TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond
Tarkvarateaduse instituut

Marten Muru 123862IAPB

ELEKTRIJALGRATASTE MOODULI JA ANALÜÜTIKA VEEBIRAKENDUSE VAHELISE SUHTLUSE IMPLEMENTEERIMINE

Bakalaureusetöö

Juhendaja: Martin Rebane
MSc

Tallinn 2018

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: Marten Muru

14.05.2018

Annotatsioon

Selles töös tuleb vastu võtta ja saadavaks teha andmeid, mis saadetakse välja elektrijalgrattasse integreeritud moodulitest. Mooduli poolt saadetud andmed on sõltuvalt elektrijalgratta tootjast erinevad ja kuna oluline on jalgrataste hetkeolukorra jälgimine, siis andmeid liigub palju.

Lõputöö eesmärgiks oli luua töötav serverirakenduste süsteem, mis võtaks vastu moodulite poolt saadetud andmeid ja teeks need saadavaks jalgrataste andmete analüüsimise jaoks tehtud veebirakendusele läbi REST api.

Töö käigus loodi kaks serverirakendust, millest ühe eesmärk on andmeid vastu võtta ja talletada ning teise eesmärk on teenindada jalgrataste analüütika veebirakendust.

Lõputöö on kirjutatud eesti keeles ning sisaldab teksti 29 leheküljel, 34 peatükki, 13 joonist, 0 tabelit.

Abstract

Implementation of Communication Between an Analytics Web Application and a Module for Electric Vehicles

In this work it is important to receive and make available data, that is broadcasted by a module which is integrated into electric bicycles. The data that is broadcasted by the module differs depending on the manufacturer of the electric bicycle and since surveillance of the bicycle has to be conducted in real time, the amount of data being broadcasted is significant.

The goal of this thesis was to create a working system of backend applications, which would receive the data broadcasted by the modules and make it available using for the web application that was created for analyzing the data of bicycles using REST API.

As a result of this thesis two server applications were created. The purpose of one of them is to receive and store the data and the purpose of the other one is to service the bicycle analytics web application.

The thesis is in Estonian and contains 29 pages of text, 34 chapters, 13 figures, 0 tables.

Lühendite ja mõistete sõnastik

GCP	*Google Cloud Platform*
AWS	*Amazon Web Services*
GAE	*Google App Engine*
PaaS	*Platform as a Service*
API	*Application Program Interface.* Serveri osa, mis v]tab vastu ja saadab välja päringuid.
HTTP	*Hypertext Transfer Protocol.* Kommunikatsiooniprotokoll andmete saatmiseks ja vastuvõtmiseks internetis.
REST	*Representational State Transfer* API, mis kasutab HTTP päringuid andmete vastuvõtmiseks, väljasaatmiseks, muutmiseks ja kustutamiseks.
JSON	*JavaScript Object Notation.* Kergesti arusaadav ja väikese mahuga andmevahetusformaat.
CSV	*Comma-separated values.* Tekstifail, mis kasutab väärtuste eristamiseks komasid.
SQL	*Scripted Query Language*. Päringukeel relatsioonandmebaasi haldamiseks.
NoSQL	*No Scripted Query Language*. Andmebaas, mis ei ole relatsiooniline ja ei kasuta haldamiseks SQL lahendust.
ACID	*Atomicity, Consistency, Isolation, Durability.* Andmebaasi transaktsiooni standard, mis rahuldab neid nelja omadust.
GQL	*Google Query Language.* SQL laadne andmete pärimise meetod Google Datastore andmebaasist.
JAX-RS	Java API, mis pakub toetust REST veebiteenuste loomisele.
MQTT	*MQ Telemetry Transport.* Lihtne ja väikese mahuga sõnumite saatmise protokoll, mida kasutatakse ernevate masinate vahelise suhtluse loomiseks.

Servlet	Java serveripoolne programm, mis võtab vastu, töötleb ja vastab kliendi poolt tehtud päringutele.

Sisukord

1 Sissejuhatus 10
1.1 Taust 10
1.2 Probleem 11
1.3 Eesmärk 11
2 Planeerimine 13
2.1 Arhitektuuri planeerimine 13
2.1.1 Mikroteenused 15
2.1.2 Serverirakendus andmete käsitlemiseks 15
2.1.3 Serverirakendus kliendiga suhtlemiseks 17
2.2 Tehnoloogiate valik 18
2.2.1 Google Cloud Platform 18
2.2.2 Google App Engine 18
2.2.3 Pub/Sub 19
2.2.4 Datastore 20
2.2.5 BigQuery 20
2.2.6 Java 21
2.2.7 Gradle 21
2.2.8 Jersey 22
2.2.9 Objectify 22
3 Lahendus 23
3.1 Andmete haldamise serverirakenduse lahendus 23
3.1.1 Module-Data 26
3.1.2 Module-Module 29
3.1.3 Module-Type 30
3.1.4 Presentation 32
3.2 Analüütika veebirakenduse serverirakenduse lahendus 32
3.2.1 Autentimine 33
3.2.2 Erinevate klientide andmete eraldamine 34
3.2.3 Andmete haldamise serverirakendusega suhtlemine 34

4 Hinnang loodud lahendusele 37
4.1 Implementatsioonis esinenud keerukused 37
4.2 Tehtud töö tulemus 38
4.3 Edasise arengu võimalused 38
5 Kokkuvõte 39

Jooniste loetelu

Joonis 1. Planeeritud arhitektuuri üldine ülevaade. 14
Joonis 2. Pilt andmete haldamise serverirakenduse ülesehitusest. Module-Module, Module-Data ja Module-Type on mikroteenused. 24
Joonis 3. Mooduli andmete liikumise diagramm 25
Joonis 4. Filter meetod, mis kontrollib päringu ligipääsuõigust. 25
Joonis 5. ModuleParserService liidese kood 27
Joonis 6. Koodinäide andmebaasi salvestamise operatsioonist. 28
Joonis 7. Näide API meetodist ressursi klassist. 29
Joonis 8. *ModuleService* liides 30
Joonis 9. VehicleConfigService liidese poolt nõutud meetodid. 31
Joonis 10. Module-Data mikroteenuse servleti konfiguratsioon. 32
Joonis 11. Osa filter meetodist, mis viib läbi päringu autentimist. 33
Joonis 12. *SharingVehicle* objekti väljad 35
Joonis 13. Mooduli andmete liikumine läbi analüütika veebirakenduse serverirakenduse. 37

1 Sissejuhatus

Elektrijalgrataste müük on tänapäeval väga kiiresti kasvav äri. Paljud inimesed soovivad vältida linnas autoga liikumist ja kasutada keskkonnasõbralikumat transpordi varianti. Tänu sellele on elektrijalgrattad kogunud populaarsust arenenud riikides nagu Saksamaa ja Hiina.

Lõputöö esimeses osas vaadatakse üle autori poolt tehtud tehnoloogiavalikud ja kirjeldatakse arhitektuuri planeerimist. Põhjendatakse ära tehtud valikud ning arhitektuuri metoodikate valikud.

Teises osas on kirjeldatud lõputöö raames valminud rakenduse implementatsiooni. Autor toob välja lõputöö arhitektuuri erinevad osad ja kirjeldab iga osa eesmärki ning põhjendab kuidas iga osa aitab lahendada sissejuhatuses kirjeldatud probleeme.

1.1 Taust

Comodule [1] on firma, mis toodab platvormi kergete elektrisõidukite jaoks. Comodule-i eesmärk on toetada elektrijalgrataste arengut ja nende populaarsuse kasvu, tehes elektrijalgrataste kasutamise rohkem kasutajasõbralikumaks ja veelgi modernsemaks. Selle eesmärgi saavutamiseks ühendab Comodule elektrisõidukid asjade internetiga.

Infosüsteem jälgib sõiduki aku parameetreid ja kuvab informatsiooni kasutajale läbi IOS ning Android mobiilirakenduste. Üheks oluliseks infosüsteemi osaks on ka analüütika veebirakendus, mis on spetsiifiliselt loodud tootjate jaoks.

Mobiilirakendused on loodud lõppkasutajate jaoks, kes ostavad edale sõiduki, mis on varustatud Comodule tehnoloogiaga. Lõppkasutajad on jalgrataste tootjate kliendid. Kasutades mobiilirakendust on kasutajatel võimalik näha kasulikku informatsiooni nende enda ratta kohta ja lisaks sellele on ka lihtne ja mugav hoida sidet ratta tootjaga, kes saab kasutaja sõnumitele vastata kasutades tootjate jaoks loodud veebirakendust.

Analüütika veebirakendus rakendus korjab kokku spetsiifilise tootja info kõigi nende sõidukite hetkeolukorrast ja ajaloost, mis lubab tootjal teha paremaid arendusotsuseid ja

lisaks analüütikale pakub ka otsest ühendust nende lõppkasutajatega, tänu Comodule mobiilirakendustele. Jalgrataste tootjad on Comodule kliendid.

1.2 Probleem

Analüütika veebirakenduse jaoks on vaja serverirakendust, mis peab kokku korjama sisseloginud kliendi moodulite andmed Comodule infosüsteemist ja tegema need andmed kättesaadavaks veebirakendusele. Klient võib soovida analüütika veebirakenduse kasutamiseks ka mitut kontot, seega tuleb lahendada probleem sellisel viisil, et erinevatel kontodel võib olla täpselt sama, või mitu erinevat ligipääsu kliendi poolt grupeeritud andmetele.

Comodule infosüsteemi jaoks on vaja serverirakendust, mis oleks võimeline vastu võtma ja talletama kõikide jalgrataste andmeid. Elektrijalgratta tootjaid on väga palju. Sellepärast on antud serverirakenduses oluline andmebaasi ülesehitus ja andmete käsitlemine. Arvesse tuleb võtta, et andmed oleksid hoitud sellisel viisil, et igal kliendil oleks ligipääs ainult enda andmetele ja et andmed oleksid käsitletud vastavalt iga kliendi spetsiifilistele nõuetele.

See tähendab et serverirakendus Comodule infosüsteemi jaoks peab olema piisavalt dünaamiline, et vastu võtta paljude erinevate klientide spetsiifilisi andmeid ja serverirakendus analüütika platvormi jaoks peab olema spetsiifiline iga kliendi jaoks.

Selleks, et elektrijalgrataste asukohta ja hetkeseisukorda reaalajas jägida ning et hiljem oleks kõik need andmed saadaval ka ajaloona, tuleb serverirakendusel vastu võtta ja salvestada andmeid väga suurtes koguses.

1.3 Eesmärk

Lõputöö üldiseks eesmärgiks on luua kaks serverirakendust. Üks serverirakendus peab teenindama analüütika veebirakendust ja teine peab vastu võtma kõikide moodulite andmed ning tegema vastu võetud andmed kättesaadavaks kõikidele teistele Comodule infosüsteemi osadele.

Täpsemalt tuleb töö käigus täita järgmised eesmärgid:

- Luua serverirakendus andmete haldamiseks

- Luua serverirakendus analüütika veebirakenduse jaoks.
- Andmete haldamise rakendus peab olema suuteline vastu võtma ja edasi saatma suurtes kogustes andmeid.
- Andmete haldamise rakendus ei tohi välisel maailmal enda API-dele ligipääsu lubada.
- Andmeid tuleb hoida selliselt, et iga klient pääseks ligi ainult iseenda andmetele.
- Klient peab saama olema võimeline saama andmetele ligipääsu mitme erineva konto alt.
- Andmete haldamise rakendus peab olema võimeline vastu võtma erineva struktuuriga andmeid
- Rakendused peavad olema skaleeritavad.
- Rakendused peavad olema modulaarsed.
- Rakendused peavad olema võimelised ühilduma mistahes teiste Comodule serverirakenduste osadega.
- Lõputöö valmides peavad serverirakendused analüütika veebirakenduse jaoks tegema kättesaadavaks kogu andmete ajaloo ja moodulite hetkeseisu.

2 Planeerimine

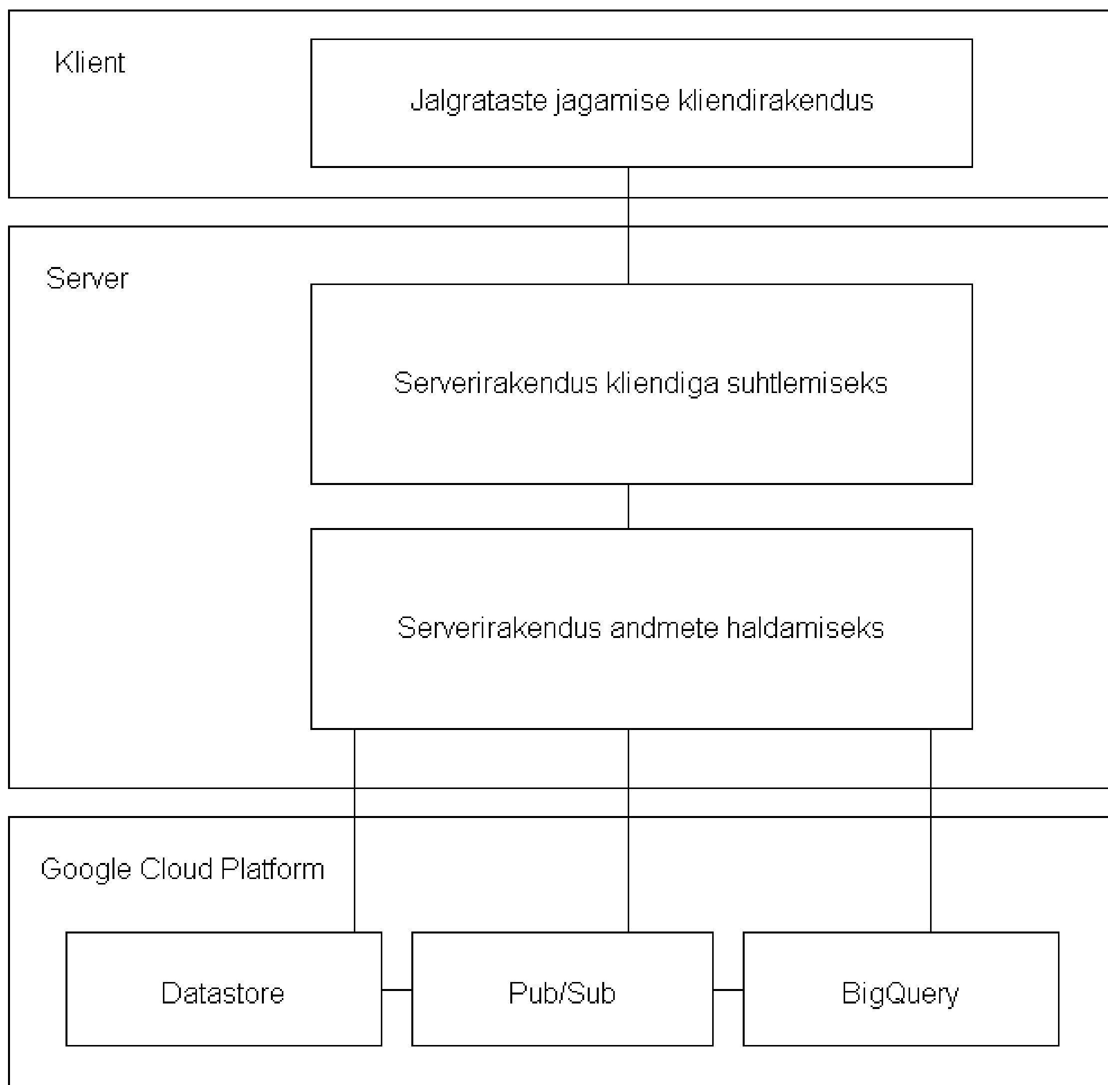
Antud peatükis on välja toodud kogu lõputöö arhitektuuri planeerimise osa, kus kirjeldatakse milline peaks välja nägema lõputöö valmimisel loodud projekt. Lisaks sellele on siin peatükis ka välja toodud lõputöös kasutatud tehnoloogiate valikud ja on põhjendatud miks iga tehnoloogiavalik selle lõputöö jaoks oluline on.

2.1 Arhitektuuri planeerimine

Käesolevas peatükis kirjeldab autor planeeritavat projekti arhitektuuri. Välja on toodud kõige olulisem arhitektuuristiili valik, selle kirjeldus ning põhjendus, miks valiti just selline arhitektuuristiil. Lisaks sellele on järgnevates alampeatükkides üldiselt kujutatud, milline projekti arhitektuur peaks valmides välja nägema.

Eesmärgiks on täita kõik nõuded mis on paika pandud eesmärkide peatükis. Pidades meeles modulaarsuse ja skaleeritavuse nõudeid, on plaanis implementeerida projekt kasutades *Google Cloud Platformi* (edaspidi GCP) [2] teenuseid ja mikroteenuste arhitektuuri. Põhilised GCP teenused, mida lõputöö eesmärkide täitmiseks tuleks ära kasutada on *Pub/Sub*, *Datastore* ja *BigQuery*. Alljärgneval joonisel (Joonis 1) on kujutatud üldine ülevaade, kuidas töö käigus implementeeritavad projektid üksteisega suhtlema peavad.

Andmete haldamise serverirakendus on plaanis üles ehitada kasutades mikroteenuste arhitektuuri. Mikroteenuste arhitektuuri ei ole plaanis kasutada analüütika platvormi serverirakenduse puhul, kuna selle peamine eesmärk on teenindada analüütikaplatvormi, kasutades selle lõputöö puhul ainult andmete haldamise serverirakendust, kuid tulevikus ka teisi Comodule infosüsteemi osasid. Mikroteenuseid ja nende implementatsiooni kirjeldatakse lähemalt lahenduse dokumentatsiooni peatükis.



Joonis 1. Planeeritud arhitektuuri üldine ülevaade.

Esimese asjana peaksid saabuma mooduli andmed andmete haldamise serverirakenduse API meetodisse. See serverirakendus peab andmed ümber struktureerima JSON formaadile, salvestama GCP andmebaasi ja seejärel edastama andmed teistele Comodule infosüsteemi osadele.

Klient peaks autenditud klient saama läbi kliendiga suhtlemiseks tehtud serverirakenduse API meetodied kasutuades ära kasutada andmeid API meetodeid, mis on loodud andmete haldamise serverirakenduse poolt, et ühendust võtta GCP andmebaasiga.

Töö implementeerimise protsess peaks välja nägema selline:

- Implementatsiooni planeerimine.
- Andmete käsitlemise serverirakenduse loomine.
- Andmete väljasaatmise serverirakenduse loomine.

2.1.1 Mikroteenused

Mikroteenuste [3] arhitektuur on hiljuti populaarsust kogunud meetod tarkvarasüsteemide arendamiseks. Kuigi hetkel puudub täpne definitsioon, mida see arhitektuuristiil endast kujutab, on sellel siiski mõned kindlad defineerivad omadused. Mikroteenuste stiil, on projekti arhitektuuriline lähenemine, kus luuakse rakendus, mis koosneb paljudest väikestest teenustest, millest igaüks täidab ühte kindlat ja väikest eesmärki ning mis suhtlevad üksteisega kastuades väga lihtsaid mehhanisme, nagu näiteks REST üle HTTP [3].

Monoliitse arhitektuuri stiili puhul tuleb iga kord, kui projektis midagi muudetakse, uuendada kogu serveripoolset rakendust. Võrreldes monoliitse stiiliga, mille puhul tüüpiliselt ehitatakse üles serverirakendus ühe suure projektina, on mikroteenuste arhitektuuristiiliga ehitatud projekt palju modulaarsem, kuna ühe kindla ülesande täitmises eksisteerib eraldi rakendus, mida saab arendada ilma ülejäänud projekti segamata [3].

Tänu mikroteenuste modulaarsusele, on selline arhitektuuristiil ideaalne käesoleva lõputöö implementatsiooniks. Mikroteenuste modulaarsus täidab antud lõputöös kahte nõuet. See teeb projekti rohkem skaleeritavaks ja modulaarseks. Modulaarsus tuleb mikroteenuste arhitektuuri põhimõttest, kuna see nõuab väikeseid, iseseisvaid teenuseid, mida saab lihtsalt ära kustutada, juurde lisada või muuta. Skaleeritavus tuleneb modulaarsusest, sest tänu modulaarsusele on väga lihtne projekti tulevikus muutuvate nõuete jaoks lisaarendust läbi viia.

2.1.2 Serverirakendus andmete käsitlemiseks

Serverirakenduse loomine andmete käsitlemiseks on lõputöö esimeseks eesmärgiks. Selle rakenduse ülesanne on vastu võtta, hoida ja ülejäänud Comodule infosüsteemi jaoks saadavaks teha andmeid, mis saadetakse välja elektrijalgrattasse paigaldatud moodulilt.

Kõigepealt tuleb luua mikroteenus, mille ülesanne oleks andmeid vastu võtta, need läbi töödelda ja andmebaasi salvestada. Moodul võib andmeid välja saata ükskõik millal ja mikroteenus peab alati olema valmis andmeid vastu võtma. Andmed tulevad mikroteenusesse CSV stringina, kus ridu võib olla rohkem kui üks ja iga rida võib sisaldada ükskõik kui palju andmeid. Andmete arv sõltub kliendist ja iga arvväärtuse tähendus sõltub riistvara konfiguratsioonist.

Andmete arv, mida moodul korjab on suur ning selle rakenduse nõuetes on oluliseks osaks suurtes kogustes andmete kiire pärimine. Selle lahendamiseks on plaanis ära kasutada *Google Cloud BigQuery* teenust. Nagu kasutatude tehnoloogiate peatükis sai mainitud, on *Google Cloud BigQuery* spetsiaalselt sellise probleemi lahendamiseks tehtud. Seal saab hoida lõputult suures koguses andmeid, ning nende pärimine ei võta üle mõne sekundi ja andmete koguse suurenemisel andmete hoidmise ja pärimise skaleeritavuse eest hoolitseb *Google Cloud BigQuery* automaatselt [4].. Vastavalt andmete kogusele tehakse Google-i poolt andmete töötlemiseks ja hoidmiseks saadavaks vajalik arv ressursse [4].

Päringuid, mida tehakse tihedalt, on plaanis liigutada kasutades *Pub/Sub* teenust. Selle teenuse abil hoolitseb andmeliikluse skaleeritavuse eest GCP. Päringud, mida ei tehta nii tihedalt, on plaanis liigutada kasutades tüüpilist REST API-t üle HTTP.

Kuna serverirakenduses läheb vaja konfiguratsiooni loomist, hoidmist ja kasutamist, siis tuleb luua vastav mikroteenus, mis selle eest hoolitseb. Mikroteenus peaks sisaldama API-t, mille kaudu oleks võimalik luua kindla kliendi jaoks riistvara konfiguratsiooni, mis defineerib ära mooduli poolt välja saadetud andmete tähenduse. Sama API kaudu peaks olema võimalik ka konfiguratsioone küsida.

Selleks, et iga moodul teaks, mis riistvara konfiguratsiooni versiooniga ta seotud on, peab andmebaasi olema salvestatud mooduli objekt, iga füüsilise mooduli kohta. Lisaks sellele eesmärgile, peab see objekt sisaldama ka väärtuseid, mis on vajalik mooduli peal oleva tarkvara jaoks ja ka üldiseks olemasolevate moodulite haldamiseks. Selle jaoks tuleks luua kolmas mikroteenus, mis lahendab need probleemid.

Seega lisaks põhieesmärkidele peab andmete käsitlemise serverirakendus täitma järgmiseid alameesmärke:

- Rakendus peab lubama sisestada ja küsida informatsiooni sõiduki tüübi kohta vastavalt mooduli riistvara versioonile.
- Rakendus peab lubama sisestada ja küsida informatsiooni iga mooduli kohta.

2.1.3 Serverirakendus kliendiga suhtlemiseks

Serverirakenduse loomine kliendiga suhtlemiseks on teiseks lõputöö eesmärgiks. Andmete väljastamiseks on vajalik eraldi rakendus sellepärast, et üheks lõputöö nõudeks on tingimus, et serverirakendus andmete käsitlemiseks peab oma API ligipääsu välismaailmale kinni panema. Andmete väljastamise serverirakendus asub samas *AppEngine* keskkonnas ja seega on tema päringu päises identifikaator, mida kontrollib filter andmete käsitlemise serverirakenduses.

Serverirakendus andmete väljasaatmiseks on põhimõtteliselt värav, mis lubab piiratud ligipääsu andmete käsitlemise serverirakendusse. API ligipääs on olemas ainult nendele spetsiifilistele REST meetoditele, mis väraval vaja on. Ülejäänud REST API meetodid on mõeldud serverirakenduse siseseks tegevuseks.

Selle lõputöö põhieesmärgi raames peab andmete väljasaatmise serverirakendus analüütika veebirakenduse jaoks saadavaks tegema kogu moodulite andmete ajaloo ja kõikide moodulite hetkeseisu.

2.2 Tehnoloogiate valik

Käesoleva projekti loomiseks on valitud sellised tehnoloogiad, mis lahendaksid keerulisemad skaleeritavuse probleemid automaatselt. Selle jaoks on lõputöö implementeerimiseks valitud tehnoloogiad enamjaolt seotud GCP poolt pakutavate teenustega.

2.2.1 Google Cloud Platform

Lõputöö Serverirakenduste aluseks saab olema GCP. Kuigi kõige populaarsem pilveteenuse pakkuja on AWS, on lõputöö implementeerimiseks valitud GCP, kuna tegemist on pilveteenusega, mis on tulevikukindlam tänu unikaalsele infrastruktuurile, mida suudab pakkuda ainult Google [5]. Samuti on GCP odavam kui AWS [6].

GCP aluseks on nii füüsiline riistvara nagu arvutid ja kõvakettad kui ka virtuaalsed ressurssid nagu virtuaalmasinad, mis asetsevad Google-i poolt loodud andmekeskustes üle kogu maailma. Andmekeskused on hetkel jagatud seitsmeteistkümnesse globaalsesse regiooni. Regioonid on veel omakorda jaotatud tsoonideks, mis on üksteistest isoleeritud oma regiooni siseselt ja igaüks neist sisaldab oma füüsilist andmekeskust. Tänu sellele on garanteeritud, et kui ühes andmekeskuses midagi valesti läheb, siis teine andmekeskus võtab töö üle.

GCP on Google-i poolt loodud pilveandmetöötluse keskkond mis pakub mitmeid teenuseid, mis lubavad arendajatel pääseda ligi GCP füüsilistele ja virtuaalsetele ressurssidele. Kogu seda sama infrastruktuuri kasutavad ka Google-i enda rakendused, nagu näiteks Youtube ja Google otsingumootor. Google-i poolt loodud infrastruktuur lubab arendajatel ehitada rakendusi, mis on kindlad, turvalised ja skaleeritavad. Need omadused on ideaalsed selle lõputöö eesmärgi täitmiseks ning sellel põhjusel on Comodule infosüsteemi ehitamiseks valitud just see tehnoloogia.

2.2.2 Google App Engine

Google App Engine on GCP poolt pakutav PaaS teenus, mis lubab arendajal valmiskirjutatud koodi väga lihtsalt Google-i poolt pakutud serverisse üles laadida. Rakenduse üles laadimiseks pole vaja servereid ise luua, selle eest hoolitseb GAE [7]. Rakenduse kasutamiseks tuleb lihtsalt kood valmis kirjutada ja see GAE teenusesse üles laadida [7].

GAE kasutab GCP infrastruktuuri automaatselt proportsionaalselt rakenduse kasutajate hulgaga [7] [8]. See tähendab, et rakendus, mis on loodud kasutades GAE teenust skaleerib automaatselt vastavalt kasutajate arvule.

2.2.3 Pub/Sub

Pub/Sub on andmevahetuse muster, mis lahutab sõnumite saatja ja vastuvõtja täielikult [9]. Sõnumite saatja ei tea sõnumite vastuvõtjast mitte midagi ja samamoodi ei tea ka sõnumite vastuvõtja mitte midagi sõnumite saatjast. Seega, vastuvõtjad saavad ainult andmeid, mis neile huvi pakuvad, teadmata mis saatjad, kui neid üldse on, eksisteerivad. Pub/Sub muster pakub suuremat skaleeritavust ja dünaamilisust andmevahetuse käsitlemisel [9].

Google Pub/Sub teenus on ehitatud spetsiifiliselt pilveteenuste jaoks. See lubab mitmel iseseisval rakendusel üksteise vahel asünkroonselt suhelda. Omalt poolt lisab google enda *Pub/Sub* teenusesse ka turvalisuse ja garantii, et iga sõnum jõuab kohale [9]. Kui mõni sõnum ei jõua mingil põhjusel vastuvõtjani, siis sõnum salvestatakse Google-i poolt ja proovitakse saata uuesti senikaua, kuni sõnum kohale jõuab.

Sõnumite filtreerimiseks on kasutusel teemadel põhinev süsteem. Süsteem töötab nii, et sõnumite saatja defineerib teema ja võib saata sinna sõnumeid. Sõnumite kuulajad saavad ennast selle teema alla registreerida, kui sõnumi kuulaja on ennast mõne teema alla registreerunud, siis kõik sõnumid, mis sinna teemasse saadetakse, jõuavad sõnumi kuulajani.

Pub/Sub mustri eelised:

- Madal sidestus
- Skaleeritavus
- Puhtam kood
- Paindlikkus
- Kergesti testitav

Pub/Sub mustri puudused:

- Sõnumi saatja ja vastuvõtja ei tea üksteise seisukorrast
- Vahendaja on vajalik ja lisab keerukust

- Vahendaja ei pruugi sõnumi kohaletoimetamisest teada anda

Pub/Sub tehnoloogia eeliseid saab ära kasutada käesolevas lõputöös tehtava rakenduse modulaarsuse ja skaleeritavuse nõude täitmiseks.

2.2.4 Datastore

Google Cloud Datastore on NoSQL andmebaasi lahendus, mis põhineb pilvetehnoloogial [10]. *Datastore* on üles ehitatud selliselt, et ta skaleerib automaatselt vastavalt andmete hulgale [10]. Mida rohkem andmeid hoiad, seda rohkem asi maksab, kuid ülejäänu eest hoolitseb Google.

Andmebaasi sisu on automaatselt jagatud mitme serveri vahel ning suured tabelid on jagatud mitmeteks väiksemateks tabeliteks, mis kasutavad ära eelnevalt mainitud mitmeid servereid [10]. Tänu sellele on *Datastore* tehnoloogia väga skaleeritav ning salvestatud andmetele ligipääs on väga kiire.

Datastore toetab SQL-i sarnaseid päringuid tänu GQL olemasolule, ACID transaktsioone ja REST API-t [10]. Kuna tegemist on NoSQL andmebaasiga, siis andmebaasi skeemi pole vaja, kuid erinevalt teistest NoSQL lahendustest on *Datastore*-ga võimalik teha komplekseid päringuid kasutades GQL tehnoloogiat. Samas andmebaasi loomine on SQL lahendustega võrreldes väga lihtne, tuleb lihtsalt luua objekt ja see panna andmebaasi. Samas on see andmebaas ka väga paindlik, näiteks on võimalik andmebaasi salvestatavate objektide küljes hoida mitut väärtust ühes väljas.

Tänu tehnoloogia skaleeritavusele, lihtsusele ja kiirusele, on *Google Cloud Datastore* ideaalne tehnoloogia kõikide moodulite hetkeseisukorra andmete hoidmiseks.

2.2.5 BigQuery

Google Cloud BigQuery on skaleeritav ja madalate kuludega teenus, mis kasutab ära Google serveritut arhitektuuri, automaatiseeritud skaleerimist ja suurt jõudlust, et luua spetsiaalselt suurte andmekoguste hoidmiseks mõeldud andmebaas [4]. Kasutades *BigQueryt* on võimalik odavalt salvestada väga suuri andmekogumikke.

Päringute tegemine *BigQuerys* toimib läbi SQL päringute reaalajas. Kasutades Google-i olemasolevat infrastruktuuri töö tegemiseks, jooksutatakse päring paralleelselt üle

mitme kõvaketta [11]. Tänu sellele on päringud kiired, võrreldes teiste sarnaste andmebaasilahendustega.

Tänu *Google Cloud BigQuery* omadustele, on see teenus ideaalne suurte andmevoogude salvestamiseks, nagu näiteks andmevoog aktiivsest sensorist elektrisõiduki küljes. Seejärel on võimalik üle kogu salvestatud andmehulga jooksutada kiireid SQL päringuid, mis skaleerivad automaatselt vastavalt andmehulgale ja päringu tegijatele. Kiired päringud lubavad omakorda teha suurte andmekoguste analüütikat reaalajas, näiteks graafikute joonistamine.

Käesolevas lõpuöös on suurte andmekoguste salvestamine ja analüüsimine üks nõuetest. Seega on *Google Cloud BigQuery* ideaalne mooduli poolt saadetud andmete ajaloo talletamiseks. Samuti ei pea muretsema andmemahu skaleerimise pärast kuna selle eest hoolitseb Google.

2.2.6 Java

Java programmerimiskeel on klassidel põhinev ja objektorienteeritud keel, see on disainitud sellisena, et see oleks võimalikult lihtne keel ning võimaldaks tänu sellele paljudel programmeerijatel sujuvalt koodi kirjutamine selgeks õppida [12]. Java programmeerimiskeel on sarnane C ja C++ programmeerimiskeeltega ja on seega suhteliselt kõrege tasemega keel [12]. Programmid, mis on selles keeles kirjutatud, tuleb enne käima panemist kompileerida masinatele arusaadavaks keeleks [12].

Peamine põhjus, miks lõputöö projekt on implementeeritud java keeles, on *Google Appengine* piirang programmeerimiskeele kasutamisel, kus Java on üks vähestest lubatud keeltest. Põhjus Miks nendest keeltest sai valitud just Java keel tuleb sellest, et kogu firmasisene infosüsteem on kirjutatud javas ning ka autor ise on selle keelega kõige paremini tuttav.

2.2.7 Gradle

Gradle on tööriist, mis on mõeldud java projektide ehituse automatiseerimiseks [13], tänu millele muutub tarkvaraprojekti ehitamine natuke lihtsamaks. Gradle lubab iga projekti külge lisada ehitusskripti faili, millel on mitmeid eeliseid [13].

Ehitusskripti üheks suurimaks eeliseks on sõltuvuste haldamine. Tänu Gradle töörista kasutamisele, ei pea projekti ehitamisel muretsema sõltuvuste allalaadimise ja nende projekti installimise pärast - kogu selle töö teeb ära Gradle.

Teiseks ehitusskripti kasutamise suureks eeliseks on tugi, mida see pakub projekti struktuuridele, mis sisaldavad rohkem kui ühte projekti, kuna iga projekti külge on võimalik lisada ehitusskripti fail ja läbi selle on võimalik projektide vahel lihtsalt luua sõltuvuste hierarhia.

Käesoleva lõputöö projekti implementeerimiseks on Gradle kasulik mõlema eelnevalt mainitud omaduse tõttu. Sõltuvuste haldamise automatiseerimine on kasulik iga projekti jaoks, kuid kuna lõputöö projekti implementeerimisel on planeeritud kasutada mikroteenuste arhitektuuri, siis teine mainitud Gradle eelis on peamine põhjus, miks antud lõputöös seda kasutatakse.

2.2.8 Jersey

Jersey on REST tehnoloogial põhinev veebiteenuste raamistik, mis täiendab juba olemasolevat JAX-RS API-t [14]. Selle mõte on REST tehnoloogial põhinevate veebiteenuste loomise lihtsustamine Java programmeerimiskeeles.

Tänu Jersey poolt loodud annotatsioonidele võtab REST API-de loomine palju vähem aega ja koodi. Antud lõputöös tuleb suhtlus luua kasutades REST tehnoloogiat, et edasi saata JSON sõnumeid. sellel põhjusel on kasutusele võetud Jersey raamistik.

2.2.9 Objectify

Objectify on *Datastore* teenuse jaoks loodud raamistik, mis pakub meetodeid andmete salvestamiseks ja andmebaasi päringute tegemiseks. Ilma selle raamistikuta võtaks andmebaasi operatsioonide implementeerimine rohkem aega ja koodi.

Antud lõputöös kasutatakse antud raamistiku puhtalt sellepärast, et aega kokku hoida ja koodi puhtana hoida.

3 Lahendus

Alljärgnevalt on dokumenteeritud lõputöö käigus loodud serverirakenduste implementatsioon. Esmalt on kirjeldatud andmete käsitlemise serverirakenduse implementatsiooni ning seejärel andmete jagamise serverirakenduse implementatsiooni.

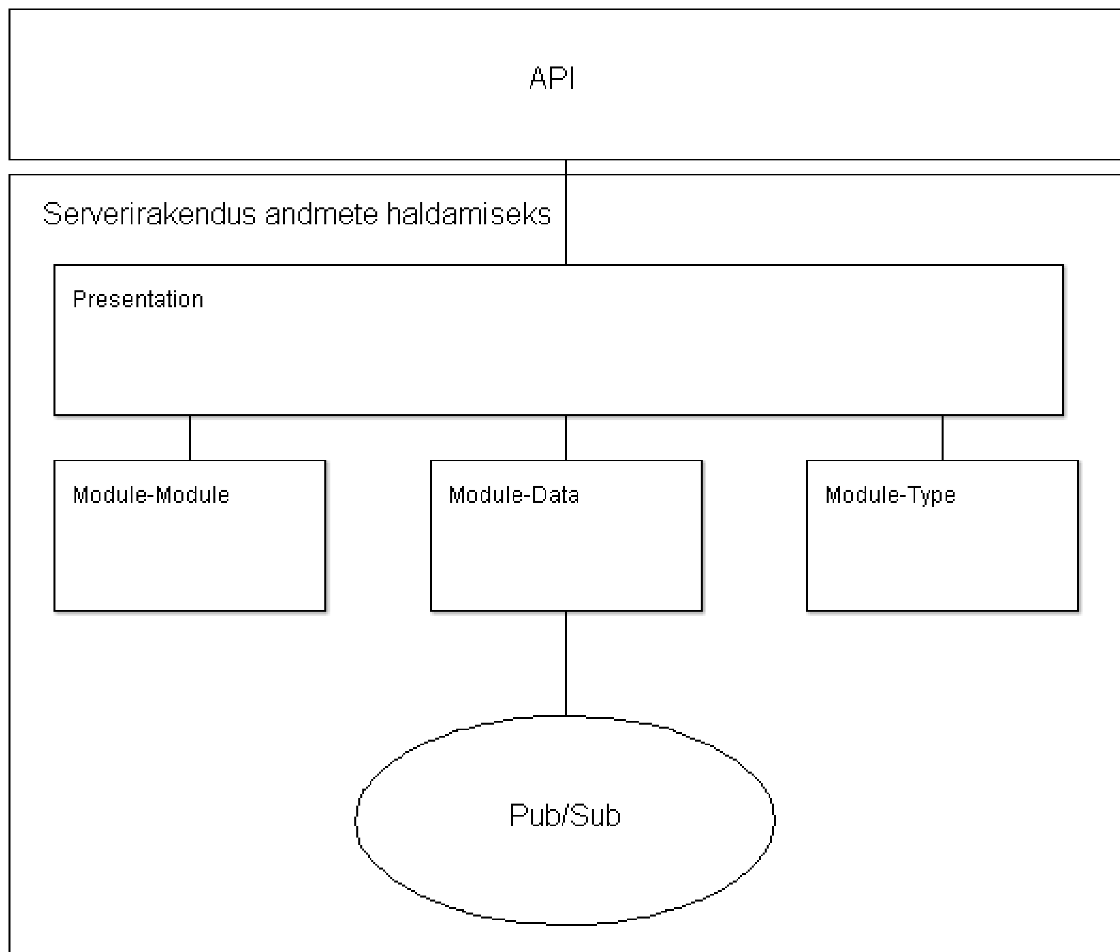
Alampeatükid on jagatud mikroteenuste ja teiste oluliste projekti osade kaupa, kuna iga mikroteenus peab täitma ainult ühte kindlat rolli on nii kõige parem kirjeldada lõputöös loodud rakendusi. Iga mikroteenuse puhul on ära kirjeldatud kõige olulisemad klassid ja nendes klassides olevad kõige olulisemad meetodid. Meetodid ja klassid, mida spetsiifiliselt kirjeldatud ei ole, eksisteerivad eraldi lihtsalt selleks, et koodi puhtamana hoida.

Ideaalis oleks iga mikroteenus GAE platvormi üles laetud eraldi rakendusena, kuid hetkel aja ning ressursside kokkuhoiu huvides on tehtud 2 rakendust, mis koosnevad mitmest projektist ning iga projekt rakenduse sees on eraldi mikroteenus.

3.1 Andmete haldamise serverirakenduse lahendus

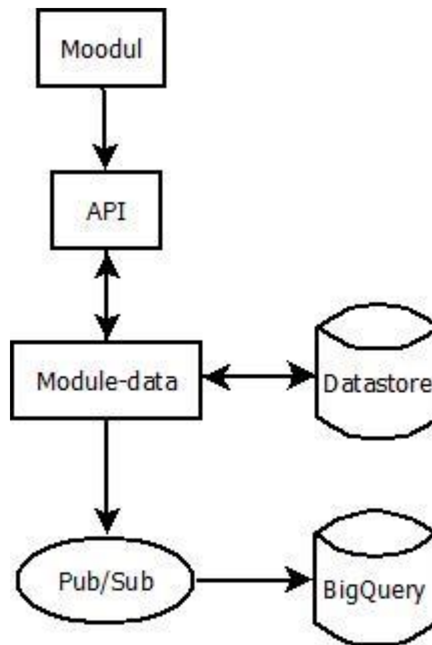
Andmete haldamise serverirakendus on üks rakendus, mis koosneb neljast modulaarsest projektist. Rakenduse koosneb ühest presentatsiooni projektist, mis tegeleb ainult päringute suunamisega õigesse projekti. Projektid *Module-Module*, *Module-Data* ja *Module-Type*, mis on nimetatud projekti poolt käsitletud objektide järgi, on mikroteenused (Joonis 2). Tegemist on nelja projektiga, mis laetakse GAE serverisse üles ühe rakendusena.

Mikroteenuste projektides on implementeeritud kogu projekti loogika. Iga mikroteenus tegeleb ühe spetsiifilise ülesandega. Nagu joonisel (Joonis 2) on näha, siis mikroteenused ei ole üksteisega mitte kuidagi seotud. Kui üks mikroteenus tahab teiselt midagi küsida, siis seda ei tehta mitte projekti siseselt, vaid tehakse päring läbi API, mis seejärel suunatakse presentatsiooni projektis õigese mikroteenuse projekti. Tänu sellele, kui tulevikus on soov mikroteenused eraldi rakendustena GAE platvormi üles laadida, siis see on väga lihtsasti teostatav.



Joonis 2. Pilt andmete haldamise serverirakenduse ülesehitusest. Module-Module, Module-Data ja Module-Type on mikroteenused.

Module-Module ja *Module-Type* mikroteenused on projektid, mis on loodud *Module-Data* mikroteenuse toetamiseks. Mooduli poolt saadetud andmed liiguvad ainult läbi *Module-Data* mikroteenuse. Mooduli andmete liikumise diagramm on kujutatud joonisel (Joonis 3).



Joonis 3. Mooduli andmete liikumise diagramm

Käesolevas peatükis on täpsemalt kirjeldatud iga mikroteenust, mis lõputöö raames loodi, kuid igal mikroteenusel on siiski mõned klassid, mis korduvad, sest iga mikroteenus peab täitma samu turvanõudeid.

Igas mikroteenuses tagab API turvalisuse kood, mis asub *PlatformFilter* klassis. Klass sisaldab ainult ühte meetodit (Joonis 4), mis kontrollib, et API-t välja kutsuva päringu päises sisaldub *X-Appengine-Inbound-Appid* väli ja et see väli oleks võrdne projekti id väärtusega. Antud väli on päringu päises ainult siis, kui päring tuleb mõnest teisest GAE projektist ja välja sisu on päringu teinud projekti id. Filtri meetod kutsutakse välja enne kui päring API meetodi implementatsiooni jõuab, seega päring kas võetakse vastu või lükatakse tagasi enne kui päringut täitma hakatakse.

```
public void filter(ContainerRequestContext requestContext) throws IOException
{
    databaseManager.setNamespace(DatabaseManagerImpl.DEFAULT_NAMESPACE);

    String appId = requestContext.getHeaders().getFirst(HEADER_KEY_APP_ID);
    if (appId == null || !appId.equals(PROJECT_ID))
        throw new NotAuthorizedException("application not authorized");
}
```

Joonis 4. Filter meetod, mis kontrollib päringu ligipääsuõigust.

Nagu joonisel (Joonis 4) on näha, siis see meetod sisaldab ka *setNamespace* meetodit *DatabaseManager* klassist. See klass on liides, mida implementeerib

DatabaseManagerImpl klass. Meetodi ülesanne on nimeruumi muutmine vastavalt mikroteenusele, mis tähendab et igal mikroteenusel, mis on andmete haldamise serverirakenduses, on oma enda nimeruum andmete hoidmiseks. Tänu sellele tehakse kõik *Datastore* andmebaasioperatsioonid ühes kindlas nimeruumis, mis pannakse paika kohe kui päring serverisse jõuab ja enne kui andmebaasi operatsioone hakatakse teostama.

3.1.1 Module-Data

Module-Data mikroteenus on projekt, mida läbivad kõik mooduli poolt välja saadetud andmed. Selles peatükis on kirjeldatud olulisemaid klasse, nende meetodeid ja on selgitatud millisel moel need aitavad täita sissejuhatuses paika pandud eesmärke.

Antud mikroteenus koosneb järgnevatest olulistest klassidest:

- DataResource
- ModuleResource
- PubSubResource
- ModuleParserService
- ModuleParserServiceImpl

Mikroteenuse eesmärk on läbi viia järgnevat protsessi:

1. Võtab vastu mooduli poolt välja saadetud andmed.
2. Teeb päringu *Module-Module* mikroteenusesse, mis tagastab andmed mooduli omadustest.
3. Kasutab *Module-Module* mikroteenusest saadud vastusest võetud objekti küljes olevat püsivara versiooni numbrit, et teha päring *Module-Type* mikroteenusesse.
4. Kasutab *Module-Type* mikroteenusesse tehtud päringu vastusest loetud konfiguratsiooni, et tuvastada mooduli poolt välja saadetud andmete tähendus.
5. Saadab JSON formaati teisendatud andmed *Pub/Sub* teenusesse.
6. Salvestab JSON formaati teisendatud andmed *Datastore* teenusesse.
7. Teeb *Datastore* teenusesse salvestatud andmed saadavaks läbi API.

Selleks, et neid eesmärke edukalt täita, sisaldab *Module-Data* mikroteenus, *ModuleParserService* liidest (Joonis 5) ja ressursiklasse, mis sisaldavad andmete kättesaamiseks API meetodeid.

```
public interface ModuleParserService {
    void handleDataMessage(String csv, String chipId);
    void handleStatusMessage(MqttData mqttData);
    JsonObject getModuleStatus(String chipId);
    JsonObject getModuleData(String chipId);
    JsonObject getModuleEvents(String chipId);
    JsonArray getDataList(List<String> ids);
    JsonArray getStatusList(List<String> ids);
}
```

Joonis 5. ModuleParserService liidese kood

Nagu joonisel on näha, siis meetodid ei tagasta kindlalt defineeritud objekti. Selle asemel liiguvad andmed JSON objektidena. Põhjuseks, miks andmed just nii liiguvad on see, et meile ei ole teada millisel kujul andmed rakendusesse sisenevad. Meile on teada ainult CSV string, mis võib sisaldada ükskõik kui palju andmeid ja nende andmete tähendus on iga kliendi puhul erinev. Andmete hulk CSV stringis on sama kõikide moodulite puhul, millel on sama püsivara versioon. Seda liidest implementeerib *ModuleParserServiceImpl* klass.

handleDataMessage meetodi kutsub välja *Pub/Sub* teenus. Seejärel võtab see meetod vastu *Pub/Sub* teenuse poolt saadetud CSV stringi ja teeb sellest JSON objekti, kasutades informatsiooni *module-module* ja *module-type* mikroteenustest. Valmis tehtud JSON objekt saadetakse edasi *Pub/Sub* teenusesse, kus üheks kuulajaks on mikroteenus, mis võtab andmed vastu ja salvestab need kõik *BigQuery* teenusesse. Lisaks sellele salvestab *handleDataMessage* teenus kõige hilisemad andmed *DataStore* teenusesse.

Oluline on ka välja tuua, et kuna *Datastore* on *noSQL* andmebaasiteenus, on võimalik andmeid salvestada ilma andmebaasi eelneva konfigureerimiseta. Salvestatud andmete andmebaasi struktuur pannakse paika või muudetakse samal ajal kui uus objekt andmebaasi salvestatakse. See lubab andmebaasi salvestada uusi andmeid, ilma andmebaasi struktuuri muutmiseta. Kasutades ära seda omadust, on võimalik salvestada erinevate klientide poolt saadetud andmeid, teadmata nende täpset sisu. Ehk kui süsteemi hakkab kasutama uus klient, kes tahab andmebaasis hoida uusi andmeid, mida mitte ükski eelnev klient pole kunagi hoida soovinud, siis väljad nende andmete jaoks luuakse automaatselt andmete salvestuse operatsiooni käigus, vastavalt andmete konfiguratsioonifailile ja andmete hulgale (Joonis 6).

```
Entity entity = getOrCreateModuleEntity(chipId);
for (int i = 0; i < csvArray.length; i++) {
	String name = getValueNameFromConfig(config, i);
	String multiplier = getMultiplierValueFromConfig(config, i);
	String value = csvArray[i];

	if (value != null && !Objects.equals(value, EMPTY_STRING)) {
		if (multiplier != null && !multiplier.equals(EMPTY_STRING))
			value = String.valueOf(Double.valueOf(value)
				* Double.valueOf(multiplier));

		createEntityValueProperty(entity, timestamp, name, value);
	}
}
ds.put(entity);
```

Joonis 6. Koodinäide andmebaasi salvestamise operatsioonist.

handleStatusMessage meetodi kutsub samuti välja *Pub/Sub* teenus. See on vajalik sellepärast, et ühe teise mikroteenuse kaudu, mis ei ole seotud antud lõputööga, on võimalik saata moodulile käske, näiteks mooduli sisse ja välja lülitamiseks. Kui selline käsk saadetakse, siis moodul saadab *Pub/Sub* teenusesse sellest sõnumi, mis sisaldab CSV stringi koos muutunud staatuste väärtustega. CSV töödeldakse läbi, mille tulemusena luuakse kaks JSON obekti. Üks objekt kõikidest mooduli praegustest staatustest ja teine staatuse muutmise sündmusest.

Mõlema meetodi puhul saadetakse läbi töödeldud informatsioon JSON kujul *Pub/Sub* teenusesse ja salvestatakse otse *Datastore* teenusesse. Andmed, mis liiguvad tihedalt saadetakse alati *Pub/Sub* teenuse kaudu edasi, sest siis hoolitseb Google arhitektuur selle skaleeritavuse eest. Kui andmeid tuleb sisse rohkem ja sõnumeid on palju, siis Pub/Sub teenus automaatselt suurendab nii füüsiliste kui ka virtuaalsete ressursside arvu selle andmehulga käsitlemiseks. Projekti omanik peab muretsema ainult suurenenud rahakulu eest.

Ülejäänud meetodid, mida *ModuleParserService* liides nõuab ja *ModuleParserServiceImpl* implementeerib on loodud API GET meetodite realiseerimiseks ja need tegelevad andmebaasist JSON objekti pärimisega ja seejärel ressursiklassidele saatmisega. Ressursiklassid sisaldavad API meetodeid (Joonis 7), mis on implementeeritud kasutades *Jersey* raamistikku. Kõik API meetodid tagastavad JSON formaadis objekte.

```
@GET
@Path("/data")
@Produces(MediaType.APPLICATION_JSON)
public JsonArray getDataListByIds(@QueryParam("id") List<String> ids) {
    return moduleParserService.getDataList(ids);
}
```

Joonis 7. Näide API meetodist ressursi klassist.

Ressursiklasse on kolm:

- *DataResource* sisaldab API meetodeid mitme mooduli andmete küsimiseks.
- *ModuleResource* sisaldab API meetodeid ainult ühe mooduli andmete küsimiseks.
- *PubSubResource* sisaldab API meetodeid, mis on mõeldud kasutamiseks ainult Google *Pub/Sub* teenusele

Antud mikroteenus täidab järgnevaid sissejuhatuses püstitatud eesmärke:

- Mikroteenus on suuteline vastu võtma ja edasi saatma suurtes kogustes andmeid, tänu *Pub/Sub* teenusele.
- Mikroteenus ei luba välisel maailmal enda API ressurssidele ligipääsu tänu filtrile, mis kontrollib, et päring tuleb meie projektisisesest mikroteenusest.
- Mikroteenus on võimeline väga lihtsalt ja skaleeritavalt võimeline ühenduma teiste Comodule serverirakenduste osadega tänu Pub/Sub teenuse omadustele ja väga lihtsale API ülesehitusele.
- Serverirakendus hoiab *Datastore* teenuses iga mooduli hetkeseisu.

3.1.2 Module-Module

Module-Module mikroteenus on loodud selle eesmärgiga, et siia saaks registreerida kõik moodulid mis olemas on ja mis tulevikus luuakse ning nende kohta informatsiooni pärimiseks. Moodul vajab API-t, millega ta saaks ise automaatselt ühendust võtta ja oma informatsiooni andmebaasi salvestada. Sellise mikroteenuse loomine on üheks lõputöö alameesmärgiks ning selle põhjuseks on iga spetsiifilise mooduli tüübi olulisus andmete vastuvõtmisel *Module-Data* mikroteenuses.

Olulised klassid antud mikroteenuses on järgmised:

- ModuleResource
- ModuleService

- ModuleServiceImpl

ModuleResource klass roll on implementeerida antud mikroteenuse REST API. Implementeeriud on *postModule*, *getModule*, *deleteModule* ja *getAllModules* API meetodid. Need on olulised operatsioonid, selle mikroteenuse eesmärkide täitmiseks.

ModuleService (Joonis 8) on liides, mis dikteerib olulised meetodid *ModuleServiceImpl* klassi jaoks. Kuna antud mikroteenus täidab väga lihtsat eesmärki, siis need meetodid vastavad täpselt *ModuleResource* klassi API meetoditele.

```
public interface ModuleService {
    void postModule(Module module);
    Module getModule(String id);
    List<Module> getModules();
    void deleteModule(String id);
}
```

Joonis 8. *ModuleService* liides

PostModule meetod kutsutakse välja siis, kui moodul aktiveeritakse esmakordselt. Moodul võtab automaatselt ühendust *ModuleResource* klassi *postModule* API meetodiga ja salvestab sinna andmed, mis kirjeldab tema riistvara. Mooduli poolt üles saadetud andmed salvestatakse *Datastore* teenusesse, sest seda andmebaasi salvestamise operatsiooni ei kutsuta välja väga tihti. Lisaks lihtsale salvestusoperatsioonile, registreerib see meetod mooduli ka cloud MQTT teenusesse, mis on oluline Comodule riistvara osakonna jaoks.

3.1.3 Module-Type

Module-Type mikroteenus on loodud eesmärgiga, et lahendada probleem, mis tekib iga kliendi andmete erinevusest. Kuna iga kliendi jaoks toodetud moodulite seeria vajab riistvaras oma isiklikku püsivara konfiguratsiooni, siis selle konfiguratsiooni ID järgi saab eristada *Module-Module* mikroteenuses registreeritud mooduleid klientide järgi.

Olulised klassid antud mikroteenuses on järgmised:

- VehicleCommandConfigResource
- VehicleConfigResource
- VehicleStatusConfigResource
- VehicleConfigService
- VehicleConfigImpl

Ressursiklassid *VehicleCommandConfigResource*, *VehicleConfigResource* ja *VehicleStatusConfigResource* eksisteerivad eraldis klassidena ainult sellepärast, et koodi puhtana ja selgemana hoida. Iga ressursiklass on kaardistatud eraldi API rajale.

VehicleConfigService on liides, mis paneb paika meetodid, mida *VehicleConfigImpl* peab implementeerima (Joonis 9).

```
public interface VehicleConfigService {
    void createVehicleConfig(VehicleConfig vehicleConfig);
    void createCommandConfig(long typeId, String commandConfig);
    void createStatusConfig(long typeId, String statusConfig);
    VehicleConfig getVehicleConfig(long typeId);
    Object getVehicleCommandConfig(long typeId);
    Object getVehicleStatusConfig(long typeId);
}
```

Joonis 9. VehicleConfigService liidese poolt nõutud meetodid.

createVehicleConfig, *createCommandConfig* ja *createStatusConfig* meetodid kutsutakse välja ressursiklasside API meetodite poolt kui nendeni autoriseeritud päring jõuab. Need meetodid salvestavad *Datstore* andmebaasi objekti, mille ID on mooduli püsivara versiooni number ja massiiv, mille pikkus ja sisu järjekord peab olema sama kui CSV stringi pikkus ja järjekord, mida iga spetsiifilise püsivara versiooniga moodul välja saadab. Selle abil saab viia kokku iga mooduli oma kliendi jaoks loodud konfiguratsiooniga ja *Module-Data* mikroteenuses tõlgendada mooduli poolt välja saadetud CSV stringi.

Sõiduki põhilised andmed, mille konfiguratsioon luuakse *createVehicleConfig* meetodiga vajab põhjalikumat konfiguratsiooni võrreldes staatuste ja käskude konfiguratsiooniga. Staatuste ja käskude konfiguratsiooni puhul koosneb konfiguratsiooni objekti massiiv ainult stringidest, kuid sõiduki põhiliste andmete konfiguratsiooni puhul koosneb massiiv *GenericDataPoint* objektidest.

See objekt sisaldab täpsemat kirjeldust elektrijalgrattasse installeeritud mooduli poolt välja saadetud andmetest. Täpsemalt on ära defineeritud *String* väärtusena andmepunkti nimi ja ühik ning *Double* väärtusena teisendusväärtus. Tänu täpsemale kirjeldusele saab andmepunktid automaatselt *Module-Data* mikroteenuses põhjalikult ära defineerida vastavalt erinevate klientide soovile.

Meetodid *getVehicleConfig*, *getVehicleCommandConfig* ja *getVehicleStatusConfig* kutsutakse samuti välja ressursiklasside API meetodite poolt ja need lihtsalt viivad läbi andmebaasist andmete väljavõtmise operatsiooni ja tagastavad vastuse.

3.1.4 Presentation

Presentation projekti ülesanne on suunata andmete haldamise serverirakendusesse tehtud päringud õigesse mikroteenusesse. Selle eesmärgi täitmiseks sisaldab antud mikroteenus *web.xml* faili, kus on eraldi konfigureeritud iga mikroteenuse servletid.

Iga servleti konfiguratsioon kaardistab url-mustri vastavalt servletile. Seejärel suunab servleti konteiner kliendi poolt tehtud päringu vastavasse mikroteenuse projekti. Näitena on toodud *Module-Data* mikroteenuse servleti konfiguratsioon (Joonis 10).

```
<servlet>
       <servlet-name>ModuleDataApplication</servlet-name>
       <servlet-class>
             org.glassfish.jersey.servlet.ServletContainer
       </servlet-class>
       <init-param>
            <param-name>javax.ws.rs.Application</param-name>
            <param-value>com.comodule.web.data.ModuleDataApplication
            </param-value>
       </init-param>
</servlet>

<servlet-mapping>
        <servlet-name>ModuleDataApplication</servlet-name>
        <url-pattern>/api/module-data/*</url-pattern>
</servlet-mapping>
```

Joonis 10. Module-Data mikroteenuse servleti konfiguratsioon.

3.2 Analüütika veebirakenduse serverirakenduse lahendus

Analüütika veebirakenduse serverirakenduse eesmärk on API-de kaudu kättesaadavaks teha veebirakendusse sisseloginud kliendile tema jalgrataste andmed. Selleks tuleb veenduda, et klient on sisse loginud ja seejärel teha vajalikud päringud serverirakendusesse, mis tegeleb jalgrataste andmete haldamisega. Antud peatükis on kirjeldatud selle serverirakenduse implementatsioon.

3.2.1 Autentimine

Veebirakenduse serverirakenduse turvalisuse nõude täitmiseks on loodud kasutaja andmete haldamiseks *PlatformUser* objekt.

PlatformUser objekti sees hoitakse sisselogimiseks ja sessiooni hoidmiseks olulisi välju *email*, *password* ja *session* ning andmete andmebaasis grupeerimise jaoks olulised väljad *lastRole* ja *roles*. Sisselogimiseks tuleb teha POST päring *SessionResource* klassi poolt implementeeritud *login* API meetodisse, mis on meetod, mida *filter* klass ei kontrolli. See meetod kutsub välja *SessionService* liidese poolt defineeritud ja *SessionServiceImpl* klassi poolt implementeeritud *login* meetodi, mis kontrollib, kas kasutajanimi ja parool on õige. Seejärel lisab *PlatformUser* objekti külge uue *session* välja väärtuse, salvestab selle *Datastore* andmebaasi ja saadab päringu vastuseks loodud *Session* objekti, mis sisaldab äsja sisse loginud kasutaja andmeid ja *PlatformUser* objekti külge lisatud sessiooni ID väärtust.

Seda kasutab ära *Filter* klass, mis võtab päringu vastu enne, kui see jõuab API implementatsiooni. Filtrist lähevad läbi ainult API meetodid, mis on tähistatud *@Authenticated* annotatsiooniga. *Filter* klassis on implementeeritud *filter* meetod, kasutades selleks ära javasse sisseehitatud liidest *ContainerRequestFilter*. See meetod kontrollib, kas turvaliste päringute kontekstis on olemas vajalik päis, mis sisaldab sessiooni id väärtust ja kas sellise sessiooniga kasutaja on serveris olemas. Õige sessiooniga kasutaja objekti olemasolu korral kontrollib kood ka sessiooni kehtivust (Joonis 11).

```
String sessionId =
requestContext.getHeaders().getFirst(HEADER_KEY_SESSION_ID);
if (sessionId == null)
	throw new NotAuthorizedException("User not authorized");
PlatformUser platformUser =
ofy().load().type(PlatformUser.class).filter("sessionId",
sessionId).first().now();
PlatformUserModel platformUserModel =
new PlatformUserModel(platformUser);

if (platformUserModel.isNull() ||
platformUserModel.hasSessionExpired(System.currentTimeMillis()))
	throw new NotAuthorizedException("User not authorized");
```

Joonis 11. Osa filter meetodist, mis viib läbi päringu autentimist.

Kui kõik kontrollid on edukalt läbitud, defineeritakse *PlatformUser* objekti *lastRole* välja poolt viidatud *Role* objekti *displayName* välja väärtuse abil nimeruum ja suunatakse päring edasi välja kutsutud *@Authenticated* annotatsiooniga tähistatud API meetodi implementatsiooni. Teised API meetodid, mis ei ole tähistatud *@Authenticated* annotatsiooniga ei lähe läbi filtri ja seega need API meetodid ei ole turvalised, sest need API meetodid võtavad vastu kõik päringud. Turvalisuse huvides selle lõputöö raames on oluline ainult andmete haldamise serverirakendusega suhtlemise turvalisus.

3.2.2 Erinevate klientide andmete eraldamine

Erinevate klientide andmete eraldi hoidmiseks ja käsitlemiseks on loodud *Role* objekt. Lisaks klientide andmete eraldamisele täidab see objekt ka eesmärki, mis nõudis, et klientidel oleks võimalik oma andmeid gruppeerida ja nendele ligi pääseda mitme erineva registreeritud konto alt.

Selle eesmärgi täitmiseks sisaldab *PlatformUser* objekt välju *lastRole* ja *roles*. Välja *lastRole* väärtust on võimalik muuta kasutades *PlatformUserResource* klassi API meetodeid. Selle välja väärtus hoiab endas informatsiooni hetkel aktiivse oleva *Role* objekti kohta. *Role* objekt hoiab endas ainult kahte välja: *displayName String* andmetüübina ja *id Long* andmetüübina. *Role* objekti displayName välja eesmärk on talletada kliendi poolt valitud nime väärtust, mis võimaldab kliendile kuvada millised rollid on tema kontole lubatud ja millise rolli andmeid ta hetkel vaatab. Tänu sellisele ülesehitusele on võimalik defineerida erienevad nimeruumid *DataStore* andmebaasis, kasutades nimeruumi loomisel *Role* objekti *displayName* välja väärtust.

3.2.3 Andmete haldamise serverirakendusega suhtlemine

Selleks, et iga kleint saaks ainult enda spetsiifiliste moodulite kohta infot, peab iga klient registreerima nende poolt kasutatavad moodulid Comodule analüütika veebiplatvormi kasutades. Eesmärgi täitmiseks on vaja API POST meetodit, mis lubab sisselogitud kliendil registreerida temale kuuluvate moodulite ID väärtused. Need väärtused salvestatakse Datastore andmebaasi *SharingVehicle* objektidena (Joonis 12).

```
@Id
private Long id;
@Index
private String moduleId;
private LocationInfo locationInfo;
private String name;
private Long addedTimestamp;
private JsonObject data;
private JsonObject status;
```

Joonis 12. *SharingVehicle* objekti väljad

SharingVehicle objekt esindab elektrijalgratast. Sellel objektil on omaenda *id* väli ning sisaldab ka sisseloginud kasutaja poolt selle alla registreeritud mooduli ID välja *moduleId*. Registreerides mooduli ID salvestatakse registreerimise dokumenteerimiseks objekti külge ka *addedTimestamp* välja väärtus. Kasutaja saab soovi korral lisaks veel veebiplatvormi abil defineerida elektrijalgratta nime väärtuse *name* välja alla, et jalgrattaid rohkem inimloetaval kujul kuvada saaks.

Andmete dünaamilisuse eesmärgi täitmiseks on *SharingeVehicle* objekti küljes ka *JsonObject* väljad, mis lubavad hoida enda väärtusena mistahes objekti. Selle lõputöö raames on olulised objektid lõputöö eesmärgi täitmiseks *data* ja *status*. Kasutades ära sellist objekti ülesehitust on võimalik teistest rakendustest küsida ja *SharingVehicle* objekti külge lisada ükskõik millist objekti. Kui tulevikus on soovi *SharingVehicle* objekti küljes olevaid väärtuseid täiendada, siis tuleb lihtsalt objekti juurde lisada uus *JsonObject* väli ja selle väärtuseks panna suvalise Comodule mikroteenuse poolt JSON formaadis tagastatud informatsiooni.

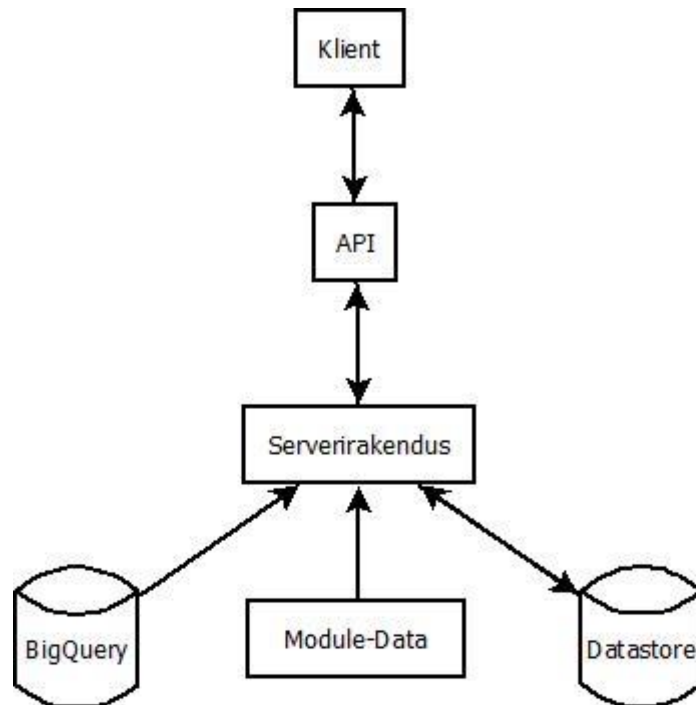
Registreeritud moodulid salvestatakse *DataStore* andmebaasis ainult hetkel aktiivse oleva rolli poolt definieeritud nimeruumi alla. Tänu sellele registreeritakse kliendi antud rolli poolt defineeritud nimeruumi alla ainult tema enda poolt registreeritud moodulid.

Suhtlemine andmete haldamise serverirakendusega toimub läbi API meetodite, mis on implementeeritud *ModuleResource* klassis. See klass kasutab omakorda meetodeid, mis on defineeritud *ModuleService* liidese poolt, neid meetodeid implementeerib *ModuleServiceImpl* klass. Olulised API meetodid on *getVehicleData* ja *getVehicleDetailData*. Ligipääs API meetoditele on lubatud ainult autenditud klientidele, kelle nimeruumi alla on registreeritud üks või rohkem mooduli ID-d.

Meetod *getVehicleData* eesmärk on küsida mitme mooduli hetke seisukorra andmed. Meetod kasutab *moduleService* liidese samanimelist meetodit, mis paneb kokku GET päringu ning saadab selle andmete haldamise serverirakenduse *Module-Data* mikroteenusesse. Kasutades registreeritud moodulite ID väärtuseid tehakse päring andmete haldamise serverirakenduse *Module-Data* mikroteenusesse ja seejärael võtab andmete haldamise serverirakendus *DataStore*-st ainult nende moodulite andmed, mille ID-d on defineeritud päringus. Päringu vastuses on küsitud moodulite kõik andmed JSON formaadis. Tagastatud andmete põhjal loob *getVehicleData* meetod täieliku SharingVehicle objekti, mis sisaldab *data* ja *status* väljade väärtuseid.

Meetod *getVehicleDetailData* implementatsioon on lihtsam. Selle meetodi eesmärk on küsida *BigQuery* andmebaasist ühe mooduli kõikide andmete ajalugu. Vajalik on ühe mooduli ID, et ehitada üles SQL stiilis päringu *String*. String luuakse klassis *DetailDataQueryBuilder* ja *BigQuery* vastus parsitakse kasutades *DetailDataParser* klassi.

Andmete liikumine on kujutatud joonisel (joonis 13). Jooniselt on näha, et serverirakendus korjab andmed kokku erinevatest allikatest. Lisaks *Module-Data* mikroteenusele, võib andmeid pärida ka teistest Comodule infosüsteemis olevatest mikroteenustest ja sellega on täidetud modulaarsuse nõue, kuid selle lõputöö raames teiste mikroteenuste kasutamine pole vajalik.



Joonis 13. Mooduli andmete liikumine läbi analüütika veebirakenduse serverirakenduse.

4 Hinnang loodud lahendusele

4.1 Implementatsioonis esinenud keerukused

Antud lõputöö implementeerimisel osutus kõige keerulisemaks osaks erinevate klientide andmete vastuvõtmine ühe dünaamilise meetodi abil. Kõikide moodulite andmete käsitlemine sellisel viisil oli keeruline realiseerida, sest puudub kindel andmestruktuur. Ainuke informatsioon, mis serverisse jõuab on CSV string ja selle põhjal peab server looma JSON formaadis informatsioonistruktuuri, mis siis andmebaasi salvestada ja teistele Comodule informatsioonisüsteemi osadele edasi saata. Veel tuleb arvesse võtta, et üks moodul võib tagantjärgi informatsiooni saata, mis tähendab, et ühe ühe CSV stringi asemel saabub mitu. Probleemi lahendamiseks tuli loobuda kindla objekti andmestruktuurist ja luua eraldi mikroteenus kliendi andmestruktuuri konfiguratsiooni jaoks, mis lisas tööle mahtu juurde.

Teine keerukus, mis implementeerimisel esile tuli oli GAE teenuste kasutamise hind. Iga projekt, mis GAE keskkonda üles on laetud maksab ja seega raha kokkuhoidmiseks tuli Mikroteenused panna ühe projekti alla.

Mikroteenuste tõttu tuli kood kirjutada nii, nagu oleks tegemist eraldi projektidega. Sellepärast tuli teatud funktsionaalsed osad kategoriseerida. Kategorisatsiooni valimine oli üsnagi keerukas ning lõputöö käigus valitud mikroteenuste arhitektuur ei ole kindlasti lõplik.

4.2 Tehtud töö tulemus

Projektid laeti üles GAE keskkonda. Tehtud töö on võimeline vastu võtma ja hoidma suures koguses jalgrataste informatsiooni. Hetkel sisaldab lõputöö jaoks loodud *BigQuery* tabel 3,806,140 rida ja päringu tegemine sellest tabelist võtab keskmiselt aega 1.6 sekundit. Kõige aktiivsem moodul on viimase 24 tunni jooksul andmeid üles saatnud 4720 korda.

Moodulite poolt välja saadetud informatsioon on kättesaadav Comodule analüütika veebiplatvormis ja edastatakse teiste informatsioonisüsteemi osadele. Rakendus on võimeline käsitlema andmeid suurtes kogudes ja kõik teised töö alguses püstitatud nõuded said samuti täidetud.

4.3 Edasise arengu võimalused

Edasise arengu puhul tuleks esimese asjana kirjutatud koodi struktuur üle vaadata ning dünaamilisuse ja modulaarsuse jaoks struktuur veelgi enam mikroteenuste arhitektuuri peale üle viia. Peale selle lõputöö implementeerimist on selge, et mikroteenuste loomisel tuleks silmas pidada ühte objekti. Iga andmebaasis oleva objekti jaoks peaks olema enda mikroteenus.

Analüütika veebirakenduse serverirakendust on võimalik sama moodi mikroteenusteks jagada. Esimese asjana tuleks eraldada autentimise kood ja viia see iseenda mikroteenuse alla, mis haldab *PlatformUser* objekti. Samamoodi tuleks eraldada *SharingVehicle* objekt ja luua selle jaoks enda mikroteenus.

GAE pakub veel mitmeid teenuseid, mida saaks antud lõputöös ära kasutada. Näiteks *Google Cloud Dataflow,* mis on mõeldud reaalajas liikuvate andmete haldamiseks. Edasises arengus on oluline ära kasutada veel rohkem GCP võimalusi.

5 Kokkuvõte

Lõputöö eesmärgiks oli luua osa infosüsteemi osa, mis võtaks vastu elektrijalgratta mooduli andmeid ja oleks võimeline teenindama analüütika veebirakendust, mis kuvab vastuvõetud informatsiooni. Kuna oluline on rataste jälgimine reaalajas, siis juba üks moodul võib andmeid serverisse saata mitu korda sekundis. Keerukust lisasid ka asjaolud, et iga erineva kliendi jalgrattad saadavad üles erinevat informatsiooni ning interneti puudumisel moodulisse kogunenud andmed saadetakse tagantjärgi korraga üles. Lisaks sellele oli oluliseks eesmärgiks loodud süsteemi turvalisus ja rakenduse modulaarsus.

Antud töö käigus tutvus autor GCP võimalustega ja kasutas neid võimalusi lõputöö eesmärkide täitmiseks. Ülejäänud rakendustele andmete saadavaks tegemiseks on kasutatud *Pub/Sub* teenust, suurtes kogustes analüütika andmete salvestamiseks on kasutatud *BigQuery* teenust ja hetkeandmete salvestamiseks *Datastore* teenust.

Probleemide lahenduseks sai loodud kaks veebirakendust. Serverirakendus, mis teenindab analüütika veebirakendust kasutades ära teist lõputöö käigus loodud serverirakendust. Teine lõputöö käigus tehtud serverirakendus on oluline, sest selle töö peab tagama, et kõikide moodulite poolt saadetud andmed saaksid vastu võetud ja õigesse andmebaasi salvestatud.

Tehtud rakendused teenindavad juba reaalseid kliente ja kõik töötab plaanipäraselt. Teenindavate moodulite arvu suurenedes tõuseb küll hind kuid rakenduse töö ei ole häiritud.

Edaspidine areng on lihtsustatud tänu mikroteenuste arhitektuurile. Seda tulevikus täiendades on võimalik arhitektuuri veelgi lihtsamaks muuta. Samuti on võimalik lahendust täiendada GCP teenustega mida hetkel ei ole lõputöös kasutusele võetud. Seda tehes peaks olema võimalik ka optimeerimisega makstud hinda alla viia.

Kasutatud kirjandus

[1] Comodule.
[WWW] http://www.comodule.com/ (14.05.2018)

[2] Google Cloud Platform
[WWW] https://cloud.google.com (14.05.2018)

[3] Fowler M., Lewis J.
Microservices
[WWW] https://martinfowler.com/articles/microservices.html (10.03.2014)

[4] Google BigQuery
[WWW] https://cloud.google.com/bigquery/ (14.05.2018)

[5] Aboukhalil R.
A Tale of Two Clouds: Amazon vs. Google
[WWW] https://medium.com/@robaboukhalil/a-tale-of-two-clouds-amazon-vs-google-4f2520516a38 (15.05.2018)

[6] Fedak V.
AWS vs. Google Cloud Platform: which cloud service provider to choose
[WWW] https://hackernoon.com/aws-vs-google-cloud-platform-which-cloud-service-provider-to-choose-94a65e4ef0c5 (11.10.2017)

[7] Google App Engine
[WWW] https://cloud.google.com/appengine/ (14.05.2018)

[8] Lowry J.
Designing for Scale
[WWW] https://cloud.google.com/appengine/articles/scalability (02.2017)

[9] Pub/Sub
[WWW] https://cloud.google.com/pubsub/architecture (11.05.2018)

[10] Google Cloud Datastore Overview
[WWW] https://cloud.google.com/datastore/docs/concepts/overview (14.05.2018)

[11] Sato K.
An Inside Look at Google BigQuery
[WWW] https://cloud.google.com/files/BigQueryTechnicalWP.pdf (2012)

[12] Gosling J., Joy B., Steele G., Bracha G., Buckley A., Smith D.
The Java Language Specification
[WWW] https://docs.oracle.com/javase/specs/jls/se9/jls9.pdf (07.08.2017)

[13] Bosa R.
Gradle
[WWW] https://www.journaldev.com/7971/gradle (02.04.2018)

[14] Jersey
[WWW] https://jersey.github.io/ (14.05.2018)