



SAVONIA

OPINNÄYTETYÖ - AMMATTIKORKEAKOULUTUTKINTO
TEKNIIKAN JA LIIKENTEEN ALA

AVOIN WEB-RAJAPINTA SENSORIDATALLE

TEKIJÄ/T: Samu Vartiainen

Koulutusala Tekniikan ja liikenteen ala	
Koulutusohjelma/Tutkinto-ohjelma Sähkötekniikan tutkinto-ohjelma	
Työn tekijä(t) Samu Vartiainen	
Työn nimi Avoin web-rajapinta sensoridatalle	
Päiväys 7.9.2020	Sivumäärä/Liitteet 64/10
Ohjaaja(t) Arto Toppinen	
Toimeksiantaja/Yhteistyökumppani(t) Savonia-ammattikorkeakoulu	
<p>Tiivistelmä</p> <p>Ohjelmistokehityksessä tarvitaan usein rajapintoja, sillä eri ohjelmat tai sovellukset vaihtavat tietoja niiden kautta. Ohjelmoijat käyttävät paljon toistensa tekemiä rajapintoja, jonka vuoksi rajapintojen tulee olla hyvin suunniteltuja. Rajapinnan peruserä on tarjota dataa tai toimintoja.</p> <p>Opinnäytetyössä tutkitaan web-rajapintojen tekniikoita ja perehdytään hyvän web-rajapinnan piirteisiin. Suunnitteluvaiheessa valitaan sopivat tekniikat työn toteutukseen, sekä suunnitellaan yksityiskohtaisesti eri ohjelmistojen osat. Toteutusvaiheessa rakennetaan avoin RESTful web-rajapinta, joka dokumentoidaan Swagger-työkalulla. Web-rajapinta tehdään työssä sensoridatalle, jota kerätään Arduino Uno -kehitysalustalta tietokantaan MQTT-protokollan yli. Lisäksi työssä tehdään sivuprojektina rajapintaa käyttävä sovellus, joka on web-käyttöliittymä sensoridatan hallintaan.</p> <p>Opinnäytetyön lopputuloksena syntyi avoin RESTful web-rajapinta sensoridatalle, jonka kautta voidaan hakea, lisätä, muokata ja poistaa dataa. Rajapintaa voi käyttää kuka tahansa, ja se on dokumentoitu Swagger:illa. Datankeruuseen liittyen ohjelmoitiin MQTT-viestejä julkaiseva ja tilaava ohjelma. Web-käyttöliittymästä tuli kätevä sovellus, jolla voi hallita rajapinnan sensoridataa.</p>	
Avainsanat Ohjelmointirajapinta, Web API, REST, Node.js, Angular	

Field of Study Technology, Communication and Transport			
Degree Programme Degree Programme in Electrical Engineering			
Author(s) Samu Vartiainen			
Title of Thesis Public API for Sensor Data			
Date	September 7, 2020	Pages/Appendices	64/10
Client Organisation /Partners Savonia University of Applied Sciences			
<p>Abstract</p> <p>Application programming interfaces allow third-party applications to communicate with other applications and programs. APIs are used a lot when programming software in general. A good API makes it much easier for the developers to do their work.</p> <p>The goal of the thesis was to create a web API for sensor data. The theoretical part of the thesis was done by researching API technologies, how APIs work and the good web API design. Choosing the technologies and planning the work was an important part of the project. The API was made public, so that it is possible for anyone to use the API for all the functions (search, add, edit and delete). The practical part included getting the sensor data from Arduino Uno to a database using the MQTT protocol.</p> <p>As a result of the thesis, a public RESTful web API for the sensor data was created, documented using Swagger tools, two programs were made for publishing and subscribing the MQTT messages, and an user interface web app was also made as a side project to see that the API actually works well in use.</p>			
Keywords Application programming interface, Web API, REST, Node.js, Angular			

SISÄLTÖ

1	JOHDANTO	8
2	OHJELMOINTIRAJAPINNAT YLEISESTI	9
2.1	Rajapintojen perusperiaate	9
2.2	Rajapintojen julkaisukäytännöt	9
2.3	Avoimen rajapinnan ehtoja	9
2.4	Rajapinnat käyttötarkoitusten mukaan	10
3	WEB-RAJAPINTOJEN TEKNIIKAT JA TOIMINTA.....	11
3.1	HTTP	11
3.1.1	HTTP-pyyntö	11
3.1.2	HTTP-vastaus	12
3.1.3	HTTP:n statuskoodit.....	12
3.2	SOAP	13
3.3	JSON.....	13
3.4	REST.....	14
3.4.1	Resurssi ja sen metodit	14
3.4.2	REST:in rajoitteet.....	15
3.4.3	Päätepisteet.....	17
4	HYVÄN WEB-RAJAPINNAN PERIAATTEET	18
4.1	Kunnollinen dokumentaatio	18
4.2	Versiointi	18
4.3	Suojaus.....	19
4.4	Hakutulosten sivutus ja suodattaminen	19
4.5	Seuranta	19
4.6	Esimerkki web-rajapinnasta	20
5	TYÖN SUUNNITTELU	21
5.1	Kokonaisuus	21
5.2	Rajapinnan avoimuus	21
5.3	Rajapinnan data.....	22
5.4	Palvelinohjelman päätepisteet.....	22
5.5	Arduino ja sensoridatan tuonti rajapintaan	23
5.6	Web-käyttöliittymä.....	23

6	TYÖN WEB-TEKNIIKAT	25
6.1	Tekniikan valinta	25
6.2	MEAN stack	25
6.2.1	MongoDB.....	26
6.2.2	Express.js	26
6.2.3	AngularJS	26
6.2.4	Node.js.....	27
7	WEB-RAJAPINNAN TOTEUTUS.....	28
7.1	Projektin aloitus.....	28
7.2	Heroku-alustalla julkaisu ja tietokannan luominen	28
7.3	Tietokantaan yhdistäminen ja Express.js	29
7.4	RESTful API palvelimen ohjelmointi ja päätepisteet	30
8	RAJAPINNAN DOKUMENTOINTI.....	34
8.1	OpenAPI Specification	34
8.2	Swagger.....	34
8.3	API:n dokumentointi	34
9	DATAN KERÄYS.....	37
9.1	Arduino	37
9.2	MQTT.....	38
9.3	Arduino Uno:n kytkentä.....	39
9.4	Arduino-ohjelma	39
9.5	Node.js -ohjelma	41
10	KÄYTTÖLIITTYMÄN TOTEUTUS	43
10.1	Angular-projektin aloitus	43
10.2	Käyttöliittymän ohjelmointi	43
10.3	Lopputulos	47
11	KÄYTETYT KEHITYSTYÖKALUT	49
11.1	Git-versionhallinta	49
11.2	Visual Studio Code	49
11.3	Heroku	49
11.4	Postman.....	50
12	YHTEENVETO.....	51

13 LÄHDELUETTELO.....	52
LIITE 1. PALVELINOHJELMAN KOODI.....	54
LIITE 2. ARDUINO-OHJELMAN KOODI.....	58
LIITE 3. NODE.JS-OHJELMAN KOODI	60
LIITE 4. LISTA-KOMPONENTIN TS-KOODI.....	62

LYHENTEET JA MÄÄRITELMÄT

API = Application Programming Interface. Ohjelmointirajapinta.

AWS = Amazon Web Services. Amazon.comin tarjoama pilvipalvelukokonaisuus.

BSON = Binary JSON. JSON:in binäärimuotoinen esitys.

CLI = Command-line interface. Komentorivi, komentokehote.

CORS = Cross-Origin Resource Sharing. Mekanismi, joka sallii toisen domainin pyytää rajoitettuja resursseja.

CRUD = Create, Read, Update, Delete. Tiedonhallintaan liittyvät neljä perustoimintoa.

HATEOAS = Hypermedia as the Engine of Application state. Hypermedia sovelluksen tilakoneena on REST:in yhdenmukaisen rajapinnan rajoitteeseen liittyvä kohta.

HTML = Hypertext Markup Language. Hypertekstin merkinäkieli.

HTTP = Hypertext Transfer Protocol. Hypertekstin siirtoprotokolla, jota selaimet ja WWW-palvelimet käyttävät tiedonsiirtoon.

IDE = Integrated development environment. Ohjelmointiympäristö.

IoT = Internet of Things. Esineiden internet.

IP = Internet Protocol. Protokolla, jolla internet toimii. Yhdistää internettiin liittyneitä laitteita palvelimiin.

ISO = International Organization for Standardization. Kansainvälinen standardisointijärjestö.

JSON = JavaScript Object Notation. Tiedostomuoto tiedonvälitykseen.

MAC-osoite = Media Access Control. Yksilöi Ethernet-verkkosovittimen.

MQTT = Message Queuing Telemetry Transport. Tiedonsiirtoprotokolla.

MVC = Model-view-controller. Ohjelmistoarkkitehtuuri.

NoSQL = Not only SQL. NoSQL-tietokannat poikkeavat perinteisestä relaatiomallista.

REST = REpresentational State Transfer. HTTP-protokollaan perustuva arkkitehtuurimalli ohjelmointirajapinnoille.

SOAP = Simple Object Access Protocol. Viestipohjainen tietoliikenneprotokolla.

UI = User interface. Käyttöliittymä.

URI = Uniform Resource Identifier. Yhdenmukainen resurssin tunniste.

URL = Uniform Resource Locator. Yhdenmukainen resurssin paikannin.

USB = Universal Serial Bus. Sarjaväyläarkkitehtuuri oheislaitteiden liittämiseksi tietokoneelle.

XML = Extensible Markup Language. Merkinäkielten standardi, määrittää tietojen merkinämuodon.

YAML = YAML Ain't Markup Language. Datan serialisointiin käytetty kieli.

1 JOHDANTO

Tämä opinnäytetyö käsittelee ohjelmointirajapintoja ja siinä syvennyttään web-rajapinnan toteuttamiseen sensoridatalle. Web-rajapintojen tekniikoita, hyvää suunnittelua ja dokumentointia tutkitaan kokonaisvaltaisesti ennen varsinaista toteutusta. Web-rajapinta rakennetaan sensoridatalle, jota kehitetään Arduino Uno -kehitysalustalta. Työn loppupuolella toteutetaan lisäksi sivuprojektina sovellus, joka käyttää tekemääni rajapintaa.

Kiinnostuin aiheesta opiskeluaikana, sillä web-rajapintojen käyttäminen oli monesti tarpeellista ohjelmoidessa. Koska uusien sovellusten kehitys on ollut suuressa kasvussa, on jatkuva tarve myös kehittää uusia rajapintoja, jotka tarjoavat dataa tai toimintoja. Web-rajapinnat ovat tärkeä osa ohjelmointia, joten halusin perehtyä aiheeseen tarkemmin.

Opinnäytetyön tavoitteena on rakentaa toimiva web-rajapinta, jota kuka tahansa pystyy käyttämään. Tätä varten web-rajapinta täytyy ensiksi suunnitella hyvin, ja valmis rajapinta tulee dokumentoida asianmukaisesti, jotta rajapinta olisi mahdollisimman selkeä muillekin käyttäjille. Rajapinnan sensoridatan osalta tavoitteena on yksinkertaisesti saada mittaustietoa eli sensoridataa Arduino Uno -kehitysalustalta. Lisäksi tavoitteena on työn loppupuolella tehdä rajapintaa käyttävä käyttöliittymäsovellus, jolla voidaan hallita rajapinnan dataa. Sovelluksen tarkoituksena on osoittaa rajapinnan toimivuus ja ylipäättään havainnollistaa rajapintojen tarpeellisuutta. Työn pääpaino on kuitenkin web-rajapintojen tutkimuksessa ja toteutuksessa.

Työn tarkoituksena on olla opettavainen sekä itselleni että muille aiheesta kiinnostuneille. Käytän tutkimukseen verkosta löytyviä oppaita ja artikkeleita, joista koostan oppimaani tietoa yhteen. Koitan esitellä selkeästi tekemääni koodia tekstissä kuvien avulla.

2 OHJELMOINTIRAJAPINNAT YLEISESTI

Ohjelmointirajapinta tai lyhemmin rajapinta (englanniksi "Application programming interface", API) määrittelee, miten ohjelmat ja sovellukset kommunikoivat ja vaihtavat tietoa toistensa kanssa.

2.1 Rajapintojen peruseräite

Rajapinnan peruseräiteena on tarjota toimintoja tai dataa, joita voidaan käyttää hyödyksi ohjelmistokehityksessä. Rajapinnat ovat laajalti käytössä helpottamaan ohjelmiojien työtä, ettei kaikkea tarvitse aloittaa tyhjistä.

Käyttöjärjestelmät tarjoavat omia rajapintojaan, jotta sovelluskehittäjien ei tarvitse aloittaa vaikkapa mobiilisovellusta luomalla uusiksi kamera ohjelmaa, vaan voidaan käyttää kameralle luotua rajapintaa, jolla puhelimen sisäänrakennetun kameras toiminnot saadaan käyttöön. Web-sovellusten toiminnoissa, kuten säätiedon haussa tai sosiaalisen median syötteen näkemisessä, käytetään useimmiten web-rajapintoja, sillä käyttäjälle palautuu valittua dataa juuri rajapinnasta. (Hoffman 2018)

API:t mahdollistavat muun muassa sovellusten kehittämisen nopeammin sekä hyvän käyttökokeuksen luomisen asiakkaille, sillä sovellukseen voidaan toteuttaa esimerkiksi mahdollisuus kirjautua Facebook-tunnusta käyttäen. Ohjelmoijat voivat käyttää toisten julkaisemia rajapintoja hyödykseen omissa sovelluksissa ilman erillistä lupaa, mikäli rajapinnat ovat avoimia. (Hoffman 2018)

2.2 Rajapintojen julkaisukäytännöt

Rajapintojen julkaisu muiden käyttöön riippuu pitkälti siitä, mitä varten kyseinen rajapinta on luotu. Ohjelmointirajapinta voi olla täysin avoin käytettäväksi, julkaistu tietyn rajoituksin tai kokonaan yksityinen. Yleisesti ottaen netissä näkemämme API:t ovat avoimia, jos niihin pääsee käsiksi ilman salasanan kysymistä palvelun tarjoajalta. (Wodehouse 2016)

Avoimet rajapinnat ovat hyvä vaihtoehto, kun halutaan levittää ja mainostaa brändiä, saada uusia yhteistyökumppaneita tai ylipäätään kehittää bisnestä. Toisaalta tällöin rajapinnan omistajan on vaikeampaa hallita sen käyttäjien tekemien hakujen määrää, jolloin huonosti toteutettu rajapinta voi ylikuormittua. (Wodehouse 2016)

Rajapinnat voivat olla rajoitettu esimerkiksi yhteistyökumppaneille, eli rajapintojen käyttö voi vaatia käyttöoikeuksia tai lisenssiä. Tällöin on helpompi hallita asiakkaita ja rajapinnan käyttö voi olla maksullista, tai voidaan toteuttaa rajoitus, kuinka paljon hakuja voi tehdä ilmaiseksi. Yksityiset rajapinnat ovat monesti yritysten tapa toteutukseen, sillä rajapintoja tarvitaan ohjelmistokehityksessä, mutta ne halutaan yrityksessä vain omaan käyttöön. (Wodehouse 2016)

2.3 Avoimen rajapinnan ehtoja

Rajapinnan avoimuuteen on kehitelty muutamia ehtoja, sillä täsmällistä määrittelyä tarvitaan esimerkiksi julkisissa järjestelmähankinnoissa, joissa tavallisesti vaaditaan avoimia rajapintoja. Avoimen rajapinnan ehtoja ovat dokumentointi, käyttöönottettavuus ja testattavuus. Dokumentaation tulee olla verkon kautta avoimesti saatavilla, ja sen tulee olla riittävän tarkka, jotta rajapinnan käyttöön-otto ja hyödyntäminen on mahdollista. Rajapinnan käyttöönoton pitää olla mahdollista itsenäisesti, eli ilman käyttöavaimen pyytämistä ylläpitäjältä tai palveluntarjoalta. (Kivekäs 2014)

Rajapinnan testattavuuden ehto puolestaan täyttyy, jos testijärjestelmään, jossa on realistista esimerkkidataa, on avoin pääsy. Ehdon voi täyttää myös tuotantojärjestelmän avoin pääsy, jotta palveluun voi integroitua tai testijärjestelmän tulee olla ladattavissa vapaasti omaan käyttöön. Huomion arvoista on, että avoimesta rajapinnasta puhuttaessa datan ei tarvitse olla avointa dataa. Tämä tarkoittaa, että esimerkiksi potilastietojärjestelmässä voisi olla avoin rajapinta, vaikkei data eli potilastiedot olisivatkaan avoimia. (Kivekäs 2014)

2.4 Rajapinnat käyttötarkoitusten mukaan

Rajapintoja luokitellaan usein niiden käyttötarkoitusten, eli niiden järjestelmien mukaan, joille ne on suunniteltu. Database API:t ovat nimensä mukaisesti tietokannan rajapintoja, jotka mahdollistavat tiedonsiirron sovellusten ja tietokannan hallintajärjestelmien välillä. Esimerkiksi Drupal 7 Database API antaa käyttäjille mahdollisuuden kirjoittaa yhtenäisiä kyselyitä eri tietokannoille, niin omille kuin avoimille lähteille. (AltexSoft 2019)

Käyttöjärjestelmän API:t (eng. operating systems APIs) määrittelevät kuinka sovellukset käyttävät käyttöjärjestelmien resursseja ja palveluita. Jokaisella käyttöjärjestelmällä on joukko rajapintoja, joita järjestelmän laitteistot ja sovellukset käyttävät. Lisäksi useimmat käyttöjärjestelmät tarjoavat rajapintoja, joiden avulla ohjelmoijat voivat kirjoittaa käyttöympäristön mukaisia sovelluksia. (Beal ei pvm)

Web API:t perustuvat internet- ja web-protokollien avulla viestintään. Web API:t toimittavat pääasiassa verkkosovellusten pyyntöjä ja vastauksia palvelimilta, jotka käyttävät http-protokollaa (Hypertext Transfer Protocol). Web API:t tarjoavat dataa ja toimintojen siirtoa web-pohjaisten järjestelmien välillä, jotka edustavat client-server eli asiakas-palvelin-arkkitehtuuria. Käytännössä Web API:t tekevät internetissä olevista resursseista käytettäviä paikallisesti. Web API:t tulivatkin suosituiksi juuri internet-palvelujen yleistymisen myötä, kun pystyttiin tallentamaan sisältöä ja tietoa verkossa. (Pedro 2017)

3 WEB-RAJAPINTOJEN TEKNIIKAT JA TOIMINTA

Tässä luvussa käydään läpi kahta eri web-rajapinnan toteutustapaa. REST ja SOAP ovat tällä hetkellä tekniikat, joilla todella suuri osa web-rajapinnoista on tehty. Lisäksi käydään läpi olennaisia tekniikoita, jotka auttavat ymmärtämään web-rajapintojen toimintaa.

3.1 HTTP

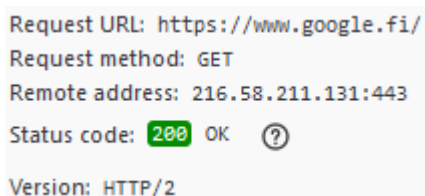
HTTP (HyperText Transfer Protocol) on selainten ja WWW-palvelimien käyttämä protokolla tiedon siirtoon. HTTP on web-rajapintojen ja web-ohjelmoinnin kannalta erittäin oleellinen protokolla ymmärtää, sillä mm. REST API:t rakentuvat käytännössä http-protokollan ominaisuuksien päälle.

HTTP noudattaa asiakas-palvelin-mallia (engl. Client-Server Model). Yksinkertaistettuna asiakas, kuten selaimen käyttäjä, ottaa yhteyden palvelimeen HTTP-pyyntöllä. Palvelin eli tietokone tai ohjelmisto, joka on määritetty palvelimeksi mm. kiinteän verkkoyhteyden ja pysyvän IP-osoitteen avulla, vastaa tähän HTTP-pyyntöön esimerkiksi näyttämällä nettisivun (HTML-dokumentti), tekstitiedostolla tai virheellä. (Tampere University of Technology ei pvm)

HTTP on tilaton, eli palvelin ei muista aiempia pyyntöjä, vaan joka HTTP-pyyntöissä on oltava kaikkien käsittelyyn tarvittava tieto. Tilattomuus yksinkertaistaa palvelimen puolen toteutusta, mutta toisaalta esimerkiksi verkkokaupan ostoskorin sisällön säilömistä varten, joudutaan käyttämään muita ratkaisuja kuten evästeitä. (Tampere University of Technology ei pvm)

3.1.1 HTTP-pyyntö

HTTP-pyyntön alkuosa, ns. pyyntöriivi (engl. start line), rakentuu HTTP-metodista eli verbistä (kuten GET, PUT, POST), pyyntön kohteesta eli yleensä URL-osoitteesta, sekä HTTP:n versiosta, joka määrittelee jäljellä olevan viestin rakenteen ja osoittaa vastauksessa odotettavissa olevaa versiota. (Tampere University of Technology ei pvm)



```
Request URL: https://www.google.fi/  
Request method: GET  
Remote address: 216.58.211.131:443  
Status code: 200 OK  
Version: HTTP/2
```

Kuva 1 HTTP-pyyntön alkuosa

Lisäksi HTTP-pyyntöön kuuluu aina otsikkotiedot (headers), joka kertoo muun muassa kohdepalvelimen osoitteen (Host), tietoja asiakkaasta eli web-selaimesta (User-Agent), pyyntön esitysmuoto (Accept) ja kieli (Accept-Language), jolla palvelimen halutaan vastattavan, sekä paljon muita kohtia.

```
Host: www.google.fi
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:73.0) Gecko/20100101 Firefox/73.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
Accept-Language: fi-FI,fi;q=0.8,en-US;q=0.5,en;q=0.3
Accept-Encoding: gzip, deflate, br
Connection: keep-alive
Cookie: SID=ugc7MPGOUarGDwtb-fqkH95SJqhGX0EcXUC8mFg511zeNMrf7HHZZIdaGjgR4zUXCPr5iW.; HSID=AGqIVJzGMdxCB2jFa; SSID=AAUG7DE-Pp
Upgrade-Insecure-Requests: 1
TE: Trailers
```

Kuva 2 HTTP:n otsikkotiedot

HTTP-pyyntöön viimeinen osa on kappale (body), jota kaikilla pyynnöillä ei ole. Resursseja hakeva pyyntö, kuten GET, HEAD tai DELETE ei tällaista tarvitse, kun taas POST-pyyntö usein lähettää esimerkiksi lomaketietoja tietoa palvelimelle. (Tampere University of Technology ei pvm)

3.1.2 HTTP-vastaus

HTTP-vastaus rakentuu ensimmäisenä olevasta status-rivistä (status line), otsikkotiedoista (headers) ja mahdollisesta kappale (body) osiosta. Status-rivi sisältää HTTP:n version, statuskoodin ja selityksen statuskoodille. Muu viestin rakenne on avain-arvo-pareina kuten pyynnöissäkin. (Tampere University of Technology ei pvm)

```
HTTP/2 200 OK
date: Fri, 06 Mar 2020 10:11:20 GMT
expires: -1
cache-control: private, max-age=0
content-type: text/html; charset=UTF-8
strict-transport-security: max-age=31536000
content-encoding: br
server: gws
content-length: 56275
x-xss-protection: 0
x-frame-options: SAMEORIGIN
set-cookie: 1P_JAR=2020-03-06-10; expires=Sun, 05-Apr-2020 10:11:20 GMT; path=/; domain=.google.fi; Secure; SameSite=none
alt-svc: quic=":443"; ma=2592000; v="46,43",h3-Q050=":443"; ma=2592000,h3-Q049=":443"; ma=2592000,h3-Q048=":443"; ma=2592000
X-Firefox-Spdy: h2
```

Kuva 3 HTTP-vastaus

3.1.3 HTTP:n statuskoodit

HTTP-vastauksen sisältämä statuskoodi ilmoittaa pyynnön onnistumisesta. HTTP:n statuskoodeja on suuri määrä, mutta tärkeimmät niistä näkyvät kuvassa 4. Etenkin statuskoodi 404 on monelle tuttu, kun etsittyä sivua ei löydy tietyistä osoitteesta. Näitä samaisia statuskoodeja voidaan hyödyntää aina HTTP:n kanssa tekemisissä ollessa, kuten web-rajapinnoissa. (Tampere University of Technology ei pvm)

Statuskoodi	Selitys	Esimerkkejä
1xx	Pyyntö kesken	100 Continue
2xx	Pyyntö onnistui	200 OK 201 Created
3xx	Uudelleenohjaus	302 Found
4xx	Asiakkaan virhe	400 Bad Request 403 Not Allowed 404 Not Found
5xx	Palvelimen virhe	500 Internal Server Error

Kuva 4 Yleisimmät HTTP-statuskoodit (Tampere University of Technology ei pvm)

3.2 SOAP

Ennen REST-rajapintojen yleistymistä, web-rajapinnat olivat tyypillisesti SOAP-pohjaisia. SOAP (Simple Object Access Protocol) on viestiprotokolla, jonka avulla sovelluksen eri elementit voivat kommunikoida keskenään. SOAP toimii useiden alemman tasojen protokollien yli, mukaanlukien http-protokollan yli. (SmartBear 2020)

SOAP voi käyttää ainoastaan XML-formaattiin perustuvia viestejä, joten kaikki data SOAP-pohjaisen rajapinnan ja ohjelman välillä vaihtuu XML-muodossa. XML-formaatin viestit ovat melko hyvin ihmisen luettavissa, mutta viestit sisältävät suhteellisen paljon tietoa, joten tiedonsiirron kannalta viestit ovat hitaita käsitellä. (Rouse 2019)

SOAP on suunniteltu tukemaan laajennuksia, joten luotettavuuteen ja tietoturvaan SOAP:in sanotaan pystyvän REST-rajapintoja paremmin. Tästä huolimatta REST-rajapinnat ovat korvanneet melkein kokonaan SOAP:in web-rajapinnoissa. Erityisesti viestin tiedonsiirron hitauden, sekä REST:in tarjoamien dataformaattien takia SOAP-pohjaiset rajapinnat nähdään nykyisin hitaampina, vaikeammin lähestyttävänä ja käytettävänä kuin REST-rajapinnat. (SmartBear 2020)

3.3 JSON

JavaScript Object Notation, tästedes JSON, on formaatti datan tallettamiseen ja vaihtoon. JSON on erittäin yleinen datan muoto, jota käytetään usein muun muassa, kun dataa halutaan palvelimelta esitettäväksi web-sivulle. JSON-muotoinen data on avain-arvo-pareina (name/value pair). Syntaksiin kuuluu, että data on eroteltu pilkuilla. Aaltosulkeilla erotellaan objektit, kun taas hakasulkuja käytetään taulukkoihin. Kuvassa 5 on "nimet" taulukko, joka sisältää kaksi objektia. (w3schools ei pvm)

```
"nimet": [
  { "etunimi": "Juha", "sukunimi": "Korhonen" },
  { "etunimi": "Annina", "sukunimi": "Kettunen" }
]
```

Kuva 5 JSON esimerkki

Nimestään huolimatta JSON:ia voidaan käyttää muillakin ohjelmointikielillä kuin JavaScriptillä. Monet modernit ohjelmointikielet tukevat datan muuntoa JSON-muotoon ja pois siitä, tai vähintäänkin löytyy kirjastoja, joilla JSON:ia on helppo käsitellä.

3.4 REST

REST on yleisin arkkitehtuurimalli web-rajapintojen toteuttamiseen. REST:in suosio perustuu sen yksinkertaisuuteen; ominaisuudet nähdään käyttäjäystävällisenä, ja ohjelmoijien on helppo ymmärtää sen koodaamista. REST eli REpresentational State Transfer nimi on hyvin kuvaava, sillä rajapinta kutsuessa palvelin siirtää (transfer) asiakkaalle esityksen (representation) pyydetyn resurssin tilasta (state). REST määrittelee mm. operaatiot, jolla palvelinten dataa pyydetään, lisätään ja käsitellään. REST ei määrittele standardia tiedon formaattia, joten se on paljon SOAP:ia joustavampi eri ohjelmointiympäristöissä, kun voidaan käyttää esimerkiksi JSON:ia. (Avraham 2017)

3.4.1 Resurssi ja sen metodit

Resurssilla tarkoitetaan mitä tahansa kohdetta, josta API antaa tietoja. Esimerkiksi Instagram API:n resurssi voi olla käyttäjä, valokuva tai hashtag. Kokonaisuudessaan REST-rajapinta toimisi siis näin: Kun kehittäjä kutsuu Instagram API:a hakemaan tiettyä käyttäjää eli resurssia, API palauttaa kyseisen käyttäjän tilan, joka sisältää hänen nimensä, kuinka monta seuraajaa käyttäjällä on ja käyttäjän lähettämien postausten määrän. (Avraham 2017)

Resurssit identifioidaan toisistaan URI:n (Uniform Resource Identifier) avulla. URI sisältää resurssin nimen ja osoitteen, mistä resurssi haetaan. Jokaisella osoitettavissa olevalla tietoyksiköllä tulee siis olla osoite, joka muodostuu usein suoraan linkin ja id-määreen avulla tai epäsuorasti esimerkiksi mediatyyppin ja esitysrakenteen perusteella. Uniikkia id:tä käyttäminen on yksinkertainen keino, jolla tietoa erotellaan toisistaan. Web-rajapinnan kaikki data voisi esimerkiksi sijaita osoitteessa "https://esimerkki.fi/api/data/". Jos resurssi olisi määritelty id:n avulla, "https://esimerkki.fi/api/data/1" ja "https://esimerkki.fi/api/data/2" voisivat palauttaa kuvan 6 ja 7 mukaisesti JSON-muodossa dataa. REST ei aseta rajoitusta resurssien esittämismuotoon, joten periaatteessa mitä tahansa muotoa voitaisiin käyttää, mutta kaksi yleisintä muotoa resurssien esittämiseksi ovat JSON ja XML. (restfulapi ei pvm)

```
{
  "id" : 1,
  "pvm" : "1.12.2006",
  "arvo" : 600
}
```

Kuva 6 Esimerkki data 1

```
{
  "id" : 2,
  "pvm" : "6.3.2020",
  "arvo" : 1020
}
```

Kuva 7 Esimerkki data 2

REST:in resursseja operoidaan http-metodien avulla. Käytetyimmät metodit ovat GET, POST, PUT, DELETE ja PATCH. Näitä metodeja kutsutaan myös CRUD (Create, Read, Update, Delete) operaatioiksi, jotka ovat neljä tarvittua toimintoa tiedon varastoinnissa. Taulukko 1 kertoo mitä kullakin metodilla tehdään REST:issä, ja millainen pyyntö URI-osoitteeseen voitaisiin esimerkiksi tehdä.

TAULUKKO 1. Http-metodien tehtävät REST:issä

Http-metodi	CRUD	Metodin käyttö REST:issä	Esimerkki pyynnöstä
POST	Create	Luo uuden resurssin kokoelmaan.	HTTP POST https://esimerkki.fi/api/data
GET	Read	Käytetään resurssien hakemiseen ja esitykseen.	HTTP GET https://esimerkki.fi/api/data
PUT	Update/ Replace	Päivittää olemassa olevan resurssin kokonaan. Mikäli resurssia ei ole olemassa, voidaan käyttää myös luomaan uuden resurssin.	HTTP PUT https://esimerkki.fi/api/data/1
PATCH	Partial Update/Modify	Päivittää resurssin osittain.	HTTP PATCH https://esimerkki.fi/api/data/1
DELETE	Delete	Poistaa resurssin.	HTTP DELETE https://esimerkki.fi/api/data/1

3.4.2 REST:in rajoitteet

Roy Fielding:in vuonna 2000 esittelemään ja määrittelemään REST arkkitehtuurimalliin kuuluu kuusi rajoitetta, jotta web-rajapintaa voitaisiin kutsua aidosti REST:iksi. Rajoitteita ovat asiakas-palvelin, tilattomuus, välimuisti, yhdenmukainen rajapinta, kerroksittainen järjestelmä ja ladattava koodi (vapaaehtoinen rajoite). (Fielding 2000)

- 1) **Asiakas-palvelin** rajoitteella tarkoitetaan asiakas-palvelin-mallia, jossa palvelin ja asiakas ovat itsenäisiä. Aivan kuten HTTP:ssäkin, vuorovaikutus niiden välillä toimii asiakkaan aloittamien pyyntöjen mukaan, jolloin palvelin lähettää vastauksen pyyntöön. Palvelin vain odottaa tulevia pyyntöjä, eikä ala lähettämään tietoja joidenkin resurssien tilasta ilman pyyntöjä. Rajoitteen taustalla on ns. huolenaiheiden erottaminen toisistaan, jotta siirrettävyys ja skaalautuvuus alustojen välillä paranee, kun "palvelinkomponentit" pysyy yksinkertaisena. Tällöin sekä asiakasohjelmisto, että palvelinohjelmisto, saavat kehittyä itsenäisesti ja järjestelmä pysyy paremmin ylläpidettävänä. (Fielding 2000) (GeeksforGeeks ei pvm)
- 2) **Tilattomuus** tarkoittaa, että palvelin ei muista mitään API:a käyttävästä käyttäjästä eli palvelin ei muista, jos käyttäjä on jo lähettänyt pyynnön aiemmin. Jokainen yksittäinen pyyntö sisältää kaikki tiedot, jotka palvelin tarvitsee pyynnön suorittamiseksi ja vastauksen palauttamiseksi, riippumatta muista käyttäjän tekemistä pyynnöistä. Sessiointi pidetään siis kokonaan asiakkaalla. Tämä rajoitus perustuu mm. skaalautuvuuteen, sillä palvelimen ei tarvitse tallentaa tilaa pyyntöjen välillä. (Fielding 2000) (GeeksforGeeks ei pvm)

- 3) **Kerrostettu järjestelmä.** Resurssien tilan esittämistä pyytävän asiakkaan ja vastauksen takaisin lähettävän palvelimen välillä voi olla useita palvelimia. Nämä palvelimet voivat tarjota suojakerroksen, välimuistikerroksen, kuormituksen tasapainotuskerroksen tai jonkin muun toiminnallisuuden. Eri kerroksia voidaan esimerkiksi käyttää ns. kapseloimaan vanhoja palveluita (legacy services) ja suojaamaan uusia palveluita vanhoja järjestelmiä käyttäviltä asiakkailta siirtämällä harvoin käytettyjä toimintoja jaettuun välittäjään. Välittäjiä voidaan käyttää myös palvelujen kuormituksen tasapainotukseen useiden verkkojen ja prosessorien välillä. (Fielding 2000) (MuleSoft ei pvm)
- 4) **Välimuisti** rajoitteella tarkoitetaan, että palvelimen lähettämät tiedot sisältävät kohdan, onko tietoja välimuistilla vai ei. Rajoite ei siis käytännössä ole pakollinen, jos välimuistia ei päätetä käyttää. Etuna välimuisti parantaa tehokkuutta ja käyttäjän havaitsemaa suorituskykyä, sillä viive vähenee, kun tiedot haetaan välimuistista palvelimen sijaan. Välimuistin tekeminen on mahdollista versionumeron avulla, jonka perusteella asiakkaalla olevan dataversioiden avulla asiakas voi välttää pyytämästä samaa tietoa uudelleen, jos tiedot eivät ole päivittyneet. Välimuistille siis annetaan oikeus käyttää vastauksen tietoja uudelleen, myöhempää vastaavaa pyyntöä varten. Välimuistia ei kuitenkaan aina ole järkeä käyttää, jos esimerkiksi resurssit muuttuvat nopeasti. Huonosti toteutettuna välimuisti heikentää luotettavuutta, sillä välimuistin sisällä olevat tiedot ovat voineet vanhentua ja erota huomattavasti tiedoista, jotka olisi saatu suoraan palvelimelta. (Fielding 2000) (restfulapi ei pvm)
- 5) **Yhdenmukainen rajapinta** tarkoittaa, että eri asiakkaiden pyynnöt näyttävät samoilta, riippumatta onko asiakas selain, Android sovellus, Python-skripti tai jokin muu. Yhdenmukainen rajapinta sisältää neljä rajoitetta, jotka liittyvät resurssin tunnistamiseen ja muokkaamiseen (URI ja HTTP-metodit), pyyntöjen sisältämiin tietoihin ja HATEOAS:iin. HATEOAS (Hypermedia as the Engine of Application State) eli hypermedian käyttö sovelluksen tilakoneena on uniikki REST'in ominaisuus, joka erottaa sen muista web-rajapinnoista. Hypermedialla tarkoitetaan mitä tahansa dataa, joka sisältää linkkejä muihin median muotoihin, kuten kuviin, videoihin tai tekstiin. HATEOAS rajoite edellyttää siis, että asiakkaalle välitettyjen vastauksien tulee sisältää linkkejä. Esimerkiksi normaalisti rajapinnasta saataisiin pankin saldon tiedot, muttei mitään toimenpiteitä siihen liittyen. HATEOAS:n kanssa saataisiin myös linkit, joilla voitaisiin tallentaa, nostaa tai siirtää rahaa. (Fielding 2000) (Karanam 2019) (Avraham, Medium 2017)
- 6) **Ladattava koodi** on edellisistä poiketen valinnainen rajoitus. Palvelimen vastaus voi sisältää koodia, jolla voidaan laajentaa asiakassovelluksen toiminnallisuutta suorittamalla koodia asiakaspäässä. Näin voidaan tehdä esimerkiksi harvoin käytetyille toiminnallisuuksille, jotka otetaan käyttöön vasta sitten kun tarvitaan. (Fielding 2000) (restfulapi ei pvm)

REST termi on alkuperäisestä määritelmästäan hieman elänyt, sillä näitä rajoitteita ei noudateta aina kovinkaan tarkasti, vaikka puhuttaisiin REST tai RESTful rajapinnasta. Rajoitteiden tunteminen on kuitenkin hyvä pohja web-rajapinnan toteutukseen.

3.4.3 Päätepisteet

Ennestään tuntemattoman rajapinnan käyttö on parhaimmillaan helppoa ohjelmoijille, kun resurssit ja päätepisteet nimetään hyvin. Mikäli ne taas nimetään huonosti, rajapinta voi tuntua vaikealta ymmärtää. Siispä on tärkeää noudattaa nimeämistapoja, joita yleisesti noudatetaan. Vaikka päätepisteiden nimeämistä ei ole REST:ille määritelty tai standardoitu, oppaita ja materiaaleja tutkimalla huomaa, että tapa nimetä päätepisteitä on hyvin vakiintunut. Ennen kaikkea siinä pyritään yksinkertaisuuteen ja loogisuuteen.

REST API:n ns. päätepisteillä (engl. end points) tarkoitetaan polkua, josta resurssit löytyvät. Päätepisteet tulee siis suunnitella niin, että niissä otetaan huomioon resurssien ns. alaresurssit. REST:issä suositeltava nimeämistapa kokoelmille on päätepisteen nimi monikossa, esimerkiksi `"/asiakkaat"`. Yksittäinen asiakas sen sijaan saadaan päätepisteestä `"/asiakkaat/{asiakasId}"`. Resurssilla voi olla myös alaresursseja, kuten `"/asiakkaat/{asiakasId}/laskut"`. Tällöin taas yksittäinen lasku voitaisiin tehdä saataville osoitteesta `"/asiakkaat/{asiakasId}/laskut/{laskuId}"`. (Kapadnis 2018)

Resurssin URI-osoitteessa tulee käyttää substantiiveja, kuten `"/asiakkaat"`, eikä esimerkiksi `"/luo-Asiakas"`. Kuten aikaisemmin mainittua, REST:issä http-metodeilla määritellään resurssille suoritettava toiminto. Taulukossa 2 on yhteenveto http-metodien käytöstä yksittäiseen resurssiin sekä kokoelmaan. (Kapadnis 2018)

TAULUKKO 2. Http-metodien käyttö resurssille

Resurssin polku	GET	POST	PUT	DELETE
<code>/asiakkaat</code>	Palauttaa listan asiakkaista	Luo uuden asiakkaan	Muokkaa kaikkia asiakkaita	Poistaa kaikki asiakkaat
<code>/asiakkaat/123</code>	Palauttaa tietyn asiakkaan	Metodi ei ole sallittu (405)	Tietyn asiakkaan muokkaus	Poistaa tietyn asiakkaan

4 HYVÄN WEB-RAJAPINNAN PERIAATTEET

Web-rajapintaa kehittäessä on syytä muistaa, että rajapintaa tehdään useimmiten muille ohjelmajille. He ovat todennäköisesti yhtä kriittisiä rajapinnan suhteen, kuin olisit itse tutustuessasi uuteen rajapintaan. Rajapinnan suunnittelussa tulisikin alusta asti miettiä toimintoja käyttäjän näkökulmasta, sitä millä eri tavoilla rajapintaa voidaan haluta käyttää, ja kuinka se tehdään mahdollisimman helpoksi käyttäjälle. Rajapintaa voidaan ajatella tuotteena, jonka pienetkin puutteet voidaan huomata. (Ambra 2014)

4.1 Kunnollinen dokumentaatio

Rajapinnan dokumentaatio on välttämätöntä, sillä muuten ainoastaan sen suunnitelleet henkilöt tietävät miten rajapintaa käytetään. Dokumentaatio on usein myös ensimmäinen asia, jonka käyttäjät näkevät, joten on äärimmäisen tärkeää tehdä dokumentaatio hyvin.

Dokumentoinnin lähtökohtana on kertoa mihin rajapintaa käytetään, kertoa rajapinnan pyynnöistä ja vastauksista, sekä saatavasta dataformaatista. Näiden tietojen dokumentointiin on olemassa ohjelmistotyökaluja, kuten OPENAPI/Swagger, jotka helpottavat ja yksinkertaistavat dokumentoinnin tuottamista. Lisäksi hyvin dokumentoiduissa rajapinnoissa on käyttöesimerkkejä ja mieluiten jonkinlainen tutoriaali. Nämä auttavat käyttäjää ymmärtämään rajapintaa ja mistä aloittaa. Tutoriaalien kannattaa olla tiiviitä, jotka auttavat käyttäjää nopeasti alkuun. Sen jälkeen voidaan linkillä ohjata yksityiskohtaisempaan dokumentaatioon vaikkapa toiminnallisuuksista, joten käyttäjä ymmärtää miten alun jälkeen pystytään laajentamaan rajapinnan käyttöä. (Ambra 2014)

Dokumentaation valmistuttua, on hyvä varmistaa, että se on ymmärrettävissä muillekin kuin sen tekijälle. Hyvin dokumentoidun rajapinnan avulla rajapinnan käytön pitäisi olla niinkin helppoa, että helpon rajapintaa käyttävän perusohjelman tekoon menisi vain noin 15 minuuttia. (Ambra 2014)

4.2 Versiointi

Rajapinnan versiointi on todella tärkeää, jos rajapintaa halutaan kehitettävä ajan myötä. Pelkkä toimintojen lisääminen rajapintaan onnistuu ilman versiointiakin, mutta jo julkaistun API:n olemassa olevia toimintoja ei tulisi koskaan muuttaa, varsinkaan jos se on käytössä asiakkaila. Koska API on kuin tuote, asiakkaat odottavat siltä oikeita vastauksia ja muutokset saattavat rikkoa asiakkaan ohjelmiston. Tämän vuoksi API pyritään julkaisemaan alun perinkin valmiina, mutta jos muutoksia tai lisäyksiä tehdään, on syytä kirjata ja julkaista jonkinlainen lista muutoksista. (Ambra 2014)

Versioinnin ansiosta rajapinnan kehittäjät voivat siis julkaista uuden version rikkomatta ohjelmistoja, jotka on tehty vanhan version toiminnallisuuksilla. Versioinnilla voidaan antaa myös asiakkaalle aikaa siirtyä uuteen versioon. Uudesta versiosta on hyvä julkaista jonkinlainen lista muutoksista (engl. changelog), jotta asiakkaat tietävät mitä eroa versioiden välillä on.

Versiointitiedot voidaan toteuttaa rajapinnassa erilaisilla tavoilla. Usein versiotieto päätetään lisätä URL-osoitteeseen suoraan, sillä se on käytännöllistä. Esimerkiksi URL-osoitteessa "https://esimerkki.fi/api/v1/data/" kohta v1 voisi tarkoittaa ensimmäistä versiota, ja rajapinnasta toinen versio olisi osoitteessa "https://esimerkki.fi/api/v2/data/". Versioinnissa kannattaa käyttää yksinkertaisia numeroita, sillä esimerkiksi "v1.2":n piste ei näkyisi URL-osoitteessa. (Ambra 2014)

4.3 Suojaus

Rajapinnan käyttötarkoituksesta ja julkisuudesta riippuen, rajapinnan käyttö voidaan haluta sallia ainoastaan autentikoituneille käyttäjille. Usein ei myöskään haluta sallia kaikkia toimintoja jokaiselle käyttäjälle, sillä resurssien lisäämisen, muokkaamisen ja poistamisen salliminen antaisi kaikille luvan muokata rajapinnan dataa vapaasti.

Suojaus voidaan toteuttaa esimerkiksi yksinkertaisella token-pohjaisella todennuksella, jossa käyttäjätunnukseksi on vaikkapa rekisteröinnin yhteydessä luotu satunnainen tunnistus. Koska REST on tilaton, tunnistus tulee jokaisen käyttäjän tekemän pyynnön mukana. Tunnistusta voidaan verrata tietokannasta löytyviin tunnistuksiin ja saada tieto käyttäjän oikeuksista. Tunnistautumisen toteuttamiseen on olemassa muitakin vaihtoehtoja, kuten OAuth 2 -protokolla, joka myös käyttää tunnistusta todennukseen. OAuth-kirjastoja on saatavilla monille yleisille ohjelmointikielille, ja sen sanotaan olevan kohtuullisen helppo tapa toteuttaa palvelimen puolella. Lisäksi käyttäjätunnukset, salasanat, tunnistukset ja API-avaimet eivät saa näkyä URL-osoitteessa, sillä ne voidaan tallentaa palvelinlokeihin, jolloin ne voidaan kaapata helposti. (Ambra 2014)

4.4 Hakutulosten sivutus ja suodattaminen

Jos rajapinta sisältää tai tulee sisältämään valtavan määrän dataa, siihen on suositeltavaa toteuttaa palautettavien tuloksien rajaaminen jollain keinolla. Suuren datamäärän lataaminen joka kerta voi näkyä kuluttajan sovelluksen hitaudessa, tai sivusto voi toimia ja näyttää toisenlaiselta, jos siihen ei ole varauduttu. Yksinkertainen tapa tähän on sivutus (engl. pagination), eli suurempi datamäärä jaetaan pienemmiksi palasiksi ja jokainen sivu sisältää maksimissaan tietyn määrän tuloksia. Sivutus voidaan toteuttaa URI-osoitteen avulla, esimerkiksi "https://esimerkki.fi/api/v1/data?page=12" voisi palauttaa 25 datatietuetta sivulta 12. (Haldar 2016)

Suodattaminen puolestaan tarkoittaa usein erilaisten kyselyparametrien kautta hakutulosten rajaamista. Tämänkin toteutukseen yksinkertainen tapa on URI-osoite, joten esimerkiksi "https://esimerkki.fi/api/v1/data?sijainti=Kuopio" palauttaisi kaiken datan, jossa sijainti on Kuopio.

4.5 Seuranta

Yritysmailmassa rajapinnan kehittämisen ja testaamisen jälkeen on jatkettava tuen tarjoamista, kun rajapinta otetaan käyttöön ja sitä halutaan käyttää tuotantoon. Jos jokin menee pieleen, rajapintaa kehittäneille ihmisille on syytä raportoida ongelmista niin tarkasti kun mahdollista, jotta oikeat

ihmiset tietävät tällöin vastata tarvittavilla keinoilla. Kehitysvaiheessa kannattaakin ennakoida ja testata rajapintaa tarpeeksi. Lisäksi API-valvontatyökaluilla, kuten Runscope:lla voidaan ajaa läpi ja monitoroida rajapinnassa esiintyviä ongelmia. Tällöin voidaan tunnistaa, ratkaista ja estää ongelmat nopeasti, ennen kuin esimerkiksi API kaatuu kokonaan ja tulee pitempiä katkoksia, joista on haittaa asiakkaille. (Despoudis 2018)

4.6 Esimerkki web-rajapinnasta

Tarkastellaan esimerkkinä Traffic Management Finland:in omistamaa rautatieliikennetietoa tarjoavaa avointa rajapintaa. Rajapinta jakaa tietoa koko Suomen rataverkolla kulkevien junien aikatauluista, sijainneista sekä muista tarkoista tiedoista. Kyseiseen rajapintaan koostetaan dataa eri lähteistä. Käyttäjät voivat ainoastaan hakea dataa tämän rajapinnan kautta. Muita operaatioita kuten poistamista ja muokkaamista ei ole käyttäjille toteutettu, ymmärrettävistä syistä. (Traffic Management Finland ei pvm)

Rajapinta on hyvin esitelty ja dokumentoitu verkkosivustolla. Sivustolla on ohjeita ja esimerkkejä eri tiedon saannista rajapinnan kautta. Lisäksi on saatavilla Swagger-dokumentaatio, jossa näkyy hienon lyhyesti kaikki haut mitä voidaan tehdä ja mitä dataa mikäkin haku palauttaa. Rajapinnassa on tällä hetkellä REST- ja WebSocket-rajapinnat, joiden vastaukset ovat JSON-formaattia. (Traffic Management Finland ei pvm)

- [Junien tiedot \(/trains\)](#)
- [Aktiivisten junien seuranta \(/live-trains\)](#)
- [Junan GPS-sijainnit \(/train-locations\)](#)
- [Kulktietoviestit \(/train-tracking\)](#)
- [Kokoonpanotiedot \(/compositions\)](#)
- [Metatiedot \(/metadata\)](#)

Kuva 8 Rautatieliikenne API:n päätepisteet (Traffic Management Finland ei pvm)

Rajapinta on datamäärältään suuri, joten se on jaettu kuuteen ns. päätepisteeseen kuvan 8 mukaisesti. Nämä päätepisteet helpottavat käyttäjää käyttämään rajapintaa oikein, eli käytännössä suuri datamäärä on jaettu pienempien otsikoiden alle. Esimerkiksi `/trains` palauttaa junien tietoja kuten junanumeron, lähtöpäivämäärän ja aseman. Metatiedot puolestaan sisältävät muun muassa kaikkien asemien paikkakunnat ja niitä vastaavat lyhenteet. (Traffic Management Finland ei pvm)

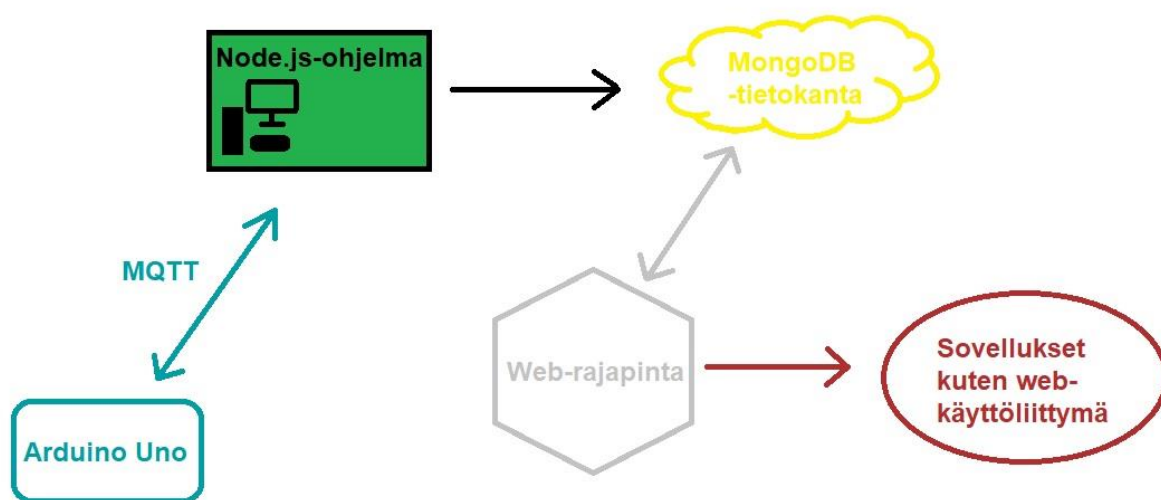
Rajapinnan versiointi on otettu huomioon suoraan osoitteessa (<https://rata.digitraffic.fi/api/v1/trains/latest/1>). Lisäksi rajapinnalle on kerrottu muun muassa sen käyttöehdot, tietolähteen lähteen nimeäminen, toteutetut ominaisuudet, suunnitellut ominaisuudet, sekä palvelulle on olemassa julkinen Google-ryhmä kehittäjäyhteisöä varten. (Traffic Management Finland ei pvm)

5 TYÖN SUUNNITTELU

Web-rajapinnan toteutukseen valitsin REST-arkkitehtuurimallin, sen yksinkertaisuudesta ja joustavuudesta johtuen. Tässä luvussa käsitellään työn eri osat ja suunnitellaan tarkemmin rajapinnan toimintaa. Lisäksi suunnitellaan yksityiskohtaisesti työhön kuuluvien eri ohjelmistojen osat. Työssä käytettyjä web-tekniikoita puolestaan käsitellään tarkemmin luvussa 6.

5.1 Kokonaisuus

Opinnäytetyössä toteutetaan web-rajapinta sensoridatalle. Projektiin kuuluu muitakin osia, joten kokonaisuudessaan systeemi on esitelty kuvassa 9. Sensoridata kerätään Arduino Uno -laitteella ja MQTT-protokollaa käyttäen lähetetään dataa paikallisella tietokoneella ajettavalle Node.js-ohjelmalle. Node.js-ohjelma lisää sensoridatan pilvipalveluna olevaan tietokantaan. Web-rajapintaan rakennetaan datan esittämiseen ja käsittelyyn liittyviä toimintoja. Valmis rajapinta dokumentoidaan, sekä sitä voidaan käyttää erilaisiin sovelluksiin eri alustoilta. Työn loppupuolella tehdään esimerkkisovelluksena web-käyttöliittymä rajapinnalle.



Kuva 9 Projektin eri osat

5.2 Rajapinnan avoimuus

Päätökseni on tehdä avoin rajapinta, sillä rajapinta ja sen sisältämä sensoridata voidaan hyvin pitää julkisena. Rajapinnan kaikkien toimintojen käyttäminen sallitaan vapaasti, joten muutkin ohjelmoijat pystyisivät halutessaan rakentamaan omia ohjelmia ja sovelluksia siihen. Rajapinta toteutetaan REST-arkkitehtuurimallilla, mutta aivan kaikkia REST:in rajoitteita ei toteuteta. En näe käytännöllisenä tässä tapauksessa esimerkiksi HATEOAS:n toteutusta, joten rajapinnasta tulee periaatteessa REST-pohjainen.

5.3 Rajapinnan data

Rajapinnan datana tulee olemaan Arduino-laitteelta saatua sensoridataa sekä myöhemmin rakennettavan käyttöliittymän kautta lisättyä dataa. Rajapinnan datan haluan palautuvan JSON-muotoisena, joten datalle määritellään avain-arvo-pareja. Määrittelemäni sensoridatalle sopivat avain-arvo-parit näkyvät kuvassa 10. Sensoridatan arvo lisätään yksinkertaisesti arvo-avaimeen, ja datan yksikkö taas annetaan yksikko-avaimelle. Yksikko-avaimelle annetaan mittauksen yksikkö, ja sensori-avaimelle annetaan sensorin nimi.

Avaimella "mitattu" tulee olemaan ajankohta, jolloin sensoridata on mitattu. Muokattu-avaimeen puolestaan lisätään päivämäärä, jos dataa muokataan käyttöliittymän kautta. Alinta kohtaa eli lisätty-avainta käytetään niin, että sen arvo on "Arduino Uno", jos sensoridata on saatu Arduino Uno:lta. Jos lisätty-avaimen arvona lukee "UI", se tarkoittaa käyttöliittymän (engl. user interface) kautta lisättyä dataa.

```
{
  "id": "123",
  "arvo": "20",
  "yksikko": "C",
  "sensori": "LM35",
  "mitattu": "2020-03-11T09:57:16.282Z",
  "muokattu": "2020-05-22T09:21:59.282Z",
  "lisetty": "Arduino Uno"
}
```

Kuva 10 Datan rakenne

Muutamia huomioita datan rakenteesta on, että ID:n täytyy olla aina uniikki jokaiselle datalle. Tämä voidaan toteuttaa esimerkiksi JavaScript'in ObjectID:n avulla. Päivämäärät kannattaa mielestäni olla ISO 8601 -standardin mukaan eli kansainvälisessä muodossa, sillä muut esitystavat aiheuttavat helposti sekaannusta, eivätkä ole niin helposti lajiteltavissa aikajärjestykseen. Avain-arvo-pareissa kannattaa välttää erikoisia merkkejä ja kirjaimia kuten å, ä ja ö, sillä ne voivat aiheuttaa ongelmia toisille, esimerkiksi vieraskielisille rajapinnan käyttäjille tai rajapintaa lukeville ohjelmalle. (Korpela 2013)

5.4 Palvelinohjelman pääteipisteet

Web-rajapinnan palvelinohjelmaan pääteipisteet toteutetaan Express.js-sovelluskehityksellä. REST API:n pääteipisteitä eli polkua tiettyyn resurssiin käsiteltiin luvussa 3.4.3, jota noudattaen web-rajapinnalle suunnitellaan seuraavaksi pääteipisteet. Pääteipisteiden kautta tulee olla mahdollista lisätä, lukea, muokata ja poistaa dataa. Toteutan vaihtoehdot, joilla saadaan näytettyä rajapinnan kaikki data, yksittäinen tieto tai jaettua data eri sivuille. Jos datamäärä kasvaa suureksi, voidaan käyttää sivunumeron sisältävää pääteipistettä, jolloin kukin sivu näyttää tietyn määrän tuloksia. Nämä toiminnot voidaan toteuttaa neljällä http-metodilla ja kolmella eri pääteipisteellä, josta taulukko 3 kertoo.

TAULUKKO 3. Suunnitellut päätepisteet ja http-metodien käyttö

Päätepiste	GET	POST	UPDATE	DELETE
/api/v1/data	Palauttaa kaiken rajapinnan datan	Luo uuden tietueen	-	-
/api/v1/data/:id	Palauttaa yhden tietueen	-	Muokkaa yksittäistä tietuetta	Poistaa yksittäisen tietueen
/api/v1/data/page/:page-number	Palauttaa dataa sivulta X	-	-	-

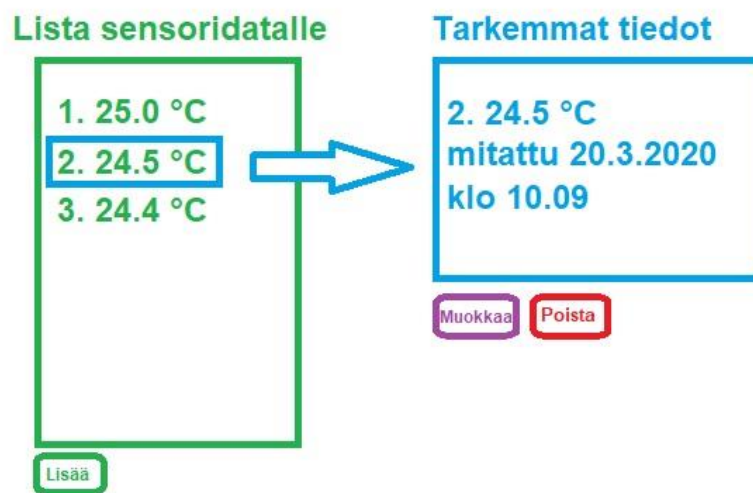
5.5 Arduino ja sensoridatan tuonti rajapintaan

Työssä rajapintaa täytetään sensoridatalla. Arduino-laitteelta halutaan sensoridataa MQTT-protokollan yli MongoDB-tietokantaan. Tätä varten tarvitaan ohjelma Arduinon sensoridatan keräämiseen ja lähettämiseen MQTT:n yli, sekä MQTT:hen perustuva viestejä tilaava ohjelma. Arduinon ohjelma toteutetaan Arduino Ide:llä, jota olen aiemmin käyttänyt vastaavanlaisiin yksinkertaisiin sovelluksiin. Arduinolle tehtävä kytkentä sisältää lämpötilasensorin ja Ethernet-yhteyteen tarvittavan Arduino Ethernet Shield -komponentin. Viestejä tilaavan ohjelman toteutan Node.js:llä, jota voi pyörittää yksinkertaisesti tietokoneella. Node.js-ohjelmassa voidaan tarvittaessa käsitellä tai parsia dataa ennen tietokantaan lisäämistä.

5.6 Web-käyttöliittymä

Rajapinnan valmistuttua tehdään sovellus, joka käyttää rajapintaa. Vaikka rajapintaa tullaan testaamaan työn aikana muun muassa Postman:illa, sovelluksen teko viimeistäänkin osoittaa sen, että rajapinnan toiminnot toimivat halutulla tavalla. Sovelluksena tehdään web-käyttöliittymä, jolla voidaan hallita rajapinnan dataa kätevämmiin. Käyttöliittymä tehdään AngularJS:llä, jota opettelen projektin aikana. Käyttöliittymän avulla on tarkoitus voida lisätä, muokata ja poistaa dataa. Toteutusta on hahmoteltu kuvassa 11. Vasemmalla puolella sivua on lista, josta voi valita klikkaamalla yhden mitatun tiedon. Sen jälkeen oikealle puolelle avautuu tarkemmat tiedot näkymä, jossa esitetään kaikki tiedot kyseiselle mittaukselle. Lisääminen, muokkaaminen ja poistaminen on mahdollista painikkeilla.

Web-käyttöliittymä



Kuva 11 Käyttöliittymän hahmotelma

6 TYÖN WEB-TEKNIIKAT

REST-rajapinnan luominen on mahdollista monilla ohjelmointikielillä ja kehitykseen on olemassa paljon erilaisia sovelluskehyskiä. Tässä luvussa tarkastellaan valitsemaani MEAN-stack:ia, joilla myöhemmin rakennetaan rajapinta ja web-käyttöliittymä.

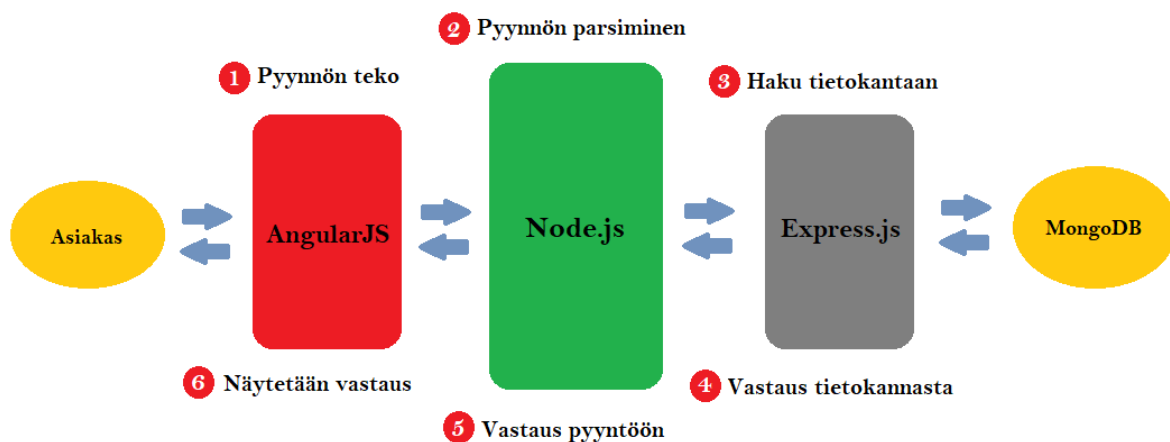
6.1 Tekniikan valinta

Mielestäni nykyaikaisen web-rajapinnan kannattaa palauttaa JSON-muotoista dataa, sillä sitä on helppo lukea ja käsitellä. Tällöin myös datan tallentaminen tietokantaan JSON-muodossa pitää projektin yksinkertaisena. Pienen selvittelyn jälkeen selvisi, että useat REST-rajapinnan luontia käsittelevät artikkelista ja oppaista useat suosivat MEAN stack:iä, joka sopii hyvin myös tähän projektiin.

MQTT:n yli tulevan datan käsittelyyn käytän NodeJS:ää, jolla JavaScript-koodia voidaan suorittaa palvelimella. Web-rajapinnan rakentamiseen JavaScript:illä käytän apunani Express-sovelluskehystä (engl. framework), joka toimii ilmeisen hyvin NodeJS:n kanssa juuri web-rajapintojen ja web-sovelusten tekoon. Rajapinnan projektin pohjana käytän itselleni ennestään tuntematonta Angular:ia, jolla teen myöhemmin myös web-käyttöliittymän. Tietokantana käytän pilvipalveluna toimivaa versiota MongoDB:stä.

6.2 MEAN stack

MEAN stack tarkoittaa MongoDB, Express.js, AngularJS ja Node.js tekniikoiden yhdistelmää eli ”MEAN” tulee yksinkertaisesti kyseisten tekniikoiden alkukirjaimista. Kyseiset tekniikat käyvät hyvin yhteen JSON-muotoisen tiedon käsittelyssä tietokannan, asiakasohjelman ja palvelinohjelman välillä. Kuvassa 12 on esitetty MEAN stack:in toimintaa kaaviossa. (Guru99 2020)



Kuva 12 MEAN stack -kaavio

1. Asiakasohjelmalla tehdään pyyntö, jonka AngularJS käsittelee.
2. Pyyntö siirtyy Node.js:än, jossa se parsitaan.

3. Express.js tekee kutsun MongoDB -tietokantaan. Kutsu voi olla datan hakemista (get) tai muita operaatioita kuten set, patch ja update, jotka asettavat datan.
4. MongoDB ottaa vastaan pyynnön ja palauttaa vastauksen Express.js:lle.
5. Node.js palauttaa pyynnön asiakasohjelmaan.
6. Tuloksena asiakasohjelmassa näytetään haettu tieto.

6.2.1 MongoDB

MongoDB on suosittu ilmainen NoSQL-tietokanta. NoSQL (Not Only SQL) tarkoittaa perinteisestä relaatiomallista poikkeavaa tietokantaa, joka on muun muassa kyselykieleltään (engl. query language) ja tietorakenteeltaan joustavampi kuin relaatiotietokanta. MongoDB on ns. JSON:in kaltaisten dokumenttien tietokanta, jossa kentät dokumenteissa voivat vaihdella vapaasti tietokannan sisällä. Datastruktuuria ei siis ole tiukasti määritelty, vaan esimerkiksi avain-arvo-parien avaimetkin voivat muuttua vapaasti. Tarkemmin ottaen MongoDB käyttää JSON:in kaltaista BSON (Binary JSON) datamuotoa, jonka itse MongoDB on kehittänyt laajentamaan JSON:ia muun muassa datatyypeillä. BSON-muoto on helppo ymmärtää, mikäli käyttäjä tuntee JSON:in entuudestaan. MongoDB:n voi ladata käyttöönsä paikallisesti, tai sitä voi käyttää pilvipalveluna ja se löytyy muun muassa Google Cloud:ista ja AWS:stä (Amazon Web Services). (MongoDb 2020)

MongoDB on kehittäjien keskuudessa suosittu, sillä se on helppokäyttöinen, tarjoaa työkaluja, sekä ylipäättänsä soveltuu hyvin moderneihin web-sovelluksiin ja suuren datamäärän kanssa. MongoDB:n tärkeimpiä ominaisuuksia on muun muassa valmiiksi hyvä turvallisuus, täysin hallittu ja jatkuva varmuuskopiointi, automaattisesti luodut kaaviot, sekä reaaliaikainen suorituskykypaneeli. (Violino 2018)

6.2.2 Express.js

Express.js on erittäin suosittu, minimaalinen sovelluskehys Node.js:lle, jolla voidaan toteuttaa yksinkertaisemmin palvelinpuolen ohjelmistoja. Express yksinkertaistaa muun muassa rajapinnan päätepisteiden ohjelmointia, virheenkäsittelyä ja ns. väliohjelmiston (engl. middleware) kehitystä http-protokollan mukaisesti. (Express ei pvm)

Express:in sanotaan olevan helppo sovelluskehys oppia. Se ei pakota käyttämään mitään tiettyä suunnittelumallia (engl. design pattern), kuten MVC:tä, joten sovelluksia voi rakentaa omien mieltymysten mukaan ilman suurempaa oppimiskäyrää. (Yang 2016)

6.2.3 AngularJS

AngularJS on Googlen omistama JavaScript-sovelluskehys web-sovellusten luontiin. Angular laajentaa HTML:n syntaksia, jotta se vastaisi paremmin nykypäivän web-kehityksen vaatimuksia. Sillä voidaan siis lisätä toiminnallisuuksia monelle tuttuun HTML-kieleen, joka ei itsessään riitä monimutkaisten sivustojen tekoon. Angular on kirjoitettu TypeScript-ohjelmointikielellä, jota käytetään paljon

web-kehityksessä. TypeScript on ylivoimaisesti suosituin kieli käytettäväksi myös Angular:illa kehittäessä, vaikka Angular toimii muun muassa JavaScript- ja Dart-ohjelmointikieltenkin kanssa. Angular:ia käytetään dynaamisten web-sovellusten luontiin ja erityisesti yksisivuisten sovellusten kehityksessä. (Marx 2017)

Angular:ista puhutaan joskus myös alustana, sillä siihen on kehitetty avuksi paljon työkaluja, joilla voidaan kehittää ilman kolmansien osapuolten tekemiä ohjelmointikirjastoja. Työkaluja ovat esimerkiksi Angular-Cli, Angular Universal ja Angular Material, joilla saadaan lisää ominaisuuksia kuten nopea projektin generointi, palvelinpuolen renderointi ja tyylikkäästä käyttöliittymän komponentit. Työkalut eivät itsessään kuulu Angular:iin vaan niitä voidaan ottaa käyttöön valinnaisesti. (Marx 2017)

6.2.4 Node.js

Node.js on sovelluskehys palvelinpuolen JavaScript-sovelluksille. Node.js tuo JavaScript:in muuallekin, kuin selainten käyttöön. Ennen Node.js:än syntyä vuonna 2009, JavaScript-koodia oli mahdollista suorittaa vain selaimella. Kaikilla selaimilla on ns. JavaScript-moottori, joka vastaa JavaScript-koodin suorittamista. Esimerkiksi Firefox käyttää SpiderMonkey nimistä moottoria, kun taas Google Chrome käyttää V8:a. Node.js on rakennettu Google Chromen V8 JavaScript-moottorin päälle. (Henderson 2019)

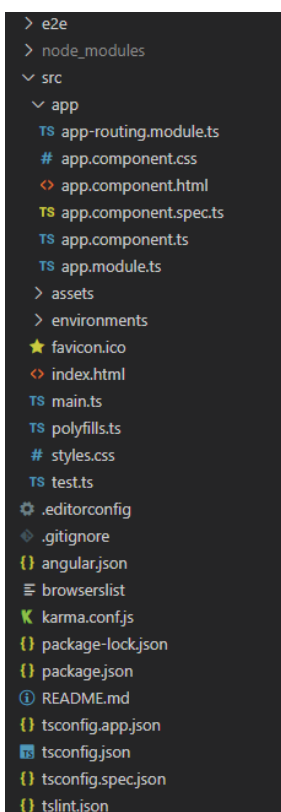
Node.js soveltuu hyvin dataa käsitteleviin reaaliaikaisiin sovelluksiin. Se on tehokas, sillä se käyttää esteetöntä ja tapahtumapohjaista I/O-järjestelmää, jossa mikään käsky ei estä toisen käskyn samanaikaista ajoa. Esimerkiksi rajapintojen tapauksessa tämä tarkoittaa, että eri käyttäjille voidaan suorittaa yhtä aikaa toimintoja, kuten ottaa pyyntö vastaan ja alkaa hakea dataa, vaikka palvelin olisi vastaamassa toisen käyttäjän pyyntöön. Node.js:ää käytetään muun muassa komentorivin sovelluksiin, reaaliaikaisiin chat-sovelluksiin ja REST API palvelinpuolen ohjelmistoon. (Patel 2018)

7 WEB-RAJAPINNAN TOTEUTUS

Tässä luvussa rakennetaan avoin RESTful web-rajapinta MEAN-stack:ia käyttäen. Ohjelmointiprojektin tärkeimmät osuudet pyritään kertomaan tekstissä selkeästi, kuvia apuna käyttäen.

7.1 Projektin aloitus

Ohjelmointiprojekti alkoi Angular CLI:llä uuden projektin luomisella. Komento "ng new projektin-nimi" asentaa Angular:in tarvitsemat npm-paketit, alustaa projektin hakemiston ja generoi valmiiksi kansiorakenteen ja tiedostoja projektiin. (Kuva 13) Komento siis generoi yksinkertaisen projektin, josta lähtee liikkeelle. Kansiossa nimeltä "src" on tarkoitus pitää sovelluksen logiikka, data ja muut lähteet. Muun muassa projektin pääsivu, index.html, sijaitsee src-kansiossa. Ylimmällä tasolla kansioiden kanssa olevien tiedostojen taas on tarkoitus tukea testausta ja sovelluksen ajamista, joten muun muassa koko projektin laajuiset konfiguraatiodokumentit ovat täällä. Projektin juuressa sijaitsee esimerkiksi "package.json", jossa on metatietoa projektiin liittyen, jotta esimerkiksi npm-pakettien riippuvuudet ovat saatavilla kaikille projektin tiedostoille.

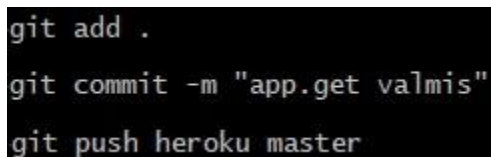


Kuva 13 Angular-projektin hakemisto

7.2 Heroku-alustalla julkaisu ja tietokannan luominen

Luvussa 11.3 tarkemmin läpikäymälleni Heroku-alustalle sovelluksen lisääminen tapahtuu yksinkertaisesti menemällä projektikansioon komentokehoteessa ja kirjoittamalla komento "heroku create". Tämä komento generoi satunnaisen verkko-osoitteen (<https://immense-bayou-52372.herokuapp.com/>).

kuapp.com), joka on nyt varattu sovellukselle. Komento myös luo git-versionhallintaan liittyvän "remote" version, jotta projektin koodi ei ole ainoastaan paikallisena koneella. Ohjelmointiprojektin edetessä ja muutoksia tehdessä uusin versio Heroku-alustalle saadaan lisättyä esimerkiksi Git:in avulla kuvan 14 mukaisilla komennoilla. Kuvan komennoilla valitaan kaikki tiedostot projektin juuresta, lisätään viesti ja siirretään tiedostot säilytyspaikkaan, sekä siirretään push-komennolla Heroku-alustalle.



```
git add .
git commit -m "app.get valmis"
git push heroku master
```

Kuva 14 Git-komennot

Käytän projektiin MongoDB Atlas:ta, joka on pilvipalveluna toimiva tietokanta. Tietokannasta on sekä ilmaisia että maksullisia versioita, joista valitsin ilmaisen version, sillä se vaikuttaisi olevan riittävä tähän projektiin ja sen sanotaan olevan ikuisesti ilmainen. Valmiille tietokannalle täytyy sallia yhteys IP-osoitteesta, josta sitä aikoo käyttää. Yhteys voidaan sallia kaikkialta käyttämällä "0.0.0.0/0" IP-osoitetta. Tietokannalle täytyy myös luoda käyttäjä, joten tämä suojaa pääsyn muilta. Tämän jälkeen tietokannan klusteriin voidaan yhdistää esimerkiksi käyttäen MongoDB:n Node.js-ajuria.

7.3 Tietokantaan yhdistäminen ja Express.js

Node.js:n ja Express.js:n käyttöä varten Angular-projektissa lisäsin projektin juureen "server.js" -tiedoston, jossa luodaan yhteys tietokantaan ja ohjelmoidaan RESTful API:n päätepisteet. Tämän tiedoston määritin myös package.json-tiedoston start-kohtaan eli käynnistymään ensimmäisenä, jotta sovellus toimii myös Heroku-alustalla.

Express.js:n ja MongoDB:n käyttöä varten Node.js:n kanssa tarvitsee asentaa MongoDB:n Node.js ajuri ja Express.js sovelluskehys, jotka on helppo asentaa npm-paketteina. Lisäksi asennetaan "body-parser" -kirjasto. Komento "npm install express mongodb body-parser --save" asentaa ja tallentaa nämä, sekä tallentaa riippuvuudet package.json -tiedostoon. MongoDB ajurin ja Express.js sovelluskehysten saa käyttöönsä Node.js:n require -funktiolla kuvan 15 osoittamalla tavalla riveillä 1 ja 2.

```

JS server.js > app.get("/api/v1/data") callback > toArray() callback
1 var express = require("express");
2 var mongodb = require("mongodb");
3 var bodyParser = require("body-parser");
4 var app = express();
5 app.use(bodyParser.json());
6 var data_collection = "data";
7 // Luodaan tietokanta muuttuja, jota voidaan uudelleenkäytetään yhteyteen (connection pool)
8 var db;
9
10 // Yhdistetään tietokantaan ennen palvelinohjelmiston käynnistämistä
11 | mongodb.MongoClient.connect("mongodb://käyttäjätunnus:salasana@cluster0-shard-00-00-2pav5.mongodb.net:27017,
12 | if (err) {
13 |   console.log(err);
14 |   process.exit(1);
15 | }
16
17 // Tallennetaan tietokantaobjekti callback:ista uudelleenkäyttöä varten
18 db = client.db();
19 console.log("Yhteys tietokantaan valmis");
20
21 // Alustetaan sovellus
22 var server = app.listen(process.env.PORT || 8080, function () {
23 |   var port = server.address().port;
24 |   console.log("Sovellus käynnissä portissa", port);
25 | });
26

```

Kuva 15 Muodostetaan yhteys tietokantaan

Kuvassa 15 MongoClient() -konstruktorilla luodaan uusi instanssi ja connect-metodilla otetaan yhteys tietokantaan. Käytin yhteyteen tietokannan sivulta löytyviä kolmea klusterin osoitetta. Osoite sisältää tietokannan käyttäjätunnuksen ja salasanan, joten ne ovat tämän työn kuvissa ja liitteissä muutettu.

Tietokantaan muodostettuja yhteyksiä kannattaa uudelleenkäyttää mahdollisimman paljon sen sijaan, että luodaan ja tuhotaan yhteys useita kertoja ohjelmassa, koska ne ovat kalliita suorituskyvyn ja viiveen kannalta. Express.js sovelluskehyksellä puolestaan alustetaan sovellus ja app.listen() -funktioilla saadaan luotua Node.js palvelin määriteltyyn porttiin ja isännöitsijään (engl. host). Kuvan 15 rivillä 22 annetaan portiksi "process.env.PORT", joka on eräänlainen ympäristömuuttuja (eng. environment variable), jolla Herokun tapauksessa annetaan web-palvelimelle mahdollisuus määrittellä portti. Vaihtoehtoisesti portiksi asetettiin 8080, jotta kehitysvaiheessa sovellusta voidaan ajaa paikallisena portissa 8080.

7.4 RESTful API palvelimen ohjelmointi ja päätepisteet

Rajapinnan toiminta ja päätepisteet suunniteltiin luvussa 5.4, joten ohjelmointi tapahtuu näiden suunnitelmien pohjalta. Päätepisteiden ohjelmointiin käytin Express.js -sovelluskehystä. Sen avulla voidaan määrittellä reititys (engl. routing), eli kuinka web-rajapinta vastaa asiakasohjelman pyyntöön mistäkin päätepisteestä, ottaen huomioon http-metodin. Express:issä reitti määritellään seuraavan struktuurin mukaisesti: app.METHOD(PATH, HANDLER)

- app tarkoittaa express:in instanssia
- METHOD on HTTP-pyyntöön metodi, koodissa kirjoitetaan pienellä
- PATH on polku palvelimella
- HANDLER on funktio, joka suoritetaan tähän reittiin tullessa. Reitillä voi olla yksi tai useampia handler-funktioita.

Kuvassa 16 on toteutettu reitti rajapinnan kaiken datan hakemiseen. Koodissa haetaan tietokannassa olevat tietueet ja näytetään ne JSON-muodossa, kun pyyntö tehdään get-metodilla polussa `"/api/v1/data"`. Lisäksi reitille on toteutettu virheen käsittely yksinkertaisella funktiolla, jotta virhetilanteessa saadaan tarkempaa tietoa kuten http-statuskoodi ja selitys virheelle.

```
// virheen käsittely
function handleError(res, reason, message, code){
  console.log("ERROR: " + reason);
  res.status(code || 500).json({error: message});
}
// GET -- rajapinnan kaiken datan palauttava reitti
app.get("/api/v1/data", function(req, res) {
  db.collection(data_collection).find({}).toArray(function(err, docs) {
    if (err) {
      handleError(res, err.message, "Datan hakeminen epäonnistui.");
    } else {
      res.status(200).json(docs);
    }
  });
});
```

Kuva 16 Virheen käsittely ja get-metodi

Polulla määritellään käytännössä aiemmin generoidun osoitteen perään lisättävä osuus. Kun sovellus päivitetään Herokuun, rajapinnan kaikki data saadaan nyt get-metodilla osoitteesta `"https://immense-bayou-52372.herokuapp.com/api/v1/data/"`. Get-metodia voidaan testata menemällä selaimella edellä mainittuun osoitteeseen, tai ohjelmien kuten Postman avulla. Tuloksena nähdään kaikki rajapinnan data JSON-muodossa eli avain-arvo-pareina. (Kuva 17)

```
▼ 0:
  _id: "5ee9daa378c12e36ac256d70"
  arvo: "25.26"
  yksikko: "C"
  sensori: "LM35"
  mitattu: "2020-03-17T08:56:03.613Z"
  muokattu: null
  lisetty: "Arduino Uno"
▼ 1:
  _id: "5ee9daad78c12e36ac256d71"
  arvo: "25.26"
  yksikko: "C"
  sensori: "LM35"
  mitattu: "2020-03-17T08:56:13.601Z"
  muokattu: null
  lisetty: "Arduino Uno"
▼ 2:
  _id: "5ee9dab778c12e36ac256d72"
  arvo: "25.36"
  yksikko: "C"
  sensori: "LM35"
  mitattu: "2020-03-17T08:56:23.590Z"
  muokattu: null
  lisetty: "Arduino Uno"
```

Kuva 17 Rajapinnan palauttama data

Sivunumeron sisältävässä reitissä rajoitin haettavan datan 25 tietueeseen. Yksinkertainen tapa mielestäni oli käyttää MongoDB:n `skip()`- ja `limit()`-funktioita, kuten kuvassa 18. Kehitysvaiheessa tein sellaisen huomion, että ns. tarkempi reitti täytyy olla koodissa ennen kaiken datan hakevaa reittiä, sillä rajapinta kaatui tietyissä tapauksissa. Tähän löytyi apua muun muassa stackoverflow:sta, että `http`-metodin olleessa sama useammalle reitille, tarkempi reitti tulee aina olla ensin.

```
// GET -- rajapinnan dataa sivulta X palauttava reitti, limit 25
app.get("/api/v1/data/page/:pagenumber", function(req, res) {
  var skips = 25 * req.params.pagenumber - 25;
  db.collection(data_collection).find({}).skip(skips).limit(25).toArray(function(err, docs) {
    if (err) {
      handleError(res, err.message, "Datan hakeminen epäonnistui.");
    } else {
      res.status(200).json(docs);
    }
  });
});
```

Kuva 18 Sivunumeron sisältävä reitti

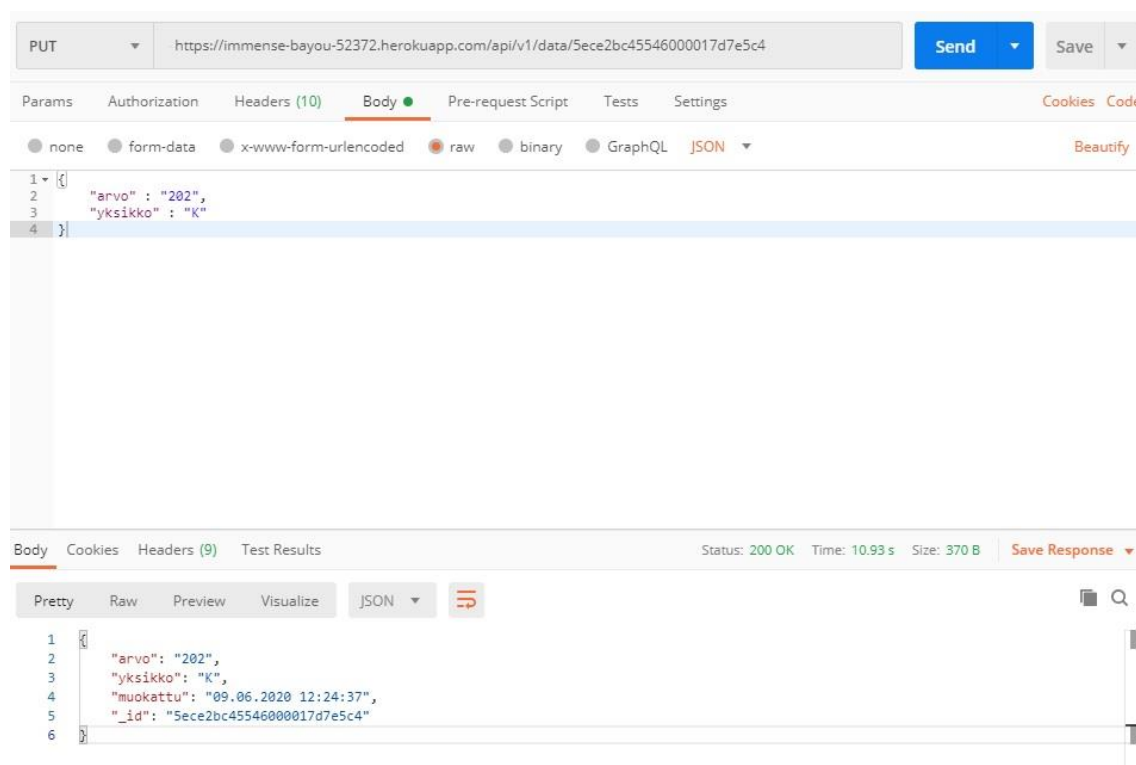
Muut rajapinnan toiminnot kehitetään vastaavalla tavalla, eri reittien avulla. Reiteissä määritellään ja muokataan arvoja muuttujien avulla, jotta ne saadaan haluttuun muotoon, kuten kuvassa 19 on tehty `post`-metodin reitissä, jossa uusi tietue luodaan. Tähän reittiin lisäsin ehdoksi, että jokaisella tiedolla on oltava jokin arvo. Uuteen tietoon lisätään myös avaimille arvoja, kuten myöhemmin tehtävää käyttöliittymää varten "UI", sekä päivämäärä ellei sitä ole syötetty pyynnössä. Tietokannan muutoksia tehdään koodissa MongoDB -ajurin `collection`-metodin avulla. Kokonaisuudessaan koodi löytyy liitteestä 1.

```
// POST -- uuden tietueen luonti
app.post("/api/v1/data", function(req, res) {
  var newData = req.body;
  newData.lisatty = "UI";
  if (!req.body.arvo) {
    handleError(res, "Virhe tiedon syöttämisessä", "Lisää 'arvo:'", 400);
  } else {
    if (!req.body.mitattu) { // jos mittaukselle ei anneta päivämäärää, otetaan tämänhetkinen aika automaattisesti
      var now = new Date();
      newData.mitattu = now;
    }
    db.collection(data_collection).insertOne(newData, function(err, doc) {
      if (err) {
        handleError(res, err.message, "Tiedon luonti epäonnistui.");
      } else {
        res.status(201).json(doc.ops[0]);
      }
    });
  }
});
```

Kuva 19 Uuden tietueen luonnin reitti

Kehitysvaiheessa rajapinnan reittejä on kätevää testata esimerkiksi Postman-ohjelman avulla. Ohjelmalla voidaan tehdä pyyntöjä eri `http`-metodeilla ja kirjoittaa pyyntöihin sisältöä JSON-muodossa, jonka jälkeen pyyntö lähetetään `send`-painikkeesta. Pyyntö onnistuessa näytetään onnistunut statuskoodi (200-299) ja esitetään vastaus. Kuvassa 20 on esimerkki onnistuneesta pyynnöstä, statuskoodista ja vastauksesta. Mikäli pyyntö epäonnistuu, vastauksena tulee reitille määrittelemämme

virheilmoitus tai muunlainen virheellinen vastaus, jos reitti on syötetty väärin. Käyttäjän tekemän virheellisen pyynnön statuskoodi on väliltä 400-499.



Kuva 20 Postman-ohjelman näkymä

8 RAJAPINNAN DOKUMENTOINTI

Rajapinnan dokumentaation täytyy tuoda ilmi tietoa, jota tarvitaan rajapinnan onnistuneeseen käyttöön ja integrointiin. Kuten aiemmin todettua, dokumentoinnin kannattaa sisältää teknisen kirjoituksen lisäksi koodinäytteitä ja esimerkkejä, jotta rajapinnan toimintaa ymmärretään paremmin ja nopeammin. REST API:en tapauksessa melko vakiintunut käytäntö on Swagger-työkalun avulla dokumentointi, joten tässä luvussa käsitellään tätä työkalua ja sen käyttöä työssä.

8.1 OpenAPI Specification

OpenAPI-spesifikaatio standardoi REST API:en kuvauksen muodon. Kuvauksen muoto on helppo oppia, sillä se on tarkoitettu ihmisten sekä koneiden luettavaksi. Kuvaus voidaan kirjoittaa YAML:illa tai JSON:illa. OpenAPI-tiedostossa voidaan kuvailla eli selittää auki koko rajapinnan toiminta, kuten esimerkiksi päätepisteet, operaatiot eli toiminnot, parametrit operaatioille, autentikaatio, yhteystiedot, lisenssi, käyttöehdot ja muuta tietoa. (Smartbear, 2020)

OpenAPI-spesifikaatiota käytetään työkaluissa, joita on kehitetty muun muassa rajapinnan dokumentaation generoimiseen ja testaukseen. OpenAPI:n kehitystä tukee OpenAPI-aloite, johon osallistuu yli 30 organisaatiota, mukaan lukien Microsoft, Google ja IBM. (Pinkham, 2017)

8.2 Swagger

Swagger on ohjelmoijien keskuudessa suosittu joukko avoimen lähdekoodin työkaluja, jotka käyttävät OpenAPI-spesifikaatiota. Swagger-työkalujen avulla voidaan suunnitella, rakentaa, dokumentoida ja käyttää REST-rajapintoja. Työkaluja kehittää Smartbear Software niminen yritys. (Smartbear ei pvm)

Swagger-työkaluista tärkeimpiä ovat Swagger Editor, Swagger UI ja Swagger Codegen. Swagger Editor on editori, jolla dokumentaatiota voidaan kirjoittaa. Swagger UI on rajapinnan resursseja visualisoiva, interaktiivinen työkalu. Swagger Codegen puolestaan mahdollistaa API koodin generoimisen tehdyn dokumentaation pohjalta. Nämä työkalut on tuotu yhteen SwaggerHub-alustalla, joka toimii verkkopohjaisena. (Smartbear ei pvm)

Swagger-dokumentaation lopputuloksena syntyy tiivis ja ytimekäs listaus rajapinnan operaatioista eli päätepisteistä ja metodeista. Dokumentaatio mahdollistaa operaatioiden kokeilun, joka helpottaa sovellusten kehitystä, sillä tiedetään millainen vastaus onnistuneesta tai virheellisestä pyynnöstä tulee. (Smartbear ei pvm)

8.3 API:n dokumentointi

Swagger:in kotisivuilta löytyi ohjeita API dokumentoinnin aloittamiseen, jossa lähestymistapoja sanotaan olevan kaksi; suunnittelu ensin tai koodi ensin. Dokumentoinnin teko ennen koodaamista voi

auttaa näkemään rajapinnan suunnitteluun liittyvät ongelmat, joten tämä lähestymistapa voi osoittautua tehokkaaksi projektissa. Lisäksi se voi auttaa kaikkia rajapinnan parissa työskenteleviä ohjelmiojia tietämään paremmin mitä tavoitteita rajapinnalla on ja kuinka resurssit esitetään. Perinteisempi lähestymistapa on vielä toistaiseksi ollut rajapinnan koodaaminen ensin, kuten myös tässä työssä tehtiin. (SmartBear ei pvm)

Aloitin dokumentoinnin työkalulla nimeltä Swagger Inspector, jossa rajapinnan päätepisteitä voi testata eri metodeilla, aivan kuten Postman-ohjelmassakin. Kun kaikki rajapinnan päätepisteet oli testattu ja valittu, Swagger Inspector:in ”Create API definition” painike loi pohjan dokumentaatiolle ja ohjasi SwaggerHub-työkaluun. Dokumentaatioon syntyi valmiiksi jo päätepisteet, joten ainoastaan niiden tekstejä, kuvauksia ja esimerkkejä täytyi päivittää manuaalisesti. SwaggerHub:issa YAML-koodia pystyy muokkaamaan editorissa, ja vieressä näkyy dokumentaatio interaktiivisena, joten työkalu oli mielestäni hyvin kätevä.

Dokumentaatio luodaan samalla myös verkko-osoitteeseen, joten valmis dokumentaatio voidaan yksinkertaisesti jakaa linkkinä rajapinnan käyttäjille. Vaihtoehtoisesti dokumentaatio voidaan ladata JSON:ina, YAML:ina tai viedä html-koodiksi. Kuvassa 21 on valmis Swagger-dokumentaatio. Menetelmät esitetään Swagger:issa eri väreillä.

Sensor data REST API

0.1 OAS3

API for sensor data.
Base url: <https://immense-bayou-52372.herokuapp.com/>

Servers

<https://immense-bayou-52372.herokuapp.com>

data Operations for data

GET	/api/v1/data/{id}	Returns datum by ID
PUT	/api/v1/data/{id}	Updates datum by ID
DELETE	/api/v1/data/{id}	Deletes datum by ID
GET	/api/v1/data/page/{pagenumber}	Returns data from given page number
GET	/api/v1/data/	Returns all data from API
POST	/api/v1/data/	Creates new datum

Kuva 21 Valmis Swagger-dokumentaatio

Jokainen metodi laajenee Swagger-dokumentaatioissa klikkaamalla metodia kuvan 22 näköiseen muotoon. Metodia voidaan testata dokumentaatioon lisättyjen esimerkkien avulla, ja pyyntö on myös muokattavissa. Näin dokumentin lukijan on helpompi kokeilla ja havainnollistaa rajapinnan toimintaa.

PUT

/api/v1/data/{id} Updates datum by ID

Parameters

Cancel

Name	Description
id * required	id
string	
(path)	<input type="text" value="id - id"/>

Request body

application/json

Examples:

Example

```
{  "arvo": "25.00",  "ykeikko": "C",  "sensori": "IM35"}
```

Servers

These operation-level options override the global server options.

Execute

Responses

Kuva 22 Swagger:in metodille avautuva näkymä

9 DATAN KERÄYS

Rajapinta suunniteltiin ja tehtiin sensoridataa varten, joten seuraavaksi halutaan täyttää rajapintaa Arduino Uno:lla kerättävällä sensoridatalla. Tässä luvussa käsitellään datan keruuta kokonaisuudessaan, eli tutustutaan Arduinoon, MQTT-protokollaan, kehitysalustaan tehtyyn kytkentään, sekä kehitetään tarvittavat ohjelmat.

9.1 Arduino

Arduino on avoimen laitteiston ja lähdekoodin alusta elektroniikan projekteihin. Arduinon kehitysalustaan liitetään ulkoinen kytkentä, joka sisältää erilaisia komponentteja kuten antureita, ledejä tai kytkimiä. Tämän jälkeen Arduinoon syötetään ohjelma, joka käyttää näitä elektroniikan komponentteja eri tavoin mikrokontrollerin ansiosta. Arduinon kehitysalustat sisältävät mikrokontrollerin lisäksi muun muassa USB-liittimen koodin syöttöä varten, virtaliitännät kytkentöjä varten, sekä digitaalisia ja analogisia kytkentäpinnejä, jotka lukevat jännitettä.



Kuva 23 Arduino Uno -kehitysalusta (Arduino 2020)

Arduino on todella suosittu, koska alkuun pääseminen sillä on helppoa ja halpaa verrattuna muihin vastaaviin alustoihin. Arduinoa käytetään paljon harrastelijoiden projekteissa ja opetuskäytössä, ja siihen löytyy paljon ohjeita ja esimerkkiprojekteja. Arduino-ohjelmoinnissa alkuun pääsee kenties parhaiten Arduino IDE -ohjelmalla (Integrated Development Environment), joka löytyy Arduino-yrityksen sivuilta ilmaiseksi. Arduino IDE:llä ohjelmoidaan C/C++ -ohjelmointikielillä, joskin ohjelmointi tapahtuu pitkälti käyttäen valmiita C/C++ -funktioita, joilla Arduino-kortin hallitseminen on tehty todella helpoksi. (Arduino 2020)

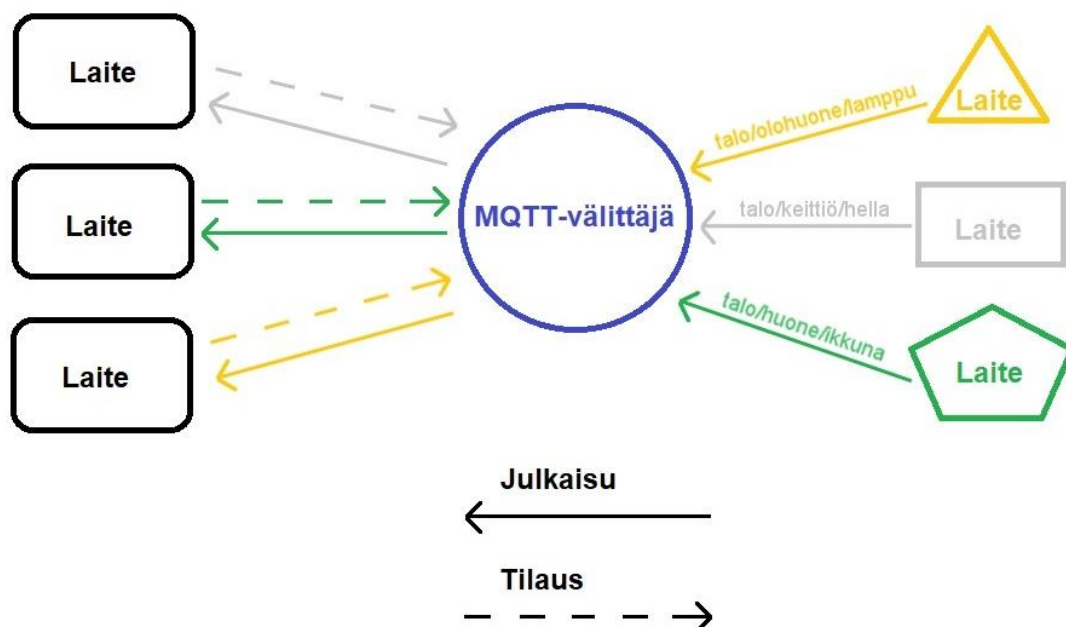
Sulautettujen järjestelmien kokonaiskuvassa ja etenkin teollisuudessa Arduino ja Arduino IDE eivät ole kovinkaan käytettyjä, sillä ne eivät ole useinkaan siihen sopivia. Arduino on kokonainen kehitysalusta, kun taas sulautetuissa järjestelmissä valitaan komponentteja kuten mikrokontrolleri kehitettävän tuotteen vaatimusten mukaan. Arduino IDE puolestaan on huono debuggaukseen ja monia-kan mikrokontrollereita Arduinon käyttämän Atmelin lisäksi ei voida ohjelmoida. Sulautettujen jär-

jestelmien ammattilaiset käyttävät ohjelmistoja, joilla voidaan tehdä vapaasti ja tehokkaasti ohjelmia, sillä mikrokontrolleria voidaan hallita täydellisesti. Esimerkiksi Texas Instrumentsin, ARM:n ja Atmelin tuotteet ovat Arduinoa suositumpia teollisuudessa.

9.2 MQTT

MQTT (Message Queuing Telemetry Transport) on kevytrakenteinen viestintäprotokolla, joka perustuu tietojen julkaisuun ja tilaukseen viestien muodossa. MQTT:llä voidaan muodostaa yhteys useiden laitteiden välille, joten sille löytyy käyttöä koneiden välisessä viestinnässä ja IoT:ssä. Esimerkiksi http:n ongelmana on, että asiakas-palvelin-arkkitehtuuri ja vastausten odottaminen lisäävät viivettä ja käyvät äkkiä raskaaksi, kun laitteita on paljon kuten usein IoT:ssä.

MQTT-protokollaan liittyy muutamia konsepteja, jotta sen toimintaa voidaan ymmärtää. Kuvassa 24 on esitelty MQTT:n toimintaa. Laite voi olla viestejä julkaiseva tai tilaava. Viesteillä tarkoitetaan tietoa, joka halutaan vaihtaa laitteiden välillä eli esimerkiksi jokin käsky tai dataa. Viestejä julkaistaan tiettyyn aihepiiriin (engl. topic) ja niitä tilataan tietyistä aihepiiristä. Aihepiirien avulla käytännössä ohjataan, minne viestejä julkaistaan, jotta voidaan määritellä, haluaako laite tilata ja näin ollen saada minkäkin viestin. Aiheista ja viestien kulusta on vastuussa MQTT:n välittäjä (engl. broker), joka suodattaa viestejä ja ohjaa niitä aihepiiristä kiinnostuneille laitteille. MQTT-välittäjiä on saatavilla pilvipalveluina ja ladattavina versioina. Myös yhteyden muodostamista helpottavia MQTT-kirjastoja laitteiden ohjelmointia varten löytyy monelle eri ohjelmointikielelle. (Santos 2017)



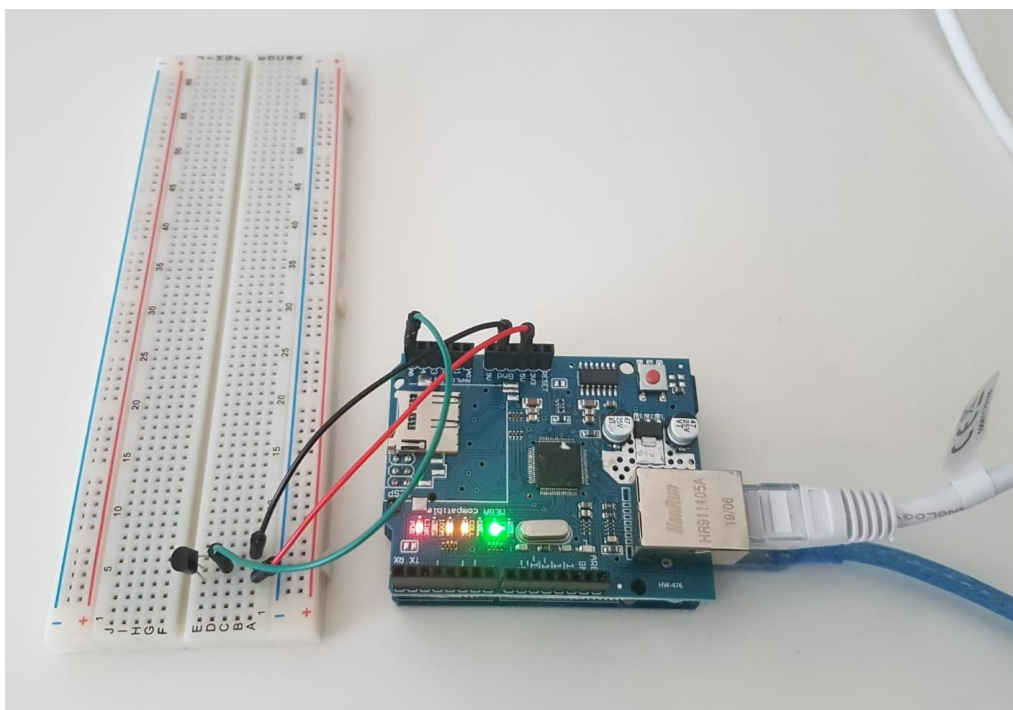
Kuva 24 MQTT viestintä

MQTT:tä käytetään yleensä TCP/IP (Transmission Control Protocol / Internet Protocol) tietoliikenne-protokollien kanssa, joten laitteella ja välittäjällä tulee myös olla TCP/IP stack eli ns pino. Yksinker-

taistettuna TCP/IP pino määrittelee miten laitteet ovat yhteydessä muihin laitteisiin, ja tekee tiedonsiirrosta luotettavaa. TCP/IP:n avulla laitteet osoitteistetaan ja verkkopaketit reititetään verkossa, jotta tavujonoja voidaan lähettää ja vastaanottaa luotettavasti.

9.3 Arduino Uno:n kytkentä

Työssä käytetään Arduino Uno -kehitysalustaa, johon liitetään Arduino Ethernet Shield W5100 -komponentti. Arduino Uno on tässä työssä MQTT:n viestejä lähettävä laite ja tietokone puolestaan viestejä tilaava laite. Ethernet Shield:iä tarvitaan, jotta voimme ohjelmoida viestien julkaisun TCP/IP:n avulla MQTT-välittäjään. Arduino Uno:n kytkentään lisätään LM35 lämpötilasensori, jolla mittauksen arvot saadaan. Arduino Uno liitetään tietokoneeseen USB-kaapelilla ja Ethernet Shield puolestaan liitetään reitittimeen RJ45-kaapelilla.



Kuva 25 Arduinon kytkentä

9.4 Arduino-ohjelma

Arduino Uno:n ohjelma tehdään Arduino IDE:llä, sillä projektissa halutaan edetä nopeasti ja tavoite on yksinkertaisesti saada sensoridataa MQTT:n viestejä tilaavalle ohjelmalle. Kuvan 26 koodissa tärkeimpänä seikkana on MQTT-välittäjän määrittäminen ja siihen yhdistäminen, sillä myöhemmässä vaiheessa koodia julkaisemme viestin välittäjälle. MQTT-välittäjän IP-osoitteeksi asetetaan reitittimen antama tietokoneen IP-osoite, sillä MQTT-välittäjän ohjelmaa ajetaan tietokoneelta. IP-osoite on järkevää asettaa staattiseksi, jottei se vaihdu itsestään projektin aikana.

Arduino Ethernet Shield:in MAC-osoitteena voidaan käyttää keksittyä osoitetta, sillä ainoastaan saman MAC-osoitteen käyttäminen eri laitteelle samassa verkossa aiheuttaisi ongelmia. Arduinon Ethernet Shield on yhdistetty verkkokaapelilla reitittimeen, joten Arduinon IP-osoitteena käytetään reitittimen antamaa IP-osoitetta, joka on eri osoite kuin tietokoneelle.

```

9 // Arduinon MAC- ja IP-osoite
10 byte mac[] = { 0xDE, 0xAD, 0xBE, 0xEF, 0xFE, 0xED };
11 IPAddress ip(192, 168, 100, 12);
12
13 // MQTT-välittäjän IP-osoite
14 byte server[] = { 192, 168, 100, 13 };
15
16 // Tarvitaan yhteyden muodostamiseen
17 EthernetClient ethClient;
18 PubSubClient mqttClient(ethClient);
19
20 void setup()
21 {
22     // 1,1V referenssijännite
23     analogReference(INTERNAL);
24     Serial.begin(9600);
25
26     // Aloitetaan Ethernet-yhteys
27     Ethernet.begin(mac, ip);
28     delay(3000);
29
30     // Määritetään MQTT-välittäjä
31     mqttClient.setServer(server, 1883);
32
33     // Yritetään yhdistää MQTT-välittäjään ID:llä "myClientID"
34     if (mqttClient.connect("myClientID"))
35     {
36         Serial.println("Yhteys on avattu onnistuneesti");
37     }
38     else
39     {
40         Serial.println("Yhteyden avaus epäonnistui");
41     }
42 }

```

Kuva 26 Arduino-ohjelmassa yhteyden muodostaminen

Kuvan 27 koodissa puolestaan luetaan lämpötilasensorilta arvo, joka muutetaan celsiusasteiksi jakamalla se luvulla 9,31. Luku tulee pohjimmiltaan analogRead:in määrittelemästä 1024 "askeleesta". Referenssijännite on asetettu 1,1 volttiin, joten kun se jaetaan 1024:llä askeleella, saadaan noin 1,0742 millivoltia. LM35-lämpötilasensorin on kerrottu havaitsevan yhden celsiusasteen muutoksen per 10 millivoltia, joten tämän jakamalla 1,0742 millivoltilla päästään lukuun 9,31. Näin saadaan mahdollisimman tarkka arvo kyseiseltä sensorilta.

Koodissa arvo lisätään char array eli C-kielen taulukkotyyppiseen muuttujaan, jota pubsubclient:in publish() -funktio vaatii. Lämpötilatieto julkaistaan aiheeseen "arduino_uno/sensoridata/lampotila/", joten viestejä tilaavassa ohjelmassa lämpötilatieto saadaan samaisesta aiheesta. Koodissa on käytetty muutamia Arduino IDE:stä vakiona löytyviä kirjastoja, joiden avulla yhteyden muodostaminen käy helposti. Kokonaisuudessaan Arduino-ohjelman koodi löytyy liitteestä 2.


```

48 // Luetaan LM35-sensorilta arvo
49 int reading=analogRead(sensorPin);
50
51 // Muunnos celsiusasteiksi
52 float temperature = reading / 9.31;
53
54 // Lisätään arvo char array -muuttujaan
55 char fValue[16];
56 dtostrf(temperature, 3, 2, fValue);
57
58 // Jos aikaa kuluu tarpeeksi..
59 if(millis()>(lastTime+1000)) {
60     // Lisätään arvo viestinä lähetettävään bufferiin
61     snprintf(buffer, sizeof(buffer), "%s", fValue);
62     Serial.println(buffer);
63
64     // Julkaistaan viesti "arduino_uno/sensoridata/lampotila/" aiheeseen
65     mqttClient.publish("arduino_uno/sensoridata/lampotila/", buffer);
66     lastTime=millis();
67 }

```

Kuva 27 Arduino-ohjelman koodia

9.5 Node.js -ohjelma

MQTT:n viestejä tilaava ohjelma tehdään Node.js:llä. Ohjelmaa pyöritetään paikallisella tietokoneella, eli käynnissä ollessaan se voi vastaanottaa viestejä tilatusta aiheesta. Viestissä tullut Arduino Uno:n lukema lämpötilatieto lisätään ohjelmassa muiden tietojen kanssa MongoDB-tietokantaan. Ohjelmassa käytetään apuna Node.js:lle tehtyjä MQTT.js ja MongoDB -kirjastoja. Kirjastot on jälleen asennettu npm-paketteina komennon "npm install mqtt --save" tyylisesti.

Kirjastojen dokumentteja lukemalla löytää tietoa ja esimerkkejä, jotta kirjaston muuttujia ja funktioita osaa käyttää oikeaan tarkoitukseen. Esimerkiksi mqtt.connect() -funktiolla kuvan 28 koodissa rivillä 21 yksinkertaisesti otetaan yhteys MQTT-välittäjään määrittelemällä isännöitsijä ja portti samoiksi, kuin Arduinon ohjelmassa. Tämän jälkeen subscribe() -funktiolla tilataan sama aihe, johon Arduinon ohjelma lähettää viestejä. Viestin tultua suoritetaan funktio, jossa viestin lämpötilatieto luetaan, ja tietokantaan lisätään kyseisen mittauksen tiedot eli id, arvo, yksikkö, sensori, mittauksen ajankohta ja mittauksen laite. Kokonaisuudessaan Node.js -ohjelman koodi on liitteessä 3.

```

10 // aihe, josta viestejä tilataan
11 var topicName = "arduino_uno/sensoridata/lampotila/";
12 // yhdistetään tietokantaan kokoelman (collection) kanssa
13 MongoClient.connect(mongodbURI, setupCollection);
14
15 function setupCollection(err, db) {
16     if (err) throw err;
17
18     // kokoelman nimi tietokannassa
19     collection = db.collection("data");
20     // yhdistetään MQTT-välittäjään, host ja port oltava samat kuin Arduinon ohjelmassa
21     client = mqtt.connect({ host: 'localhost', port: 1883 });
22     // tilataan aihe
23     client.subscribe(topicName + "+");
24     // viestin tultua suoritetaan funktio
25     client.on('message', insertEvent);
26
27     console.log("MQTT-välittäjä on yhdistetty tietokantaan.");
28 }
29
30 // funktio, jossa määritellään mitä mongodb-tietokantaan menee
31 function insertEvent(topic, message) {
32
33     // lämpötilatieto tulee viestissä, muunnetaan se string-tyyppiseksi
34     var messageBuffer = Buffer.from(message, 'base64');
35     var messageString = messageBuffer.toString();
36     console.log(messageString);
37     var now = new Date();
38
39     // kokoelmaan lisätään tiedot
40     collection.insertOne(
41         {
42             _id: new ObjectId(),
43             arvo: messageString,
44             yksikko: "C",
45             sensori: "LM35",
46             mitattu: now,
47             muokattu: null,
48             lisatty: "Arduino Uno"
49         },

```

Kuva 28 Node.js MQTT-viestin tilaus ja tietojen lisääminen tietokantaan

10 KÄYTTÖLIITTYMÄN TOTEUTUS

Osana projektia halusin toteuttaa erillisen sovelluksen, joka käyttää tekemääni web-rajapintaa. Tämä demonstroi tilannetta, jossa kenellä tahansa on mahdollisuus tehdä avointa rajapintaa käyttäviä sovelluksia vapaasti. Käyttäjän ei tarvitse ottaa yhteyttä rajapinnan omistajaan tai murehtia tietokantaan liittyvistä asioista, kunhan rajapinnan tarjoamia toimintoja osaa käyttää. Tässä luvussa rajapintaa tulee testattua käytännön sovellukseen, ja mahdolliset virheet tai huonosti toteutetut asiat voivat tulla helpommin esiin.

10.1 Angular-projektin aloitus

RESTful API:a käyttäviä sovelluksia on mahdollista tehdä hyvin monelta eri alustalta, kuten Androidilta, iOS:ltä, Windowsilta tai web-sovelluksena. Päätin tehdä web-sovelluksen, joka on käyttöliittymä rajapinnalle. Web-sovellukseen käytettiin Angular-sovelluskehystä.

Angular-projekti aloitettiin generoimalla se Angular CLI:n "ng new projektin-nimi" komennolla, kuten aiemmin web-rajapinnan ohjelmointiprojektia aloittaessa. Kuvassa 29 näkyvät muut käyttämäni komennot, joista ensimmäisellä luotiin kansio projektin "src/app/" -polkuun. Sen jälkeen luotiin luokka, jossa esitämme datan rakenteen. Lisäksi luotiin käyttöliittymän molemmat komponentit, jotka ovat lista ja tarkemmat tiedot. Luotiin myös Angular service, jota käytetään komponenteissa tiedon lähettämiseen ja vastaanottamiseen. Komentoja on hyvä käyttää, sillä esimerkiksi komponenteille syntyy automaattisesti niiden css-, html- ja ts-tiedostot.

```
D:\Projektit\angularwebapp>mkdir src\app\sensor-data
D:\Projektit\angularwebapp>ng generate class sensor-data\sensor-datum
? Would you like to share anonymous usage data about this project with the Angular Team at
Google under Google's Privacy Policy at https://policies.google.com/privacy? For more
details and how to change this setting, see http://angular.io/analytics. No
CREATE src/app/sensor-data/sensor-datum.spec.ts (175 bytes)
CREATE src/app/sensor-data/sensor-datum.ts (29 bytes)
D:\Projektit\angularwebapp>ng generate component sensor-data\sensor-datum-list
CREATE src/app/sensor-data/sensor-datum-list/sensor-datum-list.component.html (32 bytes)
CREATE src/app/sensor-data/sensor-datum-list/sensor-datum-list.component.spec.ts (693 bytes)
CREATE src/app/sensor-data/sensor-datum-list/sensor-datum-list.component.ts (317 bytes)
CREATE src/app/sensor-data/sensor-datum-list/sensor-datum-list.component.css (0 bytes)
UPDATE src/app/app.module.ts (527 bytes)
D:\Projektit\angularwebapp>ng generate component sensor-data\sensor-datum-details
CREATE src/app/sensor-data/sensor-datum-details/sensor-datum-details.component.html (35 bytes)
CREATE src/app/sensor-data/sensor-datum-details/sensor-datum-details.component.spec.ts (714 bytes)
CREATE src/app/sensor-data/sensor-datum-details/sensor-datum-details.component.ts (329 bytes)
CREATE src/app/sensor-data/sensor-datum-details/sensor-datum-details.component.css (0 bytes)
UPDATE src/app/app.module.ts (673 bytes)
D:\Projektit\angularwebapp>ng generate service sensor-data\sensor-datum
CREATE src/app/sensor-data/sensor-datum.service.spec.ts (383 bytes)
CREATE src/app/sensor-data/sensor-datum.service.ts (140 bytes)
D:\Projektit\angularwebapp>
```

Kuva 29 Komennot Angular-projektin alustukseen

10.2 Käyttöliittymän ohjelmointi

Angularissa komponenttien tehtävä on ainoastaan mahdollistaa käyttökokemus. Komponenttien ulkoasu ja ohjelmointilogiikka toteutetaan kunkin komponentin tiedostoihin. Service-tiedostoa puoles-

taan käytetään tehtäviin, kuten datan noutamiseen palvelimelta tai käyttäjän syötteiden tarkistamiseen, jolloin nämä tehtävät ovat kaikkien komponenttien käytössä. Näitä periaatteita noudattaen service:en määriteltiin web-rajapinnan url-osoite ja kehitettiin rajapinnan toimintoja vastaavat osuudet käyttöliittymään. (Kuva 30) (Angular 2020)

```

6  export class SensorDatumService {
7
8      private dataUrl = "https://immense-bayou-52372.herokuapp.com/api/v1/data";
9
10     constructor(private http: Http) { }
11
12     // get("/api/v1/data")
13     getData(): Promise<void | SensorDatum[]> {
14         return this.http.get(this.dataUrl)
15             .toPromise()
16             .then(response => response.json() as SensorDatum[])
17             .catch(error);
18     }
19
20     // post("/api/v1/data")
21     createNewDatum(newDatum: SensorDatum): Promise<void | SensorDatum> {
22         return this.http.post(this.dataUrl, newDatum)
23             .toPromise()
24             .then(response => response.json() as SensorDatum)
25             .catch(error);
26     }
27
28     // put("/api/data/v1/:id")
29     updateDatum(updData: SensorDatum): Promise<void | SensorDatum> {
30         var updUrl = this.dataUrl + "/" + updData._id;
31         return this.http.put(updUrl, updData)
32             .toPromise()
33             .then(response => response.json() as SensorDatum)
34             .catch(error);
35     }
36
37     // delete("/api/data/v1/:id")
38     deleteDatum(delById: String): Promise<void | String> {
39         return this.http.delete(this.dataUrl + "/" + delById)
40             .toPromise()
41             .then(response => response.json() as String)
42             .catch(error);
43     }
44 }

```

Kuva 30 Projektin service-tiedosto

Service käyttää kuvassa 31 näkyvää luokkaa, jossa määritellään mitä kohtia yksittäinen tieto sisältää.


```
export class SensorDatum {
  _id?: string;
  arvo: string;
  yksikko: string;
  sensori: string;
  mitattu: string;
  muokattu: string;
  lisatty: string;
}
```

Kuva 31 Luokka

Määrittelin alkuvaiheessa käyttöliittymälle kaksi komponenttia eli listan datalle ja yksityiskohdat tiedolle. Tiedon yksityiskohdat sisältävän komponentin html-tiedostossa määrittelin painikkeet, joilla voidaan lisätä, muokata ja poistaa tieto. Input-kenttään sai sidottua dataa NgModel'in avulla. (Kuva 32)

```
src > app > sensor_data > sensor-datum-details > sensor-datum-details.component.html > ...
1 <div *ngIf="sensorsdatum" class="row">
2   <div class="col-md-12">
3     <h2 *ngIf="sensorsdatum._id">Tiedot</h2>
4     <h2 *ngIf="!sensorsdatum._id">Uusi sensoridata</h2>
5   </div>
6 </div>
7 <div *ngIf="sensorsdatum" class="row">
8   <form class="col-md-12">
9     <div class="form-group">
10      <label for="sensorsdatum-arvo">Id</label>
11      <input class="form-control" name="sensorsdatum-id" [(ngModel)]="sensorsdatum._id" readonly/>
12    </div>
13    <div class="form-group">
14      <label for="sensorsdatum-arvo">Arvo</label>
15      <input class="form-control" name="sensorsdatum-arvo" [(ngModel)]="sensorsdatum.arvo" placeholder="20"/>
16    </div>
17    <div class="form-group">
18      <label for="sensorsdatum-arvo">Yksikkö</label>
19      <input class="form-control" name="sensorsdatum-yksikko" [(ngModel)]="sensorsdatum.yksikko" placeholder="C"/>
20    </div>
21    <div class="form-group">
22      <label for="sensorsdatum-arvo">Sensori</label>
23      <input class="form-control" name="sensorsdatum-sensori" [(ngModel)]="sensorsdatum.sensori" placeholder="LM35"/>
24    </div>
25    <div class="form-group">
26      <label for="sensorsdatum-createDate">Mittauksen pvm</label>
27      <input class="form-control" name="sensorsdatum-mitattu" [(ngModel)]="sensorsdatum.mitattu" placeholder=""/>
28    </div>
29    <div class="form-group">
30      <label for="sensorsdatum-editDate">Muokkauksen pvm</label>
31      <input class="form-control" name="sensorsdatum-muokattu" [(ngModel)]="sensorsdatum.muokattu" placeholder="" readonly/>
32    </div>
33    <div class="form-group">
34      <label for="sensorsdatum-addedFrom">Lisätty (käyttöliittymästä tai Arduinolta) </label>
35      <input class="form-control" name="sensorsdatum-lisatty" [(ngModel)]="sensorsdatum.lisatty" placeholder="UI/Arduino" readonly/>
36    </div>
37    <button class="btn btn-primary" *ngIf="!sensorsdatum._id" (click)="createNewDatum(sensorsdatum)">Lisää</button>
38    <button class="btn btn-info" *ngIf="sensorsdatum._id" (click)="updateDatum(sensorsdatum)">Muokkaa</button>
39    <button class="btn btn-danger" *ngIf="sensorsdatum._id" (click)="deleteDatum(sensorsdatum._id)">Poista</button>
40  </form>
41 </div>
```

Kuva 32 Yksityiskohdat komponentin html-koodia

Painikkeihin täytyi sijoittaa ohjelmointilogiikkaa, joten tämä tehtiin yksityiskohdat komponentin ts-tiedostossa. Tämä täytyi tehdä handler-funktioiden avulla, jotta on mahdollista muokata tietoja, joita lista näyttää. (Kuva 33)

```

10 export class SensorDatumDetailsComponent {
11
12   @Input()
13   sensordatum: SensorDatum;
14
15   @Input()
16   createHandler: Function;
17   @Input()
18   updateHandler: Function;
19   @Input()
20   deleteHandler: Function;
21
22   constructor (private sensordatumService: SensorDatumService) {}
23
24   createNewDatum(sensordatum: SensorDatum) {
25     this.sensordatumService.createNewDatum(sensordatum).then((newDatum: SensorDatum) => {
26       this.createHandler(newDatum);
27     });
28   }
29
30   updateDatum(sensordatum: SensorDatum): void {
31     this.sensordatumService.updateDatum(sensordatum).then((updatedDatum: SensorDatum) => {
32       this.updateHandler(updatedDatum);
33     });
34   }
35
36   deleteDatum(sensordatumId: String): void {
37     this.sensordatumService.deleteDatum(sensordatumId).then((deletedDatumId: String) => {
38       this.deleteHandler(deletedDatumId);
39     });
40   }
41
42 }

```

Kuva 33 Yksityiskohdat komponentin ts-koodia

Sivun latautuessa listan halutaan esittävän tiedot, joten varsinaiset funktiot, kuten "updateDatum" ovat lista komponentin ts-tiedostossa. Samaisessa tiedostossa kutsutaan ngOnInit() -funktiota, ja määritellään sovelluksen käynnistyessä se käyttämään service-tiedostoa, jotta saadaan listaan kaikki data API:lta. Lista komponentin koodi on melko pitkä, joten se löytyy kokonaisuudessaan liitteestä 4.

Käyttöliittymän kehityksessä käytin Angular CLI:n komentoa "ng serve", jolla käyttöliittymää pystyi testaamaan paikallisena. Lopuksi lisäsin käyttöliittymäsovellukseni Herokuun, jota varten täytyi luoda JS-tiedosto, jossa määritellään sovellus käynnistymään Herokun oletusporttia käyttäen (Kuva 34). Lisäksi package.json-tiedostoon täytyi määrittää samainen JS-tiedosto, jotta sovellus käynnistyy ensimmäisenä myös Herokussa.

```

const express = require('express');
const path = require('path');

const app = express();

app.use(express.static(__dirname + '/dist/angularwebapp'));

app.get('/*', function(req,res) {

  res.sendFile(path.join(__dirname+'/dist/angularwebapp/index.html'));
});

// Sovellus käynnistetään kuuntelemalla Herokun oletusporttia
app.listen(process.env.PORT || 8080);

```

Kuva 34 JS-tiedosto, jossa määritellään sovellus käyttämään Herokun oletusporttia

10.3 Lopputulos

Web-rajapinta toimi käyttöliittymäsovelluksen tekoon lähestulkoon kuten kuuluikin. Rajapinnan koodia tarvitsi muokata kuitenkin yhdeltä osalta, sillä koodissa täytyi sallia CORS, kun halusin lisätä myös käyttöliittymän Heroku-alustalle. Osoittautui, että tämä Cross-Origin Resource Sharing eli CORS:in ongelma ilmenee http-pyyntöjen vuoksi. CORS mekanismi käytännössä sallii rajoitettujen resurssien pyytämisen toisesta domainista. Tämä ongelma ei ollut ilmennyt PostMan-sovelluksella testaamisessa, eikä missään tutkimissani web-rajapintaa käsittelevissä artikkeleissa mielestäni ollut puhuttu CORS:in sallimisesta, joten tämä tuli yllätyksenä. (Codeacademy ei pvm)

Käyttöliittymän osalta lopputuloksena syntyi sensoridatan hallintaan kätevä työkalu. Listasta voidaan valita sensoridata, jota tarkastellaan. Perustoiminnot eli uuden tiedon lisääminen, tiedon muokkaaminen ja poistaminen toimivat. Muokattavia kohtia ovat mittaustiedon arvo, yksikkö, sensori ja mittauksen pvm. Käyttäjä ei voi muokata id:tä, muokkauksen päivämäärää tai lisätty-kenttää, sillä ne ovat html-koodissa määritelty readonly-tyyppiseksi. Uuden tiedon lisäyksessä mittauksen päivämäärä syntyy automaattisesti, jos kentän jättää tyhjäksi. Myös muokkauksen päivämäärä syntyy automaattisesti, kun käyttäjä painaa muokkaa nappia.

Käyttöliittymässä on parannettavaa muun muassa kenttien tarkistuksessa. Lisäksi lista voi kasvaa pitkäksi, koska siihen haetaan kaikki rajapinnan tiedot käyttäen "api/v1/data" päätepistettä. Listaa voisi parantaa esimerkiksi tekemällä jonkinlaisen lajittelu- tai hakutoiminnon, mutta tämän työn osalta käyttöliittymä on riittävä. (Kuva 35)

Sensoridata

5.5.2020 200K
09.06.2020 12:20:09 202K
09.06.2020 13:22:09 25.09C
09.06.2020 13:22:39 28.79C
09.06.2020 13:22:29 28.79C
09.06.2020 13:22:39 28.79C
09.06.2020 13:22:49 28.79C
09.06.2020 13:22:59 28.79C
09.06.2020 13:23:09 28.79C
09.06.2020 13:23:19 28.79C
09.06.2020 13:23:29 28.89C
09.06.2020 13:23:39 28.89C
09.06.2020 13:23:49 28.89C
09.06.2020 13:23:59 28.89C
09.06.2020 13:41:59 100000Pa

Uusi

Kuva 35 Valmis käyttöliittymä

Tiedot

Id

5edf62d1cc2c273ad8dfb024

Arvo

25.09

Yksikkö

C

Sensori

LM35

Mittauksen pvm

09.06.2020 13:22:09

Muokkauksen pvm

Lisätty (käyttöliittymästä tai Arduinolta)

Arduino Uno

Muokkaa

Poista

11 KÄYTETYT KEHITYSTYÖKALUT

Tässä luvussa käydään läpi työn aikana käyttämiäni kehitystyökaluja. Työn ohjelmointiosuudessa käytin Git-versionhallintaa, jota käytetään yleisesti ohjelmointiprojekteihin työelämässä. Editorina käytin Visual Studio Code:a, sillä pidän muun muassa sen koodin täydennyksestä ja ehdotuksista.

11.1 Git-versionhallinta

Versionhallinta on palvelu, joka säilöö koodia. Sen käyttö on tärkeä osa ohjelmointia monestakin syystä. Versionhallinnalla käytännössä luodaan varmuuskopio ohjelman nykyisestä versiosta, jotta voidaan tarvittaessa palata siihen, jos jokin menee pieleen uusien ominaisuuksien kehityksessä. Sen avulla nähdään ohjelman kehitys kokonaisuudessaan paremmin, sillä nähdään mitä muutoksia on tehty, milloin, ja kenen toimesta. Versioita kannattaa tallennella mieluummin liian usein kuin liian harvoin. Versionhallintatyökalut helpottavat myös koodin jakamista muille, sekä mahdollistavat muiden projekteihin osallistumisen.

Git on eräs versionhallintatyökalu, joka on todella paljon käytetty muun muassa siksi, että se toimii kaikilla eri käyttöjärjestelmillä ja käytännössä millä tahansa ohjelmointikielellä tehdyt projektit saadaan talteen. Git on itsessään komentoriviohjelma, mutta sen lisäksi on myöhemmin kehitetty muun muassa GitHub-verkkosivusto, jossa voidaan julkaista ohjelmointiprojekteja ja tarkastella niitä helpommin. Git toimii hajautettuna, eli on paikallinen versio ja palvelimella sijaitsevat versiot. Paikallinen versio on kopio, johon tehdään muutokset, ja viedään versionhallintaan ”commit” ja ”push” -komennoilla. Git:issä on mahdollista käyttää eri kehityshaaroja (engl. branch), jotta samassa projektissa voi työskennellä moni yhtä aikaa. (Ngan, 2018)

11.2 Visual Studio Code

Visual Studio Code on Microsoftin kehittämä avoimen lähdekoodin tekstieditori. Sen ominaisuuksia ovat muun muassa IntelliSense, sisäänrakennettu versionhallinta ja debuggauksen eli virheenjäljityksen tuki. IntelliSense:llä tarkoitetaan automaattista koodin täydennystä, ehdotuksia ja parametrien tietoja. Sisäänrakennettu versionhallinta mahdollistaa Git-komentojen käytön suoraan editorista, ja halutessaan voi katsoa mitä muutoksia tiedostoihin on tehty viimeiseen versioon nähden. Lisäksi Visual Studio Code on laajalti mukautettavissa ja laajennettavissa käyttäjän tarpeisiin. (Microsoft, 2020)

11.3 Heroku

Suunnitteluvaiheessa löysin alustan, jonne voin julkaista tekemäni web-rajapinnan ja web-käyttöliittymän ilmaiseksi. Heroku on pilvipalveluna toimiva alusta, jonka avulla ohjelmoijat voivat kehittää, julkaista ja seurata sovelluksia. Sen ideana on olla helppo- ja nopeakäyttöinen, jotta niin yritykset kuin harrastelijatkin voivat julkaista ja näyttää sovelluksia toisille nopeasti. Kuten monilla webhotelleilla, Heroku:lla on erihintaisia kuukausimaksuja, riippuen meneekö sovellus tuotantoon ja tarvitaanko parempaa suorituskykyä tai muita ominaisuuksia. Harrastelijoille ja omiin projekteihin se on

kuitenkin ilmainen. Heroku tarjoaa palveluina useita tunnettuja ja turvallisia tietokantoja, kuten PostgreSQL:n ja MongoDB:n. (Heroku ei pvm)

Käytin ohjelmointiprojektissani Heroku CLI komentoriviohjelmaa, joka toimii Git-versionhallinnan kanssa. Kun Heroku CLI:n lataamisen ja kirjautumisen tililleen saa tehtyä, sen komentoja voi alkaa käyttää tavallisesta komentokehotteesta.

11.4 Postman

Postman on työkalu, jolla rajapintojen toimivuutta on kätevää testata. Postman-ohjelmaa voidaan käyttää sekä oman rajapinnan kehityksessä että muiden tekemien rajapintojen käytön opettelussa. On huomattavasti helpompaa tehdä pyyntöjä Postman-ohjelman käyttöliittymän kautta, kuin kirjoittaa koodia rajapinnan toimivuuden testaamiseksi. (Farmer ei pvm)

Ohjelmassa laitetaan rajapinnan reitti osoitepalkkiin, valitaan käytettävä http-metodi ja määritetään mahdollinen sisältö pyyntöön. Pyyntö lähetetään, ja rajapinnan vastaus näytetään ohjelmassa http:n statuskoodien ja vastausajan kanssa. Postman:illa tehdyt haut jäävät historiatietoihin ja ohjelmalla pystyy tekemään myös automatisoituja testejä. (Hooda ei pvm)

12 YHTEENVETO

Opinnäytetyön tavoitteena oli tutkia web-rajapintoja ja rakentaa oma web-rajapinta sensoridatalle. Vaikka työ sisälsi paljon tutkittavaa ja tietoa täytyi hankkia paljon, onnistuin mielestäni tutkimaan ja kirjoittamaan keskeisimmistä kohdista työhön liittyen hyvin. Web-rajapinnan rakentaminen onnistui mielestäni jopa yllättävän helposti Express.js -sovelluskehityksen avulla, joten suunnittelulla ja tekniikan valinnalla voidaan sanoa olleen tärkeä osa työn onnistumista.

Aihe oli tekijän mielestä kiinnostava ja opettavainen alusta loppuun saakka. Web-rajapintojen ja etenkin avoimen RESTful-rajapinnan toiminnasta opin paljon, ja ymmärrän nyt millaisia tekijöitä rajapinnan suunnittelussa ja toteutuksessa tulee ottaa huomioon. Myös rajapinnan dokumentointi oli olennainen osa työtä, sillä opin käyttämään Swagger:iä, joka on laajasti käytössä yrityksissäkin.

Rajapinnan sensoridatan keräämiseen työssä käytettiin Arduino Uno -kehitysalustaa ja Arduino IDE:ä, jotka ovat käytännössä yksinkertaisin mahdollinen toteutustapa sulautettujen järjestelmien maailmassa. Sensoridatasta olisi saanut luotettavamman ja ammattimaisemman käyttämällä parempaa alustaa, mutta halusin saada sensoridataa yksinkertaisesti ja nopeasti rajapintaan, joten tämä ratkaisu oli hyvä työn etenemisen kannalta. Sensoridatan osuudessa tulin tutkineeksi ja käyttäneeksi MQTT-protokollaa tiedonsiirtoon.

Työn sivuprojektina halusin tehdä rajapintaa käyttävän sovelluksen, jotta pystyin varmasti toteamaan rajapinnan olevan toimiva, mikäli joku muu haluaisi tehdä sovelluksen, joka käyttää tekemääni rajapintaa. Rajapinnan koodia täytyikin muokata yhdeltä osalta sallimalla CORS eli mekanismi, jolla voidaan sallia rajoitettujen resurssien pyytäminen toisesta domainista. Tämä ongelma ei ollut kehitysvaiheessa ilmennyt Postman-ohjelmalla testatessa, eivätkä tutkimani web-rajapintaa käsittelevät artikkelit tai oppaat olleet tästä erikseen maininneet.

Ohjelmointiosuudesta erityisesti Angular -sovelluskehityksen opettelu vei aikaa, mutta lopputuloksena syntyi sensoridatan hallintaan kätevä web-käyttöliittymä. Myös opinnäytetyön kirjoittaminen vei odotettua enemmän aikaa, mutta kokonaisuudessaan lopputulokseen olen tyytyväinen.

13 LÄHDELUETTELO

- AltexSoft. *altexsoft*. 18. Kesäkuu 2019. <https://www.altexsoft.com/blog/engineering/what-is-api-definition-types-specifications-documentation/> (haettu 17. Maaliskuu 2020).
- Ambra, Jordan. *Toptal*. 2014. <https://www.toptal.com/api-developers/5-golden-rules-for-designing-a-great-web-api> (haettu 18. Maaliskuu 2020).
- Angular. *Angular*. 2020. <https://angular.io/guide/architecture-services> (haettu 8. Kesäkuu 2020).
- Arduino. *Arduino.cc*. 2020. <https://store.arduino.cc/arduino-uno-rev3> (haettu 12. Toukokuu 2020).
- Avraham, Shif Ben. *Medium*. 5. Syyskuu 2017. <https://medium.com/extend/what-is-rest-a-simple-explanation-for-beginners-part-1-introduction-b4a072f8740f> (haettu 5. Huhtikuu 2020).
- Beal, Vangie. *webopedia*. ei pvm. <https://www.webopedia.com/TERM/A/API.html> (haettu 18. Maaliskuu 2020).
- Despoudis, Theofanis. *DZone*. 9. Kesäkuu 2018. <https://dzone.com/articles/7-tips-for-building-an-api> (haettu 21. Maaliskuu 2020).
- Express. *Expressjs*. ei pvm. <https://expressjs.com/en/guide/routing.html> (haettu 22. Kesäkuu 2020).
- Farmer, Kevin. *Digitalcrafts*. ei pvm. <https://www.digitalcrafts.com/blog/student-blog-what-postman-and-why-use-it> (haettu 17. Kesäkuu 2020).
- Fielding, Roy Thomas. *ics.uci.edu*. 2000. https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm (haettu 20. Huhtikuu 2020).
- Guru99. *Guru99*. 2020. <https://www.guru99.com/mean-stack-developer.html> (haettu 9. Huhtikuu 2020).
- Haldar, Mahesh. *Hackernoon*. 25. Maaliskuu 2016. <https://hackernoon.com/restful-api-designing-guidelines-the-best-practices-60e1d954e7c9> (haettu 20. Maaliskuu 2020).
- Henderson, Michael. *Medium*. 13. Maaliskuu 2019. <https://medium.com/@michaelhenderson/what-is-nodejs-and-why-you-need-to-learn-it-f0760ba9a76a> (haettu 7. Huhtikuu 2020).
- Heroku. *Heroku*. ei pvm. <https://www.heroku.com/what> (haettu 23. Kesäkuu 2020).
- Hoffman, Chris. *How-To Geek*. 21. Maaliskuu 2018. <https://www.howtogeek.com/343877/what-is-an-api/> (haettu 10. Maaliskuu 2020).
- Hooda, Parikshit. *geeksforgeeks*. ei pvm. <https://www.geeksforgeeks.org/introduction-postman-api-development/> (haettu 22. Kesäkuu 2020).
- Kapadnis, Jay. *Medium*. 1. Maaliskuu 2018. <https://medium.com/hashmapinc/rest-good-practices-for-api-design-881439796dc9> (haettu 23. Maaliskuu 2020).
- Karanam, Ranga. *DZone*. 25. Marraskuu 2019. <https://dzone.com/articles/rest-api-what-is-hateoas> (haettu 10. Huhtikuu 2020).
- Kivekäs, Otso. *Otso Kivekäs*. 16. Kesäkuu 2014. <http://otsokivekas.fi/2014/06/avoin-rajapinta/> (haettu 13. Maaliskuu 2020).
- Korpela, Jukka. *jkorpela*. 29. Toukokuu 2013. <http://jkorpela.fi/iso8601.html> (haettu 23. Huhtikuu 2020).
- Marx, Lukas. *Malcoded*. 9. Marraskuu 2017. <https://malcoded.com/posts/angular-beginners-guide/> (haettu 6. Huhtikuu 2020).
- Microsoft. *Visual Studio Code*. 2020. <https://code.visualstudio.com/> (haettu 9. Huhtikuu 2020).
- MongoDb. *mongoDB*. 2020. <https://www.mongodb.com/> (haettu 27. Maaliskuu 2020).
- Ngan, Kayla. *Microsoft*. 5. Huhtikuu 2018. <https://docs.microsoft.com/en-us/azure/devops/learn/git/what-is-git> (haettu 9. Huhtikuu 2020).
- Patel, Priyesh. *freeCodeCamp*. 18. Huhtikuu 2018. <https://www.freecodecamp.org/news/what-exactly-is-node-js-ae36e97449f5/> (haettu 7. Huhtikuu 2020).

- Pedro, Bruno. *Hackernoon*. 7. Marraskuu 2017. <https://hackernoon.com/what-are-web-apis-c74053fa4072> (haettu 18. Maaliskuu 2020).
- Pinkham, Ryan. *Swagger*. 26. Lokakuu 2017. (haettu 11. Kesäkuu 2020).
- restfulapi*. ei pvm. <https://restfulapi.net/rest-architectural-constraints/> (haettu 9. huhtikuu 2020).
- restfulapi. *Restfulapi*. 2018. <https://restfulapi.net/>.
- Rouse, Margaret. *SearchAppArchitecture*. Helmikuu 2019.
<https://searchapparchitecture.techtarget.com/definition/SOAP-Simple-Object-Access-Protocol> (haettu 28. Maaliskuu 2020).
- Santos, Rui. *randomnerdtutorials*. 2017. <https://randomnerdtutorials.com/what-is-mqtt-and-how-it-works/> (haettu 13. Toukokuu 2020).
- SmartBear. *SmartBear*. 2. Tammikuu 2020. <https://smartbear.com/blog/test-and-monitor/soap-vs-rest-whats-the-difference/> (haettu 22. Maaliskuu 2020).
- Smartbear. *Swagger*. 2020. <https://swagger.io/docs/specification/about/> (haettu 11. Kesäkuu 2020).
- . *Swagger*. ei pvm. <https://swagger.io/docs/specification/2-0/what-is-swagger/> (haettu 23. Kesäkuu 2020).
- SmartBear. *swagger*. ei pvm. <https://swagger.io/resources/articles/documenting-apis-with-swagger/> (haettu 23. Kesäkuu 2020).
- Tampere University of Technology. *cs.tut*. Tampere University of Technology. ei pvm.
<http://www.cs.tut.fi/~seitti/2015/kalvot/http/all.html> (haettu 23. Maaliskuu 2020).
- Traffic Management Finland. *digitraffic*. ei pvm. <https://www.digitraffic.fi/rautatieliikenne/> (haettu 22. Kesäkuu 2020).
- w3schools. *w3schools*. ei pvm. https://www.w3schools.com/js/js_json_syntax.asp (haettu 3. Huhtikuu 2020).
- Violino, Bob. *InfoWorld*. 9. Maaliskuu 2018. <https://www.infoworld.com/article/3260184/how-to-choose-the-right-nosql-database.html> (haettu 27. Maaliskuu 2020).
- Wodehouse, Carey. *Upwork*. 10. Lokakuu 2016. https://www.upwork.com/hiring/development/public-apis-vs-private-apis-whats-the-difference/?utm_campaign=Submission&utm_medium=Community&utm_source=GrowthHackers.com (haettu 11. Maaliskuu 2020).
- Yang, Chuoxian. *toptal*. 2016. <https://www.toptal.com/nodejs/nodejs-frameworks-comparison> (haettu 22. Kesäkuu 2020).

LIITE 1. PALVELINOHJELMAN KOODI

```

var express = require("express");
var bodyParser = require("body-parser");
var mongodb = require("mongodb");
var ObjectID = mongodb.ObjectID;
var data_collection = "data";
var moment = require('moment-timezone');
var cors = require('cors');
var app = express();
app.use(bodyParser.json());

// Tietokanta muuttuja, jota voidaan uudelleenkäyttää yhteyteen (connection pool)
var db;

// Luodaan linkki Angularin build hakemistoon
var distDir = __dirname + "/dist/";
app.use(express.static(distDir));

// Sallitaan cors (cross-origin resource sharing)
app.use(cors());

mongodb.MongoClient.connect("mongodb://käyttäjätnus:salasana@cluster0-shard-00-00-2pav5.mongodb.net:27017, cluster0-shard-00-01-2pav5.mongodb.net:27017, cluster0-shard-00-02-2pav5.mongodb.net:27017/test?ssl=true&replicaSet=Cluster0-shard-0&authSource=admin" ||
"mongodb://localhost:27017/test", function (err, client) {
  if (err) {
    console.log(err);
    process.exit(1);
  }

  db = client.db();
  console.log("Tietokanta yhdistetty");

  var server = app.listen(process.env.PORT || 8080, function () {
    var port = server.address().port;
    console.log("Sovellus käynnissä portissa ", port)
  });
});

// virheenkäsittely

```

```

function handleError(res, reason, message, code) {
  console.log("ERROR: " + reason);
  res.status(code || 500).json({ error: message });
}

// API:N REITIT
// GET -- rajapinnan dataa sivulta X palauttava reitti, limit 25
app.get("/api/v1/data/page/:pagenumber", function (req, res) {
  var skips = 25 * req.params.pagenumber - 25;
  db.collection(data_collection).find({}).skip(skips).limit(25).toArray(function (err, docs) {
    if (err) {
      handleError(res, err.message, "Datan hakeminen epäonnistui.");
    } else {
      res.status(200).json(docs);
    }
  });
});

// GET -- yhden tietueen hakeminen ID:llä
app.get("/api/v1/data/:id", function (req, res) {
  db.collection(data_collection).findOne({ _id: new ObjectID(req.params.id) }, function (err, doc) {
    if (err) {
      handleError(res, err.message, "Tietoa ei löytynyt tällä ID:llä.");
    }
    else {
      res.status(200).json(doc);
    }
  });
});

// GET -- rajapinnan kaiken datan palauttava reitti
app.get("/api/v1/data/", function (req, res) {
  db.collection(data_collection).find({}).toArray(function (err, docs) {
    if (err) {
      handleError(res, err.message, "Datan hakeminen epäonnistui.");
    }
    else {
      res.status(200).json(docs);
    }
  });
});

// POST -- uuden tietueen luonti

```

```

app.post("/api/v1/data", function (req, res) {
  var newData = req.body;
  newData.lisatty = "UI";
  if (!req.body.arvo) {
    handleError(res, "Virhe tiedon syöttämisessä", "Lisää 'arvo:', 400);
  } else {
    if (!req.body.mitattu) { // jos mittaukselle ei anneta päivämäärää, otetaan tämänhetkinen aika
      automaattisesti
      var now = new Date();
      newData.mitattu = now;
    }
    db.collection(data_collection).insertOne(newData, function (err, doc) {
      if (err) {
        handleError(res, err.message, "Tiedon luonti epäonnistui.");
      } else {
        res.status(201).json(doc.ops[0]);
      }
    });
  }
});

// PUT -- yhden tietueen muokkaaminen
app.put("/api/v1/data/:id", function (req, res) {
  var updateDoc = req.body;
  delete updateDoc._id; // poistetaan pyynnön body:stä id
  var now = new Date();
  updateDoc.muokattu = now;

  db.collection(data_collection).updateOne({ _id: new ObjectId(req.params.id) }, { $set: updateDoc }, function (err, doc) {
    if (err) {
      handleError(res, err.message, "Tiedon päivittäminen epäonnistui.");
    }
    else {
      updateDoc._id = req.params.id;
      res.status(200).json(updateDoc);
    }
  });
});

// DELETE -- yhden tietueen poistaminen
app.delete("/api/v1/data/:id", function (req, res) {

```



```
db.collection(data_collection).deleteOne({ _id: new ObjectID(req.params.id) }, function (err, result) {  
  if (err) {  
    handleError(res, err.message, "Tiedon poistaminen epäonnistui");  
  }  
  else {  
    res.status(200).json(req.params.id);  
  }  
})  
});
```

LIITE 2. ARDUINO-OHJELMAN KOODI

```

#include <SPI.h>
#include <Ethernet.h>
#include <PubSubClient.h>

int sensorPin=5;
unsigned long lastTime;
char buffer[32];

// Arduinon MAC- ja IP-osoite
byte mac[] = { 0xDE, 0xAD, 0xBE, 0xEF, 0xFE, 0xED };
IPAddress ip(192, 168, 100, 12);

// MQTT-välittäjän IP-osoite
byte server[] = { 192, 168, 100, 13 };

// Tarvitaan yhteyden muodostamiseen
EthernetClient ethClient;
PubSubClient mqttClient(ethClient);

void setup()
{
  // 1,1V referenssijännite
  analogReference(INTERNAL);
  Serial.begin(9600);

  // Aloitetaan Ethernet-yhteys
  Ethernet.begin(mac, ip);
  delay(3000);

  // Määritetään MQTT-välittäjä
  mqttClient.setServer(server, 1883);

  // Yritetään yhdistää MQTT-välittäjään ID:llä "myClientID"
  if (mqttClient.connect("myClientID"))
  {
    Serial.println("Yhteys on avattu onnistuneesti");
  }
  else
  {
    Serial.println("Yhteyden avaus epäonnistui");
  }
}

```

```

    }
}

void loop()
{
    mqttClient.loop();

    // Luetaan LM35-sensorilta arvo
    int reading=analogRead(sensorPin);

    // Muunnos celsiusasteiksi
    float temperature = reading / 9.31;

    // Lisätään arvo char array -muuttujaan
    char fValue[16];
    dtostrf(temperature, 3, 2, fValue);

    // Jos aikaa kuluu tarpeeksi..
    if(millis()>(lastTime+1000)) {
        // Lisätään arvo viestinä lähetettävään bufferiin
        snprintf(buffer, sizeof(buffer), "%s", fValue);
        Serial.println(buffer);

        // Julkaistaan viesti "arduino_uno/sensoridata/lampotila/" aiheeseen
        mqttClient.publish("arduino_uno/sensoridata/lampotila/", buffer);
        lastTime=millis();
    }

    // 60000ms = 60s väli tiedon lisäyksessä
    delay(60000);
}

```

LIITE 3. NODE.JS-OHJELMAN KOODI

```

var mqtt = require('mqtt');
var mongodb = require('mongodb');
var ObjectID = mongodb.ObjectID;
var collection, client;

// alustetaan mongodb-client
var MongoClient = mongodb.MongoClient;
// tietokannan URI-osoite
var mongodbURI = 'mongodb://käyttäjätunnus:salasana@cluster0-shard-00-00-2pav5.mongodb.net:27017, cluster0-shard-00-01-2pav5.mongodb.net:27017, cluster0-shard-00-02-2pav5.mongodb.net:27017/test?ssl=true&replicaSet=Cluster0-shard-0&authSource=admin';
// aihe, josta viestejä tilataan
var topicName = "arduino_uno/sensoridata/lampotila/";
// yhdistetään tietokantaan kokoelman (collection) kanssa
MongoClient.connect(mongodbURI, setupCollection);

function setupCollection(err, db) {
    if (err) throw err;

    // kokoelman nimi tietokannassa
    collection = db.collection("data");
    // yhdistetään MQTT-välittäjään, host ja port oltava samat kuin Arduinon ohjelmassa
    client = mqtt.connect({ host: 'localhost', port: 1883 });
    // tilataan aihe
    client.subscribe(topicName + "+");
    // viestin tultua suoritetaan funktio
    client.on('message', insertEvent);

    console.log("MQTT-välittäjä on yhdistetty tietokantaan.");
}

// funktio, jossa määritellään mitä mongodb-tietokantaan menee
function insertEvent(topic, message) {

    // lämpötilatieto tulee viestissä, muunnetaan se string-tyyppiseksi
    var messageBuffer = Buffer.from(message, 'base64');
    var messageString = messageBuffer.toString();
    console.log(messageString);
    var now = new Date();

```

```
// kokoelmaan lisätään tiedot
collection.insertOne({
  _id: new ObjectID(),
  arvo: messageString,
  yksikko: "C",
  sensori: "LM35",
  mitattu: now,
  muokattu: null,
  lisatty: "Arduino Uno"
},
function (err, docs) {
  if (err) {
    console.log("Lisääminen epäonnistui - " + err.message);
  }
});
console.log("Lisätty tietokantaan.");
}
```

LIITE 4. LISTA-KOMPONENTIN TS-KOODI

```

import { Component, OnInit } from '@angular/core';
import { SensorDatum } from '../sensor-datum';
import { SensorDatumService } from '../sensor-datum.service';
import { SensorDatumDetailsComponent } from '../sensor-datum-details/sensor-datum-details.component';

var moment = require('moment-timezone');

@Component({
  selector: 'sensor-datum-list',
  templateUrl: '../sensor-datum-list.component.html',
  styleUrls: ['../sensor-datum-list.component.css'],
  providers: [SensorDatumService]
})

export class SensorDatumListComponent implements OnInit {
  sensorDataArr: SensorDatum[]
  selectedDatum: SensorDatum

  constructor(private datumService: SensorDatumService) { }

  public ngOnInit() {
    this.datumService
      .getData()
      .then((sensorDataArr: SensorDatum[]) => {
        this.sensorDataArr = sensorDataArr.map((sensordatum) => {

          sensordatum.mitattu = moment(sensordatum.mitattu).tz("Europe/Helsinki").format("DD.MM.YYYY HH:mm:ss"); // Moment-timezone-kirjastolla formatoidaan päivämäärät esitettäväksi Suomen esitystavassa.

          if(sensordatum.muokattu != null){
            sensordatum.muokattu = moment(sensordatum.muokattu).tz("Europe/Helsinki").format("DD.MM.YYYY HH:mm:ss");
          }

          return sensordatum;
        });
      });
  }
}

```

```

private getIndex = (datumId: String) => {
  return this.sensorDataArr.findIndex((sensordatum) => {
    return sensordatum._id === datumId;
  });
}

```

```

selectDatum(sensordatum: SensorDatum) {
  this.selectedDatum = sensordatum;
}

```

```

createNewDatum() {
  var sensordatum: SensorDatum = {
    arvo: "",
    yksikko: null,
    sensori: null,
    mitattu: null,
    muokattu: null,
    lisatty: null
  };

  // Uusi, luotu data valitaan listalta oletuksena
  this.selectDatum(sensordatum);
}

```

```

deleteDatum = (datumId: String) => {
  var idx = this.getIndex(datumId);
  if (idx !== -1) {
    this.sensorDataArr.splice(idx, 1);
    this.selectDatum(null);
  }
  return this.sensorDataArr;
}

```

```

addDatum = (sensordatum: SensorDatum) => {
  this.sensorDataArr.push(sensordatum);
  this.selectDatum(sensordatum);
  return this.sensorDataArr;
}

```

```

updateDatum = (sensordatum: SensorDatum) => {
  var idx = this.getIndex(sensordatum._id);

```

```
if (idx !== -1) {  
  this.sensorDataArr[idx] = sensordatum;  
  this.selectDatum(sensordatum);  
}  
return this.sensorDataArr;  
}  
}
```