Tampereen yliopisto

Niko Vuorinen

IOT-LAITTEEN HALLINTA PILVIPALVELUNA

Informaatioteknologian ja viestinnän tiedekunta
Diplomityö
Huhtikuu 2024

TIIVISTELMÄ

Niko Vuorinen: IoT-laitteen hallinta pilvipalveluna
Diplomityö
Tampereen yliopisto
Tietotekniikan diplomi-insinöörin tutkinto-ohjelma
Huhtikuu 2024

Tässä työssä tutkittiin, miten luoda laitehallintapalvelu IoT-laitteille. Pääajatuksena oli luoda web-palvelu, jota voitaisiin käyttää ympäri maailmaa, joten luonnollinen ratkaisu oli toteuttaa palvelu pilveen. Valmiita sovelluksia IoT-laitteiden antureiden mittauksiin on useita, mutta tähän käyttötarkoitukseen toimivaa ratkaisua, jossa tutkitaan IoT-laitteen tilaa ei löytynyt valmiina palveluna. Laitteidenhallinta-palvelu oli osa isompaa kokonaisuutta, josta tässä työssä käsitellään vain laitehallintaan liittyviä osia.

Laitehallintapalvelun testilaitteena käytettiin IoT-Gatewayta, joka oli sopivan yksinkertainen laite, jolla voitiin testata palvelun toimivuutta. Kyseisen laitteen kanssa on tullut aiemmin vastaan tilanteita, joissa on selvinnyt laitteen katkaiseen yhteyden verkkoon, mutta ei syytä tälle. Palvelun kannalta oleellista oli myös ymmärtää tarve useamman laitteen yhtäaikaiselle hallinnalle, joka sisältää sekä laitteen päivitykset että tilan tarkastelun.

Työn teoria perustuu esineiden internettiin ja pilvipalveluihin. Teoriaosuudessa syvennytään siihen, miten web-palvelut on rakennettu ja millaisia teknologioita niissä käytetään. Työssä esitellään myös toteutukseen valitut teknologiat yleisellä tasolla. Sovelluksen rajapinta on kirjoitettu Pythonilla käyttäen Djangoa ja Django REST Frameworkia. Käyttöliittymät on kirjoitettu myös Pythonilla käyttäen Djangoa, mutta esimerkiksi mittaustietojen kuvaajia ja HTTP-pyyntöjä on kirjoitettu Javascriptilla. Relaatiotietokantana palvelussa toimii PostgreSQL.

Työn toteutusvaiheessa saatiin luotua toimiva pilvipalvelu, joka toteuttaa laitehallinnan IoT-laitteille. Palveluun on yksinkertaista lisätä, muokata ja poistaa laitteita. Laitteiden tilaa on myös yksinkertaista seurata. Työssä ilmeni, että laitteilta tulee niin paljon mittaustuloksia, että niiden hallinnointi relaatiotietokannassa on haastavaa. Lisäksi päivitysten hallinnointi ei toiminut niin yksinkertaisesti kuin kuviteltiin ja sen kehitys jäi hieman kesken. Jatkokehityksenä pitäisi tarkastella perusteellisemmin mittaustulosten tallennusta ja kehittää päivitystenhallinta loppuun. Työn lopputuloksena havaittiin, että laitehallinnan toteuttaminenweb-palveluna oli hyvä valinta toteutuksen pohjaratkaisuksi, mutta relaatiotietokannan hyödyntämistä pitäisi tutkia tarkemmin.

Avainsanat: IoT, Pilvipalvelut, Web-sovellus

Tämän julkaisun alkuperäisyys on tarkastettu Turnitin OriginalityCheck -ohjelmalla.

ABSTRACT

Niko Vuorinen: IoT-device Management as Cloud Service
Master's thesis
Tampere University
Master´s Degree Programme in Information Technology
April 2024

In this work, we explored how to create a device management service for IoT devices. The main idea was to create a web service that could be used around the world, so the natural solution was to implement the service in the cloud. There are several ready-made applications for measuring sensors on IoT devices, but a solution that would work for this application, which would examine the status of an IoT device, could not be found as a ready-made service. The device management service was part of a larger whole, of which only the device management parts are discussed in this thesis.

The test device used for the device management service was the IoT Gateway, which was a simple enough device to test the functionality of the service. With this device, situations have been encountered in the past where it was discovered that the device had lost connection to the network, but no reason for this. It was also essential for the service to understand the need for simultaneous management of multiple devices, including both device updates and status checking.

The theory of the work is based on the Internet of Things and cloud computing. The theoretical part of the thesis delves into how web services are built and what technologies are used. It also presents the technologies chosen for implementation at a general level. The application interface is written in Python using Django and the Django REST Framework. The user interfaces are also written in Python using Django, but for example the measurement data graphs and HTTP requests are written in Javascript. The relational database used in the service is PostgreSQL.

In the implementation phase of the work, a functional cloud service was created that implements device management for IoT devices. It is simple to add, edit and delete devices. It is also simple to monitor the status of the devices. The work revealed that there are so many measurement results from devices that managing them in a relational database is challenging. In addition, the management of updates was not as simple as imagined and its development was somewhat incomplete. Further development should look more in depth at the storage of measurement results and finalise the update management. The final conclusion of the work was that the implementation of the instrument management as a web-based service was a good choice as a basic implementation solution, but the use of a relational database should be further investigated.

Keywords: IoT, Cloud services, Web application

The originality of this thesis has been checked using the Turnitin OriginalityCheck service.

ALKUSANAT

Tämän diplomityön tarina on monivaihenen. Työtä on alunperin alettu tekemään Tampereen teknillisen yliopiston signaalinkäsittelyn laitokselle. Yliopistojen yhdistymisen ja organisaatiomuutosten jälkeen työ on ilmeisesti tehty Tampereen yliopiston tietotekniikan yksikölle. Työtä lähdettiin aluperin tekmään Realin Oy:lle, jotta saataisiin palvelu, jolla hallita IoT-Gatewayta. Kiireiden myötä työn tekeminen hidastui ja välillä pysähtyi. Lopulta työ toteutettiin loppuun Vincit Oyj:llä. Realinilla työtä eniten ohjaili TkK Juha-Pekka Sarin ja Vincitillä DI Juha Simola ja TkT Essi Isohanni.

Realin Oy on Tampereen teknilliseltä ylopistolta lähtöisin oleva ohjelmistoyritys. Yrityksen ohjelmistot liittyvät IoT-laitteiden antureiden mittausten tallentamiseen ja mittaustiedon esittämiseen ja analysointiin. Yritys on vakavarainen ja yrityksessä on kuusi työntekijää.

Vincit Oyj on tamperelainen ohjelmistoyritys, joka tuottaa digitaaliseen kiihdyttämiseen ja kokonaisvaltaiseen kauppaan liittyviä palveluita. Yritys toimii tällä hetkellä Suomessa, Ruotsissa, Puolassa ja Yhdysvalloissa.

Työn osalta haluaisin kiittää erityisesti seuraavia henkilöitä. Työn ohjauksesta TkK Juha-Pekka Sarin, TkT Marko Hännikäinen, DI Juha Sarin ja TkT Essi Isohanni. Lisäksi haluaisin kiittää Vincit Oyj:tä kahden viikon diplomityövapaasta. Myös erityismaininnat ansaitsevat työn potkimisesta eteenpäin TkT Veli-Pekka Eloranta ja Sosionomi Piritta Karppelin. Työn tarkastaja professori Hannu-Matti Järvinen ansaitsee myös suuret kiitokset pitkästä pinnasta työn suhteen ja yhteistyöstä erinäisissä hallinnon elimissä. Viimeisenä haluan myös kiittää rakasta tee-kerhoa, te ette auttaneen ollenkaan tämän työn kanssa, mutta muuten oli hauskaa.

Tampereella, 30. huhtikuuta 2024

Niko Vuorinen

SISÄLLYSLUETTELO

1 Johdanto . . . 1
2 Käyttötapaus . . . 3
2.1 Esineiden Internet . . . 3
2.2 Pilvipalvelu . . . 6
2.3 IoT-Gateway . . . 7
3 Valitut tekniikat . . . 9
3.1 Web-sovellus . . . 10
3.2 Arkkitehtuurivalinnat . . . 11
3.3 Käytetyt tekniikat . . . 12
4 Palvelun toteutus . . . 14
4.1 Valitut ratkaisut . . . 15
4.2 Toteutus . . . 16
4.3 Toiminnallinen kuvaus . . . 17
4.4 Laitteen toiminta . . . 21
4.4.1 Laitevaatimukset . . . 22
4.4.2 Laitetyypit . . . 23
4.5 Päivitysten hallinta . . . 23
4.6 Testaus . . . 24
4.7 Käyttöönotto . . . 24
5 Jatkokehitys ja arviointi . . . 26
5.1 Ratkaisu: Välimuistin hyödyntäminen tietokantaoperaatioissa . . . 26
5.2 Ratkaisu: Tietokantataulujen sovittaminen dataan . . . 27
5.3 Ratkaisuehdotusten analysointi . . . 28
5.4 Muita jatkokehitysajatuksia . . . 28
6 Johtopäätökset . . . 31
Lähteet . . . 33

LYHENTEET JA MERKINNÄT

ACID	Atomicity, Consistency, Isolation, Durability
AWS	Amazon Web Services
DRF	Django Rest Framework
IoT	Internet of Things
JSON	Javascript Object Notation
NoSQL	Not Structured Query Language
REST	Representational state transfer
RFID	radio frequency identification
SQL	Structured Query Language
WSN	Wireless Sensor Network

1 JOHDANTO

Ihmisillä on aina ollut tarve selittää, millaiset ovat tämän hetken olosuhteet. Jokaiselle on tuttua, että kun keitetään perunoita, niin oikean keittolämpötilan tietää siitä, että vesi kiehuu. Mittaustekniikan kehittyessä ihmiset ovat kehittäneet tapoja myös tarkempaan mittaukseen. Elohopeamittarilla saadaan selville asteen tarkkuudella vallitseva lämpötila. Tekniikan kehittyessä anturit ovat vähitellen tulleet tarkemmiksi ja digitalisoituneet. Digitalisoitumisen ja Internetin kehittymisen seurauksena anturit alkoivat siirtyä Internettin. Internetin kautta tarkoituksena on tutkia anturin mittauksia, mutta kuka tutkii sitä, missä kunnossa varsinainen anturi on ja onko IoT-laite toimintakunnossa.

Työssä tutustutaan IoT-laitteiden hallintapalveluun, joka on toteutettu web-palveluna pilviympäristössä. Aiempi järjestelmä, joka perustui pääasiassa kynä ja paperi -menetelmiin, oli osoittautunut riittämättömäksi ja kankeaksi. Lisäksi laitteiden kehittyessä ne pystyivät lähettämään mittaustietoja itsestään ja päivittämään itsensä. Yrityksessä, johon palvelu tehtiin tilanne oli juuri sen kaltainen, että käytännössä laitehallintana toimi kokoelma excel-tiedostoja. Tämän lisäksi ymmärrettiin tarve kehittää uusi palvelu, joka mahdollistaisi laitteiden tilan tarkastelun ja päivitykset.

Vaikka laitehallinta sinänsä ei ole uusi keksintö, monet aiemmat sovellukset ovat olleet monimutkaisia tai tarjonneet vain rajoitetusti tietoa. Esimerkiksi perinteiset sovellukset saattoivat ilmoittaa ainoastaan, että reititin on verkossa, mutta eivät tarjonneet tarkempaa tietoa reitittimen tilasta tai ongelmista, kuten muistin täyttymisestä. Uusi palvelu tarjoaa yksinkertaisen tavan hallita useita laitteita ja samalla monipuolista ja reaaliaikaista tietoa laitteiden tilasta. Näin loppukäyttäjä voi selvittää, miksi laite ei toimi ja miten ongelma voidaan ratkaista tai välttää tulevaisuudessa. Tällä tavoin varmistetaan, että laitteiden tuottama data on jatkuvasti eheää ja luotettavaa. Palvelun suunnittelussa kiinnitettiin huomiota myös siihen, että palvelun haluttiin olevan yleiskäyttöinen, palvelun ei kuulu ottaa kantaa siihen, millaisia laitteita siihen liitetään.

Relaatiotietokantoihin jatkuva datan tallentaminen ei ole erityisen tehokasta. Lisäksi relaatiotietokannoissa datalla pitäisi olla valmiina rakenne, ja jatkuvan datavirran tapauksessa tämä ei ole itsestään selvää. Näiden lisäksi laitepäivityksiin liittyy asioita, jotka ei alkuvaiheessa ole vielä kovin selkeitä. Tästä syystä työn tutkimuskysymykset ovat seuraavat:

- Miten hyödyntää relaatiotietokantoja jatkuvan datavirran kanssa?
- Kun yllä olevan ongelman ratkaisee, niin mitä muita ongelmia ratkaisu voi aiheuttaa? Miten nämä saadaan ratkaistua?

- Mitä pitää huomioida tarjottaessa päivityspaketteja?

Näihin kysymyksiin vastattiin rakentamalla sovellus, joka ottaa jatkuvaa datavirtaa sisään ja tarjoaa päivityspaketteja. Työn asiakasyrityksessä on jo aiemmin luotu palveluita, jotka tallentavat relaatiotietokantaan IoT-laitteiden antureiden mittauksia.

Projektissa kehitetty sovellus on osa laajempaa kokonaisuutta, jonka avulla voidaan käsitellä IoT-laitteilta tulevia mittaustuloksia. Tässä työssä keskitytään kuitenkin pääasiassa laitehallintapalvelun kehittämiseen eikä itse laitteiden toimintaan. Laitteille tarjotaan rajapinnat, joiden avulla mittaustiedot voidaan siirtää palveluun. Työssä käsitellään laajempaa järjestelmää vain niiltä osin, jotka liittyvät suoraan laitehallintaan.

Ohjelmistokehitysprojektissa on noudatetaan ketteriä kehitysmenetelmiä, mutta tietyt viitekehykset tai menetelmät eivät olleet projektin alkaessa vielä lukittuja. Projektin alussa ei ollut olemassa olemassa valmista suunnitelmaa projektin toteuttamiseksi, mikä antaa joustavuutta kehitystyölle. Toisaalta se myös esti vesiputousmallin käyttämisen. Työn tekemisen aikana dokumentoitiin ohjelmiston toiminta tähän diplomityöhön. Lopuksi palataan pohtimaan, että mitä kehitettävää järjestelmään on vielä jäännyt. Kehitysmenetelmät on valittu lähinnä sen takia, että yrityksen organisaatiossa työt ovat vahvasti henkilöityneet, jolloin pyrähdyksiin perustuva kehittäminen ei ole tarkoituksenmukaista.

Tässä diplomityössä ongelman ratkaistiin konstruktiivisella menetelmällä kehittämällä webpalvelu, joka perustui PostgreSQL-tietokantaan. Tietokantaa käytetään Python-pohjaisella Django-ohjelmistokehyksellä luodulla REST-rajapinnalla. Tätä rajapintaa käytetään Djangolla luodulla käyttöliittymällä.

Aluksi työssä esitellään taustalla olevaa teoriaa pilvipalveluista ja esineiden internetistä, jotta lukijalle muodostuu kokonaiskuva aiheesta. Tämän jälkeen käsitellään valittuihin tekniikoihin liittyvää teoriaa, joka auttaa ymmärtämään niiden käyttöä ja hyötyjä projektin kontekstissa. Kun teoriat on käsitelty, seuraavassa vaiheessa esitellään tarkemmin, miten laitehallintapalvelu on toteutettu ja miten eri komponentit toimivat yhdessä. Lopuksi pohditaan mahdollisia kehityssuuntia tulevaisuudessa ja kuinka nykyisiä ominaisuuksia voitaisiin parantaa. Lopullisessa arvioinnissa tarkastellaan myös työn onnistumista ja siihen liittyviä saavutettuja tuloksia.

2 KÄYTTÖTAPAUS

Luvussa esitellään perusteellisesti IoT:n, eli esineiden Internetin (Internet of Things) käsitteitä ja taustateoriaa, jotta lukijalle muodostuu vahva ymmärrys aihepiiristä. IoT-laitteiden pääasiallinen tarkoitus on tarjota käyttäjille internetin välityksellä etäpalveluita, yleensä jonkin tyyppisiä mittauksia. Esimerkkilaitteena käytetään IoT-reitititintä, joka toimii antureiden tukiasemana. Reitittimen avulla voidaan ottaa yhteyttä antureihin ja välittää niiden keräämää tietoa palvelimelle, johon tiedot tallennetaan. Reitittimen tehtävänä on myös varmistaa lähetettävän datan eheys, joten se voi suodattaa pois tarpeettoman tai päällekkäisen datan. Kriittistä kuitenkin on varmistaa reitittimen toiminta ja laitteen kunto, jotta tiedetään, että laite toimii oikein ja datan tallentuminen sekä oikeaoppinen parittaminen antureiden ja mittaustulosten välillä onnistuvat.

Aliluvussa 2.1 esitellään tarkasti, mitä termi "esineiden internet" tarkoittaa ja käsitellään termin historiaa. Tämän jälkeen syvennytään datan tallentamiseen tietokantoihin ja eritellään haasteita, joita esiintyy esineiden internetissä. Aliluvun loppupuolella paneudutaan myös tietoturvaan ja käsitellään datan yhdistämistä muihin lähteisiin esineiden internetissä.

Aliluvussa 2.2 käsitellään pilvipalveluita ja niiden perusteita. Lukija saa kattavan kuvan pilvipalvelujen toimintaperiaatteista ja käsitteistä. Erilaiset palvelutyypit selitetään esimerkkipalveluiden avulla. Lisäksi pohditaan eri pilvipalveluntarjoajien välisiä eroja ja annetaan näkökulmia tietyn palveluntarjoajan valintaan.

Aliluvussa 2.3 esitellään esimerkkilaitetta, IoT-Gatewayta, ja kerrotaan yleisellä tasolla sen tarkoituksesta ja siitä, millaisia ongelmia laite ratkaisee. Tämän yhteydessä käsitellään myös LoRaWAN-verkkoja, koska ne ovat läheisesti yhteydessä IoT-Gatewayn toimintaan.

Luvun tarkoituksena on antaa perusteellinen johdatus aiheeseen ja luoda vankka pohja seuraavalle luvulle. Näin lukija pystyy paremmin seuraamaan ja ymmärtämään luvun 3 sisältöä.

2.1 Esineiden Internet

Esineiden Internet (Internet of Things, IoT), on käsite, jolla tarkoitetaan datan keräämistä jatkuvasti ympäröivästä maailmasta ja datan analysointia reaaliajassa. Esineiden Internet tarkoittaa, että ihminen ei syötä dataa palvelimelle, vaan laitteet itse keräävät datan palvelimelle. Esineiden Internetin tarkoituksena on kehittää älykkäämpää ympäristöä ja vähentää resurssien käyttöä. Alunperin ajateltiin laitteiden olevan RFID-lukijoita mutta nykyään IoT-laite käsitteen alla olevat laitteet voivat olla myös muunlaisia antureita, tai jopa virtuaalisia antureita, joiden

tulokset on johdettu muista antureista. Internetin ja prosessointitehon kehitys on viime vuosina mahdollistanut datan laajemman keräämisen ja siirron, jonka takia IoT:sta on tullut suosittu konsepti. [1]

Kuitenkin IoT:ssa on ongelmana turvallinen tiedonsiirto, jonka toteutumista yritetään saavuttaa esimerkiksi erilaisilla salausmekanismeilla, jotta ulkopuoliset eivät pääsisi dataan käsiksi. Jos ajatellaan hypoteettista tilannetta, jossa huoneessa olisi aktiviteettianturi, sen avulla olisi helppo tietää, milloin paikalla on ihmisiä ja milloin ei. Jos tämän datan saa käsiksi esimerkiksi murtovaras, hänen on helppo ajoittaa toimintansa siihen hetkeen, kun paikalla ei ole ketään ihmisiä. Tietoturvaa IoT:ssa kuuluu myös pohtia sen kannalta, tekeekö Internetiin kytkeytyminen muita järjestelmiä suojamattomaksi. Esimerkiksi voi olla tilanne, että lähiverkon saman portin kautta, josta data menee ulkomaailmaan pääseekin vahingossa käsiksi ohjausjärjestelmiin ja tämän jälkeen jokin kriittinen järjestelmä on avoin hyökkäyksille. [2]

Antureiden lähettämään dataan liittyy monesti myös kysymys, miksi dataa tallennetaan. James Manyika kuvasi, kuinka paljon dataa tallennetaan ja kuinka paljon sitä voisi hyödyntää. Esimerkkinä käytettiin öljynporauslauttaa, jossa on 30000 anturia, mutta vain yhtä prosenttia näistä antureista hyödynnetään. Tämä tarkoittaa luonnollisesti sitä, että suurin osa datasta menee hukkaan. Artikkeli on kirjoitettu vuonna 2015, mutta tilanne tuskin on paljoa parantunut. [3]

Manyika myös väitti, että vuonna 2025 kroonisten sairauksien mittarointi voisi luoda $1,1$ biljoonan dollarin arvon vuodessa. Tämä kuvastaa sitä, kuinka paljon dataa voisi hyödyntää ja kuinka paljon dataa nykypäivänä päivänä jää hyödyntämättä. Dataa hyödyntämällä päätöksenteossa voisi kuitenkin saada tehtyä mittavia säästöjä ja parempia päätöksiä terveydenhuollossa. Tämä johtuu siitä, että datan määrän lisääntyessä voidaan saada parempaa tietoa ja sitä kautta myös parempaa tietämystä. [3]

Johtuen datamääristä, jota IoT-laitteet tuottavat, on monesti otettu käyttöön erilaisia tietokantoja ja viestinvälityspalveluita perinteisen relaatiotietokannan rinnalle. Relaatiotietokannoissa ongelmiksi muodostuvat aikaleimatun tiedon tallentaminen, tiedon koko, skaalaus ja indeksointi. SQL-tietokannat omaavat jo määritelmän mukaan rakenteen ja IoT-laitteiden datan rakenne voi olla erilainen jo eri suureiden kohdalla, jolloin vaaditaan tarkkaa suunnittelua tietokannan rakenteelle. IoT-laitteissa on mahdollista samalta laitteelta tulevat useamman suuren mittauksia, jotka voivat olla sekä kokonaislukuja että liukulukuja. On myös mahdollista, että monidimensioinen data on tiivistetty yhteen dimensioon tai IoT-laite lähettää historiadataa mittauksista. [4]

Tietokannoissa perinteisesti käytetään ACID-periaatetta, jonka tarkoituksena on varmistaa tietokannan eheys ennen ja jälkeen tietokantaoperaatioiden. ACID-akronyymi tulee sanoista atomicity, consistency, isolation ja durability, eli suomeksi atomisuus, eheys, eristyneisyys ja pysyvyys. Atomisuus tarkoittaa sitä, että tietokantatapahtuman pitää tapahtua kokonaan tai ei ollenkaan. Eheys tarkoittaa sitä, että tietokantatapahtuma noudattaa tietokannalle asetettuja sääntöjä ja operaatioiden jäelkeen tietokanta jää eheään tilaan. Eristyneisyys tarkoittaa sitä, että jos tietokantatapahtumia tapahtuu samanaikaisesti niiden ei pidä vaikuttaa toisiinsa. Py-

syvyys tarkoittaa sitä, että kun tietokantatapahtuma on tapahtunut, se selviytyy häiriöistä, kuten tietokantapalvelimen sammutuksesta. Näiden periaatteiden avulla voidaan olla melko varmoja, että tiedot säilyvät tietokannassa. [5]

IoT-laitteille ACID-periaatteet eivät ole itsestään selviä, koska dataa voi esimerkiksi tietoliikennehäiriöiden takia kadota. Eri IoT-laitteilla voi olla myöskin erilaisia tapoja esittää samaa dataa, ja tämä pitää huomioida datan tallentamisessa. Lisäksi tiedoista osa on julkisia ja osa suojattuja. Ongelmia ratkaisevat erilaiset NoSQL-kannat kuten Memcached tai Redis, jotka molemmat toimivat kuten web-palveluissa käytettävät välimuistit. Näillä pystytään nopeasti sekä tallentaa että hakea tietoa. Tarkoituksena ei ole kuitenkaan säilöä dataa pitkiksi ajoiksi vaan tallentaa tieto lyhyeksi ajaksi. Yleensä datat lopulta tallennetaan myös SQL-tietokantaan, jossa ne säilyy paremmin kuin NoSQL-tietokannoissa. Tietyissä NoSQL-tietokannoissa on esimerkiksi mahdollista, että vanhat datat pyyhitään pois, kun tietokannan koko tulee täyteen, tämän jälkeen luonnollisesti ei vanhaa dataa pysty enää käsittelemään. SQL-tietokannoista dataa voidaan taas myöhemmin hyödyntää lukuoperaatioissa, datan lukeminen on myös merkittävästi nopeampaa kuin datan kirjoittaminen. Kun data on tallennettuna tietokantaan, sitä voidaan hyödyntää esimerkiksi luomalla virtuaalisia antureita tai päätelemällä ympäristön tilaa. [4]

Toukokuussa 2018 voimaan tulleen Euroopan unionin GDPR-tietosuojalain johdosta tarkasti henkilöidyt tiedot pitää anonymisoida, olla aina tietoihin liittyvän henkilön saatavilla ja olla aina poistettavissa. Samalla pitää olla myös tiedossa, missä tiedot ovat fyysisesti, joka voi tuoda erinäisiä ongelmia kun kyseessä on pilvipalvelu. Tällöin pitää varmistaa, että on tiedossa, missä tallennetut tiedot sijaitsevat. Esineiden Internetissä ongelmana saattaa olla, että mistä tietää, mikä tieto on henkilöityä ja mikä ei. [6]

GDPR-laki tulee myös huomioida siinä mielessä, että monien eri antureiden dataa yhdistämällä on mahdollista luoda tietämystä hyvinkin kompleksista tilanteista ja luoda tietoa, jota ei oikeasti edes millään tavalla mitata. Tällöin voi olla mahdollista luoda tietoa siitä, että mitä joku tietty henkilö ihminen tekee milloinkin ja millainen päivärytmi henkilöllä on esimerkiksi jossain huoneessa. Uuden tietosuojalain kuitenkin suurimpana voittona voidaan pitää, että ihmiset voivat paremmin luottaa siihen, että heidän henkilötietonsa ovat turvassa. [6]

Datan yhdistäminen ei itsessään ole pelkästään huono asia. Datan yhdistämisellä voidaan luoda esimerkiksi virtuaalisia antureita ja datan yhdistäminen on myös tärkeä osa data-analyysiä. Esimerkkinä voisi miettiä älykaupunkeja, niitä varten on olemassa monia erilaisia datalähteitä ja antureita, joiden avulla voidaan luoda kaupungista suurempaa tietoisuutta. Esimerkiksi Aarhusin kaupungissa ennustetaan liikenneruuhkia käyttäen tukivektorikonetta esineiden Internetin dataan. Yksi aspekti älykaupungeissa datan keräämiseen on myöskin kaupunkisuunnittelu. Sen lisäksi, että kerätään dataa näyttämään tämän hetken tilaa voidaan myös pyrkiä ennustamaan, miten tulevaisuudessa kaupunki tulee muuttumaan. Datan keräämisessä pitää kuitenkin muistaa, että kerättävän datan pitäisi olla hyödyllistä ratkaistavan ongelman kannalta. [7]

IoT on tärkeä palvelun kannalta, koska se tarjoaa olennaisen komponentin monille palveluratkaisuille. Tässä työssä käytettävään esimerkkilaitteeseen liittyvät anturit, jotka kuuluvat esineiden internetiin, ovat keskeisessä roolissa tiedon keräämisessä ja siirtämisessä palveluihin.

Näiden antureiden tuottama data voi olla arvokasta monissa eri sovelluksissa, kuten ympäristön seurannassa, teollisuuden valvonnassa, terveydenhuollossa tai älykotiratkaisuissa.

Vaikka esineiden internetin laaja-alaisuus ja sen mahdollistama datan hyödyntäminen ovat mielenkiintoisia, tämän työn pääpaino on kuitenkin pilvipalveluna toimivassa laitehallinnassa. Laitehallinta käsittää laitteiden etähallinnan, valvonnan ja konfiguroinnin pilvipalvelun avulla. Pilvipalvelu mahdollistaa laitteiden hallinnan ja datan analysoinnin reaaliajassa, tarjoten skaalautuvuutta, joustavuutta ja helppokäyttöisyyttä.

2.2 Pilvipalvelu

Pilvipalvelut ovat palveluita, jotka toimivat isoissa laskentayksiköissä, jotka on yhdistetty IP-verkon kautta toisiinsa. Pilvipalvelut voidaan jakaa julkisiin ja yksityisiin pilvipalveluihin. Julkista pilveä voisi ajatella yksinkertaisesti, että vuokrataan tarjoavalta yhtiöltä laitekapasiteettia omien tarpeiden mukaan, kun taas yksityinen pilvi tarkoittaa käytännössä yrityksen omaa pilvipalvelua. Julkisten pilvipalveluiden etuna pidetään yleensä sitä, että yrityksellä ei tarvitse olla omia konesaleja, eikä myöskään henkilöstöä, joka vastaa tietokoneiden fyysisestä hallinnasta. Myös tilanteessa, jossa esimerkiksi yrityksen tiloissa syttyy tulipalo, data säilyy edelleen pilvessä ja suurimmat menetykset ovat paikallisia. Pilvipalveluita pystytään käyttään myös hybridinä, eli osa pilvestä on julkisessa pilvessä ja osa yksityisessä. [8]

Pilvipalveluita on nykyään monia erilaisia, ja ne ratkaisevat todella kattavasti erinäisiä ongelmia. Pilvipalveluita on karkeasti rajattuna kolmea tyyppiä. Ensimmäinen tyyppi on infastruktuurin tarjoaminen palveluna (IaaS), esimerkkinä Amazonin EC2-palvelu, joka tarjoaa asiakkaalle palvelimen. Toinen tyyppi on alustan tarjoaminen palveluna (PaaS), esimerkiinä Amazonin Elastic Beanstalk -palvelu, joka tarjoaa sopivat resurssit perustuen siihen, minkälaisen ohjelman antaa palvelulle. Viimeinen tyyppi on sovelluksen tarjoaminen palveluna (SaaS), esimerkkinä Googlen Stadia, jonka tarkoituksena on pelaaminen pilvessä ilman, että lokaalisti käytetään tietokoneen resursseja. Pilvialustat lupaavat tietoturvan IaaS- ja PaaS-palveluissa laitteiston puolesta, mutta ohjelmiston tietoturva on asiakkaan vastuulla. Jos kyseessä on SaaS-palvelu, niin silloin palveluntarjoaja vastaa itse myös ohjelmiston tietoturvasta. Kuluttajalle näkyvimpiä palveluita ovat yleensä erilaiset verkkotoisto-, tiedostojenhallinta- ja tiedostojenjakopalvelut. Näiden palveluiden etuna on, että niitä voi käyttää missä tahansa ja aina on mahdollisuus käyttää palvelua. Yrityksillä on pilvipalveluina käytännössä mitä tahansa tietoliikenneverkkojen virtualisoinnista neuroverkon laskennan kautta palvelin ylläpitoon. Käytännössä kyse on monesti siitä, että halutaan päästä eroon laitteiden hankinnasta ja ylläpidosta, jolloin yleensä saadaan aikaiseksi säästöjä. Monesti myös tarvitaan varmuutta siitä, että laskenta pysyy ajossa tarpeeksi pitkään ja toimii tarpeeksi tehokkaasti. Omalla tietokoneella ongelmaksi voi muodostua, että laskentakapasiteettia ei ole riittävästi tai yksinkertaisesti aikaa ei ole tarpeeksi. Yritysmaailman oikea ongelma voi olla, että että käynnistätteässä pitkää laskentaa käyntiin, joku käy sen aikana laskenta-ajan aikana sammuttamassa tietokoneen. Kun laskenta on pilvessä voidaan varmistua siitä, että laskenta pyörii kuten on tarkoitettu, mikäli laskenta itsessään on kunnossa. [9, 8]

Pilvipalvelutarjoajia on nykyään useita, yleisimmät näistä varsinkin Suomessa on Amazon Web Services (AWS), Google Cloud Platform (GCP) ja Microsoft Azure. Eri palveluntarjoajilla on hieman erilaiset palveluvalikoimat, joten yleensä pilvipalvelun luominen eri ympäristöihin vaatii erilaiset konfiguraatiot. Pohjimmiltaan kuitenkin kaikki palveluntarjoajat tarjoavat virtuaalipalvelimen, jota voi käyttää kuten mitä tahansa palvelinta. Palveluntarjoajilla on kuitenkin eroa palvelusopimuksissa, joten kannattaa huomioida, mikä palveluntarjoaja tarjoaa sopivaa palvelusopimusta omille tarpeille. Palveluntarjoajaa valittaessa kannattaa myös kiinnittää huomiota siihen, miksi on valitsemassa juuri kyseistä palvelua. Esimerkkejä valintaperusteista voi olla esimerkiksi tietoturva tai laitteiston ominaisuudet. Huomioitavaa on myös, että eri palveluntarjoajilla on erilaiset hinnoittelumallit, jotka voivat perustua resurssien käyttöön, käyttäjämäärään tai kiinteään kuukausimaksuun. [10]

Tässä diplomityössä olevassa ratkaisussa pilvipalvelut ovat erittäin merkittävässä roolissa useista syistä. Ensinnäkin, pilvipalveluiden avulla palvelun skaalaaminen ja resurssien hallinta ovat huomattavasti helpompaa. Pilvipalveluntarjoajat tarjoavat joustavia resurssien skaalausominaisuuksia, joilla voidaan esimerkiksi lisätä ja vähentää resursseja palvelun kuormituksen mukaan. Toiseksi, pilvipalveluiden käyttö mahdollistaa paremmat ja luotettavammat verkkoyhteydet palvelimille, vaikka IoT-laitteet olisi hajautettuna ympäri maailmaa. Pilvipalveluntarjoajien globaalit verkostot ja tietoliikenneinfrastruktuuri tarjoavat tehokkaat ja nopeat yhteydet käyttäjien ja palvelinten välillä.

Kaiken kaikkiaan pilvipalvelut tarjoavat toteutettavan ratkaisun kannalta ratkaisevan tärkeitä etuja. Ne mahdollistavat palvelun skaalautumisen vaivattomasti, tarjoavat paremmat verkkoyhteydet hajautetuille laitteille ja tarjoavat laajan valikoiman palveluita, jotka voidaan skaalata ja räätälöidä omiin tarpeisiin. Pilvipalvelut ovat siis välttämättömiä infrastruktuurikomponentteja, jotka tukevat ja mahdollistavat tehokkaan ja toimivan palvelun toteutumisen.

2.3 IoT-Gateway

IoT-anturit eivät yleensä osaa keskustella suoraan ulkomaailman kanssa, vaan anturit ottavat yhteyttä IoT-Gatewayhin, josta asiakaslaitteet käyvät lukemassa antureiden lähettämän datan.

IoT-Gatewayn komponentit ovat yleensä antenni, joka vastaanottaa signaalit, mahdollinen signaaliprosessori ja verkkokortti. IoT-Gatewayksi riittää yksinkertainen yhden piirilevyn tietokone, kuten Raspberry Pi. IoT-Gateway on tärkeä laite siinä mielessä, että se mahdollistaa IoT-antureiden lukemisen Internetin yli. IoT-antureilta haetaan antureiden data käyttäen suunniteltua verkkoa, joka voi olla esimerkiksi Bluetooth tai LoRaWAN. Täten IoT-gateway mahdollistaa datan keräämisen reaaliajassa ja myös datan eteenpäin jakelun. Lisäksi IoT-Gatewayn avulla voidaan lähettää viestejä antureille, esimerkiksi älypistorasiaa ohjataan reitittimen avulla. IoT-Gateway on siis portti antureiden ja Internetin välillä. Internetin ei kuulu päästä suoraan kiinni antureihin eikä antureiden kuulu päästä kiinni Internettiin. IoT-Gateway lähettää eteenpäin metriikkaa antureista siten annotoituna, että metriikkatietoon on merkitty, miltä laitteelta data on tulossa, jotta jatkossa osataan käsitellä viestit oikein [11].

LoRaWAN (Long Range Wide Area Network) on radioteknologia, joka toimii pienellä taajuusa-

lueella (125 kHz - 500 kHz) ja se on suunniteltu käytettäväksi paristokäyttöisillä laitteilla. Tämän johdosta LoRaWAN teknologia vie vähän virtaa ja on toimiva IoT:n kanssa. Koska taajuusalueet ovat pieniäm tiedonsiirtonopeudet ovat luokkaa 3 $kbps$ - 30 $kbps$. Tämä on kuitenkin aivan riittävä IoT-antureille, koska yksittäinen IoT-anturin lähettämä datamäärä ei ole kovin suuri. [12]

Tässä työssä IoT-Gateway valittiin esimerkkilaitteeksi useista syistä. Ensinnäkin, IoT-ympäristössä laitteiden toimintahäiriöt voivat ilmetä ilman ilmeistä syytä, ja siksi on tärkeää pystyä päivittämään laitteita reaaliajassa tarvittaessa. Toiseksi, käsiteltävä laite on suhteellisen yksinkertainen, mutta sen rooli on erittäin merkittävä mittauksien tarkkuuden varmistamisessa ja datan luotettavassa välittämisessä. Laite on suunniteltu keräämään tietoa erilaisista mittausantureista ja lähettämään ne eteenpäin. Tämä vaatii laitteelta luotettavuutta ja tarkkuutta, jotta mittaukset ovat oikein ja datan siirto tapahtuu virheettömästi.

Kokonaisuudessaan IoT-Gatewayn käyttöönotto ja sen käyttö esimerkkilaitteena tuovat monia etuja. Se parantaa laitteiden toimintavarmuutta ja mahdollistaa niiden päivittämisen reaaliajassa tarvittaessa. Lisäksi se takaa mittauksien tarkkuuden ja datan luotettavan välittämisen. IoT-Gatewayn käyttö on tehokasta ja hallittua, mikä edistää parempaa palvelua ja parempia tuloksia IoT-ympäristössä.

3 VALITUT TEKNIIKAT

Tässä luvussa keskitytään perusteellisesti laitehallintapalvelun toteutukseen, valittuihin tekniikoihin ja valintapäätösten taustoihin.

Tekniikoiden valinnassa keskeistä on niiden sopivuus tarkoitukseen. Ohjelmistoissa on laaja valikoima erilaisia tekniikoita, ja valintakriteereinä voivat olla esimerkiksi aikataulu, jossa ohjelman on tarkoitus valmistua, sekä tehokkuusvaatimukset. Yleisesti voi todeta, että mitä helpompi sovellus on kirjoittaa, sitä enemmän se yleensä vaatii laskentatehoa.

Tekniikoiden valintaan vaikutti merkittävästi sovelluksen käyttötapaus. Koska kyseessä on pilvipalvelu, web-sovelluksen kehittäminen oli luonnollinen valinta, ja tämän myötä myös ohjelmistoarkkitehtuuri valikoitui helposti. Ohjelmointikieleksi valittiin Python, koska se oli jo entuudestaan tuttu ja laajasti käytetty yrityksessä. Vaikka web-sovellusten arkkitehtuuriin vaikuttaa myös valittu ohjelmistokehys, voidaan käytännössä todeta, että lähes kaikki web-sovellukset hyödyntävät MVC (Model-View-Controller) -mallia.

Aliluvussa 3.1 määritellään, mikä on web-sovellus ja mitä hyötyjä web-sovelluksilla on. Tämän jälkeen käsitellään web-sovellusten toiminnallista testausta ja esitellään erilaisia ei-toiminnallisen testauksen tyyppejä. Testauksen jälkeen tarkastellaan lyhyesti web-sovelluksille oleellista HTTP-teknologiaa. Aliluvussa esitellään yksinkertainen esimerkki web-sovellusta, joka toteuttaa verkkokaupan, ja käsitellään myös web-sovellusten hajautetun toiminnan ominaisuutta.

Aliluvussa 3.2 keskitytään arkkitehtuurivalintoihin ja niihin liittyvään teoriaan. Aluksi käsitellään laajasti käytettyä MVC-mallia, joka on suosittu monissa web-viitekehyksissä. Kun MVC-malli on tuttu, siirrytään tarkastelemaan sovelluksen keskeistä teknologiaa, Djangoa, ja kiinnitetään huomiota siihen, miten Django hyödyntää MVC-mallia. Lopuksi käsitellään REST-rajapintoja ja niiden yleisiä toimintaperiaatteita.

Aliluvussa 3.3 esitellään käytetyt tekniikat, jotka mahdollistavat sovelluksen toiminnan. Aluksi käsitellään JavaScriptiä, joka on olennainen web-sovellusten kehittämisessä. Tämän jälkeen tarkastellaan JSON-viestejä, jotka välittävät tietoa REST-rajapintojen kautta käyttöliittymän ja palvelimen välillä. JSON-viestien muotoon valittiin sovelluksessa WSN OpenAPI, jonka periaatteet käydään läpi. Viimeisenä käsitellään myös PostgreSQL-tietokannan ominaisuuksia.

Ottaen huomioon sovelluksen käyttötarkoituksen teknologiavalinnat olivat normaaleja.

3.1 Web-sovellus

Web-sovellus tarkoittaa sovellusta, jota käytetään Internetin yli yleensä web-selaimella. Web-sovellusten etuna on, että sovellus ei ole laitekohtainen, joten sitä pystyy käyttämään millä laitteella tahansa, josta pääsee yhdistämään Internetiin, ja laitteessa on web-selain. Web-sovelluksesta tehtäessä pitää yleensä pohtia erityisen paljon käyttöliittymää, koska on täysin mahdollista, että laitetta käytetään kosketusnäytöllä, puhelimella tai tietokoneella. Web-sovelluksessa on myös täysin mahdollista, että sovellus näyttää erilaiselta riippuen web-selaimesta tai laitteen käyttöjärjestelmästä. Tässä sovelluksessa on käytetty oletusta, että sovellusta käytetään tietokoneella, kuitenkin sovellusta on testattu jonkin verran älypuhelimella. [13]

Aliluvussa 3.2 käsitellään web-sovellukselle tyypillisiä rakenteita. Näistä rakenteista syntyy ohjelmistoon monimutkaisuutta, jonka pitää huomioida testaamisessa. Web-sovellusten testaaminen vaatii yksikkötestien lisäksi myös kattavia integraatio- ja järjestelmätestejä. Yksikkötestit voi jakaa palvelimen testaukselle ja selaimen testaukselle. Palvelintesteillä varmistetaan, että palvelin tuottaa oikeanlaista dataa selaimelle. Selaintestit varmistavat, että palvelimen data näytetään oikein loppukäyttäjälle. Integraatiotesteillä varmistetaan, että palvelimen integraatiot toimivat oikein. Järjestelmätesteillä varmistetaan, että koko järjestelmä selaimesta integraatioihin asti toimii oikein. [14]

Kaikki edellä mainitut testaamisen lajit ovat funktionaalista testausta. Funktionaalisen lisäksi web-sovellukselle voi tehdä myös ei-funktionaalista testausta, jotka jakaantuu ainakin seuraaviin lajeihin: suorituskyky-, kuormitus-, stressi-, yhteensopivuus-, käytettävyys-, saavutettavuus- ja turvallisuustestaus. Suorituskykytesteillä testataan, että tomiiko järjestelmä kuten suorituskyky on määritelty. Kuormitustesteillä testataan, miten järjestelmä toimii erilaisten kuormitusten kasvaessa. Stressitesteillä testataan, että ohjelmisto toimii jokseenkin hallitusti, kun järjestelmän maksimikapasiteetti ylitetään. Yhteensopivuustesteillä testataan, että sovellus toimii halutuissa ympäristöissä, esimerkiksi selaimilla. Käytettävyystesteillä testataan, että sovellus toimii kuten käyttäjät haluavat sitä käyttää. Saavutettavuustesteillä varmistetaan, että sovellus toimii vaikka käyttäjällä olisi joku sovellukseen käyttöön vaikuttava vamma. Turvallisuustestauksella pyritään varmistamaan, että tiedot tallentuvat turvallissesti ja kukaan ulkopuolinen ei pääse niihin käsiksi. [14]

Hypertext Transfer Protocol, eli HTTP on web-sovelluksen kannalta merkittävä asia erityisesti sen takia, että sen avulla voidaan hakea dataa sovelluksesta, myös sen takia, että sen avulla voidaan myös tallentaa sovellukseen tietoja. HTTP-protokollan perusajatus on, että palvelimelle lähetetään pyyntö ja palvelin lähettää siihen vastauksen. HTTP:lla on erilaisia metodeja, näistä yleisin on GET, jolla haetaan palvelimelta tietoa. Toinen yleinen metodi on POST, jolla lähetetään palvelimelle tietoa. Metodit ovat olennaisia REST-arkkitehtuurin kannalta, jota käsitellään aliluvussa 3.2. Suuremmassa mittakaavassa voidaan myös ajatella, että web-sovellukset ovat HTTP-protokollan päälle kehitetty tapa luoda sovelluksia. [15]

Tässä työssä käsitelty laitesovellushallinta ei ole kovin tyypillinen web-sovellus. Tyypillisempi esimerkki web-sovelluksesta olisi verkkokauppa. Verkkokaupassa on valmiina tietokannassa erilaisia tuotteita, joita voi ostaa. Taustalla jokainen uniikki tuote on oma tietokantarivinsä ja

tietokannassa lukee tuotteiden lukumäärä. Käyttötapaus alkaa siitä, kun asiakas kirjautuu web-selaimella verkkosivulle. Tällöin asiakas näkee tietokannassa olevat tuotteet ja voi valita sieltä kiinnostavat tuotteet. Asiakkalla on mahdollisuus mennä tuotesivulle, jolla näytetään tuotteen tiedot, jotka haetaan tietokannasta. Tuote voidaan lisätä lomakkeen avulla ostoskoriin, joka toimii välimuistissa. Lisätessä tuote ostoskoriin varmistetaan, että tuotteita on olemassa haluttu määrä. Kun tuote on lisätty ostoskoriin, voi asiakas käydä katsomassa, mitä ostoskorista löytyy, jolloin välimuistista haetaan halutut tuotteet. Ennen tilausta voidaan lomakkeella joko lisätä tuotteiden lukumääriä tai poistaa tuotteita ostoskorista. Kun ostoskori on valmis, asiakas syöttää lomakkeen avulla tietonsa palvelimelle, joka tallentaa asiakastiedot tietokantaan. Kun tuote on maksettu, tehdään tilauksesta vielä oma tietokantamalli, joka sisältää tiedot mitä tuotteita on tilauksessa ja mihin ne kuuluu lähettää, lisäksi tilauksessa on myös tilauksen tila. Tilauksen tila on siinä mielessä tärkeä, että kun esimerkiksi tila muuttuu vaikka lähetetyksi, asiakas saattaa haluta tästä sähköpostin. [16]

Web-sovelluksessa rajapintapyynnöt ovat sovelluksen suorituskyvyn pullonkaula. Sovellus voi olla hyvinkin hajautettuja, kuten aliluvussa 3.2 sivutaan. Tämän sovelluksen rajapinta voi olla hajautettuna muualle, joten osa sovelluksessa on ajossa toisaalla. Tämä on johtanut siihen, että viimeisimpiä datapisteitä on ladattuna valmiiksi välimuistiin, koska muuten jouduttaisiin tekemään hitaita rajapintapyyntöjä. Ongelma on korjattu tallentamalla datapiste suoraan välimuistiin, joten se on sieltä helposti saatavissa, kun se halutaan lukea. Tällöin on helppoa saada tietoonsa laitteiden tämän hetkinen tila. Synkronoinnin kannalta ainoa ongelma liittyy laitteiden ja palvelimen kellojen eroavaisuuteen. Tämä on ratkaistu siten, että laitteilta ei oteta kellonaikoja vastaan ollenkaan ja oletetaan lähetetyn datan olevan reaaliaikaista. Sovelluksessa luotetaan sokeasti siihen, että data jota rajapinta antaa on eheää. Rajapintaan on tehty erinäisiä suunnitteluratkaisuja, jotka pyrkivät ehkäisemään mahdollisia konfiguroinnin virheitä. Esimerkiksi rajapinta ei lähtökohtaisesti ota koskaan laitteelta aikaleimoja, koska aika voi laitteella olla väärin ja monissa tapauksissa tämä ei aiheuta merkittäviä ongelmia laitteen toimintaan.

3.2 Arkkitehtuurivalinnat

Koska kyseessä on web-sovellus sovellusarkkitehtuuri on noudattanut MVC-mallia (Models, View, Controller). Models, eli malleilla tarkoitetaan sovelluksen osaa, joka koostuu olennaisista käsitteistä ongelmanratkaisussa, tässä tapauksessa esimerkiksi laite, ja myös käsitteistä, jotka tukevat ongelmanratkaisua, tässä tapauksessa esimerkiksi lähde, joka voi koostua useasta laitteesta. View, eli näkymällä tarkoitetaan sovelluksen osaa, joka toteuttaa varsinaisen käyttöliittymän. Controller, eli käsittelijä, on sovelluksen osa joka koostaa malleista halutun datan näkymälle. [17]

Sovellus on tehty Djangolla, joten sovellusarkkitehtuuria on muokattu hieman MVC-mallista, ja Djangossa puhutaankin yleisesti MTV-mallista, joka koostuu malleista (models), sapluunoista (template) ja näkymistä (view). Django valittiin projektiin, koska se on toteutettu Pythonilla, joka oli ennestään kehittäjille tuttu kieli. Lisäksi Djangoon on olemassa REST-lisäosa, jonka

avulla REST-rajapinnan toteuttaminen on helppoa. Djangon MTV-malli eroaa MVC-mallista siinä mielessä, että sapluunat piirtävät käyttöliittymän ja näkymät hakevat sapluunoille malleista tarvittavan datan. Sapluunassa otetaan vastaan käyttäjän syötteitä ja näytetään käyttöliittymä, mutta toisaalta näkymien URL-osoitteet tai kyselyparametrit voivat muokata sitä, mitä sapluunassa näytetään tai mitä näkymä tekee. Pilvipalvelun käyttöliittymästä kyseinen arkkitehtuuri on hyvinkin helposti huomattavissa, mutta itse sovelluksen rajapinnassa arkkitehtuuri on hieman haastavampi huomata, koska rajapinnassa ei ole sapluunoita. [18]

Pilvipalvelun rajapintaa käyttää niin sanottua REST (Representational State Transfer) --arkkitehtuuria. REST-arkkitehtuuri koostuu seuraavista osista. Ensimmäinen osa on strukturoitu datamuoto, joka on tapa jolla haettavat resurssit näytetään. Seuraava osa on HTTP, joka kertoo REST-pyynnölle, mitä halutaan tehtäväksi. Kolmas osio on URI, yleensä URL, joka kertoo, mistä haettavat resurssit löytyvät. Viimeinen osio on MIME-tyyppi, joka kertoo, missä muodossa data on lähettetty. REST:ssä puhutaan monesti CRUD-operaatioista, joka tulee sanoista create, retrieve, update ja delete. HTTP:n vastaavat operaatiot ovat OPTIONS, POST, GET, PATCH, PUT ja DELETE. OPTIONS-operaaatio kertoo, mitä rajapinalla voi tehdä. POST-operaatio yleensä luo uuden resurssin, eli rivin, tietokantatauluun. GET-operaatio hakee tietokannasta yhden tai useamman resurssin, mutta ei muokkaa resursseja. PATCH ja PUT -operaatiot muokkaavat resursseja. DELETE-operaatio poistaa resurssin. Olennaista REST-arkkitehtuurissa on myös URI-osa, koska siinä määritellään, mistä resurssit haetaan, lisäksi URIssa voi pyyntöparametreilla myös rajoittaa tuloksia. [19]

3.3 Käytetyt tekniikat

Käytetyt tekniikat valikoituivat myös jokseenkin tavanomaisilla perusteilla. Erityisiä poikkeuksia ei juurikaan tehty, paitsi Django-käyttöliittymän tekeminen ilman varsinaista tietokantaa. Toisaalta käyttöliittymän toteutustavaksi olisi myös voinut valita Reactin.

Rajapinnan kanssa keskustellaan JSON-muotoisilla viesteillä. Rajapinnan taustalla toimii SQL-tietokanta, johon on tallennettu laitteiden mittaustiedot ja laitteiden yleiset tunnistetiedot. Sekä rajapinta että itse sovellus on kirjoitettu Djangolla. Web-sovelluksen näkymiin on käytetty Javascriptiä, jotta ollaan saatu tehtyä sopivia rajapintapyyntöjä.

Javascript on käytännössä tärkein teknologia, koska se mahdollistaa web-sovelluksen toiminnan ja lataa rajapinnalta tulevan datan käyttöliittymäksi selaimelle. Javascript mahdollistaa komentojen tekemisen web-sivulla, joten Javascript mahdollistaa web-sivuilta toisille web-sivuille. Sovelluksessa on käytössä graafikirjasto, joka on kirjoitettu Javascriptilla. Monia Javascript-kirjastoja on myös käytetty sovelluksessa, näkyvimpänä näistä on Bootstrap, jolla on tehty koko käyttöliittymän asettelu. Suurin osa Javascriptistä ohjelmassa on nimenomaan valmiita kirjastoja ja itsekirjoitetut osuudet on lähinnä tehty kiinnittämään kirjastototeutukset oikeisiin HTML-komponentteihin. [20]

JSON, eli Javascript Object Notation on Javascriptista peräisin oleva datan tallennusmuoto, jota käytetään paljon web-sovelluksissa. Tähän syynä on se, että suurin osa web-sovelluksista on kirjoitettu Javascriptillä ja Javascript luonnollisesti tukee JSON:ia suoraan. Muoto, jossa data

JSON:ssa on avain–arvo-tyyppistä, JSON:ssa on myös ominaisuus, että on täysin mahdollista laittaa arvoksi uusi objekti. Kun avaimen arvoksi laittaa objektin, saa luotua objektin, joka voi muistuttaa esimerkiksi tietokantarelaatioita. JSON:n hyviä puolia on myös, että sitä on suhteellisen helppo myös ihmisen kirjoittaa ja lisäksi muilla ohjelmointikielillä on suhteellisen helppo jäsentää JSON-objekti kielen omiksi objekteiksi. [21]

Mittaustuloksia varten käytettiin WSN OpenAPI -tyyppisiä viestejä. Kyseessä on formaatti, jolla on pyritty luomaan mahdollisimman monikäyttöinen tapa vastaanottaa ja lähettää anturidataa. Alunperin formaatti on luotu käyttäen XML:ää, mutta formaatin pystyi myös suhteellisen helposti muuttamaan JSON:ksi. Formaatin yksi suurimpia etuja on, että aina saadaan selville, mistä verkosta ja mikä laite on kyseessä, sekä mistä mittaus on tullut rajapinnalle. [22]

Postgre SQL -tietokanta on valittu käytettäväksi, koska se oon avointa lähdekoodia, lisäksi se on monimuotoinen ja tehokas. Monin paikoin Postgre SQL-tietokanta on hyvin yksinkertainen ja sillä saa tehtyä todella paljon asioita tehokkaasti. Kuitenkin tässä sovelluksesta tietokantaa käytetään käytännössä pelkästään Django ORM:n kautta, joten tehokkuus ei ollut pääsyy valintaan. Periaatteessa tietokanta olisi voinut olla myös joku muu SQL-tietokanta, mutta muut Djangon tukemat vaihtoehdot eivät olleet yhtä laajoja kuin PostgreSQL. Lisäksi Postgressa oli helpottavia asioita, kuten HStoreFieldit, joihin pystyttiin tallentamaan avain–arvopareja. Tällä tavalla pystyttiin lisäämään lisätietoja esimerkiksi lukijainstanssiin. [23]

Käytetyt tekniikat osoittautuivat lopulta oikein hyviksi valinnoiksi. Jälkikäteen voisi pohtia, että olisiko voinut olla tarpeellista käyttää jotain NoSQL-tietokantaa useasti haettavan datan tallentamiseen, jolloin palvelun suorituskykyä olisi voinut parantaa.

4 PALVELUN TOTEUTUS

Tässä kappaleessa perehdytään yksityiskohtaisesti palvelun toteutukseen ja arvioidaan luvussa 3 tehtyjen teknologiavalintojen onnistumista.

Palvelun toteutukseen liittyvät valinnat ovat ohjelmistoarkkitehtuurin ohella keskeisiä päätöksiä, joita ohjelmistoprojektissa tehdään. Nämä valinnat vaikuttavat merkittävästi siihen, miten eri ratkaisut konkreettisesti toteutetaan, ja riittävän monta valintaa tehtyään voi törmätä myös rajoituksiin, joita valituilla ratkaisuilla voi olla.

Palvelu on toteutettu AWS-ympäristössä, ja käyttöjärjestelmänä on käytetty Ubuntu Linuxia. Ubuntu Linux ei ole välttämätön vaatimus, palvelun voi halutessaan asentaa myös muille käyttöjärjestelmille. Käytännössä kuitenkin Linuxin käyttö alustana on helpottanut asennusta huomattavasti erityisesti NoSQL-tietokannan käytön vuoksi, jota hyödynnetään rajapinnassa laitehallinnan ulkopuolisissa sovelluksissa. Kehitysympäristössä on myös käytetty samaa Ubuntu Linuxia Vagrant box -virtuaalikoneessa. Kuten aiemmin mainittiin, rajapinnan ja sovelluksen ei tarvitse olla samalla palvelimella, joten sovelluksen voi hyvin asentaa myös Windows-palvelimelle.

Yritys, johon palvelu tehtiin tuottaa IoT-pilviratkaisuja SaaS-tyyppisesti erilaisille asiakkaille. Sovelluskohteita ovat esimerkiksi kiinteistöautomaatiot, varastonhallinta ja ilmanlaatu. Yrityksen palvelut perustuvat Djangolla rakennettuun tietokantaan, johon tallennetaan rajapinnan kautta IoT-antureilta saadut mittaustulokset. Rajapintaa käytetään käyttöliittymillä, jotka on tehty myös Djangolla.

Koska lähtökohtaisesti rajapinta pysyy samana kaikissa erilaisissa IoT-projekteissa, niin projekteja ei tarvitse aloittaa miettimällä, miten data tallennetaan tietokantaan. Tällöin jää ratkottavaksi, miten data halutaan näyttää ja onko sovelluksessa olemassa joitain erityisiä määreitä, joita ei ole muuten huomioitu. Esimerkiksi, jos käsitellään RFID-tunnisteita, niin silloin ei pääde aivan samanlaiset määreet kuin, mitä tulisi vastaan mittauksista, jossa mittaustulos on lähtökohtaisesti numero.

Näin ollen yrityksen liiketoiminnaksi jääkin sovelluksen muokkaamminen käyttötapaukseen ja asiakkaalle sopivaksi. Hyötynä tässä on, että tietokantaa ei tarvitse joka kerta rakentaa alusta ja lisäksi analysointityökaluja voidaaan käyttää ristiin usean asiakkuuden suhteen. Parhaimillaan pystytään jopa kehittämään useampaan asiakkuuteen analysointikirjastoja samanaikaisesti. Kolmas hyöty on, että yhtä pääsovellusta on helpompi ylläpitää kuin montaa erillistä sovellusta. Tällöin voidaan olla luottavaisia sen suhteen, että uusimmat tietoturvapäivitykset on aina asennettu.

Luvussa käsitellään ensin valittuja ratkaisuja. Aluksi tarkastellaan tietokannan ja rajapinnan rakennetta. Lähde-instanssi on eriytetty omaksi osiokseen sen monimutkaisuuden ja mahdollisten ongelmien vuoksi. Lisäksi käsitellään mittaustuloskonfiguraatioita, joiden avulla normalisoidaan lähde instanssin tuottama data. Näiden käsitteiden jälkeen tarkastellaan, miten Django-sovellukset on rakennettu. Viimeisenä asiana käydään läpi valittujen ratkaisujen pohjalta muodostuneita tietokannan relaatioita.

Seuraava aliluku keskittyy sovelluksen toiminnalliseen kuvaamiseen käyttöliittymäkuvien avulla. Aluksi käsitellään laitteiden lisäämistä ja listauksen toteuttamista. Tämän jälkeen esitellään, miten laitteiden tiedot ja mittaustulokset näytetään käyttöliittymässä. Viimeisenä asiana käsitellään päivitysten hallintaa ja julkaisun luontia.

Aliluvun jälkeen tarkastellaan sovelluksen vaatimuksia laitteelle. Käydään läpi, millaisia vaatimuksia sovelluksella on laitteen kannalta, sekä mitä erilaisia laitteita järjestelmään on liitetty. Kun kaikki palaset on käyty läpi, tarkastellaan vielä sovelluksen testaamista ja huomioon otettavia asioita.

Viimeinen aliluku keskittyy sovelluksen käyttöönottoon. Aluksi käsitellään käyttöönoton haasteita, ja sen jälkeen siirrytään käsittelemään sovelluksen asentamista ja käyttöönottoa. Viimeisenä asiana käydään läpi, miten laite liitetään sovellukseen onnistuneesti.

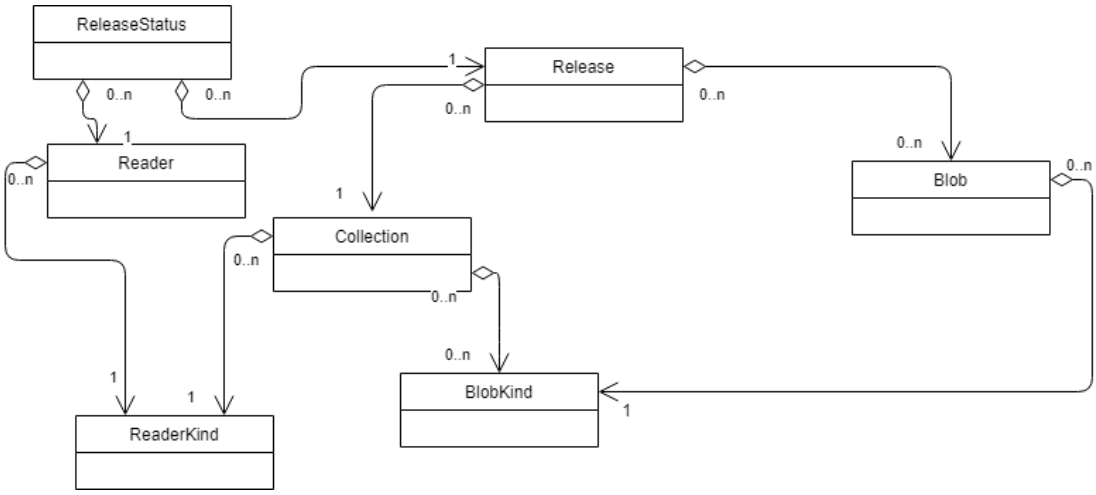
4.1 Valitut ratkaisut

Valitut ratkaisut voidaan jakaa kolmeen osaan, tietokantaan, rajapintaan ja käyttöliittymään. Nämä osat muodostavat yhdessä sovelluksen koko toiminnan. Näiden lisäksi valituksi ratkaisuksi voidaan myös nähdä pavelun alkuperäinen tarkoitus olla yleiskäyttöinen.

Tietokanta sisältää taulut jokaiselle ohjelman peruskäsitteelle ja peruskäsitteiden tietoja hallitaan tietokannassa. Suunnitteluperiaatteena oli, että tietokanta pysyisi mahdollisimman yksinkertaisena ja taulujen määrä olisi mahdollisimman pieni. Tietokannassa kuitenkin jouduttiin tekemään tauluja, joilla yhdistettiin tietokantatauluja ja luotiin näiden liitoksille lisätietoa. Syynä tähän oli se, että rajapinnan kannalta oli helpompaa luoda uusi taulu kuin kasvattaa taulujen kokoa.

Rajapinnassa käytännössä jokaiselle tietokantataululle luotiin uusi rajapinta. Rajapintoja ei suunniteltu käyttöliittymä edellä, vaan ideana oli että käyttöliittymän kuuluu hakea tarvittavista rajapinnoista data ja kerätä se yhteen. Rajapinnoilla tallennetaan tietokantaan vain yhtä asiaa kerrallaan, jos tarvitsee tehdä useampia operaatioita, niin starvitaan useampia rajapintapyyntöjä.

Rajapinnan suunnittelun pohjana oli ajatus siitä, että rajapintaa voisi käyttää yleiskäyttöisesti muihinkin tapauksiin, joissa tarvitsee hallita erilaisia laitteita. Tämän johdosta rajapinnassa on suhteellisen vähän IoT-reittittimeen liittyviä erityispiirteitä. Lopulta kuitenkin laitteen päivitykseen liittyvissä rajapinnoissa jouduttiin lisäämään ominaisuus, jossa varmistettiin että laite on saanut päivityksen asennettua. Varsinaisesti tämäkään ominaisuus ei estä muunlaisten laitteiden käyttöä, mutta alunperin tarkoituksena oli, että laite tietäisi itse, mikä versio sillä on ja



Kuva 4.1. *UML-kuvaus tietokannasta*

päättelisi itse, tarvitseeko se päivitystä.

Käyttöliittymien toiminta perustuu siihen, että rajapinnasta haetaan dataa ja käsitellään se haluttuun muotoon käyttöliittymässä. Rajapinnoissa on ominaisuus, että käyttäjälle suodatetaan vain käyttäjälle itselleen kuuluvat datat. Tämä mahdollistaa sen, että asiakkaille voidaan myös tarjota hallintapalvelua, jolloin asiakkaat näkevät vain omat laitteensa. Suurin osa käyttöliittymistä siirtää rajapinnasta saadut datat samassa muodossa käyttöliittymään. Poikkeuksena on kuvaajat, koska kuvaajissa tehdään analyysiä, esimerkiksi kuvaajien akselit sovitetaan automaattisesti mittauspisteiden mukaan.

4.2 Toteutus

Rajapinta on jaettu neljään osaan, Ensimmäinen osuus on asiakaskonfiguraatio, jonka avulla luodaan asiakas instanssi ja niihin liittyvät käyttäjät. Asiakaskonfiguraatiossa tehdään myös laite instanssi. Toinen osuus on päivityshallinta, jossa voidaan määritellä laitteiden päivityksiin liittyviä asioita, kuten tiedostot ja tiedostojen suhteet toisiinsa. Kolmas osuus on mittaustulokset, jonka avulla laite lähettää mittaustuloksia ja sen avulla saadaan haettua laitteen mittaustulokset käyttöliittymään. Neljäs osuus on laitehallinta, jonka avulla laitteet yhdistetään asiakaskonfiguraation laiteinstanssiin ja yhdistetään palveluun.

Mittaustulosten yhteyteen on tehty lähde -instanssi. Lähde-instanssi voi olla kokonainen laite tai pelkästään osa laitteesta. Tämän johdosta instanssin olisi tarkoitus olla puumainen rakenne, joka mahdollistaisi, että yhdellä päälaitteella voi olla monta alilaitetta ja yhdellä alilaitteella voi olla esimerkiksi monta antennia. Tämä mahdollistaa sen, että voidaan tarkastella kokonaista laitetta tai pelkästään spesifiä osaa laitteesta, esimerkiksi WLAN-antennien toimintaa. Periaatteessa kyseinen ominaisuus mahdollistaisi myös datan kokoamisen yhteen paikkaa monesta laitteesta, mutta tälle ei varsinaisesti ole nähty tarvetta. Ongelmana tässä olisi, että lähettäessä data pitäisi tarkemmin annotoida, onko kyse laitteesta, alilaitteesta vai antennista, jotta tiedetään, mistä laitteesta data on peräisin. Ilman annotointia voisi esimerkiksi käydä niin, että muistinkulutukset menisivät kahdelta laitteelta sekaisin, jos ne on ilmoitettu samanlaiseksi

dataksi.

Merkittävä osuus mittaustulososuutta on myös mittaustuloskonfiguraatio. Mittaustuloskonfiguraatioon on määritelty, millaisilla suureilla tuetaan palveluun tulevaa dataa ja samalla suureisiin on määritelty kertoimet, joilla datan voi normalisoida. Tällä tavalla saadaan tarvittaessa varmistettua, että suureet ovat oikein ja data on helposti luettavaa. Myös datan analysoinnin kannalta on hyvin olennaista, että pystytään varmistumaan siitä, että suureet ovat oikein.

Sovelluksessa on myös käytetty suunnittelupäätöstä, että varsinaisella laitehallinnalla ei ole omaa tietokantaa, vaan tietokantana käytetään suoraan rajapinnan tietokantaa. Ohjelman tietokanta sisältää lähinnä vain tiedot käyttäjistä ja käyttäjiin liitetyistä oikeuksista. Käyttäjät pitää kuitenkin olla yhdistettynä rajapintapalvelimeen joko tokenilla tai käyttäjätunnisteiden pitää olla samat. Tässä on omat hyvät ja huonot puolensa. Hyvänä puolena on, että sovelluksen voi asentaa käytännössä mihin tahansa palvelimelle ja sovellus voi ottaa yhteyttä rajapintaan oli rajapinta käytössä missä tahansa. Samalla saa varmennettua, että käyttäjällä on oikeudet palvelimella oleviin resursseihin. Huonona puolena on, että rajapintapyynnöt eivät ole järin nopeita ja tietokannan muokkaaminen sovelluksessa on hankalampaa rajapinnan kautta. Toisaalta sunnittelupäätös on suhteellisen normaali moderni web-sovellus, mutta toisaalta myös epätyypillinen, jos käytössä on Django käyttöliittymässä.

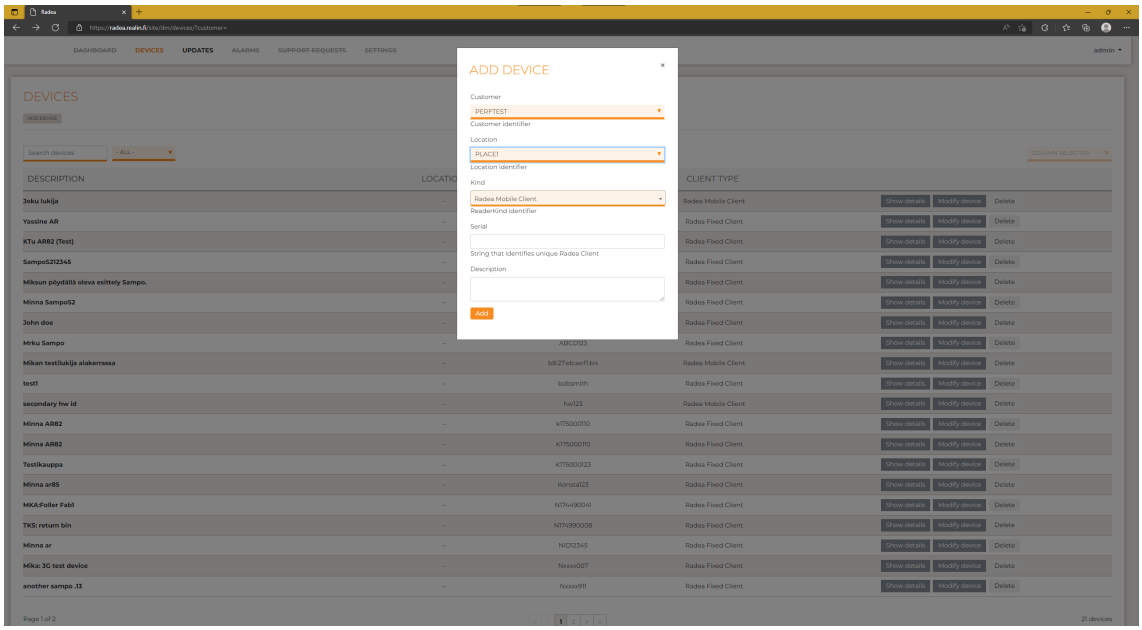
Kuvaajassa 4.1 on kuvattu tietokannan relaatiot ja millaisista tauluista tietokanta koostuu. Blob kuvastaa binääritiedostoa, joka sisältää laiteohjelmiston, jokaisella binääritiedostolla on joku tyyppi eli BlobKind. Eli Blobissa on vierasavain tauluun BlobKind. Collection kuvastaa kokoelmaa erilaisia binääritiedostotyyppejä, jotka kohdistuvat yhteen lukijatyyppiin, eli ReaderKindiin. Tietokannan tapauksessa tämä tarkoittaa, että Collectionin ja BlobKindin välillä on monesta moneen relaation, samalla Collectionissa on vierasavain ReaderKindiin. Release kuvastaa julkaisua, joka kohdistuu yhteen kokoelmaan ja koostuu binääritiedostoista. Releasetaulussa on siis vierasavain Collection-tauluun, lisäksi siinä on monesta moneen relaatio Blobtauluun. ReleaseStatus on taulu, joka kertoo, mikä on julkaisun tila tietylle laitteelle. Tämä taulu siis on laitteen ja julkaisun monesta moneen relaation lisäinformaatiotaulu.

Laitehallinnan toteutus on lopulta melko perinteinen web-sovellus. Mielenkiintoisimpina ominaisuuksina siinä on päivitysten synkronointi, jota käsitellään tarkemmin aliluvussa 4.3.

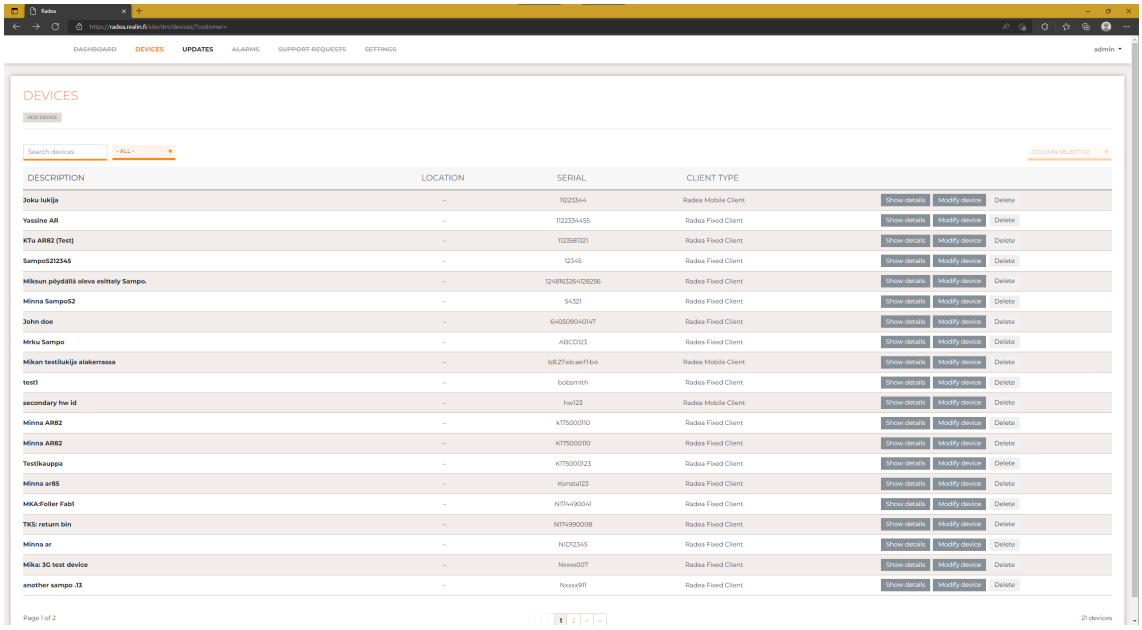
4.3 Toiminnallinen kuvaus

Tässä aliluvussa kuvataan, miten laitteet ja ihmiset käyttävät sovellusta.

Alkuoletuksena asiakas haluaa lisätä laitteen palveluun. Tämä alkaa siten, että asiakas lisää käyttöliittymän kautta laitteelle nimen ja vapaasti valittavan yksilöivän tunnisteen, joka on esitelty kuvassa 4.2. Kyseisessä kuvassa näkyy modaali, jonka alla näkyy laitelistaus. Laitteelle voi myös lisätä paikan, tyypin ja kenen laite on kyseessä. Nämä näkyvät myös kuvassa 4.2, mutta ne eivät ole pakollisia kenttiä. Olennaisin kenttä näistä on tunniste, koska tunnisteen pitää olla sama, jolla laite keskustelee rajapinnan kanssa. Tunnisteet ovat itsessään selkokielisiä, mutta rajapinnat vaativat token-kirjautumisen. Jos rajapinta olisi avoin, se tarkoittaisi, että tunnis-



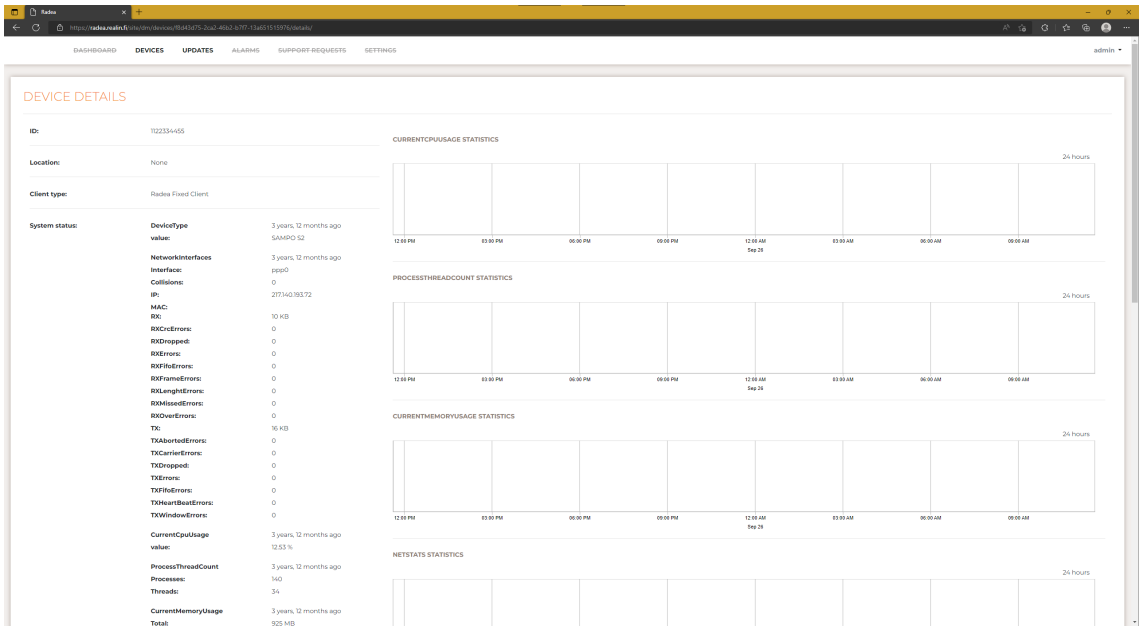
Kuva 4.2. *Kuvankaappaus laitteenlisäyksestä*



Kuva 4.3. *Kuvankaappaus laitelistauksesta*

teen tietäessään mikä tahansa laite voisi esittää olevansa kyseinen laite, mutta kirjautuminen estää tämän.

Kun laite on luotu, se tulee näkymään laitelistaukseen, joka on esitelty kuvassa 4.3. Oletuksena laitelistausnäkymässä näkyvät perustiedot, jotka on syötetty laitteen lisäyksessä. Nämä perustiedot ovat laitteen kuvaus, sijainti, tunniste ja tyyppi. Viimeisessä sarakkeessa on painikkeet, joista pystyy tarkastelemaan yksittäisen laitteen tietoja, muokkaamaan laitteen tietoja tai poistamaan laitteen. Laitteen poiston painike on eri värillä, jotta sitä ei painaisi vahingossa. Laitelistauksen lisäksi listaan on tehty ominaisuus, että taulukon sarakkeita voi muokata vapaasti ja halutessaan pääsee katsomaan mittastulosten viimeisimpiä arvoja. Näkymään tehtiin



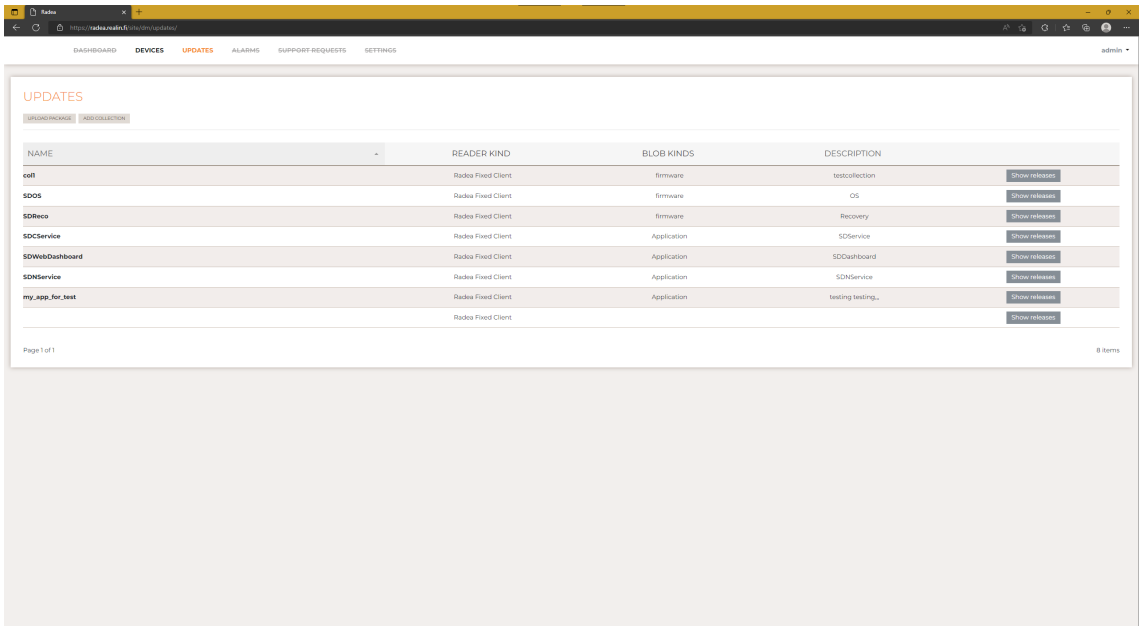
Kuva 4.4. *Kuvankaappaus laitteen tiedot -sivusta*

myös ominaisuus, joka kertoi laitteen tilan liikennevaloilla. Todellisuudessa liikennevalot kertoivat lähinnä suorittimen sen hetkisestä kapasiteetista. Koska näkymässä on mahdollisuus näyttä useiden laitteiden viimeisimpiä mittaustuloksia, tehtiin taustalle järjestelmä, joka tallentaa viimeisimmät arvot, jotta näkymä olisi nopeampi ladata.

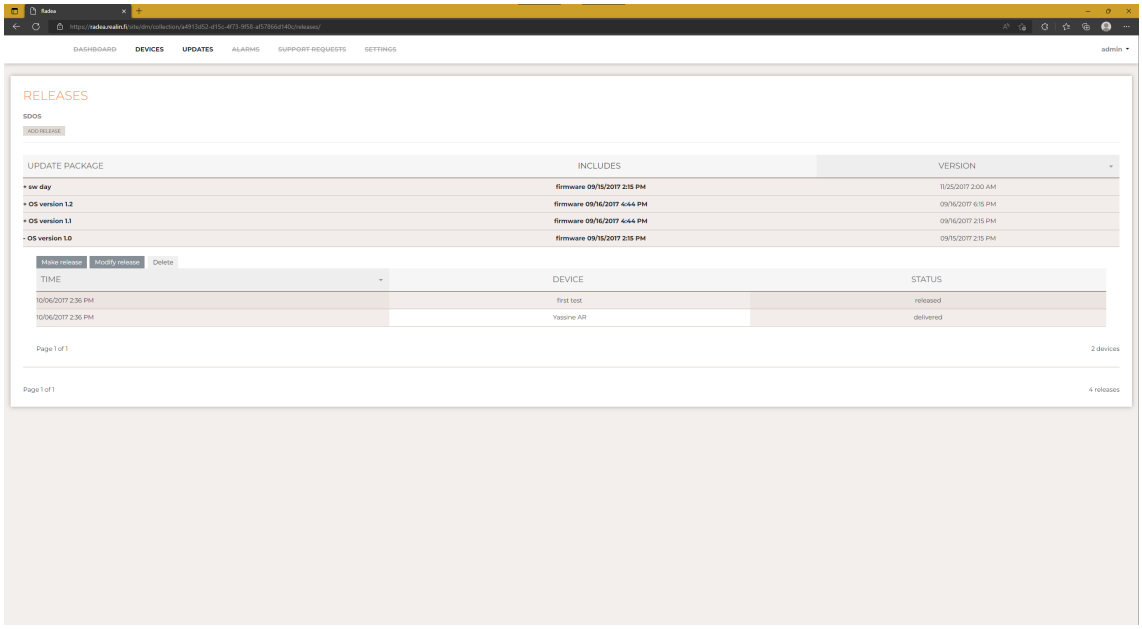
Kun tunnisteet ovat kunnossa sekä laitteella että ohjelmassa, voi laite kysyä tunnisteen avulla osoitetta, johon mittaustulokset kuuluisi lähettää. Tämän jälkeen laite alkaa lähettää mittaustuloksia osoitteeseen ja asiakas pääsee katsomaan laitteen kunnon käyttöliittymän kautta, joka on esitelty kuvassa 4.4. Sivu näyttää samat tiedot mitä laitelistaus-sivukin, mutta tämän lisäksi on mahdollista myös näyttää historiadataa kuvaajana. Kuvan vasemmassa laidassa on laitteen tämän hetkiset tiedot ja kuvan oikeassa laidassa on kuvaajat. Sekä kuvaajat että laitteen tiedot ryhmittäytyvät sarakkeiksi, jolloin tiedot ja kuvaajat ovat päällekäin. Kuvaaja-komponentti on kirjoitettu Javascriptilla ja siihen on tehty aggregointiominaisuus, jolla saa muokattua kuvaajaa vähemmän tietokoneen resursseja vieväksi.

Kun laite on liitetty palveluun, sille voidaan lisätä palveluun sovelluspäivityspaketteja, joiden lisääminen on esitetty kuvassa 4.5. Kuvan yläreunassa näkyy painikkeet, joilla voidaan luoda uusi paketti ja voidaan lisätä tiedostoja. Tämän alla on taulukko, jossa on listattuna paketit, paketin nimi, laitteen tyyppi, tiedostotyypit, kuvaus ja viimeisenä on painike, josta painamalla näkee pakettien julkaisut. Kun luodaan tiedostoja, pitää liittää mukaan, mikä on tiedoston tyyppi ja mikä on tiedoston versio. Versiot erotellaan toisistaan aikaleimoilla, jotta tiedetään onko tiedosto uusi. Kun tiedostot on ladattu palvelimelle, nämä tiedostot voi paketoida, jolloin valitaan tiedostotyypit, jotka muodostavat paketin. Tiedostopaketti kohdistetaan tietylle laitetyypille ja laitteen omistajalle.

Tiedostopaketeista voidaan luoda tarvittaessa julkaisuja. Julkaisuja voidaan hallinta julkaisusivulta, joka on esitelty kuvassa 4.6. Kuvassa näkyy ensimmäisenä paketin nimi. Paketin nimen



Kuva 4.5. *Kuvankaappaus päivityshallinta -sivusta*



Kuva 4.6. *Kuvankaappaus julkaisujenhallinta -sivusta*

alla on painike, josta voi luoda julkaisun. Tämän jälkeen alkaa taulukko, jossa yhdellä rivillä on plus-merkki, julkaisun nimi, sisälletyt paketit ja versionumero. Jos painaa plus-merkkiä aukeaa sisäikkäinen valikko. Valikon alussa on napit julkaisulle, julkaisun muokkaukselle ja julkaisun poistolle. Valikon alla näkyvät laitteet, joille julkaisu on viety. Julkaisu kohdistetaan aina yhteen tiedostopakettiin ja julkaisuun valitaan julkaisun tarvitsemat tiedostot. Kun julkaisu on luotu, niin tämän jälkeen sen voi julkaista. Tämä tarkoittaa sitä, että julkaisun tila muutetaan tilaan released, jolloin laitteet voivat hakea palvelimelta tiedostot ja päivittää itsensä uuteen versioon. Tämän jälkeen päivityksen tilaksi muodostuu, että se on jo päivitetty, jolloin käyttäjä voi olla varmistua, että päivitys on onnistunut. Jos päivitys syystä tai toisesta epäonnistuu,

Kuva 4.7. *Kuva IoT-Gatwaysta*

niin laitteen olisi hyvä päivittää tieto siitä rajapintaan. Päivitysten soveltuvuus laitteelle on joko laitteen tai loppukäyttäjän vastuulla. Sovellus ei ota kantaa siihen, mahtaako päivitys toimia laitteella. Tietokantatasolla vastaava toiminnallisuus on esitetty kappaleessa 4.2.

Laitteelta vaaditaan ominaisuuksina sen verran loogista päättelyä, että se osaa syöttää mittaustulokset oikeaan paikkaan ja myös käsitellä mahdolliset päivityspaketit itsenäisesti.

4.4 Laitteen toiminta

Kyseessä oleva laite, joka on kuvassa 4.7, on suunniteltu keräämään dataa IoT-antureilta ja lähettämään kerätyt tiedot asiakaspalvelimelle. Laitteessa on Linux-käyttöjärjestelmä, joka mahdollistaa sen toiminnan suorittamisen nopeasti ja tehokkaasti. Laite pystyy myös vastaanottamaan mittaustuloksia palvelimelta, mikä antaa käyttäjille mahdollisuuden saada reaaliaikaista tietoa antureiden tilasta.

Laitteen sisään rakennetut ominaisuudet antavat käyttäjille mahdollisuuden antaa yksinker-

taisia käskyjä antureille. Tämä tarkoittaa sitä, että käyttäjä voi esimerkiksi pyytää laitetta aloittamaan tai lopettamaan tietyn anturin tiedonkeruun tai määrittämään tiettyjä hälytysrajapyykkejä. Käyttäjän ei tarvitse käyttää erillistä ohjelmistoa näiden käskyjen antamiseen, vaan ne voidaan antaa suoraan laitteelle.

Laitteen ohjelmistopäivitykset ovat tärkeä osa sen toimintaa, sillä ne mahdollistavat uusien ominaisuuksien lisäämisen ja virheiden korjaamisen. Laiteohjelmiston päivittäminen onnistuu lataamalla uusi ohjelmisto kokonaisuudessaan laitteelle, jolloin vanha ohjelmisto korvataan uudella. On kuitenkin hyvä huomata, että laitepäivityksen aikana tulevat mittaustulokset häviävät, sillä laite joutuu käynnistymään uudelleen. Tämä on kuitenkin otettu huomioon laitteen suunnittelussa, ja sen pitäisi olla toimintavalmis heti käynnistyksen jälkeen.

Laitteen hallinta on mahdollista myös etänä, sillä laite voi lähettää tietoa itsestään asiakaspalvelimelle ja kyselee palvelimelta aika ajoin mahdollisista ohjelmistopäivityksistä. Näin laitteen ylläpito on helppoa ja nopeaa. Laite ilmoittaa myös itsestään rajapintaan, kun se on takaisin käytössä, joten käyttäjän ei tarvitse manuaalisesti tarkkailla sen toimintaa. Kaiken kaikkiaan tämä esimerkkilaite on suunniteltu erittäin tehokkaaksi ja käyttäjäystävälliseksi IoT-laitteeksi, joka täyttää erilaisten käyttäjien tarpeet.

4.4.1 Laitevaatimukset

Tässä aliluvussa käsitellään millaisia vaatimuksia laitehallintaa käyttävällä laitteella on pilvipalvelun käytössä. Pilvipalvelut mahdollistavat tiedon tallentamisen ja jakamisen verkon yli.

Pilvipalvelun käyttö vaatii laitteelta pääsyn Internetiin, joten yksi laitevaatimus on, että laitteesta on oltava internet-yhteys. Tämän lisäksi laitteen on pystyttävä lähettämään ja vastaanottamaan rajapintapyyntöjä pilvipalvelun kanssa kommunikoimiseksi. Tämä edellyttää laitteen ohjelmointia, jotta se kykenee käsittelemään tulevia pyyntöjä ja lähettämään vastauksia pilvipalveluun.

Tallennustilan tarve on myös tärkeä laitevaatimus laitehallinnan käytössä. Laitteella on oltava tarpeeksi tallennustilaa, jotta se pystyy tallentamaan laitehallinnasta vastaanotetut tiedot. Erityisesti tärkeää on, että tallennustilaa on vähintään kaksinkertaisesti verrattuna laitteen käyttöjärjestelmän tai ohjelmiston kokoon, jotta laite voidaan päivittää uudempaan versioon.

On huomattava, että pilvipalvelua voidaan käyttää käytännössä millä tahansa tietokonealustalla, joka on julkaistu 2010-luvulla tai sen jälkeen. Kuitenkin suurin haaste on, että pilvipalvelusta vastaanotettavat tiedot voivat olla suurikokoisia, joten laitteiden on kyettävä käsittelemään suuria määriä dataa tehokkaasti. Tätä on pyritty ratkaisemaan REST-pyyntöjen sivutuksella, jossa vastaukset palvelimelta jaetaan pienempiin osiin. Tämä auttaa pienentämään yksittäisten vastausten kokoa, mutta kokonaispyyntöjen lukeminen saattaa vaatia useamman osan lataamista.

Yhteenvetona voidaan todeta, että laitevaatimukset pilvipalvelun käytössä ovat melko yksinkertaiset, mutta laitteen on oltava riittävän tehokas käsittelemään suuria määriä dataa.

4.4.2 Laitetyypit

Tässä aliluvussa käsitellään laitehallinnan kautta hallittavia laitteita. Laitehallinta mahdollistaa laitteiden tehokkaan hallinnan etäältä, mikä on erittäin hyödyllistä monimutkaisten järjestelmien ylläpidossa. Laitehallinnan avulla voidaan muun muassa suorittaa ohjelmistopäivityksiä, seurata laitteiden tilaa ja kerätä lokitietoa.

Yksi laitetyyppi, jota voidaan hallita laitehallinnan kautta, on Raspberry Pi. Raspberry Pi on suosittu pieni tietokone, joka on ohjelmoitavissa monella eri tavalla. Laitehallinnan avulla Raspberry Pi -laitteelle voidaan ladata uusia ohjelmistoja, ja sen tilaa voidaan seurata esimerkiksi lokitiedon avulla. Raspberry Pi -laitteiden lisäksi laitehallinta mahdollistaa myös muiden laitteiden, kuten RFID-lukijoiden, hallinnan.

Laitehallintaa voidaan hyödyntää myös erilaisten mobiililaitteiden, kuten Windows- ja Android-pohjaisten laitteiden, hallinnassa. Tämä on erityisen hyödyllistä, jos organisaation käytössä on suuri määrä erilaisia mobiililaitteita. Laitehallinnan avulla voidaan esimerkiksi suorittaa päivityksiä kaikille laitteille samanaikaisesti, mikä säästää aikaa ja vaivaa. Lisäksi laitehallinnan avulla voidaan kerätä tietoa mobiililaitteiden käytöstä ja tilasta.

4.5 Päivitysten hallinta

Yhtenä työn tutkimuskysymyksenä oli, mitä pitää huomioida tarjottaessa päivityspaketteja. Tämän vuoksi tässä uudesta näkökulmasta samoja asioita, jotka tulivat aliluvussa 4.3 ilmi kehitysprosessia käsitellessä.

Ensimmäinen asia, joka kehityksen aikana tuli vastaan, oli että päivityspakettien tiedoistoille täytyy kehittää hakemistopuu, joilla voidaan jäljittää päivityspaketin sijainti palvelimella. Varsinaisesti tästä ei ole hyötyä laitteille, mutta ohjelmistokehittäjän näkökulmasta on helpompaa, kun tietää mihin tietokantataulujen riveihin tiedostot liittyvät.

Päivitysten tiedostoille piti myös antaa tyyppi, koska binääritiedostoista olisi vaikea päätellä, mitä tiedoston kuuluisi tehdä. Tiedoston tyyppi voi lähtökohtaisesti olla mitä tahansa, mutta sen pitää olla yksilöivä, koska sitä tietoa käytetään päivitysten paketoinnissa. Tiedoston tyyppi liitettynä aikaleimaan luo tiedoston version.

Seuraava ongelma liittyi päivityspaketteihin. Alunperin ajateltiin, että päivityspaketit olisivat valmiiksi paketoituja. Myöhemmin selvisi, että pakettien purku ei välttämättä olisi mahdollista laitteella. Lopputuloksena piti rakentaa tietokantamalli, joka paketoi yksittäiset tiedostot tietokannassa "paketiksi". Tämän lisäksi piti myös luoda laitteen tyyppi taulu, koska oli tarve liittää tietyt paketit tietyille laitteille.

Kun paketointi oli tehty, selvisi lopulta, että on mahdollista, että useassa julkaisussa on samoja paketteja. Tämä tarkoitti sitä, että paketoitu kooste tiedostoista ei voisi olla suoraan julkaisu tai vähintään se olisi tehotonta, koska samoja tiedostoja jouduttaisiin lataamaan pavelimelle useita kertoja. Tämän johdosta luotiin julkaisumalli tietokantaan. Julkaisu koostuu useista paketeista, lisäksi julkaisuun kirjattiin luontiaika, jotta olisi mahdollista selvittää pakettien versiohistoria.

Varsinainen päivityspaketti oli saatu näin luotua. Viimein piti selvittää, miten saadaan päivitys laitteelle luotettavasti. Tämän takia luotiin tietokantataulu julkaisun tilalle, ja se liitettiin julkaisu- ja laitetaulujen välille. Tämän taulun yksinkertainen idea oli vain kertoa, oliko yksittäiselle laitteelle toimitettu paketit ja laitteen kuuluisi viestiä takaisin, jos päivitys on onnistunut. Tällä tavalla saadan tietoon, mitkä julkaisujen versiot on asennettuna millekin laitteelle.

4.6 Testaus

Tässä aliluvussa käsitellään, miten laitehallintaa testattiin. Pythonille on useita erilaisia tapoja kirjoittaa testejä. Oikean tavan valintaan liittyy paljon se, mitä sovellus tekee.

Kyseessä on monia metodeja käyttävä järjestelmä, joten järjestelmä kuului testata kattavasti. Kattavuutta tarkasteltiin ekvivalenssiluokkien avulla. Ekvivalenssiluokat määriteltiin ohjelmakoodissa olevien if- ja try-lohkojen perusteella. Tarkoituksena oli, että testeillä käydään vähintään jokainen lohko läpi. Ensinnäkin kaikki metodit testattiin erikseen, jotta ne toimivat kuten ne oli suunniteltu. Rajapintatestauksessa käytettiin erityistä huomiota oikeuksiin ja tarkasteltiin erityisesti objektien oikeuksia, jotta tietyn asiakkaan laitteet näkyisivät vain asiakkaalla. Käyttöliittymän testaaminen tehtiin pääasiassa käsin, käyttöliitymän automatisoidut testit testasivat lähinnä oikeuksia ja lisäksi testattiin, että käyttöliittymä saa sisäänsä oikeanlaista dataa. Testauksessa vähäiselle huomiolle jäi tietokannan suorituskyky, jota ei etukäteen nähty varsinaisesti ongelmaksi.

Sovellus testattiin käyttötarkoitukseen nähden kattavasti. Kuitenkin jälkikäteen ajateltuna varsinaiset testit olisi voinut kirjoittaa Pytestillä Django test runnerin sijaan. Pytestillä testien kirjoittaminen olisi ollut yksinkertaisempaa, eikä olisi vaatinut yhtä suuria alustuksia kuin Django test runner. Lisäksi Pytest olisi tukenut suoraan valmiita datamalleja, jollaista toimintoa ei django test runnerilla suoraan ole. Sovellusta olisi voitu testata esimerkiksi robot frame-workilla, jotta olisi saatu tietoon toimiiko käyttöliittymä oikein. Robot frameworkin käyttöönoton raskauden takia tätä ei tehty, käyttöliittymässä kuitenkaan ei tullut vastaan merkittäviä virheitä sinä aikana, kun sovellus oli toiminnassa testi ja stage-ympäristöissä.

4.7 Käyttöönotto

Tässä aliluvussa käsitellään, miten laitehallinnan käyttöönotto lopulta onnistui.

Käyttöönotto osoittautui laitteille hieman monimutkaiseksi. Tämä johtui siitä, että jokaiselle laitteelle piti luoda omat tunnisteet, jotta niiden mittaustulokset saatiin näkyviin. Tätä varten kehitettiin apuväline, jossa lähdeinstanssiin pystyi syöttämään dataan, jos tiesi laitteen tunnisteen. Tällöin kuitenkin lähteen hierarkia tulisi ohitetuksi, jolloin lähde oli sama asia kuin laite. Kuitenkin itse järjestelmän käyttöönotto muuten osoittautui hyvin yksinkertaiseksi prosessiksi.

Järjestelmän käyttöönotto yksinkertaisimmillaan toimii siten, että PyPi-palvelimelta haetaan uusimmat yhteensopivat versiot rajapinnasta ja käyttöliittymästä. Rajapinnan ja käyttöliittymän Python-paketit osaavat asentaa tarvittavat riippuvuudet itsekseen. Tämän jälkeen pitää alustaa tietokantamallit ja luoda pääkäyttäjä, jotta voi luoda lisää käyttäjiä ja saa ylläpito-

käyttöliittymän käyttöön.

Laitteen käyttöotto järjestelmään tehdään siten, että ensin luodaan järjestelmään laite ja laitteelle ainutlaatuinen tunniste. Kun tunniste on luotu, voidaan tunnisteen avulla kysyä, mikä on laitteen UUID järjestelmässä. Kun UUID on selvillä, voidaan sen avulla kysellä mahdollisia uusia ohjelmistopäivityksiä. Jos halutaan lähettää mittaustuloksia, pitää kysyä laitteeseen liittyvät lähde-UUID, jonka avulla lähetetään mittaustulokset.

Sovelluksen elinkaari jäi kuitenkin pieneksi, koska ilmeni ohjelmistopäivityksiin liittyvä ongelma, jonka ratkaisun jatkokehittäminen olisi jäännyt tarpeettoman työlääksi. Vaikka ohjelmistopäivitykset toimivatkin, niin rajapinnassa ei ollut ominaisuutta, joka olisi kertonut mikä on laitteella käytössä oleva versio. Tästä syystä ohjelmistopäivityksiä ei uskallettu kokeilla tuotannossa, koska pelättiin että laitteet alkavat hallitsemattomasti päivittää itseään. Tietokantamallit korjausta varten tehtiin, mutta laitteita ei päivitetty käyttämään uusia rajapintoja, joten tätä ei päästy koskaan testaamaan. Tämän jälkeen priorisoitiin enemmän IoT-laitteiden mittaustuloksia kuin laitehallintaa, josta syystä kehitys laitettiin toistaiseksi jäihin. Työlistalla olisi ollut paljon jatkokehitysajatuksia, joita ei ehditty toteuttamaan.

5 JATKOKEHITYS JA ARVIOINTI

Tässä luvussa käsitellään ja arvioidaan sovelluksessa ilmenneitä ongelmia, erityisesti mittaustietokantaan liittyviä, ja niihin mahdollisia jatkokehitysideoita. Lisäksi vastataan tutkimuskysymyksiin kertomalla millaisia ratkaisuja niihin voitaisiin kehittää. Kappaleen lopussa esitetään vielä hajanaisia ajatuksia muista kehitysideoista, joita kehityksen aikana ilmeni, mutta ei toteutettu vielä.

Aliluvussa 5.1 paneudutaan ratkaisuun, jossa lisätään välimuistin käyttöä datan lataamisessa. Tässä käsitellään, mitä hyötyjä kyseisestä ratkaisusta olisi ja miten tämä ratkaisu voitaisiin yksinkertaisesti luoda. Kappaleen lopussa myös pohditaan mahdollisia ongelmia, joita ratkaisusta syntyy.

Toinen ratkaisu, jota aliluvussa 5.2 esitellään, on sovitettujen tietokantataulujen käyttö tallentamisessa. Tässä kohdassa pohditaan esimerkiksi sitä, mitä suureita halutaan tallentaa tietokantaan. Viimeisenä analysoidaan ajatusta, että oliko tämä lopulta hyvä idea, ja kannattaako sovitettuja tietokantatauluja rakentaa.

Ratkaisujen esittelyn jälkeen vertailemme molempia ratkaisuja. Aliluvussa 5.3 pohditaan, voiko ratkaisuja käyttää yhdessä ja olisiko jompi kumpi ratkaisuista parempi. Lopuksi pohditaan vielä ratkaisujen työmääriä

Lopuksi aliluvussa 5.4 käydään myös läpi muita kehitysideoita, kuin varsinaisesti mittaustietokantaan liittyviä. Nämä kehitysideat ovat hyvin moninaisia, suurimman osan tähtäin on saada optimoitua resurssien käyttöä palvelimella. Kappaleessa pohditaan erikseen myös kuinka järkevää näitä olisi toteuttaa.

5.1 Ratkaisu: Välimuistin hyödyntäminen tietokantaoperaatioissa

Kun dataa tallennetaan jatkuvasti, syntyy datavirta. Laitehallintajärjestelmään tallennetaan datapisteitä laitteiden tarpeiden mukaan. Jos laite on kriittisessä käytössä, voi olla järkevää kerätä datapisteet joka minuutilta. Toisaalta, jos laite ei ole kovin kriittinen, niin kerran tunnissa tulevat datapisteet voivat olla riittävä määrä ratkaisun kannalta. Mitä enemmän dataa tallennetaan, sitä enemmän se luonnollisesti kuormittaa mittaustietokantaa ja tämä ongelma täytyy ratkaista tavalla tai toisella.

Johtuen mittaustietokantaan tulevista suurista datavirroista lähtökohtaisesti voisi olla järke-

vää hyödyntää mahdollisimman paljon välimuistia tai NoSQL-tietokantaa. Nämä olisi järkevä ratkaisu, koska rajapintaan tulee jatkuvasti dataa. Ne onkin todettu monesti IoT-ratkaisussa oikein toimiviksi. Syy tähän on se, että NoSQL-tietokannoissa ei ole nimensä mukaisesti datassa rakennetta. Datan rakenteettomuus lisää mahdollisuuden, että tarvittaessa voidaan vain luoda uusi sarake uudelle datatyypille. NoSQL-tietokantojen käyttöä voisi harkita laitehallinnan käyttötapauksessa, harmillisesti kuitenkin Djangon ORM tukee käytännössä vain SQL-tietokantoja, jolloin NoSQL-kantoja joutuisi ylläpitämään käsin. NoSQL:ää käytetään tässä sovelluksessa käytännössä pelkästään datan hakemisessa, jolloin sieltä tarkistetaan onko tullut uutta dataa. NoSQL-tietokantana tässä sovelluksessa käytetään Redistä. Koska Redis on konfiguroitu käyttämään sisäistä muistia, siinä ei kannata säilyttää dataa pitkiä aikoja, siitä huolimatta niiden käyttömahdollisuuksia olisi voitu pohtia enemmän. Tällöin jos tietoa ei ole tallennettu muualle, niin tieto saattaa katoaa.

Välimuistin käytön hyvänä puolena olisi se, että datan käyttö käyttöliittymässä olisi varsinkin koostenäkymissä nopeaa. Lisäksi tällä saataisiin vähennettyä kuormaa tietokantapalvelimelta, koska ei tarvitsisi kokoajan olla tallentamassa uusia mittauksia. Välimuistin käyttö myös aiheuttaisi ainakin kaksi erilaista ongelmaa, jotka pitäisi ratkaista tavalla tai toisella.

Ensimmäinen ongelma liittyy mittaushistorian tarkasteluun. Välimuistissa mittaushistoriaa ei ole tallennettuna kovinkaan pitkälle, jolloin dataa pitäisi hakea relaatiotietokannasta. Tietokantahaun jälkeen pitäisi tarkastella, onko kaikki data tallentunut. Jos eivät ole tallentuneet, silloin tarvitsee sovittaa datat tietokannasta ja välimuistista. Lisäksi pitää myös tarkastella, ettei ole mukana kahdennettua dataa, joka voisi näyttäytyä mittaushistoriassa ei toivotulla tavalla.

Tietokantataulujen loputonta kasvua ei myöskään korjata pelkästään välimuistin käyttöönotolla. Välimuistista jatkossakin lisättäisiin rivejä aina vain suurenevaan mittaustulostauluun. Vaikka PostgreSQL on hyvin suorituskykyinen tietokannan hallintajärjestelmä, silti haut jatkuvasti hidastuvat. Tilannetta voi yritää pelastaa indeksoinnilla siten, että mittaustyypistä tekee luetellun tyypin, joka luo automaattisesti PostgreSQL-tietokannassa indeksin. Mutta jossain vaiheessa tämäkään ei auta enää, koska indekseistä tulee liian suuria.

5.2 Ratkaisu: Tietokantataulujen sovittaminen dataan

Sovelluksessa käytetty WSN OpenAPI SIDF -malli on laajuutensa ja yleiskäyttöisyytensä takia mahdollisesti liian raskas tähän käyttötarkoitukseen. Kyseisen mallin käytöstä johtuen tietokantaan tuli todella suuri tietokantataulu mittaustiedoille. Sen sijaan, että on käytössä yleiskäyttöinen mittaustietorajapinta, niin olisi voinut olla järkevää tehdä käyttötarkoitukseen suunniteltuja mittaustietotietokantatauluja. Tällöin tietokantataulujen määrää olisi järkevä lisätä siten, että määriteltäisiin millaisia suureita halutaan mitata ja tallennettaisiin käyttöympäristön mukaan omiin tauluihin. Sovitetut tietokantataulut mahdollisesti helpottaisivat erinäisten operaatioiden tekemistä hyvin paljon. Esimerkiksi tällä tavalla saataisiin mahdollisesti nopeutettua tiedonhakua ja pystyttäisin hakemaan keskenään tärkeät tiedot nopeasti. Sovittaminen tosin vaatisi päätöksiä sisääntulevasta datasta. Tällöin tietokantataulut olisi optimoitu pelkästään haluttuja

suureita varten, joten tietokantataulujen koko olisi helpommin hallittavissa. Mahdollisesti voisi olla myös järkevää analysoida arvoja tallentaessa. Tämä toki hidastaa tallennusprosessia, mutta voisi huomattavasti nopeuttaa tärkeän datan hakua, jos datasta olisi valmiiksi merkittynä asioida, joissa on jotain vikaa. Toinen asia mitä pitäisi pohtia tietokannoissa on tietokantojen indeksointi. Tämä on lähinnä ongelma siksi, että tietokanta on luotu Django ORMia käyttäen, mikä ei luo indeksejä muuten kuin indeksoitavien kenttien tapauksessa, esimerkiksi lueteltu tyyppi -kentän.

Sovitetut tietokantatulut nopeuttaisivat ja yksinkertaistavat tietokantaoperaatioita. Niiden ansiosta käyttöliittymässä ei tarvitsisi miettiä ollenkaan, mitä pitää tehdä, jos rajapinnasta tulee uudenlainen suure vastaan. Lisäksi tällä tavalla saataisiin myös rajattua muistinkäyttöä tietokannassa, koska merkityksettömät mittaustulokset voitaisiin suoraan hylätä.

Aliluvussa 5.1 otettiin kantaa, kuinka tietokanta ajanmyötä hidastuu. Tietokantataulujen sovittaminen ei kuitenkaan ratkaise tätä ongelmaa. Kun kuluu tarpeeksi aikaa, mittaustulosten määrä kasvaa lineaarisesti, ja tietokantahaut hidastuvat. Sovitetuissa tietokantatauluissa on myös se riski, että sovelluspäivitysten takia mittaustuloksen nimi muuttuu, jolloin nimi ei vastaa tietokantatauluun sopivaa nimeä, jolloin mittaustulokset jäisivät tallentumatta.

5.3 Ratkaisuehdotusten analysointi

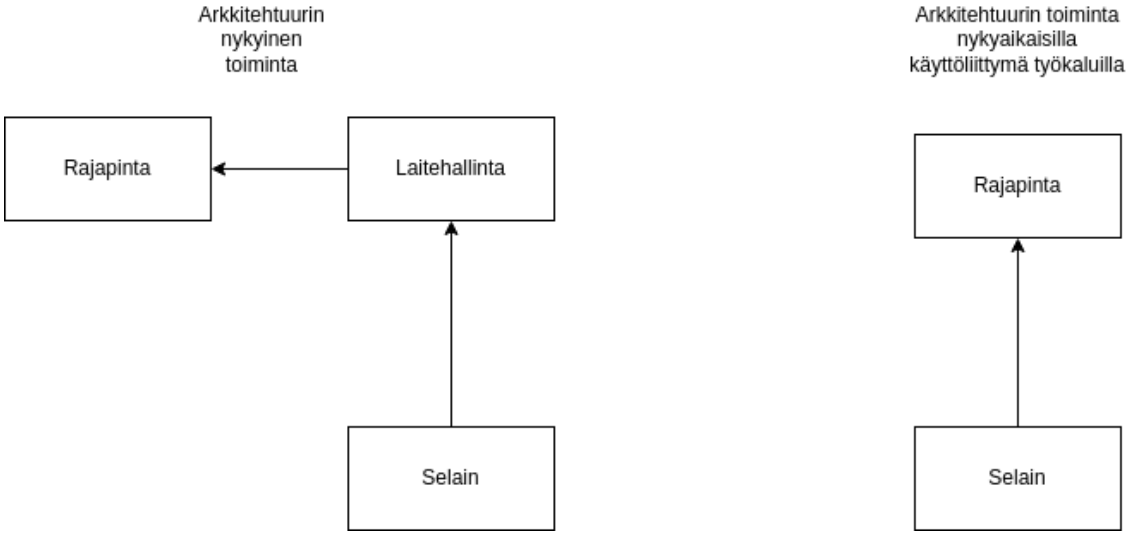
Ratkaisuehdoitukset eivät ole toisiaan poissulkevia. On täysin mahdollista, että tehtäisiin järjestelmä, jossa on dataan sovitettu tietokanta ja data tulee tietokantaan NoSQL-tietokannan läpi. Kuitenkin kummassakin ratkaisussa on omat ongelmansa, jotka on syytä tiedostaa. Jos ratkaisuja käytetään yhdessä, niin silloin voi esiintyä yhdistelmiä niihin liittyvistä ongelmista.

Yksinkertaistaminen on ohjelmistotekniikassa yleisesti hyve. Näin sovitetut tietokantataulut vaikuttaisivat hyvältä ratkaisulta. Ketterän kehittämisen näkökulmasta voisi olla jopa järkevää aluksi pitäytyä sovitetuissa tietokantatauluissa. Vasta siinä vaiheessa, kun tarvittaisiin joko lisää suureita tai ratkaisu paljastuisi liian hitaaksi joka tapauksessa, olisi syy vaihtaa monimutkaisempaan välimuistiratkaisuun.

Välimuistiratkaisua taas puoltaa se, että välimuistia joudutaan todennäköisesti joka tapauksessa käyttämään. Joko sen joutuu tekemään sovitettuihin tietokantoihin tai yleiskäyttöisiin tietokantoihin. Lisäksi jos lähtötilanteessa on olemassa yleiskäyttöinen tietokanta, niin sen purkaminen sovitetuksi tietokannaksi voi olla turhan työlästä tuotettuun arvoon nähden.

5.4 Muita jatkokehitysajatuksia

Tässä aliluvussa esitellään muita kehittämiskohteita kuin mittaustietokannan optimointi. Yleisesti voidaan todeta laitehallinnan ja käyttökokemuksen olevan jatkuvan kehityksen kohteita, ja on tärkeää löytää hyviä ratkaisuja erilaisiin haasteisiin. Aliluvussa aluksi keskitytään REST-arkkitehtuuriin, jossa pohditaan sen järkevyyttä tässä sovelluksessa. Seuraavassa kappaleessa keskitytään arkkitehtuurivalintoihin ja olisiko joku muu arkkitehtuuri voinut olla käytännöllisempi. Tämän jälkeen käsitellään, miten REST-rajapintaa olisi voitu optimoida. Kun arkkiteh-



Kuva 5.1. *Yksinkertainen arkkitehtuurikuvaus*

tuuri on käsitelty, käsitellään päivityspakettien toimivuutta ja puutteita. Lopuksi vielä analysoidaan sovelluksen onnistumista.

Djangolla tehty REST-rajapinta ei välttämättä ollut aivan paras ratkaisu ongelmaan. Vaihtoehtoinen ratkaisu olisi voinut olla käyttää esimerkiksi GraphQL:ää. REST-rajapinnan kanssa ongelmaksi muodostui erityisesti monien erilaisten mallien kerros, jotka tarkastelivat ongelmaa hieman eri lähtökohdista. Itse sovellus ottaa kantaa lähinnä siihen, miltä data näyttää, kun taas rajapinta käsittelee dataa hyvin yleisluontoisesti. Tämä tarkoittaa sitä, että monille käyttöliittymän näkymille tarvitsee tehdä tarpeettoman monta rajapintapyyntöä. Jos käytössä olisi ollut GraphQL, olisi rajapintapyynnöissä suoraan voinut koota sopivan datan käyttöliittymälle. Myöskin voidaan pitää hieman kyseenalaisena, että SQL-tietokantaan tallennettiin binääritiedostoja ja aikaleimattua mittausdataa.

Kuten aiemmin aliluvussa 3.2 käsiteltiin web-sovelluksen arkkitehtuurivalintaa, mahdollisesti oltaisiin voitu tehdä erilainen sovellus, joka ei olisi käyttänyt rajapintaa vaan olisi ollut monoliittinen kokonaisuus. Tämä johtuu siitä, että sovellus rakennettiin kuvan 5.1 mukaisesti. Kuvassa esitetään, miten pyynnöt välitetään rajapinnalle. Järjestelmässä tehdään ensin pyynnöt laitehallintasovellukseen, joka tekee pyynnöt rajapintapalvelimelle. Tämä aiheuttaa tarpeettomasti odotusaikaa, kun olisi myös mahdollisuus siihen, että pyynnöt tehtäisiin selaimesta suoraan rajapintaan. Tämmöinen toiminnallisuus on esimerkiksi moderneissa React-sovelluksissa. Tähän ongelmaan monoliitinen kokonaisuus olisi ollut hyvä, koska silloin tietokanta olisi suoraan ollut laitehallintasovelluksen alla. Mutta toisaalta jonkinlainen rajapinta olisi vaadittu siihen, että laitteet keskustelevat sovelluksen kanssa. Tällöin sovelluksen ei tarvitsisi jatkossa keskustella toisen sovelluksen kanssa, vaan kaikki mahdolliset tietokantapyynnöt tehtäisiin tietokantaan ilman rajapintaa. Toki tietokantaa käyttäessä pitää tehdä käyttäjille sopivat oikeudet, jotta käytettävien laitteiden data ei vuoda.

Rajapintapyyntöjä itsessäänkin voisi jonkin verran nopeuttaa. Rajapinnan näkymät ovat käytännössä kaikki kirjoitettu tyylillä, jossa näkymään on laitettu serialisaattori. Ilman serialisaattoriakin näkymät toimisivat ja tekisivät paljon vähemmän tarkistuksia, joista suurinosa ei

yleensä ole tarpeellisia. Jos serialisaattori halutaan poistaa näkymistä, tarvittaisiin todennäköisesti muita keinoja datan muuntamiseen ja lähetämiseen HTTP-pyynnön mukana. Vaatisi esimerkiksi oman koodin kirjoittamista ja testaamista, jotta datan integriteetti voidaan varmistaa. Vaikka tämä lisätyö voikin olla merkittävä, se voisi parantaa rajapinnan suorituskykyä huomattavasti, mikä voi olla erittäin tärkeää suurten datamäärien käsittelyssä.

Aliluvussa 4.5 käsiteltiin päivityspakettien tarjoamista laitteille, mutta niiden ongelmia ei käsitelty ollenkaan. Tietokannassa oikaistiin ohjelmistopäivitysten yhteydessä ja ajateltiin riittävän, että laite tietää versionsa ja rajapinta ei ota siihen mitään kantaa. Lopulta selvisi, että laitteella voi olla väärä tieto ja on hyvä varmistaa rajapinnalta, että onko laitteen tieto oikein. Tämän johdosta tietokantaan piti rakentaa tietokantataulu, joka sisältää tiedon laitteelle asennetuista tiedostoista. Johtuen projektin ajasajosta kyseistä ominaisuutta ei ehditty koskaan testaamaan, mutta ajatuksen tasolla tämä ei kuitenkaan merkittävästi lisännyt kompleksisuutta tietokantaan. Tätä varten myös rakennettiin oma rajapinta, josta voi kysyä asennetut tiedostot ja niiden tiedostojen versiot. Ratkaistavaksi kysymykseksi jäi, voiko jotenkin varmistaa laitteen ja palvelimen olevan samaa mieltä asennetuista tiedostoista. Luonnollinen tapa olisi luoda tiivisteet tiedostoista ja vertailla niitä. Tämän toimivuutta ei päästy koskaan testaamaan.

Loppu analyysina voidaan todeta web-sovelluksen olevan oikein toimiva ratkaisu itsessään. Sovelluksen kautta oli hyvin yksinkertaista tehdä yksinkertaisia laitehallintaan liittyviä toimenpiteitä ja seurata laitteiden tilaa. Ongelmat sovellukseen liittyen olivat hyvin pitkälle rajapinnassa johtuen yleensä siitä, että optimointeja ei oltu tehty tarpeeksi ajoissa. Optimointiongelmat johtivat siihen, että asiakas havaitsi sovelluksen toimivan liian hitaasti. Näistä kuitenkin yleensä selvittiin laittamalla dataa välimuistiin tai optimoimalla tietokantakyselyjä.

6 JOHTOPÄÄTÖKSET

Työssä toteutettiin sovellus, jossa voidaan rakentaa IoT-laitteiden hallintapalvelu, ja paneudutiin ohjelmistoteknisiin valintoihin, jossa käsiteltiin miten web-palvelu on rakennettu ja minkälaisia asioita yleensä palvelun rakentamiseen liittyy. Työ esittelee sovelluksen toiminnan, millaisia näkymiä ohjelmistossa on ja miten tietokanta on rakennettu. Lopuksi vielä arvioitiin työn onnistumista ja kävimme läpi keskeiset jatkokehitysajatukset, joita ohjelmistoon työn aikana kerkesi tulla. Samalla vastasimme johdannossa esitettyihin tutkimuskysymyksiin.

Analysoinnin tuloksena voidaan todeta työn onnistuneen hyvin. Ensinnäkin laitehallinnan siirtäminen tietokantaan oli oikein onnistunut päätös. Laitteiden tilasta saa irti helpommin haluttuja tietoja sovelluksen näkymien kautta kuin aiemmin sai koostetuista laskentataulukoista. Lisäksi sietoisuus virheille kasvoi, koska enää ei ollut mahdollista tyhjentää laskentataulukon soluja vahingossa niin helposti kuin aiemmin. Suurin voitto kuitenkin oli se, että nyt laitehallinta oli keskitetysti yhdessä paikassa ja saavutettavissa käytännössä mistä tahansa. Samalla pystyttiin myös luottamaan siihen, että laitteiden tiedot ovat ajantasalla. Lisäominaisuutena mittaustuloksien kerääminen toi lisäarvoa, koska vastaavaa ominaisuutta ei ollut aiemmin olemassa. Päivityshallinta myös helpotti ihmisten työtä, koska enää ei tarvinnut yksitellen käydä SSH-yhteydellä jokaista laitetta erikseen läpi.

IoT-laitehallinnan rakentaminen Djangoa ja Django REST Frameworkia hyödyntäen oli varsin hyvä ratkaisu. Djangon käyttäminen teki ohjelmoinnista hyvin yksinkertaista ja DRF:n rajapinnasta sai tehtyä helposti rajapintapyyntöjä. Myöskin yhteinen rajapinta sekä käyttöliittymälle että laitteille oli oikein hyvä ratkaisu. Laitehallinnan suunnitteluongelmat olivat suhteellisen pitkälle asioita, mitä ei tule normaalisti ajateltua. Esimerkiksi mitä pitää huomioida tarjotessa päivityspaketteja? Onko kaikki päivitykset samanlaisia? Onko aina päivitysten mukana konfiguraatiotiedostot? Miten ilmoittaa käyttöliittymän käyttäjälle, että päivitykset on asennettu? Myöskin ongelmat metriikkatiedon tallentamisesta eivät tulleet aluksi mieleen, vaan tallennusmekanismi vaikutti kaikinpuolin hyvältä idealta.

Mittaustietokantaan liittyvät tutkimuskysymykset olivat:

- Miten hyödyntää relaatiotietokantoja jatkuvan datavirran kanssa?
- Kun yllä olevan ongelman ratkaisee, niin mitä muita ongelmia ratkaisu voi aiheuttaa? Miten nämä saadaan ratkaistua?

Näihin saatiin vastauksiksi saatiin kaksi tapaa, joilla voidaan parantaa tietokannan suorituskykyä jatkuvan tiedonkäsittelyn kanssa. Ensimmäinen tapa käsiteltiin aliluvussa 5.1, jossa ratkaisuna oli lisätä välimuistin käyttöä, jotta dataa saataisiin luettua nopeammin. Toinen tapa

käsiteltiin aliluvussa 5.2, jossa yritettiin tehdä tietokantataulujen sovittaminen dataan, jotta luku- ja kirjoitusoperaatiot olisi yksinkertaisempia. Aliluvussa 5.3 käsiteltiin näiden ratkaisuiden ongelmia ja tultiin lopputulokseen, jossa tietokantataulujen sovittaminen olisi työläs ja todennäköisesti väliaikainen ratkaisu. Välimuistin käyttöä sen sijaan kannattaa tarkastella enemmän ja lisätä sen käyttöä. Toisaalta datan kirjoittamiseen välimuisti ei tuota merkittäviä hyötyjä.

Päivityspaketteihin liittyi tutkimuskysymys, mitä pitää huomioida tarjottaessa päivityspaketteja. Vastauksen ratkaisua käsiteltiin aliluvussa 4.5, jossa esiteltiin tietokantamallit, joilla tuotiin päivityspaketeille tarpeeksi rakennetta, jotta ne oli helppo parittaa laitteelle. Lisäksi tietokantamalleihin lisättiin aikaleimat, tämä lisäsi päivityspaketeille versiohistorian, jotta laite pystyi tarkistamaan onko uusia versioita olemassa. Viimeisenä tietokantaan luotiin mahdollisuudet siihen, että laite voi tarkistaa onko mikä versio asennettuna laitteelle.

Siitä huolimatta, että tutkimuskysymyksiin saatiin vastaukset, palveluun jäi vielä jatkokehitettävää. Nämä aiheet käsiteltiin aliluvussa 5.4. Arkkitehtuuriin olisi kaksi mahdollista kehityssuuntaa. Ensimmäinen idea oli vaihtaa REST-arkkitehtuuri esimerkiksi GraphQL-arkkitehtuuriin. Tällöin saataisiin luotua järkevämpiä pyyntöjä rajapintaan. Toinen suunta olisi ollut tehdä monoliittinen rakenne, jossa ei olisi ollut rajapintaa. Tässä ratkaistavana ongelmana olisi, miten keskustella laitteiden kanssa. Näiden lisäksi pohdittiin REST-rajapintojen optimointia, jotta ne voisivat toimia nopeammin. Viimienen käsitelty jatkokehitysidea liittyi päivityspaketteihin, jotta voidaan varmistaa laitteella olevan ohjelmistoversio.

Toteutettu web-sovellus oli oikein hyvä ratkaisu käyttötarkoitukseensa. Myös metriikan tallentaminen itsessään havaittiin oikein toimivaksi ominaisuudeksi.

LÄHTEET

[1] Ashton, K., That 'internet of things' thing, RFID journal, Vol. 22, No. 7, 2009, s. 97–114.

[2] Alaba, F. A. et al., Internet of Things security: A survey, Journal of Network and Computer Applications, Vol. 88, 2017, s. 10–28.

[3] Manyika, J. et al., Unlocking the Potential of the Internet of Things, McKinsey Global Institute, Vol. 1, 2015.

[4] Cooper, J., James, A., Challenges for database management in the internet of things, IETE Technical Review, Vol. 26, No. 5, 2009, s. 320–329.

[5] Haerder, T., Reuter, A., Principles of transaction-oriented database recovery, ACM computing surveys (CSUR), Vol. 15, No. 4, 1983, s. 287–317.

[6] Wachter, S., Normative challenges of identification in the Internet of Things: Privacy, profiling, discrimination, and the GDPR, Computer law & security review, Vol. 34, No. 3, 2018, s. 436–449.

[7] Mahdavinejad, M. S. et al., Machine learning for Internet of Things data analysis: A survey, Digital Communications and Networks, Vol. 4, No. 3, 2018, s. 161–175.

[8] Qian, L. et al., Cloud computing: An overview, teoksessa: IEEE international conference on cloud computing, Springer, 2009, s. 626–631.

[9] Sadiku, M. N., Musa, S. M., Momoh, O. D., Cloud computing: opportunities and challenges, IEEE potentials, Vol. 33, No. 1, 2014, s. 34–36.

[10] Ghosh, N., Ghosh, S. K., Das, S. K., SelCSP: A framework to facilitate selection of cloud service providers, IEEE transactions on cloud computing, Vol. 3, No. 1, 2014, s. 66–79.

[11] Chen, H., Jia, X., Li, H., A brief introduction to IoT gateway, teoksessa: IET international conference on communication technology and application (ICCTA 2011), IET, 2011, s. 610–613.

[12] Noreen, U., Bounceur, A., Clavier, L., A study of LoRa low power and wide area network technology, teoksessa: 2017 International Conference on Advanced Technologies for Signal and Image Processing (ATSIP), IEEE, 2017, s. 1–6.

[13] Gortmaker, J., Janssen, M., Wagenaar, R. W., The advantages of web service orchestration in perspective, teoksessa: Proceedings of the 6th international Conference on Electronic Commerce, 2004, s. 506–515.

[14] Di Lucca, G. A., Fasolino, A. R., Testing Web-based applications: The state of the art and future trends, Information and Software Technology, Vol. 48, No. 12, 2006, s. 1172–1186.

[15] Fielding, R. et al., *Hypertext transfer protocol–HTTP/1.1*, tekninen raportti, 1999.

[16] Yan, G., Ng, W.-K., Lim, E.-P., Toolkits for a distributed, agent-based web commerce system, teoksessa: Proceedings of the International IFIP Working Conference on Trends in Distributed Systems for Electronic Commerce (TrEC'98), Hamburg, Germany, Citeseer, 1998.

[17] Deacon, J., Model-view-controller (mvc) architecture, Online][Citado em: 10 de março de 2006.] http://www. jdl. co. uk/briefings/MVC. pdf, Vol. 28, 2009.

[18] Django Software Foundation, *Django*, versio 3.2, 14. kesäkuuta 2022, Saatavilla: URL: https://djangoproject.com.

[19] Richards, R., "Representational state transfer (rest)", teoksessa: *Pro PHP XML and web services*, Springer, 2006, s. 633–672.

[20] Wilton, P., Beginning JavaScript, John Wiley & Sons, 2004.

[21] Peng, D., Cao, L., Xu, W., Using JSON for data exchanging in web service applications, Journal of Computational Information Systems, Vol. 7, No. 16, 2011, s. 5883–5890.

[22] Suhonen, J. et al., Unified service access for wireless sensor networks, teoksessa: 2012 Third International Workshop on Software Engineering for Sensor Network Applications (SESENA), IEEE, 2012, s. 49–55.

[23] Drake, J. D., Worsley, J. C., Practical PostgreSQL, "O'Reilly Media, Inc.", 2002.