LAADUNVALVONTASOVELLUKSEN TAUSTAJÄRJESTELMÄN SUUNNITTELU JA TOTEUTUS

LAB-AMMATTIKORKEAKOULU
Insinööri (AMK)
Tieto- ja viestintätekniikka
Syksy 2020
Johannes Syvämeri

Tiivistelmä

Tekijä(t)	Julkaisun laji	Valmistumisaika
Syvämeri Johannes	Opinnäytetyö, AMK	Syksy 2020
	Sivumäärä	
	37	

Työn nimi

Laadunvalvontasovelluksen taustajärjestelmän suunnittelu ja toteutus

Tutkinto

Insinööri (AMK)

Tiivistelmä

Opinnäytetyön tavoitteena oli suunnitella ja toteuttaa laadunvalvontasovelluksen taustajärjestelmä. Toimeksiantajana toimi Raute Oyj, joka on maailmanlaajuisesti puutuotetoimialaa palveleva yritys.

Puutuotetehtailla tehdään erilaisia laadunvalvontamittauksia. Niillä varmistetaan, että tuotteet vastaavat laadultaan niille määrättyjä kriteereitä ja vaatimuksia. Raute halusi tuottaa laadunvalvontasovelluksen, johon laadunvalvojat voisivat syöttää mittaustietoja, sekä lähettää ne Rauten kehittämään tiedonhallinta- ja raportointijärjestelmään MillSIGHTS:iin.

Työssä käytettiin Express.js:a, joka on Node.js sovelluskehys verkkosovelluksien, taustajärjestelmien ja rajapintojen kehitykseen. Vastaanotetut mittaukset tallennettiin MySQL-tietokantaan. Mittausten lähettämiseen MillSIGHTS:iin, sovellukseen tehtiin mittaustenlähetys moduuli, joka ajoittain tarkistaa tietokannasta lähettämättömät mittaukset ja hoitaa niiden lähettämisen.

Työn tuloksena saatiin toimiva taustajärjestelmä, joka on kevyt ja helposti skaalattavissa.

Asiasanat

Taustajärjestelmä, REST, API

Abstract

Author(s)	Type of publication	Published
Syvämeri, Johannes	Bachelor's thesis	Autumn 2020
	Number of pages	
	37	

Title of publication

Design and Implementation of Backend-system for Quality Control Application

Name of Degree

Engineer (UAS)

Abstract

The goal of the thesis was to plan and implement a backend-system part of the Quality Control mobile application. The client was Raute Oyj. Raute is a company who serves the wood products industry worldwide.

In the wood product producing factories, they do quality control measurements. They want to make sure, that the product meets the standards and is of good quality. Raute wanted to develop an application, where a quality controller could input measurements. From the application, measurements would be sent to Raute's data control- and reporting system MillSIGHTS.

Express.js is a framework for Node.js. With it, it is easy to create web applications, backend systems and APIs. Its functionality is based on using middleware-functions. All the measurements that, the backend system receives, are stored in a MySQL-database. To send received measurements to MillSIGHTS, a module was built. It handles the sending events and makes sure that all measurements are sent to correct MillSIGHTS instance.

The light and easily scalable backend system became a product of this thesis.

Keywords

Backend-system, REST, API

SISÄLLYS

1 JOHDANTO

Verkko- ja mobiilisovellukset tuotteena yleistyvät ja kehittyvät kovaa vauhtia. Nykypäivänä useimmilla moderneilla yrityksillä on käytössään jonkunlainen sovellus tai verkkosivusto. Sovelluksella asiakas voi käyttää yrityksen tarjoamia palveluja helposti omalla päätelaitteellaan. Modernina yrityksenä Raute tuottaa asiakkaillensa digitaalisia palveluja.

Raute on puutuotetoimialan yritys, jonka päätoimipaikka sijaitsee Nastolassa. Yrityksen liikevaihto oli vuonna 2019 151,3 miljoonaa euroa (Raute b). Rauten asiakkaita ovat viilun, vanerin, viilupalkin ja sahatavaran tuottajat. Rauten tarjoaa asiakkaillensa koko tuotantoprosessin koneet ja laitteet. Asiakkaita palvelee noin 800 Rauten työntekijää. Raute panostaa jatkuvasti tuotekehitykseensä, jonka tavoitteena on parantaa asiakkaan toiminnan kannattavuutta ja kilpailukykyä. Tuotekehitys on koneiden ja palveluiden kehitystä tai uusien tuotteiden ja tuotantokonseptien luontia. (Raute a.) Tehtailla suoritetaan laadunmittaamista, jonka avuksi Raute halusi kehittää mobiilisovelluksen.

Opinnäytetyön tavoitteena on suunnitella ja toteuttaa laadunvalvontasovelluksen taustajärjestelmä. Taustajärjestelmä toimii rajapintana ja kommunikoijana mobiilikäyttöliittymän ja Rauten tiedonhallinta -ja raportointijärjestelmän MillSIGHTS:in välillä. Sovelluksella pyritään helpottamaan laadunvalvojien työtä digitalisoimalla laadunvalvontamittausten keruu.

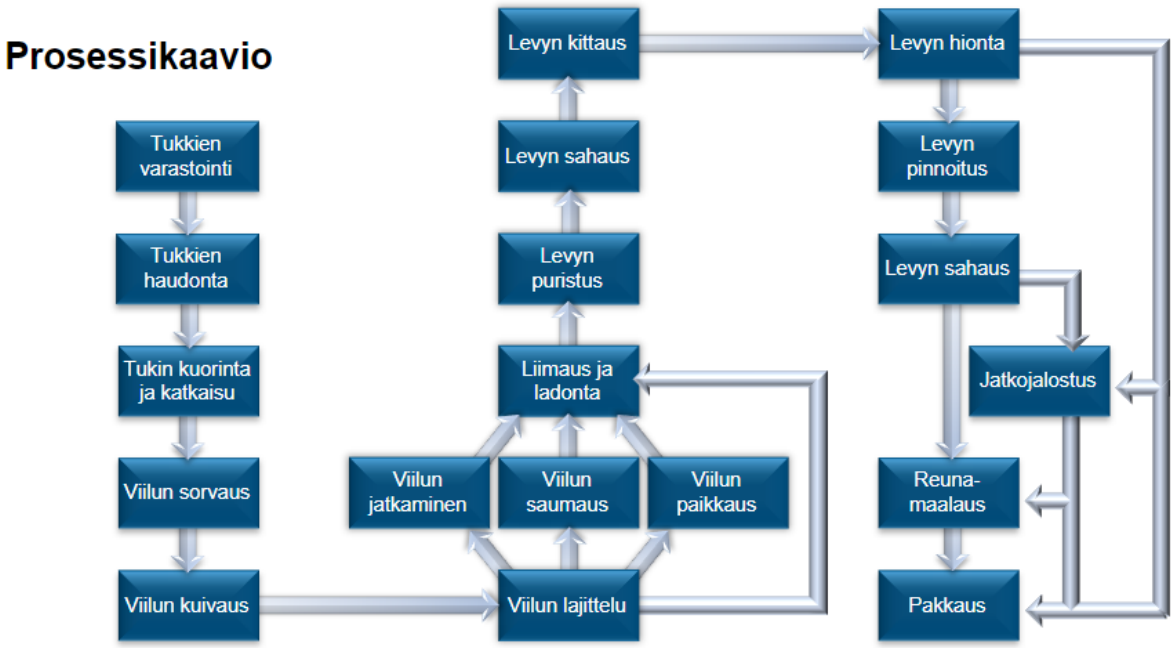
Suunnitteluvaiheessa Raute antoi vaatimuslistan, joka määritteli käytettävän arkkitehtuurimallin. Aikaisemman JavaScript kokemuksen myötä, luonnollisin valinta teknologioille oli Node.js ja Express.js.

2 TOIMINTAYMPÄRISTÖ

Raute on vanerin ja viilupuun tehdaskokonaisuuksien toimittaja. Luvussa 2.1 tarkastellaan Rauten ja heidän asiakkaidensa toimintaympäristöä. Luvussa 2.2 tutkitaan mitä tarkoitetaan tehtailla tapahtuvalla laadunvalvonnalla. Viimeisessä alaluvussa tarkastellaan taustajärjestelmälle asetettuja asiakasvaatimuksia.

2.1 Tehdas ja linjastot

Vanerin ja viilupuun tuotanto koostu useista eri tuotantovaiheista (Kuva 1). Näissä vaiheissa tukeista valmistetaan viiluarkkeja, joista sitten tehdään vaneria.



Kuva 1. Tuotantoprosessikaavio (Raute Academy)

Tukkivarastosta otettuja tukkeja haudotaan kuumassa vesialtaassa, noin 24 tunnin ajan. Haudonnan tarkoitus on lämmittää puuaines sopivaksi sorvausta varten. Toinen menetelmä haudontaan on hautoa tukkeja haudontakammiossa, jossa lämpötila on noin 90°C. Haudonnan jälkeen tukit kuoritaan ja katkaistaan oikean pituisiksi sorvausta varten. Sorvilinjalla puupöllit sorvataan viilumatoksi, josta sitten leikataan viiluarkkeja. Sorvista viilut päätyvät kuivattavaksi. Kuivauksen tarkoitus on saada viilut tavoitekosteuteen liimaamista ja kuumapuristusta varten. Kuivaimesta viilut lajitellaan lokeroihin riippuen niiden laadusta. Laatu määritellään viilun ulkonäön ja kosteuden mukaan. Kokonaiset viiluarkit pinkattaan ja kuljetetaan suoraan ladontaan, paikkauslinjalle tai jatkamislinjalle. Vialliset viilut menevät korjattavaksi saumauslinjalle. Jatkamislinjalla viiluarkkeja liimataan yhteen

pituussuunnassa ja saumauslinjastossa ne liimataan yhteen leveyssuunnassa. Paikkauslinjassa viiluarkissa oleva virhe voidaan korvata ehjällä puupaikalla. (Raute Academy.)

Liimaus ja ladontalinjastolla liimoitetut viiluarkit ladotaan vanerilevyksi ladonta-asemalla. Liimauksen ja ladontalinjaston jälkeen vaneriarkit siirtyvät puristuslinjastolle. Levyn sahaus-, kittaus- ja hiontalinjoilla vaneri sahataan, sen pintavirheitä kitataan ja se hiotaan sileäksi ja oikeaan paksuuteen. Hionnasta vaneri voi mennä pinnoituslinjastolle, jatkojalostukseen tai pakkauslinjastolle. Pinnoituslinjastolla erilaisia pinnoitteita liimataan vanerilevyn päälle. Sen avulla parannetaan vanerin pintaominaisuuksia ja laajennetaan sen käyttömahdollisuuksia. Pinnoituksen jälkeen arkkien ylimitta sahataan oikeaan levymittaan. (Raute Academy.)

Reunamaalauksella vähennetään kosteuden tunkeutumista levyyn. Jatkojalostuksessa vaneria jalostetaan asiakkaan toiveiden mukaan. Viimeisenä valmiit vanerit pakataan kuljetusta varten. (Raute Academy.)

Koko vanerintuotanto koostuu siis useasta eri linjastosta. Yksi linjasto koostuu useasta eri vaiheesta ja laitteesta. Kuvassa 2 on esiteltynä sorvilinjasto. Ensimmäisenä puunrunko sorvataan viiluksi, jonka jälkeen se menee skannerin läpi. Skanneri havaitsee, jos viilun pinnassa on jotain vikaa. Lopuksi viilu leikataan sopivan pituiseksi ja ladotaan pinoon. Pinosta Viilut siirrettäisiin seuraavaan linjastoon, eli kuivaimeen.

Kuva 2 Sorvilinjasto (Raute Academy)

Tehtaiden toiminnan parantamiseksi tehtaat tarvitsevat reaaliaikaista ja tarkkaa tietoa tuotannon suorituskyvystä ja linjojen saatavuudesta. Tämän ongelman ratkaisuksi Raute on

kehittänyt tiedonhallinta- ja raportointijärjestelmän, nimeltä MillSIGHTS. MillSIGHTS on käytössä noin 100 tuotantolinjalla ympäri maailman.

2.2 Laadunvalvonta

Laadunvalvonnalla varmistetaan, että tuote ja prosessi täyttää niille asetetut vaatimukset. Laadunvalvonta koostuu toiminnoista ja toimintatavoista. Toimintoja ja tapoja ovat esimerkiksi erilaiset mittaukset, kuten viiluarkin pituus ja paksuusmittaukset. Kaikille laatumittauksille on asetettu tavoitteet ja raja-arvot. Jos tavoitteita ei täytetä tai raja-arvoihin ei päästä, niin silloin on ryhdyttävä toimenpiteisiin ongelman korjaamiseksi. Tietyn väliajoin tehtävät laadunvalvontamittaukset varmistavat, että linjastot toimivat oikein ja tuotteet ovat laadukkaita. Laadunvalvontaa voidaan tehdä tuotannossa tai laboratoriossa.

Tällä hetkellä laadunvalvojat täyttävät paperilomakkeita, jonka tiedot merkataan myöhemmin tietokoneelle Excel-tiedostoon. Tämä paperilomakkeiden täyttö ja mittaustulosten siirtäminen tietokoneelle on ongelma, jonka Raute haluaa ratkaista.

Ongelman ratkaisuksi laadunvalvojille päätettiin tehdä sovellus, jonka mobiilikäyttöliittymään mittaustulokset voidaan tallettaa. Mobiililaitteesta tiedot voitaisiin siirtää automaattisesti MillSIGHTS: iin. Mobiilikäyttöliittymän ja MillSIGHTS: in välisen tiedonsiirron toteuttaisi siihen rakennettu taustajärjestelmä, jonka suunnittelu ja toteutus on tämän opinnäytetyön tavoitteena.

2.3 Asiakasvaatimukset

Taustajärjestelmälle on asetettu vaatimuksia, jotka ovat:

- REST-arkkitehtuuri. Taustajärjestelmän täytyy noudattaa REST-arkkitehtuurityyliä, jolloin toteutuksesta tulee kevyt ja se on helposti skaalattavissa.
- Tietokanta. Taustajärjestelmän pitäisi käyttää tietokantaa mittausten ja käyttäjien ja yritysten tietojen tallentamiseen.
- Reititys. Taustajärjestelmän pitäisi tarjota mobiilikäyttöliittymälle tapa, jolla sen kanssa voidaan kommunikoida.
- Tietoturvallisuus. Taustajärjestelmän pitäisi olla tietoturvallinen. Sen täytyy pystyä vastaanottamaan HTTPS pyyntöjä, sekä suojaamaan dataa ja reittejä.
- Erilaisten mittausten käsittely. Taustajärjestelmän pitäisi pystyä käsittelemään erilaisia mittausdatoja. Mobiilikäyttöliittymästä voi tulla esimerkiksi kuvia base64 muodossa, jotka täytyy pystyä tallentamaan.

- MillSIGHTS yhteensopivuus. Taustajärjestelmän pitäisi pystyä lähettämään mittaustiedot MillSIGHTS: iin. Lähetys osoite riippuu sovelluksen käyttäjästä. Taustajärjestelmän pitää siksi toimia yhdessä MillSIGHTS: in kanssa, mutta se ei saa olla riippuvainen siitä.
- Luotettavuus. Taustajärjestelmän ei pitäisi olla koskaan pois käytöstä. Sen pitää pystyä toipua virheistä ja tarvittaessa käynnistyä uudestaan automaatiisesti.
- Validointi. Taustajärjestelmän pitäisi pystyä validoimaan sinne saapuvat pyynnöt. On oltava tapa, jolla esimerkiksi estetään sovellukseen pääsy ulkopuolisilta.
- Lokitiedot. Taustajärjestelmän täytyy pitää kirjaa sinne saapuvista pyynnöistä. Lokitiedostoista voidaan myös tarkistaa järjestelmässä sattuneita virheitä.
- Sähköposti. Taustajärjestelmän pitäisi pystyä lähettämään sähköpostiviesti, jos MillSIGHTS:iin tulee yhteysongelmia.

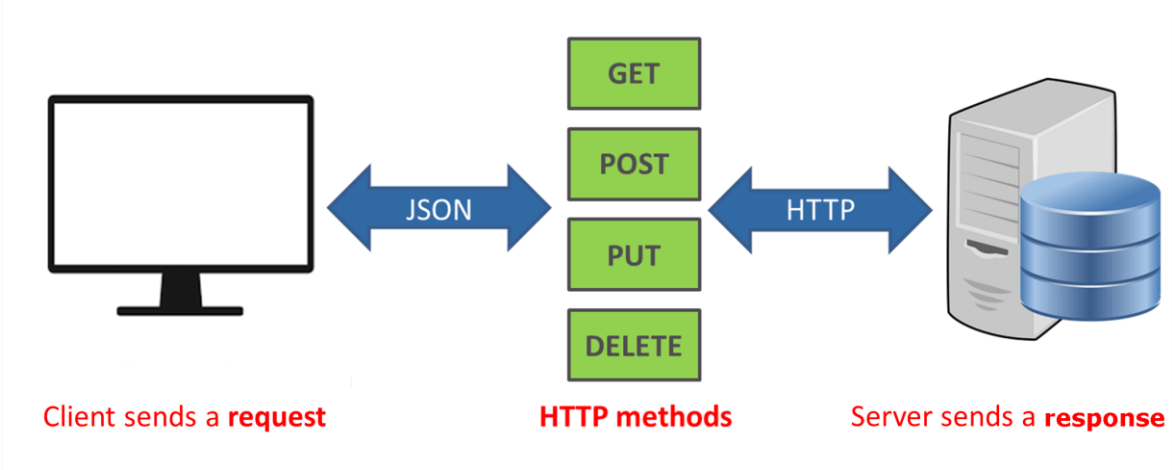
Tärkein kehityksen suuntaan vaikuttava asiakasvaatimus on REST-arkkitehtuuri. Se määrää koko ohjelman rakenteen sekä sen, miten sitä kehitettään. Sen avulla myös valittiin käytettävät teknologiat, joilla se ja muut asiakasvaatimukset täytettiin.

3 REST-ARKKITEHTUURI

Taustajärjestelmän ensimmäisenä asiakasvaatimuksena oli, että se noudattaisi REST-arkkitehtuuria. Tässä luvussa käydään ensin läpi REST-arkkitehtuuria yleisesti, jonka jälkeen tutkitaan, mitä tarkoitetaan resurssilla ja resurssimetodeilla. Lopuksi tarkastellaan REST:in asettamia rajoitteita. Tekstin lähteenä käytettiin Roy Fieldingin kuuluisaa “Architectural Styles and the Design of Network-based Software Architectures” väitöskirjaa, joka julkaistiin vuonna 2000. Fieldingia pidetään REST-tyylin keksijänä.

REST (REpresentational State Transfer) on http-protokollaan perustuva arkkitehtuurimalli ohjelmointirajapintojen (Application Programming Interface) toteuttamiseen. REST ei määrittele miten rajapinta täytyy toteuttaa käytännössä, mutta se antaa sen suunnitteluun ohjeita.

Kuvassa 3 on havainnollistava esimerkki asiakkaan ja palvelimen välisestä tiedonsiirrosta. Asiakas lähettää pyynnön, joka voi pitää sisällään JSON (JavaScript Object Notation) merkkijonon, jossa on tarvittavat tiedot, millä palvelin voi pyynnön käsitellä. Asiakaskone myös määrittää kutsussa HTTP (Hyper Text Transfer Protocol) -metodin. Palvelin käsittelee pyynnön ja palauttaa vastauksen niin ikään JSON merkkijonona.



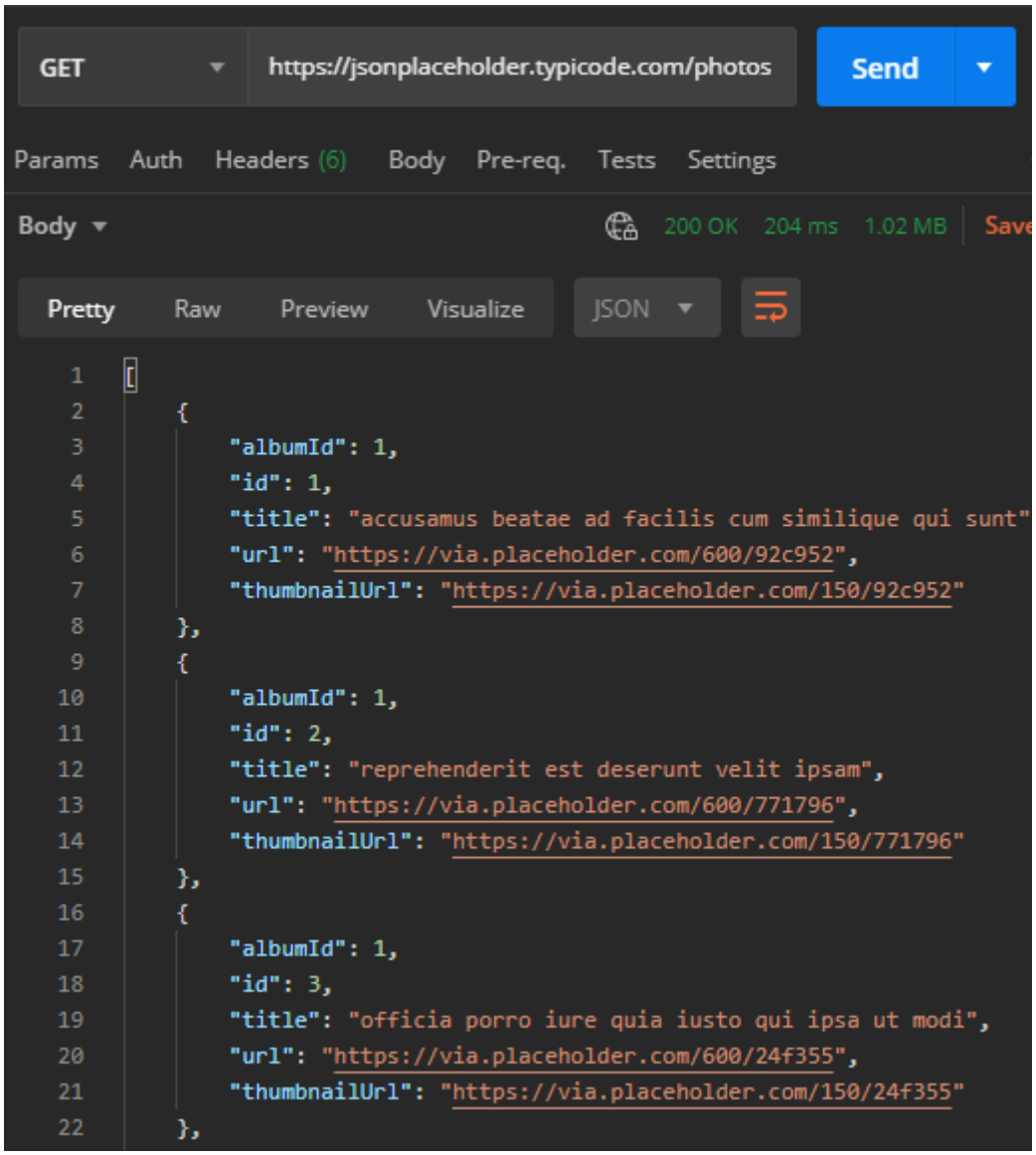
Kuva 3 Tiedonkulku palvelimen ja asiakaskoneen välillä (Benharosh 2018)

Tiedonsiirron data formaatti voi olla esimerkiksi XML (Extensible Markup Language) tai JSON. JSON on helppolukuista ja kevyttä. Kuvassa 4 on esimerkki JSON tietorakenteesta.

Kuva 4. JSON tietorakenne

3.1 Resurssi

REST:ssa informaatiota kutsutaan resurssiksi, jos sille voidaan antaa jokin nimi. Resurssi voi olla esimerkiksi dokumentti, kuva, väliaikainen palvelu tai resurssikokoelma. (Fielding 2000, 88.) Esimerkiksi ”photos” olisi kuvakokoelma ja ”photo” yksittäinen kuva. Kuvassa 5 on tehty GET-pyyntö Postman työkalulla. Postman on työkalu, jolla voidaan lähettää erilaisia pyyntöjä palveluun. Photos resurssikokoelma tunnistetaan /photos URI:lla (Uniform Resource Identifier). Yksittäinen kuva tunnistettaisiin /photos/{kuvanId} URI:lla.



Kuva 5. GET-pyyntö Postman työkalulla

3.2 Resurssimetodit

Roy Fielding ei koskaan määritellyt, mitä tapaa resurssin manipulointiin on käytettävä. Hän vain määritteli, että tapojen täytyy olla yhtenäinen. Verkkomaailmassa resurssien manipulointiin liitetään yleensä käsite CRUD (Create, Read, Update, Delete). CRUD-operaatioille on olemassa vastaavat HTTP-metodit. Resurssin luontiin käytetään POST-metodia. Resurssien hakemiseen GET-metodia. Resurssin muokkaamiseen PUT tai PATCH-metodia. Resurssin poistamiseen DELETE-metodia. CRUD ja HTTP vastaavat metodit ovat esiteltynä kuvassa 6.

HTTP Method	CRUD Operation	Purpose
GET	Read	Retrieve the requested records
POST	Create	Create a new record
PUT	Update or Replace	Set a value for a record
DELETE	Delete	Remove a record
PATCH	Partial Update	Modify a resource

Kuva 6. CRUD ja HTTP-metodit (Naeem 2020)

3.3 Rajoitteet

REST-arkkitehtuuriin kuuluu kuusi Roy Fieldingin määrittämää rajoitetta, jotka ovat:

- Asiakas-palvelin
- tilattomuus
- välimuistin käyttäminen
- asiakaspalvelin-malli
- kerroksittainen järjestelmä
- ladattava koodi

Ainoa vapaaehtoinen rajoite on koodin lataus. Jos sovellus ei täytä mitä tahansa muuta rajoitetta, sitä ei periaatteessa pitäisi kutsua REST-rajapinnaksi.

3.3.1 Asiakas-palvelin

Asiakas-palvelin rajoitteen pääperiaatteena on asiakkaan ja palvelimen erottaminen. Tällä tavoin muutokset käyttöliittymässä eivät vaikuta palvelimen toimintaan ja päinvastoin. Merkittävin hyöty onkin se, että sovelluksen eri komponenteille annetaan tilaa kehittyä itsenäisesti. Sillä myös parannetaan käyttöliittymän siirrettävyyttä eri alustojen välillä. Samalla skaalautuvuus paranee, koska palvelimen komponentit yksinkertaistuvat. (Fielding 2000, 78.)

3.3.2 Tilaton toiminta

Tilaton rajoitteella Fielding määrittelee, että kommunikaation täytyy olla luonteeltaan tilaton. Jokainen pyyntö, joka saapuu palvelimelle, täytyy käsitellä yksilönä eikä minkäänlaista istuntotilaa pidetä yllä palvelimen toimesta. Tämä rajoite siirtääkin vastuuta

enemmän asiakkaalle. Asiakkaan viestin on pidettävä sisällään kaikki tarvittavat tiedot, jotta pyyntö voidaan täyttää.

Tämä rajoite parantaa näkyvyyttä, koska palvelimen ei tarvitse ottaa vastaan kuin yksi pyyntö, jonka avulla se saa kaiken tarvittavan informaation pyynnön täyttämiseksi. Rajoite myös lisää ohjelmiston skaalautuvuutta, koska palvelimen ei tarvitse hallita resursseja useiden erilaisten pyyntöjen välillä. Rajoitteen hyötynä voidaan pitää myös luotettavuuden kasvua, koska palvelimen on helpompi toipua virheistä. (Fielding 2000, 78-79.)

3.3.3 Välimuistin käyttö

Välimuistin käyttö vähentää tietoverkon kuormaa. Rajoite vaatii, että palvelimen vastatessa pyyntöön vastaus sisältää tiedon siitä, voiko dataa säilyttää välimuistissa. Tällainen välimuistissa säilytettävä data voisi esimerkiksi olla joku lista, joka ei muutu. Välimuistia hyödyntämällä voitaisiin kokonaan eliminoida asiakkaan ja palvelimen välinen tiedonsiirto. Siten vähennettäisiin samojen pyyntöjen käsittelyä palvelimella ja näin myös palvelimen kuormitus vähenisi. Välimuistin käytöllä on myös huonot puolensa. Asiakas ei voi olla varma, onko sen käyttämä data jo vanhentunut. (Fielding 2000, 80.)

Yhtenä hyvänä esimerkkinä voidaan pitää selainten välimuistin käyttöä. Verkkosivun ollessa staattinen ei ole mitään järkeä hakea sivua aina palvelimelta uudestaan. Kun sivusto on tallennettu välimuistiin, sivuston näyttäminen käyttäjälle on nopeampaa. Näin käyttökokemuksesta tulee miellyttävämpi.

3.3.4 Yhtenäinen käyttöliittymä

Yhtenäinen käyttöliittymä on keskeinen ominaisuus, jolla REST arkkitehtuurimalli eroaa muista verkkosovellusarkkitehtuureista. Sen ansiosta koko järjestelmän arkkitehtuuri yksinkertaistuu sekä interaktioiden näkyvyys paranee. Näkyvyyden paranemisen vuoksi se kannustaa sovelluksen komponenttien kehitystä itsenäisesti. REST: in käyttöliittymä suunnitellaan toimimaan suurissa hypermedia tiedonsiirroissa. Koska tietoa siirretään standardoidussa muodossa, palvelimen kokonaistehokkuus laskee. (Fielding 2000, 81-82.)

Yhtenäinen käyttöliittymä saavutetaan täyttämällä neljä rajoitetta: resurssin tunnistamisella, resurssien manipulointi esitysmuotojen (representations) avulla, itse kuvaavat viestit ja hypermedia sovellustilan moottorina (HATEOAS). (Fielding 2000, 81-82.)

Resurssin pyytäjä saa vastauksena resurssin esitysmuodon. Muoto voi olla esimerkiksi JSON- tai XML-formaatissa. Tällä esitysmuodolla käyttäjän on mahdollista manipuloida palvelimella olevia resursseja. Itse kuvaavilla viesteillä tarkoitetaan sitä, että jokainen

viesti pitää sisällään tarpeeksi informaatioita, kuinka viesti kuuluu prosessoida. HATEOAS rajoitteella asiakkaat saavat resurssin tilan bodyn , kyselymerkkijonoparametrien, otsikkojen ja resurssitunnisteen avulla. Palvelin välittää resurssin tilaa bodyn, statuskoodien ja otsikkojen avulla. Tarvittaessa vastauksessa voi olla myös linkkejä siihen liittyviin resursseihin. (REST Api Tutorial.)

3.3.5 Kerroksittainen järjestelmä

Kerrostetussa järjestelmässä ohjelmiston eri komponentit rakennetaan hierarkkisesti kerroksiin. Komponentti voivat kommunikoida vain välittömästi ylemmällä- tai alemmalla tasolla olevien komponenttien kanssa. Tällä tyylillä asetetaan ohjelmiston yleinen arkkitehtuuri ja suositaan riippumattomuutta ohjelmiston osien välillä. Kerroksittaisella tyylillä voidaan myös kapseloida haluttuja palveluita ja suojata uusia. Harvoin käytetyt toiminnot voidaan siirtää jaettuun välittäjään. Palvelukokonaisuuden kuormitusta voidaan myös tasapainottaa tietoverkossa ja prosessoreissa. (Fielding 2000, 83)

3.3.6 Koodin lataus

Viimeinen rajoite mahdollistaa asiakasohjelmiston toiminnallisuuden laajentamisen palvelimelta saatavilla skripteillä. Palautettava skripti voi esimerkiksi olla joku käyttöliittymä komponentti. Koska tämä rajoite vähentää ohjelmiston näkyvyyttä, se on määritelty vapaaehtoiseksi. (Fielding 2000, 84)

Rajoitteella voitaisiin tehdä esimerkiksi asiakasohjelmiston käyttöliittymään komponentteja, jotka olisivat uniikkeja kirjautuneelle käyttäjälle. Käyttöliittymäkomponentti voisi olla esimerkiksi painike, jonka tyylin käyttäjä voisi muokata ja tallentaa.

4 KÄYTETYT TEKNOLOGIAT

REST-rajapinnan kehitykseen oli valittava teknologia, jolla se toteutetaan. Laadunvalvontasovelluksen mobiilikäyttöliittymä oli kehitetty JavaScriptillä, joten taustajärjestelmän toteutus samalla kielellä tuntui luonnolliselta valinnalta. Rajapinnan kehittämiseen päätettiin käyttää Node.js:a ja Express.js:a.

Tässä luvussa käydään läpi kehityksessä käytettyjä teknologioita. Ensimmäisessä alaluvussa käydään läpi Node.js ajoympäristöä ja sitä, miten se toimii. Lisäksi esitellään Noden moduulien hallinta järjestelmää. Toisessa alaluvussa käsitellään Express.js sovelluskehystä ja sen ominaisuuksia. Kolmannessa alaluvussa tutkitaan, mitä tarkoittaa HTTP:n suojattu versio HTTPS. Neljännessä alaluvussa kuvataan MySQL tietokantaa ja miten siihen otetaan yhteys Node-ajoympäristöstä.

4.1 Node.js

Node.js on palvelinpuolen alusta, joka pohjautuu Google Chromen V8 JavaScript-moottoriin. Sen kehitti Ryan Dahl vuonna 2009. Se on avoimen lähdekoodin alustariippumaton ajoaikainen ympäristö palvelinpuolen ja verkkosovellusten kehittämiseen. Node.js-sovellukset on kirjoitettu JavaScriptilla, ja niitä voidaan ajaa OS X:ssä, Microsoft Windowsissa ja Linuxissa. (Tutorialspoint a). Kuvassa 7 on esimerkki Node.js palvelinkoodista.

node.js

node.js > server

```
1   const hostname = '127.0.0.1'
2   const port = process.env.PORT
3
4   const server = http.createServer((req, res) => {
5       res.statusCode = 200
6       res.setHeader('Content-Type', 'text/plain')
7       res.end('Hello World!\n')
8   })
9
10  server.listen(port, hostname, () => {
11      console.log(`Server running at http://${hostname}:${port}/`)
12  })
```

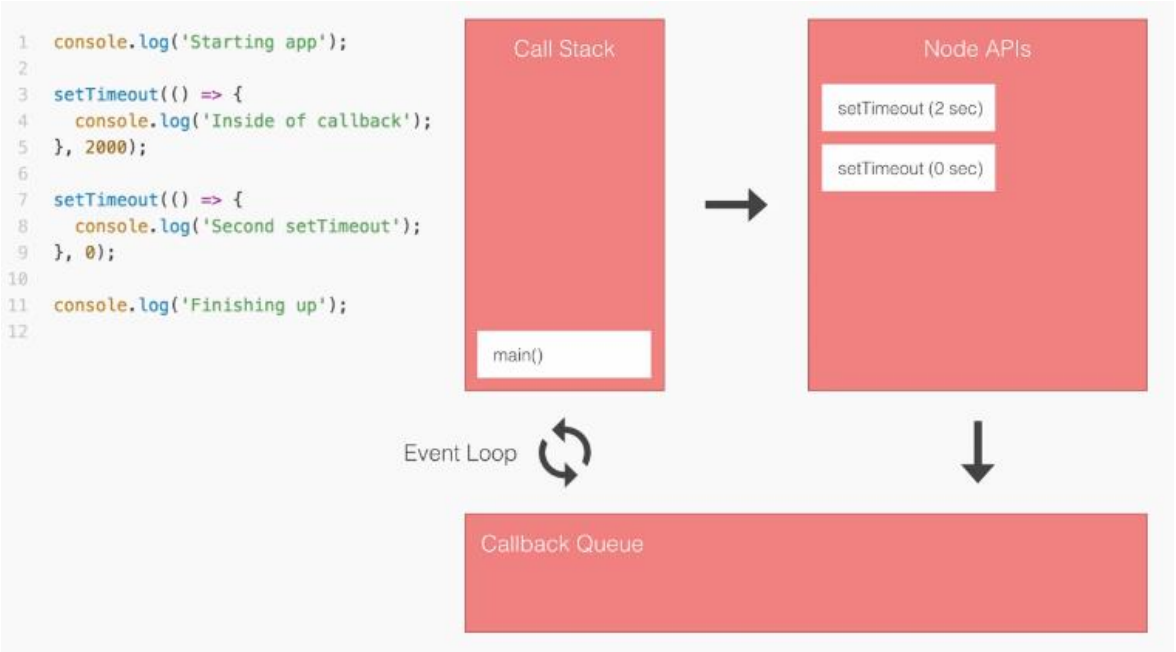
Kuva 7. Esimerkki Node.js palvelinkoodi

4.1.1 I / O-malli

Node.js käyttää tapahtumavetoista ei-keskeyttävää I / O-mallia (non-blocking I / O model), joka tekee siitä kevyen ja tehokkaan. Ei-keskeyttävässä I / O-mallissa operaatiot eivät blokkaa toisten operaatioiden toimintaa, vaan ne suoritetaan rinnakkain. Operaatioita voivat olla esimerkiksi tietokannasta tiedon haku tai http-kutsu. Molemmat operaatiot vievät oman aikansa. Ei-keskeyttävässä I / O-mallissa tietokannan kanssa tehtävä toiminto ei estä http-kutsun käsittelemistä. (Patel 2018.)

4.1.2 Tapahtumasilmukka

Ei-keskeyttävän I / O operaatioiden suorittamisen mahdollistaa tapahtumasilmukka (Node.js). Kuvassa 8 on esitelty miten Node.js hoitaa synkronisen ja asynkronisen koodin suorittamisen. Kaikki synkroniset funktiokutsut lisätään suoraan kutsupinoon (Call Stack). Asynkroniset funktiot käyttävät hyväkseen Noden rajapintoja (Node API's). Kun asynkroninen funktio on valmis, se lisätään takaisinkutsujonoon (Callback Queue). Tapahtumasilmukka (Event Loop) odottaa, että kutsupino on tyhjä, jonka jälkeen se lisää takaisinkutsujonossa olevan prosessin kutsupinoon. (Roberts 2014.)



Kuva 8 Node.js tapahtumasilmukka (Roberts 2014)

4.1.3 Node Moduulit ja NPM

Node.js:n rakenne perustuu moduuleihin. Moduulit ovat kirjastoja, jotka ovat luotu erilaisiin käyttötarkoituksiin. Niitä voidaan kirjoittaa itse ja niitä voidaan luomisen jälkeen käyttää

muissa Node.js sovelluksissa. Node.js:ssa on sisäänrakennettuja moduuleja, joita voidaan käyttää ilman lisäasennuksia.

NPM on maailman suurin ohjelmistorekisteri ja se on myös Node-ohjelmistojen hallinta-manageri ja asentaja. Rekisteri koostuu yli 800,000 ohjelmistopaketista. NPM pitää sisällän myös oman komentorivi työkalun, jota käytetään ohjelmistojen lataamiseen ja asentamiseen. (W3schools.) Moduulit asennetaan komentorivityökalulla kuvan 9 mukaisesti.

```
$ npm install express
```

Kuva 9 Paketin asentaminen NPM komentorivityökalulla

4.2 Express.js

Express.js on suosituin Node-verkkosovelluskehys, jolla voidaan kehittää ohjelmistorajapintoja, verkkosivuja, verkkosovelluksia ja taustajärjestelmiä. Express.js:n kehitti TJ Holowaychuk. Tällä hetkellä sitä ylläpitää Node.js Foundation ja lukuisat avoimen lähdekoodin toimijat. (Tutorialspoint 2020)

Express.js tuo lukuisia hyviä ominaisuuksia taustajärjestelmän kehitykseen. Laadunvalvontasovelluksen näkökulmasta tärkeimmät ovat pyyntöjen käsittelijöiden luominen erilaisille http kutsuille, sekä mahdollisuus lisätä välikäsittelijöitä (middleware).

Express.js itsessään on minimalistinen kehys, mutta kehittäjät ovat luoneet siihen lukuisia välikäsittelijäpaketteja, joilla voidaan ratkaista melkein mitkä tahansa ongelmat verkkosovelluskehityksessä. On olemassa kirjastoja muun muassa evästeille, istunnoille, käyttäjän kirjautumisille, URL parametreille, post-datalle ja tietoturvaotsikoille. Kuvassa 10 on lyhyt esimerkki Express.js sovelluksesta.

```
1   const express = require("express");
2   const app = express();
3
4   app.get("/", (req, res) => {
5       res.send("hello world");
6   })
7
8   app.listen(3000, () => {
9       console.log("app listening on port 3000")
10  })
```

Kuva 10. Esimerkki Express.js:lla

4.2.1 Middleware

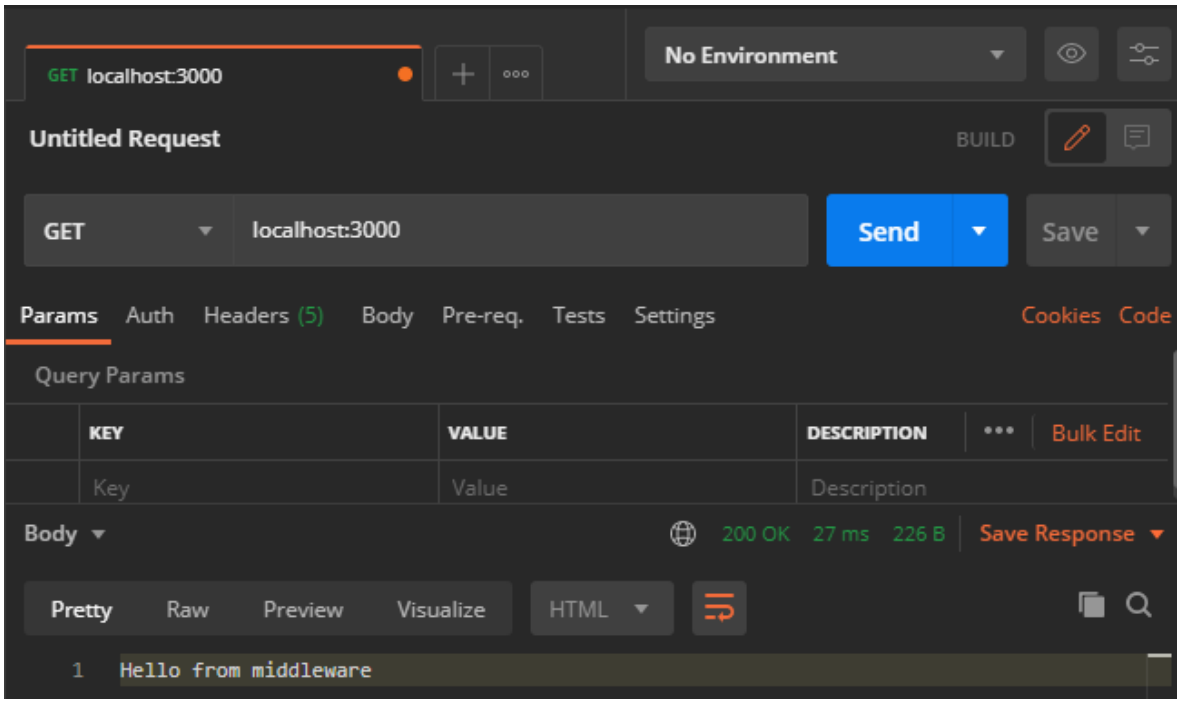
Expresskehyksen toiminta perustuu suurimmaksi osaksi välifunktioiden, eli middleware-funktioden käyttöön. Middleware-funktioilla on käytössää request-objekti, response-objekti sekä next-funktio. Next-funktiota kutsumalla suoritetaan seuraava vaihe. Middleware-funktiossa voidaan esimerkiksi suorittaa koodia, tehdä muutoksia request- ja response-objekteihin, lopettaa request-response sykli tai kutsua next-funktiota . On äärimmäisen tärkeää huolehtia siitä, että next-funktiota kutsutaan, jos middleware ei lopeta request-response sykliä.

Kuvassa 11 Express-sovellukseen on lisätty middleware-funktio. Middleware-funktio luodaan rivillä 4 ja sen parametreiksi määritellään request-objekti, response-objekti ja next-funktio. Funktiossa response-objektiin määritellään message ominaisuus, jonka jälkeen kutsutaan next-funktiota. Middleware ladataan sovellukseen use-funktiolla, kuten rivillä 9 on tehty. Middleware-funktiot suoritetaan siinä järjestyksessä, missä ne on sovellukseen ladattu. Rivillä 11 palvelimen juureen määritellään GET-pyyntöön vastaus, joka on aiemmin määritelty response-objektin message ominaisuus.

```
1   const express = require("express");
2   const app = express();
3
4   const myMiddleware = (req, res, next) => {
5       res.message = "hello from middleware";
6       next();
7   }
8
9   app.use(myMiddleware);
10
11  app.get("/", (req, res) => {
12      res.send(res.message);
13  })
14
15  app.listen(3000, () => {
16      console.log("app listening on port 3000")
17  })
```

Kuva 11 Middleware

Kuvassa 12 on kuvakaappaus Postman työkalun käyttöliittymästä. Kuvassa on tehty pyyntö sovellukseen, josta on saatu middleware-funktiossa asetettu viesti.

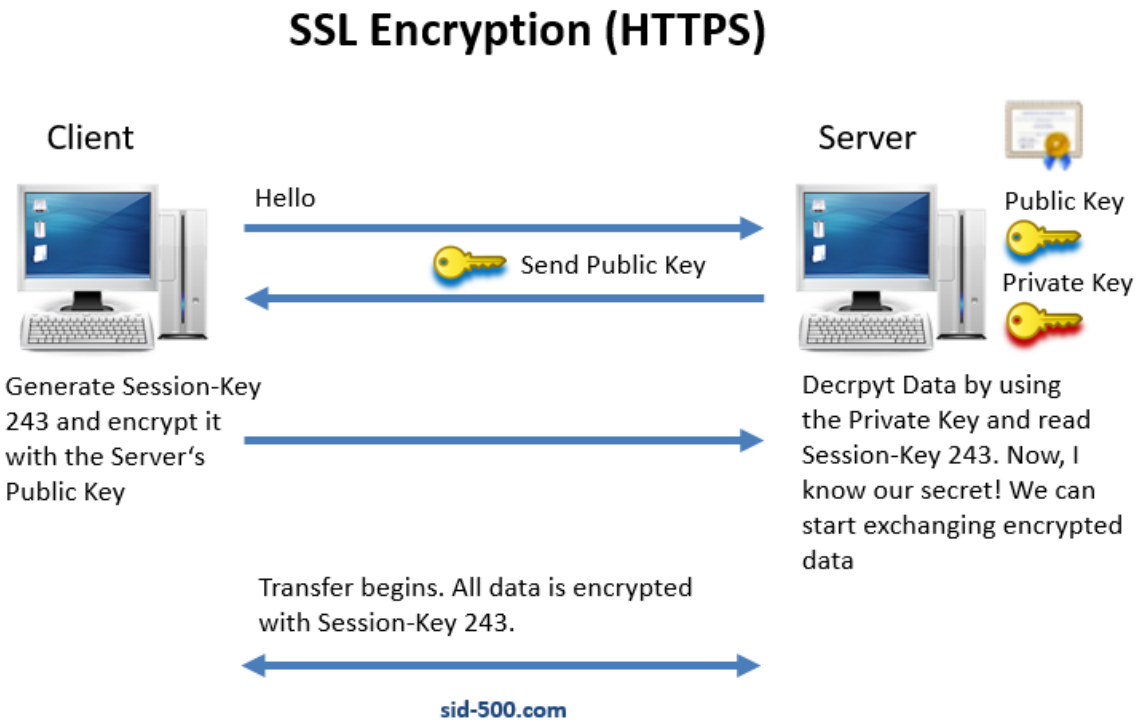


Kuva 12. Postmanilla tehty GET pyyntö

4.3 HTTPS

HTTPS (englanniksi Hyper Text Transfer Protocol Secure) on HTTP-protokollan ja TLS/SSL-protokollan yhdistelmä. Sitä käytetään tiedon siirron suojaamiseen. Suojaus tapahtuu käyttämällä sertifikaattitiedostoja, jotka pitävät sisällään muun muassa kryptaukseen ja dekryptaukseen käytettävät avaimet. Kuvassa 13 on yksinkertaistettu esitys asiakas- ja palvelinkoneen avainten vaihdosta. Asiakas lähettää viestin palvelinkoneelle, jonka jälkeen palvelin vastaa asiakkaalle sen julkisella avaimella. Asiakas luo istuntoavaimen ja kryptaa sen palvelimen antamalla julkisella avaimella. Sen jälkeen palvelin dekryptaa viestin palvelimen yksityisellä avaimella. Avainten vaihdon jälkeen kaikki lähetetty data on suojattu.

Sertifikaateilla voidaan halutessa varmistaa, että onko se kone, jonka kanssa nyt halutaan kommunikoida, juuri se, joka se väittää olevansa. Sekä lähettäjä, että vastaanottaja voi halutessaan vaatia toisen osapuolen identiteetin todentamista, ennen kuin mitään muuta dataa aloitetaan ottamaan vastaan.



Kuva 13. HTTPS kommunikointi (Patrick Gruenauer 2019)

Tiedonsiirrossa käytettävät avaimet säilytetään sertifikaattitietostoissa.

4.4 MySQL

MySQL on avoimenlähdekoodin, relaatiotietokannan hallinta ohjelmisto (Relational Database Management System). Tietokanta yksinkertaisuudessaan on kokoelma jäsenneltyä tietoa. Relaatiotietokanta tarkoittaa, että tietojoukkoon tallennetut tiedot on järjestetty taulukoiksi. Taulukot voivat sisältää riippuvuuksia toisiinsa. Muutamat isot sovellukset, kuten Facebook, Twitter, YouTube, Google ja Yahoo käyttävä MySQL:a tietojen tallentamiseen. (Dwika 2020)

Node.js ja MySQL tietokannan yhteys tehdään siihen tarkoitetulla moduulilla. Moduuli asennetaan syöttämällä konsoliin kuvan 14 mukainen komento.

```
$ npm install mysql
```

Kuva 14 MySQL asennus

Kuvassa 15 on kirjoitettu koodiesimerkki yhteyden muodostamisesta. Rivillä 10 tietokantaan tehdään kysely, jossa haetaan kaikki viestit viesti taulukosta. Konsolissa näkyy saatu viesti.

```
1   var mysql = require('mysql');
2   var pool = mysql.createPool({
3       connectionLimit: 10,
4       host: 'localhost',
5       user: 'testuser',
6       password: 'Password123',
7       database: 'test'
8   });
9
10  pool.query('SELECT * from messages', function (error, results, fields) {
11      if (error) throw error;
12      console.log('Message received: ', results[0].message);
13  });
```

```
PROBLEMS   OUTPUT   TERMINAL   DEBUG CONSOLE
C:\Program Files\nodejs\node.exe .\mySQL
Message received:  hello from MySQL
```

Kuva 15. MySQL yhteys Node.js:lla

5 TAUSTAJÄRJESTELMÄN TOTEUTUS

Taustajärjestelmä toteutettiin noudattaen REST arkkitehtuurityyliä. Sovellus ja rajapinnat tehtiin Node.js:lla ja Express.js:lla.

Testaamiseen tarkoitettu virtuaalikone ja välityspalvelin oli jo valmiiksi asennettu Azure-pilvipalveluun. Virtuaalikoneen käyttöjärjestelmänä oli Windows 10. Työ aloitettiin asentamalla palvelimelle tarvittavat ohjelmistot. Vaaditut ohjelmistot olivat MySQL-tietokannanhallintajärjetelmä ja Node.js. Nämä asennukset sujuivat ilman sen suurempia ongelmia.

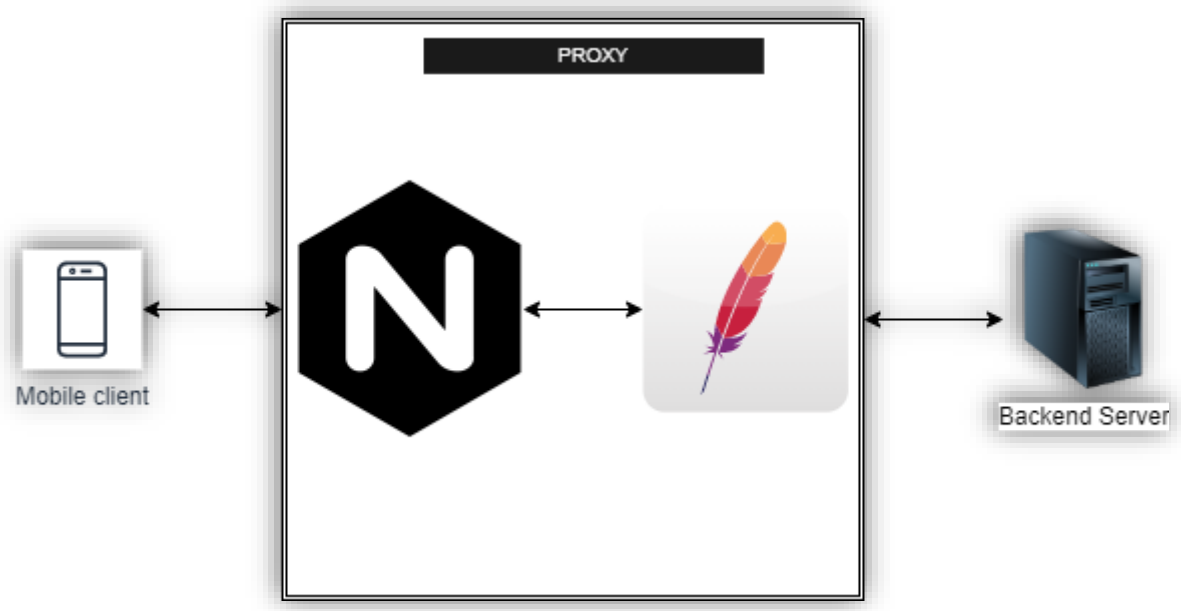
Kehitystyötä helpottamaan ja parantamaan palvelimelle päätettiin asentaa myös TypeScript ohjelmisto. TypeScriptilla tuodaan vakio JavaScriptiin lisäominaisuuksia. TypeScript koodi muunnetaan ajettavaksi JavaScript koodiksi.

Lukujen rakenteessa pyritään noudattamaan samaa kaava, kuin miten data kulkee järjestelmässä eri komponenttien läpi. Ensimmäisenä esitellään koko sovelluskokonaisuuden datankulku ja katsotaan, mitä asioita piti tehdä ennen itse sovelluskoodin kirjoitusta. Tiedonkulun rakenteen esittelyn jälkeen käydään läpi itse taustajärjestelmäsovelluksen toteutetusta ja tarkastellaan, miten jotkin asiat toteutettiin koodi näkökulmasta.

5.1 Datan kulku

Yksi tärkeimmistä asioista sovelluksen kehityksessä on varmistaa sen tietoturvallinen toteutus. Sovellusta ajetaan Azuren pilvipalvelun yksityisessä verkossa. Jotta taustajärjestelmään saisi yhteyttä mobiilikäyttöliittymästä, oli niiden väliin luotava välityspalvelin. Välityspalvelin toimii tiedon välittäjänä mobiilisovelluksen ja taustajärjestelmän välillä. Tällä tavoin saadaan koko järjestelmä kokonaisuuteen luotua lisäturvaa.

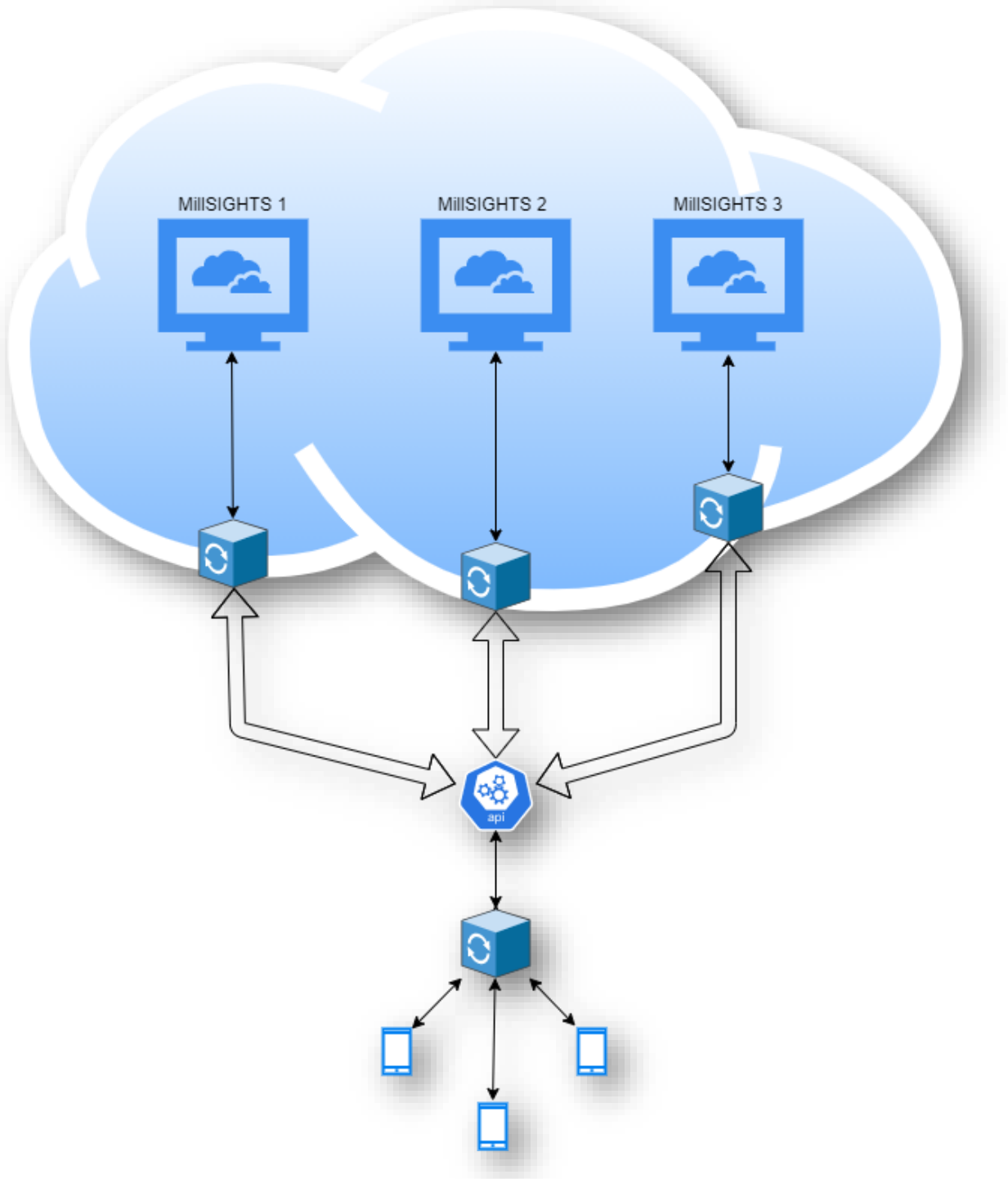
Kuvassa 16 mobiilikäyttöliittymästä lähtevä viesti saapuu välityspalvelimelle, Nginx tarkistaa muun muassa onko lähettäjän IP tunnettu. Jos viesti hyväksytään, niin sen jälkeen viestin vastaan ottaa Apache24-ohjelmisto, joka hoitaa kommunikoinnin taustajärjestelmän kanssa.



Kuva 16. Proxy

Välityspalvelin oli suurimmaksi osaksi jo konfiguroitu lukuun ottamatta IP-rajoituksia ja sertifikaatteja. Tarvittavat HTTPS sertifikaatit luotiin Rauten omalla sertifikaatin luontiohjelmistolla, joita Apache24 käyttää kommunikointiin taustajärjestelmän kanssa. IP rajoitusten ja sertifikaattien takia voidaan olla melko varmoja, että taustajärjestelmään saapuvat viestit ovat tunnetusta lähteestä.

Kuvassa 17 on esiteltynä tiedon kulku koko järjestelmä läpi, päätelaitteesta MillSIGHTS:iin. Kuvassa mobiililaitteelta lähetetään mittausdataa taustajärjestelmän välityspalvelimelle. Taustajärjestelmästä data lähtee mittaajan yrityksen MillSIGHTS välityspalvelimelle, josta se pääty yrityksen MillSIGHTS raportointijärjestelmään.



Kuva 17. Datan kulku

5.2 Mittausten vastaanottaminen ja tallentaminen

Jokaiselle palvelimen vastaanottamalle pyynnölle täytyy tehdä erilaisia toimintoja. Ensimmäiseksi palvelimen täytyy validoida pyynnön sertifikaatti. Taustajärjestelmä konfiguroidaan siten, että vain oikeutettua sertifikaattia käyttämällä lähettäjän pyynnöt käsitellään.

HTTPS-moduulia hyödyntämällä Express saatiin ajettua HTTPS palvelimena. Ensimmäisenä askeleena oli importoida koodiin Express- ja HTTPS-moduuli (Kuva 18).

```
1   // express module
2   import express from "express";
3   const app = express();
4
5   // https module
6   import https from "https";
```

Kuva 18. Express- ja HTTPS-moduuli

Käytettyjä sertifikaatti tiedostoja oli kolmea eri muotoa. Pem-tiedosto pitää muun muassa sisällään tiedon varmenteen omistajasta (Certificate Authority (CA)). Crt-tiedosto sisältää itse allekirjoitetun sertifikaatin ja Key-tiedosto sen yksityisen salausavaimen. Palvelin asetettiin kuuntelemaan ympäristömuuttujiin määritettyä porttia, käyttäen edellä mainittuja sertifikaatteja (Kuva 19). Tällä tavoin palvelin hyväksyy vain salatun yhteyden kautta tulevat viesti ja joiden sertifikaatit ovat palvelin sertifikaattien pari.

```
79  // https server settings
80  const options: https.ServerOptions = {
81    key: fs.readFileSync(process.env.SSL_KEY_PATH),
82    cert: fs.readFileSync(process.env.SSL_CERT_PATH),
83    ca: fs.readFileSync(process.env.SSL_CA_PATH),
84    requestCert: true,
85    rejectUnauthorized: true,
86  };
87
88  // start API
89  https.createServer(options, app).listen(process.env.SERVER_PORT, () => {
90    logger.info(`HTTPS Server running on port ${process.env.SERVER_PORT}`);
91  });
```

Kuva 19. HTTPS-palvelin

Sertifikaatin validoinnin jälkeen pyyntö ohjataan oikeaan reittiin.

5.2.1 Reititys

Palvelimella on neljä pääreittiä, *login*, *measurements*, *users* ja *companies*. Ensimmäinen käyttäjän käyttämä reitti on Login, johon käyttäjä lähettää kirjautumis- pyynnön halutessaan kirjautua järjestelmän sisään. Kirjautumis- pyynnössä käyttäjä lähettää JSON objektin, joka pitää sisällään käyttäjän käyttäjänimen ja salasanan. Käyttäjänimi ja salasana

tarkistetaan tietokannasta ja jos niitä vastaava rivi löytyy, käyttäjä saa palautus viestissä tokenin. Tokenia ja sen validointia käydään tarkemmin läpi seuraavassa luvussa.

Measurements-reittiä käytetään mittausresurssien manipulointiin. Users ja Companies-reiteillä käyttäjä voi manipuloida käyttäjä- ja yritys resursseja, sillä oletuksella, että hänen käyttäjäoikeutensa siihen riittävät. Kaikki Reitit tehtiin yksittäisiksi moduuleiksi, jotka tuotiin pääohjelmistoon (Kuva 20).

```
61  // route modules
62  import login from "./routes/login";
63  import measurements from "./routes/measurements";
64  import users from "./routes/users";
65  import companies from "./routes/companies";
66
67  app.use("/login", login);
68  app.use("/measurements", measurements);
69  app.use("/users", users);
70  app.use("/companies", companies);
```

Kuva 20. Reititys-moduulit

5.2.2 JWT käyttäjän autentikointi

Sovellukseen tulevat viestit validoidaan jo sertifikaattien avulla, mutta tämä ei yksinään riitä. Erilaisten käyttäjätasojen hallitseminen täytyy tehdä mahdolliseksi. Reitit ja resurssit täytyy pystyä suojaamaan, ettei esimerkiksi tavallinen käyttäjä pysty lisäämään tai poistamaan muita käyttäjiä, vaan sen pystyy tekemään vain pääkäyttäjät. Kevyt ja sopiva ratkaisu tähän oli JWT:a (JSON Web Token) käyttävä validointimenetelmä.

JWT on avoin standardi, joka määrittelee kompaktin ja itsenäisen tavan tietojen turvalliseen lähetykseen osapuolten välillä JSON-objektina. JWT-voidaan allekirjoittaa salausavaimella tai käyttämällä julkista/yksityistä avainparia. Allekirjoitus tehdään RSA- tai ECDSA-algoritmilla. (JWT.io)

Token koostuu kolmesta osasta, jotka ovat: otsikko (header), tietosisältö (payload) ja allekirjoitus (signature).

Otsikko-osa koostuu yleensä kahdesta osasta. Tokenin tyypistä, joka on JWT ja allekirjoitus algoritmista (Kuva 21). Algoritmi voi olla HMAC SHA256 tai RSA.

```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

Kuva 21. JWT-otsikko (JWT.io)

Tietosisältö voi sisältää useitakin eri asioita, mutta tärkeimmät näistä ovat esimerkiksi, tokenin vanhentumisaika ja data, joka usein pitää sisällään joitakin käyttäjän tietoja (Kuva 22).

```
{
  "sub": "1234567890",
  "name": "John Doe",
  "admin": true
}
```

Kuva 22. JWT-tietosisältö (JWT.io)

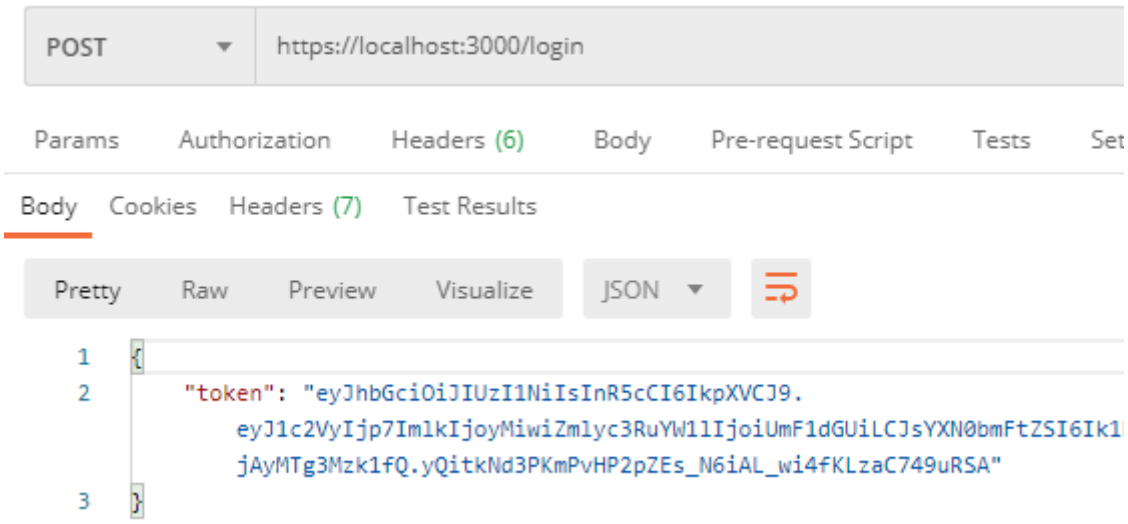
Allekirjoitus luodaan käyttämällä koodattua otsikkoa, tietosisältöä, salausavainta ja algoritmia, joka on määritelty otsikossa. Kuvan 23 esimerkissä käytetään HMAC SHA256 algoritmia.

```
HMACSHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
  secret)
```

Kuva 23. JWT-allekirjoitus (JWT.io)

Otsikko, tietosisältö ja allekirjoitus koodataan Base64 merkkijonoksi, ennen sen palautusta käyttäjälle. Lopullinen koodattu token on muotoa xxxxx.yyyyy.zzzzz. Kuvassa 24

sovellukseen on tehty kirjautumis- pyyntö, josta on saatu vastauksena token.



Kuva 24. Kirjautuminen

Sovellukseen asennettiin jsonwebtoken kirjasto. Asennus tehdään syöttämällä konsoliin kuvan 25 mukainen komento.

```
$ npm install jsonwebtoken
```

Kuva 25 jsonwebtoken kirjaston asennus

Asennuksen jälkeen kirjasto tuodaan haluttuun moduuliin (Kuva 26, rivi 1). Sign-metodiin määritellään kryptattava tietosisältö ja salausavain. Sign-metodi palauttaa tokenin, jonka käyttäjä saa kirjautumisen yhteydessä.

```
1   const jwt = require("jsonwebtoken");
2   let token = jwt.sign({ payload: "tietosisältö" }, "salausavain");
```

Kuva 26 Esimerkki jsonwektoken kirjaston käytöstä

Kun käyttäjä on kirjautunut onnistuneesti ja saanut oman tokenin, hän on valmis lähettämään oman mittauksensa. Hän tekee POST-pyynnön, measurements-päätepisteeseen. Ennen tietojen tallentamista tietokantaan, suoritetaan middleware funktio (kuva 27), joka tarkistaa ensin sisältääkö sen header authorization kentän ja onko sen sisällä token. Jos tokenia ei löydy pyyntöön vastataan 401 statuksella, joka tarkoittaa, ettei kyseisellä käyttäjällä ole oikeuksia. Jos token löytyy, sen salausavainta verrataan sovelluksen käyttämään avaimeen. Jos vertaus onnistuu, token puretaan, josta saadaan kyseisen käyttäjän tiedot.

```
105  function verifyToken(req: express.Request | any, res: express.Response, next: express.NextFunction) {
106    const authHeader = req.headers.authorization;
107    const token = authHeader && authHeader.split(" ")[1];
108    if (token == null) {
109      // token was not in headers
110      return res.sendStatus(401);
111    } else {
112      jwt.verify(token, process.env.ACCESS_TOKEN_SECRET, (err: JsonWebTokenError) => {
113        if (err) {
114          // token does not have correct key
115          logger.info("token", err);
116          return res.sendStatus(401);
117        } else {
118          // decode token and assing it to req objects user property
119          const decoded = jwt.decode(token, { json: true });
120          req.user = decoded.user;
121          next();
122        }
123      });
124    }
125  }
```

Kuva 27. Tokenin tarkistus

5.2.3 Mittausten tallentaminen

Mobiilikäyttöliittymästä piti pystyä ottamaan ja lähettämään kuvia. Yhteisen päätöksen mukaan kuvat lähetetään POST-viestin imagekentässä koodattuna base64-merkkijonoksi. Jos kuvaa, ei ole, niin imagekenttä on null. Ennen mittaustietojen tallennusta tietokantaan, sovelluksen täytyy tarkistaa, sisältääkö mittausdatan imagekenttä mitään tietoa (Kuva 28).

```
if (body.image !== null) {
  let companyname;
  companyname = await this.getCompanyName(user.companyid);
  const now = new Date().getTime();
  imageSaver(`${imagesFolderPath}\\${companyname}\\${now}.jpg`, body.image);
  body.image = `${companyname}\/${now}.jpg`;
}
```

Kuva 28. Kuvan valmistelu tallennukseen.

Jos kuva löytyy, se tallennetaan palvelimelle sille määrättyyn paikkaan. Nimeämiskäytännössä päätettiin noudattaa tyyliä yritys/aika. Ajalla varmistettiin, ettei kuvien nimet

koskaan ole samat. Kuvat tallennettiin fs-moduulin asynkronisella writefile-funktiolla (kuva 29).

```
1   import { writeFile } from "fs";
2   import logger from "../utils/logger";
3
4   export function imageSaver(path: string, base64: string) {
5     writeFile(`${path}`, base64, "base64", (err) => {
6       if (err) {
7         logger.error(`save image: ${err}`);
8       }
9     });
10  }
```

Kuva 29. Kuvan tallennusmoduuli

Kun POST-pyyntö on läpäissyt sertifikaatti- ja JWT validoinnit, sekä kuvankäsittely tehty, se voidaan tallentaa tietokantaan. Tietokanta sisältää 3 mittataulua, lähettämättömät-, lähetetyt- ja epäonnistuneetmittaukset. Taustajärjestelmään saapuvat mittaukset tallennetaan lähettämättömät mittaukset-tauluun. Epäonnistuneisiin mittauksiin luetellaan sellaiset mittaukset, jotka ovat vajaita ja joita MillSIGHTS ei hyväksynyt vastaanotettavaksi. Kaikkien taustajärjestelmän tekemien validointien jälkeen tällaisia mittauksia ei pitäisi tietokannassa olla, mutta taulu päätettiin tehdä varmuuden vuoksi.

On tärkeää, että tietokantaan tehtävät kyselyt tehdään asynkronisesti, jotta esimerkiksi tallentamisen odottaminen ei blokkaa muiden toimintojen suorittamista. Sen vuoksi tallennus tapahtuma toteutettiin käyttämällä JavaScriptin Promiseja. Promise on luonteeltaan asynkroninen. Promisella voi olla kaksi lopputulosta. Se voi olla joko täytetty (englanniksi fullfilled) tai hylätty (englanniksi rejected). Promise, eli lupaus täytetään kutsumalla resolve-funktiota ja hylätään kutsumalla reject-funktiota. Kuvassa 30 tietokantaan tehdään kysely. Tässä tapauksessa kysely on mittauksen tallentaminen. Jos mittauksen tallentaminen onnistuu, resolve-funktiota kutsutaan success parametrilla. Jos tallennus jostain

syystä epäonnistuu, reject-funktiota kutsutaan error parametrilla, joka pitää sisällä MySQL virhekoodin.

```
return new Promise((resolve, reject) => {
  pool.query(sql, [JSON.stringify(body), user.companyid], (err, results) => {
    if (err) {
      const error: IErrorResponse = { error: { code: err.code } };
      reject(error);
      return;
    } else {
      const success: ISuccessResponse = { success: { count: results.affectedRows } };
      resolve(success);
      return;
    }
  });
});
```

Kuva 30. mittauksen tallentaminen

Promisen tulosta odotellaan, jonka mukaan käyttäjälle palautetaan vastaus (Kuva 31). Kuvassa tietokantaan tallennusfunktion eteen lisätään await määre, jolloin lopun koodin suoritus lakkaa siihen asti, kunnes kyseinen funktio on saatettu loppuun. Jos lisääminen onnistuu käyttäjälle, palautetaan tieto tallennuksen onnistumisesta. Jos Promise hylätään, suoritetaan catch lohko sisältämä koodi, jossa käyttäjä saa vastauksena status koodin 500. Virhekoodi tallennetaan virhelokitiedostoon.

```
try {
  const user: IUser = req.user;
  const results = await db.saveMeasurementToUnsentTable(req.body, user);
  return res.json(results);
} catch (error) {
  logger.error("measurements_post", error);
  return res.sendStatus(500);
}
```

Kuva 31. Try-catch-lohko

5.3 Mittausten lähetyssilmukka

Oli toteutettava tapa, jolla jokainen lähettämätön mittaus saadaan lähetettyä oikeaan mittaajan yrityksen MillSIGHTS osoitteeseen. Lähetyksen täytyi olla erillinen tapahtuma, joka ei suoraan liity REST-toimintoihin. Tämän vuoksi mittausteen lähettämiseen tehtiin ”communicator” -moduuli.

Jokainen taustajärjestelmän tietokantaan tallennettu mittaus, sisältää ”isSent” kentän. Kentän tietotyyppi on boolean. Jos arvo on tosi, se tarkoittaa, että mittaus on lähetetty

onnistuneesti MillSIGHTS: iin ja jos se on epätosi, mittausta ei ole vielä lähetetty. Jotta mittaus voidaan syöttää lähetyssilmukkaan, sitä ennen täytyy tarkistaa, onko tietokannassa lähettämättömiä mittauksia.

5.3.1 Lähettämättömien mittausten tarkistaminen

Lähettämättömien mittausten tarkistus tehdään itsenäisenä funktiona, joka on erillään muusta sovelluksen toiminnasta. Kuvassa 32 on luotu lähettämättömien mittausten tarkistusmoduuli. Rivillä 5 olevalla muuttuja pitää sisällään tiedon siitä, onko edellisten mittausten prosessointi valmis. Rivillä 7 ”checkUnsentMeasurements” -funktio määritellään asynkroniseksi, jotta se ei estä muun koodin suoritusta sovelluksessa. Rivillä 11 tehdään kysely tietokantaan, onko siellä yhtään lähettämättömiä mittauksia. Jos mittauksia löytyy, suoritetaan silmukka (rivi 13), joka käy läpi jokaisen mittauksen ja hakee sen mittauksen yrityksen tiedot, jonka jälkeen ”sentDataToMillSights” -funktiota kutsutaan sen mittauksen tiedoilla.

TS index.ts

src > communicator > TS index.ts > ...

```
1   import sendDataToMillSights from "./functions/sendDataToMillSights";
2   import logger from "../utils/logger";
3   import db from "./database/db";
4
5   let checking = false;
6
7   const checkUnsentMeasurements = async () => {
8     if (!checking) {
9       checking = true;
10      try {
11        const unsentmeasurements: any = await db.getUnsentMeasurements();
12        if (unsentmeasurements.length > 0) {
13          for (const measurement of unsentmeasurements) {
14            const companyData: any = await db.getCompanysMillurlAndCertificate(measurement.companyid);
15            // if resolve is called in this function then everything is ok.
16            // if reject is called, reject message will be cathed in catch block
17            const results = await sendDataToMillSights(measurement, companyData);
18            logger.info(`measurement with id ${measurement.eventid} status: ${results}`);
19          }
20        }
21      } catch (error) {
22        logger.info(error);
23      }
24      checking = false;
25    }
26    else {
27      logger.info("processing previous records");
28    }
29  };
30
31  export = checkUnsentMeasurements;
```

Kuva 32. Lähettämättömienmittausten tarkistusmoduuli

5.3.2 Lähetys MillSIGHTS: iin

Mittausdatan lähetys MillSIGHTS: iin tapahtuu kolmen eri pyynnön avulla. URI:n alkuosa on muotoa mis/data. Lähetys tapahtumana koostuu kolmesta eri pyynnöstä, cansend, send ja isdone. Cansend pyynnöllä varmistetaan, että kyseinen MillSIGHTS-palvelin on valmis ottamaan pyyntöjä vastaan. Send pyynnössä lähetetään itse mittausdata, josta saadaan vastauksena ticket numero. Isdone pyynnöllä tarkistetaan, että onko mittausdata varmasti saapunut perille ja onko esimerkiksi sen tallentaminen tietokantaan onnistunut. Isdonea kysytään send -pyynnöstä saadulla tiketti numerolla.

Edellä mainitut kolme pyyntöä suoritetaan mainitussa järjestyksessä. Sovelluksen täytyy pystyä säilyttämään toimintansa, vaikka nämä pyynnöt epäonnistuisivat. Jokainen pyyntö suoritetaan maksimissaan viisi kertaa ja jokaisen epäonnistuneen yrityksen jälkeen pidetään kymmenen sekunnin tauko. Jos mikä tahansa pyyntö epäonnistuu 5 kerran jälkeen, silmukasta poistutaan.

”isdone” -pyynnöstä voidaan saada vastauksena neljä eri statusta. Statukset ovat:

- 0 invalid
- 1 received
- 2 success
- 3 error
- 4 dublicate

Invalid-status tarkoittaa, että kyseistä ticketti numeroa ei ole olemassa. Received-status tarkoittaa, että data on vastaanotettu, mutta sitä ei ole vielä käsitelty. Success-status tarkoittaa, että data on vastaanotettu ja se on käsitelty. Error-status tarkoittaa, että käsittelyn aikana tapahtui virhe. Dublicate-status tarkoittaa, että tietokannassa on duplikaatti rivi.

Kuvassa 33 on kuvakaappaus taustajärjestelmän lokitiedostosta, johon on kirjattu yksittäisen mittauksen koko lähetystapahtuma. Lokitietojen kerääminen ja säilyttäminen on tärkeää. Niiden implementointia käydään läpi seuraavassa luvussa.

```
12-10-2020 09:51:54 info: cansend value for 545 is true, times asked 1
12-10-2020 09:51:55 info: send id for 545 is 1, times asked 1
12-10-2020 09:51:55 info: isDone status for 545 is 1, times asked 1
12-10-2020 09:51:55 info: isDone status 1 (received) waiting 10 sec and asking again...
12-10-2020 09:52:05 info: isDone status for 545 is 2, times asked 2
```

Kuva 33. Lähetyssilmukan lokitiedot

5.4 Lokitiedot

Yksi asiakasvaatimuksista oli, että taustajärjestelmässä pidetään sen toiminnasta lokitietoja. REST: in kaltaisessa sovelluksessa on hyvä pitää kirjaa esimerkiksi siitä, mitä ja milloin pyyntöjä on vastaanotettu ja onko ne täytetty vai hylätty. On myös hyvä tietää, onko syy hylkäämiseen ollut asiakkaan syytä vai taustajärjestelmän. Virheiden sattuessa kehittäjän on hyvä nähdä tiedoston polku ja tiedoston rivi missä virhe on tapahtunut. Node-ajoympäristössä on lukuisia tapoja pitää lokia. Perinteisin tapa on console.log tai console.error. Näillä lokitietoja voidaan tulostaa konsoliin. Niiden käyttö on hyväksyttävää kehitysvaiheessa, mutta niitä ei pitäisi käyttää, kun sovellus siirtyy tuotantoon, koska ne ovat synkronisia funktioita. (Express.js)

Taustajärjestelmässä päätettiin käyttämään Winston ohjelmistoa, jonka käyttöä esitellään seuraavassa alaluvussa.

5.4.1 Winston

Winston ohjelmisto yksinkertaistaa lokitietojen kirjaamista tiedostoon ja se toimii asynkronisesti. Kuvassa 34 on Winstonin konfigurointi koodi. Logger olio luodaan Winstonin ”createLogger” -metodilla. CreateLogger-metodiin syötettään parametrina yksi asetusolio. Asetusoliossa määritellään muun muassa missä formaatissa lokirivejä kirjoitetaan ja mihin tiedostoon ne tallentuvat.

```
import { createLogger, format, transports } from "winston";

const logger = createLogger({
  level: "info",
  format: format.combine(
    format.timestamp({ format: "DD-MM-YYYY HH:mm:ss" }),
    format.metadata({ fillExcept: ["message", "level", "timestamp", "label"] }),
    format.printf((info) => {
      let out = `${info.timestamp} ${info.level}: ${info.message}`;
      if (Object.keys(info.metadata).length !== 0) {
        out = out + " " + JSON.stringify(info.metadata);
      }
      return out;
    })
  ),
  transports: [
    new transports.File({
      filename: "./logs/error.log",
      level: "error",
      maxsize: 5242880,
      maxFiles: 5,
    }),
    new transports.File({
      filename: "./logs/info.log",
      level: "info",
      maxsize: 5242880,
      maxFiles: 5,
    }),
    new transports.File({
      filename: "./logs/all-logs.log",
      maxsize: 5242880,
      maxFiles: 5,
    }),
    new transports.Console(),
  ],
});

export = logger;
```

Kuva 34. Logger-moduuli

Käyttäjä voi määrittää lokitiedoille eri tasot. Tasoja voi olla esimerkiksi info, error, warn jne. Kuvassa 35 on esimerkki infotason kirjauksesta.

```
logger.log({ level: "info", message: "info viesti" });
logger.info("info viesti");
```

Kuva 35. Winston infotaso

5.5 Sähköpostin lähetys

Yksi asiakasvaatimus oli sähköpostin lähetys. Sovelluksen pitää lähettää sähköpostia pääkäyttäjälle, jos MillSIGHTS: iin ei saada muodostettua yhteyttä lähetyssilmukassa. Sähköpostia lähetetään, jos yhteyttä ei ole saatu muodostettua viiden kerran jälkeen. Sähköpostin lähetys tehtiin Nodemailer-moduulilla. Nodemailer asennetaan syöttämällä konsoliin ”npm install nodemailer”. Asennuksen jälkeen moduuli tuodaan projektiin kuvan 36 mukaisesti.

```
1   import nodemailer from "nodemailer";
2   import Mail from "nodemailer/lib/mailer";
```

Kuva 36. Nodemailer:in tuonti

Nodemailer:in tuonnin jälkeen, luotiin uudelleenkäytettävä kuljetusolio, johon määriteltiin SMTP:n (englanniksi Simple Mail Transfer Protocol) tuottaja ja kirjautumis- tunnukset (kuva 37).

```
5   const client = nodemailer.createTransport({
6     host: "smtp.sendgrid.net",
7     auth: {
8       user: process.env.EMAIL_USER,
9       pass: process.env.EMAIL_PASS,
10    },
11  });
```

Kuva 37. Nodemailer "kuljetusolio"

Lopuksi tehtiin ”sendEmail” -funktio (kuva 38), jota kutsumalla lähetetään sähköposti ”message” -parametriin määritetyllä viestillä.

```
13  const sendEmail = (message: string) => {
14    const email: Mail.Options = {
15      from: "viestin lähettäjä",
16      to: "viestin vastaanottaja",
17      subject: "viestin aihe",
18      text: message,
19    };
20    client.sendMail(email, (err: Error, info: any) => {
21      if (err) {
22        logger.error(err);
23        return;
24      }
25      logger.info(info);
26    });
27  };
```

Kuva 38. Sähköpostinlähetysfunktio

Sähköpostia pitää lähettää, jos esimerkiksi ”cansend” -pyyntö epäonnistuu viisi kertaa. Lähetys-silmukassa kutsutaan ”sendEmail”-funktiota ennalta määrätyllä viestillä (kuva 39, rivi 111).

```
100     do {
101       try {
102         canSendResponse = await canSend(millurl, port, ca, pfx, passphrase);
103       } catch (error) {
104         logger.error(error);
105         canSendResponse = false;
106       }
107       timesCanSendAsked++;
108
109       if (canSendResponse !== true) {
110         if (timesCanSendAsked >= 5) {
111           sendEmail(canSendFailEmailMessage);
112           reject("canSend failed 5 times");
113           break;
114         }
115         await sleep(10000);
116       }
117     } while (canSendResponse !== true);
```

Kuva 39. cansend-silmukka

5.6 Sovelluksen ajo Windows Servicenä

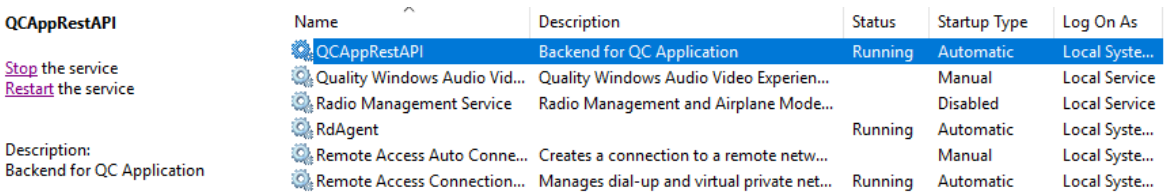
Sovellusta ajetaan Windowsin Servicenä. Sen ajamiseen käytettiin ”Quick Window Service” -moduulia. Moduuli asennetaan syöttämällä konsoliin käsky ”npm install qckwinsvc”.

Windows Service luodaan kuvan 40 mukaisesti. Ensimmäisenä konsoliin syötetään komento ”qckwinsvc”, jonka jälkeen sovellus kysyy muun muassa, millä nimellä Service luodaan ja suoritettavan ohjelmiston polkua.

```
> qckwinsvc
prompt: Service name: Hello
prompt: Service description: Greets the world
prompt: Node script path: C:\my\folder\hello.js
prompt: Should the service get started immediately? (y/n): y
Service installed.
Service started.
```

Kuva 40. Quick Windows Service (Npmjs 2020)

Kun Service on luotu, se ilmestyy Windows Service välilehteen (kuva 41). Välilehdellä Servicen voi käynnistää, pysäyttää tai käynnistää uudestaan.



Kuva 41. Windows Service ikkuna

6 JOHTOPÄÄTÖKSET

Raute halusi kehittää laadunvalvontasovelluksen, joka koostuu mobiilikäyttöliittymästä ja sen taustajärjestelmästä. Taustajärjestelmään kehitettiin myös hallintapaneeli, jolla mittaustietoja, käyttäjiä ja yrityksiä voidaan muokata. Työn laajuuden vuoksi, se jätettiin lopullisesta raportista pois ja keskityttiin vain taustajärjestelmän rajapintaominaisuuksiin ja mittaustietojen lähettämiseen MillSIGHTS:iin.

Taustajärjestelmä kehitettiin Express.js sovelluskehyksellä. Sillä oli helppoa ja nopeaa luoda sovelluksen päätepisteet.

Taustajärjestelmä pyrittiin toteuttamaan REST-arkkitehtuuri tyylillä ja se voidaan katsoa onnistuneeksi. Opinnäytetyön tuloksena oli kevyt ja helposti skaalattavissa oleva taustajärjestelmä.

Opinnäytetyö prosessi eteni suunnitelmien mukaan. Prosessin aikana tuli useita uusia ominaisuuksia, joita toimeksiantaja halusi lisätä taustajärjestelmään. Alun perin työn piti olla yksinkertainen REST-rajapinta, jonka kanssa mobiilikäyttöliittymä kommunikoi ja johon mittaukset tallennetaan. Sen laajuus kuitenkin nopeasti kasvoi hallintapaneelin ja MillSIGHTS kommunikaattorin muodossa.

Taustajärjestelmän ja mobiilikäyttöliittymän kehitys jatkuu. Tällä hetkellä koko laadunvalvontasovellusta valmistellaan testi käyttöön Koskisen puutuotetehtaalle. Sovellusta tullaan muuttamaan saadun palautteen ja kehitysehdotusten mukaan.

Haastavinta työssä oli ehdottomasti lähetyssilmukan luonti. Siinä oli niin monta asiaa, jota piti ottaa huomioon. Piti muun muassa kommunikoida MillSIGHTS: in kanssa, erotella eri yrityksen mittausdatat ja sertifikaatit, sekä kaikki piti tehdä asynkronisesti, jotta sovellus toimii tehokkaasti. Lisähaastetta toi isdone-statusten vuoksi tehtävät ohjaukset.

LÄHTEET

Benharos 2018. What is REST API? in plain English. [kuvaviittaus 13.10.2020]. Saatavissa: https://phpenthusiast.com/blog/what-is-rest-api

Express.js 2020. Production best practices: performance and reliability. [viitattu 16.10.2020]. Saatavissa: https://expressjs.com/en/advanced/best-practice-performance.html

Fielding 2000. Architectural Styles and the Design of Network-based Software Architectures [viitattu 8.10.2020] Saatavissa: https://www.ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation.pdf

Dwika 2020. What is MySQL. [viitattu 29.10.2020]. Saatavissa: https://www.hostinger.com/tutorials/what-is-mysql

JWT.io 2020. Introduction to JSON Web Tokens. [viitattu 13.10.2020]. Saatavissa: https://jwt.io/introduction/

Naemm 2020. Simplify Application Integration with the REST API Browser (Beta). [kuvaviittaus 13.10.2020]. Saatavissa: https://www.astera.com/type/blog/rest-api-integration

Node.js. The Node.js Event Loop, Timers, and process.nextTick(). [viittaus 30.10.2020]. Saatavissa: https://nodejs.org/en/docs/guides/event-loop-timers-and-nexttick/

Npmjs 2020. Quick Windows Service. [kuvaviittaus 14.10.2020]. Saatavissa: https://www.npmjs.com/package/qckwinsvc

OpenJS Foundation 2020. About Node.js [viitattu 10.9.2020]. Saatavissa: https://nodejs.org/en/about/

Patel 2018. What exactly is Node.js. [viitattu 17.9.2020]. Saatavissa: https://www.freecodecamp.org/news/what-exactly-is-node-js-ae36e97449f5/

Raute a. Tietoa Rautesta. [viitattu 22.10.2020]. Saatavissa https://www.raute.fi/fi/tietoa-rautesta

Raute b. Näkymät. [viitattu 22.10.2020]. Saatavissa https://www.raute.fi/fi/nakymat

Raute Academy. Vanerin valmistuksen peruskurssi. [viitattu 6.10.2020] Saatavissa: https://academy.raute.com/

REST Api Tutorial. What is REST? [viitattu 30.10.2020]. Saatavissa: https://www.restapitutorial.com/lessons/whatisrest.html#

Roberts 2014. What the heck is the event loop anyway? [viittaus 17.9.2020]. Saatavissa: https://www.youtube.com/watch?time_continue=1412&v=8aGhZQkoFbQ&feature=emb_title

Tutorialspoint a 2020. What is Node.js?. [viitattu 23.10.2020]. Saatavissa: https://www.tutorialspoint.com/nodejs/nodejs_introduction.htm

Tutorialspoint b 2020. Node.js – Event Loop [viitattu 23.10.2020]. Saatavissa: https://www.tutorialspoint.com/nodejs/nodejs_event_loop.htm

W3schools. What is npm. [viitattu 23.10.2020]. Saatavissa: https://www.w3schools.com/whatis/whatis_npm.asp