TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond

Peeter Raudsepp 179725IABB, Sten-Mark Paju 179339IABB, Kaarel Rohumaa 179151IABB, Tatjana Putškova 179931IABB, Mark Porohnja 179237IABB

ELEKTRIAUTODE MÕJU SIMULEERIMINE ENERGIATURULE

Bakalaureusetöö

Juhendajad: Kristina Murtazin
MSc.
Jekaterina Tšukrejeva
MSc.

Tallinn 2021

Autorideklaratsioon

Kinnitame, et oleme 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.

Autorid: Peeter Raudsepp, Sten-Mark Paju, Kaarel Rohumaa, Tatjana Putškova, Mark Porohnja

18.05.2020

Annotatsioon

Lõputöö eesmärgiks on luua simulatsioon lähituleviku Eestist, kus leidub tuhandeid V2G laadimistehnoloogiat kasutavaid elektriautode digitaalseid kaksikuid.

Süsteem on lahendatud kahe rakendusena, millest esimene on REST API projekt ning teine veebirakendus, mis koosneb kaardist, kaardil olevatest dünaamilistest objektidest ning muudest kaarti ja kaardil olevaid objekte mõjutavatest komponentidest. Kaardirakenduse jaoks genereeritakse info REST API projektis.

Rakenduse arendamiseks kasutati põhiliselt Spring ja ReactJS raamistikke ning IntelliJ IDEA arenduskeskkonda.

Töö annab ülevaate seni projekti käigus valminud arendusetappidest ning nende jooksul arendatud kasutajalugudest ja tehnilistest ülesannetest.

Lõputöö on kirjutatud eesti keeles ning sisaldab teksti 84 leheküljel, viite peatükki, 60 kuvatõmmist kirjutatud koodist, nelja skeemi ja ühte suhtlusdiagrammi.

Abstract

Simulating the impact of electrical vehicles on energy market

The goal of this thesis is to create a simulation of a near future Estonia, where there are thousands of electrical vehicle digital twins using the new V2G charging technology. The project was developed in two parts of which the first one is a REST API application and the second a single page web application. The main use of the REST API application is to generate information about the digital twin based electrical vehicles. Information and actions for the virtual vehicle are visualized in the web application, which consist of a map, information layers of the map and buttons to trigger specific actions for the car, like making the car drive and stop, regenerating new addresses to the virtual vehicles or reset the location of the car. The many different information layers are shown on the map as markers for entities like cars, home and work locations, routes and public chargers. The layers can be toggled on and off the map. Clicking on the markers show additional information about the entities, for example distance driven for the car or the length of the route. There is also an information layer to show Enefit Volt's active network of electric vehicle public chargers on the map.

The project will be used by Eesti Energia to simulate the impact of thousands of electrical vehicles, that all use V2G smart charging system, on the energy market and grid of Estonia.

The main development framework used in REST API application is Spring and ReactJS for the web page application. Both applications were developed using IntelliJ IDEA integrated development environment.

The thesis provides an overview of the work finished in the development stages. The work is divided into milestones. These milestones consist of different user-stories and technical stories which are described more in length in the project results and analysis.

The thesis is written in Estonian and contains 84 pages of text, 5 chapters, 60 screenshots, 4 schemas and 1 communication diagram.

Lühendite ja mõistete sõnastik

API	*Application Programming Interface*, definitsioonide ja protokollide kogum, mis lubab tehnoloogia produktidel ja teenustel suhelda üle interneti.
CI/CD	*Continuous integration/Continuous delivery,* pidevintegratsioon ja - valmidus.
CORS	*Cross-Origin Resource Sharing*, mehhanism, mis võimaldab ressursse jagada teisele välisele domeenile.
CSRF	*Cross-site request forgery,* tuntud ka kui *one-click* rünne või sessioon ärandamine on rünne, mille käigus kasutatakse ära veebilehe usaldust kasutaja vastu ning petetakse laadima veebilehte, mis annab võimaluse ründajal tegutseda kasutaja nimel ja õigustega ning muutusi vastavas süsteemis.
Docker konteinerid	*Docker Containers*, lubavad arendajal kokku pakkida aplikatsiooni koos kõigi osadega, mida ta vajab, näiteks teegid ja muud sõltuvused, ning jooksutada seda ühe paketina.
EDA	*Event-Driven Microservice Architecture*, sündmuspõhine mikroteenuse arhitektuur, mida kasutavad teenused suhtlevad üksteisega läbi sündmuste teadete.
HTTP	*Hypertext Transfer Protocol,* Kommunikatsiooniprotokoll veebiserverite omavaheliseks ühendamiseks ja andmete jagamiseks läbi interneti.
JDK	*Java Development Kit*, üks kolmest tuumtehnoloogia paketist, mida kasutatakse Java programmeerimiseks, aitab arendajatel luua Java programme, mida saab jooksutada.
JSON	*JavaScript Object Notation*, lihtsustatud andmevahetusvorming, mis põhineb JavaScripti programmeerimiskeele alamhulgal.
Kasutajalugu	*User Story*, lühikene ja lihtne funktsiooni kirjeldus kasutaja või kliendi vaatenurgast.
Mähisklass	*Wrapper Class*, pakub mehhanismi, mille abil teisendatakse primitiivne väärtus objektiks ja vastupidi.
OOP	*Object-oriented programming,* programmeerimise tüüp, kus defineeritakse andmestruktuuride tüübid, sealhulgas ka operatsioonid, mida andmestruktuurile omistada.

Püstijala koosolek	*Stand-up Meeting*, lühike tiimi koosolek, mida peetakse püsti seistes. Selle eesmärk on läbi rääkida tähtsad ülesanded, mis on valmis, tegemisel või alustamise järgus.
REST	*Representational State Transfer*, arhitektuuri stiil, mis loob standardeid arvutisüsteemide vahel veebis. Lihtsustab omavahelist suhtlemist.
Scrum meister	*Scrum Master,* agiilse arendustiimi juhendaja.
SPA	*Single Page Application*, ühe leheline rakendus, mis ei pea selle kasutamise ajal lehte uuesti laadima.
Sõltuvuste sisestamine	*Dependency Injection,* fundamentaalne osa Spring raamistikust, läbi mille Springi konteinerid sisestavad objekte teistesse objektidesse või sõltuvustesse.
Toote omanik	*Product Owner*, projekti peamine osanik, osa toote omaniku vastutusest on omada visiooni, mida ta soovib ehitada ja edastada see visioon arendusmeeskonnale.
URL	*Uniform Resource Locator,* ressursi aadress internetis ja protokoll sellele ligipääsemiseks.
V2G	*Vehicle-to-Grid*, elektriauto laadija tehnoloogia, mis võimaldab elekriauto akust saata elektrit tagasi elektrivõrku.
Verstapost	*Milestone*, kasutatakse töö edasiliikumise jälgimiseks mõne kindla eesmärgi või sündmuse suunas.

Sisukord

Autorideklaratsioon.. 2
Annotatsioon.. 3
Abstract Simulating the impact of electrical vehicles on energy market.. 4
Lühendite ja mõistete sõnastik.. 5
Sisukord.. 7
1 Sissejuhatus.. 10
1.1 Projekti lühikirjeldus.. 10
1.2 Probleem ja projekti eesmärk.. 10
1.3 Funktsionaalsus.. 11
1.3.1 Funktsionaalsed nõuded.. 12
1.3.2 Mittefunktsionaalsed nõuded.. 13
1.4 Töö edasine struktuur.. 13
2 Metoodika.. 15
2.1 Objekti detailne kirjeldus.. 15
2.2 Tööriistade kirjeldus.. 20
2.3 Tööprotsessi kirjeldus.. 20
3 Töö tulemused.. 23
3.1 Esimene etapp.. 23
3.1.1 Ilmateave kättesaamine.. 23
3.1.2 Laadija parameetrid.. 25
3.1.3 Sõidukid liiguvad kasutades reaalseid marsruute.. 26
3.2 Teine etapp.. 30
3.2.1 Kaardipõhine simulaator.. 30
3.2.2 Aku laadimine ja tühjenemine.. 34
3.2.3 Laadimisrežiimide vahel lülitamine.. 40
3.3 Kolmas etapp.. 43
3.3.1 Sõiduki liikumine kasutajaliideses.. 43
3.3.2 Sõiduki liikumine serveripoolses osas.. 45

3.3.3 Kodu ja töö asukohtade vaheline kaugus 48
3.3.4 Sõiduki aku laadimiskõver 48
3.4 Neljas etapp 52
3.4.1 Sõiduki aku tühjenemiskõver 52
3.4.2 Avalikud laadimispunktid kaardil 57
3.4.3 Dockeri implementatsioon projektis 60
3.4.4 Pidevintegratsiooni ja -valmiduse (CI/CD) konveieri implementatsioon 62
3.5 Testid 63
3.5.1 Ühiktestid 64
3.5.2 Integratsioonitestid 65
4 Analüüs ja järeldused 67
4.1 Tulemuste tehnilise teostuse põhjendus 67
4.1.1 Ilmateave kättesaamine 67
4.1.2 Laadija parameetrid 68
4.1.3 Sõidukid liiguvad kasutades reaalseid marsruute 68
4.1.4 Kaardipõhine simulaator 69
4.1.5 Aku laadimine ja tühjenemine 71
4.1.6 Laadimisrežiimide vahel lülitamine 71
4.1.7 Sõiduki liikumine kasutajaliideses 72
4.1.8 Sõiduki liikumine serveripoolses osas 73
4.1.9 Kodu- ja tööasukohtade vaheline kaugus 74
4.1.10 Sõiduki aku laadimiskõver 74
4.1.11 Sõiduki aku tühjenemiskõver 75
4.1.12 Avalikud laadimispunktid kaardil 76
4.1.13 Dockeri implementatsioon projektis 77
4.1.14 Pidevintegratsiooni ja -valmiduse konveieri implementatsioon 79
4.2 Testid 80
4.2.1 Automaattestimine 80
4.2.2 Testjuhud 80
4.3 Kirjanduse ülevaade 84
4.3.1 Kirjanduse põhjal ülevaade ja analüüs 84
4.4 Teostatud tööde kirjeldus 86
4.4.1 Giti commit'ide väljavõte 86
4.4.2 Clockify rakendusest väljavõte 86

4.5 Hinnang projekti teostamise protsessi kohta 91
4.5.1 Projekti juhtimine ning projekti teostamise protsess 91
4.5.2 Projekti kitsaskohad ja kordaminekud 92
4.5.3 Hinnang üldisele projekti teostamise protsessile 93
4.5.4 Meeskondlik hinnang meeskonnaliikmete panuse kohta 93
5 Kokkuvõte 94
Kasutatud allikad 96
Lisa 1. Peeter Raudsepa enda panuse kirjeldus ja eneseanalüüs 100
Lisa 2. Kaarel Rohumaa enda panuse kirjeldus ja eneseanalüüs 103
Lisa 3. Tatjana Putškova enda panuse kirjeldus ja eneseanalüüs 105
Lisa 4. Sten-Mark Paju enda panuse kirjeldus ja eneseanalüüs 110

1 Sissejuhatus

1.1 Projekti lühikirjeldus

Antud töö on bakalaureuse lõpuprojekti aruanne. Projekt on järg varem alustatud projektile, mida hakati arendama õppeaine ITB1706 Infosüsteemide arendamise meeskonnaprojekt raames. Meeskond koosnes viiest Tallinna Tehnikaülikooli äriinfotehnoloogia tudengist ja projekti valmistati koostöös Eesti Energia tütarettevõtte Enefitiga.

1.2 Probleem ja projekti eesmärk

Eesti Energia arendab välja virtuaalset elektrijaama, mille idee on tekkinud tänu elektrienergia tarbijate arvu suurenemisele, kes suudavad peale tarbimise ka ise elektrit toota, näiteks päikesepaneelide kaudu. Virtuaalne elektrijaam on tuleviku kontseptsioon, mille olemus seisneb erinevatest elektriallikatest ja -salvestusseadmetest koosnevast detsentraliseeritud elektrijagamisvõrgustikust, mis pakub tarbijatele võimalust oma toodetud elektrienergiat müüa [1]. Tänu virtuaalse elektrijaama implementeerimisele on võimalik energiatootmisettevõtetel keskenduda elektrivõrgustiku varustuskindluse tagamisele ja energiajagamisplatvormi arendamisele. See vähendab elektrienergia tootmiskulusid ning sellega kaasnevaid keskkonda reostavaid mõjusid.

Käesolev projekt keskendub ühele osale virtuaalse elektrijaama võrgustikust, täpsemalt elektriautode akude energia talletamise potentsiaalile ning nende mõjudele energiaturul ja taristul. Seda on võimalik testida alles siis, kui elektriautode arv Eestis kasvab kümnete tuhandeteni.

Elektrilevi ja Eesti Energia on loonud Eestisse nutika elektriautode kiirlaadimisplatvormi Enefit Volt, mis katab kogu Eestit. Enefit Volt soovib uuendada olemasolevat kiirlaadimisplatvormi uuema generatsiooni V2G nutikate elektrisõidukite laadijatega [2], mis võimaldab lisaks elektrisõiduki laadimisele ka elektrit võrgustikku tagasi müüa. Tänu

sellist tüüpi laadijale saab ühendada elektriautod virtuaalse elektrijaama tehnoloogiaga. Ettevõte eeldab, et see lahendus võib vähendada elektrienergia tootmise kulusid ning samuti põhjustada elektri päevase tipphinna langust. Lisaks on võimalik näha olemasoleva taristu töökindlust ning energiaturu reageerimist, kui Eesti liiklusesse peaks lisanduma kümneid tuhandeid elektrisõidukeid. Seda just seetõttu, et kui on rohkem elektrisõidukeid, siis on ka rohkem elektrit nende akudesse salvestatud ning läbi loodud laadimisplatvormi hoitakse rohkem energiat liikvel. Väljatöötamisel oleva tehnoloogia kaudu väheneb oletuslikult ka energia juurde tootmise vajadus päeva elektri nõudluse tipphetkel.

Probleemiks on, et hetkeseisuga puudub piisav elektrisõidukite ressurss eelnevalt välja toodud lahenduse testimiseks. Siinkohal ilmnebki antud projekti tähtsus. Tiimi töö eesmärk Eesti Energias on luua simulatsioon lähituleviku Eestist, kus leidub tuhandeid elektriautode digitaalseid kaksikuid, mis kõik kasutavad uut V2G laadimistehnoloogiat. Digitaalne kaksik on virtuaalne mudel protsessist, tootest või teenusest [3]. Tehtud projekti kasutab Eesti Energia ühe osana suurest pildist, mis ühildatakse Enefiti poolt loodud nutikaid algoritme kasutava laadimisplatvormiga, mille abil saavad tarbijad elektrit kallima hinna puhul müüa ja odavama hinna puhul osta.

1.3 Funktsionaalsus

Antud veebirakendus on süsteemi simulatsioon, kus luuakse virtuaalsed elektrisõidukid ja neid kasutavad kasutajaprofiilid ning simuleeritakse nende käitumist reaalsetes tingimustes ja erinevate stsenaariumite järgi. Autode kodu- ja tööaadressid ning nende vahel sõidetavad teekonnad pärinevad kolmest tiimi poolt valitud linnast - Tallinnast, Tartust ja Pärnust. Sõidukeid saab laadida, sealjuures sõites ka nende akud tühjenevad. Kasutajaliides on üles ehitatud üheleherakendusena, kus tähtsaim komponent on kaart, millel kuvatakse simuleeritud autosid, nende kodu- ja tööaadresse, sõiduki teekonda ja laadimispunkte. Kaardi kõrvale on lisatud navigatsiooninupud sõidu alustamiseks ja peatamiseks, laadimisrežiimide vahel lülitamiseks ning andmete lähtestamiseks ja uuesti genereerimiseks.

1.3.1 Funktsionaalsed nõuded

1. Kaardi kihid: interaktiivsel kaardil on informatsiooni kihid, mida saab sisse ja välja lülitada (autod, kohad, laadijad, marsruudid);
2. Avalikud laadijad: kaardil kuvatakse avalikud laadijad markeritena ja neid saab sisse ning välja lülitada;
3. Kodu ja töö asukohad: kaardil kuvatakse ikoonidena kodu ja töö asukoha objektid, mille peal vajutades näeb asukoha koordinaate ning aadressi;
4. Marsruut: kaardil on joonistatud marsruut, mis ühendab omavahel auto kodu ja töö asukohad. Marsruudile arvutatakse elektritarbimine, marsruudi pikkus ja läbimiseks kuluv aeg;
5. Autod: autodel on kindel spetsifikatsioon, olek ja neid kuvatakse kaardil liikuva ikoonina, mille peale vajutades näeb auto informatsiooni;
6. Auto sõidab: auto sõidab mööda marsruuti ja kaardi kõrval saab vajutada nuppu sõidu alustamiseks ning peatamiseks. Auto ikoonile vajutamisel kuvatakse lähtepunktist läbitud teekonna pikkus;
7. Laadimine ja aku tühjenemine: elektrisõidukit saab laadida, samuti sõites ka tema aku tühjeneb;
8. Laadimise režiimid: on olemas kaks erinevat laadimisrežiimi (reaalne ja lineaarne). Režiime vahetatakse tumblernupul vajutades;
9. Reaalne laadimiskõver: sõiduki aku laadimine toimub reaalse laadimiskõvera alusel;
10. Reaalne tühjenemiskõver: sõiduki aku tühjenemine toimub reaalse tühjenemiskõvera alusel;
11. Ilmateave: rakendus kasutab reaalajas saadud ilmateavet, et simuleerida tegelikke sõidu- ja laadimistingimusi;
12. Lähtestamine ja uuesti genereerimine: elektriautot on võimalik lähtestada väärtustele, mis olid rakenduse käivitamisel või genereerida uued väärtused;
13. Reaalsete kasutajate profiilid: luuakse erinevad kasutaja profiilid, mis võimaldavad määrata kodust tööle ja tagasi sõitmise aegu, aku mahtusid ning kodu ja töö asukohtade vahelisi distantse. Profiili teave ja parameetrid kuvatakse eraldi komponendina kaardi kõrvale auto ikoonil vajutades;
14. Laadijate parameetrid: virtuaalsel laadijal on võimsuse spetsifikatsioon

1.3.2 Mittefunktsionaalsed nõuded

1. Olekumasina implementeerimine rakendusse [4];
2. Serveripoolne projekt on implementeeritud REST API-na [5];
3. Rakendus töötab Docker konteineris [6] ja rakendusi saab koos käivitada kasutades Docker Compose'i [7]
4. Rakendus kasutab CI/CD [8] ja GitHub Actions *pipeline*'i [9];
5. Kasutatakse OOP [10] ning sõltuvuste sisestamist [11];
6. Rakendus on osa sündmuspõhisest mikroteenuse arhitektuurist (EDA [12]);
7. Projektile on lisatud SonarQube [13] koodi kvaliteedi kontrollimiseks;
8. Rakendus integreeritakse ülejäänud mikroteenuste süsteemi REST API kaudu;
9. Rakendus konfigureeritakse ettemääratud parameetrite loetelu järgi (*resource* failid [14]);
10. Kliendipoolne osa on loodud üheleherakenduse arhitektuuri järgi ning suhtleb serveripoolse osaga läbi REST API;
11. Tehnoloogilised piirangud: Java [15] + SpringBoot [16], React JS [17];
12. Ühiktestidega kaetus on vähemalt 50%;

1.4 Töö edasine struktuur

Käesolev aruanne koosneb kasutatud metoodikast, saadud tulemustest, tehtud töö analüüsist ning lisadest.

Metoodika osas on välja toodud kasutatud tööriistad, tööprotsessid ja objekti detailne kirjeldus. Objekti detailse kirjelduse osas leidub projekti ülevaade ja kirjeldatakse projekti käigus planeeritud ja valminud kasutajalugusid [18]. Samuti kirjeldatakse ka, mis sai lõpuprojektile eelnenud meeskonnaprojekti käigus tehtud.

Tulemuste osas on välja toodud koodi seletused näidistega, loogika algoritmide skeemid ning disaini ja arhitektuuri kirjeldused. Kasutajalugusi kirjeldatakse arendusetappide järjekorras, sel viisil saab tööprotsessi etapiviisiliselt jälgida.

Analüüsi osas kirjeldatakse lähemalt tööprotsessi tulemusi, kordaminekuid ja kitsaskohti. Samuti on välja toodud tehtud testid ja koodi testidega kaetust. Lisaks on esitatud

meeskondlikud hinnangud, Giti *commit*'ide [19] väljavõtted ja logid iga meeskonnaliikme kohta.

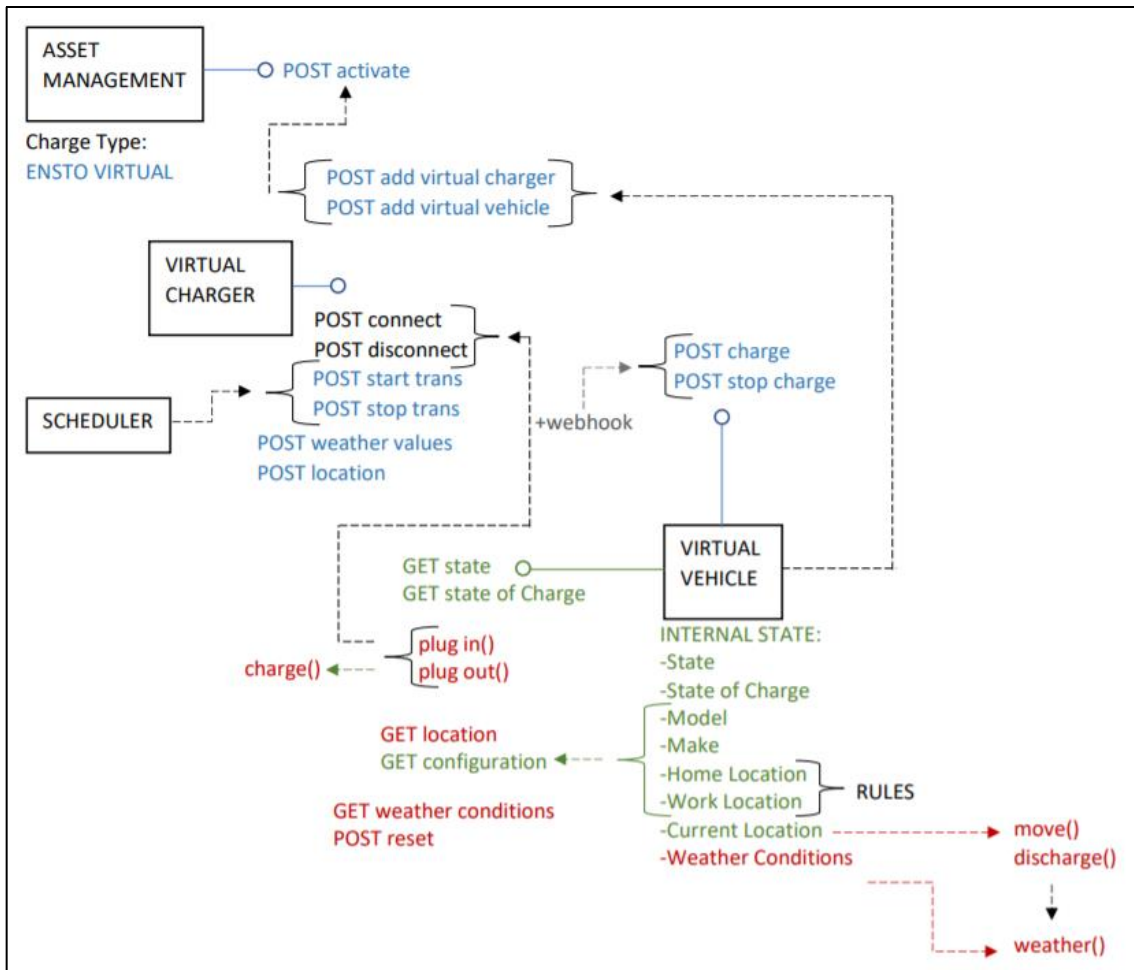
Lisade osas on iga meeskonnaliikme aruanded, mis koosnevad omapoolse panuse kirjeldusest ja eneseanalüüsist.

2 Metoodika

2.1 Objekti detailne kirjeldus

Antud projekt koosneb omavahel integreeritud REST API-st ning üheleherakendusest, mille lõppeesmärgiks on simuleerida realistlikku olukorda, kus Eestis leidub tuhandeid elektriautosid ning milline oleks nende mõju olemasolevale elektrivõrgustikule ja seeläbi ka energiaturule, kui nad kasutavad nutikaid V2G laadijaid. Lõputöö kirjutamise hetkel on valmis arendatud mõlemad rakenduse pooled, kuid funktsionaalsusest jääb veel puudu. Siiamaani on arendatud kaart, mis kuvab autode hetke asukohta, kodu ja töö asukohti ning nende vahelisi marsruute. Lisaks autodele on kaardil näha elektriautode laadimispunkte. Auto saab mööda genereeritud marsruuti edasi-tagasi sõita. Kogu info kaardirakendusele genereeritakse REST API projektis. Selle projekti raames keskenduti peamiselt virtuaalse sõiduki käitumise arendamisele. Tulevikus kavatsetakse suurendada sõidukite arvu ja luua ühendus Enefiti poolt valminud virtuaalse laadija töötamise simulatsiooniga.

Skeemis 1.3.1 on näha virtuaalauto ja -laadija projektide arhitektuuri skeem. Virtuaalse laadija rakenduse API [20] päringud on sinist värvi, virtuaalse elektriauto rakenduse API päringud ning peamised meetodid on märgitud rohelise ja punase värviga.



Skeem 1.3.1. Virtuaalse sõiduki ja virtuaalse laadija rakenduste arhitektuuri skeem

Meeskonnaprojekti käigus valminud loogika on märgitud rohelise värviga ning punase värviga on märgitud osad, mis on valminud praeguses projektis.

Eelmise projekti jooksul konfigureeriti rakendus ning loodi serveri- ja kliendipoolse osa projektid. Valmisid esimesed API lõpp-punktid (*/configuration*, */state*, */stateOfCharge*) ja teenused, kus arvutatakse ja genereeritakse API-dele andmeid. Kasutajaliidesena valmis kaart, kus nägi kodu ja töö asukohti.

Järgnevalt on välja toodud eelneva projekti jooksul valminud kasutajalood:

- Virtuaalse elektriauto kontseptsioon ja konfiguratsioon. Iga loodud virtuaalne auto omab järgnevaid omadusi: mark, mudel, aku maht, laadimisvõimsus, laadimise efektiivsus. Täidab osa funktsionaalsest nõudest 5;
- Kodu ja töö asukohad. Loodi virtuaalsele elektrisõidukile salvestatud kodu- ja tööasukohad, mis koosnevad ID'st, koordinaatidest, aadressist, linnast ja majatüübist. Täidab osa funktsionaalsest nõudest 3;

- Sõiduki olek (*Vehicle Status*). Simuleeritud virtuaalse sõiduki hetkeolekud (laeb (*charging*), laadijaga ühendatud (*plugged-in*), seisab (*idle*), sõidab (*driving*)). Täidab osa funktsionaalsest nõudest 5;
- Aku olek (*State of Charge*). Igale virtuaalsele elektrisõidukile omistati praegune aku olek, mis arvutatakse sõltuvalt sõiduki aku mahust. Täidab osa funktsionaalsest nõudest 5;

Lõpuprojekti raames lisati virtuaalse sõiduki simulatsioonile palju uusi funktsioone ja võimalusi nagu marsruutide ehitamine, auto sõitmine, aku laadimine ja tühjenemine erinevates režiimides. Samuti lisati reaalajas ilmastikuolude saamine, et simuleerida tegelikke sõidu- ja laadimistingimusi. Järgnevalt lisatakse ka kasutajate profiilid, et realiseerida erinevate kasutajate käitumist. Kasutajaliides on kaart kihtidega, mida saab sisse ja välja lülitada, et näha autode, avalike laadijate ning kodu ja töökohtade ikoone ja autode marsruute. Iga ikooni peale vajutades kuvatakse täiendavat infot objekti kohta. Samuti leiduvad kaardi kõrval navigatsiooninupud režiimide vahel lülitamiseks, sõitmise alustamiseks ja lõpetamiseks ning rakenduse käivitamisel genereeritud väärtuste lähtestamiseks või uuesti genereerimiseks.

Järgnevalt on toodud lõpuprojekti raames valminud kasutajalood ja tehnilised ülesanded nelja arendusetapi käigus:

- Ilmateave kättesaamine: ilmastikuolude küsimine OpenWeatherMap API-st [21] ja selle mõju aku laadimis- ja vähenemiskiiruse efektiivsuse arvutamisele lisamine. Täidab funktsionaalsed nõuded 10 ja 11;
- Laadija parameetrid: Virtuaalsel laadijal on võimsuse spetsifikatsioon. Kuulub funktsionaalse nõude 14 alla;
- Sõidukid liiguvad mööda reaalseid marsruute: Autole marsruudi loomine ja praeguse asukoha arvutamine. Täidab funktsionaalset nõuet 4;
- Kaardipõhine simulaator: üheleherakenduse kliendipoolse osa projekt. Põhiline osa on interaktiivne kaart, kus saab näha markeritena kaardil auto, kodu ja töö asukohti ning teekonda kodust tööle. Markereid saab infokihtidele klikkides kaardil kuvada ja ära võtta. Autole, kodu- ja tööasukohtadele ning teekonnale klikkides näeb nende kohta täpsemat infot. Täidab funktsionaalsed nõuded 1, 3, 4 ja 5;
- Aku laadimine ja tühjenemine: auto olekute (laeb (*charging*), laadijaga ühendatud (*plugged-in*), seisab (*idle*), sõidab (*driving*)) vahel lülitatakse olekumasinat

kasutades, autot laetakse kui see asub kodus või tööl, samuti sõites tema aku tühjeneb. Täidab mittefunktsionaalset nõuet 1 ning funktsionaalset nõuet 7;

- Laadimisrežiimide vahel lülitamine: simulatsioonis on kaks erinevat laadimisrežiimi (reaalne ja lineaarne) ja nende vahel lülitatakse tumblernuppu vajutades. Täidab funktsionaalset nõuet 8;
- Sõiduki liikumine kasutajaliideses: Kui auto sõidab ehk tema staatus on *driving*, liigub auto marker kaardil piki määratud teekonda - kas kodust tööle või vastupidi. Täidab funktsionaalset nõuet 6;
- Sõiduki liikumine serveripoolses osas (kliendipoolne osa): Muudeti auto teekonna kajastamist kaardil. Teekond muutub nüüd olenevalt auto sõitmise alguspunktist kodu-töö ja töö-kodu teekonna vahel. Autole klikkides on lisaks näha mitu meetrit teekonnast sõidetud on. Lisati sõida/peatu (*Drive/Stop*) nupp, mida klikkides saab mugavamalt auto sõitma panna ja peatada. Täidab funktsionaalset nõuet 4;
- Sõiduki liikumine serveripoolses osas (serveripoolne osa): Loodi loogika, mis salvestab auto läbitud teepikkuse sõidu alustamise momendist, peatamisel ning taas sõitma asumisel arvutab uue marsruudi sihtpunktini. Täidab funktsionaalseid nõudeid 4 ning 6;
- Kodu- ja tööasukohtade vaheline kaugus: loodi funktsionaalsused, et auto asukoha parameetrid lähtestada ning uuesti genereerida kodu ning töö asukohad. Täidab funktsionaalseid nõudeid 3 ja 12;
- Sõiduki aku laadimiskõver: Laadimine toimub kasutades reaalset laadimiskõverat. Täidab funktsionaalset nõuet 9;
- Sõiduki aku tühjenemiskõver: Aku tühjenemine toimub kasutades reaalset tühjenemise kõverat. Täidab funktsionaalset nõuet 10;
- Dockeri implementatsioon kasutajaliidese poolses osas: Loodi kasutajapoolse osa projektile *Dockerfile* [22] ning selle abil loodud *Docker Image* [23] konteiner laeti GitHub Packages registrisse. Täidab mittefunktsionaalset nõuet 3;
- Dockeri implementatsioon serveripoolses osas: Loodi serveripoolsele projekti osale *Dockerfile* ning selle abil loodud *Docker Image* konteiner laeti GitHub [24] Packages registrisse. Täidab mittefunktsionaalset nõuet 3;
- Avalikud laadimispunktid kaardil: Kaardil kuvatakse avalikke laadimispunkte. Laadimispunktid ilmuvad kaardi nähtavasse osasse markeritena, mida saab infokihile klikkides kaardile kuvada või ära võtta. Laadimispunktile klikkides näeb antud laadimispunkti kohta täiendavat infot. Täidab funktsionaalset nõuet 2;

- Pidevintegratsiooni ja -valmiduse konveieri implementatsioon: Peale igat koodi lisamist *master* harusse [25] loob uued *Docker Image* konteinerid, laeb need üles GitHub registrisse ja teavitab arendajaid Slackis [26]. Täidab mittefunktsionaalset nõuet 4.

Rakenduse arendamise tööprotsess oli jagatud viieks etapiks, millest töö kirjutamise hetkeks on möödunud neli ja valminud on suurem osa nõuetest. Funktsionaalsete nõuete hulgast on jäänud teha ainult üks: "Reaalsete kasutajate profiilid". Mittefunktsionaalsete nõuete hulgast on veel alustamata üks (SonarQube platvormi projekti lisamine) ja kaks on osaliselt tehtud (Docker ja GitHub Actions).

Järgnevalt on välja toodud ülesanded, mis on viimase etapi jooksul plaanitud teha:

- Reaalsete kasutajate profiilid: tuleb luua kolm kasutajaprofiili järgmiste omadustega ja erinevate sisendparemeetritega: kodunt lahkumise aeg, töölt lahkumise aeg, päevad, millal auto sõidab, aku maht ja kodu- ning tööasukohtade vaheline distants. Kõikide kasutajaprofiilide autod, kodu- ja tööasukohad ning teekonnad tuleb kuvada kaardile ning nende autode sõitmine automatiseerida sõltuvalt sisendparameetritest. Täidab funktsionaalset nõuet 13;
- SonarQube implementeerimine: Lisada projektile SonarQube platvorm, mis analüüsib Githubi erinevates harudes olevate muudatuste alla laadimist *master* harusse ning annab muudetud koodi kvaliteedi ning testide tulemuste kohta tagasisidet. Sealhulgas kiirendab arendusprotsessi edasiliikumist. Täidab mittefunktsionaalset nõuet 7;
- *Docker Compose* implementatsioon: lisada *Docker Compose* GitHub Actionsile, et kõiki GitHub Actions registrisse lisatud Dockeri konteinereid [6] samaaegselt käivitada nii, et nad saaksid üksteisega suhelda. Täidab mittefunktsionaalset nõuet 3;
- Implementeerida GitHub Actions kõikide harude ehitamiseks: loob funktsionaalsuse, millega saab iga haru ehitada ja automaatteste käivitada. Teavitab arendajaid ehituse ja testimise õnnestumisest Slack'i kaudu. Täidab mittefunktsionaalset nõuet 4.

Mitmeid mittefunktsionaalseid nõudeid saab tuua välja peaaegu igas kasutajaloos, neid peavad arendajad igapäevaselt silmas, näiteks objekt-orienteeritud programmeerimine, sõltuvuste sisestamine, testimine, serveripoolne osa on REST API ja kasutajapoolne osa on üheleherakendus.

2.2 Tööriistade kirjeldus

Arendamise keskkonnaks valiti IntelliJ IDEA [27]. Arendati RESTful API [28] Java rakendust kasutades Spring [29] raamistiku ja Gradle projekti [30] ehitustööriista. Lisaks kasutati Lombok tööriista [31], mis lihtsustab objektide loomist. Kasutajaliidese arendamiseks kasutati React JS-i ja selle testimiseks Jest raamistikku. Kogu projekt on kättesaadav GitHubis, mis võimaldab projekti paremini hallata ja meeskonnatööd optimiseerida. Serveripoolse osa testimiseks kasutatakse Mockito ja SpringRunner raamistikke. Manuaalseks API päringute testimiseks kasutati Postman rakendust. Samuti kasutati kaardi kuvamiseks Leafleti poolt loodud teeki, mis kasutab põhjaks OpenStreetMapi [32] vabavarakaarti. Virtuaalse elektrisõiduki olekute kontrollimiseks kasutati olekumasina tööriista. Marsruudi ehitamiseks kasutati OpenRouteService [33] ja Google Maps [34] teenuseid. Info ilmastikuolude kohta saadi kasutades OpenWeatherMap teenust. Serveripoolse ja kasutajaliidese projektid on automatiseeritud GitHub Actionsi abil ja üles laetud GitHub registrisse, kust neid on võimalik alla laadida ja Dockeri konteineritena käivitada. Tiimi omavaheliseks suhtlemiseks kasutati Slacki ning Skype'i ja aja dokumenteerimiseks Clockify rakendust.

2.3 Tööprotsessi kirjeldus

Eesmärkide saavutamiseks jätkatati Scrumi [35] ja Kanbani [36] agiilsete arendusmetoodikate kasutamisega. Sellel korral on arenduskäigule lisandunud ka etapid, mille käigus arendati valmis selleks planeeritud ülesanded, mis kujutasid endast serveri- või kasutajapoolse osa arendust ning olid autorite vahel võrdselt ära jagatud. Esimesed kolm etappi kestsid kolm nädalat ning viimane neli. Iga etapi lõpus toimus retrospektiiv koos järgmise etapi planeerimisega. Retrospektiivi käigus räägiti töö positiivsetest ja negatiivsetest aspektidest ning pakuti töö paremaks toimimiseks ideid.

Töö algas veebruaris ning kestab juuni lõpuni, mis tähendab, et lõputöö projekt ei ole veel valmis saanud ning aruanne tehakse pooliku töö kohta. Tööpäevadeks olid neljapäev ja reede. Iga tööpäeva alguses leidis aset püstijalakoosolek [37], millest võtsid osa tiimiliikmed ja Eesti Energia poolne scrum meister [38]. Püstijalakoosolekute ajal tutvustasid kõik kohalolevad tiimiliikmed lühidalt, mida tehti eelmisel tööpäeval, mida

on kavas teha käesoleval tööpäeval ja kas on esinenud probleeme. Kogu arendusjärgu käigus suhtlesid tiimiliikmed pidevalt omavahel suurima efektiivsuse saavutamiseks.

Alates märtsi algusest tehti tööd kodus riigi poolt kehtestatud eriolukorra tõttu, mis tõstis rakenduste nagu Skype'i ja Slacki kasutamise tiimi töös eriti tähtsale kohale.

Projekti kasutajalood on välja planeeritud Eesti Energia poolse arhitekti Aleksandr Skubi ja tooteomaniku [39] Jaan Otsa poolt. Samuti on kaasatud projekti töösse kolm juhendajat Allan Juhanson, Osama Muhamed ja Vinay Puranik ning Scrum master Ruslan Bjurkland.

Iga meeskonnaliige arendas enamasti iseseisvalt erinevaid käesoleva etapi kasutajalugusid, tehnilisi ülesandeid ning tööprotsessi jooksul tekkinud vigasid. Tavaliselt kui arendajal eelmine töö valmis sai, võeti järgmisena käsile ülesanne, mis kuulus arendusjärgule vastava etapi alla ning mida keegi veel arendama polnud hakanud. Kui antud etapp oli lõppjärgus ja ülesanded olid ära jagatud, aidati üksteisel ülesandeid lõpetada, mitte ei liigutud edasi järgmisesse arendustsüklisse kuuluvate ülesannetega. Mõnda keerulisemat ülesannet arendati juba algusest peale paarikaupa. Kõik meeskonnaliikmed olid *full-stack* arendaja rollis, Sten-Mark Paju ja Peeter Raudsepp tegelesid peamiselt kasutajalugudega, mis hõlmasid kautajaliidest. Kaarel Rohumaa, Tatjana Putškova ja Mark Porohnja tegelesid peamiselt serveripoolse osaga.

Tatjana Putškova tegeles auto sõitma paneku simuleerimisega ja marsruudi arendamisega serveripoolses osas(funktsionaalsed nõuded 4 ja 6). Samuti laadimise režiimidega(funktsionaalne nõue 8) ning reaalses režiimis aku laadimise ja tühjenemise(funktsionaalsed nõuded 9 ja 10) simuleerimisega. Peale seda arendas ta ühikteste implementeeritud loogika jaoks(mittefunktsionaalne nõue 12).

Mark Porohnja arendas lineaarset sõiduki aku laadimist ja tühjenemist(funktsionaalne nõue 7) ning sama kasutajaloo raames implementeeris ta olekumasina(mittefunktsionaalne nõue 1). Lisaks tegeles ta laadija parameetrite arendamisega(funktsionaalne nõue 14). Samuti arendas ta ühikteste(mittefunktsionaalne nõue 12). Peale selle töötas ta ka konfiguratsiooni failide uuesti laadimise võimalusega(mittefunktsionaalne nõue 9)

Peeter Raudsepp arendas serveripoolses osas avalikud laadimispunktid(funktsionaalne nõue 2) ja lisas rakendusse ilmateave küsimise(funktsionaalne nõue 11). Peale selle arendas ta peamiselt kasutajaliidest ja seal infokihtide abil kujutatavate sõidukite, teekondade ja avalike laadimispunktide dünaamilist muutmist (funktsionaalsed nõuded 1, 2, 3, 4, 5, 6). Arendatud loogika jaoks kirjutas ta ühikteste (mittefunktsionaalne nõue 12). Lisaks tegeles ta Dockeri projekti lisamisega(mittefunktsionaalne nõue 3).

Sten-Mark Paju tegeles peamiselt kasutajaliidese arendusega, mille käigus arendas ta projekti jaoks kasutajaliidese üheleherakenduse arhitektuuri järgi(mittefunktsionaalne nõue 10). Ta tegeles kaardi ja sellele kihtide(funktsionaalne nõue 1) loomisega, auto ja teiste objektide kasutajaliideses kuvamisega(funktsionaalsed nõuded 3, 4 ja 5), aitas Peeter Raudsepal avalikke laadijaid infokihina kaardile saada(funktsionaalne nõue 2), samuti tegeles auto liikumise simuleerimisega kasutajapoolses projekti osas(funktsionaalne nõue 6). Lisaks tegeles auto asukoha lähtestamise ja uuesti genereerimise võimalusega kasutajaliideses(funktsionaalne nõue 12).

Kaarel Rohumaa tegeles serveripoolses osas marsruudi arendamisega ja auto sõitma panemise simuleerimisega(funktsionaalsed nõuded 4 ja 6). Samuti implementeeris ta GitHub Actions konveieri(mittefunktsionaalne nõue 4). Sealhulgas arendas auto asukoha lähtestamise ja uuesti genereerimise serveripoolses rakenduses(funktsionaalne nõue 12). Lisaks kogu enda arendatud loogikale kirjutas ta ühikteste (mittefunktsionaalne nõue 12).

Meeskonnaliikmed täitsid kõiki nõudeid, mis olid seotud rakenduse arhitektuuriga(mittefunktsionaalsed nõuded 5, 6, 8 ja 11). Kuna rakendus pole kirjutamise hetkeks täielikult valmis, jäid mõned nõuded täitmata.

3 Töö tulemused

Järgnev osa on tehniline dokumentatsioon, mis koosneb täidetud nõuetest ning arhitektuuri ja disaini kirjeldusest. Samuti on välja toodud skeemid, mis selgitavad arendatud loogika algoritme ning koodi näidiseid.

Kuna esimene verstapost [40] (*milestone* 1) on eelmise projekti raames tehtud töö, siis selle projekti töö protsess oli jaotatud neljaks etapiks: esimene etapp (*milestone* 2), teine etapp (*milestone* 3), kolmas etapp (*milestone* 4) ja neljas etapp (*milestone* 5). Tööd kirjeldatakse etappide järjekorras, sel viisil saab tööprotsessi etapiviisiliselt jälgida. Iga etapp koosneb erinevatest kasutajalugudest, tehnilistest ülesannetest ja vigade (ingl *bug*) parandusest.

3.1 Esimene etapp

3.1.1 Ilmateave kättesaamine

Käesoleva kasutajaloo peamine kriteerium oli hetke ilmastiku info kättesaamine ja selle kasutamine simulatsiooni realistlikumaks muutmiseks. Põhiline ja arendajatele peamiselt vajalik ilmastikuolu on temperatuur, mis mõjutab elektriautode aku laadimise ja tühjenemise kiiruse efektiivsust. Vajalikud kalkulatsioonid efektiivsuse muutumise arvutamiseks saadi Eesti Energia poolselt arhitektilt, mida on võimalik näha kuvatõmmisel 3.1.1.2. Ilmastiku saamiseks kasutatakse OpenWeatherMap API-t. Sinna saadetakse API GET päring koos koordinaatidega ning tagastatakse hetkeline ilm antud koordinaatidega kohas. Selleks kasutajalooks loodi kolm klassi: *WeatherConditions*, *WeatherService* ja *WeatherConfiguration*. *WeatherConditions* on ilma klass koos vajalike tunnustega, millele omistatakse väärtused OpenWeatherMap API-st *WeatherService* klassi abil. *WeatherService* klassi peamine ülesanne ongi API-ga suhelda ja iga minuti tagant *WeatherConditions* infot muuta. Selleks on loodud meetodid *getWeatherFromAPI* ja *updateWeatherConditions* (vt. kuvatõmmis 3.1.1.1). *WeatherConfiguration* klass hoiab endas API päringu URL-i [41] ja aplikatsiooni identifikaatorit, mis on vajalikud projekti API-ga ühendamiseks. See klass sai loodud,

kuna projekti disainis tuleb vältida sisse kodeeritud väärtusi ning kõike peaks saama vajadusel konfigureerida. *WeatherConditions* klassi saab vajadusel ka ise tulevikus kasutada simulatsiooni ilmastikuolude määramiseks, kui selleks vajadus tekib. See kasutajalugu on ka kajastatud sügissemestril tehtud Meeskonnaprojekti dokumentatsioonis, kuid see ei saanud selle jooksul täiesti valmis ning oli vaja ära lõpetada.

```java
@RequiredArgsConstructor
@Service
@Slf4j
public class WeatherService {
    private final VehicleProperties vehicleProperties;
    private final WeatherConfiguration weatherConfiguration;
    private WeatherConditions weatherConditions;

    public WeatherConditions getWeather() { return weatherConditions; }

    public WeatherConditions getWeatherFromAPI() {
        double[] coordinates = getCoordinates();
        if (coordinates.length < 2) return null;
        return Unirest.get(weatherConfiguration.getWeatherUrl())
                .queryString( name: "lat", coordinates[0])
                .queryString( name: "lon", coordinates[1])
                .queryString( name: "units",  value: "metric")
                .queryString( name: "appid", weatherConfiguration.getAppId())
                .asObject(WeatherConditions.class)
                .getBody();
    }

    @Scheduled(cron = "0 0/1 * * * ?")
    public void updateWeatherConditions() {
        log.info("Updating weather conditions from API. Initial value: {}", weatherConditions);
        weatherConditions = getWeatherFromAPI();
        log.info("Internal weather conditions was set to the new value from API. Value: {}", weatherConditions);
    }

    protected double[] getCoordinates() {
        var latitude = vehicleProperties.getHomeAddress().getCoordinate().getLatitude();
        var longitude = vehicleProperties.getHomeAddress().getCoordinate().getLongitude();
        return new double[]{latitude, longitude};
    }
```

Kuvatõmmis 3.1.1.1. *WeatherService* klass ilmastikuolude saamiseks API-st ja projektis loodud *WeatherConditions* info uuendamiseks.

```java
@Override
public double getChargeCoefficientBasedOnTemp() {
    double baseTemperature = properties.getBaseTemperature();
    double temp = weatherService.getWeather().getMain().getTemp();
    double currentEfficiency = properties.getChargeEfficiencyPercentage() * 1.0 / 100;
    double coefficient = currentEfficiency - (baseTemperature - temp) * 0.01;
    return Math.min(1, coefficient);
}

@Override
public double getDischargeCoefficientBasedOnTemp() {
    double baseTemperature = properties.getBaseTemperature();
    double temp = weatherService.getWeather().getMain().getTemp();
    return 1 + (baseTemperature - temp) * 0.01;
}
```

Kuvatõmmis 3.1.1.2. Laadimiskiiruse ja aku vähenemise kiiruse kalkulatsioonid *VehicleServiceImpl* klassis.

3.1.2 Laadija parameetrid

Antud kasutajaloo valmimisnõudeks oli laadimisjaamade kontseptsiooni loomine. Selle eesmärgi saavutamiseks oli vaja luua klass *ChargerProperties*(vt. kuvatõmmis 3.1.2.1).

Laadimisvõimsuse järgi arvutatakse akule lisatav laeng. Laadimisprotsess ise loodi hilisemas arendusetapis.

```
@Configuration
public class ChargerProperties {
    @Getter
    private final double CHARGING_POWER_KWH = 50.0;
}
```

Kuvatõmmis 3.1.2.1. *ChargerProperties* klass

Projekti praeguses versioonis kasutatakse laadimisvõimsusena *CHAdeMO* [42] standardi ELMO laadija võimsuse väärtust, mis vastab kiirlaadimise standarditele.

3.1.3 Sõidukid liiguvad kasutades reaalseid marsruute

Käesoleva kasutajaloo esimene eesmärk oli virtuaalse elektriauto sõitma panemine kodust tööle ja vastupidiste marsruutidega, kasutades selleks reaalseid teid. Teiseks eesmärgiks oli arvutada tegelik energiatarbimine marsruudi jaoks.

Antud loos toimus arendamine ainult serveri poolel. Selleks, et auto sõita saaks pidi olemas olema marsruut. Marsruudi genereerimisega on seotud kümme klassi: *OpenRouteService*, *RouteRequest*, *RouteResponse*, *RouteResponseMetadata*, *RouteResponseQuery*, *Summary*, *RouteConfiguration*, *RouteController* ning *RouteService*.

Reaalse kalkulatsiooni teeb ära OpenRouteService teenus. Selleks saadetakse *https://api.openrouteservice.org/v2/directions/driving-car* lõpp-punktile POST taotlus, mis sisaldab endas algus- ja lõppkoordinaate. Nendeks koordinaatideks on projektis kodu ja töökoha koordinaadid. Taotluses on veel tõeväärtuse andmetüüpi *instructions*, mis on väärtusega *false*. See määrab ära, et vastusesse pole vaja panna navigeerimise sõnalisi juhiseid. Taotluse sisu loob *createRequest* meetod ja taotluse saadab *getResponse* meetod, kasutades Unirest teeki. Kuvatõmmises 3.1.3.1 on välja toodud *createRequest* ja *getResponse* meetodid.

```java
public OpenRouteServiceResponse getResponse(GeoCoordinate start, GeoCoordinate destination) {
    return Unirest.post(routeConfiguration.getRouteUri().toString())
            .header( name: "Authorization", routeConfiguration.getAuthorizationToken())
            .header( name: "Accept", value: "application/json")
            .header( name: "Content-Type", value: "application/json; utf-8")
            .body(createRequest(start, destination))
            .asObject(OpenRouteServiceResponse.class).getBody();
}

private RouteRequest createRequest(GeoCoordinate start, GeoCoordinate destination) {
    RouteRequest request = new RouteRequest(start, destination);
    request.setInstructions(false);

    return request;
}
```

Kuvatõmmis 3.1.3.1. *getResponse* ja *createRequest* meetodid *OpenRouteService* klassis

Kõik *OpenRouteService* klassist saadud tulemused kaardistatakse vastavatesse klassidesse. Kõige suurem klass, mis hõlmab kõiki teisi klasse on *OpenRouteServiceResponse*. *RouteResponse* klassis peitub kõige vajalikum info. *Summary* klassis hoitakse teepikkust ja arvatavat aega, mis läbimiseks kulub. Sõne *geometry* on kodeeritud loend koordinaatidest, mis moodustavad marsruudi. Selle dekodeerimiseks kasutavad autorid Google Maps Platform klassi *PolylineEncoding* meetodit *decode*. Loendit *wayPoints* saab kasutada dekodeeritud marsruudi kontrollimiseks. *RouteResponseMetadata* klass sisaldab endas infot päringu metaandmete kohta. *OpenRouteServiceResponse*, *RouteResponse* ja *RouteResponseMetadata* klassid on näidatud kuvatõmmises 3.1.3.2.

```java
@Data
public class OpenRouteServiceResponse {
    private List<RouteResponse> routes;
    private List<Double> bbox;
    private RouteResponseMetadata metadata;
}

@Data
public class RouteResponse {
    private Summary summary;
    private List<Double> bbox;
    private String geometry;
    @SerializedName("way_points")
    private List<Integer> wayPoints;
}

@Data
@JsonIgnoreProperties(ignoreUnknown = true)
public class RouteResponseMetadata {
    private String attribution;
    private String service;
    private Long timestamp;
    private RouteResponseQuery query;
}
```

Kuvatõmmis 3.1.3.2. *OpenRouteServiceResponse*, *RouteResponse* ja *RouteResponseMetadata* klassid

Kuvatõmmises 3.1.3.3 on näidatud marsruudi päringu tulemus, mis sisaldab alg- ja lõpp koordinaate, marsruudi pikkust, marsruudi elektritarbimisest ja teekonnapunktide koordinaatide loendit.

GET http://localhost:8080/routeFromHomeToWork

Params Authorization Headers (7) Body Pre-request

Query Params

KEY

Key

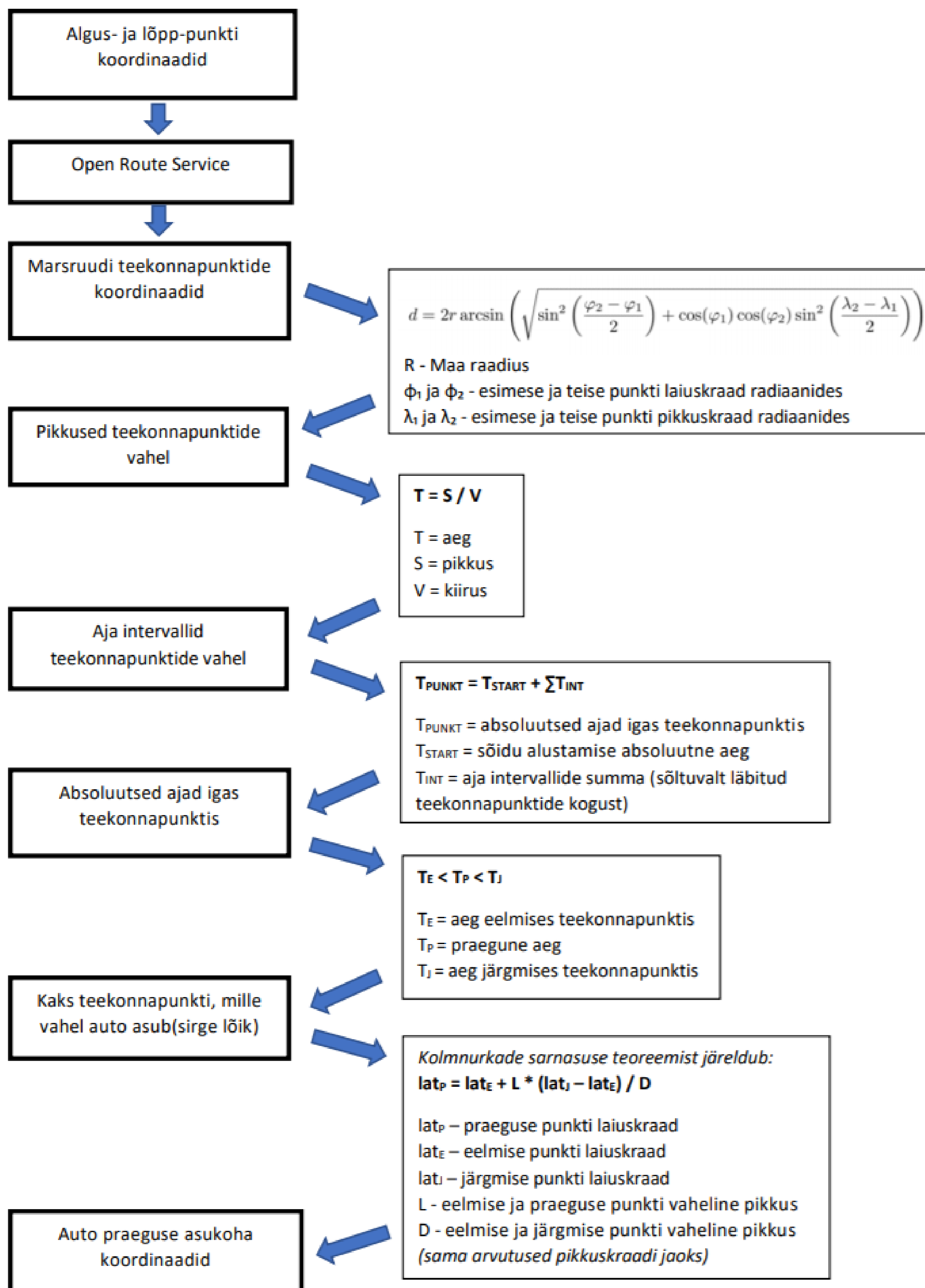
Body Cookies Headers (8) Test Results

Pretty Raw Preview Visualize JSON


```
{
    "startingPoint": {
        "lat": 59.41577,
        "lng": 24.720860000000002
    },
    "destinationPoint": {
        "lat": 59.42963,
        "lng": 24.725160000000002
    },
    "distance": 2078.5,
    "electricityConsumptionWh": 342.95250000000004,
    "wayPointsCoordinates": [
        {
            "lat": 59.41577,
            "lng": 24.720860000000002
        },
        {
            "lat": 59.415850000000006,
            "lng": 24.720540000000003
        },
```


Kuvatõmmis 3.1.3.3. */routeFromHomeToWork* GET päringu tulemus

Kui auto sõidab, tähendab et see asub kuskil marsruudil, mitte alg- või lõpp-punktis. Sõiduki praegune asukoht arvutatakse skeemis 3.1.3.4 näidatud algoritmi kasutades.



Skeem 3.1.3.4. Sõiduki praeguse asukoha koordinaatide leidmine

Kuvatõmmises 3.1.3.5 on näidatud *RouteService* klassis asuv praeguse asukoha leidmise privaatne meetod.

```java
private GeoCoordinate getCurrentLocationWhenStatusDriving(long time) {
    int indexOfNextWayPoint = getIndexOfNextWayPointAfterCurrent(route.getTimesInEachWayPoint(), time);

    if (isArrivedInTargetPoint(indexOfNextWayPoint)) {
        return vehicleState.getTargetLocation();
    }

    List<Long> timesInWayPoints = route.getTimesInEachWayPoint();
    List<LatLng> wayPoints = route.getWayPointsCoordinates();

    long previousWayPointTime = timesInWayPoints.get(indexOfNextWayPoint - 1);

    LatLng previous = wayPoints.get(indexOfNextWayPoint - 1);
    LatLng next = wayPoints.get(indexOfNextWayPoint);

    var current = new GeoCoordinate();
    double diffLat = next.lat - previous.lat;
    double diffLng = next.lng - previous.lng;

    long timeBetweenCurrentAndPrevious = time - previousWayPointTime;
    long timeBetweenNextAndPrevious = route.getTimesBetweenWayPoints().get(indexOfNextWayPoint - 1);
    double distancesRatio = (double) timeBetweenCurrentAndPrevious / (double) timeBetweenNextAndPrevious;
    current.setLatitude(previous.lat + distancesRatio * diffLat);
    current.setLongitude(previous.lng + distancesRatio * diffLng);

    return current;
}
```

Kuvatõmmis 3.1.3.5. *RouteService* klassis asuv *getCurrentLocationWhenStatusDriving* meetod

Kuvatõmmises 3.1.3.6 on näidatud asukoha päringu tulemus, mis saadi */location* lõpp-punktilt GET päringu kaudu. Saadud vastus sisaldab praeguse asukoha koordinaate.

```java
@RestController
@AllArgsConstructor
public class RouteController {
    private RouteService routeService;

    @GetMapping("/routeFromHomeToWork")
    public Route getRouteFromHomeToWork() {
        return routeService.getRouteFromHomeToWork();
    }

    @GetMapping("/routeFromWorkToHome")
    public Route getRouteFromWorkToHome() {
        return routeService.getRouteFromWorkToHome();
    }

    @GetMapping("/location")
    public GeoCoordinate getLocation() {
        return routeService.getLocation();
    }
```

GET http://localhost:8080/location

Params | Authorization | Headers (7) | Bod

Query Params

KEY

Key

Body | Cookies | Headers (8) | Test Results

Pretty | Raw | Preview | Visualize | JSC


```json
{
    "latitude": 59.415497462063136,
    "longitude": 24.720614321378946
}
```


Kuvatõmmis 3.1.3.6. *RouteController*'i */location* lõpp-punkt ja selle GET päringu tulemus

3.2 Teine etapp

3.2.1 Kaardipõhine simulaator

Antud kasutajaloo vastuvõtvad kriteeriumid muutusid korduvalt. Selle põhjuseks oli kasutajaloo mittevalmimine ITB1706 - Meeskonnaprojekt käigus ning aja möödudes uute

nõuete lisandumine. Kuna meeskonnaprojektiga jätkati jaanuaris lõputööna, siis kandus see kasutajalugu üle esialgu teise ning sealt edasi kolmandasse etappi, kus see lõpuks ka valmis sai. Selle aja jooksul lisandus juurde mõningaid eraldiseisvaid kasutajalugusid antud kasutajaloosse. Lõplikeks kriteeriumiteks kujunes luua interaktiivne kaart, millel kujutatakse kasutajate elektriautosid, kodu- ja tööaadresse ning auto marsruuti kodust tööle (vt. kuvatõmmis 3.2.1.6). Samuti saab autode, kodu ja töö asukohtade, teekonna kaardile kuvamist sisse ja välja lülitada ning nendele klikkides saab kuvada ka nende kohta infot. Kogu info nende jaoks tuleb API päringute kaudu, mis saavad oma info serveripoolsest projekti Rest API osast, kus toimub info genereerimine. Valminud kaart on baasiks edasiseks arenduseks. Kaardi komponentide kujunduseks kasutati Material UI disaine ning testimiseks kasutati Jest testimise raamistikku.

Selle kasutajaloo valmistamiseks kasutati React.JS ja Leaflet teeke. Leaflet on Javascripti vabavara teek(ingl *open source library*), millega saab luua kaardi ja sellele lisada erinevaid komponente nagu näiteks markereid. Leafleti teek kasutab OpenStreetMapi oma baasina. OpenStreetMap on vabavarakaart, mis on loodud ja täiendatud kasutajate poolt. Kergemaks Leaflet teegi integreerimiseks kasutasid autorid React-Leafleti, mis muudab Leafleti Javascript koodi JSX kujule ehk muudab nad React komponentideks.

Antud kasutajaloo valmimiseks loodi kaks klassi, *MapComponent* ja *LayersComponent*. *LayersComponent* on *MapComponent*'i alamklass, kus luuakse kaardile infokihid ning markerid. *LayersComponent* koosneb Leaflet pistikprogrammidest, mis on võetud Leaflet raamistikust ning muudab need ümber Javascript keelest JSX-i React Leaflet funktsiooniga *createLeafletElement*.

MapComponent on nö. olekuga (ingl *stateful*) ehk tark komponent (ingl *smart component*), mille olekus asub kogu serveripoolse projekti REST API-lt saadud info. *MapComponent* klassi renderdamisel saadetakse kogu vajalik info atribuutidena (*property*) edasi *LayersComponent* klassile, mis kasutab antud informatsiooni markerite ja kihikihtide loomiseks kaardile, millega saab eelmainitud markereid sisse ja välja lülitada. *LayersComponent* on olekuta(ingl *stateless*). Kui *MapComponent* klassi olek muutub, toimub uus info vahetus *LayersComponent* ja *MapComponent* klasside vahel.

MapComponent klassi funktsionaalsus koosneb kahest peamisest meetodist: *componentDidMount* ja *render* (vt. kuvatõmmised 3.2.1.1. ja 3.2.1.2).

ComponentDidMount meetod suhtleb Axios teegi abil serveripoolse API-ga kahe GET päringuga ning annab *MapComponent* klassi olekus asuvatele *vehicle* ja *routeFromHomeToWork* massiividele väärtused. Render kasutab neid samu massiivi väärtuseid ning tagastab renderdatud kaardi. Kui kaarti renderdatakse, kutsutakse välja *LayersComponent*, millele antakse edasi atribuutide kujul massiivide *vehicle* ja *routeFromHomeToWork* väärtused.

```
componentDidMount() {
    axios.get( url: "http://localhost:8080/configuration").then(
        result => {
            this.setState( state: {vehicle: result.data});
        },
        error => {
            this.setState( state: {
                isLoaded: true,
                error
            });
        }
    );
    axios.get( url: "http://localhost:8080/routeFromHomeToWork").then(
        result => {
            this.setState( state: {isLoaded: true, routeFromHomeToWork: result.data});
        },
        error => {
            this.setState( state: {isLoaded: true, error});
        }
    );
}
```

Kuvatõmmis 3.2.1.1. *MapComponent componentDidMount* meetod

```
render() {
    const {error, isLoaded, vehicle, routeFromHomeToWork} = this.state;
    if (error) {
        return <div>Error: {error.message}</div>
    }
    if (!isLoaded) {
        return <div>Loading...</div>
    }

    const routes = this.mapData(routeFromHomeToWork.wayPointsCoordinates);
    const routeDistance = routeFromHomeToWork.distance;
    const routeElectricityConsumptionWh = routeFromHomeToWork.electricityConsumptionWh;

    return (
        <div>
            <Map className="map" zoom={this.state.zoom} ref={this.mapRef} center={[58.88, 25.54]}>
                <TileLayer
                    attribution='&amp;copy <a href="http://osm.org/copyright">OpenStreetMap</a> contributors'
                    url="https://{s}.tile.openstreetmap.org/{z}/{x}/{y}.png"
                />
                <SearchComponent/>
                <LayersComponent vehicleData={[vehicle]} routeData={routes} routeDistanceData={routeDistance}
                                 routeElectricityConsumptionData={routeElectricityConsumptionWh}/>
            </Map>
        </div>
    )
}
```

Kuvatõmmis 3.2.1.2. *MapComponent render* meetod

LayersComponent klassi peamine meetod on *createLeafletElement*, mille sisendiks on *MapComponent*'ist saadud omadused (vt. kuvatõmmis 3.2.1.3). Neid kasutatakse infokihi gruppide (ingl *layersGroup*) loomiseks *addLayerGroups* abimeetodi abil, millega lisatakse kaardile infokihi komponent, mis hoiab endas infokihi gruppidena auto, töö- ja koduasukoha markereid ning teekonda (vt. kuvatõmmis 3.2.1.4). Kõik need on ka omakorda Leaflet elemendid. Infokihtide komponent on koodis tähistatud *L.control.layers*'iga, infokihi grupid *L.layergroup*'iga, auto ning töö- ja koduasukohad on *L.marker*'id ja teekond on *L.polyline*. L nende ees tähendab, et need on Leafleti elemendid.

```
createLeafletElement(props :Props ) {
    let vehicleVar = (this.props.vehicleData);
    let routeVar = (this.props.routeData);
    let routeDistance = (this.props.routeDistanceData);
    let routeElectricityConsumption = (this.props.routeElectricityConsumptionData);
    let vehicleMarkers = [];
    let homeMarkers = [];
    let workMarkers = [];
    let randomMarkers = [];
    let routes = [];
    let routeMarkers = [];
    for (let i = 0; i < vehicleVar.length; i++) {
        pushMarkers(vehicleVar, vehicleMarkers, i, getIcon( iconName: 'directions_car'),  id: 1);
        pushMarkers(vehicleVar, homeMarkers, i, getIcon( iconName: 'house'),  id: 2);
        pushMarkers(vehicleVar, workMarkers, i, getIcon( iconName: 'business'),  id: 3);

    }
    for (let i = 0; i < routeVar.length; i++) {
        routes.push(L.latLng(routeVar[i].lat, routeVar[i].lng));
    }
    var polyline = L.polyline([routes], {
        color: "#B80000",
        weight: 3,
        opacity: 1,
        smoothFactor: 1
    }).bindPopup("Route distance: " + routeDistance + "m" + "<dl>Route electricity consumption in Wh: " + (routeElectricityConsumption).toFixed( fractionDigits: 2));
    routeMarkers.push(polyline);
    return addLayerGroups(vehicleMarkers, homeMarkers, workMarkers, randomMarkers, routeMarkers);

}
```

Kuvatõmmis 3.2.1.3. *LayersComponent createLeafletElement* meetod

```
function addLayerGroups(vehicleMarkers, homeMarkers, workMarkers, randomMarkers, routeMarkers) {
    const vehicles = L.layerGroup(vehicleMarkers);
    const homes = L.layerGroup(homeMarkers);
    const workplaces = L.layerGroup(workMarkers);
    const routes = L.layerGroup(routeMarkers);

    const overlayMaps = {
        "Vehicles": vehicles,
        "Homes": homes,
        "Workplaces": workplaces,
        "Routes": routes
    };
    return new L.control.layers(null, overlayMaps);
}
```

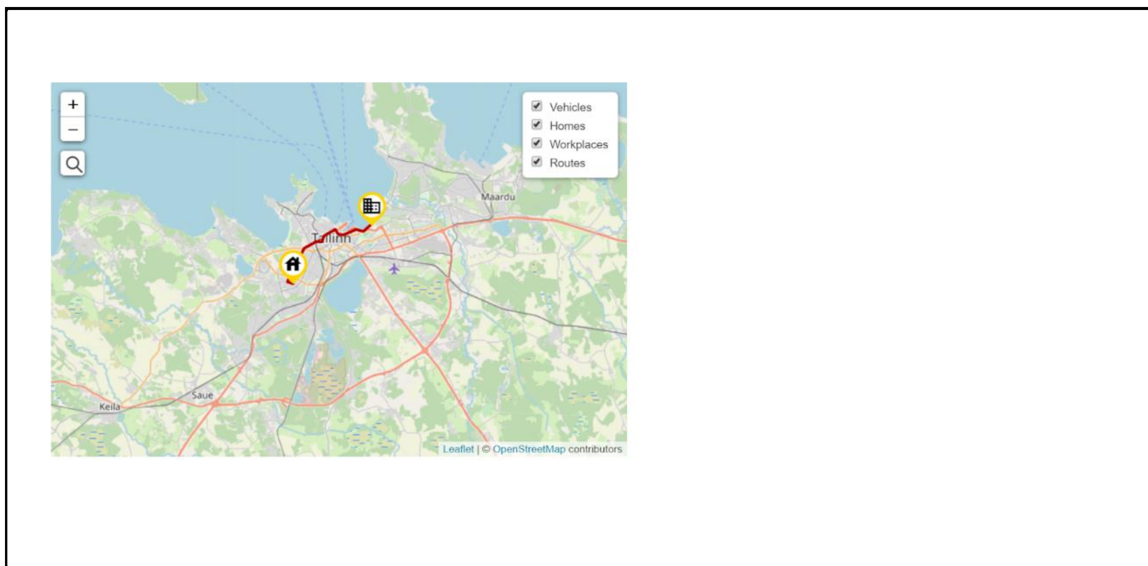
Kuvatõmmis 3.2.1.4. *LayersComponent addLayerGroups* meetod

Testida oli vaja *MapComponent* klassi. Selleks tuli luua imiteeritud väärtused ning nende abil kasutades Jest meetodit *shallow MapComponent* renderdada ja siis testida, et

MapComponent renderdab ja saab sisendiks õiged väärtused. Teste on võimalik näha kuvatõmmisel 3.2.1.5.

```
beforeEach(() =>{
    wrapper = shallow(<MapComponent/>);
    wrapper.setState({
        routeFromHomeToWork: routeData,
        vehicle: vehicles,
        isLoaded: true
    });
});

describe("MapComponent", () => {

    describe("when state changed", () => {
        it("should fetch route from home to work data and vehicles", () => {
            expect(wrapper.exists()).toBe(true);
            expect(wrapper.state().isLoaded).toEqual(true);
            expect(wrapper.state().routeFromHomeToWork).toEqual(routeData);
            expect(wrapper.state().vehicle).toEqual(vehicles);
        });
        it("renders", () => {
            expect(wrapper.exists()).toBe(true);
            expect(wrapper.state().isLoaded).toEqual(true);
            console.log(wrapper.debug());
        });
        it("has distance data from routeFromHomeToWork", () =>{
            expect(wrapper.state().routeFromHomeToWork.distance).toEqual(routeData.distance);
        });
        it("has electricityConsumptionWh data from routeFromHomeToWork", () =>{
            expect(wrapper.state().routeFromHomeToWork.electricityConsumptionWh).toEqual(routeData.electricityConsumptionWh);
        });
        it("has wayPointsCoordinates data from routeFromHomeToWork", () =>{
            console.log(wrapper.state().routeFromHomeToWork.wayPointsCoordinates);
            expect(wrapper.state().routeFromHomeToWork.wayPointsCoordinates).toEqual(routeData.wayPointsCoordinates);
        });
        it("Contains component LayersComponent", () => {
            expect(wrapper.contains(<LayersComponent vehicleData={[wrapper.state().vehicle]} routeData={wrapper.state().routeFromHomeToWork.wayPointsCoordinates}
                                                     routeDistanceData={wrapper.state().routeFromHomeToWork.distance} routeElectricityConsumptionData={wrapper.state().routeFromHomeToWork.electricityConsumptionWh}/>)).toBe(true);
        });
        it("Contains component SearchBar", () => {
            expect(wrapper.contains(<SearchComponent/>)).toBe(true);
        });
    });
});
```

Kuvatõmmis 3.2.1.5. *MapComponent* testiklass



Kuvatõmmis 3.2.1.6. interaktiivne kaart koos markerite ja infokihtidega

3.2.2 Aku laadimine ja tühjenemine

Selle kasutajaloo jooksul tuli lisada serveripoolsele rakendusele elektriauto olek ja selle järgi aku laadimistaseme lineaarne muutmine. Näiteks kui auto sõidab ehk tema olek on *DRIVING*, siis hakkab aku tühjenema. Aku tühjenemise arvutamisel arvestatakse aku tühjenemise kiirusega kilomeetri ja läbitud vahemaa kohta. Auto laadimise protsess toimub sarnaselt. Laadimise alustamiseks tuleb auto seada olekusse *CHARGING*.

Lisatava laengu arvutamisel võetakse aluseks omadused nagu laadimisjaama võimsus ja aku laadimisefektiivsus.

Olekute vahetamiseks otsustati kasutada Springi poolt loodud olekumasinat *Spring Statemachine*, kus on 2 peamist elementi: olek ja sündmus. Olekumasina konfiguratsioonis registreeritakse kõik olekute üleminekud koos neid käivitavate sündmustega. Elektrisõidukid saavad ühest olekust minna ainult kindlatesse teistesse olekutesse. Neid üleminekuid näeb kuvatõmmisel 3.2.2.1. Antud projektis kasutati olekuteks ja sündmusteks sama *enum* tüüpi klassi, mis koosneb neljast elemendist: *IDLE*, *DRIVING, PLUGGED_IN ja CHARGING*. Oleku muutmisel käivitatakse sündmusi Action-tüüpi meetodite abil, näiteks *chargingAction* kui auto olek muutub *PLUGGED_IN* olekust *CHARGING* olekusse.

```
stateMachineBuilder.configureTransitions()  StateMachineTransitionConfigurer<VehicleState.Status, VehicleState.Status>
        .withExternal()  ExternalTransitionConfigurer<VehicleState.Status, VehicleState.Status>
        .source(Status.IDLE).target(Status.PLUGGED_IN)
        .event(Status.PLUGGED_IN)
        .and()  StateMachineTransitionConfigurer<VehicleState.Status, VehicleState.Status>

        .withExternal()  ExternalTransitionConfigurer<VehicleState.Status, VehicleState.Status>
        .source(Status.PLUGGED_IN).target(Status.CHARGING)
        .event(Status.CHARGING)
        .action(chargingAction())
        .and()  StateMachineTransitionConfigurer<VehicleState.Status, VehicleState.Status>

        .withExternal()  ExternalTransitionConfigurer<VehicleState.Status, VehicleState.Status>
        .source(Status.IDLE).target(Status.DRIVING)
        .event(Status.DRIVING)
        .action(drivingAction())
        .and()  StateMachineTransitionConfigurer<VehicleState.Status, VehicleState.Status>
```

Kuvatõmmis 3.2.2.1. Olekumasina üleminekute määratlemine

Auto oleku muutmiseks tehakse API PUT päring. Näiteks oleku muutmiseks väärtuseks *DRIVING* tuleb teha PUT taotlus lõpp-punktile */state/status*, mille sisendiks on *DRIVING*. Kuvatõmmisel 3.2.2.2 on näidatud REST lõpp-punkt auto oleku muutmiseks HTTP PUT päringuga.

```
@PutMapping("/state/status")
public Status updateStatus(@RequestBody Status status) {
    vehicleService.updateStatus(status);
    return vehicleService.getStatus();
}
```

Kuvatõmmis 3.2.2.2. REST lõpp-punkt elektrisõiduki oleku muutmiseks *VehicleController* klassis

Kuvatõmmisel 3.2.2.3 on näidatud meetod *chargingAction*. See meetod kasutab *ThreadPoolTaskScheduler* klassi, et käivitada meetod *chargeTask* eraldi lõimes iga ajavahemiku tagant. Antud juhul on ajavahemikuks üks sekund.

```
@Bean
public Action<Status, Status> chargingAction() {
    return context -> {
        Duration interval = Duration.ofSeconds(1);
        var trigger = new PeriodicTrigger(interval.toMillis());
        taskScheduler.initialize();
        taskScheduler.schedule(chargeTask(interval), trigger);
    };
}

private Runnable chargeTask(Duration interval) {
    return () -> {
        synchronized (VehicleState.class) {
            vehicleService.charge(interval);
        }
    };
}
```

Kuvatõmmis 3.2.2.3. *Action*-tüüpi meetod *chargingAction*, mis käivitatakse *CHARGING* olekusse üleminekul

Kuvatõmmisel 3.2.2.4. on kujutatud aku laadimise meetodid. Lisatava laengu arvutamiseks kasutatakse meetodit *getChargeRateInWattHours*, mille käigus arvutatakse, kui palju vatt-tunde teatud aja jooksul lisatakse, võttes arvesse aku laadimise efektiivsust ja laadimisvõimsust. Kui aku on laetud (maksimaalne maht on saavutatud), siis autot enam ei laeta.

```java
@Override
public void charge(Duration interval) {
    var stateOfChargeWh = vehicleState.getStateOfChargeWh();
    var batteryCapacityWh = vehicleProperties.getBatteryCapacitykWh() * 1000;

    if (stateOfChargeWh >= batteryCapacityWh - 100) {
        var SoC = stateOfChargeWh / batteryCapacityWh * 100;
        log.info(String.format("Battery is charged, SoC: %s%%", SoC));
        return;
    }

    var chargeRate = getChargeRateInWattHours(interval);
    var newStateOfChargeWh = stateOfChargeWh + chargeRate;
    vehicleState.setStateOfChargeWh(newStateOfChargeWh);
    log.info(String.format("State of Charge: %s watt-hours", newStateOfChargeWh));
}

private double getChargeRateInWattHours(Duration interval) {
    double chargingEfficiency = vehicleProperties.getChargeEfficiencyPercentage() / 100d;
    double chargingPowerInWattHoursPerInterval = ChargerProperties.CHARGING_POWER_KW / 60 / 60 * interval.toMillis();
    return chargingPowerInWattHoursPerInterval * chargingEfficiency;
}
```

Kuvatõmmis 3.2.2.4. Aku laadimiseks meetodid *VehicleService* klassis

Kuvatõmmisel 3.2.2.5 näha olev meetod *drivingAction* käivitatakse kui elektriauto läheb *DRIVING* olekusse. Siin kasutatakse meetodeid *getRoute* ja *updateTargetLocationAccordingToCurrent*, mis muudavad auto sihtpunkti praeguse asukoha järgi.

```java
@Bean
public Action<Status, Status> drivingAction() {
    return context -> {
        vehicleService.updateTargetLocationAccordingToCurrent();

        try {
            routeService.getRoute(vehicleState.getCurrentLocation(), vehicleState.getTargetLocation());
        } catch (IOException e) {
            log.warn(e.getMessage());
        }

        Duration interval = Duration.ofSeconds(1);
        var trigger = new PeriodicTrigger(interval.toMillis());

        taskScheduler.initialize();
        vehicleState.setDrivingStartTime(System.currentTimeMillis());
        taskScheduler.schedule(dischargeTask(), trigger);
    };
}

private Runnable dischargeTask() {
    return () -> {
        synchronized (VehicleState.class) {
            vehicleService.updateCurrentLocation();
            vehicleService.discharge();
        }
    };
}
```

Kuvatõmmis 3.2.2.5. *Action*-tüüpi meetod *drivingAction*, mis käivitatakse *DRIVING* olekusse üleminekul

Meetodis *drivingAction* kasutatakse ka *ThreadPoolTaskScheduler* klassi, et käivitada *dischargeTask* eraldi lõimes iga ajavahemiku järel. Kuvatõmmisel 3.2.2.6. on näidatud, et praeguse asukoha värskendamiseks kasutatakse *RouteService*'i meetodit *getLocation*.

```
@Override
public void updateCurrentLocation() {
    var newCurrent = routeService.getLocation();
    vehicleState.setPreviousLocation(vehicleState.getCurrentLocation());
    vehicleState.setCurrentLocation(newCurrent);
}
```

Kuvatõmmis 3.2.2.6. Meetod *updateCurrentLocation*, mis värskendab praegust auto asukohta

Meetodit *updateTargetLocationAccordingToCurrent* näeb kuvatõmmisel 3.2.2.7. Sihtpunkti muudetakse vastavalt järgmisele reeglile: kui auto on kodus, siis sihtpunkt on töö, kui auto on tööl, siis sihtpunkt on kodu.

```
@Override
public void updateTargetLocationAccordingToCurrent() {
    var homeLocation = properties.getHomeAddress().getCoordinate();
    var workLocation = properties.getWorkAddress().getCoordinate();
    var currentLocation = vehicleState.getCurrentLocation();

    if (currentLocation.equals(homeLocation)) {
        vehicleState.setTargetLocation(workLocation);
    }
    if (currentLocation.equals(workLocation)) {
        vehicleState.setTargetLocation(homeLocation);
    }
}
```

Kuvatõmmis 3.2.2.7. Meetod *updateTargetLocationAccordingToCurrent*, mis värskendab auto sihtkohta

Kuvatõmmisel 3.2.2.8 näha olev meetod *discharge* kutsutakse iga sekundi tagant välja. *Discharge* meetod kasutab omakorda meetodit *getDischargeInWattHours*, et teada saada, kui palju elektrisõiduki aku tühjeneb. Selle jaoks võetakse koordinaadid, kus auto oli üks sekund tagasi ja koordinaadid, kus auto parasjagu asub. Seejärel arvutatakse nende kahe koordinaadi vaheline kaugus meetrites ning korrutatakse saadud tulemus vahemaa konstandiga, mis näitab, kui palju tühjeneb elektrisõiduki aku ühe meetri kohta.

```java
@Override
public void discharge() {
    double stateOfChargeWh = vehicleState.getStateOfChargeWh();

    if (stateOfChargeWh <= 100d) {
        log.info(String.format("low battery: %s watt-hours left", stateOfChargeWh));
        return;
    }

    double discharge = getDischargeInWattHours(
            vehicleState.getPreviousLocation(), vehicleState.getCurrentLocation());
    double newStateOfChargeWh = stateOfChargeWh - discharge;
    vehicleState.setStateOfChargeWh(newStateOfChargeWh);
    log.info(String.format("%s watt-hours remaining", newStateOfChargeWh));
}

private double getDischargeInWattHours(GeoCoordinate previousLocation, GeoCoordinate currentLocation) {
    double distance = addressGenerator.calculateDistanceInMetersBetween(previousLocation, currentLocation);
    return VehicleState.DISCHARGE_RATE_IN_WH_PER_METER * distance;
}
```

Kuvatõmmis 3.2.2.8. Aku tühjenemise meetodid *VehicleService* klassis

Kui aku tase langeb alla 100 vatt-tunni, siis *DRIVING* olekus elektrisõiduki aku enam ei tühjene ja kuvatakse info alles jäänud aku laadimistaseme protsendist.

Et auto laadimist lõpetada või autot peatada, tuleb auto seada olekusse *IDLE*. Auto *IDLE* olekusse muutmise tagajärjel käivitatakse *stopAction* meetod, mis lülitab välja *ThreadPoolTaskScheduler*'i. Need protsessid on nähtavad kuvatõmmisel 3.2.2.9.

```java
.withExternal() ExternalTransitionConfigurer<VehicleState.Status, VehicleState.Status>
.source(Status.CHARGING).target(Status.IDLE)
.event(Status.IDLE)
.action(stopAction())
.and() StateMachineTransitionConfigurer<VehicleState.Status, VehicleState.Status>

.withExternal() ExternalTransitionConfigurer<VehicleState.Status, VehicleState.Status>
.source(Status.DRIVING).target(Status.IDLE)
.event(Status.IDLE)
.action(stopAction());
```

```java
@Bean
public Action<Status, Status> stopAction() {
    return context -> {
        if (!taskScheduler.getScheduledExecutor().isShutdown()) {
            taskScheduler.shutdown();
        }
    };
}
```

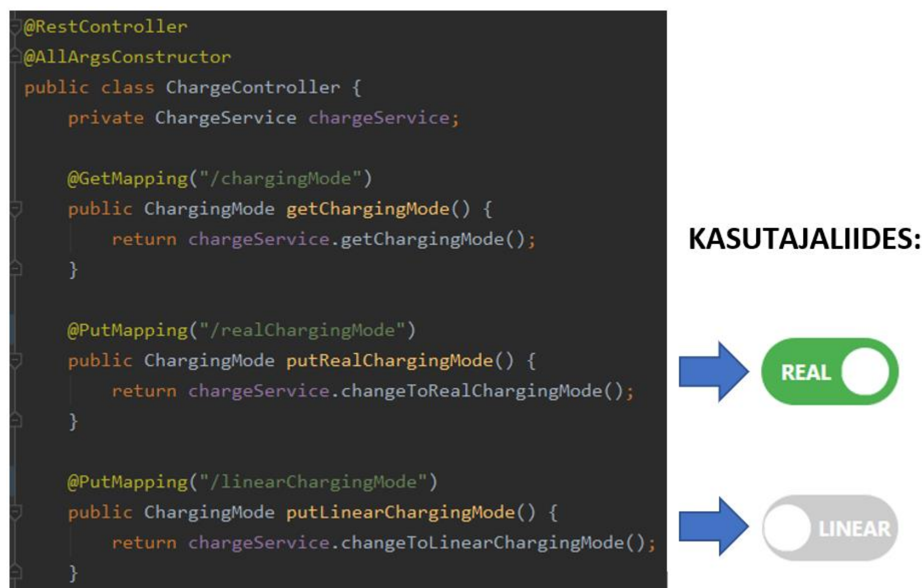
Kuvatõmmis 3.2.2.9. Olekumasina üleminekute, mis kasutavad *stopAction* meetodi, määratlemine ja *stopAction* meetod

3.2.3 Laadimisrežiimide vahel lülitamine

Selle kasutajaloo kriteeriumiteks olid kahe erineva laadimisrežiimi (reaalne ja lineaarne) ja nende vahel lülitamise võimaluse loomine.

Reaalne laadimisrežiim pakub reaalse laadimise- ja tühjenemiskõveraga laadimist ning aku tühjenemist. Laadimist ja tühjenemist mõjutavad mitmed erinevad tegurid, mida lineaarses režiimis arvesse ei pea võtma. Lineaarses režiimis on aku laadimine ja tühjenemine konstantselt samasugused. Lineaarne aku laadimine ja tühjenemine on loodud aku laadimise ja tühjenemise kasutajaloo raames, aga laadimine ja tühjenemine reaalsete kõveratega on loodud sõiduki aku laadimis- ja tühjenemiskõvera kasutajalugudes.

Selle kasutajaloo raames loodi eraldi teenus, mudel, *enum VehicleState* klassis, kontroller GET meetodiga */chargingMode* ja kahe PUT meetodiga */realChargingMode* ning */linearChargingMode*, mis võimaldavad vahetada režiimi kasutajaliideses tumblernupule vajutades. Kuvatõmmises 3.2.3.1 on näidatud kontrolleri lõpp-punktid ja nende tulemused kasutajaliideses.



Kuvatõmmis 3.2.3.1. *ChargeController* klassi lõpp-punktid ja tumblernupp kasutajaliides

Kasutajaliidese poolel loodi *ChargingModeComponent* klass, mis vastutab režiimi kuvamise ja muutuste tegemise eest. Kuvatõmmises 3.2.3.2 on näidatud tema peamised meetodid - *componentDidMount* ja *componentDidUpdate*, mis suhtlevad Axios teeki kasutades serveripoolse API-ga.

```
21      componentDidMount() {
22          axios.get( url: this.API_URL + "/chargingMode").then(
23              result => {
24                  var newChecked = result.data.chargingMode === "REAL";
25                  this.setState( state: {checked: newChecked});
26              }
27          );
28      }
29
30      componentDidUpdate() {
31          if (this.state.checked) {
32              axios.put( url: this.API_URL + "/realChargingMode").then(r => console.log(r));
33          } else {
34              axios.put( url: this.API_URL + "/linearChargingMode").then(r => console.log(r));
35          }
36      }
```

Kuvatõmmis 3.2.3.2. GET ja PUT päringud *ChargingModeComponent* klassis

Kuvatõmmises 3.2.3.3 on näidatud *render* meetod võtab sisend väärtuseid ja tagastab andmed kuvamiseks.

```
render() {
    return (
        <div
            className={"toggle-switch" + (this.props.Small ? " small-switch" : "")}
        >
            <input
                type="checkbox"
                name={this.props.Name}
                className="toggle-switch-checkbox"
                id={this.props.id}
                checked={this.state.checked}
                defaultChecked={this.props.defaultChecked}
                onChange={this.onChange}
                disabled={this.props.disabled}
            />
            {this.props.id ? (
                <label className="toggle-switch-label" htmlFor={this.props.id}>
                    <span
                        className={
                            this.props.disabled
                                ? "toggle-switch-inner toggle-switch-disabled"
                                : "toggle-switch-inner"
                        }
                        data-real={this.props.Text[0]}
                        data-linear={this.props.Text[1]}
                    />
                    <span
                        className={
                            this.props.disabled
                                ? "toggle-switch-switch toggle-switch-disabled"
                                : "toggle-switch-switch"
                        }
                    />
                </label>
            ) : null}
        </div>
    );
}
```

Kuvatõmmis 3.2.3.3. *render* meetod *ChargingModeComponent* klassis

Kuvatõmmises 3.2.3.4 on näidatud *ChargeService* avalikud meetodid *getChargingMode*, mis saab režiimi ning *changeToRealChargingMode* ja *changeToRealChargingMode*, mis salvestavad *VehicleState* repositooriumi õige režiimi *enum* väärtuse. Samuti on näidatud privaatne abimeetod *createChargingMode*, mille abil saab uusi režiime luua.

```
@RequiredArgsConstructor
@Service
@Slf4j
public class ChargeService {

    private final VehicleProperties properties;
    private final VehicleState vehicleState;
    private final ChargerProperties chargerProperties;

    @Value("#{'${vehicle.chargingCurveValuesKW}'.split(',')}")
    @Getter @Setter
    private double[] chargingCurveValuesKW;

    public ChargingMode getChargingMode() {
        return createChargingMode();
    }

    public ChargingMode changeToRealChargingMode() {
        vehicleState.setChargingMode(ChargingModeEnum.REAL);
        return createChargingMode();
    }

    public ChargingMode changeToLinearChargingMode() {
        vehicleState.setChargingMode(ChargingModeEnum.LINEAR);
        return createChargingMode();
    }

    private ChargingMode createChargingMode() {
        return new ChargingMode(vehicleState.getChargingMode().name());
    }
```

Kuvatõmmis 3.2.3.4. *ChargingModeService* klass

Kuvatõmmises 3.2.3.5 on näidatud *VehicleState* repositooriumis asuv *enum* režiimidega. Samuti on välja toodud, et rakenduse käivitamisel on vaikimisi režiim reaalne laadimisrežiim.

```
@Component
@Data
public class VehicleState implements Cloneable {
    private double stateOfChargeWh;
    private Status status;
    private GeoCoordinate previousLocation;
    private GeoCoordinate currentLocation;
    private GeoCoordinate targetLocation;
    private long drivingStartTime;
    private long lastSocIncrementingTime;
    private ChargingModeEnum chargingMode = DEFAULT_CHARGING_MODE;
    private double distanceDrivenOnRoute;
    private double vehicleMileage;

    public final static ChargingModeEnum DEFAULT_CHARGING_MODE = ChargingModeEnum.REAL;
    public final static double DISCHARGE_RATE_IN_WH_PER_METER = 0.165;
    public final static double SPEED_IN_KM_PER_H = 50;
    private StartLocationEnum startLocationEnum;

    public enum ChargingModeEnum {
        REAL,
        LINEAR
    }
```

Kuvatõmmis 3.2.3.5. *VehicleState* klass

3.3 Kolmas etapp

3.3.1 Sõiduki liikumine kasutajaliideses

Käesoleva kasutajalooga oli vaja kaardile loodud auto liikuma panna. See auto liigub mööda teekonda, mida kuvatakse kaardile kasutajaloos sõiduki liikumine serveripoolses osas loodud marsruudi abil(vt. kuvatõmmis 3.3.1.4).

Et saada auto kaardile liikuma, on kõigepealt vaja saata API POST päring serveripoolsele osale. Seda tehakse hetkel kasutades programmi Postman. Autole on serveripoolses osas loodud ka marsruut OpenRouteService abil, mille loomisest on juttu teise etapi kasutajaloos sõidukid liiguvad mööda reaalseid marsruute.

Kasutajaliidese poolel toimub jällegi kogu tegevus kahes klassis - *MapComponent* ja *LayersComponent*. Auto asukoht serveri poolsest projektist jõuab React aplikatsiooni läbi REST API GET päringu(vt. kuvatõmmis 3.3.1.1). Seejärel lisatakse saadud informatsioon massiivi *vehicleLocation* ning seda informatsiooni ehk auto asukohta päritakse iga kahe sekundi tagant. Seejärel saadab ta hetkeasukoha info edasi *LayersComponent* klassi.

```
setInterval( handler: () => axios.get( url: "http://localhost:8080/location").then(
    result => {
        this.setState( state: {isLoaded: true, vehicleLocation: result.data});
    },
    error => {
        this.setState( state: {isLoaded: true, error});
    }), timeout: 2000);
```

Kuvatõmmis 3.3.1.1. *MapComponent* klassi *componentDidMount* meetodis asuv GET päring asukoha saamiseks

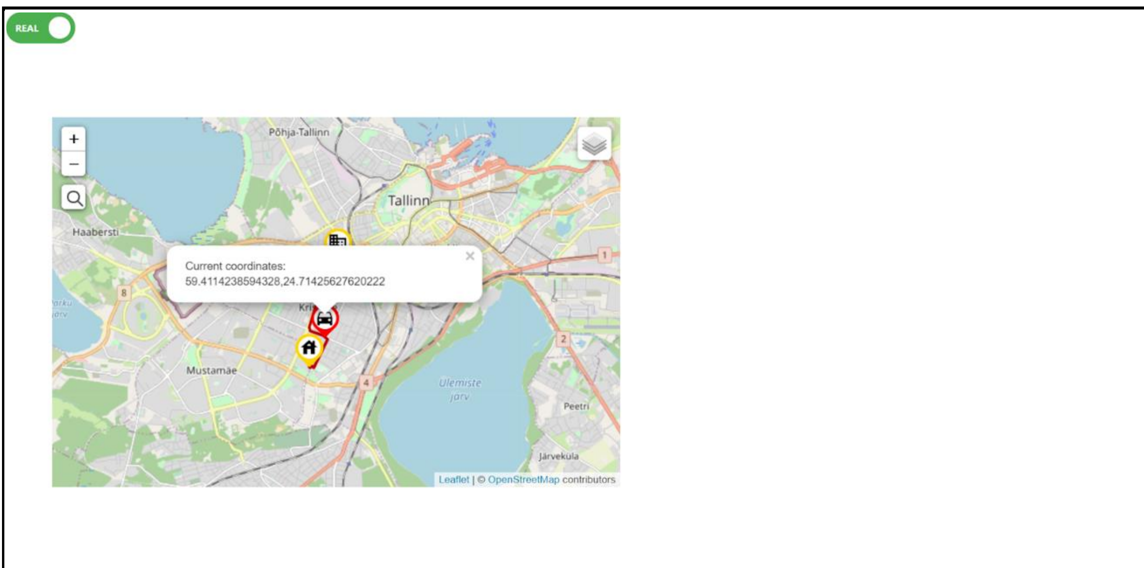
Kui auto sõidab, muutub sõiduki asukoht iga kahe sekundi tagant ehk kui edasi antud atribuudid on teised, kutsutakse välja *updateLeafletElement* meetod ja *Drift Marker*'i *slideTo* meetod ning kaardil on näha et auto marker liigub (vt. kuvatõmmised 3.3.1.2 ja 3.3.1.3).

```
this.vehicleLocation = new Drift_Marker(this.props.vehicleLocationData, this.props.icon)
    .bindPopup("Current coordinates: " + this.props.vehicleLocationData);
```

Kuvatõmmis 3.3.1.2. *Drift Marker* liikuva auto markeri loomiseks

```
updateLeafletElement(fromProps : Props , toProps : Props ) {
    if (fromProps.vehicleLocationData !== toProps.vehicleLocationData) {
        this.vehicleLocation.slideTo(toProps.vehicleLocationData, options: {duration: 2000});
        this.vehicleLocation.bindPopup("Current coordinates: " + toProps.vehicleLocationData);
    }
    if (toProps.icon !== fromProps.icon) {
        this.vehicleLocation.setIcon(toProps.icon);
    }
}
```

Kuvatõmmis 3.3.1.3. *updateLeafletElement* meetod *LayersComponent* klassis auto liikuma panemiseks



Kuvatõmmis 3.3.1.4. Auto teel kodust tööle koos koordinaatidega, mis saadakse API päringu kaudu

3.3.2 Sõiduki liikumine serveripoolses osas

Antud kasutajaloo käigus keskenduti nii serveri- kui ka kasutajapoolsele arendusele. Selle kasutajaloo kriteeriumiks oli, et autod kasutavad reaalseid marsruute, et kodust tööle sõita ja vastupidi.

Autod sõidavad mööda reaalseid marsruute kasutajaloo käigus loodud teekonna loogikale lisati juurde uusi meetodeid, mis talletavad elektriauto läbitud teekonna pikkuse(vt. kuvatõmmis 3.3.2.1). Marsruut ei ole algus- ja lõpp-punkti vahel sirge, vaid koosneb mitmetest väiksematest sirgetest, seetõttu tuli välja mõelda meetod, mis salvestaks teeviidad, millest auto on möödunud ning nende viitade põhjal arvutaks läbitud teepikkuse. *VehicleController* klassi loodi kaks uut meetodit *drive* ja *stop*, mis on ühenduses kasutajapoolse projektiga ning vastavalt muudavad staatust *DRIVING* ja *IDLE* vahel(vt. kuvatõmmis 3.3.2.5).

```
public double getDistanceTravelled(){
    int indexOfNextWayPoint = getIndexOfNextWayPointAfterCurrent(route.getTimesInEachWayPoint(), System.currentTimeMillis());
    List<Double> list;

    if (indexOfNextWayPoint == -1){
        list = route.getDistancesBetweenWayPoints();
    }
    else {
        list = route.getDistancesBetweenWayPoints().subList( i: 0,  i1: indexOfNextWayPoint - 1);
    }
    return list.stream().mapToDouble(Double::doubleValue).sum();
}
```

Kuvatõmmis3.3.2.1. *getDistanceTravelled* meetod klassis *RouteService*

Kaardil kasutati siiamaani teekonnana ainult marsruuti kodunt tööle, aga kuna sõiduki liikumine kasutajaliideses kasutajaloo käigus tuli panna auto kaardil sõitma, ilmnes vajalikkus ka uuendada teekonda sõltuvalt auto hetke asu- ja sihtkohast. *LayersComponent* klassi *updateLeafletElement* meetodisse sai lisatud kaardile kujutatava marsruudi muutmise loogika, mis sõltub auto sõidu alguskohast(vt. kuvatõmmis 3.3.2.3). Seda tehakse *setLatLngs* meetodiga, mis on *L.polyline* üks meetoditest. See saadakse omadusena *MapComponent* klassist, kust küsitakse serveripoolsest APIst GET päringuga auto staatust(vt. kuvatõmmis 3.3.2.2). Ehk kui auto jõuab tööle, muutub tema teekond töö-kodu teekonnaks ja kui auto jõuab koju, muutub ta tagasi kodu-töö teekonnaks, mille muutust kaardil ka uuendatakse.

```
setInterval( handler: () => axios.get( url: "http://localhost:8080/state").then(
    result => {
        this.setState( state: {isLoaded: true, vehicleState: result.data});
    },
    error => {
        this.setState( state: {
            isLoaded: true,
            error
        });
    }
), timeout: 2000);
```

Kuvatõmmis 3.3.2.2. *componentDidMount* meetodis asuv GET päring auto staatuse kohta

```
updateLeafletElement(fromProps :Props , toProps :Props ) {
    if (JSON.stringify(fromProps.vehicleLocationData) !== JSON.stringify(toProps.vehicleLocationData)) {
        this.vehicleLocation.slideTo(toProps.vehicleLocationData, options: {duration: 2000});
        this.vehicleLocation.bindPopup("Current coordinates: " + toProps.vehicleLocationData + "<dl>Distance driven: " + toProps.distanceDrivenData);
    }
    if (toProps.icon !== fromProps.icon) {
        this.vehicleLocation.setIcon(toProps.icon);
    }
    if (fromProps.startLocationEnum === "HOME") {
        this.polyline.setLatLngs([toProps.routeData]);
    }
    if (fromProps.startLocationEnum === "WORK") {
        this.polyline.setLatLngs([toProps.routeData]);
    }
}
```

Kuvatõmmis 3.3.2.3. uuendatud *updateLeafletElement* meetod *LayersComponent* klassis lisatud teekonna muutmise loogikaga

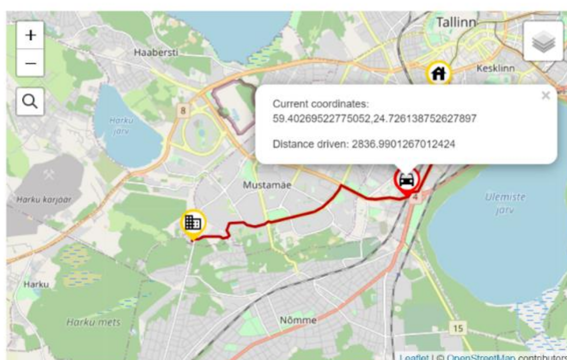
Lisaks tuli autole juurde selle kasutajaloo raames loodud info juba sõidetud teekonna pikkusest, mis uueneb sõites ja mida saab näha autole vajutades. See saadakse samuti auto staatusest. Senimaani kasutati Postmani, et teha õige API POST *request* auto sõitma panemiseks. Et seda natukene mugavamaks muuta, loodi *DriveComponent*, mis on nupp. Seda vajutades saab auto kas sõitma panna või peatada, olenevalt auto hetkelisest staatusest. Kui auto sõidab, kuvatakse nupule sõna *Stop*. Seda vajutades auto peatub, tema staatus muutub *IDLE*'ks ja nupule ilmub sõna *Drive*. Kui seda uuesti vajutada, hakkab auto uuesti sõitma. Nupuvajutus käivitab API POST päringu, mis saadetakse serveripoolsesse projekti, kus omakorda muudetakse auto staatust. *DriveComponent* klassi ja selle meetodeid on näha kuvatõmmisel 3.3.2.4.

```
export default function DriveComponent() {

    let checked = false;
    return <div>
        <Button variant="contained" onClick={() => {
            if(checked === true){
                changeCarStatus("stop");
                checked = false;
            }
            if(checked === false){
                changeCarStatus("drive");
                checked = true;
            }
        }}>Drive/Stop</Button>
    </div>;

    function changeCarStatus(status) {
        axios.post("http://localhost:8080/" + status).then(r => console.log(r));
    }
}
```

Kuvatõmmis 3.3.2.4. *DriveComponent* nupu loogika auto sõitma panemiseks ja POST API päring projekti serveripoolsele osale

```
@CrossOrigin(origins = "*", maxAge = 3600)
@PostMapping("/drive")
public void drive() { updateStatus(Status.DRIVING); }

@CrossOrigin(origins = "*", maxAge = 3600)
@PostMapping("/stop")
public void stop() { updateStatus(Status.IDLE); }
```

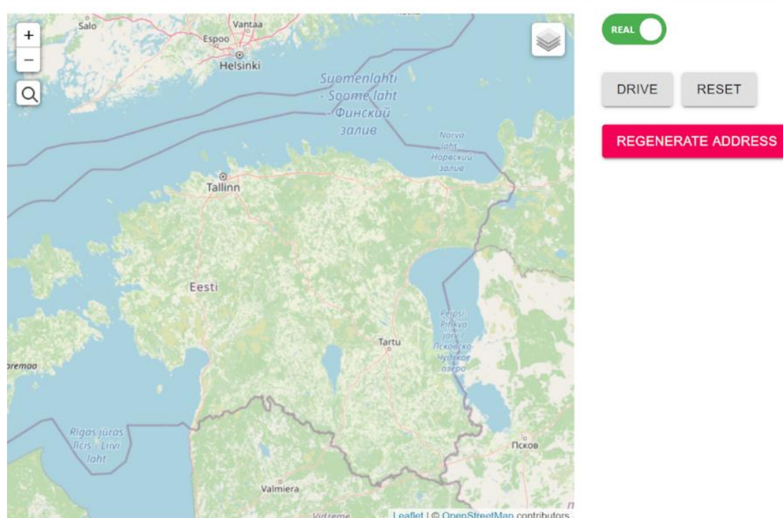
Kuvatõmmis 3.3.2.5. *Drive* ja *stop* meetodid REST API projekti *VehicleController* klassis auto staatuse uuendamiseks sõitmise ja peatumise vahel



Kuvatõmmis 3.3.2.6. auto teel kodust tööle, sõidetud on ümardatult 2900 m, kaardi kohal on näha ka *Drive/Stop* nupp

3.3.3 Kodu ja töö asukohtade vaheline kaugus

Kodu ja töö asukohtade vahelise kauguse kasutajaloos oli vaja luua funktsionaalsused, et auto asukoha parameetrid lähtestada ning genereerida uuesti auto kodu ja töö asukohad. Serveri poolsesse projekti loodi *VehicleServiceImpl* klassi auto lähtestamiseks meetod *resetVehicle*, mis määrab auto omadustele väärtuseks *null* või mõne teise algväärtuse. Et seda kasutajaliidese kaudu juhtida saaks on *VehicleController*'is meetod *resetVehicle*, mis võtab vastu POST päringuid, käivitab meetodi *resetVehicle* ja saadab tagasi auto staatuse. Lisaks loodi kasutaja poolsesse projekti juurde ka nupud, mis teevad POST API päringuid serveri poolsesse projekti, mis omakorda käivitavad serveris kas asukoha parameetrite lähtestamise või uuesti genereerimise asukoha ja kodu- ning töökoha asukohtadele. Hetkel toimib ainult üks funktsioon - auto asukoha parameetrite lähtestamine. Auto asukoha ja aadresside uuesti genereerimise loogikat kirjutatakse hetkel teises kasutajaloos ning see funktsionaalsus lisatakse hiljem projektile juurde. Lisaks sellele, lisati siin juurde ka mõningad visuaalsed muutused, mida on võimalik näha ka kuvatõmmisel 3.3.3.1, mis teevad kogu projekti silmale natukene ahvatlemaks ning kasutamise mugavamaks.



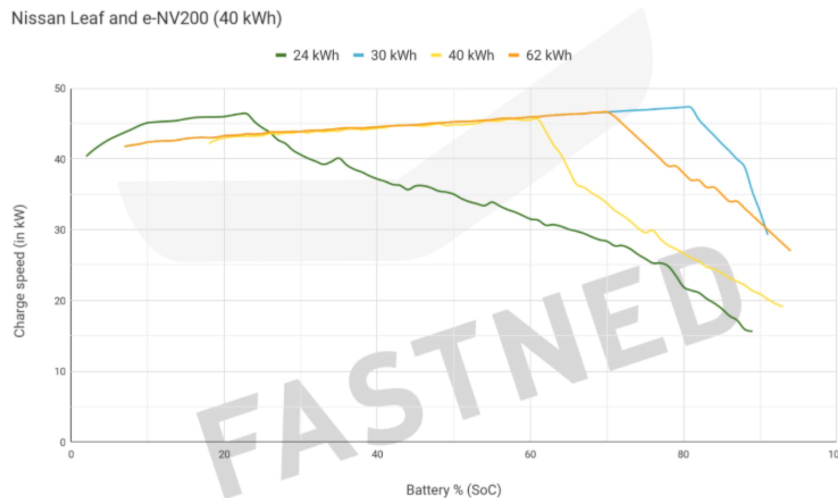
Kuvatõmmis 3.3.3.1 Kasutajaliidesele juurde lisatud nupud ning muutused.

3.3.4 Sõiduki aku laadimiskõver

Selle kasutajaloo eesmärk on reaalse laadimiskõveraga laadimise implementeerimine. See tähendab, et laadimine toimub erineva kiirusega, sõltuvalt aku täituvusest kui ka aku temperatuurist.

Kuvatõmmises 3.3.4.1 on näidatud erinevate akumahtudega laadimiskõverad Nissan Leaf mudeli jaoks [43].

- 24 kWh edition: fast charging until 25%, charging will gradually go slower after this
- 30 kWh edition: fast charging until 80%, charging will go slower after this
- 40 kWh edition: fast charging until 60%, charging will go slower after this
- 62 kWh edition: fast charging until 70%, charging will go slower after this



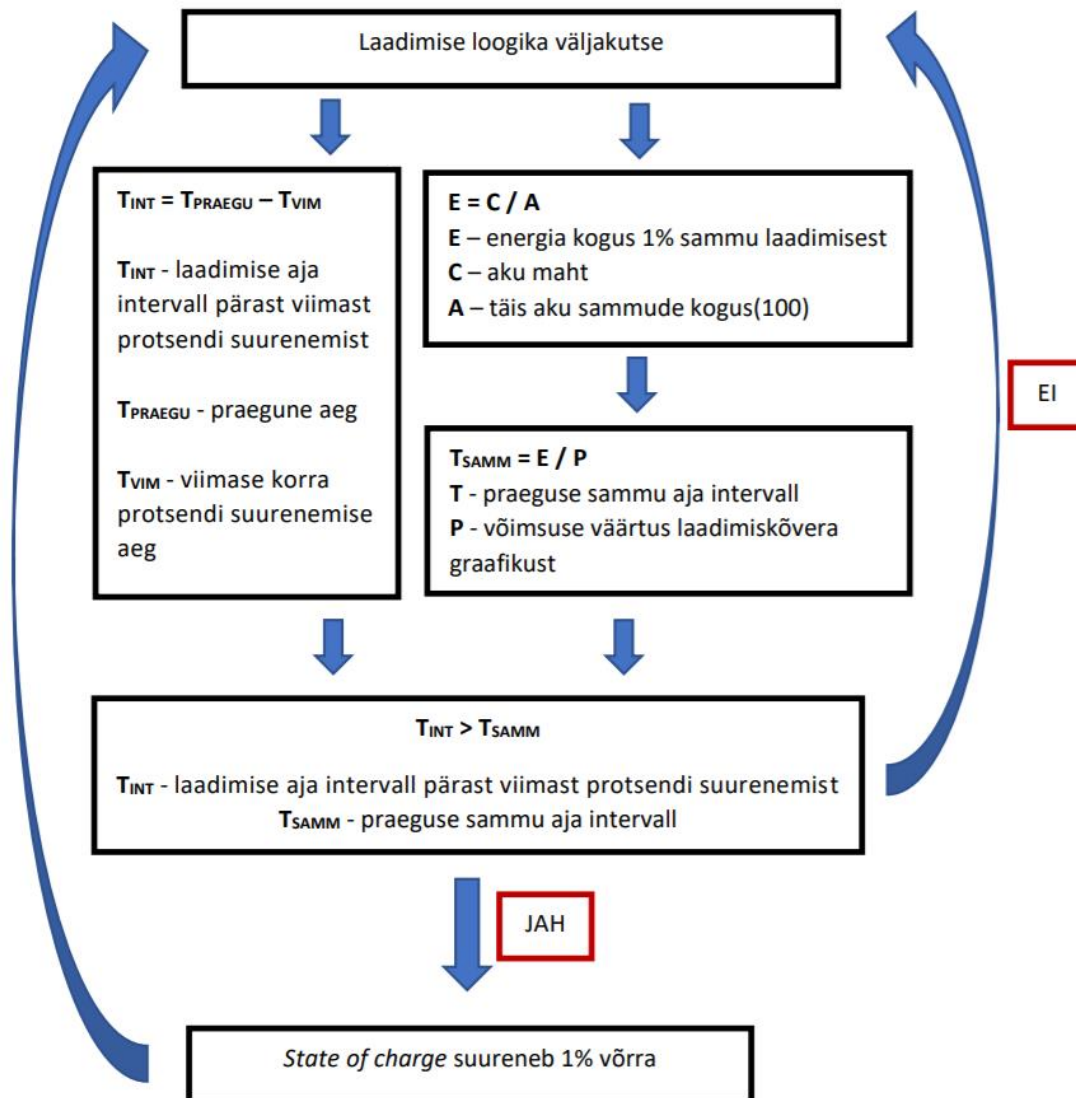
Kuvatõmmis 3.3.4.1. Laadimiskõverad

Kuvatõmmises 3.3.4.2 on näidatud *application.properties* fail, kus hoitakse ligikaudseid võimsuse väärtuseid ühe protsendilise sammuga aku täituvuse muutumisega laadimiskõvera graafikust.

```
application.properties
1  generated_vehicle_properties_path=backend/config/vehicle-generated.properties
2  route.url=https://api.openrouteservice.org/v2/directions/driving-car
3  route.authorization=5b3ce3597851110001cf6248b4316014682b4c7381024fa6f5d7e7a7
4  weather.appid=53473283ae89967c3ee476f05247e56d
5  weather.url=https://api.openweathermap.org/data/2.5/weather
6  allowedOrigins=http://localhost:3000
7  vehicle.chargingCurveValuesKW=40.0, 40.1, 40.2, 40.3, 40.4, 41.5, 41.6, 41.7,
8  vehicle.dischargingCurveValuesKW=0.50, 0.50, 0.50, 0.50, 0.50, 0.50, 0.62, 0.7
```

Kuvatõmmis 3.3.4.2. 40kWh aku mahuga sõiduki laadimisekõvera väärtused *application.properties* failis

Reaalses režiimis laadimine toimub „sammude kaupa“. Iga sammuga suurendatakse aku täituvust ühe protsendi ehk kindla energiahulga võrra (aku maht/100). Aga ühe protsendilise sammu laadimine võtab erineva aja sõltuvalt praegusest aku täituvusest. Laadimine toimub skeemis 3.3.4.3 näidatud algoritmi kasutades.



Skeem 3.3.4.3. Reaalse režiimis laadimise algoritm

Kuvatõmmises 3.3.4.4 on näidatud kõveraga laadimise loogika implementeerimine *ChargeService* klassis.

```java
        } else {
            if (shouldIncrementSoc(vehicleState.getLastSocIncrementingTime(), currentTime)) {
                realModeCharge(stateOfChargeWh, properties.getBatteryCapacitykWh());
            }
        }
    }

    private void realModeCharge(double stateOfChargeWh, double batteryCapacityKWh) {
        double newStateOfChargeWh = getNewStateOfChargeWh(stateOfChargeWh, batteryCapacityKWh);
        vehicleState.setStateOfChargeWh(newStateOfChargeWh);
        vehicleState.setLastSocIncrementingTime(System.currentTimeMillis());
        int stateOfCharge = StateOfCharge.calculateStateOfChargeValue(newStateOfChargeWh, properties);
        log.info(String.format("State of Charge: %s%% %s watt-hours", stateOfCharge, newStateOfChargeWh));
    }

    private boolean shouldIncrementSoc(long previousSocIncrementTime, long currentTime) {
        long timePassedAfterIncrementing = currentTime - previousSocIncrementTime;
        double currentStepTime = calculateTimeOfChargingOnePercent(
                vehicleState.getStateOfChargeWh(),
                properties.getBatteryCapacitykWh());

        return currentStepTime < timePassedAfterIncrementing;
    }

    private long calculateTimeOfChargingOnePercent(double stateOfChargeWh, double batteryCapacityKWh) {
        double amountOfEnergyToChargeOnePercentKWh = batteryCapacityKWh / 100;
        int socPercent = Math.min(StateOfCharge.calculateStateOfChargeValue(stateOfChargeWh, properties), 99);
        double chargingEfficiency = getChargingEfficiency(properties.getChargeEfficiencyPercentage());

        double currentStepChargingCurveValueKWh = Math.min(
                chargingCurveValuesKW[socPercent],
                chargerProperties.getChargingPowerKw());
        double currentStepPowerKWh = currentStepChargingCurveValueKWh * chargingEfficiency;

        return (long) ((amountOfEnergyToChargeOnePercentKWh / currentStepPowerKWh) * 3600 * 1000);
    }

    public long calculateLastIncrementingTime(long currentTimeMillis) {
        double stateOfChargeWh = vehicleState.getStateOfChargeWh();
        double batteryCapacityKWh = properties.getBatteryCapacitykWh();
        double socPercent = stateOfChargeWh * 100 / batteryCapacityKWh / 1000;
        double modulo = socPercent % 1;
        long timeToChargeCurrentPercent = calculateTimeOfChargingOnePercent(stateOfChargeWh, batteryCapacityKWh);
        long timePassedAfterIncrementing = (long) (timeToChargeCurrentPercent * modulo);

        return currentTimeMillis - timePassedAfterIncrementing;
    }
```

Kuvatõmmis 3.3.4.4. *ChargeService* klass

Kui sõiduk on liikumise ehk aku tühjenemise protsessis, siis kuvatakse info konsoolis iga aku täituvuse langusega. Kuvatõmmises 3.3.4.5 on näidatud, et aku laadimine toimub erineva kiirusega sõltuvalt tema tasemest.

```
2020-04-09 06:14:37.945  INFO 13400 --- [ivityScheduler2] enefit.vxg.services.ChargeService        : State of Charge: 1% 380.0 watt-hours   Time left until SoC incrementing: 37.905 sec
2020-04-09 06:15:15.985  INFO 13400 --- [ivityScheduler2] enefit.vxg.services.ChargeService        : State of Charge: 2% 760.0 watt-hours   Time left until SoC incrementing: 37.81 sec
2020-04-09 06:15:54.021  INFO 13400 --- [ivityScheduler2] enefit.vxg.services.ChargeService        : State of Charge: 3% 1140.0 watt-hours   Time left until SoC incrementing  37.717 sec
2020-04-09 06:16:32.060  INFO 13400 --- [ivityScheduler2] enefit.vxg.services.ChargeService        : State of Charge: 4% 1520.0 watt-hours   Time left until SoC incrementing  37.623 sec
2020-04-09 06:17:10.096  INFO 13400 --- [ivityScheduler2] enefit.vxg.services.ChargeService        : State of Charge: 5% 1900.0 watt-hours   Time left until SoC incrementing  36.626 sec
```

```
2020-04-09 06:32:49.914  INFO 13400 --- [ivityScheduler2] enefit.vxg.services.ChargeService        : State of Charge: 31% 11780.0 watt-hours   Time left until SoC incrementing: 34.545 sec
2020-04-09 06:33:24.944  INFO 13400 --- [ivityScheduler2] enefit.vxg.services.ChargeService        : State of Charge: 32% 12160.0 watt-hours   Time left until SoC incrementing: 34.467 sec
2020-04-09 06:33:59.974  INFO 13400 --- [ivityScheduler2] enefit.vxg.services.ChargeService        : State of Charge: 33% 12540.0 watt-hours   Time left until SoC incrementing: 34.467 sec
2020-04-09 06:34:35.009  INFO 13400 --- [ivityScheduler2] enefit.vxg.services.ChargeService        : State of Charge: 34% 12920.0 watt-hours   Time left until SoC incrementing: 34.389 sec
2020-04-09 06:35:10.041  INFO 13400 --- [ivityScheduler2] enefit.vxg.services.ChargeService        : State of Charge: 35% 13300.0 watt-hours   Time left until SoC incrementing: 34.389 sec
```

```
2020-04-09 06:50:02.863  INFO 13400 --- [ivityScheduler2] enefit.vxg.services.ChargeService        : State of Charge: 61% 23180.0 watt-hours   Time left until SoC incrementing: 33.853 sec
2020-04-09 06:50:36.895  INFO 13400 --- [ivityScheduler2] enefit.vxg.services.ChargeService        : State of Charge: 62% 23560.0 watt-hours   Time left until SoC incrementing: 34.004 sec
2020-04-09 06:51:10.927  INFO 13400 --- [ivityScheduler2] enefit.vxg.services.ChargeService        : State of Charge: 63% 23940.0 watt-hours   Time left until SoC incrementing: 34.157 sec
2020-04-09 06:51:45.954  INFO 13400 --- [ivityScheduler2] enefit.vxg.services.ChargeService        : State of Charge: 64% 24320.0 watt-hours   Time left until SoC incrementing: 34.311 sec
2020-04-09 06:52:20.983  INFO 13400 --- [ivityScheduler2] enefit.vxg.services.ChargeService        : State of Charge: 65% 24700.0 watt-hours   Time left until SoC incrementing: 38.0 sec
```

```
2020-04-09 07:16:14.296  INFO 13400 --- [ivityScheduler2] enefit.vxg.services.ChargeService        : State of Charge: 91% 34580.0 watt-hours   Time left until SoC incrementing: 76.0 sec
2020-04-09 07:17:30.366  INFO 13400 --- [ivityScheduler2] enefit.vxg.services.ChargeService        : State of Charge: 92% 34960.0 watt-hours   Time left until SoC incrementing: 77.948 sec
2020-04-09 07:18:48.439  INFO 13400 --- [ivityScheduler2] enefit.vxg.services.ChargeService        : State of Charge: 93% 35340.0 watt-hours   Time left until SoC incrementing: 80.423 sec
2020-04-09 07:20:09.510  INFO 13400 --- [ivityScheduler2] enefit.vxg.services.ChargeService        : State of Charge: 94% 35720.0 watt-hours   Time left until SoC incrementing: 83.06 sec
2020-04-09 07:21:32.585  INFO 13400 --- [ivityScheduler2] enefit.vxg.services.ChargeService        : State of Charge: 95% 36100.0 watt-hours   Time left until SoC incrementing: 86.857 sec
```

Kuvatõmmis 3.3.4.5. *Log.info* reaalne režiim

3.4 Neljas etapp

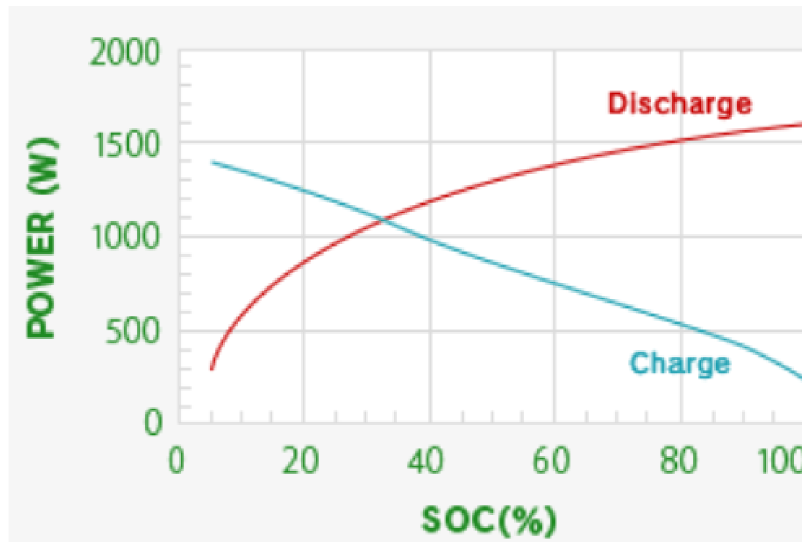
3.4.1 Sõiduki aku tühjenemiskõver

Siin kasutajaloos oli vaja simuleerida aku tegelikku tühjenemist kasutades reaalset tühjenemise kõverat. See tähendab, et kui sõiduk liigub pidevalt sama kiirusega, siis aku tühjenemine ei toimu konstantselt sama kiirusega, vaid sõltuvalt aku täituvusest ehk aku temperatuurist.

Selle loo loogika on umbes samasugune nagu aku laadimiskõvera kasutajaloos, kuid laadimisvõimsuse asemel arvutatakse teepikkus, mille sõiduk oleks kindla aja jooksul läbinud ning kui palju energiat oleks selleks kulunud. Nende numbrite põhjal itereeritakse üle praeguse aku oleku tarbitud energia võrra kuni aku saab tühjaks.

Tegelik elektritarbimine mõjub tühjenemise protsessile sõltuvalt praegusest aku täituvusest. Mida vähem energiat on akusse jäänud, seda aeglasemalt aku tühjeneb.

Kuvatõmmises 3.4.1.1 on näidatud tühjenemiskõver Nissan Leaf 24 kWh akuga mudeli jaoks [44].



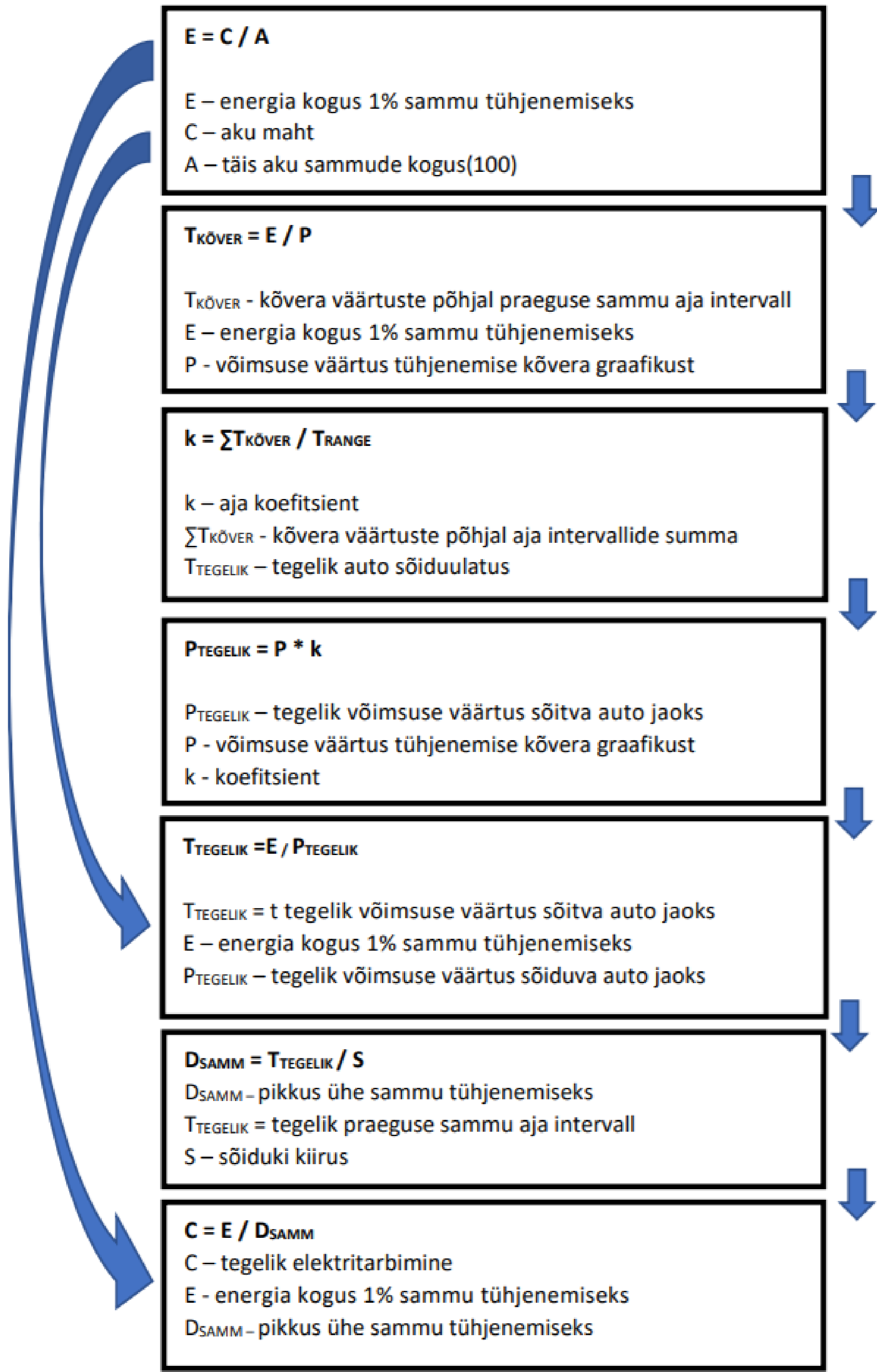
Kuvatõmmis 3.4.1.1. tühjenemiskõver Nissan Leaf 24 kWh aku mahuga

Aku laadimise ja tühjenemise kasutajaloo raames implementeeriti lineaarne aku tühjenemine. Reaalne laadimine toimub sama algoritmiga nagu lineaarne laadimine. Aku täituvusest lahutatakse iga *discharge* meetodi väljakutse korral tarbitud energia, väljaarvatud, et reaalse laadimise korral kasutatakse tegelikku elektritarbimist, mis kalkuleeritakse iga sammu (1%) jaoks, aga lineaarses laadimises võetakse elektritarbimiseks keskmine tarbimise väärtus. Kuvatõmmises 3.4.1.2 on näidatud koodis implementeeritud reaalse ja lineaarse laadimise erinevus.

```java
@RequiredArgsConstructor
@Service
@Slf4j
public class DischargeService {
    private final VehicleState vehicleState;
    private final VehicleProperties properties;
    private final VehicleService vehicleService;
    private final AddressGenerator addressGenerator;

    @Value("#{'${vehicle.dischargingCurveValuesKW}'.split(',')}")
    @Getter
    @Setter
    private double[] dischargingCurveValuesKW;

    @Setter
    private VehicleState previousVehicleState;

    public void discharge() {
        double stateOfChargeWh = vehicleState.getStateOfChargeWh();

        double consumption;
        if (vehicleState.getChargingMode() == VehicleState.ChargingModeEnum.LINEAR) {
            consumption = VehicleState.DISCHARGE_RATE_IN_WH_PER_METER;
        } else {
            consumption = getRealConsumption(stateOfChargeWh, properties.getBatteryCapacitykWh());
        }

        double distance = addressGenerator.calculateDistanceInMetersBetween(
                previousVehicleState.getCurrentLocation(),
                vehicleState.getCurrentLocation());
        double dischargedForDistance = distance * consumption;

        boolean isBatteryEmpty = stateOfChargeWh - dischargedForDistance < 100.0;
        if (isBatteryEmpty) {
            stopDriving();
            return;
        }

        double newStateOfChargeWh = stateOfChargeWh - dischargedForDistance;
        vehicleState.setStateOfChargeWh(newStateOfChargeWh);

        log.info(String.format("%s watt-hours remaining, SoC is %s%%",
                newStateOfChargeWh,
                StateOfCharge.calculateStateOfChargeValue(newStateOfChargeWh, properties)));

        previousVehicleState = (VehicleState) vehicleState.clone();
    }
```

Kuvatõmmis 3.4.1.2. aku tühjenemise loogika *DischargeService* klassis

Tegelik elektritarbimine kalkuleeritakse skeemis 3.4.1.3 näidatud algoritmi kasutades.



Skeem 3.4.1.3. Tegeliku elektritarbimise kalkuleerimise algoritm

Koodis implementeeritakse tühjenemise algoritmi *DischargeService* klassis kahe abimeetodi abil - *getRealConsumption*, kus kalkuleeritakse tegelik elektritarbimine ning *getCurveBasedTimeOfDischargingFullBattery*, kus *for* tsükkliga arvutakse kõvera väärtuste põhjal aja intervallide summa(vt kuvatõmmis 3.4.1.4).

```
private double getRealConsumption(double stateOfChargeWh, double batteryCapacityKWh) {
    double stateOfCharge = stateOfChargeWh * 100 / batteryCapacityKWh / 1000;
    double realDistanceOfDischargingFullBatteryKm = batteryCapacityKWh / VehicleState.DISCHARGE_RATE_IN_WH_PER_METER;
    double realTimeOfDischargingFullBatteryH = realDistanceOfDischargingFullBatteryKm / VehicleState.SPEED_IN_KM_PER_H;

    double coefficient = getCurveBasedTimeOfDischargingFullBattery(dischargingCurveValuesKW) / realTimeOfDischargingFullBatteryH;
    double currentStepDischargingCurveValue = dischargingCurveValuesKW[Math.min((int) stateOfCharge, 99)];

    return (currentStepDischargingCurveValue * coefficient / VehicleState.SPEED_IN_KM_PER_H) / 1000;
}

private void stopDriving() {
    Log.info(String.format("low battery: %s watt-hours left", vehicleState.getStateOfChargeWh()));
    vehicleService.updateStatus(VehicleState.Status.IDLE);
}

private double getCurveBasedTimeOfDischargingFullBattery(double[] curve) {
    double sum = 0;
    for (double value : curve) {
        double time = (properties.getBatteryCapacitykWh() * 10) / value;
        sum += time;
    }
    return sum;
}
```

Kuvatõmmis 3.4.1.4. tegeliku elektritarbimise arvutamine *DischargeService* klassis

Kui sõiduk on liikumise ehk aku tühjenemise protsessis, kuvatakse infot konsoolis iga sekundilise intervalliga. Kuna sõiduk liigub konstantsel kiirusel, on tühjenemise loogika väljakutsete vahel sama pikad intervallid.

Kuvatõmmises 3.4.1.5 on näidatud, et sõltuvalt aku olekust kulutatakse erinev kogus energiat samale teepikkusele.

```
enefit.vxg.services.DischargeService     : 37634.63842368411 watt-hours remaining, SoC is 99%,  battery discharged on 3.8403940789740822 WH
enefit.vxg.services.DischargeService     : 37630.79802960514 watt-hours remaining, SoC is 99%,  battery discharged on 3.8403940789740822 WH
enefit.vxg.services.DischargeService     : 37626.95763552617 watt-hours remaining, SoC is 99%,  battery discharged on 3.8403940789740822 WH
enefit.vxg.services.DischargeService     : 19354.789627655395 watt-hours remaining, SoC is 50%,  battery discharged on 3.1447755288202863 WH
enefit.vxg.services.DischargeService     : 19351.644852126574 watt-hours remaining, SoC is 50%,  battery discharged on 3.1447755288202863 WH
enefit.vxg.services.DischargeService     : 19348.500076597753 watt-hours remaining, SoC is 50%,  battery discharged on 3.1447755288202863 WH
enefit.vxg.services.DischargeService     : 4177.130573480616 watt-hours remaining, SoC is 10%,  battery discharged on 1.420221206564003 WH
enefit.vxg.services.DischargeService     : 4175.710352274052 watt-hours remaining, SoC is 10%,  battery discharged on 1.420221206564003 WH
enefit.vxg.services.DischargeService     : 4174.290131067488 watt-hours remaining, SoC is 10%,  battery discharged on 1.420221206564003 WH
```

Kuvatõmmis 3.4.1.5. *Log.info* reaalne režiim

Kuvatõmmises 3.4.1.6 on näidatud, et lineaarses režiimis samadel tingimustel tagastatakse sama väärtust sõltumatult aku täituvusest.

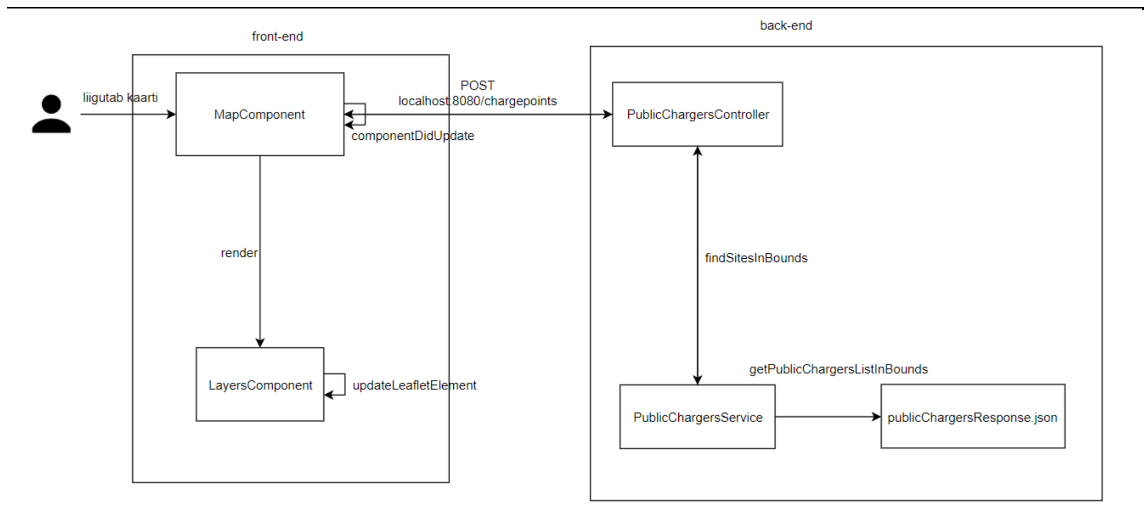
```
enefit.vxg.services.DischargeService     : 19378.541666666602 watt-hours remaining, SoC is 50%,  battery discharged on 2.2916666666666665 WH
enefit.vxg.services.DischargeService     : 19376.249999999935 watt-hours remaining, SoC is 50%,  battery discharged on 2.2916666666666665 WH
enefit.vxg.services.DischargeService     : 19373.958333333267 watt-hours remaining, SoC is 50%,  battery discharged on 2.2916666666666665 WH
enefit.vxg.services.DischargeService     : 4115.771370555898 watt-hours remaining, SoC is 10%,  battery discharged on 2.2916666666666665 WH
enefit.vxg.services.DischargeService     : 4113.479703889231 watt-hours remaining, SoC is 10%,  battery discharged on 2.2916666666666665 WH
enefit.vxg.services.DischargeService     : 4111.188037222564 watt-hours remaining, SoC is 10%,  battery discharged on 2.2916666666666665 WH
```

Kuvatõmmis 3.4.1.6. *Log.info* lineaarne režiim

3.4.2 Avalikud laadimispunktid kaardil

Antud kasutajaloo nõue oli näidata kaardil elektriautode laadimispunkte(vt. kuvatõmmis 3.4.2.6). Laadimispunkte kuvatakse ainult kaardi nähtavasse osasse ning nende leidmiseks kasutatakse kaardi nähtava osa kõige kirde- ja edelapoolsemaid koordinaatpunkte. Kui kaarti liigutada, muutuvad koordinaatpunktid. Koordinaatide kätte saamiseks kasutatakse Leafleti meetodit *getBounds*, mis annab arendajatele vajalikud koordinaadid. *GetBounds* meetodit kasutatakse kasutajaliidese projekti *MapComponent* klassi *componentDidUpdate* meetodis, mis kutsutakse iga kord välja kui kaardi olek muutub. Selles samas meetodis tehakse kaardi oleku muutumisel API POST päring serveripoolses projektis asuvasse *PublicChargersController*'ile, mis sai ka selle kasutajaloo käigus loodud(vt. kuvatõmmis 3.4.2.2). POST päringuga küsitakse kõiki laadimispunkte antud koordinaatide vahemikus. *PublicChargersController* suhtleb *PublicChargersService* klassiga, mis otsib ülesse nende punktide piires olevad elektriautode laadimispunktid(vt. kuvatõmmis 3.4.2.3) ning need tagastatakse API päringu vastusena *MapComponent*'ile(vt. kuvatõmmis 3.4.2.4). Kõik Eestimaa piires olevad elektriautode laadimispunktid ja nende info on salvestatud ajutiselt projekti JSON tüüpi failina [45]. Ka *PublicChargersController* ja *-Service* on loodud ajutiselt, kuni luuakse Enefiti poolt loogika, mis võimaldaks suhtlemist Enefit Volt API-ga.

MapComponent klass edastadab laadimispunktide info atribuutidena *LayersComponent*'i. *LayersComponent* kasutab laadimispunktide koordinaate, et luua markereid kaardile kuvamiseks. Samuti luuakse infokiht *Chargers*, mille peale klikkides saab kuvada kaardil laadimispunkte. Kaarti liigutades või sisse-välja suumides kutsutakse välja *updateLeafletElement* meetod, mis vajadusel lisab *Chargers* infokihti laadimispunkti markereid juurde või eemaldab neid(vt. kuvatõmmis 3.4.2.5). Järgnevalt on näha ka klassidevaheline suhtlusdiagramm.



Suhtlusdiagramm 3.4.2.1. klasside vahel

```
@RestController
public class PublicChargersController {

    private PublicChargersService publicChargersService = new PublicChargersService();

    @PostMapping("/chargePoints")
    public List<PublicCharger> findSitesInBounds(@RequestBody PublicChargerFilterBody body) throws IOException {

        return publicChargersService.getPublicChargerListInBounds(body.filterByIsManaged,
                new GeoCoordinate(body.filterByBounds.getNorthEastLat(), body.filterByBounds.getNorthEastLng()),
                new GeoCoordinate(body.filterByBounds.getSouthWestLat(), body.filterByBounds.getSouthWestLng()));
    }
}
```

Kuvatõmmis 3.4.2.2. *PublicChargersController* klass API POST päringu jaoks

```java
public class PublicChargersService {
    ObjectMapper mapper = new ObjectMapper();
    ClassLoader classLoader = getClass().getClassLoader();
    File jsonFile = new File(classLoader.getResource( name: "publicChargersResponse.json").getFile());

    public PublicChargerResponse getPublicChargerResponse() throws IOException {
        return mapper.readValue(jsonFile, PublicChargerResponse.class);
    }

    public List<PublicCharger> getPublicChargerList() throws IOException {
        var response = getPublicChargerResponse();
        if (response.getErrors().isEmpty() && response.getSuccess()) {
            List<PublicCharger> publicChargers = mapper.convertValue(
                    response.getData().get(1), new TypeReference<List<PublicCharger>>() {}
            );
            return publicChargers;
        }
        return null;
    }

    public List<PublicCharger> getPublicChargerListInBounds(boolean filterByIsManaged, GeoCoordinate northEast, GeoCoordinate southWest) throws IOException {
        List<PublicCharger> listOfAllPublicChargers = getPublicChargerList();
        List<PublicCharger> listOfPublicChargersInBounds = new ArrayList<>();
        for (var publicCharger : listOfAllPublicChargers) {
            if (publicCharger.latitude > southWest.getLatitude() && publicCharger.latitude < northEast.getLatitude()
                    && publicCharger.longitude > southWest.getLongitude() && publicCharger.longitude < northEast.getLongitude()) {
                if (publicCharger.managed == filterByIsManaged) {
                    listOfPublicChargersInBounds.add(publicCharger);
                }
            }
        }
        return listOfPublicChargersInBounds;
    }
```

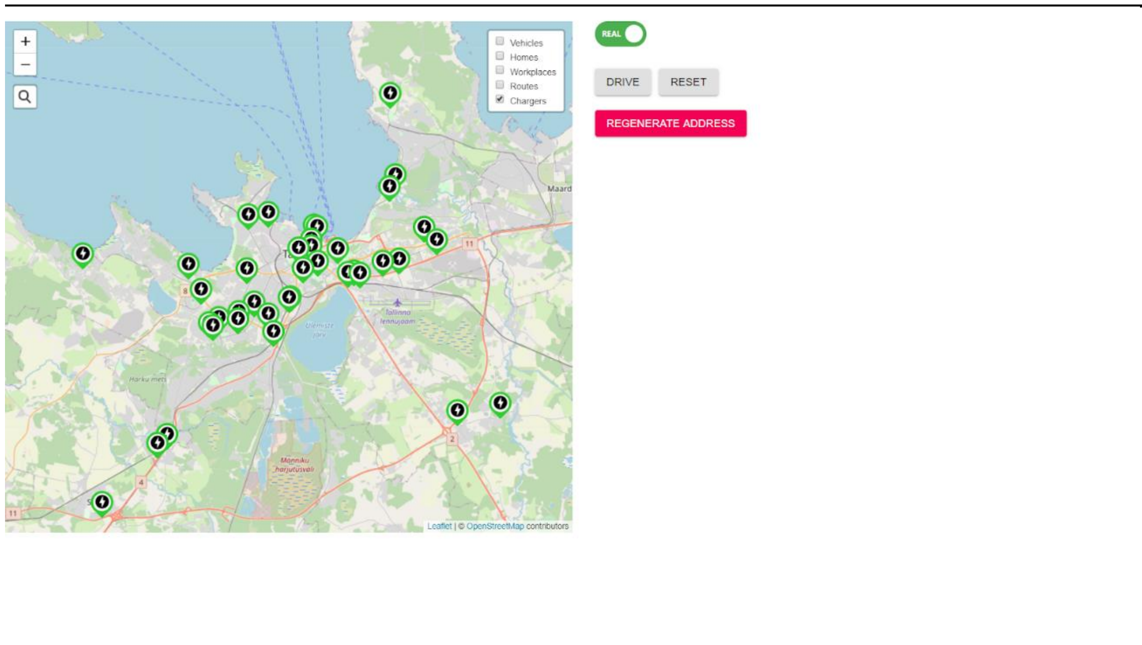
Kuvatõmmis 3.4.2.3. *PublicChargersService* klass laadimispunktide otsimiseks json tüüpi failist publicChargersResponse

```javascript
componentDidUpdate(prevProps: Readonly<P>, prevState: Readonly<S>, snapshot: SS): void {
    if (this.mapRef.current != null) {
        if (JSON.stringify(prevState.mapBounds) !== JSON.stringify(this.mapRef.current.leafletElement.getBounds())) {
            this.setState( state: {mapBounds: this.mapRef.current.leafletElement.getBounds()});
            axios({
                method: "post",
                url: this.API_URL + "/chargePoints",
                data: {
                    filterByIsManaged: true,
                    filterByBounds: {
                        northEastLat: this.mapRef.current.leafletElement.getBounds()._northEast.lat,
                        northEastLng: this.mapRef.current.leafletElement.getBounds()._northEast.lng,
                        southWestLat: this.mapRef.current.leafletElement.getBounds()._southWest.lat,
                        southWestLng: this.mapRef.current.leafletElement.getBounds()._southWest.lng
                    }
                }
            }).then(result => {
                    this.setState( state: {publicChargers: result.data});
                },
                error => {
                    this.setState( state: {
                        error
                    })
                });
        }
    }
```

Kuvatõmmis 3.4.2.4. *componentDidUpdate* meetod *MapComponent* klassis koos API POST päringuga laadimispunktide saamiseks serveri poolsest projektist

```javascript
updateLeafletElement(fromProps : Props , toProps : Props ) {
    if (JSON.stringify(fromProps.vehicleLocationData) !== JSON.stringify(toProps.vehicleLocationData)) {
        this.vehicleLocation.slideTo(toProps.vehicleLocationData,  options: {duration: 2000});
        this.vehicleLocation.bindPopup("Current coordinates: " + toProps.vehicleLocationData + "<dl>Distance driven: " + toProps.distanceDrivenData);
    }
    if (toProps.icon !== fromProps.icon) {
        this.vehicleLocation.setIcon(toProps.icon);
    }
    if (fromProps.startLocationEnum === "HOME") {
        this.polyline.setLatLngs([toProps.routeData]);
    }
    if (fromProps.startLocationEnum === "WORK") {
        this.polyline.setLatLngs([toProps.routeData]);
    }
    if (JSON.stringify(fromProps.publicChargers) !== JSON.stringify(toProps.publicChargers)) {
        this.chargers.clearLayers();
        this.chargerMarkers = [];
        for (let i = 0; i < toProps.publicChargers.length; i++) {
            this.chargerMarkers.push(L.marker([toProps.publicChargers[i].latitude, toProps.publicChargers[i].longitude],
                {icon: getIcon( iconName: 'charger')}).bindPopup(toProps.publicChargers[i].dn));
        }
        const newChargers = L.layerGroup(this.chargerMarkers);
        newChargers.addTo(this.chargers);
    }
}
```


Kuvatõmmis 3.4.2.5. Uuendatud *updateLeafletElement* meetod *LayersComponent* klassis, lisatud laadimispunktide uuendamine kaardil



Kuvatõmmis 3.4.2.6. Kõik elektriautode laadimispunktid kaardi nähtaval osal

3.4.3 Dockeri implementatsioon projektis

Käesolev punkt koosneb mitmest Dockeri implementatsioonist nii serveri- kui ka kasutajapoolses osas. Nendes kasutajalugudes tuli autoritel kirjutada *Dockerfile*, mille ülesanne on luua *Docker Image* konteiner. *Dockerfile* on teksti dokument, mis hõlmab endas kõiki käske, mida kasutaja peab kirjutama, et luua *Docker Image*. *Dockerfile* kirjutamiseks kasutati mitmeastmelist *Dockerfile* ülesehitust.

Serveripoolne osa:

Esimene osa *Dockerfile*'st laeb alla Gradle lähtefailid ning hakkab kasutama seda projekti ehitajana(ingl *builder*). Seejärel kopeeritakse allalaetud failid kausta, mis luuakse Dockeri poolt loodud virtuaalarvutisse. Järgmine käsurida paneb töökeskkonnaks selle sama kausta ning ehitab seal projekti üles. Seejärel tuleb sisse Java ning selle failid laetakse alla. Antud projekti puhul on selleks Java JDK 13 [46]. Peale seda luuakse kasutajagrupp ning kasutaja ja määratakse see samune kasutaja aktiivseks. Järgmisena hakatakse kasutama uut töökeskkonda nimega *app* ning sinna kopeeritakse ehitatud projekti *jar* tüüpi fail ning konfiguratsioonifail *vehicle-generated.properties* ja viimane neist määratakse keskkonna muutujaks samas konfiguratsioonifailis. Viimane rida loob Docker konteineri, mida hakatakse kasutama käivitatava failina.

Kui *Dockerfile* abil on loodud *Docker Image*, siis see tuleb üles laadida Github Packages registrisse. Tulevikus saab seda sealt ka alla laadida ning käivitada. *Dockerfile*'i ülesehitus on näha kuvatõmmisel 3.4.3.1.

```
FROM gradle:6.3-jdk13 AS build
COPY --chown=gradle:gradle . /home/gradle/src
WORKDIR /home/gradle/src
RUN gradle build

FROM openjdk:13 AS vxg-backend

RUN groupadd -r app && useradd --no-log-init -r -g app app
USER app

WORKDIR /app
COPY --from=build --chown=app:app /home/gradle/src/build/libs/backend-0.0.1-SNAPSHOT.jar /app/vxg-virtual-vehicle-backend-0.0.1-SNAPSHOT.jar
COPY --from=build --chown=app:app /home/gradle/src/config/vehicle-generated.properties /app/config/
ENV generated_vehicle_properties_location file:///app/config
ENTRYPOINT ["java","-jar","vxg-virtual-vehicle-backend-0.0.1-SNAPSHOT.jar"]
```

Kuvatõmmis 3.4.3.1. Serveri poolse *Dockerfile* ülesehitus *Docker Image* loomiseks

Kasutajaliidese poolne osa:

Dockerfile alustab Node.js keskkonna alla laadimisega, mis hakkab projekti koodi käivitama, seejärel luuakse töökeskkond ning kopeeritakse kasutajapoolse projekti *package.json* ja *package-lock.json* failid loodud töökeskkonda. Peale seda tehakse *npm install* ja *npm audit fix* käsud, mis tõmbavad alla kõik *package.json* failis kirjas olevad vajalikud sõltuvused ja vajaduse korral uuendavad või parandavad neid. Needki kopeeritakse töökeskkonda. Seejärel jooksutatakse läbi kõik testid. Peale seda määratakse vajalik keskkonna muutuja ning käivitatakse projekti ülesehituse protsess. Selleks on vaja nginx-i veebiserverit, mis tõmmatakse alla. *Expose* 80 tähendab, et aplikatsiooni jooksmise jaoks avatakse port 80.

Et *Docker Image* registrisse üles laadida, tuleb konsoolis vastavaid käske kasutada. Kõigepealt tuleb luua *Docker Image*, kasutades *Dockerfile*'i. Seejärel tuleb see Github Packages registrisse üles laadida. Tulevikus saab seda vajadusel alla laadida ja käivitada. *Dockerfile*'i ülesehitus on näha kuvatõmmisel 3.4.3.2.

```
FROM node:10 as build-stage
WORKDIR /app
COPY package*.json /app/
RUN npm install
RUN npm audit fix
COPY ./ /app/
RUN CI=true npm test
ENV REACT_APP_API_URL http://localhost:8080
RUN npm run build
FROM nginx:1.15
COPY --from=build-stage /app/build/ /usr/share/nginx/html
COPY nginx.conf /etc/nginx/conf.d/default.conf
EXPOSE 80
```

Kuvatõmmis 3.4.3.2. Kasutaja poolse projekti *Dockerfile* ülesehitus *Docker Image* loomiseks

3.4.4 Pidevintegratsiooni ja -valmiduse (CI/CD) konveieri implementatsioon

Loo eesmärgiks on projekti implementeerida CI/CD konveier, mille eesmärgiks on automatiseerida *Docker Image* loomist ja Github Actions registrisse üles laadimist ning seejärel teavitada Slacki vahendusel protsessi lõpptulemusest. Algne plaan oli kasutada Jenkinsit, kuid kui lõpuks looga tegelema hakati oli kinnisest beetaversioonist välja tulnud Github Actions. Github Actions otsib projekti *.github* kaustast *.yml* lõpuga faile, mida nimetatakse töövoo(ingl *workflow*) failideks. Failid algavad *workflow* nimega, millele järgneb sündmus, mis käivitab tööd. Sündmuseid on erinevaid ja neid saab omakorda ühendada kindlate harudega. Näiteks on projektis kasutatud *push* käsu sündmust *master* harus, mis tähendab seda, et iga kord kui *master* harusse laetakse uus muudatus käivitub ettenähtud töö. Töödesse kirjutatakse nende nimi ja seejärel pannakse paika missugust keskkonda(ingl *runner*) töö kasutab. Github pakub omalt poolt nelja eelkonfigureeritud keskkonda: Windows Server 2019, Ubuntu 18.04, Ubuntu 16.04, macOS Catalina 10.15. Teine võimalus on ise selline keskkond kogu vajaliku tarkvaraga üles seada. Peale keskkonna valikut pannakse kirja tegevused, mida selles kindlas töös tuleb teha. Tegevusi on võimalik ise programmeerida kasutades selleks samu käske, mida kasutatakse ka käsureal. Tihti kasutatavate tegevuste jaoks on olemas ka "turg", kust saab neid lihtsalt kaasata oma projekti. Käesolevas loos on kasutatud kolme sellist tegevust: *Github Actions Checkout, mr-smithers-excellent Docker Build & Push Action, rtCamp Slack Notify - Github Action.* Github Actions *Checkout* võimaldab ligipääsu repositooriumile. *Docker Build & Push Action* ehitab *Docker Image*'i ja laeb selle üles

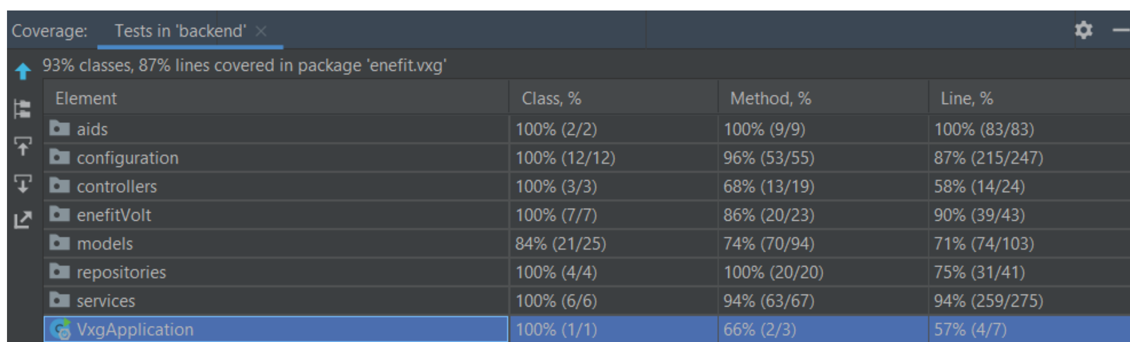
ettemääratud registrisse. *Slack Notify* postitab *secrets.SLACK_WEBHOOK* väljas defineeritud ühenduse järgi teavituse tehtud muudatustest.

```
name: Build and push to Docker image, then notify in Slack

on:
  push:
    branches:
      - master

jobs:
  buildPushBackend:
    name: Build & push Backend Docker image
    runs-on: ubuntu-latest

    steps:
      - uses: actions/checkout@v2
      - uses: mr-smithers-excellent/docker-build-push@v3
        with:
          image: vxg-virtual-vehicle/vxg-virtual-vehicle-backend
          registry: docker.pkg.github.com
          githubOrg: enefit
          dockerfile: backend/Dockerfile
          username: ${{ github.actor }}
          password: ${{ secrets.GITHUB_TOKEN }}
```

Kuvatõmmis 3.4.4.1. osa töövoo faili ülesehitusest

3.5 Testid

Rakenduses on 87% koodiridadest ja 95% klassidest kaetud testidega. Kuvatõmmises 3.5.1 on näidatud iga paketi koodi kaetust ning terve projekti üldine kaetus. Kokku on 74 testi. Neist 56 on ühiktestid ning 18 integratsioonitestid.


Coverage: Tests in 'backend'

93% classes, 87% lines covered in package 'enefit.vxg'

Element	Class, %	Method, %	Line, %
aids	100% (2/2)	100% (9/9)	100% (83/83)
configuration	100% (12/12)	96% (53/55)	87% (215/247)
controllers	100% (3/3)	68% (13/19)	58% (14/24)
enefitVolt	100% (7/7)	86% (20/23)	90% (39/43)
models	84% (21/25)	74% (70/94)	71% (74/103)
repositories	100% (4/4)	100% (20/20)	75% (31/41)
services	100% (6/6)	94% (63/67)	94% (259/275)
VxgApplication	100% (1/1)	66% (2/3)	57% (4/7)

Kuvatõmmis 3.5.1 Projekti koodi kaetus testidega

3.5.1 Ühiktestid

Iga teenuse jaoks on arendatud ühikteste, mille jaoks kasutatakse Mockito raamistikku. Kuvatõmmises 3.5.1.1 on näidatud *ChargeServiceUnitTest* klass. Testi meetodis *chargeShouldSaveIncrementedSocValue_SocIs25Percent* testitakse, et *setStateOfChargeWh* meetodit kutsutaks välja ainult üks kord ning õige väärtuse *stateOfCharge* muutuja sisse salvestamist. Privaatses abimeetodis *mockChargingValues* luuakse imiteeritud väärtuseid(ingl *mock values*). Testimisel on kasutatud *Given-When-Then* testide arendamise stiili.

```
@RunWith(MockitoJUnitRunner.class)
public class ChargeServiceUnitTest {
    @Mock
    private VehicleProperties vehicleProperties;
    @Mock
    private ChargerProperties chargerProperties;
    @Mock
    private VehicleState vehicleState;
    @InjectMocks
    private ChargeService chargeService;

    private long currentTime = System.currentTimeMillis();

    private void mockChargingValues(double stateOfCharge, long timePassedAfterIncrementing, double chargingPower) {
        when(vehicleState.getStateOfChargeWh()).thenReturn(stateOfCharge);
        when(vehicleState.getChargingMode()).thenReturn(VehicleState.ChargingModeEnum.REAL);
        when(vehicleState.getLastSocIncrementingTime()).thenReturn(currentTime - timePassedAfterIncrementing);

        when(vehicleProperties.getBatteryCapacitykWh()).thenReturn(38.0);
        when(vehicleProperties.getChargeEfficiencyPercentage()).thenReturn(90);
        when(chargerProperties.getChargingPowerKw()).thenReturn(50.0);

        chargeService.setChargingCurveValuesKW(new double[100]);
        int socPercent = (int) (stateOfCharge * 100 / vehicleProperties.getBatteryCapacitykWh() / 1000);
        chargeService.getChargingCurveValuesKW()[socPercent] = chargingPower;
    }

    private ArgumentCaptor<Double> getDoubleArgumentCaptor() {
        return ArgumentCaptor.forClass(Double.class);
    }

    @Test
    public void chargeShouldSaveIncrementedSocValue_SocIs25Percent() {
        //given
        mockChargingValues( stateOfCharge: 9500.0,  timePassedAfterIncrementing: 34950,  chargingPower: 43.5);
        ArgumentCaptor<Double> captor = getDoubleArgumentCaptor();

        //when
        chargeService.charge(Duration.ofSeconds(1), currentTime);

        //then
        verify(vehicleState, times( wantedNumberOfInvocations: 1)).setStateOfChargeWh(anyDouble());
        verify(vehicleState).setStateOfChargeWh(captor.capture());
        assertThat(captor.getValue(), closeTo( operand: 9880.0d,  error: 0.0));
    }
```

Kuvatõmmis 3.5.1.1. *ChargeServiceUnitTest* klass

Kuvatõmmises 3.5.1.2 on näidatud *RouteController*'i jaoks ühiktesti näidis, kus testitakse, et kontroller kutsub *RouteService*'t ainult ühe korra.

```java
@RunWith(MockitoJUnitRunner.class)
public class RouteControllerUnitTest {
    @Mock
    private RouteService routeService;
    @InjectMocks
    private RouteController routeController;

    @Test
    public void getRouteFromHomeToWorkShouldCallGetRouteFromHomeToWorkFromVehicleServiceOnlyOneTimeTest() {
        //given
        when(routeService.getRouteFromHomeToWork()).thenReturn(Route.builder().build());

        //when
        routeController.getRouteFromHomeToWork();

        //then
        verify(routeService, times( wantedNumberOfInvocations: 1)).getRouteFromHomeToWork();

        assertNotNull(routeService.getRouteFromHomeToWork());
    }
```

Kuvatõmmis 3.5.1.2. *RouteControllerUnitTest* klass

3.5.2 Integratsioonitestid

Kontrollerite jaoks arendati ka integratsiooniteste, kus kasutati SpringRunner raamistikku.

Kontrollerite integratsioonitestide klassides testitakse kontrolleri ja teenuse omavahelist toimimist. Kuvatõmmises 3.5.2.1 on näidatud *ChargingControllerIntegrationTest*, kus testitakse, et saadud režiim ei oleks tühi.

```java
import ...

@RunWith(SpringRunner.class)
@SpringBootTest
public class ChargeControllerIntegrationTest {
    @Autowired
    private ChargeController chargeController;

    @Test
    public void chargingModeIsNotNull() { assertNotNull(chargeController.getChargingMode()); }

    @Test
    public void realChargingModeIsNotNull() { assertNotNull(chargeController.putRealChargingMode()); }

    @Test
    public void linearChargingModeIsNotNull() { assertNotNull(chargeController.putLinearChargingMode()); }
}
```

Kuvatõmmis 3.5.2.1. *ChargeControllerIntegrationTest* klass

Kontrolleri MVC testide klassides testitakse, kas vastuse kood on 200 *successful*. Kuvatõmmises 3.5.2.2 on näidatud *VehicleControllerIntegrationTest*, kus testitakse kontrolleri */configuration*, */state* ja */stateOfCharge* GET meetodite ning */state/status* PUT meetodit @*RequestBody* parameetriga.

```java
@RunWith(SpringRunner.class)
@SpringBootTest
@AutoConfigureMockMvc
public class VehicleControllerMvcTest {
    @Autowired
    private MockMvc mockMvc;

    @Test
    public void getConfigurationResponseIsSuccessfulTest() throws Exception {
        mockMvc.perform(get( urlTemplate: "/configuration")).andExpect(status().is2xxSuccessful());
    }

    @Test
    public void getStateResponseIsSuccessfulTest() throws Exception {
        mockMvc.perform(get( urlTemplate: "/state")).andExpect(status().is2xxSuccessful());
    }

    @Test
    public void getStateOfChargeResponseIsSuccessfulTest() throws Exception {
        mockMvc.perform(get( urlTemplate: "/stateOfCharge")).andExpect(status().is2xxSuccessful());
    }

    @Test
    public void updateStatusResponseIsSuccessfulTest() throws Exception {
        mockMvc.perform(put( urlTemplate: "/state/status").contentType(MediaType.APPLICATION_JSON)
                .content("\"PLUGGED_IN\"")).andExpect(status().is2xxSuccessful())
                .andExpect(content().string( expectedContent: "\"PLUGGED_IN\""));
    }
}
```

Kuvatõmmis 3.5.2.2. *VehicleControllerIntegrationTest* klass

4 Analüüs ja järeldused

4.1 Tulemuste tehnilise teostuse põhjendus

Selles osas teostatakse lähemalt töö protsessi tulemuste peatükis olevate kasutajalugude analüüsi ning arutletakse arendamise kordaminekute ja kitsaskohtade üle. Samuti on välja toodud tehtud testide meetodeid ja koodi kaetust testidega.

4.1.1 Ilmateave kättesaamine

Ilmateave kättesaamine kasutajalugu jäi sügissemestri meeskonnaprojekti lõppemise ajaks poolikuks ning tuli ära lõpetada. Kuna antud kasutajaloo esialgset autorit enam tiimis polnud, pidi selle lõpuni viima uus arendaja. Väga palju muutma ei pidanud, sest suurem osa tööst oli juba tehtud. Suurem osa ajast kulus koodi efektiivsemaks muutmisele, refaktoorimisele ja testimisele. Eelmise autori poolt oli valmis tehtud *WeatherService* ja *WeatherConditions* klassid. *WeatherConditions* klassis on kõik ilmastiku omadused nagu temperatuur, tuule kiirus jne. *WeatherService* klass küsis iga minuti tagant *OpenWeatherMap* API-lt hetkeilma koduaadressi linna järgi. Antud projektis muudeti aga seda koordinaatide järgi küsima. Samuti modifitseeriti API päringut. Võeti kasutusele Unirest HTTP kliendi teeki, mis muutis POST päringu koodis kirjutamise palju lihtsamaks ja arusaadavamaks. API päringus olevate URL-i ja aplikatsiooni identifikaatori jaoks loodi *WeatherConfiguration* klass ning antud URL ja *appID* lisati *application.properties* faili. *WeatherConfiguration* klass saab neid väärtuseid sealt küsida ja *WeatherService* klassile edastada. See viis on parem, sest siis pole antud väärtused koodis püsiprogrammeeritud ning vajadusel saab neid lihtsasti *application.properties* failis muuta. Viimase tegevusena lõpetati ära testimine. Antud kasutajalugu oli üpriski lihtne, pidi küsima API-lt väärtuseid ja neid salvestama.

WeatherService klassi testimiseks matkiti *VehicleProperties* ja *WeatherConfiguration* klasse. Nende abil testiti õigete koordinaatide olemasolu API päringu tegemisel, ilmastikuolude olemasolu, API päringu õnnestumist ja ilmastiku uuendamist.

WeatherService klassi koodi testidega kaetus on 100%.

4.1.2 Laadija parameetrid

Antud kasutajalugu oli üks lihtsamatest, sest ainus ülesanne oli serveripoolsesse projekti laadija klassi loomine. Algselt plaaniti lisada laadimisjaama objektile peale laadimisvõimsuse omadusele ka mark ja mudel, aga hiljem otsustati, et neid pole siiski vaja lisada, sest need ei mõjuta laadimist.

4.1.3 Sõidukid liiguvad kasutades reaalseid marsruute

Antud kasutajaloo raames on marsruudi ehitamise ja praeguse asukoha leidmise loogika eraldatud kahte eraldi teenusesse, seda asukoha leidmise paljude arvutuste tõttu. OpenRouteService'ist marsruudi saamine tõsteti peateenusest eraldi klassidesse mugavaks automaattestimiseks.

Kõik keerulised arvutused pandi eraldi privaatsetesse meetoditesse ja väärtused edastati parameetrite kaudu koodi loetavuse ja vajadusel ilmneva tulevase refaktoreerimise lihtsustamiseks. Mitmed väärtustega loetelud, mis sisaldavad aja intervalle, teekonnapunktide vahelisi pikkuseid ja absoluutseid ajaväärtuseid igas teekonnapunktis salvestatakse marsruudi sisse, et vältida igakordseid pikaajalisi arvutusi.

Kasutajaloo arendamise protsessis tekkisid erinevad refaktoreerimise vajadused puhta ja loetava koodi saavutamiseks. Näiteks, selleks et vältida rohkem kui kolme parameetrit ühes meetodis oli vaja koordinaatide laius- ja pikkuskraadi jaoks luua uus mudeli klass - *GeoCoordinate*.

Seda ülesannet arendati Aku laadimine ja tühjenemine kasutajalooga samaaegselt, kuna aku tühjenemise loogika on tihedalt seotud auto liikumise loogikaga. Paralleelse arendamisega esinesid töö tegemisel mõned raskused, kuid koostöö abil see probleem lahendati.

Automaattestimine võttis palju aega ja tekitas raskusi välise teenuse kasutamise tõttu. *RouteService*'i ja *RouteController*'i jaoks arendati ühiktestid, mille jaoks kasutatakse Mockito raamistikku, imiteeritud objekte ja *Given-When-Then* testide arendamise stiili.

Arendati ka integratsiooniteste, kus kasutati SpringRunner raamistikku. *RouteControllerIntegrationTest* klassis testitakse kontrolleri ja teenuse omavahelist toimimist. *RouteControllerMvcTest* klassis testitakse, kas vastuse kood on 200 *successful*.

RouteService testidega koodi kaetus on 94% ja *RouteController* testidega koodi kaetus on 60%.

4.1.4 Kaardipõhine simulaator

Selle kasutajaloo käigus tuli kaardile esitada serveripoolsest osast tulnud kodu ja töö asukohad, linnulennult marsruuti nende vahel ning elektriautosi. Selle arenduseks valis arendustiim Meeskonnaprojekti alguses välja kasutajaliidese arenduse platvormiks Angulari. Seda just seetõttu, et varasem semester oli sellega olnud kokkupuude ning oli autoritele familiaarsem. Õppides rohkem kaardilahenduste kohta, leidsid autorid, et kõige sobivamini toetatud kaardimoodul asub React.JS raamistikus ning autorid võtsid vastu otsuse minna üle Reactile. Kahe raamistiku õppimise alla läks töö autoritel palju aega ning seetõttu kujunes ka see kasutajalugu kõige ajakulukamaks kasutajalooks. Lõppude lõpuks pidid arendajad tutvuma viie täiesti uue asjaga, milleks olid Typescript, mida kasutas Angular, Javascript, mis oli vajalik React.JS-i arenduseks, React.JS ise, Leaflet raamistik ning React-Leaflet, mis on loodud, et teha hõlpsamaks Leafleti kasutamine React raamistikus.

Meeskonnaprojekti semestri lõpuks oli valminud arendajatel interaktiivne kaart, millel oli Leafleti enda poolt arendatud Leaflet Routing Machine pistikprogrammi, kuid mis oli ainult ajutine viis, et kuvada kaardile marsruut kodu ja töö asukohtade vahele. Ka testimine valmistas arendajatele alguses palju muret, sest internetist otsides jäi mulje, et väga paljud arendajad ei testi Reacti koodi just seetõttu, et väga palju abi sealt ei osanud leida. Sealjuures võib jällegi välja tuua arendajate kogenematuse Reactiga.

Lisaks sai kuvada kaardile kodu ja töö asukoha markerid ning auto markeritele peale vajutades saab rohkem teada infot selle markeri kohta, nt. automark, kodu ja töökoha asukoha aadressid jne. Võib öelda, et meeskonnaprojekti lõpuks olid põhiteadmised kätte saadud ning asi hakkas kiiremini edasi liikuma. Kokkuvõtteks võib öelda, et antud kasutajalugu esimesel semestril oli autorite jaoks aeglane kuid vajalik protsess, kuna see kasutajalugu on baasiks kogu ülejäänud kaardiga seotud arendusele ning autorid tegid suure arengu just kasutajaliidese osa arenduses.

Kuna kasutajalugu ei olnud eelmise semestri aine Meeskonnaprojekt - ITB1706 jooksul valmis saanud, sest veel oli kaardile vaja lisada linnulennult marsruut kodu ja töökoha vahele ning testimine oli jäänud ebapiisavaks. Selle semestri alguseks olid tekkinud ka

mõned uued nõudmised ning vanade vigade parendused. Nendeks kujunes välja marsruudi kujutamine reaalsete autoteede peal ja mitte enam linnulennult, testimise parem implementatsioon ning koodi refaktoreerimine.

Esimese arendusetapi käigus arendajad nägid vaeva, et saada kõige esimesena korda testimine, mis võttis aega, kuid lõpuks peale paari nädalat valmis saadi. Esimese arendusetapi lõpuks oli valmis saanud ka teekond kodust tööle esialgu küll linnulennult, kuid arhitekt palus, et nüüdseks oleks juba reaalne marsruut sõidetavate autoteede peal. See ülesanne jäi lahendada teise arendusetapi jooksul, mis seal ka valmis sai.

Selle kasutajaloo suurimateks probleemideks kujunesid välja varasemalt mainitud testimine ja React keskkonnas lubaduste mitte defineeritud (ingl *undefined*) seisundite lahendamised. Lubadused on Mozilla Firefoxi definitsiooni järgi objekt, mida kasutatakse, et töödelda asünkroonseid arvutusi, millel on mõningad olulised garantiid, mida on keeruline käsitleda kasutades tagasikutsumise (ingl *callback*) metoodikat. Lihtsalt öeldes on lubadus objekt, mis ümbritseb väärtust, mis võib olla arendajatele teada või mitte, kui objekt on loodud ning pakub meetodit väärtuse käsitlemiseks pärast seda, kui ta on teada ehk *Resolved* või tõrke tõttu mitte kättesaadav ehk *Rejected* [47].

Undefined objekti definitsioon kujunes arendajate jaoks välja enda kogemusest, kui prooviti saata kahe komponenti vahel objekte, kuid need jäid teisele komponendile kättesaamatuks ning seetõttu oli ta seisund ka määratlemata.

Selle kasutajaloo lõpuks olid arendajad saanud oma pagasisse mitmeid vajalikke õppetunde ja teadmisi React.JS'i kohta, mis muutsid edaspidise arengu märgatavamalt lihtsamaks ja kiiremaks.

MapComponent testimiseks loodi imiteerivaid objekte väärtused API päringutest ja sisestati need testobjekti mähisklassi [48], mis on sama mis *MapComponent* komponent. Vaadati, et testiobjekti väärtused võrduvad imiteeritud väärtustega ning testiti ka, et komponenti renderdatakse. Sealhulgas testiti ka seda, kas *LayersComponent*'i renderdatakse. Testimiseks kasutati Jest testimisraamistikku.

4.1.5 Aku laadimine ja tühjenemine

Antud kasutajalugu osutus üsna aeganõudvaks. Põhiliseks ülesandeks oli auto olekute vahetamise funktsionaalsuse loomine. Näiteks sõiduki liikuma panemiseks on vaja välja kutsuda meetod *drivingAction*, mis määrab sõitmise olekuks.

Algselt loodi meetod, mis muutis auto olekut *switch*'i abil, kuid selline teostus osutus probleemseks, sest see blokeeris rakenduse põhilõime, s.t. juhtlõime. Probleemi uurimise käigus leiti, et on hoopis vaja kasutada mitut protsessori lõime, mis jookseksid peamise lõimega paralleelselt. Samuti avastati, et olekute efektiivseks vahetamiseks on Springi poolt juba loodud Olekumasina teek, mis pakub paindlikku ja mugavat võimalust olekute vahetamise juhtimiseks.

Vajalike protsesside käivitamiseks eraldi lõimedes kasutab rakendus klassi *ThreadPoolTaskScheduler*, mis koos *PeriodicTrigger* klassi abiga võimaldab käivitada vajalikke toiminguid iga määratud intervalli tagant, mis antud juhul oli üks sekund. Näiteks kui sõiduk on sõitmise olekus, käivitatakse aku tühjenemise funktsioon igal sekundil.

Kasutajaloo arendamise käigus avastati ka, et mitme töölõime ohutuks toimimiseks on vaja lukustada ühes lõimes muudetava objekti omaduste muutmine teiste lõimete jaoks, ilma selleta võib tekkida rakenduse töötamise käigus ootamatuid probleeme(ingl *unexpected behaviour*).

Lõpptulemusena loodi vajalik funktsionaalsus, mille kaudu saab elektrisõiduki olekut mugavalt ja ohutult muuta ning selle kaudu käivitada aku tühjenemise või laadimise protsesse.

4.1.6 Laadimisrežiimide vahel lülitamine

Antud kasutajaloos arendati nii serveripoolset osa kui ka kasutajaliidest. Serveripoolset projekti osa oli üsna lihtne rakendada. Kuigi režiimide vahel lülitamise loogika oli lihtne ja selle rakendamine ei nõudnud palju koodiridu, see oli tehtud eraldi teenuses tulevikus laadimise loogikate arendamist.

Kliendipoolne osa võttis palju aega, sest see oli antud loo arendajale esimene kogemus React JS raamistikuga. Kõige keerulisem oli nupu komponendi ühendamine serveripoolse

lõpp-punktiga, et nupule vajutades uuendataks olekut serveripoolses osas. Lahendust oli raske arendada, sest rakenduses sarnaseid lahendusi veel ei olnud.

Kasutajaloo raames arendati integratsiooniteste kontrolleri jaoks, kus kasutati SpringRunner raamistikke. *ChargeControllerIntegrationTest*'is testiti *ChargeController*'i ja *ChargeService*'i integratsiooni. *ChargeControllerMvcTest*'is testiti, kas vastuse kood on 200 - edukas.

ChargeController'i koodi kaetus testidega on 100%.

4.1.7 Sõiduki liikumine kasutajaliideses

Antud kasutajaloo arendamine osutus üpriski keerukaks. Selle kasutajaloo arendamise alustushetkeks oli kaardile loodud elektriauto ning kodu- ja tööasukohtade markerid, samuti kujutati kaardil auto teekonda kodust tööle. Serveripoolses osas oli loodud loogika, mis paneb auto sõitma, kui POST API päringuga auto staatus muuta sõitmiseks. Loogika oli olemas, kuid kaardil seda näha polnud. Esialgu tehti *MapComponent* klassi *componentDidMount* meetodi sisse uus API GET päring, mis küsib serveripoolse osa projektist iga 2 sekundi tagant auto asukohta ning uuendab *MapComponent*'i olekus hoiustatud andmeid. Järgmisena tuli saata olekus hoiustatud andmed atribuutidena *LayersComponent*'i. Peale selle valmis tegemist oli väga pikalt probleemiks atribuutide uuendamine *LayersComponent* klassis. Kuigi *MapComponent* klassis olekus uuenes iga 2 sekundi tagant, jäi auto asukoha atribuut *LayersComponent*'is alati samaks. Probleem oli selles, et *LayersComponent*'i renderdatakse ainult üks kord ja atribuudid on staatilised. *LayersComponent* uuendamiseks leidsid autorid Leafleti dokumentatsioonist uue meetodi - *updateLeafletElement*, mis kutsutakse iga kord välja kui Leafleti element saab endale sisse uued atribuudid. Peale selle meetodi avastamist prooviti igat pidi panna auto marker liikuma, mida siiski ei juhtunud. Tuli välja, et ka marker on staatiline ning seda ei saa liigutada. Pärast pikemat internetis otsimist leiti *Drift Marker*. *Drift Marker*iga kaasnes väga lihtsasti üks meetod nimega *slideTo*. Peale seda läks juba väga lihtsasti, kuna *slideTo* meetodi sisse sai panna alati uued koordinaadid ning auto liikus eelmiselt koordinaadilt uuele. Ehk siis peale kasutajaliidese projekti käivitamist ning kaardi renderdamist, tuli teha Postmani kaudu POST päring, mis muutis auto staatuse sõitvaks, mistõttu auto asukohakoordinaadid muutuma hakkasid. *MapComponent* küsib iga kahe sekundi tagant auto koordinaate ja tuvastab muudatuse ning uuendab olekut, mille kaudu omakorda uuendatakse *LayersComponent* klassi atribuudid auto koordinaatide kohta.

UpdateLeafletElement'is tuvastatakse jälle, et atribuudid on muutunud ning kutsutakse välja *Drift Marker*'i meetod *slideTo*, kuhu sisestatakse atribuudi kujul uued koordinaadid. Ja nii iga 2 sekundi tagant auto liigub edasi, kuni ta jõuab oma sihtkohta ning jääb seisma. Kokkuvõtteks arenesid arendajad selle kasutajaloo käigus veelgi rohkem kasutajaliidese arenduses, arendajad õppisid veelgi rohkem React koodi manipuleerima ning Leaflet'i enda jaoks ära kasutama.

Kasutajaloo testimiseks lisati *MapComponent* testiklassi juurde vajaminevaid *mock* API päringute vastuste väärtuseid, sisestati need mähisklassi objekti ning kontrolliti nende olemasolu renderdamisel.

4.1.8 Sõiduki liikumine serveripoolses osas

Sõiduki liikumine serveripoolse osa kasutajalugu esialgselt ei sisaldanud kliendipoolse osa loogika programmeerimist, kuid selle kasutajaloo töötamise käigus saadi aru, et see vajadus siiski eksisteerib. Senimaani oli kaardil näidatud ainult teekonda kodust tööle kuid reaalsuses auto ei saa sama sõiduteed kasutades töölt koju sõita. Serveripoolses osas oli õnneks juba saadaval teekond töölt koju ning tuli välja mõelda viis, kuidas teekonda kaardil vahetada olenevalt auto lähte- ja sihtkohast. Kõigepealt loodi uus atribuut serveripoolse osa projekti klassi *VehicleState* nimega *StartLocationEnum*, mis saab olla kas *HOME* või *WORK*, olenevalt kust asukohast auto oma teekonda alustas ning mis muutub vastupidiseks, kui auto jõuab oma sihtkohta. Kuna auto staatuse küsimise loogika oli kliendipoolse projekti *MapComponent* klassis juba olemas, lisati juurde lihtne *if-else*-lausetega kontroll, mis vaatab järele, kas auto *StartLocationEnum* oli kas *HOME* või *WORK*.

Samuti ei teatud enne, et ka Leaflet Polyline, mida kasutatakse teekonna kujutamiseks kaardil, omab lihtsat meetodit, mis infokihti ka uuendab. Lõpptulemuse saavutasidki autorid kui jõudsid Polyline meetodini *setLatLngs*. Peale pikka pusimist, kus prooviti saada Polyline tööle varasemalt tuttavate ja kasutatud meetoditega, leidsid arendajad väga kerge ja sarnase lahenduse nagu sõiduki liikumise kasutajaliidese kasutajaloos. See tuli asetada *LayerComponent* klassis asuvasse *updateLeafletElement* meetodisse ning too hakkas uuendama ennast iga kord, kui *MapComponent* sai uued koordinaadid enda olekusse läbi API GET serveripoolse osa päringu. Tuli välja, et ka *updateLeafletElement* meetodisse tuli panna kontroll, et kas tegu *StartLocationEnum* on kas *HOME* või *WORK*,

sest muidu oleks teekonda konstantselt kaardil uuendanud, mis tähendas et see kadus iga 2 sekundi tagant kaardilt ja auto marsruudi infokihti pidi uuesti vajutades sisse lülitama.

Siinkohal võib jälle välja tuua Leafleti dokumentatsiooni ebamäärasuse ning arendajate kogenematuse Leafleti kasutamisega. Lõpuks andis see kasutajalugu arendajatele kõige enam lisateadmisi Leafleti kasutamise kohta ning kogu kasutajaliidese arendus muutus peale seda kasutajalugu väga palju arusaadavamaks ning lihtsamaks.

Serveripoolses osas on läbitud teepikkuse loogika testitud *RouteService* klassis, mille koodi kaetus on 94% ja *RouteController* klassi koodi kaetus on 60%.

Front-end poole testimiseks lisati jällegi uusi vajaminevaid API päringute vastuste väärtuseid *MapComponent* testklassi ning kontrolliti nende olemasolu mähisklassi objekti renderdamisel.

4.1.9 Kodu- ja tööasukohtade vaheline kaugus

Kasutajaloo läbiviimiseks oli vaja tööd teha mõlemas projekti serveripoolses ja kliendipoolses osas. Selleks oli vaja luua serveripoolses osas meetodid, mis lähtestavad ja regenereerivad uued väärtused. Nende meetodite arendamisel ja testimisel raskusi polnud, sest antud töö serveripoolsest osast tegelenud arendaja oli ka varem põhjalikult kokku puutunud uute lõpp-punktide loomise ja loogikate ühendamisega.

Kasutajaliidese poolel töötanud arendaja muutis suuresti, kuidas kasutajaliides välja nägi ning lisas juurde nupud lähtestamiseks ja uuesti genereerimiseks, millele lisati juurde ka REST API päringud, mis ühendavad projekti serveripoolsest osast loodud meetoditega. Arendaja jaoks ülesanne suurt raskust ei valmistanud, kuid lisas enda pagasisse väärtuslikku kogemust juurde disaini ning Material UI React.JS komponentide koha pealt.

4.1.10 Sõiduki aku laadimiskõver

Laadimise loogika oli implementeeritud laadimisrežiimide vahel lülitamise kasutajaloo raames loodud teenuses. Loogika sisaldab palju keerulisi arvutusi, mis pandi eraldi privaatsetesse meetoditesse ja väärtused antakse edasi parameetrite kaudu koodi loetavuse ja tulevase refaktoreerimise lihtsustamise jaoks.

Kõige keerulisem töö protsessis oli laadimise kõvera graafiku füüsilisest tähendusest arusaamine ja laadimise algoritmi koodis implementeerimine.

Samuti reaalses režiimis laadimine on mõistlik ainult kiirlaadimise tüüpi laadijaga(nt *CHAdeMO*), kus pakutakse kõrgemaid võimsuseid(50kW). Kui laadimine toimub kodus (nt tavapistikust 230V ja 10A), siis laadida võiks maksimaalselt selle võimsusega, mida suudetakse pakkuda(2.3kW). Sellise olukorraga oli vaja arvestada ja lisada simulaatorisse kõveraga kiirlaadimise võimalus.

Koodi testiti põhjalikult ja kiiresti, sest ühiktestimine ei tekitanud enam raskusi nagu varem. Selle kasutajaloo raames arendati ühikteste *ChargeService*'i jaoks. Kasutati Mockito raamistikku, imiteeritud objekte ja *Given-When-Then* testide arendamise stiili.

ChargeService testidega koodi kaetus on 100%.

4.1.11 Sõiduki aku tühjenemiskõver

Aku tühjenemise loogika jaoks loodi eraldi teenus, sest oli planeeritud palju arvutusi. Kuigi loogika oli laadimise loogikaga sarnane, oli see siiski erinev, seda peamiselt, sest see on tihedalt seotud auto liikumise loogikaga.

Algne implementatsioon kasutas laadimise algoritmi, kus laadimine toimub „sammude kaupa" ja iga sammuga aku täituvus suurendatakse ühe protsendi ehk kindla energiahulga võrra. Aga see implementatsioon oli liiga massiivne ja ebamõistlikult keeruline. See nõudis lisa väärtuste salvestamist sõiduki siseolekus ja sisaldas liiga palju arvutusi. Seetõttu implementeeriti kasutajalugu uuesti. Kõveraga tühjenemine toimub sama lihtsa algoritmiga, nagu lineaarses režiimis, välja arvatud tegeliku elektritarbimise kalkuleerimine.

Töö käigus avastati vanad vead, mida oli vaja parandada. Näiteks aku täieliku tühjenemise juhul sõiduk jätkas liikumist, kuigi seda ei oleks tohtinud juhtuda. Samuti salvestati eelmiseid oleku väärtuseid sõiduki siseolekus. See viga lahendati oleku kloonimisega ja selle sügav koopia(ingl *deep copy*) teenuses kasutamisega.

Samuti esines probleeme õige sõiduki mudeli jaoks tühjenemise kõvera saamisega. Nissani keskusest saadeti vastus, et seoses COVID-19 pandeemiaga olid nemad ajutiselt osad tegevused peatanud ja ei suutnud andmeid pakkuda. Probleemi lahendamiseks

otsustati kasutada teise mudeli kõverat, mille kaudu arvutati alternatiivsed kõvera väärtused.

Pärast mentorite poolset ülevaatust oli vaja teha mitmeid muudatusi. Koodi parandati ja refaktoreeriti seni kuni see heaks kiideti.

Selle kasutajaloo raames loodi eraldi teenus *DischargeService*. Selle jaoks arendati ühikteste, mis kasutavad Mockito raamistikku, imiteeritud objekte ja *Given-When-Then* testide arendamise stiili.

DischargeService testidega koodi kaetus on 97%.

4.1.12 Avalikud laadimispunktid kaardil

Käesoleva kasutajaloo jaoks tuli kliendipoolses osas jällegi töötada *MapComponent* ja *LayersComponent* klassidega. Esialgne plaan oli teha API POST päring Enefit Volt API-le ning saada kogu info selle kaudu, see aga kahjuks ei töötanud, millega oldi ka arvestatud. Päring ei töötanud, kuna seda segasid Enefit Volt API poolsed faktorid nagu *sessionID*, CORS [49] seaded ja CSRF [50] tõend (ingl *token*). Neid faktoreid võis veel olla, aga nende segavate faktorite olemasolu võimalus oli autoritele teada. Varuplaan oli luua ajutine kontroller serveripoolse osa projekti. Samuti tuli alla laadida Enefit Volt API õige vastus internetibrauserist. Selleks suumis arendaja Enefit Volt kodulehel kaardi täielikult Eesti raames välja, et näha oleks kõik olemasolevad laadimispunktid. Seejärel sai brauseri konsoolist võtta välja kõikide laadimispunktide info *json* andmetena ning lisada need *json* tüüpi faili. Lühidalt selgitades omasid autorid nüüd kõikide laadimispunktide infot just sellisel kujul, nagu oleks olnud API vastus. See lisati ka serveripoolsesse projekti. Kui see tehtud sai, tuli luua serveripoolse osa projekti ka klass *PublicChargersService*, mis filtreerib sealt samast *json* tüüpi failist välja vajalikud laadimispunktid. Sisendina võetakse sisse kaardi nähtava osa kõige kirde- ja edelapoolsemad punktid. Need saadakse kliendipoolse osa projekti klassist *MapComponent*. *MapComponent*'ile loodi referents (*ref*), mis on viide kogu kaardile ning mille kaudu saadi kätte kaardi omadused nagu kaardipiirid, mis ongi kaardi nähtava osa koordinaatvahemik kõige kirdepoolsemast punktist edelapoolseima punktini. *MapComponent* klassi loodi ka *componentDidUpdate* meetod, mis uueneb iga kord kui *MapComponent* klassi *state* muutub. *GetBounds* kaudu saidki autorid vajalikud koordinaadid ning tegid need oleku osaks, mis tähendab et kui kaarti liigutada, muutuvad

ka koordinaadid ja nende kaudu olek, peale mida kutsutakse välja *componentDidUpdate* meetod. *ComponentDidUpdate* meetodi sees tehakse seejärel koordinaatsisendiga POST päring serveripoolse osa projektis asuvale kontrollerile, mis suhtleb *PublicChargersService* klassiga. See klass omakorda filtreerib välja kõik koordinaatide vahemikus olevad laadimispunktid ja tagastab need *MapComponent* klassile. *MapComponent* klass edastab need *LayersComponent* klassile, mis need laadimispunktid ka kaardile kuvab. Laadimispunktide jaoks loodi ka *LayersComponent* klassis eraldi infokiht ning seda uuendatakse iga kord *updateLeafletElement* kaudu kui laadimispunktide info muutub. Kuna selleks hetkeks oli autoritel juba kaardi interaktiivne uuendamine üpris selge, saadi sellega väga kiiresti hakkama. Lõppkokkuvõtteks ilmuvadki kaardi nähtavasse osasse laadimispunktid ning neid uuendatakse, kui kaarti liigutatakse või kaardil suumitakse sisse/välja. Sellega sai ka antud kasutajalugu valmis. Hiljem on plaanis kaotada ajutine fail serveripoolse osa projektist ning suhelda hoopiski Enefit Volt API'ga, et laadimispunktide info kätte saada. Hetkel aga oligi ülesanne luua võimalikult ligilähedane laadimispunktide kaardile kuvamise protsess, nagu see toimiks Enefit Volt API'ga suheldes. Antud kasutajaloo lahendamine oli autorite jaoks väga huvitav protsess.

Nii nagu Kaardipõhise simulaatori, Sõiduki liikumise serveripoolse ja kasutajaliidese projekti puhul, imiteeriti API päringute vastuste väärtused, mis sisestati mähisklassi objekti ning testiti nende olemasolu renderdamisel.

4.1.13 Dockeri implementatsioon projektis

Antud juhul oli tegemist tehnilise looga. Projekt oli jõudnud sinnamaale, kus tõusis vajalikkus implementeerida Docker. Docker töötab kui virtuaalmasin eraldi konteineris, mis võimaldab ükskõik millises operatsioonisüsteemis projekti käivitada, kuna kõik vajalik info, allalaetud failid jne asuvad Dockeri konteineris ning kasutaja peab ainult Dockeri kaudu antud konteineri käivitama. Docker võimaldab kasutajatel aplikatsioone palju kiiremini käivitada kui tavaliselt. Tavaolukorras peaks kasutajal olema õiged platvormid alla laetud, õigesti konfigureeritud, õiged andmed alla laetud jne. Docker teeb seda kõike kasutaja eest ning kasutajal peab olemas olema ainult Docker ja projekti *Docker Image*. Kuna loodud projekti maht aina kasvab, on see hea võimalus ka teistel projekti käivitada, kes selle kallal iga päev ei tööta, nt. tiimi Eesti Energia poolsed juhendajad.

Serveri poolne osa:

Serveripoolse osa projektil oli valmis loodud *Dockerfile* juba eelmise aasta oktoobrist, kuid Eesti Energia poolne arhitekt eelistas, et *Dockerfile* oleks mitmik astmelise kujuga ning see tuli ümber teha. Ümber kirjutamise käigus esinesid autoril mitmed probleemid, mida polnud internetist otsides kerge leida, nimelt just seetõttu, et *Dockerfile*'i loomiseks võib olla mitmeid erinevaid viise, kuid lõpptulemus on sama. Kuna on mitmeid erinevaid viise, kuidas *Dockerfile* implementeerida, on paljudel neist üleliigset informatsiooni, kuid antud tehnilise ülesande tingimus oli luua *Dockerfile*'i võimalikult efektiivselt - minimaalselt käske, et täita kõik vajalikud kriteeriumid. Seetõttu tuli mitmeid kordi isegi töötavaid lahendusi ümber kirjutada ning kogu aeg uut materjali lugeda ja läbi töötada.

Mure pakkus fakt, et arendaja ei olnud kokku puutunud varasemalt Docker'iga ning kõike tuli nullist õppida. Suurimaks probleemiks kujunes keskkonnamuutujate integreerimine *Dockerfile*'i, mis tavaliselt sai lisatud IntelliJ programmis, kuid Docker ei ole seotud IntelliJ'ga ning seetõttu tuli leida uus lahendus, et projekt töötaks.

Fail sai loodud formaadis - laadida alla Gradle lähtefailid, neid kasutades ehitada üles konteineris projekt, mis loob *jar* tüüpi projekti faili koos vajalike keskkonna muutujatega ning alla laadida Java DK tööriistad, mille abil on võimalik hiljem *Docker Image*'t jooksutades luua Docker konteiner ning selle abil aplikatsioon tööle panna. Seejärel tuli saadud *Docker Image* üles laadida Github Actions registrisse. Tänu sellele on võimalik edaspidi seda projekti Github Actions konveieri kaudu uuendada ja käivitada, kuid see võimalus luuakse alles järgmises arendusetapis.

Kasutajaliidese poolne osa:

Erinevalt serveripoolse osa projektist polnud kliendipoolse osa projekti jaoks seni veel *Dockerfile* loodud ning tuli alustada sellest. Enne seda tehti ka väga põhjalik uurimistöö Dockeri kohta, kuna arendaja jaoks oli Docker seni täiesti tundmatu. Õnneks on Docker väga lihtsasti ja arusaadavalt üles ehitatud ning selle kasutamine on väga levinud, mispärast leidis internetist selle kohta väga palju dokumentatsiooni ja juhiseid. Kui *Dockerfile* valmis sai, tuli seda testida. Selle testimiseks tuli *Dockerfile*'st luua *Docker Image*, see käivitada ning vaadata kas kõik töötab nagu peaks. Pärast mõnda katset ja

Dockerfile'i ümberkirjutamist oligi *Docker Image* valmis, mis tuli üles laadida Github registrisse. Ka see polnud eriti keeruline, kuna selle jaoks oli olemas GitHub enda poolt väga sisukas juhend, mida järgides *Docker Image* ka registrisse üles laeti ning sellega sai ka antud tehnilise ülesande nõue täidetud. Kuna *Docker Image* on nüüdsest leitav GitHub pakettide registrist, saavad kõik, kellel sinna ligipääs on, selle alla laadida ja lihtsasti käivitada ilma midagi muud tegemata. Tänu sellele saavad autorid edaspidi seda ja ka serveripoolse osa projekti Github Actions konveier kaudu koos käivitada, mille jaoks tuleb see integreerida käesolevasse töösse järgmisena.

4.1.14 Pidevintegratsiooni ja -valmiduse konveieri implementatsioon

Antud tehnilise ülesande eesmärgiks oli lisada projekti automatiseerimise valmidus. Ülesande kirjeldus oli valmis kirjutatud juba lõputööle eelneva meeskonnaprojekti käigus ning toodi üle. Algne plaan oli kasutada automatiseerimiseks Jenkinsi tarkvara, kuid ülesande arendamise hetkeks oli avalikkusele kasutamiseks välja tulnud Github Actions. Arhitekti huvi uue tehnoloogia vastu ja tõsiasi, et projekti koodi hoiti juba Githubi repositooriumis, suunas arendajaid kasutama automatiseerimiseks Github Actions'it.

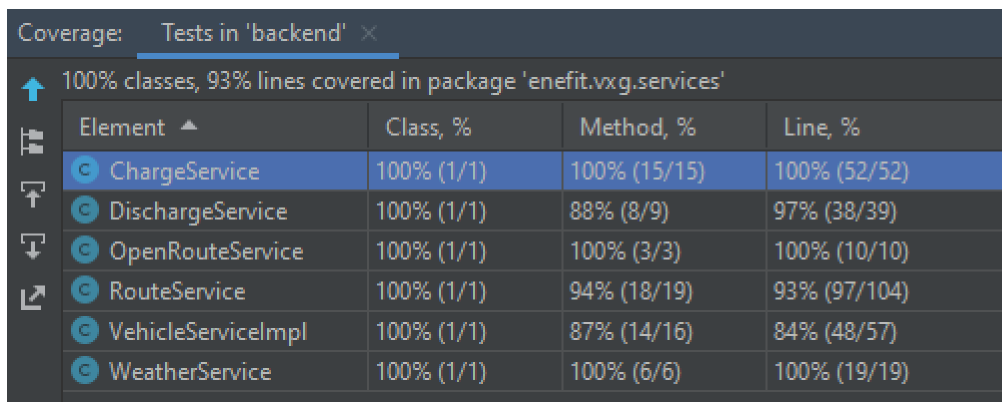
Github Actions'i rakendamine võttis omajagu aega, sest arendaja pidi selgeks tegema endale nii Github Actions'i kui ka põhilise osa Dockerist. Kuigi mõlema rakenduse dokumentatsioon on väga põhjalik, ei leidu veel palju õpetusi, kuidas neid kahte omavahel ühendada. Tihti jäävad õpetused liiga pinnapealseks. Teise probleemina võib välja tuua ühe suure ebamugavuse: lukustatud peaharus ei saa muudatusi koheselt sisse viia. Iga muudatuse jaoks tuleb luua uus tõmbe nõudlus, mis vajab ka mentorite poolset ülevaatust. Lisaks sellele polnud ülesandes täpselt välja toodud, mis peab olema automatiseeritud. Arendaja lõi algselt *workflow*, mis seadis üles Gradle keskkonna, kus käivitati ja testiti rakenduse serveripoolset osa. Ülevaatusel paluti muuta faili nii, et see looks ja laeks üles uued *Docker Image*'d. Ülesande arendamise lõppedes tuli välja, et tuleb järjekordselt ümber teha *Dockerfile*'d, mis näitab vähese integreerimise informatsiooni olemasolu.

4.2 Testid

4.2.1 Automaattestimine

Üheks mittefunktsionaalseks nõudeks oli ühiktestide arendamine. Ühiktestidega kontrollitakse erinevate koodi osade korrektsust ning võimaldab jälgida, et järjekordne koodi muutus ei põhjustaks varem testitud kohtades uusi vigu. See tähendab, et iga autorite poolt loodud funktsiooni või meetodi jaoks tuli luua mitmeid ühikteste.

Teenuste loogika testimiseks kasutati Mockito raamistikku ning selle abil loodud imiteeritud objektid sisestati erinevatesse testistsenaariumitesse. Nõutav ühiktestidega koodi kaetus oli vähemalt 50%. Kuvatõmmises 4.1.5.1 on näidatud teenuste pakettide ühiktestidega koodi kaetust, mis on 93%.



Coverage: Tests in 'backend'

100% classes, 93% lines covered in package 'enefit.vxg.services'

Element	Class, %	Method, %	Line, %
ChargeService	100% (1/1)	100% (15/15)	100% (52/52)
DischargeService	100% (1/1)	88% (8/9)	97% (38/39)
OpenRouteService	100% (1/1)	100% (3/3)	100% (10/10)
RouteService	100% (1/1)	94% (18/19)	93% (97/104)
VehicleServiceImpl	100% (1/1)	87% (14/16)	84% (48/57)
WeatherService	100% (1/1)	100% (6/6)	100% (19/19)

Kuvatõmmis 3.1.5.1. Teenuste koodi kaetus

Tuli testida ka rakenduse osade omavahelist suhtlust integratsioonitestidega, mille sisend olid eraldi ühiktestidega juba testitud. Kontrollereid testiti integratsioonitestidega, kus kasutati SpringRunner raamistikku. Testiti kontrollerite ja teenuste omavahelist toimimist ning API päringute vastuste staatuseid.

4.2.2 Testjuhud

Järgnevalt on välja toodud testkomplektid süsteemi funktsionaalsuse testimiseks, mida kasutati erinevatel testimise staadiumitel. API-sid testiti manuaalset Postmani rakenduse ja kasutajaliideses asuvate nupuvajutuste vastuste abil.

Testjuhtum praeguse asukoha saamiseks, kui auto sõidab

Kirjeldus: GET päring lõpp-punkti */location* tagastab asukoha koordinaati, kui auto on sõitmise olekus.

Eeltingimused: serveripoolse osa ja Postmani rakendused on käivitatud.

Testi sammud:

1. Saada PUT päring *http://server:[port]/state/status* päringu kehaga “*DRIVING*”.
2. Veendu, et vastuse keha on “*DRIVING*”.
3. Saada GET päring *http://server:[port]/location*

Oodatav tulemus: vastuse keha sisaldab asukoha laius- ja pikkuskraade.

Testjuhtum auto aku laadimiseks reaalses laadimisrežiimis:

Kirjeldus: sõiduki akut laetakse kasutades laadimiskõverat ning laadimise protsessi saab jälgida konsoolis.

Eeltingimused: serveripoolse osa ja Postmani rakendused on käivitatud

Testi sammud:

1. Saada PUT päring *http://server:[port]/realChargingMode*.
2. Veendu, et vastuses laadimisrežiim on “*REAL*”.
3. Saada PUT päring *http://server:[port]/state/status* päringu kehaga “*PLUGGED_IN*”.
4. Veendu, et vastuse keha on “*PLUGGED_IN*”.
5. Saada PUT päring *http://server:[port]/state/status* päringu kehaga “*CHARGING*”.
6. Veendu, et vastuse keha on “*CHARGING*”.

Oodatav tulemus: log.info konsoolis kuvatakse aku täituvuse suurenemise protsess. Iga aku täituvuse suurenemise sammu pikkus on erineva intervalliga.

Testjuhtum auto aku laadimiseks lineaarses laadimisrežiimis:

Kirjeldus: sõiduki akut laetakse lineaarselt ning laadimise protsessi on võimalik jälgida konsoolis.

Eeltingimused: serveripoolse osa rakendus ja Postman rakendused on käivitatud.

Testi sammud:

1. Saada PUT päring *http://server:[port]/linearChargingMode*.

2. Veendu, et vastuses laadimisrežiim on “*LINEAR*”.
3. Saada PUT päring http://server:[port]/state/status päringu kehaga “*PLUGGED_IN*”.
4. Veendu, et vastuse keha on “*PLUGGED_IN*”.
5. Saada PUT päring *http://server:[port]/state/status* päringu kehaga “*CHARGING*”.
6. Veendu, et vastuse keha on “CHARGING”.

Oodatav tulemus: *log.info* konsoolis kuvatakse iga sekundilise intervalliga aku täituvuse suurenemine sama elektrienergia koguse võrra.

Testjuhtum auto sõitmiseks mööda reaalset marsruuti

Kirjeldus: auto peab sõitma mööda reaalset marsruuti kodust tööle

Eeltingimused: serveripoolse osa, kasutajaliidese osa ja Postmani rakendused on käivitatud

Testi sammud:

1. Liigu hiirega kasutajaliidese kihtide nuppu kohale kaardi üleval paremas nurgas.
2. Vali infokihi *Home*s juures olev märkeruut.
3. Veendu, et kodu asukoha ikooni kuvatakse kaardil.
4. Vali infokihi *Workplaces* juures olev märkeruut.
5. Veendu, et töö asukoha ikooni kuvatakse kaardil.
6. Vali infokihi *Routes* juures olev märkeruut.
7. Veendu, et marsruut, mis ühendab omavahel töö ja kodu ikoone kuvatakse kaardile.
8. Vali infokihi *Vehicles* juures olev märkeruut.
9. Veendu, et sõiduki ikoon kuvatakse kaardile.
10. Vajuta *DRIVE* nuppu

Oodatav tulemus: sõiduki ikoon liigub mööda marsruuti kodu asukoha ikoonist töö asukoha ikoonini.

Testjuhtum auto aku tühjenemiseks lineaarses laadimisrežiimis.

Kirjeldus: sõiduki aku tühjeneb lineaarselt, laadimise protsessi saab jälgida konsoolis.

Eeltingimused: serveripoolse osa ja kasutajaliidese rakendused on käivitatud.

Testi sammud:

1. Vajuta kasutajaliidesel tumblernupp, et lülita staatus *LINEAR*.
2. Vajuta *DRIVE* nuppu.

Oodatav tulemus: *log.info* konsoolis kuvatakse iga sekundilise intervallidega aku täituvuse vähenemine sama elektrienergia koguse võrra.

Testjuhtum auto aku tühjenemiseks reaalses laadimisrežiimis.

Kirjeldus: sõiduki aku tühjeneb reaalse laadimiskõvera järgi, laadimise protsessi saab jälgida konsoolis.

Eeltingimused: serveripoolse osa ja kasutajaliidese rakendused on käivitatud.

Testi sammud:

1. Vajuta kasutajaliides tumblernupp, et lülita staatus *REAL*.
2. Vajuta *DRIVE* nuppu.

Oodatav tulemus*: log.info* consolis kuvatakse iga sekundilise intervallidega aku täituvuse vähenemine erineva elektrienergia koguse võrra

Testjuhtum tumblernupu testimiseks:

Kirjeldus: tumblernupuga on võimalik lülitada lineaarse ja reaalse režiimi vahel.

Eeltingimused: serveripoolse osa ja kliendipoolse osa ning Postman rakendused on käivitatud.

Testi sammud:

1. Vajuta kasutajaliides tumblernuppu, et lülita staatus *LINEAR*.
2. Saada GET päring *http://server:[port]/getChargingMode*.
3. Veendu, et vastuses laadimisrežiim on "*LINEAR*".
4. Vajuta kasutajaliides tumblernuppu, et lülita staatus *REAL*.
5. Saada GET päring *http://server:[port]/getChargingMode*.

Oodatav tulemus: GET päringu *http://server:[port]/getChargingMode* vastuses laadimisrežiim on "*LINEAR*".

4.3 Kirjanduse ülevaade

4.3.1 Kirjanduse põhjal ülevaade ja analüüs

Antud projekti puhul on tegu täiesti uue ja innovatiivse ideega, mille tulemus on veel ainult spekuleeritav ning kirjandust selle kohta väga palju ei leidu. Antud idee on saanud alguse virtuaalse elektrijaama kontseptsioonist, mida ka Eesti Energia poolt välja arendatakse. Autorite poole loodud projekti keskpunkt, elektriautod ja nende laadimise simuleerimine V2G laadijatega, on üks tükk suuremast pildist ehk virtuaalsest elektrijaamast.

Virtuaalne elektrijaam põhineb tulevikus plaanitud taastuvenergiale ülemineku ja selle jaoks vajaliku energia salvestamise strateegial. Elektriautode akud on üks viis energiat salvestada.

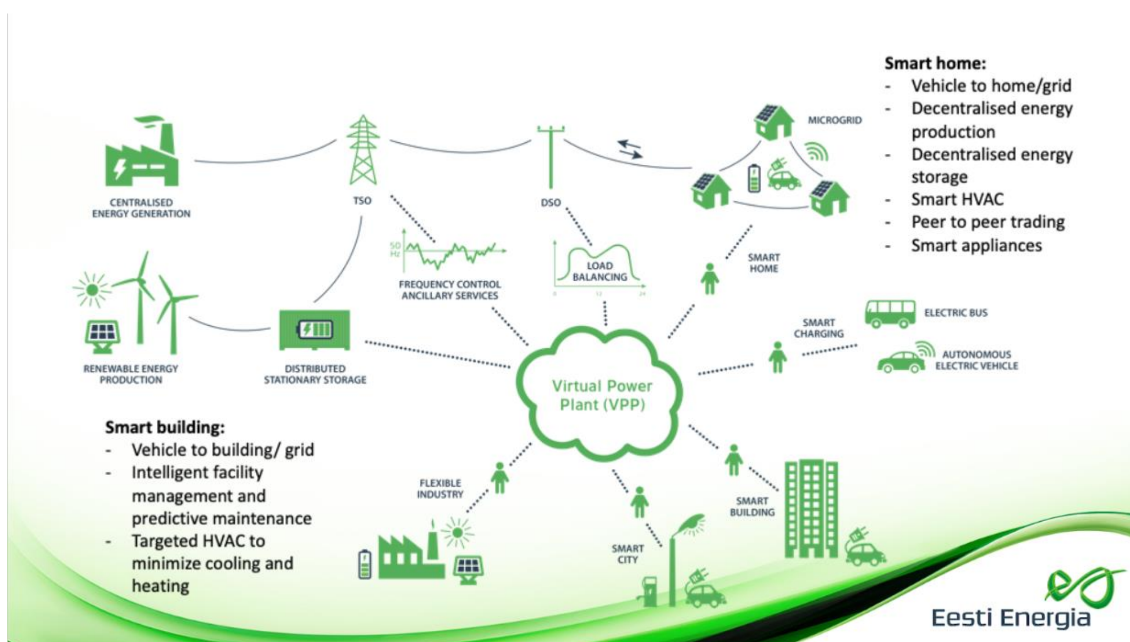
23. märts 2020 toimunud QCon konverentsi Tesla Virtual Power Plant esitluses öeldi, et elektrivõrgustik on üks suuremaid ja keerulisemaid masinaid, mis on iial ehitatud ning hetkeolukorras tuleb selle energia peamiselt 20. sajandi tsentraliseeritud fossiilkütuse baasil töötavatest elektrijaamadest. Arvestades kui palju on tekkinud alternatiivseid taastuvelektrienergia tootmisviise, pole need siiski piisavad, et asendada täielikult fossiilkütuste kasutamist energia tootmiseks, kui soovitakse säilitada hetkelist energianõudlust. Üheks võimaluseks on elektrienergia salvestamine tuhandete tarbijate kodustes patareides, et luua virtuaalseid elektrijaamu(ingl *virtual power plants*) mis pakuvad väärtust nii energia tootjale kui ka tarbijale [51]. Olukorras, kus elektriautode tootmine kasvab ning turule kui ka liiklusesse tuleb juurde aina võimsamate akudega elektriautosid, võimaldavad need samad elektriautod olla kas osa või täielikult asendada koduseid akusid. 2020. aasta veebruaris ilmunud ERR-i artiklis "Elektriautode müük võib tänavu kasvada mitu korda" on toodud välja, et maanteeameti registris oli jaanuari lõpu seisuga 1361 elektriautot ning kui eelmisel ja üle-eelmisel aastal oli elektriautode müük Eestis alla saja elektriauto, siis tänavu esitati aasta algul jõustunud riigitoetuse tagajärjel esimese nelja tunniga taotlusi 240 auto ostuks [52]. Praeguse seisuga moodustab elektriautode arv Eestis 0,2% kogu riigis arvel olevatest sõidukitest. Elektriautode nõudluse kasvavat trendi arvesse võttes, tulebki Eesti Energial vaadata tuleviku suunas ning mõelda olukorrale, kus elektriautode osa kõikidest sõidukitest tõuseb kahekohalise protsendini nagu Norras saavutatud 10% aastal 2018 [53]. Elektriautode arvu kasvu ja

selle mõju taristule, kui ka autode laadimisvajaduse testimiseks ongi loodud antud simulaator.

Kuna kõik uued elektriautod kasutavad innovatiivseid ja võimsaid akusid, mis suudavad minimaalse kaoga elektrienergiat talletada [54], on elektriautod nö. ideaalsed patareid. Kasutades Eesti Energia poolt loodud V2G laadijavõrgustikku saavad nad elektrivõrgustikult elektrienergiat osta ja võimalusel seda ka tagasi müüa, kus tulebki mängu väärtuse loomine nii energia tootjale kui ka tarbijale - tootja ei pea enam nii suures mahus energiat tootma kui varem ja tarbijad saavad endale vajalikku energiat tarbida odavama kuluga. Tuleb siiski mainida, et hetkel on see vaid spekulatsioon ning tiimi ülesanne ongi luua platvorm, et seda testida.

Kasvava CO2 heitmekvoodi hinna ning elektriautode nõudluse juures tuleb Eesti Energial kui riigi omandis oleva ettevõttena nii elektri tootja, varustuskindluse tagaja kui ka võrguoperaatorina mõelda uute ja innovatiivsete lahenduste poole, sealjuures tekibki vajadus saavutada võimalikult efektiivne tasakaal elektritootmise ja tarbimise vahel ning see projekt mängib selles väikest, kuid tähtsat rolli.

Allpool on näha skeem Eesti Energia plaanist luua detsentraliseeritud energiajagamise platvorm virtuaalse elektrijaama näol. Autorite poolt arendatud osa sellest on nutikas laadimine(ingl *smart charging*) (vt skeem 4.2.1.1).



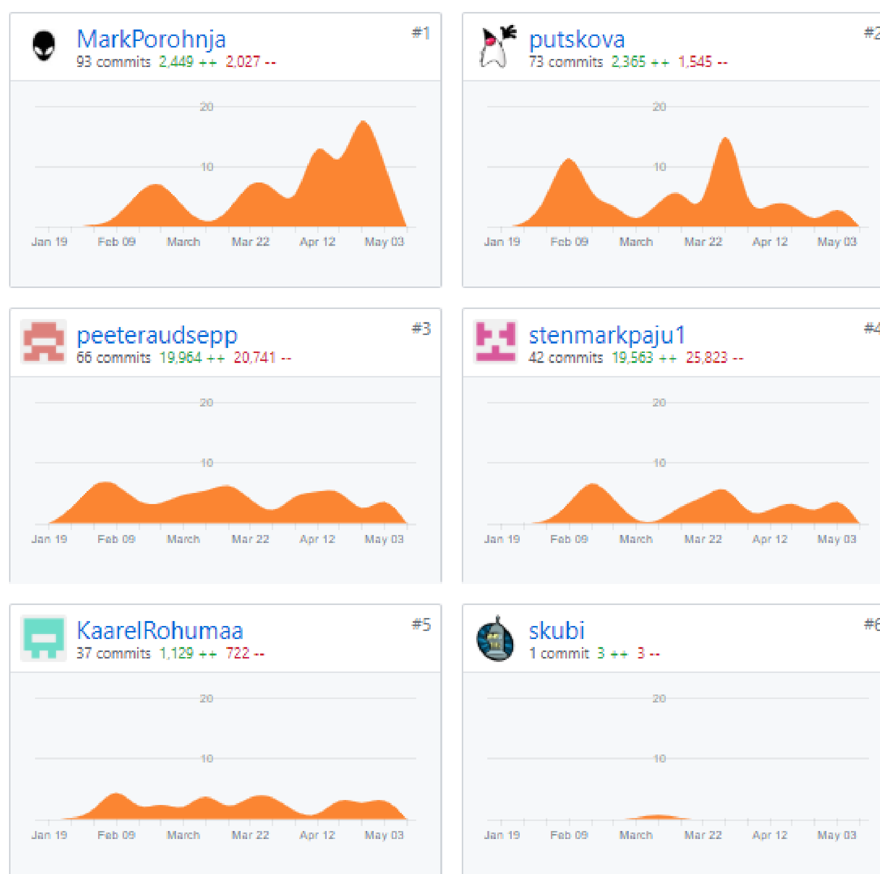
Skeem 4.2.1.1. Virtuaalne elektrijaam

4.4 Teostatud tööde kirjeldus

Eesti Energia ja arendajate vahelise lepingu järgi oli tiimil kaks kindlat tööpäeva nädalas - neljapäev ja reede. Projekt on antud hetkeks kestnud 14 nädalat. Sellest selgub, et tööle ettevõttes kulus igal tiimi liikmel 224 tundi. Ülejäänud aja jooksul töötasid autorid ka rakenduse kallal kodust ning kirjutasid projekti aruannet.

4.4.1 Giti commit'ide väljavõte

Kuvatõmmises 4.3.1.1 on välja toodud graafikud Giti *commit*'ide väljavõtetega iga tiimiliikme kohta aastal 2020.



Kuvatõmmis 4.3.1.1 GitHubist *commit*-de sageduste väljavõte

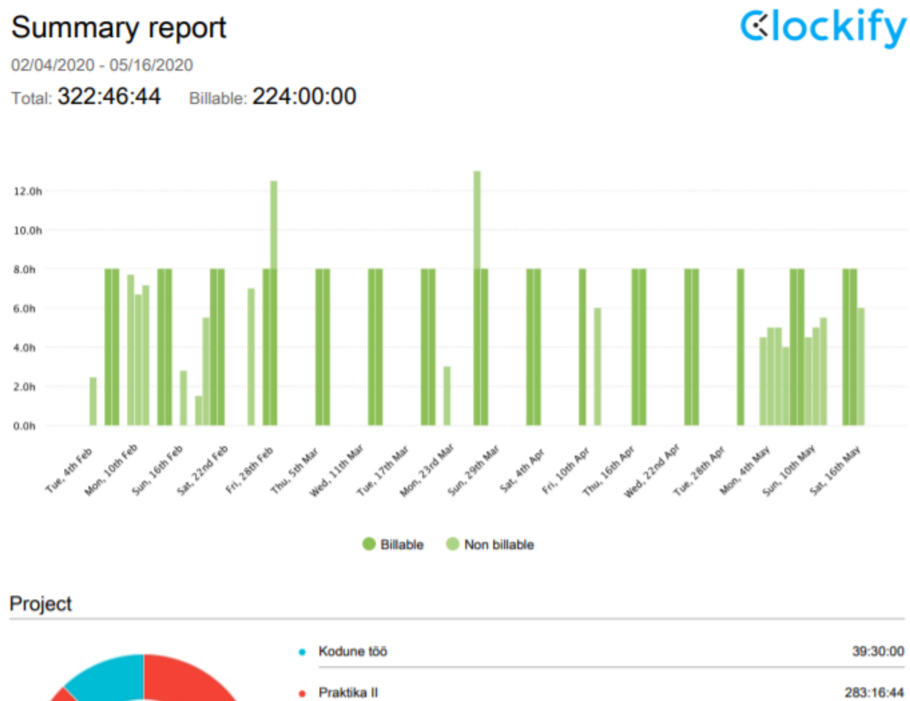
4.4.2 Clockify rakendusest väljavõte

Järgnevalt on välja toodud iga projektimeeskonna liikme kohta personaalne logi. Clockify rakendusest võetud graafikute ja rõngasdiagrammide kaudu on näha iga liikme poolt projektile kulutatud aeg.

Tatjana Putškova:

Käesolev aruanne näitab Tatjana Putškova ajakulu projektile vahemikus 6. veebruar - 16. mai.

Projekti arendamisele Eesti Energias kulus umbes 224 tundi, iseseisvaks tööks umbes 59 tundi ja aruande kirjutamiseks umbes 39 tundi(vt kuvatõmmis 4.3.2.1).



Kuvatõmmis 4.3.2.1 Tatjana Putskova tundide väljavõte Clockify rakendusest

Peeter Raudsepp:

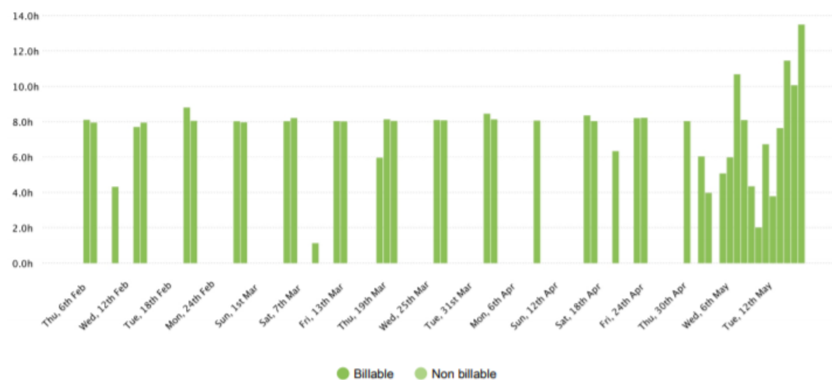
Käesolev aruanne näitab Peeter Raudsepa ajakulu projektile vahemikus 6. veebruar - 16. mai.

Projekti arendamisele Eesti Energias kulus umbes 226 tundi, iseseisvaks tööks umbes 34 tundi ja aruande kirjutamiseks umbes 51 tundi(vt kuvatõmmis 4.3.2.2).

Summary report

02/06/2020 - 05/16/2020

Total: 311:29:00 Billable: 311:29:00 Amount: 0.00 USD



Billable Non billable

Project

Lõputöö kirjutamine	51:01:30
Kodune töö II	33:41:17
Praktika II	226:46:13

Kuvatõmmis 4.3.2.2 Peeter Raudsepa tundide väljavõte Clockify rakendusest

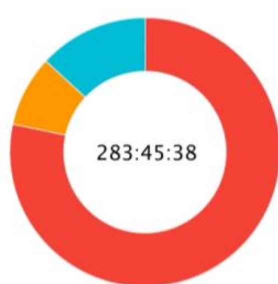
Kaarel Rohumaa:

Käesolev aruanne näitab Kaarel Rohumaa ajakulu projektile vahemikus 6. veebruar - 16. mai.

Projekti arendamisele Eesti Energias kulus umbes 222 tundi, iseseisvaks tööks umbes 24 tundi ja aruande kirjutamiseks umbes 37 tundi(vt kuvatõmmis 4.3.2.3).

Total: 283:45:38 Billable: 283:45:38

Project



Lõputöö kirjutamine	37:23:36
Kodune töö II	24:07:59
Praktika II	222:14:03

Kuvatõmmis 4.3.2.3 Kaarel Rohumaa tundide väljavõte Clockify rakendusest

Mark Porohnja:

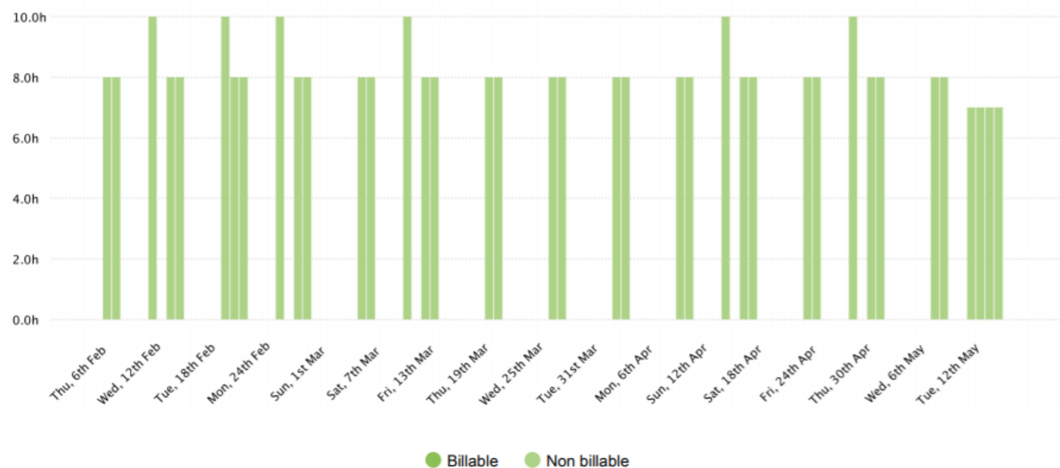
Käesolev aruanne näitab Mark Porohnja ajakulu projektile vahemikus 6. veebruar - 16. mai.

Projekti arendamisele Eesti Energias kulus 224 tundi, iseseisvaks tööks 60 tundi ja aruande kirjutamiseks umbes 28 tundi(vt kuvatõmmis 4.3.2.4).

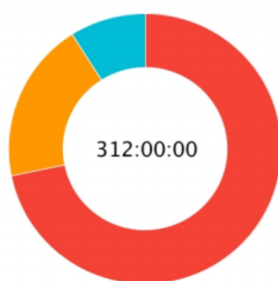
Summary report

06-02-2020 - 16-05-2020

Total: 312:00:00 Billable: 00:00:00



Project



Lõputöö kirjutamine	28:00:00
Kodune töö II	60:00:00
Praktika II	224:00:00

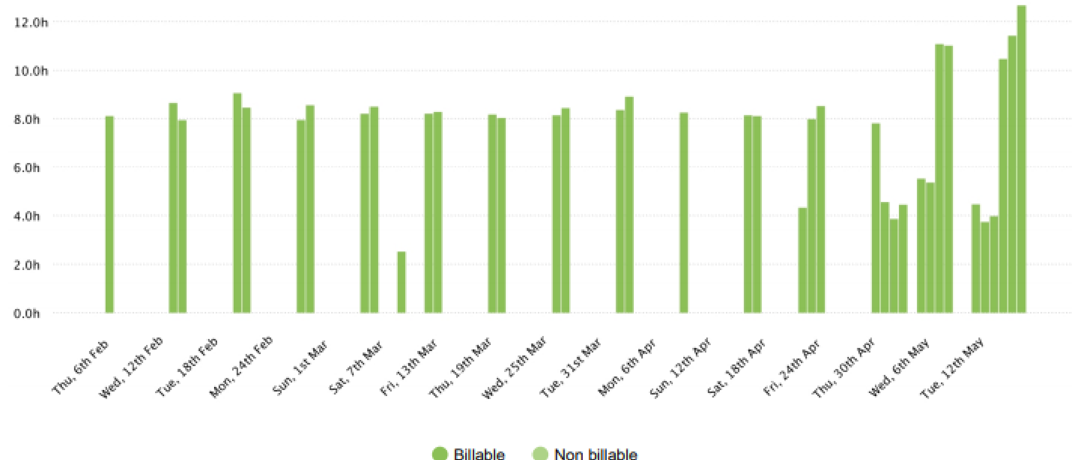
Kuvatõmmis 4.3.2.4 Mark Porohnja tundide väljavõte Clockify rakendusest

Sten-Mark Paju:

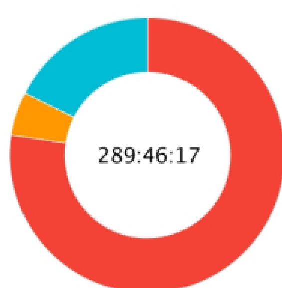
Käesolev aruanne näitab Sten-Mark Paju ajakulu projektile vahemikus 6. veebruar - 16. mai.

Projekti arendamisele Eesti Energias kulus umbes 224 tundi, iseseisvaks tööks umbes 15 tundi ja aruande kirjutamiseks umbes 51 tundi(vt kuvatõmmis 4.3.2.5).

Total: 289:46:17 Billable: 289:46:17



Project



Lõputöö kirjutamine	50:57:27
Kodune töö II	14:51:09
Praktika II	223:57:41

Kuvatõmmis 4.3.2.5. Sten-Mark Paju tundide väljavõte Clockify rakendusest

4.5 Hinnang projekti teostamise protsessi kohta

4.5.1 Projekti juhtimine ning projekti teostamise protsess

Töö algas veebruari alguses ning kestab juuni lõpuni, mis tähendab, et kirjutamise hetkel ei ole projekti arendus veel lõppenud ning aruanne on koostatud seni tehtud töö kohta.

Küsimustega, mis puudutasid valmis kirjutatud kasutajalugusi ja tehnilisi ülesandeid, aitasid autoreid arhitekt Aleksandr Skubi ja tooteomanik Jaan Ots. Koodi ja arendamisega seotud probleemide lahendamiseks said arendajad Eesti Energia poolt kaks juhendajat: Allan Juhanson ja Osama Muhamed. Seoses pikaaegsema haigestumisega asendati Allan Juhanson märtsis Vinay Puraniku vastu. Igal tööpäeval toimuvate püstijala koosolekude ja retrospektiivide sujumise eest vastutas *Scrum* meister Ruslan Bjurkland.

Ülikooli poolsed juhandajad olid Kristina Murtazin ja Jekaterina Tšukrejeva.

Projekti paremaks haldamiseks ja meeskonnatöö optimiseerimiseks kasutati Githubi. Iga arendusetapi alguses arutati ülesanded läbi ja jagati need omavahel ära. Kui arendusetapis oli rohkem ülesandeid kui viis, hakkasid arendajad, kes oma ülesande varem valmis said, mõne uue järelejäänud ülesandega tegelema. Kui alustamata ülesandeid enam polnud, aga arendusetapp polnud läbi saanud, aidati üksteisel antud kasutajalugu, tehniline ülesanne või arendusprotsessi käigus tekkinud vigade parandamine ära lõpetada, kuna arendusprotsessi jaoks oli väga tähtis asjadega arendusetapi lõpuks valmis saada, et tööd ei kuhjuks.

Alates märtsi alguses kehtestatud eriolukorra tõttu tavapärast kontoritööd ei jätkatud vaid tehti tööd kodus. Seetõttu muutusid rakendused nagu Skype ja Slack omavahelise suhtluse jaoks hädavajalikuks. Tihtipeale olid arendajad kogu tööpäeva Skype'i kõnes ning Slack'is saadaval ja aitasid üksteist probleemide korral.

Iga meeskonnaliikme panus projekti on välja toodud lisade osas.

4.5.2 Projekti kitsaskohad ja kordaminekud

Projekti arendus algas väga sujuvalt, kuna jätkati aine ITB1706 - Meeskonnaprojekt aine käigus alustatud töö edasi arendamisega ja tiimiliikmete oskused olid sellest ajast kõvasti arenenud. Samuti võib võrreldes varasemaga vägagi positiivseks osaks välja tuua tiimi väiksema koosseisu, mistõttu pidid seekord arendajad rohkem iseseisvalt tööd tegema ning seetõttu arenesid tiimiliikmed individuaalselt kiiremini. Kuigi tehti rohkem iseseisvalt tööd, viis see tiimi omavahelise suhtluse ning ülesannete valmimiskiiruse uuele tasemele, kuna oli vähem segavaid faktoreid ning asjad liikusid edasi palju konstruktiivsemalt.

Kuigi alguses oli riigis kehtestatud eriolukorra tõttu töö natuke aeglasem, suudeti olukorraga kiiresti kohaneda ning lõppude lõpuks töö ja omavaheline suhtlus sellest ei kannatanud.

Asi, mida täpselt ei oska hinnata kas ta on positiivne või negatiivne, on ülesannete mitmekülgsus, mis täpsemalt tähendab seda, et iga arendaja tegeles erineva asjaga ning arenes selles valdkonnas rohkem kui teised. Näiteks võib tuua, et mõni arendaja tegeles enamjaolt ainult serveripoolne ja mõni kliendipoolne osa arendamisega. Kuigi selline on tõsiasi, saavad arendajad aru, et see käibki suurema projekti arendusprotsessi juurde. Et

olukorda parandada, otsustasid arendajad kolmanda tsükli retrospektiivis alustada koodi ülevaadetega (ingl *code review*), mille käigus seletatakse üksteisele detailselt oma kirjutatud koodi ning selle funktsionaalsust, et kõik koodist aru ning vajadusel lihtsamalt koodi täiendada saaksid.

Mõnevõrra negatiivse asjana võib välja tuua keelebarjääri arendajate kui ka mentorite vahel, mis mõnikord aeglustas üksteisega suhtlust, kuid sellegipoolest suudeti asjad alati omavahel läbi arutada.

4.5.3 Hinnang üldisele projekti teostamise protsessile

Meeskonna hinnang üldisele projekti teostamise protsessile on väga hea. Meeskonnaliikmed on väga rahul, et said antud projektist osa võtta ning võtavad sellest projektist kaasa palju positiivseid kogemusi. Samuti on kõik tiimiliikmed üksteise tööga väga rahul, tiimitöö oli sujuv ning tiimis oli meeldiv töötada.

Ülikooli poolsed juhendajad andsid meeskonnale palju nõu ja näpunäiteid lõputöö kirjutamise ning vormistamise osas ning andsid meile võimalikult palju konstruktiivset tagasisidet. Lisaks sellele aitasid nad autoritel kinni pidada tähtaegadest.

Eesti Energia poolt projekti kaasatud inimesed vastasid väga kiiresti tekkinud küsimustele ning jagasid alati kasulikke kommentaare ja märkmeid rakenduse paremaks arendamiseks, meeskond on väga rahul ettevõtte poolsest toetusega. Ka eriolukorra ajal olid mentorid alati Skype'i või Slacki kaudu kättesaadavad. Nende abi ja toetus oli väga kasulik. Positiivse osana võib välja tuua ka igal tööpäeval toimunud püstijala koosolekud ja arendustsükli lõpus toimunud retrospektiivid, mida korraldas projekti *Scrum* meister Ruslan.

4.5.4 Meeskondlik hinnang meeskonnaliikmete panuse kohta

Meeskond jõudis ühisele hinnangule, et kõik panustasid võrdselt (0).

5 Kokkuvõte

Antud lõpuprojekt valmis Eesti Energia tütarettevõtte Enefiti ja viie TalTechi äriinfotehnoloogia tudengi koostööl.

Käesoleva projekti lõppeesmärk oli luua realistlik simulatsioon lähituleviku Eestist, kus on tuhandeid elektriautosid, mis kasutavad Eesti Energia poolt välja töötatud V2G nutikat laadijat, mille põhjal genereerida andmeid sellise olukorra mõjust nii elektrivõrgustikule kui ka turule. Antud simulatsiooni loomine on väga tähtis, sest hetkel pole võimalik sellist olukorda füüsiliselt elektriautode vähesuse tõttu testida.

Lõpuprojekt loodi kasutades agiilseid arendusmetoodikaid kasutades REST API ja kaardirakenduse kujul. REST API arendamiseks kasutati põhiliselt Spring raamistikku ning kasutajaliidese arendamiseks SPA [55] arhitektuuri, React raamistikku ning Leaflet teeki. Enamik serveripoolse osa genereeritud infot kujutatakse kaardirakendusel. Projekti arendus oli jagatud viite erinevasse etappi, mis koosnesid erinevatest kasutajalugudest ja tehnilistest ülesannetest. Projekti haldamiseks ja meeskonnatöö optimeerimiseks kasutati GitHubi, mille abil sai kasutajalugude ja tehniliste ülesannete eraldi arendamiseks kasutada erinevaid harusid.

Dokumentatsiooni kirjutamise hetkeks pole projekt veel täielikult läbi saanud ning selle arendamisprotsess läheb jätkub antud tiimiga kuni 2020 aasta juuni lõpuni. Suurem osa nõuetest said valmis ehk hetkeseisuga on arendajatel valmis saanud serveri ja kasutajaliidese projektide osas nelja etapi jagu kasutajalugusi ja tehnilisi ülesandeid. Loodud on üheleherakenduse arhitektuuri järgi dünaamiline kaardirakendus, millel on võimalik kujutada elektrisõidukeid, sõidukitega seotud kodu- ja tööasukohti, teekondasid kodu ja töö asukohtade vahel ning avalikke elektrisõidukite laadimispunkte. Autod saavad sõita kodu ja töökoha vahet, nende infot saab lähtestada ning samuti uuesti genereerida. Peale selle on ka võimalik elektriautode akude laadimise ja tühjenemise režiimi vahetada. Tulevikus on plaan luua sõidukitele võimalus kasutada avalikke laadimispunkte ning kasutajaprofiilid, mille läbi on võimalik sõidukite käitumist automatiseerida ning andmeid genereerima hakata.

Käesolev projekt on andnud võimaluse viiele projektis osalenud tudengile head töökogemust reaalses töö ja arenduskeskkonnas saada ning koguda palju uusi kogemusi

raamistikega, millega varem polnud kokku puutunud. Samuti saadi kogemusi agiilse töökeskkonna protsesside ja selle sees töötamise kohta. Uute raamistike kasutama õppimine ja agiilne töökeskkond andsid tudengitele juurde palju väärtuslike teadmisi ja kogemusi, mida koolis praktikas ei õpetata. Nendele lisaks avardas projektist osavõtmine ka tudengite silmaringi energiasektori kohta, millest kindlasti tulevikus kasu võib saada.

Kasutatud allikad

[1] „Peer to peer solar energy trading guide,“ Energymatters, [Võrgumaterjal]. Available: https://www.energymatters.com.au/misc/peer-to-peer-solar-energy-trading-guide/. [Kasutatud 18 05 2020].

[2] „Vehicle-to-Grid,“ Virta, [Võrgumaterjal]. Available: https://www.virta.global/vehicle-to-grid-v2g. [Kasutatud 18 05 2020].

[3] B. Marr, „What Is Digital Twin Technology And Why Is It So Important,“ Forbes, 2017. [Võrgumaterjal]. Available: https://www.forbes.com/sites/bernardmarr/2017/03/06/what-is-digital-twin-technology-and-why-is-it-so-important/#1812d0282e2a. [Kasutatud 15 05 2020].

[4] „State Machine,“ Techopedia, 17 11 2015. [Võrgumaterjal]. Available: https://www.techopedia.com/definition/16447/state-machine. [Kasutatud 18 05 2020].

[5] „What is Rest,“ Codeacademy, [Võrgumaterjal]. Available: https://www.codecademy.com/articles/what-is-rest. [Kasutatud 18 05 2020].

[6] „What is a Container,“ Open Source, [Võrgumaterjal]. Available: https://www.docker.com/resources/what-container. [Kasutatud 18 05 2020].

[7] „Compose file versions and upgrading,“ Docker, [Võrgumaterjal]. Available: https://docs.docker.com/compose/compose-file/compose-versioning/. [Kasutatud 18 05 2020].

[8] M. Rehkopf, „Continuous Delivery/Continuous Integration,“ Atlassian, [Võrgumaterjal]. Available: https://www.atlassian.com/continuous-delivery/continuous-integration. [Kasutatud 18 05 2020].. [Kasutatud 18 05 2020].

[9] „Actions,“ GitHub, [Võrgumaterjal]. Available: https://github.com/features/actions. [Kasutatud 18 05 2020].

[10] V. Beal, „Object Oriented Programming,“ Webopedia, [Võrgumaterjal]. Available: https://www.webopedia.com/TERM/O/object_oriented_programming_OOP.html. [Kasutatud 18 05 2020].

[11] „Spring Dependency Injection,“ Baeldung, 18 02 2020. [Võrgumaterjal]. Available: https://www.baeldung.com/spring-dependency-injection. [Kasutatud 18 05 2020].

[12] B. Tasdemir, „"Event Driven Microservice Architecture",“ Medium, 20 10 2019. [Võrgumaterjal]. Available: https://medium.com/trendyol-tech/event-driven-microservice-architecture-91f80ceaa21e. [Kasutatud 18 05 2020].

[13] „SonarQube Documentation,“ SonarQube , [Võrgumaterjal]. Available: https://docs.sonarqube.org/latest/. [Kasutatud 18 05 2020].

[14] Mkyong, „Java Read A File From Resources Folder,“ Mkyong, [Võrgumaterjal]. Available: https://mkyong.com/java/java-read-a-file-from-resources-folder/. [Kasutatud 18 05 2020].

[15] „Java,“ TechTerms, 19 04 2012. [Võrgumaterjal]. Available: https://techterms.com/definition/java. [Kasutatud 18 05 2020].

[16] M. Mulders, „What Is Spring Boot,“ Stackify, 16 09 2019. [Võrgumaterjal]. Available: https://stackify.com/what-is-spring-boot/. [Kasutatud 18 05 2020].

[17] N. Pandit, „C# Corner," 05 03 2020. [Võrgumaterjal]. Available: https://www.c-sharpcorner.com/article/what-and-why-reactjs/. [Kasutatud 18 05 2020].

[18] „"User Stories"," Mountain Goat Software, [Võrgumaterjal]. Available: https://www.mountaingoatsoftware.com/agile/user-stories. [Kasutatud 18 05 2020].

[19] „Git Commit," Atlassian, [Võrgumaterjal]. Available: https://www.atlassian.com/git/tutorials/saving-changes/git-commit. [Kasutatud 18 05 2020].

[20] „What is an API," Rapidapi, [Võrgumaterjal]. Available: https://rapidapi.com/blog/api-glossary/api/. [Kasutatud 18 05 2020].

[21] „OpenWeatherMap API guide," Openweathermap, [Võrgumaterjal]. Available: https://openweathermap.org/guide. [Kasutatud 2020 05 18].

[22] „Dockerfile reference," Docker, [Võrgumaterjal]. Available: https://docs.docker.com/engine/reference/builder/. [Kasutatud 18õ 05 2020].

[23] „Docker Image," Search ITOperations Tech Target, [Võrgumaterjal]. Available: https://searchitoperations.techtarget.com/definition/Docker-image. [Kasutatud 18 05 2020].

[24] „What Is GitHub, and What Is It Used For," Howtogeek, [Võrgumaterjal]. Available: https://www.howtogeek.com/180167/htg-explains-what-is-github-and-what-do-geeks-use-it-for/. [Kasutatud 18 05 2020].

[25] V. Driessen, „A successful Git branching model," Nvie, 2010. [Võrgumaterjal]. Available: https://nvie.com/posts/a-successful-git-branching-model/.

[26] „What is Slack," Slack, [Võrgumaterjal]. Available: https://slack.com/intl/en-ee/help/articles/115004071768-What-is-Slack-. [Kasutatud 18 05 2020].

[27] „IntellIJ IDEA," Techopedia, [Võrgumaterjal]. Available: https://www.techopedia.com/definition/7755/intellij-idea. [Kasutatud 18 05 2020].

[28] „REST Architectural Constraints," restfulapi, [Võrgumaterjal]. Available: https://restfulapi.net/rest-architectural-constraints/. [Kasutatud 18 05 2020].

[29] „Spring Framework," Spring, [Võrgumaterjal]. Available: https://spring.io/projects/spring-framework. [Kasutatud 18 05 2020].

[30] „What is Gradle," Gradle, [Võrgumaterjal]. Available: https://docs.gradle.org/current/userguide/what_is_gradle.html. [Kasutatud 18 05 2020].

[31] V. Kasegaonkar, „Project Lombok," Dzone, 18 12 2018. [Võrgumaterjal]. Available: https://dzone.com/articles/project-lombok-boilerplate-code-reducer. [Kasutatud 18 05 2020].

[32] „About Openstreetmap," Openstreetmap, [Võrgumaterjal]. Available: https://www.openstreetmap.org/about. [Kasutatud 18 05 2020].

[33] „Openrouteservice Services," Openroute, [Võrgumaterjal]. Available: https://openrouteservice.org/services/. [Kasutatud 18 05 2020].

[34] „What is Google Maps," GFCGlobal, [Võrgumaterjal]. Available: https://edu.gcfglobal.org/en/google-maps/what-is-google-maps/1/. [Kasutatud 18 05 2020].

[35] „About Scrum," ScrumAlliance, 11 2017. [Võrgumaterjal]. Available: https://www.scrumalliance.org/about-scrum/definition. [Kasutatud 18 05 2020].

[36] „What Is Kanban,“ Digite, [Võrgumaterjal]. Available: https://www.digite.com/kanban/what-is-kanban/#kanban-in-lean-agile. [Kasutatud 18 05 2020].

[37] A. Novkov, „Running A Better Stand Up Meeting,“ Kanbanize, 02 03 2019. [Võrgumaterjal]. Available: https://kanbanize.com/blog/running-a-better-stand-up-meeting/. [Kasutatud 18 05 2020].

[38] „What Is Scrum Master,“ Visual Paradigm, [Võrgumaterjal]. Available: https://www.visual-paradigm.com/scrum/what-is-scrum-master/. [Kasutatud 18 05 2020].

[39] „ Product Owner,“ Mountain Goat Software, [Võrgumaterjal]. Available: https://www.mountaingoatsoftware.com/agile/scrum/roles/product-owner. [Kasutatud 18 05 2020].

[40] M. Racasan, „What Is A Milestone In Agile Project Management,“ ZenHub, 06 08 2019. [Võrgumaterjal]. Available: https://www.zenhub.com/blog/what-is-a-milestone-in-agile-project-management/. [Kasutatud 18 05 2020].

[41] „Uniform Resource Locator Url,“ Techopedia, 30 03 2020. [Võrgumaterjal]. Available: https://www.techopedia.com/definition/1352/uniform-resource-locator-url. [Kasutatud 18 05 2020].

[42] „Eurostauto OÜ koostöös Elektritransport OÜ-ga avas kahesüsteemse avaliku elektriautode kiirlaadimispunkti,“ Eurost Auto, 18 12 2019. [Võrgumaterjal]. Available: https://www.eurostauto.ee/uudised-menu;news/1548/Eurostauto-O-koost-s-Elektritransport-O-ga-avas-kahes-steemse-avaliku-elektriautode-kiirlaadimispunkti. [Kasutatud 18 05 2020].

[43] „Charging With a Nissan Leaf-e or e-NV200,“ Fastned, 29 04 2020. [Võrgumaterjal]. Available: https://support.fastned.nl/hc/en-gb/articles/204784998-Charging-with-a-Nissan-Leaf-e-or-e-NV200. [Kasutatud 15 05 2020].

[44] P. Lima, „2018 Nissan Leaf battery real specs,“ PushEVs, 29 01 2018. [Võrgumaterjal]. Available: https://pushevs.com/2018/01/29/2018-nissan-leaf-battery-real-specs/. [Kasutatud 15 05 2020].

[45] „JSON,“ Unionpedia, [Võrgumaterjal]. Available: https://et.unionpedia.org/i/JSON. [Kasutatud 18 05 2020].

[46] M. Tyson, „Introduction to the Java Development Kit,“ Javaworld, 17 01 2020. [Võrgumaterjal]. Available: https://www.javaworld.com/article/3296360/what-is-the-jdk-introduction-to-the-java-development-kit.html. [Kasutatud 18 05 2020].

[47] A. Lerner, „Introduction to Promises,“ Newline, [Võrgumaterjal]. Available: https://www.newline.co/fullstack-react/30-days-of-react/day-15/. [Kasutatud 17 05 2020].

[48] „Wrapper classes in Java,“ Javatpoint, [Võrgumaterjal]. Available: https://www.javatpoint.com/wrapper-class-in-java. [Kasutatud 18 05 2020].

[49] „Cross-Origin Resource Sharing,“ Mozilla, 16 05 2020. [Võrgumaterjal]. Available: https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS. [Kasutatud 18 05 2020].

[50] T. Ollema, „Cross-Site Request Forgery,“ 2013. [Võrgumaterjal]. Available: https://wiki.itcollege.ee/index.php/CSRF_(Cross-Site_Request_Forgery). [Kasutatud 18 05 2020].

[51] P. Link ja C. Breck, „Tesla Virtual Power Plant,“ InfoQ, 23 03 2020. [Võrgumaterjal]. Available: https://www.infoq.com/presentations/tesla-vpp/. [Kasutatud 15 05 2020].

[52] M. Otsmaa, „Elektriautode müük võib tänavu kasvada mitu korda,“ ERR, 13 02 2020. [Võrgumaterjal]. Available: https://www.err.ee/1034584/elektriautode-muuk-voib-tanavu-kasvada-mitu-korda. [Kasutatud 15 05 2020].

[53] M. Kane, „10% Of Norway's Passenger Vehicles Are Plug Ins,“ InsideEVs, 07 11 2018. [Võrgumaterjal]. Available: https://insideevs.com/news/341060/10-of-norways-passenger-vehicles-are-plug-ins/. [Kasutatud 15 05 2020].

[54] K. Eljand, „Surge Towards Electric Cars in Europe,“ Mediumv, 16 10 2019. [Võrgumaterjal]. Available: https://medium.com/@EnefitIT/surge-towards-electric-cars-in-europe-c5604f68b86e. [Kasutatud 15 05 2020].

[55] Z. Gimon, „What is a Single Page Application,“ Huspi, [Võrgumaterjal]. Available: https://huspi.com/blog-open/definitive-guide-to-spa-why-do-we-need-single-page-applications. [Kasutatud 18 05 2020].

[56] „RapidAPI,“ [Võrgumaterjal]. Available: https://rapidapi.com/blog/api-glossary/api/. [Kasutatud 18 05 2020].

Lisa 1. Peeter Raudsepa enda panuse kirjeldus ja eneseanalüüs

Panustasin lõpuprojekti kasutajaliidese ja serveri poole kasutajalugude kui ka tehniliste ülesannete arendamisega. Nelja arendusetapi jooksul töötasin kokku viie kasutajaloo ja ühe tehnilise ülesande kallal.

Arendusprotsessi alustasin ilmateave kättesaamise kasutajalooga, mis jäi eelmisel semestril arendatud meeskonnaprojekti lõppemise ajaks poolikuks. Kuna kasutajaloo algne arendaja tiimiga lõpuprojekti raames ei jätkanud, jäi selle lõpetamine minu teha. Esimese asjana viisin ennast juba tehtud tööga kurssi ning kuigi antud ülesande edasi arendamine tundus esialgu üpris keeruline, oli see tegelikult üllatavalt lihtne. Kõige keerulisemaks ja ajakulukamaks osutus testide kirjutamine, sest antud testide kirjutamiseks tuli kasutada mulle seni tundmatut Mockito raamistikku ja Given-When-Then testide struktuuri.

Peale ilmateave kättesaamise kasutajaloo oli meeskonnaprojektist lõpetamata jäänud ka kasutajalugu kaardipõhine simulaator ning järgmisena asusingi koostöös Sten-Mark Pajuga seda valmis arendama. Siinkohal võib välja tuua, et ka edasised kasutajaliidese lahendused on suuremas osas meie koostööl valminud. Olin eelmisel semestril antud kasutajalooga ning selle arendamiseks kasutatava React raamistiku ja Leaflet teegiga tutvunud ja omajagu aega ka selle kasutajaloo valmimisele panustanud. Antud hetkeks oli serveri pool kasutajaliidesest kõvasti ees ning seetõttu oli antud kasutajaloo valmimiseks ka aja jooksul rohkem nõudeid juurde tekkinud. Sellel arendusprotsessi hetkel oli kaardipõhise simulaatori kasutajaloo valmis arendamine prioriteet number üks, sest üheks kogu lõpuprojekti suurimaks nõudeks oli, et kogu back-end'is loodud loogikat peab saama kaardirakenduse kaudu ka demonstreerida. Kuigi olin juba varasemalt umbes kuu aega Reacti ja Leafletiga tutvunud, ei valdanud ma siiski neid veel piisavalt hästi ja kasutajaliidese arendamine oli minu jaoks endiselt üpris keeruline. Väga suur osa ajast kulus internetist lahenduste näidete otsimisele ja sealt erinevate leitud asjade läbi katsetamisele. Tuleb mainida, et Leafleti kohta internetist peale tema enda dokumentatsiooni väga palju leida pole ning minu arvates pole nende dokumentatsioon eriti hästi ja selgelt arusaadavalt koostatud, mis kogu protsessi veelgi aeglustas. Sellest hoolimata hakkasin ma aga ajapikku kõigest paremini aru saama ning kasutajalugu sai

teisel arendusetapil kenasti valmis. Kasutajaloo lõpuks tundsin ma ennast juba kasutajaliidese arenduse poolel palju mugavamalt ning tunnen, et õppisin palju juurde.

Järgmisena alustasin tööd paralleelselt kahe kasutajaloo kallal. Nendeks olid sõiduki liikumine kasutajaliideses ja sõiduki liikumine serveripoolses osas. Mõlema kasutajaloo puhul pidi hakkama kaarti ja kaardi komponente dünaamiliselt muutma. Näiteks tuli panna auto liikuma ja auto teekonda vahetada. Kaardi dünaamiline muutmine osutus minule seni suurimaks katsumuseks. Nende kasutajalugude arendamisel tuli väga suures mahus erinevaid asju läbi katsetada, et midagi kasvõi natukenegi tööle saada. Kuigi osa lahendusest oli tööle saadud ja võis tunduda, et olen jõudnud lõpplahendusele lähedale, leidsin tihtipeale ennast jällegi tupikust ning tuli leida mingi muu lahendus. Kogu katsetamine tasus ennast lõpuks ära ja asi sai valmis ning tunnen, et mõlemad kasutajalood andsid mulle enim juurde kogemust Reacti ja Leafleti komponentide olekute manipuleerimisele. Asjadest arusaamine muutus mitu korda lihtsamaks kui lõpuprojekti alustades.

Minu uueks ülesandeks oli avalikud laadimispunktid kaardil kasutajaloo arendamine. See oli esimene kasutajalugu, mille kasutajaliidese poolel tegutsesin ma peamiselt üksi ning seda tehes tundsin ennast ka juba piisavalt mugavalt. Antud kasutajaloo puhul oli minu esimeseks ülesandeks katsetada, kas on võimalik saada läbi API päringu Enefit Volt teenusest kätte elektriautode laadimispunktide info ja neid meie loodud kaardirakendusel dünaamiliselt kuvada. Erinevatel põhjustel polnud info kättesaamine aga võimalik, mida oli tegelikult ka oodata. Seetõttu pidin leidma teise viisi, et laadimispunkte kaardil kuvada. Selle jaoks kopeerisin ma Enefit Volt veebilehelt nende API päringu vastuse ja lisasin selle eraldi failina meie projekti. Seejärel lõin ma serveripoolsesse projekti uue REST API teenuse, mille kaudu ma saan projekti lisatud failist laadimispunktide info välja filtreerida ja info API päringuna kaardirakendusse edasi saata. Kõige keerulisemaks osutus API päringu identne ülesehitus Enefit Volt päringuga, sest tulevikus plaanime siiski leida lahenduse selle teenuse kasutamiseks. Identne pidi see olema seetõttu, et siis ei pea kaardirakenduses enam lisamuudatusi sisse viima. Minu üllatuseks oli selle kasutajaloo juures kõige lihtsam osa kasutajaliides ehk laadimispunktide dünaamiline kuvamine kaardil ning üleüldse kulus mul selle kasutajaloo arendamiseks poole vähem aega kui olin arvestanud.

Neljandas arendamise etapis keskendus kogu tiim enamjaolt tehniliste ülesannete lahendamisele ning peale eelmise kasutajaloo lõpetamist jäi minu teha tehniline ülesanne Dockeri implementatsioon kasutajaliidese projektis. Antud juhul oli tegu minu esimese kokkupuutega Dockeri platvormiga. Dockeri dokumentatsioon on aga nii hästi valmistatud, et sellest aru saamiseks kulus mul väga vähe aega. Tegin endale selgeks täpselt selle, mida mul oma ülesande efektiivseimaks arendamiseks vaja läheb ning kogu arendus möödus väga kiirelt. Tunnen, et sain omale hea algbaasi Dockeri kasutamiseks ning tunnen, et need teadmised tulevad mulle tulevikus oma erialal palju kasuks.

Kogu lõpuprojekti suurimaks katsumuseks oli Reactist ja Leafletist aru saamine ja nende kasutama õppimine. Üleüldse kogu kasutajaliidese arendamise valdkond oli minu jaoks suhteliselt tundmatu ning minu kogemus sellega oli peaaegu minimaalne. Olen arvamusel, et äriinfotehnoloogia õppekaval peaks olema selle jaoks kindlasti üks baaskursus. Päris mitu korda ilmus tõsiasi, et pärast mõne asja igat viisi läbi proovimist piisas lõpuks ainult mõnest koodireast, et asi tööle saada ning tunnen, et kui oleksin varasemalt osa võtnud mõnest seda teemat õpetavast õppeainest või koolitusest, oleksin säästnud mitukümmend tundi, mis kulusid iseseisvalt asjade katsetamisele ja lahenduste välja nuputamisele. Sellegipoolest olen rahul, et selle protsessi läbi tegin ning tunnen, et sain tohutul hulgal kogemust just selles valdkonnas juurde. Tegelikult on minu arvates veebirakenduse kasutajaliidese arendamine ka palju rahuldust pakkuvam protsess kui serveripoolne algoritmine arendamine, sest oma kirjutatud koodi tulemusi on koheselt veebirakenduses näha. Kui projekti alguses eelistasin rohkem Java arendust, siis nüüd eelistaksin seda vähem. Seetõttu olen projektiga igati rahul, sest leidsin enda jaoks midagi uut ja huvitavat.

Üldiselt olen projektiga väga rahul, tiimi ja Eesti Energia juhendajate näol oli tegu mõnusa kollektiiviga ning sain näha, kuidas toimub suuremas ettevõttes agiilne projekti arendus. Kindlasti tegin ma tarkvaraarendajana ja tiimiliikmena väga suure enesearengu ja sain suurel hulgal kogemust juurde, mis aitab mul tulevikus arendajana tööd leida. Tulevikus soovitan ma jätkata taoliste ülikooli ja ettevõtte vaheliste projektidega, sest need tulevad minu arvates kõikidele osapooltele väga palju kasuks. Antud projekt avardas minu silmaringi ning äratas minus huvi energiasektori vastu. Samuti olen õnnelik, et sain osa võtta projektist, millest Eesti ühiskond ning Eesti loodus tulevikus kasu võivad lõigata.

Lisa 2. Kaarel Rohumaa enda panuse kirjeldus ja eneseanalüüs

Täitsin projekti arendamises suures osas serveri poolse arendaja rolli, kuid sain põgusalt tutvuda ka kasutajaliideses kasutatavate tehnoloogiatega. Tegelesin kolme kasutajaloo, ühe tehnilise ülesande ja mitmete arendamise käigus tekkinud vigade kõrvaldamisega.

Esimese arendusetappi jooksul töötasin sõidukid liiguvad kasutades reaalseid marsruute kasutajalooga. Alustasin tegelikult selle looga lõputööle eelnenud meeskonnaprojekti jooksul, kuid ei jõudnud seda lõpuni viia. Meeskonnaprojekti vältel sai marsruudi küsimiseks valmis väga algeline lahendus, mis oli jäänud testimata. Esimese asjana kirjutasin teste loodud teenuse ja kontrolleri jaoks. Peale seda aitasin Tatjanal kirjutada teste tema poolt loodud loogikale. Kuigi kasutajaloos defineeritud nõuded olid saavutatud, ei olnud mentorid koodiga rahul ja lõid kaks uut probleemi ülesannet.

Teist arendusetappi alustasin eelmisel etapil tekkinud vigade parandamisega. Loodud marsruudi pärimise teenus oli vaja muuta konfigureeritavaks, päringud ja vastused tuli luua lihtsate Java objektidena. Peale mentoritega rääkimist oli ülesanne selge ja parandused said kahe päevaga tehtud. Vahepeal oli tekkinud veel üks ülesanne, mis vajas kiirelt lahendust. Nimelt jooksis rakendus kokku kui seda jooksutada kaustast, mille nimes oli tühik. Selle probleemi põhjustasid sõned, mida kasutati .properties failide asukohtadele viitamiseks. Probleemi lahendamiseks tuli sõned asendada URI'dega. Lahendus oli kiire ja lihtne, mistõttu võtsin järgmise kasutajaloo, millega töötasin kolmanda arendustsükli lõpuni.

Kolmandas etapis jätkasin tööd sõiduki liikumine serveripoolses osas, kus tuli luua muuta auto loogikat, et peatumisel genereeritakse uus marsruut peatumispunktist sihtkohani. Lisaks sellele tuli salvestada seni läbitud teekond. Selle kasutajaloo arendamiseks tuli mulle appi Peeter. Uue marsruudi genereerimisega raskuseid polnud, aga läbitud teekonna salvestamisega tekkisid mitmed probleemid, mille põhjust ei suudetud tuvastada. Näiteks hakkas läbitud teepikkus eksponentsiaalselt kasvama. Lõpuks otsustasime pöörduda tagasi lahenduse juurde, mis polnud väga täpne, kuid töötas. Parajat peavalu tekitas ka selle lahenduse testimine, sest Mockito raamistik on siiani võõras. Peale testimist sain väikese ülevaate ka kasutajaliidese tehnoloogiatest, kui tuli

salvestatud teepikkust autole vajutades kuvada. Antud kasutajalugu oli minu jaoks kindlasti kõige mahukam ja huvitavam, sest see puudutas rakendust igast võimalikust küljest. Selles etapis aitasin veel Stenil arendada auto lähtestamise ja uuesti genereerimise loogikat. Mõlema tegevuse jaoks vajalikud meetodid valmisid kiiresti, sest uuesti genereerimiseks oli vaja kustutada auto omaduste väärtused. Genereerimise loogika oli juba varem loodud. Auto lähtestamist sai teha sisse ehitatud setter meetodite abil.

Neljandas etapis sain lahendada tehnilist ülesannet pidevintegratsiooni ja -valmiduse konveieri implementatsioon. Antud ülesande kirjeldus oli üpris lakooniline ja seetõttu alustasin Github Actions'i poolt loodud näidistega tutvumisega ning sarnase lahenduse loomisega. Mentoritele ülevaatusel tuli välja, et selline lahendus ei sobinud ja tuli nullist alustada. Lisaks tuli selgeks teha Docker'i põhilised tehnoloogiad. Põhiline aeg kulus teiste inimeste poolt loodud tegevuste arusaamisele ja implementeerimisele. Kiiret arendust takistas aga lukustatud peaharu, kuhu koodi laadimiseks oli vaja mentorite poolset ülevaatust. Seetõttu tuli isegi väikeste muutuste jaoks teha uus tõmbe nõudlus. Lõpuks saadi katse eksitus meetodil konveier toimima, mille käigus ilmnes, et Dockerfile'id vajavad ümber kirjutamist.

Aasta aega Eesti Energias on avardanud minu silmaringi energiavaldkonnas. Tutvumine uuenduslike tehnoloogiatega nagu V2G on pannud rohkem mõtlema maailma ja Eesti tulevikule. Arendamise poole pealt olen tänulik, et sain kogeda tööd suures ettevõttes. Omamoodi panustas sellele kogemusele ka üleriigiline eriolukord, mille tõttu on see praktika sedavõrd erilisem. Olen projektis osalenud tiimidega väga rahul, nii Eesti Energia poolse kui ka teiste arendajatega. Enda arvates olen Eesti Energiast saanud väga hea põhja edasisteks seiklusteks arendusvaldkonnas.

Lisa 3. Tatjana Putškova enda panuse kirjeldus ja eneseanalüüs

Antud projektis ma osalesin *full-stack* arendaja rollis. Olen tegelenud antud projektis nii *back-end* kui ka *front-end*'iga. Ikkagi rohkem olin keskendunud *back-end*'i poolele. Arendasin ise kokku kolm kasutajalood ning ühe kasutajaloo arendasin teise tiimliikmega koos. Nende kasutajalugude jaoks arendasin ka automatteste. Neljast kasutajaloost kolm olen arendanud *back-end* poolel ning ainult üks neist koosnes mõlemast poolest. Samuti tööprotsessi käigus olen parandanud mitu korda rakenduses avastatud vigu.

Projekti algusest kuni 19 veebruarini tegelesin *Vehicles Driving on Routes* kasutajalooga. Antud kasutajalugu oli keeruline ja koosnes mitmest osast, seetõttu me jagasime seda kaheks ülesandeks. Kaarel Rohumaa tegeles OpenRoute teenusest marsruudi teave saamisega. Selle andmete põhjal mina arendasin marsruute ning arendasin marsruudis sõiduki praeguse asukoha arvutamist. Kõige keerulisem osa minu jaoks oli sõiduki asukoha leidmine. Mul oli loetelu teekonnapunktide kordinaatidest, mis saadi OpenRoute teenusest. Ma teadsin, et punktide vahel on sirged teelõigud. Ma arvutasin iga lõigu pikkust matemaatilise valemi “kahe koordinatide vahel pikkuse leidmise” kasutades. Seejärel arvutasin masina muutuva kiiruse abil iga punkti vahelise ajavahemikku. Seejärel arvutasin absoluutsed ajad igas teekonna punktis, liikumise algusaega kasutades. Seejärel, tsükkli kasutades, ma leidsin, millise kahe teekonnapunkti vahel sõiduk asub. Seejärel arvutasin praeguse asukoha teepikkust laius- ja pikkuskraadi kasutades. Selles kasutajaloos asukoha arvutamiseks ma kasutasin tsükli konstruktsioone, matemaatilisi ja kinemaatika valemeid. Selle kasutajaloo implementeerimiseks mul kuulus umbes 58 tundi. Neist 4 tundi kuulus uurimiseks, 10 tundi marsruudi endpointe ja selle jaoks loogika arendamiseks ja 15 tundi praeguse asukoha leidmise loogika implementeerimiseks. Automaattestimine võttis palju aega ja tekitas raskusi seoses välise teenuse kasutamisega. Testimiseks kuulus umbes veel 19 tundi. Lisaks kuulus koodi parandamiseks pärast juhataja ülevaadet veel 10 tundi.

20 veebruarist kuni 19 märtsini arendasin *Charging Mode* kasutajalugu, mille tulemuseks oli tumblernupp kasutajaliides, mille vajutades vahetatakse laadimisrežiime. Kokku arendamiseks kulus 58 tundi. Antud kasutajalugu arendati *back-end*'i ja front-end'i poolt. *Back-end*'i osa oli üsna lihtne rakendada ja võttis mul umbes 12 tundi aega. *Front-end*'i

osa võttis rohkem aega - 25 tundi, sest see oli minu esimene kogemus React JS raamistikuga. Samuti uurimiseks kuulus 8 tundi aega, testimiseks umbes 2 tundi ning parandusteks pärast juhataja ülevaadet kulus terve tööpäeva.

27 ja 28 veebruaril kasutajaloo arendamise protsessis oli vaja pausi teha. Rakenduses tekkisid vead: loogika, mis ma arendasin eelmise kasutajaloo raames praeguse asukoha arvutamisega läks katki. See juhtus uue kasutajaloo, millega tegi teine tiimiliige, integreerimise tõttu. See nõudis kahe uue GitHubi probleemi loomist. 27 veebruaril ma leidsin veapõhjuse ning lõin kaks uut ülesannet selle lahendamiseks. Ühe *neist Location of the car is not on the route* parandasin samal päeval. Taastasin varem implementeeritud loogikat. 28 veebruaril koos Mark Porohnjaga parandasime *Bug in returning of current location* ülesannet. Leidsime kust viga tuli ja tegime vajalikud koodiparandused. Selle päeva jätku kulutasime ülesandepüstituse lõputöö kirjutamisele. Mõlema vea lahendamiseks kuulus kokku kaks tööpäeva.

20 märtsist kuni 11 aprillini arendasin *Real Charging Curve* kasutajalugu. Minu eesmärgiks oli luua mittelineaarset laadimist laadimiskõvera kasutades. Kõige keerulisem tööprotsessis oli laadimise kõvera graafiku füüsilisest tähendusest arusaamine ja laadimise algoritmi koodis implementeerimine. Samuti antud kasutajalugu eeldas füüsika valemite teadmist ja rakendamist. Mulle anti laadimiskõvera graafikut, kus abstsissteljel oli *State of Charge* ja ordinaatteljel oli võimsus. Samuti juhendati kasutama varem rakendatud lineaarse algoritmi asemel "sammude kaupa" laadimisalgoritmi. Nendest andmetest leidsin viisi, kuidas arvutada laadimiseks kuluvat aega iga üheprotsendilise sammu laadimisest, ja implementeerisin selle loogika nõutava algoritmi kasutades. Samuti reaalses režiimis laadimine on mõistlik ainult kiirlaadimise tüüpi laadijaga(nt *CHAdeMO*), kus pakutakse kõrgemaid võimsuseid. Laadimine toimub maksimaalselt selle võimsusega, mis laadija suudab pakkuda. Seetõttu laadimiskõvera võimsuseid "lõigati" võimsuseks, mis laadija seade suudab pakkuda. Selle jaoks kasutati Math.min funktsiooni. Kasutajaloo arendamiseks kuulus kokku umbes 40 tundi. Neist 4 kuulus uurimiseks ja 23 loogikat implementeerimiseks. Koodi testiti põhjalikult kuid kiiresti, sest ühiktestimine ei tekitanud enam raskusi nagu varem. Erinevate juhtumi testimiseks oli ühiktestid arendatud erineva graafiku osades: alguses, keskkohas ning lõppus. Testimiseks kuulus umbes 5 tundi ning parandusteks pärast juhataja ülevaadet kuulus terve tööpäev.

16 aprillist kuni 3 maini ma arendasin *Real Discharging Curve*. Kuigi loogika oli laadimise loogikaga sarnane, oli see siiski erinev, sest see on tihedalt seotud auto liikumise loogikaga. Alguses implementatsioon kasutas laadimise algoritmi, kus laadimine toimub "sammude kaupa" ja iga sammuga *State of Charge* suurendatakse ühe protsendi ehk mõne energiahulga võrra. Aga see implementatsioon oli liiga massiivne ja ebamõistlikult keeruline. See nõudis lisa väärtuste salvestamist sõiduki olekus ja sisaldas liiga palju arvutusi. Seetõttu vastavalt arhitektri juhistele 7. mail implementeeriti kasutajalugu uuesti. Kõveraga tühjenemine toimub nüüd sama lihtsa algoritmiga, nagu lineaarses režiimis, välja arvatud tegeliku elektritarbimise kalkuleerimist. Arvutasin tegeliku elektritarbimist füüsika valemite ja kõvera väärtuste kasutades. Kasutajaloo arendamiseks kulus kokku umbes 64 tundi. Neist 8 kulus uurimiseks, loogikat implementeerimiseks kulus 24 tundi. Testimiseks kulus umbes 7 tundi ning parandusteks pärast ülevaadet kulus umbes 5 tundi. Aga teise implementatsiooniks kulus veel 20 tundi.

Real Discharging Curve kasutajaloo arendamise käigus avastati vead, mis oli vaja parandada. Näiteks salvestati eelmise asukoha väärtust sõiduki olekus, kuid eelmiseid oleku väärtuseid hoida muutujana oli halb meetod ning juhataja palus seda lahendada teistpidi. See nõudis uue GitHubi probleemi loomist ning 7 mail ma tegin Remove previous location from VehicleState and use clone ülesannet ja alustasin vigu parandama. Selle lahendamiseks ma kasutasin oleku kloonimist ja selle *deep copy* otse teenuses kasutamist. Seda vahendit rakendati järgmisel viisil. Luuakse sisemise oleku klooni. Olek on sellesse eelnevalt täielikult salvestatud. Edasi eelneva asukoha muutuja oleku asemel kasutatakse praegust asukohta kloonist. Selle lahendamiseks kulus aega umbes terve tööpäev.

Mai alguses rakenduse arendamisega paralleelselt alustasin lõputöö aruannet kirjutama, varem tehtud märkuste ja detailse logi kasutades. 4 ja 5 mail kirjutasin minu poolt tehtud kasutajalugude tulemusi. Joonistasin skeeme algoritmidega ja vormistasin koodi kuvatõmmiste seletamisega. 6 ja 9 mail kirjutasin tulemuste osas kirjeldatud kasutajalugudele alnalüüsi ja järeldusi. 10 mail kirjutasin sissejuhatuse peatükkis projekti funktsionaalsed ja mittefunktsionaalsed nõuded ning osaliselt metoodika peatüki. Järgnevalt koos meeskonnaga olen teinud parendusi ja muutusi meie juhatajate soovituste alusel. Kokku kuulus aruande kirjutamiseks umbes 40 tundi.

Alates 14 maist ma hakkasin tegelema uute vigade parandamisega. Enne selle lahendamist ma lõin GitHubi probleemi *Car travels different distances with the same time and speed*. Vead koosnesid auto vale liikumisest mööda marsruuti. Vead olid avastatud *back-end*'is OpenRoute teekonnapunktide saamises: kasutajaliides võib näha, et auto liigub mõnikord sirgetel marsruutide osadel tagasi ja lahkub marsruudilt tee pöördel ja sõidu alustamisel. See ülesanne on praegu tööprotsessis, aga lahendamiseks kindlasti vajab põhjalikult kiiret uurimist.

Lisaks umbes 18 tundi ma kuulutasin iga arenguetapi lõpus korraldatud üritusteks, kus meil oli retrospektiivide ja järgmise etapi planeerimised ning valmis kasutajaloode demonstreerimine. Umbes 5 tundi kulus teise meeskonnaüritusteks, nimelt püstijala koosolekud(ingl *stand-up*), kus me planeerisime tööprotsessi. Lisaks me uurisime koodi koos meskonnaga, et õigesti aru saad arendatud kasutajalugusid. Samuti meil oli React JS'i seminar, mis võttis mitu tundi. Lisaks umbes 20 tundi aega kulus õppekirjanduse ja -videote uurimiseks. Projekti käigus ma täiendasin oma oskusi *back-end*'i arendamises ja automaattestimises. Sain uue kogemuse *front-end*'i arenduses React JS raamistikku kasutades. Projekti arendamises minu peamiseks ülesandeks oli lahendust leidmine ja ainult pärast seda koodi implementeerimine. Sel põhjusel mul oli võimalus iseseisvalt keerulisi ülesandeid lahendada ning erinevate probleemide jaoks lahendust leiutada. Sel viisil ma õppisin ise otsustada - mis algoritme ja meetode kasutada ning mis andmestruktuure ja konstruktsioone valida.

Kõige keerulisem tööprotsessis minu jaoks oli uue *front-end*'iga töötamine, sest see oli minu esimene kogemus React JS raamistikuga. Samuti ülesannete lahenduste leidmiseks oli vaja süveneda probleemi keerulise tehnilisse külge. Kõige lihtsam oli *back-end*'i koodi arendamine ja testimine, sest selline kogemus oli saadud eelmise semestri meeskonnaprojektis ja ülikooli õppimise perioodil. Mina isiklikult hindan oma semestri jooksul tehtud tööd edukaks. Seatud ülesanded olid keerulised, kuid suurem osa neist on täidetud. Väga oluline on see, et suur kogemus ja palju uusi teadmisi on saadud. Tunnen ennast nüüd palju kindlama tarkvara arendajana.

Töö selle projekti kallal pole minu jaoks mitte ainult väga kasulik, vaid ka nauditav. Osalesime projektis, mis on osa suurest süsteemist, mis võib ühiskonnale ja keskkonnale suurt kasu tuua. Projekti aruande kirjutamise käigus pidin selle teemaga veelgi süvenema, et probleemi uurida mitte ainult seestpoolt, vaid ka laiemalt. Tehtud tööd analüüsides ja

võrreldes sain selgeks, et meie töö toob kasu mitte ainult ettevõttele, vaid ka meie riigile tervikuna. Samuti töö Eesti Energias on töö meeldivas kollektiivis, kus juhatajad andsid palju nõusid minu töö parendamiseks. Nende abi ja toetus oli väga kasulik. Meeskonnas ka igaüks on valmis teineteisele appi tulla.

Minu arvates, Eesti Energia ja TalTech'i koostöö oli tore võimalus tudengitele, et tõelist kogemust tarkvara arenduses saada ja valmista ennast tõeliseks tööks pärast ülikooli lõpetamist.

Lisa 4. Sten-Mark Paju enda panuse kirjeldus ja eneseanalüüs

Käesoleva töö ühe autorina tegelesin enamasti kogu projekti käigus kasutajaliidese arendamisega. Põhjus seisnes selles, et kui saime projekti esimest korda kätte, asusid kõik tiimiliikmed kohe serveripoolse arenduse peale ja tundsin, et kui lähen sama teed pidi, jääb üks osa projektist tegemata või teisest poolest liialt maha, mis lõppude lõpuks ka juhtus.

Projekti esimeses etapis – sügissemestril – tegelesid meie üheksa liikmelises tiimis kaks inimest kasutajaliidese arendamise ja seitse liiget serveripoolse arendusega. Seetõttu tekkis probleem, kus üks osa projektist jõudis palju kaugemale kui teine. Töös on ka mainitud, et kuna ülikooli poolt jäi kasutajaliidese arenduse õpetamine väheseks, siis kahe faktori kokku sattumisel – vähene *front-end*'i tundmine ning tiimi koondumine pigem serveripoolse arenduse peale, jäi kasutajaliidese arendus serveripoolsest märkimisväärselt maha. Väga suur osa ajast läks esialgselt Angulari ning hiljem ka Reacti õppimisele, kasutusele võetud pistikprogrammide kasutama õppimisele ning arendusele endale. Kokkuvõttes ei jõutud projekti esimeses pooles kasutajaliidese arendusega kuigi kaugele, kuid kõik õpitu ja kogetu võeti kaasa projekti järgmisesse etappi

Projekti teine etapp algas muudatustega, milledest suurim oli tiimi pea poole väiksemaks muutumine. Selle käigus tuli kasutajaliidese arendajana appi Peeter Raudsepp, kellega suur osa arendusest selles ajavahemikus ka läbi viidi.

Esmalt oli vaja lõpule viia pooleliolev kasutajalugu ehk arendusülesanne - kaardipõhine simulaator. Suur põhjus, mis selle kasutajaloo pikaks venitas oli see, et pidevalt lisandus uusi nõudeid, mis oli vaja jooksvalt sisse viia. Esialgselt oli kasutajaloos kirjas, et on vaja kasutajaliideses näha kaarti ning sellele lisada markerite kuvamise võimalus. Markereid oli vaja erinevaid, näiteks autode, kodu ja töö asukohtade ning linnulennult teekonna kaardile kuvamise sisse-välja lülitamist. Nendel ei pidanud olema esialgselt loogikat taga, ainult kuvamisevõimalus. Kasutajaloo lõpuks oli neile juurde pandud API päringud, mida kasutasid iseloodud React-Leaflet komponendid. Komponendid töötasid läbi API päringutest saadud informatsiooni ning suunasid selle õigesse asukohta. Näiteks auto asukoha koordinaadid töötati läbi ning sellele omistati auto marker. Lisaks alguses ei

olnud vajalik auto teekonna loogika, kuid see arenes linnulennult autoteekonnaks ning väga kiirelt oli vajalik, et luua teekond, mis ühendab kodu ja töö asukohti kasutades olemasolevaid liiklusteid. Lisaks muutis väga pikaks kasutajalooga lõpuni jõudmise ka asjaolu, et tuli ära õppida, kuidas toimib kasutajaliidese testimine, mille kohta ei leidunud väga palju informatsiooni internetis.

Selles kasutajaloos tegin suure osa mainitud tööst ära mina, kuid abiks olid ka tiimiliikmed nagu näiteks Peeter Raudsepp, kes aitas mind testimise ja teekonna loogikaga.

Õppetund sellest olukorrast oli see, et oleksin pidanud uued tekkinud nõuded tagasi lükkama ja ütlema, et nende kohta on vaja teha uus kasutajalugu, kuid seda teadmist tol ajal veel ei omanud. Kasutajaloo lõpuks oli mul tekkinud hea baas arusaamine Javascriptist, Reactist ning Leaflet raamistikust ning olin õppinud neid kasutama kesktasemel.

Järgmisteks ülesanneteks sai kasutajaliideses implementeerida kasutajalood - sõiduki liikuma panemine *front-end* kaardil ning kodu- ja tööasukohtade vaheline kaugus. Esmalt alustasin töötamist kodu- ja töökohtade vahelise distantsidega, mille käigus oli vaja läbi viia tööd nii serveri kui ka kasutajaliidese poolel. Kasutajaliidese poolel viisin sisse mitmed muudatused, näiteks lisasin autode lähtestamise ja uuesti genereerimise nupud ning viisin läbi rekonstruktsiooni kasutajaliidese visuaalis. Selle kasutajaloo läbiviimiseks oli suuresti abiks väga kergesti arusaadav Material UI dokumentatsioon ning peavalu valmistas iseenesest lihtne CSS, millega viisin läbi kasutajaliidese rekonstruktsiooni. Antud kasutajalugu raskusi ei valmistanud ja valmis kiirelt.

Seejärel liikusin appi Peeter Raudsepale, kes tegeles sõiduki liikuma panemisega *front-end* kaardil. See kasutajalugu osutus esialgselt arvatust raskemaks, sest kaardi komponendid oli vaja panna dünaamiliselt liikuma. Viisime läbi väga palju teste ning seda eeskätt teadmiste ja kogemuste puudusest, kuidas panna Leaflet kaardil erinevad komponendid dünaamiliselt liikuma. Raske pähkel siin oli see, et ei osanud piisavalt hästi manipuleerida Reacti olekuid ning seetõttu tekkis vajadus kasutada pistikprogrammi. Esialgsed otsingud antud programmi järele vilja ei kandnud, kuid lõpuks jõudsime Drift Marker pistikprogrammini, millest esialgselt üle vaatasime, sest selle dokumentatsioon oli halb. Lugedes dokumentatsiooni mitmeid kordi saime Peetriga lõpuks probleemist

jagu ning lahendus oli oodatust lihtsam. Ütleks jällegi, et põhjalikum Reacti tundmine oleks andnud arendajatele suure eelise, kuid tänu sellele kasutajaloole said mõlemad arendajad tähtsa kogemuse võrra rikkamaks ning omandasid teadmised komponentide oleku manipuleerimisest.

Neljandas arendusetapis oli minu ülesandeks muuta olemasolevat *Docker*-i implementatsiooni projekti serveri poolel. Peamine raskus siinkohal oli see, et kuidas muuta olemasolev *Dockerfile* võimalikult efektiivseks minimaalse käskude hulgaga. Isegi kui *Dockerfile* töötas ning kõik tingimused olid täidetud, oli neid lahendusi siiski vaja mitu korda efektiivsuse ja kompaktsuse nimel ümber kirjutada. Probleemi tekitas minule vajadus kasutada mitmik astmelist *Dockerfile* ülesehitust, sest internetis oli väga palju erinevaid näiteid, kuidas seda luua. Väga paljudel näidetel oli küljes palju mitte vajalikku informatsiooni ning lõplikku lahendust pidin mitmeid kordi ümber tegema. Siinkohal sain jällegi kaasa hea õppetunni: isegi kui lahendus töötab, ei pruugi ta olla veel kõige õigem ning seda saab muuta paremaks.

Eesti Energia ja Taltech ülikool pakkusid mulle kahe semestriga kogemuse, millist ei oleks suutnud isegi ette aimata. Üks suurimaid õppetunde projektile tagasi vaadates taandub teadmistele ja ressursside jagamise tasakaalule. Kui mõni ülesande osa tundub keerukam ja selle lahendamiseks pole piisavalt teadmisi, kipub inimressurss kalduma nende ülesannete poole, millest saadakse aru ja osatakse lahendada. Just see juhtus ka meie projekti algfaasis ning see mõjutas otseselt projekti erinevate osade edenemise kiirust ja tulemust.

Tehnilise poole pealt tunnen, et õppisin väga palju uute tehnoloogiate kohta, millega polnud varem kokku puutunud. Serveripoolse projekti tehnoloogiate seas on Gradle-t kasutav Spring Framework ning selle programmerimiskeel - Java. Kasutajaliidese tehnoloogiate poolt õppisin hästi tundma Javascripti ja Typescripti, sest neid kasutasid meie poolt kasutatavad raamistikud React ja Angular. Õppisin väga palju koodiloogika kohta, kuidas mingisugune informatsioon peaks liikuma ühest protsessist teise. Tunnen, et nende tehnoloogiate tundma õppimine annab mulle väga tugeva alustala tulevikuks, sest on töömaastikul suhteliselt laialt kasutatavad tehnoloogiad.

Samuti sain selgemaks, mida tähendab suures ettevõttes töötamine, milline näeb välja arendustiimi igapäevatöö ja kuidas üleüldse töötada suure meeskonna liikmena. Kokku

annavad need kogemused pagasi, mida ei ole võimalik omandada kirjutades tavalist lõputööd ning annavad ühele tudengile, kes alustab teekonda infotehnoloogia maailmas, märkimisväärse eelise teiste tudengite seas. Kindlasti soovitan Taltech ülikoolil jätkata sarnaste projektidega ning hea on näha, et Äriinfotehnoloogia õppekava on seda teed pidi ka minemas.