WWW-sovelluspalvelurajapinnat - REST

Samuli Anola-Pukkila

Tiivistelmä
Tutkielman päämääränä on luoda käytäntöpohjainen kirjallisuuskatsaus www-sovelluspalvelurajapintatoteutuksissa käytettävästä REST-arkkitehtuurimallista. Käytäntöpohjaisen näkökulman vahvistamiseksi tutkielma sisältää kirjoittajan luomia käytännön esimerkkejä, joiden tavoitteena on havainnollistaa käsiteltyjen aiheiden toimintaperiaatteita. Tutkielmassa tarkastellaan REST-arkkitehtuurimallin määritelmästä koituvia etuja sekä haittoja käymällä läpi REST:ssä määritellyt rajoitteet ja erittelemällä kunkin rajoitteen päämäärä sekä siitä aiheutuvat seuraukset kokonaisohjelmistoarkkitehtuuriin. Tutkielman loppupuolella käsitellään myös yleisellä tasolla REST-mallia mukailevan eli RESTful:n www-sovelluspalvelurajapinnan toteutusta. Tutkielmassa avataan REST-arkkitehtuurimallin lisäksi muita www-kehitykseen rinnastettavia aiheita, kuten www-sovellusrajapintoja ja HTTP-protokollaa.

Avainsanat: REST, HTTP, URI, rajapinta, www-sovelluspalvelu

1 Johdanto

WWW-pohjaisten ohjelmistojen suosion myötä automatisoitujen prosessien tarve kasvaa ja kehittyy [Mumbaikar and Padiya, 2013]. Web-sovelluksien automatisoinnin kulmakivenä toimii ohjelmistojen välisen vuorovaikutuksen mahdollistavat *www-sovelluspalvelut* (engl. web service).

Tässä tutkielmassa tutkitaan julkaisuhetkestään tasaisesti suosiotaan kasvattanutta, yhdeksi alan standardiksi www-sovelluspalveluiden toteutuksissa noussutta REST-arkkitehtuurimallia [Liu *et al.*, 2017; Fielding, 2000]. Suosiosta kielii esimerkiksi suurten tai muuten merkittävien organisaatioiden, kuten Googlen ja Wikipedian, tarjoamat julkiset *RESTful-www-rajapinnat* (engl. RESTful Web API) sekä aiheesta löytyvän kirjallisuuden määrä.

Kirjoittajan motiivina toimii yleisluontoinen kiinnostus web-teknologioita kohtaan, www-sovelluspalvelurajapintojen, -arkkitehtuurimallien sekä -protokollien tuntemuksen tuomat hyödyt www-sovellusta kehittävässä työympäristössä sekä REST-arkkitehtuurimallin tämänhetkinen suosio ja ajankohtaisuus.

Lukijalta *ei odoteta* laajaa teoriapohjaa tai käytännön kokemusta aiheesta. Tästä syystä tutkielmassa tullaan käsittelemään www-sovelluspalveluita

ja REST-arkkitehtuurimalliin liittyviä teknologioita sekä konsepteja perustasolta lähtien.

2 WWW-sovelluspalvelut

WWW-sovelluspalveluiden päämääränä on mahdollistaa ohjelmistojen välinen vuorovaikutus internetin välityksellä. Termiä web-sovelluspalvelu ei tule siis sekoittaa termin *web-sovellus* (engl. web application) kanssa; web-sovelluksen päämääränä on toimia ihmisen ja koneen välisessä vuorovaikutuksessa, kun taas web-sovelluspalvelun tehtävänä on toimia esimerkiksi kahden web-sovelluksen välillä.

WWW-sovelluspalvelu on pohjimmiltaan semanttisesti tarkkaan määritelty abstraktio joukosta erityyppisiä tapahtumia, joiden päämääränä on vaikuttaa ennalta määrättyihin resursseihin ennalta määrätyin tavoin [Sheng *et al.*, 2014]. Tämä tapahtumajoukko määritellään käyttämällä standardioitua ohjelmointikieltä. Vuorovaikutus joukkoon kuuluvien tapahtumien kanssa tapahtuu jonkin standardioidun internet-pohjaisen protokollan välityksellä.

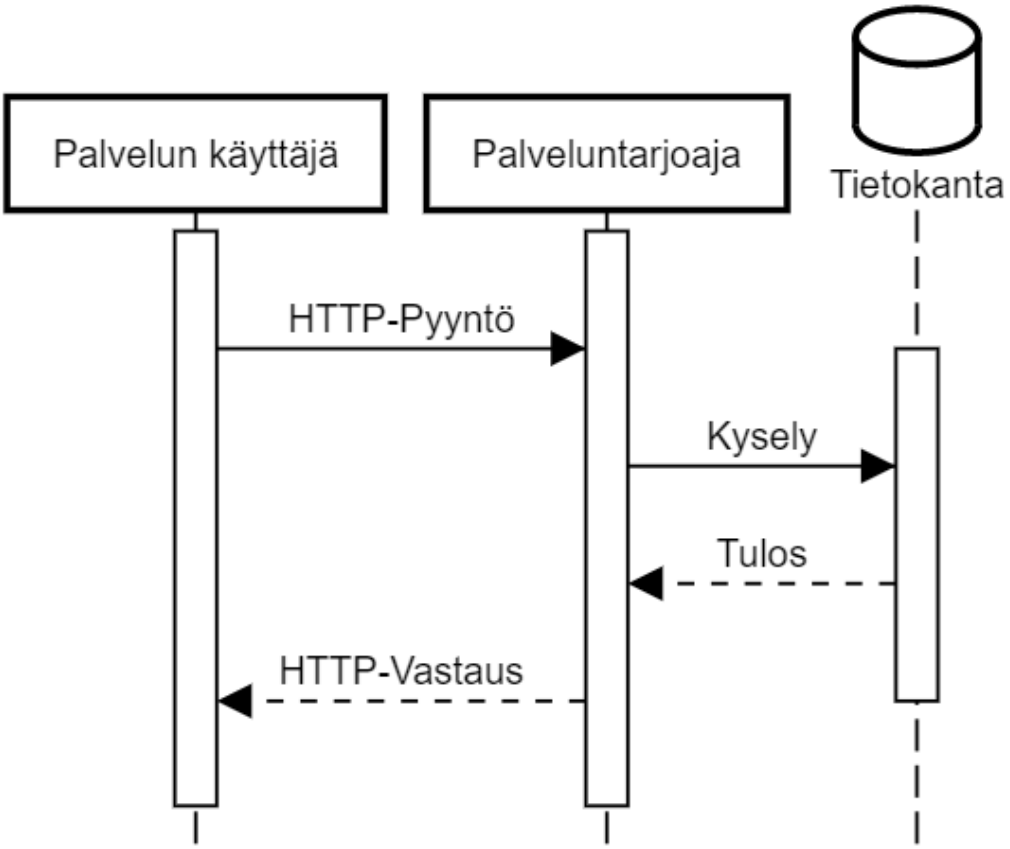
Standardioidun *tiedonsiirtoprotokollan* (engl. transfer protocol), esimerkiksi HTTP:n, käyttö johtaa siihen, että tapahtumia määrittelevä osapuoli ei määrittele tapahtumia käyttävän osapuolen ominaisuuksia. Tämä tarkoittaa sitä, että määritellyt tapahtumat ovat käytettävissä *aina*, ottamatta kummankaan osapuolen kohdalla huomioon esimerkiksi ohjelmointikieltä, palvelinohjelmistoa tai käyttöjärjestelmää. "Alustariippumattomuus" sekä hyvien suunnittelumenetelmien tuoma *löyhä sidonnaisuus* (engl. loose coupling) palveluja käyttäviin ohjelmistoihin tekee hyvin toteutetusta www-sovelluspalvelusta potentiaalisesti hyvin jatkokehitettävän ja pitkäikäisen kokonaisuuden. [Medjahed, 2004].

WWW-sovelluspalvelussa on kolme osapuolta: *palveluntarjoaja* (engl. service provider), *palvelun käyttäjä* (engl. service requester) ja *palveluhakemisto* (engl. service registry) [Sheng *et al.*, 2014]. Palveluntarjoaja tuottaa ja julkaisee palvelunsa palveluhakemistoon, josta palvelun käyttäjä löytää ja hyödyntää palvelun tarjoamia resursseja. Palveluhakemiston käyttö ei ole pakollista, mutta erityisesti julkisen palvelun löydettävyyden kannalta suositeltavaa.

2.1 Käyttö

Eräs yksinkertaisimmista www-sovelluspalveluiden käyttötapauksista on toimia palvelun käyttäjälle, eli esimerkiksi web-sovellukselle resursseja julkaisevana rajapinnan ja tietokannan yhdistelmänä (kuva 1). Tutkielmassa käytettyjen sekvenssikaavioiden merkintätapa on seuraava: yhtenäinen viiva merkit-

see *pyyntöä* (engl. request) tai tietokantakyselyä, katkoviiva *vastausta* (engl. response) tai tietokantakyselyn tulosta ja pystypalkki sekvenssin osapuolten aktiivisuuden tilaa.

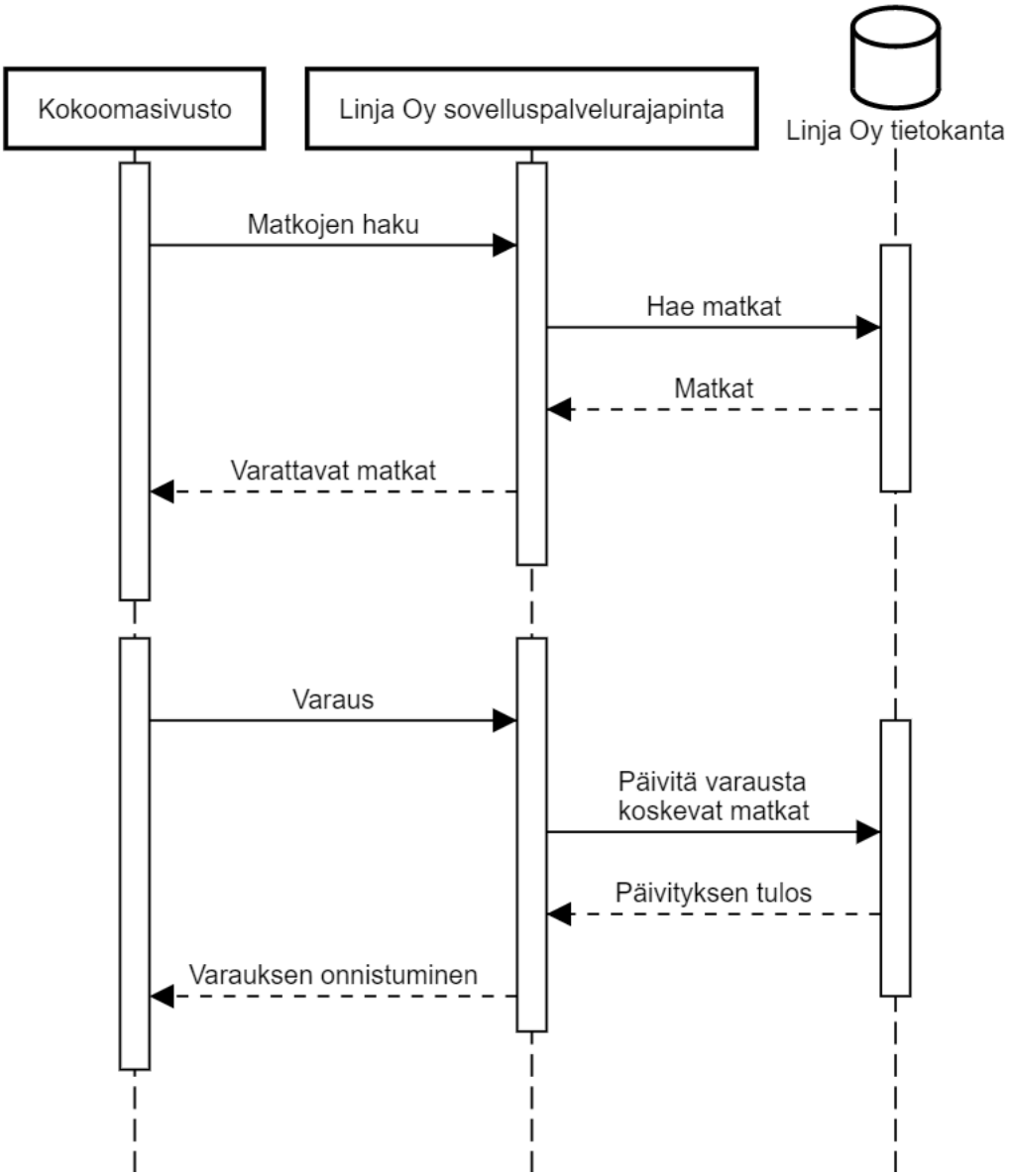


Kuva 1: Yksinkertaisen www-sovelluspalvelun toiminta sekvenssikaaviona.

Yksittäinen taho voi toimia sekä palveluntarjoajana, eli tietokannan ja www-sovelluspalvelun hallinnoijana, että palvelun käyttäjänä. Yleisemmin www-sovelluspalvelun päämääränä on kuitenkin tuoda erinäisten tahojen (esimerkiksi yritysten) sisäisiä prosesseja julkiseen tai rajattuun käyttöön internetin välityksellä [Sheng *et al.*, 2014].

2.2 Esimerkkipalvelu - Linja Oy

Kuvitteellisella linja-autoyhtiö *Linja Oy*:llä on verkkosivut, joiden kautta on mahdollista varata lippuja yhtiön tarjoamille matkoille. Alhaisten kävijämäärien vuoksi yhtiö päättää nostaa näkyvyyttään julkaisemalla lipunvarausaikataulunsa sivustolla, joka kokoaa eri linja-autoyhtiöiden aikatauluja yhteen myyden asiakkailleen yhtiöiden tarjoamia matkoja. Tästä sivustosta käytetään jatkossa termiä "*kokoomasivusto*".



Kuva 2: Linja Oy:n matkanvarausrajapinnan matkanvaraussekvenssi.

Yhteistyön mahdollistamiseksi Linja Oy:n täytyy antaa kokoomasivustolle reaaliaikaista tietoa varattavista matkoista sekä ottaa vastaan tietoa kokoomasivustolta varatuista matkoista. Näiden toimintojen automatisointia varten Linja Oy luo rajapinnan vuorovaikuttamaan kokoomasivustolle lähetetyn sekä sieltä saapuvan tiedon kanssa. Koska kokoomasivusto tekee yhteistyötä useampien linja-autoyhtiöiden kanssa, luodaan rajapinta kokoomasivuston esittämien standardien mukaan.

Kuvatuista prosesseista on luotu havainnollistava, esimerkinomainen sekvenssikaavio (kuva 2) on yksinkertaistettu versio todellisesta sekvenssistä;

pyyntöjen muoto on abstraktioitu puhekielelle, ja kaaviosta puuttuu esimerkin kannalta epäolennaisia elementtejä, kuten auktorisointi. Puuttuvia asioita käsitellään tarkemmin tutkielman myöhemmissä luvuissa. Sekvenssikaaviossa ei ole myöskään otettu huomioon esimerkiksi kokoomasivuston todennäköisesti käyttämää omaa rajapintaa, vaan oletetaan, että kokoomasivuston pyynnöt tulevat suoraan käyttöliittymästä.

Kuvan 2 kaaviossa kuvataan käyttötapaus, jossa käyttäjä hakee vapaita vuoroja kokoomasivustolta, jonka jälkeen hän suorittaa varauksen Linja Oy:n matkalle. Kuvattu "Matkojen haku" -pyyntö lähetetään kaikille kokoomasivuston kanssa yhteistyötä tekeville linja-autoyhtiöille, jolloin kokoomasivusto kokoaa vastauksina saaduista varattavissa olevista vuoroista koostuvan kokoelman ja esittää tämän käyttäjälle käyttöliittymässä. Kun käyttäjä päättää varata jonkin linja-autoyhtiön tarjoaman matkan, lähetetään tästä tieto kyseisen yhtiön rajapintaan. Tämän jälkeen yhtiön rajapinta päivittää yhtiön tietokannan tilaa halutulla tavalla ja palauttaa kokoomasivustolle tiedon siitä, onnistuiko varaus odotetusti. Tämä mahdollistaa sen, että virhetilanteessa kokoomasivusto kykenee välittämään tiedon epäonnistuneesta varauksesta käyttäjälle.

Kuvassa 2 esitetty www-sovelluspalvelurajapinnan ja tietokannan kokonaisuus on siis kuvitteellisen Linja Oy:n tarjoama www-sovelluspalvelu. Kuvattu palvelusekvenssi mahdollistaa matkojen automatisoidun, reaaliaikaisen esityksen kokoomasivustolla. Koska palvelu suoritetaan standardioiduilla pyynnöillä käyttäen standardioitua tiedonvälitysprotokollaa, Linja Oy:n tietojen integrointi kokoomasivuston palveluihin ei oikein toteutettuna luo tarvetta muokata aiemmin käytössä olevia järjestelmiä, palvelimia, tietokantoja tai rajapintoja.

3 WWW-sovelluspalvelurajapinnat

WWW-sovelluspalvelurajapinta on www-sovelluspalvelun julkaistu esitys palvelun tarjoamista tapahtumista. Siinä määritellään julkaistavien tapahtumametodien käyttämät *viestityypit* (engl. message types) ja *viestinvaihdon mallit* (engl. message exchange patterns) [Brown and Haas, 2004].

WWW-sovelluspalvelurajapinta on siis jokin semanttisesti yhteensopiva kokonaisuus erityyppisiä toimintoja. Työpöytäsovellusrajapintoihin tottuneelle konsepti itsessään on tuttu: kirjataan ylös joukko metodeja, joista jokaisella on jokin tehtävä. Tämän tehtävän suoritukseen metodeille voidaan antaa parametreja, ja ne voivat palauttaa paluuarvoja, kuten esimerkiksi tiedon tehtävän onnistumisesta tai metodin avulla haetun muuttujan arvon.

WWW-sovelluspalvelurajapintoihin annettavat parametrit annetaan rajapintoihin tehdyissä, jotakin internet-pohjaista tiedonsiirtoprotokollaa käyttävissä pyynnöissä, ja paluuarvot palautetaan vastauksissa. Tässä tutkielmassa keskitytään HTTP-protokollan käyttöön, joten tästä lähtien tullaan käyttämään termejä *HTTP-pyyntö* (engl. HTTP-request) ja *HTTP-vastaus* (engl. HTTP-response). Tiedonsiirtoprotokollan käyttöön liittyvissä kohdissa viitataan aina HTTP:n käyttöön, jollei toisin mainita. HTTP-protokollaa käsitellään tarkemmin luvussa 4.

3.1 Esimerkkirajapinta - Linja Oy

Aiemmin käsittelyssä olleen Linja Oy:n ja kokoomasivuston integraatio tapahtui käytännössä toteuttamalla www-sovelluspalvelurajapinta Linja Oy:n tietokannan, ja kokoomasivuston välille. Yksinkertaistetussa esimerkkitapauksessa kokoomasivuston täytyi suorittaa Linja Oy:n rajapinnassa seuraavia toimintoja:

- noutaa tieto varattavista matkoista
- tallentaa tieto varatusta matkasta.

Jos rajapintaa halutaan kehittää eteenpäin, muokataan Linja Oy:n rajapintaa niin, että se tukee aiemmin kuvattujen toimintojen lisäksi myös seuraavia toimintoja:

- mahdollisuus noutaa tieto varattavista matkoista tietyltä aikaväliltä
- mahdollisuus noutaa tietyn matkan tiedot
- mahdollisuus päivittää varauksen tietoja
- mahdollisuus perua varaus.

Kuvataan nyt kyseisen rajapinnan metodijoukkoa pseudokoodiesityksellä (esimerkkikoodi 1).

```
getTrip(tripId);
getAvailableTrips();
getAvailableTrips(startDate, endDate);
createReservation(reservationInfo);
updateReservation(reservationId, reservationInfo);
deleteReservation(reservationId);
```

Esimerkkikoodi 1: Linja Oy rajapinta.

Linja Oy voisi toteuttaa kyseisen rajapinnan sellaisenaan haluamallaan ohjelmointikielellä. Oleellisin ero työpöytärajapintatoteutukseen on se, että

kyseisiä metodeja täytyy pystyä kutsumaan internetin välityksellä. Tämä tapahtuu erillisen tiedonsiirtoprotokollan avulla.

4 HTTP

HTTP-protokollaa käsitellään tutkielmassa näennäisen laajasti. Tämä perustellaan HTTP-protokollan ja sen liitännäiskonseptien (esimerkiksi URI ja HTTP-metodien käyttö) olevan *kirjoittajan mielestä* sujuvan REST-www-sovelluspalvelurajapintasuunnittelun sekä -toteutuksen kulmakiviä. Näiden perusteellinen opiskelu saattaa joissain tapauksissa säästää ensimmäistä RESTful web-rajapintaansa kehittävän ohjelmoijan useiden tuntien refaktoroinnilta.

HTTP on sovellustason protokolla hajautetuille, yhdistetyille, hypermediainformaatiojärjestelmille. Se on geneerinen, tilaton protokolla, jota käytetään esimerkiksi tiedonvälitykseen ja hajautettuun hallinnointiin hyödyntäen HTTP-pyyntöjen metodeja, virhekoodeja ja otsikoita. [Fielding and Reschke, 2014a]. Sovellustasolla viitataan *TCP/IP-mallin* ylimpään kerrokseen [Parziale *et al.*, 2006; Braden, 1989a,b].

Uusin käytössä oleva HTTP-versio on nimeltään *HTTP/2*, jonka erot vanhempaan HTTP/1.1 -versioon liittyvät pääasiassa suorituskykyparannuksiin, ja yksi HTTP/2:n ensisijaisista päämääristä onkin yhteensopivuus (engl. *backwards compatibility*) edeltävien versioiden (HTTP/1.x) kanssa [Belshe *et al.*, 2015]. Tässä tutkielmassa ei keskitytä tarkemmin HTTP-protokollan versioiden eroihin, tai oteta kantaa HTTP/2:n käyttöönoton kannattavuuteen. On kuitenkin huomionarvoista tutkielmassa käsiteltyjen asioiden pätevän kaikilla yleisesti käytössä olevilla protokollan versioilla (HTTP/2, sekä HTTP/1.x).

4.1 HTTP-viestien rakenne

Koneellinen kommunikaatio HTTP:n avulla koostuu HTTP-viesteistä. HTTP-viesti voi olla joko johonkin resurssiin kohdistettu HTTP-pyyntö tai kyseisestä resurssista tuleva HTTP-vastaus. Nämä viestit koostuvat *aloitusrivistä* (engl. start-line), $N \geq 0$ kappaleesta *otsikkorivejä* (engl. request-header) sekä mahdollisesta *viestirungosta* (engl. message-body). Syntaktisesti ainoa ero HTTP-pyynnöllä ja HTTP-vastauksella on aloitusrivi, joka on joko *pyyntörivi* (engl. request-line) tai *tilarivi* (engl. status-line). [Fielding and Reschke, 2014a].

```
POST /post HTTP/1.1
cache-control: no-cache
content-type: text/plain
user-agent: PostmanRuntime/6.4.1
host: postman-echo.com
content-length: 256

Duis posuere augue vel cursus pharetra. In luctus a ex nec pretium.
```

Esimerkkikoodi 2: HTTP-pyyntö.

```
HTTP/1.1 200
status: 200
content-encoding: gzip
content-type: application/json; charset=utf-8
cache-control: no-cache
date: Mon, 27 Nov 2017 16:26:24 GMT
server: nginx
vary: X-HTTP-Method-Override, Accept-Encoding
content-length: 524
connection: keep-alive
{
  "args":{},
  "data":"Duis posuere augue ve.."
}
```

Esimerkkikoodi 3: HTTP-vastaus.

Tarkastellaan HTTP-viestien rakennetta esimerkkien (esimerkkikoodit 2 ja 3) avulla. Kummassakin esimerkkikoodissa ensimmäisenä oleva rivi on aiemmin mainittu aloitusrivi. HTTP-pyynnön kohdalla siihen on merkitty käytetty metodi (POST), resurssin sijainti palvelimella (/post) sekä käytetty HTTP-protokollan versio (HTTP/1.1). HTTP-vastauksen kohdalla aloitusriville on merkitty HTTP-protokollan versio sekä pyynnön käsittelyn tuottama status (200). Aloitusrivin jälkeen tulevat viestien mukana kulkevat otsikkorivit. Näistä riveistä käy ilmi esimerkiksi resurssia ajavan palvelimen juuriosoite (host: postman-echo.com). Otsikkorivien jälkeen tulee otsikkorivien päättymisestä ilmaiseva tyhjä rivi sekä viestirungot.

4.2 Metodit

HTTP-pyynnöt lähetetään aina käyttäen jotakin HTTP-protokollaan määriteltyä, pyynnön päämäärää kuvaavaa metodia. Tämä päämäärä voi olla esimerkiksi tiedonhaku tai -tallennus. Nämä metodit jakautuvat kolmeen eri kategoriaan, *turvallisiin* (engl. safe methods), *idempotentteihin* (engl. idempotent methods) sekä muihin metodeihin (taulukko 1 ja esimerkkikoodi 4). Kappaleessa käsitellään vain tutkielman näkökulmasta oleellisimpia metodeja.

Metodi	**Turvallinen**	**Idempotentti**
OPTIONS	Kyllä	Kyllä
GET	Kyllä	Kyllä
HEAD	Kyllä	Kyllä
POST	Ei	Ei
PUT	Ei	Kyllä
DELETE	Ei	Kyllä
PATCH	Ei	Ei

Taulukko 1: Yleisten HTTP-metodien turvallisuus ja idempotenttisuus.

```
class Example {

  int global = 0

  safeAndIdempodentMethod() {
    return global
  }

  idempotentMethod(value) {
    global = value
  }

  notSafeNorIdempotentMethod() {
    global++
  }
}
```

Esimerkkikoodi 4: Havainnollistava pseudokoodiesitys esimerkkiluokasta.

Turvallinen metodi on metodi, jolla ei voida muuttaa kohderesursseja. Tämä tarkoittaa käytännössä metodia, joka on semanttisesti "vain luku"

-tilassa. [Fielding and Reschke, 2014b]. Yleisesti turvallisia metodeja käytetään tiedonhakuun (esimerkkikoodi 4, *safeAndIdempodentMethod*-metodi).

Idempotentilla metodilla voidaan muuttaa resursseja. Idempotentilla metodilla on kuitenkin rajoituksena $N > 0$ identtisen pyynnön lopputulosten identtisyys. Tämä tarkoittaa sitä, että yhden pyynnön toteutuessa kyseistä pyyntöä seuraavat identtiset pyynnöt johtavat aina samaan lopputulokseen kuin alkuperäinen pyyntö. Idempotentin metodin käyttö jättää siis järjestelmän aina samaan tilaan ensimmäistä pyyntöä seuraavien identtisten pyyntöjen määrästä riippumatta. [Fielding and Reschke, 2014b]. Yleisesti idempotentteja metodeja käytetään tiedon hakuun, muokkaamiseen, tai poistamiseen (esimerkkikoodi 4, *idempotentMethod*-metodi).

Muut metodit ovat metodeja, jotka eivät täytä turvallisen tai idempotentin metodin kriteerejä. Tällaisia metodeja käytetään esimerkiksi uuden tiedon luontiin tai tiedon muuttamiseen esimerkkikoodi 4:n *notSafeNorIdempotentMethod*-nimisessä metodissa kuvatulla inkrementoivalla tavalla.

GET-metodilla haetaan haluttuja resursseja jossakin tarjolla olevassa esitysmuodossa. Se on HTTP:n ensisijainen tiedonhakuun tarkoitettu metodi. [Fielding and Reschke, 2014b].

HEAD-metodi on muuten identtinen GET-metodin kanssa, mutta HEAD-pyynnön HTTP-vastauksessa *ei saa* olla body-osaa eli viestirunkoa [Fielding and Reschke, 2014b]. HEAD-metodia voidaan käyttää esimerkiksi haettaessa resurssiin liittyvää metatietoa.

POST-metodia käytetään pyyntönä, jonka päämääränä on käskeä kohderesurssia prosessoimaan pyynnön mukana tulevaa tietoa resurssin omien semanttisten sääntöjen mukaisesti. Yleisesti POST-metodia käytetään tiedon inkrementoivaan lisäämiseen (esimerkiksi viestin lähettäminen keskustelualueelle), tiedon lisäämiseen jo olemassa olevaan, kohderesurssissa sijaitsevaan tietorakenteeseen tai tiedon kuljettamiseen tietojenkäsittelyprosessia varten (esimerkiksi HTML form -elementin käyttö). [Fielding and Reschke, 2014b].

PUT-metodia käytetään tiedon idempotenttiin lisäykseen tai muokkaukseen. Tämä tarkoittaa sitä, että PUT-pyyntö joko luo uuden tai korvaa olemassaolevan resurssin. [Fielding and Reschke, 2014b]. Sekä POST:ia, että PUT:ia voidaan käyttää tiedon luomiseen tai päivittämiseen, eikä niiden välillä ole suurta semanttista eroa. Pääasiallinen ero kehittäjän näkökulmasta on toiminnon idempotenttisuus. Jos päämääränä on idempotentti tietojenkäsittelytapahtuma, käytetään PUT:ia ja muutoin käytetään tilanteen mukaan joko POST:ia tai PATCH:ia.

PATCH-metodia käytetään olemassa olevan resurssin osittaiseen muokkaukseen. Tämä eroaa PUT:sta sillä, että PATCH ei ole idempotentti metodi. Käytettäessä PUT-metodia resurssin muokkaukseen, korvataan kohdere-

surssi kokonaan, kun taas PATCH muokkaa resurssia vain osittain. Joskus PUT:n käyttö PATCH-metodin sijaan on suositeltavaa myös näennäisesti osittaisessa muokkauksessa, esimerkiksi silloin, kun PATCH-metodin toteutukseen tarvittava tietomäärä on huomattavasti suurempi kuin kokonaan uuden resurssin luominen PUT:lla. POST-metodia käytetään pääasiassa resurssien luontiin, joten PATCH:ia voi yleisesti käyttää tilanteissa, joissa POST:ia käytettäisiin resurssin muokkaukseen. [Dusseault and Snell, 2010].

DELETE-metodia käytetään kohderesurssin poistoon. Fielding ja Reschke [2014b] vertaavat DELETE-metodia UNIX-käyttöjärjestelmän *rm*-komentoon tuoden esiin DELETE-metodin päämäärän poistaa nimenomaan tietyssä osoitteessa sijaitseva resurssi jättäen aiemmat informaatioassosiaatiot huomiotta.

OPTIONS-metodilla pyydetään tietoja kohderesurssiin kohdistuvista tietojenkäsittelymahdollisuuksista [Fielding and Reschke, 2014b]. Näin voidaan hakea esimerkiksi tarkat ohjeet jonkin resurssin luontia varten.

4.3 URI

Uniform Resource Identifier, eli URI, on kompakti merkkijono, joka yksilöi ja paikallistaa joko abstraktin tai fyysisen resurssin. URI voidaan tarkemmin luokitella kuvaamaan joko resurssin sijaintia (URL), nimeä (URN) tai kumpaakin edeltävistä. HTTP:ta käytettäessä pyritään paikantamaan kohderesurssit tietoverkossa, joten metodien kohderesursseista puhuttaessa voidaan yleisesti käyttää myös termiä URL. URI:n oletussyntaksi koostuu neljästä pääkomponentista: skeemasta, auktoriteetista, polusta, ja kyselystä (esimerkkikoodi 5). [Berners-Lee *et al.*, 2005].

```
<scheme>://<authority><path>?<query>
```

Esimerkkikoodi 5: Geneerisen URI syntaksin neljä pääkomponenttia.

Skeema viittaa URI:n syntaksia määrittelevään, rekisteröityyn protokollaan, jollainen myös HTTP on [Thaler *et al.*, 2015]. Auktoriteetti koostuu valinnaisesta autentikaatio-osasta, johon kuuluvat käyttäjänimi ja salasana, sekä *isäntäosasta* (engl. host), joka määrittelee HTTP:ta käytettäessä joko rekisteröidyn verkkosivun nimen tai IP-osoitteen eli resurssin juuren. Polku määrittelee resurssin sijainnin suhteessa aiemmin määriteltyyn resurssin juureen. Kyselyn avulla voidaan rajata resurssia tai vaikuttaa sen esitystapaan. [Berners-Lee *et al.*, 2005].

HTTP:n oma URI-syntaksin määritelmä (esimerkkikoodi 6) [Fielding and

Reschke, 2014a, 17-18] mukailee yleistä URI:n määritelmää poislukien HTTP-kohtaiset tarkennukset.

```
"http:" "//" authority path-abempty [ "?" query ][ "#" fragment ]
```

Esimerkkikoodi 6: HTTP:n URI-syntaksi.

HTTP-URI alkaa protokollan määrittelyllä, eli merkkijonolla "*http:*". Tämän jälkeen merkitään *auktoriteettiosa* (authority), jota edeltää kaksinkertainen kenoviiva "*//*", ja joka päättyy joko kenoviivaan "*/*", kysymysmerkkiin "*?*", numeromerkkiin "*#*" tai URI:n loppuun. Auktoriteettiosa koostuu valinnaisesta autentikaatio-osasta, isäntäosasta (palvelimen juuriosoite), sekä portista (oletuksena 80). Komponentti *path-abempty* viittaa yleisessä URI-syntaksin määrittelyssä määriteltyyn *polku*-osaan, joka alkaa kenoviivalla "*/*" tai on tyhjä. *Kysely* (query) viittaa URI-kyselyyn ja *kappale* (fragment) viittaa tiettyy resurssin osaan, esimerkiksi artikkelin alaotsikkoon. [Berners-Lee *et al.*, 2005, 17-25].

```
http://10.0.0.30:8080/article?title=http&date=17-06-18#basics
```

Esimerkkikoodi 7: HTTP:n URI-syntaksin mukainen URI.

5 REST

REST, eli *representational state transfer*, on web-sovellusarkkitehtuurissa laajalti käytössä oleva ohjelmistoarkkitehtuurimalli. Tässä tutkielmassa REST-arkkitehtuurimallia käsitellään vahvasti www-sovelluspalvelurajapintojen kautta, mutta mallin alkuperäinen kuvaus ei suoranaisesti sido REST:iä rajapintoihin, vaan mallin esiintuonut Fielding [2000] kuvaa sitä "*arkkitehtuaaliseksi tyyliksi hajautetuille hypermediajärjestelmille*".

Nykyaikaisessa www-sovellusarkkitehtuurissa käytettynä REST rakentuu URI:lla tunnistettavien *esitysmuotoisten* (engl. representational) tietyssä *tilassa* (engl. state) olevien resurssien HTTP-protokollalla tehtyjen *siirtojen* (engl. transfer) ympärille. Ne koostuvat erinäisistä arkkitehtuaalisista rajoitteista, joiden pyrkimyksenä on rakentaa skaalautuvia, uudelleenkäytettäviä, löyhästi sidonnaisia, korkeasuorituskykyisiä ja matalalla viiveellä toimivia rajapintoja [Mumbaikar and Padiya, 2013; Rodriguez, 2008; Pautasso, 2014].

Tutkielman aiemmissa luvuissa on käytetty yleistajuista sanaa resurssi kuvaamaan tietoa tai tietolähdettä. REST:ssä resurssi määritellään erikseen:

resurssi on mitä tahansa nimettävissä olevaa tietoa. Tämä tieto voi olla esimerkiksi dokumentti, kuva, merkkijono, kokoelma muita resursseja tai täysin ei-virtuaalinen objekti (kuten ihminen).

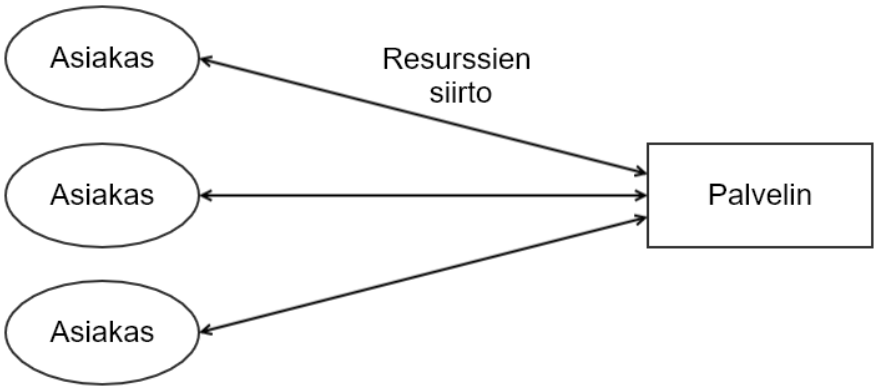
REST-resurssi ei kuitenkaan viittaa itse tietoalkioon, vaan se on viite vastaavista tietoalkioista koostuvaan tietojoukkoon. Tällöin resurssi ei käytännössä viittaa alkioon tai alkiojoukkoon, joka jonakin tiettynä ajankohtana edustaa viitattavan tietojoukon tilaa. Kuitenkin se samalla viittaa tietojoukon, eli tietoalkion arvoon. [Fielding, 2000; Vinoski, 2008].

Tutkielman seuraavissa luvuissa käsitellään REST:n määritteleviä arkkitehtuaalisia rajoitteita sekä tarkastellaan REST-arkkitehtuurimallilla toteutettavaa www-sovelluspalvelurajapintaa käytännön tasolla.

6 REST - Arkkitehtuaaliset rajoitteet

REST-rajapinta-arkkitehtuurimallin rajoitteet rakentuvat ns. *tyhjän tyylin* (engl. null style) päälle. Tyhjä tyyli viittaa järjestelmään, joka ei sisällä rajoitteita järjestelmän eri komponenttien välillä. [Fielding, 2000, 77].

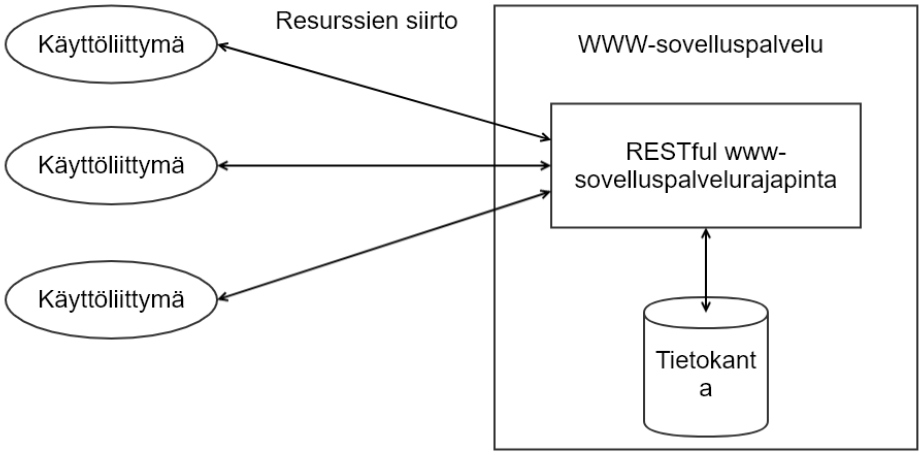
6.1 Asiakas-palvelin -arkkitehtuuri



Kuva 3: Asiakas-palvelin -arkkitehtuurimalli.

Asiakas-palvelin -arkkitehtuurin (engl. client-server architecture) (kuvat 3 ja 4) päämääränä on erottaa asiakas palvelimesta. Käytännössä tämä tarkoittaa usein käyttöliittymän erottamista kohderesursseja hallinnoivasta osapuolesta eli esimerkiksi www-sovelluspalvelurajapinnasta. Näiden osapuolten erottelemisella pyritään parantamaan käyttöliittymän siirrettävyyttä eri alusto-

jen välillä edistäen skaalautuvuutta yksinkertaistamalla palvelinkomponenttien toimintaa. Alkuperäisessä määritelmässä mahdollisesti merkittävimpänä seikkana asiakkaan ja palvelimen erotuksessa pidettiin kykyä kehittää kumpaakin osapuolta itsenäisesti. [Fielding, 2000, 78].



Kuva 4: Asiakas-palvelin -arkkitehtuurimalli www-sovelluspalveluna.

6.2 Tilattomuus

Tilattomuus (engl. statelessness) lisää aiemmin mainittuun asiakas-palvelin -arkkitehtuuriin seuraavan rajoitteen: kommunikaatio näiden kahden entiteetin välillä tulee olla luonteeltaan tilatonta [Fielding, 2000, 78].

Tilattomuus merkitsee jokaisen asiakkaalta palvelimelle lähetetyn pyynnön täytyvän sisältää kaikki tarvittava tieto pyynnön jäsentelyä varten sekä asiakasosapuolen kykenemättömyyttä hyväksikäyttää palvelimelle tallentunutta kontekstuaalista tietoa. Palvelimen ja asiakkaan välille muodostuvaan *istuntoon* (engl. session) liittyvä tieto, kuten käyttäjätiedot, tallennetaan siis aina asiakaspuolelle, ja jokaisen osapuolten välisen vuorovaikutustapahtuman jälkeen asiakas ja palvelin eivät jaa yhteisiä tilatietoja. [Fielding, 2000; Pautasso, 2014].

Tämä rajoite varmistaa sen, että resurssiin kohdistuvat pyynnöt eivät ole riippuvaisia toisistaan [Pautasso, 2014, 33]. Se edistää näkyvyyttä, luotettavuutta sekä skaalautuvuutta. Näkyvyys paranee, sillä järjestelmää monitoroiva osapuoli kykenee havainnoimaan asiakkaan lähettämän pyynnön luonteen tarkastelemalla vain yhtä pyyntötietuetta. Luotettavuus paranee, sillä pyyntöjen tilattomuus helpottaa palautumista osittaisista virhetilanteista

[Fielding, 2000, 79: Waldo *et al.* 1996]. Skaalautuvuus paranee, sillä potentiaalisesti tarpeellisten tilatietojen puuttuessa palvelimella on mahdollisuus vapauttaa resurssejaan nopeasti. Lisäksi tilattomuus yksinkertaistaa palvelintoteutuksen rakennetta, joka helpottaa palvelinohjelmiston täytäntöönpanoa.

Tilattomuudella on myös haittapuolia. Näihin lukeutuu verkkoliikenteen mahdollinen kuormittuminen asiakkaan ja palvelimen välisen kontekstin ollessa mukana jokaisessa palvelimelle tehdyssä pyynnössä, joka nostaa *toistuvan tiedon välityksen* (engl. per-interaction overhead) määrää järjestelmässä. Tämänkaltaiselta tarpeettomalta tiedonvälitykseltä ei voida välttyä, jos osapuolien välistä kontekstia ei saa säilyttää palvelimella. [Fielding, 2000, 79].

Tilan tallentuminen asiakkaan puolelle tarkoittaa myös sitä, ettei palvelinohjelmisto tai sen hallinnoija kykene varmistamaan palvelinohjelmiston tarjoamien resurssien tarkoituksenmukaista käyttöä. [Fielding, 2000, 79]. Tämä voi vaarantaa tiedon eheyden esimerkiksi tapauksessa, jossa palvelinohjelmiston julkaisemat resurssit ovat muuttuneet, mutta logiikkavirheen vuoksi asiakasohjelma ei ole pyytänyt päivitettyjä tietoja palvelimelta näyttäen käyttöliittymässään virheellistä tietoa.

6.3 Välimuisti

HTTP-välimuisti (engl. HTTP cache) on paikallinen varasto HTTP-pyyntöjen vastauksille ja välimuistin toimintoja hallinnoivalle osajärjestelmälle. Välimuisti varastoi *välimuistikelvollisia* (engl. cacheable) HTTP-vastauksia vähentääkseen vastausaikoja ja verkon kaistanleveyden käyttöä tulevilla vastaavilla HTTP-pyynnöillä. [Fielding *et al.*, 2014, 4].

Välimuistirajoite pyrkiikin kompensoimaan tilattomuusrajoitteesta aiheutuvia mahdollisia suorituskykyhaittoja. Välimuistirajoitteen mukaan palvelimen antaman vastauksen sisältämän tiedon täytyy olla merkitty joko välimuistikelvolliseksi tai välimuistikelvottomaksi. Jos vastaus on välimuistikelvollinen, asiakasohjelma saa oikeuden käyttää jatkossa vastauksen sisältämää tietoa vastaaviin pyyntöihin. [Fielding, 2000, 79]. Välimuistikelvollisuus voidaan määritellä HTTP-viestin otsikossa *cache-control* (esimerkkikoodi 2).

Tiedon välimuistiin tallentamisen etuna on joidenkin vuorovaikutustapahtumien osittainen tai täysinäinen poisto tapahtumaketjusta. Tämä parantaa tehokkuutta, skaalautuvuutta sekä käyttäjän havainnoimaa suorituskykyä laskemalla keskiarvoista viivettä tapahtumaketjujen välillä.

Välimuistiin tallentamista täytyy kuitenkin käyttää harkiten, sillä se saattaa vähentää tiedon luotettavuutta. Ongelma ilmenee, kun vanha, välimuistiin tallennettu tieto eroaa merkittävästi suoraan palvelimelle lähetetyn pyyn-

nön palauttamasta tiedosta. [Fielding, 2000, 80].

6.4 Rajapinnan yhdenmukaisuus

Fielding [2000] kuvailee rajapinnan yhdenmukaisuuden olevan keskeisin ero REST:n ja muiden verkkopohjaisten arkkitehtuurimallien välillä. Järjestelmän komponentit yhdistetään toisiinsa yhdenmukaisen rajapinnan kautta, jolloin kaikkien järjestelmässä käsiteltävien resurssien on kuljettava kyseisen rajapinnan läpi. Tämä rajapinta tarjoaa kokoelman pieniä, geneerisiä, funktionaalisesti riittäviä metodeja tukemaan kaikkia mahdollisia vuorovaikutustapahtumia resurssien välillä. [Pautasso, 2014, 33].

Yhdenmukaisen rajapinnan käyttö yksinkertaistaa järjestelmäarkkitehtuuria kokonaisuudessaan, ja parantaa vuorovaikutustapahtumien näkyvyyttä niiden kulkiessa aina saman rajapinnan kautta. Yhdenmukainen rajapinta helpottaa myös komponenttien itsenäistä kehitystä asiakas-palvelin -arkkitehtuurin pohjalta rakennetussa järjestelmässä. [Fielding, 2000, 81].

Yhdenmukaisen rajapinnan huonoksi puoleksi luetaan tehokkuuden heikentyminen eli suorituskyvyn lasku. Tämä johtuu käsiteltävän tiedon standardoidusta muodosta, joka ei ole välttämättä optimaalisin jokaiselle rajapintaa käyttävälle sovellustoteutukselle. [Fielding, 2000, 82].

WWW-sovelluspalveluiden ja HTTP-protokollan näkökulmasta aiemmin kuvattu yhdenmukainen rajapinta koostuu HTTP-metodeista, jotka voidaan kohdentaa käsittelemään tiettyä resurssia yksilöimällä kyseinen resurssi URI:lla [Pautasso, 2014, 33].

REST määrittelee yhdenmukaiselle rajapinnalle lisärajoitteita, joiden mukaan kyseinen komponentti tulee toteuttaa: *resurssien tunnistus* (engl. identification of resources), *resurssien manipulointi esitysten kautta* (engl. resource manipulation through representations), *viestien itseselitteisyys* (engl. self-descriptive message) ja *hypermedia sovelluksen tilan ajurina* (engl. hypermedia as the engine of application state, tai *HATEOAS*) [Fielding, 2000, 82].

Resurssien tunnistus www-pohjaisissa RESTful-rajapinnoissa tapahtuu URI:n avulla [Pautasso, 2009, 854]. REST-malli kannattaa intuitiivisten ja helppolukuisten URI:en käyttöä [Rodriguez, 2008; Pautasso *et al.*, 2008]. REST-mallin mukaisia URI:ja käsitellään tarkemmin kohdassa 7.1.

Esitys tarkoittaa REST:n kontekstissa mitä tahansa hyödyllistä informaatiota jonkin resurssin tilasta. Resurssilla voi olla useita eri esityksiä tai esitystapoja. [Feng *et al.*, 2009]. Nämä esitystavat ovat käytännössä resurssien *mediatyyppisiä* (engl. media type, aiemmin MIME-tyyppi, esimerkiksi *text/plain* tai *image/jpeg*) esityksiä sekä mahdollisia metatietoja kyseisistä

esityksistä [Freed and Borenstein, 1996; Alarcón and Wilde, 2010]. Mediatyyppi eli resurssin esitysmuoto määritellään HTTP-viestin otsikossa *content-type* (esimerkkikoodit 2 ja 3).

Itseselitteisellä viestillä tarkoitetaan sitä, että jokainen viesti sisältää kaikki tarvittavat tiedot ja metatiedot halutun tapahtuman tarkkaan toteutukseen. Tällaisia tietoja ovat esimerkiksi aiemmin mainitun mediatyypin tai jonkin resurssin, kuten artikkelin esityskielen määrittely. [Pautasso, 2014, 33].

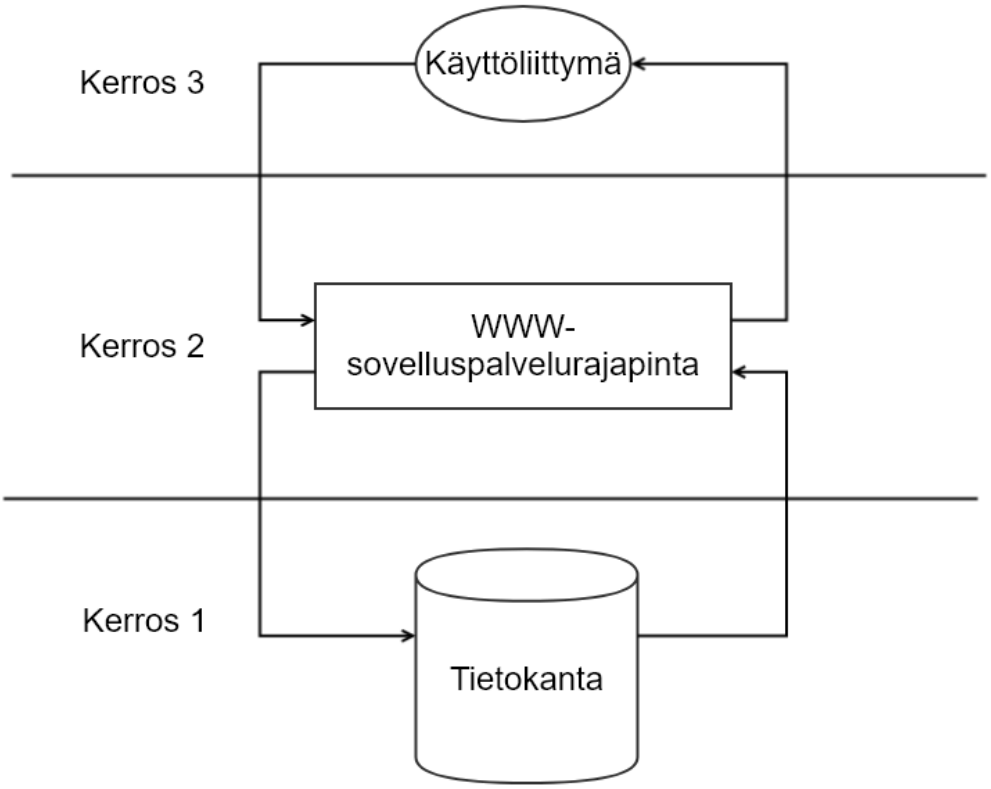
Hypermedia sovelluksen tilan ajurina rajoite pyrkii siihen, että RESTful API:n resurssien yhteyksiä sekä mahdollisia tilanmuutoksia mallinnettaisiin eräänlaisella hypermediasta koostuvalla viitteiden verkolla. Tämä mahdollistaa sen, ettei REST-palvelu toimi pelkkänä *päätepisteiden* (engl. endpoint) kokoelmana, vaan se voi tarjota pienen määrän URI:lla kohdennettuja resursseja, jotka toimivat *sisääntulopisteinä* (engl. entry-point) suuremmalle palvelukokonaisuudelle. [Alarcon *et al.*, 2010].

Hypermedia auttaa siis hajautetussa resurssien löytämisessä (engl. decentralized resource discovery), dynaamisessa löytämisessä (engl. dynamic discovery) ja vuorovaikutusprotokollien määrittelyssä palveluiden välillä [Pautasso, 2014].

Hypermedian käyttöä ei kuitenkaan aina huomioida REST-mallia mukailevan rajapinnan toteutuksissa. Fielding [2008], Vinoski [2008] ja Pautasso [2014] nostavat esiin ongelman hypermedian ja aiemmin kuvatun palvelukokonaisuuden verkotuksen vähäisestä käytöstä niin kutsutuissa RESTful-rajapinnoissa, ja arvioivat HATEOAS:n olevan yksi REST:n useimmin rikotuista rajoitteista. Rajoitteen käyttöä havainnollistetaan kohdassa 7.2.

6.5 Järjestelmän kerroksittaisuus

Kerroksittainen järjestelmä (kuva 5) koostuu kerroksista, jotka koostuvat komponenteista, joita ovat esimerkiksi käyttöliittymä ja www-sovellusrajapinta. Kerrokseen kuuluva komponentti kykenee vuorovaikuttamaan vain välittömästi vierekkäisiin kerroksiin kuuluvien komponenttien kanssa [Fielding, 2000, 82-83]. Tämä merkitsee sitä, ettei kerroksessa oleva komponentti ole tietoinen muiden kuin sen välittömässä läheisyydessä olevien kerrosten olemassaolosta, jolloin esimerkiksi käyttöliittymäkomponentti ei voi tietää, onko se yhteydessä suoraan palvelun sisältämään palvelimeen vai johonkin välittäjäpalveluun.



Kuva 5: Kerroksittainen järjestelmä.

Rajoittamalla komponenttien näkyvyyttä saadaan rajoitettua järjestelmän potentiaalista monimutkaisuutta. Lisäämällä kerroksia voidaan yksinkertaistaa monimutkaisten ongelmien käsittelyä pilkkomalla ne useaan abstraktioituun ohjelmiston tasoon [Garlan and Shaw, 1993, 11]. Kerroksia voidaan hyödyntää vanhentuneiden palveluiden kapseloinnissa ja uusien palveluiden suojaamisessa vanhentuneilta asiakasohjelmilta.

Välittäjäpalveluilla voidaan tasapainottaa esimerkiksi tietokantakuormitusta jakamalla pyynnöt useiden prosessointiyksiköiden ja verkkojen välille. Niiden avulla pystytään esimerkiksi autentikoimaan käyttäjiä ja parantamaan tietoturvaa lisäämällä palomuurien vaatimia turvallisuutta koskevia toimia organisaation verkon rajalle. Koska kerrosten välinen vuorovaikutus tapahtuu ketjuttaen, komponenttien väliset tapahtumat kulkevat aina välittäjäpalveluiden kautta, jolloin implementointi on yksinkertaista.

Kerroksittaisen järjestelmän ja yhdenmukaisen rajapinnan rajoitteissa mainittujen itseselitteisten viestien yhteistoiminnan avulla voidaan muokata ja suodattaa kohderesurssia halutulla tavalla. Tämä on mahdollista, sillä itseselitteisten viestien semantiikka, kohderesurssi sekä haluttu toiminto ovat

näkyvissä jokaiselle välittäjäpalvelulle kaikissa sen läpikulkemissa kerroksissa.

Lisäämällä järjestelmään kerroksia kasvatetaan myös tiedonkäsittelystä johtuvaa viivettä. Tämä näkyy suoraan käyttäjälle suorituskyvyn laskuna. Välimuistin käyttöä tukevassa verkkopohjaisessa järjestelmässä tätä voidaan kompensoida hyödyntämällä välittäjäpalveluiden jaettua välimuistia. Välimuistia jakavien palveluiden asettaminen organisaation verkon rajalle voi johtaa merkittäviin suorituskykyparannuksiin. [Fielding, 2000, 82-84].

6.6 Koodinsiirron mahdollisuus

REST-malli tarjoaa asiakas-palvelin -arkkitehtuurin asiakasosapuolelle mahdollisuuden laajentaa toiminnallisuuttaan lataamalla ja suorittamalla palvelimella sijaitsevaa ohjelmakoodia. Tämä yksinkertaistaa asiakasohjelmia vähentämällä vaadittujen etukäteen toteutettavien ominaisuuksien määrää, sekä parantaa järjestelmän laajennettavuutta. Toisaalta koodinsiirto vähentää näkyvyyttä. Näkyvyyden vähentymisestä johtuen koodinsiirron mahdollisuus on ainoa vapaaehtoinen rajoite REST:ssä.

Ristiriitaiselta kuulostavan rajoitteen vapaaehtoisuuden päämääränä on ylläpitää rajoitetta sellaisissa järjestelmän osissa, joissa se on käytössä. Esimerkiksi organisaation sisäverkossa ajettavassa asiakasohjelmassa hyödynnetään organisaation omilta palvelimilta ladattua koodia, mutta palomuurirajoitteiden vuoksi tätä toiminnollisuutta ei voida ladata sisäverkon ulkopuolisista lähteistä. Tällöin ulkopuolisesta verkosta katsottuna vaikuttaa siltä, että sisäverkossa ajettava asiakasohjelma ei tue koodinsiirtoa. Vapaaehtoisen rajoitteen ansiosta järjestelmäarkkitehtuuri voi tukea haluttua toiminnallisuutta yleisessä käyttötapauksessa, ollen kuitenkin tietoinen siitä, että joissain tapauksissa toiminnallisuus voi olla rajattua. [Fielding, 2000, 84-85].

7 RESTful www-sovelluspalvelurajanpinnat

WWW-sovelluspalvelun toteutus tapahtuu käytännössä usein jonkin webkehitykseen soveltuvan *sovelluskehyksen* (engl. software framework) avulla. Sovelluskehyksen hallinnoidessa esimerkiksi HTTP-viestien reittejä ja tietokantayhteyttä kehittäjän vastuulle jää REST:lle ominaisten suunnitteluvalintojen lisäksi esimerkiksi sovelluksen tietoturva-arkkitehtuuri, testaus ja dokumentointi. Tässä luvussa tarkastellaan REST-kohtaisia suunnitteluparadigmoja www-sovelluspalveluiden kannalta laajentaen ja tarkentaen REST-arkkitehtuurissa määriteltyjä rajoitteita.

On huomionarvoista, että vaikka rajoitteissa puhutaan paljon asiakaspalvelin -arkkitehtuurin dynamiikasta, niin RESTful:n www-sovelluspalvelun toteutuksessa ei tarvitse suoranaisesti puuttua asiakasohjelman eli käyttöliittymän toimintaan. Vaikka ohjelmistokokonaisuuden tarkoituksenmukainen toiminta riippuu myös asiakasohjelman toteutustavasta, REST:n asettamia rajoitteita ja niiden toteutustapoja hallinnoidaan pääosin palvelinohjelmistosta käsin. Tärkeää tämä on siksi, että www-sovelluspalvelulla on usein useita, palvelinohjelmistoa hallinnoivasta osapuolesta riippumattomia käyttäjiä eli asiakasohjelmia.

Esimerkkikoodissa 1 kuvatut metodit toteuttavat CRUD-joukkoon (create, read, update, delete) kuuluvia toimintoja. Metodeilla pyritään siis luomaan, hakemaan, muokkaamaan ja poistamaan resursseja. Jos kuvattu rajapinta toteutettaisiin vapaavalinnaisella ohjelmointikielellä, voitaisiin sitä käyttää kuin perinteistä ohjelmistorajapintaa. Koska kuitenkin halutaan toteuttaa web-pohjainen www-sovelluspalvelurajapinta, täytyy toteutettuja metodeja pystyä kutsumaan verkon välityksellä. Tämä onnistuu luomalla rajapintaan resursseille URI:lla osoitettavat päätepisteet sekä määrittelemällä kyseisille resursseille mahdolliset HTTP-metodit.

7.1 URI

REST www-sovelluspalvelun URI-päätepisteiden tulee olla intuitiivisia, kuvaavia ja helposti ymmärrettäviä. Niiden ei ole tarkoitus toimia koneellisessa vuorovaikutuksessa, vaan mahdollistaa asiakasohjelmaa kehittävän ihmisen määritellä rajapintayhteyteen vaadittavia *URI-reittejä* eli URI-osoitteita [Rodriguez, 2008].

```
http://www.linjaoy.fi/affB6k9/12?topic=technology
```

Esimerkkikoodi 8: Huono URI.

Esimerkiksi syystä tai toisesta koneellisesti generoitujen resurssipolkujen (esimerkkikoodi 8) sekä tarpeettomien URI-kyselyosien (kohta 4.3) käyttö ei ole suositeltua, sillä ne heikentävät luettavuutta. Parhaassa tapauksessa rajapinnan URI-kokoelma dokumentoi itse itseään tarviten hyvin vähän todellista dokumentaatiota tai lisämäärittelyjä tuekseen.

Eräs yleistynyt tapa luoda käytettävyydeltään edistyneitä URI-reittejä on käyttää hakemistorakenteen kaltaista URI:a. Tällainen URI, kuten hyvä hakemistorakennekin, on hierarkkinen tietopuu, joka alkaa yhdestä juuresta haarautuen useaksi eri alireitiksi. Tarkastellaan esimerkkikoodissa 9 määri-

teltyä aiemmin kuvatun Linja Oy -rajapinnan URI-muotoista esitystä.

```
http://www.linjaoy.fi/trips/{tripId}
http://www.linjaoy.fi/trips/available
http://www.linjaoy.fi/trips/available/{startDate}/{endDate}
http://www.linjaoy.fi/reservation/
http://www.linjaoy.fi/reservation/{reservationId}
```

Esimerkkikoodi 9: Linja Oy -rajapinnan URI-päätepisteet.

Esimerkissä voidaan huomata resurssien yksilöllistäminen eräänlaisen puurakenteen kautta, jonka juurina toimivat matkat ja varaukset. Aaltosulkeilla merkittyjen muuttujien avulla yksilöidään resursseja esimerkiksi tietokannassa sijaitsevan id:n tai päivämäärien mukaan. Koska resurssi voi olla resursseista koostuva kokoelma, myös esimerkiksi vapaiden matkojen kokoelma on resurssi.

URI-muotoisesta rajapinnasta puuttuu muutamia esimerkkikoodissa 1 esiintyviä asioita, kuten metodikutsujen CRUD-etuliitteet ja parametreina annettavat *reservationInfo*-muuttujat. Näistä johtuen pääasiallisia API-päätepisteitä on myös vähemmän. Syy päätepisteiden erilaisuuteen on se, että RESTful -rajapinnan URI-päätepisteiden kuuluu olla ensisijaisesti resursseja, ei operaatioita [Fielding, 2000, 109-111]. Tästä syystä rajapintaa ei käännetä suoraan esimerkkikoodissa 10 kuvattuun muotoon.

```
http://www.linjaoy.fi/getAvailableTrips
```

Esimerkkikoodi 10: Operaatiokeskeinen URI.

Rodriguez [2008] antaa resurssikeskeisyyden ja hakemistorakennepohjaisuuden lisäksi muutamia lisäohjeita RESTful URI-päätepisteiden luomiseen:

- Piilota palvelimella ajettavien skriptikielten tiedostopäätteet (.php, .js), jotta voit vaihtaa kieltä myöhemmin muuttamatta URI:a.
- Käytä vain pieniä kirjaimia.
- Käytä välilyöntien tilalla joko väliviivaa (-), tai alaviivaa (_).
- Vältä kyselyosien (kuvattu kohdassa 4.3) käyttöä.
- Jos käytetty URI on osittaisella reitillä, palauta 404-koodin sijaan jokin oletussivu tai -resurssi.
- Substantiivien käyttö resurssipoluissa parantaa HTTP-metodien semanttista ymmärrettävyyttä.

7.2 HTTP

HTTP on ensisijainen RESTful-www-sovelluspalvelurajapintojen tiedonsiirtoon käytetty tiedonsiirtoprotokolla. Se noudattaa REST:n arkkitehtuaalisia rajoitteita (luku 6), kuten resurssien esitysmuotoista manipulointia ja itseselitteisiä viestejä. [Fielding, 2000, 116-121].

Koska HTTP-metodit ovat käytännössä vain semantiikkaa kuvaavia merkkijonoja, HTTP-metodien (kohta 4.2) oikeellinen ja täsmällinen käyttö on avainasemassa RESTful-www-sovellusrajapinnan suunnittelussa. Rodriguez [2008] esittelee ongelman, jossa tiedonhakuun käytettävää GET-metodia käytetään tiedon luontiin (esimerkkikoodi 11).

```
GET /adduser?name=Matti HTTP/1.1
```

Esimerkkikoodi 11: Virheellinen HTTP-pyynnön aloitusrivi, GET-metodin käyttö tiedon luonnissa.

Esimerkkikoodin 11 pyyntö korjataan muuttamalla GET-metodi semanttisesti oikeelliseksi POST-metodiksi (oletetaan, että käyttäjä lisätään idempotenttisesti) ja antamalla käyttäjätiedot pyynnön viestirungossa (esimerkkikoodi 12).

```
POST /users HTTP/1.1
host: localhost
content-type: application/json
cache-control: no-cache

{"name":"Matti"}
```

Esimerkkikoodi 12: Korjattu HTTP-pyyntö.

Semanttisen virheen lisäksi GET-metodin käytöllä voi olla muitakin seuraamuksia. Esimerkiksi internetin indeksointiin käytetyt *web-indeksoijat* (engl. web crawler) saattavat linkkiä indeksoidessaan laukaista tietoa muokkaavan operaation, sillä GET-metodia voidaan käyttää kirjoittamalla URI-polku suoraan www-selaimen URL-palkkiin.

Virheellisen HTTP-metodin käytöllä voi olla muitakin tietoturvakriittisiä seuraamuksia kuin tiedon eheyden vaarantaminen. Esimerkiksi kirjautumistietojen antaminen GET-parametreina URI:ssa tallentaisi jokaisella kirjautumiskerralla käyttäjän käyttäjätunnuksen sekä salasanan selaimen sivuhistoriaan.

```
GET /trips/{tripId}
GET /trips/available
GET /trips/available/{startDate}/{endDate}
GET /reservation/{reservationId}
POST /reservation/
PATCH /reservation/{reservationId}
DELETE /reservation/{reservationId}
```

Esimerkkikoodi 13: Linja Oy -rajapinnan HTTP-metodit ja kohderesurssit.

```
POST /reservation/ HTTP/1.1
host: localhost
content-type: application/json
cache-control: no-cache

{
   "info": {
      "name": "Meikäläinen Matti",
      "seats": 2
   }
}
```

Esimerkkikoodi 14: Matkan varaus Linja Oy:n palvelusta.

```
HTTP/1.1 400
status: 400
content-encoding: gzip
content-type: application/json; charset=utf-8
cache-control: max-age=16400
connection: keep-alive

{
   "messages": [
      {
         "message": "reservation was succesfully created",
         "type": "resource_created",
         "resource": "reservation/12"
      }
   ]
}
```

Esimerkkikoodi 15: HATEOAS:n toteutus.

Kohdassa 7.1 huomattiin, että URI-päätepisteistä puuttuu CRUD:n mukaiset funktioetuliitteet. Nämä määritellään HTTP-metodeilla, joista suurin osa kuuluu CRUD-joukkoon. Puuttuvat reservationInfo -muuttujat annetaan POST- ja PATCH-metodeilla tehtyjen pyyntöjen viestirungossa (esimerkkikoodi 14).

Kohdassa 6.4 nostettiin esiin hypermedia sovelluksen ajurina -rajoitteen vähäinen käyttö. Linja Oy:n tapauksessa rajoitteen toteuttaminen onnistuisi esimerkiksi palauttamalla luodun tilauksen yhteydessä HTTP-vastauksen viestirungossa kyseisen tilauksen resurssipolun eli URI:n, joten lisäämme kyseisen toiminnollisuuden esimerkkikoodissa 13 kuvattuun rajapintaan. Toiminnollisuutta havainnollistetaan esimerkkikoodissa 15.

7.3 HTTP-vastaukset

HTTP-vastaus sisältää aina statuskoodin [Fielding and Reschke, 2014b], esimerkiksi useimmille tutun 404 (Not Found) -koodin. Statuskoodin lisäksi HTTP-vastaus sisältää tyypillisesti metatietoja (esim. mediatyyppi) vastauksen sisällöstä sekä viestirungon sisällön.

Fielding ja Reschke [2014b] määrittelevät HTTP:n statuskoodien jakautuvan viiteen eri kategoriaan: informatiivisiin (1xx), onnistuneisiin (2xx), uudelleenohjattaviin (3xx), asiakasvirheisiin (4xx) ja palvelinvirheisiin (5xx).

Jokaiselle statuskoodille ei kannata pyrkiä etsimään käyttötarkoitusta, vaan parasta olisi pitää www-sovelluspalvelurajapinnan palauttamien statuskoodien joukko suhteellisen suppeana. Statuskoodien liiallinen laajuus johtaa siihen, että asiakasohjelman kehittäjä joutuu ottamaan tai pahemmassa tapauksessa ei huomaa ottaa kehitystyössään huomioon suurta määrää erilaisilla statuksilla kuvattuja tilanteita, esimerkiksi käyttöliittymässä näytettäviä virheilmoituksia varten.

Statuskoodi itsessään ei aina riitä kuvaamaan pyynnön onnistumista, vaan lisäksi saatetaan tarvita vapaamuotoista lisäselvitystä esimerkiksi ilmenneestä virheestä. Tämä annetaan vastauksen viestirungossa. Tällaisessa viestissä voidaan esimerkiksi ilmoittaa, miksi vastattiin kyseisellä statuskoodilla tai mitä jonkin virheen korjaamiseksi voitaisiin tehdä (esimerkkikoodi 16). HTTP-vastauksissa tulevien viestirunkojen tulee olla johdonmukaisia läpi järjestelmän. Tämä onnistuu parhaiten niin, että päätetään järjestelmälle yhteinen joukko mahdollisia statuskoodeja. Tämän lisäksi määritellään jokin yleinen, viestirungossa käytettävä, mediatyyppinen tietomuoto tai -rakenne, kuten JSON tai XML.

```
HTTP/1.1 400
status: 400
content-encoding: gzip
content-type: application/json; charset=utf-8
cache-control: no-cache
date: Mon, 04 Dec 2017 12:48:27 GMT
connection: keep-alive

{
   "messages": [
      {
         "message": "name field in request body required",
         "type": "empty_body_field"
      },
      {
         "message": "message string length must be between 1 and
            255",
         "type": "invalid_message_length"
      }
   ]
}
```

Esimerkkikoodi 16: Esimerkinomainen HTTP-viestirunkorakenne.

Erityisen tarkkana kannattaa olla asiakasohjelmasta johtuvien virheiden kanssa, sillä niiden havainnointi ja selkeä dokumentointi voi mahdollisesti nopeuttaa ja suoraviivaistaa asiakasohjelmaa kehittävän sovelluskehittäjän työprosessia huomattavasti.

8 Yhteenveto

Tutkielmassa tarkasteltiin REST-sovellusarkkitehtuurimallia ja sen liitännäisteknologioita www-sovelluspalveluiden näkökulmasta. Tämän lisäksi havainnollistettiin käsiteltyjä aiheita kuvitteelliselle Linja Oy:lle luodun www-sovelluspalvelurajapinnan avulla. Tutkielman aikana aiheita käsiteltiin progressiivisesti esimerkkien avulla alkaen www-sovelluspalveluiden perusteista, päättyen REST-arkkitehtuurimallin määrittelemiin rajoitteisiin ja REST:n käyttöön www-sovelluspalvelurajapinnan suunnittelussa.

WWW-sovelluspalveluita ja -sovelluspalvelurajapintoja käsiteltiin tutkielman alussa melko popularisoidusti. Tällä pyrittiin luomaan lukijalle vakaa

ymmärrys kyseisten teknologioiden käyttötarkoituksista. Käsittelemällä HTTP-protokollan perusteita yleisellä tasolla pohjustettiin erityisesti oikeellisten HTTP-metodien käyttöä RESTful-www-sovelluspalvelurajapinnoissa.

REST-arkkitehtuurimalliin kuuluvia rajoitteita käsitellessä huomattiin rajoitteiden keskittyvän tiettyjen päämäärien ympärille. Suurin osa rajoitteista pyrkii edistämään järjestelmän suorituskykyä, skaalautuvuutta tai kehityskelpoisuutta esimerkiksi löyhien sidonnaisuuksien avulla. Useat rajoitteista ovat kuitenkin pohjimmiltaan kompromisseja ja saattavat näin esimerkiksi parantaa sidonnaisuutta suorituskyvyn kustannuksella.

Rajoitteita käsiteltäessä havaittiin myös, että REST:n keskittyessä yhdenmukaisen rajapinnan toimintaan ja rajoitteisiin ottaa se kantaa rajapinta-arkkitehtuurin lisäksi järjestelmän kokonaisarkkitehtuuriin. Vaikka REST määritteleekin ohjelmistoarkkitehtuurin rakennetta ja ominaisuuksia osittain melko tarkasti, se ei ota kantaa mallin mukaisesti luodun järjestelmän toteutustekniikoihin tai teknisiin yksityiskohtiin, jolloin RESTful-rajapinnan toteutuksessa käytetyt teknologiat voidaan valita tarkoituksenmukaisesti. Tällöin, yhdistettynä HTTP:n käyttöön tiedonsiirtoprotokollana, REST toteuttaa osaltaan www-sovelluspalveluille ominaisen alustariippumattomuuden.

Tutkielman rajatusta laajuudesta johtuen tutkielmassa ei käsitelty tarkemmin muita www-sovelluspalvelurajapintojen toteutuksessa oleellisia aiheita, kuten autentikaatiota, ohjelmistotestausmetodeja (esimerkiksi automatisoitu yksikkötestaus), CSRF/XSS -haavoittuvuutta tai HTTP:n *kuljetuskerroksen turvallisuus* -protokollalla (engl. transport layer security, tai TLS) salattua versiota: HTTPS:ää. Samasta syystä ei myöskään käsitelty vaihtoehtoisia teknologioita, kuten RCP:tä tai SOAP:ia.

REST-arkkitehtuurimallissa määriteltyjen rajoitteiden sekä tutkielmassa käsiteltyjen www-sovelluspalvelurajapintakohtaisten suunnitteluperiaatteiden seuraaminen ei yksin riitä hyvän www-sovelluspalvelurajapinnan tai -sovelluspalvelun toteutukseen. RESTful-rajapinnan kehittäjän on aina huomioitava myös muut joustavan ja hyvän suunnittelun keinot, sekä arvioitava käytettävien toteutustekniikoiden ja -teknologioiden osuus kokonaisarkkitehtuurissa. Varsinkin tilanteessa, jossa kehitetään www-sovelluspalvelua tietoturvakriittisen tiedon julkaisemiseen rajatulle asiakaskunnalle, on otettava huomioon rajapinnan tietoturva-aspekti. Tietoturvan kokonaisvaltainen kartoitus RESTful-rajapinnoissa olisikin tärkeä jatkotutkimusaihe.

Viitteet

Alarcón, R., and Wilde, E. 2010. Restler: crawling restful services. In: *Proc.*

of the 19th International Conference on World Wide Web, 1051–1052.

Alarcon, R., Wilde, E., and Bellido, J. 2010. Hypermedia-driven restful service composition. In: *Proc. of ICSOC Workshops,* 111–120.

Belshe, M., Peon, R., and Thomson, M. 2015. *Hypertext transfer protocol version 2 (http/2)* (RFC nro 7540). RFC Editor. Internet Requests for Comments. Checked 14.12.2017, Available `http://www.rfc-editor.org/rfc/rfc7540.txt`.

Berners-Lee, T., Fielding, R., and Masinter, L. 2005. *Uniform resource identifier (uri): Generic syntax* (STD nro 66). RFC Editor. Internet Requests for Comments. Checked 14.12.2017, Available `http://www.rfc-editor.org/rfc/rfc3986.txt`.

Braden, R. 1989a. *Requirements for internet hosts - application and support* (STD nro 3). RFC Editor. Internet Requests for Comments. Checked 14.12.2017, Available `http://www.rfc-editor.org/rfc/rfc1123.txt`.

Braden, R. 1989b. *Requirements for internet hosts - communication layers* (STD nro 3). RFC Editor. Internet Requests for Comments. Checked 14.12.2017, Available `http://www.rfc-editor.org/rfc/rfc1122.txt`.

Brown, A., and Haas, H. 2004. *Web services glossary* (W3C Note). Checked 14.12.2017, Available `http://www.w3.org/TR/2004/NOTE-ws-gloss-20040211/`.

Dusseault, L., and Snell, J. 2010. *Patch method for http* (RFC nro 5789). RFC Editor. Internet Requests for Comments. Checked 14.12.2017, Available `http://www.rfc-editor.org/rfc/rfc5789.txt`.

Feng, X., Shen, J., and Fan, Y. 2009. Rest: An alternative to rpc for web services architecture. In: *Proc. of the First International Conference on Future Information Networks,* 7–10.

Fielding, R. 2000. Rest: Architectural styles and the design of network-based software architectures. *Doctoral dissertation, Department of Information and Computer Science, University of California.*

Fielding, R. 2008. *REST APIs must be hypertext-driven.* Checked 3.12.2017, Available `http://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven`.

Fielding, R., Nottingham, M., and Reschke, J. 2014. *Hypertext transfer protocol (http/1.1): Caching* (RFC nro 7234). RFC Editor. Internet Requests for Comments. Checked 14.12.2017, Available `http://www.rfc-editor.org/rfc/rfc7234.txt`.

Fielding, R., and Reschke, J. 2014a. *Hypertext transfer protocol (http/1.1): Message syntax and routing* (RFC nro 7230). RFC Editor. Internet Requests for Comments. Checked 29.11.2017, Available `http://www.rfc-editor.org/rfc/rfc7230.txt`.

Fielding, R., and Reschke, J. 2014b. *Hypertext transfer protocol (http/1.1): Semantics and content* (RFC nro 7231). RFC Editor. Internet Requests for Comments. Checked 4.12.2017, Available `http://www.rfc-editor.org/rfc/rfc7231.txt`.

Freed, N., and Borenstein, N. S. 1996. *Multipurpose internet mail extensions (mime) part two: Media types* (RFC nro 2046). RFC Editor. Internet Requests for Comments. Checked 3.12.2017, Available `http://www.rfc-editor.org/rfc/rfc2046.txt`.

Garlan, D., and Shaw, M. 1993. An introduction to software architecture. In: V. Ambriola and G. Tortora (eds.), *Advances in Software Engineering and Knowledge Engineering.* World Scientific, 1–39.

Liu, S., Li, Y., Sun, G., Fan, B., and Deng, S. 2017. Hierarchical rnn networks for structured semantic web api model learning and extraction. In: *Proc. of 2017 IEEE International Conference on Web Services (ICWS)* 708–713.

Medjahed, B. 2004. *Semantic web enabled composition of web services* (unpublished dissertion).

Mumbaikar, S., and Padiya, P. 2013. Web services based on soap and rest principles. In: *International Journal of Scientific and Research Publications* 3, 5, 1–4.

Parziale, L., Liu, W., Matthews, C., Rosselot, N., Davis, C., Forrester, J., Britt, D. T., et al. 2006. *TCP/IP Tutorial and Technical Overview*, IBM Redbooks.

Pautasso, C. 2009. Restful web service composition with bpel for rest. In: *Data and Knowledge Engineering* 68, 9, 851–866. Elsevier.

Pautasso, C. 2014. Restful web services: principles, patterns, emerging technologies. In: A. Bouguettaya, Q. Z. Sheng, and F. Daniel (eds.), *Web Services Foundations.* Springer, 31–51.

Pautasso, C., Zimmermann, O., and Leymann, F. 2008. Restful web services vs. big'web services: making the right architectural decision. In: *Proc. of the 17th International Conference on World Wide Web,* 805–814.

Rodriguez, A. 2008. *Restful web services: The basics.* Checked 14.12.2017, Available `https://www.ibm.com/developerworks/library/ws-restful/ws-restful-pdf.pdf`.

Sheng, Q. Z., Qiao, X., Vasilakos, A. V., Szabo, C., Bourne, S., and Xu, X. 2014. Web services composition: A decade's overview. In: *Information Sciences* 280(Supplement C), 218–238.

Thaler, D., Hansen, T., and Hardie, T. 2015. *Guidelines and registration procedures for uri schemes* (BCP nro 35). RFC Editor. Internet Requests for Comments. Checked 14.12.2017, Available `https://www.rfc-editor.org/rfc/rfc7595.txt`.

Vinoski, S. 2008. RESTful Web Services Development Checklist. In: *IEEE Internet Computing* 12, 6, 96–95.

Waldo, J., Wyant, G., Wollrath, A., and Kendall, S. 1996. A note on distributed computing. In: *Proc. of International Workshop on Mobile Object Systems,* 49–64.