TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond
Tarkvarateaduse instituut

Rasmus Lelumees 144097

VABAVARALISTE IOT PILVEPLATVORMIDE VÕRDLUS JA VALITUD PLATVORMI RAKENDAMINE

Magistritöö

Juhendaja: Priit Ruberg
Nooremteadur
Kaasjuhendaja: Mairo Leier
Teadur

Tallinn 2017

Autorideklaratsioon

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

Autor: Rasmus Lelumees

01.05.2017

Annotatsioon

Käesolev töö keskendub IoT andmete kogumiseks ning visualiseerimiseks vajaliku platvormi loomisele, mille sihtgrupiks on väiksemad nutikodud ja lihtsamad IoT projektid. Platvormi loomiseks rakendatakse pilveteenust, mille valik põhineb hetkel turul olevate lahenduste analüüsimisel ja võrdlemisel. Töö eesmärgiks on näidata, et pilvelahenduste baasil on võimalik kiire prototüüpimise tulemusena vajalik andmete kogumise ja visualiseerimise platvorm luua.

Lõputöö tulemuseks on põhjalik pilveplatvormide võrdlus ning töö eesmärkideks sobivaima pilveplatvormi rakendamine, mis seisneb vastavale platvormile prototüübi loomisel. Loodav prototüüp on rakendus, mis võimaldab nutikodu sensorite andmeid pilveplatvormil koguda ning veebirakenduse kaudu kogutud andmeid reaalajas atraktiivselt välja kuvada. Rakendus on ühtlasi tõestuseks, et pilvelahenduste baasil on võimalik eesmärgiks seatud andmete kogumise ja visualiseerimise prototüüp luua.

Lõputöö on kirjutatud eesti keeles ning sisaldab teksti 56 leheküljel, 6 peatükki, 25 joonist, 6 tabelit.

Abstract

Comparison of IoT cloud platforms and application of a chosen platform

The aim of this thesis is to create a platform that would enable IoT data collection and visualisation. The platform is targeted for smaller smart homes and other simpler IoT projects. The platform is created using a cloud computing service that is chosen based on the existing solutions on the market. The existing solution could be a cloud service that already provides an interface for collecting and visualising IoT data or a universal platform which provides an infrastructure for building an application that would achieve the same results. These existing solutions are analysed and compared. As a result, the best possible cloud platform is chosen which is used for building a prototype application. The goal of this work is to show that it is possible to create a cloud solution based platform for IoT data collection and visualisation as a result of rapid prototyping.

The result of this thesis is a thorough comparison of cloud platforms and an application that enables collecting and visualising smart home sensor data. The created application is also an evidence to the given goal that it is possible to use cloud platforms to rapidly prototype a solution that would enable data collection and displaying the sensor data.

The thesis is in Estonian and contains 56 pages of text, 6 chapters, 25 figures, 6 tables.

Lühendite ja mõistete sõnastik

AMQP	*Advanced Message Queuing Protocol*, avatud standardina kirjeldatud rakenduskihi sõnumivahetusprotokoll
API	*Application Programming Interface*, rakendusliides ehk reeglistik valmisprogrammiga suhtlemiseks
Back end	Tarkvarakomponendi andmetöötluskiht, mis üldjuhul teenindab presentatsioonikihti ehk front end kihti
Deadlock	Paralleelse arvutustegevuse lukustunud paigalseis, kus tegevuste grupi kõik liikmed ootavad mõne teise grupi liikme järel, et need luku vabastaks
DOM	*Document Object Model*, dokumendi objektimudel, rakendusliides, mis tõlgendab HTML, XML või XHTML dokumenti kui puustruktuuri
ECMAScript	ECMA-262, Ecma International'i poolt standardiseeritud programmeerimiskeel, mille tuntuimateks dialektideks on JavaScript ja ActionScript
Front end	Tarkvarakomponendi presentatsioonikiht
GB	*Gigabyte*, Gigabait ehk 2^{30} baiti
HTML	*Hypertext Markup Language*, standardiseeritud veebilehtede ja veebirakenduste märgendamise keel
HTTP	*Hypertext Transfer Protocol*, rakenduste protokoll teabe edastamiseks arvutivõrkudes
IaaS	*Infrastructure as a Service*, pilveteenuse mudel, mis pakub virtualiseeritud riistvara ning sellega seotud peamisi toiminguid teenusena
IoT	*Internet of Things*, Asjade Internet, ka Värkvõrk, interneti kaudu seotud asjade (autode, olmeelektroonika jt seadmete) võrk
Java	Objektorienteeritud klassipõhine programmeerimiskeel
JavaScript	ECMAScript'i dialekt, dünaamiline tüübivaba programmeerimiskeel, mida kasutatakse valdavalt veebilehtede kodeerimiseks
JSON	*JavaScript Object Notation*, programmeerimiskeelest sõltumatu andmevahetusformaat, mis pärineb JavaScript'ist
JSX	*JavaScript XML*, JavaScript'i laiendsüntaks

JWT	*JSON Web Token*, JSON'il põhinev avatud standard, mida kasutatakse ligipääsu piiramiseks tähiste (ingl *token*) abil
M2M	*Machine to Machine*, viide kommunikatsioonikanalist sõltumatule seadmetevahelisele otsesuhtlusele
MATLAB	*Matrix Laboratory*, numbriline arvutuskeskkond ja programmeerimiskeel
MB	*MegaByte*, Megabait ehk 2^{20} baiti
MQTT	*Message Queue Telemetry Transport*, ISO standardil (ISO/IEC PRF 20922) põhinev "kergekaaluline" sõnumivahetusprotokoll
Node.js	Asünkroonne sündmustepõhine JavaScripti versioon
NoSQL	*Non-SQL* või *not only SQL*, andmebaasi liik, mis kasutab andmete salvestamiseks ning vahendamiseks mehhanisme, mis erinevad relatsiooniliste andmebaaside tabelipõhisest lähenemisest
PaaS	*Platform as a Service*, pilveteenuse mudel, mis pakub rakenduste arendamise, jooksutamise ja haldamise platvormi teenusena
PHP	*Hypertext Preprocessor,* originaalis *Personal Home Page*, peamiselt veebirakenduste loomiseks mõeldud programmeerimiskeel
RAM	*Random-access Memory*, muutmälu ehk operatiivmälu
Skript	Programm, mis automatiseerib ülesandeid, mida alternatiivselt teostaks manuaalselt ja ükshaaval inimene
SPA	*Single-page Application*, veebirakendus või veebisait, mis on loodud ühele leheküljele ning mille vajalik kood laetakse serverist ühekordse lehelaadimisega
SQL	*Structured Query Language*, standardiseeritud (ISO/IEC 9075) struktuurpäringukeel ehk andmebaasi päringukeel, mida kasutatakse relatsioonilistes andmebaasides
SSL	*Secure Sockets Layer* või ka *Transport Layer Security (TLS)*, transpordikihi turbeprotokoll, mis tagavad kommunikatsiooni turvalisuse arvutivõrkudes
Thread	Lõim ehk paralleelne protsess, mis teeb koostööd või on muul moel seotud sama programmi teiste paralleelsete protsessidega
UI	*User Interface* ehk kasutajaliides, ruum või kontekst, milles toimub inimese ja masina vaheline interakteerumine
WebSocket	Standardiseeritud kommunikatsiooniprotokoll, mis võimaldab mitut samaaegset suhtluskanalit üle ühe TCP ühenduse
XP	Extreme Programming, üks enim tuntud agiilne tarkvaraarendusmetoodika

Sisukord

1 Sissejuhatus 13
1.1 Ülesandepüstitus 14
1.2 Metoodika 14
1.3 Ülevaade tööst 15
2 Teoreetiline taust 17
2.1 IoT lahenduste loomiseks pakutavad platvormid 18
2.1.1 ThingSpeak 19
2.1.2 Kaa 19
2.1.3 Dweet.io 19
2.1.4 BeeBotte 20
2.1.5 Carriots 20
2.1.6 Ubidots 20
2.1.7 Amazon Web Services 20
2.1.8 Google Cloud 22
2.1.9 IBM Watson IoT 22
2.1.10 Microft Azure 22
2.1.11 RedHat OpenShift (Next Generation Developer Preview) 23
2.1.12 Heroku 24
2.1.13 Nanobox 24
2.2 Prototüübi testimiseks kasutatud seadmed, sensorid ja tarkvara 24
2.3 Kasutatavus, kasutajasõbralikkus ja kasutajakeskne disain 25
2.4 Kasutajakeskne disain ja disainiprotsessi tehnikad 26
2.5 Iteratiivne disainimudel 27
2.6 Agiilne tarkvaraarendusprotsess 29
3 Pilveplatvormide analüüs ja võrdlus 31
3.1 Mittefunktsionaalsed nõuded 31
3.2 IoT andmete kogumisele spetsialiseerunud platvormid 32
3.2.1 ThingSpeak 33
3.2.2 Kaa 33

3.2.3 Dweet.io 33
3.2.4 BeeBotte 34
3.2.5 Carriots 34
3.2.6 Ubidots 34
3.2.7 Spetsialiseerunud platvormide vastavus baasnõuetele ja võrdluse kokkuvõte 35
3.3 Universaalsed pilveplatvormid 36
3.3.1 Amazon Web Services 37
3.3.2 Google Cloud 38
3.3.3 IBM Watson IoT 38
3.3.4 Microsoft Azure 39
3.3.5 RedHat OpenShift (Next Generation Developer Preview) 39
3.3.6 Heroku 40
3.3.7 Nanobox 40
3.3.8 Universaalsete platvormide vastavus baasnõuetele ja võrdluse kokkuvõte 40
4 Rakenduse arhitektuur ja kasutatavad tehnoloogiad 42
4.1 Prototüübi arhitektuur valitud pilveplatvormi põhjal 42
4.2 Programmeerimiskeele valik 44
4.3 Rakenduse sisemine arhitektuur 44
4.4 Versioonihaldus 47
5 Disaini- ja arendusprotsess 48
5.1 Disainiprotsessi ülevaade 48
5.1.1 Intervjuud 48
5.1.2 Prototüüpimine ja kasutajatestimine 50
5.1.3 Valideerimine disainikriitika abil 52
5.2 Arendusprotsessi ülevaade 52
5.2.1 Pilveprojekti loomine 53
5.2.2 Rakenduse arendamine ja jooksutamine 54
5.2.3 Arendusprotsessi kokkuvõte 63
5.3 Süsteemi funktsionaalsused 63
5.3.1 Aktorid 63
5.3.2 Kasutuslood 64
6 Võimalikud edasiarendused 67

7 Kokkuvõte 68
Kasutatud kirjandus 69
Lisa 1 – Ülesanded kasutajale disainiprototüüpide kasutajatestimisel 72

Jooniste loetelu

Joonis 1. Amazon Web Services IoT mooduli arhitektuurne skeem (Amazon Web Service'i veebileht 22.03.2017). 21
Joonis 2. Microsoft Azure IoT Hub arhitektuurne skeem (Microsoft Azure veebileht, 22.03.2017). 23
Joonis 3. Lihtne iteratiivse disaini mudel (Rogers jt 2011, 186). 29
Joonis 4. Ekraanipilt ThingSpeak platvormi andmekogumise ja -visualiseerimise graafikust (ThingSpeak'i veebileht, 17.03.2017). 36
Joonis 5. Loodava projekti Google Cloud platvormi konsooli avaleht. 42
Joonis 6. Prototüübi arhitektuursed komponendid. 43
Joonis 7. Esimesed kasutajaliidese visandid, mis koostati intervjuude põhjal. 49
Joonis 8. Paberprototüübi vaade sensori positsioneerimiseks plaanil. 50
Joonis 9. Digitaalse disainiprototüübi esimese versiooni kasutajaliidese elemendid. 51
Joonis 10. Digitaalse disainiprototüübi uuendatud avalehe väljavõte. 52
Joonis 11. Käsureaprogrammi konfiguratsioon Google Cloud pilveteenuse kasutamiseks. 53
Joonis 12. Programmikood: paketihalduse abil moodulite installimine ning rakenduse käivitamine. 54
Joonis 13. Programmikood: rakenduse pilve laadimine Google Cloud käsureatööriista abil. 54
Joonis 14. Programmikood: Yarn paketihalduse käsud. 54
Joonis 15. Programmikood: rakenduse jooksutamise ja ehitamise konfiguratsiooniskriptid package.json failis. 55
Joonis 16. Programmikood: rakenduse pilvekonfiguratsioon app.yaml failis. 55
Joonis 17. Programmikood: primitiivse veebiserveri näide Express.js põhjal (Express.js veebileht, 18.04.2017). 56
Joonis 18. Programmikood: serveri juurfaili väljavõte, mis tagastab kliendirakenduse. 56
Joonis 19. Programmikood: seadmelt vastuvõetud andmete salvestamine lubaduste ehk *promise*'ite abil. 57
Joonis 20. Programmikood: JSON Web Token märgendi genereerimise koodinäide. .. 58

Joonis 21. Programmikood: React'i rakenduse formuleerimine. 59
Joonis 22. Programmikood: React'i ruuteri seadistamine..... 59
Joonis 23. Programmikood: React'i komponendi lähtestamise meetod. 60
Joonis 24. Programmikood: React'i komponendi render meetod ja JSX süntaksi näide.61
Joonis 25. Programmikood: Webpack'i konfiguratsioon loodud rakenduses..... 62

Tabelite loetelu

Tabel 1. IoT andmete kogumisele spetsialiseerunud platvormide võrdlus. 35
Tabel 2. AWS'i arhitektuursete komponentide hind ja mahupiirangud. 38
Tabel 3. Google Cloud’i arhitektuursete komponentide hind ja mahupiirangud. 38
Tabel 4. Microsoft Azure’i arhitektuursete komponentide hind ja mahupiirangud. 39
Tabel 5. Heroku arhitektuursete komponentide hind ja mahupiirangud. 40
Tabel 6. Universaalsete pilveplatvormide võrdlus. 41

1 Sissejuhatus

Asjade Internet (ingl *Internet of Things, IoT*) on üha enam inimeste teadvusse ja elu- või töökeskkonda jõudmas. See märgib uue arvutiajastu algust, kus traditsioonilise lauaarvuti ja mobiilsete nutiseadmete kõrval on ka erinevad koduelektroonika seadmed, sensorid ning aktuaatorid võrku ühendatud. Tulemusena elame keskkondades, kus info- ja kommunikatsioonitehnoloogia on nähtamatult juurdunud kõikjale meie ümber, tootmas tohututes kogustes andmeid, mida peab salvestama, protsessima ning presenteerima tõhusalt, veatult ja lihtsasti tõlgendataval moel. (Gubbi jt, 2013)

IoT turu suuruseks ennustatakse 2020. aastaks ligi 3.7 miljardit USA dollarit (kasvades mitmekordseks 2015. aasta 900 miljonilt). (Ip, 2016) Tänaseks on IoT turule jõudnud mitmeid kodutarbijale mõeldud seadmeid, mis üha enam populaarsust koguvad ning järjest rohkem kodu- või töökeskkonna elemente Asjade Internetiga ühendada suudavad. IoT lahendused on seega perspektiivikad ja läbimõeldud teostamise puhul kasumlikud.

Tallinna Tehnikaülikoolis tegeletakse igapäevaselt IoT maailma probleemide lahendamisega ning tõenäoliselt pole Infotehnoloogia teaduskonna Arvutisüsteemide instituudi teadurid ainukesed, kes oma eriala raames antud valdkonnaga kokku puutuvad. Käesoleva lõputöö raames aga keskendutakse just viimastele, lahendamaks probleemi, mis antud instituudi ekspertiisist välja jääb. Nimelt on teaduritel keeruline ja ajakulukas iga targa hoone või muu IoT projekti raames andmete visualiseerimiseks ja sensorite kontrollimiseks vajalik programm kirjutada. Töö on manuaalne ja iga kord veidi erinev, mistõttu ei toimu korduvkasutamist, kvaliteet on madal ning andmete kuvamine pole atraktiivne ega kasutajasõbralik. Kiiret andmete kuvamist on ka tudengitele vaja näiteks aine "IAX0230 Sardsüsteemide alused" raames.

1.1 Ülesandepüstitus

Käesolev töö keskendub probleemile, mis IoT lahenduste loojate ekspertiisist tihti välja jääb - andmete kogumiseks ning visualiseerimiseks vajaliku platvormi loomine. Töö eesmärgiks on luua vastava platvormi universaalne prototüüp, mis võimaldaks lihtsate vahenditega koguda ning visualiseerida näiteks nutikodu või mõne muu väiksema IoT projekti sensorite andmeid. Platvormi põhirõhk on hõlpsasti ülesseataval andmekogumismoodulil ning andmete atraktiivsel ning kasutajasõbralikul visualiseerimisel, mis lahendatakse pilveplatvormil (ingl *cloud computing platform*). Sobiva platvormi valimiseks teostatakse töös põhjalik analüüs ning võrdlus pakutavatele pilveteenustele. Vaatamata sellele, et pilveplatvormi kasutamine toetab skaleeruva lahenduse loomist, ei panda töös rõhku suure hulga seadmete toetamisele ning kergemate andmesideprotokollide toetamisele. Eesmärgiks on näidata, et pilvelahenduste baasil on võimalik kiire prototüüpimise tulemusena vajalik andmete kogumise ja visualiseerimise platvorm luua, mida saaks visualiseerimise eesmärgil kasutada ka mõne eksisteeriva IoT lahendusega API kaudu liidestatuna. Loodava prototüübi kasutamiseks kirjutatakse valitud pilveplatvormi ja loodud lahenduse kasutusjuhend, mille abil huvitatud osapool suudaks iseseisvalt lahenduse kasutusele võtta. Kasutusjuhend ning kogu prototüübi programmikood asub projekti koodirepositooriumis.

Loodav prototüüp võiks esmalt lihtsustada ning kiirendada teadurite ja tudengite tööd IoT projektides, pikemas perspektiivis oleks prototüüp aluseks kodukasutajatele mõeldud tervikliku nutikodu monitoorimise ja kontrollimise lahendusele.

1.2 Metoodika

Probleemi lahendamiseks on vajalik erinevate pilvelahenduste otsimine, analüüsimine ja võrdlemine. Võrdlusesse kaasatakse valmislahendusi, mis oma olemuselt võiksid probleemile lahendust pakkuda ilma lisanduva tarkvara kirjutamise vajaduseta ning ka universaalsemaid platvorme, mis eeldaksid andmete kogumiseks või visualiseerimiseks vajaliku rakenduse kirjutamist. Võrdluses olevaid platvorme analüüsitakse nende võimaluste, kulukuse ja teiste tööks oluliste parameetrite põhjal. Platvormide puhul luuakse minimaalne proovirakendus, millega saab valideerida platvormide kasutamise keerukust ning detailsemalt analüüsida platvormide võimekust. Võrdluse tulemusena

valitakse sobivaim platvorm, millele disainitakse ning arendatakse probleemi lahendamiseks vajalik rakendus.

Rakenduse disainimiseks viiakse läbi intervjuud kasutajatega, et kaardistada nende esmased soovid ja vajadused. Seejärel luuakse kaks erineva detailsusega disainiprototüüpi, mida arendatakse iteratiivselt, kuni jõutakse lõpliku disainilahenduseni. Prototüüpide abil viiakse läbi kasutajatestimist veel valmimata lahendusega. Disainiprototüüp saab olema aluseks valitud pilveplatvormile loodavale rakendusele, olles sisendiks tarkvara loomisele.

Arenduse etapis valitakse pilveplatvormi võimaluste, loodud disaini ning muude tehnoloogiliste nõudmiste põhjal andmete kogumiseks ja visualiseerimiseks vajaliku tarkvara kirjutamise keel, teegid ning raamistikud. Arendusprotsessi tehakse iteratiivselt agiilseid arendustehnikaid silmas pidades, mille tulemusena valmistatakse rakendus, mis andmete kogumist ning reaalajas kuvamist võimaldab.

1.3 Ülevaade tööst

Teises peatükis kirjeldatakse töö teoreetilist tausta - nimetatakse täna eksisteerivaid valmislahendusi, mis võimaldavad seadmetelt andmeid koguda, räägitakse pilveplatvormidest, nende võimalustest ja piirangutest, viidatakse disaini- ja arendusprotsessidele ja mudelitele, mille alusel prototüübi loomine toimub ning kirjeldatakse antud prototüübi testimiseks kasutusel olevaid seadmeid, sensoreid ja tarkvara.

Kolmandas peatükis teostatakse pilveplatvormide süvaanalüüs ja võrdlus nende võimalusi, hinda ning lihtsust arvesse võttes. Lisaks sätestatakse platvormide valideerimiseks vajalikud baasnõuded. Platvormide võrdluse ja analüüsimise tulemusena valitakse sobivaim lahendus prototüübi loomiseks.

Neljandas peatükis kirjeldatakse loodava prototüübi arhitektuuri ning kasutatavaid tehnoloogiaid, arvestades valitud pilveplatvormi võimalusi ning piiranguid. Samuti selgitatakse selles peatükis lähemalt, miks ja kuidas vastavaid tehnoloogilisi otsuseid tehti.

Viiendas peatükis kirjeldatakse disainiprotsessi alates esimese prototüübi loomisest kuni lõpliku disainilahenduse kasutajatestimiseni. Samuti kirjeldatakse arendusprotsessi kuni lõpliku lahenduse valmimiseni ning tuuakse koodinäiteid valminud rakendusest. Lisaks loetletakse lühidalt süsteemi funktsionaalsed nõuded, mille põhjal rakendus loodi.

Kuuendas peatükis pakutakse prototüübi edasiarendamise võimalusi, mida prototüübi järgmises versioonis teostada plaanitakse.

2 Teoreetiline taust

Asjade Internet ehk Internet of Things leidis terminina esimest korda kasutust Kevin Ashtoni poolt tarneahela juhtimise kontekstis (Ashton, 2009) ning jõudis laialdasemalt käibesse 2005. aastal. Tänaseks võib Asjade Interneti alla liigitada seadmeid ja süsteeme kõikvõimalikest eluvaldkondadest, sealhulgas transport, tervishoid, koduautomaatika jpt. (Sundmaeker jt, 2010) IoT termini looja sõnul põhineme meie, inimesed, ning kogu meid ümbritsev - majandus, ühiskond, ellujäämine - asjadel, mistõttu loevad asjad palju rohkem kui ideed või informatsioon. Kui meie loodud süsteemid teaksid rohkem asjadest, kasutades selleks iseseisvalt kogutud andmeid, suudaksime märkimisväärselt vähendada jääke, kulusid ja kaotusi. (Ashton, 2009)

IoT areng seab erinevate valdkondade ettevõtetele ning spetsialistidele uusi väljakutseid, aga loob ka seninägematuid võimalusi. Palju Asjade Interneti valdkonda liigituvaid tooteid on juba turule toodud ning üha enam jõuavad ka seni kallid ja keerukad seadmed või lahendused tee kodukasutajani. Järjest rohkem soovivad inimesed oma kodu või muid keskkondi automatiseeritumaks ja targemaks muuta ning seeläbi erinevaid seadmeid distantsilt kontrollida või monitoorida.

IoT turu suuruseks on ennustatud 2020. aastaks ligi 3.7 miljardit USA dollarit. Selleks ajaks on IoT seadmete arv ületanud 50 miljardi piiri. Sellise tempoga kasvava turu puhul on ärilise tulu saamiseks valmis paljud oma aega ja vaeva investeerima, et enda turuosa hõivata. (Ip, 2016) Täna leidub turul nii erinevaid IoT seadmeid, mis aitavad elukeskkonna temperatuuri, niiskust, valgust ja muid parameetreid reguleerida (Philips Hue, Sensi WiFi Thermostat jpt), aga ka komplekssemaid tooteid, mis häälkäsklustele reageerides teiste IoT seadmete kontrollimist teostavad (Amazon Echo, Google Home jt). Kõrgtasemeliste seadmete ja toodete kõrvale on tekkinud ka pilvepõhised IaaS või PaaS lahendused, mis tehnilisemate oskustega kasutajate jaoks IoT projektideks erinevaid võimalusi pakuvad.

2.1 IoT lahenduste loomiseks pakutavad platvormid

Olgu eesmärgiks lihtsalt andmete kogumine ja talletamine või keerukam pilvepõhine seadmete kontrollimine, leidub tänasel IoT turul palju erinevaid teenuseid, mis vastavaid eesmärke adresseerivad. On olemas nii spetsiifiliselt IoT suunitlusega teenuseid kui ka väga laiahaardelisi pilveplatvorme, mis ka kõige nõudlikuma kliendi soove rahuldavad. Järgnevalt on toodud nimekiri erinevatest platvormidest, mis antud projekti raames võimaliku teenusepakkujana kaalul olid. Kuna eesmärgiks oli luua võimalikult soodne, eelistatult tasuta lahendus, siis vastavalt on analüüsitud ka platvormide funktsionaalsusi ja võimalusi. IoT andmete kogumisele spetsialiseerunud lahendused oleksid suuremalt jaolt välistanud (vähemalt andmete kogumiseks vajaliku) koodi kirjutamise, ent nendel oli andmete visualiseerimine suurel määral piiratud. Seetõttu võib juba etteruttavalt öelda, et IoT'le spetsialiseerunud platvormid ei vastanud lõpuni projekti nõudmistele, mistõttu kaasati analüüsi ka universaalsemad platvormid, milles kogu rakenduse loogika tuli nullist üles ehitada.

IoT andmete kogumisele spetsialiseerunud platvormide ühisosaks on teenus, mis HTTP (*Hypertext Transfer Protocol*, rakenduste protokoll teabe edastamiseks arvutivõrkudes), MQTT (*Message Queue Telemetry Transport*, standardil põhinev sõnumivahetusprotokoll) või mõne muu kergekaalulise protokolli abil võimaldab kindla arvu seadmete andmeid vastu võtta, salvestada ja tihti ka platvormil visualiseerida. Mõned platvormid pakuvad lisaks ka erinevaid reageerimise mehhanisme ja liidestamise võimalusi. Kodeerimise vajadus on selliste platvormide puhul minimaalne, aga võimalused on selle võrra piiratumad, eriti võrreldes universaalsete pilveplatvormidega.

Universaalsete pilveplatvormide alla võib liigitada pea kõik PaaS (*Platform as a Service*, pilveteenuse mudel, mis pakub rakenduste arendamise, jooksutamise ja haldamise platvormi teenusena) ja IaaS (*Infrastructure as a Service*, pilveteenuse mudel, mis pakub virtualiseeritud riistvara ning sellega seotud peamisi toiminguid teenusena) lahendused, mis võimaldavad kasutaja loodud rakendust pilves käitada ning selle kaudu seadmetelt saadetud andmeid töödelda ning salvestada. Et kirjeldatud kriteeriumitele vastavate pakkujate hulk on võrdlemisi suur, keskendub autor antud töös populaarsematele ja enamtuntud lahendustele (mis on üldjuhul ka suuremad ja

komplekssemad platvormid). Võrdluseks kaasatakse mõned väiksemad teenusepakkujat, et mitmekülgsemalt teenuseid võrrelda ning prototüübiks parim lahendus leida.

Universaalsete platvormide puhul tuleb andmete töötlemiseks ja visualiseerimiseks vajalik rakendus ise kirjutada. Seetõttu on lisaks hinnale antud platvormide puhul oluline ka lihtsus, arusaadavus ja õpitavus, et rakenduse üleslaadimine ja jooksutamine oleks teostatav ka neile, kes vastava platvormiga pole varem kokku puutunud. Mõned universaalsed platvormid pakuvad lisaks eraldi IoT moodulit, mis võimaldab andmete kogumist ja salvestamist, ent visualiseerimise rakendus tuleb neis siiski ise kirjutada.

2.1.1 ThingSpeak

ThingSpeak pakub andmekogumise API (*Application Programming Interface* ehk rakendusliides) kõrval ka andmete visualiseerimist ja analüüsimist MatLab'i (*Matrix Laboratory*, numbriline arvutuskeskkond ja programmeerimiskeel) abil. See teenus on kõige lähedasem loodava prototüübi ideele, pakkudes funktsionaalsuse mõttes sarnaseid võimalusi. Lisaks võimaldab ThingSpeak andmete kogumisel reegleid kehtestada ning kriteeriumitele vastates ka reageerida mõne teise API abil - näiteks populaarse avatud lähtekoodiga elektroonikaplatvormi Arduinoga ühenduda ja seadet seeläbi kontrollida või e-maili ning sõnumi teel olukorra kohta infot edastada. (ThingSpeak veebileht, 17.03.2017)

2.1.2 Kaa

Kaa reklaamib end kui juhtiv tasuta IoT platvorm. Kaa toimib vahevarana seadmete ja loodud rakenduse vahel. Traditsioonilisest PaaS platvormist eristab teda suurem vabadus ja kontroll seadmeid kontrolliva tarkvara üle ning seadmete arvust sõltumatu tasuta pakett. Samas sõltub loodava rakenduse reaalne ülalpidamine kolmandatest osapooltest - näiteks saab Kaa platvormi jooksutada kohalikus arenduskeskkonnas või Amazon S3 abil pilves, aga mitte Kaa enda pilveplatvormil. Seega täiesti sõltumatu lahendusega Kaa puhul tegu ei ole ning suure tõenäosusega nõuab Kaa-põhine lõplik lahendus tasulise lisateenuse kasutamist. (Kaa veebileht, 20.03.2017)

2.1.3 Dweet.io

Dweet pakub ThingSpeak'ile sarnast lihtsat ja minimalistlikku tasuta andmekogumise võimalust. Erinevalt ThingSpeak'ist pole Dweet'is võimalik peale lihtsate hoiatuste

kuidagi andmetele reageerida või andmeid põhjalikumalt visualiseerida. Dweet'is on andmekanalid vaikimisi avalikud ning nende privaatseks muutmine on tasuline. (Dweet.io veebileht, 18.03.2017)

2.1.4 BeeBotte

BeeBotte pakub PaaS'i, mis toetab andmete edastamiseks HTTP kõrval ka WebSocket'i ja MQTT protokolle. WebSocket on standardiseeritud kommunikatsiooniprotokoll, mis võimaldab mitut samaaegset suhtluskanalit üle ühe TCP ühenduse. BeeBotte platvorm keskendub reaalaja IoT rakendustele ning võimaldab API kaudu välistel rakendustel andmete kohta infot pärida ja vastavalt reageerida. Selles mõttes on BeeBotte sarnane ThingSpeak'ile, aga eksisteerivate liidestuste arv on BeeBotte'is väiksem. BeeBotte hoiustab andmeid maksimaalselt 3 kuud. (BeeBotte veebileht, 17.03.2017)

2.1.5 Carriots

Carriots on platvorm, mis on mõeldud IoT ja Machine to Machine (M2M, kommunikatsioonikanalist sõltumatu seadmetevaheline otsesuhtlus) projektide loomiseks. Tasuta paketti saab lisada maksimaalselt 2 seadet ning platvorm toetab kuni 500 ühendust päevas. Andmeid hoiustatakse Carriots'is 3 kuud. Esimene pakutav tasuline pakett on 2€ seadme kohta kuus, mis on konkurentidega võrreldes kõige väiksem. (Carriots veebileht, 19.03.2017)

2.1.6 Ubidots

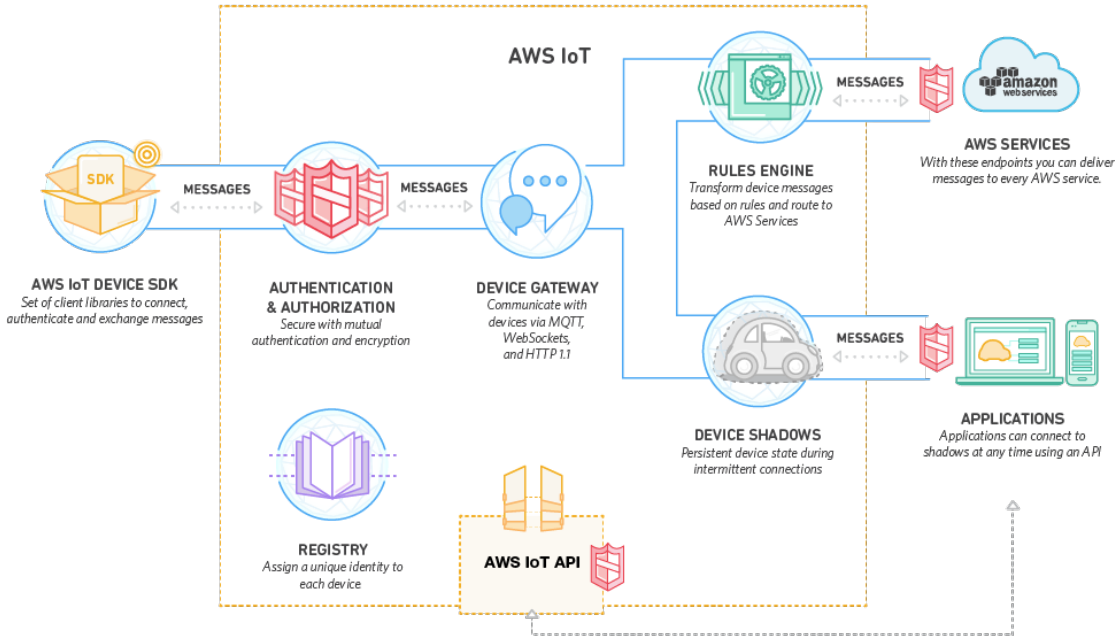
Ubidots'i tasuta pakett lubab kuni 20 seadmelt andmeid koguda, lihtsamaid hoiatussõnumeid kasutada ning andmeid nende platvormilt jälgida. Maksimaalne andmete hoiustamine on 3 kuud, toetatud on nii HTTP kui ka MQTT andmesideprotokollid. (Ubidots veebileht, 18.03.2017)

2.1.7 Amazon Web Services

Amazon Web Services ehk AWS pakub laiaulatuslikku platvormi, milles sisalduvaid komponente ja alamtooteid on üle saja. Antud prototüübi raames olulisteks osadeks on mitmed erinevad andmebaasi ja andmeladustamise komponendid ning pilverakenduse jooksutamiseks vajalikud konteinerid. Nendest on nö igavesti tasuta mahupiirangutega paketis NoSQL (*Non-SQL* või *not only SQL*) andmebaas DynamoDB ja pilvekomponente ühendavad Lambda funktsioonid. Esimese aasta jooksul (*Free Tier*

paketi raames) saab tasuta tarbida AWS'i selgrooks olevat EC2 skaleeruvat konteinerklastrit, RDS andmebaasimoodulit, API'de publitseerimiseks mõeldud API Gateway'd ja paljusid teisi teenuseid. Kõige kiiremaks viisiks on kasutada AWS'i Elastic Beanstalk nimelist toodet, mis EC2 konteinerite instantside baasil rakenduse pilves jooksutamist kõige lihtsamal moel pakub. (Amazon Web Services veebileht, 22.03.2017)

AWS pakub *Free Tier* paketis ka eraldi IoT moodulit, mis hõlmab endas nii pilveplatvormi (IoT Platform) kui ka lokaalselt kontrollmehhanismi (AWS Greengrass), mis võimaldab seadmetel omavahel väga kiirelt sõnumeid vahetada ja ka võrgust väljas olles omavahel suhelda. Joonis 1 kujutab AWS'i IoT Platform mooduli arhitektuurset skeemi. Moodul pakub IoT seadme jaoks arendustööriistu (*AWS IoT Device SDK*), mis lihtsustavad pilveteenusega suhtlemist. Sõnumeid võetakse vastu läbi ühtse autentimise ja autoriseerimise komponendi, mis delegeerib sõnumid edasi seadmete keskusele (*Device Gateway*). See komponent tõlgendab erinevate suhtlusprotokollide (näiteks MQTT, WebSocket või HTTP) põhjal päringuid ning saadab need edasi AWS'i teistele pilveteenustele. Lisaks asub pilves seadmete register (*Registry*), mis hoiustab seadmete unikaalseid identifikaatoreid. Välised rakendused suhtlevad AWS IoT pilvemooduliga läbi rakendusliidese või *Device Shadows*'i (seadme "varjutamine" pilvekoopiana, mis hoiab viimast aktiivset staatust, kuni seade jälle ühenduse saavutab).



Joonis 1. Amazon Web Services IoT mooduli arhitektuurne skeem (Amazon Web Service'i veebileht 22.03.2017).

2.1.8 Google Cloud

Google Cloud on AWS-iga võrreldav mahukas platvorm, milles sisalduvad samuti mitmed alamtooted ja komponendid. Käesolevas projektis olulisteks komponentideks on rakenduste jooksutamiseks mõeldud App Engine, andmete ladustamist võimaldavad NoSQL Cloud DataStore ning SQL andmebaase pakkuv Cloud SQL. Sarnaselt AWS'i Lambdale on ka Google loomas pilvefunktsioonide moodulit nimega Cloud Functions, mis on hetkel veel beta versioonis. Google Cloud pakub alatiseks tasuta lahendusena mahupiirangutega App Engine'i, Compute Engine'i, DataStore'i ja paljude teiste moodulite kasutamist. Lisaks on 12 kuu jooksul võimalik 300-dollarilist krediiti kasutada. (Google Cloud veebileht, 20.03.2017)

Google Cloud pakkus mõnda aega tagasi ka eraldiseisvat IoT moodulit, ent antud töö koostamise ajaks oli see asendunud erinevate väiksemate ja universaalsemate teenustega. (Google Cloud veebileht, 20.03.2017)

2.1.9 IBM Watson IoT

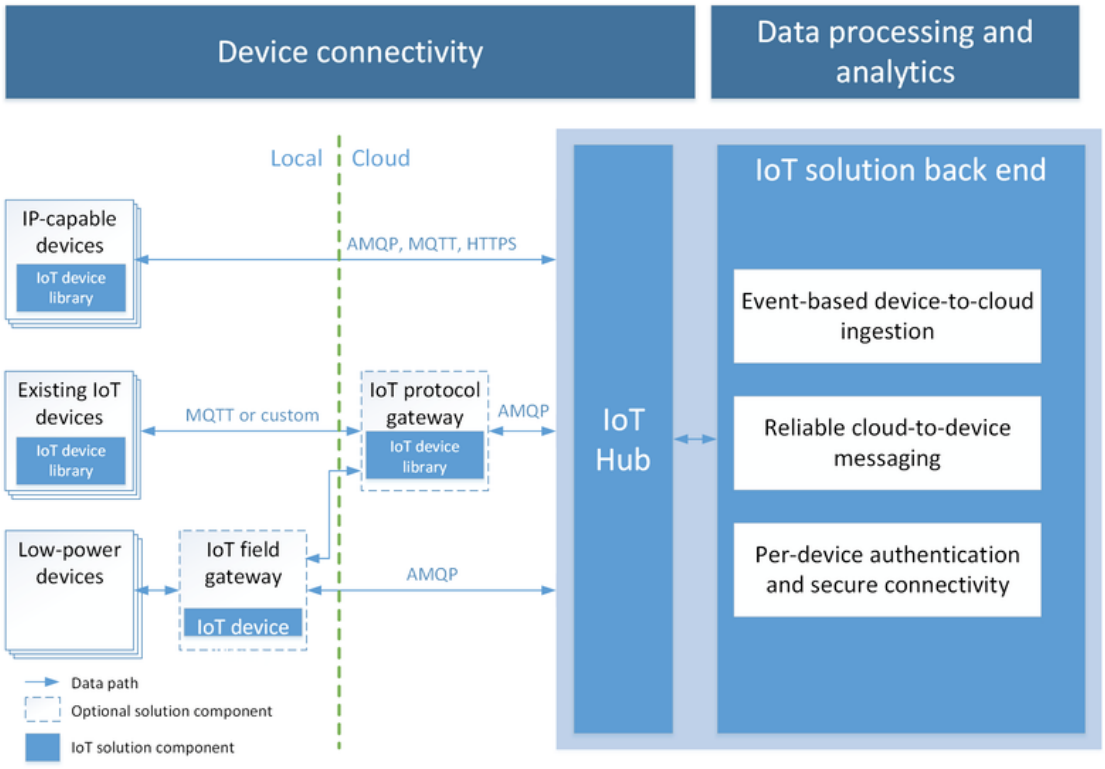
IBM pakub puhtalt IoT-le keskendunud Watson IoT platvormi. Sihtgrupiks on korporatiivsemad ettevõtted ja organisatsioonid ning pakutavad andmemahud on kõrged. Platvormi saab 30 päeva tasuta proovida, mille raames saab kuni 500 seadet ühendada, kuu andmemahu piirang on 200MB. Tegemist on AWS'i IoT moodulile sarnase teenusega, küll veidi lihtsustatud kujul - puudub lokaalne seadmete haldus ja seadmete varjutamise funktsionaalsus. (IBM Watson veebileht, 18.03.2017)

2.1.10 Microft Azure

Microsoft Azure on kolmas suurem teenusepakkuja, mis sarnaselt Google'i ja Amazoni lahendustele katab kõikvõimalikud pilvelahenduse nõuded, pakkudes nii DocumentDB-nimelist NoSQL andmebaasi, SQL andmebaasi, rakenduste jooksutamiseks mõeldud App Service'it ja palju muid komponente. Tasuta paketi raames saab piirangutega kasutada mitmeid teenuseid, sealhulgas App Service'it. Sarnaselt Google Cloud'iga pakub Azure rahalist krediiti tasuliste teenuste proovimiseks, krediidi suuruseks on 170€. (Microsoft Azure veebileht, 22.03.2017)

Azure'is on sarnaselt AWS'ile ja IBM Watsonile eraldi IoT platvorm nimega IoT Hub, mis lubab tasuta baaspaketis 8000 sõnumit päevas edastada. Protokollidena toetatakse

lisaks HTTP'le ka kergemaid - näiteks MQTT või sõnumitele orienteeritud AMQP'd. (Microsoft Azure veebileht, 22.03.2017) Joonis 2 kujutab IoT Hub platvormi arhitektuurset skeemi, millel näidatud IP-võimekusega seadmed võivad otse pilvemooduliga suhelda, kasutades HTTP, AMQP või MQTT protokolle sõnumite saatmiseks. Madala energiatarbivusega ning eksisteerivad IoT seadmed ühenduvad pilvega läbi *IoT protocol Gateway* komponendi, mis MQTT või mõne muu suhtlusprotokolli kaudu infot vastu võtab ning selle AMQP protokolli kaudu pilveplatvormi keskusesse edastab. Madala energiatarbivusega seadmete jaoks on kohalikus keskkonnas vaja kasutada komponenti *IoT field Gateway*, mis sõnumid otse IoT keskusesse (*IoT Hub*) või *IoT Protocol Gateway*'le edastab. IoT keskus tõlgendab saadud sõnumeid ning edastab need teistele pilvekomponentidele.



Joonis 2. Microsoft Azure IoT Hub arhitektuurne skeem (Microsoft Azure veebileht, 22.03.2017).

2.1.11 RedHat OpenShift (Next Generation Developer Preview)

RedHat pakub OpenShift nimelist PaaS teenust, mis põhineb Docker'i tarkvarakonteinerite platvormil. Tarkvarakonteiner on isoleeritud arenduskeskkond, mis toimib nagu virtuaalmasin, aga ei sisalda täielikku operatsioonisüsteemi, vaid ainult rakenduse jooksmiseks vajalikke teeke ja seadeid. (Docker veebileht, 28.04.2017)

OpenShift v2 ehk RedHat'i eelmise põlvkonna pilveteenus on hetkel küll toimiv, aga uusi kasutajaid sinna alates 2016. aasta augustist juurde luua ei saa. Hetkel on Next Generation Developer Preview nimeline versioon küll juba avatud, aga selle valmidus on hetkel pigem poolik. Developer Preview versioon võimaldab tasuta kolme tarkvarakonteineri (nimetatud ka *Gear*'iks) jooksutamist, erinevate andmebaaside seadistamist ning mahupiirangute raames on võimalik rakendusi sellel platvormil täiesti tasuta jooksutada. (RedHat OpenShift veebileht, 24.03.2017)

2.1.12 Heroku

Heroku on esimene väiksem PaaS'i pakkuja, reklaamides end kui tarkvarale, mitte infrastruktuurile keskenduvat teenust. AWS'i, Google Cloud'i ja Azure'iga võrreldes on Heroku palju kergekaalulisem ja lihtsam hallata. Baaspakett sisaldab ühe väiksema ressursiga virtuaalmasina (nimetatud ka *Dyno*'ks) käitamist, milles saab mahupiirangutega enamlevinud programmeerimiskeeltes veebirakendust jooksutada ning lisadena kataloogist erinevaid mooduleid (näiteks SQL või NoSQL andmebaas või monitooringu pakett) juurde tellida. Lisade hinnapoliitika pole Heroku enda defineerida, kui lisa autoriks on kolmas osapool, ent enamjaolt oli lisadel olemas esimese astme tasuta pakett, mis rakenduse arendamisel ja testimisel piisav peaks olema. (Heroku veebileht, 21.03.2017)

2.1.13 Nanobox

Nanobox on paljuski sarnane Herokule, pakkudes samuti lihtsamat PaaS-lahendust, mille baaspaketis sisaldub ühe väiksema ressursiga virtuaalmasina jooksutamine. Vastav pakett on tasuta, kui tegemist on isikliku (või avatud lähtekoodiga) projektiga. Kommertskasutuseks on baaspaketi hind 9 dollarit kuus. Nanobox'i eristab lisaks pilveplatvormile ka lokaalse arenduskeskkonna pakkumine, mis sisuliselt on Docker'i baasil ehitatud tarkvarakonteinerite süsteem. (Nanobox veebileht, 22.03.2017)

2.2 Prototüübi testimiseks kasutatud seadmed, sensorid ja tarkvara

Käesolevas projektis loodava prototüübi testimiseks kasutati füüsilist seadet, mis edastas niiskus- ja temperatuurisensorilt kogutud infot. Füüsilise seadme riistvaralisteks komponentideks olid Espressif Systems'i ESP8266 WiFi moodul (Espressif Systems ESP8266EX WiFi mooduli spetsifikatsioon, 01.05.2017) ning Texas Instruments'i

MSP430F5529 mikrokontroller koos vastava platvormiga (LP ehk *LaunchPad)*. (Texas Instruments MSP430F5529 LaunchPad spetsifikatsioon, 01.05.2017) Sensorina oli kasutusel Aosong Electronics'i poolt toodetud DHT22, mis mõõdab nii temperatuuri kui ka suhtelist õhuniiskust. (Aosong Electronics DHT22 sensori spetsifikatsioon, 01.05.2017) Seade edastas üle HTTP protokolli enda unikaalse identifikaatori, sensorilt mõõdetud temperatuuri ja niiskuse väärtused ning neile vastavad andmetüübid POST päringuna.

Alternatiivseks testimiseks kasutati programmi Postman, mida on saadaval enamlevinud operatsioonisüsteemidele ning mis graafilise kasutajaliidese abil võimaldab HTTP päringuid teha ning hilisemaks kasutamiseks salvestada. (Postman veebileht, 01.05.2017) Käesoleva rakenduse testimisel simuleeriti Postman'i abil nii seadmetelt saadetava andmete salvestamise päringut kui ka teisi veebirakenduse serveripoolseid funktsionaalsusi.

2.3 Kasutatavus, kasutajasõbralikkus ja kasutajakeskne disain

Kaasaegsete rakenduste loojad puutuvad üha enam kokku kasutajasõbralikkuse ja kasutatavuse mõistetega. Erinevatelt seadmetelt tarbitava digitaalse informatsiooni kogus üha kasvab, mistõttu on kasutajate nõudmised meeldivate, loogiliste ja arusaadavate liideste järele suurenenud. Tõenäoliselt on Jakob Nielseni lahtikirjutatud kasutatavuse mõiste üks tuntumaid, koondades enda alla õpitavuse (ingl *learnability*), meeldejäävuse (ingl *memorability*), tõhususe (ingl *efficiency*), eksimused (ingl *errors*) ja rahulolu (ingl *satisfaction*). (Nielsen, 1993) Ajalooliselt on kasutusmugavus pärit eelkõige sõjatööstusest, kus sõjatehnika intuitiivsemaks ja ergonoomilisemaks muutmine vähendas lahingus tehtud vigu. (Shackel jt 1991) Rahvusvahelise standardiorganisatsiooni (ISO 9241-11) põhjal võib kasutatavust defineerida läbi kasutaja eesmärkide efektiivse (ingl *effectiveness*), rahulolu pakkuva (ingl *satisfaction*) ja ressursisäästliku (ingl *efficiency*) rahuldamise. Rahvusvahelisele standardi definitsioonile on eelnevalt viidanud oma magistritöödes ka Mihkel Uukkivi (Uukkivi 2006, 11), Kadi Lauk (Lauk 2015, 12) ja Oliver Ainsalu (Ainsalu 2016, 14).

Kasutajasõbralik disain peaks olema seega kasutatav, pakkudes kindlasti rahulolu, olles efektiivne ning võimalikult intuitiivne, et kasutaja ei eksiks. Kasutajasõbraliku disaini saavutamiseks rakendatakse erinevaid kasutajakeskse disaini tehnikaid.

2.4 Kasutajakeskne disain ja disainiprotsessi tehnikad

Kasutajakeskne disain (ingl *user-centered design*) on olemuslikult lähenemine, mida on varasemalt paljude erinevate nimetustega kutsutud, näiteks ergonoomika, kasutatavuse konstrueerimine (ingl *usability engineering*) ja inimfaktorite konstrueerimine (ingl *human factors engineering*). Kasutajakeskne disain esindab tehnikaid, protsesse, meetodeid ja protseduure kasutatavate toodete ja süsteemide disainimiseks, pannes kasutaja kogu protsessi keskmesse. (Rubin jt 2008, 12)

Kasutajakeskse disainiprotsessi meetodeid ja tehnikaid on mitmeid ning nende rakendamine toimub tootearenduse erinevates etappides. Etnograafiline uuring kätkeb endas kasutaja süvaanalüüsi vaatlustehnika baasil. Sealjuures on oluline jälgida kasutajaid keskkonnas, kus nad toodet kasutaksid ning tuvastada kasutaja eesmärgid ja ülesanded, mille baasil luuakse kasutajate kohta üldistavaid profiile. Fookusgrupid (ingl *focus groups*) ja kaasav disain (ingl *participatory design*) kaasavad lõppkasutajaid juba arenduse algusfaasides disaini valideerima - esimesel juhul selleks mõeldud sessioonides ning teisel juhul pühendunud tiimiliikmena. Uuringud aitavad suurema valimi peal mõista kasutajate eelistusi. Läbikäigud (ingl *walk-through*) ja paberprototüüpimine aitavad väga tõhusalt ja väikese vaevaga prototüübi algusfaasides kasutajate tagasiside põhjal suunamuutusi teha ning komponente valideerida. Andmete leitavuse hindamiseks kasutatakse kaartide sorteerimise tehnikat ning viimaks võib disaini valideerida heuristilise hindamise või kasutajatel testimise baasil (Rubin jt 2008, 16-20).

Käesoleva töö raames rakendati ülaltoodud tehnikatest kaasavat disainimist, paberprototüüpimist ning kasutajatel testimist. Vastavad tehnikad sobitusid probleemi konteksti kõige paremini ning kombinatsioonina katsid need meetodid erinevaid disaini etappe, hoides kasutajat pidevalt protsessi keskmes.

Kaasava disaini rakendamine tähendab, et üheks disainimeeskonna liikmeks peaks olema lõppkasutaja ise, kes on kaasatud projekti algusest peale ning kes panustab oma teadmiste, oskuste ning isegi emotsioonidega. Tihti kasutatakse seda tehnikat organisatsioonisiseste süsteemide loomisel. Antud tehnika ohuks on kasutajate liigne lähedus disainitiimile, jättes seetõttu vajalikul määral disaini kritiseerimata. (Rubin jt 2008, 17)

Paberprototüüpimine on, nagu nimigi ütleb, paberi abil disainiprototüübi loomine ning sellele kasutajatelt tagasiside küsimine küsimuste või muude tegevuste abil. Paberprototüübiga võib kujutada ühte kindlat süsteemi vaadet või kitsamat moodulit, mille kohta tagasisidet soovitakse. Paberprototüübi abil saab vaadete järjestust ning mingi funktsionaalsuse voogu kiirelt ja lihtsa vaevaga testida. Paberprototüüp ei pea, aga võib olla interakteeruv, see tähendab, et testimise läbiviija võib kasutaja tegevuste peale vaateid muuta või sellel olevaid elemente liigutada, lisada või ära võtta. (Rubin jt 2008, 18-19)

Kasutajatel testimine valideerib olemasolevat valmistoodet või selle prototüüpi, andes kasutajatele täitmiseks ülesanded. Testimise läbiviija jälgib kasutajate tegevusi ning kogub tagasisidet kasutajate õnnestumiste ja ebaõnnestumiste kohta ülesannete täitmisel. Seda tehnikat saab kombineerida teiste meetoditega (näiteks ka paberprototüübi puhul võib kasutajatele anda ülesanded, mitte küsida küsimusi) ning kasutajatel testimist võib läbi viia iteratiivselt, toodet või selle prototüüpi järk-järgult parendades. (Rubin jt 2008, 19-20) Antud projektis rakendatakse kasutajatel testimist nii paberprototüübil kui ka digitaalsel prototüübil.

Lisaks eeltoodule kasutatakse disaini valideerimiseks ning täiendavate ideede kogumiseks disainikriitika koosoleku metoodikat, mida kasutab aktiivselt AirBnB disainitiim. (Spec 73, 2015) Selle meetodiga kaasatakse inimesi väljaspoolt senist disainiprotsessi kaasa mõtlema, konstruktiivselt kritiseerima ning arusaamatuste puhul küsimusi küsima. Disainikriitika koosolek on heaks viisiks saada värskemaid ja nö kastist-väljas mõtteid loodava lahenduse kohta. Disainikriitika koosolekul presenteeritakse lahenduse ideed väljaspool senist protsessi olevale huvigrupile, milles iga liige kirjutab oma tagasiside erinevate kategooriate - ideed, küsimused, miinused ja plussid - paberitele. Vastavad märkmed koondatakse ühiseks tabeliks loetud kategooriate alla. Saadud märkmetabeli punkte analüüsitakse ja arutatakse, mille tulemusena saadakse komplektne tagasiside loodud lahendusele. (Spec 73, 2015)

2.5 Iteratiivne disainimudel

Lisaks kasutajakesksele lähenemisele on disainiprotsessi õnnestumiseks oluline jälgida muidki tegureid. Gould ja Lewis toovad välja kolm printsiipi, mida peaks edukas disainiprojektis jälgima: (Gould jt 1985)

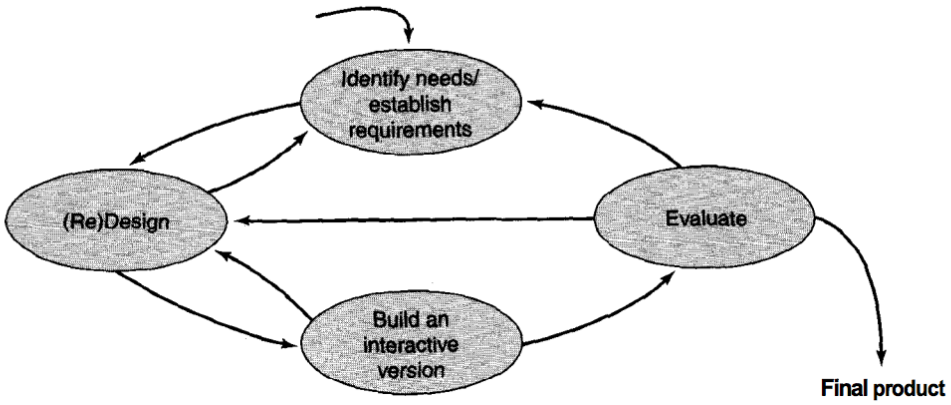
1. Varajane fookus kasutajatele ning nende eesmärkidele ja ülesannetele
2. Empiiriline mõõtmine
3. Iteratiivne disain

Sisuliselt viitavad Gould ja Lewis tõdemusele, et süsteemide loomisel on keeruline asju ühe korraga õigesti teha, sest üldjuhul ei oska ka lõppkasutajad ise enda reaalseid vajadusi täiuslikult täpselt kirjeldada. See tähendab, et eriti keerukamate süsteemide või funktsionaalsuste puhul on mõistlik loodav disain juba algusfaasis prototüübina korduvalt läbi testida, sealjuures kasutajate reaktsioone jälgida ning vastavalt tagasisidele kohe vajalikke muudatusi teha. Sellist protsessi võib nimetada iteratiivseks, sest korduvate tsüklitena luuakse, testitakse ja valideeritakse ning seejärel parendatakse disaini. Iteratiivse disaini mudel kirjeldab nelja põhilist tootearenduse etappi, millest disainimisel lähtuma peaks: (Rogers jt 2011, 186)

1. Vajaduste identifitseerimine ja nõuete fikseerimine
2. (Korduv) Disainimine
3. Interaktiivse versiooni ehitamine
4. Hindamine

Üldjuhul algavad projektid esimesest punktist ehk esmalt sätestatakse nõuded kasutajate vajadusi arvestades, seejärel luuakse üks või mitu disainikavandit ning vastavad kavandid viiakse interaktiivsete prototüüpide tasemele, et saaks neid hinnata ning testida. Testimiselt saadud tagasiside põhjal võib protsess jätkuda vajaduste ja nõuete korrigeerimisega või otse korduvdisainimise etapis (Rogers jt 2011, 186).

Joonis 3 illustreerib kirjeldatud mudelit, milles ülevalt alustades ning vastupäeva liikudes on kujutatud neli peamist protsessi etappi: vajaduste identifitseerimine ja nõuete fikseerimine, (korduv) disainimine, interaktiivse versiooni ehitamine ning hindamine. Samuti kujutatakse joonisel projekti liikumist erinevate etappide vahel projekti algusest kuni lõpliku toote valmimiseni.



Joonis 3. Lihtne iteratiivse disaini mudel (Rogers jt 2011, 186).

2.6 Agiilne tarkvaraarendusprotsess

Sarnaselt disainiprotsessile on ka tarkvaraarenduse metoodikad liikunud iteratiivse mudeli juurde ning soovitanud ehitada väiksemaid tükke korraga. Kuigi sellised agiilsed mudelid erinevad nii praktika kui ka rõhuasetuse poolest, jagavad nad kõik mõningaid ühiseid karakteristikuid, näiteks iteratiivne arendus, kommunikatsioonile keskendumine ning ressursimahukate tugielementide vähendamine. (Cohen jt 2003, 12) Sarnaselt sõnastatakse agiilse arendusmetoodika põhitõed ka Agiilse Tarkvaraarenduse Manifestis, hinnates inimesi ja nendevahelist suhtlust rohkem, kui protsesse ja arendusvahendeid; töötavat tarkvara rohkem, kui kõikehõlmavat dokumentatsiooni; koostööd kliendiga rohkem, kui läbirääkimisi lepingute üle; reageerimist muutunud oludele rohkem, kui algse plaani järgimist. (Beck jt 2001) Traditsiooniliste tarkvaraarenduse viiside puhul algas töö kõikide nõuete täielikust kaardistamisest ja dokumenteerimisest, millele järgnes arhitektuuri ja süsteemi disainimine, arendamine ja inspekteerimine. (Cohen jt 2003, 2-4) Agiilsed metoodikad on vastuseks traditsioonilistele tarkvara arendamise viisidele, teadvustades vajadust alternatiivsele lähenemisele, mis ei seaks dokumentatsiooni esikohale ning ei oleks olemuselt "raskekaaluline" arendusprotsess. (Beck jt 2001)

Üheks enimtuntud ja tänasel hetkel rohkem praktiseeritud agiilseks arendusmetoodikaks on Scrum, mis aktsepteerib tõsiasja, et arendusprotsess on etteaimamatu. Scrum'i ideoloogia põhineb sprintidel, mis on kindla ajaraamiga arendustsüklid, mida eraldi planeeritakse ja mille lõpus toimub tagasivaade tehtud tööle. (Cohen jt 2003, 14-15) See tagab väiksemate pakettidena tehtavad muudatused, mida on lihtsam planeerida ja

hallata, vähendades tõenäosust eksida spetsifikatsioonis või implementatsioonis. Lisaks sprintide iteratiivsele olemusele on Scrum'is ka väiksem tsükkel, mille pikkuseks on 24 tundi. Iga päev toimub Scrum'i tiimis 15-minutiline koosolek, mis nõuab vastuseid igalt liikmelt kolmele küsimusele: mida oled teinud alates viimasest sarnasest koosolekust, kas sinu tegevusel on takistusi ning mida plaanid teha järgmise koosolekuni? (Cohen jt 2003, 14) Igapäevane sünkroniseerimine tagab parema kommunikatsiooni tiimiliikmete vahel ning aitab hoiduda takistustest ning olla maksimaalselt produktiivne. Eriti oluline on kommunikatsioon suuremates meeskondades, kus on keeruline kõikide tegemistega kursis olla.

Teiseks agiilseks metoodikaks, mida on peetud ka kõige populaarsemaks, on *Extreme Programming*, lühendina XP. Sarnaselt Scrum'ile soovitab XP jagada süsteemi väljalasked (ingl *release*) väiksemateks tükkideks ning viia läbi nö "planeerimismäng" iga iteratsiooni alguses, mille raames identifitseeritakse nõuded, luuakse nendest kasutuslood, mida hinnatakse ja prioritiseeritakse ning planeeritakse järgmise iteratsiooni skoop. XP defineerib veel palju muid juhiseid nagu näiteks paarisprogrammeerimine, pidev refaktoreerimine ning testide kirjutamine enne koodi kirjutamist (ingl *test-driven development*), mida tasuks eriti jälgida suuremates projektides, mille kood kipub tihti keerukaks ja arusaamatuks muutuma. Kokku defineerib XP 12 reeglit, mis peaksid aitama arendustiimil projekt edukalt läbi viia. (Cohen jt 2003, 12-13)

Lisaks Scrum'ile ja XP'le leidub palju vähem praktiseeritud metoodikaid nagu *Lean Development*, *Feature Driven Development* või *Crystal Methods*, millel aga pikemalt ei peatuta. Käesolevas projektis lähtutakse agiilsete arendusmetoodikate ühisosaks, milleks on iteratiivne arendus, efektiivne kommunikatsioon ning muu üleliigse vähendamine, mis liigselt aega vi energiat võiks võtta.

3 Pilveplatvormide analüüs ja võrdlus

Töö teoreetilises osas toodi välja erinevad pilveteenused, mis võimaldaksid ülesandepüstituses sätestatud probleemi lahendada - koguda sensoritelt tulevaid andmeid, neid pilves hoiustada ning kasutajale atraktiivselt ja arusaadavalt välja kuvada. Sellised pakkujad jagunesid peamiselt kaheks.

Esimesse gruppi liigituvad erinevad PaaS teenused, mis rohkemal või vähemal määral koodi kirjutamise vajaduse elimineerivad, olles spetsiifilisemalt mõeldud just IoT projektide loomiseks, haldamiseks ja monitoorimiseks. Sellisteks platvormideks on näiteks ThingSpeak, Kaa, BeeBotte jt. Teise gruppi kuuluvad univeraalsemad ja tihti rohkemaid võimalusi pakkuvad teenused, mida võib liigitada sõltuvalt paketist või alamtootest nii IaaS kui ka PaaS pakkujate alla. Universaalsete suuremate teenuste hulka kuuluvad kindlasti Amazon Web Services, Microsoft Azure ja Google Cloud, väiksemate ja lihtsamate sekka liigituvad Heroku ja Nanobox.

3.1 Mittefunktsionaalsed nõuded

Pilveplatvormide võrdlemiseks, valideerimiseks ning järjestamiseks sätestati järgnevad süsteemsed mittefunktsionaalsed nõuded.

1. Süsteem peaks lubama andmekogumist vähemalt 5 seadmelt
2. Süsteem peaks vastuvõetud sõnumeid salvestama ja hoiustama vähemalt 1 aasta jooksul
3. Süsteem peaks võimaldama päringute vastuvõtmist minimaalselt 15-sekundilise intervallina (ehk minimaalselt 5760 sõnumit seadme kohta päevas)
4. Süsteem peaks toetama SSL'i (*Secure Sockets Layer*, transpordikihi turbeprotokoll, mis tagab kommunikatsiooni turvalisuse arvutivõrkudes)
5. Teenuse hind peaks olema tugeva eelistusena tasuta, alternatiivselt kuni 5 dollarit kuus

6. Süsteem peaks võimaldama andmete privaatset salvestamist (seadmelt tulev andmevookanal ei tohiks olla avalik)

7. Süsteemi haldamine peaks olema lihtne, loogiline ja kiirelt õpitav

8. Süsteem peaks võimaldama atraktiivset (või atraktiivseks muudetavat) andmete kuvamist

Ülaltoodud nõuded on sätestatud eelkõige valmislahenduste ehk IoT'le spetsialiseerunud teenuste valideerimiseks, ent teatud kohandustega lähtuti nendest nõuetest ka universaalsete platvormide analüüsis. Universaalsete platvormide võrdlemisel ja analüüsimisel pole mõtet kõiki baasnõudeid rakendada, sest näiteks loodava lahenduse atraktiivsus ei sõltu sellisel juhul platvormist, vaid sellel jooksvast rakendusest. Autori ekspertiisist tingituna lisandub universaalsete platvormide nõuetesse ka programmeerimiskeelte tugi, milles loodav rakendus võib olla kirjutatud. Sarnaselt IoT-le keskenduvate platvormide võrdlusele on viimaseks nõudeks lihtsus ja õpitavus, mida 3-pallisüsteemis hinnatakse. Kohaldatud nõuded universaalsetele platvormidele on järgmised:

1. Teenus peaks pakkuma tasuta paketti, mis võimaldaks vähemalt 28 800 päringu vastuvõtmist - vähemalt 5 seadmelt minimaalselt 5760 sõnumit seadme kohta päevas

2. Süsteem peaks toetama SSL'i (*Secure Sockets Layer*)

3. Süsteem peaks toetama rakenduse jooksutamisel järgmisi programmeerimiskeeli: PHP, Node.js või Java

4. Süsteemi haldamine peaks olema lihtne, loogiline ja kiirelt õpitav

3.2 IoT andmete kogumisele spetsialiseerunud platvormid

Spetsiifilisemaid teenuseid, mis keskenduvad vaid IoT-le, võib pidada kergemini üles seatavateks ning nende õpikõver ei ole liiga järsk. Samas tähendab see, et selliste platvormide puhul on kasutajal konfigureerimise ja kohaldamise võimalused väiksemad. Rakenduse kirjutamise vaev küll puudub, aga ühtlasi tähendab see, et teenuse või platvormi kulukus on selle võrra suurem. Testprojektiks pakutakse kõikides loetletud

spetsialiseerunud platvormides piiratud mahuga tasuta paketti, aga lõpliku lahenduse loomiseks kippusid vastavad paketid või platvormi enda võimalused olema ebapiisavad. Alljärgnevalt on toodud spetsialiseerunud platvormide peamised tugevused ja nõrkused ning nimekirja lõpus on toodud platvormide vastavus baasnõuetele.

3.2.1 ThingSpeak

Tugevused:

- Väga lihtne ja kiire üles seada
- Palju integratsioone ja võimalusi andmetele reageerimiseks
- Andmete töötlemise võimalus MatLab'is

Nõrkused:

- Väga primitiivne ja väheatraktiivne andmete kuvamine
- Tasuta pakett lubab ühel seadmel maksimaalselt ligi neli korda minutis andmeid edastada, rohkemate seadmete puhul jaguneb intervall veelgi pikemaks.

3.2.2 Kaa

Tugevused:

- Avatud lähtekood ja väidetavalt täiesti tasuta rakendus
- Võimalus rakenduse koodi ise oma vajaduste järgi kohaldada

Nõrkused:

- Ülesseadmine nõuab rohkem tehnilisi teadmisi ja konfigureerimist
- Rakenduse pilves jooksutamiseks peab ise eraldi AWS'is konto ja rakenduse konteineri looma
- Sisuliselt ei pakuta tootega *hosting*'ut, mistõttu võib öelda, et Kaa pole päris tasuta PaaS

3.2.3 Dweet.io

Tugevused:

- Äärmiselt lihtne süsteem, mille ülesseadmine pole mingi vaev
- Meeldiv keskkond ja keskmisest atraktiivsem liides

Nõrkused:

- Kanali privaatseks muutmine on tasuline
- Seadme unikaalse nime reserveerimine on tasuline
- Andmeid hoiustatakse vaikimisi vaid 24 tundi, tasulises paketis kuni 30 päeva

3.2.4 BeeBotte

Tugevused:

- Erinevad andmete kuvamise ja vormindamise võimalused
- Tasuta plaan lubab lõpmatu arvu kanaleid
- Tasuta sõnumite arv - kokku lubatakse saata 50 000 sõnumit päevas

Nõrkused:

- Kõiki sõnumeid ei salvestata, tasuta plaan lubab vaid 5000 sõnumi salvestamist päevas (ehk ligi 20 sekundiline intervall, kui kasutada ühte seadet/kanalit)
- Andmetele reageerimiseks ja muuks lisakonfiguratsiooniks on vähe võimalusi

3.2.5 Carriots

Tugevused:

- Palju lisakonfiguratsiooni võimalusi
- Võimalik kirjutada sisuliselt oma *back-end*

Nõrkused:

- Tasuta pakett võimaldab vaid kahe seadme kasutamist
- Mahukatele päringutele orienteeritud - maksimaalne päringute lubatud arv on madal (500 päevas)

3.2.6 Ubidots

Tugevused:

- Suur lubatud seadmete arv (tasuta paketis kuni 20 seadet)
- Parim andmete saatmise lubatud intervall (1440 päringut päevas seadme kohta)
- Palju erinevaid andmete kuvamise ja vormindamise võimalusi

Nõrkused:

- Andmete hoiustamine kuni kolm kuud
- Mahukam (korporatiivsem) ja seetõttu keerukam süsteem

3.2.7 Spetsialiseerunud platvormide vastavus baasnõuetele ja võrdluse kokkuvõte

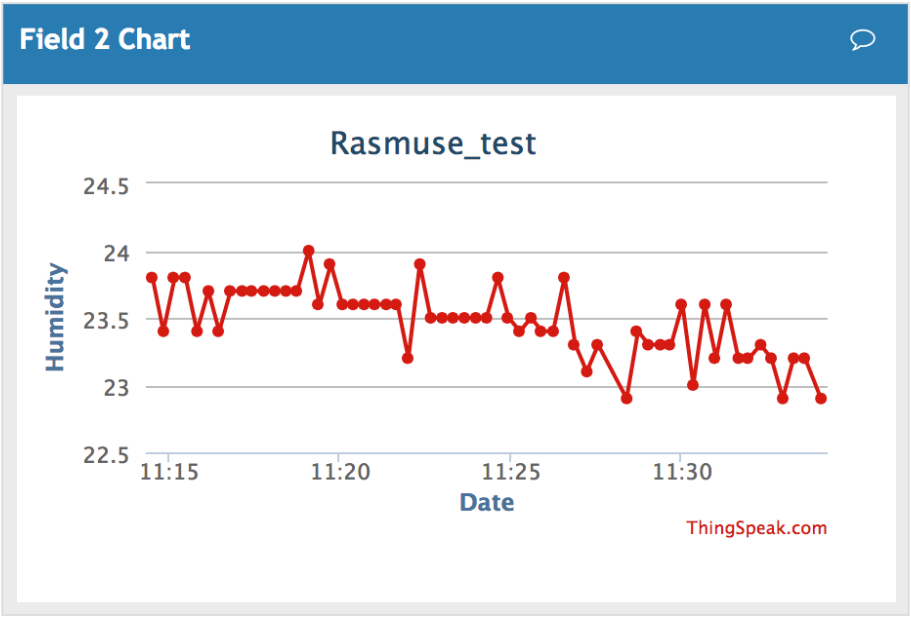
Võrreldud platvormide seas olid mitmed põnevad ja võimekad tooted, mis võiksid kiiremaks katsetamiseks ja seadmete testimiseks väga head lahendused olla. Samas kõikidele baasnõuetele ei vastanud ükski toode. Paljude platvormide puhul pakuti võrdlemisi lühikest andmete hoiustamise aega, mis jäi tihti 3 kuu või poole aasta piiresse. Teiseks nõudeks, millele leidus vähe vasteid, oli andmete saatmise mahupiirangud, mis seadmete või päringute arvult olid pigem piiravad. Tasuliste lahenduste puhul oleks olnud võimalik mahupiirangutesse edukamalt mahtuda, mistõttu võib öelda, et kui ligi 10 dollarit kuus on kasutaja jaoks vastuvõetav, siis leidub ka valmislahendusi, mis kasutaja ootustele võiks vastata. Käesolevas töös polnud aga selline lahendus aktsepteeritav.

Tabelis 1 on toodud spetsialiseerunud platvormide võrdlus ning vastavus baasnõuetele. Nõudeid 1...6 hinnati jah/ei (+/-) väärtustena, nõue 5 ehk hind oli rahuldatud, kui platvorm võimaldas tasuta paketti. Nõudeid 7 ja 8 hinnati 3-pallisüsteemis, kus 3 pall tähistab kõige paremat ning 1 pall kõige kehvemat tulemust.

Tabel 1. IoT andmete kogumisele spetsialiseerunud platvormide võrdlus.

	ThingSpeak	**Kaa**	**Dweet.io**	**BeeBotte**	**Carriots**	**Ubidots**
(1) Vähemalt 5 seadet	-	+	+	+	-	+
(2) Sõnumite hoiustamine 1 aasta jooksul	+	+	-	-	-	-
(3) Intervall min 15s (5760 / seade / päev)	-	+	+	-	-	+
(4) SSL	+	-	+	+	+	+
(5) Hind (tasuta)	+	-	+	+	+	+
(6) Andmete privaatsus	+	+	-	+	+	+
(7) Lihtsus	***	*	***	**	**	*
(8) Atraktiivsus	*	*	**	**	**	**

Enamik lahendusi pakkusid väga minimalistlikku andmete kuvamist, mis tihti oli lahendatud lihtsa graafiku kujul. Et töö eesmärgiks oli luua prototüüp, mis andmeid atraktiivsemalt reaalajas välja suudaks kuvada, ei vastanud kesise kasutatavusega valmislahendused paljuski ootustele. Joonis 4 kujutab ekraanipilti ühest ThingSpeak platvormi andmekogumise graafikust, mis visualiseerib viimase 20 minuti jooksul kogutud andmeid niiskust mõõtvalt sensorilt.



Joonis 4. Ekraanipilt ThingSpeak platvormi andmekogumise ja -visualiseerimise graafikust (ThingSpeak'i veebileht, 17.03.2017).

Kokkuvõttes ei leitud analüüsi käigus ideaalselt sobivat valmistoodet, mistõttu võrreldi lisaks eksisteerivatele spetsialiseerunud teenustele ka universaalsemaid platvorme. Seda otsust toetas ka pikema perspektiivi plaan, mille eesmärgiks oli prototüübi põhjal eraldiseisev toode luua, mis IoT turuprognoosi põhjal võiks äriliselt edukas olla.

3.3 Universaalsed pilveplatvormid

Universaalsete pilveplatvormide puhul tuli analüüsida iga juhtumit eraldi, sest pakutavad lahendused on kohati väga erinevad, nagu ka vastavate platvormide eesmärgid. Ühe grupina võib eristada väiksemaid PaaS teenuseid, mis pakuvad lihtsat veebirakenduse jooksutamist ning lisamoodulite kaudu ka andmebaasi ja muude

pilvefunktsioonide võimalust. Sellised platvormid on näiteks Heroku ja Nanobox. Teise grupi alla võiks liigitada kõik ülejäänud PaaS ja IaaS lahendused - näiteks Amazon Web Services, Google Cloud, Microsoft Azure, RedHat OpenShift jt. Viimaste puhul tasub eristada neid teenuseid, mis pakuvad ka eraldi IoT moodulit, mistõttu on rakenduse loomiseks võimalik tootesiseselt erinevaid arhitektuurseid lahendusi luua. Näiteks AWS'is võib kasutada seadmete halduseks IoT moodulit ning see siduda läbi Lambda funktsioonide DynamoDB-ga. Tulemuseks on prototüübi esimene pool, mis võimaldab seadmetelt andmeid koguda ja neid pilves andmebaasi salvestada. Alternatiivina saaks aga kasutada EC2'l jooksvat Elastic Beanstalk'i API loomiseks, mida saaks samuti DynamoDB-ga ühendada ning kuhu seadmed saaksid üle HTTP andmeid saata. Seega on suuremate platvormide puhul prototüübi loomiseks erinevaid arhitektuurseid võimalusi.

3.3.1 Amazon Web Services

Amazon Web Service'i selgrooks on EC2-nimelised virtuaalkonteinerid, milles saab pilverakendusi jooksutada. Staatilist salvestusruumi haldab S3 komponent. Rakenduste loojatele pakutakse tasuta Elastic Beanstalk'i moodulit, mis sisuliselt kasutab S3 salvestusruumi ning EC2 instantse rakenduse jooksutamiseks. AWS'i nö ajutiselt tasuta paketi (*Free Tier*) raames saab mõningaid tasulisi teenuseid 12 kuu jooksul mahupiirangutega ilma rahata kasutada, lisaks on mõned komponendid väiksema mahukasutuse puhul alati tasuta. Tabelis 2 on toodud kirjeldatud AWS'i sobivaimate komponentide hinnad ja mahupiirangud. Kuna AWS pakub SQL andmebaasi tasuta vaid ajutiselt tasuta paketis, aga NoSQL baasi saab alati tasuta tarbida, siis on AWS'is mõistlik andmebaasina NoSQL DynamoDB komponenti kasutada. Elastic Beanstalk'i jooksutamiseks vajalikud S3 ja EC2 ning alternatiiviks olev IoT moodul on kõik ajutiselt tasuta paketis, mistõttu pole AWS'is võimalik päris nullkuludega igavesti rakendust jooksutada. Küll aga esimese 12 kuu jooksul.

Tabel 2. AWS'i arhitektuursete komponentide hind ja mahupiirangud.

Komponent	**Hind**	**Mahupiirang**
DynamoDB	**Tasuta** (alati)	200 miljonit päringut kuus, 25GB salvestusruumi
EC2	**Tasuta** (Free Tier)	750 tundi kuus
S3	**Tasuta** (Free Tier)	5 GB salvestusruumi
IoT	**Tasuta** (Free Tier)	250 000 sõnumit kuus

3.3.2 Google Cloud

Google Cloud on küll sarnaselt Amazon Web Service'ile väga suur platvorm, aga seal rakenduse loomiseks ja jooksutamiseks piisab tegelikult kahest komponendist. Tabelis 3 on loetletud sobivaimad Google Cloud komponendid hindade ja mahupiirangutega. Cloud Datastore on NoSQL andmebaas, mida tasub SQL baasidele eelistada just hinna tõttu - SQL baasi jooksutamine võib minimaalsete mahtude korral maksta ligi 130 dollarit aastas, Datastore'i saab aga tasuta kasutada. Teiseks vajalikuks komponendiks on App Engine, mis ühendab endas kõik vajaliku rakenduse jooksutamiseks. App Engine on samuti mahupiirangute raames tasuta kasutatav. Lisaks pakub Google esimeseks aastaks 300 dollari suurust krediiti, mida saab soovi korral tasulistele komponentidele kulutada.

Tabel 3. Google Cloud'i arhitektuursete komponentide hind ja mahupiirangud.

Komponent	**Hind**	**Mahupiirang**
Cloud Datastore	**Tasuta**	1GB salvestusruumi, 50 000 lugemist, 20 000 kirjutamist, 20 000 kustutamist päevas
App Engine	**Tasuta**	28 tundi päevas, 5GB salvestusruumi

3.3.3 IBM Watson IoT

IBM'i pilveplatvormist on mistahes rakenduse jooksutamine vaid esimesed 30 päeva tasuta. Teoreetiliselt saaks IBM'i teenuses ühendada IoT mooduli, konteinerite süsteemi ning MongoDB andmebaasi, aga kindlasti osutuks see juba tasuliseks (ja sealjuures

küllaltki kalliks), mistõttu jääb edasisest analüüsist see platvorm välja. IBM keskendub oma platvormiga just suurematele klientidele, kelle andmemahud pole käesoleva projektiga võrreldavad.

3.3.4 Microsoft Azure

Azure'i rakenduste mootoriks on App Service, mida piiratud mahus tasuta tarbida saab. Kahjuks on aga mõlemad andmebaasimoodulid (NoSQL ja SQL) tasulised, kusjuures suurele võimsusele orienteeritud NoSQL DocumentDB moodulit pakutakse üsna suurte mahupiirangutega, mistõttu on selle hind isegi kallim, kui väikseimal SQL andmebaasi instantsil. IoT Hub on samuti tasuta, aga etteruttavalt öeldes ei vastaks selle mahupiirang (8000 sõnumit päevas) loodava lahenduse baasnõuetele, milleks on ligi 30 000 päringut päevas. Seega on ka Azure'il soovitud rakenduse loomine tasuline, jäädes küll ainult SQL andmebaasi ja App Service'it kasutades 5 dollari kuueelarvesse. Tabelis 4 on toodud sobivaimad Azure'i komponendid ja nende hinnad koos mahupiirangutega.

Tabel 4. Microsoft Azure'i arhitektuursete komponentide hind ja mahupiirangud.

Komponent	**Hind**	**Mahupiirang**
DocumentDB	~ $20 kuus	1GB salvestusruumi kuus
SQL DB	~$5 kuus	2 GB salvestusruumi
App Service	**Tasuta**	1GB salvestusruumi, 1GB RAM-i, jagatud tuumad, 1 instants
IoT Hub	**Tasuta**	8000 sõnumit päevas

3.3.5 RedHat OpenShift (Next Generation Developer Preview)

OpenShift'is saab väiksema mahukusega rakendust tasuta jooksutada ning autori katsetusel sai tõepoolest platvormile esimese instantsi (*Gear*'i) ilma rahata üles seada. Andmebaasimooduli lisamisel aga tekkis korduvalt süsteemitõrge, mistõttu polnud võimalik lõplikult veenduda, kas loodava rakenduse jaoks RedHat OpenShift'i kasutamine on võimalik. Süsteemitõrked võivad olla tingitud sellest, et tegemist on arendusjärgus teenusega. Kuna platvormi veebilehel polnud informatsiooni selle kohta, kas teenuse valmimisel see endiselt tasuta kasutada jääb ja milliste piirangutega, siis jäi

süsteemitõrgete ja lisainformatsiooni puudumise tõttu RedHat OpenShift edasisest võrdlusest välja.

3.3.6 Heroku

Heroku pakub sisuliselt virtuaalmasinaid (*Dyno*), milles erinevates keeltes loodud rakendusi väga lihtsalt jooksutada saab. Lisakomponendid nagu andmebaas tuleb Heroku lisade kataloogist juurde tellida. Need võivad, aga ei pruugi, olla Heroku enda loodud ja hallatud, mistõttu on hinnapoliitika iga komponendi puhul erinev. Antud projektis vajalik rakenduse konteiner on mahupiirangutega tasuta paketis olemas, andmebaasidest oli mõistlikuim kasutada mLab'i poolt pakutavat MongoDB'd. Tabelis 5 on toodud Heroku sobivaimad komponendid koos vastavate hindade ning mahupirangutega.

Tabel 5. Heroku arhitektuursete komponentide hind ja mahupiirangud.

Komponent	**Hind**	**Mahupiirang**
mLab MongoDB	**Tasuta**	0,5GB salvestusruumi
Dyno	**Tasuta**	1000h kuus, 512MB RAM-i, 1 worker

3.3.7 Nanobox

Nanobox'i puhul ei saa rääkida erinevate arhitektuursete komponentide omavahelisest sidumisest või kombineerimisest. Nanobox'i arenduskeskkond ja rakendus hoolitsevad kogu konfiguratsiooni eest, vajalik on vaid rakendus luua ja valida, milliseid komponente see rakendus tarbib. Kasutatavate komponentide nimekiri kattis enamlevinud SQL ja NoSQL andmebaasid ning kohaldatud baasnõuetes kirjeldatud programmeerimiskeeltes loodud rakenduste jooksutamise konteinereid. See platvorm võimaldab ka komponentide jaotamist eri konteineritesse, aga tasuta versioon sisaldab vaid ühe konteineri kasutust, mille täpsemaid parameetreid ega mahupiiranguid pole kuskil kajastatud.

3.3.8 Universaalsete platvormide vastavus baasnõuetele ja võrdluse kokkuvõte

Analüüsi tulemusena välistati mõned universaalsed platvormid, mille puhul oli esmase analüüsi põhjal järeldatav nende sobimatus antud projekti jaoks. Selle tulemusena jäi

lõplikku võrdlusesse viis platvormi, millest Microsoft Azure siiski mahupiirangu baasnõudele ei vastanud. Universaalsete pilveplatvormide seas jäi sõelale seega neli sobivat valikut: AWS, Google Cloud, Nanobox ja Heroku. Tabelis 6 on toodud universaalsete pilveplatvormide võrdlus ning vastavus kohaldatud baasnõuetele. Nõudeid 1...3 hinnati jah/ei (+/-) väärtustena, nõue 1 ehk hind oli rahuldatud, kui platvorm võimaldas tasuta paketti oodatud mahtudega. Lihtsuse ja õpitavuse nõuet hinnati 3-pallisüsteemis, milles 3 palli tähistab maksimaalset tulemust ehk lihtsat ja kergesti õpitavat toodet, 1 pall aga väga keerukat toodet, mille selgeks õppimine nõuab rohkem vaeva.

Tabel 6. Universaalsete pilveplatvormide võrdlus.

	AWS	**Google Cloud**	**Microsoft Azure**	**Heroku**	**Nanobox**
(1) Tasuta pakett oodatud mahtudega (28 800 päringut päevas)	+ (12 kuud)	+	-	+	+ (open-source projektile)
(2) SSL	+	+	+	+	+
(3) Keeleline tugi (PHP, Node, Java)	+	+	+	+	+
(4) Lihtsus ja õpitavus	*	**	**	***	***

AWS'i puhul pakutakse tasuta paketti vaid 12 kuuks ning platvormi haldamine, dokumentatsiooni loetavus ja üldine UI oli konkurentidest oluliselt keerulisem. Nanobox'i hinnainfo ja mõningad jõudlust määravad parameetrid (sh piirangud) ei olnud leitavad, mistõttu jäi ebakindlus, kas päris tasuta seda siiski kasutada saab. Seega toimus lõplik valik Google Cloud'i ja Heroku vahel. Google Cloud'i suurim nõrkus on platvormi suurus, mis aga väga hea kasutajaliidese ja dokumentatsiooniga suureks takistuseks ei kujunenud. Heroku nõrkuseks on ebakindlad lisad - antud projekti puhul andmebaasimoodul, mille pakkuja pole Heroku ise ning mille tasuta maht võiks kujuneda ebapiisavaks.

Eeltoodut arvesse võttes osutus parimaks valikuks pilveplatvormide seas Google Cloud oma täielikult tasuta paketiga, piisavate ressursipiirangutega ning väga laialdaste võimalustega.

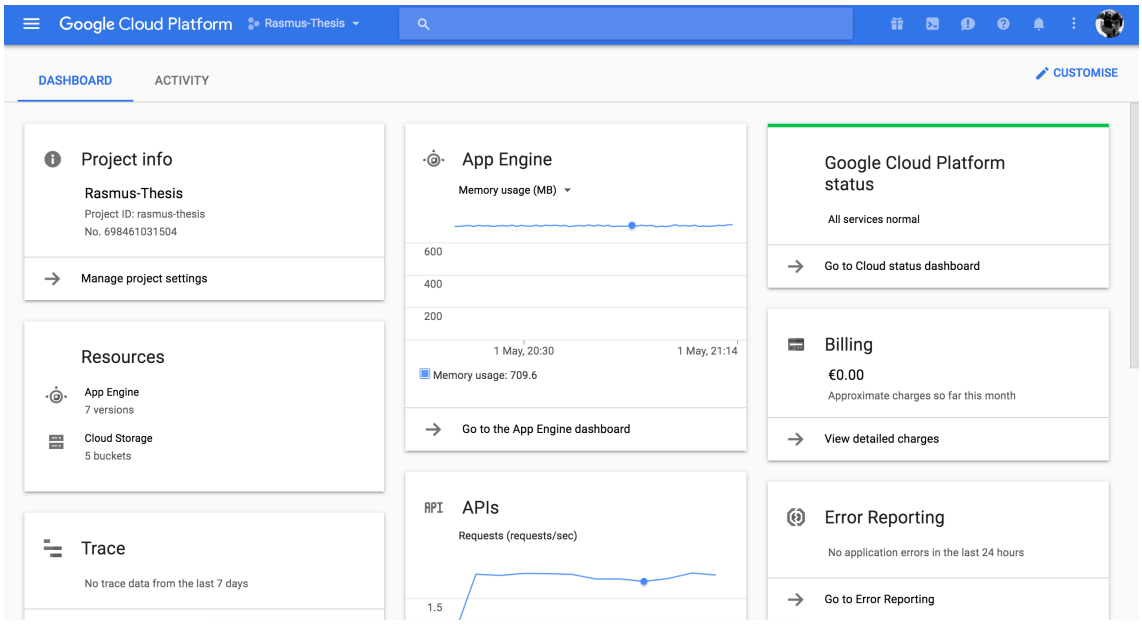
4 Rakenduse arhitektuur ja kasutatavad tehnoloogiad

Käesolevas peatükis antakse ülevaade kogu prototüübi arhitektuurist valitud pilveteenuses ning tuuakse välja kasutatud tehnoloogiad koos valikut toetava põhjendusega. Loodava lahenduse puhul saab eristada ka rakenduse enda sisemist arhitektuuri, mille struktuuri ning mustreid veidi lähemalt tutuvustatakse.

4.1 Prototüübi arhitektuur valitud pilveplatvormi põhjal

Prototüübi jaoks valitud pilveplatvormiks osutus Google Cloud, milles oli plaanitud kasutada andmebaasimoodulina Datastore komponenti ning ülejäänud rakenduse jooksutamiseks App Engine komponenti. Google Cloud ise on oma suuruse, keerukuse ja komponentide poolest IaaS teenus, ent vähemalt Google ise defineerib App Engine komponendi PaaS lahendusena, mille kasutaja infrastruktuuri peale liigselt vaeva ja energiat kulutama ei pea.

Joonis 5 on ekraanipilt Google Cloud platvormi konsooli avalehest, kus kuvatakse projekti üldinfot, kasutatavaid ressursse ning aktiivsete komponentide analüütikat.

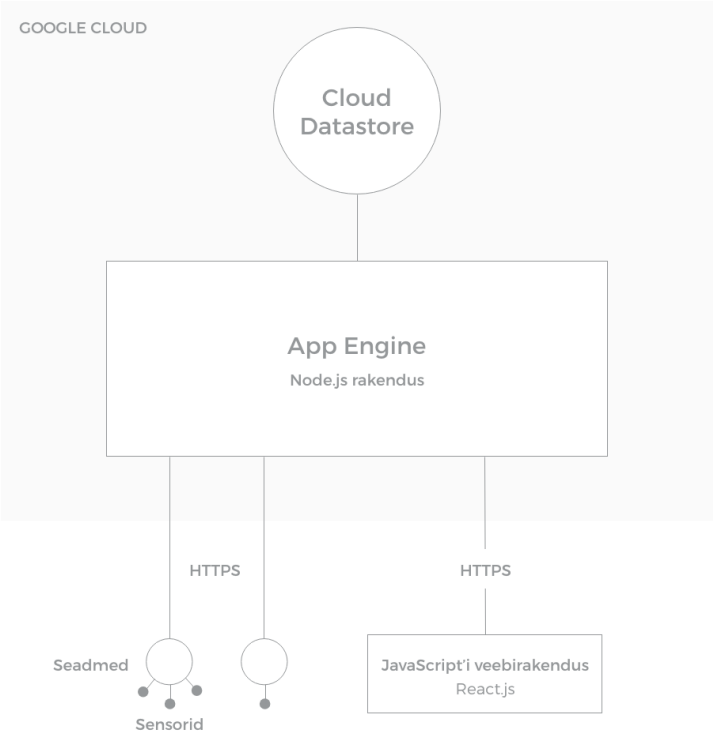


Joonis 5. Loodava projekti Google Cloud platvormi konsooli avaleht.

App Engine toetab rakenduste jooksutamist mitmes eri keeles - C#, Go, Java, Node.js, PHP, Python ja Ruby. Allalaetava käsurea tööriista abil on võimalik rakendusi luua, üles laadida ja hallata. App Engine toimib vaikimisi automaatselt skaleeruvana, mistõttu ei pea lihtsamate rakenduste puhul instantside haldamisega tegelema. Lisaks on App Engine'i moodulis juba sisse ehitatud primitiivsem versioonihaldus ja monitoorimine.

Cloud Datastore on NoSQL andmebaas, mis on (tüüpiliselt mitterelatsioonilistele NoSQL andmebaasidele) skaleeruvusele orienteeritud. Datastore'is loodud objekti kutsutakse Entity'ks ning nende haldamine on võimalik ka pilveteenuse graafilise kasutajaliidese abil. Lisaks on võimalik Datastore'is kasutada transaktsioone ning luua indekseid keerulisemate filtreerimiste kiirendamiseks.

Joonis 6 kujutab loodava prototüübi arhitektuurseid komponente. Pilves asuvad Google Cloud platvormi komponendid - andebaasina kasutatav Cloud Datastore ning rakenduse jooksutamiseks mõeldud App Engine. Seadmed ning JavaScript'i veebirakendus suhtlevad pilvega üle HTTPS protokolli. Ühel seadmel võib olla üks või mitu sensorit.



Joonis 6. Prototüübi arhitektuursed komponendid.

4.2 Programmeerimiskeele valik

Kõikidest App Engine'i poolt toetavatest programmeerimiskeeltest oli autori jaoks tuttavaim Node.js. Selle keele valikut toetasid nii mugav ja kiire rakenduse ülesseadmine ja paketihaldus kui ka vähene konfigureerimisvajadus. Lisaks oli plaanis kirjutada rakenduse brauseris jooksev osa JavaScripti "ühelehelise rakendusena" (ingl *single-page application*, SPA), kasutades selleks React.js raamistikku. Ühe programmeerimiskeele kasutamine rakenduse üleselt lihtsustab administreerimist, hilisemat täiendamist ning rakenduse mõistmist, mistõttu on samuti JavaScriptil põhinev Node.js eelistatum võrreldes PHP ja Javaga.

Node.js on asünkroonne sündmustepõhine JavaScripti versioon, mis on disainitud skaleeruvate veebirakenduste loomiseks ning ehitatud Chrome V8 JavaScript mootorile. Node'i asünkroonsus toimib veidi teisiti kui traditsiooniliste keelte (näiteks Java) operatsioonisüsteemi lõimede (ingl *Thread*) kasutamine. Lõim on paralleelne protsess, mis teeb koostööd või on muul moel seotud sama programmi teiste paralleelsete protsessidega. Node'i loojate väitel on operatsioonisüsteemi lõimedepõhine võrgupäringute haldamine võrdlemisi ebaefektiivne, mistõttu Node'is nende kasutamist välditakse. Node ei kasuta ka lukke, mistõttu rakenduse nö lukustunud paigalseisu (ingl *deadlock*) pole võimalik tekitada. Disaini poolest on Node sarnane Ruby Event Machine'i ning Python Twisted'i moodulitele, olles mõlemast suurel määral inspireeritud. (Node.js veebileht, 30.03.2017)

Kokkuvõttena on Node.js autori isikliku kogemuse ning Node'i loojate tutvustuse põhjal üks parimaid keeli, mida hetkel skaleeruva rakenduse loomiseks kasutada. Käesoleva projekti spetsiifikast lähtudes välistati seetõttu muud keeled ning kirjutati rakendus Node'is.

4.3 Rakenduse sisemine arhitektuur

Rakenduse sisemine arhitektuur jaguneb serveripoolseks API'ks ning kliendipoolseks veebirakenduseks. Poolte omavaheline kommunikatsioon toimib üle HTTP, kusjuures ressursside pärimiseks ja muutmiseks kasutatakse HTTP lauseid nagu GET, POST,

PUT ja DELETE. Andmete saatmiseks kasutatakse JSON'i (*JavaScript Object Notation*) formaati.

Node'i serveri loomiseks ning HTTP päringute haldamiseks kasutatakse Express raamistikku. Express'i kasutamist tingib peamiselt selle lihtsus, laialdane kasutus ja populaarsus. Aktiivse ja suure kasutajaskonna tõttu on raamistikus leitud vead kiiresti parandatud ning kasutajate kommuuni abil on arendusel tekkivatele küsimustele lihtne vastuseid leida. Express ise reklaamib end kiire ja minimalistliku veebirakenduse raamistikuna, mis antud prototüübi pigem lihtsa rakenduse jaoks tundus hästi sobivat. (Express.js veebileht, 18.04.2017)

Andmete haldamiseks on Google loonud Datastore'i mooduli Node'i jaoks, mis vahendab andmepäringuid ning lihtsustab Datastore'iga ühendumist. Kõikide rakenduste sõltuvuste pakettide (või moodulite) haldamiseks kasutatakse Yarn'i, mis on traditsioonilise Node Package Manager'i ees eelistatud tema kiiruse ja turvalisuse tõttu.

Rakenduse kliendipoolne osa ehk *front-end* lahendatakse JavaScript'i ühelehelise rakendusena (SPA), et hallata suuremat osa äriloogikat brauseris ning vähendada vajadust rakendusesiseselt kogu lehekülge serverist laadida. SPA on rakendus, mis ei lae kasutamise ajal kogu lehekülge uuesti. Sisuliselt on tegu nn "paksu" kliendirakendusega, mis laetakse ühekordselt serverist. (Mikowski jt 2013) See tähendab, et SPA on kiire ja efektiivne ning ei pidurda rakenduse kasutamist. JavaScripti raamistikke, mis toetavad SPA ideed, leidub väga erinevaid. Võrreldes tavalise HTML'i ja JavaScripti rakendusega üritavad enamik JavaScripti raamistikke lahendada andmete sidumise (ingl *data binding*) probleemi. HTML'is mõne DOM (*Document Object Model*, dokumendi objektimudel) elemendi siseste andmete muutmine tähendaks standardses JavaScript'is iga kord vastava elemendi tuvastamist, selle sisu analüüsimist ja modifitseerimist ning DOM'i uuendamist. JavaScript'i raamistikud üritavad andmete sidumist automatiseerida, nii et kui andmed muutuvad, siis DOM uuendatakse automaatselt vastavalt muutunud andmetele.

Üheks tuntumaks selliseks raamistikuks on Google'i poolt loodud Angular.js, mis jagab rakenduse loogika teenusteks, staatilised vaated mallideks ning korduvkasutatavad ning eristatavad komponendid direktiivideks. (Angular.js veebileht, 02.04.2017) React.js on teine populaarne alternatiiv, mis pole küll päris iseseisev raamistik. React kasutab

virtuaalset DOM'i ning lahendab samuti andmete sidumise probleemi, ent ülejäänud rakenduse loogika ja arhitektuur võib olla mistahes raamistiku põhjal loodud. Seetõttu loetakse React'i pigem kasutajaliidese paketiks või teegiks, mida üldjuhul kombineeritakse Flux'i arhitektuurimustriga. (React.js veebileht, 20.04.2017) Ka Flux pole mahukas raamistik, vaid pigem just muster, mille järgimine ei tähenda palju lisakoodi kirjutamist. (Flux veebileht, 03.04.2017) Küll aga leidub järjekordne moodul nimega Redux, mis Flux'i arhitektuurimustri põhjal React'ist välja jäävat andmete oleku haldamist lahendab. (Redux veebileht, 10.04.2017) Leidub ka komplektsemaid ning mahukamaid JavaScripti raamistikke nagu Ember.js või Meteor.js, mis ühel või teisel moel ka *back-end*'i mõjutavad. Et mahukamate raamistike järele antud prototüübis vajadus puudus, siis piisas SPA loomiseks just React'i lihtsast kasutajaliidese manipuleerimise teegist, millele rakenduse kasvades keerukamaid arhitektuurseid komponente lisada võiks. Seega sai JavaScript'i kliendirakenduse jaoks valituks React.js.

React'i rakendus lähtestatakse rakenduse konteineriks oleva HTML elemendile viitamise ning rakenduse marsruutide (ingl *routes*) seadistamise kaudu. Ruuter tagab kasutaja tegevuse delegeerimise õigele komponendile. React'i komponendid võimaldavad jagada kasutajaliidese taaskasutatavateks osadeks ning keskenduda igale osale isoleeritud keskkonnas. Komponente võib defineerida funktsiooni või klassina, millest viimase kasutamine eeldab küll ES6 ehk ECMAScript 6 (ECMA-262 programmeerimiskeele 6. väljalase) kasutust. Lisaks kasutatakse React'is JSX süntaksit (JavaScript'i laiendsüntaks), mis võimaldab kirjeldada JavaScript'i objekte kasutajaliidesele lähedasel moel, see tähendab, et süntaktiliselt on see midagi HTML'i ja JavaScript'i vahepealset. JSX tõlgendatakse JavaScript'i objektiks ning sisuliselt saab React'i kasutada ka ilma JSX'ita, kuigi React'i loojad seda ei soovita. (React.js veebileht, 20.04.2017)

React'i komponentidel on oma sisemine olek (ingl *state*) ning elutsükkel. Komponenti saab kontrollida ja seadistada elutsükli meetodite (ingl *lifecycle methods*) abil. Näiteks formuleerimise meetod *render* tagastab kasutajaliidese struktuuri JSX süntaksi või JavaScript'i objektina. Lisaks võib komponenti kontrollida algsel initsialiseerimisel konstruktori (ingl *constructor*) ning lähtestamise meetoditega (*componentWillMount* ning *componentDidMount*). Komponendi uuenemisega seotud meetodid on *componentWillReceiveProps*, *shouldComponentUpdate*, *componentWillUpdate* ning

componentDidUpdate. Lisaks kasutatakse ka lahtiühendumise meetodit nimega *componentWillUnmount*. Loetletud meetodid teevad komponendi haldamise arendajale ligipääsetavamaks ning hoiavad ära suure osa senisest manuaalsest DOM'i manipuleerimisest. (React.js veebileht, 20.04.2017)

Et kliendipoolne rakendus oleks üheks vähendatud serveri poolt tagastatavaks komplektiks kokku ehitatud, kasutatakse moodulite ja muude failide komponeerimiseks Webpack'i komplekteerijat. Lisaks on kasutusel Babel'i pistikprogramm Webpack'i jaoks, mis teisendab uuema JavaScript'i (näiteks ECMAScript 6) koodi ka vanematele brauseritele käitatavaks standardseks JavaScriptiks.

Kliendirakenduse funktsionaalsused nägid ette ka süsteemi autentimise võimekust - see tähendab, et süsteem peab võimaldama kasutajal sisse logida ning autentimata kasutajatele näidatakse vaid sisselogimise vormi. Kasutajate tegevusi on võimalik piirata React'i enda sisemise ruuteriga (*react-router*), mis aga nõuab turvalisuse tagamiseks ka serveripoolset sessioonihaldust. Selleks kasutab rakendus JSON Web Token'i (JWT) nimelist moodulit, milles loodud tähised (ingl *token*) aitavad turvaliselt päringuid veebirakendustes representeerida. (JWT veebileht, 12.04.2017)

4.4 Versioonihaldus

Loodava rakenduse paremaks haldamiseks ning tööprotsessi hõlbustamiseks kasutatakse versioonihaldusena Git'i. Git on tasuta versioonihaldussüsteem, mis on väga universaalne, sobides nii suuremate kui ka väiksemate projektide haldamiseks. Git'i loojate sõnul tasub seda versioonihaldust eelistada näiteks Subversion'ile, CVS'ile ja teistele sarnastele süsteemidele odava ja hõlpsa kohaliku versiooniharude haldamise, mugava võrdluskeskkonna ning mitmete paralleelsete töökeskkondade võimekuse tõttu. (Git veebileht, 15.04.2017) Koodi hoiustamiseks kasutatakse juhendajate soovitusel Arvutisüsteemide instituudi Git'i keskkonda, mis põhineb GitLab'i platvormil ning kus on võimalik privaatseid Git'i repositooriumeid luua. GitLab pakub ka graafilist kasutajaliidest, ent seda vaid erinevatele Linuxi versioonidele, mistõttu kasutatakse käesoleva projekti puhul Git'i käsurearakendust.

5 Disaini- ja arendusprotsess

Järgnevalt räägitakse projekti disaini- ja arendusprotsessist ning kirjeldatakse lahenduse erinevaid arenguetappe. Kirjeldatakse ka toimunud muudatusi skoobis ja kontseptsioonis ning valitud metoodikate ja tehnikate rakendamist. Eraldi peatükis on lühidalt välja toodud ka rakenduse funktsionaalsed nõuded, mis kujunesid kogu protsessi vältel, andmaks paremat ülevaadet loodava rakenduse lõplikust skoobist. Lisaks eelnevalt pilveplatvormide võrdlemiseks kirjeldatud süsteemsetele nõuetele kaardistavad funktsionaalsed nõuded kasutajate ootusi andmete visualiseerimise ning seadmete ja sensoite haldamise osas.

5.1 Disainiprotsessi ülevaade

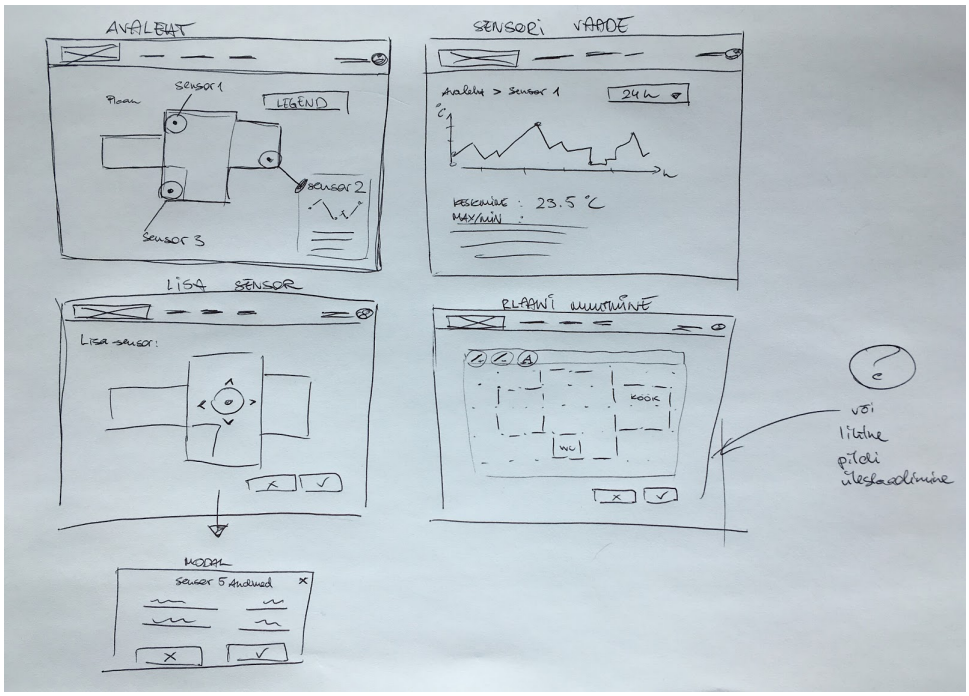
Käesoleva töö disainiprotsess järgib iteratiivse disainiprotsessi mudelit, tagamaks disaini kasutajakesksuse igas projekti etapis. See tähendab, et toimub tsükliline vajaduste ja nõuete identifitseerimine, disainimine, interaktiivse versiooni ehitamine ning lahenduse hindamine. Selline lähenemine tagab toode sujuva arengu, milles kasutaja on pidevalt protsessi kaasatud ning disainer saab oma tööle pidevat ja kiiret tagasisidet. Iteratiivse mudeli toetamiseks rakendati erinevaid disainiprotsessi tehnikaid, mille kombineerimisel on võimalik tsüklilist mudelit järgida. Järgnevalt on kirjeldatud vastavate tehnikate rakendamist disainiprotsessis.

5.1.1 Intervjuud

Esimese etapina identifitseeriti kasutajate vajadused ja kirjeldati vastavad nõuded. Selleks viidi kahe teaduriga läbi intervjuud, milles küsiti nii avatud kui ka suletud küsimusi. Avatud küsimustega sai minna sügavuti teemadesse, mis aitasid kasutajate tegelikke vajadusi mõista ning suletud küsimuste abil hoiti piisavalt fookust ning välditi liigset teemast kõrvalekaldumist. Intervjuud olid tõhusaks vahendiks esmaste vajaduste mõistmiseks ning see on disainiprotsessi puhul esmatähtis - kasutajate vajadusi ja tahet pole võimalik iseseisvalt arvata, samuti pole mõtet enda eelistustest juhindudes disainida ning loota, et tulemus kasutajatele sobib. (Carlson jt 2006)

Intervjuude tulemusena selgus disaini mõttes ühe olulisema komponendina kasutajate soov kujutada nutikodu sensoreid maja või korteri plaanil või muul skeemil. Senised valmislahendused pakuvad üldjuhul sensori vaadet graafikuna, tabelina või loeteluna, aga tegelikult on kasutajate peamiseks ootuseks andmete või sensorite hetkeolek. Näiteks toodi targa kasvuhoone projekt, milles võib olla mitu termoandurit ning nende eristamine nimedepõhiselt oleks võrdlemisi tüütu ja keeruline. Sensorite kuvamine skeemil või plaanil aitaks suurel määral andmeid atraktiivsemaks ning kasutajasõbralikumaks muuta. Edasine disain võttis kasutaja soove arvesse ning seadis kuvatava plaani või skeemi avalehele vaate keskmesse.

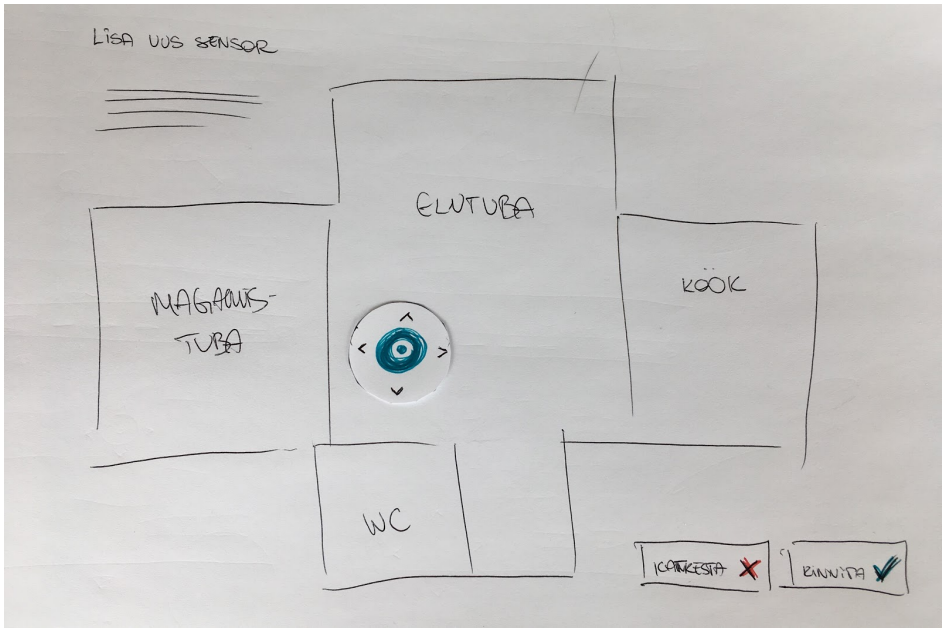
Lisaks üldisemale disainikontseptsioonile aitasid intervjuud paika panna ka projekti täpsemat skoopi. Kasutajate vajaduste analüüsimisel tulid välja mitmed nõuded, mis lõppskoopi ei mahtunud, aga intervjuude ajal oli võimalik kohe kasutajate ootused prioritiseerida ning seeläbi projekti fookust paremini suunata. Loodava süsteemi lõplikud funktsionaalsed nõuded on loetletud peatükis 5.3, milles mõned defineeriti juba intervjuude ajal. Joonis 7 kujutab esimesi kasutajaliidese visandeid, mida koostati intervjuude põhjal, illustreerimaks paremini kasutajate ootusi süsteemi osas.



Joonis 7. Esimesed kasutajaliidese visandid, mis koostati intervjuude põhjal.

5.1.2 Prototüüpimine ja kasutajatestimine

Prototüübi disain arenes kahes prototüüpimise iteratsioonis, millest esimene oli paberprototüüp ning teine digitaalne prototüüp. Digitaalse prototüübi loomiseks kasutati InVision veebirakendust, milles loodud disainivaateid on võimalik tüüpiliste sisendsündmuste (*click*, *hover* jt) abil omavahel siduda. (InVision veebileht, 02.04.2017) Prototüüpe testiti viie kasutaja peal, et näha, kas nad mõistavad lahenduse üldist ideed ning kas nad oskaksid seda kasutada. Kasutajate valim oli seekord juhuslikum, kaasates autori tutvusringkonnast potentsiaalseid nutikodu omanikke, kes ei oleks teadurid. See andis võimaluse valideerida lahendust seni protsessist väljaspool olnud inimeste peal ning nagu testimisel selgus, tõi protsessi uusi vaatenurki ja ideid. Kasutajatestimisel anti kasutajatele ette ülesanded (vaata Lisa 1), mis eeldasid loodavate funktsionaalsuste rakendamist ning kasutajaid jälgiti samal ajal, et tuvastada nende võimalikke arusaamatusi. Lisaks paluti kasutajatel valjult mõelda (*Think-out-loud* metoodika), (Rogers jt 2011) et koguda rohkem tagasisidet. Joonis 8 on pilt paberprototüübi sensori lisamise vaatest, kus kasutaja saab sensorit ruumiplaanil liigutada ning oma tegevuse katkestada või kinnitada.

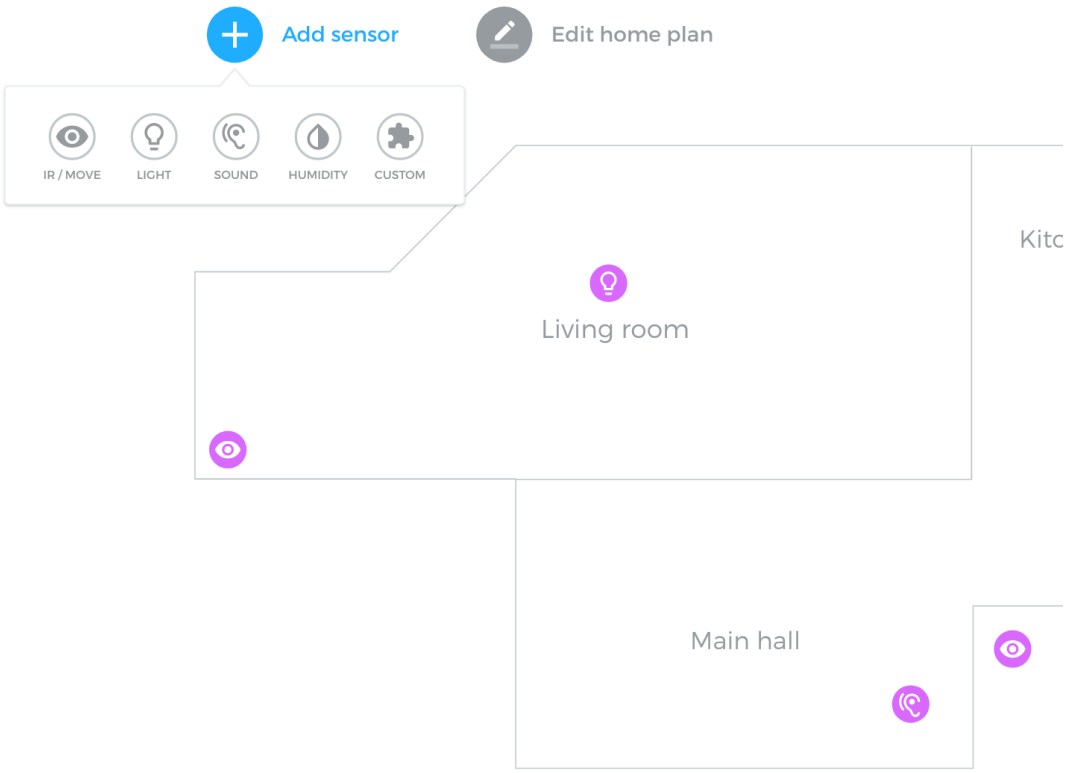


Joonis 8. Paberprototüübi vaade sensori positsioneerimiseks plaanil.

Kasutajatestimise mõlemad etapid aitasid kasutajate ootusi paremini mõista ning lisaks tulid välja mitmed vead, mis süsteemi üldisemat kontseptsiooni palju mõjutasid. Näiteks

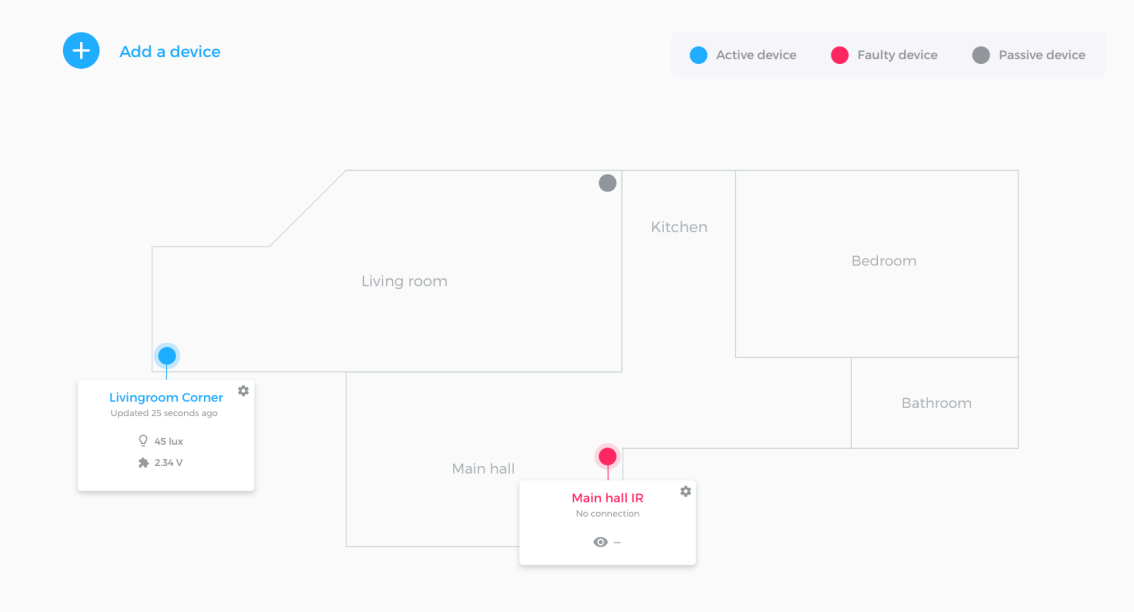
selgus digitaalse prototüübi testimisel ning hilisemal täpsemal arutelul teaduritega, et füüsilisel kujul oleks üheks positsioneeritavaks komponendiks seade, mitte sensor. Ühe seadme küljes võib seega olla mitu sensorit, mis samas toanurgas erinevaid andmeid mõõdavad ja serverile saadavad. Sensoril üksinda aga pole andmete edastamise võimekust ega ka unikaalset tunnusmärki, millega teda süsteemis eristada.

Joonis 9 kujutatakse esimese digitaalse prototüübi kasutajaliidese elemente. Tegemist on väljavõttega avalehest, millel on võimalik uut sensorit tüübipõhiselt lisada ning ruumiplaani muuta. Hiljem asendus sensori lisamine seadme lisamisega.



Joonis 9. Digitaalse disainiprototüübi esimese versiooni kasutajaliidese elemendid.

Joonis 10 kujutab digitaalse prototüübi uuendatud versiooni, millel peamiseks tegevuseks on seadmetelt saadetud andmete reaalajas jälgimine ning uue seadme lisamine. Lisandunud on ka seadme staatust indikeerivad värvid.



Joonis 10. Digitaalse disainiprototüübi uuendatud avalehe väljavõte.

5.1.3 Valideerimine disainikriitika abil

Kasutajatestimisele lisaks kasutati disaini valideerimiseks ning täiendavate ideede kogumiseks disainikriitika koosoleku metoodikat, mida kasutab aktiivselt AirBnB disainitiim. (Spec 73, 2015) Loodava lahenduse ideed presenteeriti väljaspool senist protsessi olevale huvigrupile, milles iga liige kirjutas oma tagasiside erinevate kategooriate paberitele. Vastavad märkmed liigitati nelja kategooria alla - ideed, küsimused, miinused ja plussid. Saadud märkmetabeli punkte arutati ning analüüsiti, mille tulemusena formuleeriti komplektne tagasiside loodud lahendusele.

5.2 Arendusprotsessi ülevaade

Käesolevas projektis ei rakendata ühe kindla arendusmetoodika reegleid, vaid keskendutakse agiilsete metoodikate ühistele väärtustele ja parimatele praktikatele. Näiteks kulgeb arendus iteratiivsete tsüklitena, mille pikkuseks on maksimaalselt 2 nädalat. Funktsionaalsused hoitakse võimalikult väikestena ning iga funktsionaalsuse valmimisel tehakse tootest uus väljalase (ingl *release*), mida saab demonstreerida projekti huvigruppidele. Arendusel ollakse valmis selleks, et nõuded võivad muutuda ning sellistele muutustele reageeritakse kiiresti. Kommunikatsiooni ja koostööd väärtustatakse kõrgelt, mistõttu hoitakse pidevat sidet lahenduse vastuvõtjate ning loojate vahel - antud juhul süsteemi kasutajateks olevate teadurite ning autori vahel.

Paljuski on ühe kindla metoodika vältimine tingitud projekti mittestandardsest arendusmeeskonnast, mille ainukeseks liikmeks on autor ise. Samuti lähtub autor enda senisest kogemusest, mille põhjal ühe kindla metoodika liiga range jälgimine pigem vähendab efektiivsust. Iga tiim peaks leidma endale sobivad metoodikad või protsessid ning neid oma vajadustele vastavalt kohaldama.

5.2.1 Pilveprojekti loomine

Esimese tehnilise tegevusena loodi uus projekt Google Cloud'i pilveplatvormile. Projekti hinnapaketiks valiti tasuta pakett, mis lubab teatud komponente piiratud andmemahuga kasutada ning ülejäänud platvormi peal 300-dollarilise krediidiga aasta jooksul teisigi komponente proovida. Google Cloud'il on väga hea dokumentatsioon, mille põhjal tutvustatakse kohe ka näidisrakenduse loomist ja üleslaadimist. Selleks on vaja lokaalselt installeerida Google Cloud'i arendustööriistade teek ning kloonida Git'i kaudu näidisprojekt. Näidisprojekti puhul on juba võimalik valida, millise keelega soovid jätkata, mis antud projekti puhul on Node.js. Seega sai esimese sammuna kloonitud Node'i testprojekt.

Google Cloud'i projekti määramiseks ning projekti lokaalse ligipääsu tagamiseks peaks lokaalselt sätestama ka projekti nime (GCLOUD_PROJECT) ning viite volitustele (GOOGLE_APPLICATION_CREDENTIALS) käsureaprogrammi konfiguratsioonis (näiteks bash'i puhul .bash_profile failis). Joonis 11 on kujutatud loodud rakenduse konfiguratsiooni näitel Google Cloud'i seadistamist käsureaprogrammile. Projekti nimeks on määratud "rasmus-thesis" ning volituste viiteks on aadress failile *rasmus-thesis.json*.

```
export GCLOUD_PROJECT="rasmus-thesis"
export GOOGLE_APPLICATION_CREDENTIALS="/Users/rasmus/Thesis/GCP_key/rasmus-
thesis.json"
```

Joonis 11. Käsureaprogrammi konfiguratsioon Google Cloud pilveteenuse kasutamiseks.

Node'i projekti moodulite installeerimiseks tuleb paketihaldustarkvara abil käivitada vastav skript. Käesolevas projektis on tegemist Yarn'i paketihaldusega, mistõttu tuleb pakettide allalaadimiseks ning rakenduse lokaalseks käivitamiseks jooksutada Joonis 12 kujutatud käsud projekti juurkaustas. Käsk *yarn install* käivitab pakettide allalaadimise ning *yarn start* käivitab rakenduse.

```
yarn install
yarn start
```

Joonis 12. Programmikood: paketihalduse abil moodulite installimine ning rakenduse käivitamine.

Rakenduse Google Cloud'i pilveplatvormile laadimiseks tuleb esmalt luua vajalikus pilvekomponendis ehk App Engine'is uus rakendus. App Engine rakenduse konfiguratsioonis on ainuke antud kontekstis oluline parameeter regioonivalik, mis määrab, millises serveripargis rakendust hoitakse. Kui rakendus on pilves loodud, piisab näidisprojektis, kus on Google Cloud konfiguratsioon juba seadistatud, vaid Joonis 13 näidatud üleslaadimise käsu jooksutamisest.

```
gcloud app deploy
```

Joonis 13. Programmikood: rakenduse pilve laadimine Google Cloud käsureatööriista abil.

5.2.2 Rakenduse arendamine ja jooksutamine

Esimeseks oluliseks osaks Node'i rakenduste projektide puhul on paketihalduse konfiguratsioonifail nimega *package.json*. Selles failis defineeritakse rakenduse nimi, Node'i versioon, rakenduse käivitamise konfiguratsioon ning vajalike teekide viited.

Pakettide installeerimiseks võib küll *package.json* faili ise muuta, aga üldjuhul on soovituslik kasutada selleks paketihalduse käske. Kuna valitud paketihalduseks on Yarn, siis saab Joonis 14 kujutatud käskudega vastavalt paketihaldus initsialiseerida (*yarn init*), teeke lisada (*yarn add*) ning teeke eemaldada (*yarn remove*).

```
yarn init
yarn add <teegi_nimi>
yarn remove <teegi_nimi>
```

Joonis 14. Programmikood: Yarn paketihalduse käsud.

Konfiguratsioonifailis määratakse ka rakenduse jooksutamise skriptid, mis lihtsustavad ja automatiseerivad rakenduse ehitamise ning üleslaadimise tegevusi. Rakenduse startimise skriptina piisab rakenduse serveripoolse juurfaili määramisest, milleks on käesolevas projektis *server.js*. Kliendirakenduse testimiseks on kasutusel Webpack'i arendusserver, mis lubab kliendirakendust kiirelt ja mugavalt arendada või testida ilma serveripoolse osa käivitamiseta. Joonis 15 kujutatakse Webpack'i arendusserveri käivitamiseks mõeldud skripti nimega "start-dev". Lisaks on koonisel näha ka teised

rakenduse jooksutamise skriptid. Skript "start" jooksutab rakendust, skript "build" ehitab rakenduse koodi Webpack'i abil kokku ning skript "deploy" kutsub eelnevalt defineeritud "build" skripti ning seejärel laeb rakenduse Google Cloud käsureatööriista abil serverisse.

```
"scripts": {
    "start": "node server.js",
    "build": "NODE_ENV=build webpack",
    "deploy": "yarn build && gcloud app deploy app.yaml",
    "start-dev": "NODE_ENV=development webpack-dev-server"
}
```

Joonis 15. Programmikood: rakenduse jooksutamise ja ehitamise konfiguratsiooniskriptid package.json failis.

Rakenduse üleslaadimine nõuab samuti konfiguratsiooni, mis peaks olema Google'i dokumentatsiooni põhjal failis nimega *app.yaml*. Vastava faili sisus on võimalik defineerida palju erinevaid rakenduse jooksutamise ning üleslaadimise seadeid, nendest olulisimad on rakenduse jooksutamise keel ning keskkonna skaleerimine. Joonis 16 näidatakse loodud rakenduse pilvekonfiguratsiooni, milles defineeritakse *runtime* ning *env* parameetrid. *Runtime* parameeter määrab rakenduse keskkonnaks Node.js ning *env* parameeter defineerib paindliku skaleeruvuse sätte pilveserveritele.

```
runtime: nodejs
env: flex
```

Joonis 16. Programmikood: rakenduse pilvekonfiguratsioon app.yaml failis.

Loodava rakenduse võib jagada kaheks: serveri pool ehk kogu pilves jooksev rakenduse osa ning kliendi pool ehk React.js põhjal loodud veebirakendus, mis jookseb kasutaja veebibrauseris. Serveripoolne rakenduse osa algab juba konfiguratsioonis mainitud *server.js* failist, mis defineerib rakenduse raamistiku ning käivitab serveri. Antud projektis osutus valituks raamistikuks Express.js, mis võimaldab kiirelt ja vähese koodiga luua minimalistlikke veebirakendusi. Primitiivse *hello-world* serveri saab Express'i abil luua vaid 10 koodireaga nagu näidatud ka Joonis 17. Esmalt lähtestatakse rakendus *express* teegi abil, seejärel defineeritakse juuraadressile vastav funktsioon, mis tagastab lihtsa "Hello World!" sõne ning lõpuks sätestatakse rakendus kuulama pordile 3000.

```
var express = require('express')
var app = express()

app.get('/', function (req, res) {
  res.send('Hello World!')
})

app.listen(3000, function () {
  console.log('Example app listening on port 3000!')
})
```

Joonis 17. Programmikood: primitiivse veebiserveri näide Express.js põhjal (Express.js veebileht, 18.04.2017).

Käesolevas projektis laetakse serveri juurfailis ka vajalik vahevara HTTP päringute parsimiseks ning logimiseks. Lisaks defineeritakse päringute aadressid ning nende haldamiseks vastavad funktsioonid. Kindlate aadresside puhul vastab server vastavalt aadressile määratud funktsiooniga, ülejäänud päringute puhul tagastatakse React'i rakendus, mis sisemiselt ülejäänud aadressidega tegeleb. Vajalikud teegid laetakse Node'i rakenduses *require* funktsiooni abil. Joonis 18 on näidatud loodud rakenduse põhjal funktsiooni, mis tagastab *index.html* faili igale päringule, mis eelnevate funktsioonide aadressidega ei sobitunud.

```
app.route("*").get((req, res) => {
    res.sendFile('client/dist/index.html', { root: __dirname });
});
```

Joonis 18. Programmikood: serveri juurfaili väljavõte, mis tagastab kliendirakenduse.

Ülejäänud serveripoolne koodibaas on jagatud vastavalt moodulite eesmärgile eraldi kataloogidesse. *Routes* kataloogi jäävad päringutega tegelevad funktsionaalsused ning *models* kataloogis asuvad andmebaasiga tegelevad süsteemi osad, sealjuures on need moodulid vastavalt mõjutatud ressurssidele (näiteks seade, seadmegrupp või kasutaja) eraldi failideks jaotatud.

Node.js toetab asünkroonsust ning tegelikult on Node'i rakendustes asünkroonsuse kasutamine rangelt soovituslik. (Node.js veebileht, 30.03.2017) Sünkroonsete funktsioonide kasutamine on võimalik, ent selline lähenemine ei kasutaks kogu Node'i potentsiaali. Käesolevas projektis on asünkroonsust maksimaalselt ära kasutatud, tehes näiteks seadmelt tulnud sensori andmete salvestamiste päringud paralleelselt. Üldjuhul toetavad sellist lähenemist ka kõik Node'is kasutatavad teegid, mille funktsioonid

ootavad tavaliselt ühe parameetrina viidet funktsioonile, mida käivitada asünkroonse tegevuse õnnestumisel (ingl *callback function*) või tagastavad nö lubaduse midagi teha ehk *Promise*'i. Promise on oma kirjapildi poolest oluliselt loetavam ning ei tekita palju üksteise sisse pesastunud (ingl *nested*) funktsioone. Lubadusi on võimalik pikemateks "kettideks" (ingl *promise chain*) kokku ehitada, võimaldades keerukaid asünkroonsete päringute kogumeid seeläbi kontrollida ja järjestada.

Joonis 19 on näidatud, kuidas loodava rakenduse seadmetelt tulevate andmete salvestamise loogika kasutab lubadusi selleks, et seadme erinevate sensorite andmeid paralleelselt hallata. Selleks defineeritakse iga sensori andmetele vastav päring ning käivitatakse päringud paralleelselt *Promise.all* funktsiooniga. Joonisel kuvatud näites tehakse iga päringu puhul ka seadme andmete konverteerimised, sest andmeid edastatakse ilma komata väärtustega. See nõuab eksponendi lisaparameetri kasutamist, millega saadetud väärtus serveris kümne astmena läbi korrutada ning seejärel ümardada.

```
sensorModel.getSensorsByDeviceId(deviceId).then(sensors => {
    let sensorDataRequests = _.map(sensors, sensor => {
        const sensorId = sensor[datastore.KEY].id;
        const dataKey = dataPrefix + sensor.number;
        const exponentKey = dataPrefix + sensor.number + exponentSuffix;

        const intValue = parseInt(data[dataKey]);
        const exponent = parseInt(data[exponentKey]);
        const value = (intValue * Math.pow(10, exponent)).toFixed(5);

        return sensorId && value ?
            sensorDataModel.createSensorData(sensorId, value) :
            Promise.reject('One or more sensor data incomplete');
        });

        Promise.all(sensorDataRequests)
            .then(() => { res.send('Data saved successfully') })
            .catch(err => { res.status(400).send(err) });

}).catch(err => {
    res.status(500).send(err);
});
```

Joonis 19. Programmikood: seadmelt vastuvõetud andmete salvestamine lubaduste ehk *promise*'ite abil.

Kasutajate päringute autentimiseks kasutatakse süsteemis *JSON Web Token*'eid ehk JWT märgendeid. Vastavad märgendid genereeritakse serveris õnnestunud kasutaja tuvastamise ehk süsteemi sisselogimise korral. JWT eeliseks on tema kompaktsus, mis

lubab märgendeid saata näiteks HTTP päises (ingl *header*). Lisaks sisaldab märgend ise juba kogu vajaliku informatsiooni kasutaja kohta, mis vähendab lisapäringute tegemise vajadust. (JWT veebileht, 12.04.2017) Kasutaja ressurssidega (näiteks seadmegrupi või seadmete info küsimine) nõuab kliendirakenduselt kasutaja märgendi saatmist, mille põhjal tuvastatakse kasutaja ning kontrollitakse, kas vastavale ressursile on kasutajal õigus ligi pääseda. JWT'ga seotud funktsioonid on kasutatavad läbi *jsonwebtoken* teegi, mida Yarn'i kaudu hõlpsalt projekti kaasata saab. Joonis 20 on kujutatud märgendi genereerimist, milleks on vaja täpsustada kasutajat eristav subjekt ehk *sub* parameeter ning aegumise ajatempel ehk *exp* parameeter. Käesolevas näites on subjektiks unikaalne kasutaja e-mail. Aegumise ajatempel genereeritakse *moment* teegi abil, lisades käesolevale ajahetkele ühe päeva, tulemuseks on järgneval päeval samal kellaajal aeguv märgend.

```
const createToken = email => {
    var payload = {
        sub: email,
        exp: moment().add(1, 'day').unix()
    };

    return jwt.sign(payload, config.TOKEN_SECRET);
}
```

Joonis 20. Programmikood: JSON Web Token märgendi genereerimise koodinäide.

Rakenduse kliendipoolne osa on React'i üheleheline veebirakendus (ingl *single-page application*), mis on kokku ehitatud üheks minimeeritud JavaScript'i failiks. See laetakse serveri poolt tagastatud *index.html* failiga kasutaja brauserisse ning edasi saab rakendust kasutada ilma lehte värskendamata. React'i rakenduse käivitamiseks piisab rakenduse konteineriks oleva HTML elemendile viitamisest ning rakenduse marsruutide (ingl *routes*) seadistamisest. Joonis 21 illustreerib rakenduse formuleerimist, milles imporditakse vajalikud teegid ning seatakse React'i konteineriks HTML element, mille identifikaator on *content*. Lisaks viidatakse käivitamisel defineeritud marsruutide konfiguratsioonile (*Routes* muutuja).

```
import '../dist/css/style.css';
import React from 'react';
import ReactDOM from 'react-dom';
import Routes from './routes';

ReactDOM.render(
  Routes,
  document.getElementById('content')
);
```

Joonis 21. Programmikood: React'i rakenduse formuleerimine.

Igale rakenduse marsruudile vastab kindel React'i komponent, lisaks võib React'i sisemise ruuteri seadistada aadressimuudatuse peale kindlaid tegevusi tegema. Näiteks võib mõned komponendid ja marsruudid kaitsta funktsiooniga, mis kontrollib autentimist. Käesolevas töös tähendab see kehtiva JWT märgendi olemasolu kontrollimist. Joonis 22 on näidatud, kuidas loodud rakenduse näitel saab React'i ruuterit konfigureerida ning teatud komponente kaitsta sellega, et kontrollitakse enne marsruudile minekut *onEnter* parameetriga kehtiva JWT märgendi olemasolu, viidates parameetri abil *requireAuth* funktsioonile, mis omakorda rakendab *Auth* moodulit.

```
import React from 'react';
import { Router, Route, browserHistory, IndexRoute } from 'react-router';
import { App, Dashboard, Devices, Login } from './components';
import Auth from './modules/auth';

const requireAuth = (nextState, replace) => {
    if (!Auth.isUserAuthenticated()) {
        replace({
            pathname: '/login'
        })
    }
};

const routes = (
    <Router history={browserHistory}>
        <Route path="/" component={App} onEnter={requireAuth}>
            <IndexRoute component={Dashboard} />
            <Route path="/devices" component={Devices} />
        </Route>
        <Route path="/login" component={Login} />
    </Router>
);

export default routes;
```

Joonis 22. Programmikood: React'i ruuteri seadistamine.

React'i tõeline olemus paljastub tema komponentide koostamisel, mida elutsükli meetoditega kontrollida saab. Antud töös on komponendid defineeritud ECMAScript 6 põhiste klassidena, mis vastavaid elutsükli meetodeid üle kirjutavad (ingl *override*) ning seeläbi rakenduse loojale komponendi kontrollimist võimaldavad. Näiteks on konstruktoris võimalik seadistada komponendi algne olek ning konfigureerida komponendi funktsioone. Kui komponent vajab oma töös väliseid andmeid, siis võib neid sisestada ülemkomponendi (ingl *parent component*) kaudu või teha vastav asünkroonne päring serverile komponendi sees. Sellised päringud tuleks teha pärast komponendi lähtestamist (ingl *mount*) ja samas enne komponendi väljanäitamist *render* meetodis. Heaks tavaks peetakse *componentDidMount* meetodis väliste päringute tegemist, mille tulemusena komponendi sisendandmed, olek või muud tegurid muutuvad, sest siis ei toimu mitmekordset taasesitamist (ingl *rerendering*) ehk *render* meetodi väljakutsumist. Joonis 23 on toodud rakenduse avalehe (*Dashboard*) komponendi *componentDidMount* koodinäidis, mis esmalt seab komponendi oleku *isFetching* tõeväärtuse tõeseks (et kasutajaliideses laadimise indikaatorit kuvada), seejärel kutsub seadmegrupi andmete saamiseks *getUserDeviceGroup* funktsiooni ning õnnestumise korral seadmegrupile vastava avalehe andmete pärimise funktsiooni (*getDashboardData*). Vea tekkimisel seatakse laadimise tõeväärtus valeks ning logitakse veasõnum konsooli.

```
componentDidMount () {
    this.setState(assign({}, this.state, { isFetching: true }));
    this.getUserDeviceGroup().then(deviceGroup => {
        this.getDashboardData(deviceGroup);
    }).catch(err => {
        this.setState(assign({}, this.state, { isFetching: false }));
        console.error(err);
    });
}
```

Joonis 23. Programmikood: React'i komponendi lähtestamise meetod.

Serveriga suhtlevaid komponente on käesolevas rakenduses mitmeid. Avalehe komponent pärib ühekordselt kasutajale vastava seadmegrupi andmeid ning perioodiliselt seadmete hetkeolekut. Lisaks suhtlevad serveriga sisselogimise ja väljalogimise komponent ning seadmete seadistusega tegelevad rakenduse osad. Antud rakenduses haldas iga komponent enda olekut eraldi, ent süsteemi kasvades näeb autor

mõistliku lahendusena siiski mõne lisaraamistiku kasutamist (näiteks Redux), mis rakenduse olekut keskselt haldaks ning sedasi komponentidevahelist suhtlust ja oleku keerukust vähendaks.

Komponendi visuaalne külg defineeritakse *render* meetodis, mis kasutab JSX süntaksit HTML'i ja JavaScript'i vahepealse objekti loomiseks. Komponentidepõhine arhitektuur võimaldab defineerida rakenduse osad loogilisteks ja taaskasutatavateks üksusteks ning JSX võimaldab rakenduse komponente mugavalt märgendada. Kirjeldatud elutsükli meetodeid ja formuleerimise loogikat kasutades on kirjutatud kõik rakenduse komponendid. Joonis 24 on näidatud rakenduse avalehe komponendi render meetodit, milles kasutajaliidese elemendid on kirjeldatud JSX süntaksi abil.

```
render () {
    const { deviceGroup, isFetching } = this.state;

    if (isFetching) {
        return <p>Loading data...</p>;
    }

    if (!deviceGroup || isEmpty(deviceGroup)) {
        return <p>No deviceGroup found for this user!</p>;
    }

    const devices = deviceGroup.devices;
    return (
        <div className="page">
            <h1 className="title">Test apartment</h1>
            <div className="flex">
                <div className="flex__item flex__item_3">
                    <DeviceList devices={devices} />
                </div>
                <div className="flex__item flex__item_7">
                    <div className="devicegroup-image-wrap">
                        <div className="devicegroup-image">
                            <DeviceMap devices={devices} />
                            <img src={floorplan} />
                        </div>
                    </div>
                </div>
            </div>
        </div>
    );
}
```

Joonis 24. Programmikood: React'i komponendi render meetod ja JSX süntaksi näide.

Selleks, et kliendipoolse rakenduse kõik osad oleksid üheks failiks kokku ehitatud, kasutatakse Webpack'i nimelist komplekteerimise tööriista. Webpack'i konfiguratsioon defineeritakse projekti juurkaustas *webpack.config.js* failis ning see sisaldab kogu ehituslikku loogikat - näiteks seda, millistes kaustades on sisendfailid ning kuhu ja kuidas salvestada väljundiks kompileeritud fail. Lisaks saab Webpack'i kasutada arendusversiooni jooksutamiseks, mis automaatselt muudetud failid (sealhulgas CSS'i ja JavaScript'i koodifailid või pildid ja muud lisad) rakendusse laeb ning seeläbi arendamist oluliselt mugavamaks ja kiiremaks muudab. Joonis 25 on koodinäidis Webpack'i konfiguratsioonist, mis on kirjutatud loodud rakenduse jaoks. Konfiguratsiooni *entry* parameeter määrab rakenduse juurfaili ning käesolevas näites ka Babel'i pistikprogrammi viite, mis garanteerib parema brauseritega ühilduvuse. *Output* parameeter defineerib genereeritud koodifaili nime ning väljundkausta. *Module* parameeter määrab reeglid erinevate failitüüpide laadimiseks ning *resolve* parameeter defineerib töödeldavate koodifailide laiendid.

```
const webpackConfig = {
    entry: {
      app: ['babel-polyfill', PATHS.src]
    },
    output: {
      path: PATHS.dist,
      filename: 'bundle.js'
    },
    module: {
        rules: [
            loaders.babel,
            loaders.css,
            loaders.font,
            loaders.image
        ]
    },
    resolve: {
        extensions: ['.js', '.jsx']
    }
};
```

Joonis 25. Programmikood: Webpack'i konfiguratsioon loodud rakenduses.

Kuna React'i rakenduse loomisel kasutati töös palju ECMAScript 6 süntaksit, siis tuli brauserite ühilduvuse saavutamiseks kasutada Babel'i pistikprogrammi. Babel'i abil saab React'i JSX ning JavaScript'i ES6 vormingus failid konverteerida nö harilikuks JavaScript'iks, mis on loetav kõikidele JavaScript'i toetavatele brauseritele. See tagab

rakenduse toimimise ka vanemates veebibrauserites, võimaldades kasutada arendamiseks uusimaid tehnoloogiaid ning keelevorminguid.

5.2.3 Arendusprotsessi kokkuvõte

Rakenduse loomisel kasutatavad tehnoloogiad on populaarsed valikud tänaste arendajate seas ning üldjuhul sisaldasid kõikide raamistike ja teiste kasutatud tehnoloogiate kodulehed väga esinduslikku ja põhjalikku dokumentatsiooni, mille põhjal polnud keeruline väiksemat rakendust luua. Võib öelda, et tehtud valikud olid mõistlikud ning ei põhjustanud liigset tüli koodi kirjutamisel. Webpack on oma võimalustelt küll väga võimekas ja see teeb tema konfigureerimise esmakordsel kasutamisel pigem valulikuks ja vearohkeks, aga dokumentatsiooni täpsel järgimisel on võimalik kõik probleemid lahendada. Valikutest raskeim oli ühelehelise rakenduse raamistiku valimine, mis võinuks kulmineeruda ka Angular'i kasutamisega, ent käesoleva töö jaoks toimis React.js ideaalselt. React on väga "kerge" tehnoloogia, mis pigem väikese rakendusega ka ilma lisanduva arhitektuurse teegita toimib. Rakenduse kasvades oleks mõistlik oleku haldamiseks kasutada näiteks Redux tehnoloogiat, mis Flux arhitektuurimustrist inspireerituna oleku haldamist ning serveripäringute tegemist koordineerib.

5.3 Süsteemi funktsionaalsused

Iteratiivsele arendus- ja disainiprotsessile tuginedes loodi süsteemile kasutuslood, et kaardistada loodava rakenduse jaoks vajalikke funktsionaalsusi. Agiilsele arendusprotsessile kohaselt kirjeldati kasutuslugusid järk-järgult vastavalt süsteemi valmimisele. Käesolevas peatükis on toodud projekti skoopi mahtunud funktsionaalsuste kokkuvõte läbi kasutuslugude jutustustena (ingl *use case narrative*).

5.3.1 Aktorid

Kuna tegemist on võrdlemisi lihtsa prototüüplahendusega, siis on aktoreid antud süsteemis vaid üks. Lihtsuse mõttes võib seda aktorit (kes tõenäoliselt on teadur, aga mitte ilmtingimata) kutsuda lihtsalt kasutajaks. Esimeses prototüübi versioonis rohkem rolle ei defineerita.

5.3.2 Kasutuslood

1. **Seadmete hetkeseisu vaatamine** - Kasutaja soovib süsteemi avalehel näha reaalajas saabunud seadmete hetkeandmeid. Andmed peavad sisaldama seadme nime, viimati saadetud paketi aega ning kõikide seadme sensorite mõõteväärtusi. Lisaks peab süsteem visualiseerima seadme positsiooni seadmegrupi plaanil.

2. **Sisse logimine** - Kasutaja soovib end süsteemi sisse logida e-maili ning parooli abil. Sisselogimata kasutaja ei tohiks näha ühtegi süsteemi alamlehte peale sisselogimisvormi. Sisseloginud kasutajale kuvatakse süsteemi avalehte.

3. **Välja logimine** - Kasutaja soovib oma sessiooni lõpetada, vajutades selleks väljalogimise lingile. Väljaloginud kasutaja saadetakse sisselogimisvormile.

4. **Seadme lisamine** - Kasutaja soovib süsteemi lisada uut seadet. Seadme lisamist peab olema võimalik igal hetkel katkestada.

    a. Kasutaja vajutab avalehel seadme lisamise nuppu

    b. Kasutajale kuvatakse seadme lisamise vorm

    c. Kasutaja täidab vormis seadme andmed (seadme nimi, identifikaator, sensorite arv ning sensorite tüübid)

    d. Kasutaja kinnitab seadme andmed

5. **Seadmete nimekirja kuvamine** - Kasutaja soovib näha kõiki oma seadmeid seadmete halduseks mõeldud alamlehel. Nimekirjas kuvatakse seadme nimi, sensorite arv ning seadmegrupi nimi (kui eksisteerib). Lisaks näidatakse nimekirjas nuppe seadme muutmiseks ja kustutamiseks ning seadmegrupi plaanile lisamiseks.

6. **Seadme muutmine** - Kasutaja soovib seadme andmeid muuta. Seadme muutmist peab olema võimalik igal hetkel katkestada.

    a. Kasutaja vajutab seadme sätestamise/muutmise nupule kas avalehel või seadmete nimekirjas

    b. Kasutajale kuvatakse seadme muutmise vorm

c. Kasutaja muudab vormis seadme andmed (seadme nimi, identifikaator, sensorite arv ning sensorite tüübid)

d. Kasutaja kinnitab seadme andmed

7. **Seadme eemaldamine** - Kasutaja soovib seadet eemaldada.

    a. Kasutaja vajutab seadme eemaldamise nuppu seadmete nimekirjas

    b. Kasutajale kuvatakse tegevuse kinnitamise aken

    c. Kasutaja kinnitab või katkestab seadme kustutamise avanenud aknas

8. **Seadmegrupi plaani lisamine** - Kasutaja soovib lisada enda seadmegrupile vastava plaani, millel seadmeid positsioneerida. Plaan on pildifail, mida kuvatakse avalehel seadmete taustana, illustreerimaks seadmete asukohti skeemil või ruumiplaanil.

    a. Kasutaja vajutab seadmegrupi plaani lisamise nupule

    b. Avanenud vormil kuvatakse plaani üleslaadimise nuppu

    c. Kasutaja vajutab üleslaadimise nupule, mis avab süsteemiakna faili valimiseks

    d. Kasutaja kinnitab faili valiku süsteemiaknas

    e. Kasutajale kuvatakse vormil valitud faili eelvaadet

    f. Kasutaja kinnitab või katkestab plaani lisamise vormiaknas

9. **Seadmegrupi plaani eemaldamine** - Kasutaja soovib eemaldada enda seadmegrupile vastava plaani.

    a. Kasutaja vajutab seadmegrupi plaani kustutamise nupule

    b. Kasutajale kuvatakse tegevuse kinnitamise aken

    c. Kasutaja kinnitab või katkestab plaani kustutamise avanenud aknas

10. **Seadme lisamine seadmegrupi plaanile** - Kasutaja soovib määrata seadme asukoha seadmegrupi plaanil ning sellega muuta seadme andmete kuvamise avalehel aktiivseks.

    a. Kasutaja vajutab seadme plaanile lisamise nupule seadmete nimekirjas

    b. Kasutajale kuvatakse aken, milles saab seadet seadmegrupi plaani eelvaatel liigutada

    c. Kasutaja lohistab seadme sobivale asukohale plaanil

    d. Kasutaja kinnitab või katkestab seadme positsioneerimise aknas

11. **Seadme eemaldamine seadmegrupi plaanilt** - Kasutaja soovib eemaldada seadme seadmegrupi plaanilt ning sellega muuta seadme andmete kuvamise avalehel mitteaktiivseks.

    a. Kasutaja vajutab seadme plaanilt eemaldamise nupule seadmete nimekirjas

    b. Kasutajale kuvatakse tegevuse kinnitamise aken

    c. Kasutaja kinnitab või katkestab seadme plaanilt kustutamise avanenud aknas

6 Võimalikud edasiarendused

Võimalike töö edasiarendustena võib tuua järgnevaid funktsionaalsusi, mis jäid esimesest lahenduse versioonist välja. Esiteks võiks süsteem pakkuda võimalust seadmete sisemise konfiguratsiooni muutmiseks ning tarkvara uuendamiseks läbi pilveplatvormi. Lisaks võiks süsteem võimaldada sarnaselt eksisteerivatele IoT andmete kogumisele keskenduvatele platvormidele seadmetelt tulnud andmetele reageerimist. Näiteks infrapunasensorilt saadetud andmete põhjal saaks luua koduse valvesüsteemi, mis sõnumi teel kasutajat liikumisest informeerib. Selles osas on loodud süsteem väga laiaks platvormiks ning võimalusi täiendamiseks on väga palju. Tehnilises mõttes on prototüübi esimeses versioonis suurimaks puudujäägiks seadmete ning pilveplatvormi andmevahetuse turvalisus. Suhtlus toimub küll üle HTTPS'i, ent seadme unikaalse identifikaatori äraarvamisel võiks pahatahtlik kõrvaline isik saata pilve valeandmeid. Eriti kriitiliseks muutubki turvaline suhtlus siis, kui platvorm võimaldab andmetele reageerida ning seeläbi saab pahategija näiteks kodust valvesüsteemi eksitada. Kui

Rakenduse järgmine versioon keskendubki paremale seadmehaldusele ning turvalisusele - eesmärgiks on kaotada vajadus seadmeid manuaalselt konfigureerida ja uuendada ning parendada andmevahetuse turvalisust. Vastavad edasiarendused jäid esimeses versioonis teostamata põhiliselt seadmete endi võimekuse tõttu. Käesoleval hetkel ei toeta seadmed veel läbi võrgu uuendamist või konfigureerimist, samuti puudus seadmetel võimekus pilvega suhtlust turvalisemaks muuta. Vastavad edasiarendused viiakse seega läbi paralleelselt nii pilverakenduses kui ka seadmete tarkvaras.

Kasutajaliidese ja andmete visualiseerimise üks võimalikke edasiarendusi on avalehel kuvatava seadmegrupi plaani joonistamise võimekus. Esimene prototüübi versioon eeldab plaani pildifaili olemasolu ning võimaldab selle kasutajal üles laadida. Samas oleks visuaalse ühtsuse tagamiseks mõistlik pakkuda primitiivsemat plaani või skeemi joonistamise võimalust. Lisaks võiks rakenduse sisu muuta vastavalt kasutatava seadme ekraanilaiusele, mille tulemuseks oleks adaptiivne ehk kohalduv disain.

7 Kokkuvõte

Käesoleva töö eesmärgiks oli IoT andmete kogumiseks ning visualiseerimiseks vajaliku platvormi loomine. Selleks analüüsiti ning võrreldi erinevaid IoT lahenduste teostamist toetavaid pilveplatvorme ning sobivaima platvormi baasil loodi prototüüp-rakendus, mis võimaldab teaduritel väikese vaevaga süsteemis seadmeid registreerida, neilt andmeid koguda ning andmeid reaalajas visualiseerida. Rakendus valmis iteratiivse protsessi tulemusena, mistõttu oli lõplik skoop lahtine. Süsteemi arendati vastavalt ajaraamile ning loodavad funktsionaalsused olid reastatud prioritiseerituna, et olulisimad elemendid saaksid implementeeritud.

Süsteemi funktsionaalsusi kirjeldavad kasutuslood defineerivad skoobi, mida antud projekti raames jõuti valmis teha. Võib öelda, et prototüübi loomine õnnestus plaanipäraselt ning miinimumnõuded said esimese versiooni puhul täidetud. Prototüüp on tõestuseks sellele, et pilveplatvormil on võimalik väikekasutajatele mõeldud lihtsama nutikodu või muu targa keskkonna seadmete andmeid koguda ning visualiseerida. Samuti andis loodud prototüüp piisavalt informatsiooni, aitamaks edasiarendatud versioonina loodavat lahendust teenusena pakkuda.

Samas töö prototüübiga jätkub, et valmiks esmalt täiustatum versioon ning hiljem ka edasiarendatud rakendus, mis oleks aluseks uuele IoT andmete kogumise teenusele või platvormile. Süsteemi pilveplatvormiks valitud Google Cloud on piisavalt võimekas ja pakub palju skaleerimise võimalusi, mistõttu sobib loodud rakenduse edasiarendus teenusena samuti Google'i pilveplatvormis käitamiseks.

Kasutatud kirjandus

Ainsalu, O. (2016). Veebilehtede käideldavuse ja kasutatavuse hindamine erivajadustega inimestele avaliku sektori veebilehtede näitel : magistritöö. Tallinn, Tallinna Tehnikaülikool

Amazon Web Services veebileht. [WWW] https://aws.amazon.com/ (22.03.2017)

Angular.js veebileht. [WWW] https://angular.io/ (02.04.2017)

Aosong Electronics DHT22 sensori spetsifikatsioon. [WWW] https://www.sparkfun.com/datasheets/Sensors/Temperature/DHT22.pdf (01.05.2017)

Ashton, K. (2009) That ‘‘Internet of Things’’ thing. RFiD Journal

Beck, K., Cockburn, A., Jeffries, R., Highsmith, J. Agile Manifesto. [WWW] http://www.agilemanifesto.org (12.03.2017)

BeeBotte veebileht. [WWW] https://beebotte.com/ (17.03.2017)

Carlson, C.R., Wilmot, W.W. (2006) Innovation: The Five Disciplines For Creating What Customers Want. New York: Crown

Carriots veebileht. [WWW] https://www.carriots.com/ (19.03.2017)

David Cohen, M. L. (2003). Agile Software Development, Data & Analysis Center for Software

Docker veebileht. [WWW] https://www.docker.com/what-docker (28.04.2017)

Espressif Systems ESP8266EX WiFi mooduli spetsifikatsioon. [WWW] http://download.arduino.org/products/UNOWIFI/0A-ESP8266-Datasheet-EN-v4.3.pdf (01.05.2017)

Express.js veebileht. [WWW] https://expressjs.com (18.04.2017)

Fielding, R. (2000) Architectural Styles and the Design of Network-based Software Architectures : doktoritöö, University of California, Irvine

Flux veebileht. [WWW] https://facebook.github.io/flux (03.04.2017)

Git veebileht. [WWW] https://git-scm.com/ (15.04.2017)

Gould, J.D., Lewis, C. (1985) Designing for usability: key principles and what designers think New York: Association for Computing Machinery

Gubbi, J., Buyya, R., Marusic, S., Palaniswami, M. (2011) Internet of things (IoT): A vision architectural elements and future directions, Future Gen. Comput. Syst., vol. 29, no. 7, pp. 1645-1660

Heroku veebileht. [WWW] https://www.heroku.com (21.03.2017)

IBM Watson IoT veebileht. [WWW] https://www.ibm.com/internet-of-things/platform/watson-iot-platform/ (18.03.2017)

InVision veebileht. [WWW] https://www.invisionapp.com/ (02.04.2017)

Ip, C. (2016) The IoT opportunity – Are you ready to capture a once-in-a lifetime value pool?, McKinsey Global Institute Analysis

JWT veebileht. [WWW] https://jwt.io (12.04.2017)

Kaa veebileht. [WWW] https://www.kaaproject.org/ (20.03.2017)

Lauk, K. (2015). Veebilehtede kasutatavuse hindamine elektrimüügi ettevõtete näitel : magistritöö. Tallinn, Tallinna Tehnikaülikool

Microsoft Azure veebileht. [WWW] https://azure.microsoft.com (22.03.2017)

Mikowski, M., Powell, J. (2013) Single Page Web Applications: JavaScript end-to-end, Manning Publications

Nanobox veebileht. [WWW] https://nanobox.io (22.03.2017)

Nielsen, J. (1993) Usability Engineering, Academic Press, Inc., San Diego

Node.js veebileht. [WWW] https://nodejs.org (30.03.2017)

Postman veebileht. [WWW] https://www.getpostman.com/ (01.05.2017)

React.js veebileht. [WWW] https://facebook.github.io/react (20.04.2017)

RedHat OpenShift veebileht. [WWW] https://www.openshift.com (24.03.2017)

Redux veebileht. [WWW] http://redux.js.org (10.04.2017)

Rogers, Y., Sharp, H., Preece, J. (2015) Interaction design: beyond human-computer interaction. 4th ed., J. Wiley & Sons

Rubin, J., Chisnell, D. (2008). Handbook of Usability Testing. Second Edition: How to Plan, Design, and Conduct Effective Tests. Indianapolis, Indiana : Wiley Publishing, Inc

Shackel, B., Richardson, S. J. (1991) Human Factors for Informatics Usability. Cambridge, University Press

Spec. 73. (2015) Feedback Friends (feat. Katie Dill), Spec Network, Inc [WWW] http://spec.fm/podcasts/design-details/19155 (08.02.2017)

Sundmaeker, H., Guillemin, P., Friess, P., Woelfflé, S. (2010) Vision and challenges for realising the Internet of Things, Cluster of European Research Projects on the Internet of Things — CERP IoT

Texas Instruments MSP430F5529 LaunchPad spetsifikatsioon. [WWW] http://processors.wiki.ti.com/index.php/MSP430F5529_LaunchPad (01.05.2017)

ThingSpeak veebileht. [WWW] https://thingspeak.com/ (17.03.2017)

Ubidots veebileht. [WWW] https://ubidots.com/ (18.03.2017)

Uukkivi, M. (2006). Kasutajakeskne veebidisain: Õppevahendi loomine ja kasutajakeskuse testimine : magistritöö. Tallinn, Tallinna Ülikool

Lisa 1 – Ülesanded kasutajale disainiprototüüpide kasutajatestimisel

Paberprototüübi testimiseks kasutatud ülesanded

1. Leia magamistoa termoanduri hetkeväärtus
2. Lisa süsteemi uus sensor
3. Eemalda sensor süsteemist

Digitaalse prototüübi testimiseks kasutatud ülesanded

1. Leia elutoas asuva valgussensori hetkeväärtus
2. Lisa süsteemi uus seade, millel on üks termoandur
3. Eemalda lisatud seade süsteemist
4. Muuda seadmegrupi plaani