Osaamista
ja oivallusta
tulevaisuuden
tekemiseen

Arttu Kiiskinen

REST-rajapinta Node.js-ympäristössä

Metropolia Ammattikorkeakoulu

Insinööri (AMK)

Tieto- ja viestintätekniikan tutkinto-ohjelma

Insinöörityö

24.4.2019

Tekijä Otsikko Sivumäärä Aika	Arttu Kiiskinen REST-rajapinta Node.js-ympäristössä 33 sivua 16.4.2019
Tutkinto	insinööri (AMK)
Tutkinto-ohjelma	Tieto- ja viestintätekniikan tutkinto-ohjelma
Ammatillinen pääaine	Ohjelmistotuotanto
Ohjaajat	Yliopettaja Auvo Häkkinen Tiimivetäjä Oskari Larjamo

Tässä insinöörityössä tarkastellaan REST-rajapintoja ja niiden toteuttamista verkkoon Node.js-ympäristössä. Aihetta käsitellään Helsingin kaupungille toteutetun web-rajapinnan kautta. Projektissa selvitetään miten REST-rajapinnat rakentuvat verkkoympäristössä ja miten Node.js toimii rajapinnan alustana, mitä lisäosia tai työkaluja se tarvitsee ja millaisia niiden eri vaihtoehdot ovat.

Työssä esitellään myös Helsingin kaupunkiympäristölle tehty Node.js-pohjainen laite- ja ohjelmistotarpeiden hallintajärjestelmä. Sitä tarkastellessa katsottiin sen tärkeimpiä osia, kuten toimintaympäristöä, itse rajapintaa ja sen lisäosia

Moderni ohjelmistokehitys on siirtynyt yhä enenevissä määrin web-ympäristöön, ja sen myötä työkaluja on tullut valtavasti lisää, ja niiden käyttö on helpompaa ja monipuolisempaa kuin koskaan. Node.js toimii hyvin alustana web-rajapintojen rakentamiselle, ja tarjoaa monipuoliset työkalut REST-mallin toteuttamiselle, kuten esitellystä projektista käy ilmi.

Avainsanat	Node.js, javascript, express, api, web-ohjelmointi

Author Title Number of Pages Date	Arttu Kiiskinen REST interface with Node.js runtime 33 pages 16 April 2019
Degree	Bachelor of Engineering
Degree Programme	Software engineering
Professional Major	Software production
Instructors	Auvo Häkkinen, Principal Lecturer Oskari Larjamo, Team Manager

In this thesis we are looking at REST-interfaces and how you should go about making one in web environment with Node.js. The topic is examined by scrutinizing interface made for web application ordered by city of Helsinki. In this project we found out how REST-interfaces are built and what options you have when you want to build one. We also considered what tools you might need if you want to add special features or expand its purpose.

We also look at a project that was made for city of Helsinki. The project was a software- and hardware requirement system and it was made with Node.js as its base. We examined its most important parts like operating environment, the API and its extensions.

Modern software development has more and more moved to web environment and with that, the library of tools have expanded rapidly and using those tools is easier and versatile than ever. Node.js is a great platform for this and provides excellent tools to make REST-based web-interfaces as we see in project that is presented in this thesis.

Keywords	Node.js, javascript, express, api, web-programming

Sisällys

Lyhenteet

1 Johdanto 1

2 REST-rajapinta 2

2.1 Historiaa 2
2.2 REST-rajapinnan rajoitteet 3
2.2.1 Asiakas-palvelin-arkkitehtuuri 4
2.2.2 Yhtenäinen rajapinta 5
2.2.3 Kerrostettu 6
2.2.4 Tilattomuus 6
2.2.5 Code on Demand 7
2.2.6 Välimuistin hyödyntäminen 8

3 Node.js-ympäristö 9

3.1 Perustiedot 10
3.2 Historia 10
3.3 Toiminta 11
3.4 Moduulit ja NPM-pakkaustenhallinta 12
3.5 Node.js-rajapintakehykset 13

4 Node.js-pohjainen laite- ja ohjelmistotarpeiden hallintajärjestelmä 15

4.1 Projektin tavoite 16
4.2 Toimintaympäristö 17
4.3 Tietokanta ja muut resurssit 18
4.4 Sovelluksen REST-rajapinta 19
4.5 Rajapintaa tukevat kirjastot 22
4.5.1 Autentikointi 22
4.5.2 Dokumentointi 23
4.5.3 Tapahtumaloki 24
4.6 Testaaminen 26
4.7 Projektin tulos 28

5 Yhteenveto 30

Lähteet 32

Lyhenteet

API	Application Programming Interface. Määrittely, jonka perusteella eri ohjelmat tai niiden osat voivat keskustella keskenään.
HTML	Hypertext Markuup Language. Verkossa käytettävä standardoitu kuvauskieli, jolla kuvataan hyperlinkkejä sisältävää kuvatekstiä.
HTTP	Hypertext Transfer Protocol. Protokolla jota käytetään verkossa tiedonsiirtoon.
MSSQL	Microsoft SQL Server. Microsoftin kehittämä SQL-pohjainen tietokantajärjestelmä, johon tallennetaan ja josta haetaan dataa.
Node.js	Asynkroninen Javascript-ympäristö, joka on suunniteltu web-sovellusten rakentamiseen.
REST	Representional State Transfer. Arkkitehtuurityyli, joka ohjaa web-kehitystä hyödyntämällä tilattomia operaatioita.
RPC	Remote Procedure Call. Sovellusarkkitehtuuri, jonka tavoitteena on abstraktoida etäkutsut lokaaleiksi.
SOAP	Simple Object Access Protocol. Tapa rakentaa rajapintoja.
SQL	Structured Query Language. Kieli, jota käytetään relaatiotietokantojen hallinnassa.
web	Maailmanlaajuinen tietojärjestelmä, joka koostuu komponenteista, jotka kommunikoivat keskenään.
XML	Extensible Markup Language. Merkintäkieli, joka määrittelee säännöt dokumenttien muuntamisesta koodikielelle.

1 Johdanto

Viime vuosina web-kehityksen tärkeys on kasvanut huomattavasti yhä useampien palveluiden siirryttyä verkkoon. Varsinkin web-sovellusten suosio on kasvanut, koska ne mahdollistavat monipuolisempien sovellusten tarjoamisen suoraan selaimen välityksellä staattisempiin verkkosivuihin nähden. Näiden sovellusten rakentamiseksi tarvitaan myös moderneja arkkitehtuurimalleja sekä työkaluja.

Arkkitehtuurien puolesta REST (Representational State Transfer) on ollut kovassa nosteessa lukuisista sen tarjoamista hyödyistä johtuen, kuten tilattomuudesta ja palvelimen ja käyttöliittymän erottelusta, joka mahdollistaa muutokset toisessa päässä ilman, että toinen hajoaa. Siitä, mitä REST-konseptina varsinaisesti sisältää on, jonkin verran keskustelua, joten sen sisältöä tullaan myös tarkastelemaan ja katsotaan, millaisia rajoituksia sillä on. Tarkastellaan myös, mitä vaihtoehtoisia toteutusmalleja moderneille web-sovelluksille on olemassa ja mitkä näiden heikkoudet ja vahvuudet ovat. Ehkä myös suositussa REST-tyylissä on jotain parantamisen varaa.

Kun on päätetty, mitä arkkitehtuuria käytetään, tarvitaan vain työkalu, jolla se toteutetaan. ”JavaScript on erittäin suosittu kieli kiitos sen läsnäolon verkkoselaimissa, se myös vertailee hyvin muita kieliä vastaan ja sisältää monia moderneja konsepteja” [Herron, 2013]. JavaScript ei yksinään kuitenkaan riitä, joten tässä vaiheessa Node.js astuu kuvaan. Se toimii sovelluksen REST-rajapinnan palvelimena sekä yleisenä alustana sen kehitykselle. Node.js on REST-mallin tapaan kasvattanut suosiotaan viimevuosina mm. nopeutensa ja juurikin JavaScriptiin pohjautumisensa ansiosta. Näin sekä käyttöliittymässä että palvelinpäässä käytetään yhtä ja samaa kieltä. Se on valtava etu verkkosovelluksien kehityksessä. Tämä muun muassa siksi, koska JavaScriptiä käytetään jo huomattavasti verkossa valmiiksi, jolloin taitavat JavaScript-koodaajat pääsevät hyödyntämään osaamistaan myös palvelinpuolella. JavaScriptin suosion vuoksi myös sovelluksen tekoon myöhemmin mukaan tulleet ymmärtävät koodia todennäköisemmin. Kolmas tärkeä seikka on koodin uudelleenkäyttö. Palvelimen ja käyttöliittymän ollessa samaa kieltä voidaan niiden välistä koodia jakaa ja käyttää uudelleen.

Tässä työssä on tehty REST-rajapinta sovellus- ja laitetarpeiden hallintasovellusta varten käyttäen Node.js-ympäristöä. Työssä tarkastellaan, miten rajapinta rakennetaan Node.js:lla ja mitä moduuleita tai kirjastoja siinä voidaan hyödyntää. Lisäksi työssä tutustutaan myös tarkemmin sekä Node.js:ään että REST-rajapintaan ja pohditaan, mitkä seikat tekevät niistä modernin web-kehityksen kulmakiviä.

2 REST-rajapinta

REST ei ole varsinaisesti tiukka arkkitehtuurimalli, vaan ennemminkin kokoelma rajoitteita, joita seuraten ohjelma rakennetaan. Arkkitehtuurimallit tässä yhteydessä tarkoittavat sellaisia malleja, jotka vaikuttavat sovelluksen rakenteeseen ja toimintaan suoraan ja tarkemmin kuin laveammat REST-rajoitteet. REST-rajoitteet määrittävät, millaisia asioita sovelluksen tulisi sisältää, mutteivat kerro, kuinka ne kuuluisi toteuttaa. Näitä rajoitteita seuraamalla REST-mallin mukaan saadaan tehtyä järjestelmiä, jotka ovat skaalautuvia, hyödyllisiä, helppokäyttöisiä ja niissä voidaan hyödyntää dataa monesta eri lähteestä. ”Tämä on siis tietenkin vain teoriaa, mutta on huomattu, että REST-periaatteita noudatettaessa päästään tuloksiin, jossa toteutukset ovat vähemmän riippuvaisia toisistaan” [Wilde & Pautasso, 2011].

Aliluvuissa käydään REST-mallin historiaa ja aikaisemmin mainittuja rajoitteita, jotka määrittävät sen rajat. Katsotaan myös, mitä mahdollisia hyötyjä niistä on web-sovelluksissa

2.1 Historiaa

REST-mallin isänä pidetään Roy Fieldingiä, joka oli myös yksi HTTP-protokollan (Hypertext Transfer Protocol) pääsuunnittelijoista. Fielding oli mukana myös Apache-palvelimen luomisessa. Vaikka nämä ovatkin erittäin merkittäviä suorituksia IT-kehityksen alalla, tunnetaan hänet kuitenkin nykyään parhaiten hänen vuonna 2000 julkaisemastaan väitöskirjasta *Architectural Styles and the Design of Network-based Software Architecture* [Morris, 2010]. Tässä kirjassa hän määritteli termin REST ja määritteli sen rajoit-

teet. REST on siis jo käsitteenä melkein 20 vuotta vanha. Sen kehitys lähti käyntiin aikanaan, kun Fielding työskenteli HTTP-standardisoinnin parissa ja halusi parantaa web-kehitystä. Myös Fieldingin näkemät SOAP-protokollan (Simple Object Acces Protocol) ongelmat olivat este web-kehityksen edistykselle.

SOAP on tietoliikenneprotokolla, joka sisältää REST:n tapaan sisältää ajatuksen siitä, kuinka web-palveluita tulisi luoda. Sekään ei kuitenkaan ole ensimmäinen ratkaisu ongelmaan. Microsoft kehitti sen alun perin korvaamaan muita vanhempia teknologioita, jotka eivät toimineet kovin hyvin web-ympäristössä. Ilmestymisensä jälkeen siitä tuli erittäin suosittu esimerkiksi kieli- ja alustavapauden ansiosta. Se on myös standardisoitu ja sisältää sisäänrakennetun virheenhallinnan. SOAP on myös täysin XML (Extensible Markup Language) -pohjainen, joka tekee dokumenteista kone- ja ihmisluettavia.

SOAP:lla oli suuri vaikutus REST:n syntymiseen. Kuten aikaisemmin mainittiin, motivaatio sen kehittämiselle alun perin syntyi, kun SOAP-malli alkoi saavuttaa laajempaa käyttöä. SOAP oli Fieldingin mielestä monilla tavoin rajoittunut ja monimutkainen. Hän halusi, että olisi olemassa jokin malli, jonka avulla HTTP-protokollasta, jota hänkin oli mukana kehittämässä, voisi ottaa kaiken irti. REST ei kuitenkaan julkaisunsa jälkeen saavuttanut välitöntä suosiota, vaan se enemmänkin jakoi mallien puolestapuhujat eri leireihin. SOAP:n kannattajat moittivat REST:iä liian yksinkertaistetuksi ja REST:in kannattajat SOAP:ia vanhanaikaiseksi ja kankeaksi. Tämä jako jatkui varsin pitkään, ja vasta viime vuosina REST on alkanut saavuttaa laajempaa suosiota uusien alustojen ja web-palveluiden jatkuvan lisääntymisen ansiosta.

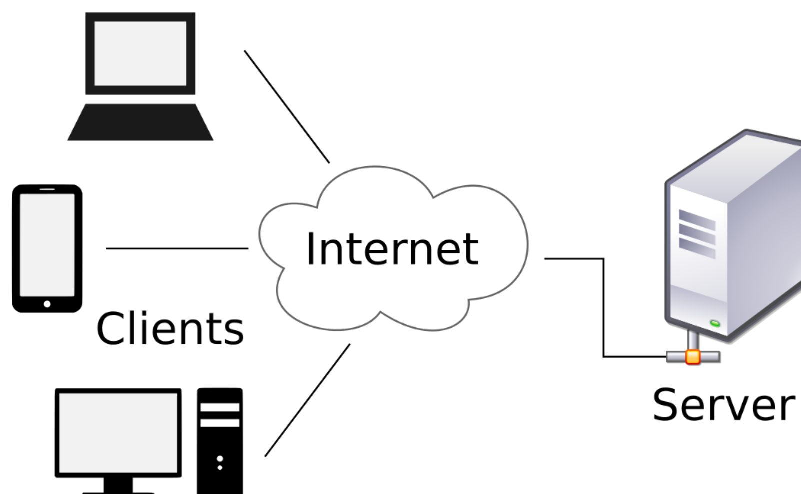
2.2 REST-rajapinnan rajoitteet

Alkuperäiset REST-rajapinnan rajoitteet on määritelty Fieldingin väitöskirjassa [Fielding, 2000: 76-106]. Mutta myöhemmin, varsinkin kun REST saavutti laajempaa suosiota, ovat muut laajentaneet rajoituksia omillaan. Aiheesta on siis tehty paljon tutkimusta Fieldingin jälkeenkin ja osaa näistä tutkimusteoksista lainataan tässäkin työssä. Tämän seurauksena REST:in määritelmästä on jonkin verran kiistaa, ja se tarkoittaa eri ihmisille eri

asioita. Yksinkertaisuuden vuoksi ainakin rajoitteista puhuttaessa pidättäydytään Fieldingin kirjassa esitetyissä kategorioissa. Tarkastellaan, mitä nämä rajoitteet ovat ja miten ne parantavat järjestelmiä, joihin ne toteutetaan.

2.2.1 Asiakas-palvelin-arkkitehtuuri

Rest-rajapinnan ensimmäinen rajoite on asiakas-palvelin-arkkitehtuuri. Se tarkoittaa lyhykäisyydessään sitä, että palvelin ja asiakas erotetaan toisistaan. Tällöin käyttöliittymää ja palvelinta voidaan kehittää itsenäisesti toisistaan, ja käyttöliittymä on helpompi siirtää eri alustoille palvelimen pysyessä staattisena. Tämä erottelu on yksi REST-mallin kulmakivistä.



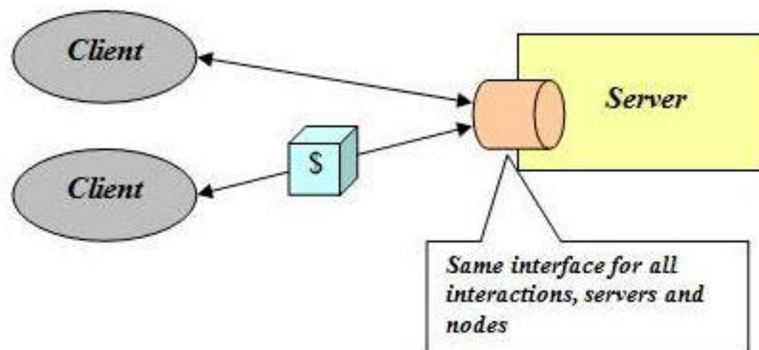
Kuva 1. Asiakas-palvelin arkkitehtuuri (Wikipedia, 2019)

Kuva 1 esittää asiakas-palvelin-arkkitehtuuria, jossa oikealla oleva palvelin siis tarjoilee dataa rajapinnan kautta asiakaslaitteille. Näin vältytään sovelluksen asentamiselta ja rajapinta ja käyttöliittymä voidaan näin pitää erotettuna.

2.2.2 Yhtenäinen rajapinta

Käyttöliittymän puolesta REST painottaa yhtenäisyyttä järjestelmän eri osien välillä. Tällä jälleen haetaan yksinkertaisuutta ja halutaan helpottaa eri osien itsenäistä kehitystä. Terminä yhtenäinen rajapinta on vielä ehkä hieman laaja. Se yleensä jaetaan vielä neljään alakategoriaan, jotka ovat resurssien tunnistaminen, resurssien manipulointi representaation avulla, itseään kuvaavat viestit ja HATOAS-periaate (Hypermedia As The Engine Of Application State). HATEOAS-periaatteen tavoitteena on käyttää hypermedialinkkejä ohjelmien eri osien välillä liikkumiseen aina, kun sovelluksessa kohdataan informaatiota, joka on relevanttia toisen osan kannalta.

Iso osa yhtenäisen rajapinnan tarpeista saadaan jo siis tyydytettyä sillä, että HTTP-kutsuja tehdessä niille annetaan standardin mukaiset metodien nimet riippuen niiden tarjoamasta toiminnallisuudesta. Nämä ovat siis POST, GET, PUT, PATCH ja DELETE. Ne vastaavat post-, read-, update- ja delete-operaatioita.

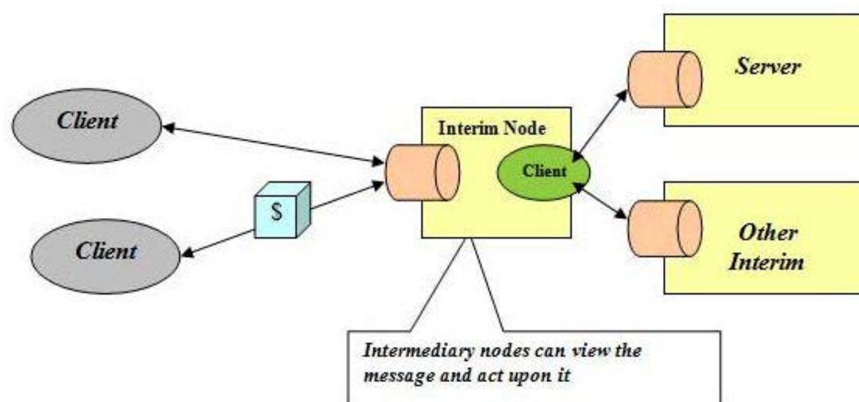


Kuva 2. Yhtenäinen rajapinta (Patil, 2013).

Kuvassa 2 on koitettu hahmottaa yhteisen rajapinnan käsitettä. Siinä rajapinta toimii palvelimen ja asiakkaan välissä ja tarjoilee asiakkaille nämä yhtenäiset operaatiot.

2.2.3 Kerrostettu

Kerrostetuissa järjestelmissä arkkitehtuuri järjestetään hierarkkisiin kerroksiin, jossa osat tietävät ainoastaan niiden osien toiminnasta, minkä kanssa ne ovat suoraan kosketuksissa. Näin yksittäisten osien toiminallisuutta saadaan yksinkertaistettua. Tästä hyötyvät erityisesti erittäin monimutkaiset järjestelmät, jotka kasvavat jatkuvasti. Pienille järjestelmille tällaisesta kerrostamisesta saattaa kuitenkin aiheutua ylimääräistä vaivaa sekä datan kulun hidastumista sen kulkiessa monien kerrosten läpi.



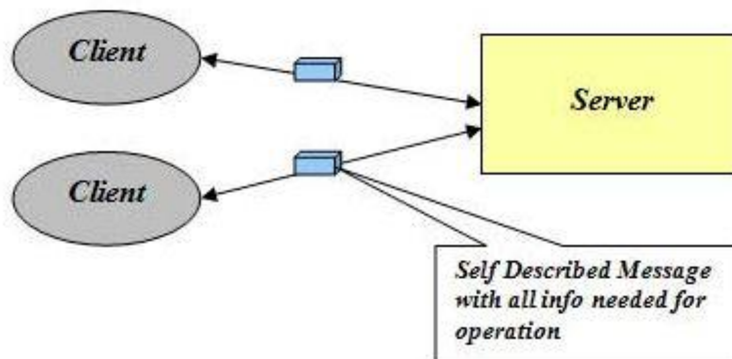
Kuva 3. Kerrostettu järjestelmä (Patil, 2013).

Kuvassa 3 on esitys kerrostetusta järjestelmästä. Tässä palvelimen ja asiakkaan välissä on välikappale, joka osaa reagoida sen lähettämään viestiin ja toimia sen mukaisesti, muttei tiedä muuta sen toiminnasta. Palvelimen alla näkyvä toinen erillinen välikappale tarjoaa ensimmäiselle välikappaleelle jotain toista toiminnallisuutta, josta palvelin ei taas tiedä mitään.

2.2.4 Tilattomuus

Tilattomuudella tässä yhteydessä tarkoitetaan sitä, että palvelin, joka vastaanottaa käyttöliittymän kutsut, ei sisällä mitään dataa siitä, missä tilassa se on. Tämä aiheuttaa sen,

että kaikki kutsut, jotka lähetetään sisältävät itsessään kaiken tarvittavan tiedon, eli kutsut pitää ymmärtää ilman kontekstia. Tämän avulla päästään eroon käyttöliittymän tilan sisältävästä datasta, joka muuten kuormittaisi palvelinta. Tämän takia tilattomuus on erityisen hyvä ratkaisu, jos järjestelmään haluaa jättää mahdollisuuden ylöspäin skaalaamiselle.



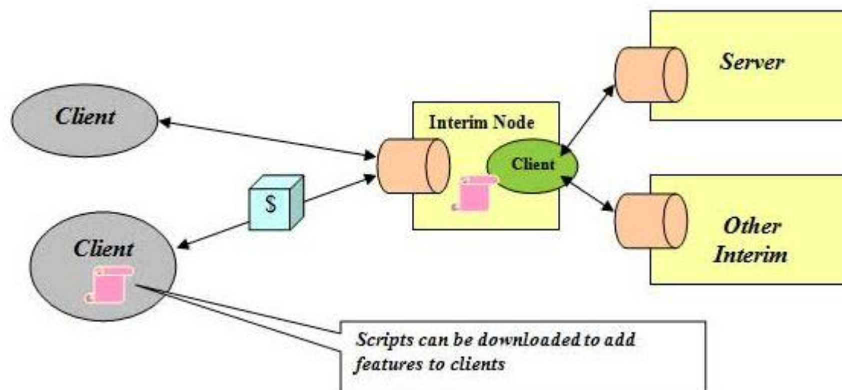
Kuva 4. Tilaton järjestelmä (Patil, 2013).

Kuvassa 4 on esitys aikaisemmassa luvussa kuvatusta tilattomasta järjestelmästä. Palvelin siis lähettää kaiken asiakkaalle tarpeellisen tiedon sinisissä laatikoissa olevissa viesteissä, jolloin sen ei tarvitse jäädä mihinkään tiettyyn tilaan odottamaan vastausta asiakkaalta.

2.2.5 Code on Demand

Joskus on hyödyllistä, että käyttöliittymä pystyy itse suorittamaan koodia skriptien avulla. Näin sen käytettävyyttä tai ominaisuuksia voidaan lisätä pitäen samalla sen pohjan yksinkertaisena. Code on Demand tunnetaan myös nimellä client side scripting ja sitä käytetään paljon myös REST-mallin ulkopuolella. Uusien ominaisuuksien lisääminen jälkikäteen mahdollistaa myös järjestelmän eliniän pidentämisen. Toisaalta se saattaa myös heikentää näkyvyyttä ohjelman sisällä, jonka takia sitä pidetään vapaaehtoisena REST-

mallin kannalta. Seuraavassa kuvassa havainnollistetaan hieman Code on Demand-käsitettä.



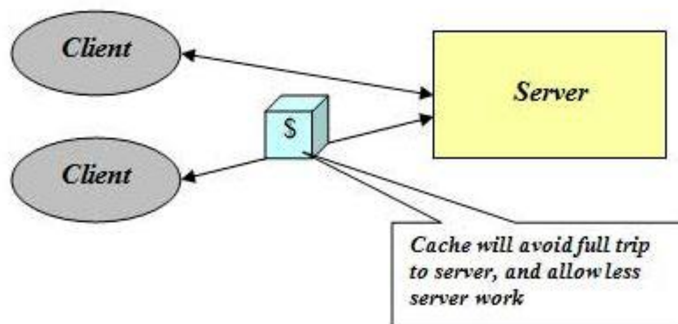
Kuva 5. Code on Demand (Patil, 2013).

Kuvassa 5 on esimerkki koodin ajamisesta asiakkaan laitteessa. Tässä tapauksessa asiakkaan laite on ladannut palvelimen päädystä skriptin, jota se tämän jälkeen voi suorittaa omalla laitteellaan jonkin toiminnon toteuttamiseksi.

2.2.6 Välimuistin hyödyntäminen

Modernit web-selaimet sisältävät mahdollisuuden käyttää asiakaspään laitteen muistia selaimessa toimivan järjestelmän tarkoituksiin. REST-malli vaatii, että aina, kun asiakas tekee pyynnön palvelimelle, pitää vastaus tarvittaessa olla mahdollista tallentaa tähän välimuistiin. Kutsuihin on siis määriteltävä, saadaanko sen vastaus tallentaa välimuistiin myöhempää käyttöä varten. Tästä on selvänä hyötynä se, että välimuistissa olevaa dataa voidaan hyödyntää järjestelmän toiminnassa ilman uusien kutsujen tekemistä ja näin vähentää palvelimen kuormitusta. Järjestelmän käyttö nopeutuu myös asiakkaan päädyssä, kun mahdollisesti suuria datamääriä ei aina tarvitse pyytää palvelimelta uudestaan.

Välimuistin käytössä piilee kuitenkin myös mahdollinen ongelma. Jos palvelinpäästä saatava data muuttuu paljon ja usein, voi tämän datan välimuistiin tallentaminen heikentää datan luotettavuutta, sillä välimuistissa oleva data ei välttämättä enää vastaakaan palvelimella olevaa uusinta tietoa. Kuva 6 pyrkii havainnollistamaan välimuistin käyttöä sovelluksissa.



Kuva 6. Välimuistin hyödyntäminen (Patil, 2013).

Kuvassa 6 näkyvä sininen laatikko symboloi väliaikaisia tiedostoja, jotka on talletettu asiakkaan laitteeseen. Tällöin sen ei tarvitse jokaisella kerralla hakea tietoa uudestaan palvelimelta, vaan se voi ladata sen laitteella olevasta sijainnista.

3 Node.js-ympäristö

Tässä luvussa tutustumaan myöhemmin käsiteltävän rajapinnan pohjana toimivaan Node.js-ympäristöön. Katsotaan aluksi sen historiaa ja motivaatioita sen luomisen taustalla. Toisena katsomme sen perustietoja sekä hieman sen toimintaa, joka tekee siitä uniikin. Moduulit ja NPM (Node Package Manager) ovat aina olleet Node.js:n kulmakiviä. Näistä moduuleista erityisesti myöhemmin käsiteltävän projektin kannalta tärkeimpiä ovat rajapintakehykset. Niiden erilaisia ratkaisuvaihtoehtoja tarkastellaan tämän luvun viimeisessä luvussa ja sen aliluvuissa.

3.1 Perustiedot

Nyt kun on tarkasteltu, mitä rajoitteita REST käsittää, tutustutaan Node.js-ympäristöön, jossa rajapinnan toteutus tehdään. Kuten aikaisemmin puhuimme, JavaScriptin merkitys verkossa on kasvanut 2000-luvulla merkittävästi. Tähän yhdistettynä myös web-palveluiden nopea kasvu ovat luoneet tarpeen JavaScript-pohjaiselle kehitysympäristölle ja Node.js tarjoaa siihen ratkaisun.

Node.js-ympäristö on rakennettu JavaScript-moottori v8:n päälle, jota myös Googlen Chrome-selain käyttää. Se on suunniteltu erityisesti vaativia I/O-applikaatioita silmällä pitäen hyödyntäen tapahtumapohjaista arkkitehtuuria. Suurin osa funktioista tehdään myös asynkronisesti, vaikkakin mahdollisuudet synkronisten funktioiden tekemiseen löytyy. Kaikki sovellukset Node.js:sa kirjoitetaan JavaScriptillä, mikä kuten aikamaisemmin mainitsimmekin, on läsnä verkossa kaikkialla. Näin pystytään tekemään sekä käyttöliittymä että palvelin samalla kielellä, joka on vakiintunutta teknologiaa web-kehityksen alalla.

3.2 Historia

Ryan Dahl julkisti Node.js:n alun perin vuonna 2009, JSConf-nimisessä JavaScript- teemaisessa tapahtumassa. Tässä esityksessä hän yllätti kaikki esittelemällä JavaScript-koodia, joka ei toimi lainkaan selaimessa, vaan määrittelee palvelimen sen ulkopuolella.

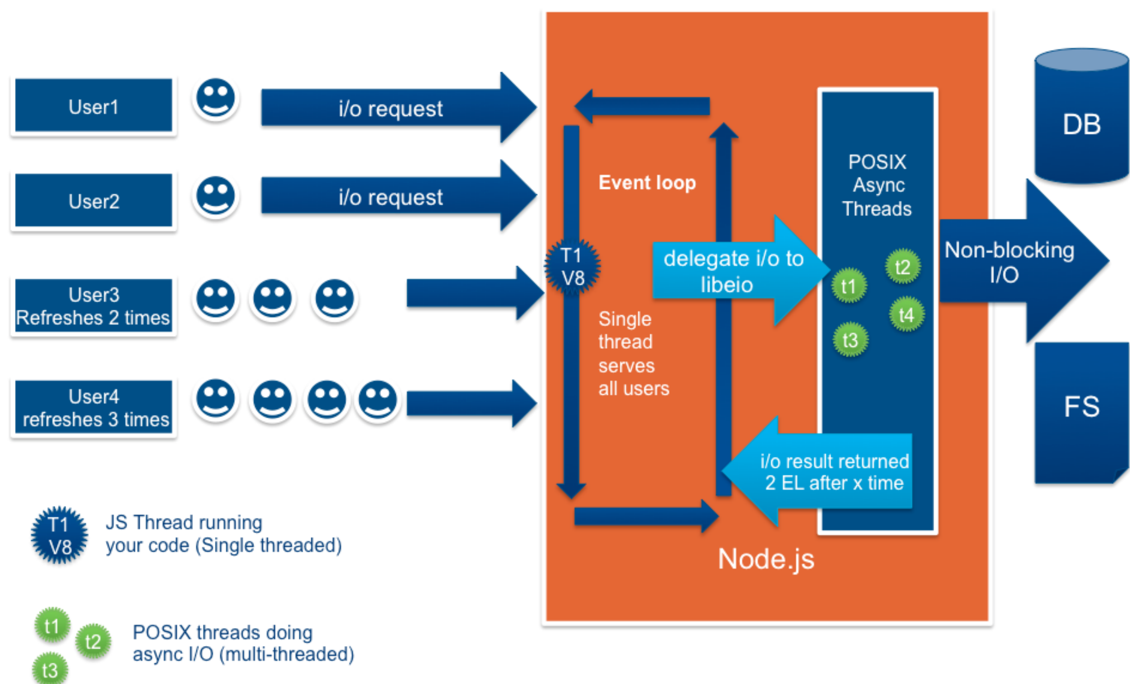
Ensimmäinen versio julkaistiin virallisesti seuraavan vuoden alussa NPM:n kanssa Mac OS X- ja Linux-käyttöjärjestelmille. Kuitenkin jo vuotta myöhemmin julkaistiin Microsoftin ja Joyentin yhteistyössä kehittämä natiivi Windows-versio. Vuonna 2012 Dahl väistyi Node.js:n pääkehittäjän paikalta ja NMP:n luoja Isaac Schlueter alkoi ohjata projektia. Tästä kaksi vuotta myöhemmin projektin johtaja vaihtui uudestaan, kun Timothy J. Fontaine otti ohjat Node.js-kehityksestä.

Myös Node.js:n ympärillä toimivat organisaatiot ovat kokeneet paljon muutoksia. Alun perin Joylent sponsoroi sen kehitystä, mutta myöhemmin päädyttiin neutraalin säätiön perustamiseen, jotta kehitys pysyisi mahdollisimman vapaana.

3.3 Toiminta

Suurin osa perinteisistä palvelinratkaisuista, kuten PHP, ASP.NET ja Java käyttävät kaikki lukuisia säikeitä toiminnassaan. Tällöin jokaisella kutsulla luodaan uusi säie, joka palvelee tätä kutsua.

Node.js taas poikkeaa tästä toimintatavasta radikaalisti, ja jokainen palvelinkutsu käsitellään samassa säikeessä. Jotta tämä saadaan toimimaan, käyttää Node.js ”Single Threaded Event Loop”-mallia. Tämä on kuitenkin hieman hämäävää, sillä Node.js:n ”päälooppi” on yksisäkeinen, mutta I/O eli input- ja output-operaatiot pyörivät omissa säikeissään, koska ne ovat asynkronisia, jotta malli saadaan toimimaan. Node.js-ympäristö on siis tavallaan yksisäikeinen, mutta ei täysin.



Kuva 7. Node.js-arkkitehtuuri (Chandrayan, 2017).

Kuvassa 7 on kuvattu Node.js-arkkitehtuuri, joka toimii edellä mainitun Singe Threaded Event Loopin ympärillä. Vasemmalla näkyy käyttäjiä, jotka antavat I/O-kutsuja järjestelmälle. Event loop ottaa kutsut vastaan ja välittää ne oranssin laatikon sisällä sijaitseville

taustasilmukoille. Nämä silmukat käsittelevät kutsut loppuun asynkronisissa silmukoissa, jonka jälkeen tulos palautetaan takaisin event looppiin. Se lopulta palauttaa tulokset takaisin käyttäjille

3.4 Moduulit ja NPM-pakkaustenhallinta

Node.js-ympäristössä kirjastoja kutsutaan moduuleiksi, ja ne sisältävät funktioita ja toiminnallisuutta, kuin missä tahansa muussa kielessä tai ympäristössä, ja niitä käytetäänkin Node.js-ympäristössä erittäin paljon. Myös niiden sisältäminen sovelluksiin on varsin yksinkertaista ja tapahtuu yksinkertaisella require()-funktiolla. Tämän jälkeen kaikki moduulin funktiot ovat sovelluksen käytössä.

```
1  var express = require('express');
2  var app = express();
3  var sql = require("mssql");
4  var bodyParser = require("body-parser");
5  var proxy = require("express-http-proxy")
6  var winston = require('winston');
7  var passport = require('passport')
```

Itse moduulien asentaminen on myös helppoa NPM:n (Node.js Package Manager) avulla. Siitä pidetään myös yhtenä Node.js:n parhaista ominaisuuksista. NPM:n kautta on mahdollista asentaa tuhansia erilaisia moduuleita Node.js-ympäristöön suoraan editorin komentoriviltä. Näitä moduuleja ylläpidetään samannimisessä verkkotietokannassa, joka sisältää ilmaisten pakettien lisäksi myös maksullisia tai täysin yksityisiä moduuleja. Moduulien varsinainen asennus tapahtuu yksinkertaisesti kirjoittamalla ”npm install (paketin nimi)”-komentoriville, jolloin NPM hakee moduulin repositoriosta ja asentaa sen. Tämän jälkeen se on heti valmiina käyttöön.

```
$ npm install passport
```

Kuva 8. Moduulin asennus

Kuten edeltävästä kuvasta näkyy, moduulien asennus NPM:lla on erittäin yksinkertaista. Jos sellaista ei vielä ole, NPM luo node_modules-nimisen kansion, jonne kaikki moduulit työtilassa tallennetaan jatkossa. Kuten mainittiin, suurin osa moduuleista on julkisia, mutta jos käyttäjä haluaa asentaa yksityistä moduuleita, täytyy ennen sen nimeä tehdä @scope-määritys. Tällöin edellisen kuvan komento olisi esimerkiksi npm install @myorg/passport.

3.5 Node.js-rajapintakehykset

Nyt kun on puhuttu REST-periaatteesta sekä Node.js-ympäristöstä pohditaan, miten, REST-rajapinnan voisi toteuttaa sillä. Vaikka rajapinta on mahdollista toteuttaa täysin ilman erillistä kehystä, tähän ei kuitenkaan ole mitään erityisen hyvää syytä, ellei välttämättä halua tutustua Node.js:n toimintaan aivan perin pohjin. Rajapintaa toteuttaessa Node.js:lla käytetään yleensä jotain sen lukuisista rajapinta-kehyksistä, jotka tekevät kehitysprosessista nopeamman ja helpomman. Node.js-kehysten laadusta kertoo myös se, että vuonna 2017 tehdyssä Stackoverflow'n kyselyssä 47,1 % kaikista, jotka käyttävät jotain kehystä verkkojärjestelmissään, käyttävät Node.js-ympäristöä.

Kehyksen perimmäinen ajatus on toimia rajapinnan pohjana, jonka päälle järjestelmä rakennetaan. Jokaisessa on omat tavoitteensa ja erikoisuutensa, ja valinta tehdään sen perusteella, millaiset järjestelmän tarpeet ovat. Jotkut keskittyvät web-rajapintaan, kun toiset taas tarjoavat sen lisäksi HTML-pohjia, tai kenties tukea tietynlaisille tietokantarakaisuille. Seuraavissa luvuissa tarkastellaan, mitkä ovat suosituimpia vaihtoehtoja kehyksiksi ja millaisia ominaisuuksia ne tarjoavat. Niitä tarkastellaan myös sen pohjalta, miten ne auttavat täyttämään REST-periaatteen tavoitteet tai joudutaanko niissä joustamaan siitä. Tietenkin sen periaatteet ovat hyvin korkeatasoisia jo valmiiksi ja työkalut vaikuttavat ainoastaan rajallisesti niiden toteuttamiseen ja paremminkin tukea pohjaa, jonka Node.js jo valmiiksi tarjoaa.

Express

Suosituin kaikista verkkokehyksistä on ehdottomasti Express. Sitä käytettiin myös myöhemmin esiteltävässä projektissa ja sitä pidetään yleisesti Node.js-ympäristön standardina verkkorajapintojen kehittämiseen. Rakenteeltaan Express on hyvin kevyt ja pelkistetty, mutta siihen on saatavilla lisäominaisuuksia erilaisilla lisäosilla (plugin). Tästä huolimatta sillä on mahdollista rakentaa hyvinkin monimutkaisia sovelluksia juurikin näiden lisäosien sekä Node.js:n API-integraation ansiosta. Express pystyy siis käyttämään kaikkea sitä, mitä Node.js pystyy.

Näissä samoissa ominaisuuksissa piilevät kuitenkin myös sen heikkoudet. Express ei tarjoa juurikaan työkaluja koodin ylläpitoon ja erittäin rajoitetun virheenhallinnan, joka saattaa tehdä kehittäjän työstä hankalaa, jos toteutusta ei ole rakennettu huolella. Sen käyttämistä täysin sellaisenaan kannattaa siis harkita, sillä sen lukuisat lisäkirjastot paikkaavat sen puutteita. Expressin suosiosta johtuen sen kilpailijoita verrataankin usein suhteessa siihen ja tietenkin suosituimpana kehyksenä se on yleensä hyödyllistä, kun aletaan valita työkaluja.

Sails.js

Sails.js on Expressin päälle rakennettu MVC-verkkokehys. Sen arkkitehtuuri on samankaltainen Ruby on Railsin arkkitehtuurin kanssa, mutta se tukee enemmän modernin kaltaista rajapintakehitystä. Se kuitenkin eroaa Expressistä tarjoamalla enemmän ominaisuuksia ilman ylimääräisiä lisäosia kuten automaattisesti generoidun REST-rajapinnan, helpon WebSocket-integraation sekä tärkeimpänä yhteensopivuuden lähes kaikkien käyttöliittymäalustojen kanssa. Voisi jopa siis sanoa, että Sails.js on Express, joka tarjoaa laajemman REST-mallia tukevan ominaisuuskokoelman ilman lisäosia, mutta samalla myös tutun rakenteen, jos on työskennellyt aikaisemmin Ruby on Railsin parissa.

Restify

Express ja Sails.js ovat erittäin suosittuja kehyksiä REST-rajapintaa rakentaessa, vaikka etenkään Express ei suoraan sisällä REST-kehitystä tukevia ominaisuuksia. Restify on Koa.js:n ohella erityisesti REST-rajapintojen rakentamiseen tarkoitettu web-kehys. Sen

tavoitteena on helpottaa esimerkiksi versionhallintaa, ja virheiden käsittelyä. Mukana tulee myös DTrace-niminen työkalu, jonka avulla rajapinnasta voi etsiä suorituskykyongelmia. Restify erottuu muista olemalla kaikista kehyksistä pelkistetyin. Tämän ansiosta kehittäjä saa täyden kontrollin HTTP-interaktioista ja DTracen antaman näkyvyyden suorituskykyyn ja näin antaa vapauden toteuttaa REST-mallia niin pitkälle kuin haluaa.

4 Node.js-pohjainen laite- ja ohjelmistotarpeiden hallintajärjestelmä

Tässä luvussa esitellään Helsingin kaupungin kaupunkiympäristölle suunniteltu REST-rajapinta ja sovellus, jota se palvelee. Esittely aloitetaan tarkastelemalla projektin tavoitteita ja sitä, mitkä sen tärkeimmät tehtävät ovat ja mitkä olivat ne ideat, minkä takia koko projektia lähdettiin alun perin kehittämään.

Teknisellä puolella ensimmäisenä katsotaan sovelluksen toimintaympäristöä, jossa sen pitää toimia. Tämä tarkoittaa lähinnä asioita kuten sitä, millainen on kaupungin verkon rakenne. Tämä on erityisen tärkeää web-sovelluksen toiminnan kannalta, koska se pitää mahdollisesti integroida jo olemassa olevaan verkkojärjestelmään. Toinen tärkeä kokonaisuus on sovelluksen tietokanta, joka toimii rajapinnasta saatavan informaation lähteenä. Seuraavaksi katsotaan, mikä tietokantaratkaisu rajapinnan takana on ja tarkastellaan hieman sen yksityiskohtia ja erikoisuuksia.

Kolmantena tässä luvussa päästään tärkeimpään osaan eli itse rajapintaan. Kyseinen rajapinta on siis tietokannan ja käyttöliittymän välinen osa, joka tarjoilee kaiken sovelluksen tarvitseman tiedon käyttäjän näkemään osaan. Sen rakentamisessa on mahdollisuuksien mukaan seurattu REST-mallin rajoitteita, joista puhuttiin toisessa luvussa. Tästä syystä on myös oleellista pohtia, kuinka hyvin niiden toteuttamisessa onnistuttiin ja olisiko joissain osissa parannettavaa. Rajapintaan on sen perustoiminnallisuuden lisäksi tarvinnut myös ohjelmoida erilaisia lisätoimintoja, jotka parantavat sen toiminnallisuutta. Näistä lisätoiminnoista tarkastellaan kolmea eri kategoriaa, jotka ovat autentikointi, dokumentointi ja tapahtumaloki. Näistä kolmesta erityisesti autentikointi on oleellinen, sillä virastossa työskentelee yli tuhat työntekijää ja sen käyttöä on haluttu rajoittaa. Tämä tehdään sen vuoksi, että sovellusta ei vahingossa käytä kukaan, joka ei osaa sen käyttöä ja mahdollisesti sotkisi siellä olevia tietoja. Tietenkin myös ulkopuolisten pääsy

sovellukseen on huolenaihe, vaikka se sijaitseekin kaupungin sisäisillä palvelimilla, jotka eivät näy viraston sisäisen verkon ulkopuolelle. Kaikki lisätoiminnot on toteutettu Node.js:n NPM-työkalulla asennettavilla lisäosilla, jotka kuuluvat alustan vahvuuksiin.

Rajapinnan tarkastelun jälkeen katsomme myös hieman rajapinnan testaamista, tai pikemminkin sen puutetta. Tiukkojen aikarajoitteiden ja henkilöstöpuutteen vuoksi automatisoitu testaus jouduttiin jättämään täysin pois. On kuitenkin hyvä miettiä hieman, kuinka se olisi mahdollisesti kannattanut toteuttaa.

Aivan viimeisenä katsotaan vielä projektin lopputulosta ja sen tulevaisuutta. Onko sovellusta otettu lainkaan käyttöön vai olisiko siinä tarvetta mahdollisesti jatkokehitykselle? Miten sovelluksessa käytetyt työkaluvalinnat toimivat vai olisivatko kenties muut vaihtoehdot olleet parempia? Tarkoituksena on muutenkin yleisesti arvioida koko projektin onnistumista sekä mahdollista tulevaisuutta.

4.1 Projektin tavoite

Helsingin kaupungin kaupunkiympäristön toimiala työllistää yli 1600 henkilöä, ja näistä suurin osa käyttää päivittäisessä työssään tietokonetta ja erilaisia ohjelmistoja. Työntekijöiden tarpeet poikkeavat myös rajusti toisistaan. Sen takia tätä kyseistä prosessia haluttiin suoraviivaistaa, jotta yksittäinen työntekijä saisi laitteet ja ohjelmistot, joita hän tarvitsee. Näin hän voisi helposti aloittaa työtehtäviensä tekemisen ilman edestakaista vuorokeskustelua It-hallinnon kanssa. Tämä säästäisi merkittävästi kyseisestä toiminnasta sillä hetkellä vastaavien työntekijöiden aikaa. Se toisi myös paljon mahdollisia säästöjä, kun ylimääräisistä ohjelmistolisensseistä päästäisiin eroon tai saataisiin ne siirrettyä niille, jotka niitä tarvitsevat. Monet viraston käyttämistä ohjelmistoista ovat nimittäin erittäin kalliita mallinnusohjelmia, joita kaupunkisuunnittelijat ja insinöörit taritsevat työhönsä

Kaikki tämä saadaan aikaan luomalla profiileja, jotka vastaavat erilaisia työtehtäviä. Tämän jälkeen näihin ”profiileihin” pyritään määrittelemään siihen työtehtävään tarvittavat laitteet sekä ohjelmistot ja lupalisenssit muihin järjestelmiin. Näiden ohjelmien yksityiskohtiin voidaan myös merkitä, ovatko ne kenties kuukausi- vai kertamaksullisia. Sovellus

toimii siis samalla myös tavallaan lisenssirekisterinä, jonka avulla voidaan pitää kirjaa kalliiden ohjelmistojen määrästä.

Sovelluksen avulla voidaan sitten katsella, muokata ja järjestää tätä dataa. Näin se pystyy tarjoamaan helpon ja kätevän referenssin sen käyttäjälle siitä, mitä hänen tulisi asentaa uuden käyttäjän koneelle. Tämän ansiosta työntekijä pystyy aloittamaan työnsä mahdollisimman helposti ja nopeasti, mikä säästää molempien osapuolien arvokasta työaikaa.

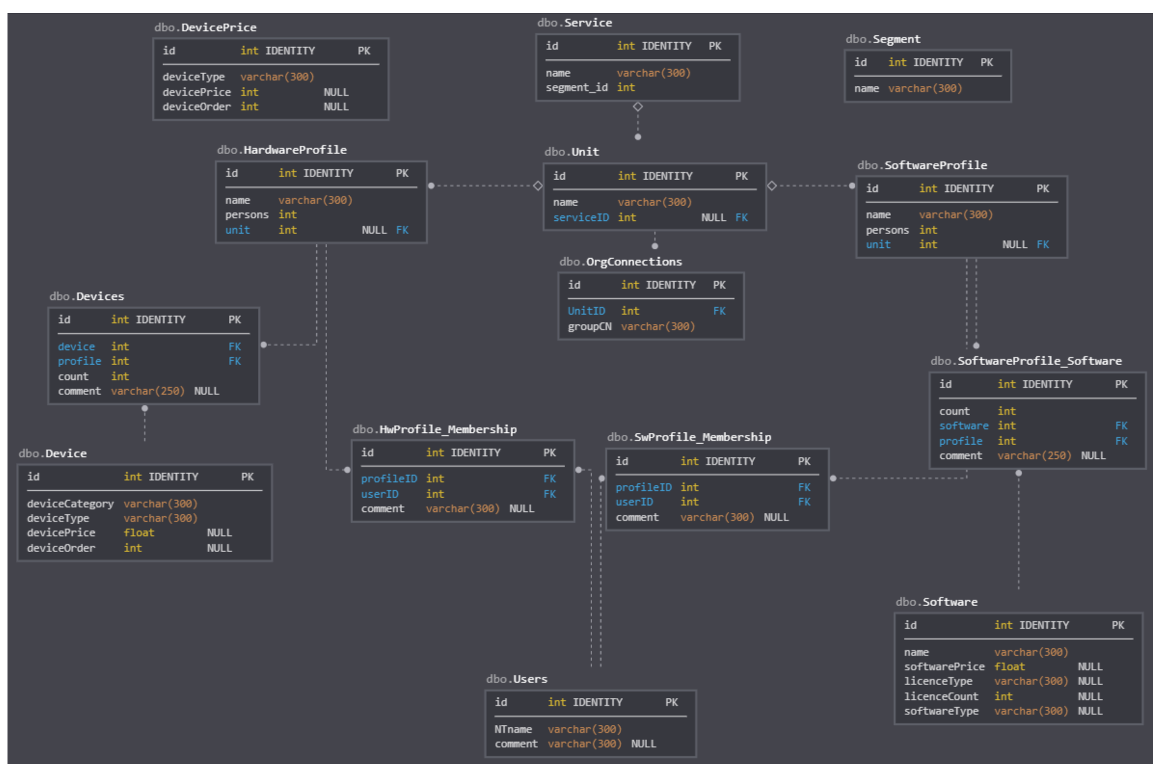
4.2 Toimintaympäristö

Kaupunkiympäristö ylläpitää lukuisia verkkopalveluja sisäiseen viestintään tarkoitetusta intranetistä ja tikettijärjestelmistä, julkiseen käyttöön luoduista karttajärjestelmistä. Oli siis alusta asti selvää, että myös tämä järjestelmä olisi verkkoselaimissa toimiva web-ohjelma, jota pyöritetään viraston omilla palvelimilla. Käyttäjät voivat siis avata ohjelman ollessaan viraston sisäisessä verkossa, jolloin sen käyttäminen vaatii jatkuvan verkkoyhteyden. Sovellus on optimoitu toimimaan Chrome-selaimella, joka on viraston käytettäväksi suositeltu selain. Muiden selainten optimointia ei suunniteltu rajallisen käyttäjäkunnan vuoksi.

Virastolla on myös valmiiksi oma sisäinen verkkopalvelu, joka sisältää useita erillisiä sovelluksia keskustelupalstoista lukuisiin karttapalveluihin. Kenelläkään projektiin osallisella ei kuitenkaan ollut tarkkaa tietämystä tämän valtavan järjestelmän toiminnasta, joten sovelluksen siihen integroiminen päätettiin jättää tämän projektin ulkopuolelle. Oli myös epäilys, onko sovelluksen liittäminen osaksi vanhaa järjestelmää edes mahdollista, sillä monet sen osista ovat huomattavan vanhoja uuteen Node.js-pohjaiseen sovellukseen nähden. Projektin lopussa sovellus jäi tilaan, jossa sovellusta käyttäessä on kirjoitettava vain sen osoite selaimen osoitekenttään.

4.3 Tietokanta ja muut resurssit

Kun rajapintaa alettiin toteuttaa, ei valmista dataa ollut lainkaan, joten pohja tietokannalle luotiin myös Microsoft SQL Serverillä. Se toimii rajapinnan pääasiallisena resurssina, mutta sen lisäksi käytetään myös viraston Active Directorya. Sen avulla jokainen työntekijä voidaan liittää johonkin profiiliin ja näin pitää kirjaa siitä, mitä ohjelmia työntekijällä pitäisi käytössään olla. Kuvassa 9 tutustutaan tietokannan rakenteeseen ja niihin ratkaisuihin, joihin siinä on päädytty.



Kuva 9. Tietokannan kuvaus.

Tietokantaa tarkastellessa luontevinta on lähteä tarkastelemaan katsomalla kuvaa 9 sen yläreunassa olevasta Service-taulusta. Tämä kuvastaa viraston rakenteessa olevia palveluita, jotka ovat sen toiseksi ylimmän tason rakenne. Niiden yläpuolella on vielä kuvan oikeassa ylänurkassa näkyvä Segment-taulu. Tämä kuitenkin jätettiin pois, sillä sen kuvaaminen sovelluksen sisällä ei lopulta ollutkaan tarpeellista tai edes hyödyllistä. Myös

DevicePrice-taulu oli alun perin suunniteltu olevan osa tietokantaa, mutta jäi pois suunnitelmien muututtua. Palveluista loogisesti alaspäin mentäessä päästään Unit-tauluun, joka puolestaan kuvaa viraston eri yksiköitä ja siihen liitetty OrgConnections, jonka avulla sovellukseen yhdistetään Active Directory -toiminnallisuutta. Unit-taulusta oikealle ja vasemmalle mentäessä tulevat HardwareProfile ja SoftwareProfile. Nämä edustavat aikaisemmin mainittuja profiileita, jotka ovat sovelluksen olennaisin osa. Kuvaukseltaan ne ovat lähes identtiset, mutta toinen edustaa sovellus- ja toinen laiteprofiileja. Näihin vuorostaan taas on liitetty taulut Devices ja SoftwareProfile_Software, jotka kuvaavat yhteen profiiliin liitettyä laitetta tai ohjelmaa ja kertovat esimerkiksi niiden määrän kyseisessä profiilissa. Software- ja Device-taulut sisältävät staattisen listan laitetyypeistä ja ohjelmista sekä niiden hinnoista, joita niiden yläpuolella olevat taulut käyttävät näyttäessään profiilin sisältöä. Nämä ovat oman polkunsa alimmat taulut ja jäljellä on enää HwProfile_Membership sekä SwProfile_Membership. Nämä molemmat yhdistyvät vielä Users-tauluun. Näiden tarkoitus on yhdistää membership-taulujen avulla Users-taulusta saatavat käyttäjät profiilien jäseniksi. Tämä kattaa kokonaisuudessaan koko rajapinnan käyttämän tietokannan ja sen, kuinka eri osat toimivat keskenään tarjotakseen kaiken tarpeellisen toiminnallisuuden.

4.4 Sovelluksen REST-rajapinta

Kuten aikaisemmissa osioissa mainittiin, rajapinnan pohjaksi valittiin Node.js. Se soveltuu erittäin hyvin REST-mallisen web-järjestelmän tekoon ja tarjoaa rajapinnalle hyvät mahdollisuudet skaalautua suuremmaksi datamäärän kasvaessa. Koko sovellus rakennettiin alusta alkaen sieltä saatavaa dataa mukaan lukien, ja se on elänyt ja muuttunut projektin aikana useasti, kun projektin skaalaa laajennettiin. Rajapinnan laajuuden vuoksi sen kokonaan läpikäyminen ei ole järkevää, mutta pyritään tarkastelemaan, millainen sen toiminta on pääpiirteittäin. Rajapintaa rakennettaessa lähdettiin liikkeelle siitä, mitä dataa haluttiin näyttää ja miten sitä olisi tarkoitus muokata. Yksinkertaistettuna sovellus siis sisältää erilaisia profiileita, joilla on tietyt ohjelmat sekä laitteet. REST-periaatteen mukaisesti näille tiedoille tehtiin GET-, POST- ja DELETE-käskyjä, jotka koskevat kerrallaan yhtä tai useampaa tietokantataulua. REST-periaatteen noudattaminen on muutenkin ollut tavoitteena, vaikka siitä on jouduttu paikoiten joustamaan paljonkin.

Kuvassa 10 näkyy tyypillinen esimerkki projektiin tehdystä rajapintakutsun määrityksestä. Rivillä kaksi määritetään kutsun URL-osoite, jonka jälkeen kirjautumisen ja admin-oikeudet tarkistava funktio.

```
//HardwareProfile
app.delete("/api/deletehwp", checkAuth, checkAdmin, (req, res) =>{
    get.selectHardwareProfile(req.body.id).then(result=>{
        logger.log({
            level: 'info',
            action: 'Delete',
            target: 'HardwareProfile',
            message: result,
            author: req.user._json.sAMAccountName
        })
    })
    del.deleteHwP(req.body).then(result=>{
        res.send(JSON.stringify(result))
    }).catch(err=>{
        handleError(err, res, logger)
    })
})
```

Kuva 10. Esimerkki kutsun määrityksestä

Sen jälkeen kutsutaan selectHardwareProfile()-funktiota, joka saa parametrikseen id:n. Tämä tehdään sitä varten, jotta lokitiedostojen keräin voi kirjata ylös poistettavan profiilin, joka tapahtuu tämän lokitiedoston luomisen jälkeen. Jos jompikumpi toiminnoista epäonnistuu, siirrytään virheenkäsittelijään, joka antaa virheviestin ja kirjaa sen virhelokiin.

Virheenkäsittelijälle täytyy myös antaa parametrina instanssi logger-oliosta, jotta se toimii oikein. Kuvassa 11 taas näkyy deleteHwPM()-funktio, joka poistaa annetun id:n perusteella laiteprofiilin.

```
//HwProfile_Membership
this.deleteHwPM = (insertInfo) => {
    return new Promise((resolve, reject) => {
        db.request()
        .input("id", sql.Int, insertInfo.id)
        .query("DELETE FROM HwProfile_Membership WHERE id = @id ")
        .then(RequestedSoftware=>{
            resolve(RequestedSoftware)
        }).catch(err=>reject(err))
    })
}
```

Kuva 11. Kuvan 9 poistofunktio.

Nämä funktiot ovat hyvin samankaltaisia toistensa kanssa. Pääosin mikä niissä vaihtelee, on tietokantakyselyjen pituus. Pidemmissä kyselyissä lupausten käyttäminen pitää huolen, että pitkät kyselyt toimivat moitteettomasti ja tilattomuuden REST-rajoite pysyy hallinnassa. Lupaus on siis olio, joka palauttaa arvon, kun sen ”lupaus” on täytetty. Aluksi lupaus on pending-tilassa, jossa se odottaa, että sitä kutsutaan. Kun lupaus on täytetty, se siirtyy fulfilled-tilaan ja tällöin sillä on arvo. Jos lupaus hylätään, siirtyy se rejected-tilaan, ja tällöin hylätty lupaus saa arvokseen hylkäyksen syyn.

Entä miten rajapinnan REST-rajoitteet ylipäätään täyttyvät? Asiakas-palvelinmalli on otettu huomioon, sillä rajapinta on täysin itsenäinen palvelunsa, kun käyttöliittymänä toimii projektissa mukana olleen toisen koodaajan rakentama Vue.js-käyttöliittymä. Rajapinta käyttää myös aikaisemmin mainittuja yhdenmukaisia HTTP-metodeja oikein, joka on oleellista kerrostetun järjestelmän aikaansaamiseksi. Myös client side scriptingiä on hyödynnetty käyttöliittymässä pääasiassa rajapinnan tarjoileman datan muokkaamiseen. Kutsut ovat myös täysin tilattomia, sillä ne eivät sisällä mitään oletusta tilasta, vaan tarjoilevat vain dataa annetuilla parametreilla.

Rajapinnan suurimmaksi puutteeksi jääkin ehkä tässä järjestelmässä se, että välimuistia ei hyödynnetä lainkaan, vaan käyttöliittymä joutuu joka kerta tekemään uuden kutsun, kun se haluaa. Tähän ratkaisuun päädyttiin lähinnä aikarajoitusten takia ja siksi, että nykyisen nopeuden koettiin riittävän. Kutsujen palauttaman datan määrä, jota kutsut palauttavat ei luultavasti kasva ajan myötä paljoakaan. Toinen puute löytyy yhtenäisen rajapinnan rajoitteesta. Oikeastaan ainoastaan resurssien tunnistaminen sen neljästä alakategoriasta toteutuu: resurssit tunnistetaan URL:n avulla. Esimerkiksi yksittäistä sovellusta tarkastellessa osoitepalkissa näkyy sen uniikki id. Tämän id pysyy aina samana, joten tämä kohta yhtenäisestä rajapinnasta täyttyy. Edellä mainitut puutteet johtuvat paljolti siitä, että käyttöliittymää rakennettiin samanaikaisesti rajapinnan kanssa. Tämä johti moniin hyvin spesifien kutsujen luontiin, jotta käyttöliittymä olisi helpompi ja nopeampi rakentaa. Suurilta osin rajapinta kuitenkin seuraa REST-periaatteita ja hyötyy niistä lukuisilla tavoilla puutteistaan riippumatta.

4.5 Rajapintaa tukevat kirjastot

Koska kolmannessa osiossa käsiteltävää järjestelmää rakennettaessa käytettiin Express-kehystä, olisi hyvä myös puhua lyhyesti lisäkirjastoista, joita etenkin siihen löytyy runsain määrin, ja ne ovat varsin oleellisessa asemassa, kun sillä rakennetaan rajapintoja. Seuraavissa luvuissa esiintyvät kategoriat heijastelevat tarpeita, joita tätä työtä tehdessä on tullut vastaan. Siksi seuraavaksi tarkastellaan lyhyesti niiden tarkoitusta ja toimintaa rajapinnassa.

4.5.1 Autentikointi

Rajapintaa tukevista kirjastoista tärkein on ehdottomasti autentikointi, sillä ilman sitä järjestelmä jäisi haavoittuvaksi hyväksikäytölle. Suojaus on tietenkin aina mahdollista toteuttaa lokaalilla sisäänkirjautumisella. Kaupungilta tulleiden rajoitteiden vuoksi valittiin kuitenkin sellainen työkalu, joka mahdollistaa MicroSoftin Active Directory-ohjelman käyttämisen kirjautumisstrategiana. Passport on middleware eli väliohjelma, joka toimii tässä tapauksessa ohjelman eri osien välissä ja tarjoaa jotain palveluita sille. Passportin käyttö mahdollista satojen eri strategioiden käytön, kuten Facebook, Twitter, tai lokaalin tietokannan. Active Directorylla on myös toinenkin hyvä puoli: sen avulla voi helposti

määritellä, kenellä on oikeus tehdä tietynlaisia toimintoja. Tämä onnistuu siten, että kirjautumisstrategiaa määritellessä tarkistetaan, kuuluuko käyttäjä tiettyyn Active Directory-ryhmään Tämän jälkeen funktio määrittää, onko nykyinen käyttäjä oikeutettu admin-tason funktioihin. Tämän jälkeen funktio tarvitsee vain lisätä kutsujen määritykseen haluttaessa. Seuraavassa kuvassa vielä tarkastellaan kuinka, admin-oikeudet tarkastetaan.

```
65  function checkAdmin(req, res, next) {
66      if(req.user.admin) {
67          return next()
68      } else {
69          return res.status(403).send(JSON.stringify({'forbidden': true, 'message': 'needs privilege elevation'}))
70      }
71  }
```

Kuva 12. Admin funktion määritys

Kuvassa 12 näkyy checkAdmin()-funktion määritys, joka tarkastaa if-lauseessa, onko käyttäjällä admin-oikeudet vai ei. Saadessaan myönteisen vastauksen se palauttaa arvon true ja päästää käyttäjän jatkamaan. Muussa tapauksessa käyttäjä saa ruudulleen forbidden-ilmoituksen ja viestin, jossa pyydetään käyttäjää hankkimaan korkeamman tason oikeudet.

4.5.2 Dokumentointi

Tässä työssä dokumentoinnilla tarkoitetaan lähinnä rajapinnan dokumentaatioita, joka määrittelee sen resurssit, eli millaisia päätepisteitä sieltä löytyy ja mitä parametreja niille voi antaa. Tämä toimii myöhemmin sovelluksen pariin palaaville tai uusille kehittäjille referenssinä ja helpottaa tulevaa kehitystä. Se on myös frontend-koodaajan tärkein työkalu, kun sitä kehitetään itsenäisesti backendistä riippumattomana. Nämä työkalut luovat olemassa olevasta koodista pohjan, jota voi itse täydentää työkalun tarjoamalla annotaatiotyylillä. Kutsuja voi esimerkiksi ryhmitellä toimintojen mukaan sekä antaa niille tarkentavia kuvauksia. Määrällisesti dokumentointityökaluja ei ole kovinkaan paljoa, mutta olemassa olevat ovat varsin hyviä ja tekevät kaiken tarpeellisen.

ApiDoc.js on useita eri kieliä tukeva erityisesti REST-rajapintojen dokumentointiin tehty työkalu. Tämä lisäosa aikaisemmin mainittuun tapaan generoi valmiin pohjan koodista,

jota pystyy itse täydentämään lisää. Vaihtoehtona tälle olisi mm. Swaggerin Express-liitännäinen. Se on ominaisuuksiltaan ehdottomasti laajempi, mutta myöskin maksullinen. Tämän seurauksena se jätettiin pois harkinnasta ja apiDoc.js otettiin käyttöön helppona ja yksinkertaisena vaihtoehtona. Kuvassa 13 harmaalla näkyy esimerkki työkalun annotaatiosta.

```
524    /**
525     * @api {get} /api/selectunit/:id Get certain Unit
526     * @apiName Get Unit
527     * @apiGroup SingleGets
528     *
529     * @apiParam {Number} id unique ID of the Unit.
530     *
531     * @apiSuccess {Number} id Id of the unit
532     * @apiSuccess {String} name name of the unit
533     */
534    app.get("/api/selectunit/:id", checkAuth, (req, res)=>{
535        get.selectUnit(req.params.id).then(results=>{
536            res.send(JSON.stringify(results))
537        }).catch(err=>{
538            handleError(err, res, logger)
539        })
540    })
```

Kuva 13. Rajapintadokumentaation annotaatio

Kuvaus aloitetaan /** -merkinnällä, jonka jälkeen @-merkillä ja sen jälkeisellä sanalla kerrotaan, mitä dokumentaatiossa halutaan ilmaista. Esimerkiksi kuvassa näkyvä @apiName kertoo dokumentaatiossa kutsun nimen ja @apiGroup sen alakategorian. @apiParam puolestaan määrittää kutsun parametrit ja sen jälkeen tuleva {Number} parametrin tyypin. Viimeisenä @apiSucces kertoo, mitä ja minkä tyyppistä dataa kutsu palauttaa. Lopulta kuvaus lopetetaan */-merkinnällä.

4.5.3 Tapahtumaloki

Tapahtumalokien tavoitteena tässä projektissa oli edesauttaa vahingollisista toiminnoista palautumista. Tämä tehdään siksi, koska jos käyttäjä epähuomiossa esimerkiksi poistaa lopullisesti jotain saadaan lokitiedostosta selville mitä käyttäjä oli poistanut. Myös

tietoturvan kannalta on hyödyllistä, että jokaisesta poisto- ja lisäystoiminnosta jää merkintä, josta käy ilmi, kuka on käyttäjä, joka sen on tehnyt. Tämän lisäksi myös virheiden tallettaminen erilliseen tiedostoon koettiin hyödylliseksi, koska se helpottaa virheiden korjaamista sellaisissa tilanteissa, kun käyttäjät ovat jo ehtineet saada suuren määrän virheilmoituksia. Näin ongelmaa voidaan tarkastella suurempana kokonaisuutena ilman, että virheviestejä tarvitsee etsiä konsolista erikseen. Mitään tätä erityisempää toimintaa ei työkalulta vaadittu. Asian olisi hoitanut lähes mikä tahansa useasta Express:n kanssa yhteensopivasta vaihtoehdosta. Oikeastaan ainoaksi rajoitukseksi jäi tällöin keveys ja yksinkertaisuus.

Winston, joka lopulta valikoitui työkaluksi, on ominaisuuksiltaan varsin laaja ja sisältää lokitiedostojen luonnin lisäksi myös paljon niiden personointimahdollisuuksia. Näistä tärkeimpinä erilaisten formaattien ja itsemääriteltyjen varoitustasojen käyttö. Nämä eivät kuitenkaan olleet ominaisuuksia, joita järjestelmässä oli tarvittu, joten siitä hyödynnettiin vain perusominaisuuksia, ja niiden käyttöönotto oli erittäin helppoa. Seuraavassa kuvassa tutustutaan siihen, miten sen määritys tehdään ja miten sitä voidaan kustomoida.

```
// creates winston logger
const logger = winston.createLogger({
    format: winston.format.combine(
        winston.format.timestamp(),
        winston.format.json()
    ),
    transports: [
      new winston.transports.File({ filename: 'error.log', level: 'error' }),
      new winston.transports.File({ filename: 'combined.log', level: 'info'}),
      new (winston.transports.Console)({'timestamp':true})
    ]
  });
```

Kuva 14. Winston lokikirjaimen määritys

Määrityksen teko alkaa luomalla instanssi Winstonista, jonka jälkeen siirrytään varsinaiseen kustomointiin. Neljännellä ja viidennellä rivillä määritellään formaatti, jossa esim. tässä tapauksessa siihen lisätään aikaleima ja sen jälkeen määritellään se antamaan viestit JSON-muodossa. Tämän jälkeen transports-kohdassa riveillä 8 ja 9 määritetään

lokitiedostojen nimet sekä virhetasot. Tässä tapauksessa error.log-tiedosto saa ainoastaan virhetason viestit, kun taas combined.log saa lisäksi myös infotason viestit.

4.6 Testaaminen

Rajapintojen testaaminen vähänkään suuremmissa ohjelmistoprojekteissa kuuluu sen tärkeimpiin osiin. Sen avulla ollaan jatkuvasti perillä siitä, että kaikki rajapinnan osat toimivat niin kuin kuuluu ja eivätkä näin haittaa sovelluksen toimintaa. Tämä tulee erityisen tärkeäksi silloin, kun rajapintaan aletaan toteuttaa uutta toiminnallisuutta. Uudet ominaisuudet voivat usein rikkoa vanhoja ominaisuuksia, jolloin testien avulla voidaan helposti selvittää vian alkulähde ja näin nopeuttaa korjausprosessia. Valitettavasti testaamisen tärkeydestä huolimatta tähän projektiin ei ollut mahdollisuutta toteuttaa automatisoitua testaamista lainkaan. Tähän on useita syitä, mutta niistä oleellisin on jo luvun neljä alussakin mainittu ajan puute. Projektilla oli alun perin varsin erilaiset tavoitteet, ja sen oli tarkoitus olla työkalu, jolla syötetään Excel-tiedostoja tietokantaan. Tämä kuitenkin muuttui aikaisemmin mainittuun ohjelma- ja laitetarve sovellukseen. Tämän vuoksi aika ei riittänyt kaikkien kriittisten ominaisuuksien tekoon, ellei jotain karsittaisi. Testaaminen oli varsin selvä kohde poisjättämiselle, sillä se ei sinänsä vaikuta sovelluksen toimivuuteen. Rajapintaa ja käyttöliittymää sovelluksessa kyllä testattiin jonkin verran, mutta tämä kaikki tapahtui käsin. Postman oli tähän erittäin kätevä työkalu, jolla pystyy helposti tekemään rajapintakutsuja erilaisilla arvoilla ja näin testaamaan sen toimintaa.

Vaikka testaus puuttuukin täysin, on hyödyllistä silti puhua siitä, kuinka se olisi toteutettu, jos aikaa olisi riittänyt. Yksi suosittu vaihtoehto testaamiselle Node.js-ympäristössä on Mocha. Mocha on JavaScript-testikehys, joka mahdollistaa esimerkiksi asynkronisen testaamisen sekä tuen, jos asynkroniset testit epäonnistuvat aikarajoituksen vuoksi. Pelkkä testikehys ei kuitenkaan riitä, vaan lisäksi tarvitaan myös assertaatiokirjasto. Tämän kirjaston avulla testataan itse HTTP-kutsut sekä annetaan testeille kyky tietää, onko esimerkiksi GET-kutsusta saama JSON-tiedosto odotetun kaltainen. Tällaisia kirjastoja on Mocha-kehykselle useita, mutta etenkin monissa verkosta löytyvissä ohjeissa käytetään useimmiten Chai-nimistä kirjastoa. Chai tekee aikaisemmin mainitun kaltaisten testien tekemisestä helpompaa ja sujuvampaa. Mocha ja Chai olisivat todennäköisesti vali-

koituneet tämän rajapinnan testaustyökaluiksi jo pelkästään kattavan ohjevalikoiman ansioista. Kattavat ja helposti ymmärrettävät ohjeet ovat tärkeitä etenkin silloin, kun rajapinta-testaus ei ole ennestään tuttua.

Kuvassa 15 nähdään vielä lyhyt esimerkki siitä millaisia Mochalla ja Chailla tehdyt testit ovat ja mitä niissä varsinaisesti testataan.

```
chai.use(chaiHttp);
//Our parent block
describe('Books', () => {
    beforeEach((done) => { //Before each test we empty the database
        Book.remove({}, (err) => {
           done();
        });
    });
/*
  * Test the /GET route
  */
  describe('/GET book', () => {
      it('it should GET all the books', (done) => {
        chai.request(server)
            .get('/book')
            .end((err, res) => {
                  res.should.have.status(200);
                  res.body.should.be.a('array');
                  res.body.length.should.be.eql(0);
              done();
            });
      });
  });

});
```

Kuva 15. Esimerkki rajapinnan testauksesta (Zaza, 2016)

Kuvassa 15 on esimerkki rajapintaan kohdistuvasta GET-operaation testistä. Kuvassa on kaksi describe-funktiota. Ensimmäisestä näissä määritellään tietokanta tälle testille ja tyhjennetään se ylimääräisestä datasta. Tätä ei tämä sovelluksen testissä tarvitsisi käyttää, sillä tietokanta on erillinen osa rajapinnasta. Toisessa osassa määritellään itse testi.

It-funktiossa kerrotaan, mitä testin tulisi palauttaa, jonka jälkeen chai.rquest-funktiossa annetaan rajapintakutsu, jota halutaan testata, eli tässä tapauksessa /book. Viimeisenä määritellään vielä tärkein osa, eli mitä kutsun tulisi palauttaa, jotta testiä pidetään onnistuneena. Tässä tapauksessa kutsun pitäisi palauttaa tilaviesti 200, paluuviestin pitäisi olla taulukko ja sen pituuden nolla. Jos kaikki nämä ehdot täyttyvät, testi onnistuu ja kaikissa muissa tapauksissa epäonnistuu ja näyttää ne kohdat, joissa kohdattiin virhe.

4.7 Projektin tulos

Kuten projektin tavoitteista kertovassa luvussa puhuttiin, tavoitteena oli luoda sovellus, jolla voidaan hallinnoida viraston laite- ja ohjelmistorapeita. Projektille asetettiin tiettyjä vaatimuksia, jotka koskivat lähinnä sen ominaisuuksia ja muita tärkeitä seikkoja, kuten tietoturvaa ja dokumentointia. Tässä luvussa katsotaan, millainen sovelluksesta lopulta tuli, mitkä ovat sen tärkeimmät toiminnot ja miten niiden toteuttamisessa onnistuttiin.

Projektin päätyttyä sovelluksen lopulliseksi nimeksi annettiin Inno Importer. Nimi viittaa projektin alkuperäiseen tavoitteeseen, jossa sen oli tarkoitus olla vain työkalu, jolla viedään Excel-taulukoita tietokantaan. Tavoitteiden muuttuessa se kuitenkin sai nykyisen muotonsa. Tämä toiminnallisuus on kuitenkin mukana myös lopullisessa versiossa, vaikka se ei välttämättä näe paljoakaan käyttöä käyttöönoton jälkeen. Kuvan saaminen sovelluksen käyttöliittymästä ei valitettavasti ollut mahdollista, joten sen ominaisuudet joudutaan kattamaan vain sanallisesti. Varsinaiset päätoiminnallisuudet ovat siis aikaisemminkin mainittujen profiilien katselu ja muokkaus. Nämä profiilit ovat käytännössä eri työtekijöiden yleistettyjä toimenkuvia, ja ne sisältävät tähän toimenkuvaan tarvittavat ohjelmistot ja laitteet. Kaikki profiilit listataan käyttöliittymän sivuvalikosta avautuvaan listaan ja klikkaamalla haluttua viraston yksikköä, saadaan lista kyseisen yksikön profiileista. Avaamalla yksittäinen profiili saadaan puolestaan taulukkonäkymä, jossa on listattu laitteet tai ohjelmistot sekä niiden tiedot. Näitä tietoja pääsee muokkaamaan taulukon yläpuolella sijaitsevasta painikkeesta, joka avaa muokkausikkunan.

Toinen tärkeä ominaisuus on edellä mainittujen ohjelmien ja laitteiden selaus erillisessä taulukkonäkymässä, jotka ovat omilla sivuillaan. Tässä näkymässä näkyvät kaikki laitteet tai ohjelmistot sekä niiden tiedot. Näitä tietoja pääsee jälleen muokkaamaan klikkaamalla

haluttua ohjelmistoa ja valitsemalla muokkaa-painike sivun ylälaidasta. Kolmas pääominaisuus on profiilien lataaminen Excel-tiedostoina, joka avataan myös sovelluksen sivupalkista. Tämän toiminnon avulla mikä tahansa profiili on mahdollista muuttaa Excel-muotoon, jonka jälkeen sen arvoja voi muuttaa kaikissa Excel-tiedostomuotoa tukevissa taulukko-ohjelmissa. Kun tietoja on muokannut, voi taulukon ladata samasta ikkunasta takaisin, ja näin muuttaa sovelluksen sisäiset tiedot vastaamaan tätä Excel-taulukkoa. Tämä ominaisuus helpottaa suurien tietomäärien analysointia Excelin erinomaisilla tilasto-ominaisuuksilla tai antaa vaihtoehtoisen tavan muokata useita profiilitietoja kerralla.

Viimeisenä pääasiallisen toiminnan kannalta tärkeä ominaisuus on yksittäisten käyttäjien tietojen katselu. Tältä sivulta on mahdollista nähdä sovelluksen sisäiset yksittäisen käyttäjän tiedot. Näihin kuuluvat käyttäjän kytkökset hänen profiiliinsa, laitteet ja ohjelmistot jotka hänellä pitäisi olla, sekä viraston laiterekisteriohjelmasta saatu data ohjelmista joita hänellä kuuluisi olla. Tämä näkymä on tehty, jotta olisi mahdollisimman helppo nähdä onko käyttäjällä jo hänen tarvitsemansa ohjelmat, vai onko hänellä kenties jotain ylimääräisiä asennuksia. Tämä on erityisen tärkeää, kun asennetaan kalliita ohjelmia jotka vaativat lisenssin toimiakseen. Näkemällä nämä tiedot voidaan ottaa yhteys työntekijään ja tiedustella ohjelman tarpeellisuutta. Jos tarvetta sille ei ole, voidaan lisenssi lakkauttaa tai siirtää toiselle työntekijälle.

Projektissa käytettiin myös lukuisia eri työkaluja, joista puhuttiin myös aikaisemmin, mutta seuraavaksi arvioidaan hieman niiden soveltuvuutta. Sovelluksen pohjana käytetty Node.js-ympäristö soveltui tähän projektiin erittäin hyvin. NMP:n ansiosta moduulien asennus ja käyttö oli helppoa, eivätkä pitkätkään JSON-tiedostot hidastaneet juurikaan sen toimintaa. Myös laajan moduulivalikoiman ansiosta työkalut pysti valitsemaan varsin vapaasti. Itse rajapinnan rakentamiseen käytetty Express.js oli myös toimiva valinta. Sen määrittely ja käyttö oli yksinkertaista, ja se salli erittäin vapaan rakenteen käyttämisen. Muut rajapintakehykset tuskin olisivat tarjonneet mitään sellaista toiminnallisuutta, jota ei Expressistä löydy. Rajapintaa tukevat kirjastot toimivat myös hyvin dokumentointia lukuunottamatta. ApiDoc.js vaati kattavaan dokumentointiin huomattavia määriä kirjoitusta, koska automaattisesti generoidut dokumentit eivät olleet riittäviä. Se kuitenkin ajoi asiansa, kun siihen panosti aikaa. Muut kirjasotot kuten Passport ja Winston toimivat

moitteetta ja toteuttivat vähällä vaivalla sen toiminnon, jota niiltä odotettiinkin. Ainut tärkeä työkalu, jota ei pysty kommentoimaan, ovat tietenkin luvussa 4.6 mainitut testaustyökalut. Tämä on myös projektin suurin epäkohta ja voi aiheuttaa jatkokehittäjille päänvaivaa.

Kokonaisuutena projekti oli kuitenkin varsin onnistunut, sillä kaikki sille asetetut tavoitteet saatiin täytettyä huolimatta siitä, että projektin suunta muuttui hyvinkin radikaalisti sen aikana. Sovelluksesta ei myöskään löytynyt mitään merkittäviä bugeja sisäisen testauksen aikana. Projektin päätyttyä sovellus jäi tilaan, jossa sen täyttä käyttöönottoa suunniteltiin. Sen käyttämiseksi järjestetään viraston sisäisiä koulutustilaisuuksia ja kokouksia, jossa keskustellaan siihen liittyvistä asioista, kuten ylläpitovastuista ja muista käytännön asioista. Tämä ei kuitenkaan enää kuulunut tähän projektiin, joten käyttöönoton sujuvuutta tai laajempia käyttäjäkokemuksia ei pysty arvioimaan.

5 Yhteenveto

Tämä työn tarkoituksena oli tutustua REST-periaatteeseen ja erityisesti sen rakentamiseen paljon suosiota viime vuosina nauttineella Node.js-ympäristöllä. Tarkastellaan hieman aluksi mitä REST konseptina sisältää. Monet sen rajoitteista ovat hyvinkin yksinkertaisia, ja ne tähtäävät parempaan ja helpompaan web-kehitykseen.

Tämän jälkeen tutustuttiin Node.js-ympäristöön, sen moduuleihin sekä muutamaan rajapintakehykseen. Saatiin selville Node.js-ympäristön vahvuuksia sekä toimintaperiaatteita sekä pohdittiin, mikä tekee siitä alustana hyvän ja suositun. Sen vahvuudet piilivät paljolti sen suorituskyvyssä sekä erittäin laajassa kehysten ja kirjastojen tarjonnassa, jotka on helppo asentaa sen omalla NPM-työkalulla.

Seuraavaksi esiteltiin Helsingin kaupunkiympäristölle rakennettua web-rajapintaa, jossa keskityttiin puhumaan sen tavoitteista sekä pääpiirteittäin kertomaan, millainen siitä tuli. Tarkasteltiin myös sitä, kuinka rajapinta oli onnistunut tavoitteessaan seurata REST-rajoituksia. Sen lisäksi otettiin vielä lähempään vilkaisuun muutaman projektissa käytetyn lisäkirjaston, jotka täydentävät rajapinnan toiminnallisuutta ja ovat olennainen osa erityisesti Node.js-ympäristössä, jonka suosio perustuu paljolti niille.

Viimeisenä arvioitiin työkaluvalintojen osuvuutta sekä katsottiin, mitä projektista lopulta tuli. Myös sovelluksen onnistumista pohdittiin lyhyesti ja mietittiin sen mahdollista tulevaisuutta.

Lähteet

Ali Syed Bsarat, Beginning Node.js Apress, 2014.

Fielding Roy Thomas, Architectural Styles and the Design of Network-based Software Architectures. Kalifornian yliopisto, Irvine. 2000.

Doglia Fernando, REST API Developement with Node.js, Apress 2015.

Hoffmann Jay. 2017. SOAP And REST At Odds. Verkkodokumentti. <https://thehistoryoftheweb.com/soap-rest-odds/>. Luettu 20.02.2019.

Nole Tim. 2018. Node.js and Web Frameworks for 2019. Verkkodokumentti. <https://blog.checklyhq.com/Node.js-js-api-and-web-frameworks-for-2019/>.Luettu 23.02.2019.

Kili Aaron, 2019. 14 Best Node.jsJS Frameworks for Developers in 2019. Verkkodokumentti. <https://www.tecmint.com/best-Node.jsjs-frameworks-for-developers/>. Luettu 25.02.2019.

Chandryan Parmod, 2017. How Node.Js Single Thread mechanism Work? Understanding Event Loop in Node.jsJs. Verkkodokumentti. <https://codeburst.io/how-Node.js-js-single-thread-mechanism-work-understanding-event-loop-in-Node.jsjs-230f7440b0ea>. Luettu 03.03.2019.

Rouse Margaret. REST (REpresentational State Transfer). Verkkodokumentti. <https://searchmicroservices.techtarget.com/definition/REST-representational-state-transfer>. Luettu 04.03.2019.

Santamaria Maso Rolando, 2018. NO, you most probably don't need Express in your Node.js REST API. Verkkodokumentti. <https://medium.com/sharenowtech/there-are-expressjs-alternatives-590d14c58c1c>. Luettu 06.03.2019.

Sandoval Kristopher, 2017. 13 Node.js Frameworks to Build Web APIs. Verkkodokumentti. <https://nordicapis.com/13-Node.js-frameworks-to-build-web-apis/>. Luettu 20.03.2019.

Paresh Sagar, 2018. Why Node.js is so popular amongst developers? Verkkodokumentti. <https://www.socpub.com/articles/why-Node.jsjs-so-popular-amongst-developers-16039>. Luettu 21.03.2019.

Node.js. Verkkodokumentti. < https://en.wikipedia.org/wiki/Node.js >. Luettu 11.04.2019.

Patil Manisha, REST Architectural Elements and Constraints. Verkkodokumentti. <http://mrbool.com/rest-architectural-elements-and-constraints/29339>. Luettu 12.04.2019.

Morris Richard, 2010. Roy Fielding Geek of the week. Verkkodokumentti. <https://www.red-gate.com/simple-talk/opinion/geek-of-the-week/roy-fielding-geek-of-the-week/>. Luettu 15.04.2019.

Zaza Samuele, 2016. Test a Node RESTful API with Mocha and Chai. Verkkodokumentti. < https://scotch.io/tutorials/test-a-node-restful-api-with-mocha-and-chai> Luettu 16.04.2019.