Asiakkuuden ja asiakkaan käsitteen yhdistäminen IoT-laitteiden laitehallintaan

HAMK
HÄMEEN AMMATTIKORKEAKOULU
HÄME UNIVERSITY OF APPLIED SCIENCES

Ammattikorkeakoulututkinnon opinnäytetyö

Riihimäki, tieto- ja viestintätekniikka

Kevät, 2018

Valtteri Virta

HÄMEEN AMMATTIKORKEAKOULU
HÄME UNIVERSITY OF APPLIED SCIENCES

Tieto- ja viestintätekniikka
Riihimäki

Tekijä Valtteri Virta **Vuosi** 2018

Työn nimi Asiakkuuden ja asiakkaan käsitteen yhdistäminen IoT-laitteiden laitehallintaan

Työn ohjaaja Petri Kuittinen

TIIVISTELMÄ

Tämän opinnäytetyön tarkoituksena on yhdistää asiakkuus ja asiakkaan käsite jo olemassa olevaan IoT-laitteiden laitehallintaan. Työn toimeksiantajana toimii UnSeen Technologies Oy.

Opinnäytetyö alkaa teoriaosuudella, jossa käydään läpi projektissa käytettäviä teknologioita ja perehdytään syihin, jotka vaikuttivat kyseisten teknologioiden valintoihin. Teoriaosuutta seuraa käytännönosuus, missä käydään läpi tehtyä työtä kolmessa eri osassa. Ensimmäisenä on palvelinpuolen ohjelmointi, jota seuraa erillinen osuus tietokantaintegraatiosta, ja lopuksi käydään läpi asiakaspuolella tehtyä työtä.

Projekti saatiin valmiiksi aikataulussa, vaikka toiminnallisuutta oli lopulta suunniteltua vähemmän. Projektin kehittämistä aiotaan kuitenkin jatkaa myöhemmin, ja aiemmin asetetut tavoitteet toteutetaan.

Avainsanat CoAP, LwM2M, ohjelmistokehitys, Smart Objects

Sivut 32 sivua

HÄMEEN AMMATTIKORKEAKOULU
HÄME UNIVERSITY OF APPLIED SCIENCES

Information and Communication Technology
Riihimäki

Author Valtteri Virta **Year** 2018

Subject Connecting clientship and the concept of client with an already existing IoT-device management service

Supervisor Petri Kuittinen

ABSTRACT

The aim of this thesis project was to connect clientship and the concept of client a with an already existing IoT-device management service. The commissioner of the project is UnSeen Technologies Oy.

This thesis starts with a theoretical part, where the technologies that were used in the project are introduced, and the reasons for choosing these technologies are examined. The empirical part follows the theoretical part, and in the practical part we examine hands-on work is examined under three sections. The first section covers the backend, the second one is about the database integration and lastly, the frontend is described.

The project was finished within the given time frame, although it was less functional than originally planned. The project will be developed further in the future, so that the previously set goals can be met.

Keywords CoAP, LwM2M, Smart Objects, software development

Pages 32 pages

SANASTO

API Application programmin interface eli sovelluksen ohjelmointirajapinta on ennalta määritelty kokoelma funktioita, joita voidaan käyttää rajapinnan kautta.

CoAP Constrained Application Protocol on rajoittuneisiin verkkoihin kehitetty tiedonsiirtoprotokolla.

DB Database eli tietokanta, mihin tietoa tallennetaan. Projektin kontekstissa käytettävä tietokanta on nimeltään MariaDB.

HTML Hypertext Markup Language on standardisoitu web-kehityksessä käytettävä kuvauskieli.

IMEI International Mobile Equipment Identity on 15-merkkinen tunnus, joka tunnistaa matkapuhelinverkossa olevia laitteita.

IoT Internet of Things eli esineiden internet kuvaa kokonaista infrastruktuuria ja yksittäisiä laitteita, jotka ovat yhteydessä Internettiin.

JDBC Java Database Connectivity on ohjelmointirajapinta, joka määrittelee, että kuinka asiakkaat pääsevät käsiksi tietokantaan.

JSX JavaScript XML on mm. React-sovelluskehyksessä HTML:n tilalla käytettävä laajennus JavaScript-ohjelmointikielen syntaksiin.

LwM2M Lightweight machine-to-machine on Open Mobile Alliancen (OMA) kehittämä protokolla sensoriverkkoihin ja kevyeen koneiden väliseen kommunikointiin.

MVP Minimum viable product eli pienin mahdollinen tuote on versio tuotteesta, jossa on juuri tarpeeksi ominaisuuksia ja toiminnallisuutta, että se voi tyydyttää asiakkaan käyttötarpeet.

NB-IoT Narrowband-IoT on kevyt pieniin tiedonsiirto määriin ja vähäiseen virrankulutukseen suunniteltu radiostandardi.

ORM Object/Relational Mapping on relaatiotietokantojen kanssa käytettävä tekniikka, jossa luokat pystytään sovittamaan suoraan näitä vastaaviin tietokantatauluihin.

SISÄLLYS

1 JOHDANTO

Tässä opinnäytetyössä tavoitteena on yhdistää asiakkuus ja asiakkaan käsite jo olemassa olevaan IoT-laitteiden laitehallintaan. IoT-laitteista käytetään jatkossa nimikettä asiakaslaitteet, jotta yhteys asiakaslaitteiden ja laitehallinnan asiakkuuden välillä olisi selkeämpi.

Sekä asiakaslaitteet että laitehallinta ovat toteutettu LwM2M-protokollan mukaisesti, joten protokollaan tutustuminen on ensimmäinen askel opinnäytetyön toteutuksessa. Projektin edetessä tutustutaan myös moneen ennestään tuntemattomaan sovelluskehykseen, jotka tulisi hallita. Lisäksi vastaan tulee erilaisia toimintatapoja, joita tulisi noudattaa, kuten yksikkötestien kirjoittaminen.

Asiakkuuden ja asiakkaan käsitteen lisääminen alkaa laitehallinta palvelimelta, jossa tulee yhdistää asiakaslaitteelta saatu tunnus oikeaan asiakkaaseen. Tunnistamisen jälkeen asiakaslaitteet tallennetaan asiakaskohtaiseen tietokantaan.

Asiakaspuolella tehdään karkea vedos käyttöliittymästä, johon kuuluu sisäänkirjautumisnäkymä asiakkaalle. Lopuksi asiakaspuolen sisäänkirjautuminen yhdistetään palvelinpuolen toiminnallisuuteen.

2 KEHITTÄMISTYÖN TIETOPERUSTA

Kehittämistyön tietoperustassa käydään läpi opinnäytetyössä käytettyjä teknologioita, ja perehdytään syihin, jotka vaikuttivat kyseisten teknologioiden valintoihin. Teknologioita käydään läpi niiden käyttöjärjestyksessä projektissa.

Ensimmäisenä käydään läpi laitehallinnassa käytettävää LwM2M-palvelinta ja sen kanssa kommunikoivaa moniasiakkuus-arkkitehtuurimalliin pohjautuvaa tietokantaa. Viimeisenä tutustutaan asiakaspuolen toteutukseen käytettyihin sovelluskehyksiin.

2.1 Leshan

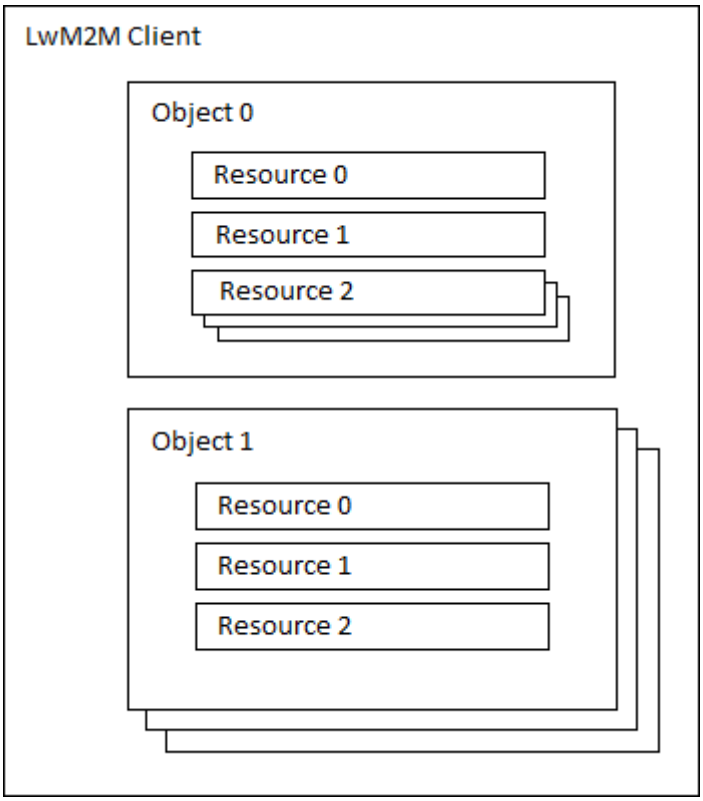
Leshan on Eclipse Foundationin kehittämä avoimen lähdekoodin Java-kirjasto, jonka tarkoituksena on helpottaa LwM2M-palvelinsovellusten toteuttamista. CoAP implementaationa Leshan käyttää Eclipse Foundationin Californiumia. (Eclipse Foundation, 2015.)

Oletuksena kirjastosta löytyy suora tuki Jetty web -palvelimelle, jossa asiakaspuolen sovellus pyörii. Lähdekoodista löytyy esimerkki palvelintoteutuksesta, jossa on mukana AngularJS-sovelluskehyksellä toteutettu asiakaspuolen applikaatio.

2.2 Smart Objects

Jokaista LwM2M-asiakaslaitteen esittämää yksittäistä tietoa, kuvataan yhtenä resurssina. Resurssit järjestellään objektien instansseiksi, ja instanssit järjestetään edelleen objekteiksi. Jokaiselle resurssille annetaan uniikki tunnus objektin sisällä. (Open Mobile Alliance, 2017.)

Yhdellä asiakaslaitteella voi olla *n* määrä resursseja, jotka kaikki kuuluvat johonkin objektiin. Kuva 1 kuvaa asiakaslaitteen, ja siihen kuuluvien objektien sekä resurssien rakennetta.



Kuva 1. Rakenne LwM2M-asiakaslaitteen, siihen kuuluvien objektien sekä resurssien välillä.

Yksittäinen resurssi tai objekti voi esiintyä useammin kuin kerran ja näitä esiintymisiä kutsutaan instansseiksi. Oletuksena instansseja on yksi ja se on järjestysnumeroltaan ensimmäinen, eli tietotekniikassa 0. Standardisoitu LwM2M objektin esitysmuoto näyttää seuraavalta:

```
<object ID>/<object instance ID>/<resource ID>
```

Jokaiselle objektille määritellään uniikki LwM2M-objekti tunnus OMA-objektirekisteristä. Rekisteristä löytyy vapaasti käytettäviä standardisoituja objekteja erilaisiin käyttötarkoituksiin. Rekisterin ylläpitäjänä toimii OMA Naming Authority (OMNA).

Esimerkkinä, lähetetään palvelimella olevalle paikkatieto objektille DISCOVER-komento. Saadaan selville, että objektista (ID:6) löytyy yksi instanssi (ID:0). Vapaaehtoiset resurssit ID:2 ja ID:3 ovat käytössä, kun taas vapaaehtoiset resurssit ID:4 ja ID:6 eivät ole käytössä.

Vastaus joka saadaan palvelimelta, kun lähetetään DISCOVER /6 pyyntö:

```
</6/0/0>,</6/0/1>,</6/0/2>,</6/0/3>,</6/0/5>
```

OMA objekti rekisteriin on myös allokoitu tilaa järjestöille ja yksityisille yrityksille, jos he haluavat rekisteröidä oman objektinsa (IPSO Alliance, 2018). Uusien objektien luomiseen voi käyttää objektieditoria, jonka tarjoaa OMA (Open Mobile Alliance, 2018).

2.3 **Docker**

Docker on ns. kontteja käyttävä alusta, jolla voidaan ajaa erilaisia sovelluksia ja/tai palveluita kevyesti ja yksittäin, erillään muista. Virtuaalikoneiden tapaisesti docker käyttää konteissa vedoksia, joissa prosessit pyörivät. (Docker, n.d.)

Vedokset Dockerissa ovat todella kevyitä, muista riippumattomia ajettavia paketteja. Vedoksista löytyy ainoastaan pienin mahdollinen ympäristö, joka vaaditaan prosessin ajoon.

Dockerin suuren suosion ansiosta, lähes kaikki halutut sovellukset löytyvät laajasta pakettivarastosta. Paketteja voi hakea avainsanojen avulla versioiden mukaan *docker pull* -komennolla:

```
docker pull mysql/mysql-server:latest
docker run --name=mysql-db -d mysql/mysql-server:latest
```

Ajossa olevia prosesseja voi pysäyttää komennolla *docker stop [name]* ja taas käynnistää komennolla *docker start [name]*. Luodut prosessit nähdään kirjoittamalla *docker ps*. Syntaksissa on paljon samankaltaisuuksia Unix-komentoriviin nähden.

Oletuksena ajettavissa sovelluksissa on kaikki portit suljettuna ulkomaailmaan. Jos näitä halutaan avata, niiden muokkaaminen tapahtuu käyttämällä *-p* -asetusta komennon yhteydessä.

2.4 **MariaDB**

MariaDB on avoimen lähdekoodin relaatiotietokanta, joka on alkuperältään suora haara MySQL-tietokannasta. Syntaksi molemmissa tietokannoissa on samanlainen, ja MariaDB:n esitellään olevan suora korvaaja MySQL-tietokannalle. Tietokannat ovat siis helposti ja nopeasti vaihdettavissa keskenään. (MariaDB Foundation, 2018)

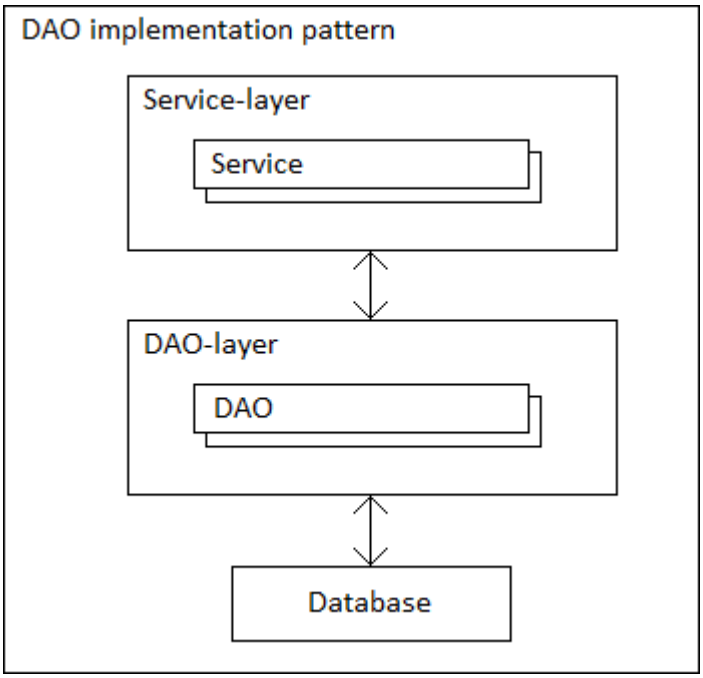
2.4.1 MariaDB vs. MySQL

Tietokantojen yhteensopivuus voi tehdä oikean ratkaisun valitsemisesta vaikeaa. Yleisesti ottaen MySQL tuntuu monelle ihmiselle tällä hetkellä tutummalta, ja se on myös vaihtoehdoista se suositumpi (DB-Engines, 2018).

Tällä hetkellä MariaDB on toimintorikkaampi vaihtoehto – etenkin ilmaisversioiden kohdalla – ja siksi se valittiin projektiin tietokannaksi. Tärkein ilmaisversioista löytyvä ero on tietojen kryptaus -toiminnallisuus, joka suojaa tietokantaan tallennettavan datan (Percona, 2016).

2.4.2 Data Access Object

Data Access Object (DAO) on objekti, joka välittää abstraktin rajapinnan jonkin typpiseen tietokantaan, tai muuhun tiedon säilytys mekanismiin. DAO välittää sovellukselle tietynlaista dataa paljastamatta kriittisiä yksityiskohtia tietokannasta. (Oracle, n.d.)



Kuva 2. DAO implementaation perusmalli.

Saatavana etuna tulee selvä erotus kahden eri applikaation osan välillä kuten esitetty kuvassa 2. Applikaation osien ei tarvitse tietää mitään

toisistaan, ja niiden voidaan olettaa kehittyvän usein ja toisistaan riippumattomasti. Tällä tavoin tallennus mekanismeihin kohdistuvat muutokset voidaan toteuttaa muuttamalla ainoastaan DAO-tasoa ja kaikki muut applikaation osat jäävät koskemattomiksi.

Abstraktin rajapinnan implementointi saattaa myös tuoda mukanaan negatiivisia haittavaikutuksia. Yhteyksien ottaminen tietokantaan kasvaa usein huomaamattomasti, ja samalla unohtuu kuinka paljon lukuisat yhteydet kuormittavat applikaatiota. (IBM, 2016.)

2.4.3 Moniasiakkuus

Moniasiakkuus kuvaa tilannetta, jossa yksi ohjelmiston instanssi, jota ajetaan palvelimella, palvelee montaa eri käyttäjää (Gartner, n.d.). Tietokanta näkökulmasta moniasiakkuus voidaan toteuttaa kolmella eri tavalla (Microsoft, 2018).

Kaikista ehkä suora viivaisin ja radikaalein ratkaisu, on toteutus monella tietokannalla. Jokaiselle sovelluksen käyttäjälle, eli asiakkaalle, luodaan oma tietokanta, jonne asiakkaan tiedot tallennetaan. Lisäksi, jos asiakkaan fyysiselle tietokannalle tapahtuu jotain, sillä ei ole vaikutusta muihin asiakkaihin.

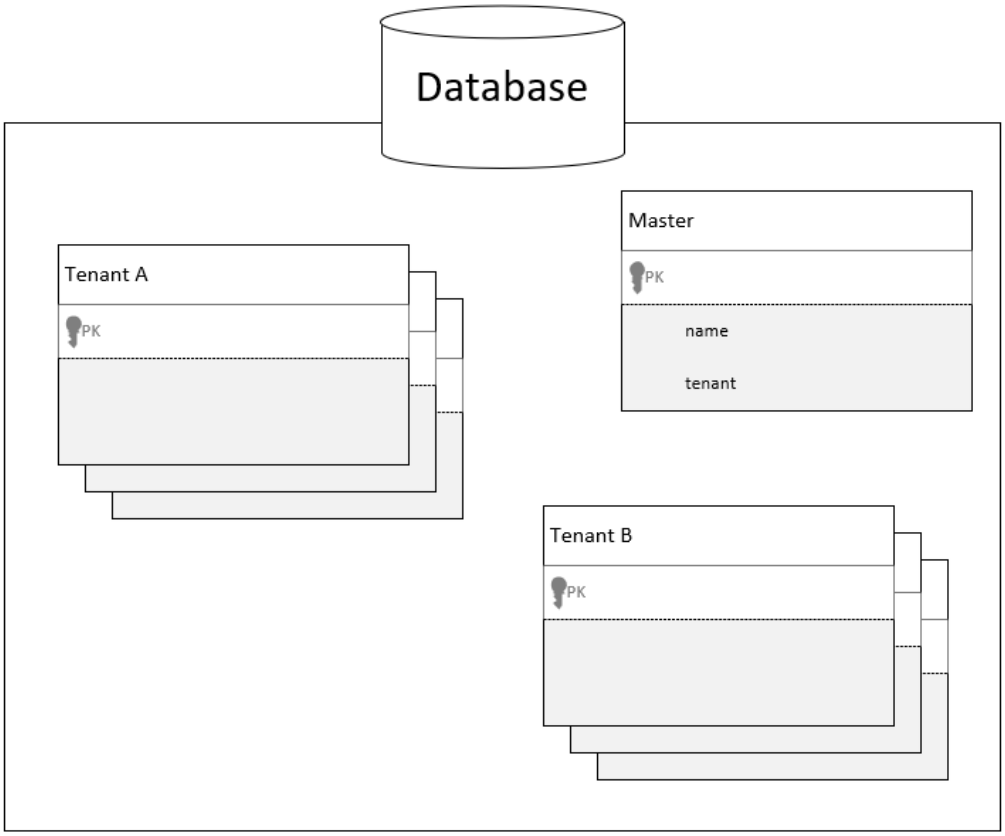
Toisen ääripään ratkaisu, on toteutus yhdellä kaaviolla ja yhdellä fyysisellä tietokannalla. Tällöin kaikkien asiakkaiden tiedot ovat tallennettu tietokannassa samoihin tauluihin.

Asiakkaiden tiedot erotellaan toisistaan jonkinlaisella tunnuksella, esimerkiksi asiakas tunnuksella. Suurille data määrille tämä ratkaisu ei kuitenkaan ole suositeltava, sillä yksittäisen tietokannan koko tällä ratkaisulla kasvaa äkkiä valtavaksi.

Näiden kahden ratkaisun välillä on vielä toteutus monella kaaviolla ja yhdellä fyysisellä tietokannalla. Kaikki tieto tallennettaisiin jälleen yhteen tietokantaan, mutta joka asiakkaalla on oma kaavio, jonka mukaan tiedot tallennetaan asiakkaan omiin tauluihin. Tämä vaihtoehto on välimalli kahden aikaisemmin esitetyn toteutuksen väliltä.

Projektissa käytetään monen kaavion toteutusta, koska tallennettavaa dataa on erittäin paljon ja asiakaskohtaiset data määrät vaihtelevat suuresti. Monen kaavion toteutus tuo myös turvaa, kun tiedetään että jos yhden asiakkaan tiedot katoavat, niin muut pysyvät koskemattomina.

Kuvassa 3 on kuvattu moniasiakkuus tietokannan rakennetta, jossa on mukana erillinen isäntä-tietokanta.



Kuva 3. Moniasiakkuus tietokannan rakenne.

2.5 Hibernate ORM

Hibernate on sovelluskehys, jonka tehtävänä on sovittaa Java-tietotyypit tietokannan tietotyyppeihin, sekä sovittaa kokonaisia Java-luokkia niitä vastaaviin tietokantatauluihin (Hibernate, n.d).

Suuremmissakin sovelluksissa Hibernate skaalautuu erinomaisesti, sen lähdekoodi on avoin ja dokumentaatio on harvinaisen kattava. Suurin syy sovelluskehyksen valintaa oli kuitenkin toimeksiantaja yritykseltä löytyvä valmis geneerinen DAO implementaatio, joka käyttää Hibernatea.

2.6 HikariCP

HikariCP on erittäin kevyt, vain 130Kb kokoinen, todella nopea JDBC-yhteysallas -kirjasto. HikariCP lähdekoodi on avoin, ja kokonaisuudessaan kirjasto on hyvin helppokäyttöinen. (Brett Wooldridge, n.d.)

Esimerkkinä suora HikariCP toteutus käyttäen *db* -nimistä MySQL-tietokantaa portissa *3306*, käyttäjällä *'root'* ja salasanalla *'root'*:

```
HikariDataSource ds = new HikariDataSource();
ds.setJdbcUrl("jdbc:mysql://localhost:3306/db");
ds.setUsername("root");
ds.setPassword("root");
```

2.7 **Angular 6**

Leshan-palvelimesta tehdyn demo applikaation asiakaspuoli oli toteutettu käyttäen Angular 1 -nimellä tunnettua AngularJS-sovelluskehystä. Tammikuussa 2018 AngularJS sovelluskehyksen kehittäjä, Google, ilmoitti että AngularJS siirtyy ylläpitoon, mikä tarkoittaa aktiivisen kehityksen lakkauttamista.

Ilmoituksen johdosta – ja osittain siksi että AngularJS on hidas ja hankala kehittää – tulevan projektin asiakaspuoli päätettiin kirjoittaa uudestaan käyttäen modernimpaa web-sovelluskehystä.

Vaihtoehtoista sovelluskehystä valittaessa vertailtiin suosituimpia vaihtoehtoja. Lopulta valinta kohdistui kuitenkin Angular-sovelluskehykseen, ja käyttöön otettiin maaliskuussa 2018 julkaistu versio 6 Angularista.

Vertailtaessa suosituimmat web-sovelluskehykset olivat kiistatta Angular ja React. Altavastaajana mukaan vertailuun tulee vielä hieman vähemmän suosittu, mutta räjähdysmäisessä kasvussa oleva Vue.js (NPM, 2018).

2.7.1 Angular vs. React vs. Vue.js

Vertailtavat web-sovelluskehykset ovat hyvin erilaisia toisistaan. Angular sekä React ovat suurten yritysten, Angular Googlen ja React vastaavasti Facebookin, kehittämiä sovelluskehyksiä, kun taas Vue.js on alun perin yksittäisen henkilön kehitystyön aikaansaannos.

Nykyään kaikki sovelluskehykset ovat avointa lähdekoodia, Angularin sekä Reactin kehitys jatkuu suurten yritysten tukemina, ja myös Vuen ympärille on muodostunut harras ja vannoutunut kehitystiimi.

Kolmesta vertailtavasta React eroaa ehkä eniten kaikista. Reactissa on käytössä sen oma JSX eikä perinteinen HTML. JSX syntaksi on lähellä XML syntaksia, ja kaikki JSX kirjoitetaan JavaScriptin sisään.

Reactilla toteutettu “Hello, world!” -esimerkki:

```
ReactDOM.render(
  <h1>Hello, world!</h1>,
  document.getElementById('root')
);
```

Vue.js on monella tapaa saman tyylinen Angularin kanssa. Molemmissa on käytössä HTML, dependenssi injektointi, direktiivit ja mahdollisuus käyttää suosittua TypeScript-ohjelmointikieltä.

JavaScript ei ole paras mahdollinen valinta suurien monimutkaisten ohjelmistojen kehitykseen. JavaScript ei ole käännettävä kieli, joten virheiden

löytäminen hankalampaa kuin käännetyissä kielissä. Lisäksi olio-ohjelmoinnissa mukana olevat piirteet puuttuvat täysin, joten suurien applikaatioiden kehitys turvallisesti on hankalaa. TypeScriptin tehtävä on sulkea tämä rako, ja paikata JavaScriptin puutokset. (Rozentals 2017, 9.)

TypeScriptillä toteutettu “Hello, world!” esimerkki:

```
function hello(target) {
  return "Hello, " + target + "!";
}
let tmpl = "world";
document.body.innerHTML = hello(tmpl);
```

Lopulta Angularin valinnan kanssa oli hyvin vähän tekemistä sovelluskehyksien toiminnallisuuksien kannalta, etenkin kun yksikään sovelluskehys ei erottunut erityisen paljoa positiivisesti edukseen.

Valinnan avulla pyrittiin laskemaan jo korkealla olevaa oppimiskäyrää, joka projektilla oli lukuisista uusista teknologioista johtuen. Angular oli ainoa, jonka käytöstä löytyi aiempaa kokemusta ja toimeksiantajalta löytyi siihen valmiita uusiksi käytettäviä komponentteja.

2.7.2 MDBootstrap

MDBootstrap (MDB) on suosittu ja helppokäyttöinen sovelluskehys, joka painottaa responsiivista suunnittelua sekä mobiili-ensin -lähestymistapaa, jonka tärkeys korostuu nykypäivänä mobiili laitteiden ollessa yhä suositumpia webissä. (MDBootstrap, 2018.)

MDB tarjoaa laajan valikoiman visuaalisesti miellyttäviä ja valmiita HTML-komponentteja kuten nappeja, pudotusvalikkoja ja graafeja. Sovelluskehyksen sivuilta löytyy koodipätkiä komponenttien luomiseen.

Kaikki tarjolla olevat perus komponentit ovat ilmaisia, mutta joitakin monimutkaisempia komponentteja on laitettu lisenssi tyylisesti maksettavaksi ns. maksumuurin taakse.

MDB valittiin projektiin koska se on erittäin helppo käyttöinen ja se yksinkertaisesti näyttää hyvältä. Sovelluskehyksen käyttö myös säästää huomattavasti aikaa, koska komponenttien ulkonäön hiomista ei tarvitse tehdä itse.

3 SUUNNITTELU JA TOTEUTUS

3.1 Työn suunnittelu

Ennen työn toteutusta, on hyvä miettiä mitä asiakkuus konseptina merkitsee toteutettavassa sovelluksessa. Lisäksi tulee pohtia erilaisia rajatapauksia, joita tulee vastaan asiakkaiden tunnistusta toteuttaessa.

Palvelimelle rekisteröityessä asiakaslaitteille on asetettu tiettyjä vaatimuksia ja virheelliset rekisteröitymisyritykset tulee täten estää. Vastaavasti asiakaspuolen sovelluksessa tulee estää virheelliset kirjautumisyritykset.

Asiakkaislaitteiden tallentuessa asiakaskohtaisiin tietokantoihin tulee varmistaa, että asiakkailla ei ole pääsyä tai käyttöoikeuksia toisten asiakkaiden tietokantoihin. Ainoastaan toimeksiantaja yrityksen käyttäjät pääsevät hakemaan tietoja kaikkien asiakkaiden tietokannoista.

Toimeksiantaja yrityksellä on lisäksi oma tietokanta, johon kehityksessä mukana olevat testilaitteet ja niiden dataa tallennetaan. Tämän rinnalle tehdään vielä perinteinen isäntä-tietokanta mihin tallennetaan asiakkaiden tunnukset, yhteystiedot ja muu yleinen metatieto.

Tähän liitettynä tulee estää virheellisten asiakas laitteiden rekisteröitymisyritykset palvelimelle. Vastaavasti asiakaspuolen sovelluksessa tulee estää virheellisten käyttäjien kirjautumisyritykset ja lisäksi asiakkailla olevan näkymän tulee olla asiakaskohtainen.

Testaukseen käytetään asiakaslaitteiden lisäksi Leshan-kirjastosta löytyvää asiakaslaiteohjelmaa, jolla voidaan nopeasti simuloida kymmenkuntaa asiakaslaitetta yhtäaikaisesti.

3.2 Asiakkuuden lisääminen Leshan-palvelimelle

Ensimmäiset vaiheet asiakkuuden lisäämisessä projektiin voivat tuntua monimutkaisilta, koska alussa on hankala hahmottaa kokonaiskuvaa. Tästä huolimatta täytyy tyytyä siihen, että alussa asiakkuus on vain jokin abstrakti käsite, joka tulee itse kehitellä.

Yksinkertaisimmillaan asiakkuutta voidaan ajatella jonkinlaisena tunnuksena tai numerona, joka on uniikki jokaiselle asiakkaalle. Näin asiakkaat voidaan erottaa toisistaan, ja ennalta määritellyillä tunnuksilla voitaisiin estää niiden asiakaslaitteiden rekisteröintiyritykset, joissa tunnus olisi kelvoton.

Asiakkaan tunnistuksessa päädyttiin käyttämään 32-merkkistä satunnaisesti luotua tunnusta, joka koostuu kirjaimista ja numeroista. Asiakkaiden ei itse tarvitse tietoturvasyistä tietää tätä tunnusta.

Uusi tunnus tulisi generoida uutta asiakasta luodessa, ja se pistettäisiin talteen palvelimelle sekä asiakaslaitteisiin. Asiakkaiden tunnuksien lisäksi, tallessa olisi IMEI jokaiselta asiakaslaitteelta.

Tässä vaiheessa, kun järjestelmän testausta olisi edessä vielä paljon, päätettiin toistaiseksi mahdollistaa laitteiden rekisteröitymiset ilman tunnusta. Rekisteröityminen ilman tunnusta toimisi sillä ehdolla, että laitteen IMEI kuuluisi jollekin tiedossa olevalle laitteelle.

3.2.1 Asiakkuuden käsitteen lisäys

Myöhemmin asiakkaiden tunnukset tullaan tallentamaan tietokantaan. Tässä vaiheessa kuitenkin todistetaan konseptin toimivuus, ja siihen riittää väliaikainen ratkaisu. Palvelimen tulee osata erotella asiakkaille määritellyt tunnukset epäkelvollisista tunnuksista.

Luodaan taulukko, johon asetetaan muutama testikäyttöön generoitu tunnus. Tehdään väliaikaisesti myös toinen taulukko, johon laitetaan kuviteltuja asiakkaiden nimiä.

Tällä tavoin koko asiakkuuden käsitteen abstraktisuutta saadaan hieman konkreettisemmaksi. Näin on helpompi hahmottaa *asiakkuus*, ja myös kehitys sen ympärillä sujuu selkeämmin.

Seuraavaksi tehdään validointi laitteelta saataville imei ja token (tunnus) attribuuteille. Ensimmäisenä imei tarkistetaan käymällä läpi sen sisältö ja pituus. Sisällöltään imei tulee koostua pelkistä numeroista, ja sen pituus ei saa olla yli eikä alle 15 merkkiä. Sisältö on helppo tarkistaa käyttäen säännöllistä lauseketta (engl. regular expression):

```
!imei.matches("[0-9]+") || imei.length() != 15
```

Tämän jälkeen tarkistetaan tuliko tunnus laitteen rekisteröintiyrityksen mukana. Jos tunnusta ei tullut, niin määritellään laitteelle geneerinen testitunnus.

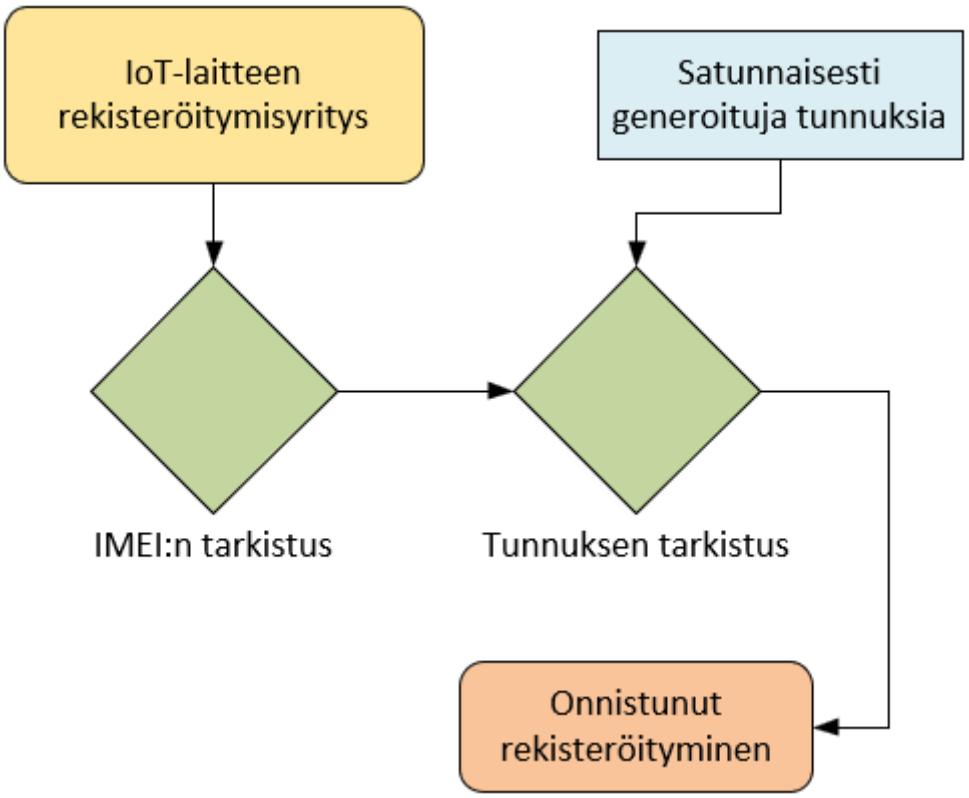
Jos laite yrittää rekisteröityä tunnuksen kanssa, niin verrataan sitä taulukossa oleviin arvoihin. Vastaavien arvojen löytäminen on helppoa Java 8:n mukana tulleella Stream API:lla:

```
Arrays.stream(getClientTkns()).noneMatch(token::equals)
```

Asiakkuutta käsitteenä täydennetään lisää myöhemmissä vaiheissa tietokanta integraatiolla ja sen mukana tulevilla lisäyksillä. Tässä vaiheessa asiakkuus on vaikea erotella, koska se on sidoksissa pelkkään tunnukseen.

3.2.2 Asiakkaan tunnistus

Konfiguroidaan testikäytössä oleva asiakasohjelma siten, että imei ja tunnus lähetetään rekisteröintiviestin mukana. Palvelinpuolella otetaan käyttöön aikaisemmin läpi käydyt validointi menetelmät, ja tarkistetaan rekisteröinti kuten kuvassa 4 on esitetty.



Kuva 4. Asiakaslaitteen rekisteröityessä tapahtuva asiakkaan tunnistus.

Ongelmana on, että käytännössä kaikki rekisteröitymisen aikana ajettava toiminnallisuus on Leshan-kirjastossa. Vaikka kirjasto on avoin, sitä ei haluta kuitenkaan muokata, koska kirjasto on edelleen aktiivisessa kehityksessä ja sen muokkaaminen vaikeuttaisi päivittämistä.

Alkuperäisen *RegisterResource* -luokan määrittely kuitenkin löytyi, mutta senkin käyttöönotto, eli lisäys ajettavaan *LeshanServer* -luokkaan tapahtui kirjaston sisällä.

Luokasta täytyi siis tehdä oma implementaatio, jolla alkuperäinen versio voitaisiin korvata:

```
// Overwrite existing /rd resource
RegisterResource rdResource = new RegisterResource(new
    RegistrationHandler(registrationService, authorizer));

lwServer.getCoapServer().add(rdResource);
```

Juuri korvatussa *RegisterResource* -luokassa on määritelty *handleRegister* -niminen funktio, joka vastaa laitteiden rekisteröitymisen validoinnista ja oikean vastauksen antamisesta rekisteröitymisyritykseen.

Rekisteröitymisyrityksestä riippuen asiakas saa vastaukseksi success 2.xx viestin, tai client error 4.xx viestin. Tilakoodit (engl. status codes) CoAP protokollassa ovat karsittu versio monille tutuista http-tilakoodeista.

Funktioon asetetaan validoinnit imei ja tunnus attribuuteille, ja jos toisessa näistä ilmenee virhe, niin palvelin vastaa laitteelle *Bad Request* -viestillä, joka on http-tilakoodi 400:

```
// Validate imei
imei = additionalParams.get("imei");
if (!RegisterUtils.validImei(imei)) {
  exchange.respond(ResponseCode.BAD_REQUEST);
  return;
}

if(!additionalParams.containsKey("token"))
  additionalParams.put("token", RegisterUtils.getCommonTo-
ken());



// Validate token
token = additionalParams.get("token");
if (!RegisterUtils.validToken(token)) {
  exchange.respond(ResponseCode.BAD_REQUEST);
  return;
}
```

Asiakaskohtainen rekisteröinti on nyt palvelimella käytössä siten, että tunnus vastaa asiakasta ja rekisteröityessä asiakas tunnistetaan tälle kuuluvan tunnuksen kautta.

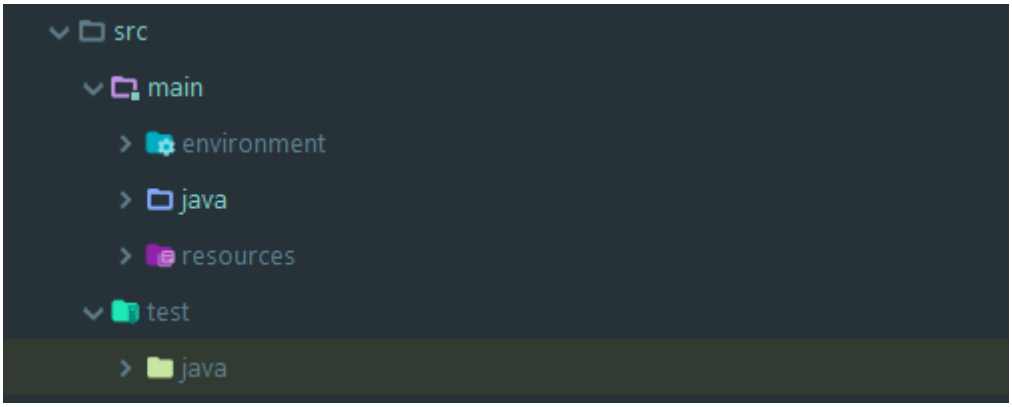
3.2.3 Yksikkötestaus

Yksikkötestaus otettiin mukaan projektiin oppimisen kannalta; on hyvä taito osata käyttää erilaisia testaukseen erikoistuneita sovelluskehyksiä. Lisäksi yksikkötestien tekeminen isommissa sovelluksissa on hyvä käytäntö, joka helpottaa sovelluksen kehittämistä eteenpäin myöhemmin.

Käydään läpi yksi yksikkötesti, joka on tehty aiempaa validointi metodia varten, ja avataan hieman testaukseen käytettävän JUnit 5 -sovelluskehyksen syntaksia.

Kiteytettynä yksikkötesteissä testataan funktiota tai metodia, ja oletetaan tämän toimivan tietyllä tavalla, eli lopputuloksen tulee olla ennalta arvattavissa. Jos testistä saatu lopputulos ei vastaa odotuksia, on metodin toiminnallisuudessa joitakin puutteita.

Yksikkötesteillä voi hyvin pitää huolen siitä, että jos esimerkiksi funktiota muutetaan, samalla ei muuteta sen alkuperäistä toiminnallisuutta. Yksi testien ideoista on pitää ohjelma toimivana, kun sitä kehitetään. Kuvassa 5 näkyy Java-projektin hakemisto rakenne, jossa on mukana yksikkötestit.



Kuva 5. Yleisesti käytetty hakemisto rakenne Java-projekteissa, joissa on mukana yksikkötestit.

Lisäetuna yksikkötestit tuovat metodien nopean testaamisen. Yhden funktion testaaminen yksikkötesteillä on usein nopeampaa, kuin vastaavien toiminnallisuuksien testaaminen manuaalisesti.

Aloitetaan testaus luomalla uusi testi luokka, määritellään testauksessa käytettävä *MockitoRule* sekä alustetaan testattavaa luokkaa:

```
public class RegisterUtilsTest {
    @Rule
    public MockitoRule mockitoRule = MockitoJUnit.rule();

    RegisterUtils registerUtils;
}
```

Tehdään seuraavaksi testien alustus funktio, jossa määritellään jäljitelmä, eli *mock* testattavasta luokasta:

```
@BeforeAll
static void init() {
    registerUtils = mock(RegisterUtils.class);
}
```

Alustus funktio on merkitty *@BeforeAll* -asetuksella, mikä tarkoittaa, että funktio ajetaan aina kerran, ennen kuin mitään luokassa olevia testejä.

Seuraavaksi voidaan testata validointi funktiota. Funktio, jolle annetaan imei pitäisi palauttaa *true* jos annettu imei on kelvollinen ja *false* jos annetun arvon sisällössä on vikaa.

```
@Test
void validImeiPositive (){
    assertTrue(registerUtils.validImei("19480904643408"));
}
```

Yllä olevassa testissä annetaan validointi funktiolle kelvollinen imei ja oletetaan että funktion antama palautus on *true*. Funktio palauttaa oikean arvon, ja näin testi on hyväksytty.

Vastaavalla logiikalla kannattaa testata funktiota erilaisilla arvoilla. Etenkin *null* arvoja ja tyhjiä muuttujia voi koittaa syöttää funktioihin, ja katsoa miten ne reagoivat.

3.3 Tietokanta integraatio

Tietokanta integraatiossa monimutkaisin osuus on toimivan moniasiakkuusratkaisun toteuttaminen. Tavoitteena työssä olisi, että asiakkaan oma tietokanta luotaisiin dynaamisesti samalla kun asiakkaalle tehdään tunnukset asiakaspuolen sovelluksessa.

Toistaiseksi tyydytään kuitenkin staattiseen ratkaisuun, eli tietokannat luodaan itse käyttäjien kanssa. Integraatiota tehdessä tulee kuitenkin pitää mielessä sovelluksen haluttu käyttötarkoitus.

Palvelimen tulee kuitenkin osata hakea oikean asiakkaan tietokanta dynaamisesti. Tietokannan nimi on linkitetty laitteelta tulevalta tunnukseen, ja näin laite voidaan tallettaa oikeaan kantaan.

3.3.1 Tietokannan ajo Dockerissa

Haetaan kaikki pakettivarastosta löytyvät MariaDB Docker vedokset kirjoittamalla seuraava komento:

```
docker search mariadb
```

Tulokseksi saadaan laaja valikoima vedoksia eri palvelun tarjoajilta. Vain yksi näistä on kuitenkin virallinen, ja se on listassa yleensä ensimmäisenä. Haetaan viimeisin virallinen vedos:

```
docker pull mariadb:latest
```

Sovellusta ajaessa täytyy muistaa avata portit ulkomaailmaan, jotta tietokantaan päästäisiin käyttämään. Siirretään kuitenkin tietokanta sovelluksessa käytössä oleva oletus portti 3306 muualle:

```
docker run -p 33060:3306 --name mariadb-server -e
MYSQL_ROOT_PASSWORD=password -d mariadb
```

Tämä tehdään siksi, että jos pilvipalvelussa, jossa sovellus myöhemmin otetaan käyttöön, halutaan ajaa natiivi tietokantaa. Näin kaksi sovellusta ei käytä samaa porttia, eikä päällekkäisyyksiä tule.

Komennossa vaihdettiin Docker portista 3306 tuleva liikenne hostin porttiin 33060. Nimeksi annettiin mariadb-server, sitten ilmoitettiin root-käyttäjän salasana ja lopuksi vielä mitä vedosta käytetään.

Sovellus lähtee *run*-komennon jälkeen automaattisesti käyntiin, ja siihen pääsee käsiksi omalla suosikki tietokanta työkalulla.

3.3.2 Hibernate ja HikariCP konfigurointi

Ennen varsinaisen konfiguroinnin aloittamista lisätään Hibernate, HikariCP ja MariaDB driver *build.gradle* -tiedostoon, josta projekti hakee tarvittavat kirjastot:

```
// Hibernate ORM & HikariCP
compile group: 'org.hibernate', name: 'hibernate-core',
version: '5.2.12.Final'
compile group: 'org.hibernate', name: 'hibernate-hikaricp',
version: '5.2.12.Final'
compile group: 'com.zaxxer', name: 'HikariCP', version:
'2.7.8'

// MariaDB
compile group: 'org.mariadb.jdbc', name: 'mariadb-java-cli-
ent', version:'2.1.2'
```

Lisätään seuraavaksi *hibernate-schema-multitenancy.properties* -tiedosto, minne halutut konfiguraatiot asetetaan:

```
# driver
hibernate.connection.driver_class=org.mariadb.jdbc.Driver
# connection pool path
hibernate.hikari.jdbcUrl=jdbc:mariadb://localhost:33060
hibernate.connection.username=root
hibernate.connection.password=password
hibernate.connection.autocommit=true
# dialect
hibernate.dialect=org.hibernate.dialect.MariaDBDialect
hibernate.show_sql=true
```

```
hibernate.multiTenancy=SCHEMA
hibernate.multi_tenant_connection_provider=fi.unseen.le-
shan.server.persistence.dao.hibernate.multitenancy.Sche-
maMultitenantConnectionProvider
# 20sec connection timeout
hibernate.hikari.connectionTimeout=20000
hibernate.hikari.minimumIdle=5
hibernate.hikari.maximumPoolSize=10
# 5min idle timeout
hibernate.hikari.idleTimeout=300000
hibernate.hikari.leakDetectionThreshold=5000
```

Ajuriksi on määritelty aiemmin lisätty MariaDB-ajuri ja HikariCP-yhteysaltaan polku osoittaa aikaisemmin määriteltyyn porttiin 33060. Konfiguroinnin ollessa valmis, täytyy vielä alustaa Hibernate AppMain-luokassa:

```
// Initialize Hibernate
LOG.trace("Initializing Hibernate");
HibernateUtilities.getInstance().getSessionFactory();
```

3.3.3 Laite DAO implementaatio

Aiemmin todettiin, että projektiin löytyy valmis geneerinen DAO-implementaatio. Implementaatiota ei käydä läpi yksityiskohtaisesti, mutta raportin kannalta keskeisimpiä osia tarkastellaan käyttökohtaisesti.

Luodaan asiakaslaitteiden tallentamista varten uusi DeviceEntity-luokka, joka laajentaa implementaatiossa olevaa Entity-luokkaa. Luokassa Entity on ainoastaan määritelty muuttuja nimeltä *id*. Tämä muuttuja on tärkeä koska relaatiotietokantoja käyttäessä jokaiselle tietokannassa olevalle riville annetaan tauluun nähden uniikki tunnus tai id.

Perittävän Entity-luokan etuna saadaan siis periytyvät kentät. Kaikkien uusien entiteettiluokkien laajentaessa Entity-luokkaa, ei yhteenkään niistä tarvitse erikseen määritellä muuttujaa id.

```
@javax.persistence.Entity
@Table(name="device")
public class DeviceEntity extends Entity {
    private static final long serialVersionUID = 1L;

    @Column(name="imei")
    private long imei;

    // getters & setters omitted for simplicity...
}
```

Tehdään asiakaslaitteita varten tietokantaan taulu nimeltä *device*. Lisätään tauluun DeviceEntity-luokkaa vastaavat kentät, eli id ja imei.

Luodaan seuraavaksi itse DeviceDAO-rajapinta, joka laajentaa geneeristä DAO-rajapintaa. Uuteen rajapintaan alustetaan myös findWithImei-funktio, jolla myöhemmin haetaan laitteita kannasta:

```
public interface DeviceDAO extends GenericDAO<DeviceEntity,
Long> {
    DeviceEntity findWithImei(String imei);
}
```

Lopuksi luodaan HibernateDeviceDAO -luokka, jossa määritellään aiemmin luotu hakufunktio tarkemmin:

```
public class HibernateDeviceDAO extends
HibernateGenericDAOImpl<DeviceEntity, Long> implements
DeviceDAO {

    @Override
    public DeviceEntity findWithImei(String imei) {
        Predicate restriction = this.getCriteriaBuilder()
                    .equal(getRoot().get("imei"), imei);

        return this.findUnique(restriction);
    }
}
```

Seuraavaksi testataan, miten laitteiden haku kannasta toimii aiemmin tehdyn findWithImei-nimisen funktion avulla. Luodaan uusi funktio, jolla on parametrina haettavan laitteen imei.

```
private DeviceEntity getDevice(String imei) {
    DeviceEntity deviceEntity = null;

    try(DeviceDAO dao = DAOFactory.instance(DAOFactory
            .HIBERNATE).getDeviceDAO()) {

        dao.openSession(“master”);
        deviceEntity = dao.findWithImei(
                regAttributes.get("imei"));
    } catch(Exception e) {
        LOG.error("Device fetch from database failed: {}",
                e.getLocalizedMessage());
        return null;
    }

    return deviceEntity;
}
```

Funktio yrittää avata yhteyden master-nimiseen isäntä-tietokantaan, jonka jälkeen laitetta yritetään hakea kannasta. Tietokanta operaatiossa tulee varautua virheisiin, ja siksi suurin osa funktiosta on laitettu *try – catch* -blokin sisään.

Onnistuneessa haussa funktio palauttaa laitteen, ja myöhemmin laite palautetaan jälleen eteenpäin. Jos yhteyden avaamisessa tai haussa ilmenee ongelmia, näytetään virhe viesti ja palautetaan *null*.

3.3.4 Monen asiakkaan tietokanta

Seuraavaksi tuodaan asiakkuus mukaan tämän hetkiseen tietokanta implementaatioon. Myöhemmässä vaiheessa halutaan, että asiakkaiden yleiset tiedot, mukaan lukien heidän tietokantojen tunnisteet löytyvät isäntä-tietokannasta.

Tässä vaiheessa yritetään jälleen testata konseptia ja todistaa sen toimivuus. Riittää siis, että käytettävät tietokannat ns. kovakoodataan siten, että ne ovat haettavissa käyttäen asiakkaan tunnusta.

```
// Initialize HashMap directly with values
private static final Map<String, String> tenantIdentifiers
= createTenants();

private static Map<String, String> createTenants()
{
    Map<String,String> tenantIdentifiers = new HashMap<>();
    tenantIdentifiers
            .put("0123456789abcdefghijklmnopqrstuv",
                 "client");

    return tenantIdentifiers;
}
```

Alustetaan HashMap suoraan arvoilla siten, että tunnisteen voi hakea asiakkaan tunnuksella. Otetaan tunnistusten haku käyttöön lisäämällä se aiemmin tehtyyn funktioon, jossa haettiin laite tietokannasta.

Toteutetaan funktio, joka suorittaa tunnisteen haun:

```
private String getTenantIdentifier(String token) {
    if (TokenUtils.getTenantIdentifiers()
            .get(token) == null) {
        return "master";
    }

    return TokenUtils.getTenantIdentifiers().get(token);
}
```

Funktio kutsuu getTenantIdentifiers -metodia, joka palauttaa aiemmin alustetun HashMapin. Seuraavaksi tarkistetaan, löytyykö annetulle tunnukselle tunnistetta.

Mikäli tunnistetta ei löydy, palautetaan oletus tunniste eli *master*. Tämä toiminnallisuus on vain testikäyttöön, ja se tulee myös tulevaisuudessa vaihtaa. Laitteelta tullaan siis vaatimaan oikeaa tunnistetta, joka täsmää olemassa oleviin tietokantoihin.

Lisätään tunnisteen määrittely laitehaku-funktion alkuun, ja käytetään haettua tunnistetta session avaamiseen tietokannan kanssa:

```
String tenantIdentifier = getTenantIdentifier(token);

[...]

dao.openSession(tenantIdentifier);
```

Nyt kaikki tietokantoihin kohdistuvat toiminnot tapahtuvat niin, että tieto saadaan talteen asiakkaiden omiin kantoihin.

3.4 **Käyttöliittymän toteutus**

Käyttöliittymän toteutuksessa käydään läpi, kuinka luodaan uusi Angular-projekti, otetaan käyttöön MDBootstrap-sovelluskehys sekä määritellään applikaation reititys.

Aikataulun salliessa aletaan lisäksi implementoimaan asiakkuuteen liittyviä elementtejä, kuten kirjautumisnäkymä ja hallintapaneeli asiakkaalle.

3.4.1 Uuden Angular-projektin luominen

Uuden Angular-projektin luominen tapahtuu komentoriviltä, ja komentorivillä käytettävä työkalu on nimeltään node package manager (npm). Toimiakseen npm vaatii Node.js sovelluskehyksen, molemmat tulee ladata ennen uuden Angular-projektin luomista.

Dependenssien latausten valmistuttua, asennetaan angular-cli -niminen npm paketti, joka mahdollistaa Angular-projektien ja uusien komponenttien luomisen komentoriviltä. Asennus tapahtuu komennolla:

```
npm install -g @angular/cli
```

Komennossa oleva *-g* asetus tarkoittaa, että sovellus asennetaan globaalisti eli sitä voi käyttää missä sijainnissa tahansa, eikä tarvitse olla samassa kansiossa kuin itse ohjelma on.

Seuraavaksi luodaan uusi Angular-projekti komennolla:

```
ng new webapp --style=scss --routing
```

Projektin tyylit määritellään *--style=scss* asetuksella, joka tarkoittaa, että projektissa ei ole käytössä perinteinen css vaan sass. Komennon lopussa oleva *--routing* asetus tekee valmiin reitityspohjan projektille.

Angular-projekti on nyt valmis käynnistettäväksi. Siirrytään projektin hakemistoon ja käynnistetään sovellus:

```
cd webapp
ng serve --open
```

Komennossa oleva *--open* asetus avaa selaimen automaattisesti osoitteeseen *localhost:4200*, mikä on applikaation oletus portti paikallisella koneella. Kuvassa 6 on Angular-applikaation ensimmäistä kertaa käynnistyessä näkyvä teksti ja logo.

Kuva 6. Näkymä Angular-applikaation käynnistyessä ensimmäistä kertaa.

Applikaatiota voi kehittää ilman että sivua tarvitsee päivittää, kunhan pitää *ng serve* -komennon päällä komentorivillä. Angular-projektien mukana tuleva WebPack-kirjasto, joka tuntuu tänä päivänä olevan standardi web-applikaatioita kehittäessä, mahdollistaa tämän tyyppisen kehityksen.

3.4.2 MDBootstrapin käyttöönotto

Ladataan projektin kansioon MDBootstrap-sovelluskehys kirjoittamalla seuraava npm komento:

```
npm install angular-bootstrap-md --save
```

Komennossa oleva *--save* asetus tallentaa ladattavan sovelluskehyksen projektiin pysyväksi dependenssiksi.

Lisätään *app.module.ts* -tiedostoon seuraavat rivit:

```
import { NgModule, NO_ERRORS_SCHEMA } from '@angular/core';
import { MDBBootstrapModule } from 'angular-bootstrap-md';

@NgModule({
    imports: [ MDBBootstrapModule.forRoot() ],
    schemas: [ NO_ERRORS_SCHEMA ]
});
```

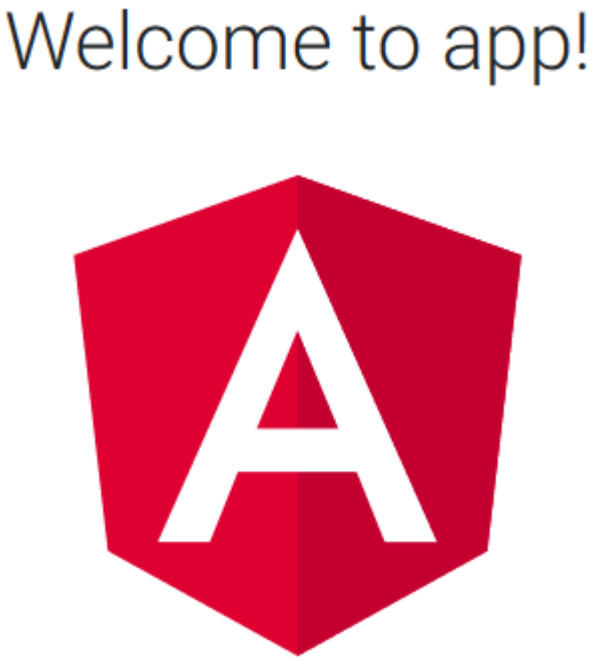
Seuraavaksi lisätään *angular.json* -tiedostoon rivit:

```
"styles": [
    "../node_modules/font-awesome/scss/font-awesome.scss",
    "../node_modules/angular-bootstrap-md/scss/boot-
strap/bootstrap.scss",
    "../node_modules/angular-bootstrap-md/scss/mdb-
free.scss",
    "./styles.scss"
],
"scripts": [
  "../node_modules/chart.js/dist/Chart.js",
  "../node_modules/hammerjs/hammer.min.js"
],
```

Lopuksi asennetaan vielä ulkopuoliset kirjastot, joita MDBootstrap tarvitsee toimiakseen:

```
npm install --save chart.js@2.5.0 font-awesome hammerjs
```

MDBootstrap on nyt asennettu ja valmis käytettäväksi projektissa. Asennuksen voi vielä varmistaa avaamalla applikaatio selaimessa. MDBootstrap tuo mukanaan Roboto -nimisen fontin, joka on oletuksena käytössä kaikkialla projektissa, kuten kuvassa 7 näkyy.



Kuva 7. Näkymä Angular-applikaatio MDBootstrapin kanssa.

Lisätään sivulle navigaatiopalkki, jossa olevat linkit reititetään myöhemmin. Sovelluksessa käytettävä navigaatiopalkki löytyy MDBootstrapin sivuilta.

```
<mdb-navbar SideClass="navbar navbar-expand-lg navbar-dark
elegant-color ie-nav" [containerInside]="true">
  <links>
      <ul class="navbar-nav ml-auto">
          <li class="nav-item waves-light" mdbWavesEffect>
            <a class="nav-link">Client</a>
          </li>
          <li class="nav-item waves-light" mdbWavesEffect>
            <a class="nav-link">Login</a>
          </li>
      </ul>
  </links>
</mdb-navbar>
```

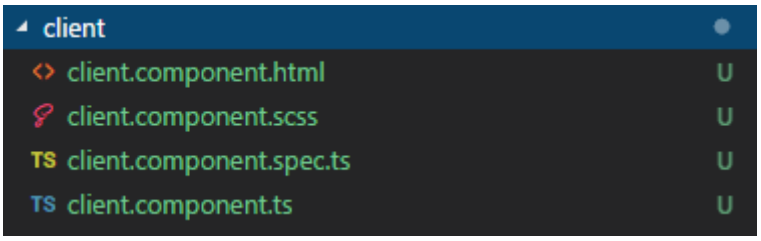
Listassa olevissa osioissa on oletuksena *mdbWavesEffect*, joka lisää animaation painettaviin nappeihin. Linkit *Client* ja *Login* -sivuille löytyvät, mutta itse komponentit ja niiden reititys puuttuu.

3.4.3 Sivujen reititys

Luodaan ensimmäisenä reititettävät *Client* ja *Login* -komponentit käyttämällä angular-cli -työkalua:

```
ng generate component client
ng generate component login
```

Komponenttien lisäksi työkalulla voidaan myös luoda palveluita (engl. service) tai vaikka moduuleita. Projektissa käytettävät tyylit otetaan myös huomioon. Kuvassa 8 näkyy luodun komponentin hakemisto rakenne.



Kuva 8. Uuden komponentin luomisen mukana tulevat tiedostot.

Aloitetaan reititys tuomalla ne komponentit, jotka halutaan reitittää, tiedostoon *app-routing.module.ts*.

```
import { LoginComponent } from './login/login.component'
import { ClientComponent } from './client/client.component'
```

Seuraavaksi lisätään samaan tiedostoon reittimääritykset:

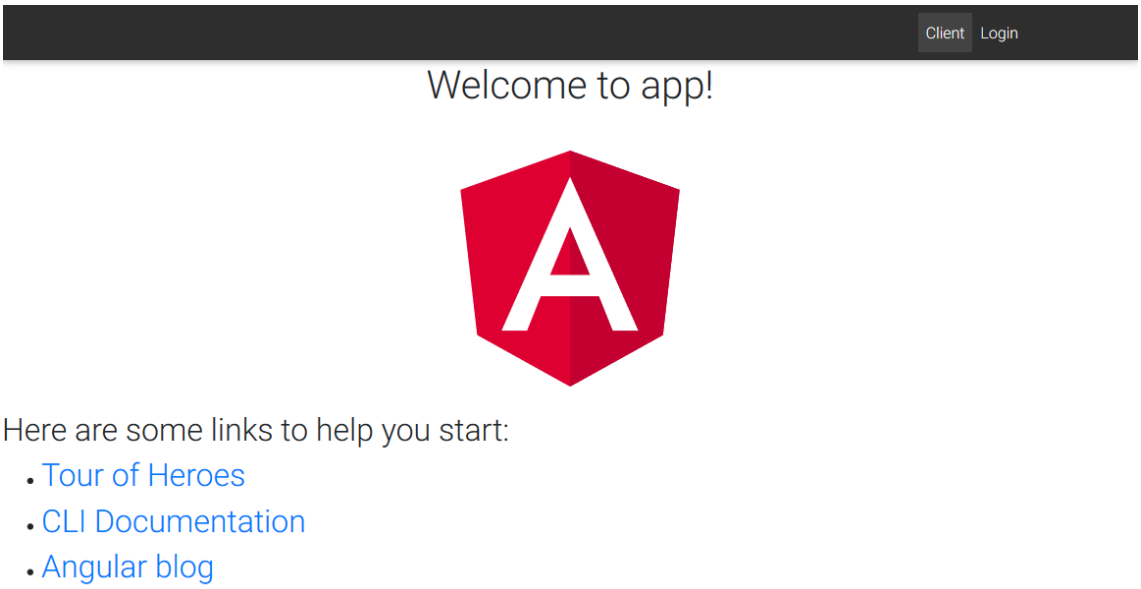
```
const appRoutes: Routes = [
  {
    path: '',
    component: ClientComponent
  }, {
    path: 'login',
    component: LoginComponent
  }
];
```

Reiteissä on polku ja komponentti mihin annetusta polusta reititys tapahtuu. *ClientComponent* on siis oletus reittinä sovelluksessa, ja */login* -reitillä päästään näkymään, jossa on *LoginComponent*.

Applikaation reitittimelle täytyy vielä kertoa, että mistä annettuihin polkuihin päästään. Lisätään navigaatiopalkkiin reitit ja käytetään reitteihin määriteltyjä arvoja:

```
<a class="nav-link" routerLink="">Clients</a>
<a class="nav-link" routerLink="login">Login</a>
```

Reititys applikaatiossa on nyt toiminnassa ja helposti laajennettavissa jos navigaatiopalkkiin halutaan lisätä linkkejä. Kuvassa 9 näkyy toteutetun applikaation ulkoasu.



Kuva 9. Applikaatio navigaatiopalkin kanssa.

3.5 Projektin jatkokehitys

Projektissa löytyy vielä paljon tehtävää jatkokehityksessä, vaikkakin toimiva MVP palvelinpuolesta on nyt olemassa. Eniten työstämistä tällä hetkellä vaatii asiakaspuolelta löytyvä käyttöliittymä, josta puuttuu kaikki yleiset asiakkuutta koskevat toiminnot.

Ensimmäisenä käyttöliittymässä tulisi mahdollistaa uuden asiakkaan luonti ja samalla uudelle asiakkaalle generoitaisiin tunnus, tunniste sekä tietokanta luodun tunnisteen pohjalta.

Yleiset toiminnot kuten kirjautumissivu asiakkaalle sekä asiakkaille näkyvä hallintapaneeli ovat korkealla tärkeysjärjestyksessä. Myös projektin omistajan näkökulmasta oleva hallintapaneeli, josta hallita asiakkaita, tulee pakolliseksi silloin, kun asiakkaita alkaa olla paljon.

Palvelinpuolelta halutaan poistaa kovakoodattuna olevat asiakkaiden tunnukset ja tunnisteen, sekä siirtää nämä isäntä-tietokantaan, josta ne voidaan tarvittaessa hakea. Myös testikäyttöön luotu toiminnallisuus, joka mahdollistaa laitteiden rekisteröitymisen ilman tunnusta, aiotaan poistaa käytöstä.

4 JOHTOPÄÄTÖKSET JA POHDINTA

Opinnäytetyötä aloittaessa, oli selkeä käsite siitä mitä pitää tehdä. Projektin suuruuden ja lukuisten uusien teknologioiden takia oli aika ajoin vaikea hahmottaa mikä osuus toiminnallisuudesta tulisi toteuttaa seuraavaksi ja miten.

Tekemistä kuitenkin helpotti suurten kokonaisuuksien pilkkominen pienemmiksi, yksittäisiksi tehtäviksi. Projektin vaikeuttamisen lisäksi, uudet teknologiat myös nostivat mielenkiintoa projektia kohtaan. Sovelluskehyksiä käyttäessä ne tulee myös puoliväkisin opittua.

Projektin suunnitelmasta suurin osa saatiin toteutettua. Asiakkaan tunnistuksessa palvelinpuolella ei ollut mitään ongelmia, ja kehitys siellä eteni virtaviivaisesti. Asiakaslaitteiden tallettamisessa ilmeni haasteita, etenkin olemassa olevan tietokanta kokonaisuuden integroinnin kanssa.

Asiakaspuolen sovellukseen tehty työ taas jäi hyvin keskeneräiseksi. Kehityksessä käytettävä MDB-sovelluskehys saatiin integroitua, mutta mitään asiakkuuteen liittyvää toiminnallisuutta ei ehditty toteuttaa.

Tuli opittua, että ohjelmistoprojektin kehitykseen tarvittavaa aikaa on todella vaikea arvioida. Joskus aikaisemmin on tullut luettua, että kun arvio kehitykseen tarvittavasta ajasta on tehty, se kerrotaan vielä kahdella. Nyt tietää miksi tätä suositellaan; kehityskaaret ovat usein hyvin arvaamattomia ja ne vain monimutkaistuvat matkan varrella.

LÄHTEET

Brett Wooldridge (n.d.). HikariCP. Haettu 28.5.2018 osoitteesta https://github.com/brettwooldridge/HikariCP/

DB-Engines (2018). DB-Engines Ranking. Haettu 28.5.2018 osoitteesta https://db-engines.com/en/ranking/

Docker (n.d.). What is Docker. Overview. Haettu 28.5.2018 osoitteesta https://www.docker.com/what-docker/

Eclipse Foundation (2015). Leshan. OMA Lightweight M2M server and client in Java. Haettu 28.5.2018 osoitteesta https://www.eclipse.org/leshan/

Gartner (n.d.). Multitenancy. Haettu 28.5.2018 osoitteesta https://www.gartner.com/it-glossary/multitenancy/

Hibernate (n.d.). Hibernate ORM. Haettu 28.5.2018 osoitteesta http://hibernate.org/orm/

IBM (2016). Don't repeat the DAO! Haettu 28.5.2018 osoitteesta https://www.ibm.com/developerworks/java/library/j-genericdao/index.html/

IPSO Alliance (2018). IP for Smart Objects – IPSO Objects. Creating new Objects. Haettu 28.5.2018 osoitteesta https://github.com/IPSO-Alliance/pub/

Jersey (2018). About. Haettu 19.06.2018 osoitteesta https://jersey.github.io/

MariaDB Foundation (2018). About MariaDB. Haettu 28.5.2018 osoitteesta https://mariadb.org/about/

MDBootstrap (2018). About MDB. Haettu 28.5.2018 osoitteesta https://mdbootstrap.com/material-design-for-bootstrap/

Microsoft (2018). Multi-tenant SaaS database tenancy patterns. Haettu 28.5.2018 osoitteesta https://docs.microsoft.com/en-us/azure/sql-database/saas-tenancy-app-design-patterns/

Node Package Manager (2018). The State of JavaScript Frameworks, 2017. Haettu 28.5.2018 osoitteesta https://www.npmjs.com/npm/state-of-javascript-frameworks-2017-part-1/

Open Mobile Alliance (2017). Introduction to LightweightM2M. What is LwM2M. Haettu 28.5.2018 osoitteesta https://wiki.openmobilealliance.org/display/TOOL/What+is+LwM2M/

Open Mobile Alliance (2018). OMA LWM2M Management Object Editor. Haettu 28.5.2018 osoitteesta http://devtoolkit.openmobilealliance.org/OEditor/Default/

Oracle (n.d.). Core J2EE Patterns – Data Access Object. Haettu 28.5.2018 osoitteesta http://www.oracle.com/technetwork/java/dataaccessobject-138824.html/

Percona (2016) MySQL Data at Rest Encryption. Data at Rest Encryption: Database-Level Options. Haettu 28.5.2018 osoitteesta https://www.percona.com/blog/2016/04/08/mysql-data-at-rest-encryption/

Rozentals, N. (2017). *Mastering TypeScript. Second Edition.* Packt Publishing Ltd.