
...
```

Joonis 8 Tüüpiline objekti salvestamise protsess *MongoDB* andmebaasisüsteemi kasutates *Mongoose*t

Lisaks võimaldab Mongoose kasutajatel peale sisendite valideerimise ka *schema*del defineerida funktsioone, mida kasutatakse enamasti selliste päringute tegemiseks, mida rakenduses mitmes kohas tehakse. [37]

Järgneb näide sellisest projektis kasutatavast päringufunktsioonist (joonis 9):

```
...

TripSchema.statics.findLatest = (email, callback) => {
    User.findByEmail(email, (err, user) => {
        if (!err && user) { //err objektis on esitatud info päringu
põrumise kohta
            return mongoose.model('Trip').findOne({
                "$query": {
                    user: user._id,
                    endingTimestamp: {
                        $exists: true
                    }
                },
                "$orderby": {
                    startingTimestamp: -1
                }
            }, callback); //Andmebaasipäringud on MongoDBs asünk-
roonsed, mistõttu kasutatakse callback-funktsiooni
        }
    });
}
...
```

Joonis 9. *Mongoose'*i viimast lõppenud reisi tagastav *schema* funktsioon

Mongoose sobis *Express*i raamistikuga hästi, andmebaasiga suhtlus oli koodi mõttes lihtsasti arusaadav, sest serveris kasutatavatele objektidele lisab *Mongoose* hulga funktsioone, mis andmebaasioperatsioone teostavad. Nii saab ühte ja sama objekti ilma

teisendusteta kasutada serveris arvutuste ning muude tegevuste tegemiseks kui ka selle andmebaasi salvestamiseks või uuendamiseks.

6.5 Serveri tööpõhimõte

6.5.1 Kasutajate autentimine

Kuna projektis loodud infosüsteem on kasutajapõhine, on olulisel kohal kasutaja autentimine. *HTTP* on olekuta protokoll, mis tähendab, et iga päringut koheldakse kui iseseisvat – sõltuvust eelnevatest või järgnevatest päringutest ei ole. Sellest tulenevalt peab kogu olekuga seotud info iga päringuga kaasas olema, et serveris otsuseid vastu võtta. Traditsiooniliselt on kasutajate autentimiseks kasutatud sessioonipõhis autentimist, mida loetakse olekuliseks. Kasutaja autentimine toimub järgmiselt: 1. klient saadab serverile kasutajanime ja parooli 2. server kontrollib kombinatsiooni õigsust, juhul kui detailid on korrektsed, liigutakse järgmisesse sammu, vastasel juhul tagastatakse veateade 3. server salvestab andmebaasi uue sessiooni, kus on kasutaja andmed ning sessiooni id 4. klient salvestab sessiooni id küpsisena 5. iga järgneva päringuga paneb klient küpsisest saadud sessiooni id kaasa 6. server saab sessiooni id põhjal andmebaasist vajalikud kasutaja detailid mille põhjal päringu töötlemisel erinevaid otsuseid vastu võtta 7. väljalogimisel sessioon serveris ja kliendis hävitatakse. [38]

Sellise lähenemise puhul on mitu nõrka külge: 1. server peab lisatööd tegema, et sessiooni salvestada ja kustutada 2. sessiooniinfo on ainult ühes serveris, mis tähendab, et kui mitu serverit jagab omavahel tööd, on kasutaja ühe masinaga seotud ning teisele päringu tegemisel ei ole ta justkui ennast autentinud 3. andmebaasis hoitavad sessioonid raiskavad mälumahtu 4. võimalikud on *CSRF* ründed, kus ründaja saab enda valdusesse sessiooni id ning esineb serveri ees kui ründe ohver. Põhiprobleemiks on asjaolu, et tegemist on olekulise skeemiga. Nende nõrkade kohtade lahendamiseks on võimalik kasutada olekuta *token*ipõhist autentimist. Selle kohaselt toimub kasutaja autentimine järgnevalt: 1. klient saadab serverile kasutajanime ja parooli 2. server kontrollib nende õigsust, kui detailid on korrektsed jätkatakse tööd, vastasel korral tagastatakse veateade 3. server genereerib ja tagastab kliendile unikaalse piisavalt suure entroopiaga ajutise

allkirjastatud stringi – *token*i, mis sisaldab mingil hulgal infot kasutaja kohta 4. klient salvestab saadud *token*i ja paneb selle iga järgmise päringuga kaasa 5. server kontrollib *token*i õigsust ja kui see validatsiooni läbib, jätkab tööd kasutades *token*is sisalduvat infot kasutaja tuvastamiseks. [39]

Autor rakendas kasutajate autentimiseks *token*ipõhist lähenemist. Tokenite genereerimiseks on erinevaid võimalusi, tuntuimaks millest on *JWT*d, ehk *JSON* web *token*id. Need koosnevad kolmest punktiga eraldatud base64 kujul sõnest, mis esitavad *JSON* objekte. Esimene osa on päis, mis sisaldab kasutatud räsialgoritmi (alg) ja *token*i tüüpi (typ). Teine osa sisaldab tavaliselt andmeid kasutaja ja tema õiguste kohta ning esineb ka muud infot, näiteks *token*i väljastaja (iss) ning kehtivusaeg (exp). Viimane osa on allkiri, mida kasutatakse terve *token*i valideerimiseks – selle abil saab olla kindel, et *token*i saatja on see kellena ta esineb ning transpordi käigus sõnud muutunud pole. [40]

*Token*ite genereerimiseks ja valideerimiseks kasutati *express*i vahevara nimega *jsonwebtoken*. Selleks, et vältida vabatekstina kasutajate paroolide hoidmist, kasutatakse teeki bcrypt, mille abil kõik paroolid enne andmebaasi salvestamist räsitakse. Samuti lisatakse kõigile räsidele ka sool, et vältida *raindow-table* tüüpi rünnakuid, kus ründajad koostavad suuri tabeleid laialtlevinud paroolide ja sõnade räsiväärtustest, mida siis järgemööda rakenduse lekkinud andmebaasiga võrreldakse lootuses viia kokku kasutaja email ja parool, mida kasutajad pahatihti ka muudel veebilehtedel kasutavad.

6.5.2 Päringute töötlemine

Serveri sisuline töö algab pärast kliendi päringu vastuvõtmist, kui *express*i raamistik kutsub välja kontrollerites defineeritud päringu tüübile ja *URL*ile vastava funktsiooni. Selliste e*ndpoint*ide puhul, mida peab kasutada saama ainult autenditud kasutaja, võetakse päringu päisest *authorization* väljast kasutajale eelnevalt eduka autentimise vastusena saadetud *token*. Edasi valideeritakse saadud *token*it ning selle puudumisel või vale väärtuse esinemise korral vastatakse kasutajale veateatega, kusjuures päringu *HTTP* staatuskoodiks määratakse 401 – unauthorized. Eduka valideerimise korral jätkatakse funktsiooni täitmist, kusjuures otsuste tegemisel kasutatakse *token*i genereerimisel selle kehaossa pandud kasutaja emaili, mis kasutaja üheselt määrab.

Järgnevalt on toodud näide sellest, kuidas kontrollitakse, kas päringu tegijal on õigus selle päringu tulemusele (joonis 10):

```
...
//jwt on expressi jsonwebtoken vahevara
jwt.verify(req.headers.authorization, app.get('config').jwtSecret,
(err, decodedToken) => {
        if (err) {
            //autentimine kukkus läbi
            res.status(401).json({success: false});
        } else {
            //kasutajal on õigus päringu tulemusele, jätkame tööd
            findAndReturnTrips(req, res, decodedToken);
        }
    });

...
```

Joonis 10. Serverile päringu teinud kasutaja autentimine enne funktsiooni käivitamist

Eduka autentimise tulemusena saadakse dekodeeritud kujul *token*, ning tööd jätkatakse. Kuna sageli tehakse vaid andmebaasipäringuid, mille tulemused tuleb kliendile saata on *Node*i kasutamine väga mugav, sest mingit teisendust vastuse saatmiseks *JSON* kujule teha ei ole vaja. Järgneb näide tüüpilisest päringu töötlemisest (joonis 11):

```
function findAndReturnTrips(req, res, decodedToken) {
...
//leiame kõik kasutaja reisid kasutades tema emaili, mille saame //to-
keni kehaosast
Trip.findByEmail(res, decodedToken.user.email, 'locationName distance
duration startingTimestamp loggingType', (err, trips) => {
            if (!err && trips && trips.length > 0) {
                res.json(trips);
            } else {
                res.json({success: false});
            }
      });
```

Joonis 11. Päringu töötlemine – kasutaja reiside leidmine ja tagastamine

Lisaks päringutele vastamisele jooksutab server veel töid, mida *cron*iga ajastatakse. Näitena sellisest tööst on lõpetamata jäänud reiside automaatne lõpetamine. Võib juhtuda, et reisi kestel ühendus serveriga kaob ja klient ei saa serverit reisi lõpetamise korral sellest teavitada. Sellises olukorras peatab server ise reisi, kuna kindla intervalli tagant läheb tööle funktsioon, mis kõik sellised reisid lõpetab, millele pole pika aja vältel ühtegi andmekomplekti lisatud (asukoht ja/või mootori andurite näitude komplekt).

6.5.3 Masinõpe

Loodud infosüsteemi abi diagnostika tegemisel seisneb selles, et reiside hulgast tuuakse eriliselt välja sellised, mille jooksul esines mootoris vigu (loeti veakood) või ületati seatud piirväärtusi. Nii on olemas eraldi nimekiri kõigist sellistest reisidest. Lisaks kuvatakse andurite väärtuste graafikutel piirväärtusi kujutavaid jooni. Selliselt on kohe näha, kas väärtused sõidu kestel piiridest väljusid. Nimetatud väärtuste seadmiseks on kaks võimalust: esiteks kasutaja ise määrab need vastavalt oma teadmistele ja kogemustele ning teiseks server seab need väärtused automaatselt kasutades masinõpet.

Kasutaja ei pruugi osata piirväärtusi ise kuigi täpselt seada, mistõttu on loodud võimalus lasta need serveri poolt määrata. Kuna igal mootoril on erinevad normväärtused, on piiride seadmisel oluline arvesse võtta konkreetse mootori eripärasid. Selleks peab kasutaja võimalikult üheselt määrama, mis mootoriga tegu on. Maailmas läbi aegade toodetud suure hulga mootorite koondamist nimekirja ei ole praktiliselt võimalik teha. Selle asemel laseb loodud infosüsteem kasutajal oma mootorit kirjeldada, selleks määrab ta järgmised parameetrid:

1. Mark

2. Mudel

3. Kubatuur

4. Kütusetüüp (bensiin, diisel)

5. Ülelaadimistüüp (vabalthingav, turbo, kompressor, mõlemad)

6. Maksimaalne jõud

Antud parameetrid on valitud selliselt, et kasutajal oleks neid lihtne sisestada, see tähendab, et ta teaks neid peast ja et viia kahe erineva mootori kokkulangemiste arv võimalikult väiksele tasemele.

Pärast reisi lõppemist on teada iga logitud anduri väärtuste vahemik ja toimub piirväärtuste värskendamine. Juhul, kui antud mootori kohta pole veel ühtegi teist reisi salvestatud, määrab server anduri väärtuse miinimumpiiri leides reisi kestel salvestatud

minimaalse väärtuse ja lahutades sellest väikese suuruse ning analoogselt maksimumpiiri lisades väikese suuruse salvestatud maksimaalsele väärtusele. Eelneva piirväärtuste komplekti olemasolul arvutatakse järgmiselt:

1. uus miinimumväärtus = (hetkel salvestunud keskmestatud miinimumväärtus + uus miinimumväärtus) / 2

2. uus maksimumväärtus = (hetkel salvestunud keskmestatud maksimumväärtus + uus maksimumväärtus) / 2

Kirjeldatud lihtsate reeglite põhjal uuendatakse piirväärtusi pidevalt, mis tähendab, et mida rohkem inimesi süsteemi kasutab, seda rohkem need piirväärtused mootori normväärtustega kooskõlla viiakse. Suure hulga kasutajate korral on üpris täpselt teada, mis piirid konkreetse mootori jaoks normaalsed on.

7 Kokkuvõte

Käesoleva lõputöö eesmärgiks oli luua reiside logimist ja sõidukitele diagnostika tegemist hõlbustav tööriist, mis oleks mugava ja lihtsa liidesega ning ei nõuaks lõppkasutajalt täiendavate programmide alla laadimist. Eesmärgi täitmiseks loodi infosüsteem, mis koosnes serverist ja kahest eri tüüpi klientrakendusest. *Android*i klientrakenduse abil kogutakse reiside ja/või mootori andurite väärtusi, mille server andmebaasi talletab. Veebipõhise klientrakenduse ülesandeks jäi eelnevalt kogutud info intuitiivsel kujul esitamine. Selline lähenemine lubab kasutajal sõiduki juhtimisele keskenduda, kui samaaegselt mobiilirakendus infot kogub ning hiljem mugavast keskkonnast vaid arvuti ja internetiühenduse olemasolul saadud andmeid visualiseerida graafikul, kaardil ning ülevaatlikul kalendril.

Loodud lahendus on tehniliselt väga mitmekülgne – esindatud on veebi- ja mobiilirakenduste kategooria ning loodud on ka server. Autor tegeles tuli nii kõrge abstraktsioonitasemega lähtekoodiga, nagu vaadete loomine üheleheküljelises veebirakenduses kui ka madalatasemeliste ülesannetega, nagu erilise riistvaraga (*OBD-II* spetsifikatsiooni kasutav *ELM327*-tüüpi seade) suhtlemisega. Kasutusel oli mitu programmeerimiskeelt – *Java*, *JavaScript* ning *Typescript* kui ka modernseid tehnoloogiaid, nagu *SPA*, olekuta autentimissüsteem *JWT*de abil, *NoSQL*-tüüpi andmebaasisüsteem ning *JavaScript* serveri pool. Kogu arendusprotsessi vältel tuli ka kõikvõimalikke alternatiive ja nende tugevusi ning nõrkusi uurida, mistõttu tegeles autor nii praktilise kui ka teoreetilisega.

Käesolev töö andis autorile teadmise, et valitud tehnoloogiaid kasutades on võimalik luua lihtne reaalselt töötav rakendus.

Kasutatud kirjandus

[1] Torque Lite [WWW]
https://play.google.com/store/apps/details?id=org.prowl.torquefree&hl=et (09.05.2017)

[2] Torque [WWW]
http://torque-bhp.com/ (22.04.2017)

[3] native application Definition from PC Magazine Encyclopedia [WWW]
http://www.pcmag.com/encyclopedia/term/47651/native-application (09.05.2017)

[4] What's different when designing apps for iOS or Android? [WWW]
https://blog.dropsource.com/whats-different-when-designing-apps-for-ios-or-android-99648738206b (12.04.2017)

[5] Alternatives to Native Mobile App Development [WWW]
https://auth0.com/blog/alternatives-to-native-mobile-app-development/ (12.04.2017)

[6] Platform Specific Code [WWW]
https://facebook.github.io/react-native/docs/platform-specific-code.html (12.04.2017)

[7] Architectural overview of Cordova platform [WWW]
https://cordova.apache.org/docs/en/latest/guide/overview/ (12.04.2017)

[8] Benefits and Disadvantages of Hybrid Mobile Applications – Brooks Canavesi [WWW]
http://brookscanavesi.com/uncategorized/benefits-disadvantages-hybrid-mobile-applications/ (12.04.2017)

[9] Steve Mcconnell, (1996). Rapid Development – Taming Wild Software Schedules. Redmond, Washington: Microsoft Press

[10] google/volley [WWW]
https://github.com/google/volley (09.05.2017)

[11] HttpClient | Android Developers | Android Developers [WWW]
https://developer.android.com/reference/org/apache/http/client/HttpClient.html (09.05.2017)

[12] An Introduction to Volley [WWW]
https://code.tutsplus.com/tutorials/an-introduction-to-volley--cms-23800 (14.04.2017)

[13] Serialization and Unserialization, C++ FAQ [WWW]
https://web.archive.org/web/20150405013606/http://isocpp.org/wiki/faq/serialization (24.04.2017)

[14] The Ultimate JSON Library: JSON.simple vs GSON vs Jackson vs JSONP [WWW]
http://blog.takipi.com/the-ultimate-json-library-json-simple-vs-gson-vs-jackson-vs-json/ (15.04.2017)

[15] obd-java-api [WWW]
https://github.com/pires/obd-java-api (09.05.2017)

[16] AT_commands [WWW]
https://www.sparkfun.com/datasheets/Widgets/ELM327_AT_Commands.pdf
(09.05.2017)

[17] About On-Board Diagnostic II (OBD II) Systems [WWW]
https://www.arb.ca.gov/msprog/obdprog/obdfaq.htm (15.04.2017)

[18] OBD – Elm Electronics [WWW]
https://www.elmelectronics.com/products/ics/obd/ (15.04.2017)

[19] Broadcasts [WWW]
https://developer.android.com/guide/components/broadcasts.html (15.04.2017)

[20] Single-page application vs multiple-page application [WWW]
https://medium.com/@NeotericEU/single-page-application-vs-multiple-page-application-2591588efe58 (16.04.2017)

[21] SEO Strategies for JavaScript-Heavy Single Page Applications or AJAX Sites | Search Engine Watch [WWW]
https://searchenginewatch.com/sew/how-to/2358775/seo-strategies-for-javascript-heavy-single-page-applications-or-ajax-sites (16.04.2017)

[22] The Pros and Cons of Single Page Applications (SPAs) [WWW]
https://www.dialogtech.com/blog/technically-speaking/technically-speaking-the-pros-and-cons-of-single-page-applications-spas (16.04.2017)

[23] Single Page Applications (SPA) and the SEO Problem | ADK Group [WWW]
http://adkgroup.com/insights/single-page-applications-spa-and-seo-problem (09.05.2017)

[24] The disadvantages of single page applications by Adam Silver [WWW]
http://adamsilver.io/articles/the-disadvantages-of-single-page-applications/ (16.04.2017)

[25] Top JavaScript Frameworks & Topics to Learn in 2017 [WWW]
https://medium.com/javascript-scene/top-javascript-frameworks-topics-to-learn-in-2017-700a397b711 (16.04.2017)

[26] React without JSX [WWW]
https://facebook.github.io/react/docs/react-without-jsx.html (09.05.2017)

[27] Tutorial: Intro to React [WWW]
https://facebook.github.io/react/tutorial/tutorial.html (18.04.2017)

[28] angular-cli [WWW]
https://github.com/angular/angular-cli (16.04.2017)

[29] Quickstart - ts – QUICKSTART [WWW]
https://angular.io/docs/ts/latest/quickstart.html (20.05.2017)

[30] Lifecycle hooks [WWW]
https://angular.io/docs/ts/latest/guide/lifecycle-hooks.html (18.04.2017)

[31] Dependency Injection [WWW]
https://angular.io/docs/ts/latest/guide/dependency-injection.html (18.04.2017)

[32] Client Side vs. Server Side [WWW]
http://www.codeconquest.com/website/client-side-vs-server-side/ (19.04.2017)

[33] Introduction v8/v8 Wiki [WWW]
https://github.com/v8/v8/wiki/Introduction (09.05.2017)

[34] Java vs. Node.js: An epic battle for developer mind share | InfoWorld [WWW]
http://www.infoworld.com/article/2883328/java/java-vs-nodejs-an-epic-battle-for-developer-mindshare.html (19.04.2017)

[35] Using middleware [WWW]
https://expressjs.com/en/guide/using-middleware.html (19.04.2017)

[36] Relational databases vs Non-relational databases | James Serra's Blog [WWW]
http://www.jamesserra.com/archive/2015/08/relational-databases-vs-non-relational-databases/ (19.04.2017)

[37] Mongoose Schemas [WWW]
http://mongoosejs.com/docs/guide.html (19.04.2017)

[38] Cookies vs Tokens: The Definitive Guide [WWW]
https://auth0.com/blog/cookies-vs-tokens-definitive-guide/ (19.04.2017)

[39] The Ins and Outs of Token Based Authentication | Scotch [WWW]
https://scotch.io/tutorials/the-ins-and-outs-of-token-based-authentication (19.04.2017)

[40] JSON Web Token Introduction – jwt.io [WWW]
https://jwt.io/introduction/ (19.04.2017)

Lisa 1 – Võimalikud jälgitavad mootori andurid

Tabel 1. Võimalikud jälgitavad mootori andurid

Nimi	**Kirjeldus**
Absolute load	Mootori koormus protsentuaalselt
Fuel level	Kütusetase protsentuaalselt
Barometric pressure	Õhurõhk
Fuel consumption rate	Kütusekulu
Fuel pressure	Kütuserõhk kütusepumba juures
Fuel rail pressure	Kütuserõhk kõrgsurveanumas (fuel rail)
Fuel trim	Kütusesegu muutmise suund (negatiivse väärtuse korral on segu liiga rikastunud ning juhtaju vähendab kütusehulka ja vastupidi)
Engine oil temp	Mootoriõli temperatuur
Wideband Air-fuel ratio	Täpne küttesegu muutmise juhis (vastandudes tavalisele air-fuel *ratio*le, mis määrab vaid suuna aga mitte koguse)
Air-fuel ratio	Õhu ja kütuse vahekord segus, protsentuaalselt (näiteks 16:1 korral siseneb mootorisse iga 1 kütuseühiku kohta 16 ühikut õhku)
Vehicle speed	Sõiduki kiirus
Ambient air temp	Välisõhu temperatuur
Engine coolant temp	Mootori jahutusvedeliku temperatuur
Air intake temp	Sissevõetava õhu temperatuur
MAF	Mass air flow – mootorisse siseneva õhu kogus
Intake manifold press	Sisselaskekollektori rõhk
Battery voltage	Aku pinge
Throttle position	Gaasiklapi asend protsentuaalselt
Engine speed	Mootori kiirus (pöördeid minutis)
VIN	Sõiduki VIN (*vehicle identification number* – sõidukile antav number, mis selle

	ülemaailmselt üheselt määrab)
Timing advance	Juhtaju saadetav signaal, mis sunnib kütusesegu varem süttima, protsentuaalselt TDCst (*top dead center* – kolvi kõrgeim asend põlemiskambris)

Lisa 2 – Tüüpiline *Angular2*e komponent

```
//raamistiku poolt pakutud ning kasutaja loodud komponentide ning teenuste importimine
import { Component, OnInit } from '@angular/core';
import { TripsService } from './trips.service';
import { ReversePipe } from './reverse.pipe';
@Component({
  selector: 'app-trips', //määratakse html'is kasutatav tag, kuhu raamistik komponendi render'dab
  templaten: './trips.component.html', //vaate html malli asukoht
  styleUrls: ['./trips.component.scss'], //vaate stiili määravate failide asukoht
  providers: [TripsService, ReversePipe] //sõltuvuse sisestamine
})
export class TripsComponent implements OnInit {
  trips: any; //mudeli muutujad, mille väärtusi saab sidumise abil ka html'i mallis kasutada
  loading: boolean = true;
  failure: boolean;
  constructor(private tripsService: TripsService) { } //sõltuvuse sisestamine, raamistik salvestab teenuse this.tripsService välja
  ngOnInit() { //raamistiku poolt pakutud elutsükli meetod
    this.loading = true;
    this.getTrips();
  }
  getTrips() {
//päring serverile ning saadud andmete salvestamine mudelisse, et hiljem html'is neid kuvada
    this.tripsService.getTrips(false).subscribe((data) => {
      this.trips = data;
      if ('success' in this.trips && !this.trips.success) {
        this.failure = true;
      }
      console.log(this.trips);
      this.loading = false;
    }, (err) => {
      console.log(err);
    });
  }
}
```

Joonis 12. Tüüpiline *Angular2*e komponent

Lisa 2 – Klientrakenduse komponendid ja teenused

Tabel 2. Veebipõhise klientrakenduse komponendid ja teenused

Nimi	**Tüüp**	**Ülesanne**
account-settings	Komponent	Kuvada vorm läbi mille on võimalik kasutaja andmeid muuta.
account-settings	Teenus	Küsida serverilt hetkel aktiivsed kasutaja andmed ja saata uuendatud andmed salvestamiseks serverisse.
auth-guard	Teenus	Tagastada *router*ile tõeväärtus, kas hetkel aktiivne kasutaja (*JWT* põhjal) valitud *route*i aktiveerida saab või mitte. Seda kasutatakse kontrollimaks, millistele lehtedele autentimata kasutaja ligi pääseb.
auth	Teenus	Küsida serverilt järgnevate päringutega kaasa pandav *token*. Hoida *token*it alles ja väljalogimisel see kustutada. Pakkuda komponentidele *token*is sisalduvat kasutaja infot.
dashboard	Komponent	Kuvada kasutajale konsolideeritud infot kõigi reiside kohta ja pakkuda mugav link viimasele reisile. Näidata hoiatusteateid kui viimasel reisil esines mootoris vigu või seatud piirväärtusi ületati.
dashboard	Teenus	Küsida serverilt reiside kohta konsolideeritud infot

engine-chart	Komponent	Kuvada mootori andurite väärtuste muutumist ajas graafikul.
form	Teenus	Pakkuda kõigile vormidele väljade valideerimiseks kasutavaid mugavaid funktsioone.
home	Komponent	Kuvada autentimata kasutajatele ülevaatlikku lehte tarkvarast ja selle funktsionaalsusest. Anda viited uue kasutaja registreerimiseks.
login	Komponent	Kuvada vorm, mille abil kasutaja ennast autentida saab.
navigation	Komponent	Kuvada mitmele leheküljele päisesse navigatsiooniriba, mis sisaldab linke reiside vaatamisele, ülevaatlikule kalendrile ning piirväärtusi ületanud reiside vaatamisele.
overview	Komponent	Näidata kasutajale kalendri kujul konsolideeritud infot reiside kohta ning võimaldada mingis ajavahemikus tehtud reiside eksporti faili mõeldud kasutamiseks sõidupäeviku koostamisel.
password-recovery	Komponent	Näidata kasutajale vormi, mille abil saab ta oma parooli muutmise lingi emailile saata.
password-recovery	Teenus	Saata serverile päring parooli muutmiseks mõeldud emaili saatmiseks.
register	Komponent	Kuvada huvilisele vorm, mille abil uus kasutaja registreerida.
register	Teenus	Saata serverile uue kasutaja andmed.
set-new-password	Komponent	Emailist saadud lingi põhjal kasutajale uue parooli

		määramiseks kasutatava vormi kuvamine.
set-new-password	Teenus	Saata serverile uus parool.
set-threshold	Komponent	Kuvada kasutajale vorm, mille abil muuta või seada andurite piirväärtused, mille ületamisel rakendus kasutajat teavitab.
set-threshold	Teenus	Saata uued või muudetud piirväärtused serverile.
threshold-trips	Komponent	Kuvada kasutajale nimekiri reisidest, mille käigus seatud piirväärtusi ületati.
trip-details	Komponent	Kuvada kasutajale asukoha logimise puhul kaart sõidutrajektooriga. Mootori väärtuste logimise puhul lisaks veel iga asukoha punkti kohta mootori andurite väärtuste komplekt.
trip-details	Teenus	Küsida serverilt valitud reisi kohta kõik detailandmed.
trip-navigation	Komponent	Kuvada kasutajale menüüriba, mis sisaldab linke kaardivaatele ja graafikuvaatele.
trips	Komponent	Kuvada kasutajale nimekiri kõigist tehtud reisidest.
trips	Teenus	Küsida serverist kõigi reiside üldandmeid.
registration-confirmation	Komponent	Kuvada kasutajale pärast edukat emaili valideerimist kinnitus registreerimise õnnestumisest.
registration-confirmation	Teenus	Teha serverile päring, mis registreerimise lõpuni viib.
set-engine	Komponent	Kuvada kasutajale vorm, mille abil identifitseerida sõiduki mootor, et masinõppe abil

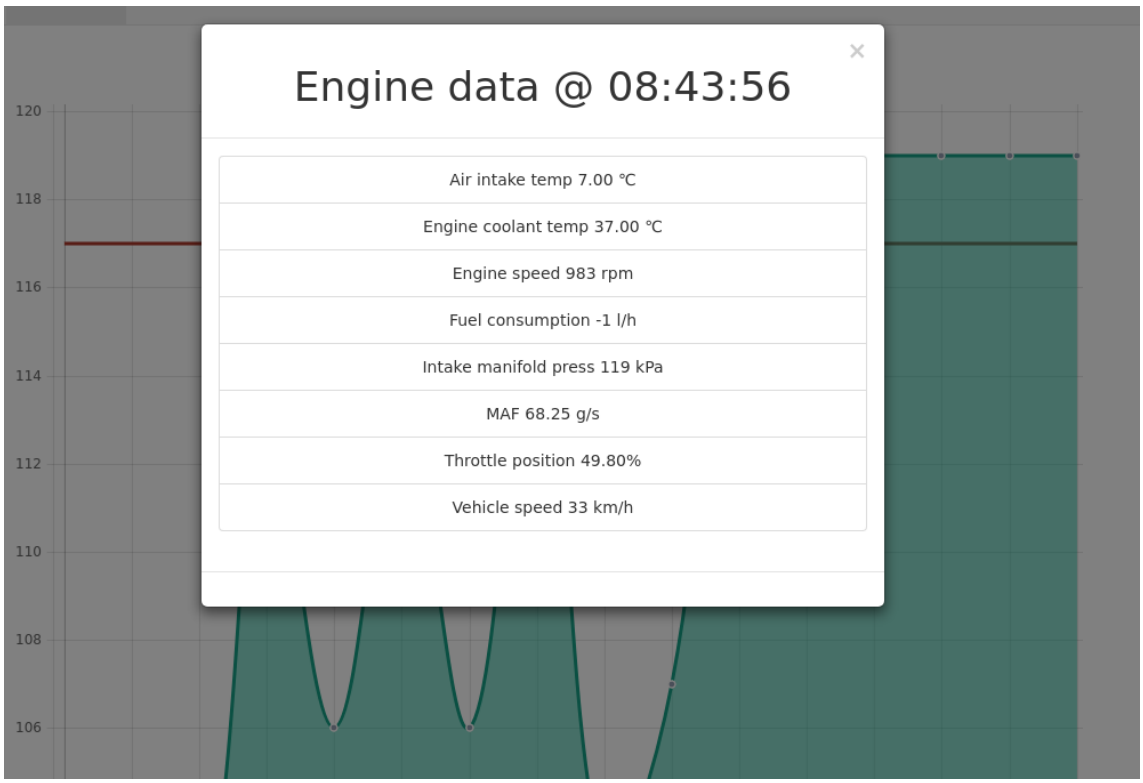
		automaatselt piirväärtuseid seada.
set-engine	Teenus	Küsida serverilt markide ja mudelite nimekiri ning hetkel seatud mootori parameetrid. Samuti saata uuendatud või esimest korda seatud parameetrid.

Lisa 3 – *Angular2*e komponendi sidumine *html* malliga

```
<div class="row loading-icon-div" *ngIf="loading"> //ngIf kontrollib
komponendi väljal oleva muutuja väärtust
...
<div class="container">
...
    <div class="list-group step" *ngFor="let trip of trips | reverse"
[class.show]="!loading"> //ngFor abil saab itereerida mudeli väljal asuvat
listi
      <div *ngIf="trip.loggingType != 'ENGINE_LOGGING'">
        <a href="#" [routerLink]="['/trip', trip._id]" class="list-group-
item" [ngClass]="trip.loggingType">
          <h4 class="list-group-item-heading">{{trip.locationName}}
{{trip?.startingTimestamp | date:'dd.MM.yyyy HH:mm'}}</h4>
          <p class="list-group-item-text">{{trip.duration}} min |
{{trip.distance}} km</p>
        </a>
      </div>
      <div *ngIf="trip.loggingType == 'ENGINE_LOGGING'">
        <a href="#" [routerLink]="['/chart', trip._id]" class="list-group-
item" [ngClass]="trip.loggingType">
          <h4 class="list-group-item-heading">{{trip.locationName}}
{{trip?.startingTimestamp | date:'dd.MM.yyyy HH:mm'}}</h4>
          <p class="list-group-item-text">{{trip.duration}} min |
{{trip.distance}} km</p>
        </a>
      </div>
    </div>
  </div>
</div>
```

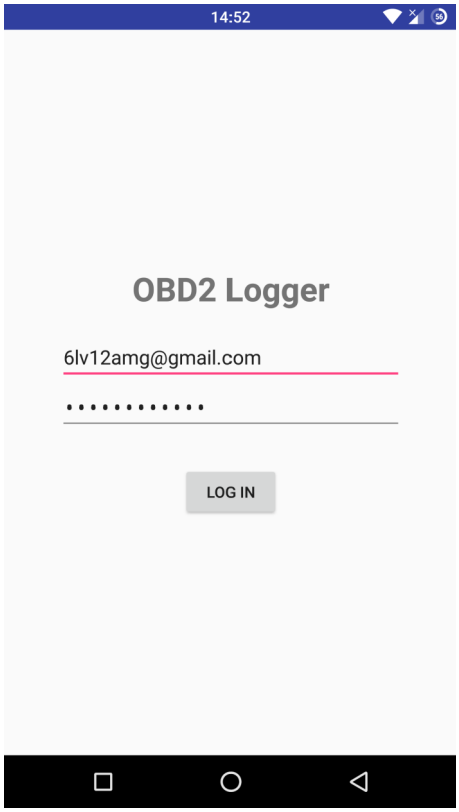
Joonis 13. *Angular2*e *HTML* mall ja selle sidumine komponendi muutujatega

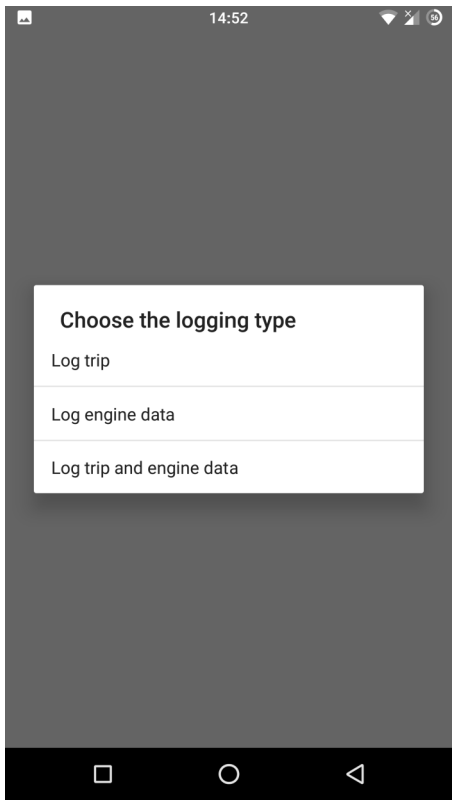
Lisa 4 – Veebirakenduse ekraanipildid

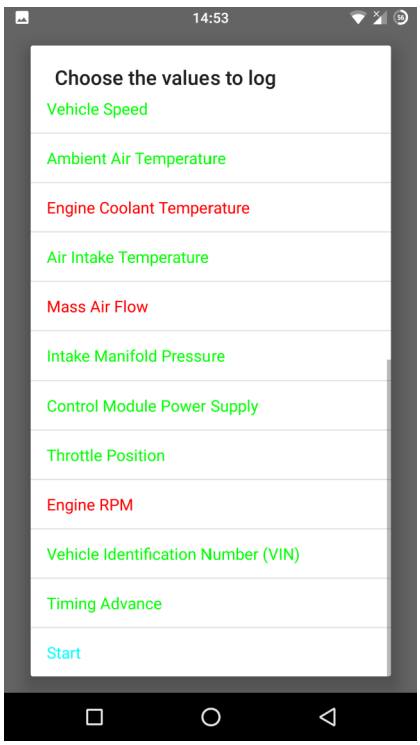


Rohkem pilte leiab giti repositooriumi documentation kaustast failist OBD2Logger-webapp.

Lisa 5 – Mobiilirakenduse ekraanipildid







Lisa 6 – Veebirakenduse asukoht

Veebirakendus on kättesaadav asukohas http://ec2-52-38-17-194.us-west-2.compute.amazonaws.com/