Topi Takkunen

Haasteet REST-pohjaisten web-ohjelmointirajapintojen kehityksessä

Tietotekniikan Pro gradu -tutkielma

14. huhtikuuta 2019

Jyväskylän yliopisto

Informaatioteknologian tiedekunta

Tekijä: Topi Takkunen

Yhteystiedot: topi.a.takkunen@student.jyu.fi

Ohjaaja: Timo Hämäläinen

Työn nimi: Haasteet REST-pohjaisten web-ohjelmointirajapintojen kehityksessä

Title in English: Challenges in RESTful Web API Development

Työ: Pro gradu -tutkielma

Opintosuunta: Ohjelmistotekniikka

Sivumäärä: 40+0

Tiivistelmä: Tässä tutkimuksessa perehdytään REST-pohjaisiin web-ohjelmointirajapintoihin, niiden erinäisiin toteutushaasteisiin sekä ratkaisuihin.

Avainsanat: web-ohjelmointirajapinta, web-sovellusten integrointi, REST-pohjainen ohjelmointirajapinta, hypermedia, JSON, HTML, REST-pohjaisten ohjelmointirajapintojen testaus

Abstract: This thesis focuses on RESTful web APIs, the challenges in their development and solutions for those challenges.

Keywords: web API, web application integration, RESTful API, hypermedia, JSON, HTML, testing RESTful APIs

Sisältö

1 Johdanto

API on kirjainlyhenne, joka tarkoittaa ohjelmointirajapintaa. Ohjelmointirajapinnat ovat jo pitkään olleet osa ohjelmistokehitystä esimerkiksi käyttöjärjestelmien ohjelmien integraatioiden kautta. Higginbotham (2015, s. 1) kirjoittaa, että ohjelmointirajapinta määrittää, miten kaksi erillistä ohjelmiston osaa voivat toimia yhdessä. Hän jatkaa myös, että organisaatioiden sisäisten järjestelmien integraatioiden lisäksi moderneja ohjelmointirajapintoja käytetään myös liiketoimintavalmiuksien ja datan jakamiseen, yhteisöjen luomiseen sekä innovoinnin tukemiseen. Tämä on ilmeistä erityisesti web-ohjelmointirajapintojen käytössä, joiden toiminta keskittyy verkkosovellusten integrointiin ja yhteistoimintaan.

Viime vuosien aikana verkkosovellusten suosio on ollut kasvussa yleisen fokuksen siirtyessä enemmän perinteisistä työpöytäsovelluksista verkkosovellusten suuntaan. Tähän ovat vaikuttaneet monet asiat, kuten nykyaikaiset selaimet, mobiililaitteiden suosio sekä esineiden internet. Tämä suuntautuminen on vaikuttanut myös web-ohjelmointirajapintojen yleistymiseen, ja kun etenkin verkkosovellusten kehityksessä siirrytään jatkuvasti eteenpäin kohti nykyaikaisia ja uusia ratkaisuja, on myös ohjelmointirajapintojen kehittäjien pysyttävä mukana muutoksessa. On siis selkeästi havaittavissa, että myös ohjelmointirajapintojen parissa ollaan siirtymässä ohjelmistokehityksen trendien mukaisesti yhteistyötä, innovaatioita sekä uudelleenkäytettävyyttä tukevaan suuntaan.

Tässä tutkimuksessa perehdytään web-ohjelmointirajapintoihin, niiden erinäisiin toteutusvaihtoehtoihin, etuihin sekä heikkouksiin. Tutkimuksessa keskitytään nykyisten ja etenkin REST-pohjaisten web-ohjelmointirajapintojen ratkaisuihin ja ongelmiin, testattavuuteen, pinnalla oleviin ja tuleviin suuntauksiin, sekä pilvipalveluiden vaikutukseen web-ohjelmointirajapintojen toteutuksessa. Tämän lisäksi tutkitaan ja vertaillaan myös hieman muita mahdollisia tapoja sovellusten integroimiseen verkossa.

Lähdemateriaalin pohjalta tutkimusaihe on hyvin ajankohtainen. Aihetta on käsitelty useammasta näkökulmasta ja aiheen suosio itsessään on hyvä indikaattori tutkimuksen tarpeesta. Vaikka tutkimusaihetta on käsitelty monipuolisesti, sisältää se edelleen ongelmia, joiden ratkaisemisella on painoarvoa. Tutkimusaiheen nykyhetken ongelmat ja aiheen suosio luovat

myös otollisen pohjan tulevaisuuden ratkaisujen kehittämiselle sekä tutkimukselle.

Tutkimuksen tarkoituksena on selvittää, minkälaisia puutteita nykyhetken web-ohjelmointirajapintojen ratkaisuissa on, miten ja mihin ne vaikuttavat, sekä minkälaisia ratkaisuja ongelmiin on kehitetty. Tutkimuksen aikana selvitetään myös web-ohjelmointirajapintojen testattavuuden vaikutuksia kyseisiin ongelmiin. Tämän lisäksi tutkitaan, minkälainen vaikutus pilvipalveluihin siirtymisellä on web-ohjelmointirajapintoihin ja tutkimuksen aikana havaittuihin ongelmiin sekä ratkaisuihin. Lopuksi käydään läpi tulevaisuuden ratkaisuja sekä mahdollisia korvaavia integraatioratkaisuja web-ohjelmointirajapinnoille.

Tutkimus toteutetaan kirjallisuuskatsauksena. Menetelmä sopii hyvin aiheen tutkimiseen useammasta näkökulmasta sekä syvempään perehtymiseen itse aihealueeseen. Kirjallisuuskatsauksen avulla pääsee myös hyvin käsiksi siihen, miksi tutkimusaihe on ajankohtainen ja se helpottaa mahdollisten ongelmien kartoitusta. Menetelmä takaa myös hyvän pohjan mahdollisten tulevaisuuden ratkaisujen ja suuntauksien tarkasteluun.

Toisessa luvussa käydään tarkemmin läpi web-ohjelmointirajapintoja ja nykyhetken toteutuksia. Kolmannessa luvussa perehdytään hieman lähemmin REST-pohjaiseen arkkitehtuuriin ja vertaillaan sen toiminnallisuutta SOAP-protokollaan. Neljännessä luvussa perehdytään itse tutkimusongelmaan ja web-ohjelmointirajapintojen haasteisiin käymällä läpi mm. miten ne syntyvät, kuinka vakavia ongelmia ne voivat aiheuttaa sekä miksi ratkaisuvaihtoehtojen läpikäyminen on tärkeää. Viidennessä luvussa käydään läpi nykyhetken ratkaisuja, niiden toimivuutta, vahvuuksia sekä mahdollisia puutteita. Kuudennessa luvussa kerrataan pääkohdat ja tehdään johtopäätöksiä tutkimuksesta.

2 Web-ohjelmointirajapinnat

Verkkosivujen ja -sovellusten toimintatavat ovat jo vuosia eläneet muutosaikaa. Monet sivustot ovat jo siirtyneet ns. vanhanaikaisista staattisista kokonaisista sivuista modernimpiin, dynaamisista ja interaktiivisista osista koostuviin kokonaisuuksiin. Nykyaikaisilla verkkosovelluksilla ei ole tarvetta luoda kokonaisia verkkosivuja sovelluksen palvelinpuolella, vaan sivustoilla voidaan hakea dataa dynaamisesti JavaScriptin avulla yhden tai useamman web-ohjelmointirajapinnan kautta (Higginbotham 2015, s. 2). Web-ohjelmointirajapintojen avulla verkkosivuista voidaan siis tehdä interaktiivisempia käyttäjää varten, ja muokata sivun sisältöä sekä käyttäjälle tarjottavia toimintoja lataamatta koko sivua uudelleen. Tämä ensinnäkin tekee verkkosivun käyttämisestä sulavampaa ja toisekseen tuo mukaan lisää vaihtoehtoja verkkosovellusten kehittämisen työkaluihin sekä käytettävissä oleviin ohjelmointikieliin. Tätä kautta verkkosovellusten kehittäminen siirtyy myös enemmän ja enemmän palvelinpuolelta asiakaspuolelle. Ei ole tavatonta, että yksittäinen verkkosivu koostuu nykypäivänä useammasta verkkosovelluksesta tai -palvelusta, joilla kaikilla on omat erilliset toimintonsa verkkosivulla. Hetken aikaa oli yleistä nimittää useampien, toisiaan tai muita ulkoisia palveluita kutsuvien verkkosovellusten tai pilvipalveluiden yhdistelmiä nimellä mashup, mutta tuo termi on jäänyt pois käytöstä, koska tällaisesta rakenteesta tuli poikkeuksen sijaan normi (Sill 2015, s. 81). Yksi verkkosovellus voi esimerkiksi lisätä verkkosivulle arvosteluja, toinen kuvia, kolmas kommentteja ja neljäs esimerkiksi videoita.

Higginbotham (2015, s. 3) mukaan web-ohjelmointirajapintojen idea on välttää ylimääräisiä standardeja ja määrityksiä, ja määrittää toimintaperiaatteensa vain HTTP-standardin perusteella. Hän jatkaa, että näin toimittaessa mikä tahansa sovellus tai laite, jolla on verkkoyhteys (kuten mobiililaitteet, verkkoselaimet, autot), voi käyttää web-ohjelmointirajapintaa vain sisäänrakennettuja ohjelmointikirjastoja käyttäen, ja vältytään kalliilta ohjelmistoratkaisuilta sekä monimutkaisilta standardeilta. Huomion arvoista on, että tämä on parhaiten havaittavissa juuri REST-pohjaisissa ratkaisuissa. On selkeästi havaittavissa, että verkkopalveluiden integroinnin ei haluta lisäävän kompleksisuutta Internetin jo muutoin monitahoiseen rakenteeseen. Tässä voidaan myös nähdä viittauksia esineiden Internetiin, joka on vähitellen leviämässä erinäisiin osiin ihmisten arkea. Esineiden Internetillä on kuitenkin rajapinnoille

ja datan siirrolle melko lailla erilaiset vaatimukset kuin verkkosovelluksilla, sillä esineiden Internetin sovelluksilla resurssit ovat todennäköisesti vähäisemmät. Joka tapauksessa ohjelmointirajapinnat ovat yksi tapa, jolla esineiden Internetinkin sovellukset voivat kommunikoida keskenään.

Ohjelmointirajapintojen kautta myös yhteistyö lisääntyy sekä toiminnallisuuksien uudelleenkeksimisen tarve vähentyy. Sen sijaan, että jokainen ominaisuus kehitettäisiin sisäisesti, kehitysryhmät voivat kolmansien osapuolien ohjelmointirajapintoja hyödyntämällä lisätä uusia toimintoja kehitettävään sovellukseen ja täten keskittyä itse sovelluksen uniikkeihin ominaisuuksiin (Higginbotham 2015, s. 5). Tämä vaatii itse sovelluksen kehittäjiltä kuitenkin edelleen panostusta, sillä jokainen kolmannen osapuolen tarjoama ominaisuus on jollakin tavalla integroitava kehitettävään sovellukseen. Uskoisin, että kun sovelluksen ominaisuuksia täten ulkoistetaan ohjelmointirajapintojen kautta, eivät sovelluksen kehittäjät halua käyttää tolkuttomasti aikaa, rahaa tai muitakaan resursseja näiden integraatioiden toteuttamiseen. Kuten jo aiemmin todettiin, web-ohjelmointirajapintojen halutaan pysyvän mahdollisimman yksinkertaisina, ja on vaikea uskoa, että itse kehitysryhmätkään haluaisivat opetella monimutkaisten rajapintojen ominaisuuksia vain lisätäkseen sovellukseensa toimintoja, jotka olisivat mahdollisesti voineet toteuttaa itse. Kehittäjillä onkin web-ohjelmointirajapintojen maailmassa tärkeä osa. Higginbotham (2015, s. 6) mukaan ohjelmointirajapintoja markkinoidaan suoraan kehittäjille, ja kehittäjät vaativat intuitiivisia, helppokäyttöisiä sekä nopealla aikataululla integroitavissa olevia ohjelmointirajapintoja. Kirjoittajan mukaan näitä vaatimuksia kutsutaan kehittäjäkokemukseksi, ja niiden kautta ohjelmointirajapintojen suunnittelu muuttuu yhä enemmän kehittäjäkeskeisemmäksi. Tämä herättää ajatuksia siitä, kuinka ohjelmointirajapinnat nykyaikaistavat sovelluskehitystä. Usein perinteisesti vastaavaa sovelluskehityksessä käytettävien työkalujen, teknologioiden tai esimerkiksi ohjelmointikielten ja kirjastojen määrittelyä tekevät vain rajattu asiantuntijajoukko. Ja vaikka prosessin aikana monesti selvitetäänkin asianosaisten kehittäjien mielipiteitä, jää päätösvalta usein hyvin pienelle joukolle. Tavoitteetkin ovat kuitenkin monesti hieman erilaiset ja monimutkaisemmat kuin ohjelmointirajapinnoilla, mutta ohjelmointirajapintojen määrittelyn ja kehityksen yhteisöllisyydessä on paljon asioita, joita voitaisiin soveltaa yleisemmin.

Web-ohjelmointirajapinnat vaativat myös enemmän yhteistoimintaa kehittäjien välillä kuin

perinteiset ohjelmointirajapinnat. Higginbotham (2015, s. 23) mukaan perinteinen ohjelmistokehitys keskittyy usein sisäisten kehitysryhmien ympärille, joilla on kehitettävän ohjelman lähdekoodi käytettävissään. Kirjoittaja jatkaa, että web-ohjelmointirajapinnat sen sijaan vaativat yhteistyötä ohjelmointirajapintaa käyttävien sekä sen toteuttaneiden kehittäjien välillä, ja ohjelmointirajapintaa käyttävillä kehittäjillä on vain ohjelmointirajapinnan malli sekä dokumentaatio käytettävissään ja ohjaamassa heidän toimintaansa. Kirjoittaja lisää vielä, että web-ohjelmointirajapinnan käyttäjillä ei siis ole pääsyä ohjelmointirajapinnan lähdekoodiin tai tietokantaan, eivätkä he voi sitä kautta selvittää ohjelmointirajapinnan toiminnallisuuksien yksityiskohtia. Käytännössä tämä tarkoittaa sitä, että web-ohjelmointirajapinnan asiakassovelluksen kehittäjien on saatava tarvittavat tiedot web-ohjelmointirajapinnan toteutuksesta ja toiminnasta muulla tavalla. Usein tämä tarkoittaa kommunikointia web-ohjelmointirajapinnan toteuttajien tai ylläpitäjien kanssa, sekä web-ohjelmointirajapinnan dokumentaation läpikäyntiä.

Edellä mainitulla sovelluksen toiminnallisuuksien yksityiskohtien piilottamisella voi olla monia haitallisiakin vaikutuksia, jos asiakassovellusten kehittäjät eivät ole siihen tottuneet. Tämä voi helposti vaikuttaa muun muassa ohjelmointirajapinnan toteutukseen, jos asiakassovellusten kehittäjät eivät ole tyytyväisiä. Hadley, Pericas-Geertsen ja Sandoz (2010, s. 11) mukaan ohjelmointirajapinnan palvelinpuolen ja sen asiakassovellusten välillä voi myös olla eräänlainen sopimus rajapinnan tarjoamista resursseista ja niiden suhteista, joka on joko staattinen tai dynaaminen. Kirjoittaja jatkaa, että staattisen sopimuksen mukaan ainakin osa palvelinpuolen toteutuksen tuntemuksesta on sulautettu asiakassovelluksiin, eikä tätä sopimusta voida päivittää kirjoittamatta asiakassovelluksia uudelleen. Kirjoittaja mainitsee myös, että dynaamisen sopimuksen toteutus taas on asiakassovellusten selvitettävissä ajon aikaisesti, ja asiakassovellukset voivat sopeutua sen mukaisesti. Jos näitä vaikutuksia ei oteta huomioon ohjelmointirajapinnan suunnittelu- ja toteutusvaiheessa, vaan toimitaan sellaisten asiakassovellusten kehittäjien toiveiden mukaan, jotka eivät syystä tai toisesta halua luopua perinteisemmän ohjelmistokehityksen mukaisesta palvelinpuolen toteutuksen tuntemuksesta, on ohjelmointirajapinnan toteuttajilla edessään paljon haasteita. Nämä haasteet liittyvät mm. tiukkaan kytkentään, rajapinnan päivittämiseen ja versiointiin, sekä asiakassovellusten hajoamiseen, ja niitä käsitellään vielä tarkemmin tässä tutkimuksessa.

Joka tapauksessa ohjelmointirajapintojen käyttö on tasaisessa nousussa ja levinnyt myös nykyaikaisten suuntausten mukaisesti. Ohjelmien siirtyessä yritysten omiin tiloihin asennetuista ratkaisuista ohjelmiston hankintaan palveluna (Software-as-a-Service) on myös rajapintojen suosio kasvanut etenkin datan käsittelyssä sekä integraatioiden automatisoinnissa (Higginbotham 2015, s. 4). Itse asiassa iso osa pilviohjelmistojen erillisten komponenttien välisestä interaktiosta tapahtuu etämetodien kautta, ja näitä parametrisyötteisiä rajapintoja kutsuttiin ennen nimellä RPC (Remote Procedure Call), mikä on edelleen hyvin tarkka kuvaus niistä (Sill 2015, s. 80-81). Kirjoittajan mukaan termiä API käytetään pilviympäristöissä rajapinnoista, jotka kelpuuttavat parametrien vaihdon sellaisten ohjelmistokomponenttien välillä, joiden fyysinen sijainti eroaa usein toisistaan ja jotka eivät ole riippuvaisia rajapinnan toteutuksen yksityiskohdista. Pilviympäristöjen sekä sitä mukaa myös rajapintojen yleisyys kasvaa edelleen, mutta termimääritykset vaikuttavat olevan vielä hieman epämääräisiä. Esimerkiksi laajoja luokkia konseptuaalisesti erilaisia rajapintoja kutsutaan pilviympäristöissä ohjelmointirajapinnoiksi, vaikka niillä ei ole sovellusten ohjelmoinnin kanssa paljoakaan yhteistä, ja vaikka ne loppujen lopuksi ovat yksinkertaisesti vain parametrisyötteisiä rajapintoja, joita vain on mahdollista käyttää URI:n kautta (Sill 2015, s. 80-81). Kirjoittajan mukaan yksi ongelma tästä on se, että tämä parametrisyötteinen osa sovellusta on usein vain ainoa osa, jota palveluntarjoaja tarjoaa ulospäin käytettäväksi. Kirjoittaja jatkaa, että verkkopalveluiden piirissä tätä voidaan kutsua web-ohjelmointirajapinnaksi, mutta vastaavaa termiä ei ole vielä kehitetty vastaavaa käyttöä varten pilviympäristöissä, jotta tämän tyyliset rajapinnat voitaisiin erottaa suorista operaatioista, jotka ovat mahdollisia rajapintakerroksen kautta suoraan lähdekoodin tasolla. Pilviympäristöissä vaatimukset ja käyttötarkoitukset ovat selkeästi monimuotoisempia perinteisempiin verkkopalveluihin verrattuna, eivätkä web-ohjelmointirajapinnat sovellu suoraan kaikkiin pilviympäristöjen käyttötarkoituksiin. Pilviympäristöissä onkin selvästi havaittavissa tarve verkkopalveluissa käytettävistä web-ohjelmointirajapinnoista eroaville rajapinnoille.

3 REST ja SOAP

Web-ohjelmointirajapintojen maailmassa palvelut on useimmiten jaoteltu joko SOAP- tai REST-pohjaisiin palveluihin. Näistä SOAP (Simple Object Access Protocol) oli aikanaan monien mielestä johtava verkkopalveluiden kommunikoinnin standardiprotokolla (Kankanamge 2012, luku 1.3). Kirjoittajan mukaan se suunniteltiin laajennettavaksi niin, että muita standardeja pystyttäisiin integroimaan sen kanssa. SOAP:ssä on kuitenkin puutteita, joista kirjoittaja mainitsee suorituskyvyn laskun raskaan XML-prosessoinnin seurauksena, sekä usean WS-* -spesifikaation käytöstä johtuvan kompleksisuuden. Muun muassa näiden seurauksena syntyi REST (Representational State Transfer), jota voidaan pitää kevyenä vaihtoehtona raskaille ja monimutkaisille SOAP-pohjaisille verkkopalvelustandardeille (Kankanamge 2012, luku 1.3). Kirjoittajan mukaan REST-pohjaisissa verkkopalveluissa painotus on pisteestä pisteeseen -kommunikaatiossa HTTP:n välityksellä käyttäen alun perin enimmäkseen XML-viestejä. Moni suosii REST-pohjaista toteutustapaa SOAP:n sijaan käytännössä REST:n helppokäyttöisyyden ja verrattavan vaivattomuuden takia. Näihin REST:n vahvuuksiin viittaa selkeästi ainakin sen samankaltaisuus WWW:n kanssa, kuten pohjautuminen samoihin standardeihin ja protokolliin.

REST:n helppokäyttöisyydessä on kuitenkin myös ongelmansa, eikä SOAP:n käyttö ole täysin lakannut ohjelmointirajapintojen toteutusvaihtoehtona. Tähän on omat syynsä, sillä myös SOAP:ssä on omat vahvuutensa. SOAP:n voidaan esimerkiksi nähdä omaavan enemmän vaihtoehtoja viestien lähetysmetodiksi kuin REST:n (Makkonen 2017, s. 37). SOAP:ssä on myös turvallisuuden kannalta monipuolisemmat valmiudet tukien sekä HTTPS:ää että standardoitua WS-Security -moduulia (Makkonen 2017, s. 38). Kirjoittaja jatkaa, että SOAP on ollut käytössä myös pidempään, ja SOAP-pohjaista kehitystä varten on täten myös kehitetty enemmän hyödyllisiä työkaluja kuin modernia REST:tä varten. Nykypäivänä ei siis ole mikään ihme, että SOAP-pohjaisia ratkaisuja toteutetaan edelleen tiettyjä käyttötarkoituksia varten, joissa esimerkiksi turvallisuus ja toimintavarmuus ovat etusijalla.

Muun muassa useimmat nykypäivän pilvipalveluiden SLA-spesifikaatioiden WS-Agreement -toteutuksista tukeutuvat tunnustettuun ja ohjeelliseen SOAP-verkkopalveluiden päällä toimivaan SOA-lähestymistapaan, mutta REST-pohjaisten arkkitehtuurien käyttö on nopeas-

ti leviämässä, luvaten parempaa suorituskykyä sekä skaalautuvuutta (Benedictis ym. 2015, s. 93). Nykyisistä ohjelmointirajapinnoista siis ainakin osalla SOAP on edelleen hyvin varteenotettava vaihtoehto REST:lle, ja molempia tullaan varmasti vielä käyttämään myös tulevaisuudessa. Myös Sill (2015, s. 81) mainitsee, että itse asiassa URI-pohjaisia etäkutsuja käyttäviä ohjelmointirajapintoja pidetään usein ainoana mahdollisena tapana tehdä etäkutsuja pilviympäristöissä, ja suurin osa näistä on nykypäivänä toteutettu REST:n tai SOAP:n avulla HTTP:n tai HTTPS:n välityksellä välittäen parametrejä tekstinä, JSON- tai XML-muodossa. Kirjoittajan mukaan muitakin kuljetusprotokollavaihtoehtoja kuitenkin on, kuten AMQP, eli Advanced Message Queuing Protocol tai XMPP, eli Extensible Messaging and Presence Protocol, joita molempia käytetään viestipainotteisissa väliohjelmistoissa, ja joissa molemmissa on ominaisuuksia, joiden kautta ne ovat suotavampia asynkronisissa viestivälityssovelluksissa kuin HTTP. Tällä ei kuitenkaan vaikuta tällä hetkellä olevan suurempaa vaikutusta REST-pohjaisen tai SOAP-pohjaisen ratkaisun valintaan, sillä valinta näiden kahden toteutustekniikan välillä perustuu usein hieman kokonaisvaltaisempiin ominaisuuksiin.

REST:n ja SOAP:n ominaisuuksissa on havaittavissa paljon selkeitäkin eroja. Esimerkiksi toiminnan ideologia on erilainen REST:n ja SOAP:n välillä, sillä REST-pohjaiset palvelut perustuvat resurssin käsitteeseen, kun taas SOAP käyttää toimintoja hakeakseen tarvittavan datan (Makkonen 2017, s. 36). Kirjoittajan mukaan molemmissa tyyleissä on omat hyötynsä, sillä REST:n resurssien käsittely mahdollistaa muun muassa helposti seurattavan datan käytön, kun taas SOAP mahdollistaa luotettavan tavan prosessoida tiettyjä tehtäviä. Sekä REST:ssä että SOAP:ssä on muitakin hyötyjä, joiden vaikutukset voivat näkyä pitkällekin ohjelmointirajapinnan elinkaaressa. Valinnan näiden kahden välillä on suositeltavaa perustua ensi sijassa siihen, mitkä ovat ohjelmointirajapinnan käyttötarkoitukset, ja siihen, mitkä ominaisuudet ovat tärkeimpiä.

Moni ominaisuus tai käyttötarkoitus voi siis vaikuttaa valintaan REST- ja SOAP-pohjaisen ratkaisun välillä, ja yksi vaikuttavimmista asioista on niiden pohjimmainen ero. REST voidaan määritellä arkkitehtuurityylinä, jonka avulla suunnitellaan ohjelmistoja (Kankanamge 2012, luku 8.1). Kirjoittaja jatkaa, että REST ei ole spesifikaatio tai W3C-standardi kuten SOAP, ja tästä syystä REST-pohjaisten palveluiden kanssa työskentely on suhteellisesti helpompaa, eivätkä ne yleensä vaadi väliohjelmistojen viitekehyksiä ollakseen käytettävis-

sä, vaan useimmiten toteutus onnistuu suoraan ohjelmointikielten standardeilla kirjastoilla. Tässä piilee kuitenkin samalla vaara sille, että REST antaa kehittäjälle liikaa vapautta, josta seuraa helposti ongelmia toteutuksen myöhemmissä vaiheissa. On kuitenkin hyvä, että REST:n hyödyntämistä varten on tarjolla ohjeita ja hyviä käytänteitä, joita kehittäjä voi seurata saadakseen toteutuksensa mahdollisimman kestävälle ja toimivalle tasolle. Muun muassa Kankanamge (2012, luku 8.1) mukaan REST:n avainperiaatteita ovat kaiken edustaminen yksilöllisillä tunnisteilla (URI), standardien HTTP-metodien käyttö (GET, POST, PUT ja DELETE), resurssien linkittäminen keskenään, resurssin useamman representaation mahdollistaminen sekä tilaton viestintä.

REST:n periaatteiden pohjalta voidaan tehdä myös muitakin johtopäätöksiä. Etenkin HTTP-metodien painotuksen sekä resurssien tunnisteiden pohjalta voidaan todeta, että REST:ssä on monia samankaltaisuuksia Internetin kanssa. Tästä HTTP-pohjaisesta interaktiosta johtuen WWW:tä voidaan pitää täydellisenä representaationa REST-pohjaisesta arkkitehtuurista (Kankanamge 2012, luku 8.1). Tämän lisäksi kirjoittaja mainitsee myös, että hyvä REST-pohjaisen suunnittelun käytäntö on, ettei sisällytä liikaa informaatiota yhteen resurssiin, vaan resurssin pitäisi sisältää linkkejä lisäinformaatioon HTML-sivujen linkkien tapaan. Tältä pohjalta, kun ajattelee REST:n suosion kasvua ja vahvuuksia, on yhteys melko selkeästi nähtävissä. Kun REST on näinkin vahvasti linkittynyt Internetin ja WWW:n toimintaan ja samankaltaisuuksia on paljon, eivät REST:n suosio ja helppokäyttöisyys ole mikään ihme.

REST:n periaatteiden ja WWW:n välillä on myös muita yhteyksiä, jotka on helppo havaita, kun REST:n periaatteita avaa hieman tarkemmin. Esimerkiksi useamman representaation mahdollistaminen voidaan muotoilla myös esitysmuodon tai mediatyypin valinnaksi niin, että samaa resurssia voidaan käsitellä useammassa eri esitysmuodossa, tai resurssin representaatio voi käyttää useampaa eri mediatyyppiä. Tähän tarkoitukseen voidaan soveltaa HTTP:n sisältöneuvottelun periaatetta, jonka avulla sovelluksen informaatioon pääsee käsiksi useamman tyyppisten asiakassovellusten kautta, jotka käyttävät eri tyyppisiä representaatioita, kuten HTML:ää tai XML:ää (Kankanamge 2012, luku 8.1). Tässä HTTP mahdollistaa hyvin yksinkertaisen tavan hyödyntää useampaa esitysmuotoa, joka myötäilee REST:n periaatteita. REST:n periaatteissa viitataan resurssien linkittämisellä myös hypertekstipohjaiseen navigaatioon, joka on myös sidoksissa edellä mainittuun esitysmuodon valintaan.

Hypertekstipohjaisen navigaation periaatteen mukaan resurssin mediatyypin määrittäminen resurssin URI:ssä ei ole hyvä käytäntö, koska se tekee resurssin representaatiosta ja tunnisteesta riippuvaisia toisistaan, representaation kehittämisestä tunnistetta muuttamatta vaikeaa, sekä poistaa asiakassovellukselta mahdollisuuden neuvotella sille parhaiten sopivaa mediatyyppiä (Li ym. 2016, s. 157). Hypertekstipohjaista navigointia käydään tarkemmin läpi seuraavissa kappaleissa.

URI:n, HTTP-metodien, hypertekstipohjaisen navigaation ja resurssin useamman representaation lisäksi REST:n periaatteissa mainittiin myös tilaton viestintä. REST määrittää, että palvelimen ei pitäisi säilyttää viestinnän tilaa yhtä pyyntöä enempää, mikä on erittäin tärkeää löysän kytkennän saavuttamiseksi REST-pohjaisen palvelun ja sen asiakassovellusten välillä (Kankanamge 2012, luku 8.1). Kirjoittajan mukaan yhden asiakkaan tekemän pyynnön pitäisi sisältää kaikki tarpeellinen tieto, jota tarvitaan palvelinyhteyteen. Hyötynä tästä kirjoittaja mainitsee, että palvelimen ei näin ollen tarvitse säilyttää informaatiota asiakassovellusten tilasta, eikä palvelin täten mene tukkoon useamman asiakkaan samanaikaisista palvelupyynnöistä. Tässä on havaittavissa eroavaisuutta REST:n ja WWW:n toiminnan välillä. Monilla verkkosivuilla on tapana säilyttää tietoja verkkosivun käyttäjästä ja tämän sessiosta. Tämä kuitenkin mahdollistaa verkkosivujen monipuolisemman toiminnan, ja REST:n käyttötarkoitukset ovat usein rajoitetumpia eikä REST:ltä vaadita yhtä monimuotoista toiminnallisuutta.

Web-ohjelmointirajapinnoista on tullut tärkeä perusta monille verkkosovelluksille, kuten pankki-, vakuutus- ja vähittäiskaupan sovelluksille (Vu, Fertig ja Braun 2018, s. 1881). Kirjoittajan mukaan web-ohjelmointirajapintojen määrä kasvaa, ja ohjelmointirajapinnan asianmukaisen suunnittelun tarve on tärkeämpää kuin koskaan, sillä hyvin tehty ohjelmointirajapinta voi olla yrityksen suurimpia etuja, mutta huonosti toteutettu ohjelmointirajapinta voi olla yksi yrityksen pahimmista rasitteista. Ohjelmointirajapintojen suunnitteluun ja toteutukseen liittyy täten paljon liiketoimintakriittisiä vaatimuksia sekä erikseen teknisiä haasteita, joiden käsittely voi vaikuttaa hyvin paljon jokaisen ohjelmointirajapinnan lopulliseen toteutukseen ja laatuun.

4 Haasteet

REST-pohjaisten ohjelmointirajapintojen suunnittelussa, toteutuksessa sekä ylläpitämisessä on lukuisia haasteita. Koska REST on enemmän arkkitehtuurityyli kuin määrittely tai standardi, REST-pohjaisen ohjelmointirajapinnan asianmukainen suunnittelu ei ole yksinkertaista, lähinnä siitä syystä että kehittäjien on käsiteltävä suurta määrää suosituksia ja hyviä käytäntöjä, ja etenkin asianmukainen hypermediarajoitteen soveltaminen vaatii kokemusta (Vu, Fertig ja Braun 2018, s. 1881). Haasteita on kuitenkin myös ohjelmointirajapintojen käyttäjillä, joiden on valittava käyttämänsä rajapinnat joidenkin kriteerien avulla. Näitä kriteerejä on määritelty useammankin tutkimuksen pohjalta, ja tärkeimmät kriteerit ovat kuitenkin verrattavan vaivattomasti valittavissa. Muun muassa Higginbotham (2015, s. 5) mukaan kolmannen osapuolen ohjelmointirajapinnan tarjoajalla olisi oltava pitkäikäisyyttä indikoiva liiketoimintamalli, jotta ohjelmointirajapintakin olisi pitkäikäinen, ja hyvin dokumentoitu ohjelmointirajapinnan versiointi- sekä päivitysprosessi, jotta jo olemassa olevat sovellukset eivät hajoa. Kirjoittajan mukaan hyvä ohjelmointirajapinta on myös kattavasti dokumentoitu kehittäjiä varten, ja sen olisi myös tarjottava tukiprosesseja hankalia tilanteita varten. Jo nämä kriteerit tuovat mukanaan erinäisiä haasteita ohjelmointirajapintojen käyttäjille ja erityisesti ohjelmointirajapintojen kehittäjille.

4.1 Versiointi

Yksi hankalimmista web-ohjelmointirajapintojen haasteista on versiointi. Suurin osa yrityksistä, jotka tarjoavat ohjelmointirajapintoja, eivät koskaan päivitä ohjelmointirajapintaansa sen ensimmäisen julkaisun jälkeen, vain koska eivät voi (Richardson, Amundsen ja Ruby 2013, s. 185). Kirjoittaja jatkaa, että tämä johtuu ohjelmointirajapinnan tarjoajien välinpitämättömyydestä hypermediarajoitteita kohtaan. Versiointi voi siis käytännössä olla haasteellista silloin, kun muita ohjelmointirajapinnan kehitysvaiheissa aiemmin vastaan tulleita haasteita ja ongelmia ei ole kunnolla ratkaistu. Versioinnin haasteethan tulevat esiin vasta silloin, kun jo julkaistuun ohjelmointirajapintaan halutaan tehdä muutoksia, ja versioinnin haasteet nähdään usein pikemminkin osana toisia ohjelmointirajapinnan kehityksessä havaittavia isompia ongelmia. Versiointiin liittyviä haasteita käydäänkin täten tässä tutkimuksessa

tarkemmin läpi muiden REST-pohjaisten haasteiden yhteydessä myöhemmissä kappaleissa.

4.2 JSON

Haasteita liittyy kuitenkin muihinkin ohjelmointirajapintojen aspekteihin. Yksi näistä on käytettävä formaatti, ja yksi suosituimpia formaatteja tällä hetkellä on JSON. JSON, eli Javascript Object Notation on kevyt tiedonvaihtoformaatti, joka pohjautuu JavaScript-ohjelmointikielen osajoukkoon käyttäen avain-arvo-pareja kuvaamaan viestissä kulkevaa tietoa (Kankanamge 2012, luku 1.3). JSON on kevyt, yksinkertainen ja suosittu esitysmuoto, mutta myös siinä on omat puutteensa. JSON ei ensinnäkään ole hypermediaformaatti, sillä vaikka sen standardi määrittelee käsitteitä kuten numerot, listat, merkkijonot ja objektit, se jättää määrittelemättä linkit sekä linkkien suhteet, joten sillä ei ole hypermedian vaatimia valmiuksia (Richardson, Amundsen ja Ruby 2013, s. 168). REST:n periaatteiden noudattamisen kannalta tämä on huono asia, sillä hypermedialla on tärkeä rooli REST:n periaatteissa. Hypermedian tärkeyttä käydään tarkemmin läpi seuraavissa kappaleissa.

Suosittuja formaatteja on muutama muukin, ja osa niistä on oikeastaan suositumpi muissa käyttötarkoituksissa. Yksi tällainen formaatti on HTML. HTML on hallitseva esitysmuoto ihmisten käyttämässä verkossa, mutta se ei silti ole suosituin formaatti web-ohjelmointirajapinnoissa, vaan sitä suositumpia ovat ainakin JSON sekä XML (Richardson, Amundsen ja Ruby 2013, s. 143-144). Kirjoittajan mukaan tämä tarkoittaa kuitenkin sitä, että suurin osa nykypäivän ohjelmointirajapinnoista ei hyödynnä mikroformaatteja tai mikrodataa edes tilanteissa, joissa se olisi järkevää, koska näitä profiileja ei voida käyttää JSON tai XML -dokumenteissa. Kirjoittaja jatkaa vielä, että JSON-pohjaista esitysmuotoa käytettäessä ei ole mitään tapaa käyttää muiden kehittäjien luomia sovellussemantiikkoja uudelleen, vaan profiili on luotava itse. Tämä tarkoittaa ensinnäkin asioiden turhaa uudelleen keksimistä ja toistoa. Tämän lisäksi, kun profiili luodaan näillä käytänteillä, pitää se sovellussemantiikan erillään itse sovelluksesta. Ja kun ohjelmointirajapintaa varten halutaan luoda asiakassovelluksia, on käyttäjien luettava profiili erikseen jotta voivat luoda toteutuksen ohjelmointirajapinnan sovellussemantiikalle (Richardson, Amundsen ja Ruby 2013, s. 143-144). Tämän pohjalta voidaan todeta, että vaikka JSON onkin yksi käytetyimpiä viestiformaatteja ohjelmointirajapinnoissa, on se sellaisenaan hyvin vajavainen. Näitä vajavaisuuksia ja puutteita varten on kuitenkin kehitetty

erinäisiä ratkaisuja, joita käydään läpi tämän tutkimuksen myöhemmissä kappaleissa.

4.3 REST

Käytetystä viestiformaatista riippumatta REST-pohjaisten ohjelmointirajapintojen toteutuksen ja ylläpidon suurimmat haasteet kohdistuvat kuitenkin REST:n omiin ominaisuuksiin. REST-pohjaisissa ohjelmointirajapinnoissa on suosiostaan huolimatta edelleen paljon haasteita, eikä niiden toteutus todellakaan ole suoraviivaista tai yksinkertaista. Vu, Fertig ja Braun (2018, s. 1881) mukaan REST-pohjaisten ohjelmointirajapintojen kehitys on vaikeaa ohjelmistorunkojen puutteen takia, joiden tarkoitus on ohjata toteutusta. Kun sovelluksen suunnittelusta ja kehityksestä puuttuu runko, jonka ympärille toteutuksen voisi rakentaa, ei ole ihme, että lopputuloksena monissa REST-pohjaisissa ohjelmointirajapinnoissa on puutteita. Tämän lisäksi Davis (2012, s. 3) mielestä yksi suurin syy puutteellisten REST-pohjaisten ohjelmointirajapintojen vajaavaisuuksille on se, että kehittäjät eivät todellisuudessa ymmärrä REST-arkkitehtuuria, vaikka uskovatkin ymmärtävänsä. Tässä on melko iso kontrasti itse REST-pohjaisen ohjelmointirajapinnan kehittäjien ja asiakassovelluksen kehittäjien välillä. Se näkyy esimerkiksi siinä, että Schreibmann ja Braun (2015, s. 5) mukaan REST:n pääasiallinen etu muihin vaihtoehtoihin kuten SOAP:hen nähden on REST-pohjaisen ohjelmointirajapinnan käytön yksinkertaisuus, mikä onkin yksi syy sille miksi REST-pohjaiset rajapinnat kasvattavat suosiotaan verkko- ja mobiilisovellusten kehittäjien keskuudessa. Kirjoittajan mukaan muita syitä ovat REST-pohjaisen rajapinnan selkeät tarkoitusperät sekä rajapinnan käyttäjän opastus hypermedian avulla. Monille REST-pohjaisten ohjelmointirajapintojen kehittäjille REST siis on käytännössä täysin erilainen verrattuna siihen, miltä se vaikuttaa ohjelmointirajapintojen käyttäjille. Toisin muotoiltuna ongelman ydin voi olla siinä, että sekä REST-pohjaisen rajapinnan kehittäjät että sen käyttäjät pitävät REST:tä yksinkertaisena ja helppokäyttöisenä, mutta todellisuudessa tämä on totta vain käyttäjille.

Edellä mainitut syyt ja edut ovat kuitenkin vain ohjelmointirajapinnan ulkokuorta, eivätkä kata koko rajapinnan toiminnallisuutta tai tarkoitusta. Ja jos Ohjelmointirajapinnan perusta sekä toteutus palvelimella on heikko, heikentää se myös rajapinnan asiakassovelluksille näkyviä osa-alueita. Myös Schreibmann ja Braun (2015, s. 5) toteaa, että REST-pohjaisten rajapintojen taustalogiikan suunnittelu ja toteutus on jokseenkin hankalaa, virhealtista ja aikaa

vievää, koska käsitteellinen aukko pohjimmaisten REST:n periaatteiden sekä niiden toteutuksen välillä on todella suuri. Kirjoittajan mukaan tälle on kaksi syytä, joista ensimmäinen on se, että REST on arkkitehtuurityyli, ja täten avoin tulkinnalle. Kirjoittaja jatkaa, että kehittäjä voi halutessaan yksinkertaisesti valita vain muutaman REST:n periaatteen, joita noudattaa, ja jättää muut periaatteet huomioimatta. Toinen syy kirjoittajan mukaan on kattavien työkalujen puute REST-pohjaisten rajapintojen suunnittelua, dokumentointia ja testausta varten, sillä ohjelmointikielten tasolla käytettävissä olevat sovellusrungot ja kirjastot keskittyvät vain alhaisen tason toteutuksiin. Tältä pohjalta ei ole yllättävää, että monet näennäisesti REST-pohjaiset ohjelmointirajapinnat eivät tue REST:n periaatteita täysin. Ensinnäkin, jos REST:n periaatteita ei ymmärretä, on niitä myös vaikea toteuttaa. Toisekseen, mikäli kyseessä on yksinkertaista tarkoitusta varten luotava rajapinta, joka mahdollisesti hyödyntää vain muutamia resursseja, eivät monet kehittäjät todennäköisesti näe esimerkiksi hypermediaa ollenkaan tarpeellisena. Tähän kun liitetään saatavilla olevien sovellusrunkojen puute, joiden avulla voitaisiin esimerkiksi painottaa ja kontrolloida REST:n periaatteiden toteutusta, on edellä mainittu toimintatapa hyvin ymmärrettävissä. Näissä tapauksissa ei rajapintaa voida loppujen lopuksi kuitenkaan kutsua REST-pohjaiseksi.

Eivätkä nämä ole ainoat haasteet. Vaikka REST-pohjaisten ohjelmointirajapintojen suosio on kasvussa, ei siitä huolimatta ole olemassa standardoitua ja konekielistä tapaa kuvata REST-pohjaista ohjelmointirajapintaa samalla tasolla mihin WSDL pystyy RPC-tyylisten verkkopalveluiden osalta (Li ym. 2016, s. 156). Ja koska REST-pohjaisilla palveluilla ei ole standardia jonka avulla kuvata palvelua syntaktisesti ja semanttisesti, kuvaillaan REST-pohjaisten palveluiden toiminnallisuus sekä käyttöohjeet useimmiten erillisen tekstidokumentaation avulla Cretella ja Martino (2012, s. 33). Kirjoittajan mukaan konekielisen kuvauksen puute haittaa myös REST-pohjaisten palveluiden automaattista etsintää, joka usein hyödyntää juurikin konekielisten kuvausten läpikäymistä. Hakutoiminnallisuuden lisäksi konekielisen kuvauksen puute on helppo yhdistää myös REST-pohjaisen ohjelmointirajapinnan päivittämisen ja versioinnin haasteisiin. Erillinen tekstidokumentaatio kun ei kuitenkaan päivity itsestään ohjelmointirajapinnan muuttuessa, vaan sekin on päivitettävä erikseen.

Tätä puutetta korjaamaan on REST:lle Li ym. (2016, s. 156) mukaan ehdotettu montaa kuvauskieltä, mukaan lukien WADL, RAML, Swagger, RSML, API Blueprint, RADL ja REST

Chart. Cretella ja Martino (2012, s. 31) sen sijaan puhuu samaisesta puutteesta verkossa käytettävien pilvipalveluiden REST-rajapinnoissa natiivien ja formaalien perustietojen tarjonnan puutteena, ja hänen mukaansa tämän puutteen korjausvaihtoehtoja ovat muun muassa hRESTS, SA-REST ja RESTdesc. Cretella ja Martino (2012, s. 33) mukaan hRESTS, eli HTML for RESTful Services on mikroformaatti, joka mahdollistaa web-ohjelmointirajapinnan konekielisen kuvauksen luonnin HTML-dokumenttina. Kirjoittaja kuvailee myös SA-REST:n, joka rikastuttaa ihmisluettavaa HTML-dokumentaatiota selitteillä, joiden avulla dokumentaatiosta saadaan konekielinen, sekä WADL:n (Web Application Description Language), joka sen sijaan kuvailee syntaktisesti koko palvelun kattavan XML-tiedoston kautta, jota voidaan edelleen rikastuttaa kirjoittajan mukaan semanttisesti esimerkiksi OWL-S:n avulla. Kirjoittaja painottaa kuitenkin edelleen, että WADL ei ole saavuttanut kovin laajaa suosiota eikä siitäkään ole tullut REST-pohjaisten palveluiden standardia, sekä mainitsee monien yllä mainittujen kuvausteknologioiden rajoitteena niiden kohdistumisen teknisiin aspekteihin kuten parametreihin tai kutsumuotoihin. Myös Schreibmann ja Braun (2015, s. 6) mainitsee, että WADL pystyy kyllä muodollisesti kuvaamaan REST:n, mutta ei REST-pohjaisia ohjelmointirajapintoja siitä puuttuvan hypermedian tuen takia. Vaikka mitään kuvauskieltä ei REST:lle ole vielä standardoitu, ei se kuitenkaan tarkoita sitä, että edellä mainitut kuvauskielet olisivat hyödyttömiä. Moni kuvauskieli korjaa monia REST:n puutteita, ja osa kuvauskielistä nousee selkeästi suositummaksi kuin toiset.

RAML (RESTful API Modeling Language) on yksi viime aikoina enemmän esiinnousseista REST:n kuvauskielistä, mikä näkyy muun muassa RAML:n suosimisena viimeisimmissä tutkimuksissa ja kirjallisuudessa. Schreibmann ja Braun (2015, s. 6) mukaan RAML on ensimmäinen korkean tason kieli, jonka avulla kehittäjät voivat määritellä, luoda, testata ja julkaista REST-pohjaisia ohjelmointirajapintoja. RAML tarjoaa omaan sovellusaluekohtaiseen kieleen (Domain Specific Language, DSL) perustuvan formaalin mallin REST-pohjaisten ohjelmointirajapintojen määrittämiselle (Vu, Fertig ja Braun 2017, s. 341). Kirjoittajan mukaan RAML:n sovellusaluekohtainen kieli on suunniteltu kuvailemaan koko ohjelmointirajapinnan elinkaari ihmisluettavassa muodossa, käsittäen kuitenkin myös monia REST-pohjaisia määritelmiä, kuten URI:t, valtuutus, nimiavaruudet, mediatyypit ja HTTP-verbit. Schreibmann ja Braun (2015, s. 6) mukaan RAML käyttää YAML:ää merkintäkielenä ja pohjautuu ajatukselle resurssien ja niiden representaatioiden määrittämisestä JSON-skeemoina. Mutta

myös RAML:ssä on puutteita REST:n rajoitteiden mukaan. RAML:n mallin pohjalta voidaan kyllä luoda täysin toimiva REST-pohjainen ohjelmointirajapinta, mutta malli ei kuitenkaan tue hypermediaa, ja tästä syystä se rikkoo REST:n hypermediarajoitteita (Vu, Fertig ja Braun 2017, s. 341). Tällöin voidaan jopa sanoa, että itse ohjelmointirajapintaakaan ei voida pitää täysin REST-pohjaisena. RAML:llä on kuitenkin käyttötarkoituksensa, ja vaikka se ei ratkaise kaikkia REST:n ongelmia, on se esimerkkinä silti selkeä parannus täysin kuvauskielettömille toteutuksille ja irrallisille tekstidokumentaatioille.

5 Nykyiset ratkaisut

5.1 Hypermedia

Hypermediapohjaisella ohjelmointirajapinnalla on rajoitetut mahdollisuudet ilmaista sovelluksen palvelimella toteutettuja muutoksia rikkomatta sen asiakassovelluksia, mutta tämä ominaisuus ei ole automaattisesti käytössä (Richardson, Amundsen ja Ruby 2013, s. 189-190). Kirjoittaja käyttää esimerkkinä verkkosivujen sivukarttaa, joka on verkkosivun protokollasemantiikan täydellinen konekielinen kuvaus yhdessä HTML-dokumentissa, ja jonka pohjalta voidaan automaattisesti luoda ohjelmointirajapinnan asiakassovellus. Kirjoittaja kuitenkin jatkaa, että kun tällaisen ohjelmointirajapinnan toteutus muuttuu palvelimella, asiakassovellus rikkoutuu, koska se pohjautuu vanhentuneeseen sivukarttaan.

Monilla hypermediapohjaisilla ohjelmointirajapinnoilla on myös muita haasteita. Usein käytetään standardeja tai geneerisiä hypermediakieliä kuten HTML, joiden avulla voidaan toki määritellä ohjelmointirajapinnan protokollasemantiikka, mutta ei sovellussemantiikkaa, jonka seurauksena ohjelmointirajapinnan tarjoamien representaatioiden tarkoituksia ja selityksiä joudutaan pitämään erillisellä dokumentilla (Richardson, Amundsen ja Ruby 2013, s. 131-132). Kirjoittajan mukaan tämä taas johtaa siihen, että jokainen ohjelmointirajapinnan suunnittelija muodostaa sovellussemantiikan jo olemassa olevan palvelinpuolen mallin mukaan ja dokumentoi tuon semantiikan irrallisena.

Richardson, Amundsen ja Ruby (2013, s. 185) mainitsee kuitenkin myös hyvänä esimerkkinä hypermedian käytöstä verkkosivustot, joiden hyödyntämä hypermediadokumenttien muutosten vaikutus näkyy kaikilla asiakassovelluksilla jotka ovat niitä vastaanottaneet, ja tästä syystä verkkosivustot voivat uudistua täysin rikkomatta käyttäjiensä selaimia. Kirjoittajan mukaan verkkosivusto sisältyy täysin sen tarjoamaan representaatioon, eikä mitään ylimääräistä ole piilotettuna vain ihmisluettavaan dokumentaatioon. Kirjoittaja jatkaa, että siirtämällä ohjelmointirajapinnan semantiikka ihmisluettavasta dokumentaatiosta hypermediadokumentteihin saadaan ohjelmointirajapinnasta joustavampi muutoksia kohtaan, ja hyvän hypermediaformaatin avulla voidaan lisätä uusia resursseja ja tilasiirtymiä olemassa olevaan ohjelmointirajapintaan sekä muokata sen protokollasemantiikka vaikuttamatta sen olemassa

oleviin asiakassovelluksiin.

Hypermediassa on paljon muitakin hyviä puolia, ja se mahdollistaa myös joitain tekniikoita, jotka eivät tähän tutkimukseen läpikäymäni REST-painotteisen kirjallisuuden pohjalta ole keränneet kovin suurta kannattajakuntaa. Muun muassa Schreibmann ja Braun (2015, s. 8) mukaan hypermediapohjainen ohjelmointirajapinta voidaan määritellä äärellisen automaatin (Finite State Machine, NFA) mukaan niin, että sovelluksen toisiinsa yhteyksissä olevat resurssit ja HTTP-verbit edustavat automaatin tiloja, ja hyperlinkit edustavat siirtymiä tilojen välillä. Kirjoittaja jatkaa, että sovelluksen palvelin lähettää hyperlinkkejä osana vastausviestejä, ja näiden hyperlinkkien avulla asiakassovellukset voidaan saada käymään kaikki sovelluksen tilat läpi. Kirjoittajan mukaan äärellisten automaattien käyttö sovelluksen käyttäytymisen mallintamiseen vastaa yleisesti verkko- tai mobiilisovellusten mallintamista, joissa usein käytettäviä tekniikoita ovat rautalankamallit ja ruutusiirtymät.

Myös itse verkkoselaimet ja verkkosivustot ovat monella tavalla hyvä esimerkki hypermedian käytöstä. Richardson (2010, s. 4) mukaan verkkoselaimet perustuvat hypermediaan sovelluksen tilan moottorina (Hypermedia As The Engine Of Application State, HATEOAS). Kirjoittaja kuvailee verkkosivun toiminnan algoritmia REST-pohjaisella tavalla seuraavasti: ensimmäisenä haetaan hypermediarepresentaatio verkkosivun kotisivusta, tämän representaation tulkinnan perusteella määritellään resurssin tila, ja saman representaation semantiikan perusteella päätellään, mikä hypermedialinkki tai lomake voi mahdollisesti johtaa lähemmäksi haluttua tavoitetta, sekä klikataan haluttua linkkiä tai täytetään haluttu lomake, jonka seurauksena selain tekee uuden HTTP-pyynnön jonka vastauksena saadaan toinen hypermediarepresentaatio, ja toistetaan edeltävät askeleet kyseisellä representaatiolla, kunnes resurssin tila vastaa haluttua tavoitetta.

Hypermedian hyödyntäminen sovelluksen tilan moottorina on hyvin vahvasti liitoksissa sovelluksen tilasiirtymiin. Vu, Fertig ja Braun (2018, s. 1882) mukaan Jokaisella sovelluksen tilalla on oltava vähintään yksi sisään tuleva tilasiirtymä, jotta asiakassovellus voi saavuttaa sen, ja jokaisella sovelluksen tilalla on oltava vähintään yksi ulospäin suuntaava tilasiirtymä, jotta asiakassovellus ei jää jumiin. Hypermediaa käyttävän ohjelmointirajapinnan asiakassovelluksen ei kuitenkaan voida olettaa olevan tietoinen kaikista ohjelmointirajapinnan mahdollisista tilasiirtymistä etukäteen, vaan onkin järkevämpää, että hypermediakontrollit

jaetaan niin, että jokainen representaatio sisältää vain ne kontrollit joilla on merkitystä sen hetkiselle sovelluksen ja resurssien tilalle (Richardson, Amundsen ja Ruby 2013, s. 189-190). Kirjoittajan mukaan tämä pakottaa asiakassovellusten kehittäjät ottamaan hypermedia huomioon, eivätkä kehittäjät voi enää teeskennellä voivansa sivuuttaa hypermediaa.

Kehittäjien suunnalta on kuitenkin havaittavissa vastustusta hypermedialle sovelluksen tilan moottorina, ja yksi juurisyy tälle vastustukselle on se, että kehittäjät suosivat usein ennalta määritettyjä sääntöjä, joiden pohjalta voidaan rakentaa juuri sen resurssin URI, jota halutaan käsitellä (Richardson 2010, s. 4). Tämä johtaa helposti siihen, että määritellään valmiit tunnisteet jokaiselle resurssille dokumentaatioon, jolloin ollaan taas helposti siinä tilanteessa, että semantiikka on erillään itse sovelluksesta. Eikä se ole ainoa ongelma. Vu, Fertig ja Braun (2017, s. 342) mukaan hypermediapohjaisten asiakassovellusten pitäisi edelleen toimia ohjelmointirajapinnan palvelinpäivityksen jälkeen, eikä asiakassovelluksille täten pitäisi sallia minkäänlaista olettamusta ohjelmointirajapinnan resurssien tunnisteiden suhteen. Kirjoittajan mukaan tällaiset hypermediapohjaiset asiakassovellukset ovat kestävämpiä ja sopeutumiskykyisempiä palvelinpäivityksille, mikä vuorostaan vähentää versioinnin tarvetta. Tästä huolimatta Richardson (2010, s. 4) mukaan osa kehittäjistä jopa suosii resurssien metainformaation dokumentoimista ihmisluettavassa muodossa ja pitää konekielisiä hypermediakuvauksia tarpeettomina.

Hypermediaan ja semantiikan kuvaamiseen liittyy kuitenkin paljon sellaisia yksityiskohtia, joiden sivuuttamisen vaikutuksia ei monesti osata ottaa huomioon. Jos ohjelmointirajapinta ei hyödynnä hyperlinkkejä, johtaa tämä siihen, että sovelluksen tilat eivät myöskään luo seurattavia hyperlinkkejä ja asiakkaiden on pakko luoda nämä hyperlinkit pala palalta, joka vaatii aikaisempaa tietoa kaikista ohjelmointirajapinnan päätepisteistä, mikä paljastaa palvelimen toteutuksen yksityiskohdat (Vu, Fertig ja Braun 2018, s. 1882). Toisekseen kirjoittaja mainitsee hyperlinkkien puutteen johtavan siihen, että ohjelmointirajapinnalla ei ole ennalta määriteltyä sovelluksen toiminnankulkua, vaan se on pikemminkin staattinen ohjelmointirajapinta. Kolmanneksi kirjoittaja lisää, että edellä kuvattu asiakas-palvelin-arkkitehtuuri on tiukasti kytketty ja menee todennäköisesti rikki muutoksien johdosta joko palvelimella tai asiakkaalla.

5.2 REST ja hypertekstipohjainen navigointi

REST-pohjainen ohjelmointirajapinta voidaan suunnitella niin, että sen rakenne helpottaa hypertekstipohjaisen navigaation sekä siihen liittyvien mekanismien toimintaa ohjelmointirajapinnan rakennemuutoksien yhteydessä (Li ym. 2016, s. 154-155). Tämä tarkoittaa käytännössä laajennettavuutta. Kirjoittajan mukaan laajennettavuus REST-pohjaisissa ohjelmointirajapinnoissa tarkoittaa sitä, että ohjelmointirajapinta voi tarjota erilaisia funktioita samanaikaisesti ja se voi myös tehdä tiettyjä muutoksia näihin funktioihin ajan myötä rikkomatta asiakassovelluksiaan. Kirjoittaja jatkaa, että edellä mainittu joustavuus voidaan saavuttaa keskenään liitoksissa olevien resurssien hypertekstipohjaisen navigaation mekanismeilla.

On olemassa paljon tapauksia, joissa REST-pohjaiseen ohjelmointirajapintaan on tehtävä muutoksia tai päivityksiä nopeassa tahdissa, esimerkkinä monet avoimen lähdekoodin kehitysprojektit kuten OpenStack ja Docker (Li ym. 2016, s. 154-155). Kirjoittajan mukaan tämän seurauksena laajennettavuuden parantamisesta ja taaksepäin yhteensopivuuden säilyttämisestä on tullut akuutti ongelma etenkin tässä REST-pohjaisten ohjelmointirajapintojen nopeassa päivitystahdissa.

Hypertekstipohjainen navigaatio tarjoaa asiakassovelluksille tehokkaan tavan kestää tiettyjä REST-pohjaisen ohjelmointirajapinnan muutoksia automaattisesti (Li ym. 2016, s. 155). Kirjoittajan mukaan tämän helpottamiseksi ohjelmointirajapinta on suunniteltava REST:n periaatteiden mukaan, ja tämä suunnitelma on välitettävä käyttäjille täsmällisesti, jotta he voivat luoda hyvin suunniteltuja asiakassovelluksia. Kirjoittaja jatkaa, että hypertekstipohjaiseen navigaatioon liittyvä ylijäämä on myös huomioitava, jotta voidaan tasapainottaa REST-pohjaisen ohjelmointirajapinnan joustavuutta ja tehokkuutta.

Yllä mainitut REST:n periaatteet on määrittänyt REST:n kehittäjä Roy T. Fielding (ks. “REST APIs must be hypertext-driven” 2008), ja ne voivat olla hieman vaikeasti ymmärrettävässä muodossa. Tähän viittaa ainakin se, että jopa itse Roy T. Fielding mainitsee suurimman osan ohjelmointirajapintojen kehittäjistä sivuuttavan ainakin osan kyseisistä periaatteista (ks. “REST APIs must be hypertext-driven” 2008). Olenkin seuraavaksi pyrkinyt tulkitsemaan ja avaamaan näitä REST:n periaatteita.

Roy T. Fieldingin mukaan REST-pohjaisen ohjelmointirajapinnan ei pitäisi ensinnäkään olla

riippuvainen mistään yksittäisestä viestintäprotokollasta, ja minkä vain URI:tä tunnistautumiseen käyttävän protokollaelementin on yleisesti sallittava minkä vain URI-skeeman käyttö tunnistautumisen yhteydessä (ks. “REST APIs must be hypertext-driven” 2008). Näiden sääntöjen kohdalla epäonnistuminen antaa Fieldingin mukaan ymmärtää, että tunnistautuminen ei ole erillään itse viestinnästä. Hänen mukaansa REST-pohjaisen ohjelmointirajapinnan ei myöskään pitäisi tehdä muutoksia viestintäprotokollaan pieniä yksityiskohtia lukuun ottamatta. Tämän lisäksi REST-pohjaisen ohjelmointirajapinnan pitäisi hyödyntää suurinta osaa sen kuvauspotentiaalista määritelläkseen sen resursseja edustavat ja sovelluksen tilaa ohjaavat mediatyypit, tai määritelläkseen laajennettuja yhteysnimiä ja/tai hypertekstipohjaista merkintää olemassa oleville standardeille mediatyypeille. Kaikkien metodien ja tunnisteiden välisten suhteiden kuvausten tulisi olla täysin määritelty mediatyypin käsittelysääntöjen puitteissa. Puutteet tässä viittaavat Fieldingin mukaan siihen, että ns. out-of-band -informaatio ohjaa toimintaa hypertekstin sijaan. REST-pohjaisen ohjelmointirajapinnan ei myöskään pitäisi määritellä muuttumattomia resurssinimiä tai -hierarkioita, vaan palvelimella on oltava vapaus kontrolloida omaa nimiavaruuttaan sekä mahdollisuus ohjeistaa asiakasrajapintaa sopivien tunnisteiden luonnissa HTML-lomakkeiden sekä URI-kaavojen tapaan määrittelemällä ohjeet mediatyypissä ja linkkirelaatiossa. Tämän lisäksi REST-pohjaisen ohjelmointirajapinnan ei koskaan pitäisi käyttää tyypitettyjä resursseja, jotka ovat merkittäviä asiakassovelluksille, vaan ainoita asiakassovelluksille merkittäviä tyyppejä pitäisi olla nykyisen representaation mediatyyppi sekä standardoidut relaationimet. Viimeisimpänä Fielding mainitsee, että REST-pohjaista ohjelmointirajapintaa pitäisi pystyä käyttämään pelkän alustavan URI:n ja muutaman standardoidun sekä kohdeasiakkaiden ymmärrettävissä olevan mediatyypin turvin. Ensimmäisen kutsun jälkeen kaikkien sovelluksen tilasiirtymien olisi tapahduttava palvelimen tarjoamien vaihtoehtojen pohjalta, jotka ovat asiakkaan saatavilla kulloisenkin representaation kautta.

Yllä mainitut Roy T. Fieldingin “REST APIs must be hypertext-driven” (ks. 2008) määrittämät REST:n periaatteet voivat olla monille vaikeasti ymmärrettäviä ja ajoittain kankeita. Muitakin tapoja kuvata REST:tä on ajan saatossa luotu, ja yksi niistä on ns. Richardsonin Maturiteettimalli, jonka ovat kuvanneet “Richardson Maturity Model” (ks. 2010) ja “Act Three: The Maturity Heuristic” (ks. 2009). Heidän mukaansa Richardsonin Maturiteettimalli jakaa REST:n tärkeimmät elementit kolmeen tasoon, resursseihin (URI:t), HTTP-verbeihin ja hy-

permediakontrolleihin, ja suurin osa verkkopalveluista voidaan karkeasti jakaa sen mukaan, kuinka monta näistä tasoista verkkopalvelu toteuttaa, ja kuinka hyvin. Tämä maturiteettimalli on hyödyllinen, koska se on käytännön toteutus REST:n rajoitteista, ja koska on hankalaa puhua hypermediasta sovelluksen tilan moottorina, mutta on helpompaa puhua HTML:stä ja URI:stä, jotka ovat vastaava rajoite WWW:ssä (ks. “Act Three: The Maturity Heuristic” 2009).

“Richardson Maturity Model” (ks. 2010) ja “Act Three: The Maturity Heuristic” (ks. 2009) mukaan Richardsonin Maturiteettimalli määrittelee aluksi alimman tason, jolla operoivat palvelut hyödyntävät käytännössä yhtä URI:tä, yhtä HTTP-metodia eivätkä ollenkaan hypermediakontrolleja. Seuraavan tason, eli tason yksi verkkopalvelut erittelevät resurssit toisistaan käyttämällä erillistä URI:tä jokaista erillistä resurssia varten. Taso kaksi tuo tähän päälle tuen useammalle HTTP-metodille, mukaan lukien GET-metodi, joka mahdollistaa muun muassa turvalliset operaatiot sekä välimuistin käytön. Tällä tasolla tulee mukaan myös HTTP-paluukoodien käyttö. Kolmannella ja viimeisellä Richardsonin Maturiteettimallin tasolla tuodaan mukaan myös hypermediakontrollit. Tällä tarkoitetaan esimerkiksi resurssien mukana kulkevia linkkejä, jotka kertovat käyttäjälle, minkä tunnisteen kautta mikäkin resurssi on käsiteltävissä, HTML-lomakkeita, jotka kertovat, mitä millekin tietylle resurssille voidaan tehdä, tai mediatyyppiä, joka kertoo, miten linkit tai lomakkeet saadaan poimittua resurssin representaatiosta (ks. “Act Three: The Maturity Heuristic” 2009).

Richardsonin Maturiteettimalli ei kuitenkaan ole itse REST:n tasojen määritelmä, mutta sen avulla on mahdollisesti helpompi ymmärtää REST-pohjaista ajattelutapaa (ks. “Richardson Maturity Model” 2010). Kirjoittajan mukaan maturiteettimallin avulla päästään lähelle yleisiä suunnittelutekniikoita, koska mallin taso yksi jakaa laajan palvelun useammaksi resurssiksi, taso kaksi poistaa tarpeetonta vaihtelua käsittelemällä samankaltaiset tilanteet aina samalla tavalla HTTP-metodien avulla, ja taso kolme mahdollistaa protokollan itsedokumentoinnin. Mutta on kuitenkin muistettava, että Roy T. Fieldingin “REST APIs must be hypertext-driven” (ks. 2008) mukaan Richardsonin Maturiteettimallin tason kolme mukainen hypermedian käyttö on REST-pohjaisten ohjelmointirajapintojen ennakkoehto. Lisäksi huomion arvoista on, että Benedictis ym. (2015, s. 96) mukaan nykyhetken ohjelmointirajapinnoista vain hyvin pieni osa kunnioittaa kaikkia Richardsonin Maturiteettimallin taso-

ja. Kirjoittajan mukaan suurin osa ohjelmointirajapinnoista tukee kahta ensimmäistä tasoa, eli URI:n käyttöä resurssin tunnisteena sekä HTTP-metodien täyttä semantiikkaa resurssien käyttöä varten. Hypermediaa ja hypertekstipohjaista navigaatiota hyödyntävien ohjelmointirajapintojen määrä on paljon pienempi.

Hypertekstipohjaisen navigoinnin päätavoite on selviytyä ohjelmointirajapinnan muutoksista vähentämällä suunnitteluajan riippuvuutta asiakassovellusten ja ohjelmointirajapinnan välillä niin, että asiakas voi navigoida sen kohderesurssille hypertekstin sekä ohjelmointirajapinnalle tehtyjen kutsujen ohjaamana (Li ym. 2016, s. 157). Kirjoittajan mukaan hypertekstipohjainen navigointi toimii viiden REST-pohjaisen ohjelmointirajapinnan kerroksen päällä. Ensimmäinen näistä kerroksista on yhteys sellaisten resurssien välillä, joiden toteutusohjelmointikieli on vapaa ja joita voidaan ajaa vapaavalintaisilla laitteilla. Toinen kerros on yhteistoiminta eli metodit ja protokollat, joiden avulla voidaan olla yhteydessä resursseihin. Kolmas kerros on tunnistautuminen eli tunnisteet kuten URI:t joiden avulla resurssit tunnistetaan. Neljäs kerros on esitysmuoto tai representaatio eli hyperteksti kuten XML, joka liikkuu yhteistoiminnan kautta resurssien ja asiakassovelluksen välillä. Ja viides kerros on kuvaus eli kaikkien mahdollisten resurssien representaatioiden kuvailu. Li ym. (2016, s. 157) jatkaa edelleen, että asiakassovelluksen suunnitteluvaiheessa sen toteutus tehdään ohjelmointirajapinnan kuvausta vasten tietämättä miten sen resurssit ovat tunnistettavissa tai yhteyksissä toisiinsa, ja ohjelman ajon aikana asiakassovellus navigoi robottimaisesti alustavasta tunnisteesta kohderesurssille asti hypertekstipohjaisen representaation ohjaamana. Kirjoittajan mukaan tämän navigoinnin aikana asiakassovellus liikkuu yllä mainittujen tasojen neljännestä tasosta ensimmäiselle tasolle sykleissä, ja jokaisen syklin aikana asiakassovellus käyttää representaatioita selvittääkseen tunnisteet, tunnisteita selvittääkseen yhteistoimintaan vaadittavat metodit ja protokollat, ja yhteistoimintaa selvittääkseen yhteydet sekä representaatiot.

Hyperlinkkien seuraamisen lisäksi sisällön päättelyä sekä uudelleenohjausta voidaan myös hyödyntää hypertekstipohjaisen navigaation helpottamiseksi (Li ym. 2016, s. 157). Kirjoittajan mukaan sisällön päättelyssä resurssin tunniste, eli URI, ei jaa tietoa kyseisen resurssin representaation käyttämästä mediatyypistä, vaan ehdottaa yhteistoimintaa kuten HTTP:tä, joka voi päätellä oikean representaation, ja tämän ansiosta resurssin representaatio voi muuttua muuttamatta sen tunnistetta. Tätä kutsutaan hajautettujen resurssien löysäksi kytkennäk-

si. Kirjoittaja jatkaa, että HTTP-uudelleenohjauksessa URI:n ei tarvitse tunnistaa sen kohderesurssia, vaan se voi aloittaa yhteistoiminnan, joka voi päätellä oikean yhteyden kohderesurssiin, minkä ansiosta resurssi voi muuttaa yhteyksiään muuttamatta tunnisteitaan. Li ym. (2016, s. 157) mukaan URI-resoluution riippuessa hypertekstistä johon se sisältyy, on myös mahdollista muuttaa yhteistoimintaa resurssin kanssa muuttamatta resurssin tunnistetta, ja näin ollen voidaan todeta hypertekstipohjaisen navigaation mahdollistavan tietyt aiemmin mainittujen REST-pohjaisen ohjelmointirajapinnan kerroksien muokkaukset sillä rajoituksella, että kuvauskerros ne sallii.

5.3 HTML

HTML on Alkuperäinen hypermediaformaatti ja hyvin aliarvioitu valinta ohjelmointirajapinnoille, joka voi suoraan hyödyntää mikroformaatteja sekä mikrodataa, ja jonka script-tagin avulla voidaan suorittaa koodia asiakassovelluksella REST-pohjaisen arkkitehtuurin ominaisuuksien mukaisesti (Richardson, Amundsen ja Ruby 2013, s. 216). Kirjoittajan mukaan vastaavaa ominaisuutta ei tue mikään muu hypermediaformaatti, ja tämän lisäksi HTML-dokumentteja voidaan esittää graafisesti, mikä on tärkeä ominaisuus ongelmanselvitystä varten sekä yleisesti ohjelmointirajapinnoille, joita on suunniteltu kutsuttavan Ajax:n avulla. Myös Darmadi ym. (2018, s. 122) toteaa verkkoselainten HTML-linkkien ja lomakkeiden renderöintikyvykkyyden mahdollistavan sen, että hypermediapohjaisen ohjelmointirajapinnan määrittäminen HTML:n avulla antaa käyttäjille mahdollisuuden selailla ohjelmointirajapintaa selaimen kautta.

HTML:llä on kuitenkin myös selkeitä puutteita. Osa puutteista on osittain syynä sille, miksi HTML ei ole niin suosittu hypermediaformaatti ohjelmointirajapinnoissa, vaikka HTML onkin verkkosovellusten muissa käyttötarkoituksissa suuressa suosiossa. Teknisessä mielessä HTML ei ole geneerinen hypermediaformaatti vaan alakohtainen standardi, ja vaikka sen rajoitteita ei juuri huomata Internetissä, dataformaattina sen rajat tulevat nopeasti vastaan suunniteltaessa HTML:ää käyttävää ohjelmointirajapintaa (Richardson, Amundsen ja Ruby 2013, s. 124-125). Kirjoittajan mukaan HTML:n hypermediakontrolleilla ei voida kuvata koko HTTP:n protokollasemantiikkaa, sillä HTML-asiakas ei muun muassa pysty tekemään PUT tai DELETE -kutsuja käyttämättä JavaScriptiä.

Olisi ollut hyvin mahdollista, että HTML 5 olisi tuonut mukanaan muutoksia HTML:n puutteisiin ohjelmointirajapintojen dataformaattina. Muutokset jäivät kuitenkin hyvin vähäisiksi. HTML 5 toi mukanaan muutaman uuden hypermediakontrollin, joiden avulla voidaan luoda upotettuja linkkejä, muun muassa audio, video, source ja embed -elementit (Richardson, Amundsen ja Ruby 2013, s. 124-125). Kirjoittajan mukaan mikään niistä ei ole kovinkaan hyödyllinen muissa kuin multimediaohjelmointirajapinnoissa, eikä HTML 5 täten juurikaan muuta HTML:ää hypermediaformaattina.

5.4 HAL

HTML:llä on vahvuutensa, mutta siinä on myös monia puutteita, eikä se ratkaise kaikkia ongelmia, joita ohjelmointirajapintojen kehittäjät kohtaavat. HTML on vanha ja suunniteltu ihmisluettavia dokumentteja varten, mutta se on silti toiminut pohjana monelle uudelle hypermediaformaatille jotka on luotu erityisesti ohjelmointirajapintoja varten (Richardson, Amundsen ja Ruby 2013, s. 125). Kirjoittajan mukaan yksi näistä on Hypertext Application Language (HAL), joka on hyvä esimerkki yleisestä hypermediakielestä, jolla ei ole HTML:n historiallista taakkaa, sillä HAL hyödyntää HTML:n ominaisuuksista vain hyperlinkkiä ja karsii niistä pois kaikki muut. HAL on siis yksinkertaisempi ja helposti lähestyttävämpi ratkaisu ohjelmointirajapintojen hypermediarajoitteille, joka kuitenkin minimalistisesta rakenteestaan huolimatta tarjoaa paljon tarvittavia työkaluja ohjelmointirajapintojen toteuttamiseen. Martins, Mazayev ja Correia (2017, s. 20059) mukaan HAL on luotu helppokäyttöiseksi tunnisteiden ja niiden keskinäisten yhteyksien kuvaamistyökaluksi hyödyntäen vain pientä hypermediakontrollien osajoukkoa, eli linkkejä. Ja tällä hypermediakontrollilla voidaan tehdä esimerkiksi GET, POST, PUT, DELETE tai UNLINK -kutsuja, käytännössä siis esimerkiksi HAL+XML -dokumentin link-tagin avulla voidaan tehdä kaikkia HTTP-pyyntöjä (Richardson, Amundsen ja Ruby 2013, s. 126). HAL ei kuitenkaan ole puutteeton. Vaikka HAL:n linkkirelaation avulla voidaan tehdä mitä tilasiirtymiä tahansa, näitä tilasiirtymiä voidaan kuitenkin kuvata vain ihmisluettavan dokumentaation avulla, mikä ei ole hyvä yhdistelmä (Richardson, Amundsen ja Ruby 2013, s. 128). Kirjoittajan mukaan HAL toimisi kuitenkin hyvin vain lukuoperaatioita tekeville ohjelmointirajapinnoille, joiden kaikki tilasiirtymät ovat turvallisia.

5.5 JSON-LD

JSON-LD (JavaScript Object Notation for Linked Data) on profiilikieli, jonka avulla voidaan yhdistää konekielinen dokumentti (ns. konteksti) tavallisen JSON-dokumentin kanssa (Richardson, Amundsen ja Ruby 2013, s. 150-151). Kirjoittajan mukaan näin on helppo määritellä profiili jo olemassa olevalle ohjelmointirajapinnalle muuttamatta dokumentin formaattia ja täten rikkomatta olemassa olevia asiakassovelluksia. Toisin sanottuna JSON-LD:n avulla voidaan kuvata JSON-pohjaisen ohjelmointirajapinnan semantiikkaa ilman erillistä dokumenttia. Richardson, Amundsen ja Ruby (2013, s. 156) mukaan JSON-LD kontekstin avulla voidaan kuvata JSON-pohjaisen ohjelmointirajapinnan sovellus- ja protokollasemantiikka konekielisesti, sekä lisätä ohjelmointirajapinnalle yksinkertaisia hypermediakontrolleja edelleen kuitenkin rikkomatta olemassa olevia asiakassovelluksia.

JSON-LD siis käytännössä korjaa kaikki tässä tutkimuksessa havaitut JSON-kohtaiset haasteet ja ongelmat. Vaikka JSON ei ole hypermediaformaatti, siihen voidaan lisätä hypermediakyvykkyyksiä JSON-LD:n avulla. Ja vaikka JSON ei tue mikroformaatteja tai mikrodataa, eikä täten ole täysin konekielinen kuvaus, voidaan JSON-dokumenttia vastaava konekielinen kuvaus kuitenkin yhdistää sen kanssa JSON-LD:n avulla. JSON-kohtaisina haasteina havaittiin myös sovellussemantiikkojen kertakäyttöisyys ja erillisyys itse sovelluksesta. Mutta koska JSON-LD pystyy kuvaamaan sovellussemantiikan konekielisenä ja yhdistämään tämän kuvauksen itse JSON-dokumentin kanssa, ei sovellussemantiikka enää ole erillään eikä kertakäyttöinen. JSON sekä JSON-LD yhdessä ratkaisevat siis monia REST-pohjaisten ohjelmointirajapintojen formaatti- ja mediatyyppikohtaisia ongelmia sekä haasteita.

5.6 AtomPub

AtomPub (Atom Publishing Protocol) on sovellustason protokolla verkkoresurssien julkistamista ja muokkaamista varten (Steiner ja Algermissen 2011, s. 12). AtomPub toimii Atom-formaatin päällä, joka Parastatidis ym. (2010, s. 17) mukaan käyttää hypermediakontrolleja yhdistääkseen listoja informaatiota, jotka puolestaan viittaavat muihin resursseihin. Kirjoittajan mukaan Atomia käyttämällä voidaan luoda hajautettuja, yhdistettyjä ja duplikaateista siivottuja aineistoja kokoamalla sekä navigoimalla Atom-syötteitä verkossa. Atom-formaatti

määritellään mediatyypillä application/atom+xml, ja Atom itse määrittää sekä skeeman että kontekstin jossa linkkien yhteydet voidaan ymmärtää (Davis 2012, s. 8). Kirjoittajan mukaan Atom-formaatti määriteltiin huolellisesti juuri sellaista laajennosta varten, jonka AtomPub toteuttaa. AtomPub on klassinen esimerkki protokollasta, joka perustuu vakiintuneeseen mediatyyppiin (Parastatidis ym. 2010, s. 18-19). Kirjoittajan mukaan AtomPub laajentaa Atom-formaatin linkkirelaatioita luodakseen uuden käsittelymallin Atom-syötteitä varten.

AtomPub implementoi ensimmäisten joukossa kokoelmamallin ja yleisesti koko REST-pohjaisen lähestymistavan ohjelmointirajapintoihin, ja vaikka se onkin XML-pohjaisena standardina JSON-pohjaisten kuvausten dominoimalla alalla hieman vanhanaikainen, on AtomPub toiminut inspiraationa useille muille standardeille sekä linkkirelaatiolle joita toiset hypermediaformaatit voivat hyödyntää (Richardson, Amundsen ja Ruby 2013, s. 208). Kirjoittaja jatkaa, että vaikka AtomPubin sovellussemantiikka vihjaakin standardin olevan tarkoitettu uutisvirtasovelluksia kuten bloggaus- tai sisällönhallintaohjelmointirajapintoja varten, on AtomPub standardina kuitenkin helposti laajennettavissa. Kirjoittajan mukaan yksi AtomPubin merkittävimpiä laajennoksia on Google Data Protocol, jonka päällä Googlen ohjelmointirajapintaalusta toimii, ja Google voi aluekohtaisia tageja lisäämällä kuvailla jokaisen sivustonsa sovellussemantiikan, jolloin AtomPubin virta voi muuttua esimerkiksi video- tai laskutaulukkokokoelmaksi. Hieman tarkemmin kuvailtuna Steiner ja Algermissen (2011, s. 12) mukaan Google Data Protocol laajentaa AtomPub-protokollaa kyselyjen käsittelyllä, tunnistautumisella, ja eräpyynnöillä.

AtomPub ei kuitenkaan ratkaise kaikkia rajapintojen ongelmia. Richardson, Amundsen ja Ruby (2013, s. 131) mukaan suurin osa hypermediapohjaisista ohjelmointirajapinnoista hyödyntävät kokoelmastandardeja kuten AtomPub tai geneerisiä hypermediakieliä kuten HTML määrittääkseen ohjelmointirajapinnan protokollasemantiikan, mutta eivät kuitenkaan juurikaan määrittele sovellussemantiikkaansa.

5.7 Testaus

REST-pohjaisten ohjelmointirajapintojen testaus on käytännössä puuttuva aihe kirjallisuudessa, ja etenkin hypermedian testausta ei mainita ollenkaan (Vu, Fertig ja Braun 2018,

s. 1881). Tämän kirjallisuuskatsauksen pohjalta voin myös itse todeta, että ohjelmointirajapintojen testaukseen liittyvää kirjallisuutta ja tutkimusta löytyy hyvin vähän. Tämä herättääkin hieman kysymyksiä siitä, olisiko aiheellista tutkia testausta enemmän, etenkin kun REST-pohjaista ja ohjelmointirajapintoihin liittyvää kirjallisuutta ja tutkimusta on muuten tehty monesta eri näkökulmasta, mikä jo itsessään viittaa ohjelmointirajapintojen sekä etenkin REST-pohjaisten rajapintojen suosioon. Myös Fertig ja Braun (2015, s. 1497) mukaan REST-pohjaisten palveluiden laadunvarmistuksen informaatiosta on puute, ja tämä voi johtaa kehittäjiä harhaan ja esimerkiksi laskemaan laadunvarmistuksen tasoa kehityksen aikaisen tiukan aikataulun takia, joka voi aiheuttaa sivuvaikutuksia kaikille kehitettävän ohjelmointirajapinnan asiakassovelluksille.

REST-pohjaisten ohjelmointirajapintojen testausta ei kuitenkaan ole täysin sivuutettu. Vu, Fertig ja Braun (2018, s. 1881) mukaan yleinen ja hyvin yksinkertainen käytäntö REST-pohjaisten ohjelmointirajapintojen testauksessa on HTTP-kutsujen lähettäminen ja niihin saatujen vastauksien varmentaminen. Tämä ei kuitenkaan ole erityisen kattava testauskäytäntö, sillä ainakin hypermedian hyödyntäminen jää tällöin täysin testauksen ulkopuolelle, eikä täten voida täysin todentaa ohjelmointirajapinnan REST-pohjaisuutta. Myöskään yksikkötestaukseen tähtäävät ja yleisesti käytössä olevat ohjelmistorungot kuten JUnit tai NUnit eivät täysin sovellu verkkopalveluiden testaukseen, sillä ne ovat puolestaan tiukasti kytkeytyneitä testattavan sovelluksen toteutuskieleen (Fertig ja Braun 2015, s. 1498).

Paremminkin soveltuvia vaihtoehtoja kuitenkin on. Vu, Fertig ja Braun (2018, s. 1881) esittävätkin REST-pohjaisten ohjelmointirajapintojen testausvaihtoehdoksi mallipohjaiseen ohjelmistokehitykseen (Model-Driven Software Development, MDSD) perustuvaa mallipohjaista testausta (Model-Driven Testing, MDT), jossa ideana on REST-pohjaisen ohjelmointirajapinnan mallin pohjalta vahvistaa ohjelmointirajapinnan oikeellisuus. Heidän testauksensa pohjautuu ajatukseen siitä, että hypertekstipohjainen REST-rajapinta suunnitellaan äärelliseksi automaatiksi, ja siitä luodun metamallin kautta voidaan testata, että jokaisella suunnitellun rajapinnan tilalla on vähintään yksi ulospäin ja vähintään yksi sisäänpäin ohjautuva tilasiirtymä. Myös RAML ajaa mallipohjaisen ohjelmistokehityksen ideaa, ja sen mallin pohjalta voidaankin sekä luoda että testata täysin toimivia REST-pohjaisia ohjelmointirajapintoja (Vu, Fertig ja Braun 2017, s. 341).

Sen sijaan että käytettäisiin Turing-täydellistä ohjelmointikieltä lausekkeineen, ehtoineen ja silmukoineen, mallipohjainen ohjelmistokehitys mahdollistaa kehittäjien korkeamman abstraktiotason ajattelutavan (Schreibmann ja Braun 2015, s. 6). Kirjoittajan mukaan mallipohjaisessa ohjelmistokehityksessä luodaan ohjelmistogeneraattorin avulla sovellusaluekohtaiseen kieleen pohjautuvasta mallista käyttöön otettava REST-pohjainen rajapinta. Kirjoittaja jatkaa, että tämä mallipohjainen lähestymistapa vähentää kompleksisuutta REST-pohjaisen rajapinnan kehitysvaiheessa, parantaa laadunvarmistusta luomalla automatisoituja testitapauksia, ja yksinkertaistaa rajapinnan käyttöä luomalla kirjastoja, joiden avulla rajapintaa voidaan käyttää erityyppisten asiakkaiden kautta.

Mallipohjaisella testauksella on paljon muitakin hyviä puolia. Vu, Fertig ja Braun (2017, s. 340) mukaan manuaalinen hypermedian testaus on aikaa vievää ja haastavaa ylläpitää. Kirjoittaja jatkaa, että hypermediapohjaisten ohjelmointirajapintojen testaus vaatii monia rakenteellisesti samankaltaisia testitapauksia etenkin, kun halutaan huomioida myös käyttäjien roolit sekä virhetapaukset, ja mallipohjainen testaus soveltuu tähän erittäin hyvin. Samankaltaisuuksistaan huolimatta näkisin, että testitapaukset voidaan mahdollisesti jaotella tarkemmin käyttötarkoituksien mukaan. Fertig ja Braun (2015, s. 1498) mukaan tarpeelliset REST-pohjaisten ohjelmointirajapintojen testitapaukset voidaankin jakaa funktionaalisiin testeihin, turvallisuustesteihin, suorituskykytesteihin, käyttäytymistesteihin ja REST:n rajoitteiden noudattamiseen. REST:n rajoitteita lukuun ottamatta tämä luokittelu ei juurikaan vaikuta erityisesti REST-pohjaisiin ohjelmointirajapintoihin painottuneelta, vaan pikemminkin hyvin yleiskäyttöiseltä. Tämä herättääkin lisää kysymyksiä siitä, miksi REST-pohjaisten ohjelmointirajapintojen testaus on kirjallisuudessa jäänyt niin huomiotta. Voi olla, että REST:n testaaminen on nähty yleiskäyttöisillä työkaluilla toteutettavana tehtävänä, joka ei syvällisempää perehtymistä tai erikoistumista vaadi. Todellisuudessa REST-pohjaisten ohjelmointirajapintojen testaus on kuitenkin melko yksityiskohtaista, ja esimerkiksi funktionaalinen testaus vaatii jo vähintään osittaista kokemusta REST-pohjaisista rajapinnoista. Fertig ja Braun (2015, s. 1498) listaa funktionaalisen testauksen tärkeimpinä osina virheettömän vastausrungon, joka sisältää haetun resurssin representaation, pyydetyn mediatyypin käytön tai vähintään validin HTTP-virheviestin, paluuviestin validit HTTP-otsikkorivit, sekä palautetun URI:n oikeellisuuden siten, että jokaisen URI:n on osoitettava olemassa olevaan resurssiin, koska yksikin rikkinäinen linkki voi hajottaa hypermediamekanismin. REST-pohjaisten oh-

jelmointirajapintojen periaatteisiin ja käytänteisiin tutustuttuani yllä listatut tärkeimmät osat vaikuttavat yksiselitteisiltä ja osittain jopa itsestäänselvyyksiltä. Teorian ja tavoitteiden tasolla asia vaikuttaa hyvin selkeältä. Mutta toteutuksen tasolla tiedon leviämisessä on havaittavissa puutteita.

Testauksen pitäisi olla yksi tärkeimmistä ohjelmistokehityksen osa-alueista, ja tästä syystä informaation puute onkin jossain määrin erikoista, ja voi johtaa vakaviinkin ongelmiin. Myös Fertig ja Braun (2015, s. 1502) mukaan on oikeastaan olemassa kaksi syytä miksi REST-pohjaisten palveluiden testaus voi epäonnistua, ja niistä ensimmäinen on testausinformaation puute sekä kirjallisuudessa että verkossa. Toisena syynä kirjoittaja mainitsee projektien tiukat aikataulut, joiden seurauksena projektipäälliköt yleensä tinkivät laadunvarmistuksesta toiminnallisuuksien sijaan. Itse näkisin asian niin, että syyt eivät myöskään sulje toisiaan pois, vaan pikemminkin voi olla, että aikatauluongelmat ovat ainakin osittain saaneet alkunsa juurikin informaation puutteesta. Testauksen roolin tärkeys REST-pohjaisia ohjelmointirajapintoja käsittelevässä tutkimuksessa ja kirjallisuudessa vaikuttaakin täten yhä suuremmalta. Onkin hyvä, että tähän tarpeeseen on vähitellen herätty.

6 Yhteenveto

Ohjelmointirajapintoja on hyödynnetty ohjelmistokehityksessä jo pidemmän aikaa. Viime vuosien aikana kuitenkin etenkin web-ohjelmointirajapintojen suosio on ollut kovassa nousussa. Ja näistä web-ohjelmointirajapinnoista etenkin REST-pohjaiset ohjelmointirajapinnat ovat keränneet paljon huomiota ja saaneet vähitellen valta-aseman ohjelmointirajapintojen yleisimpänä toteutusvaihtoehtona. Samalla myös ohjelmointirajapintojen käyttötarkoitus on muuttunut. Ohjelmointirajapinnat, joita käytettiin ennen vain sisäisten integraatio-ongelmien ratkaisuina ovatkin muuttuneet yhdeksi organisaatioiden digitaalisten strategioiden kulmakivistä (Higginbotham 2015, s. 1). Monet yritykset tarjoavat esimerkiksi sisältöä ja erinäisiä toiminnallisuuksia halukkaiden verkkosivustoille ja -sovelluksille ohjelmointirajapintojen avulla. Tämän seurauksena myös verkkosovellusten kehityssuunta on muuttunut staattisemmasta mallista dynaamisempaan muotoon, ja yhteistoiminta sekä toiminnallisuuksien jakaminen ovat lisääntyneet huomattavasti verkkosovelluskehityksessä.

Toteutusvaihtoehtoja web-ohjelmointirajapinnalle on toki useita, mutta REST on kaikista vaihtoehdoista suosituin. Itse-asiassa suosituin REST-pohjainen toteutus on HTTP:n päällä toimiva WWW (Liskin, Singer ja Schneider 2011, s. 3). Jo tämä itsessään kertoo paljon REST:n vahvuuksista verrattuna muihin rajapintojen toteutusvaihtoehtoihin. REST on kevyt, työskentely sen kanssa on suhteellisesti helpompaa, ja oikein toteutettuna REST-pohjainen ohjelmointirajapinta on monipuolisempi formaattien sekä laajennettavuuden suhteen. Tämän tutkimuksen aiheena ei kuitenkaan ollut REST:n vahvuudet, vaan pikemminkin REST-pohjaisten toteutusten haasteet, ongelmat ja heikkoudet, sekä niille mahdollisesti jo kehitetyt ratkaisut.

REST-pohjaisten ohjelmointirajapintojen suosiosta huolimatta niiden toteutuksen aikaisia haasteita on edelleen useita. Monet niistä pohjautuvat REST:n pohjimmaiseen olemukseen. Ikävä kyllä, REST tarkoittaa eri asioita eri ihmisille, sillä jotkut sanovat REST:tä standardiksi, toiset määrittelyksi, ja toisille REST on arkkitehtuurityyli (Vu, Fertig ja Braun 2018, s. 1882). Virallisestihan REST on arkkitehtuurityyli, ja juuri tästä syystä hyvin tulkinnanvarainen. Kehittäjien on prosessoitava paljon suosituksia, eivätkä REST:n alkuperäisen kehittäjän viralliset periaatteet ole kokemattomalle tai edes kokeneemmalle kehittäjälle kovinkaan

selkeitä. Tämä näkyy esimerkiksi haasteina versioinnissa ja ohjelmointirajapinnan päivityksessä, sillä mikäli jotkin REST:n periaatteista jäävät ohjelmointirajapinnan kehitysvaiheessa huomioimatta, ovat ne yleensä ensimmäisenä hypermedia ja hypermediarajoitteet. Ja kun ohjelmointirajapintaa ei toteuteta hypermediapohjaisena, muutosten tekeminen yleensä katkaisee sen asiakassovellusten toiminnan.

Hypermedia vaikuttaa myös muihin REST-pohjaisen ohjelmointirajapinnan osa-alueisiin, muun muassa tiedonvaihtoformaattiin. JSON on tällä hetkellä yksi suosituimmista ohjelmointirajapintojen käyttämistä formaateista, vaikka sillä ei ole minkäänlaisia hypermediavalmiuksia. Käytännössä tämä tarkoittaa sitä, että JSON-formaattia käyttävien REST-pohjaisten ohjelmointirajapintojen toimintojen tarkoitus on dokumentoitava erilliselle dokumentille, jota on ylläpidettävä rajapinnan ylläpidon rinnalla. Vastaavat irralliset sovellussemantiikat eivät ole uudelleenkäytettävissä, ja mikäli rajapinnan sovellussemantiikka päivittyy, eivät sen pohjalta luodut asiakassovellukset päivity automaattisesti.

Vastaavia ongelmia on myös muilla formaateilla. XML on yleisesti raskaampi, ja vaikka HTML:llä onkin hypermediavalmiuksia, ei senkään avulla silti pystytä täysin kuvaamaan REST-pohjaisen ohjelmointirajapinnan toimintojen tarkoitusta tai tiedonsiirtoprotokollaa eli HTTP:tä. HTML:ää ei myöskään koskaan luotu ohjelmointirajapintojen käyttötarkoitukseen, ja siinä on paljon ylimääräistä. HAL on käytännössä HTML:n ohjelmointirajapintoja varten luotu kevyempi versio, mutta sekään ei täysin tue sovellussemantiikan yhdistämistä representaatiodokumenttiin, kuten ei myöskään Atom-formaattia käyttävä AtomPub protokolla.

REST-pohjaisten ohjelmointirajapintojen kehityksessä on kuitenkin muitakin haasteita. Ohjelmistorungon puute vaikeuttaa kehitystä, yleisesti kattavia työkaluja rajapintojen suunnitteluun, dokumentointiin ja testaukseen ei juuri ole, ja standardoitua konekielistä kuvauskieltä ei ole, mikä johtaa rajapinnan kuvauksen luomiseen erilliseen ihmisluettavaan dokumenttiin. Monia kuvauskieliä REST:tä varten on kyllä ehdotettu, mutta mitään niistä ei kuitenkaan ole standardoitu, niissä on monesti rajoitteita, eivätkä useimmat niistä tue hypermediaa. Yleisesti voidaankin sanoa, että REST-pohjaisten ohjelmointirajapintojen kehityksen aikaisten haasteiden pohjimmaisena syynä on yleensä REST:n periaatteiden sivuuttaminen.

Suurin osa tämän tutkimuksen aikana havaituista ja edellä mainituista haasteista voidaan pe-

riaatteessa ratkaista toteuttamalla REST-pohjaiset ohjelmointirajapinnat aina REST:n periaatteiden mukaisesti. Käytännössä ratkaisu ei kuitenkaan ole niin yksinkertainen, ja osa haasteista syntyy juurikin REST:n periaatteiden monimutkaisuudesta. Selkeästi hankalin ja eniten vastustusta synnyttävä REST:n periaate on hypermedia. Mutta vaikka rajapinnan toteuttaminen hypermediapohjaisena vaikuttaakin haastavalta, ovat hypermediasta saatavat hyödyt hyvin selkeitä. Hypermedian avulla ohjelmointirajapinnasta voidaan tehdä kestävämpi ja alttiimpi muutoksille. Hypermedia mahdollistaa myös sen, että ohjelmointirajapinnan toteutuksen yksityiskohtia palvelimella ei tarvitse paljastaa asiakassovelluksille, ja näin yhteys asiakassovelluksen ja palvelimen välillä pidetään mahdollisimman löysänä sekä ohjelmointirajapinta dynaamisena.

Hypermedian lisäksi REST-pohjaisten ohjelmointirajapintojen tärkeimpiä osa-alueita ovat tämän tutkimuksen pohjalta resurssit ja niiden tunnisteet, HTTP-protokolla, mediatyypit sekä kuvaukset. Näistä resurssit, HTTP ja hypermedia ovat mahdollisesti yksi selkeimmistä tavoista jakaa ohjelmointirajapinnat sen mukaan, kuinka kattavasti niiden toteutus on tehty REST:n periaatteiden mukaan. Useimmat ohjelmointirajapinnat hyödyntävät resursseja jakamaan ohjelmointirajapintaa toiminnallisuuksien mukaan, sekä mahdollistavat erinäiset operaatiot resurssien kanssa HTTP-metodien avulla. Resurssien ja tuettujen HTTP-metodien määrä kuitenkin vaihtelee, ja käytännössä mitä laajemmin ohjelmointirajapinnan toiminnallisuudet on jaettu erillisiin resursseihin ja mitä useampi HTTP-metodi on tuettuna, sitä lähemmin ohjelmointirajapinta on toteutettu REST:n periaatteiden mukaan. Suurin osa nykyisistä ohjelmointirajapinnoista on tällä tasolla. Seuraavalla tasolla ohjelmointirajapinnat hyödyntävät resurssien ja HTTP:n lisäksi hypermediaa, ja tällä tasolla ohjelmointirajapinta on itse asiassa vasta täysin REST-pohjainen. Tällä hetkellä kuitenkin vain hyvin harva ohjelmointirajapinta pääsee tälle tasolle.

Omalla tavallaan yksi tapa ratkaista REST-pohjaisten ohjelmointirajapintojen haasteita on testaus. Ohjelmointirajapintojen testauksesta ei kuitenkaan löydy kovinkaan paljon tutkimusta ja kirjallisuutta, vaikka ohjelmointirajapintojen käyttö onkin vakaassa kasvussa. Sen informaation pohjalta, mitä REST-pohjaisten ohjelmointirajapintojen testaukseen liittyen löytyi, voidaan testauksesta löytää sekä omat haasteensa että myös selkeät etunsa. Muun muassa informaation puutteesta johtuen ohjelmointirajapintojen testaus jää usein vajavaiseksi, ei-

kä kata kaikkia ohjelmointirajapinnan ominaisuuksia. REST:n periaatteet ja etenkin hypermedia jäävät usein helposti testauksen ulkopuolelle. Kattavampiakin testaustekniikoita kuitenkin on, ja yksi niistä on mallipohjaiseen ohjelmistokehitykseen perustuva mallipohjainen testaus, jossa perusajatuksena on toteuttaa testaus ohjelmointirajapinnasta luodun mallin pohjalta. Tämä mahdollistaa melko kattavan REST:n periaatteiden ja etenkin hypermediarajoitteen testauksen, sekä yleisesti testitapausten automatisoinnin. Lopputuloksena testaus ja laadunvarmistus ovat kuitenkin ensimmäisenä kärsimässä muun muassa projektien tiukoista aikatauluista, sillä niistä lähdetään useimmiten ensimmäisenä karsimaan, kun kohdataan aikataulullisia haasteita.

REST-pohjaisten ohjelmointirajapintojen testauksessa, kehityksessä ja ylläpidossa on siis selkeästi havaittavissa haasteita. Voidaan kuitenkin sanoa, että mikään niistä ei ole ylitsepääsemätön nykyaikaisten ratkaisujen avustuksella. Monet ratkaisut tuovat toki mukanaan myös omia haasteitaan, eikä yksittäinen ratkaisu välttämättä riitä kattamaan kaikkia haasteita, mutta sopivasti teknologioita, kehityssuuntauksia ja -työkaluja yhdistelemällä päästään melko lähelle kelvollista ratkaisua. Tämän tutkimuksen pohjalta voidaan todeta, että ratkaisut REST-pohjaisten ohjelmointirajapintojen ongelmiin ja haasteisiin ovat jo olemassa ja saatavilla. Suurin haaste onkin informaation jakaminen, REST:n periaatteiden selkeyttäminen ja niiden mukaan toimiminen.

Lähteet

"Act Three: The Maturity Heuristic". 2009. Viitattu 18. helmikuuta 2019. https://www.crummy.com/writing/speaking/2008-QCon/act3.html.

Benedictis, Alessandra De, Massimiliano Rak, Mauro Turtur ja Umberto Villano. 2015. "REST-based SLA Management for Cloud Applications". *2015 IEEE 24th International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises:* 93–98. doi:10.1109/WETICE.2015.36.

Cretella, Giuseppina, ja Beniamino Di Martino. 2012. "Semantic Web Annotation and Representation of Cloud APIs". *2012 Third International Conference on Emerging Intelligent Data and Web Technologies:* 31–37. doi:10.1109/EIDWT.2012.61.

Darmadi, Herru, Suryadiputra Liawatimena, Bahtiar Saleh Abbas ja Agung Trisetyarso. 2018. "Hypermedia Driven Application Programming Interface for Learning Object Management". *3rd International Conference on Computer Science and Computational Intelligence 2018:* 120–127. doi:10.1016/j.procs.2018.08.157.

Davis, Cornelia. 2012. "What if the Web Were not RESTful?" *WS-REST '12 Proceedings of the Third International Workshop on RESTful Design:* 3–10. doi:10.1145/2307819.2307823.

Fertig, Tobias, ja Peter Braun. 2015. "Model-driven Testing of RESTful APIs". *Proceedings of the 24th International Conference on World Wide Web:* 1497–1502. doi:10.1145/2740908.2743045.

Hadley, Marc, Santiago Pericas-Geertsen ja Paul Sandoz. 2010. "Exploring Hypermedia Support in Jersey". *WS-REST '10 Proceedings of the First International Workshop on RESTful Design:* 10–15. doi:10.1145/1798354.1798378.

Higginbotham, James. 2015. *Designing Great Web APIs.* 1. painos. Sebastopol, CA: O'Reilly Media.

Kankanamge, Charitha. 2012. *Web Services Testing with soapUI.* Birmingham, UK: Packt Publishing Ltd.

Li, Li, Wu Chou, Wei Zhou ja Min Luo. 2016. “Design Patterns and Extensibility of REST API for Networking Applications”. *IEEE Transactions on Network and Service Management* 13 (1): 154–167. doi:10.1109/TNSM.2016.2516946.

Liskin, Olga, Leif Singer ja Kurt Schneider. 2011. “Teaching old services new tricks: adding HATEOAS support as an afterthought”. *WS-REST ’11 Proceedings of the Second International Workshop on RESTful Design:* 3–10. doi:10.1145/1967428.1967432.

Makkonen, Joni. 2017. *Performance and usage comparison between REST and SOAP web services.* Aalto-yliopisto perustieteiden korkeakoulu.

Martins, Jaime A., Andriy Mazayev ja Noélia Correia. 2017. “Hypermedia APIs for the Web of Things”. *IEEE Access: Special Section on Intelligent Systems for the Internet of Things* 5:20058–20067. doi:10.1109/ACCESS.2017.2755259.

Parastatidis, Savas, Jim Webber, Guilherme Silveira ja Ian S Robinson. 2010. “The Role of Hypermedia in Distributed System Development”. *WS-REST ’10 Proceedings of the First International Workshop on RESTful Design:* 16–22. doi:10.1145/1798354.1798379.

“REST APIs must be hypertext-driven”. 2008. Viitattu 18. helmikuuta 2019. http://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven.

“Richardson Maturity Model”. 2010. Viitattu 18. helmikuuta 2019. https://martinfowler.com/articles/richardsonMaturityModel.html.

Richardson, Leonard. 2010. “Developers Like Hypermedia, But They Don’t Like Web Browsers”. *WS-REST ’10 Proceedings of the First International Workshop on RESTful Design:* 4–9.

Richardson, Leonard, Mike Amundsen ja Sam Ruby. 2013. *RESTful Web APIs.* 1. painos. Sebastopol, CA: O’Reilly Media.

Schreibmann, Vitaliy, ja Peter Braun. 2015. “Model-driven Development of RESTful APIs”. *Proceedings of the 11th International Conference on Web Information Systems and Technologies (WEBIST-2015):* 5–14. doi:10.5220/0005411200050014.

Sill, Alan. 2015. “When to Use Standards-Based APIs (Part 2)”. *IEEE Cloud Computing:* 80–84.

Steiner, Thomas, ja Jan Algermissen. 2011. “Fulfilling the Hypermedia Constraint Via HTTP OPTIONS, The HTTP Vocabulary In RDF, And Link Headers”. *WS-REST ’11 Proceedings of the Second International Workshop on RESTful Design:* 11–14. doi:10.1145/1967428.1967433.

Vu, Henry, Tobias Fertig ja Peter Braun. 2017. “Towards Model-driven Hypermedia Testing for RESTful Systems”. *Proceedings of the 13th International Conference on Web Information Systems and Technologies:* 340–343. doi:10.5220/0006353403400343.

———. 2018. “Verification of Hypermedia Characteristic of RESTful Finite-State Machines”. *WWW ’18 Companion: The 2018 Web Conference Companion:* 1881–1886. doi:10.1145/3184558.3191656.