Tampereen yliopisto

Kim Katainen

WORLD WIDE WEB -SOVELLUSPALVELUN TOTEUTTAMINEN REPRESENTATIONAL STATE TRANSFER -ARKKITEHTUURITYYLIN MUKAISESTI

Kandidaatintyö
Informaatioteknologian ja viestinnän tiedekunta
Tarkastajat: Tiina Schafeitel-Tähtinen
Lokakuu 2020

TIIVISTELMÄ

Kim Katainen: World Wide Web -sovelluspalvelun toteuttaminen Representational State Transfer -arkkitehtuurityylin mukaisesti
Kandidaatintyö
Tampereen yliopisto
Ohjelmistotekniikka
Lokakuu 2020

World wide web (www) -sovelluspalvelut ovat tietokoneohjelmien käyttämiä palveluja, jotka ovat isossa roolissa nykyaikaisessa sovelluskehityksessä. Tässä tutkielmassa tarkastellaan niiden toteuttamista Representational State Transfer (REST) -arkkitehtuurityylillä. Tavoitteena on selvittää ja arvioida REST-tyylin suosion taustalla olevia heikkouksia ja vahvuuksia erilaisissa www-sovelluspalveluiden käyttötapauksissa.

Työ aloitetaan määrittelemällä www-sovelluspalvelut ja REST-tyyli kirjallisuuden pohjalta. Molemmissa tapauksissa käy ilmi, että käsitteistä löytyy ristiriitaisia tulkintoja eri lähteisiin nojaten. Eritoten ei-akateemisista lähteistä saattaa löytyä jopa vääriä tulkintoja REST-tyylistä. Haasteita arviointivaiheeseen tuottaa myös se, että REST-tyylille ei löydy suoranaista vastinetta, johon vertailla.

Kehittäjän näkökulma tuodaan työssä esille toteuttamalla yksinkertainen esimerkkisovellus, joka toimii pohjana empiiriselle arvioinnille. Sovelluksessa hyödynnetään moderneja kehitystapoja ja teknologioita ja se on omiaan havainnollistamaan REST-tyylin soveltamista käytännössä. Sovelluksen yksityiskohtia kuvaava osio antaa nopeasti kattavan kuvan aiheesta.

Arviointivaihe tutkii REST-tyylin www-sovelluspalveluita hyödyntäen vapaamuotoisia sanallisia mittareita, sillä konkreettisia mittareita on abstraktin tason arkkitehtuurityylistä vaikea muodostaa. RESTiä verrataan ominaisuuskohtaisesti erilaisiin toteutustapoihin, kuten etäproseduurikutsuun (engl. Remote Procedure Call, RPC) ja SOAP-protokollaan, silloin kun se on mielekästä. Tulokseksi saadaan, että REST on pätevä tyyli lähes kaikkiin käyttötarkoituksiin. Se on erityisen hyvä palvelimen skaalautuvuuden kannalta tilattoman mallin ja löyhien kytkösten vuoksi. REST-tyylin ohjelmia on helppo jatkokehittää ja testata. Tehokkuuden kannalta REST ei kuitenkaan ole se paras vaihtoehto oikeastaan koskaan. Lisäksi palvelinsovelluksen painolasti siirtyy asiakassovelluksille, jotka hoitavat suuren osan laskennasta.

Avainsanat: www-sovelluspalvelu, REST, arkkitehtuurityyli, ohjelmointirajapinta, RPC, SOAP

SISÄLLYSLUETTELO

1 JOHDANTO

Nykyään verkkokehityksessä jaetaan isommat sovellukset pienempiin itsenäisiin kokonaisuuksiin. Erillisten palveluntarjoajien palvelimilla ajettavia ohjelmia käytetään oman käyttäjille tarjottavan sovelluksen rakennuspalasina. Näitä muiden sovellusten käyttämiä ohjelmia kutsutaan tässä työssä World Wide Web -sovelluspalveluiksi (www-sovelluspalvelut). Www-sovelluspalvelut tarjoavat käyttäjille verkon yli käytettävän rajapinnan, jota kutsumalla asiakassovellukset pääsevät käyttämään sen palveluita. Tyypillinen esimerkki tällaisesta palvelusta on staattisia karttoja tarjoava Googlen Maps Static API [1], jota kutsumalla voi vaikka kotisivuilleen lisätä karttakuvan halutusta paikasta maapallolla. Nykyään yrityksen koko toiminta voi perustua tällaiseen ohjelmallisesti käytettävään palveluun. Julkisia www-sovelluspalveluita on esitelty sivustolla programmableweb.com, joka esittäytyy eniten siteerattuna tietolähteenä muuan muassa rajapintojen statistiikkaan liittyvässä tutkimuksessa [2].

Kuten ohjelmistotuotannossa yleensä, myöskään www-sovelluspalvelujen tapauksessa ei ole olemassa yhtä kaiken kattavaa toteutustapaa. Ympäristöt eroavat muun muassa asiakkaiden määrässä, erilaisten käyttötapausten määrässä, turvallisuusvaatimuksissa ja tehokkuusvaatimuksissa. Vuosien varrella alalle on kehittynyt monia standardinomaisia toimintamalleja, joille yhteistä on HTTP ja URI -protokollien suosiminen. Aikaisemmin rajapintoja toteutettaessa suosittiin muun muassa SOAP-protokollaa ja RPC-arkkitehtuurityyliä, mutta tänä päivänä suureen suosioon on noussut Representational State Transfer (REST) -arkkitehtuurityyli.

REST ja Restful ovat muotisanoja, jotka ovat osaksi sekoittuneet kaikkien HTTP rajapintojen kanssa [3]. Näitä termejä käytetään niin löyhästi, että L. Richardson [4] on kehittänyt mittarin kuvaamaan www-sovelluspalvelun REST-tyylin astetta. Sekaannukselle on syynsä, sillä HTTP/1.1 ja URI ovat kehitetty REST-tyylin mukaisesti, Roy Fieldingin ollessa osana molempien standardien kehitystä [5]. Tästä huolimatta REST-arkkitehtuurityylin www-sovelluspalvelut eivät kuitenkaan nousseet suureen suosioon heti HTTP/1.1-standardin yleistyessä, vaan niiden käyttö on yleistynyt verkossa toimivien laitteiden määrän ja tiedonsiirron nopeuden kasvaessa. RESTin vahvuudet nousevat esiin yleishyödyllisissä, moneen eri käyttötarkoitukseen soveltuvissa palveluissa, joissa ei vaadita reaaliaikaista yhteyttä osapuolten välillä. Nykyään suositut yhden sivun sovellukset hyödyntävät REST-pohjaista rajapintaa keskusteluun palvelimen ja asiakassovellusten välillä.

Tämän kandidaatintyön tarkoitus on selvittää REST-tyylillä toteutun www-sovelluspalvelun haittoja ja hyötyjä verrattuna toisenlaisiin toteutuksiin. Aluksi esiteltävästä RESTin teo-

reettisesta pohjasta edetään käytännön toteutukseen käyttäen Node.js ohjelmointiympäristöä ja erinäisiä toteutusta helpottavia työkaluja. Hyötyjä ja haittoja arvioidaan ohjelmoinnin ja rajapinnan toimivuuden näkökulmasta käyttäen pohjana esimerkkitoteutusta ja aiheeseen liittyvää kirjallisuutta. Työssä ei luoda tai käsitellä varsinaisia REST-tyylin pohjalta luotuja arkkitehtuureja, vaan tarkastellaan tyylin toteuttamista käytännön tasolla mm. esimerkkikoodin myötä. Tähän päädyttiin aiheen pitämiseksi kandidaatintutkielman rajoissa, sekä siksi, että tällaisista arkkitehtuureista on todella vaikea löytää tietoa. Luvussa 2 määritellään www-sovelluspalvelut ja esitellään niiden erilaisia toteutustapoja. Luku 3 määrittelee REST-arkkitehtuurityylin yleisesti. Luvussa 4 käsitellään työn osana toteutettavan REST-tyylin www-sovelluspalvelun toteutusta. Luvussa 5 määritellään mittarit ja arvioidaan niiden pohjalta REST-tyylin www-sovelluspalveluita toteutetun sovelluksen ja kirjallisuuden pohjalta. Viimeisessä luvussa työstä muodostetaan yhteenveto.

2 WORLD WIDE WEB -SOVELLUSPALVELUT

Www-sovelluspalveluita on toteutettu eri muodoissa jo internetin varhaisista ajoista lähtien. Ennen puhuttiin Remote Procedure Call -arkkitehtuuritylistä, joka mahdollisti verkon yli tehtävien sovellusten kutsumisen. Sittemmin alalla ovat jalostuneet termit, kuten palvelukeskeinen arkkitehtuurityyli (Service-oriented architecture, SOA), ja sen pohjalta syntynyt tiedonsiirtoprotokolla SOAP (Simple Object Access Protocol). Www-sovelluspalveluiden toteutustapojen muovautumisessa tulee ottaa huomioon myös laitteiston ja eritoten internetin kehitys sen alkuvaiheesta nykypäivän laajaksi, miltei kaikkien saatavilla olevaksi jokapäiväiseksi kokonaisuudeksi. On selvää, että murroksen myötä myös www-sovelluspalveluiden käyttökohteet ja optimaalinen arkkitehtuuri on kokenut merkittäviä muutoksia.

2.1 Määritelmä

World Wide Web Consortium (W3C) määrittelemästä [6] englanninkielisestä käsitteestä Web Services käytetään tässä työssä termiä www-sovelluspalvelut Sanastokeskus TSK:n TEPA-termipankin mukaisesti [7]. Sen mukaan www-sovelluspalvelu tarkoittaa “verkkopalvelimessa toimivaa ohjelmaa, joka tarjoaa standardoitujen internetyhteyskäytäntöjen avulla palveluja sovellusten käytettäväksi”. Tämä korostaa sitä, että kyseessä on kahden tietokoneen välinen (machine-to-machine) palvelu/rajapinta, eivätkä viestinnän osapuolina ole esimerkiksi tietokone ja ihminen. Termiä ei pidä sekoittaa Web-palveluun tai verkkopalveluun, jotka tarkoittavat tällä määrittelyllä hieman poikkeavaa asiaa.

W3C:n vuonna 2004 julkaisema sanasto määrittelee Web Servicet ohjelmistoksi, joka tukee yhteensopivaa tietokoneiden välistä kanssakäymistä verkon ylitse. Määritelmän mukaan ohjelman rajapinta on määritelty koneellisesti prosessoitavassa muodossa ja muut järjestelmät kommunikoivat sen määrittämällä tavalla käyttäen SOAP-viestejä. Tämä on eksakti määritelmä liittyen W3C:n Web Services Architecture -dokumenttiin [8], mutta siihen liittyy maininta, että käsite “Web Service” saattaa tarkoittaa muuta eri kontekstissa. Se on kuitenkin erittäin hyvä pohja käsitteen ymmärtämiselle nykyaikaisen web-ohjelmoinnin ja tämän työn aiheen kannalta.

Toinen standardimääritelmä Web Serviceille löytyy Internet Engineering Task Forcelta (IETF). Tämä määritelmä pitää termiä Web APIs yhtäläisenä, sekä mainitsee Web Serviceitä joskus kutsuttavan myös HTTP APIksi tai REST APIksi [9]. Kyseessä on tuoreempi dokumentti vuodelta 2019, ja siinä mainitaankin nykyään suosittu Javascript Object No-

tation (JSON) [10] rajapinnan määrittelyyn XML:n sijasta. Kuten tässä kappaleessa on todettu, englanninkielinen Web Services ei ole tarkka standardikäsite. Eri lähteiden määritelmät ovat kuitenkin hyvin samanlaisia, yhteneväisinä piirteinä esimerkiksi palveluiden tarkoitus tietokoneiden käyttöön.

2.2 Toteutustavat

Remote procedure call (RPC) oli aikaisimpia verkon yli toteutettavien sovelluspalveluiden arkkitehtuurityylejä. Siinä on ideana käsitellä toisessa osoiteavaruudessa (esim. toisella tietokoneella) olevia proseduureja (funktioita, metodeja), kuin ne olisivat lokaalisti saatavilla. Käsite RPC vakiintui vuonna 1982 [11], mutta idean tasolla tyyli oli vaikuttanut ohjelmistoteollisuudessa jo aikaisemmin. RPC-tyylin toteuttamiseen voidaan käyttää muun muassa suosittua olio-ohjelmoinnin rajapintaa nimeltä Remote Method Invication (RMI) [12]. Toinen nykyaikainen ohjelmointikehys RPC:n toteuttamiseen on Googlen gRPC [13].

Keskeistä www-sovelluspalveluiden historian kannalta on palvelukeskeisen arkkitehtuurin (Service Oriented Architecture, SOA) nouseminen [14]. SOA on tutkittu, käytetty ja kehitetty käsite kirjallisuudessa eritoten 1990 ja 2000 -luvuilla ja se on toiminut pohjana lukuisille verkostuneiden sovellusten arkkitehtuurisille seikoille. SOA kannustaa komponenttien löyhiin kytköksiin (loose coupling) ja erilliseen kehittämiseen. Muun muassa nykyään suosittu mikropalveluarkkitehtuurityyli (Microservices) on kehittynyt tämän pohjalta.

Nykypäivänä datan siirto ei muodosta enää samanlaista pullonkaulaa kuin ennen, minkä ansiosta sovellukset voivat hyödyntää lukuisia eri www-sovelluspalveluita samanaikaisesti. Kuluttajille tarjolla olevien verkkoliittymien kaistanleveys mahdollistaa kymmenien eri pyyntöjen lähettämisen vaikka yksinkertaistakin verkkosivua avattaessa. Nykyään tyypilliset sivustot yhdistelevätkin lukuisia eri www-sovelluspalveluita muodostaakseen käyttäjälle näytettävän kokonaisuuden. Esimerkiksi suositut yhden sivun sovellukset (Single page application, SPA) lataavat ensimmäisellä latauksella vain sovelluksen loogisen rungon, joka sitten käyttää www-sovelluspalveluita hakeakseen näytettävät tiedot erikseen. Representational State Transfer -arkkitehtuurityyli on saanut paljon huomiota viime aikoina etenkin tällaisissa käyttötapauksissa.

RESTin mukainen www-sovelluspalvelu toteutetaan usein hyvin minimalistisesti, käyttäen käytännössä pelkästään www:n välttämättömiä protokollia ja standardeja, kuten HTTP:ta ja URI:a. Näiden REST-sovellusten tyypillinen ominaisuus on se, että data tarjotaan asiakkaalle prosessoitavaksi yleishyödyllisessä muodossa, kun palvelimella tapahtuva prosessointi jää vähäiseksi. Esimerkiksi Ebay:n REST-tyylinen myyntirajapinta tarjoaa keinot hakea, luoda ja muokata tuotteita, mutta tämän kaiken esittäminen käyttäjälle on asiakassovelluksen vastuulla. [15] Ero RPC arkkitehtuurityyliin voidaan ajatella verbien (tehtävät, funktiot) ja substantiivien avulla. Kun RPC-tyylillä palvelimelta kutsutaan suoritettavaksi joku verbillä kuvattava tehtävä (esim. lataa kuvat), löydettäisiin sama toiminnallisuus REST-sovelluspalvelusta substantiivilla kuvattavan resurssin (esim. kuvat)

alta.

3 REPRESENTATIONAL STATE TRANSFER-ARKKITEHTUURITYYLI

REST on arkkitehtuurityyli verkon yli toimivien hypermediajärjestelmien kehittämiseen. Sen esitteli alun perin Roy Fielding tohtorinväitöskirjassaan vuonna 2000 [5]. RESTiä käsitteenä käytetään usein virheellisesti, ja sen määrittelystä ollaan eri mieltä [3]. REST on suunniteltu nimenomaan suurimääräiseen hypermedian jakeluun World Wide Webissä, eikä se ole siten ole välttämättä tehokas muissa käyttötarkoituksissa. Tämä luku esittelee REST-tyylin sellaisena, kuin se on alunperin määritelty Fieldingin väitöskrjassa.

3.1 Arkkitehtuurityyli

REST on arkkitehtuurityyli (architectural style), eikä siis suunnittelumalli (design pattern) tai arkkitehtuuri. Roy Fielding on määritellyt arkkitehtuurityylin koordinoituna joukkona rajoitteita arkkitehtuuristen elementtien rooleihin, ominaisuuksiin ja sallittuihin yhteyksiin elementtien välillä missä tahansa arkkitehtuureissa, jotka noudattavat RESTiä. Käyttämällä arkkitehtuurista tyyliä arkkitehti rajoittaa ohjelmiston suunnittelua toivossa, että loppusovellus sopisi paremmin käyttötarpeisiin luonnollisessa ympäristössään. REST on tarkasti sanottuna hybdridityyli, sillä se pitää sisällään useita arkkitehtuurisia tyylejä. [16]

3.2 Rajoitteet

Käytännössä REST-tyyli koostuu joukosta rajoitteita, jotka esitellään tässä luvussa. Rajoitteet löytyvät alla olevasta listasta.

1. Asiakas-palvelin
2. Tilattomuus
3. Välimuisti
4. Yhtenäinen rajapinta
5. Kerrosmainen järjestelmä
6. Ladattava koodi

Ensimmäinen rajoite REST-arkkitehtuurityylissä on **asiakas-palvelin**–arkkitehtuurityylin käyttö. Tyylin ideana on jakaa järjestelmä kahdenlaisiin komponentteihin, asiakaskomponentteihin ja palvelinkomponentteihin. Asiakkaan roolina on aloittaa yhteydenpito palveli-

men kanssa ja palvelimen vastata asiakkaan pyyntöihin.

REST-tyylin mukainen sovellus ei tallenna minkäänlaista **tilaa** asiakkaan käytöstä palvelimen puolelle. Jokaisen lähetetyn pyynnön tulee sisältää kaikki tarvittava tieto sen suorittamiseen.

Vastauksissa täytyy määritellä, voiko sen tallentaa **välimuistiin**. Tulevaisuudessa asiakas saa käyttää välimuistiin tallennettua dataa lähettämättä uutta pyyntöä sen saamiseksi.

Fieldingin tohtorinväitöskirjan julkaisuaikaan **yhtenäinen rajapinta** oli suurin erottava tekijä muista verkko-ohjelmoinnin arkkitehtuurisista tyyleistä. Mahdollisimman yleishyödyllisellä rajapinnalla pyritään ohjelmiston arkkitehtuurin yksinkertaistamiseen ja toimintojen parempaan näkyvyyteen. Palvelut tarjoava rajapinta on erotettu niiden toteutuksesta. Yhtenäisen rajapinnan periaatteeseen kuuluu neljä eri lisärajoitetta.

1. Resurssien identifiointi
2. Resurssien manipulointi esitysten mukaan
3. Itsekuvavaat viestit
4. Hypermedia sovelluksen tilakoneena

Kaikki sellainen informaatio, joka voi olla käyttäjän hypertekstiviitteen kohteena täytyy määritellä resurssiksi. Resurssi on siis käsite, joka kuvaa (joukkoa) entiteettejä (esim. kuva). Resurssi voi viitata tyhjään joukkoon ja sen viittaama informaatio voi muuttua. Kohde onkin konsepti, kuten joka kerta uudelleen selvitettävä päivän sää, eikä välttämättä staattinen kohde. Tärkein asia resurssien määrittelyssä on kuitenkin se, että itse viittaukset (esim. URI) eivät muutu ajan saatossa. Tällä pyritään siihen, ettei linkkejä tarvitse päivittää aina datan muuttuessa. Resurssien tunnisteet nimeävä auktoriteetti, kuten webrajapinnan ylläpitäjä, nimeää tunnisteet tilanteessa parhaaksi näkemällään tavalla. Kyseinen auktoriteetti on vastuussa tunnisteiden semantiikan säilyvyydestä ajan kuluessa (toisin sanoen, että tunniste viittaa samaan resurssiin). Kuvassa 3.1 on esitetty esimerkkejä siitä, mihin edellä mainitut käsitteet viittaavat tyypillisessä www-sovelluksessa.

REST-komponentit käyttävät resursseja esitysten kautta, jotka kuvaavat resurssin nykyistä tai aiottua tilaa. Tällaisia esityksiä voivat olla HTML-tiedosto, kuva tai mikä tahansa tiedosto. Asiakaskomponentti lähettää muokatun esityksen palvelinkomponentille, joka sitten muokkaa siihen liittyviä resursseja uuden esityksen mukaisesti.

Resurssin esitykseen kuuluu resurssia esittävän datan lisäksi nimi-arvo pareista koostuvaa metadataa, joka kuvaa esitystä. Kontrollidata (eng. control data) määrittelee RESTkomponenttien välisen viestin tarkoitusta. Sen avulla muokataan muun muassa välimuistikäyttäytymistä (sallitaanko viestin tallennus välimuistiin). Esityksen dataformaattia kutsutaan englanninkielisellä termillä media type. Nämä lukeutuvat yleisimmin käytettyyn metadataan, jonka avulla REST-komponentit osaavat tulkita ja muokata viestien esityksiä.

Sovelluksen tila tallennetaan ja sitä kontrolloidaan asiakkaan (eng. user agent, esim.

Data Element	Modern Web Examples
resource	the intended conceptual target of a hypertext reference
resource identifier	URL, URN
representation	HTML document, JPEG image
representation metadata	media type, last-modified time
resource metadata	source link, alternates, vary
control data	if-modified-since, cache-control

Kuva 3.1. *Esimerkkejä REST-tyylin käsitteistä www:n tapauksessa [5]*

verkkoselain) toimesta. Tila voi koostua esityksistä, joita ollaan saatu useammalta, mahdollisesti täysin toisistaan riippumattomilta REST-tyylin palvelinsovelluksilta. Viesteissä tulee olla tarvittava tieto seuraaviin mahdollisiin tiloihin pääsemiseksi. Www-sovelluspalveluiden tapauksessa tämä tieto on yleensä seuraavien resurssien identifioivat URIt. Asiakassovellus navigoi tilasta toiseen käyttäen tätä viesteihin sisällytettyä tietoa.

Kerrosmaisen järjestelmän systeemi koostuu hierarkisista kerroksista, joista jokainen voi kommunikoida vain hierarkiassa yhtä pykälää ylä- tai alapuolella olevan kerroksen kanssa. Muita tällaisia systeemejä on muun muassa TCP/IP pino ja OSI-malli.

Ladattava koodi (code on demand) ei ole pakollinen rajoite RESTissä. Se mahdollistaa asiakkaan toiminnan laajentamisen lataamalla ja suorittamalla palvelimelta lähdekoodia.

4 TOTEUTUS

Tässä luvussa käsitellään työn ohella toteutettavan sovelluksen yksityiskohtia. Ensin esitellään käytettävät teknologiat, kuten ohjelmointikieli ja sovelluskehykset. Tästä edetään ohjelmiston kuvaamiseen luvussa 4.2.

4.1 Käytettävät teknologiat

Koska REST-arkkitehtuurityyli ei pakota minkään erityisten standardien tai protokollien käyttöä, onnistuu REST-pohjaisen www-sovelluspavelun toteuttaminen käyttäen vain kaikille verkkosivuille yhteisiä teknologioita. Tavoitteena on kehittää mahdollisimman vähän ylimääräistä HTTP-protokollan päälle ja tarjota kaikki data mahdollisimman intuitiivisesti käyttäen esitysmuotona JSONia.

Tässä luvussa teknologiat lukeutuvat tämän hetken suosituimpiin vaihtoehtoihin web-kehityksessä. Kaikkia niistä ei välttämättä tarvita minimalistisen sovelluksen kehittämiseen, mutta ne yksinkertaistavat luotettavan kokonaisuuden aikaansaamista. Viime kädessä olisi hyvin vaikeaa saavuttaa vastaava lopputulos hyödyntämällä vielä kevyempiä työkaluja.

4.1.1 Node.js

JavaScript [17] on verkkoselaimissa ajettava ohjelmointikieli, joka mahdollistaa toiminnallisuuden lisäämisen web-sivuille. Esimerkiksi Google Chromea käytettäessä Javascriptiä tulkataan V8-nimisen virtuaaliympäristön sisällä. Node.js on samaisen V8 ympäristön päälle rakennettu ympäristö, joka mahdollistaa Javascript-koodin kääntämisen ja ajamisen selaimen ulkopuolella, kuten verkkopalvelimella. Tämä mahdollistaa saman ohjelmointikielen, eli JavaScriptin, käyttämisen selaimessa ajettavan asiakasohjelman lisäksi myös palvelinpuolella. [18]

Node.js:n saa käyttöön asentamalla sen verkko-osoitteesta www.nodejs.org. Asennukseen kuuluu maailman eniten ohjelmistopaketteja sisältävä paketinhallintatyökalu Node Package Manager (NPM), eli valmiita ohjelmakirjastoja löytyy moneen lähtöön. Kaikki muu tämän kandidaatintyön projektissa tarvittava saadaan asennettua NPM:n komentorivityökalulla. Käytetyt paketit versioineen tallennetaan package.json-tiedostoon, jonka jälkeen ne voidaan asentaa ajamalla komento npm install samassa kansiossa.

4.1.2 Express

Vaikka REST-pohjaisen www-sovelluspalvelun voisi toteuttaa pelkästään Javascript ohjelmointikielellä Node.js-ympäristössä, tarjoaa Express-sovelluskehys siihen merkittäviä etuja. Sen sijaan, että toteuttaisi käytännössä kaikissa www-sovelluspalveluissa tarvittavia ominaisuuksia itse, tarjoaa Express valmiin, testatun ja suositun tavan ne vain muutamalla rivillä koodia. Teknisellä tasolla Expressin avulla voidaan rakentaa NodeJS ohjelmistokomponentteja, joita suoritetaan halutussa järjestyksessä ennalta määritellyn kaltaisen HTTP-pyynnön seurauksena. [19]

Express-kirjaston käyttöönotto Node.js-projektissa tapahtuu ajamalla komento “npm install express –save” projektin hakemistossa. Tämän jälkeen Express-reittejä luodaan koodissa seuraavalla tavalla. Express-instanssilla on jäsenfunktiot muun muassa HTTP-metodeille (GET, POST), joita kutsumalla voidaan asettaa ensimmäisen parametrin osoittamalle URI-polulle käsittelijä. Otetaan esimerkiksi kuvitteellinen sivusto, jonka URI alkaa "www.osoite.com". Asettamalla funktion ensimmäiseksi parametriksi “/login”, saadaan osoitteeseen “www.osoite.com/login” saapuvan HTTP-pyynnön seurauksena ajoon haluttu koodi. Alla oleva ohjelmakoodi kuvaa yksinkertaista ohjelmaa, joka palauttaa GET-pyynnölle osoitteeseen “http://localhost:3000/” vastauksen “Hello world!”.

```
const express = require('express')
const app = express()

app.get('/', (req, res) => {
  res.send('Hello World!')
})

app.listen(3000, () => {
  console.log(‘Example app listening at http://localhost:3000')
})
```

4.1.3 Swagger ja OAS

OpenAPI Specification (OAS) on SmartBearin kehittämä, aikaisemmin Swaggerinä tunnettu määritelmä rajapintojen dokumentointia varten. SmartBear Software esittelee OAS:n maailman standardina REST-rajapintojen määrittämiseen. Nykyään on tavallista aloittaa rajapinnan määrittelystä OAS:llä ja generoida lähdekoodin runko käyttäen erinäisiä työkaluja. [20]

OAS on kehittynyt erikseen, mutta nimellä Swagger viitataan vieläkin SmartBear Softwaren kehittämään ekosysteemiin, jolla helpotetaan rajapintojen kanssa työskentelyä hyödyntäen OAS:ää. SwaggerHub toimii pilvipohjaisena työkaluna kehittäjien yhteistyön helpottamiseen (vertaa GitHub [21]). Swaggeristä löytyy ilmainen editori rajapintojen määrittelemiseen OAS:n mukaisesti. Tämän lisäksi käyttäjällä on mahdollisuus generoida

palvelin- ja asiakasohjelmien rungot automaattisesti lukuisilla eri ohjelmointikielillä ja -ympäristöillä. Myös valmiista koodista on mahdollista generoida OAS:n mukaista rajapintamäärittelyä. Swagger helpottaa rajapintojen testaamista tarjoamalla graafisen esityksen ja yksinkertaisen testauksen esimerkiksi Swagger UI:n avulla. Nämä työkalut ja OAS ovat hyvin intuitiivisia käyttää ja nopeuttavat rajapintojen dokumentointia ja suunnittelua merkittävästi.

4.1.4 JSON web token

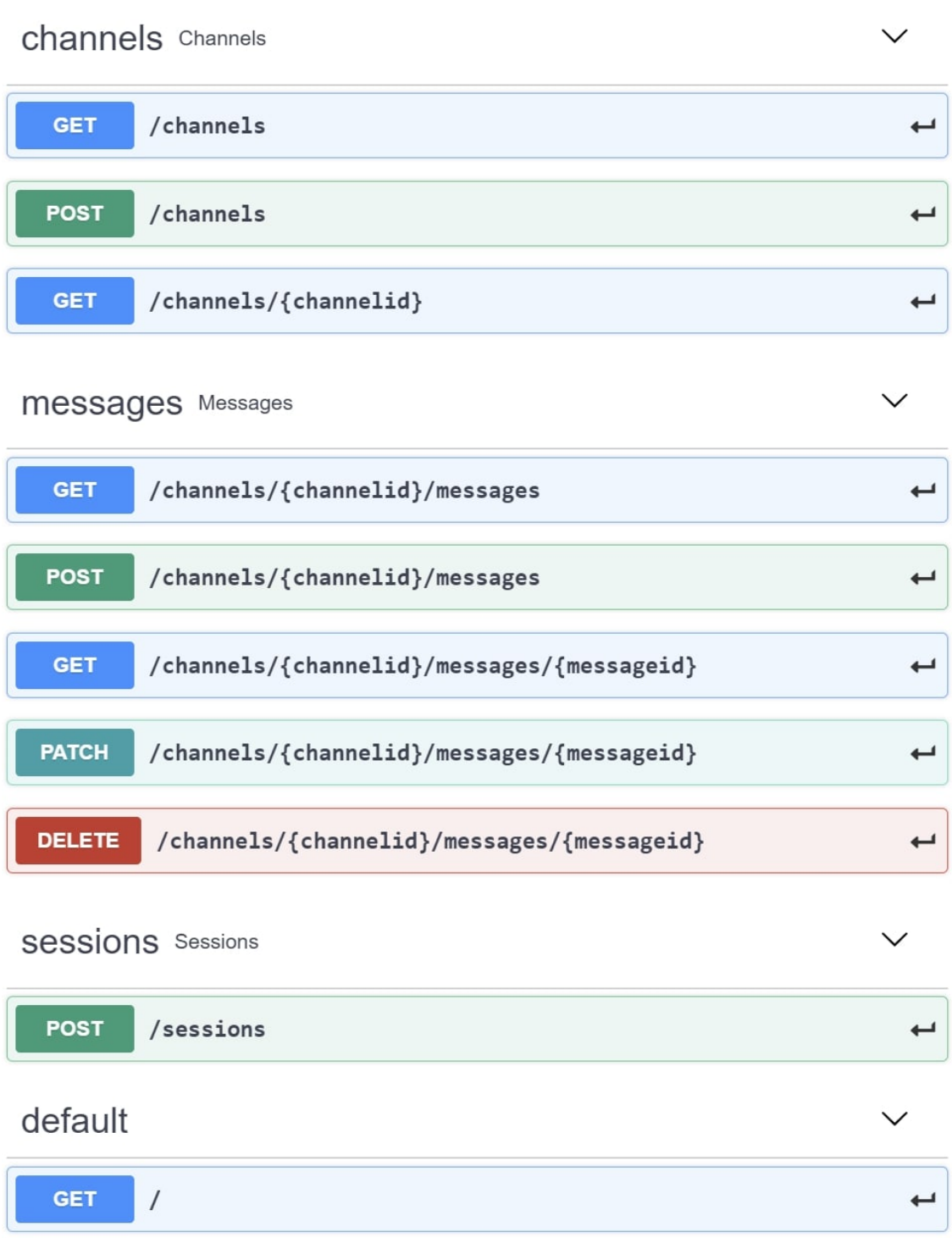
Autentikaation ja auktorisoinnin REST-arkkitehtuurityylin mukainen toteutus edellyttää, että palvelin ei tallenna asiakassovelluksen tilaa. Tämä käy helposti NPM:stä löytyvän jsonwebtoken kirjaston avulla, joka toteuttaa JSON Web Token (JWT) -standardin määrittelemän menetelmän hallita käyttäjäoikeustietoja eri sovellusten välillä. Kyseinen menetelmä ei rasita palvelinta JWT-tokenin luomisen ja tarkastamisen lisäksi millään tavalla. [22]

Ideana on tallentaa kirjautumiseen vaadittavat tiedot asiakkaalle lähetettävään merkkijonoon (Access token) sen sijaan, että mitään sessiotietoja käyttäjän kirjautumisesta tallennettaisiin palvelimen puolella. Merkkijono sisältää kirjautumistietojen (esim. käyttäjänimi ja kirjautumisaika) lisäksi palvelimen puolella kryptografisesti allekirjoitetun varmenteen. Asiakas voi sitten sisällyttää tämän merkkijonon autentikaatiota vaativiin pyyntöihinsä, josta palvelin voi sitten tarkastaa sen sisällön. Tarkastuksesta selviää, onko merkkijono luotu käyttäen palvelimen tuntemaa salaista avainta. Käytännössä siis jos palvelimen salainen avain on riittävän hyvä, kenenkään ei pitäisi pystyä väärentämään tällaista merkkijonoa, eli vain kyseinen palvelin voi niitä luoda, mahdollistaen käyttäjän tunnistamisen. [23]

4.2 Ohjelmiston yksityiskohdat

Toteutettava www-sovelluspalvelu pidetään hyvin yksinkertaisena. Pohjana on kuvitteellinen chat-sovellus, jossa käyttäjät voivat keskustella viestikanavilla. Luotavan www-sovelluspalvelun avulla chat-sovellusta voidaan käyttää ohjelmallisesti, esimerkiksi noutaa ja muokata viestejä eri kanavilla. Www-sovelluspalveluiden autentikoinnin ja auktorisoinnin toteuttamisen demonstroimiseksi luodaan myös kanava, johon pääsee käsiksi vain kirjautuneena. Palvelua havainnollistaa hyvin kuvassa 4.1 määritellyt URI:t, jotka identifioivat tarjolla olevat resurssit REST-tyylin mukaisesti. Tämä graafinen esitys on osa Swagger Editorin graafista käyttöliittymää, joka on luotu liitteen 1 OAS-määrittelyn pohjalta.

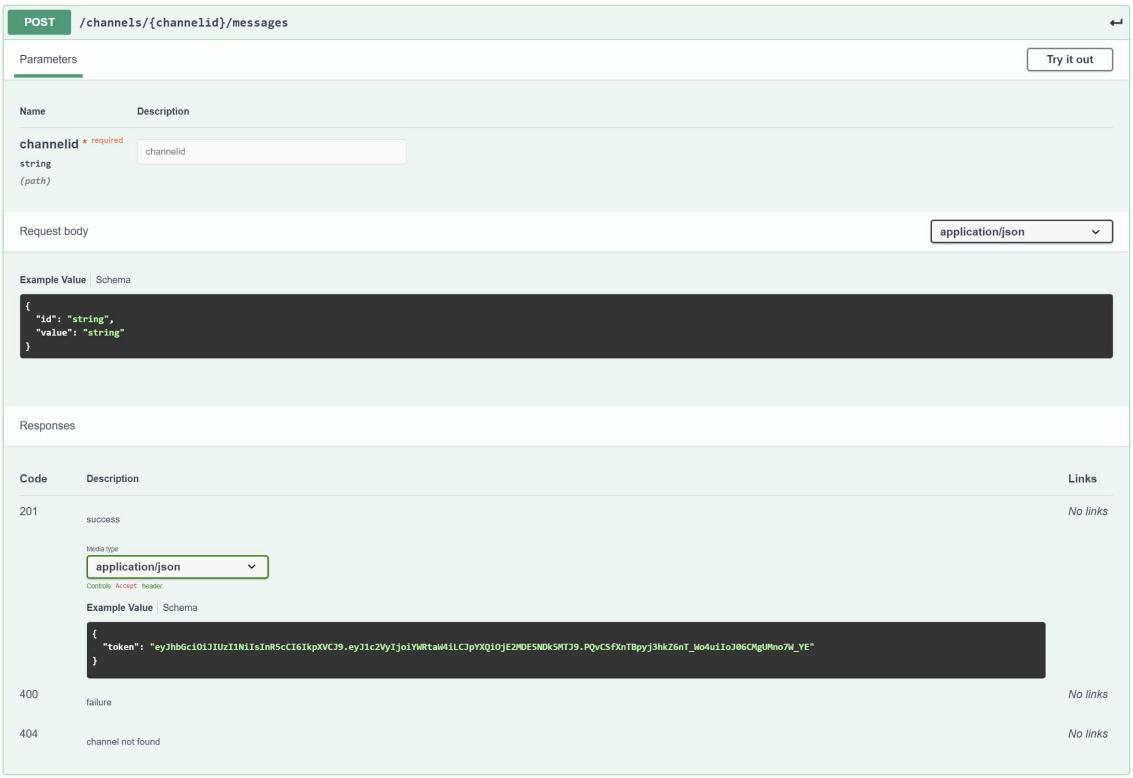
Ohjelmaa käytetään http-protokollan yli (eli esim. internet-selaimella) käyttäen osoitteena URI:ja, jotka muodostuvat www:ssä isännöidyn palvelimen kantaosoitteen perään lisätyistä kuvan 4.1 poluista. Esimerkiksi paikallisella tietokoneella portissa 8080 ajettavan palvelimen tapauksessa keskustelukanavat saataisiin osoitteella "http://localhost:8080/channels". Aaltosulkeissa olevat termit ovat muuttujia, joista saa-



Kuva 4.1. *Www-sovelluspalvelun tarjoamat resurssit Swagger Editor -työkalusta.*

daan Swagger-työkalussa enemmän tietoa avaamalla joku URI:sta (kuva 4.2). Esimerkiksi lähettämällä HTTP GET -viesti osoitteeseen http://localhost:8080/channels/1/messages/2 päästään käsiksi id:n 1 omaavan kanavan viestiin, jonka id on 2.

Huomataan, kuinka kaikkiin rajapinnan URI:hin liittyy joku HTTP metodi (GET, POST, PATCH, DELETE). REST-tyylin www-sovelluspalvelut toteutetaan usein tällä tavoin (ku-



Kuva 4.2. *Yksittäisen reitin tarkastelu Swagger Editor -työkalulla.*

ten [24] ja [25]), käyttäen HTTP-protokollan määrittelemiä viestejä niiden standardinmukaisessa [26] käyttötarkoituksissa. Näin muodostuu äärimmäisen minimalistinen sovellus ilman www:n määrittelyn päälle lisättyjä standardeja. Kun käyttäjä tietää resurssin URI:n, hän päättää siihen kohdistuvan toiminnon valitsemalla HTTP-metodin, eli resurssin identifiointi on erotettu sen muokkauksesta. Otetaan esimerkiksi edellä mainittu URI:n http://localhost:8080/channels/1/messages/2 identifioima resurssi. PATCH metodilla käyttäjä voi muokata tätä viestiä, GET metodilla hän saa haettua viestin esityksen ja DELETE-metodilla hän voi poistaa kyseisen viestin. Usein verkkosovellusten URI:t sisältävät tiedostopäätteen, kuten .html tai .php. REST-tyylisen sovelluspalvelun URI:en päähän ei lisätä tiedostotyyppiä, vaan se määritellään HTTP-pyynnön headers-osiossa. Tässä työssä kaikki viestit ovat kuitenkin JSON-muodossa.

REST-tyylin mukaan viestien tulee olla itsekuvaavia, kuten luvussa 3.2 on todettu. Asiakkaan ja palvelimen välisessä HTTP-viestinnässä tämä on toteutettu käyttämällä tiettyjä header-kenttiä, kuten RFC 4220:ssä on määritelty. Viestin välimuistikäyttäytymistä ohjataan Cache-Control–kentällä ja viestin sisällön tyyppi on ilmoitettu Content-Type–kentässä. Palvelimen lähettämän paluuviestin HTTP status kentästä käy ilmi operaation onnistuminen [26], esimerkiksi onnistuneen GET-operaation status on 200 ja POST-operaation 204. Tämä on toteutettu esimerkkisovelluksessa jokaisen Express-reitin päätteeksi. Itsekuvaaviin viesteihin liittyy usein unohdettu yksityiskohta, jonka mukaan rajapinnan käyttäjän (lue asiakassovellus) pitäisi pystyä navigoimaan haluamaansa esitykseen tietäessään ainoastaan rajapinnan juurihypertekstiviitteen (esim. URI) perusteella. Tämä on toteutettu lisäämällä palautettaviin resursseihin URI:t, joilla pääsee käsiksi toisiin resursseihin.

Kuvassa 4.1 määritellyt URI:t käsitellään Expressin funktioiden avulla, jotka sijaitsevat projektissa kansion routes tiedostoista. Esimerkiksi channels.js tiedosto kuvan 4.1 channels otsikon alla olevat URI:t. Jokaisen reitin viimeinen funktio muodostaa lähetettävän vastauksen. Niitä ennen saatetaan suorittaa muita funktioita. Autentikaatio on toteutettu reitin /sessions alla, johon käyttäjä lähettää POST-pyynnöllä käyttäjätunnuksen ja salasanan. Työssä on tämän demonstroimiseksi luotu valmis käyttäjä nimellä “admin” ja salasanalla “admin_pass”. Vastauksena palautettava JWT token tulee sisällyttää HTTP otsikkotietoihin authorization otsikolla ja lisäämällä tokenin eteen “bearer ” [27]. Id:n 0 omaava kanava on asetettu tarkastamaan tämä token, kun siihen yritetään ottaa yhteyttä. Toiminnallisuus tälle löytyy auth.js-tiedostosta.

5 ARVIOINTI

Www-sovelluspalvelun kehittämistä REST-tyylillä arvioidaan tässä kandidaatintutkielmassa kirjallisuuden pohjalta ja empiirisesti kehittäjän näkökulmasta perustuen toteutettuun esimerkkisovellukseen. Esimerkkisovelluksen kohdalla täytyy muistaa, että tulokset eivät välttämättä ole yleistettävissä kaikille REST-tyylin palveluille. Kun on määritelty arvionnin mittarit seuraavassa luvussa 5.1, edetään niiden pohjalta saatujen tulosten käsittelyyn luvussa 5.2. Mittarit tarkastelevat REST-tyylin ominaisuuksia vapaasti sanallisessa muodossa. Konkreettisten mittarien luonti (esim. suorituskyvylle) on varsin hankalaa sillä REST on abstrakti käsite kaukana käytännön toteutuksesta, eikä sille ole suoranaista vastinetta. Yksittäisille REST-tyylin piirteille esitetään kuitenkin vaihtoehtoisia toteutustapoja, joihin ratkaisua voidaan peilata.

5.1 Mittarit

Käytetyt mittarit ovat sanallisia näkökulmia, jotka pyrkivät muodostamaan mahdollisimman kattavan kuvan REST-tyylin heikkouksista ja vahvuuksista. Mittarit on jaettu kahteen osaan, kehittämisen näkökulman mittareihin ja arkkitehtuuriset laatumittarit tarkastelevat REST-tyylin ominaisuuksia ja soveltuvuutta eri tilanteissa kirjallisuuden pohjalta. Mittareina toimivat suorituskyky, turvallisuus, luotettavuus, soveltuvuus ja ylläpidettävyys. Suorituskyky kuvaa tyylin vaikutusta muun muassa muistinkulutukseen ja prosessointiaikaan. Tietoturvan kannalta merkittäviä ominaisuuksia käsittelee turvallisuusmittari. Ylläpidettävyys kertoo siitä, miten hyvin REST-tyylin sovellus on korjattavissa ja jatkokehitettävissä. Tähän liittyvät muun muassa testattavuus ja ohjelmakomponenttien modulaarisuus. Soveltuvuus kuvaa REST-tyylin mukaisen arkkitehtuurin vahvuuksia ja heikkouksia eri käyttötarkoituksissa.

Kehittämisen näkökulman mittarit kuvaavat REST-tyylin vaikutusta sovelluksen ohjelmointiin pohjautuen luvussa 4.2 esitettyyn sovellukseen. Kehittämisen helppoutta käydään lävitse REST-tyylin ymmärtämisen ja oppimisen lisäksi siltä kannalta, kuinka monimutkaista www-sovelluspalveluita on kehittää ja muokata tyylin sisäistämisen jälkeen. Toinen kehittämisen mittari on nopeus. Siinäkin eritellään tapaukset, joissa ohjelmoijat ovat tai eivät ole hyvin perillä RESTin periaatteista entuudestaan.

5.2 Tulokset

Tulosten käsittely aloitetaan keskittymällä kehittämisen mittareihin ja jatkaen siitä kirjallisuuden pohjalta suoritettavaan arkkitehtuuriseen analyysiin. Tulokset kuvataan hyvin vapaasti sanallisessa muodossa, sillä esimerkiksi numeropohjaisen arvostelun luominen aiheesta olisi hyvin hankalaa. Sen sijaan ominaisuuksia verrataan suhteessa toisiin mahdollisiin toteutustapoihin.

5.2.1 Kehittäminen

Kun ohjelmoija on sisäistänyt REST-tyylin, verkkorajapintojen toteutus on yksinkertaisimmillaan äärimmäisen nopeaa. Usein verkkorajapinnat, jotka vaikuttavat ja väittävät noudattavansa RESTiä, eivät kuitenkaan ole oikeasti täysin RESTin mukaisia. Tällaisessa tilanteessa nousevat esiin RESTin periaatteiden tarkoitusperä; tilaton ja yksinkertainen kommunikointi on äärimmäisen yksinkertaista toteuttaa, mikäli sekä verkkorajapintaa käyttävä, että sitä hyödyntävä ohjelmoija ovat todella sisäistäneet arkkitehtuurityylin. Tällöin he voivat kehittää omia osiaan ohjelmasta luottaen rajapintojen toimivuuteen ja yksinkertaiseen rakenteeseen. Tilatonta rajapintametodia tarkastellessa saa parhaimmillaan yhdellä silmäyksellä selville sen käyttötarkoituksen, mikäli kaikki resurssien kentät ovat ymmärrettävissä. Tämä helpottaa suunnittelua huomattavasti verrattuna tilalliseen sovellukseen, jossa on käytännössä yksi ulottuvuus lisää murehdittavana.

REST-arkkitehtuurityyli on osoittautunut hankalaksi ymmärtää. Esimerkkisovellusta toteuttaessa tämä kävi ilmi erityisesti kirjautumista pohdittaessa. Tässä tapauksessa hankaluuksia tuottaa erityisesti rajapinnan pitäminen tilattomana, sillä yleensä käytetty kirjautumissession tallentaminen ja tarkastaminen palvelimen puolella ei ole sallittua. Lisäksi esimerkiksi kirjautumissession esittäminen resurssina on varsin hankalaa. Usein verkkorajapinnoissa näkee /login-tyylisiä URI-päätteitä kirjautumiselle, mutta tämä menee helposti RPC-tyylin puolelle, jos se tarjoaa funktion eikä resurssia. Esimerkkisovelluksessa tämä on ratkaistu käyttämällä kuvaavampaa /session-päätteistä URI-polkua, jonka voi asettaa tietäessään salasanan ja käyttäjänimen.

REST on huomattavasti nopeampi oppia, kuin esimerkiksi tarkoin määritelty ja laaja SOAP-protokolla. SOAPia opetellessa aika kuluu standardin pikkutarkkaan lukemiseen, kun taas REST-verkkorajapinnan kohdalla vaikeus muodostuu abstraktin asian tulkitsemisesta. RESTin perusidea on ymmärrettävissä tutkimalla hyvin toteutettua esimerkkipalvelua, mutta tilattomuuden ja yhteisen rajapinnan sisäistämiseksi on tukeuduttava aineistoon. Työn esimerkkisovellus kuvastaa hyvin sitä, kuinka yksinkertainen rakenne minimalistisella REST-sovelluksella on. Esimerksiksi voidaan ottaa HTTP-metodien käyttö niiden alkuperäisessä tarkoituksessa. SOAP1.2 käyttää kaikkiin rajapintakutsuihin HTTP POST metodia ja viestin varsinainen tarkoitus löytyy HTTP:n päälle rakennetusta standardista [28].

Esimerkkisovelluksen kehitys vaikutti lyhyen REST-tutustumisen jälkeen hyvin yksinker-

taiselta. Hetken tarkemman tutustumisen jälkeen huomasi kuitenkin, että REST-tyylin alkuperäinen kuvaus täytyy todella lukea ajatuksella, jotta sen todella sisäistää. Tämä on kuitenkin mahdollista, sillä kyseinen kuvaus ei ole järin pitkä. Nyt kun REST-tyylin on kerran opetellut, voisi sitä noudattavan sovelluksen tehdä erittäin pienessä ajassa. Koska REST ei ole tarkka standardi (kuten SOAP), sitä ei välttämättä tarvitse enää lukea uudestaan lainkaan. Eritoten esimerkkisovelluksen tyylisiin www-sovelluspalveluihin REST sopii erittäin luontevasti.

Joskus SOAP www-sovelluspalvelu on yksinkertaisempaa toteuttaa kuin REST-pohjainen www-sovelluspalvelu. Tämä pätee erityisesti monimutkaisiin järjestelmiin, jossa tarvitaan paljon esimerkiksi luotettavuutta ja turvallisuutta parantavia mekanismeja. Tällöin SOAP-palvelu vaatii paljon vähemmän ohjelmointia sen sisäänrakennettujen ominaisuuksien vuoksi. Esimerkiksi SOAP1.1 määrittelee viestien uudelleenlähetyksen käytännön tilanteessa, jossa toiseen osapuoleen ei saatu yhteyttä.

Erittäin suuri etu RESTillä muodostuu dokumentoinnin kanssa. Koska resurssit nimetään mahdollisimman kuvaaviksi, ei dokumentointia usein tarvita juuri lainkaan, tai se voidaan hyvin suurilta osin automatisoida vaikkapa Swaggerillä. Tämä käy ilmi kuvan 4.1 esimerkkisovelluksen resurssikuvauksesta, joka itsessään kuvaa todella hyvin www-sovelluspalvelun toiminnan. Täytyy muistaa, että kyseessä on yksinkertainen sovellus, mutta tässä tapauksessa muuta dokumentointia ei lyhyen yleiskuvauksen lisäksi tarvita. SOAP, RPC jne. tapauksessa näin ei ole, sillä rajapinnan käyttäjä ei voi ikinä olla varma sovelluksen tilan muutoksien aiheuttamista sivuvaikutuksista. Tilannetta voidaan verrata funktionaalisen ja imperatiivisen ohjelmoinnin eroon funktion sivuvaikutusten osalta [29].

5.2.2 Suorituskyky

Suorituskyvyn osalta REST-tyylin sovellus poikkeaa suuresti muiden tyylien luontaisten toteutusten suorituskyvystä. Koska REST-tyyli vastaa yleisen tapauksen www-sovelluksen tarpeita, se ei voita spesifiin käyttötarkoitukseen erikoistunutta verkkorajapintaa maksimaalista suorituskykyä haluttaessa. Vähintään itsekuvaavat viestit muodostavat ongelman, sillä viestejä voi aina optimoida karsimalla pois kaiken muun, paitsi käytettävän datan. Tämä edellyttää sitä, että palvelin ja asiakas tietävät datan formaatin etukäteen, mikä sotii REST-tyyliä vastaan. Tästä syystä REST-arkkitehtuurityylin verkkorajapintojen käyttämä viestintä onkin varsin verkkoa kuormittavaa. Esimerkiksi RPC-tyylillä voidaan toteuttaa vähemmän dataa kuluttava verkkorajapinta. Tältä osin REST suoriutuu kuitenkin SOAPia paremmin, sillä SOAP kirjeet (envelope) sisältävät vielä raskaampaa XML-muotoista dataa, jossa on tarkkaan määriteltyä tietoa muun muassa jokaisen parametrin tyypistä. [28]

REST-tyylin erityinen etu on palvelimen vähäinen rasitus. Tässä suuressa osassa on viestinnän tilattomuus, joka vähentää suuresti fyysisten resurssien kulutusta. Sen ansiosta liittimien (connector), eli REST-ohjelmiston kommunikoinnin hoitavien ohjelmistokomponenttien, ei tarvitse tallentaa pyyntöjen välistä tilaa. Koska jokainen viesti sisältää kai-

ken tarvittavan tiedon pyynnön ymmärtämiseksi, voidaan rajapintakutsujen prosessointi myös samanaikaistaa tietämättä kutsun semantiikkaa. Tämä on merkittävä etu, sillä nykyään tietokoneiden suorituskyvyn kasvu perustuu hyvin pitkälti laskusuoritusten samanaikaistamiseen muun muassa prosessointiytimien lukumäärää kasvattamalla. Esimerkiksi SOAP-protokolla ja RPC-tyyli eivät tätä itsessään mahdollista.

Palvelimen vähäinen suorituskykytarve tarkoittaa isoa taakkaa asiakassovelluksille. Taakkaa voidaan helpottaa hyödyntämällä välimuistiin tallennettua dataa uuden hakemisen sijaan. Muilla toteutustavoilla, kuten RPC-tyylillä ja SOAP-protokollalla voidaan taakkaa siirtää palvelimen puolelle. Kerrosmaisen järjestelmän rajoite lisää latenssia datan prosessointiin, sillä rajapintakutsut eivät voi kutsua suoraan eri kerroksissa olevia ohjelmistokomponentteja. [5]

5.2.3 Soveltuvuus ja ylläpidettävyys

Soveltuvuuden kannalta REST-tyyli on varsin yleinen, ollen toimiva valinta käytännössä aina, kun asiakkaita on paljon suhteessa palvelimiin. Muita tapauksia ovat muun muassa yksittäisen yrityssovelluksen sisäiset rajapinnat. Niissä asiakas- ja palvelinsovelluksia monesti toteuttavat samat henkilöt, joten REST-tyylin rajapinnan helppo ymmärrettävyys ja dokumentoitavuus eivät luo vaikutusta. Tällöin lienee luontevampaa toteuttaa SOAP- tai RPC-tyylinen www-sovelluspalvelu. Usein toisenlaisissa sovelluksissakin löytyy täysin puhdasta REST-tyyliä toimivampi kokonaisuus. Kuten luvussa 5.2.2 mainittiin, tehokkuuskriittisiin sovelluksiin löytyy parempi ratkaisu lähes aina.

REST-tyylin www-sovelluspalvelut skaalautuvat erittäin hyvin usealle asiakkaalle. Luvun 4 esimerkkisovellus on todella yksinkertainen palvelinpuolen sovellus, joka on nopea toteuttaa. Asiakassovellusta toteuttaessa täytyy tehdä enemmän. Se, että ei tarvitse välittää viestien semantiikasta, vähentää merkittävästi ohjelmiston komponenttejen monimutkaisuutta. Jos näin ei olisi määritelty, täytyisi komponentteja toteuttaessa tietää, onko viesteillä eri vaikutus esimerkiksi niiden saapuessa eri järjestyksessä. Lisäksi voisi joutua kehittämään hyvinkin monimutkaisia mekanismeja rinnakkaisuuden ongelmien välttämiseksi. Pilvipalveluissa suoritetaan ohjelmia jopa yksittäinen funktio kerrallaan, jolloin ohjelmapätkä laitetaan suoritukseen millä tahansa alustalla löytyvällä suoritusyksiköllä. REST-tyyli sopii tähän tilanteeseen hyvin, mahdollistaen suorituksen dynaamisen uudelleenjärjestelyn yksiköiden välillä.

Reaaliaikaisten palveluiden tuottaminen ei ole optimaalista REST-tyylillä, sillä palvelin ei saa aloittaa yhteyttä asiakkaaseen, eikä palvelimen ja asiakkaan välille voi muodostaa pitkäaikaista yhteyttä. Asiakassovelluksen pitää reaaliaikaisessa REST-sovelluksessa kysellä palvelimelta muutoksista esimerkiksi tietyin väliajoin (polling), mikä rasittaa palvelinta turhaan tapauksessa, jossa uutta dataa ei ole saatavilla. Reealiaikaisen rajapinnan toteuttamiseen on monia parempia vaihtoehtoja, esimerkiksi verkkoselaimessa toimivan asiakassovelluksen tapauksessa WebHookit. Mikäli reaaliaikaisuutta kaivataan REST-pohjaiselta verkkosovellukselta, on kuitenkin mahdollista kehittää sekä palvelimen

ja asiakkaan välille erillinen palvelu, joka ilmoittaa asiakkaalle uuden datan tullessa tarjolle. Näin siis rakennetaan lisämekanismi olevassa olevan REST-sovelluksen rinnalle ja voidaan valinta uuden datan reaaliaikaisesta hakemisesta siirtää asiakkaalle.

Kerrosmainen järjestelmä vähentää kokonaisuuden kompleksisuutta ja korostaa komponenttien riippumattomuutta rajaamalla tietoa yhteen tasoon. Se mahdollistaa esimerkiksi vanhojen legacy-systeemien palveluiden piilottamisen uuden rajapinnan taakse ja välityspalvelimen (proxy) lisäämisen muuttamatta valmiita rajapintoja. Yhdistämällä kerrosmaisen järjestelmän rajoite yhtenäisen rajapinnan rajoitteeseen saadaan aikaan liukuhihnaarkkitehtuurityylin (pipeline/pipe-and-filter) kaltainen kokonaisuus [30].

Yhden sivun sovellukset (Single Page Application, SPA), ovat hyvin yleinen tyyli toteuttaa verkkosivuja. Niille tyypillistä on datan inkrementiaalinen renderöinti käyttäjälle sen sijaan, että kaikki haettaisiin ja näytettäisiin yhdellä kertaa. Tällaisten sovellusten taustalla on todella usein REST-tyylin www-sovelluspalvelu, sillä asiakassovellukselle on käytännöllistä ladata erikseen yleishyödylliset resurssit ja päättää niiden käyttö itse. [31]

Perinteisessä palvelin-asiakas–tyylissä palvelin renderöi ja lähettää datan asiakkaalle valmiisssa esitysmuodossa [32]. Tällöin asiakassovelluksen kehittäminen on yksinkertaisempaa ja alkuperäistä dataa ei altisteta asiakkaan tietoon. Tämä kuitenkin rajoittaa paljon asiakassovelluksen toimintaa ja on rasite palvelimelle. REST-tyyli ratkaisee alkuperäisen datan piilottamisen siten, että resursseista lähetetään standardimuodosssa (kuten html tai json) oleva kuvaus, eikä itse resurssia.

Ylläpidettävyys on oikein toteutetun REST-tyylin www-sovelluspalvelun parhaimpia ominaisuuksia. Testattavuuden tekee helpoksi muun muassa valmiit työkalut ja eri komponenttien erittäin löyhät kytkökset (loose coupling) yhtenäisen rajapinnan rajoitteiden ansiosta. Muuttamatta rajapintoja voidaan yksittäisten komponenttien toteutus vaihtaa ja kehittää erikseen, sekä ajaa testejä vain näille kokonaisuuksille. Esimerkkisovellusta ajateltaessa voidaan taustalla oleva Node.js-sovellus vaihtaa mihin tahansa muuhun koskematta luvussa 4.2 määriteltyyn rajapintaan. Perinteisissä monoliittisissa järjestelmissä riippuvuudet voivat tehdä esimerkiksi ohjelman osien erillisen suorittamisen ja testaamisen mahdottomaksi. Myös SOAP-protokollan kiinteät kytkökset (tight coupling) voivat muodostaa vastaavia ongelmia [33]. Osia on tällöin todella myös vaikea vaihtaa toisiin, toisin kuin REST tyylillä, jolla muodostuu luontaisesti modulaarisia sovelluksia. Jos pienetkin osat kehitetään erillisiksi REST-tyylin komponenteiksi, kuten mikropalveluarkkitehtuurityylissä, saadaan kasaan äärimmäisen modulaarinen sovellus.

5.2.4 Luotettavuus ja turvallisuus

REST-tyyli ei tarjoa juurikaan suoranaisia luotettavuuteen vaikuttavia ominaisuuksia, kuitenkin ohjatessaan tietyillä tavoilla luotettavuutta parantavaan toteutukseen. Muun muassa osittaisista virheistä on helppo palautua itsekuvaavien viestien ansiosta. Yhden viestin aloittaman suorituksen alkuun palaaminen riittää aina, toisin kuin esimerkiksi järjestelmissä, joissa suoritusketju on muodostunut useamman viestin seurauksena. Lokin ylläpitämi-

nen helpottuu samasta syystä. Välimuistin käyttäminen voi huonontaa REST-tyylin www-sovelluspalvelun luotettavuutta, mikäli välimuistin käyttäytyminen ei ole tarkkoin määriteltyä. Näin voi käydä esimerkiksi jos ei huomata datan välimuistissa olevan vanhentunutta ja sitä käytetään silti. Toisaalta välimuistin ja sovelluksen eri tilojen käsittelyn luotettavuutta edistää se, että tiedostoformaatit ovat määritelty muualla kuin URI:ssa (esim. HTTP header -kentässä). Tämä indikoi sitä, että eri tiedostomuodossa oleva esitys viittaa kuitenkin samaan resurssiin, mahdollistaen niiden esitysten samana esimerkiksi välimuistissa. Vaatimus resurssilinkkien (URI) semantiikan staattisuudelle on selvä hyöty, kun ajatellaan www-sovelluspalveluita tai verkkosivuja yleensä. Usein verkossa näkee, että saman sivun URI vaihtuu jatkuvasti. Osoitteet eivät välttämättä kuvasta mitään resurssia ja saattavat pysyä samana, vaikka sivustolla liikuttaisiin näkymästä toiseen. Tämä aiheuttaa sen, ettei aikaisemmin tunnetun osoitteen voi olettaa viittaavan samaan sisältöön, eikä sisältöä voi tallentaa välimuistiin. RESTin staattiset osoitteet korjaavat nämä ongelmat.

Turvallisuuteen vaikuttavat piirteet REST-tyylissä voidaan tiivistää kahteen asiaan. Palvelimen resurssit kannustetaan paljastamaan vain esitysten kautta ja turvallisuutta lisääviä ominaisuuksia on helppo lisätä järjestelmään. REST ei kuitenkaan itsessään määrittele näitä ominaisuuksia lainkaan. Esimerkiksi pääsynvalvontaan (access control) joutuu kehittämään tai integroimaan kolmannen osapuolen toteutuksen. Toisin on esimerkiksi SOAP-protokollan kanssa, jossa tähän löytyy virallisesti protokollassa mainittu lisä. Esimerkkisovellusta tehdessä kului merkittävä osa ajasta autentikointi- ja auktorisointiratkaisun löytämiseen ja integrointiin. Tämä kuvastaa sitä, että turvallinen ohjelmointi vaatii REST-tyylillä lisäponnisteluja.

6 YHTEENVETO

Tietokoneohjelmien välinen keskustelu sovelluskehityksessä on noussut arkipäiväiseksi asiaksi internetin suosion siivillä. Sen myötä on muodostunut paljon erilaisia standardeja ja toteutustapoja kommunikoinnin toteuttamiseen. Termi www-sovelluspalvelu tarkoittaa toisten tietokoneohjelmien käyttämää sovellusta tarjolla www:n kautta.

Ainakin verkkosivuja kehittänyt ohjelmoija on nykypäivänä varmasti törmännyt termiin REST. REST on muotisanaksi muodostunut arkkitehtuurityyli hajautettujen hypermediajärjestelmien toteuttamiseksi ja sitä hyödynnetään erityisesti www-sovelluspalvelujen tapauksessa. Abstraktin tason tyyli ei ota kantaa toteutukseen, teknologioihin tai käyttökohteisiin vaan ohjaa kehitystä tiettyyn suuntaan kuuden rajoitteen avulla. REST-tyylin erityinen piirre on se, että se tarjoaa asiakassovelluksen käyttöön niin kutsuttuja resursseja esimerkiksi RPC-tyylistä tuttujen funktioiden sijaan.

REST-arkkitehtuurityylin rajoitteet jättävät toteutuksen yksityiskohdat huomiotta keskittyäkseen ohjelmistokomponenttien rooleihin. Asiakas-palvelin–rajoite jakaa komponenttien roolit asiakkaaseen ja palvelimeen saman nimisen arkkitehtuurityylin mukaan. Tilattomuus yksinkertaistaa palvelimen toteutusta siirtämällä ohjelmistokokonaisuuden tilan käsittelyn täysin asiakkaan puolelle. Välimuistirajoite lisää kokonaisuuteen luotettavan keinon vähentää fyysistä kommunikaatiota määrittelemällä tarkat ohjeet vanhojen viestien käyttämiselle. Yhtenäisen rajapinnan rajoite voidaan jakaa neljään osaan. Palvelun tarjoamat resurssit on identifioitava asiakkaan käytettäviksi mahdollisimman muuttumattomilla osoitteilla (esim. URI). Rajapinnan käyttäminen perustuu näiden resurssien manipulointiin hyödyntämällä osoitteiden avulla saatavia esityksiä. Kaikkien viestien tulee sisältää tarvittava tieto halutun operaation kokonaiseen suorittamiseen. Lisäksi asiakas tallentaa sovelluksen tilan hypermediana. Viides rajoite on kerrosmainen järjestelmä, joka jakaa sovelluksen vain suoraan ylä- ja alapuolella olevien kerroksien kanssa kommunikoivaan kokonaisuuteen. Viimeinen rajoite, ladattava koodi, on vapaaehtoinen ja tarjoaa mahdollisuuden ohjelmakoodin lähettämiseen asiakkaalle.

Verkkoviestinnässä käytetty HTTP-protokolla on kehitetty REST-periaatteita noudattaen. Tämä kuvaa hyvin sitä, että kyseessä on yleisen tason tyyli, joka on toimiva vaihtoehto lähes jokaiseen tarkoitukseen. Tästä johtuen REST-tyylinen arkkitehtuuri häviää spesifiin tarkoitukseen optimoidulle ohjelmalle lähes aina. Se on kuitenkin hyvä lähtökohta aloittaa kehittämään myös tehokkuuskriittisiä sovelluksia, sillä se ohjaa helposti rinnakkaistettavaan ja suurille käyttäjämäärille skaalautuvaan lopputulokseen yleisesti tarkasteltuna. Yksinkertaisissa, luonnostaan tietoalkioita tarjoavissa sovelluksissa REST on erittäin toi-

miva vaihtoehto muun muassa välimuistin hyödyntämisen tarjoaman tehokkuushyödyn ansiosta. Esimerkkinä tästä on toteutettu esimerkkisovellus.

Suurin vaikutus REST-tyylin hyödyntämisellä on ohjelmiston kehittämiseen ja ylläpitoon. RESTin sisäistäneet kehittäjät voivat toteuttaa eri kokonaisuuksia samaan aikaan äärimmäisen helposti. He saavat rajapinnan toiminnallisuuden selville äärimmäisen vähäisellä dokumentoinnilla erityisesti tilattomuus-rajoitteen vuoksi. REST-tyylin www-sovelluspalveluiden luontiin luodut lukuisat työkalut helpottavat tätä entisestään. Monet RESTin mukaiseksi väitetyt sovellukset eivät kuitenkaan noudata RESTiä täysin, johtuen muun muassa abstraktin asian vaikeasta ymmärrettävyydestä käytännössä. Tämä kävi ilmi myös esimerkkisovellusta toteutettaessa; aikaa vei REST-tyylin ja teknologioiden opettelu, mutta uuden www-sovelluspalvelun tekeminen tai käyttäminen olisi nyt todella nopeaa ja vaivatonta. Turvallisuuteen REST ei käytännössä ota kantaa, vaan tietoturva jää kehittäjän vastuulle. Luotettavuutta REST parantaa muun muassa helpon vikatilanteista palautumisen, lokituksen ja staattisten resurssilinkkien avulla.

LÄHTEET

[1] *Maps Static API*. 2020. URL: https://developers.google.com/maps/documentation/maps-static/intro (viitattu 20. 04. 2020).

[2] Berlind, D. *About Programmable Web*. 2020. URL: https://www.programmableweb.com/about (viitattu 25. 05. 2020).

[3] Fielding, R. T., Taylor, R. N., Erenkrantz, J. R., Gorlick, M. M., Whitehead, J., Khare, R. ja Oreizy, P. Reflections on the REST architectural style and "principled design of the modern web architecture"(impact paper award). English. ACM, 2017, 4–14. ISBN: 9781-450351058.

[4] Richardson, L. *Act Three: The Maturity Heuristic*. 2008. URL: https://www.crummy.com/writing/speaking/2008-QCon/act3.html (viitattu 25. 08. 2020).

[5] Fielding, R. T. ja Taylor, R. N. *Architectural styles and the design of network-based software architectures*. Vol. 7. University of California, Irvine Doctoral dissertation, 2000.

[6] Haas, H. ja Brown, A. *Web Services Glossary*. 2004. URL: https://www.w3.org/TR/ws-gloss/ (viitattu 07. 09. 2020).

[7] TSK, S. *Erikoisalojen sanastojen ja sanakirjojen kokoelma*. 2012. URL: https://termipankki.fi/tepa/fi/ (viitattu 07. 09. 2020).

[8] Booth, D., Haas, H., McCabe, F., Newcomer, E., Champion, M., Ferris, C. ja Orchard, D. *Web Services Architecture*. 2004. URL: https://www.w3.org/TR/ws-arch/ (viitattu 07. 09. 2020).

[9] Wilde, E. *Link Relation Types for Web Services*. 2019. URL: https://tools.ietf.org/html/rfc8631 (viitattu 07. 09. 2020).

[10] Crockford, D. *The application/json Media Type for JavaScript Object Notation (JSON)*. 2006. URL: https://tools.ietf.org/html/rfc4627 (viitattu 07. 09. 2020).

[11] Nelson, B. J. REMOTE PROCEDURE CALL (1982).

[12] *Java Remote Method Invocation API*. 2017. URL: https://docs.oracle.com/javase/7/docs/technotes/guides/rmi/ (viitattu 09. 10. 2020).

[13] Marculescu, M. Introducing gRPC, a new open source HTTP/2 RPC Framework (2015). URL: https://developers.googleblog.com/2015/02/introducing-grpc-new-open-source-http2.html?m=1 (viitattu 10. 09. 2020).

[14] MacKenzie, C. M., Laskey, K., McCabe, F. ja Peter F Brown Rebekah Metz, B. A. H. Reference Model for Service Oriented Architecture 1.0 (2006). URL: http://docs.oasis-open.org/soa-rm/v1.0/ (viitattu 14. 09. 2020).

[15] *APIs for Selling Inventory on eBay*. URL: https://developer.ebay.com/products/sell (viitattu 29. 05. 2020).

[16] Koskimies, K. ja Mikkonen, T. *Ohjelmistoarkkitehtuurit*. Talentum, 2005.

[17] *ECMAScript 2020 Language Specification*. 2020. URL: https://www.ecma-inter%20national.org/ecma-262/11.0/index.html (viitattu 06. 09. 2020).

[18] *Node.js*. 2020. URL: https://www.nodejs.org/ (viitattu 16. 09. 2020).

[19] *Express*. 2017. URL: https://expressjs.com/ (viitattu 17. 08. 2020).

[20] *Getting started with OAS*. 2020. URL: https://swagger.io/solutions/getting-started-with-oas/ (viitattu 30. 08. 2020).

[21] *Github*. 2020. URL: https://github.com/ (viitattu 14. 09. 2020).

[22] *jsonwebtoken*. 2020. URL: https://www.npmjs.com/package/jsonwebtoken (viitattu 16. 09. 2020).

[23] Jones, M., Bradley, J. ja Sakimura, N. *JSON Web Token (JWT)*. 2015. URL: https://www.rfc-editor.org/info/rfc7519 (viitattu 08. 09. 2020).

[24] Roman, D., Kopecký, J., Vitvar, T., Domingue, J. ja Fensel, D. WSMO-Lite and hRESTS: Lightweight semantic annotations for Web services and RESTful APIs. *Journal of Web Semantics* 31 (2015), 39–58. ISSN: 1570-8268. DOI: https://doi.org/10.1016/j.websem.2014.11.006. URL: http://www.sciencedirect.com/science/article/pii/S1570826814001188.

[25] Richardson, L. *RESTful Web Services*. O'Reilly Media, 2007.

[26] Fielding, R. ja Reschke, J. *Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content*. 2014. URL: https://tools.ietf.org/html/rfc7231 (viitattu 21. 09. 2020).

[27] Jones, M. ja Hardt, D. *The OAuth 2.0 Authorization Framework: Bearer Token Usage*. 2012. URL: https://tools.ietf.org/html/rfc6750 (viitattu 11. 10. 2020).

[28] Gudgin, M., Hadley, M., Mendelsohn, N., Moreau, J.-J., Nielsen, H. F., Karmarkar, A. ja Lafon, Y. *SOAP Version 1.2 Part 1: Messaging Framework (Second Edition)*. 2007. URL: https://www.w3.org/TR/soap12/ (viitattu 20. 08. 2020).

[29] Hudak, P. Conception, evolution, and application of functional programming languages. *ACM Computing Surveys (CSUR)* 21.3 (1989), 359–411.

[30] Pearce, J. *Pipeline Architecture*. (Viitattu 22. 10. 2020).

[31] Klauzinski, P. ja Moore, J. *Mastering JavaScript Single Page Application Development*. Packt Publishing Ltd, 2016.

[32] Chin, R. S. ja Chanson, S. T. Distributed, Object-Based Programming Systems. 23.1 (maaliskuu 1991), 91–124. URL: https://doi.org/10.1145/103162.103165 (viitattu 27. 09. 2020).

[33] zur Muehlen, M., Nickerson, J. V. ja Swenson, K. D. Developing web services choreography standards—the case of REST vs. SOAP. *Decision Support Systems* 40.1 (2005). Web services and process management, 9–29. ISSN: 0167-9236. DOI: https://doi.org/10.1016/j.dss.2004.04.008. URL: http://www.sciencedirect.com/science/article/pii/S0167923604000612.