REST-rajapinnan suunnittelu ja toteutus OpenAPI specification (OAS 2.0) avulla

Case: Trafin avoin ajoneuvodata

HAMK
HÄMEEN AMMATTIKORKEAKOULU
HÄME UNIVERSITY OF APPLIED SCIENCES

Ammattikorkeakoulututkinnon opinnäytetyö

Tietojenkäsittelyn koulutusohjelma

Visamäki, syksy 2017

Petteri Saarikko

HAMK
HÄMEEN AMMATTIKORKEAKOULU
HÄME UNIVERSITY OF APPLIED SCIENCES

TIIVISTELMÄ

Tietojenkäsittelyn koulutusohjelma
Visamäki

Tekijä Petteri Saarikko **Vuosi** 2017

Työn nimi REST-rajapinnan suunnittelu ja toteutus OAS 2.0 avulla

Työn ohjaaja Erkki Laine

TIIVISTELMÄ

Opinnäytetyössä tarkastellaan REST-rajapintojen suunnittelua ja toteutusta OpenAPI specification-version 2.0 avulla. Työssä selvitetään mitä seikkoja tulisi huomioida, kun rajapintoja suunnitellaan ja tulisiko suunnittelu tehdä ennen varsinaista toteutusta. OpenAPI specification-version 2.0 osalta tarkastellaan, mitä työkaluja se tarjoaa ja miten rajapintakuvaus luodaan tällä kehikolla. Työssä myös esitellään vaihtoehtoisia työkalukehikkoja rajapintojen suunnittelua varten.

Työn tilaaja oli Hämeen Ammattikorkeakoulun Älykkäät palvelut-tutkimusyksikkö. Käytännön sovelluksessa käytettiin Trafin julkaisemaa avointa ajoneuvodataa, jonka käyttämistä varten luotiin esimerkki rajapinta REST-periaatteiden mukaisesti.

Opinnäytetyö muodostaa pohjan jonka perusteella voi lähteä laajentamaan avoimen ajoneuvodatan rajapintaa haluttuun suuntaan niin rajapinnan metodien kuin esimerkiksi virheenkäsittelyn osalta.

Avainsanat REST-rajapinta, REST, OAS 2.0, OpenAPI 2.0

Sivut 40 sivua, joista liitteitä 7 sivua

HAMK
HÄMEEN AMMATTIKORKEAKOULU
HÄME UNIVERSITY OF APPLIED SCIENCES

Degree Programme in Business Information Technology
Visamäki

Author Petteri Saarikko **Year** 2017

Subject REST API design and development on OAS 2.0

Supervisor Erkki Laine

ABSTRACT

The thesis studies REST-API design and development on the OpenAPI specification-version 2.0. The thesis presents what different aspects should be considered when API is in the development stage and whether designing should be done before codebase execution. From the OAS 2.0 framework point of view the thesis observes what kind of tools it provides and how API is written on these tools. The thesis also presents alternative frameworks for the API design and development.

The Commissioner on this thesis is Häme University of Applied Sciences Intelligent Services Unit. In practice, the application is based on open source Trafi vehicle data and study example REST API was created based on this.

The study provides the baseline for the future API development on open source vehicle data in wanted direction. Development may for example include new query methods or improved error handling.

Keywords API's, REST, OAS 2.0, OpenAPI specification

Pages 40 pages including appendices 7 pages

SISÄLLYS

1 TERMIT ... 1

2 JOHDANTO ... 3

3 RAJAPINNAN SUUNNITTELU JA TYÖKALUT ... 4

3.1 Rajapinnan elinkaaren hallinta ... 5
3.2 Rajapintojen suunnittelutyökalut ... 7
3.2.1 Swagger eli OpenAPI Specification (OAS 2.0) ... 8
3.2.2 RAML ... 10
3.2.3 API Blueprint ... 10

4 OPENAPI SPECIFICATION ELI OAS 2.0 ... 12

4.1 OAS 2.0-tietotyypit ja kenttämäärityksen laajentaminen ... 14
4.2 OAS 2.0-kuvauskielen toimintaperiaate ... 15
4.3 Tietotyyppien viittausmääritykset ... 17

5 OAS 2.0-TYÖKALUT ... 20

5.1 OAS 2.0-editori ... 20
5.2 OAS 2.0-koodigeneraattori eli codegen ... 21
5.3 OpenAPI UI ... 22

6 CASE TRAFI ... 25

6.1 Rajapinnan suunnittelu ... 25
6.2 MOCK ohjelmistokoodi ... 26
6.3 Toteutus ... 27

7 YHTEENVETO ... 31

LÄHDELUETTELO ... 32

Liitteet
Liite 1 OAS 2.0-kenttämäärityksset
Liite 2 Trafin avoin ajoneuvodata rajapintakuvaus

1 TERMIT

OAS
OAS on lyhenne sanoista OpenAPI specification, joka tunnetaan myös vanhemmissa julkaisuissa nimellä Swagger. Normaalisti OAS-lyhenteessä ilmaistaan myös sen versionumero. (Open API initiative, 2016.)

OAI
OAI on lyhenne sanoista Open API initiative ja se on pääorganisaation nimi OAS-kehitysyhteenliittymälle (Open API initiative, 2016).

JSON
JavaScript Object Notation on avoin tiedonesitys standardi. JSON:ille on ominaista esittää data avain-arvopari (name/value pair) yhdisteenä. (IETF, 2014.)

JSON objekti
JSON objekti muodostuu yhdestä tai useammasta avain-arvoparista jotka on sijoitettu aaltosulkeiden sisälle. Objektia voidaan kutsua avain-nimen perusteella. (IETF, 2014.)

JSON lista (array)
JSON lista muodostuu yhdestä tai useammasta avain–arvoparista jotka on sijoitettu hakasulkeiden sisälle. Listat ovat normaalisti osa objektia ja käyttötapaukset syntyvät tilanteesta missä objektin sisällä esitetään listattavaa tai kertautuvaa tietoa. (IETF, 2014.)

YAML
YAML Ain’t Markup Language toimintaperiaate vastaa JSON:ia, mutta aalto- ja hakasulkeiden sijaan formaatissa käytetään sisennyksiä ja miinusmerkkiä kertomaan objektin tai listan aloituksesta. Formaatti on ihmiselle helpompilukuinen kuin JSON laajoissa materiaaleissa. (Ben-Kiki, 2009.)

Markdown on tekstin kirjoitustapa, jota voidaan sellaisenaan kääntää XHTML- tai HTML-muotoon. Alkuperäisenä ajatuksena on ollut synnyttää ihmiselle luettavampi tapa esittää natiivi HTML-sivun sisältö ilman HTML-tägejä. (Gruber, 2017.)

MSON (Markdown Syntax for Object Notation) on Markdownin sisällä käytettävä tapa esittää tietotyyppejä. Tietotyyppejä voivat olla array, enum tai object. (MSON Specification, 2016.)

Tietotyyppi
Ohjelmoinnin perusperiaatteita on vakioiden ja muuttujien tietotyyppien määrittämien. Rajapintakuvaus ei tuo tässä poikkeusta ja esiteltävien tietotyyppinen kuvaaminen on osa rajapintakuvauksen tekemistä. Tarkka tietotyyppimääritys mahdollistaa generoitavan lähdekoodin halutun käyttäytymisen.

Mediatyyppi (MIME) Ilmaisee viestin tai tiedoston sisällön muotoa. Erilaisia MIME-tyyppejä löytyy RFC-määrityksen alaisuudesta runsaasti. MIME-tyypin voi myös määrittää itse jolloin sillä kuvataan yrityksen itsenäisesti hallitsemaa tapaa kuvata viestisisältöä. Formaatti tässä tapauksessa on application/vdn.yritys.tiedostomuoto, jossa yritys vastaa tiedostomuodon standardoinnista. On yleistä, että MIME-tyyppejä yhdistetään toisiinsa kuten esimerkiksi XML- tai JSON yhdistettynä toiseen MIME-tyyppiin. (IANA, 2017.)

MOCK
Termillä viitataan simuloituun palveluun tai ohjelman osaan, joka matkii kehitettyä tuotantoversiota ja sen tarjoamia toiminteita. MOCK-palveluita käytetään yleisesti yksikkötestauksen yhteydessä ilmentämään kehitettävän ohjelmiston toimintaa tai siihen liittyvää dataa. MOCK-termin käyttö ei estä tuotantodatan käyttämistä. Tuotantodatan käyttö on yleistä käyttötapauksissa missä tietoa ei päivitetä tai muuteta. Yhteistä kaikille edellä kuvatuille on se, että MOCK-palvelut eivät ole tarkoitettu tuotanto- vaan testauskäyttöä varten.

Webhook, Longbow, Stream
Yhteistä edellä kuvatuille termeille on, että kyseessä on tekniikka, jonka avulla rajapinnasta voidaan tilata tapahtumia tai dataa. REST-arkkitehtuurin perusperiaatteisiin kuuluu asiakkaan esittämä kysymys palvelulta ja siihen liittyvä vastaus. Tilanteissa missä asiakas odottaa esimerkiksi kuittausta jokin tapahtuman valmistumisesta REST-arkkitehtuurin mukaisesti asiakas kysyy, onko em. tapahtuma valmistunut ja ns. vastuu kyselystä on asiakkaalla. Otsikon tekniikat mahdollistavat toiminteet joilla asiakas tilaa kuittauksen esimerkin tapahtuman valmistumista. Tällaisessa tapauksessa kuittausviestin lähetysvelvollisuus on palvelimella (palvelulla). Tekniikka vähentää turhien kyselyiden tarvetta ja voi toisaalta mahdollistaa suurienkin datamäärien siirtämisen yksittäisen pyynnön sisällä.

2 JOHDANTO

Opinnäytetyön kirjoitusajankohdan merkittävänä trendinä voidaan pitää ohjelmistokehityksen suuntausta, jossa kehitettävät ohjelmistot tuotetaan funktio- (Serverless) tai mikropalveluina. Tämänkaltainen arkkitehtuuri mahdollistaa ohjelmistojen kehittämisen ja suorittamisen pienemmissä itsenäisissä osissa. Siinä missä aiemmin on ollut erilliset fyysiset sovellus- ja tietokantapalvelimet ovat nämä korvaantuneet funktio- tai mikropalveluina, joiden avulla lähdekoodia suoritetaan pilvipalveluperiaatteiden mukaisesti. Erityyppisten rajapintapalveluiden yleistyessä ohjelmistot itsessään eivät välttämättä ole tuotettu oman, itsenäisesti kehitetyn koodin varaan vaan ne hyödyntävät toisen ohjelmiston osia tai jopa kokonaisia ohjelmia. Ohjelmistokehittämisen kannalta tämä avaa uusia mahdollisuuksia ja toisaalta myös haasteita.

Jotta ohjelmistot tai sen osat voivat välittää tietoa keskenään tarvitaan rajapintoja. Rajapinnat voivat olla ohjelmistojen omia rajapintoja, jotka ovat tarkoitettu esimerkiksi saman ohjelmointikielen sisäisesti käytettäväksi. Vastaavasti ohjelmisto voi tuottaa rajapintoja joiden avulla se voi joko noutaa tai tuottaa tietoa ohjelmiston ulkopuolella. Tapauksissa missä ohjelmisto tuottaa tietoa ulkoisille palveluille tiedonsiirto ohjelmistojenvälillä tapahtuu useasti http-protokollan avulla.

REST on yksi rajapintojen luonti- ja kulutustapa ja tässä opinnäytetyössä keskitytään kuvaamaan tämän rajapintatyypin suunnittelua ja toteutusta. Mikäli termit *http header, body* tai verbit *get*, *post*, *put* ja *delete* ovat vieraita suositellaan lukijan tutustumaan http-protokollan ja REST-rajapintojen yleiseen toimintaperiaatteeseen esimerkiksi näiden tietolähteiden kautta, (Jussilainen 2015) kappale 5-6 ja tai (Kankaanpää 2016) kappale 3. Molemmissa näistä opinnäytetöissä on käsitelty http-protokolla ja REST-arkkitehtuurin perusperiaatteet hyvin kiteytetyllä tavalla.

Opinnäytetyön pyrkimyksenä on vastata seuraaviin kysymyksiin. Mitä seikkoja rajapinnan suunnittelussa tulisi huomioida? Mitä hyötyä rajapinnan suunnittelusta on ennen sovelluksen ohjelmointia? Miten rajapinta kuvataan? Mitä työkaluja suunnitteluun ja toteutukseen on tarjolla?

Työn tilaajana toimii Hämeen ammattikorkeakoulun älykkäät palvelut-tutkimusyksikkö. Suunniteltavan rajapintatoteutuksen datalähteenä käytetään Trafin ajoneuvojen avointa dataa. Trafin avoin data noudattaa kansainvälistä Creative Commons (CC) 4.0-lisenssiehtomallia.

3 RAJAPINNAN SUUNNITTELU JA TYÖKALUT

Tietojenkäsittelyssä termillä rajapinta viitataan yleensä ohjelmointirajapintaan (API), jonka avulla määritellään ohjelmiston tai palvelun tarjoamista toisille järjestelmille, palveluille tai ohjelmistoille. Rajapinta voi olla datarajapinta jolloin puhutaan tiedonvälittämisestä eri ohjelmistojen tai rajapintojen välillä, tai se voi olla toiminnallinen rajapinta joka tarjoaa esimerkiksi koneoppimisen (Machine learning), konenäköön tai laskentaan liittyviä palveluita. (API 2017.)

Yritykset, yhteisöt ja julkishallinto tarjoavat rajapintoja tai niihin liittyvää dataa mitä kuka tahansa voi hyödyntää omassa sovelluksessaan. Verkossa on käytettävissä useita portaaleita, joiden kautta pääsee tutustumaan em. rajapintoihin ja niiden tuottamaan dataan tai palveluun. Esimerkkeinä portaaleista mainitaan https://www.programmableweb.com/ ja https://www.avoindata.fi/

Useassa tapauksessa rajapintasuunnittelua ei tehdä ennen varsinaisen ohjelmistologiikan koodaamista. Tämä saattaa aiheuttaa tilanteen missä ohjelmiston tuottamaa rajapintaa ei dokumentoida lainkaan. Ilman rajapinnan dokumentointia sen käyttäminen on vähintään hankalaa, jos ei peräti mahdotonta. Tästä syystä rajapinta tulisi poikkeuksetta dokumentoida sellaiselle tasolle, että rajapinnan asiakas eli kehittäjä pystyy kuluttamaan sen tuottamia palveluita itsenäisesti.

Suunnittele ensin -tekniikka (Desing First) nimensä mukaisesti lähestyy toteutettavaa rajapintaa suunnittelun kautta. Suunnitelman avulla rajapinnasta luodaan konekielinen rajapintakuvaus, jota voidaan hyödyntää mm. dokumentointina kehittäjiä tai muita rajapintaa kuluttavia osapuolia varten. Rajapinnan konekielinen kuvaus mahdollistaa muitakin toiminteita joita käsitellään myöhemmin tässä dokumentissa. (Vadudevan 2017.)

Rajapintamäärityksen kohteena on URL eli polku ja siihen liittyvät http-toimintaa kuvaavat verbit kuten *get*, *post*, *put* ja *detele*. Rajapintamäärityksessä kuvataan myös rajapinnan käytössä olevat tietotyypit ja niiden ominaisuudet kuten myös tiedon välitystapa ja muoto. Kuvauksessa on myös mukana vastauksenkäsittely, joka kuvaa niin onnistuneet kuin epäonnistuneet http-tilakoodit tietotyyppeineen. Rajapintakuvaus voi myös sisältää tunnistukseen ja käyttöoikeuksiin liittyviä toiminteita, mutta nämä eivät ole rajapinnan kuvauksen kannalta pakollisia. (Koski, 2017.) (Moilanen, 2017.)

Missä vaiheessa suunnittele ensin -tekniikkaa kannattaa käyttää? Tähän kysymykseen ei ole yksiselitteistä tai oikeaa vastausta. Vaikuttavia seikkoja ovat seuraavat.

Onko rajapinnan suunnittelija ja varsinaisen ohjelmistologiikan koodin kirjoittaja yksi ja sama henkilö? Minkä kokoisesta organisaatiosta on kysymys ja miten sen resursointi on järjestetty? Tuotetaanko rajapinta sisäiseen vai ulkoisen käyttöön? Minkälainen elinkaari rajapinnalla ajatellaan olevan? Onko tiedossa, että rajapintaan on tulossa uusia metodeja, eli polkuja? Kuka on rajapinnan asiakas eli kehittäjä? Minkälainen ryhmä määrittää rajapinnan toimintatavoitteita? (Koski 2017.) (Moilanen 2017.)

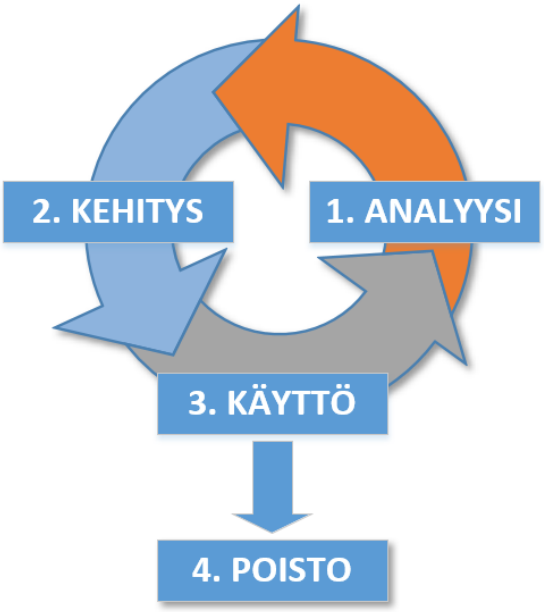
Suunnittele ensin -tekniikkaa kannattaa hyödyntää, mikäli rajapinta tuottaa julkisesti saatavilla olevan palvelun ja sen käyttö eli kuluttaminen on sidoksissa kehittäjäkokemukseen. Kehittäjät kannattaa tuoda mukaan suunnitteluun hyvin aikaisessa vaiheessa ja suunnittele ensin -tekniikka tarjoaa tälle hyvän toimintaperustan. Kehittäjän kannalta on tärkeää, että rajapinta toimii arvolupauksen määrittämällä tavalla. Päivitystilanteissa olemassa olevat toiminteet eivät saa lakata toimimasta hallitsemattomasti. Suunnittele ensin -tekniikkaa kannattaa käyttää myös siinä tapauksessa, että määritystä halutaan tarkastella tai määrittää sellaisen ryhmän avulla, joka ei välttämättä teknisesti tunne rajapinnan toimintaperiaatteita. Toimintatavan etuja ovat myös dokumentaation automaattinen syntyminen ja mahdollisuus käyttää rajapintakuvausta MOCK-palveluna. Tässä tapauksessa kuvauksesta muodostetaan http-palvelu, joka esittää olevansa rajapinta ja tuottaa kuvauksen mukaiset toiminteet MOCK-logiikan määrittämällä tavalla. (Koski 2017.) (Moilanen 2017.)

Suunnittele ensin -tekniikkaa ei välttämättä kannatta käyttää, jos rajapinnan suunnittelija ja asiakas ovat sama henkilö tai rajapinnan dokumentoinnille ei nähdä muuten tarvetta esimerkiksi resurssien puutteiden takia (Vadudevan 2017).

3.1 Rajapinnan elinkaaren hallinta

Olennainen osa suunnittele ensin -tekniikan käyttämisessä on pohtia rajapinnan tulevaa elinkaarta laajempana kokonaisuutena. Suunnitteluvaiheessa tehdyillä päätöksillä on pitkävaikutteisia seurauksia ja tästä syystä suunnitteluun kannattaa käyttää riittävästi resursseja. Huolimattomasti tehty määritys aiheuttaa turhia kustannuksia ja voi pahimmillaan aiheuttaa tilanteen, ettei tuotettua rajapintaa käytetä. Kuvassa 1 esitetyn periaatteen perusteella rajapinnalle syntyy elinkaari, jossa jokainen kierros voi tuottaa rajapinnasta tai sen osasta uuden version. Versiointi mahdollistaa uusien toiminteiden lisäämisen tai olemassa olevien päivittämisen vaikuttamatta edellisen version toimintaan. Edellä oleva mahdollistaa rajapinnan kehittymisen toivottuun suuntaan vaarantamatta käytössä olevia toiminteita ja tätä kautta määriteltyä rajapinnan arvolupausta. (Koski 2017.) (Moilanen 2017.)

Rajapinnan elinkaaren voi jakaa neljään eri vaiheeseen kuvan 1 mukaisesti.



Kuva 1. Rajapinnan elinkaaren hallinta (oma piirros)

Ensimmäisessä eli analyysi- ja suunnitteluvaiheessa tarkastellaan mm. seuraavia kysymyksiä. Miksi API luodaan, mihin sitä käytetään ja mitä se tuottaa? Kuka tai ketkä rajapintaa oletettavasti käyttävät ja miten rajapintaa käytetään tai kulutetaan? Ovatko käyttäjinä ohjelmistot vai ihmiset vai molemmat? Mitä toimintoja rajapinnan on tarkoitus pystyä tekemään nyt ja tulevaisuudessa? Millä tasolla on omistavan organisaation sitoutuminen resurssien, kustannusten ja osaamisen osalta? Edellä oleva kiteytettynä yhteen lauseeseen. Mikä on rajapinnan arvolupaus? (Koski 2017.) (Moilanen 2017.)

Toisessa eli kehitysvaiheessa tarkastellaan teknisiä kysymyksiä ja ryhdytään luomaan itse toteutusta. Miten rajapinnan pääsynhallinta ja tietoturva on tarkoitus toteuttaa? Miten asiakkaiden tunnistus ja käyttöoikeushallinta toteutetaan? Mikä on rajapinnan arkkitehtuurillinen toteutustapa, esimerkkeinä REST, SOAP, MQTT? Pitääkö rajapinnan pystyä keskustelemaan asiakkaan suuntaan tilausperiaatteella, eli webhook- longbow- tai stream- tekniikoiden käyttäminen? Tarjotaanko asiakkaille rajapinnan SDK-paketteja ja mitä ohjelmistokieliä tuetaan? Millä ohjelmointikielellä ja/tai API-kehikolla palvelu toteutetaan? Mitä käyttöjärjestelmää ja tietokantaratkaisua käytetään? Missä rajapintaa suoritetaan? Serverless, Docker-kontit, SaaS, PaaS vai itse tuotettu palvelinalusta tai jotain muuta? Käytetäänkö rajapinnan tuottamisessa jotain hallintatyökaluja tai välittäjäpalveluita? Käytetäänkö API-suunnittelukehikkoa kuten esimerkiksi OAS, RAML tai Api

Blueprint. Rajapintaa koskevat tekniset päätökset syntyvät siis tässä vaiheessa. (Koski 2017.) (Moilanen 2017.)

Kolmannessa eli käyttövaiheessa tarkastellaan seuraavia asioita. Tarkastellaan rajapinnan tuottamaa dataa, joka voi olla esimerkiksi lokitietoja, hallinta- tai välittäjäpalvelun tuottamaa dataa rajapinnan käytöstä. Lokitiedosta tarkastellaan, mitä metodeja on käytetty ja kuinka paljon. Onko jotain metodeja mitä ei käytetä lainkaan? Ketkä metodeja ovat käyttäneet? Minkälaisia datamääriä käyttöön liittyy niin kyselymäärissä kuin myös siirrettynä datana? Voidaanko lokidatasta päätellä rajapinnan väärinkäyttöä käyttöoikeus tai datamäärien perusteella? Haastatellaan rajapinnan käyttäjiä ja selvitetään, vastaako tuotettu dokumentaatio ja metodit asiakkaiden tarpeita? Saavatko asiakkaat tarvitsemansa loppukäyttäjätuen käyttämiensä kanavien kautta? Tuottaako rajapinta tarvittavat toiminteet? Pitäisikö metodeja lisätä, poistaa tai muokata? Tässä vaiheessa siis tarkastellaan rajapinnan toimintaa ja tehdään päätöksiä rajapinnan kehityksen kannalta. Päätöksien perustella palataan analyysivaiheeseen ja tehdään tarvittavia päätöksiä ja toimenpiteitä. (Koski 2017.) (Moilanen 2017.)

Neljännessä eli eläköitymisvaiheessa tarkastellaan ja toteutetaan seuraavia asioita. Kommunikoidaan asiakkaille tieto rajapinnan toiminnan loppumisesta. Tiedottaminen ja kommunikointi koskevat myös päivitystilanteita tai tilanteita joissa rajapinta ei ole saatavilla jonkin teknisen syyn takia. Eläköitymiseen voivat liittyä liiketaloudelliset syyt joita voi olla esimerkiksi rajapinnan vähäinen käyttö tai kilpailijan parempi tai käytetympi tuote. Eläköityminen voi tulla kyseeseen myös turvallisuussyiden takia tai organisaatiossa tapahtuneiden resurssi- tai linjausmuutoksien vuoksi. Eläköityminen koskee myös tilanteita missä jokin versio poistetaan käytöstä. Eläköityminen ei siis välttämättä koske koko rajapinnan käytöstä poistoa. (Koski 2017.) (Moilanen, 2017.)

3.2 Rajapintojen suunnittelutyökalut

Suunnittele ensin -tekniikan hyödyntämiseen on olemassa monia eri työkalukehikkoja. Riippumatta käytettävästä kehikosta, muodostetaan sen avulla kuvaustiedosto, joka on tarkoitettu koneluettavaan muotoon. Tästä tiedostosta on yksinkertaista lähteä jatkojalostamaan rajapintaa haluttuun suuntaan. Automaattisesti tuotetun dokumentaation lisäksi yhteistä kehikoille on valmiin ohjelmistokoodin tuottaminen kehikon reunaehtojen puitteissa. Valmiilla ohjelmistokoodilla tarkoitetaan tässä kontekstissa esim. rajapinnan MOCK-ohjelmiston koodirunkoa, joka tuottaa kuvatun rajapinnan polut ilman varsinaista toimintalogiikkaa. Erotukset kehikoiden välillä liittyvät käytettävän kuvauskielen editoreihin ja tuettuihin ohjelmointikieliin. Yleisimmin käytössä olevat kehikot ovat seuraavat.

OAS, vanha nimi Swagger, RAML ja API Blueprint. Lisäksi on suuri joukko muita kehikoita kuten esimerkiksi Mashery I/O, Google API Discovery Service, WADL, APIMATIC format, HAR 1.2, Rapid-ML.

Mikään ei estä käyttämästä suunnittelukehikkoa toiminnassa olevan rajapinnan kuvaamiseen. Tällaisessa tapauksessa kehikon tuotos on dokumentaatio ja toteutuksesta riippuen jonkinlainen MOCK-ympäristö, joka mahdollistaa yksikkötestaamisen kohti rajapintaa. Vaihtoehtoisesti dokumentointia voidaan käyttää suoraan kohti sovellusta, jolloin dokumentointia toimii rajapinnan käyttöliittymänä. Ohjelmointikielet tarjoavat eri kirjastojen avulla tapoja tuottaa rajapintakuvaus myös suoraan kirjoitetusta lähdekoodista. Näin toimimalla voidaan kirjoittaa rajapinnan ohjelmistologiikka ja dokumentointi yhtä aikaa.

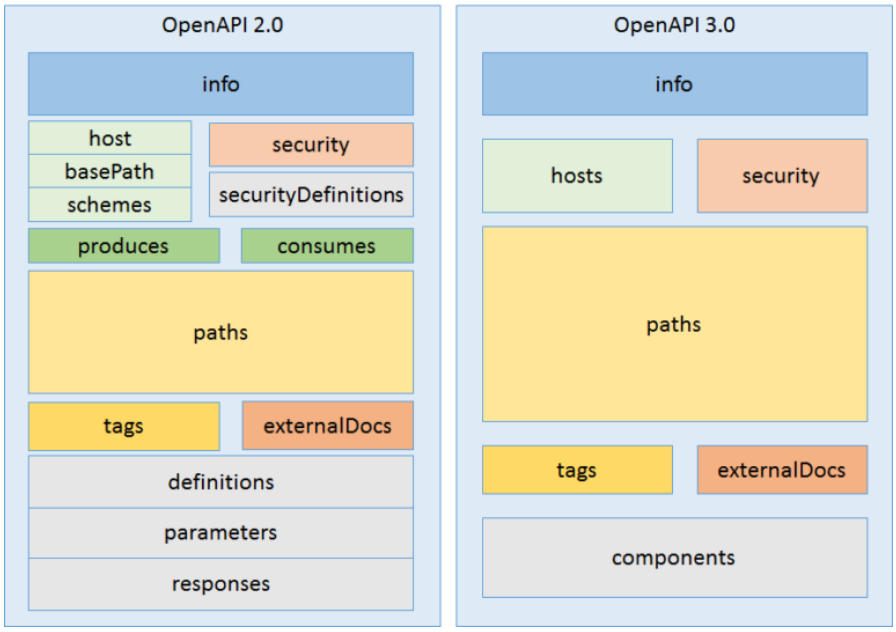
3.2.1 Swagger eli OpenAPI Specification (OAS 2.0)

Johtuen vuoden 2016 - 2017 aikana tapahtuneesta kehikon versio- ja nimimuutoksesta tässä dokumentissa puhutaan nimikkeistä Swagger, Open API initiative tai OpenAPI specification ja OAS-versionumero. Open API initiative on yhteenliittymän nimi jota voisi kuvata hallinta-organisaatioksi. OpenAPI specification on kuvauskielen nimi ja oli vanhalta nimeltään Swagger. OpenAPI specification lyhennetään muotoon OAS ja sen perässä mainitaan normaalisti käytettävän version numero. Dokumentissa viitataan eri versioihin muodossa OAS 2.0 tai OAS 3.0. (Open API initiative 2016.)

Wordnik aloitti kehittämään vuonna 2010 Swagger rajapinnan kuvauskieltä ja siihen liittyviä työkaluja. Wordnic kertoo olevansa maailman suurin verkossa toimiva Englannin kielen sanakirja. Sen tarve REST-rajapintojen kuvaamiselle syntyi sisäisen toiminnan kehitystarpeista. Vuonna 2011 Wordnic siirsi Swagger-kehikon lisensoinnin avoimen lähdekoodin alaisuuteen. Vuonna 2015 yritys nimeltä SmartBear hankki pääomistuksen Rewerb Tech- nimisestä yrityksestä, joka omistaa myös Wordnicin. Viides päivä marraskuuta 2015 Smartbear yhteistyössä 3Scalen, Apigeen, Capital Onen, Googlen, IBM:n, Microsoftin ja Paypalin kanssa sopivat yhteenliittymästä, joka tunnetaan nimellä Open API initiative. Kyseessä on avoimeen lähdekoodiin perustuva yhteenliittymä, joka toimii Linux foundationin alaisuudessa. Samassa yhteydessä Smartbear luovutti Swaggerin käyttöoikeuden Open API initiative -yhteenliittymälle. Käyttöoikeuden myötä ohjelmistokehikon nimikin muuttui Swaggerista OAS-muotoon. (Open API initiative 2016.)

Dokumentin kirjoitushetkellä OAS-määrityksestä luodaan versiota 3.0. Suurimmat eroavaisuudet 2.0- ja 3.0-kuvauksien välillä syntyvät kuvauskielen kenttämäärityksistä. Kuvasta 2 voidaan havaita, että versiossa 2.0 on huomattavasti suurempi määrä objekteja verrattuna ver-

sioon 3.0. Johtuen OAS 3.0-määrityksen keskeneräisyydestä keskitytään tässä dokumentissa kuvaamaan version OAS 2.0-toimintaa.



Kuva 2. OAS 2.0 vs. OAS 3.0 (Open API initiative 2016.)

OAS 2.0 MIME-tyyppi on text/plain;charset=utf-8 tai application/json. Rajapintamäärityksen kuvauskielen formaatti on JSON, joka mahdollistaa kuvauksen luettavuuden niin koneiden kuin ihmisienkin kannalta. Monimutkaisemmissa rakenteissa JSON-tiedostomuoto voi osoittautua ihmiselle vaikealukuiseksi ja tästä syystä OAS tukee myös YAML-formaattia. YAML-formaatti muistuttaa JSON:ia, mutta on helpompilukuista koska siellä ei käytetä JSON:sta tuttuja aalto- ja hakasulkeita eriyttämään eri objektien ja listojen osia. OAS 2.0-tietotyypit voidaan kuvata JSON- tai XML- perusteisesti.

OAS 2.0-kehikkoon kuuluu olennaisena osana editori millä rajapintakuvaus luodaan ja UI http-sovellus millä kuvauksen voi esitellä. UI-sovellus mahdollistaa myös kuvauksen MOCK-testaamisen. UI-komponenttia voidaan käyttää myös html-perusteisena dokumentaatiopisteenä tai käyttöliittymänä kohti rajapintaa. Näiden lisäksi kokonaisuuteen kuuluu koodigeneraattori, jonka avulla luotu määritys voidaan kääntää halutulle ohjelmointikielelle. Generointityökaluja on saatavilla sekä palvelin-, että asiakas SDK-lähdekoodi muodostusta varten. Asiakas-SDK tarkoittaa tässä kontekstissa sitä, että halutulle ohjelmointikielelle muodostetaan oma kirjasto, joka sisältää kuvatut rajapinnan palvelut. (IO 2017.) OAS 2.0-kuvauskieltä ja siihen liittyviä objektimäärityksiä (kuva 2) voidaan laajentaa omien tarpeiden mukaan esim. Scalalla, HTML5:llä tai esimerkiksi Javalla (Sandoval 2017).

3.2.2 RAML

RAML (Restful API modeling language) on esitelty vuonna 2013 RAML-työryhmän toimesta. Ryhmän takaa löytyy yrityksiä kuten Mulesoft, AngularJS, PayPal, Cisco (RAML 2017).

RAML määrittää mediatyypin application/raml+yaml ja tämä MIME-tyyppi ei ole IANA-rekisteröity. RAML-kuvauskielen ulkoasu vastaa OAS-esitystapaa. Huolimatta samantyylisestä esitystavasta RAML- ja OAS-määritykset eivät ole keskenään yhteensopivia. RAML-tietotyypit voidaan kuvata JSON- tai XML-perusteisesti. (IANA 2017.)

Työkalumielessä RAML tarjoaa samat ominaisuudet kuten OAS 2.0. Saatavilla on editoreita ja työkaluja jotka muodostavat dokumentaatiota määritellyn rajapintakuvaksen perusteella. Saatavilla on myös koodigeneraattoreita eri ohjelmointikieliä varten. RAML-työkalupakkiin kuuluu myös kääntäjä, jonka avulla voidaan OAS-määritys muuntaa RAML-muotoiseksi. RAML työkaluja on saatavilla niin avoimen lähdekodin periaatteella kuten myös eri yritysten tuottamina maksullisina ohjelmistoina ja palveluina. (RAML 2017.)

21.4.2017 julkaistun tiedotteen mukaan Mulesoft on liittynyt OpenAPI initiative -yhteenliittymään. Tässä vaiheessa on aikaista arvioida mitä tulee tapahtumaan RAML-kuvauskielelle pidemmällä aikavälillä. Korkealla todennäköisyydellä tämä kuitenkin tarkoittaa sitä, että OAS vahvistaa standardimaista statustaan rajapintojen kuvauskielenä. (Swagger 2017.)

3.2.3 API Blueprint

Apiary on toiminut API Blueprintin vastuullisena kehittäjänä avoimen lähdekoodin MIT-lisenssin alaisuudessa vuodesta 2011 lähtien. Kehittäminen on tapahtunut yhteistyössä kehittäjäyhteisön kanssa ja Apiaryllä on selkeästi suurempi työkalupakki verrattuna OAS:in tai RAML:in. Kehitystyötä hallintaan Githubissa toimivan RFC- (Request for change) kanavan kautta. Kuka tahansa voi siis ehdottaa uuden toiminteen lisäystä tai muutosta API Blueprint määritykseen. Apiary ilmoitti 18.1.2016, että se ryhtyy tukemaan OAS-määrittelyä käytössä olevan API blueprint kuvauksen lisäksi. 10.02.2017 päivätyllä sopimuksella Oracle on ostanut Apiaryn. (API blueprint, 2017)

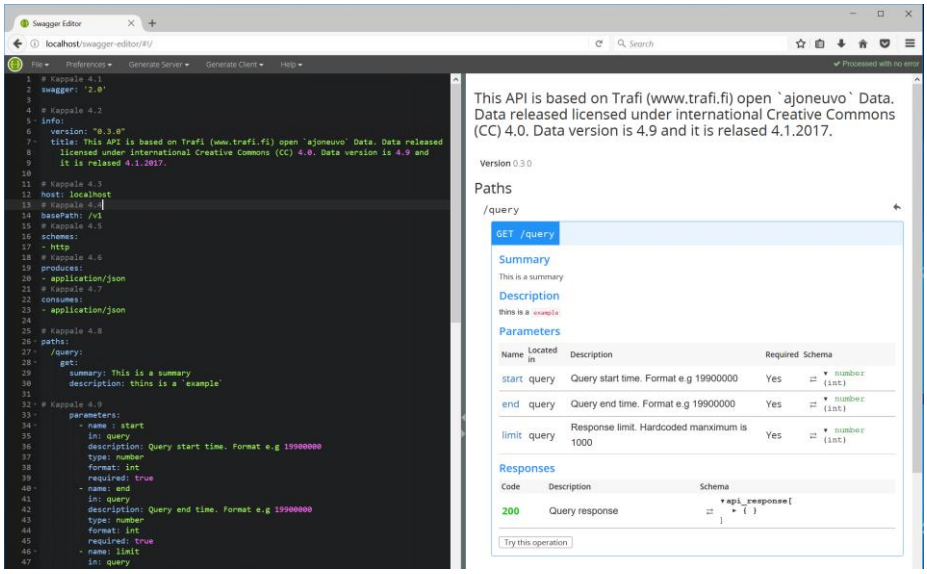
API Blueprint määrittää MIME-mediatyypin text/vnd.apiblueprint joka ei ole IANA-rekisteröity. Kuvauskieli noudattaa Markdown-syntaksia lisättynä Githubin määrittämään ja käyttämään Markdown-syntaksiin. API Blueprintin tietotyypit kuvataan MSON-muodossa. (IANA 2017.)

Merkittävin ero OAS:in ja API blueprintin välillä on kuvauskielen syntaksi YAML/JSON vastaan Markdown. API Blueprint ei tarjoa webselaimessa toimivaa editori vaihtoehtoa lainkaan. Vaihtoehtoisesti tarjotaan tekstieditoreille kuten Notepad++ tai Atom-liitännäisiä (plugin), joiden avulla kuvauksen luominen on mahdollista. Muuten työkalupuolelta löytyy kattava valikoima konverttereita, dokumentointityökaluja, MOCK-palvelinohjelmistoja. Verrattuna OAS:iin ja RAML:iin palvelin- tai asiakas SDK-kääntäjiä ei ole samalle määrälle ohjelmointikieliä. (API blueprint 2017.)

4 OPENAPI SPECIFICATION ELI OAS 2.0

Oletusarvoisesti OAS 2.0-editori tuottaa YAML-formaatissa rajapintakuvauksen. Määrityksessä esiintyvät objektit, kentät ja niiden parametrit määrittävät rajapinnan toiminnan (Liite 1). Myöhemmin kuvatuissa esimerkeissä esitysmuodon käyttäminen on pakollista ja tarjolla olevat editorit edellyttävät sisennyksien käyttämisestä (Kuva 3). Sisennyksien käyttämättä jättäminen aiheuttaa virheen ja kuvauksen toimimattomuuden. OAS 2.0-kuvauksen voi kirjoittaa myös JSON-muodossa. Näin kirjoitettu kuvaus edellyttää aalto- ja hakasulkujen käyttämistä. Swagger.io kehittäjäyhteisön tarjoamat editorit sisältävät kuvauksen syntaksin tarkastuksen kuten myös telecensen eli ehdotuksen käytettävästä kenttänimestä perustuen kirjoitettuihin muutamiin kirjaimiin. Vastaavat toiminteet ovat tuttuja mistä tahansa ohjelmointi IDE:stä jonka avulla kirjoitetaan lähdekoodia. Halutessaan kehittäjä voi kirjoittaa rajapintakuvausta esim. Notepad++ tekstieditorilla. Tässä tapauksessa rajapintakuvauksen kirjoittaja ei saa suoraan visuaalista rajapintakuvauksen ilmentymää kirjoitetun kuvauksen perustella kuten kuvassa 3 on esitetty tai tietoa onko kirjoitettu syntaksi oikein. (IO 2017.)

Kuva 3 ilmentää OAS 2.0-editoria, jossa rajapinnan toiminnankuvaus on kirjoitettu kuvan vasempaan lohkoon YAML-muodossa. Oikeassa lohkossa on rajapinnan dokumentointi-ilmentymä kirjoitetun kuvauksen perusteella. Kuvassa 3 vasen lohko on käyttäjän kirjoittama kuvaus ja oikea lohko muodostuu automaattisesti vasemman lohkon määrityksen perustella.



Kuva 3. Esimerkki 1 rajapintakuvauksesta. (Kuvakaappaus.)

Kuvassa 4 on esitetty JSON- ja YAML-kuvauksen syntaksien eroavuus. Kuvauksien tietosisältö on siis sama. Eroavuus syntyy esitystavasta ja sen määrittämästä esitysmuodosta. JSON-kuvauksessa aalto- ja haka-sulkeiden käyttäminen on pakollista, kun taas YAML-kuvauksessa si-sentämistä käytetään kuvaamaan esimerkiksi objektin parametrien määrityksiä. (OpenAPI Specification version 2.0 2017.)

```
{
  "title": "Swagger Sample App",
  "description": "This is a sample server Petstore server.",
  "termsOfService": "http://swagger.io/terms/",
  "contact": {
    "name": "API Support",
    "url": "http://www.swagger.io/support",
    "email": "support@swagger.io"
  },
  "license": {
    "name": "Apache 2.0",
    "url": "http://www.apache.org/licenses/LICENSE-2.0.html"
  },
  "version": "1.0.1"
}
```

```
title: Swagger Sample App
description: This is a sample server Petstore server.
termsOfService: http://swagger.io/terms/
contact:
  name: API Support
  url: http://www.swagger.io/support
  email: support@swagger.io
license:
  name: Apache 2.0
  url: http://www.apache.org/licenses/LICENSE-2.0.html
version: 1.0.1
```

Kuva 4. JSON vs. YAML. (OpenAPI Specification version 2.0 2017.)

4.1 OAS 2.0-tietotyypit ja kenttämäärityksen laajentaminen

Taulukko 1 OAS-tietotyypit. (OpenAPI Specification version 2.0 2017.)

Yleisnimi	OAS-kuvaustyyppi	Formaatti
integer	integer	int32
long	integer	int64
float	number	float
double	number	double
string	string	
byte	string	byte
binary	string	binary
boolean	boolean	
date	string	date
dateTime	string	date-time
password	string	password

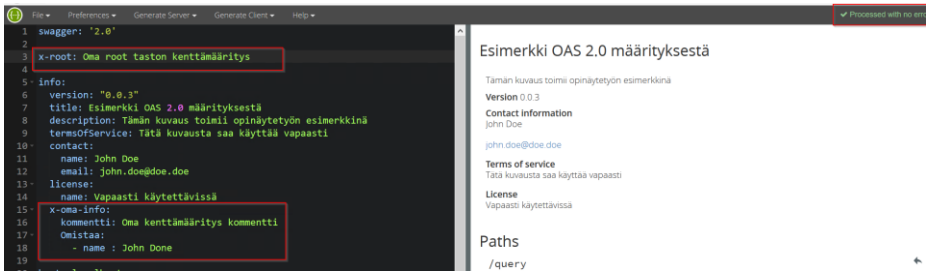
Taulukossa 1 esitellään kuvauskielen tukemat tietotyypit. OAS 2.0 noudattaa tietotyyppien osalta JSON-schema draftia nro 4.

Taulukon ”yleisnimi” sarakkeessa olevilla arvoilla viitataan yleisesti tunnistettaviin tietotyyppitermeihin jotka ovat käytössä eri ohjelmointikielissä. ”OAS-kuvaustyyppi” sarakkeessa esitellään kuvauskielessä käytettävä vastine yleisnimikkeelle. ”Formaatti” sarakkeen määrityksen avulla voidaan tarkentaa tietotyypin käyttäytymistä taulukon mukaisesti. Rajapintakuvauksen kannalta tietotyyppien määrittäminen ei ole pakollista, mutta suositeltavaa. Käytettäessä selkeää määritystä kuvauksen loppukäyttäjän ei tarvitse päätellä missä muodossa tietoa on tarkoitus käyttää. Tietotyyppien määrittäminen on yleensä pakollista siinä tapauksessa, että kuvauksen perusteella muodostetaan lähdekoodia. (OpenAPI Specification version 2.0, 2017.)

OAS 2.0 antaa kuvauskielenä mahdollisuuden käyttää omia kenttämäärityksiä eivätkä tietotyypit tuota tässä poikkeusta. Määriteltäessä käytettäväksi tietotyypiksi *string* voidaan tietotyypille määrittää mikä tahansa ”OAS-kuvaustyyppi” ja siihen sopiva formaatti. Tässä tapauksessa kuvaus ei tietenkään noudata JSON-Schma draft 4 määritystä ja ei näin ollen ole em. standardin mukainen. (IEFT 2017.)

Liitteessä 1 on kuvattu OAS 2.0-tukemat kenttämääritykset. Useassa tapauksessa nämä kenttämääritykset riittävät hyvinkin pitkälle, mutta eteen voi tulla tilanteista missä valmiiksi määritellyt objektit tai kentät eivät tuota haluttua lopputulosta. Kuvauskieltä on mahdollista laajentaa omilla kenttämäärityksillä. Kuvauskielen laajentaminen tapahtuu määrittämällä *x-kenttänimi* ja tähän mahdollisesti liittyvät parametrit. Kuvassa 5 on kuvitteellinen kenttämäärityksen laajentaminen. Editorin oikeasta lohkosta voidaan todeta, etteivät omat kenttämääritykset tule esiin kuvauksen ilmentymässä, mutta ovat käytettävissä esimer-

kiksi tilanteessa, kun kuvauksesta muodostetaan lähdekoodia. (OpenAPI Specification version 2.0, 2017.)



Kuva 5. Kenttämäärityksen laajentaminen. (Kuvakaappaus.)

4.2 OAS 2.0-kuvauskielen toimintaperiaate

Koska kysymyksessä on laaja kokonaisuus, joka sisältää merkittävän määrän objekteja, kenttämäärityksiä, parametreja ja muita arvoja lähestytään rajapintakuvauksen tekemistä ja toimintaa esimerkkien kautta. Liitteessä 1 on kuvattu OAS 2.0-kenttämääritykset taulukon avulla. Liitteen taulukko vastaa OpenAPI 2.0-määrityksen sisältöä (OpenAPI Specification version 2.0, 2017). Edellä mainitussa taulukossa sarake ”S” tarkoittaa tarvittavien sisennyksien määrää. ”V” kuvaa värikoodia eli kenttämäärityksen ryhmitystä mihin viitataan myöhemmin dokumentin edetessä.

Liitteessä 1 vihreällä merkittyjen kenttämäärityksien avulla kuvataan metatietoja ja koko rajapintaa koskevia globaaleja määrityksiä. Globaaleilla määrityksillä tarkoitetaan tässä tapauksessa esimerkiksi tietoa siitä, mitä tiedonsiirtotapaa tai tyyppiä kuvattu rajapinta tukee. Globaaleja tietoja voidaan ylikirjoittaa polun sisällä tehtävällä poikkeavalla määrityksellä.

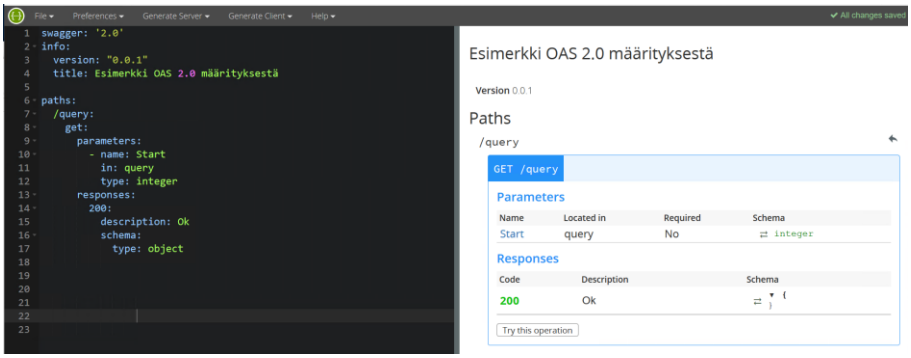
Sinisellä kuvatut kenttämääritykset määrittävät rajapinnan polkujen eli URL: ien toiminnan. Polun kuvaus sisältää URI:in, käytettävän http-verbin, mitä muuttujia http-verbi käyttää ja miten muuttujatietoa käytetään verbin kanssa. Edellä olevaan olennaisena liittyvät myös käytettävien muuttujien tietotyyppimääritykset ja syötteen tarkistaminen.

Mustat kenttämääritykset kuvaavat miten toimintaan, kun palvelin palauttaa dataa. Palautuksessa olennaista on statuskoodi ja siihen liittyvä tietotyypin määritys. Musta määritys on olennainen osa sinistä määritystä ja määritellään jokaiselle kuvatulle polulle.

Oranssilla kuvatut kenttämääritykset kuvaavat saman tietosisällön kuin musta, mutta on tarkoitettu keskitetyksi eli globaaliksi määritykseksi. Näitä määrityksiä voidaan kutsua mistä tahansa rajapintakuvauksesta.

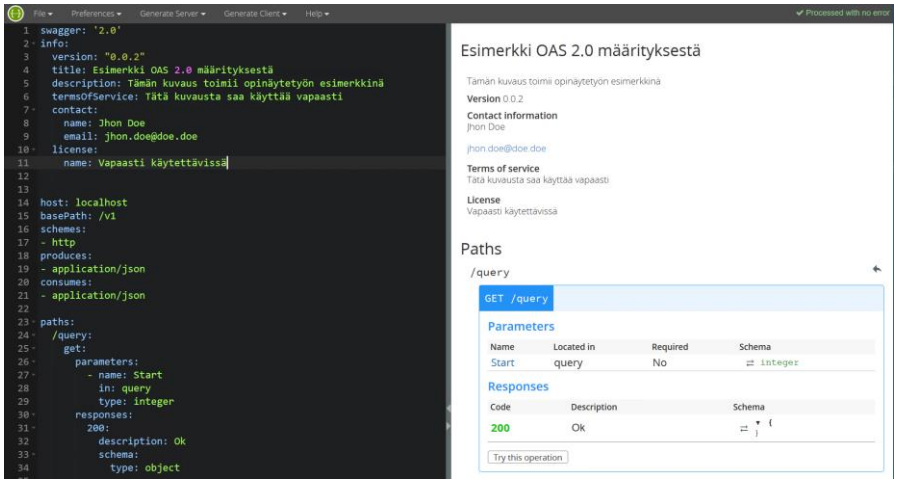
Punaisella kuvatut kenttämäärityksset toimivat kuten oranssit. Oranssin ja punaisen kenttämäärityksen erotus syntyy siitä mitkä kenttämäärityksset ovat tuettuja.

Tässä vaiheessa edellä oleva saattaa tuntua hyvinkin sekavalta, mutta ryhdytään avaamaan toimintaa esimerkkien avulla. Yksikertaisimmillaan rajapintakuvaus muodostuu liitteen 1 vihreästä, sinisestä ja mustasta osasta ja näissä olevista kenttämäärityksistä jotka on merkitty pakollisiksi. Kuvassa 6 voidaan todeta, että editorin kannalta toimiva rajapintakuvaus on alle 20 riviä pitkä. Kuvassa 6 esitetty kuvaus määrittää, että joltain palvelimelta löytyy URI nimeltä *query* ja siihen voi esittää *http-get* pyynnön. Pyynnön osana voidaan esittää muuttuja *Start*, joka on integer-lukuarvo. Lukuarvon kokoa ei ole määritelty. *Start* muuttujan käyttäminen ei kuitenkaan ole pakollista eli sen arvo voi olla myös tyhjä. Koska muuttujan lukuarvo on määritelty luettavana *query* muodossa tarkoittaa se, että muuttuja esitetään osana URL:ia. Mikäli muuttujan arvo annetaan on syöte muodossa */query?Start*=muuttujan-arvo. Palvelimen palautuksena odotetaan JSON-objektia statuskoodilla 200. Palvelimen palauttaessa jotain muuta statuskoodia ei rajapintakuvauksessa ole kuvattu sen esitystapaa.



Kuva 6. Esimerkki 1 rajapintakuvauksesta. (Kuvakaappaus.)

Kuvassa 6 esitetyllä rajapintakuvauksella ei kehittäjä vielä tee mitään, mutta ryhdytään laajentamaan kuvausta. Määritellään rajapintakuvaukseen mukaan liitteen 2 vihreästä osasta joukko yleistä meta- ja palvelintietoja, polku- ja MIME-määrityksiä jotka toimivat oletuksena koko rajapintakuvaukselle. Näiden osalta on hyvä huomioida, että ryhdytään käyttämään vapaaehtoisesti käytettäviä kenttämäärityksiä. Kuvassa 7 selviää, että kutsuttava polku eli URL löytyy nyt osoitteesta *http://localhost/v1/query*. Rajapinta lähettää ja vastaanottaa datan JSON-muodossa. Muuttujan *Start* syöte voidaan nyt antaa muodossa *http://localhost/v1/query?Start*=muuttuja-arvo.



Kuva 7. Esimerkki 2 rajapintakuvauksesta. (Kuvakaappaus.)

Kuvan 7 rajapintakuvaus tuottaa jo huomattavasti enemmän tietoa kuin kuvassa 6 oleva versio. Kehittäjän kannalta kuvauksessa kuitenkin puuttuu tieto mitä rajapinta tai sen määrittämä *get* tekee ja toisaalta ei ole kuvattu mitä tietoa palautuu kutsuttaessa määriteltyä polkua. Tälläkään versiolla kuvauksesta ei ole siis todellista hyötyä kehittäjän kannalta.

4.3 Tietotyyppien viittausmääritykset

OAS 2.0-kuvauskielessä olennainen osa on globaalit tietotyyppimääritykset ja niihin viittaaminen. Viittausmäärityksien tavoite on selkeyttää ja lyhentää rajapintakuvausta. Ideaalitilanteessa yhtä ja samaa määritystä käytetään useaan kertaan eri kohdissa kuvausta.

Aikaisemman värikoodauksen yhteydessä mainittiin, että oranssi ja punainen määritys on tarkoitettu globaaleiksi arvoiksi eli kuvauksiksi mihin voidaan viitata eri kohtaa kuvauksista. Globaaleiksi arvoiksi voidaan määrittää tietotyypit- ja niihin liittyvät määritykset, palautukseen liittyvät statuskoodit- ja muuttuja-arvot joita käytetään kuvaamaan polkurakennetta. Dokumentissa tullaan käsittelemään tietotyyppien ja palautuksien globaali kuvaaminen ja toiminta. URL:ien eli polkurakenteen kuvaaminen globaalien arvojen kautta ei välttämättä ole järkevää, mikäli kyseessä ei ole todella laaja rajapintakuvaus ja tästä syystä sitä ei käsitellä tässä dokumentissa.

Keskitetysti käytössä oleva tieto kuvataan kerran punaisen tai oranssin määrityksen määrittämällä tavalla ja sitä kutsutaan *$ref* kenttämäärityksen avulla niin monta kertaa kuin siihen on tarvetta. Punaiset määritykset on tarkoitettu kuvaamaan tietotyyppejä ja oranssit palautuksien eli statuskoodien toimintaa.

Viittausmärityksiä voidaan käyttää myös globaalin viittauksen sisällä (oranssi-punainen tai punainen-punainen). On siis mahdollista luoda monimutkaisia tietomallirakenteita yhdistämällä tietotyyppejä viittauksien avulla toisiinsa.

```
17  - http
18  produces:
19  - application/json
20  consumes:
21  - application/json
22
23  paths:
24    /query:
25      get:
26        parameters:
27          - name: Start
28            in: query
29            type: integer
30        responses:
31          200:
32            description: Ok
33            schema:
34              $ref: '#/definitions/Esimerkki'
35
36  definitions:
37    Esimerkki:
38      type: object
39      description: Objektin kuvaustieto
40      title: Objektin otsikkotieto
41      properties:
42        data1:
43          type: integer
44          format: int32
45          maximum : 100
46        data2:
47          type: string
48          minLength: 5
49          maxLength: 15
```

Paths

/query

GET /query

Parameters

Name	Located in	Required	Schema
Start	query	No	integer

Responses

Code	Description	Schema
200	Ok	Objektin otsikkotieto { Objektin kuvaustieto data1: integer data2: string }

Try this operation

Models

Esimerkki

Objektin otsikkotieto { Objektin kuvaustieto data1: integer data2: string }

Kuva 8. Esimerkki 3 rajapintakuvauksesta. (Kuvakaappaus.)

Kuvassa 8 on esitetty esimerkki viittaamisen peruskäytöstä. Luodaan globaali määritys liitteen 2 punaisten määrityksen mukaisesti ja viitataan siihen mustassa osassa. *$ref:"#polku"* kuvaa mistä haluttu määritys löytyy.

```
21
22  paths:
23    /query:
24      get:
25        parameters:
26          - name: Start
27            in: query
28            type: integer
29        responses:
30          200:
31            description: Ok
32            schema:
33              $ref: '#/definitions/Esimerkki'
34          400:
35            $ref: "#/responses/Standard400Error"
36
37  definitions:
38    Esimerkki:
39      type: object
40      properties:
41        data1:
42          type: integer
43        data2:
44          type: string
45    400:
46      type: object
47      properties:
48        id:
49          type: string
50        name:
51          type: string
52
53  responses:
54    Standard400Error:
55      description: Bad request
56      schema:
57        $ref: "#/definitions/400"
58
59
```

GET /query

Parameters

Name	Located in	Required	Schema
Start	query	No	integer

Responses

Code	Description	Schema
200	Ok	Esimerkki { data1: integer data2: string }
400	Bad request	400 { id: string name: string }

Try this operation

Models

400

400 { id: string name: string }

Esimerkki

Esimerkki { data1: integer data2: string }

Kuva 9. Esimerkki 4 rajapintakuvauksesta. (Kuvakaappaus.)

Kuvassa 9 esitellään hieman monimutkaisempi viittausrakenne. Määrityksestä selviää, että palvelimen palauttaessa statuskodin 200 odotetaan palautuksen tietotyypin eli tietosisällön tulevan siinä muodossa, miten globaali tietotyyppikuvaus ”Esimerkki” sen määrittää olevan. Standardin mukaisesti http-statuskoodi 400 tarkoittaa, että kysymys on esitetty väärin. Edellä mainitun virheen saa aikaan sillä, että kysymyksen muuttujalle annetaan syötearvoksi *string* muotoinen arvo. Tässä tapauksessa palautuksena toimisi keskitetysti määritelty ”Standard400Error” oranssin määrityksen alaisuudesta. Tämä viittausmääritys saa kuitenkin tietotyyppikuvauksen punaisen lohkon si-

sältä. Esimerkissä pyritään kuvaamaan, että mustan värikoodin sisällä voidaan viitata suoraan punaiseen tietotyyppikuvaukseen tai vastaavasti oranssiin virhekuvaukseen. Näiden lisäksi oranssista lohkosta on vielä kutsuttu punaisessa lohkossa olevaa tietotyyppikuvausta.

5 OAS 2.0-TYÖKALUT

Aikaisemmin dokumentissa on kuvattu kenttämäärityksen syntaksia ja rajapintakuvauksen luomista periaatetasolla. OAS 2.0-työkalut voidaan jakaa kolmeen luokkaan. Ensimmäinen on tarkoitettu kuvauksen luomiselle eli editorille- ja kielen syntaksin tarkastamiselle. (IO 2017.)

Toinen luokka on tarkoitettu lähdekoodin muodostamiselle halutulla ohjelmointikielellä. Tässä tapauksessa kirjoitetusta rajapintakuvauksesta muodostetaan joko palvelin- (Generate server) tai (Generate client) asiakas SDK-paketti joita vastaan kehittäjä voi testata ja/tai koodata sovelluksiaan. Rajapintakuvaus voidaan muodostaa myös kirjoitetusta lähdekoodista siellä määriteltyjen kuvauskenttien avulla. Jokaisella lähdekoodikielellä on runsas joukko kirjastoja, joiden avulla rajapintakuvauksen muodostaminen on mahdollista. Tämä lähestymistapa mahdollistaa rajapinnan dokumentaation tuottamisen samassa syklissä lähdekoodin kirjoittamisen kanssa. (IO 2017.)

Kolmantena osana on työkalut jotka mahdollistavat rajapintakuvauksen käyttämisen dokumentointi ja testaus tarkoituksissa. OAS 2.0-dokumentointi on HTML-muotoinen ja se mahdollistaa kuvattujen polkujen käyttämisen suoraan dokumentaatiosta. (IO 2017.)

5.1 OAS 2.0-editori

Editoria on mahdollista käyttää online-versiona tai sen voi asentaa omalle koneelleen. Online versio löytyy osoitteesta http://editor.swagger.io/#!/. Onlinepalvelun kautta editoria voi käyttää ilmaiseksi, mutta esimerkiksi kuvattavien API:en kappalemäärä on rajoitettu. Palvelusta on saatavilla maksulliset vaihtoehdot jotka mahdollistavat mm. tiimitoiminnan kuten Github-integroinnin ja kuvauksen näkyvyyden hallinnan. Ilman kuukausimaksullista lisenssiä kaikki luodut rajapintakuvauksen ovat julkisesti nähtävillä. Toiminta vastaa tältä osin Githubin-toimintaperiaatteita. (IO 2017.)

Mikäli OAS 2.0-editoria haluaan suorittaa paikallisesti voi ohjelmiston ladata osoitteesta https://github.com/swagger-api/swagger-editor/tree/2.x Paikallisella koneella on oltava asennettuna jokin http-palvelin, jonka alaisuudessa editorin ohjelmistokoodia suoritetaan. Editorista on saatavilla myös Docker-kontti. Editori itsessään on sama online- ja paikallisen asennuksen osalta. Paikallisesti asennettuna Github-integrointia ei ole saatavilla. (IO 2017.)

Editorin ”File”-valinnan alaisuudessa löytyy mahdollisuus tallentaa ja ladata rajapintakuvauksia. Kuvauksen voi tallentaa sekä YAML-, että JSON- muodossa. Tietoa voi myös ladata samoissa tiedostomuodoissa. ”File”-valinnan alaisuudesta löytyy myös mahdollisuus avata rajapintakuvauksien esimerkkejä. On tärkeää huomioida, että editori pysty käsittelemään yhtä kuvausta kerrallaan. Web selaimen kannalta on mahdollista avata useampia ikkunoita ja ajaa useampaa editoria tällä tavalla. Käytäntö on kuitenkin osoittanut, että useamman editorin aukipitäminen samassa selainsessiossa ei ole järkevää, mahdollisen virhetilanteen vuoksi. Varsinaista tallenna-toiminnetta ei ole vaan viimeisin kirjoitettu versio jää sellaisenaan selaimen välimuistiin. Selaimen uudelleenkäynnistys tilanteessa editointia voidaan jatkaa siitä pisteestä mihin viimeksi on jääty. Onkin suositeltavaa käyttää ”download”-toiminnetta jonka avulla saa tallennettua sen hetkisen tilanteen tiedostoon.

”Preferences”-valinnan alaisuudesta löytyvät erilaiset editorin ulkonäköön, käyttäytymiseen- ja fonttien kokoon liittyvät valinnat. Valinnoilla saa editorin helposti tilaan missä se ei vastaa tarkoitustaan. ”Preferences”-valinnan alaisuudesta löytyy myös editorin oletusarvojen palautus valinta.

Editorissa olevat niin sanotut codegen- eli koodigenerointi toiminteet edellyttävät, että suoritettavasta pisteestä on internet yhteys. Mikäli yhteyttä ei ole, editorissa olevat ”Generate Server” ja ”Generate Client” valinnat eivät ole näkyvissä. Codegen-toiminteita tarkastellaan tarkemmin seuraavassa kappaleessa.

5.2 OAS 2.0-koodigeneraattori eli codegen

Koodigeneraattoria voi käyttää paikallisesti tai editorin-avulla. Paikallisesti komentokehotteesta suoritettava koodigeneraattori on saatavilla https://github.com/swagger-api/swagger-codegen osoitteesta. Paikallisesti käytettävä generaattori edellyttää Java-SDK 7 tai uudempaa ja Apache-mavenin asentamista. Generaattorista on myös olemassa Docker-kontti, joka sisältää kaiken tarvittavan generointia varten. Koodigeneraattorista julkaistaan tasaisin väliajoin uusia versioita. Koodigeneraattoreiden tuottaman lähdekoodin paketit eivät välttämättä ole yhteensopivia eri versioiden välillä. (Community 2017.)

Editorissa oleva koodigeneraattori sisältää samat perustoiminteet kuten paikallisesti komentokehotteesta käytettävä generaattori. Erotus näiden kahden välillä syntyy siitä, että editorissa olevalle koodigeneraattorille voi kertoa pelkästään generoitavan lähdekoodin ohjelmointikielen. Komentokehotteesta käytettävälle generaattorille voi

määrittää parametreilla arvoja, miten generointi suoritetaan huomattavasti tarkemmalla tasolla.

Tuettuja ohjelmointikieliä (Generate server) palvelimia varaten ovat: C# (ASP.NET-Core, NancyFx), C++ (Pistache, Restbed), Erlang, Go, Haskell, Java (MSF4J, Spring, Undertow, JAX-RS: CDI, CXF, Inflector, RestEasy, Play Framework), PHP (Lumen, Slim, Silex, Symfony, Zend Expressive), Python (Flask), NodeJS, Ruby (Sinatra, Rails5), Scala (Finch, Scalatra). (Community 2017.)

Tuetut (Generate Client) asiakas ohjelmointikielet ovat: ActionScript, Apex, Bash, C# (.net 2.0, 4.0 or later), C++ (cpprest, Qt5, Tizen), Clojure, Dart, Elixir, Eiffel, Go, Groovy, Haskell, Java (Jersey1.x, Jersey2.x, OkHttp, Retrofit1.x, Retrofit2.x, Feign, RestTemplate, RESTEasy), Kotlin, Node.js (ES5, ES6, AngularJS with Google Closure Compiler annotations) Objective-C, Perl, PHP, PowerShell, Python, Ruby, Scala, Swift (2.x, 3.x, 4.x), Typescript (Angular1.x, Angular2.x, Fetch, jQuery, Node). (Community 2017.)

Editorilla muodostettu lähdekoodi generoituu pakatuksi paketiksi ja sisältää rajapintakuvauksessa kuvatut polut, tietotyyppikuvaukset, kontrollerit jne. Jotta muodostettua lähdekoodia voidaan suorittaa pitää paketti purkaa ja ohjelmointikielestä riippuen asentaa tarvittavat kirjastot sen suorittamista varten. Ohjeet esim. tarvittavien kirjastojen asentamisesta löytyvät paketin sisällä olevasta readme tiedostosta. Koska tuettuja kieliversioita on huomattava määrä saattaa pakettien käytössä esiintyä eroavaisuuksia kielen määrityksistä riippuen. (Community, 2017)

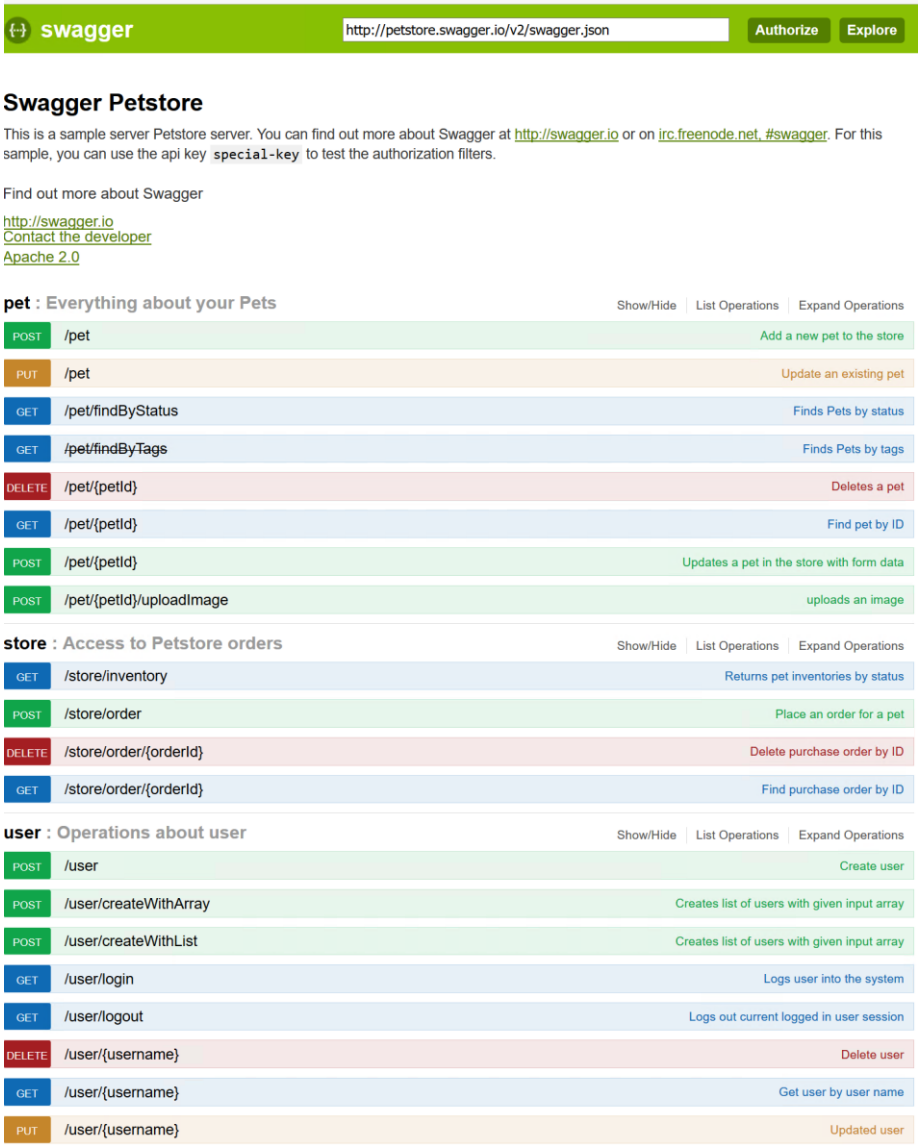
5.3 OpenAPI UI

Siinä missä Editorin oikea ikkuna toimii dokumentointina rajapinnan luonnin aikana, on UI tarkoitettu dokumentointi ja testaus tarkoitusta varten. UI-ohjelmisto asennetaan pisteeseen mistä rajapinnan dokumentointia halutaan esittää. UI-sovellus edellyttää käytössä olevaa http-palvelinta kuten editorikin. UI:n asennuspaketin saa osoitteesta https://github.com/swagger-api/swagger-ui

Kuten kuvasta 10 voidaan todeta, UI-sovellus esittää rajapintakuvauksen interaktiivisena http-sivuna. UI-sovelluksessa rajapintakuvaus avataan JSON-formaatissa. JSON-kuvaus on sama kuvaus kuin rajapinnasta muodostettu YAML-kuvaus jota käytetään editorissa. UI-ohjelmiston käyttöperiaate on melko suoraviivainen. Editorilla luodaan rajapintakuvaus. Kuvaus tallennetaan editorissa (download) JSON-muodossa ja tiedosto sijoitetaan pisteeseen mistä UI voi lukea sen.

Kuvassa 10 voidaan havaita, että polut on ryhmitelty luokkiin *pet, store* ja *user*. Ryhmittelyn pystyy toteuttaman rajapintakuvauksessa niin, että halutulle polulle kerrotaan liitteen 1 sinisten luokan *tags:* kenttämäärityksellä arvo. Yhdellä polulla voi olla useampi tagi jolloin UI:n kannalta sitä esitettäisiin useamman eri ryhmittelyn eli tägin alaisuudessa.

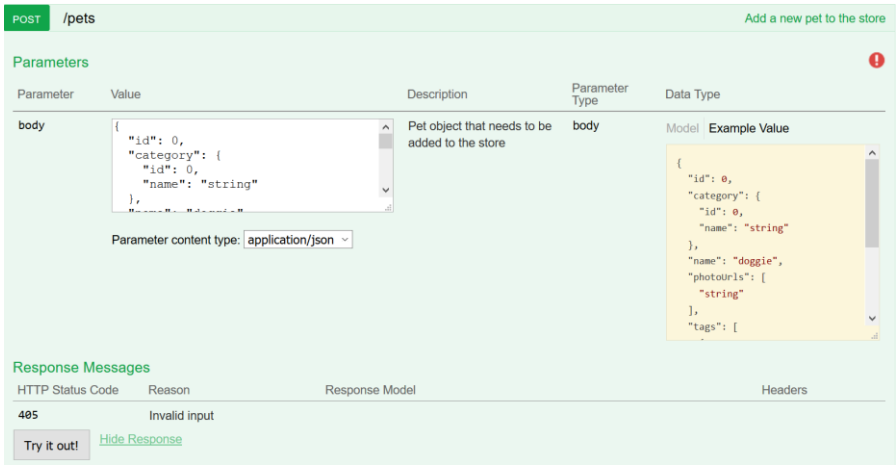
Mikäli rajapintakuvauksessa on määritelty käyttövaltuutuksia (authorization) pystytään UI-ohjelmassa tekemään valtuutus authorize-painikkeen avulla. Valtuuttaminen toteutetaan tässä tapauksessa niin kuin se on rajapintakuvauksessa määritelty toimivan.



Kuva 10. UI-ohjelman Petstore esimerkki. (Kuvakaappaus.)

Kuvassa 10 on UI:n oletusnäkymä, kun UI-sovellus on purettu ja käynnistetty ensimmäisen kerran. UI:ta kutsutaan http://localhost/swaggerui/dist/#/ osoitteesta tai vastaavasti siitä polusta minne se on purettu http-palvelimella. UI esittää rajapintaku-

vauksen dynaamisena HTML-sivuna. Määriteltyjä polkuja voi tarkastella lähemmin ”Expand operations”-valinnan avulla. ”Expand operations”-mahdollistaa rajapintakuvauksen testaamisen suoraan dokumentaatiosta kuten kuvassa 11 on esitetty. Valittaessa ”Example value”-kopioituu sen sisältö ”body”-arvoon jota voi halutessaan muokata sopivaksi käyttötapauskohtaisesti. Rajapintaa voidaan siis tarkastella polkumääritysrakenteen ja siihen liittyvien tietotyyppimääritysten kautta. Tällä tavoin rajapinnan kuluttaja tai kehittäjä saa yksityiskohtaista tietoa rajapinnan toiminnasta.



Kuva 11. UI-sovelluksen esimerkinkäyttö. (Kuvakaappaus.)

UI-käytön kannalta tulee huomioida, että se voi palauttaa dataa ainoastaan siinä tapauksessa, että siihen on kytketty jokin tietolähde ja että tietolähteellä on edellytykset vastata http-kyselyyn. UI-sovelluksen sisäänrakennettu petstore-esimerkki osoittaa Smartbearin ylläpitämään http-palveluun ja näin ollen sieltä voidaan odottaa palautuvan dataa. Mikäli UI:lle halutaan osoittaa oma rajapintakuvaus, tulee sitä kutsua kuvan 10 yläreunassa olevalla URL-rivillä ja Http-palvelimella tulee olla pääsy tiedostosijantiin. Oletusarvoisesti voidaan käyttää swagger UI-sovelluksen dist -hakemistoa. UI:n saa oletuksena avaamaan paikallisen kuvauksen editoimalla dist -hakemistossa olevaa index.html tiedostoa ja sieltä riviä url=http://petstore.swagger.io/v2/swagger.json.

6 CASE TRAFI

Opinnäytetyön toiminnallisessa osassa toteutettiin rajapinta, jonka avulla voidaan tarkastella Trafin avointa ajoneuvodataa. Samassa yhteydessä rajapintatoteutuksesta syntyi dokumentaatio. Trafin data-aineisto on melko laaja sisältäen noin 5-miljoonaa riviä ja 38-saraketta yksilöityä tietoa. Lähdeaineisto on saatavilla pilkuin eroteltuna .csv muotoisessa tiedostossa. Data sisältää tietoa kuten ajoneuvonkoko, paino, väri, ajoneuvoluokka, käyttövoima, teho ja niin edelleen. Trafin julkaisema data noudattaa kansainvälistä Creative Commons (CC) 4.0 lisenssiehtomallia. (Trafi 2017.)

Rajapintasuunnittelun lähtökohtana on tehdä rajapintakuvaus, jota voidaan tarvittaessa laajentaa kyselyiden osalta ja sen tulee tarjota JSON-palaute datasta. Koska kyseessä on data-aineisto joka tuottaa rekisterimäisesti tietoa ei rajapintaan toteuteta datan lisäys-, päivitys- tai poistotoiminteita.

Rajapintasovellus toteutettiin asentamalla XAMPP-ohjelmisto, joka tuottaa Apache- ja MySQL-palvelut. MySQL:ään luontiin tietokanta johon ladattiin edellä mainittu Trafin .csv aineisto. Aineiston mukana tuli myös lisätieto .csv aineistoja joiden avulla voisi kuvata esimerkiksi ajoneuvodatassa olevaa värikoodausta. Ajoneuvodatassa väri on ilmaistu numerokoodilla ja tätä vastaan on olemassa aputietue, joka kuvaa värikoodia määrittävän oikean värin. Näiden käyttöönottoa ei tässä vaiheessa katsottu aiheelliseksi. Edellä kuvattujen sovellusten lisäksi asennettiin OAS 2.0-editori ja UI-ohjelmistot. MOCK-palvelinlogiikan ohjelmointikieleksi valittiin Python versiossa 3.6

6.1 Rajapinnan suunnittelu

Rajapintaa ryhdyttiin sovelletusti suunnitelmaan kappaleessa 3.1 määriteltyjen kysymyksien avulla. Rajapinnan on tarkoitus toimia opinnäytetyön yksinkertaisena toteutusesimerkkinä ja näin ollen halutaan esittää kaikki data mitä on kyselyyn liittyen saatavilla. Käyttäjinä voivat olla niin ihmiset kuin ohjelmistot. Tästä syystä tiedonvälitys formaatiksi valittiin JSON. Opinnäytetyön rajaukset määrittävät, että kyseessä on REST-rajapinta ja suunnittelukehikkona käytetään OAS 2.0. Rajapintaan ei kohdistu tietoturvan tai kyselymäärän osalta rajoituksia, koska rajapintaa ei ole tarkoitus julkaista suoraan internetistä saatavaksi. Ohjelmointikieleksi valikoitui Python omien mieltymyksien takia. Rajapinnan MOCK- eli testaus ohjelmalogiikan tuottavana web-palveluna toimii ohjelmointikielen oma web-palvelin. Rajapinnan URL:iin haluttiin mukaan versionumero myöhempää käyttöä varten.

SQL-kysely toteutettiin rajapintamäärityksen mukaisesti. Data itsessään mahdollistaa hyvin monimuotoisten kyselyiden toteuttamisen, mutta määrityksen puitteissa toteutettiin datan sarakkeen ”kayttoonottopvm” aikarajaukseen perustuva haku, joka noutaa kaiken saatavilla olevan datan. Koska tietoja voi palautua huomattava määrä, määriteltiin kyselylle rajausarvo, jonka avulla on mahdollista vaikuttaa palautuvien rivitietojen enimmäismäärään. Muodostettu rajapintakuvaus löytyy liitteestä 2.

6.2 MOCK ohjelmistokoodi

Rajapintakuvauksen testausta varten muodostettiin kuvan 12 mukainen Python-ohjelma, joka osaa ottaa yhteyden MySQL-tietokantaan ja palauttaa sieltä http-kysymyksen mukaisesti tietoja. Python käyttää kirjastoa nimeltä Flask ja sen tarjoamia laajennuksia kuten Request ja Jsonify. Requestia käytetään ohjelmakoodin sisällä ottamaan vastaan rajapinnan syöttämät muuttuja-arvot URL syötteestä. Jsonifyä käytettiin muodostamaan data-aineistosta JSON-formatoitua dataa. Tietokantayhteyksien muodostamiseen käytettiin Pythonin Pymysql kirjastoa, joka mahdollista tietokannan käyttämisen nativiiveilla SQL-lauseilla. SQL-tietokantayhteyden olisi voinut toteuttaa esim. SqlAchemyn avulla jolloin tietokantayhteydestä olisi tullut ORM tyyppinen. Esimerkki koodin kannalta Pymysql käyttö on kuitenkin yksinkertaisempaa.

Pythonin flask-kirjasto tarjoaa web-palvelimen jota voi käyttää testaustarkoituksissa. Kuvassa 12 on määritelty, että sovelluksen tuottama web-palvelin toimii ns. *localhost* osoitteessa portissa *5000*.

```
1  # -*- coding: utf-8 -*-
2  # Määritellään käytettävät kirjastot
3  from flask import Flask, request, jsonify
4  import pymysql
5
6  # Määritellään solvellus ja sen nimi
7  app = Flask(__name__)
8
9  # Luodaan GET metodi joka kuuntelee polkua /querydate
10 @app.route("/querydate", methods=['GET'])
11 # Määritellään metodin toiminta
12 def querydate():
13     # Luetaan URL muuttuja arvot
14     startday = request.args['start']
15     endday = request.args['end']
16     limitit = request.args['limit']
17     # Kuvataan mistä tietokanta löytyy. Kuvataan käyttäjätunnukset
18     db = pymysql.connect("localhost", "apiread", "apiread","trafi")
19     # Määritellään, että kyselyn tulee palauttaa datan lisäksi sarakeotsikot
20     cur = db.cursor(pymysql.cursors.DictCursor)
21     # Määritellään SQL lause jolla tietoa kysytään
22     query = ("SELECT * FROM avoindata WHERE kayttoonottopvm BETWEEN %s AND %s LIMIT %s" % (startday, endday, limitit))
23     # Suoriteaan SQL kysely
24     cur.execute(query)
25     # Noudetaan suoritetun kyselun tulos
26     data = cur.fetchall()
27     # Muodostetaan noudetusta datasta JSON formaatissa oleva palaute
28     return jsonify(data)
29
30 # MOCK Web palvelin
31 if __name__ == "__main__":
32     app.debug = True
33     app.run(host="0.0.0.0", port=5000)
```

Kuva 12. Python MOCK-ohjelmistokoodi. (Kuvakaappaus.)

Kuvassa 12 kuvattua rajapintasovellusta voi kutsua web-selaimesta esimerkiksi muodossa
http://localhost:5000/?start=19900000&end=19910000&limit=10
Edellä oleva palauttaa tietokannasta 1990 ja 1991 vuosien välitä 10 ensimmäistä kannassa olevaa rivitietoa kaikkine sarakkeineen JSON-muodossa.

6.3 Toteutus

Kun rajapintatoteutus oli kuvattu OAS-editorin avulla ja siihen liittyvä MOCK-sovellus kirjoitettu, voitiin nämä kaksi toiminnetta yhdistää niin, että generoitavaa UI-sovellusta voidaan käyttää käyttöliittymänä kohti tietokantaa testaustarkoituksessa.

Kappaleessa 5.2 on kuvattu OAS 2.0-editorin mahdollisuus tuottaa lähdekoodia halutulle kielelle. Kuvauksesta generoitiin OAS 2.0-editorin (Generate server) avulla Python-palvelinpaketti joka puretaan pisteeseen missä sitä haluttiin suorittaa. Suorittaminen edellyttää, että koneella on asennettuna Python versio 3.52 tai uudempi.

Generoidun lähdekoodin mukana syntyy readme-tiedosto, jossa on mainita, että ennen suorittamista tulee asentaa sovelluksen vaatimat kirjastot. Vaaditut kirjastot on kuvattu requirements-tiedostossa. Kirjastojen asennukset tehdään Pythonin tapauksessa pipin-ohjelmiston avulla.

Puretussa paketissa ”swagger server” hakemiston alaisuudessa on hakemisto ”swagger” joka sisältää aiemmin luodun YAML-rajapintakuvauksen muutamalla kenttälisäyksellä. Tarkasteltaessa Swagger.yaml tiedostoa voidaan havaita, että generointi on lisännyt kuvan 13 mukaisesti kenttämääritykset ”x-swagger-router-controller” ja ”operationId” kenttämääritykset.

```
25 paths:
26   /query:
27     get:
28       tags:
29       - "Limited"
30       summary: "Tämä kysely palauttaa dataa perustuen `kayttoonottopaiva` kentän arvoon.\
31         \ Data palautuu JSON muodossa ja sisältää Queryresponsen mukaisen palautteeen"
32       description: "Määritä kyselylle arvot start eli aloitus, end eli lopetus limit\
33         \ eli palautettavien JSON lista-arvojen määrä."
34       operationId: "query_get"
35       parameters:
36       - name: "start"
43       - name: "end"
50       - name: "limit"
57       responses:
70       x-swagger-router-controller: "swagger_server.controllers.limited_controller"
```

Kuva 13. Generoitu swagger.yaml tiedosto. (Kuvakaappaus.)

”X-swagger-router-controller” on saanut arvon ”swagger_server.controllers.limited_controller”, joka on tiedostonimi ja

polkumääritys pisteeseen mistä polun suorituslogiikka on saatavilla. Hakemisto ja tiedostonimi edellä olevalle on siis oikeasti muodossa \swagger server\controllers\limited_controller.py Kenttämäärityksellä operationsId kuvataan mihin polun metodiin viitataan. Polkuja ja siihen liittyviä metodeita voisi siis olla useita ja kontrollerin määritystiedostoja yksi.

Kuvassa 14 nähdään miltä muokkaamaton kontrolleritiedosto ”swagger_server.controllers.limited_controller” näyttää. Kuvassa näkyy myös kuvassa 13 mainittu *operationsId* kenttämäärityksen arvo *query_get*. Kuvasta 14 voidaan todeta, että oletusarvoisesti kutsuttaessa rajapinta kuvauksen määrittämää polkua palautetaan poikkeuksena ”do some magic!”.

```
import connexion
from swagger_server.models.error import Error
from swagger_server.models.query_response import QueryResponse
from datetime import date, datetime
from typing import List, Dict
from six import iteritems
from ..util import deserialize_date, deserialize_datetime


def query_get(start, end, limit):
    """
    Tämä kysely palauttaa dataa perustuen &#x60;kayttoonottopaiva&#x60; kentän arvoon. Data palautuu JSON muodossa ja sisältää Queryresponsen mukaise
    Määritä kyselylle arvot start eli aloitus, end eli lopetus limit eli palautettavien JSON lista-arvojen määrä.
    :param start: Kyselyn aloitus aika muodossa 19900101
    :type start: int
    :param end: Kyselyn lopetusaika muodossa 19913112
    :type end: int
    :param limit: Palautettavien JSON lista objektien määrä
    :type limit: int

    :rtype: QueryResponse
    """
    return 'do some magic!'

```

Kuva 14. Generoitu limited_controller tiedosto. (Kuvakaappaus.)

Kuvassa 12 on kuvattu MOCK-logiikan lähdekoodi ja em. lähdekoodi sijoitetaan kuvan 14 kontrolleritiedostoon. Sijoituksen jälkeen lopputulos näyttää kuvan 15 mukaiselta. Lähdekoodin kannalta merkittävin erottava seikka on datan palautus. Generoidussa lähdekoodissa on sisäänrakennettu JSON-tulkki, joten MOCK-data palautetaan ns. raakamuodossa. Tämän johdosta Jsonify kirjaston käyttö on jätetty pois.

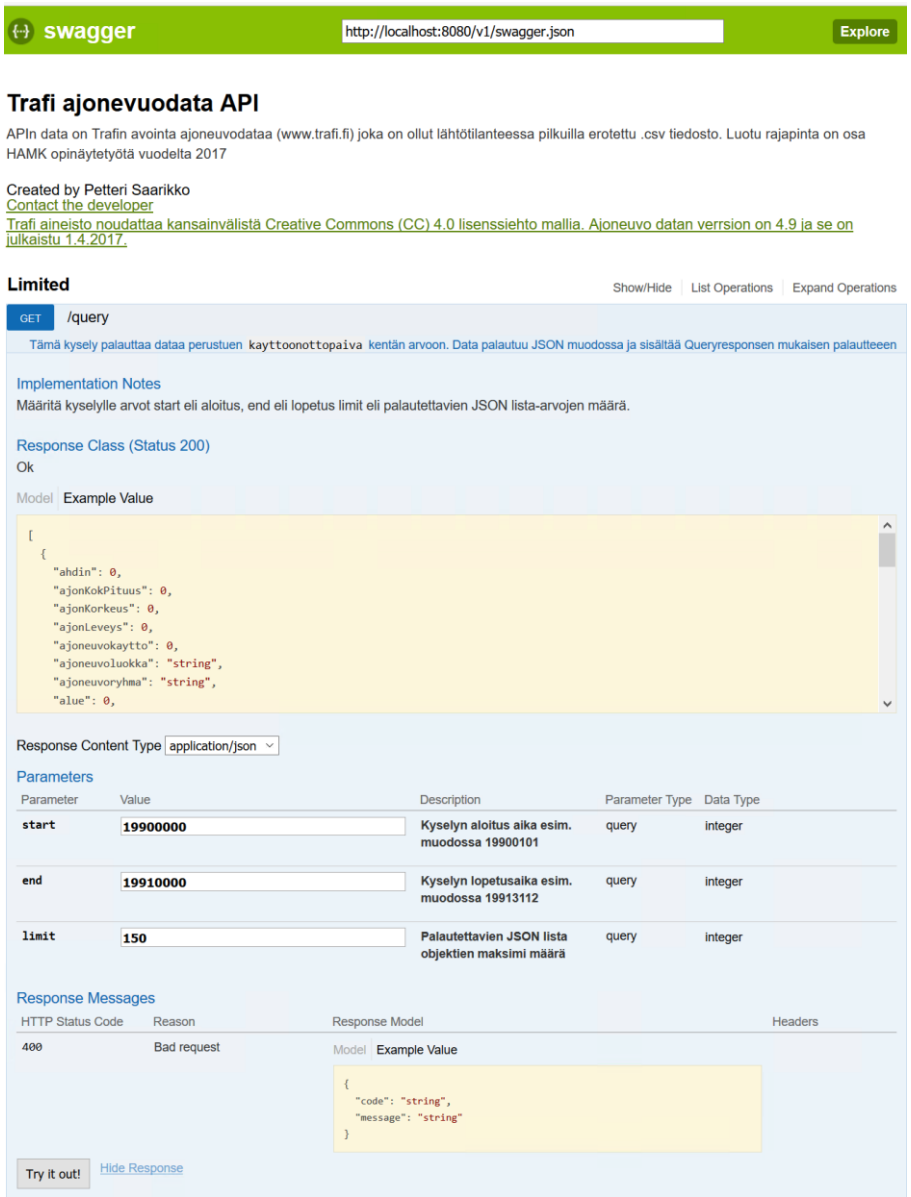
```
import connexion
import pymysql
from swagger_server.models.error import Error
from swagger_server.models.query_response import QueryResponse
from datetime import date, datetime
from typing import List, Dict
from six import iteritems
from ..util import deserialize_date, deserialize_datetime
from flask import Flask, request

def query_get(start, end, limit):
    startday = request.args['start']
    endday = request.args['end']
    limitit = request.args['limit']
    db = pymysql.connect("localhost", "apiread", "apiread","trafi")
    cur = db.cursor(pymysql.cursors.DictCursor)
    query = ("SELECT * FROM avoindata WHERE kayttoonottopvm BETWEEN %s AND %s LIMIT %s" % (startday, endday, limitit))
    cur.execute(query)
    data = cur.fetchall()
    return data

```

Kuva 15. Muokattu limited_controller tiedosto. (Kuvakaappaus.)

Edellä kuvattujen muutoksien jälkeen voidaan käynnistää generoitu python sovellus. Puretun palvelinkoodin readme tiedostossa on mainittu, että sovellusta voi kutsua osoitteesta

http://localhost:8080/v1/ui/. Edellä olevassa tulee huomioida, että rajapintakuvauksessa olevalla host- ja basepath- arvoilla on merkitystä rajapintaa kutsuttaessa ja palvelinta generoitaessa. Oletuksena generoitu palvelin odottaa käynnistyvänsä portissa 8080 joten rajapinta kuvaukseen on hyvä lisätä tämä porttitieto mukaan. Rajapintakuvauksesta tulee mukaan myös basepath-arvo ”v1” jolla tässä tapauksessa halutaan osoittaa rajapinnan versiota.

Python sovelluksen käynnistämisen jälkeen rajapintaa voidaan kutsua osoitteesta *http://localhost:8080/v1/ui/.* UI-sovellusta voidaan käyttää käyttöliittymänä kohti tietokantaa käyttöönotto päivän osalta. Kuvassa 16 on esitetty esimerkki kyselyn suoritus.



Kuva 16. Esimerkkikysely. (Kuvakaappaus.)

Kuvassa 17 on esitetty edellä kuvatun kyselyn vastaus UI-näkymän kautta. UI-sovelluksen osalta on hyvä huomioida, että ”Response bo-

dyn” esittää rajallisen määrän JSON-objekteja. Kutsuttaessa rajapintaa suoraan ”Request URL” osoitteen avulla palautuu JSON-objekteja pyydetty määrä.

Curl

```
curl -X GET --header 'Accept: application/json' 'http://localhost:8080/v1/query?start=19900000&end=19910000&limit=150'
```

Request URL

```
http://localhost:8080/v1/query?start=19900000&end=19910000&limit=150
```

Response Body


```
[
  {
    "Co2": 0,
    "ahdin": 0,
    "ajonKokPituus": 4270,
    "ajonKorkeus": 0,
    "ajonLeveys": 1680,
    "ajoneuvoluokka": "M1",
    "ajoneuvonkaytto": 5,
    "ajoneuvoryhma": 0,
    "alue": 7,
    "ensirekisterointipvm": "1990-05-08",
    "iskutilavuus": 1900,
    "istumapaikkojenLkm": 5,
    "jarnro": 7,
    "kaupallinenNimi": "BX",
    "kayttoonottopvm": 19900508,
    "kayttovoima": 1,
    "korityyppi": "",
```


Response Code

```
200
```

Response Headers


```
{
  "content-type": "application/json",
  "content-length": "149964",
  "server": "Werkzeug/0.12.1 Python/3.6.1",
  "date": "Tue, 01 Aug 2017 07:40:31 GMT"
}
```


Kuva 17. Esimerkkikyselyn vastaus. (Kuvakaappaus.)

7 YHTEENVETO

Opinnäytetyön tavoitteena oli tarkastella rajapintojen suunnitteluun ja toteutukseen liittyviä kysymyksiä. Dokumentin alussa tarkasteltiin syitä miksi rajapinta kannattaa suunnitella ennen toteutusta. Esiteltiin rajapinnan suunnittelua koskeva elinkaarimalli. Esiteltiin yleisimmät käytössä olevat rajapintojen kuvauskielet ja niiden tarjoamat työkalut. Perehdyttiin OAS-kuvauskieleen, sen syntaksiin ja käyttämiseen esimerkkien kautta. Tästä saatiin yhteys OAS-kuvauskielen työkaluihin ja kuvauksen luomiseen elinkaarisuunnitelman kautta. Käytännön osassa suunniteltiin rajapintakuvaus liittyen Trafin avoimen ajoneuvodataan. Tähän liittyen luotiin ohjelma, joka yhdistää Trafin-datan ja rajapintakuvaukseen esimerkki MOCK-palveluksi.

Rajapintojen suunnittelu- ja kuvaustyökalut mahdollistavat selkeän esitys- ja dokumentointitavan rajapinnoille. Dokumentoinnista hyötyvät niin luova organisaatio kuten myös rajapintaa kuluttavat kehittäjät. OAS-työkalujen hyödyt tulevat esiin rajapinnan suunnitteluvaiheessa. Rajapintasuunnitelmaa on helpompi muokata haluttuun suuntaan, kun osapuolet pääsevät tarkastelemaan sitä ja sen toimintaa jo kehitysvaiheessa. Suurin hyöty rajapinnankuvaamisesta tulee kuitenkin siinä vaiheessa, kun rajapintaa kuluttava kehittäjä ryhtyy hyödyntämään sitä. Rajapintakuvauksen aikana syntynyt dokumentaatio ja siihen liittyvät MOCK-esimerkit mahdollistavat kehittäjän nopean käytännön sovelluksen luomisen.

Kysymykseen kannattaako rajapinta suunnitella ennen sen toteuttamista ei ole olemassa yksiselitteistä vastausta. Suunnittelun avulla voidaan varmasti välttää turhaa työtä, mutta sen soveltuvuus riippuu täysin soveltavasta organisaatiosta ja sen sidosryhmistä. Selvää kuitenkin on, että toteutetut rajapinnat kannattaa kuvata esitettyjä tekniikoita hyödyntäen riippumatta siitä missä vaiheessa kuvaus tehdään. OAS-tarjoaa tätä varten selkeän kuvauskielen ja hyvät työkalut.

LÄHDELUETTELO

API blueprint. (2017). Haettu 14.03.2017 osoitteesta https://apiblueprint.org/

API, T. -O. (2017). *Avoimen rajapinnan määritelmä*. Haettu 16.8.2018 osoitteesta http://avoinrajapinta.fi/

Ben-Kiki, O. (2009). *YAML Ain't Markup Language (YAML™) Version 1.2*. Haettu 21.7.2017 osoitteesta http://www.yaml.org/spec/1.2/spec.html

Community, S. (2017). *Swagger Code Generator*. Haettu 18.7.2017 osoitteesta https://github.com/swagger-api/swagger-codegen

Gruber, J. (2017). *Markdown*. Haettu 14.03.2017 osoitteesta https://daringfireball.net/projects/markdown/

IANA. (2017). Haettu 14.03.2017 osoitteesta Media types: http://www.iana.org/assignments/media-types/media-types.xhtml

IANA. (2017). *Hypertext Transfer Protocol (HTTP) Status Code Registry*. Haettu 06.04.2017 osoitteesta https://www.iana.org/assignments/http-status-codes/http-status-codes.xhtml

IEFT. (2017). *JSON Schema*. Haettu 01.06.2017 osoitteesta https://tools.ietf.org/html/draft-zyp-json-schema-04#section-3.5

IETF. (2014). *The JavaScript Object Notation (JSON) Data Interchange Format*. Haettu 21.07.2017 osoitteesta https://tools.ietf.org/html/rfc7159

IO, S. (2017). *Swagger Tools*. Haettu 31.8.2017 osoitteesta https://swagger.io/tools/

Jussilainen, M. (2015). *REST-pohjaisen web servicen kehittäminen.* Opinnäytetyö, Tietojekäsittelyn koulutusohjelma. Hämeen ammattikorkeakoulu. Haettu 26.02.2017 osoitteesta http://urn.fi/URN:NBN:fi:amk-2015121420602

Kankaanpää, S. (2016). *REST-arkkitehtuurityylin käyttö web-rajapinnnoissa.* Opinäytetyö, Tietotekniikan tutkinto-ohjelma. Seinäjoen ammattikorkeakoulu. Haettu 26.02.2017 osoitteesta http://urn.fi/URN:NBN:fi:amk-2016060611967

Koski, A. (2017). API-suunnittelun lähtökohdat. Luentosarja 2016-2017. Tampereen ammattikorkeakoulu.

Moilanen, J. (2017). API-Elinkaaren hallinta. Luentosarja 2016-2017. Tampereen ammattikorkeakoulu.

MSON Specification. (2016). Haettu 14.03.2017 osoitteesta https://github.com/apiaryio/mson/blob/master/MSON%20Specification.md

Open API initiative. (2016). Haettu 13.03.2017 osoitteesta History: https://www.openapis.org/faq#OAIFAQ-History

OpenAPI Specification version 2.0. (2017). Haettu 27.03.2017 osoitteesta https://github.com/OAI/OpenAPI-Specification/blob/master/versions/2.0.md

RAML. (2017). *The simplest way to design API*. Haettu 10.3.2017 osoitteesta https://raml.org/

Sandoval, K. (2017). *Standard API Definitions Demystified*. Haettu 13.03.2017 osoitteesta http://nordicapis.com/standard-api-definitions-demystified/

Swagger. (2017). *MuleSoft Joins the OpenAPI initiative*. Haettu 29.05.2017 osoitteesta Swagger.io: http://swagger.io/mulesoft-joins-the-openapi-initiative/

Trafi. (2017). *Avoin data trafissa*. Haettu 17.07.2017 osoitteesta https://www.trafi.fi/tietopalvelut/avoin_data

Vadudevan, K. (2017). *Swaggerhub*. Haettu 27.02.2017 osoitteesta Design First or Code First?: https://swaggerhub.com/blog/api-design/design-first-or-code-first-api-development/

Liite 1

OAS 2.0-kenttämääritykset

V	S	Kenttänimi	Tyyppi	Parametri	Pakollisuus
	0	swagger:	string	Versio nro	Kyllä
	0	info:	objekti		Kyllä
	1	title:	string	Otsikko	Kyllä
	1	version:	string	Versio	Kyllä
	1	description:	string	Kuvaus	Ei
	1	termsOfService:	string	Käyttöehdot	Ei
	1	contact:	objekti		Ei
	2	name:	string	Nimi	Ei
	2	url:	string	Yritys URL	Ei
	2	email:	string	S-posti	Ei
	1	license:	objekti		Ei
	2	name:	string	Lisenssitiedot	Ei
	2	url:	string	Lisenssi url	Ei
	0	host:	string	Nimi tai IP	Ei
	0	basePath:	string	/polku	Ei
	0	schemes:	objekti		Ei
	0	- http	string lista		Ei
	0	- https	string lista		Ei
	0	- ws	string lista		Ei
	0	- wss	string lista		Ei
	0	produces:	objekti		Ei
	0	- MIME nimi	string lista		Ei
	0	consumes:	objekti		Ei
	0	- MIME nimi	string lista		EI
	0	paths:	objekti		Kyllä
	1	/polunnimi:	string		Kyllä
	1	$ref:	string	'#/polku'	Ei
	2	get:			Kyllä Joku näistä
	2	put:			
	2	post:			
	2	delete:			
	2	options:			
	2	head:			
	2	patch:			
	3	tags:	objekti		Ei
	3	- tagin nimi	string lista		Ei
	3	summary:	string	Yhteenveto	Ei
	3	description:	string	Kuvaus	Ei
	3	operationId:	string	Kuvaus	Ei
	3	cosumes:	objekti	Yli kirjoittaa glo-baalin arvon	Ei
	3	produces:	objekti		Ei
	3	schemes:	objekti		Ei
	3	deprecated:	boolean	True / false	Ei
	3	parameters:	objekti		Kyllä
	4	- name:	string	Muuttujan nimi	Kyllä
	5			query	

5	in:		header	Kyllä Joku näistä
5			paths	
5			body	
5			from	
5	description:	string	Kuvaus	Ei
5	required:	boolean	true / false	Ei
5	type:		string	Kyllä Joku näistä
5			number	
5			integer	
5			boolean	
5			array	
5			file	
5	format:	integer	int32	Ei pakollinen Joku näistä
5		integer	int64	
5		number	float	
5		number	double	
5		string		
5		string	byte	
5		string	binary	
5		boolean	true/false	
5		string	date	
5		string	date-time	
5		string	password	
5	allowEmptyValue:	boolean	True / False	Ei
5	items:	objekti		Jos type on array
5	collectionFormat:		csv	Ei pakollinen. Joku näistä
5			ssv	
5			tsv	
5			pipes	
5			multi	
5	default:	string	arvo	Ei
5	maximum	number	arvo	Ei
5	exclusiveMaximum:	boolean	true / false	Ei
5	minimum:	number	arvo	Ei
5	exclusiveMinimum	boolean	true/false	Ei
5	maxLength	integer	arvo	Ei
5	minLength	integer	arvo	Ei
5	pattern	string	Regexp	Ei
5	maxItems	integer	arvo	Ei
5	minItems	integer	arvo	Ei
5	uniqueitems	boolean	true / false	Ei
5	enum	string	- lista	Ei
5	multipleOf	number	arvo	Ei
3	responses:	objekti		Kyllä
4	statuskoodi nro:		’numero’	Kyllä
5	descritption	string	Kuvaus	Kyllä
5	schema:	objekti		Ei
6	$ref:		’#/polku’	Ei
5	examples:	string		Ei
6	mimetype:	string	mime	-
7	mimearvo	string	mimearvo	-

	0	responses:	objekti		Ei
	1	viittausnimi:			Ei
	2	description:	string	’otsikko tieto’	
	2	schema:			Ei
	3	$ref:	string	’tietotyyppi polku’	Ei
	2	examples:	string		Ei
	3	mimetype:	string		-
	4	arvo:	string	arvon_määrä	-
	0	definitions:	objekti		Ei
	1	nimike jolla kutsutaan:	string		Ei
	2	type:		object	Ei
	2	properties:			Ei
	2	description:	string	Kuvaus	
	2	title:	string	Kuvaus	
	3	tietotyypin nimi:	string		Ei
	4	type:	string	Taulukko 12	Ei
	4	format	string	Taulukko 13	Ei

Liite 2

Case trafi rajapinta kuvaus

```
# Metadata
swagger: '2.0'
info:
  version: "1.0.0"
  title: Trafi ajoneuvodata API
  description: APIn data on Trafin avointa ajoneuvodataa
(www.trafi.fi) joka on ollut lähtötilanteessa pilkuilla erotettu .csv
tiedosto. Luotu rajapinta on osa HAMK opinnäytetyötä vuodelta 2017
  termsOfService: APIa voi käyttää vapaasti mahdollisten rajoitusten
puitteissa. Rajoitus määritykset voivat muuttua ilman erillistä ilmoi-
tusta.
  contact:
    name: Petteri Saarikko
    email: petteri.saarikko@gmail.com
  license:
    name: Trafi aineisto noudattaa kansainvälistä Creative Commons
(CC) 4.0 lisenssiehto mallia. Ajoneuvo datan version on 4.9 ja se on
julkaistu 1.4.2017.

# Kaikkia polkuja koskevat yleisasetukset
host: localhost:8080
basePath: /v1
schemes:
- http
produces:
- application/json
consumes:
- application/json

# Polut eli rajapinnan metodit
paths:
  /query:
    get:
      summary: Tämä kysely palauttaa dataa perustuen `kayttoonotto-
paiva` kentän arvoon. Data palautuu JSON muodossa ja sisältää Query-
responsen mukaisen palautteen
      description: Määritä kyselylle arvot start eli aloitus, end eli
lopetus limit eli palautettavien JSON lista-arvojen määrä.
      tags:
      - Limited
# Polun eli metodin parametrit ja tietotyyppi määritykset
      parameters:
        - name: start
          in: query
          description: Kyselyn aloitus aika esim. muodossa 19900101
          type: integer
          format: int32
          required: true
          default: 19900000
        - name: end
          in: query
          description: Kyselyn lopetusaika esim. muodossa 19913112
          type: integer
          format: int32
          required: true
          default: 19910000
```

```
        - name: limit
          in: query
          description: Palautettavien JSON lista objektien maksimi
määrä
          type: integer
          format: int32
          required: true
          default: 150

# Polun eli metodin vastaukset ja niiden tietotyyppi määrityksen
      responses:
        200:
          description: Ok
          schema:
            $ref: '#/definitions/QueryResponse'
        400:
          $ref: "#/responses/Standard400Error"

# Keskitetyt tietotyyppimääritykset alkavat tästä
definitions:

# Tietotyyppi määritys paluukoodille 400. Edellyttää, että logiikka
osaa palauttaa em. JSON objektin arvon.
  Error:
    type: object
    properties:
      code:
        type: string
      message:
        type: string

# Tietotyyppimääirtys paluukoodille 200
  QueryResponse:
    type: array
    items:
      type: object
      properties:
        co2:
          type: integer
          format: int32
        ahdin:
          type: integer
          format: int32
        ajonKokPituus:
          type: integer
          format: int32
        ajonKorkeus:
          type: integer
          format: int32
        ajonLeveys:
          type: integer
          format: int32
        ajoneuvoluokka:
          type: string
        ajoneuvokaytto:
          type: integer
          format: int32
        ajoneuvoryhma:
          type: string
        alue:
          type: integer
          format: int32
```

```
ensirekisterointipvm:
  type: string
iskutilavuus:
  type: integer
  format: int32
istumapaikkojenLkm:
  type: integer
  format: int32
jarnro:
  type: integer
  format: int32
kaupallinenNimi:
  type: string
kayttoonottopvm:
  type: integer
  format: int32
kayttovoima:
  type: integer
  format: int32
korityyppi:
  type: string
kunta:
  type: integer
  format: int32
mallimerkinta:
  type: string
matkamittarilumkema:
  type: integer
  format: int32
merkkiSelvakielinen:
  type: string
ohjaamotyyppi:
  type: integer
  format: int32
omamassa:
  type: integer
  format: int32
ovienLukumaara:
  type: integer
  format: int32
sahkohybridi:
  type : integer
  format: int32
suurinNettoteho:
  type: integer
  format: int32
sylintereideLkm:
  type: integer
  format: int32
teknSuurSallkokmassa:
  type: integer
  format: int32
tieliikSiirSallKokmassa:
  type: integer
  format: int32
tyyppihyvaksuntanro:
  type: integer
  format: int32
vaihteidenLkm:
  type: integer
  format: int32
vaihteisto:
```

```yaml
          type: integer
          format: int32
        valmistusnumero2:
          type: string
        vari:
          type: integer
          format : int32
        variantti:
          type: string
        versio:
          type: integer
          format: int32
        voimanvalJaTehostamistapa:
          type: integer
          format: int32
        yksittaiskayttovoima:
          type: integer
          format: int32

# Keskitetyt palautus määritykset
responses:
  Standard400Error:
    description: Bad request
    schema:
      $ref: "#/definitions/Error"
    examples:
      application/json:
        code: "400 Bad request"
        message: "Someting wrong on the query values"
```