jamk.fi

IoT-datan tiedonkulku, varastointi ja esittäminen

Jere-Joonas Valtanen

Opinnäytetyö
Tammikuu 2019
Tekniikan ja liikenteen ala
Insinööri (AMK), Tieto- ja viestintätekniikan tutkinto-ohjelma
Ohjelmistotekniikka

Jyväskylän ammattikorkeakoulu
JAMK University of Applied Sciences

jamk.fi

Kuvailulehti

Tekijä(t)	Julkaisun laji	Päivämäärä
Valtanen, Jere-Joonas	Opinnäytetyö, AMK	Tammikuu 2019
	Sivumäärä 63	Julkaisun kieli Suomi
		Verkkojulkaisulupa myönnetty: x

Työn nimi
IoT-datan tiedonkulku, varastointi ja esittäminen

Tutkinto-ohjelma
Insinööri (AMK), tieto- ja viestintätekniikka

Työn ohjaaja(t)
Mika Rantonen, Jouni Huotari

Toimeksiantaja(t)
Digia Finland Oy

Tiivistelmä

Opinnäytetyössä suunniteltiin ja toteutettiin IoT-järjestelmä, joka kerää, lähettää, vastaanottaa, varastoi sekä esittää dataa. Kaikki järjestelmän osat toteutettiin erillisinä komponentteina, jotka suoritettiin omissa ympäristöissänsä Docker-konttien avulla. Työn tarkoituksena oli tuottaa Digia Oy:lle pohja mahdollisia tulevia IoT-projekteja varten. Lisäksi toimeksiantaja määritteli työssä käytettävät teknologiat, joiden toimivuus ja luotettavuus testattiin. Työssä tärkeää ei ollut se mitä dataa kerätään, vaan pikemminkin se miten sitä kerätään. Kerättävän datan lähde tulisi pystyä tarvittaessa pienellä vaivalla vaihtamaan.

Työssä kerättiin erilaisia ilmansaastetietoja ympäri maailmaa OpenAQ-palvelun avoimesta REST API -rajapinnasta. Dataa tuli kerätä suuria määriä, jotta järjestelmän kokema kuorma oli riittävän realistinen IoT-järjestelmälle. Dataa lähettävä ja vastaanottava applikaatio toteutettiin Java-ohjelmointikielellä sekä Spring Frameworkilla. Viestit lähetettiin IoT-järjestelmissä suositulla MQTT-protokollalla. Viestit kerättiin jonoihin sekä välitettiin oikeille vastaanottajille RabbitMQ-viestilähetin avulla. Data varastoitiin InfluxDB-aikasarjatietokantaan. Lopuksi data visualisoitiin loppukäyttäjälle Grafana-analytiikkatyökalulla.

Toteutettiin vaatimusten mukainen toimiva IoT-järjestelmä, jonka perusteella saatiin myös todennettua teknologioiden toimivuus. Kaikki työssä käytetyt teknologiat soveltuivat tehtäviinsä erinomaisesti. Työssä kerättiin InfluxDB-tietokantaan 1 958 641 datapistettä, jotka sisälsivät ilmansaastetietoja ympäri maailmaa. Suorituskykynsä puolesta aikasarjatietokanta oli hyvä valinta IoT-datan varastointiin. Data visualisoitiin kartalle sekä graafeihin Grafanassa. Lisäksi käyttäjä pystyi itse määrittelemään näytettävän saastetyypin, maan, sekä aikavälin. Yksittäisten kaupunkien saastetietoja pystyttiin tutkimaan omassa näkymässään tarkemmin.

Avainsanat (asiasanat)
Esineiden internet, Grafana, InfluxDB, IoT, MQTT, RabbitMQ, Spring

Muut tiedot
Työn GitLab-repositorio https://gitlab.com/Jere_Valtanen/iot-stack

jamk.fi

Description

Author(s)	Type of publication	Date
Valtanen, Jere-Joonas	Bachelor's thesis	January 2019
	Number of pages 63	Language of publication: Finnish
		Permission for web publication: x

Title of publication

IoT data transfer, storage and presentation

Degree programme

Information and communication technology

Supervisor(s)

Mika Rantonen, Jouni Huotari

Assigned by

Digia Finland Oy

Description

The objective of this thesis was to design and implement an IoT system that collects, sends, receives, stores and presents data. All the system components were implemented as separate components. The components were restricted to their own environments using Docker containers. The purpose of the thesis was to produce a basis for possible future IoT projects carried out by Digia Oy. The client also specified the technologies used in the project. The functionality and reliability of these technologies was tested. The collected data itself was not important in the project. The project focused on how it was collected instead. The source of the collected data had to be easily replaceable with something else.

Different types of air pollution data were collected from around the world using the OpenAQ service REST API. Large amounts of data had to be collected to create a realistic stress test for the IoT system. The applications sending and receiving data were created using Java and Spring Framework. Messages were sent using the MQTT-message protocol. The messages were gathered into queues and routed to the correct recipient using the RabbitMQ message broker. The data was stored into an InfluxDB time-series database. Finally, the data was visualized with the Grafana analytics tool.

An IoT system matching the project requirements was created and the functionality of the technologies was confirmed. All the technologies worked well in an IoT system. 1 958 641 data points containing air pollution data were saved into the InfluxDB database. Performance-wise a time-series database was an excellent choice for data storage in an IoT project. The data points were visualized on maps and graphs in Grafana. The end user could choose the displayed pollution type, country and timespan. The pollution data of single cities could be examined in detail in a separate dashboard.

Keywords (subjects)

Grafana, InfluxDB, IoT, Internet of things, MQTT, RabbitMQ, Spring

Miscellanous

Project GitLab repository https://gitlab.com/Jere_Valtanen/iot-stack

Sisältö

Kuviot

Lyhenteet

AMQP	Advanced Message Queuing Protocol
CO	Hiilimonoksidi
HTTP	Hypertext Transfer Protocol
IDE	Integrated development environment
IoT	Internet of things
J2EE	Java 2 platform, enterprise edition
JAR	Java archive
JavaEE	Java platform, enterprise edition
JDK	Java Development Kit
JSON	JavaScript object notation
LTS	Long-term-support
M2M	Machine to machine
MQTT	Message Queuing Telemetry Transport
MVC	Model-view-controller
NO2	Typpioksidi
O3	Otsoni
PM2.5	Particulate Matter of 2.5
PM10	Particulate Matter of 10
REST	Representational state transfer
SO2	Rikkioksidi

TCP	Transmission control protocol
TLS	Transport layer security
UML	Unified Modeling Language
YAML	YAML ain’t markup language

1 Opinnäytetyön lähtökohdat

1.1 Toimeksiantaja

Opinnäytetyön toimeksiantajana toimi Digia Finland Oy. Digia on IT palveluyritys, joka työllistää yli 1000 henkilöä ja tarjoaa asiakkailleen laajaa toimialaosaamista. Toimipisteitä Digialla on Suomessa ja Ruotsissa. Digia keskittyy neljälle palvelualueelle. Nämä palvelualueet ovat digitaaliset palvelut, integraatio ja tiedon hyödyntäminen, toimialaratkaisut sekä finanssiliiketoiminta. (Digia yrityksenä n.d.)

Toimialaosaamista Digialla on erityisesti kaupan, logistiikan, teollisuuden ja telecom-alalta, julkiselta sektorilta sekä pankki- ja vakuutusalalta. Digia on listattuna pörssiin Nasqad Helsingissä ja sen liikevaihto vuonna 2017 oli 94,5 miljoonaa euroa. (Digia yrityksenä n.d.)

1.2 Toimeksianto

Työn tarkoituksena oli toteuttaa toimeksiantajalle järjestelmä, jolla voidaan vastaanottaa suuria määriä IoT-dataa (ks. luku 3) MQTT-protokollan yli (ks. luku 4.3). Lisäksi data varastoitiin asianmukaiseen tietokantaan ja esitettiin muokattavissa olevalla käyttöliittymällä loppukäyttäjälle. Järjestelmän osista luotiin Docker-kontit (ks. luku 4.7), jolloin ne olivat helposti siirrettävissä esimerkiksi pilvipalveluun.

Työssä tärkeintä oli datan käsittely alusta loppuun. Se, mitä dataa järjestelmään tuotettiin, jätettiin toteuttajan itsensä päätettäväksi. Projektissa käytettiin täysin avoimen lähdekoodin teknologioita ja ne sovittiin toimeksiantajan kanssa ennen projektin aloittamista. Järjestelmän ei tarvinnut olla tuotantovalmis, mutta työn valmistuttua arvioitiin myös järjestelmän mahdollista tuotantovalmiutta.

1.3 Tavoitteet

Opinnäytetyön päätavoitteena oli tarjota kokonaiskuva IoT-projektin toteuttamisesta ja luoda pohja mahdolliselle tulevalle IoT-toteutukselle. Työn teknologiat valittiin toi-

meksiantajan toimesta ja tavoitteena oli kartoittaa niiden toimivuus ja kypsyys toimialalla. Samalla voitiin kartoittaa kuinka kypsiä avoimen lähdekoodin teknologiat ovat tuotantokäyttöön. Tehdyn projektin ansiosta mahdollista asiakasprojektia aloittaessa ei tarvitsisi käyttää aikaa pitkään teknologiasuunniteluun, vaan voitaisiin käyttää projektia pohjana suunnittelussa.

Esineiden internet on aihe, josta kuulee puhuttavan IT-alalla jatkuvasti. Esimerkiksi talouslehti Forbes ennusti verkkoon liitettyjen laitteiden määrän nousevan 11 miljardiin vuoden 2018 aikana, pois lukien tietokoneet ja puhelimet (Marr 2018). Osaaminen toimialalla vaikuttaa olevan erittäin haluttua ympäri maailmaa. Henkilökohtaisesti tämä projekti oli erinomainen mahdollisuus saavuttaa ymmärrys siitä, mitä osia IoT-projektiin kuuluu ja miten ne käytännössä toteutetaan. Samalla kehitettiin teknisiä kykyjä ja opittiin uusia mielenkiintoisia teknologioita. Projektin tulisi parantaa yleisellä tasolla suoriutumista erinäisistä tehtävistä työelämässä.

Selkeät työskentelyprosessit ovat IT-alalla tärkeitä. Pyrittiin siirtymään opiskelijaprojektien kaoottisista kehitystavoista järjestelmällisiin ja selkeisiin prosesseihin kehitystyössä. Pyrittiin etenkin projektin lähdekoodin versionhallinnan ammattimaiseen toteutukseen siitä huolimatta, että projekti toteutettiin yhden henkilön voimin.

2 Tutkimusmenetelmät

2.1 Kvantitatiivinen tutkimus

Kvantitatiivinen eli määrällinen tutkimus käsittelee yleisesti lukumääriä ja prosenttiosuuksia. Tutkimusaineiston keräyksessä hyödynnetään standardoituja tutkimuslomakkeita, joihin on valmiit vastausvaihtoehdot. Tällä tavalla vastaukset on helppo jaotella selkeisiin prosenttiosuuksiin. Tutkimus vaatii riittävän suuren otannan ollakseen pätevä. Tutkimuksessa selvinneitä asioita kuvataan yleensä numeerisesti ja selvitetään eri tekijöiden riippuvuuksia sekä mahdollisia tapahtuneita muutoksia. Kvantitatiivissa tutkimuksessa ei niinkään pyritä ymmärtämään sitä miksi jokin tapahtui vaan enemmänkin sitä, mitä tapahtui. (Heikkilä 2014, 15.)

2.2 Kvalitatiivinen tutkimus

Kvalitatiivisessa eli laadullisessa tutkimuksessa pyritään pienen tutkittavan määrän kautta ymmärtämään tutkittavaa kohdetta sekä sen käyttäytymistapoja ja miksi tiettyjä päätöksiä tehtiin. Numeerinen data ei ole kvalitatiivisessa tutkimuksessa hyödyllistä. Kvalitatiivisella tutkimuksella voidaan erinomaisesti kehittää esimerkiksi yrityksen toimintaa. Tiedot kerätään verbaalisesti haastattelujen kautta kahden tai ryhmässä. Kvalitatiivisen tutkimuksen tavoitteena on selvittää, miksi jotain tapahtuu. Se mitä tapahtuu, ei ole niinkään merkityksellistä. (Heikkilä 2014, 15.)

2.3 Valittu tutkimusmenetelmä

Opinnäytetyön tutkimusmetodiksi valittiin kvantitatiivinen tutkimus. Tutkimuksessa todettiin teknologiavalintojen onnistumisia vertailemalla teknologioita toisiinsa sekä varsinaisen projektin onnistumista kerätyllä datalla. Tulokset olivat sellaisia, joita on helppo käsitellä taulukkoina esimerkiksi numeerisessa muodossa. Tutkimuksessa ei järjestetty kyselyjä, sillä työ ei laadultaan siihen soveltunut. Sen sijaan tutkimuksessa perehdyttiin olemassa olevaan dataan ja tuotettiin omaa.

3 Internet of things

3.1 Mitä on IoT?

Internet on tavallisesti tunnetussa muodossaan tietokoneiden yhdistämistä toisiinsa globaalisti. Esineiden internetissä yhteyksiä laajennetaan tietokoneista myös kaikkiin muihin mahdollisiin laitteisiin. Termin ”Internet of Things” on keksinyt brittiläinen teknologiapioneeri Kevin Ashton jo vuonna 1999. Perustavoitteena oli kerätä ja jakaa dataa automatisoidusti ja laaja-alaisesti. (Sangaiah, Thangavelu & Sundaram 2018, 3.)

Yhteinen tekijä IoT-laitteiden välillä on tyypillisesti se, että ne sisältävät jonkinlaisen sensorin tai käyttölaitteen. Sensorit voivat mitata esimerkiksi lämpötilaa tai liikettä, ja käyttölaitteina voivat toimia esimerkiksi näytöt tai moottorit. Laitteet ovat usein

osa suurempaa kokonaisuutta, jossa useampi samanlainen laite on kytkettynä. Laitteessa tulee olla myös riittävästi laitteistoa tiedon käsittelyyn ja lähettämiseen/vastaanottamiseen. Esimerkiksi Bluetooth on yksi suosittu lyhyen matkan lähetykseen käytetty protokolla. (Sangaiah, Thangavelu & Sundaram 2018, 3-4.)

Mitä IoT-datalle sitten oikein tehdään? Dataa kerätään kaikilta yhdistetyiltä laitteilta ja sitä analysoidaan eri tavoilla. Dataa analysoimalla saavutetaan laaja tietoperusta, jonka perusteella voidaan suorittaa toimia esimerkiksi tuotannon optimointiin. Yleisesti IoT-laitteet toimivat täysin itsenäisesti ja automatisoidusti keskittyen vain datan lähettämiseen ja vastaanottamiseen. (Sangaiah, Thangavelu & Sundaram 2018, 3-4.)

3.2 Käyttökohteet ja saavutettavat hyödyt

Vaikuttaa siltä, että esineiden internetillä on potentiaalia ja käyttökohteita miltei rajattomasti. Käyttökohteita löytyy runsaasti sekä yksityiskäyttäjille että suurille yrityksille. Pyritään seuraavaksi tunnistamaan joitakin tyypillisiä käyttökohteita ja mitä hyötyjä niillä saavutetaan.

3.2.1 Kuluttajat

Normaalin kuluttajan on usein vaikea tunnistaa IoT-laitetta johtuen sen automatisoidusta ja huomaamattomasta toiminnasta. Tyypillistä IoT-laitetta ajatellessa mieleen juolahtaa helposti esimerkiksi älyjääkaappi, jonka sisältöä voi tarkastella etänä puhelimen avulla. Laajemmassa käytössä on kuitenkin esimerkiksi älykello. Hyvin varusteltu älykello sisältää useita sensoreita ja on yhteydessä internetiin. Kuluttaja voi kerätä esimerkiksi syketietojansa fyysisen harjoittelun aikana ja analysoida kehitystään ajan kuluessa.

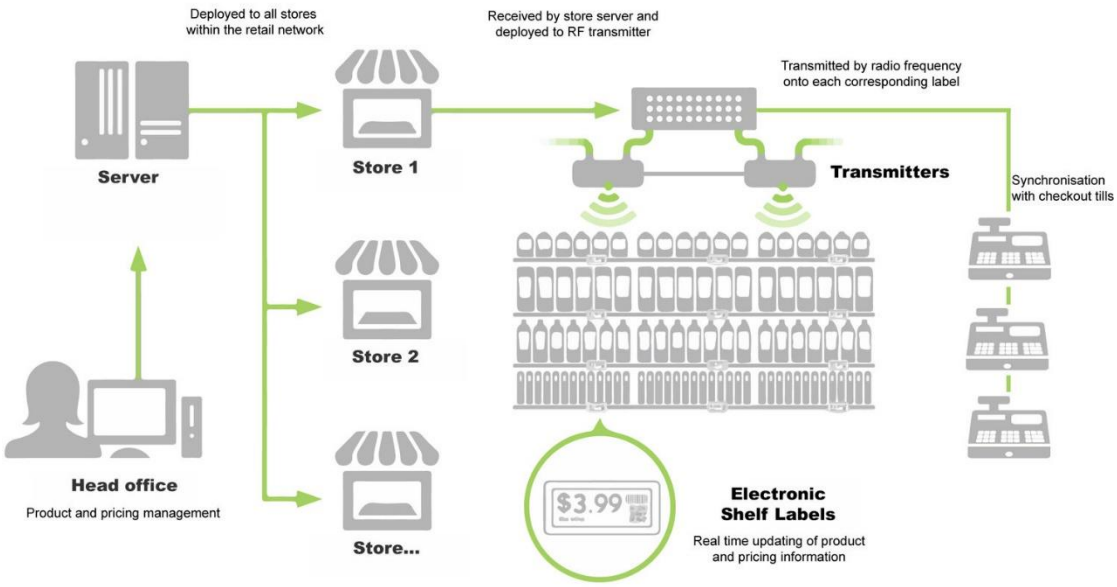
Tällä hetkellä ehkäpä näkyvin suuresti IoT-dataa hyödyntävä kuluttajatuotteiden valmistaja on Tesla. Kaikissa Tesla-merkkisissä sähköautoissa on internet-yhteys ja suuri määrä sensoreita, jotka lähettävät Teslan datakeskuksiin tietoa auton toiminnasta ja tapahtumista sen ympärillä. Teslan perustajan Elon Muskin sanoin ”Koko Tesla laivue toimii verkkona. Kun yksi auto oppii jotain, kaikki muutkin oppivat. Olemme tässä muita autojen valmistajia edellä.” (Fehrenbacher 2015.)

Kuluttajalle näkyvin Teslan IoT-ominaisuus on ”over the air” -päivitykset. Käytännössä auto vastaanottaa järjestelmäpäivityksiä samantapaisesti kuin esimerkiksi älypuhelin. Näitä päivityksiä hyödyntämällä Tesla pystyy jatkuvasti parantamaan auton toimivuutta ja reagoimaan laajoihin ongelmatilanteisiin nopeasti ja laajasti, ehkä jopa ilman takaisinkutsua. Tällaisella ohjelmistopäivityksellä Tesla korjasi esimerkiksi 29 222 ajoneuvon vian latauspistokkeissa, joiden oli havaittu aiheuttavan tulipaloja. Lisäksi Tesla on julkaissut useita päivityksiä parantaakseen auton suorituskykyä ja turvallisuutta, esimerkiksi muuttamalla auton jousitusasetuksia. (Brisbourne 2014.)

3.2.2 Yritykset

Kuluttajia tärkeämmässä asemassa IoT-teknologioiden käyttäjänä lienevät yritykset. Teollisuudessa IoT-laitteilla on mahdollista saavuttaa suuret hyödyt tuotannon tehokkuuden optimoinnissa. Tuotantoprosessissa tehtaista kerätyllä datalla voidaan seurata ja optimoida esimerkiksi energian kulutusta ja kuorman tasoa. Liittämällä IoT-järjestelmä toiminnanohjausjärjestelmään voidaan myös reaaliaikaisesti reagoida tilanteisiin, jotka vaativat huomiota. (Sangaiah, Thangavelu & Sundaram 2018, 4.)

Myös ei-teollisuuteen liittyvät yritykset voivat hyötyä esineiden internetistä suuresti. Jyväskylän seudun Citymarket-myymälöissä perinteiset hintalaput ovat korvaantuneet elektronisilla hintalapuilla. Näillä hintalapuilla tuotteiden hinnat pysyvät aina automaattisesti ajan tasalla. Hintatiedot noudetaan langattomasti kaupan taustajärjestelmästä (ks. kuvio 1).



Kuvio 1. Elektronisen hintalapun toimintakaavio (esLabels: How it works n.d.)

3.2.3 Nyt ja tulevaisuudessa

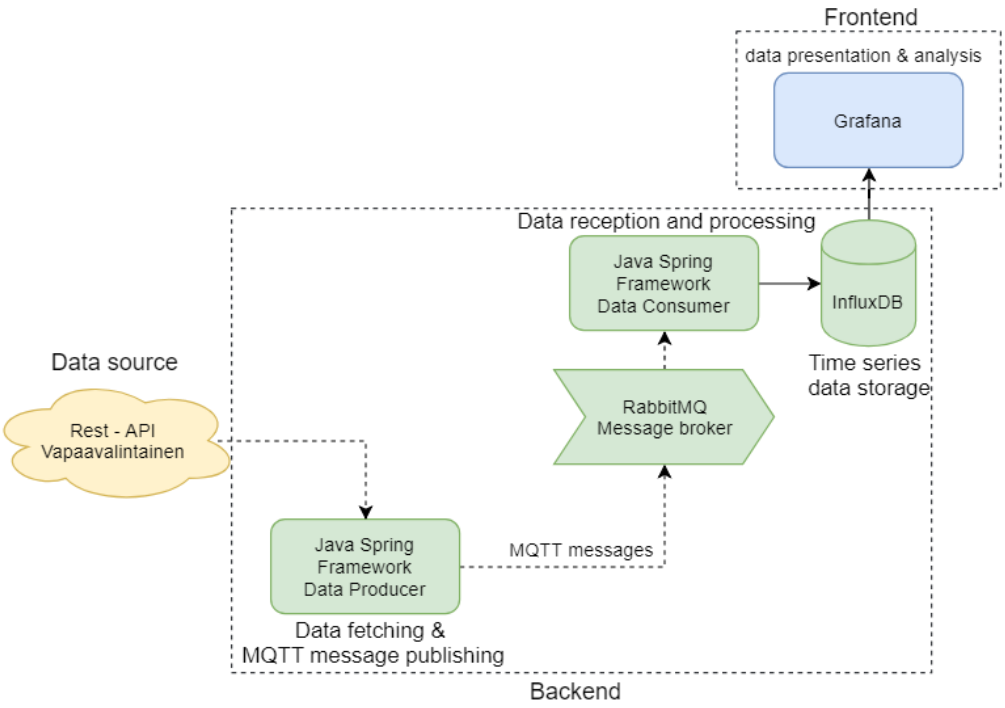
Tutkimuksen aikana kävi selväksi, että mahdollisuuksia hyödyntää esineiden internetiä yritysten toiminnassa on valtavasti. Jatkuvasti ilmestyy uusia innovatiivisia käyttötapoja, joiden avulla yritykset pyrkivät maksimoimaan tehokkuutensa ja kuluttajat saavat lisähyötyä ostamistaan tuotteista. Tutkimuksen perusteella on selvää, miksi esineiden internet on juuri nyt niin kuuma puheenaihe. Tilanne on myös IT-alan ammattilaisia ajatellen positiivinen, sillä työpaikkoja IoT-projekteihin on tarjolla runsaasti nyt ja tulevaisuudessa.

4 Teknologiavalinnat ja niiden toiminta

4.1 Yleistä

Projektin suorittamiseksi täytyi kehittäjän hallita laaja skaala teknologioita aina taustajärjestelmistä käyttöliittymään. Erilaisia teknologioita projektin toteuttamiseksi on saatavilla suuri määrä, mutta yhdessä toimeksiantajan kanssa päädyttiin tekemään valmiiksi selkeät teknologialinjaukset.

Kuviossa 2 on esillä järjestelmäkaavio, joka kuvaa järjestelmän osia ja niiden liittymistä toisiinsa. Projektin teknologiavalintoja voidaan pitää varmoina valintoina, sillä ne ovat maailmanlaajuisesti kehittäjien aktiivisessa käytössä ja sitä kautta todettu toimiviksi. Tässä luvussa käsitellään jokainen teknologiavalinta ja niiden käyttötarkoitus projektissa.



Kuvio 2. Toteutuksen järjestelmäkaavio

4.2 Spring

Spring on Java-pohjainen tuoteperhe, joka on julkaistu vuonna 2003 vastauksena J2EE-spesifikaatioiden monimutkaisuuteen. Spring ei kuitenkaan ole JavaEE:n kilpailija, vaan pikemminkin täydentää sitä. Spring perustuu täysin avoimeen lähdekoodiin, ja sillä on suuri aktiivinen yhteisö käyttäjiä. Yhteisö osallistuu sovelluskehitykseen ja tarjoaa palautetta kehitystiimille. Kirjoitushetkellä Spring on versiossa 5.1.2. Versiosta 5.0 lähtien Spring vaatii minimissään JDK-version 8 ja tukee jo alustavasti versiota 9. (Spring Framework Overview n.d.)

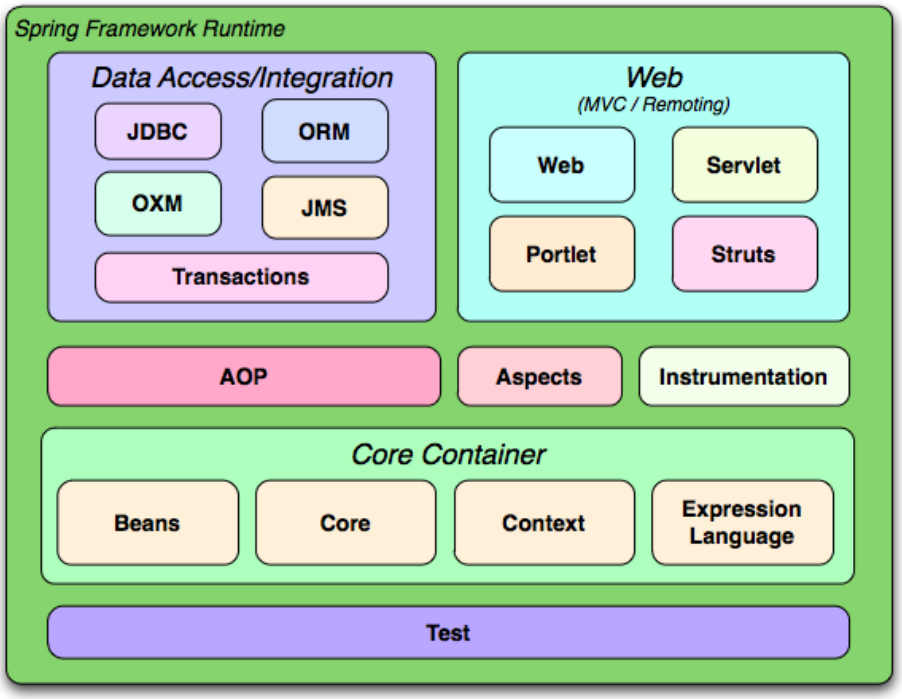
Spring on tarkoitettu helpottamaan Java-sovelluskehitystä eritoten yritysmaailmassa tarjoamalla valmiita komponentteja kehitystyön helpottamiseksi. Komponentit tar-

joavat esimerkiksi REST-rajapinnan sekä AMQP protokollan viestinnän. REST-rajapinnassa määritetään funktioita joihin kehittäjät voivat lähettää kysymyksiä ja saada vastauksia HTTP-protokollan välityksellä (Deering 2012). AMQP on viestiprotokolla, jota käytetään laajasti IoT-viestinnässä (Schneider 2013). Työssä käytettiin kuitenkin viestien lähettämiseen MQTT-protokollaa, josta kerrotaan lisää luvussa 4.3. Spring keskittyy sovellusten taustajärjestelmien pystyttämiseen stabiilisti ja tietoturvallisesti. Tällä tavalla kehittäjät voivat keskittyä sovelluksen bisneslogiikkaan.

Otetaan esimerkkinä edellä mainittu REST-rajapinta. Spring luo REST-rajapinnan taustajärjestelmät. Näihin taustajärjestelmiin kuuluvat esimerkiksi tunnistautumiseen sekä rajapinnan kutsujen vastaanottamiseen ja vastausten lähettämiseen liittyvät toiminnot. Kehittäjien työksi jää määritellä miten tunnistaudutaan, millaisia kutsuja otetaan vastaan, mitä niillä tehdään ja mitä vastaus sisältää. On selvää, että Spring mahdollistaa projekteissa suuren ajansäästön, sillä kehittäjät voivat nopeasti keskittyä olennaiseen eli sovelluskohtaisiin toimintoihin. Onko etenkin bisnesmielessä järkeä kirjoittaa jokaiseen projektiin uudelleen boilerplate-koodi REST-rajapinnan osalta, kun sellaisen saa Springin tarjoamana valmiiksi?

Spring-tuoteperhe koostuu erinäisistä projekteista. Projektit kehitetään tiettyyn tarpeeseen. Esimerkiksi Spring Data -projekti keskittyy datan käsittelyyn erilaisista datalähteistä kuten tietokannoista.

Kaikki projektit vaativat ytimekseen Spring sovelluskehyksen, Spring Frameworkin. Spring Framework on jaettu moduuleihin, joista sovellus voi valita tarvitsemansa. Kaikkien Spring-sovellusten ytimenä toimii Core Container (ks. kuvio 3), joka sisältää sovelluksen toiminnalle välttämättömät moduulit. Framework tarjoaa perustavan tuen luoda projekti useilla eri arkkitehtuurimalleilla sovelluksen vaatimuksista riippuen, esimerkiksi viestintä -tai web applikaatiolle. Ennen kehityksen aloittamista on suositeltavaa tutustua ydinkontin (core container) sisältämiin moduuleihin. Muihin moduuleihin voi tutustua lisää tarpeen vaatiessa.



Kuvio 3. Spring Framework -moduulit (Spring Framework modules n.d.)

Spring voi tuntua haastavalta aloittavalle kehittäjälle. Dokumentaatio on erittäin laaja, ja eri Spring-sovelluskehyksiä on runsaasti. Tätä helpottamaan vuonna 2014 julkaistiin Spring Boot -sovelluskehys, jonka tarkoituksena on yksinkertaistaa uuden Spring-sovelluksen luontiprosessia ja täten tarjota huomattavasti nopeampi ja helpompi tapa tutustua Spring-sovelluskehitykseen (Bhave, Bryant, Deleuze, Dupuis, Long, Nicoll, Overdijk, Pavić, Simons, Syer, Webb & Wilkinson n.d).

Spring Boot ei kuitenkaan ole vain opetustyökalu, vaan sitä hyödyntäen voidaan rakentaa täysin tuotantovalmiita applikaatioita. Myös tämän projektin toteutukseen valittiin Spring Boot juurikin tuotantovalmiuden sekä käytännönläheisen lähestymistavan vuoksi.

Spring Boot toimii hyödyntäen käynnistysmalleja (starter template). Käynnistysmallit sisältävät kaikki tarvittavat riippuvuudet, jotka tarvitaan tietyn ominaisuuden käynnistämiseksi. Esimerkiksi MVC-sovelluksen luomiseksi täytyy projektiin lisätä riippuvuus spring-boot-starter-web. MVC on yleisesti web-kehityksessä käytetty viitekehys, jossa applikaatio jaetaan model, view ja controller komponentteihin (MVC Framework - Introduction n.d). Spring hoitaa applikaation luonnin arkkitehtuuria noudattaen kehittäjän puolesta. Tässä projektissa käytettiin useita eri käynnistysmalleja

tarvittavien toimintojen saavuttamiseksi. Käytetyt käynnistysmallit käydään tarkemmin läpi toteutusosiossa.

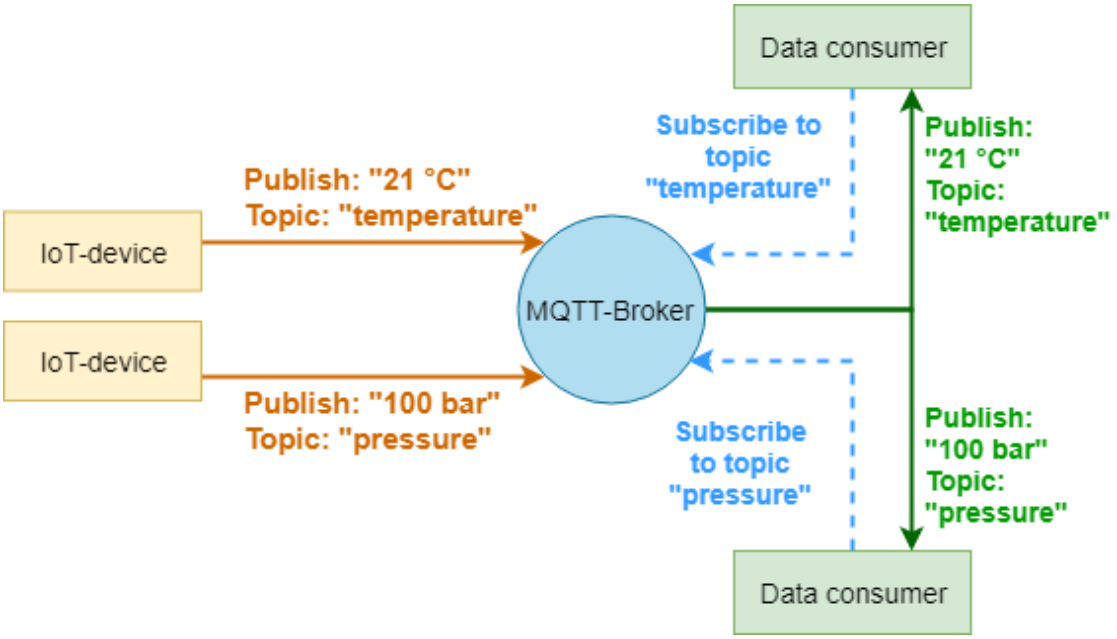
4.3 MQTT

IoT-projektissa tiedonsiirron kannalta tärkeintä on lähetettävän datan keveys sekä siirron luotettavuus. Vaihtoehtoja tiedonsiirron toteuttamiseen oli useita, ja kaikki tarjosivat hieman erilaisia vahvuuksia sekä heikkouksia. Projektin viestiprotokollaksi valittiin MQTT, joka on yksi suosituimmista ja pisimpään käytössä olleista viestiprotokollista IoT-applikaatioissa.

MQTT on kehitetty jo vuonna 1999, mutta vasta IoT-alan hurjan kasvun myötä se on noussut suosiossa räjähdysmäisesti. Syy tähän on IoT-laitteiden tarve lähettää paljon dataa matalalla virrankäytöllä sekä mahdollisesti heikolla verkkoyhteydellä. Tämän mahdollistaa protokollan rakenne, joka sisältää mahdollisimman vähän ylimääräistä dataa, jolloin viestikoot pysyvät pieninä. MQTT on erinomainen valinta M2M-viestinnälle, jossa dataa keräävä laite lähettää itse dataa toiselle laitteelle. (What is MQTT and When You Should Use It 2018.)

Tyypillisessä käyttötapauksessa IoT-laitteet lähettäisivät siis itse mittadatansa suoraan vastaanottajalle. Projektissa ei asennettu omia IoT-mittalaitteita vaan kerättiin dataa REST-rajapinnasta, joka tarjoaa mittalaitedataa usealta laitteelta. Tästä johtuen projektissa MQTT-dataa lähettää dataa keräävä producer-applikaatio, joka lähettää REST-rajapinnalta noudetun datan eteenpäin viesteinä käyttäen MQTT-protokollaa. Arkitehtuullisesti dataa lähettäviksi elementeiksi kuitenkin voitaisiin lisätä myös helposti aitoja IoT-laitteita.

MQTT-viestien jakelu toimii julkaisu/tilaus (publish/subscribe) periaatteella (ks. kuvio 4). Käytännössä tämä tarkoittaa sitä, että asiakasohjelmat voivat yhdistää datan julkaisijana, tilaajana tai jopa molempina. Julkaisijat toimivat datan lähettäjinä ja tilaajat vastaanottavat dataa. Yhteys luodaan viestilähettiin, joka hoitaa julkaistujen viestien levittämisen tilaajille. Projektissa käytetystä RabbitMQ-viestilähetistä kerrotaan lisää luvussa 4.4. (What is MQTT and When You Should Use It 2018.)



Kuvio 4. MQTT protokollan publish/subscribe-toimintakaavio

Tiedonkulun tärkein väylä on aihe (topic). Kuviossa 4 on nähtävillä kaavio, jossa esitellään myös aiheiden toimintaa yksinkertaisella tasolla. Aiheiden avulla data kulkee tiedosta kiinnostuneille vastaanottajille. Samalla aiheella voi olla samanaikaisesti useampi julkaisija sekä tilaaja. Toimintaidea on erittäin yksinkertainen ja mahdollistaa tehokkaan viestinjakelun. Aiheita hyödyntäen viestiliikenne saadaan siivilöityä ja ohjattua vain datasta kiinnostuneille vastaanottajille. (What is MQTT and When You Should Use It 2018.)

MQTT käsittelee aiheita tiedostopolkujen tapaan. Käytännössä aiheet voidaan siis erotella eri tasoihin käyttäen tiedostopolkurakennetta. Esimerkiksi aihe IoT-laitteilta tulevalle lämpötiladatalle voisi olla ”`devices/data/temperature/device-id`”. Jos haluttaisin tilata kaikki lämpötiladataa keräävien antureiden tiedot, voidaan käyttää jokerimerkkiä ”+”. Tällä merkillä voidaan tilata yhdeltä tasolta kaikki aiheet. Esimerkiksi kaikkien lämpötilatietojen keräämiseksi tilattaisiin aihe ”`devices/data/temperature/+`”. Kaiken datan noutamiseen useammalta kuin yhdeltä tasolta voidaan käyttää jokerimerkkiä ”#”. Tällä merkillä noudetaan kaikki merkin jälkeen tulevat rakenteen tasot. Esimerkiksi jos haluttaisiin noutaa kaikkien lämpötilatietojen lisäksi tiedot myös kaikki muut tiedot antureilta, tilattaisiin aihe ”`devices/data/#`”. Tämän jokerimerkin käyttö ei kuitenkaan ole suositeltavaa tilanteessa, jossa päädytään tilaa-

maan suuria määriä dataa usealta tasolta. Vastaanotettava dataliikenne saattaa kasvaa niin suureksi, ettei tilaava asiakaspääte ehdi käsittelemään viestejä riittävän nopeasti. (MQTT Essentials Part 5: MQTT Topics & Best Practices n.d.)

Yksinkertaisemmissa sovelluksissa voidaan pysyä tiedostopolun juuren tasolla, jolloin aihe voidaan nimetä yksinkertaisesti esimerkiksi lämpötiladatalle kuvaavasti ”temperature”. On kuitenkin suositeltavaa kuvailla aiheet selkeästi, ja aiheita tulisi mieluumin olla enemmän kuin vähemmän. Sekä julkistaessa että tilatessa tulee asiakaspäätteiden määritellä aihe. (MQTT Essentials Part 5: MQTT Topics & Best Practices n.d.)

Projektissa käytetty mittapistedataa lähettävä producer-applikaatio on toteutettu Javalla ja siinä käytetään ”Paho Java Client” -kirjastoa. Kirjasto mahdollistaa helpon MQTT-viestien muodostamisen sekä halutun aiheen tilaamisen tai sinne julkistamisen (Eclipse Paho Java Client n.d). Koska kirjasto hoitaa viestin muodostamisen sekä sen lähettämisen/vastaanottamisen, voidaan keskittyä viestin varsinaiseen sisältöön. Tärkeätä on muistaa kuitenkin se, että MQTT-viestin paketin maksimikoko on 256 megatavua (Cope 2018). Maksimikoko ei kuitenkaan tule missään tapauksessa olemaan ongelma pienikokoisia IoT-viestejä lähettäessä.

4.4 RabbitMQ

RabbitMQ on Erlang-ohjelmointikielellä toteutettu AMQP-viestilähetti (message broker). Ohjelmointikieleksi Erlang valittiin siksi, että sen tavoitteena on mahdollistaa luotettavia ja korkean saatavuuden applikaatioita. Tästä johtuen Erlangia käytetään esimerkiksi puhelinvaihteissa. (Dossot 2014.)

RabbitMQ on kirjoitushetkellä suosituin avoimen lähdekoodin viestilähetti. Käyttäjiä on aina pienistä startupeista suuriin yrityksiin. Suosion takana on varmastikin keveys, todistettu toimivuus sekä etenkin ilmaisuus. Realiteetti on kuitenkin se, että kalliille lisensoidulle tuotteille on alkanut ilmestymään kasvava määrä ilmaisia, hyväksi todettuja avoimen lähdekoodin vaihtoehtoja. Tärkeätä huomioida on se, että vaikka RabbitMQ on alun perin suunniteltu AMQP-viestilähettinä, tukee se nykyään myös muita

viestiprotokollia. Projektissa käytettiin IoT-viestintään MQTT-protokollaa, joka kevyestä viestikoostaan johtuen sopii täydellisesti projektin tarkoituksiin. RabbitMQ tukee MQTT-protokollaa liitännäisen kautta.

Mutta miksi edes tarvitaan viestilähetti? Lähetettävä data voitaisiin kyllä lähettää suoraan vastaanottajalle ilman välikäsiä, mutta IoT-datan lähetyksessä on viestilähetti lähes välttämätön. Suuressa IoT-kokonaisuudessa laitteet voivat lähettää tuhansia viestejä minuutissa. Jos vastaanottava komponentti ei ehdikään vastaanottamaan ja käsittelemään viestejä riittävän nopeasti, jäisivät jotkin saapuvat viestit kokonaan vastaanottamatta. Toinen tyypillinen vikatilanne olisi vastaanottavan komponentin kaatuminen tai verkkoyhteyden katkeaminen. Lähetettävät viestit jäisivät vastaanottamatta, eikä pudotetusta datasta jäisi jälkeä minnekään.

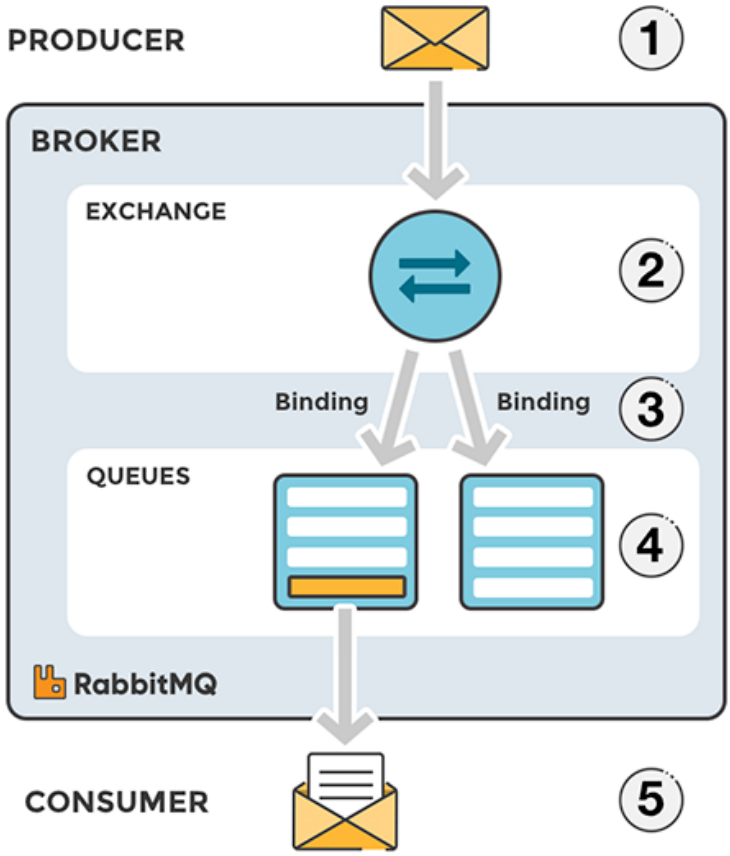
Lähettäjä ja vastaanottaja tulevat toisistaan täysin riippumattomiksi, kun väliin asetetaan viestilähetti. Kaikki keskustelu tapahtuu viestilähetin kautta, jonka tehtävänä on reitittää viestejä eteenpäin luotettavasti. Saapuvat viestit asetetaan viestilähetissä jonoon, jolloin vastaanottavan komponentin ongelmatilanteessa dataa ei menetetä. Esimerkiksi kaatumisen jälkeen vastaanotin jatkaisi viestien käsittelyä jonosta normaalisti, eikä dataa menetettäisi lainkaan.

Tämä tarkoittaa luonnollisesti myös sitä, että viestilähetin itsensä tulisi aina toimia luotettavasti. Tätä sivuttiin jo luvun alussa, jossa mainittiin, että RabbitMQ on suunniteltu juurikin luotettavuutta ajatellen.

Käydään läpi RabbitMQ:n viestinkulku (ks. kuvio 5). Käydään läpi osat, joiden kautta viesti kulkee:

1. Producer (tuottaja): Dataa tuottava lähde, joka julkaisee viestin vaihtimeen. Producer ei ole RabbitMQ:n osa. Projektissa producer on toteutettu Spring Frameworkilla.
2. Exchange (vaihdin): Vaihtimen tehtävänä on vastaanottaa julkaistu viesti ja tämän jälkeen sen tulee reitittää viesti oikeaan jonoon.
3. Binding (sidos): Vaihtimen ja jonon välille luodaan sidos. Sidoksen avulla vaihdin tietää mikä viesti reititetään mihinkin jonoon.

4. Queue (jono): Jonon tehtävänä on säilyttää viestit niin pitkään, kunnes consumer käsittelee viestit. Tämä on etenkin IoT-projektissa tärkeää, jossa viestejä voi tulla enemmän kuin niitä ehditään käsittelemään. Viestit eivät kuitenkaan saa kadota, vaan ne pidetään jonossa.
5. Consumer (käsittelijä) Käsittelijän tehtävänä on yksinkertaisesti käsitellä saapuva viesti. Lopuksi käsittelijä ilmoittaa RabbitMQ:lle että viesti on käsitelty onnistuneesti ja se voidaan poistaa jonosta. Käsittelijä ei ole RabbitMQ:n osa. Se on projektissa toteutettu Spring Frameworkilla.

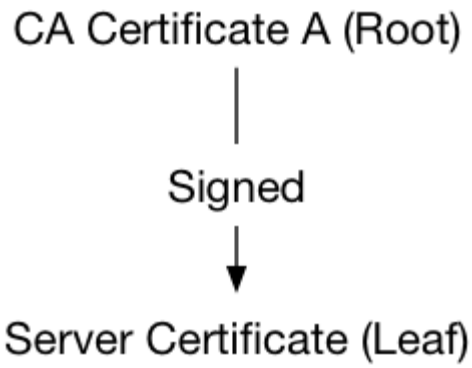


Kuvio 5. RabbitMQ toimintakaavio. (Johansson 2015.)

Saapuvat MQTT-viestit tulee saada reitittyä aiheen perusteella oikeisiin jonoihin. Reitit määritellään hyvin samantapaisesti kuin luvussa 4.3 käsitellyt MQTT-aiheet. Tiedostopolkumainen rakenne säilyy samanlaisena, mutta reitityksessä käytetään tasojen erottajana ”.”-merkkiä ja yhden tason jokerimerkkinä käytetään ”*”-merkkiä. Jokerimerkki ”#” toimii MQTT-aiheen tavoin merkin sekä sen ylempien tasojen jokerimerkkinä. (Johansson 2015.)

Reititys jonoihin on tärkeää parhaan suorituskyvyn saavuttamiseksi. RabbitMQ-jono on yksisäikeinen prosessi, jolloin se hyödyntää heikosti moniydinprosessoria. Yksittäinen jono kykenee käsittelemään noin 50 000 viestiä sekunnissa. Paras tapa hyödyntää modernia moniydinprosessoria on siis luoda useampi jono, jolloin kuorma jakautuu tasaisemmin prosessien välille ja sitä kautta useampaan ytimeen. Optimaalinen tilanne on, että jonoja on saman verran kuin prosessorin ytimiä. On kuitenkin parempi olla liikaa jonoja kuin liian vähän, sillä suorituskyky alkaa kärsimään vasta kun jonoja on tuhansia. (Johansson 2018.)

Projektissa tietoturva Producerin ja RabbitMQ:n välillä on toteutettu TLS Peer Verification -metodilla. TLS on salausprotokolla, joka tarjoaa päästä päähän -kommunikaation salauksen verkon yli (What is Transport Layer Security (TLS) 2018). Työssä TLS salaa yhteyden liikenteen ja varmistaa että yhdistävä osapuoli on luotettava. Yhteyden muodostamiseksi tarvitaan sertifikaatit, jotka projektissa generoitiin ja allekirjoitettiin itse. Tuotantoversioon olisi suositeltavaa, että sertifikaatit olisi allekirjoittanut varmenneviranomainen. Sekä Palvelin että lähettäjät tarvitsevat sekä oman sertifikaattinsa että varmenteen antajan sertifikaatin (ks. kuvio 6).



Kuvio 6. RabbitMQ palvelimen TLS sertifikaattiketju (TLS Support n.d.)

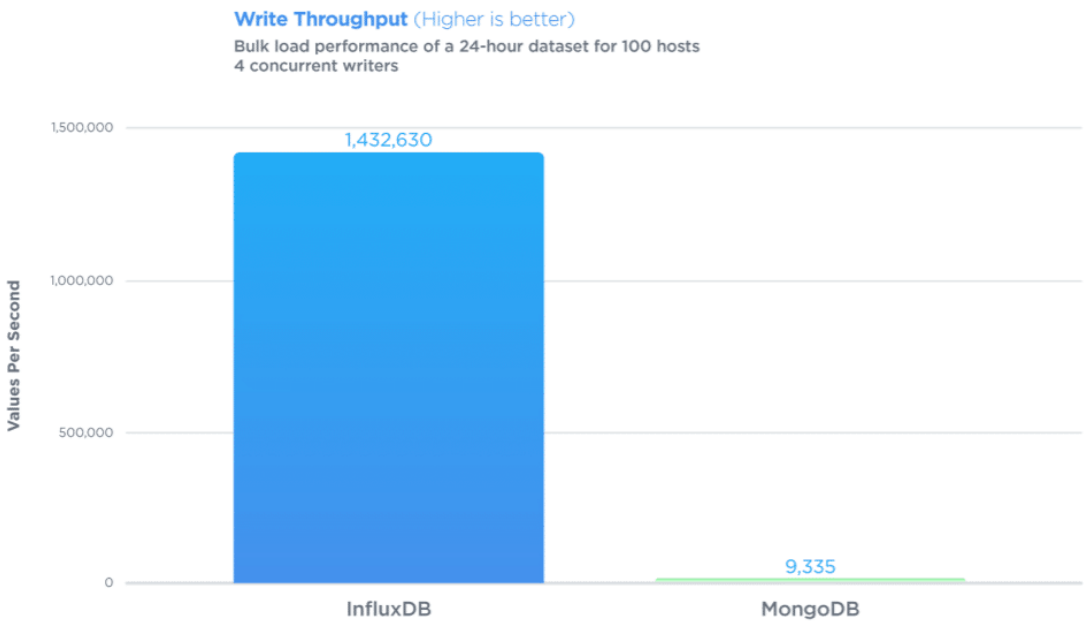
4.5 InfluxDB

IoT-teknologioiden nousun aikana on myös tiedon varastoinnin suunnalla noussut uusi trendi aikasarjatietokantojen muodossa. Aikasarjatietokannat eivät ole uusi keksintö, mutta aikaisemmat aikasarjatietokannat olivat suurimmaksi osaksi keskittyneet finanssialan datan varastointiin (Time Series Database (TSDB) Explained n.d).

IoT-järjestelmissä mittapisteiltä kerätään suuria määrä dataa eri lähteistä. Tässä työssä kerätty data tuli järjestelmään aikajärjestyksessä. Se milloin tieto kerättiin, oli tärkeää tiedon visualisoinnissa. Dataa tuli pystyä tarkastelemaan suuria määriä eri aikaväleillä. Aikasarjatietokanta soveltui tähän käyttötapaukseen hyvin. Aikasarjatietokannan avulla voidaan varastoida nopeasti suuria määriä dataa pieneen levytilaan samalla säilyttäen nopeat tietokantakyselyt.

Aikasarjatietokannan nopeus ei tietenkään perustu taikuuteen, vaan pikemminkin erityisesti aikasarjadataa varten suunniteltuun arkkitehtuuriin. Data varastoidaan aikaleimojen mukaisesti ja kompressoidaan tehokkaasti. Datan tarkkuutta voidaan muokata sen elinkaaren aikana, jolloin vanhasta datasta tehdään vähemmän tarkkaa, mutta kokonaiskuva pitkällä aikavälillä säilytetään. (Time Series Database (TSDB) Explained n.d.)

Projektiin valittu InfluxDB on avoimen lähdekoodin aikasarjatietokanta. Kyseessä on kirjoitushetkellä maailman suosituin ja suurimmassa kasvussa oleva aikasarjatietokanta (DB-Engines Ranking of Time Series DBMS. n.d). Suureen suosioon mitä todennäköisimmin liittyy suuren käyttäjäyhteisön toteama luotettavuus, nopeus sekä jälleen kerran tuotteen ilmaisuus.
Vertailut aikasarjadatan käsittelyssä InfluxDB:n ja dokumenttipohjaisen MongoDB:n välillä paljastavat hurjia eroja. InfluxDB oli vertailussa 153 kertaa nopeampi datan kirjoittamisessa (ks. kuvio 7), vei 64 kertaa vähemmän levytilaa ja suoritti tietokantakyselyt kolme kertaa nopeammin kuin MongoDB (Churilo 2018).



Kuvio 7. Datan kirjoitusnopeuden vertailu InfluxDB:n ja MongoDB:n välillä (Churilo 2018).

On tärkeää huomata, että testitulokset ovat peräisin InfluxDB:n kehittäjän tilaamista testeistä, ja täten on hyvinkin mahdollista, että testauksessa on suosittu oman tuotteen vahvuuksia. Testin tuloksiin tulee suhtautua jonkin verran kriittisesti, mutta niiden perusteella voidaan kuitenkin todeta, että InfluxDB on aikasarjadatan varastointiin paremmin sopiva tietokanta.

Vaikka InfluxDB on toiminnaltaan täysin erilainen kuin SQL-tietokanta, sen kyselyiden kirjoitusasu jäljittelee SQL-kyselyitä. Tämä tekee käytöstä helppoa ihmisille, jotka ovat tottuneet SQL-tietokantojen käyttöön.

Käydään seuraavaksi läpi tärkeimmät InfluxDB:n toimintaperiaatteet. Yksittäisessä InfluxDB:n datapisteessä tärkeimpänä kenttänä on aina aikaleima. Tämän lisäksi datapiste sisältää kenttäavaimen (field key) sekä kenttäarvon (field value). Kenttäavaimet ovat aina merkkijonoja ja toimivat kenttäarvojen tunnisteina. Kenttäarvot sisältävät varsinaisen datan, ja ne voivat olla sisällöltään erilaisia datatyyppejä kuten merkkijonoja ja numeroarvoja. Esimerkiksi kenttäavain ”syke” voi viitata arvoihin 100, 125 ja 200 eri mittapisteissä. Kyseiset arvot saataisiin noudettua tietokannasta kirjoittamalla tietokantakysely, jossa noudettaisiin kaikki arvot joihin kenttäavain ”syke” viittaa (ks. kuvio 8).

```
> SELECT "syke" FROM "HeartData"
```

Kuvio 8. InfluxDB-tietokantakysely ”syke” kentälle

Kenttäavaimien heikkoutena on se, että ne eivät ole indeksoituja tietokantaan. Heikkoudet esiintyvät, kun tietokantakysely tehdään kenttäavaimelle tietyin rajauksin, esimerkiksi haluttaisiin vain ”syke”-arvot, jotka ovat alle 150 (ks. kuvio 9). Tietokannan pitää ensin tarkistaa jokainen haettua kenttäavainta vastaava arvo ja verrata sitä tehtyihin rajauksiin ennen kuin se tuottaa vastauksen. Tämä hidastaa varsinkin laajoja tietokantakyselyitä. (InfluxDB key concepts N.d.)

```
> SELECT "syke" FROM "HeartData" WHERE "syke" < 150
```

Kuvio 9. InfluxDB-tietokantakysely ”syke” kentällä, jossa arvo alle 150.

Kyselyiden nopeuttamiseksi voidaan datarakenteeseen lisätä myös tageja. Tagit eivät ole pakollisia, mutta ne ovat indeksoituja toisin kuin kentät. Kenttien tapaan myös tageilla on sekä avain (tag key) että arvo (tag value). Varsinkin kyselyiden rajauksissa usein käytetyistä kentistä kannattaa harkita tekevänsä tageja, jolloin rajatun tietokantakyselyn ei indeksoimisen ansiosta tarvitse käydä kaikkia avainta vastaavia arvoja läpi. (InfluxDB key concepts N.d.)

4.6 Grafana

Kerätyn datan visualisointiin projektissa valikoitiin avoimen lähdekoodin analytiikka ja monitorointi työkalu Grafana. Kyseessä ei suinkaan ole mikään pieni tekijä analytiikkaohjelmistojen maailmassa, sillä Grafanaa käyttävät monet suuret yritykset mukaan lukien eBay, PayPal ja Uber. Grafana tukee yli kolmeakymmentä datalähdettä mukaan lukien MySQL, PostgreSQL ja projektin kannalta tärkeimpänä InfluxDB. (Kili 2018.)

Grafanalla pystytään noutamaan ja visualisoimaan reaaliaikaista dataa. Tärkein ja käyttäjälle näkyvin komponentti on hallintapaneeli (dashboard). Hallintapaneelin näkymä koostuu pienemmistä paneeleista (panel), jotka noutavat tietolähteistä dataa

ja esittävät sen erinäisillä tavoilla (ks. kuvio 10). Esitystapoja löytyy esimerkiksi graafin, taulukon sekä karttanäkymän muodoissa. Paneelit on helppo sijoittaa ja säätää oikean kokoisiksi hallintapaneelissa. Ulkoasu saadaan muokattua ilman minkäänlaista ohjelmointia raahaamalla ikkunoita sekä venyttämällä niiden reunoja. On jopa mahdollista esittää dataa useasta eri datalähteestä yksittäisessä paneelissa. Hallintapaneeleja voidaan luoda useita ja ne voidaan kaikki konfiguroida erikseen.

Kuvio 10. Esimerkki Grafana-hallintapaneelista ja sen sisällöstä.

Jokaisella hallintapaneelilla on oma sisäänrakennettu versionhallintansa. Jos tehdyssä päivityksessä tapahtuisikin virhe, voidaan edellisiin versioihin helposti palata. Tämä helpottaa huomattavasti tiimityöskentelyä. Tiimityöskentelyyn liittyen hieno ominaisuus on myös paneelien siirreltävyys hallintapaneelien välillä. Mikä tahansa paneeli voidaan helposti kopioida hallintapaneelista toiseen. Myös kokonaisen hallintapaneelin konfiguraatio voidaan tallentaa JSON-muodossa ja käyttää esimerkiksi pohjana kaikissa tulevissa hallintapaneeleissa.

Jos omaan käyttötapaukseen sopivaa paneelia ei löydy, voidaan sellainen tehdä myös itse. Grafana tukee liitännäisiä, joita yhteisö pystyy itse kehittämään. Myös tuki uudelle datalähteelle on mahdollista lisätä liitännäisten avulla. Grafana on loistava esimerkki avoimen lähdekoodin konseptin toimivuudesta. Yhteisö on auttanut kehittämään paljon liitännäisiä, jotka ovat parantaneet ja laajentaneet tuotteen ominaisuuksia käytännössä ilmaiseksi. Liitännäisten kehitykseen voidaan käyttää JavaScriptiä sekä kaikkia ohjelmointikieliä, jotka kääntyvät JavaScriptiksi, esimerkiksi TypeScript

(Developer Guide n.d). Projektissa omia liitännäisiä ei kehitetty, sillä se ei työssä kerätyn datan esittämiseen ollut tarpeellista.

Paneelikohtaisesti voidaan asettaa myös hälytyksiä. Hälytyksille asetetaan raja-arvot ja hälytyksen lauetessa lähetetään halutulle taholle huomautus määritellyllä tavalla. Hälytys voidaan lähettää esimerkiksi sähköpostiin tai Slack-viestintäpalveluun. Tämä mahdollistaa reagoinnin muutoksiin datassa nopeasti ilman että hallintapaneelia täytyy aina olla seuraamassa.

Grafana on keskittynyt tarjoamaan hyvän tietoturvan ja sisältääkin oman sisäisen pääsynhallintajärjestelmänsä. Sisäisen järjestelmän tilalle voidaan kuitenkin helposti integroida myös esimerkiksi tunnistautuminen Googlen tai GitHubin kautta. Tällä tavalla pystytään välttämään uusien tunnusten luonnit organisaation jäsenille käyttämällä esimerkiksi olemassa olevia Google-tunnuksia. (User Authentication Overview n.d.)

Rekisteröidyille käyttäjille määritellään käyttöoikeudet roolien mukaan. Lisäksi käyttäjät voidaan järjestellä tiimeihin, joille annetaan tietyt käyttöoikeudet. Projektissa käytettiin Grafanan sisäistä pääsynhallintajärjestelmää. Hallintapaneelit voidaan järjestellä omiin kansioihinsa, joihin voidaan määritellä käyttäjäkohtaiset pääsyoikeudet. (Permissions Overview n.d.)

4.7 Docker

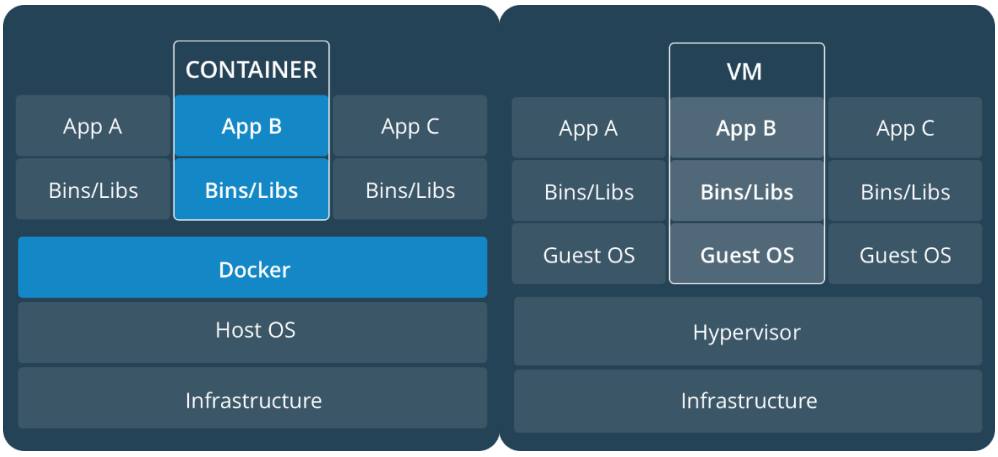
Docker on avoimen lähdekoodin alusta applikaatioiden kehittämiseen, toimittamiseen ja suorittamiseen Linuxilla, Windowsilla sekä Mac OS:llä. Dockerin pääideana on se, että applikaatio erotetaan sitä käyttävän ympäristön infrastruktuurista, jolloin se on helposti siirreltävissä ympäristöjen välillä. Tämän kaiken mahdollistaa kontti (container), joka eristää applikaation omaan ympäristöönsä. Docker on suosittu alusta ja sitä käyttää mm. PayPal, Visa sekä VR.

Dockerin käyttämä konttiteknologia on ollut olemassa jo vuosia ennen Dockerin syntyä, mutta se ei koskaan aiemmin saavuttanut samanlaista räjähdysmäistä suosiota. Tämä johtuu laajalti siitä kuinka helppokäyttöinen Docker on. Konttien paketointi,

suorittaminen ja ylläpitäminen tapahtuu helposti komentolinjan ja HTTP-rajapinnan kautta. (Docker in Production: Lessons from the Trenches 2015.)

Kuvitellaan tilanne, jossa haluat siirtää kehittämäsi Java-applikaation pilvipalvelussa olevalle virtuaalikoneelle. Saadaksesi applikaation toimimaan tulee virtuaalikoneelle ensin asentaa Java SDK, kääntää applikaatio ja suorittaa se. Docker-kontti voidaan konfiguroida tekemään kaikki edellä mainitut vaiheet oman eristetyn ympäristönsä sisällä. Docker-kontti voidaan siis helposti vain siirtää virtuaalikoneelle ja käynnistää.

Ennen Dockerin ja konttien yleistymistä applikaatioita suoritettiin usein erillisissä virtuaalikoneissa yksittäisen isäntälaitteen sisässä. Tällä tavalla saadaan applikaatiolle luotua oma eristetty ympäristö, mutta virtuaalikoneen sisällä oleva käyttöjärjestelmä syö huomattavia määriä turhia järjestelmäresursseja. Docker-kontteja voidaan suorittaa useita yksittäisessä isäntälaitteessa, mutta kontit pyörivät natiivisti isäntälaitteen käyttöjärjestelmän päällä omina prosesseinaan, jolloin ne käyttävät ainoastaan tarvitsemansa määrän resursseja. Tämä tekee konteista huomattavasti suorituskykyisempiä kuin virtuaalikoneet. Kuviossa 11 on havainnoitu edellä mainitut erot kontin ja virtuaalikoneen välillä.



Kuvio 11. Vertailu Docker-kontin ja virtuaalikoneen välillä (Docker concepts n.d)

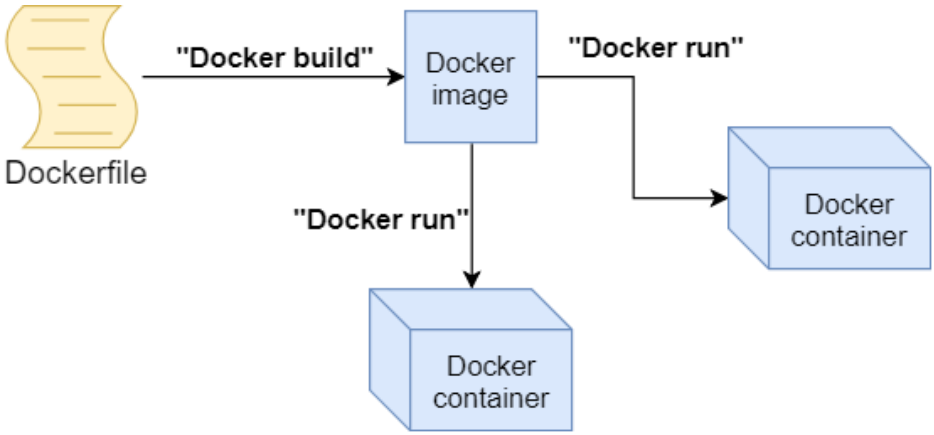
Opinnäytetyössä projektin eri komponentit paketoitiin Docker-kontteihin, jotta ne voitaisiin helposti tulevaisuudessa siirtää pyörimään eri ympäristöön, esimerkiksi pilvipalveluun. Kontit helpottivat myös omalta osaltaan projektin eri osien erottamista toisistaan selkeiksi kokonaisuuksiksi.

Docker-kontin luontiin ja käynnistykseen käytetään pohjana Docker-imagea. Docker-image on esikoottu ympäristö tietylle teknologialle tai palvelulle. Image ei siis ole ajettava prosessi vaan pikemminkin kokoelma tarvittavia tiedostoja, kirjastoja ja konfiguraatioita ympäristön koontiin. Lähes kaikille suosituille teknologioille löytyy kehittäjien ylläpitämä Docker-image. Kaikille projektissa käytettäville palveluille löytyi valmiit Docker-imaget jopa erilaisina variaatioina. Esimerkiksi RabbitMQ Docker -imagesta on versio hallintakonsolin kanssa sekä ilman. Jos graafiselle hallintakonsolille ei ole tarvetta voidaan säästää tiedostokoossa ja suorituskyvyssä valitsemalla versio ilman hallintakonsolia. Docker-image noudetaan komentolinjakomennolla ”`docker pull`”. Tämän jälkeen noudetusta imagesta voidaan käynnistää kontti ”`docker run`” -komennolla. (Takacs 2018.)

Yksi tärkeä asia ottaa huomioon Docker-kontteja käyttäessä on datan varastointi. Esimerkiksi kontti, joka pitää sisällään tietokannan ei missään nimessä saisi kadottaa tietokantaan tallennettua dataa, vaikka kontti kaatuisikin. Kontin sisään on mahdollista varastoida dataa, mutta se ei ole suositeltavaa. Tämä johtuu pääasiassa siitä, että Docker-kontin sammuessa sen sisälle tallennettu data ei säily. Kontin käynnistyessä uudelleen kaikki kontin sisään tallennettu data on menetetty. Tätä varten kontin tulee pystyä kirjoittamaan dataa kontin ulkopuolelle isäntäkoneeseen. Tästä on hyötyä myös suorituskyvyn näkökulmassa, sillä kirjoittaminen kontin sisälle on hitaampaa kuin kontista tallennus isäntäkoneelle.

Docker tarjoaa kaiken kaikkiaan kolme erilaista tapaa liittää konttiin dataa isäntäkoneelta: volume, bind mount tai tmpfs volume. Volume on näistä kolmesta suositelluin ja sitä käytettiin tässä projektissa. Volume on Dockerin itsensä hallinnoima ja se voidaan luoda terminaalissa komennolla ”`docker volume create`” tai kontin luonnin yhteydessä. Data tallennetaan kansioon isäntäkoneelle, joka luodaan automattisesti, jos sitä ei ole olemassa. Bind mount toimii samanlailla kuin volume, mutta ei ole Dockerin itsensä hallinnoima. Bind mountin käyttöä ei suositella, koska se tarvitsee valmiin kansiorakenteen mihin data tallennetaan ennen sen luontia, eikä sitä voida ohjata Docker-komennoilla. Väliaikaisen datan nopeaan varastointiin käytetään tmpfs volumea. Kyseessä on siis datavarasto kontin ulkopuolella, joka pyyhitään pois kontin sammuessa. (Manage data in Docker n.d.)

Kontin luomisen automatisoinnin mahdollistamiseksi kannattaa luoda Dockerfile. Dockerfile on tekstitiedosto, jossa määritellään mitä kontin sisällä olevassa ympäristössä tapahtuu. Konfiguraatiomahdollisuuksia on runsaasti, mutta liikkeelle pääsyyn tarvitaan ainoastaan FROM-ohjeistus, joka määrittelee mitä Docker-imagea käytetään pohjana kontin luonnissa. Lisäksi Dockerfilessä voidaan esimerkiksi kopioida kansioita isäntäkoneelta konttiin, avata portteja kontista ulkoverkkoon sekä ajaa komentoja kontin sisällä esimerkiksi applikaation käynnistämiseen. Kontti koostetaan ”`docker build`” -komennolla ja koostettu kontti käynnistetään jälleen ”`docker run`” - komennolla (ks. kuvio 12). (Dockerfile reference n.d.)



Kuvio 12. Toimintakaavio Dockerfilen käytöstä.

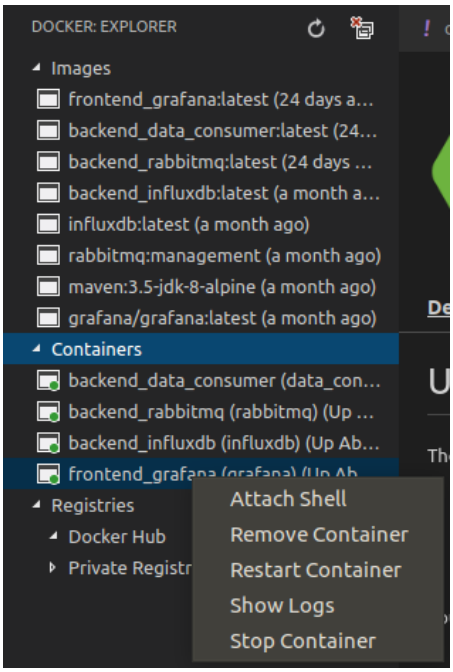
Yhden kontin automatisointi onnistuu hienosti Dockerfileä käyttäen, mutta entä jos halutaan orkestroida useamman Docker-kontin koostaminen ja käynnistys? Docker Compose on työkalu, joka mahdollistaa useiden Docker-konttien koostamisen ja käynnistämisen yksittäisen YAML-tyyppisen tiedoston avulla. Compose vaatii aluksi, että applikaatioille on luotu omat Dockerfilet, joilla ne saadaan missä tahansa automatisoidusti luotua. Tämän jälkeen määritellään palveluita, jotka koostuvat applikaatioista ”`docker-compose.yml`”-tiedostoon. Composella luodut palvelut suoritetaan yhdessä omassa eristetyssä ympäristössään. Kaikki kontit saadaan helposti koostettua ja käynnistettyä käyttämällä ”`docker-compose up`” -komentoa. (Overview of Docker Compose n.d.)

5 Toteutus

5.1 Kehitysympäristö

Kehitystyö tehtiin Linux Ubuntu 18.04.1 LTS -virtuaalikoneella. Linux valittiin johtuen aikaisemmista positiivisista kokemuksista sekä siitä, että etenkin Docker toimii luotettavammin Linuxilla. Virtualisointiohjelmana käytettiin VirtualBox-ohjelmistoa.

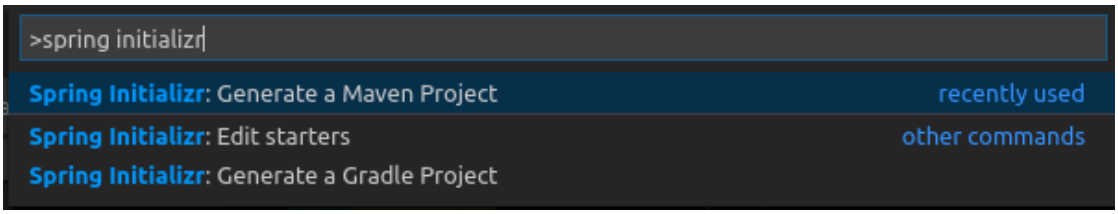
Projektissa kokeiltiin kahta erilaista koodieditoria. Aluksi Java-kehitykseen päätettiin käyttää IntelliJ IDEA IDE:tä, mutta ilmainen ”Community edition”-versio ei tue Spring Boot -applikaatioita ja on mahdollisesti raskas virtuaalikoneella käytettäväksi. Tämän jälkeen päädyttiin käyttämään huomattavasti kevyempää Visual Studio Code -koodieditoria, joka tukee useita eri ohjelmointikieliä ja teknologioita liitännäisten kautta. Docker-liitännäinen tarjoaa tuen syntaksin korostukselle Dockerfile ja docker-compose-tiedostoissa. Lisäksi liitännäisellä voidaan tarkastella luotuja Docker-imageja sekä kontteja ja niiden lokeja (ks. kuvio 13).



Kuvio 13. Visual Studio Coden Docker-liitännäisen valikko

Spring Boot Extension Pack -liitännäinen tarjosi mahdollisuuden generoida Spring Boot -projektin nopeasti suoraan Visual Studio Coden sisällä. Tämä tapahtui avaamalla Visual Studio Coden komentovalikko näppäinyhdistelmällä CTRL + shift + P ja

kirjoittamalla ”`Spring initializr`” (ks. kuvio 14). Tällä tavalla työssä generoitiin Maven Spring Boot -projektit.

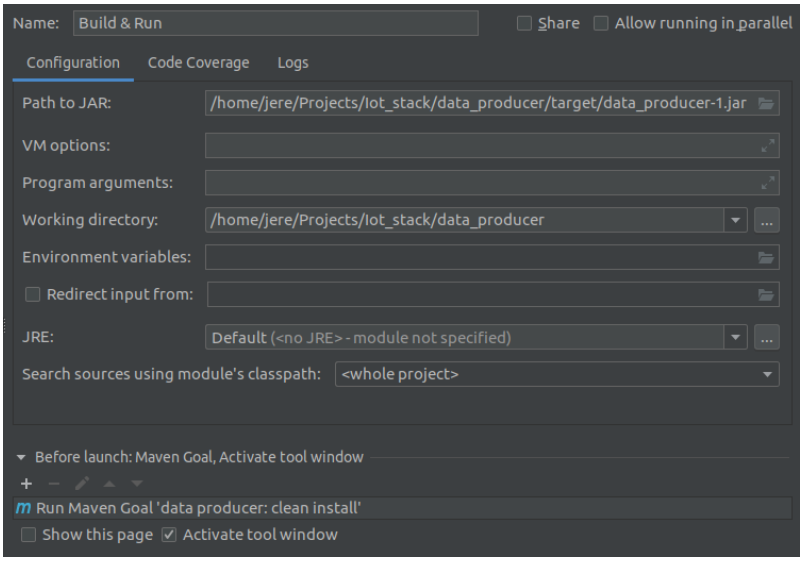


Kuvio 14. Spring Boot -projektin generointi Visual Studio Codessa

Spring Boot Dashboard -liitännäisellä voidaan tutkia työtilassa olevia Spring Boot -applikaatioita, sekä käynnistää niitä. Tämä ei kuitenkaan ole virtuaalikoneella suositeltavaa, sillä tätä kautta applikaation käynnistäminen oli raskasta ja aiheutti usein Visual Studio Coden jumiutumisen. Spring Boot Tools -liitännäinen osoittautui hyödylliseksi, sillä se tarjosi esimerkiksi koodin täydentämisen Spring-kehityksessä. Liitännäisten ansiosta Java-kehitys on siis täysin mahdollista Visual Studio Coden kanssa. Kehitetyt Java-applikaatiot kannattaa kuitenkin virtuaalikoneella suorittaa suoraan Linux-terminaalissa paremman suorituskyvyn saavuttamiseksi.

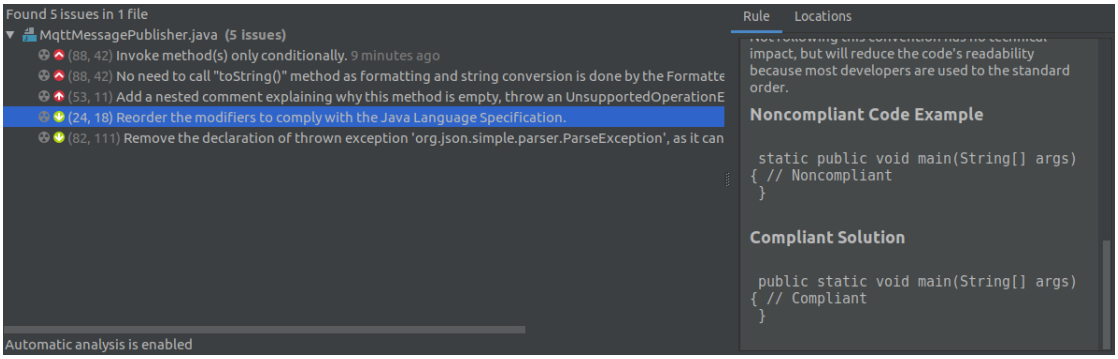
Visual Studio Code ei kuitenkaan ole koodieditori, joka on suunniteltu Java-ohjelmointikieltä varten ja se on huomattavasti suositumpi web-kehittäjien keskuudessa. Esimerkiksi Visual Studio Coden debuggeria ei saatu projektissa lainkaan toimimaan Spring Boot -applikaation kanssa.

Projektissa päätettiin palata kehityksessä IntelliJ IDEA IDE:n pariin. Tämä osoittautui hyväksi päätökseksi, sillä vaikka IDE ei suoraan tue Spring Boot -applikaatioita on niitä helppo koostaa ja suorittaa suoraan editorista, jopa debuggerin kanssa. Tätä varten piti ”Run/Debug Configurations”-valikkoon käydä lisäämässä konfiguraatio. Konfiguraatiossa määritellään Maven-työkalulla esisuoritettava ”`mvn clean install`” -komento sekä varsinainen suoritettava JAR-tiedosto (ks. kuvio 15). Tällä tavalla ohjelmistoa voidaan helposti testata suoraan editorista. Ohjelmiston suorittaminen terminaalin kautta on silti suorituskyvyn kannalta nopeinta, mutta kehitetyt suhteellisen kevyet ohjelmistot toimivat koodieditorin kautta riittävän hyvin.



Kuvio 15. IntelliJ IDEA -ohjelmiston koostamisen ja suorittamisen konfiguraatio

Myös IntelliJ IDEA tukee liitännäisiä, mutta se sisältää jo valmiiksi erinomaisen tuen Java-kieltä varten. Kirjoitetun ohjelmistokoodin laadun parantamiseen on kuitenkin erittäin suositeltavaa asentaa liitännäinen SonarLint. Liitännäinen asennetaan Asetuksien ”Plugins”-valikosta. SonarLint analysoi kirjoitettua koodia ja ehdottaa siihen tehtäviä parannuksia. Parannusehdotuksissa on esitelty esimerkkikoodien kautta, miten parannukset tehdään oikeaoppisesti. Kuviossa 16 on esillä SonarLint-näkymä, jossa on listattu koodissa havaitut ongelmat. Kehityksen aikana kirjoitetusta koodista vähentyi huonojen tapojen mukaan kirjoitettua koodia selkeästi tämän liitännäisen ansiosta.



Kuvio 16. SonarLint-koodianalyysityökalu

5.2 Kerättävä data

Työssä tarvittiin datalähde, josta saadaan paljon dataa useasta mittapisteestä. Työhön valittiin kerättävän datan lähteeksi OpenAQ-palvelu. OpenAQ on avoimen lähdekoodin projekti, joka tarjoaa maailmanlaajuisesti kerättyä avointa ilmansaastedataa. Mittapisteitä työn tekohetkellä oli 10 184:ssä sijainnissa 68:ssa maassa. Dataa työssä kerättiin palvelun tarjoamista PM10, PM2.5, SO2, CO, NO2 ja O3 saastetiedoista. (Air Quality Data n.d.; Where does OpenAQ data come from? n.d.)

OpenAQ ei itse tuota dataa mittalaitteilla vaan kerää dataa reaaliaikaisesti muilta osapuolilta koostaen ne yhteen lähteeseen. OpenAQ ei takaa datan olevan tarkkaa tai oikeanlaista, mutta alkuperäisdatan täytyy olla kuitenkin valtion viranomaisten tai kansainvälisten viranomaisten tuottamaa, jotta se hyväksytään palveluun kerättäväksi. (Where does OpenAQ data come from? n.d.)

Datan jakeluun OpenAQ käyttää REST-rajapintaa, joka tarjoaa useita tapoja noutaa dataa JSON-muodossa. Työssä tarvittiin kuitenkin vain yhtä GET-kyselyä, jolla saadaan noudettua kaikki viimeisimmät ilmansaastemittaukset. Kyselyt on rajoitettu IP-osoitteiden mukaan 2000 kyselyyn viiden minuutin aikana (Open AQ Platform API n.d). Työssä tehdään vain yksi kysely 15 minuutin välein, jossa noudetaan 10 000 viimeisintä mittausta ja tallennetaan ne tietokantaan. Kysely tehdään osoitteeseen https://api.openaq.org/v1/measurements?limit=10000. 10 000 mittauksen noutaminen kuulostaa ensikuulemalta suurelta lukemalta, mutta todellisuudessa rajapinta palauttaa mittaukset erittäin nopeasti, yleensä muutaman sekunnin sisään.

Kyseessä oli erinomainen datalähde projektiin, sillä saapuvaa dataa oli paljon ja se oli vaihtelevaa. InfluxDB-tietokantaan saatiin tallennettua paljon dataa, jolloin päästiin tarkastelemaan sen toimivuutta suurien datamäärien kanssa. Data sisälsi myös mittalaitteiden sijainnit, joka mahdollisti niiden visualisoinnin Grafanassa kartalle.

5.3 Data producer

Data producer on Java Spring Boot -applikaatio, jonka tehtävänä on kerätä IoT-dataa ja lähettää sitä MQTT-viesteinä RabbitMQ-jonoihin. Applikaatio koostetaan JAR-tiedostoksi käyttäen Maven-koostamistyökalua. Mavenin käyttämässä pom.xml-tiedostossa määritellään esimerkiksi applikaation nimi, koostamistapa sekä riippuvuudet. Producer-applikaatiota ei paketoida Docker-konttiin, vaan ideana on toimittaa tämä yksittäinen JAR-tiedosto laitteelle, joka lähettää dataa.

Producer lähettää kyselyn OpenAQ-rajapintaan 15 minuutin välein ja noutaa vastauksen. Ajastus saatiin helposti toteutettua hyödyntäen Spring Frameworkin scheduling-ominaisuutta. Scheduling saadaan päälle lisäämällä Application-luokkaan @EnableScheduling-annotaatio. Application-luokka on Spring Boot -sovelluksen sydän. Kyseessä on lyhyt pääluokka, johon tarvitaan vähintään @SpringBootApplication-annotaatio sekä main-funktio, jossa käynnistetään applikaatio (ks. kuvio 17).

```
@SpringBootApplication
@EnableScheduling
public class Application {

    ▶ Run | Debug
    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }
}
```

Kuvio 17. Data producer Application -luokka

Sovelluksen logiikan kannalta pääluokkana toimii DataFetcher-luokka. Kyseessä on luokka, jossa alustetaan MqttMessagepublisher-luokkaan perustuva olio mqttMessagePublisher, REST-rajapintaan kyselyjä lähettävä restTemplate-olio sekä määritellään ajastettu funktio scheduledTasks, joka suorittaa datan noutavan, käsittelevän ja eteenpäin lähettävän fetchData-funktion. Käydään seuraavaksi läpi tämän luokan edellä mainitut osat.

Ajastettu funktio scheduledTasks vaatii toimiakseen @Scheduled-annotaation. Tämän annotaation avulla Spring tietää taustajärjestelmissä ajastaa funktion. Annotaa-

tiolle annetaan myös argumentti, joka määrittää kuinka usein ajastettu funktio suoritetaan. Argumentti annetaan millisekunteina, tässä tapauksessa 900 000 millisekuntia, joka vastaa 15 minuuttia. Funktion ainoa tehtävä on suorittaa fetchData-funktio määritellyin väliajoin (ks. kuvio 18).

```
// Fetch new data every 15 minutes
@Scheduled(fixedRate = 900000)
public void scheduledTasks() throws InterruptedException {
    fetchData();
}
```

Kuvio 18. Ajastettu scheduledTasks-funktio

Funktiossa fetchData noudetaan ensitöiksi REST-rajapinnasta dataa. Tähän käytetään RestTemplate-luokkaa, joka on osa Spring Frameworkin web-kirjastoa. Spring Boot tekee tästäkin helppoa tarjoamalla isäntäluokan konstruktorin argumenttina annettavan RestTemplateBuilder-luokan. Luokka automatisoi restTemplate-olion luonnin, mutta sitä voi myös itse konfiguroida, esimerkiksi asettamalla rajapintaan yhdistämisen aikakatkaisun. Alustetulla oliolla noudetaan data rajapinnasta kuviossa 19 näkyvällä tavalla.

```
Pollution weather = restTemplate.getForObject(
        "https://api.openaq.org/v1/measurements?limit=10000", Pollution.class);
```

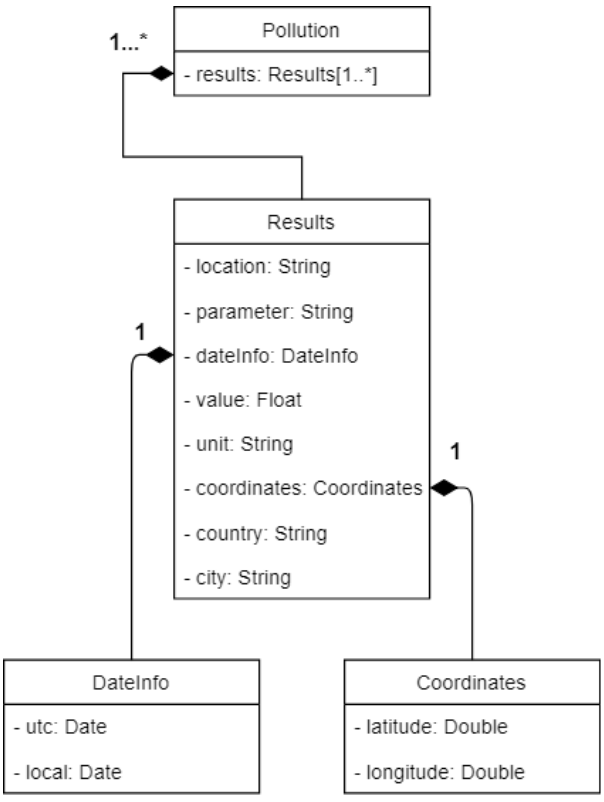
Kuvio 19. Datan noutaminen rajapinnasta

Kuviossa 19 nähdään että noudettava data tallennetaan Pollution-luokan weather-olioon. Kyseessä on luokka, jonka tehtävänä on kartoittaa vastaanotettu tekstimuotoinen JSON Java-olioksi. Luokka on toteutettu käyttäen Jackson JSON -kirjastoa. Kuviossa 20 on esillä esimerkkinä rajapinnan palauttaman datan results-taulukko, joka sisältää yhden mittaustuloksen. Huomaa, että rajapinta palauttaa applikaatiossa tuloksia 10 000.

```
"results": [
    {
        "location": "Bryn skole",
        "parameter": "pm25",
        "date": {
            "utc": "2018-12-06T18:00:00.000Z",
            "local": "2018-12-06T19:00:00+01:00"
        },
        "value": 12.570694,
        "unit": "µg/m³",
        "coordinates": {
            "latitude": 59.914106,
            "longitude": 10.82263
        },
        "country": "NO",
        "city": "Oslo"
    }
]
```


Kuvio 20. Esimerkki rajapinnan palauttaman datan ”results”-taulusta

Rajapinnan datan purkamiseen Java-luokan olioksi käyttäen Jackson-kirjastoa täytyy JSON-kentät määritellä luokan alussa. Noudetun JSON-datan kentät tallennetaan luokan vastaavasti nimettyihin kenttiin. Tästä johtuu myös se, että jos JSON-objektin sisällä on toinen JSON-objekti, täytyy sille luoda oma luokkansa, jossa objektin kentät sijaitsevat. Pollution-luokka pitää sisällään taulun Results-luokan olioita, joka taas pitää sisällään JSON-kenttien lisäksi Date ja Coordinates -oliot. Kuviossa 21 on esillä UML-kaavio, joka kuvaa rakennetta, johon vastaanotettu JSON-data puretaan.



Kuvio 21. UML-kaavio luokkarakenteesta, johon vastaanotettu JSON puretaan.

Lopuksi saapuneesta JSON-viestistä tuotetut oliot muokataan InfluxDB-tallennukseen sopiviksi ja julkaistaan yksi kerrallaan RabbitMQ-jonoon MQTT-viesteinä käyttäen mqttMessagePublisher-olion publishMessage-funktiota.

MqttMessagePublisher-luokka hyödyntää Eclipse Paho -kirjastoa, joka mahdollistaa helpon MQTT-viestien muodostamisen ja lähettämisen. Julkaistavaksi tuodun datan sisältävä olio muokataan uudelleen JSON-muotoiseksi MessageParser-luokan generateIotDataMessage-funktiolla. Tässä muutoksessa tärkeintä on luoda JSON, jossa määritellään myös mitkä vastaanotetut kentät päätyvät InfluxDB:ssä tageiksi ja mitkä kentiksi (ks. luku 4.5). Kun JSON on valmis, määritellään lähetettävän MQTT-viestin aihe luvussa 4.3 mainitun tiedostorakenteen mukaisesti ”`country/city/type`”. Tällöin RabbitMQ:sta dataa vastaanottava käsittelijä voi määritellä mitä tiettyä dataa se tahtoo noutaa. Lopuksi data lähetetään ja applikaation lokiin kirjoitetaan viesti lähetyksen sisällöstä (ks. kuvio 22).

```
2018-12-11 19:39:34.262  INFO 17802 --- [   scheduling-1] c.i.d.rabbitmq.MqttMes
sagePublisher       : Publishing message {"data":"{\"Value\":900.0,\"Latitude\":2
2.4137,\"Unit\":\"µg\\\/m³\",\"Longitude\":107.3476}","utc":1544544000000,"meta"
:"{\"tags\":[{\"id\":\"市环保局江州分局\"},{\"parameter\":\"co\"},{\"country\":\
"CN\"},{\"city\":\"崇左市\"}]}","type":"AirQualityData","uuid":"市环保局江州分局
"}
```

Kuvio 22. Data Producer -applikaation loki lähetettävästä MQTT-viestistä

Datan lähetys RabbitMQ-jonoihin suojataan itse allekirjoitetuilla TLS-sertifikaateilla (ks. luku 4.4). Sertifikaatit generoitiin käyttäen tls-gen työkalua (ks. kuvio 23). Generoidut sertifikaatit ca_certificate.pem, client_certificate.pem sekä client_key.pem kopioitiin omaan kansioonsa projektin sisällä ja määriteltiin projektin käyttöön resurssitiedostossa. On tärkeää muistaa jättää sertifikaatit pois versionhallinnasta, sillä niiden sisällyttäminen versionhallintaan on tietoturvariski. Sen sijaan sertifikaattien jakelu kehittäjien välillä tulisi toteuttaa jotakin tietoturvallisempaa reittiä.

```
jere@jere-VirtualBox:~/Projects/Iot_stack/backend$ mkdir misc
jere@jere-VirtualBox:~/Projects/Iot_stack/backend$ cd misc
jere@jere-VirtualBox:~/Projects/Iot_stack/backend/misc$ git clone https://github.com/michael
klishin/tls-gen tls-gen
Cloning into 'tls-gen'...
remote: Enumerating objects: 18, done.
remote: Counting objects: 100% (18/18), done.
remote: Compressing objects: 100% (14/14), done.
remote: Total 369 (delta 8), reused 10 (delta 4), pack-reused 351
Receiving objects: 100% (369/369), 92.37 KiB | 511.00 KiB/s, done.
Resolving deltas: 100% (211/211), done.
jere@jere-VirtualBox:~/Projects/Iot_stack/backend/misc$ cd tls-gen/
jere@jere-VirtualBox:~/Projects/Iot_stack/backend/misc/tls-gen$ cd basic/
jere@jere-VirtualBox:~/Projects/Iot_stack/backend/misc/tls-gen/basic$ make PASSWORD=[redacted]
```

Kuvio 23. TLS-sertifikaattien generointi

5.4 RabbitMQ

Projektin viestijonona toimiva RabbitMQ pyörii omassa Docker-kontissaan. Katsellaan aluksi RabbitMQ:n Dockerfile-määrityksiä (ks. kuvio 24). Pohjana käytetään rabbitmq:management Docker-imagea, joka sisältää RabbitMQ:n lisäksi sille kehitetyn graafisen järjestelmänhallinnan työkalun. Konttiin kopioidaan konfiguraatiotiedosto rabbitmq.conf, sekä sertifikaatit tiedonsiirron suojaamiseksi. Kontissa suoritetaan komento ”`rabbitmq-plugins enable --offline rabbitmq_mqtt`”, jolla aktivoidaan RabbitMQ:n MQTT-liitännäinen. Lopuksi määritellään avattavaksi RabbitMQ:n tarvitsemat portit.

Aiemmin mainitussa konfiguraatiotiedostossa määritellään sertifikaattien tiedostosijainnit, viestinnässä käytettävät portit sekä sallitaan pääsy järjestelmänhallintatyökaluun kontin ulkopuolelta.

```
# Use a docker image that has the management plugin
FROM rabbitmq:management

# Include rabbitMQ configuration file
COPY rabbitmq.conf /etc/rabbitmq/rabbitmq.conf

# To secure the transport of data, we use TLS. Certificates are fetched from files
COPY ./tls/ca/cacert.pem    /etc/rabbitmq/tls/ca/cacert.pem
COPY ./tls/server/cert.pem  /etc/rabbitmq/tls/server/cert.pem
COPY ./tls/server/key.pem   /etc/rabbitmq/tls/server/key.pem

# enable MQTT plugin, as we will be using mqtt protocol to move IoT data
# The offline option is often optimal for node provisioning automation.
RUN rabbitmq-plugins enable --offline rabbitmq_mqtt

# expose RabbitMQ TCP port
EXPOSE 5672

# Expose the port of the management tool
EXPOSE 15672

# Expose MQTT ports
EXPOSE 1883
EXPOSE 8883
```

Kuvio 24. RabbitMQ Dockerfile-tiedosto

Järjestelmähallintatyökalu sisältää myös komentolinjalta käytettävän rabbitmqadmin-työkalun, jonka toimimaan saaminen osoittautui hankalaksi. Komentolinjatyökalun noutamiseksi pitää RabbitMQ ensin käynnistää ja siirtyä selaimella osoitteeseen

http://localhost:15672/cli/index.html. Sivulta löytyy latauslinkki komentolinjatyökalulle. Työkalu sijoitetaan Docker-kontin ulkopuolelle ja sitä käytetään projektissa jonojen määrittämiseen komentorivin kautta.

Latauslinkin sisältävästä osoitteesta löytyvien ohjeiden mukaan komentorivityökalu vaatii toimiakseen Python version 2.6 tai uudemman. Tästä huolimatta, kun komentoriviä yritettiin käyttää, antoi työkalu virheen. Virhe ilmoitti, että python asennusta ei löytynyt. Tämä vaikuttaisi johtuvan siitä, että kehityskoneella käytettiin Python 3 asennusta. Työkalun tiedoston alussa oleva määritys ”`#!/usr/bin/env python`” tuli korvata määrityksellä ”`#!/usr/bin/env python3`”.

Komentorivityökalua hyödynnetään backend-järjestelmien käynnistämiseen käytettävässä komentoriviskriptissä create_rabbitmq_queues.sh, jossa luodaan jokaiselle ilmansaastetyypille oma jononsa (ks. Kuvio 25). Jonoja voidaan luoda myös graafisen järjestelmänhallistatyökalun kautta, mutta ohjelmallinen automatisointi on helposti toistettavissa useammassa järjestelmässä, jolloin se on parempi vaihtoehto. Jonon luontikomento palauttaa message-muuttujaan vastauksen luonnin onnistumisesta ja jos se sisältää termin ”queue declared” voidaan todeta jonon luonnin onnistuneen.

```
echo -n "Creating RabbitMQ message queue pm25: "
message=`./rabbitmqadmin --host=localhost --port=15672 \
--username=$QUEUE_USERNAME --password=$QUEUE_PASSWORD \
declare queue name=pm25`

if [[ $message == *"queue declared"* ]]; then
    echo "[OK]"
else
    echo "$message [ERROR]"
fi
```

Kuvio 25. RabbitMQ-jonon luominen komentorivityökalulla

Tämän jälkeen saapuvan MQTT-datan aiheet tulee vielä reitittää oikeisiin jonoihin. Tätä varten komentolinjatyökalulla määritetään reitit ykset sekä luodaan liitos jonojen välille (ks. kuvio 26). Komentorivityökalulle annetaan tärkeimpinä argumentteina reititysavain sekä kohdejono. Tallennetaan message-muuttujaan jälleen vastaus

luonnin onnistumisesta ja jos se sisältää termin ”binding declared” voidaan todeta jonon luonnin onnistuneen. Reititysten määrittelystä sekä useammasta jonosta saavutettavista hyödyistä voi lukea lisää luvussa 4.4

```
echo -n "Creating binding[MQTT->pm25]: "
message=`./rabbitmqadmin --host=localhost --port=15672 \
--username=$QUEUE_USERNAME --password=$QUEUE_PASSWORD \
declare binding source="amq.topic" destination=pm25 routing_key="*.*.pm25"`
if [[ $message == *"binding declared"* ]]; then
    echo "[OK]"
else
    echo "$message [ERROR]"
fi
```

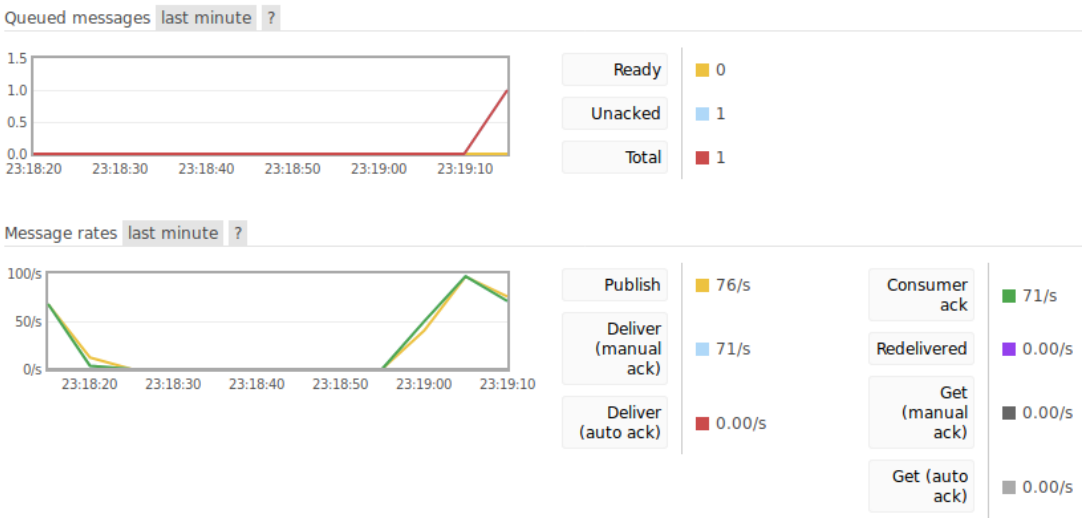
Kuvio 26. RabbitMQ-reitityksen luominen ja jonoon liittäminen

Kun RabbitMQ-kontti on käynnistetty ja jonot sekä reitit määritetty, ollaan valmiita vastaanottamaan viestejä Data producer -applikaatiolta. Helpoin tapa seurata viestien vastaanottamista sekä lähettämistä eteenpäin käsittelijälle on graafisen järjestelmänhallintatyökalun kautta. Työkaluun päästiin verkkoselaimella osoitteesta http://localhost:15672. Mielenkiintoisin tieto löytyi queues-välilehden takaa, josta nähtiin aiemmin luodut jonot sekä niiden tilat. Kuviossa 27 on esillä tilanne, jossa uusia viestejä on juuri saapumassa jonoihin ja niitä toimitetaan samanaikaisesti käsittelijäapplikaatiolle. Dataa lähettäessä viestejä saapui noin 323 sekunnissa ja niitä toimitettiin samalla nopeudella. Tässä tilanteessa viestien kulkunopeutta rajoittaa todennäköisimmin tuottaja- ja käsittelijäapplikaatiot, sillä RabbitMQ kykenisi huomattavasti suurempiinkin viestinopeuksiin.

Overview			Messages			Message rates		
Name	Features	**State**	**Ready**	**Unacked**	**Total**	**incoming**	**deliver / get**	**ack**
co	D	running	0	0	0	25/s	29/s	29/s
no2	D	running	0	0	0	85/s	85/s	85/s
o3	D	running	0	0	0	53/s	52/s	52/s
pm10	D	running	0	0	0	63/s	64/s	64/s
pm25	D	running	0	0	0	56/s	50/s	50/s
so2	D	running	0	0	0	41/s	44/s	44/s

Kuvio 27. RabbitMQ-jonojen taulukko järjestelmänhallintatyökalussa

Jonojen taulukon linkeistä päästään siirtymään myös yksittäisen jonon näkymään. Yksittäisen jonon näkymässä on esillä yksityiskohtaista tietoa jonosta. Työssä yksittäisen jonon prosessi käytti muistia toimettomassa tilassa noin 17 kilotavua ja viestejä käsitellessä noin 3 megatavua. Näkymässä piirretään graafi jonossa odottavista viesteistä sekä käsitellyistä viesteistä (ks. kuvio 28). Jonossa odottavien viestien graafi pysyi työn jonoissa lähes aina nollassa, sillä käsittelijä vastaanotti viestit lähes välittömästi. Jos käsittelijä jostakin syystä lopettaisi viestien vastaanottamisen, näkyisi graafissa odottavat viestit. Näkymässä on näkyvillä myös yhdistetyt käsittelijät IP-osoitteineen sekä liitetyt reitit ykset. Näkymässä on mahdollista julkaista jonoon viesti käsin tekstikentän kautta. Tämä ominaisuus on kätevä esimerkiksi käsittelijän toiminnan testaamisessa ilman aktiivista datan tuottajaa.



Kuvio 28. Yksittäisen jonon näkymän graafi viestien kulusta.

5.5 Data consumer

Data consumer on Java Spring Boot -sovellus, jonka tarkoituksena on käsitellä RabbitMQ-viestijonosta saapuvia viestejä ja tallentaa ne InfluxDB-tietokantaan. Data producer -sovelluksen tapaan myös Data consumer koostetaan JAR-sovellukseksi. Ainoa ero sovelluksen suorittamisen näkökulmasta on se, että consumer suoritetaan oman Docker-konttinsa sisällä.

Ohjelmiston käynnistys sovelluskoodissa tapahtuu samanlaisen Application-luokan kautta kuin producer-sovelluksessa (ks. luku 5.3). Keskitytään aluksi ConsumerConfiguration-luokkaan, jossa määritellään RabbitMQ-viestien käsittely. Konfiguraatiotiedostossa tärkeimpiä ovat @Bean-annotaatiolla varustetut funktiot.

@Bean-funktioista Spring osaa alustaa olioita sovelluksen, joita se itse hallinnoi taustajärjestelmissä. Data consumerissa alustettavat @Bean-funktiot käyttävät Spring Frameworkin AMQP-kirjastoa, johon sisältyy myös RabbitMQ tuki. connectionFactory-funktiossa alustetaan RabbitMQ-yhteyden luonnissa käytettävät osoite, käyttäjänimi sekä salasana. Tätä hyödynnetään rabbitTemplate-funktiossa, jossa konfiguroidaan RabbitMQ-viestien käsittelytapa (ks. kuvio 29). Funktiossa määritellään RabbitMQ-yhdistämiseen aiemmin käsitelty connectionFactory-olio. Reititysavaimena käytetään jokerimerkkiä ”#”, joka ohjaa kaikkien MQTT-aiheiden liikenteen tälle käsittelijälle. Viestit määritellään muutettavaksi JSON-muotoisiksi.

```
@Bean
public RabbitTemplate rabbitTemplate() {
    RabbitTemplate template = new RabbitTemplate(connectionFactory());
    template.setRoutingKey("#");
    template.setMessageConverter(jsonMessageConverter());
    return template;
}
```

Kuvio 29. rabbitTemplate-funktio.

Tämän jälkeen määritellään RabbitMQ-viestikuuntelija simpleMessageListenerContainer-funktiossa. Funktiossa määritellään jonot, joista noudetaan dataa. Erittäin tärkeää funktiossa on käytettävän viestikuuntelijan määrittäminen metodilla setMessageListener (ks. kuvio 30). Metodille annetaan argumenttina uusi instanssi MessageConsumer-luokasta, jossa varsinainen viestin käsittely tapahtuu. Nyt kun viestikuuntelijaksi on asetettu MessageConsumer-luokka, kutsutaan luokan sisällä onMessage-funktiota, aina kun uusi viesti vastaanotetaan RabbitMQ-jonosta.

```
@Bean
public SimpleMessageListenerContainer container() {
    SimpleMessageListenerContainer container = new SimpleMessageListenerContainer();
    ConnectionFactory connectionFactory = connectionFactory();
    container.setConnectionFactory(connectionFactory);
    container.setQueueNames("co", "pm25", "o3", "no2", "so2", "pm10");
    container.setMessageListener(getMessageListener());
    container.setMessageConverter(jsonMessageConverter());
    container.setConcurrentConsumers(getNumberOfConsumers());
    container.setMaxConcurrentConsumers(getMaxConcurrentConsumers());

    log.info("Hostname {}", connectionFactory.getHost());
    log.info("Port {}", connectionFactory.getPort());

    return container;
}

@Bean
public MessageListener getMessageListener() { return new MessageConsumer(); }
```

Kuvio 30. Viestikuuntelijan määrittely

MessageConsumer-luokan onMessage-funktiolle tuodaan vastaanotettu viesti JSON String -muotoisena argumenttina. Data tulee ensimmäisenä muuntaa JSON-merkkijonosta jälleen Java-olioksi. Vastaanotettu JSON kartoitetaan IotData-olioksi käyttäen samaa tekniikkaa kuin Data producer-sovelluksessa kuvatussa JSON-kartoituksessa (ks. luku 5.3).

Kun IotData-olio on luotu, olisi mahdollista tässä välissä tehdä dataan muutoksia tai lisäyksiä. Voitaisiin esimerkiksi suorittaa laskentaa ennen tallentamista tietokantaan. Tässä työssä muutokset eivät kuitenkaan ole tarpeellisia, joten tiedon tallennus tietokantaan voidaan aloittaa. MessageConsumer-luokkaan on @Autowired-annotaatiota hyödyntäen alustettu DatabaseService-luokan olio, joka on vastuussa tiedon tallennuksesta. DatabaseService käyttää Java InfluxDB -kirjastoa, jonka avulla tietokantaan yhdistäminen ja kirjoittaminen helpottuu huomattavasti.

DatabaseService-luokan alussa alustetaan InfluxDB-luokan olio influxDB. Seuraavaksi määritellään tietokantaan yhdistämiseen tarvittavat tiedot; tietokannan osoite, tietokannan nimi, käyttäjänimi sekä salasana. InfluxDB-tietokantaan luodaan yhteys connect-funktiossa käyttäen aiemmin määriteltyjä kenttiä sekä luodaan ”target”-nimi-nen tietokanta, jos sitä ei vielä ole olemassa. Tietokantaan kirjoittamiseen käytetyssä write-funktiossa luodaan Point-luokan olio p, joka on käytännössä InfluxDB-tallen-

nukseen sopiva datapiste. Datapisteeseen tallennetaan mittapiste IotData-oliosta, johon kuuluu aikaleima, tagit ja kentät. Jos InfluxDB:n tagit ja kentät eivät ole tuttuja voi niistä lukea luvussa 4.5.

Consumer sovelluksen Docker-kontin pohjana käytettiin maven:3.5-jdk-8-alpine Docker-imagea. Ideana oli aluksi koostaa JAR-tiedosto kontin sisällä käyttäen maven-työkalua, mutta tämä osoittautui hitaaksi ratkaisuksi. Sen sijaan JAR-tiedosto koostetaan ennen kontin luomista komennolla ”`mvn clean install`”. Sovelluksen Docker-filessä määritellään, että JAR-tiedosto kopioidaan kontin sisään ja tämän jälkeen sovellus käynnistetään kontin sisällä komennolla ”`java -jar -Dserver.port=8081 /data_consumer/data_consumer-1.0.0.jar`” (ks. kuvio 31).

```
# Define baseimage
FROM maven:3.5-jdk-8-alpine

WORKDIR      /data_consumer

# Copy previously compiled JAR inside the container.
ADD target/data_consumer-1.0.jar /data_consumer/data_consumer-1.0.0.jar

# Specify containers startup command
CMD java -jar -Dserver.port=8080 /data_consumer/data_consumer-1.0.0.jar
```

Kuvio 31. Dockerfile-tiedosto Data consumer -sovellukselle

5.6 InfluxDB ja backend Docker Compose

Backendin viimeinen osa on InfluxDB-tietokanta. InfluxDB pyörii omassa Docker-kontissaan, jonka Dockerfile on äärimmäisen yksinkertainen. Dockerfilessä määritellään vain kontin luonnissa käytettävä Docker-image ”influxdb” (ks. kuvio 32). Tämä image sisältää esikonfiguroidun InfluxDB-instanssin, joka käynnistyy kontin mukana automaattisesti. Tämä tarkoittaa myös sitä, että InfluxDB-tietokantaan yhdistetään oletuskäyttäjätunnuksella ja salasanalla, joten tuotantokäyttöön olisi hyvä Dockerfilessä asettaa ne tietoturvallisemmiksi.

```
# All we need to do here is to grab the influxdb docker image.
FROM influxdb
```

Kuvio 32. InfluxDB-kontin luonnissa käytettävä Dockerfile

Kaikille backend-palveluille on määritelty omat Dockerfile-tiedostot, joissa on määritelty, miten Docker-kontit rakennetaan. Lisäksi työssä kaikkien konttien yhtäaikainen käynnistäminen orkestroitiin käyttäen Docker Compose -työkalua. Huomaa, että Docker Compose sitoo myös rakennettavat kontit jaettuun verkkorakenteeseen. Tästä syystä verkkorakenteeseen pitää avata portteja palveluiden välille.

Tiedostossa docker-compose-backend.yml on määritykset jokaiselle palvelulle. Palveluille tulee vähintään määrittää kansio, jossa Dockerfile sijaitsee. Jokaiselle palvelulle määritellään koostetun kontin nimeksi ja yhteisessä verkossa käytettäväksi isäntänimeksi sama nimi kuin palvelulla itsellään, esim. RabbitMQ. Dockerfileä käytetään siis edelleen itse Docker-kontin koostamiseen.

RabbitMQ-palvelun määrityksissä avataan portit 5672 (TCP), 15672 (graafinen järjestelmänhallintatyökalu), 8883 (MQTT) ja 1883 (MQTT) avataan kontilta ulkoverkkoon. InfluxDB-palvelun määrityksissä avataan tietokannan käyttämät oletusportit 8086 ja 8083. Lisäksi määritellään data volume. Volume-määrityksessä Docker-kontin sisällä oleva ”`/var/lib/influxdb`”-kansio tallennetaan myös isäntäkoneelle ”`./data/influxdb`”-kansioon. Tallennettava kansio sisältää Tietokannan datan. Luvussa 4.7 on kerrottu lisää Docker-datavarastoinnista.

Data consumer-palvelulle määritellään ympäristömuuttuja ”`RABBITMQ_HOST=rabbitmq`” sekä ”`DATABASE_HOST=influxdb:8086`”. Näillä ympäristömuuttujilla Data consumer -applikaatiolle asetetaan osoitteet, joilla se yhdistää muihin palveluihin. Näissä hyödynnetään aiemmin muihin palveluihin asetettuja isäntänimiä. Isäntänimien avulla esimerkiksi InfluxDB löytyy Docker Composen luomassa verkossa osoitteesta influxdb:8086. Jotta isäntänimet saa Data consumer -palvelulle näkyviin, täytyy sille määritellä linkit RabbitMQ- ja InfluxDB -palveluihin. Lopuksi määritellään tallennettavien lokien maksimikooksi 100 megatavua. Kuviossa 33 on esillä Data consumer -palvelun määrittely.

```
# DATA CONSUMER SERVICE DEFINITIONS
data_consumer:
  build: data_consumer/
  container_name: data_consumer
  environment:
    - RABBITMQ_HOST=rabbitmq
    - DATABASE_HOST=influxdb:8086
  links:
    - rabbitmq
    - influxdb
  logging:
    options:
      max-size: 100m
```

Kuvio 33. Docker Composen Data consumer -palvelun määrittely

Backendin käynnistysprosessin helpottamiseksi luotiin komentoriviskripti ”start_backend_stack.sh”. Skriptin päätarkoituksena on ajaa komento ”sudo docker-compose -f docker-compose-backend.yml up -d”, joka koostaa Docker-kontit ja käynnistää ne. Tämän jälkeen odotetaan 10 sekuntia, jotta RabbitMQ-kontti ehtii käynnistyä ja luodaan ”create_rabbitmq_queues”-skriptillä RabbitMQ-viestijonot sekä niiden reitit ykset. Käynnistyneiden Docker-konttien tilan voi Linux-terminaalissa tarkistaa komennolla ”docker ps” (ks. kuvio 34).

```
jere@jere-VirtualBox:~$ docker ps
CONTAINER ID        IMAGE                  COMMAND                  CREATED             STATUS
NAMES
4e12619022b0        backend_data_consumer  "/usr/local/bin/mvn-…"   26 minutes ago      Up 19 minutes
data_consumer
b14f013ba110        backend_rabbitmq       "docker-entrypoint.s…"   26 minutes ago      Up 26 minutes
 5671/tcp, 0.0.0.0:8883->8883/tcp, 15671/tcp, 0.0.0.0:15672->15672/tcp, 25672/tcp   rabbitmq
6a7c3f93d1fe        backend_influxdb       "/entrypoint.sh infl…"   26 minutes ago      Up 26 minutes
influxdb
1e8e4809ec5d        frontend_grafana       "/run.sh"                4 weeks ago         Up 6 days
grafana
```

Kuvio 34. ”docker ps” -komento

Docker-konttien tuottamia lokeja voidaan seurata komentoriviltä. Projektissa seurailtiin eniten Data consumer -kontin lokeja komennolla ”docker logs -f data_consumer --tail 200”. Komennossa ”-f”-parametri määrittää, että lokeja halutaan seurata reaaliaikaisesti, jolloin terminaaliruutu päivittyy aina uusien lokimerkintöjen saapuessa. ”--tail 1000” -parametrilla määritetään, että lokeista näytetään vain 1000 viimeisintä merkintää. Tämä parantaa suorituskykyä estämällä vanhojen lokimerkintöjen noutamisen turhaan.

Kehityksen aikana nousi tarve päästä komentorivillä myös suorittamaan komentoja Docker-konttien sisällä. Esimerkiksi InfluxDB-kontin sisälle pääseminen onnistuu komennolla ”`docker exec -t -i influxdb /bin/bash`”. Kontin sisällä voi tutkia tietokannan sisälle tallennettua dataa (ks. kuvio 35).

```
root@influxdb:/# influx
Connected to http://localhost:8086 version 1.7.0
InfluxDB shell version: 1.7.0
Enter an InfluxQL query
> use target
Using database target
> SELECT "Value" FROM AirQualityData GROUP BY country,parameter LIMIT 10
name: AirQualityData
tags: country=AD, parameter=co
time                Value
----                -----
1542128400000000000 200
1542132000000000000 400
1542135600000000000 500
1544090400000000000 600
```

Kuvio 35. InfluxDB-datan tutkiminen tietokantakyselyllä Docker-kontissa.

5.7 Grafana ja datan visualisointi

Datan visualisointi on toteutettu Grafana-analysointityökalulla. Grafana on työn ainoa frontend-komponentti ja se pyörii oman Docker-konttinsa sisällä. Grafanan Dockerfilessä määritellään kontin sisään kansioon ”`/etc/grafana/provisioning/datasources`” kopioitavaksi tiedosto ”`datasource.yaml`”. Tässä tiedostossa määritellään Grafanalle datalähde (ks. kuvio 36). Tässä tilanteessa datalähteen tyyppi on InfluxDB ja yhdistettävän tietokannan nimi on ”target”. Lisäksi määritellään esimerkiksi tietokannan osoite, käyttäjätunnus ja salasana. Datalähde voitaisiin määritellä käsin myös Grafanan käyttöliittymässä kontin käynnistyksen jälkeen, mutta tiedostolla tehty määrittely on luotettavampi tapa määritellä datalähde, koska määrittely säilyy tiedostossa ja Grafana voidaan täten helposti monistaa.

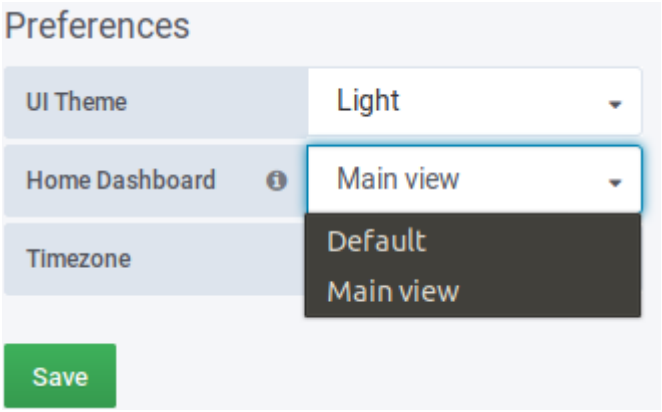
```
apiVersion: 1

datasources:
  - name: InfluxDB
    type: influxdb
    access: direct
    database: target
    user: root
    password: root
    url: http://localhost:8086
```

Kuvio 36. Grafanan datalähteen määritys datasource.yaml-tiedostolla

Grafana käynnistetään backend-palveluiden tapaan Docker Composen avulla. Grafanan docker-compose-frontend.yml-tiedoston määritykset ovat hyvin samantapaisia kuin luvun 5.6 docker-compose.backend.yml.tiedostossa. Tärkeintä on tietää, että docker-composessa määritellään Grafanalle data volume, jossa tallennetaan data isäntäkoneelle. Grafanaan luodut näkymät säilyvät siis vaikka kontti sammutettaisiin.

Lyhyt ”`start_frontend.sh`”-komentoriviskripti luo data volumea varten kansion isäntäkoneelle ja suorittaa komennon ”`sudo docker-compose -f docker-compose-frontend.yml up -d`”. Komennon valmistuessa on Grafana käynnistynyt osoitteessa “localhost:3000”. Sivulle siirtyessä pyytää Grafanan sisäinen käyttäjänhallinta käyttäjää kirjautumaan sisään. Oletuskäyttäjätunnus sekä salasana on ”admin”. Ensimmäisen kirjautumisen jälkeen Grafana pyytää käyttäjää asettamaan tietoturvallisemman salasanan.

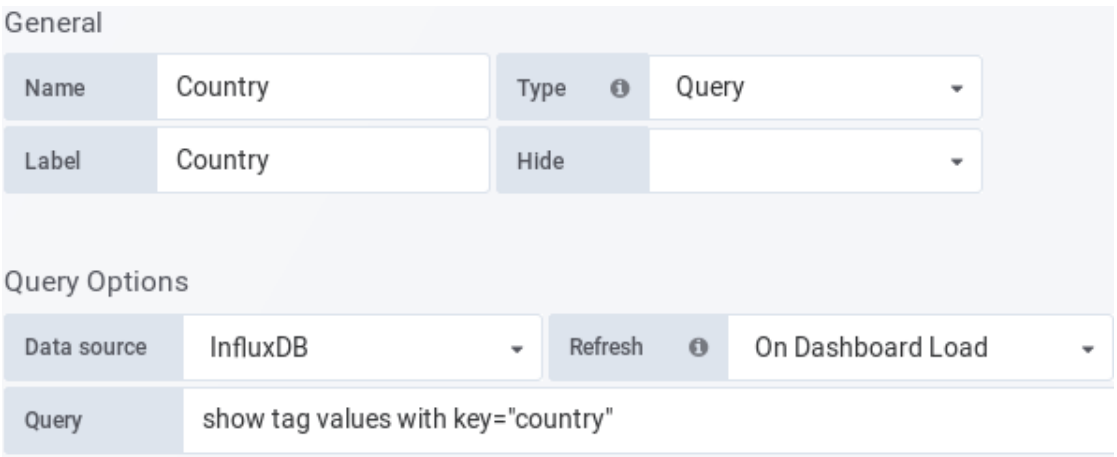
Kirjautumisen jälkeen ohjaudutaan Home-hallintapaneeliin. Kyseessä on oletushallintapaneeli, joka on työssä korvattu Main View -hallintapaneelilla. Oletushallintapaneeli voidaan vaihtaa Configuration-valikon preferences-välilehden Home Dashboard -asetuksesta (ks. kuvio 37).



Kuvio 37. Grafanan oletushallintapaneelin vaihtaminen

Työn päähallintapaneeli on yksinkertainen näkymä, joka sisältää linkit eri ilmansaastetyyppien hallintapaneeleihin. Kutsutaan tätä selkeyden vuoksi päävalikoksi. Päävalikkoon pääsee muista hallintapaneeleista aina vasemmalla sijaitsevan sivupaneelin Dashboards-näppäimestä. Kaikki ilmansaasteiden hallintapaneelit ovat asettelultaan samanlaiset ja sisältävät samat tiedot eri saastetyypeille. Tutkitaan esimerkkinä typpioksidinäkymää ja sen paneeleita.

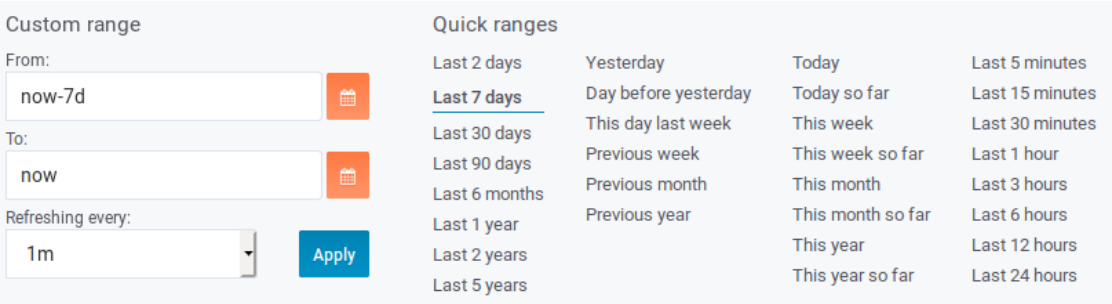
Typpioksidinäkymän yläosassa sijaitsee Country-valitsin. Kyseessä on Grafanan sisällä luotu muuttuja, jolla voidaan määrittää minkä maan tietoja esitetään paneeleissa. Muuttuja luodaan hallintapaneelin Asetuksien Variables-välilehdessä (ks. kuvio 38). Muuttujia voi olla useanlaisia, esimerkiksi tietokantakyselyn palauttamia tai käsin asetettuja arvoja. Tässä tilanteessa ”Country”-muuttuja saa arvonsa tietokantakyselystä. Muuttujalle annetaan nimi, tietolähde ja tietokantakysely. Kysely *”show tag values with key="country"”* palauttaa kaikki InfluxDB-tietokantaan tallennetut ”country”-tagin arvot



Kuvio 38. Country-muuttujan luonti

Alkuperäinen suunnitelma oli kerätä kaikki päästötyypit yhteen hallintapaneeliin ja lisätä muuttuja, jolla voitaisiin määrittää mitä päästötyyppiä esitetään paneeleissa. Tämä toteutus toimi muuten hienosti, mutta karttanäkymän värikoodausten raja-arvoja ei pysty muuttamaan muuttujien perusteella. Päästötyyppien karttanäkymän raja-arvoina käytettiin Euroopan ympäristökeskuksen käyttämiä arvoja (Air quality index n.d). Maailmanlaajuisen datan raja-arvojen määrittäminen on hankalaa, koska esimerkiksi Kiinalla on omat raja-arvonsa, jotka ovat Euroopan raja-arvoja korkeammat (China: Air quality standards n.d). Päästötyyppien jakaminen omiin hallintapaneeleihin mahdollistaa myös sen, että käyttäjien pääsyä voidaan hallita hallintapaneelikohtaisesti. Täten voidaan rajata tietoa pois käyttäjiltä, jotka eivät sitä tarvitse tai saisi nähdä.

Hallintapaneeleiden aika-akselia voi muokata ikkunan oikeassa yläkulmassa sijaitsevasta kellopainikkeesta. Painike avaa ikkunan, jonka kautta tehdään määritykset (ks. kuvio 39). Ikkuna tarjoaa valmiita aikarajoituksia, esimerkiksi viimeiset kuusi tuntia, setisemän päivää tai kaksi vuotta. Aika-akselin voi määritellä myös itse kalenterin avulla. Samasta ikkunasta voidaan määritellä myös se, kuinka usein paneelit päivittyvät automaattisesti. Vaihtoehtoja on esimerkiksi 10 sekuntia, yksi minuutti tai kaksi tuntia. Lyhyellä päivitysajalla hallintapaneeli saadaan siis käytännössä reaaliaikaiseksi.

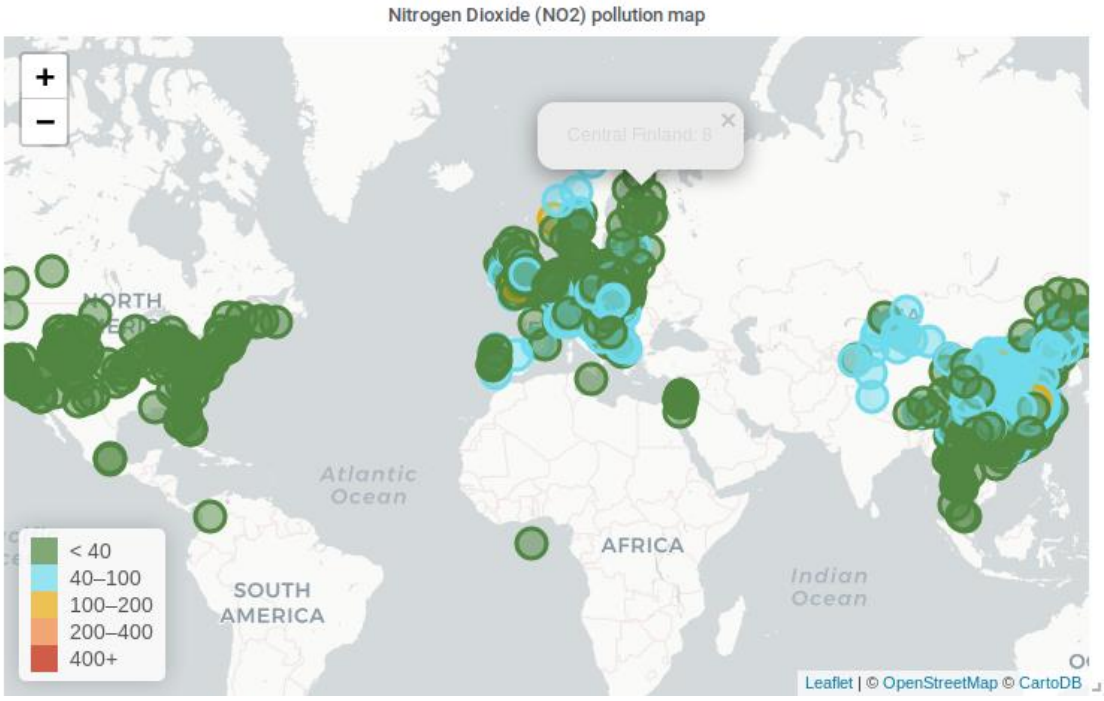


Kuvio 39. Hallintapaneelin aika-asetukset

Karttanäkymä on esillä kuviossa 40. Kuvion kartassa esitetään kaikki typpioksiditietoa keräävät mittalaitteet, joiden dataa on saapunut tietokantaan. Sijainti määritellään tietokantaan tallennetuilla koordinaateilla. Mittapisteitä on hurja määrä ja onkin sel-

vää, että aiemmin mainittu Country-muuttuja tuli tarpeeseen. Tällä muuttujalla voidaan rajata kartalta pois kaikki muut paitsi muuttujassa valitut maat. Muuttujaan voi siis valita yhden tai useamman maan näytettäväksi.

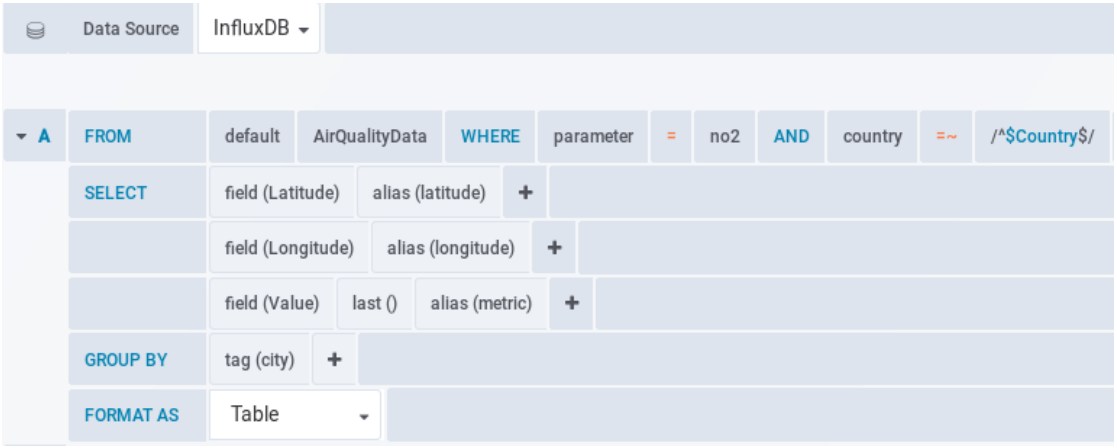
Sijoittamalla hiiren osoitin kartan mittapisteen päälle, näytetään mittapisteen viimeisin arvo. Työssä käytetty kirkas teema aiheutti ongelman arvon esittämisessä. Teksti ja tausta olivat hyvin saman sävyiset, jolloin teksti oli vaikealukuista. Pohjois-Amerikasta saapui dataa suurelta määrältä mittapisteitä, mutta niiden arvo oli 0. Täten ei voitu pitää Pohjois-Amerikasta saapuvaa dataa luotettavana. Euroopan ja Aasian maiden mittapisteissä arvot vaikuttivat olevan oikeellisia.



Kuvio 40. Karttanäkymä kerätystä typpioksididatasta

Karttapaneelin sekä muiden paneelien muokkaaminen onnistuu edit-valikossa. Valikon Metrics-välilehdellä muodostetaan tietokantakysely (ks. kuvio 41). Grafana tarjoaa InfluxDB-kyselyihin työkalun, joka mahdollistaa kyselyn luonnin graafisella käyttöliittymällä tekstin sijaan. Käyttöliittymä osaa ehdottaa tietokannasta haettavia tageja ja kenttiä, jolloin niitä ei tarvitse itse kirjoittaa. Käyttöliittymä ei kuitenkaan pysty toteuttamaan kaikkia mahdollisia tietokantakyselyitä ja tätä varten voidaan siirtyä edit-tilaan. Tässä tilassa tietokantakysely kirjoitetaan kokonaan käsin. Karttanäkymään haetaan Kentät Latitude, Longitude ja viimeisin Value-kenttä. Haetuille kentille asetetaan alias, jonka avulla karttapaneeli löytää datan. Haku rajataan niin että tagin

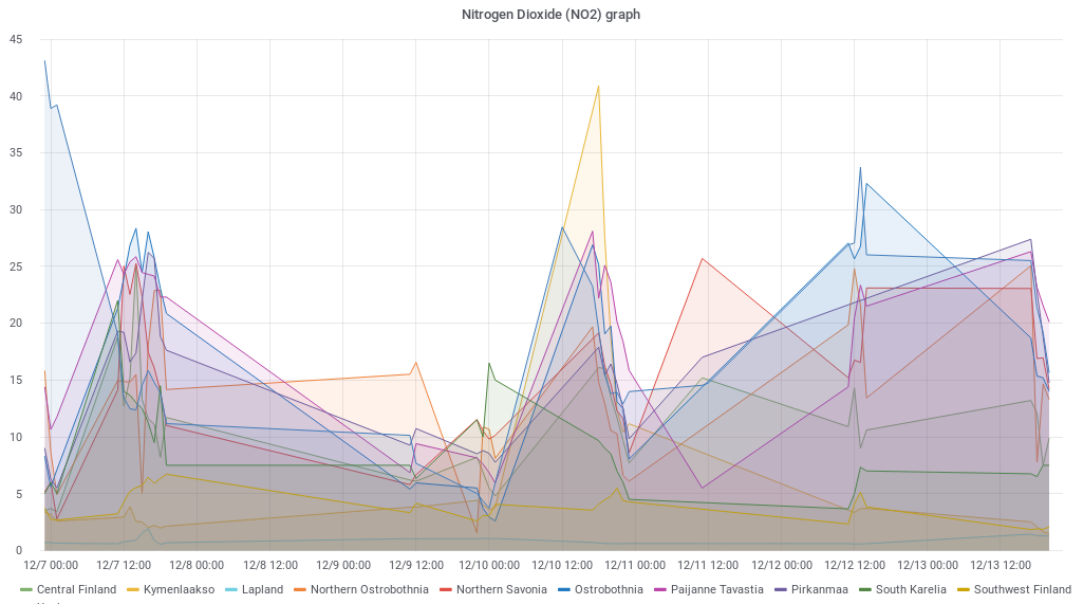
”parameter” tulee olla ”no2” ja tagin ”country” tulee olla Country-muuttujassa valittuna.



Kuvio 41. Typpioksidinäkymän karttapaneelin tietokantakysely

Kyselyn luomisen jälkeen konfiguroitiin itse graafinen kartta. Tämä tapahtuu ”Worldmap”-välilehdeltä. Ensimmäisenä määritellään kartan visuaaliset asetukset, kuten kartalle tuotettavien datapisteiden merkkien koko. Seuraavaksi kartoitetaan tietokantakyselyssä noudettu data kartan pisteisiin. Tässä hyödynnetään tietokantakyselyssä kentille asetettuja alias-nimiä. Datapiste saa nimen, arvon sekä koordinaatit. Lopuksi määritellään kartalle piirrettävien pisteiden värit ja niiden raja-arvot.

Kerätystä datasta piirrettiin graafi omaan paneeliinsa (ks. kuvio 42). Graafin tarkoituksena on kuvata ilmansaastekehitystä määritellyllä aikavälillä. Paneeli vaatii yksinkertaisen tietokantakyselyn, jossa noudetaan ”Value”-kentät hallintapaneelissa määritetyltä aika-akselilta. Data rajoitettiin samantyyppisesti kuin karttapaneelissa maan ja ilmansaastetyypin mukaan. Graafiin piirretyt viivat nimettiin kaupunkien mukaan.



Kuvio 42. Suomen typpioksidiarvoista piirretty graafi

Hallintanäkymän viimeiseen paneeliin kerättiin taulukkonäkymään viimeisimmät mittalaittaitteiden datapisteet (ks. kuvio 43). Paneelin käyttötarkoitus on siis kuvata viimeisimpiä tietoja, kun taas graafin piirtävä paneeli kuvaa muutoksia ajan kuluessa. Taulukkoon kerätään mittauksen aika, kaupunki, id, mitta-arvo sekä mittayksikkö. Tietokantakyselyssä on noudettu aikeisemmin mainittuja tietoja vastaavat kentät ja tagit.

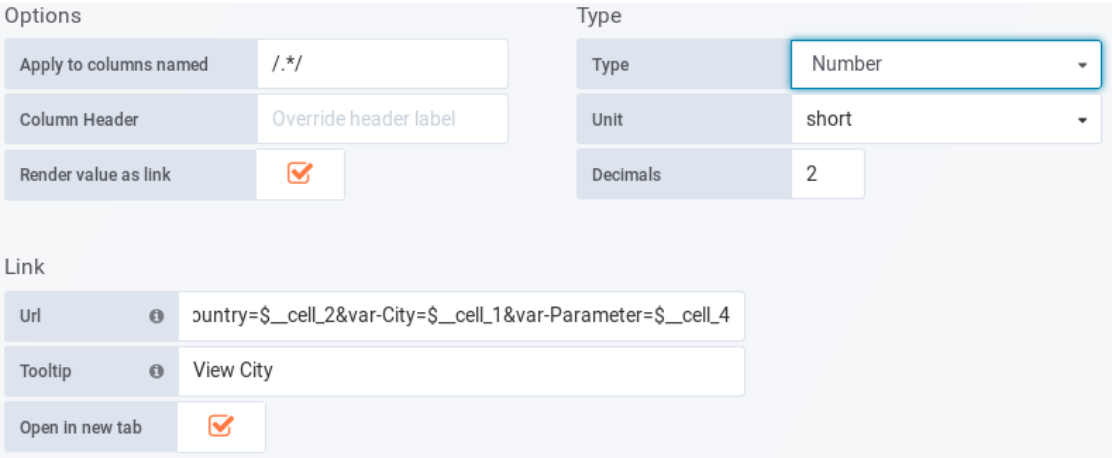
Latest values

Time	city	id	Value	Unit
2018-12-13 22:00:00	Ålesund	Karl Eriksens plass	32.79	µg/m³
2018-12-13 22:00:00	Trondheim	Elgeseter	54.07	µg/m³
2018-12-13 22:00:00	Trondheim	E6-Tiller	75.80	µg/m³
2018-12-13 22:00:00	Trondheim	Bakke kirke	44.63	µg/m³
2018-12-13 22:00:00	Tromsø	Hansjordnesbukta	19.93	µg/m³
2018-12-13 22:00:00	Szabolcs-Szatmár-Bereg	HU0028A	40.29	µg/m³
2018-12-13 22:00:00	Stavanger	Våland	4.34	µg/m³
2018-12-13 22:00:00	Stavanger	Schancheholen	7.38	µg/m³
2018-12-13 22:00:00	Stavanger	Kannik	27.60	µg/m³
2018-12-13 22:00:00	Sjælland	DK0012R	6.67	µg/m³
2018-12-13 22:00:00	Sarpsborg	Alvim	20.25	µg/m³
2018-12-13 22:00:00	Oslo	Åkebergveien	28.51	µg/m³

1 2 3 4 5 6 7 8 9

Kuvio 43. Taulukkopaneeli viimeisimmistä saapuneesta datasta

Jos tietty arvo taulukossa kiinnostaa ja mittalaitetta haluaa tutkia tarkemmin, luotiin tätä varten taulukkoon linkit yksittäisen mittalaitteen tietoihin. Tämä tapahtuu paneelin ”edit”-valikon ”Column Styles”-välilehdeltä (ks. kuvio 44). Täällä kolumneille voidaan määritellä esimerkiksi otsikko sekä datatyyppi. Kolumnit tunnistetaan säännöllisellä lausekkeella. Linkki halutaan lisätä jokaiseen kolumniin, joten säännöllinen lauseke on ”/.*/”. Linkkiin määritellään yksittäisen mittalaitteen hallintapaneelin osoite, johon annetaan lisäksi argumenteiksi maa, kaupunki ja saastetyyppi. Argumentteihin saadaan taulukosta arvo käyttämällä tunnistetta ”$__cell_1”. Tunnisteen numerolla määritetään minkä taulukon kolumnin arvo haetaan. Linkille annetaan ”Tooltip”, joka näyttää määritellyn tekstin hiiren osoittimen ollessa taulukon linkin päällä. Valittu linkki saadaan avautumaan uudessa välilehdessä ”Open in new tab” -valinnalla.



Kuvio 44. Taulukon linkin määrittäminen

Linkistä klikkaamalla siirrytään yksittäisen mittalaitteen hallintapaneeliin. Tämä hallintapaneeli sisältää samanlaiset paneelit kartan ja graafin piirtämiselle kuin edellinen näkymä, mutta esittää tietoja vain taulukossa valitulta mittalaitteelta. Linkissä asetetut argumentit sijoitetaan hallintapaneelin Country, City, Parameter ja id-muuttujiin, joita käytetään tietokantakyselyjen rajauksessa (ks. kuvio 45). Muuttujien arvoja voi itse muuttaa halutessaan. Moni kaupunki sisälsi useamman kuin yhden mittalaitteen. Kaikki kaupungin mittalaitteet saa näkyviin vaihtamalla id-muuttujan arvoksi ”all”.

Kuvio 45. Hallintapaneelin muuttujat

6 Pohdinta

Opinnäytetyön päätavoitteena oli tarjota kokonaiskuva IoT-projektin toteuttamisesta ja luoda pohja mahdolliselle tulevalle IoT-kokonaisuudelle. Työn toteutuksen jälkeen voidaan todeta, että tavoite saavutettiin. Luotu järjestelmä kerää, varastoi ja esittää datan loppukäyttäjälle. Ymmärrys datan liikkumisesta järjestelmien välillä ja IoT-laitteiden toiminnasta laajeni työnteon aikana huomattavasti. Myös aikasarjatietokantoihin perehtyminen osoittautui IoT-projektien kannalta hyödylliseksi. Järjestelmä keräsi työn toteuttamisen aikana tietokantaan arvoja mittalaitteilta yhteensä 1 958 641 kappaletta (ks. kuvio 46).

```
> select count(Value) from AirQualityData
name: AirQualityData
time count
---- -----
0    1958641
```

Kuvio 46. InfluxDB-tietokantaan tallennettujen datapisteiden kokonaismäärä

Voidaan siis todellakin todeta, että järjestelmää kuormitettiin samantyyppisillä datamäärillä kuin mitä tyypillisissä IoT-järjestelmissä käsitellään. Suuresta kerätystä datamäärästä huolimatta InfluxDB kirjoitti ja palautti dataa erittäin nopeasti. InfluxDB:n datan kompressoinnin voitiin myös todeta olevan erittäin tehokasta, sillä tietokanta vei lopulta levytilaa vain 81 megatavua (ks kuvio 47). Dataa varastoidaan tulevaisuudessa IoT-järjestelmissä aina vain suurempia määriä, jolloin muunlaiset tietokannat eivät välttämättä ole järkeviä vaihtoehtoja.

```
root@influxdb:/# du -sh /var/lib/influxdb/data/target/
81M     /var/lib/influxdb/data/target/
```

Kuvio 47. InfluxDB-tietokannan viemä levytila

RabbitMQ osoittautui erittäin nopeaksi viestilähetiksi ja olisi pystynyt toimimaan todennäköisesti nopeamminkin, jos dataa olisi julkaissut ja käsitellyt useampi applikaatio. Producer ja consumer -applikaatiot ovat helposti tulevaisuudessa monistettavissa Docker-konttien ansiosta. Tästä saavutettavaa suorituskykyhyötyä ei testattu työssä. Myös Grafana soveltui erinomaisesti noutamaan suuria määriä dataa, sillä suoritus-

kyky pysyi hyvänä koko projektin ajan. Kartalla ja graafeissa pystyttiin esittämään tuhansien eri mittalaitteiden dataa lähes reaaliaikaisesti. Lisäksi käyttäjälle tarjottiin mahdollisuus suodattaa haluamansa tieto muuttujien avulla. Grafanan mukana tulevat paneelit olivat riittäviä yksinkertaisen datan esittämiseen, mutta erilaisten asiakkaiden vaatimusten mukaisten näkymien tuottamiseen täytyisi todennäköisesti luoda uusia paneeleita liitännäisten avulla.

IoT-projektin toteuttaminen ei myöskään vaatinut kalliiden ohjelmistojen lisensointia, vaan oli täysin mahdollista toteuttaa projekti hyödyntäen ilmaisia avoimen lähdekoodin teknologioita. Työkalut olivat myös selkeitä, helppokäyttöisiä ja integroituvat helposti useisiin eri järjestelmiin. Esimerkiksi RabbitMQ tukee useita eri viestiprotokollia ja Grafana useita eri datalähteitä. Docker-kontit aiheuttivat jonkin verran ongelmia työn edetessä. Oli ajoittain vaikeaa saada kontit keskustelemaan keskenään. Docker Compose auttoi tässä rajaamalla usean toisiinsa liittyvän kontin samaan verkkoon. Lisäksi työn alussa mietittiin hetki, kuinka data varastoidaan kontin sammumisenkin jälkeen. Data volume oli tähän helppo ratkaisu. Projektin kannalta oli kuitenkin äärimmäisen tärkeää, että kontit saatiin toimimaan ja data säilymään, sillä se takaa järjestelmän helpon laajennettavuuden.

Toteutettu työ ei kuitenkaan saavuttanut täysin tuotantovalmista tilaa. Suurin puute tehdyssä toteutuksessa oli tietoturva. Suurin osa järjestelmien käyttäjätunnuksista ja salasanoista jätettiin oletusmuotoihinsa. Tuotantoversioon tulisi ehdottomasti keksiä uudet tunnukset ja salasanat tietoturvan parantamiseksi. Lisäksi Data consumer -applikaation ja RabbitMQ-viestilähetin välisen yhteyden salaamiseksi luodut sertifikaatit olivat itse allekirjoitettuja. Tuotantoversioon tulisi hankkia varmenneviranomaisen allekirjoittamat sertifikaatit. Tietoturvaan panostettiin kuitenkin riittävästi, sillä puutteet otettiin huomioon ja tuotantoversio ei vaatisi kovinkaan suuria modifikaatioita.

Kehitystyössä tuli lähes vahingossa kokeiltua kahta eri Java-kehitysympäristöä Linuxilla. Niistä IntelliJ IDEA osoittautui kokonaisvaltaisesti paremmaksi työkaluksi, mutta yllättäen myös Visual Studio Code soveltuu erinomaisesti kevyeen Java-ohjelmointiin. Kehitystyö eteni hyvää vauhtia koko työn ajan, mutta yksikkötestausta oltaisiin voitu tehdä enemmänkin. Ohjelmiston testaus hoidettiin suurimmalta osin kä-

sin testaamalla, joka ei ole kovin moderni testausmenetelmä. Spring Boot -applikaatioihin tehtiin kuitenkin yksinkertaiset yksikkötestit, jotka testaavat, että applikaatiot käynnistyivät.

Java oli ohjelmointikielenä ennalta tuttu, mutta Spring oli uudempi tuttavuus. Usein kuulee puhuttavan, että Java-kielen suosio olisi laskussa, mutta Spring tarjosi erittäin laajat mahdollisuudet helppoon taustajärjestelmien pystyttämiseen projekteissa. Java tulee varmasti pysymään suosiossa vielä pitkään suuren osaajamääränsä sekä joustavuutensa vuoksi. Spring on yksi tärkeä osa Javan suosion säilymisessä yritysmaailman projekteissa.

Työn versionhallinta onnistui hyvin. Uusia ominaisuuksia lisätessä tehtiin uusi haara, joka yhdistettiin takaisin master-haaraan merge requestin avulla ominaisuuden valmistuttua. Jos työssä olisi ollut useampi tekijä, olisi joku aina tarkistanut merge requestin ennen haaran liittämistä master-haaraan. Kun työn valmistumista lähestyttiin, jäätiin kuitenkin vahingossa yhteen ominaisuushaaraan tekemään muutoksia, jotka eivät liittyneet ominaisuuteen. Olisi ollut parempi tehdä muutoksilla oma haaransa, tai ehkä jopa tehdä pienet muutokset suoraan master-haaraan. Työn kulusta jäi kuitenkin selkeä jälki versionhallintaan, jota oli helppo seurata.

Lähteet

Air Quality Data. N.d. OpenAQ-sivuston esittely kerättävästä datasta. Viitattu 6.12.2018. https://openaq.org/#/locations?_k=gbguei.

Air quality index. N.d. Euroopan ympäristökeskuksen ilmansaastekartta. Viitattu 13.12.2018. https://www.eea.europa.eu/themes/air/air-quality-index.

Bhave, M., Bryant, J., Deleuze, S., Dupuis, C., Long, J., Nicoll, S., Overdijk, M., Pavić, V., Simons, M., Syer, D., Webb, P. & Wilkinson, A. N.d. Spring Boot Reference Guide. Spring Boot virallinen opas. Viitattu 25.11.2018. https://docs.spring.io/spring-boot/docs/2.1.0.RELEASE/reference/htmlsingle/.

Brisbourne, A. 2014. Tesla's Over-the-Air Fix: Best Example Yet of the Internet of Things?. Viitattu 19.11.2018. https://www.wired.com/insights/2014/02/teslas-air-fix-best-example-yet-internet-things/.

China: Air quality standards. N.d. Kiinan päästöjen raja-arvot ilmoittava sivusto. Viitattu 13.12.2018. https://www.transportpolicy.net/standard/china-air-quality-standards/.

Churilo, C. 2018. InfluxDB is 27x Faster vs. MongoDB for Time Series Workloads. Viitattu 28.11.2018. https://www.influxdata.com/blog/influxdb-is-27x-faster-vs-mongodb-for-time-series-workloads/.

Cope, S. 2018. Understanding the MQTT Protocol Packet Structure. Viitattu 4.12.2018. http://www.steves-internet-guide.com/mqtt-protocol-messages-overview/.

DB-Engines Ranking of Time Series DBMS. N.d. Vertailu aikasarjatietokantojen suosioista. https://db-engines.com/en/ranking/time+series+dbms.

Deering, S. 2012. Do you know what a REST API is?. Viitattu 2.1.2019. https://www.sitepoint.com/developers-rest-api/.

Developer Guide. N.d. Grafana liitännäisten kehitysopas. Viitattu 29.11.2018. http://docs.grafana.org/plugins/developing/development/.

Digia yrityksenä. N.d. Digian esittely omilla sivuillaan. Viitattu 15.11.2018.

Docker concepts. N.d. Perehdytys Dockerin toimintaan Docker-dokumentaatiossa. Viitattu 29.11.2018. https://docs.docker.com/get-started/#docker-concepts.

Dockerfile reference. N.d. Dockerfile virallinen dokumentaatio. Viitattu 6.12.2018. https://docs.docker.com/engine/reference/builder/.

Docker in Production: Lessons from the Trenches. 2015. Bleeding Edge Press.

Digitaalinen hintalappu & hintanäyttö. N.d. Zeta Displayn digitaalisen hintalapun esite. Viitattu 19.11.2018. http://www.zetadisplayfinland.fi/fi/esl/.

Dossot, D. 2014. RabbitMQ Essentials. Birmingham: Packt Publishing.

Eclipse Paho Java Client. N.d. Paho Java Client kirjaston kotisivusto. Viitattu 4.12.2018. https://www.eclipse.org/paho/clients/java/.

esLabels: How it works. N.d. elektronisen hintalapun kuvaus esLabels yrityksen sivutolla. Viitattu 19.11.2018 http://www.eslabels.com.au/how-it-works/.

Fehrenbacher, K. 2015. How Tesla is ushering in the age of the learning car. Viitattu 19.11.2018. http://fortune.com/2015/10/16/how-tesla-autopilot-learns/.

Heikkilä, T. 2014. Tilastollinen tutkimus. Helsinki: Edita.

InfluxDB key concepts. N.d. Dokumentaatio InfluxDB:n toimintaperiaatteista. https://docs.influxdata.com/influxdb/v1.7/concepts/key_concepts/.

Johansson, L. 2015. RabbitMQ Exchanges, routing keys and bindings. Viitattu 25.11.2018. https://www.cloudamqp.com/blog/2015-09-03-part4-rabbitmq-for-beginners-exchanges-routing-keys-bindings.html.

Johansson, L. 2018. Part 2: RabbitMQ Best Practice for High Performance (High Throughput). Viitattu 11.12.2018. https://www.cloudamqp.com/blog/2018-01-08-part2-rabbitmq-best-practice-for-high-performance.html.

Kili, A. 2018. Grafana – An Open Source Software for Analytics and Monitoring. Viitattu 29.11.2018. https://www.tecmint.com/install-grafana-analytics-in-centos-ubuntu-debian/.

Manage data in Docker. N.d. Docker dokumentaatio datan varastoinnista. Viitattu 6.12.2018. https://docs.docker.com/v17.09/engine/admin/volumes/.

Marr, B. 2018. The Internet Of Things (IOT) Will Be Massive In 2018: Here Are The 4 Predictions From IBM. Viitattu 2.1.2019. https://www.forbes.com/sites/bernardmarr/2018/01/04/the-internet-of-things-iot-will-be-massive-in-2018-here-are-the-4-predictions-from-ibm/#71cc273cedd3.

MQTT Essentials Part 5: MQTT Topics & Best Practices. N.d. Viitattu 4.12.2018. https://www.hivemq.com/blog/mqtt-essentials-part-5-mqtt-topics-best-practices.

MVC Framework - Introduction. MVC-arkkitehtuurin esittely. N.d. Viitattu 2.1.2019. https://www.tutorialspoint.com/mvc_framework/mvc_framework_introduction.htm.

Open AQ Platform API. N.d. OpenAQ rajapinnan dokumentaatio. Viitattu 6.12.2018. https://docs.openaq.org.

Overview of Docker Compose. N.d. Docker Compose virallinen dokumentaatio. Viitattu 6.12.2018. https://docs.docker.com/compose/overview/.

Permissions Overview. N.d. Käyttölupien ohje Grafanan dokumentaatiossa. Viitattu 29.11.2018. http://docs.grafana.org/permissions/overview/.

Sangaiah, A. K., Thangavelu, A. & Sundaram, V. M. 2018. Cognitive computing for big data systems over IoT: Frameworks, tools and applications. Cham: Springer.

Schneider, S. 2013. Understanding The Protocols Behind The Internet Of Things. Viitattu 2.1.2019. https://www.electronicdesign.com/iot/understanding-protocols-behind-internet-things.

Spring Framework modules. Dokumentaatio Spring Framework moduuleista. N.d. Viitattu 25.11.2018. https://docs.spring.io/spring/docs/3.0.0.M4/reference/html/ch01s02.html.

Spring Framework Overview. Spring Framework virallinen dokumentaatio. N.d. Viitattu 25.11.2018. https://docs.spring.io/spring/docs/5.1.2.RELEASE/spring-framework-reference/overview.html#overview.

Takacs, M. 2018. Dockerfile tutorial by example - basics and best practices. Viitattu 6.12.2018. https://takacsmark.com/dockerfile-tutorial-by-example-dockerfile-best-practices-2018/.

Time Series Database (TSDB) Explained. N.d. Viitattu 28.11.2018. https://www.influxdata.com/time-series-database/.

TLS Support. N.d. RabbitMQ TLS-tuen dokumentaatio. Viitattu 25.11.2018. www.rabbitmq.com/ssl.htm.

User Authentication Overview. N.d. Tunnistautumisen ohje Grafanan dokumentaatiossa. Viitattu 29.11.2018. http://docs.grafana.org/auth/overview/.

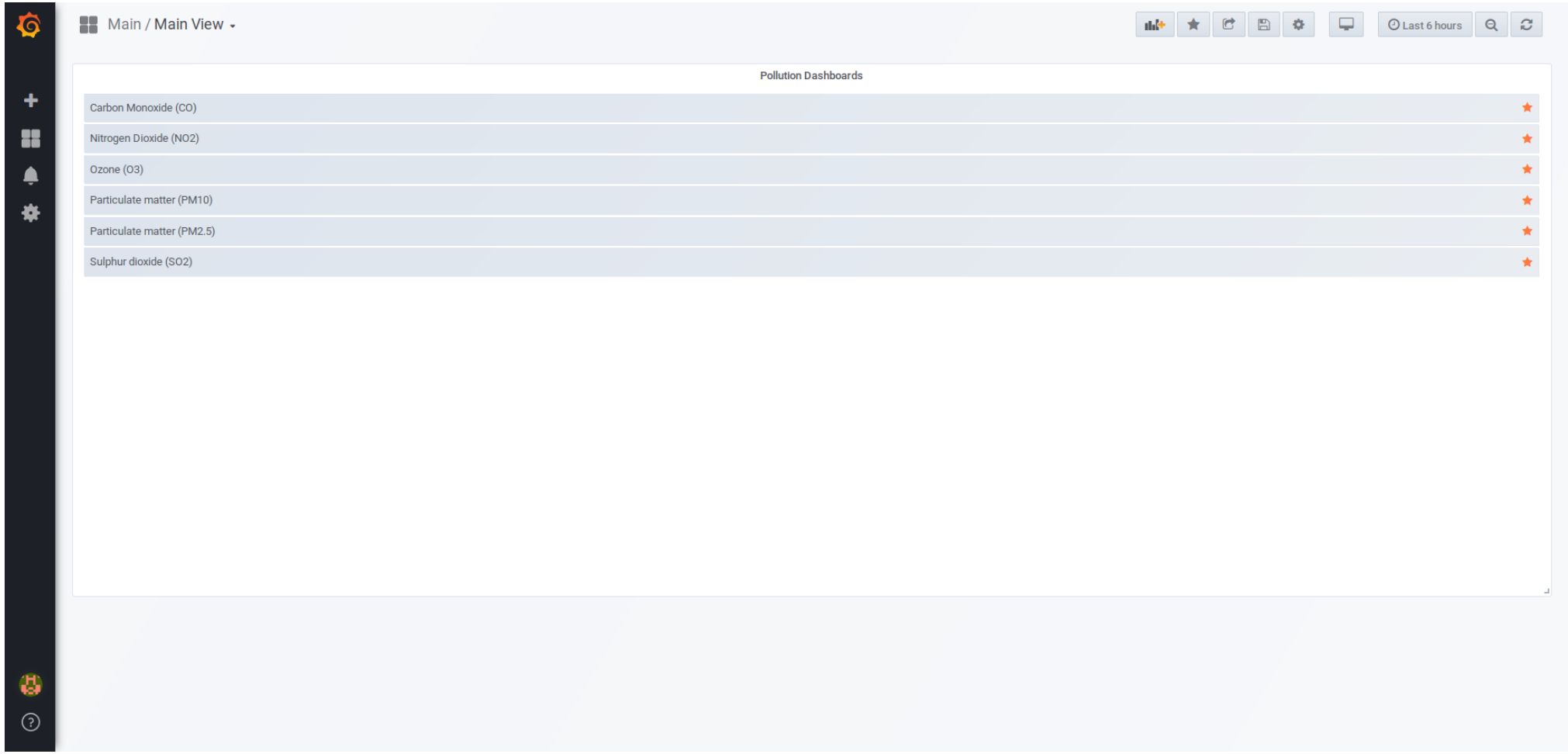
What is MQTT and When You Should Use It. 2018. Viitattu 4.12.2018. https://www.pubnub.com/blog/what-is-mqtt-use-cases/.

What is Transport Layer Security (TLS). 2018. Viitattu 2.1.2019. https://www.networkworld.com/article/2303073/lan-wan/lan-wan-what-is-transport-layer-security-protocol.html.

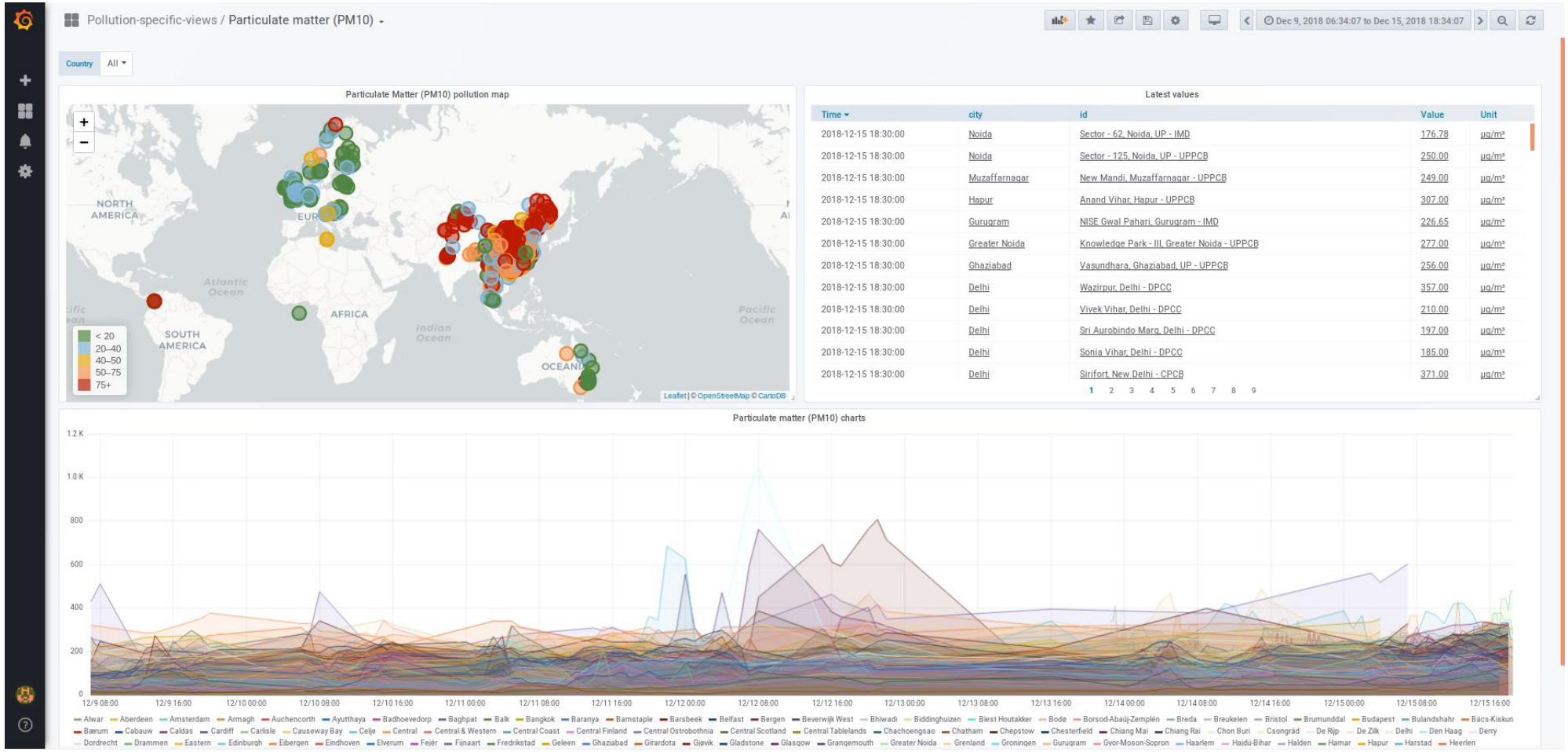
Where does OpenAQ data come from?. N.d. Viitattu 6.12.2018. https://medium.com/@openaq/where-does-openaq-data-come-from-a5cf9f3a5c85.

Liitteet

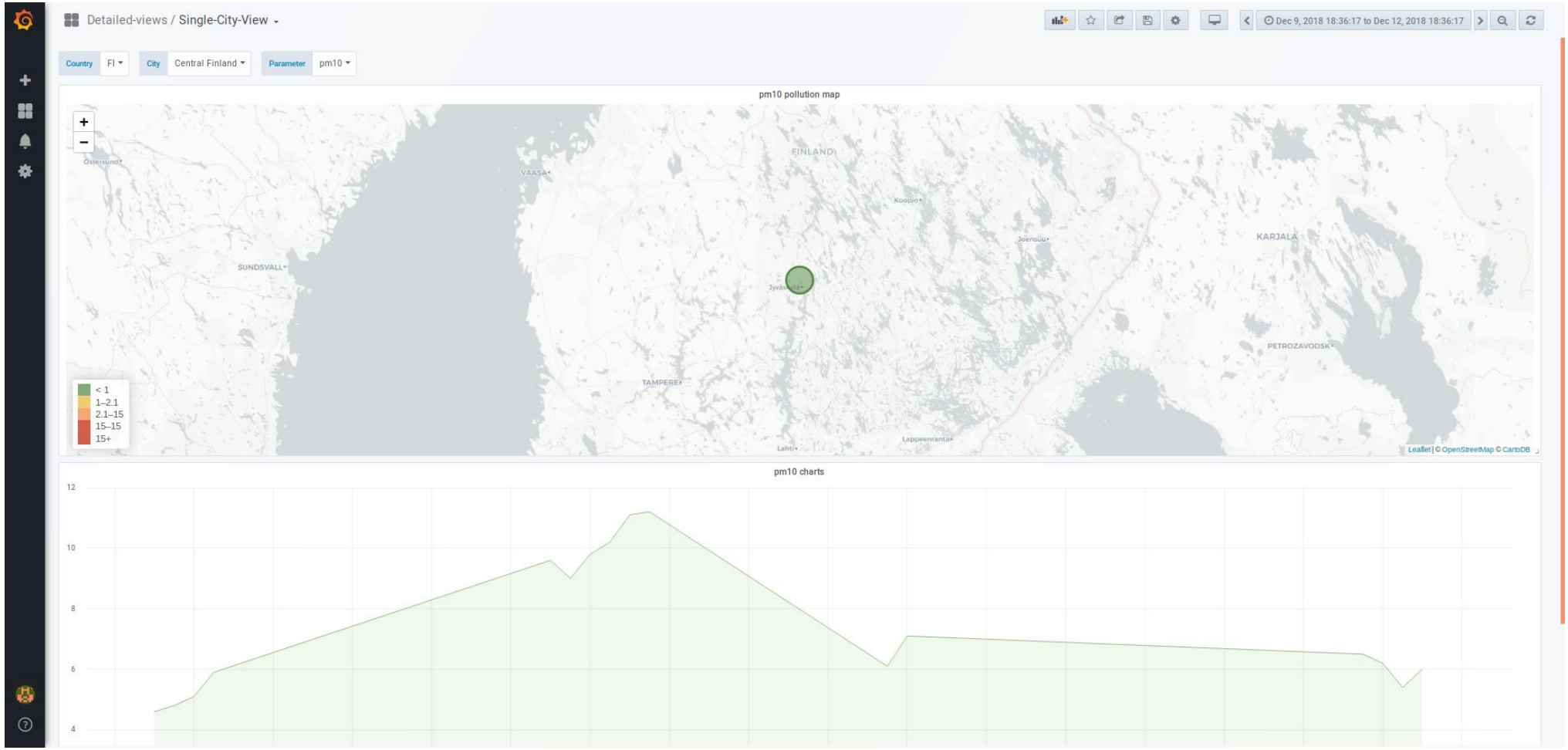
Liite 1. Grafanan päähallintapaneeli



Liite 2. Ilmansaastetietojen hallintapaneeli



Liite 3. Kaupunkikohtainen ilmansaastetietojen hallintapaneeli



Liite 4. Versionhallinnan haarojen visualisointi