SOAP-tyyppisen www-sovelluspalvelun muuntaminen REST:in mukaiseksi www-sovelluspalveluksi

Alpertti Tirronen

Tampereen yliopisto
Informaatiotieteiden yksikkö
Tietojenkäsittelyoppi
Pro gradu -tutkielma
Ohjaaja: Jyrki Nummenmaa
Maaliskuu 2013

Tampereen yliopisto
Informaatiotieteiden yksikkö
Tietojenkäsittelyoppi
Alpertti Tirronen: SOAP-tyyppisen www-sovelluspalvelun muuntaminen REST:in mukaiseksi www-sovelluspalveluksi
Pro gradu -tutkielma, 49 sivua, 7 liitesivua
Maaliskuu 2013

Tässä tutkielmassa luodaan kirjallisuuden ja aikaisemman tutkimuksen perusteella katsaus REST:in käsitteeseen. REST on www-sovelluspalveluissa (Web Services) käytetty arkkitehtuurinen tyyli. Lisäksi perehdytään REST:in mukaisen (RESTful) www-sovelluspalvelun ominaisuuksiin. Käsitteet eivät ole sisällöllisesti täysin vakiintuneita, joten esityksessä pyritään muodostamaan yhtenäinen kuva näiden käsitteiden suhteista ja olemuksesta. Lisäksi luodaan silmäys kouralliseen muita aihepiiriin läheisesti liittyviä käsitteitä ja teknologioita.

Käsitemäärittelyn lisäksi kuvataan suunnittelutieteelliseen konstruktiiviseen tutkimusperinteeseen nivoutuva tutkimus SOAP-tyyppisen www-sovelluspalvelun muuntamisesta REST:in mukaiseksi www-sovelluspalveluksi. Tutkimuksen oheistuotteena syntyy joukko menetelmiä, joita voidaan mahdollisesti käyttää hyväksi tämän tyyppisten muutostöiden yhteydessä.

Tämän tutkimuksen valossa vaikuttaa siltä, että muunnosoperaatio, jossa ainoastaan varsinainen www-sovelluspalvelu vaihdetaan toisen tyyppiseen, olisi myös yleisemmin mahdollista toteuttaa. Lisäksi tutkimuksessa saadaan viitteitä siitä, että yleisesti hyödynnettävien muunnosprosessia tukevien menetelmien luominen on mahdollista.

Tutkimuksen pohjalta ei voida kuitenkaan tehdä johtopäätöksiä luotujen menetelmien käyttökelpoisuudessa käytännön tilanteissa. Jatkotutkimuksia vaaditaan tämän arvioinnin tekemiseksi. Jatkotutkimuksia tarvitaan myös menetelmien kehittämiseksi edelleen. Lisäksi järjestelmä- ja sovellusympäristön, sekä käytössä olevien tietovarastojen merkitys muunnosprosessin onnistumiseen ja siinä käytettyjen menetelmien toimivuuteen vaatii lisäselvityksiä.

Avainsanat ja -sanonnat: HTTP, REST, RESTful, ROA, SOA, SOAP, URI, WADL, WSDL, Web Service, www-sovelluspalvelu, WWW.

Sisällys

Sanasto....1
1.Johdanto....3
2.REST:in mukainen www-sovelluspalvelu....5
2.1.REST....5
2.1.1.Asiakas-palvelin....5
2.1.2.Tilattomuus....6
2.1.3.Välimuisti....7
2.1.4.Yhtenäinen rajapinta....7
2.1.5.Kerrosmainen sovellusarkkitehtuuri....8
2.1.6.Code-on-demand....8
2.2.HTTP 1.1....8
2.2.1.HTTP-metodit....9
2.2.2.HTTP-tilakoodit....12
2.2.3.HTTP-otsikkotiedot....14
2.3.ROA....15
2.3.1.URI....15
2.3.2.Resurssit ja representaatiot....15
2.3.3.Osoitteellisuus, tilattomuus, linkitettävyys ja yhtenäinen rajapinta 17
2.3.4.ATOM....19
2.3.5.WADL....21
2.4.RMM....23
3.SOAP:ista REST:in mukaiseen www-sovelluspalveluun....27
3.1.SOAP ja WSDL....28
3.1.1.WSDL-kuvauksen rakenne....29
3.1.2.Operaatioiden tunnistaminen....36
3.1.3.Operaatioiden pilkkominen....36
3.1.4.Domain-mallin luonti....37
3.1.5.Resurssimallin luonti....40
3.1.6.Resurssimallista toteutukseen....44
3.2.Sovellustason tarkastelua....44
4.Lopuksi....46

Viiteluettelo.... 47
Liitteet

Sanasto

ASCII (American Standard Code for Information Interchange) – Tietokoneiden merkistö, joka sisältää erityisesti amerikanenglannissa käytettyjä merkkejä.
ATOM – Kaksi määrittelyä, The Atom Syndication Format ja The Atom Publishing Protocol, sisältävä käsite. Ensimmäistä käytetään uutisvirtojen esittämiseen XML-muodossa ja jälkimmäistä resurssien käsittelyn työkaluna.
HTML (hypertext markup language) – www:ssä yleisesti käytetty kieli hypertekstin tallentamiseen. HTML:llä voidaan kuvata myös tekstikokonaisuuden rakenne.
HTTP (hypertext transfer protocol) – www:ssä käytetty tiedonsiirtoprotokolla. Tässä tutkielmassa viitataan erityisesti protokollan versioon 1.1.
JSON (javascript object notation) – Pääasiassa tiedonsiirrossa käytetty merkintäkieli, jonka avulla voidaan sisällyttää tekstidokumenttiin metatietoa tietosisällöstä.
REST (representational state transfer) – Arkkitehtuurinen tyyli, joka määrittää REST:in mukaisen www-sovelluspalvelun rajoitteet.
REST:in mukainen (RESTful) – nimitys www-sovelluspalvelulle, joka täyttää REST:in vaatimukset. Suomenkielinen käsite otettu käyttöön tässä tutkielmassa luettavuuden parantamiseksi.
ROA (resource oriented architecture) – Resurssipohjainen arkkitehtuuri, joka on kehitetty erityisesti REST:in tarpeisiin.
Palveluväylä (Service bus) – SOA-arkkitehtuurissa palvelujen ja niiden kuluttajien välissä toimiva sovelluskomponentti. Helpottaa ylläpitoa ja muutoksia.
SOA (service oriented architecture) – Palvelukeskeinen arkkitehtuuri. Yleisesti www-sovelluspalvelujen toteuttamiseen käytetty arkkitehtuuri.
SOAP (simple object access protocol) – www-sovelluspalvelujen toteutukseen yleisesti käytetty, erityisesti palvelukeskeisen arkkitehtuurin tarpeisiin soveltuva, teknologia.
UML (Unified modeling language) – Mallinnukseen käytettävä kokoelma erilaisia kaavioita ja elementtejä.
URI (uniform resource identifier) – Resurssin yksilöivä tunniste. Www:ssä URI kertoo myös resurssin sijainnin.
WADL (web application description language) – XML-rakenteinen kuvauskieli www-sovelluspalvelujen kuvaamiseen. Käytetään pääasiassa REST:in mukaisten www-sovelluspalvelujen yhteydessä.

www-sovelluspalvelu (Web Service) – www:ssä toimivien palvelinsovellusten rajapinta, jota asiakasohjelmat käyttävät. Näkökulmasta riippuen myös tavanomaisten verkkosivustojen voidaan katsoa olevan www-sovelluspalveluja.

WSDL (web service definition language) – SOAP-palvelujen kuvakseen käytetty kuvauskieli.

WWW (world wide web) – Joukko sovelluskomponentteja, verkkosivuja jne., jotka yhdistyvät toisiinsa käyttäen HTTP-protokollaa. Www voidaan jakaa ihmisen käyttämään www:hen ja ohjelmasovelluksen käyttämään www:hen [Richardson and Ruby, 2007].

XML (extensible markup language) – Rakenteisissa dokumenteissa yleisesti käytetty merkintäkieli, jonka avulla voidaan sisällyttää tekstidokumenttiin metatietoa tietosisällöstä.

1. Johdanto

Representational state transfer (REST) on viime vuosina lisännyt suosiotaan www-sovelluspalveluiden toteuttamistapana [Royal pingdom, 2010]. REST on www-sovelluspalveluiden arkkitehtuurinen tyyli, jossa yhtenä pääajatuksena on hyödyntää HTTP-protokollan (tai jonkin muun yleisesti tunnetun) ominaisuuksia www-sovelluspalveluiden toteutuksessa. REST-käsitteen esitti ensimmäisen kerran Fielding [2000] väitöskirjassaan.

Www-sovelluspalveluilla tarkoitetaan perinteisesti www:ssä toimivien tietokonesovellusten rajapintaa. Www-sovelluspalvelujen toteutukseen on olemassa erilaisia tekniikoita, kuten XML-RPC, SOAP, REST jne. Voidaan kuitenkin perustellen sanoa, että myös aivan tavallinen verkkosivusto on www-sovelluspalvelu, koska vaikka sivusto olisikin suunnattu ihmiselle selaimella käytettäväksi, niin ei ole mitenkään pois suljettu, etteikö myös jokin sovellus voitaisi ohjelmoida sitä käyttämään. Tämä on erityisesti mahdollista silloin, kun rajapinta noudattaa REST:in oppeja.

Viime vuosina aiheesta on julkaistu jonkin verran kirjallisuutta ja alkuperäinen REST termi on saanut tarkennuksia ja laajennuksia. Voidaan esimerkiksi erottaa REST arkkitehtuurisena tyylinä ja toisaalta sanoa, että jokin verkkopalvelu on REST:in mukainen. REST käsitteenä ei ole siis täysin vakiintunut, joten standardit ja eri ominaisuuksien painotukset elävät.

Luvussa 2 luon kirjallisuuden pohjalta katsauksen ensin siihen mitä REST ja REST:in mukainen www-sovelluspalvelu on. Käsitteistön vakiintumattomuuden vuoksi pyrin perustellen muodostamaan johdonmukaisen kokonaisuuden käsitteen tärkeimmistä piirteistä sen alkuperäisen merkityksen valossa. Tässä luvussa käsittelen myös joitakin tiiviisti käsitteeseen liittyviä sivukäsitteitä ja teknologioita.

Luvussa 3 käyn läpi tutkimusprojektini. Tutkimusmenetelmäni sijoittuu suunnittelutieteelliseen konstruktiiviseen tutkimusperinteeseen ja seuraa Järvisen ja Järvisen [2004] kuvailemaa linjaa.

Nykyaikainen IT-alan liiketoimintaympäristö asettaa erilaisia vaatimuksia käytetyille menetelmille. Tärkeitä vaatimuksia ovat mm. kustannustehokkuus ja laatu. Yleisesti siis haetaan vastausta esimerkiksi kysymykseen: miten toteutetaan toiminta X mahdollisimman lyhyessä ajassa, lopputuloksen A laadun olennaisesti kärsimättä? Molempien ominaisuuksien osalta on merkityksellistä, että käytetty menetelmä (metodi) on systemaattisesti kuvattu, arvioitu ja testattu [Järvinen ja Järvinen, 2004]. Ainoastaan tällöin menetelmän (metodin) kehittäminen tehokkaasti on mahdollista. Tähän pohjaa myös oma tutkimukseni.

Tutkimukseni lähtökohtana on kuvitteellinen asiakasvaatimus www-sovelluspalvelurajapintojen muuttamisesta SOAP:ista REST:in mukaiseen muotoon. Varsinainen tutkimukseni pyrkii vastaamaan kahteen kysymykseen. Ensiksikin, onko mahdollista muuttaa SOAP www-sovelluspalvelu REST:in mukaiseksi muuttamalla ainoastaan varsinainen rajapinta. Toiseksi, jos muunnos SOAP:ista REST:in mukaiseksi on mahdollinen, miten se tulisi tehdä.

Vastaan näihin kysymyksiin kuvailemalla muutosprosessin (metodin), jonka avulla muunnan WebServiceX-palvelusta [2013] löytyvän alueellisten säätietojen hakuun tarkoitetun pienehkön SOAP-www-sovelluspalvelun REST:in mukaiseksi. Tutkimuksen tuotteena syntyy kuvaus metodista, jolla muunnosoperaatio on mahdollista toteuttaa.

Luvun lopussa pyrin vielä abstraktiotasoa nostamalla arvioimaan lyhyesti, onko muunnos sovellustasolla mahdollista toteuttaa ainoastaan vaihtamalla varsinaiset rajapintakomponentit. Haasteita tähän asettavat sovelluksen eri komponenttien erilaiset pohjamallit, kuten relaatiomalli ja oliomalli, sekä resurssipohjaisuus ja palvelupohjaisuus.

Luku 4 on loppuyhteenveto, jossa nivon tutkielmani sisällön yhteen ja luon silmäyksen mahdollisiin jatkokehitysvaihtoehtoihin.

2. REST:in mukainen www-sovelluspalvelu

REST on joukko sovellusarkkitehtuurisia rajoitteita, joiden toteutuessa www-sovelluspalvelu voidaan katsoa olevan REST:in mukainen [Fielding, 2000]. Tässä luvussa esittelen näitä rajoitteita ja REST:in mukaisia www-sovelluspalveluja yleisesti.

2.1. REST

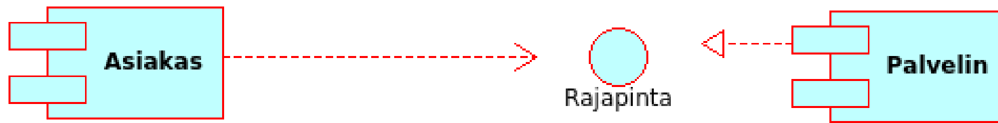
Www-sovelluspalveluiden yhteiseksi nimittäjäksi voidaan sanoa, että ne toimivat nimensä mukaisesti www:ssä. Käytännössä tämä tarkoittaa HTTP-protokollan käyttöä. On kuitenkin tärkeää huomata, että REST ei ole yhtä kuin HTTP tai www. Ensinnäkin REST on arkkitehtuurinen tyyli ja näin ollen riippumaton käytetystä protokollasta. Protokollana voidaan käyttää mitä tahansa protokollaa, joka mahdollistaa REST:in vaatimusten täyttämisen [Li and Wu, 2011]. Toisaalta voidaan kuitenkin väittää, että www on yhtä kuin HTTP-protokolla tai tarkemmin HTTP-protokollaa käyttävä asiakassovellus tai -palvelu [Richardson and Ruby, 2007]. Lisäksi, koska HTTP on tällä hetkellä tyypillisin tapa toteuttaa REST:in mukainen www-sovelluspalvelu, voitaisiin väittää, että REST on yhtä kuin HTTP. Tämä on kuitenkin siis virheellinen päätelmä.

Tämän tutkielman kontekstissa kuitenkin riittää, että todetaan näiden kolmen eri käsitteen kuvaavan, vaikkakin toisiinsa liittyviä, eri ilmiöitä. Tässä tutkielmassa www voidaan määritellä joukoksi sovelluskomponentteja, jotka yhdistyvät toisiinsa HTTP-protokollaa käyttäen. REST on siis arkkitehtuurinen tyyli, jonka toteuttajana HTTP-protokolla voidaan nähdä. Toisin sanoen REST määrittää yleiset periaatteet em. sovelluskomponenttien välisille rajapinnoille. Tässä tutkielmassa tarkastellaan erityisesti HTTP-protokollan versiota 1.1, koska se on lähes kaikkialla www:ssä tällä hetkellä käytössä oleva versio.

Fieldingin [2000] mukaan on olemassa kuusi erilaista sovellusarkkitehtuurista rajoitetta REST:ille: asiakas-palvelin, tilattomuus, välimuisti, yhtenäinen rajapinta, kerrosmainen sovellusarkkitehtuuri ja code-on-demand.

2.1.1. Asiakas-palvelin

Asiakas-palvelin -arkkitehtuuri muodostuu kahdesta komponentista: asiakas ja palvelin. Tämän arkkitehtuurin mukaisessa toteutuksessa asiakaskomponentti lähettää pyyntöjä palvelinkomponentille. Palvelinkomponentti myös tyypillisesti tarjoaa palvelunsa useille asiakkaille. Esimerkiksi staattinen www-sivu on yksinkertainen esimerkki asiakas-palvelinmallin käytännön toteutuksesta: verkkoselain hakee selaimen osoiteriville kirjoitetun URI:n osoittaman HTML-sivun. [Koskimies, 2000]



Kuva 1: Asiakas-palvelin -arkkitehtuuri UML-komponenttikaaviolla kuvattuna.

REST:in mukaisessa www-sovelluspalvelussa asiakas-palvelin -arkkitehtuurin merkitys korostuu erityisesti siksi, että jako näihin kahteen komponenttiin mahdollistaa palvelun skaalautuvuuden. Toisaalta jaottelu mahdollistaa molempien komponenttien itsenäisen kehittämisen rajapinnan säilyessä samana. [Fielding, 2000]

Lisäksi rajapinnan ollessa HTTP-protokollan mukainen ja näin ollen yleisesti tunnettu voidaan kehittää jatkuvasti uusia asiakassovelluksia, jotka käyttävät olemassa olevia palveluita. Tällöin palvelua voidaan käyttää hyvinkin monenlaisiin tarkoituksiin, myös sellaisiin, joita ei ole osattu ottaa huomioon palvelua suunniteltaessa. [Richardson and Ruby, 2007]

Esimerkkinä voitaisiin mainita vaikkapa Googlen kalenterisovellus. Kalenteri tarjoaa normaalit kalenteriominaisuudet REST:in mukaisen rajapinnan kautta. Asiakassovellukset voivat tällöin käyttää kalenteripalvelua mitä moninaisimpiin tarkoituksiin (harrastusporukan varaukset, ammattimaisemmat tilavaraukset jne.) ilman, että palvelimen tai sen kehittäjien täytyy tietää näistä tarpeista mitään.

2.1.2. Tilattomuus

HTTP-protokolla on tilaton protokolla [NWG, 1999]. Tämä tarkoittaa sitä, että jokainen HTTP-pyyntö on toisistaan riippumaton. HTTP-protokollan tilatonta luonnetta rikkoo esimerkiksi monissa verkkopalveluissa de facto -standardina käytössä oleva Netscape-selaimen laajennos: eväste [Kristol, 2001].

Eväste luodaan verkkopalvelun pyynnöstä asiakassovellukseen. Evästeen luonnin jälkeen se lähetetään jokaisen siinä määriteltyyn osoitteeseen lähetettävän HTTP-pyynnön otsikkotiedoissa, kunnes eväste poistetaan tai sen voimassaoloaika umpeutuu. Tyypillinen käyttötapaus evästeelle on verkkosovelluksen kirjautumisen jälkeen palvelimelle luotava istunto. Istunnolle luodaan tunniste, joka tallennetaan asiakassovellukseen evästeenä. Tämä mahdollistaa sovelluksen tilan säilyttämisen palvelinsovelluksessa, koska jokainen

pyyntö, joka sisältää voimassaolevan tunnisteen, voidaan yhdistää sitä vastaavaan istuntoon.

REST:in mukaisessa www-sovelluspalvelussa vaatimuksena on tilattomuus [W3C, 2004a]. Toisaalta tilattomuus käsitetään eri yhteyksissä hieman eri tavalla. Esimerkiksi www:n voidaan väittää olevan tilaton aina johtuen HTTP-protokollan luonteesta ja tästä johtuen myös siellä tarjottavat palvelut tulisi suunnitella tilattomiksi [Richardson and Ruby, 2007]. Käytännössä tämä tarkoittaisi sitä, että sovelluksen tila säilytetään aina asiakassovelluksessa.

Toinen, lievempi, näkökulma asiaan on nähdä tilattomuus teknologisia yksityiskohtia korkeammalla abstraktiotasolla. Tällöin tilattomuus nähdään lähinnä sovellustason suunnitteluratkaisuna käytännön välttämättömyyden sijaan.

Istuntojen osalta on tarjottu myös erilaisia välimuotoisina nähtäviä ratkaisuja. Kertakäyttöiset evästeet ovat yksi, etenkin tietoturvanäkökulmasta, esiin tuotu malli, joiden voisi ensi näkemältä ajatella olevan tilattomia. Tarkemmin tarkastellen kyseessä on kuitenkin tilatiedon säilyttävä tietoturvaa parantamaan pyrkivä toteutus perinteisestä evästeestä. Kertakäyttöisessä evästeessä istunnon tunnistamiseen käytettävä tunniste lasketaan asiakas- ja palvelinsovelluksen tunteman algoritmin mukaan jokaiselle HTTP-pyynnölle erikseen [Costa et al., 2012].

Käytännössä jokainen palvelinsovellus siis on edellä esitetyssä mielessä joko tilaton tai tilan säilyttävä ilman välimuotoja.

2.1.3. Välimuisti

Www-välimuistin tavoitteena on vähentää sovelluspalvelimen kuormaa ja verkkoliikennettä tallentamalla sisältöä välimuistiin [Fielding, 2000]. Välimuistin avulla sinne tallennettu sisältö voidaan jakaa asiakassovellukselle nopeammin. Yksi välimuistin käytön perusongelmista on se, mikä osa sisällöstä voidaan tallentaa ja mikä ei. Www:n alkuaikoina sisältö koostui suurelta osin staattisista HTML-dokumenteista. Dokumenteilla oli tietty sijainti ja näin ollen ne oli helppo tallentaa välimuistiin. Myös dokumenttien muutosnopeus oli mitä luultavammin nykyistä pienempi.

Nykyinen www on monimuotoisempi dynaamisesti muotoutuvine sisältöineen, joten yhä harvempi dokumentti on sellaisenaan tallennettavissa välimuistiin.

2.1.4. Yhtenäinen rajapinta

Yhtenäisellä rajapinnalla tarkoitetaan REST:in yhteydessä HTTP-protokollan määrittelemää rajapintaa [Richardson and Ruby, 2007]. Yhtenäiseksi sitä voidaan luonnehtia siksi, että se on käytännössä kaikkialla www:ssä käytössä. Myös esimerkiksi SOAP-palvelut

käyttävät HTTP-protokollaa, joten varsinainen SOAP-sanoma sisällytetään HTTP-pyynnön runkoon.

HTTP:n määrittelemä yhtenäinen rajapinta koostuu HTTP-metodeista. Näitä ovat esimerkiksi GET, PUT ja POST. Lisäksi rajapinta muodostuu HTTP-pyynnön ja -vastauksen otsikkotiedoista ja rungosta.

Yhtenäisen rajapinnan etuina verrattuna esimerkiksi SOAP-tyyppiseen www-sovelluspalvelun rajapintaan voidaan nähdä siis se, että se on yleisesti tunnettu. Tällöin julkisesti saatavilla olevalle REST-palvelulle voidaan kehittää aivan uusia spontaaneja käyttötapoja. Esimerkiksi Googlen kalenterisovellus on hyvä esimerkki monialaisten käyttötarkoituksien mahdollistamisesta.

2.1.5. Kerrosmainen sovellusarkkitehtuuri

REST:in mukaisilla www-sovelluspalveluilla tulee olla kerrosmainen sovellusarkkitehtuuri. Käytännössä tämä voi tarkoittaa esimerkiksi varsinaisen sovelluksen edessä olevaa www-palvelinta. Www-palvelin ainoastaan välittää HTTP-pyynnöt eteenpäin yhdelle tai useammalle sovelluspalvelimelle käsiteltäväksi.

Tällä rakenteella parannetaan sovelluksen skaalautuvuutta. Palvelimia voidaan lisätä teoriassa rajattomasti käytön lisääntyessä vaikuttamatta varsinaiseen rajapintaan. Haittapuolena tässä arkkitehtuurissa on mahdollisesti turhien välikerrosten olemassaolo, jolloin hukkatehoa kuluu eri kerroksissa toimivien palvelimien väliseen kommunikaatioon. [Fielding, 2000]

2.1.6. Code-on-demand

Code-on-demand tarkoittaa sitä, että asiakaskomponentti voi ladata tarvittavaa ohjelmakoodia ja suorittaa sen. Käytännössä tämä voi olla esimerkiksi javascript-kielellä toteutettu sovellus, joka suoritetaan asiakassovelluksessa ajonaikaisesti. Tämä on valinnainen rajoite. Valinnaisuus tarkoittaa tässä yhteydessä esimerkiksi sitä, että asiakassovelluksessa ajettava koodi on saatavissa ainoastaan vaikkapa yrityksen sisäverkosta, joka on palomuurilla erotettu Internetistä. [Fielding, 2000]

2.2. HTTP 1.1

Jokainen onnistunut HTTP-transaktio muodostuu pyyntö- ja vastaussanomasta. Jokainen pyyntö- ja vastaussanoma voidaan edelleen jakaa otsikkoon ja runkoon. Joissakin erityistapauksissa vastauksessa ei saa kuitenkaan olla runko-osaa. Ainoastaan otsikkotiedot sisältävä vastaussanoma palautetaan jos pyynnössä käytetty HTTP-metodi on HEAD tai vastaussanoman tilakoodi on 204 tai 304. [NWG, 1999]

```
GET /Protocols/rfc2616/rfc2616-sec4.html HTTP/1.1
Host: www.w3.org
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:19.0) Gecko/20100101 Firefox/19.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: fi-fi,fi;q=0.8,en-us;q=0.5,en;q=0.3
Accept-Encoding: gzip, deflate
DNT: 1
Connection: keep-alive
Pragma: no-cache
Cache-Control: no-cache
```

Taulukko 1: Esimerkki HTTP-pyyntösanomasta.

Taulukossa 1 on esimerkki HTTP-pyyntösanomasta. Ensimmäisen rivin alussa on käytetty HTTP-metodi. Tässä tapauksessa se on GET. Seuraavana on polku pyydettyyn resurssiin. Lyhimmillään polkumerkintä on merkki '/', joka tarkoittaa juurihakemistoa. Lopuksi ilmoitetaan vielä käytetty protokolla versioineen.

Seuraavalla rivillä alkavat varsinaiset otsikkotiedot. Otsikkotiedot ovat joukko avain-arvo -pareja. Esimerkissä ensimmäisenä määritellään avain-arvo -parina pyynnön kohteen domain-nimi. Tässä tapauksessa siis 'www.w3.org'.

Näiden perustietojen jälkeen esimerkissä näkyy tyypillisen HTTP-pyyntösanoman sisältämiä muita otsikkotietoja. Otsikkotiedot voivat sisältää tietoa käytössä olevasta asiakassovelluksesta tai käyttöjärjestelmästä sekä muuta tarvittavaa tietoa esimerkiksi siitä, minkälaista koodausta asiakassovellus tukee.

Avain-arvo -parien määrää tai muotoa ei ole rajoitettu, joten www-sovelluspalveluiden osalta otsikkotietoja voidaan hyödyntää myös palvelukohtaisella tiedolla. Tässä on kuitenkin huomioitava, että jokainen palvelukohtainen tieto muokkaa rajapintaa kauemmaksi yhtenäisen rajapinnan ideaalista [Richardson and Ruby, 2007].

2.2.1. HTTP-metodit

Pyyntösanomassa määritelty HTTP-metodi määrittää metodin, joka palvelimella halutaan suorittaa.

Tunniste	Pakollisuus	Kategoria	Kuvaus
OPTIONS	Ei	Idempotentti	Palauttaa listan palvelimen tukemista metodeista.
GET	Kyllä	Turv., idem.	Palauttaa pyydetyn resurssin.
HEAD	Kyllä	Turv., idem.	Palauttaa pyydetyn resurssin otsikkotiedostot ilman runko-osaa.
POST	Ei		Luo uuden aliresurssin pyyntösanomassa määritellyn URI:n alle. Käytännössä palvelin päättää toimenpiteet sisällön perusteella.
PUT	Ei	Idempotentti	Lisää lähetetyn resurssin pyyntösanomassa määritellyn URI:n alle.
DELETE	Ei	Idempotentti	Poistaa pyyntösanomassa määritellyn resurssin.
TRACE	Ei	Idempotentti	Palauttaa lähetetty otsikkotiedot sellaisenaan takaisin. Tarkoitettu verkon diagnostiikkaan.
CONNECT	Ei		Varattu metodi dynaamiseen tunnelointiin.

Taulukko 2: Määrittelynmukaiset HTTP-metodit [NWG, 1999].

Taulukossa 2 on esitetty HTTP 1.1 -määrittelyn nimeämät HTTP-metodit. Huomionarvoisia REST:in mukaisia www-sovelluspalveluita tarkasteltaessa näistä ovat erityisesti OPTIONS, GET, HEAD, POST, PUT ja DELETE. Nämä metodit voidaan jakaa edelleen ominaisuuksiensa perusteella kahteen kategoriaan: turvallinen ja idempotentti [NWG, 1999].

Turvallisten metodien kategoriaan kuuluvien metodien ei tulisi aiheuttaa muutoksia palvelimella. Tässä yhteydessä on kuitenkin hyvä huomata, että tämä on yksinomaan palvelinsovelluksen vastuulla. Käytännössä mikään ei estä palvelinta esimerkiksi poistamaan jotakin resurssia GET-metodin kutsumisen yhteydessä. Tämä on kuitenkin vastoin määrittelyä ja rikkoo yhtenäisen rajapinnan vaatimusta. Www:n luonteesta johtuen asiakassovelluksen on siis vain luotettava siihen, että ei-toivottuja sivuvaikutuksia ei seuraa.

Tämä ei kuitenkaan tarkoita, etteikö GET-metodin kutsusta saisi seurata minkäänlaisia tilaan vaikuttavia muutoksia [Richardson and Ruby, 2007]. Käytännössä esimerkiksi erilaiset sivulatauksia mittaavat laskurit aiheuttavat muutoksen palvelimella GET-metodia kutsuttaessa. Tärkeintä on se, että muutokset eivät ole niin suuria, että ne vaatisivat asiakassovellukselta erityistä tietämystä, jotta se voisi toimia oikein.

Idempotenssilla tarkoitetaan tässä yhteydessä sitä, että metodikutsun lopputulos pysyy samana riippumatta tehtyjen identtisten kutsujen määrästä [NWG, 1999]. Käytännössä siis esimerkiksi DELETE-metodin seurauksena haluttu resurssi poistetaan, ja jos kutsu tehtäisiin uudelleen, pysyisi resurssi edelleen poistettuna [Richardson and Ruby, 2007].

Sovelluksen kannalta ongelmallinen tilanne voi syntyä, jos samaan resurssiin on oikeudet useammalla asiakassovelluksella. Oletetaan, että asiakassovellus A suorittaa GET-metodin ja sen jälkeen DELETE-metodin. Seuraavaksi asiakassovellus B suorittaa GET-metodin samaan resurssiin ja sen jälkeen PUT-metodin. Tämän jälkeen, esimerkiksi verkkovirheestä johtuen, asiakassovellus A suorittaa idempotenssioletuksen pohjalta uudelleen DELETE -metodin kutsun. Tällaisessa, www-ympäristössä tyypillisessä tilanteessa, jossa samaa palvelua käyttää useampi asiakassovellus, esimerkin asiakassovellus B ei voi tyhjentävästi päätellä, että lopputulos on haluttu.

Edellä esitetty ei kuitenkaan sinällään poista idempotenssimäärittelyn oikeutusta, koska HTTP-protokolla on tilaton protokolla. Tällaiset potentiaaliset ongelmatilanteet on siis vain huomioitava sovelluksen suunnittelussa. Esimerkin tilanteeseen yhtenä vaihtoehtona olisi rajoittaa oikeus yksittäisen resurssin DELETE-operaatioon ainoastaan yhdelle asiakkaalle kerrallaan. Toinen vaihtoehto on estää samannimisten resurssien luominen esimerkiksi juoksevaa numerointia käyttäen. Tällöin sovelluksen on kuitenkin mahdollisesti pystyttävä erottelemaan samaan käsitteeseen liittyvät resurssit jollakin mekanismilla toisistaan.

REST:in mukaisen www-sovelluspalvelun näkökulmasta tärkeintä on kuitenkin pyrkiä suunnittelemaan sovelluksesta sellainen, joka mahdollisimman hyvin vastaa HTTP-määrittelyssä määriteltyjä tavoitteita.

POST-metodi ei ole turvallinen, eikä idempotentti. Alkujaan POST-metodi on suunniteltu esimerkiksi HTML-lomakkeen sisältämän tietojoukon lähettämistä silmällä pitäen. POST-metodilla luodaan siis edellä mainitussa tilanteessa uusi aliresurssi. Käytännössä tyypillinen ongelmatilanne on esimerkiksi, jos sama HTML-lomake lähetetään useampaan kertaan. Tällöin PUT-metodista poiketen palvelin tekee päätöksen luodun resurssin sijainnista. POST-metodia käytetäänkin mm. SOAP-palveluissa, joissa HTTP-sanomaa käytetään ainoastaan varsinaisen SOAP-sanoman kuljetuksen vuoraamiseen ja jätetään HTTP:n tarjoamat mahdollisuudet käyttämättä [Richardson and Ruby, 2007].

OPTIONS-metodin käyttöä voidaan laajentaa www-sovelluspalvelun tukemien metodien selvittämisen lisäksi esimerkiksi ajonaikaiseen URI-parametrien päättelyyn. Tällöin asiakassovellus pyytää suorittamaan OPTIONS-metodin ja pyrkii päättelemään palautesanoman sisällöstä parametrien semantiikkaa. [Steiner and Algermissen, 2011]

Myös omien metodien luominen on täysin sallittua. Ainoastaan edellä määriteltyjen varattujen metodien tulee toimia, kuten HTTP-määrittelyssä on kuvattu. [NWG, 1999]

Yhtenäisen rajapinnan vaatimuksen valossa omien metodien luomista tulisi kuitenkin välttää. Luotaessa oma HTTP-metodi luodaan samalla myös rajoite mahdollisille asiakassovelluksille. Tällöin asiakassovelluksen kehittäjän tulee etukäteen huomioida metodin olemassaolo ja tietää metodin aiheuttama toiminnallisuus. Sovelluksen voi kuitenkin toteuttaa myös siten, että omia metodeja käytetään joidenkin erityistapausten hoitamiseen, mutta sovellusta voi toisaalta käyttää perustoiminnallisuuksien osalta HTTP-määrittelyn mukaisilla metodeilla. Käytännössä tällaisille metodeille saattaisi olla tarvetta esimerkiksi yleisten ylläpitotoimien osalta. Esimerkiksi vaikkapa valokuvagallerian kaikkien kuvien siirtäminen arkistoon voisi tapahtua metodilla ARCHIVEALL. Sama toiminnallisuus voidaan saada tosin aikaan myös POST-operaatiota käyttäen. Käytännössä kyse on siis suunnitteluratkaisusta.

2.2.2. HTTP-tilakoodit

Jokaisen HTTP-vastaussanoman tulee sisältää tilakoodi [NWG, 1999].

```
HTTP/1.1 200 OK
Date: Sun, 24 Feb 2013 07:22:53 GMT
Server: Apache
Last-Modified: Fri, 22 Feb 2013 12:10:48 GMT
ETag: "42e2f-2f00-4d64f14ec80c2"
Accept-Ranges: bytes
Content-Length: 12032
Connection: close
Content-Type: text/html; charset=utf-8
```

Taulukko 3: Esimerkki HTTP -vastaussanoman otsikkotiedoista.

HTTP-vastaussanoman rakenne on sama kuin pyyntösanoman rakenne. Vastaussanoma koostuu otsikkotiedoista ja mahdollisesta runko-osasta. Taulukon 3 ensimmäisellä rivillä voidaan kuitenkin nähdä tärkein ero: metodimäärityksen sijaan palautetaan protokollatunnisteen perässä tilakoodi. Esimerkissä on onnistumista kuvaava tilakoodi '200 OK'. Tilakoodi voidaan edelleen jakaa numeeriseen koodin '200' ja lyhyeen tekstuaaliseen kuvaukseen 'OK'. Lisäksi vastaussanoman runko-osassa voidaan palauttaa vaikkapa luotu

resurssi. Lisäksi virhetilanteessa tulee vastaussanoman runkoon liittää tarkempi kuvaus virheestä [NWG, 1999].

Koodiryhmä	Kuvaus
1XX	Informatiiviset tilakoodit.
2XX	Onnistumista ilmaisevat tilakoodit.
3XX	Uudelleenohjaukseen liittyvät tilakoodit.
4XX	Asiakassovelluksen virheestä so. virheellisestä HTTP-pyyntösanomasta kertovat tilakoodit. Tämän ryhmän koodien mukana tulee aina palauttaa kuvaus virheestä ja virheen pysyvyydestä.
5XX	Palvelinsovelluksen virheestä kertovat tilakoodit. Tämän ryhmän koodien mukana tulee aina palauttaa kuvaus virheestä ja virheen pysyvyydestä.

Taulukko 4: HTTP-tilakoodiryhmät [NWG, 1999].

HTTP-määrittelyssä tilakoodit on jaettu myös tilakoodiryhmiin. Tilakoodiryhmät on esitetty taulukossa 4. Tilakoodin ensimmäinen numero toimii ryhmän tunnisteena. Määrittelyssä on lisäksi kuvattu jokaisesta ryhmästä joitakin yleisiä tilanteita kuvaavat tilakoodit. Nämä koodit ovat ns. varattuja koodeja, ja ne tulee palauttaa ainoastaan määrittelyssä määritellyissä tilanteissa. Käytännön tilanteissa ei kuitenkaan ole aina yksikäsitteistä, mikä on oikea tilakoodi. Esimerkiksi tietoturvan näkökulmasta voi olla perusteltua palauttaa tilakoodi '404 NOT FOUND', koodin '403 FORBIDDEN' sijaan, vaikka palvelinsovellus tietäisikin resurssin olevan olemassa. Tällöin ei anneta mitään ylimääräistä tietoa, joka voisi olla vahingollista mahdollisen vihamielisen käyttäjän hallussa.

HTTP-määrittely ei muulla tavoin rajoita tilakoodien käyttöä. Tämä tarkoittaa siis sitä, että on täysin sallittua luoda uusia tilakoodeja, joiden tunniste eroaa varattujen koodien tunnisteesta. Myös uusia tilakoodiryhmiä voidaan luoda.

Uusia tilakoodeja luotaessa ne tulisi kuitenkin nimetä siten, että ne merkitykseltään sopivat koodinumeron määrittämään ryhmään.

Jos esimääritellyt ryhmät eivät tule kysymykseen tai halutaan luoda sovelluskohtainen paluukoodiryhmä, tulee luoda uusi tilakoodiryhmä. Uusia tilakoodeja suunniteltaessa tulee kuitenkin huomioida, että ne tarvitsevat asiakassovellukselta HTTP-määrittelyn ulkopuolista tietoa, joka on ristiriidassa yhtenäisen rajapinnan vaatimuksen kanssa.

2.2.3. HTTP-otsikkotiedot

HTTP-otsikkotiedot muodostuvat avain-arvopareista. HTTP-määrittelyssä itsessään on määritelty runsas joukko erilaista otsikkotietoja. Otsikkotiedot voivat olla joko pakollisia tai valinnaisia. Pakollinen otsikkotieto on esimerkiksi aiemmin tässä luvussa kuvattu 'Host'. Valinnaisia otsikkotietoja ovat esimerkiksi 'Content-Type' ja' Etag'. [NWG, 1999]

Content-Type määrittää nimensä mukaisesti missä muodossa HTTP-sanoman runko-osan sisältö on. Esimerkiksi HTML-sivun Content-Type voi olla esimerkiksi 'text/html; charset=utf-8'. Ensin kerrotaan varsinainen muoto ja rakenne, tässä tapauksessa siis tekstimuotoinen ja HTML-rakenteinen dokumentti, ja lopuksi kerrotaan vielä käytetty merkistökoodaus: 'charset=utf-8'.

ETag-otsikkotieto määrittää palautetulle resurssille yksilöivän tunnisteen. Aina resurssin muuttuessa tulisi siis myös ETag tunnisteen muuttua. ETag:in avulla palvelin voi esimerkiksi päätellä, kannattaako pyydettyä resurssia lähettää lainkaan, jos Etagin perusteella tiedetään, että asiakassovelluksella on jo tuorein versio käytössä. [Richardson and Ruby, 2007]

Varsinaisessa HTML-määrittelyssä kuvattujen otsikkotietojen lisäksi on olemassa iso joukko erilaisia määrittelyn ulkopuolisia otsikkotietoja. Ehkä laajimmin käytössä olevat määrittelyn ulkopuoliset otsikkotiedot ovat 'Cookie' ja 'Set-Cookie'.

Cookie-otsikkotietoa käytetään erityisesti kirjautumisen vaativissa verkkopalveluissa. Tällöin Set-Cookie-otsikkotiedon arvona palvelin välittää asiakassovellukselle numeeriseen tunnisteen, jonka asiakassovellus tallentaa levylle ja lähettää tämän jälkeen jokaisen kyseiselle palvelimelle lähetetyn HTTP-pyynnön Cookie-otsikkotietona. Tämän jälkeen palvelinsovellus voi kytkeä pyynnöt oikeisiin resursseihin ja tilatietoon. Tätä mekanismia kutsutaan yleisesti istunnoksi.

REST:in näkökulmasta Cookie on ongelmallinen. Tilattomuuden vaatimus ei toteudu, koska palvelimen tila ja HTTP-pyynnöt sidotaan numerosarjalla toisiinsa, so. palvelin ilmoittaa asiakassovellukselle, että se ei enää hyväksy pyyntöjä, joita se hyväksyi ennen kirjautumisoperaatiota. REST:in mukaisessa www-sovelluspalvelussa kaikki mahdollinen tilatieto tulisikin säilyttää asiakassovelluksessa, jotta ei menetetä laajennettavuuden tuomia etuja. [Richardson and Ruby, 2007]

Cookie-otsikkotietoa voidaan toki käyttää muuhunkin tarkoitukseen kuin edellä kuvattuun istunnon luomiseen. Käytännössä jokaisen pyynnön mukana lähetetty tieto voi olla mitä tahansa.

HTTP-otsikkotietoja voi siis keksiä vapaavalintaisesti myös yksittäisen www-sovelluspalvelun tarpeisiin. Näidenkin osalta on kuitenkin muistettava, että tällöin asiakassovelluksen on tunnettava ja osattava näitä otsikkotietoja käyttää.

2.3. ROA

REST on arkkitehtuurinen tyyli eli joukko edellä määriteltyjä rajoitteita. REST sinällään ei siis vielä kerro, minkälaisen arkkitehtuurin avulla voidaan toteuttaa REST:in mukainen www-sovelluspalvelu. Yksi vaihtoehto tähän on ROA. [Richardson and Ruby, 2007]

2.3.1. URI

URI on merkkijono, jolla kerrotaan jonkin resurssin paikka ja nimi webissä [Berners-Lee, 1996]. Tässä yhteydessä webillä tarkoitetaan universaalia nimiavaruutta. URI:n erityistapauksia ovat URL ja URN [NWG, 1994]. URL kertoo jonkin olion sijainnin ja käytettävän protokollan. URN taas on jonkin olion, esimerkiksi teoksen, yksilöivä nimi. Tämän tutkielman kontekstissa ei kuitenkaan ole tarvetta käsitellä näitä erityistapauksia tämän tarkemmin. Riittää todeta, että yleisesti URI osoittaa sekä olion sijaintia (jos se on saatavilla), että nimeä.

```
URI { skeema:polku [ ? kysely ]  } [ # tarkenne ]
```

Taulukko 5: URI muodostuu skeemasta ja polusta, sekä valinnaisesti kyselyosasta. Lisäksi URI:n perään voidaan lisätä tarkenneosa [NWG, 1994].

Taulukosta 5 näemme URI:n perusrakenteen. Skeemaosa ja polkuosa on erotettu toisistaan kaksoispisteellä. Esimerkiksi tavanomaisesti skeemaosa on 'http' tai 'https', joka kertoo käytetyn protokollan. URI:n kohdalla kyse on kuitenkin puhtaasti nimiavaruudellisesta määreestä, jossa polkuosan nimi (ja sijainti) on määritelty. Kysymysmerkillä voidaan edelleen erottaa myös kyselyosa. Kyselyosa muodostuu tyypillisesti nimi-arvo -pareista yhtäsuuruusmerkillä erotettuna. Kyselyosalla voidaan välittää esimerkiksi hakukoneen hakulauseke: 'http://www.esimerkki.fi/haku? hakulauseke=gradu'.

Lisäksi '#'-merkillä voidaan liittää URI:n perään tarkenneosa. Www:ssä tarkennetta ei lähetetä verkon yli, vaan se on käytössä ainoastaan asiakassovelluksessa.

Kaikki edellä määritellyt erikoismerkit ovat ns. varattuja merkkejä ja niitä ei voi sellaisenaan käyttää URI:ssa. Varatut merkit voidaan esittää koodinvaihtomerkin '%' ja sopivan ASCII -numerosarjan avulla. Myös vinoviiva '/' ja piste '.' ovat varattuja merkkejä ja www:ssä käytettyjen URI:en kannalta erityisen tärkeitä. Vinoviivaa käytetään erottamaan hierarkkisia rakenteita ja pistettä erottamaan domain -nimen eri osia. Myös joitakin muita varattuja merkkejä on olemassa.

2.3.2. Resurssit ja representaatiot

W3C:n [2004b] määritelmän mukaan resurssi voi tarkoittaa mitä tahansa oliota, joka voidaan identifioida URI:lla. Jokaisella resurssilla tulee siis olla vähintään yksi URI [Berners-Lee, 1996]. Resursseja voivat siis olla vaikkapa tämä tutkielma, uusi Ubuntu

Linux -versio tai Porschen uusin automalli. Resurssit voivat olla myös abstraktimpeja käsitteitä, kuten älykkyys tai avaruuden valloitus.

ROA:ssa URI:lla ja resurssilla on myös tiiviimpi yhteys. URI:n tulisi olla muotoiltu siten, että siitä itsessään voi jo tehdä päätelmiä resurssin sisällöstä [Richardson and Ruby, 2007].

http://www.esimerkki.fi/tilaukset/2013/02/25/98765

Taulukko 6: Esimerkki URI:sta.

Taulukossa 6 esitetystä URI:sta voimme helposti päätellä, että resurssi, johon URI osoittaa, on tilaus, joka on tehty 25.2.2013 ja jonka tilausnumero on 98765. Lisäksi voimme helposti päättelyä jatkamalla tulla johtopäätökseen, että vaihtamalla numeron 25 numeroksi 24 saamme edellisen päivän tilaukset näkyville. Tämä päättely on ihmiselle helppo, mutta myös sovellukselle mahdollinen. Sovelluksia varten voidaan myös kehittää niille soveltuvampia rakenteita.

Tosin on huomattava, että URI:n takana oleva resurssi, riippuu sovelluksen toteutuksesta. URI voi johtaa aivan johonkin muuhun, vaikkapa paikallisen yrittäjän mainossivustolle. Toisaalta myös URI ilman tilausnumeroa listaa tilaukset ainoastaan, jos listaustoiminto on toteutettu. On kuitenkin ilmiselvää, että edellä esitetyn kaltainen kuvailemaan pyrkivä URI mahdollistaa paremman ymmärryksen taustalla olevista resursseista ja niihin liittyvästä jaottelusta.

URI:lle ei ole määritelty maksimipituutta [NWG, 1994]. Käytännössä kuitenkin eri sovellukset hyväksyvät vaihtelevan pituisia URI:a [BOUTEL, 2006]. Kuvailevien URI:en eräänä suunnitteluhaasteena onkin se, että niiden muodostuessa hyvin pitkiksi niiden luettavuus ihmiskäyttäjien osalta samalla luonnollisesti heikkenee. Tällöin tulisi hierarkian keinoin pyrkiä vähentämään yksittäisen URI:n pituutta. Toisaalta jotkin sovellukset, vaikkapa erilaiset kartastot, vaativat huomattavan määrän tietoa kyselyosassa, jotta haluttu karttapaikka voidaan yksikäsitteisesti kohdentaa. Näissä tapauksissa voi olla välttämätöntä luopua luettavuuden ideaalista ja pyrkiä tiivistämään kyselyosassa esitettyä tilatietoa.

Representaatiot ovat resurssien bittiesityksiä. Tämän tutkielman representaatio voi siis olla pdf-tiedosto, jonka olet ehkä juuri ladannut verkosta tai vaihtoehtoisesti representaatio voi olla kirjaston tutkielmatietokannasta löytyvä viittaus hyllypaikkaan, josta painettu versio tutkielmasta löytyy.

Representaatio voi olla myös www-sovelluspalvelun kautta toteutettu käyttöliittymä johonkin fyysiseen laitteeseen. Tällainen voi olla esimerkiksi www:n kautta toteutettu robottikäsivarren ohjaus [Richtr and Farana, 2011]. Tällaisten laitteiden osalta ohjauskontrol-

lit on monesti järkevä rajata tiettyyn päätelaitteeseen tai laitteen käyttöön koulutetulle henkilöstölle. Mikään ei kuitenkaan sinällään estä laitteen käytön sallimista kaikille www:n käyttäjille. Joka tapauksessa www-sovelluspalveluita käytettäessä skaalautuvuuden tuomat edut ovat ilmeiset. Ohjaustoiminnallisuus voidaan sallia konfiguroimalla mille tahansa verkkoselaimella varustetulle tavalliselle tietokoneelle erityisen ohjauslaitteen tilaamisen sijasta.

2.3.3. Osoitteellisuus, tilattomuus, linkitettävyys ja yhtenäinen rajapinta

ROA:n neljä päävaatimusta ovat osoitteellisuus, tilattomuus, linkitettävyys ja yhtenäinen rajapinta [Richardson and Ruby, 2007]. Tilattomuutta ja yhtenäistä rajapintaa on käsitelty jo edellä yleisellä tasolla. Jotta ROA:n pohjalta voisi tehdä aidosti REST:in mukaisen www-sovelluspalvelun, niin nämä käsitteet vaativat joitakin tarkennuksia.

Osoitteellisuudella tarkoitetaan sitä, että jokainen mahdollisesti kiinnostava asia tulee esittää resurssina eli jokaisella asialla on URI, jolla siihen voi viitata. Tällöin URI:n voi laittaa talteen ja palata myöhemmin samalle sivulle. Toisaalta on paljon verkkosovelluksia, joissa samalla URI:lla saadaan eri resurssien representaatiot esiin. Tällöin URI ei viittaa itse resurssiin, vaan sovelluksen tiettyyn tilaan. Haluttu representaatio on mahdollista saada esille suorittamalla tietty sarja sopivia toiminnallisuuksia. URI ei kuitenkaan itsessään tällöin kerro, mikä resurssi oli haluttu, vaan tämä pitää olla muuten tiedossa.

Osoitteellisuuden vaatimus on yksinkertaistava tekijä. Tällöin kaikki resurssit on suoraan saavutettavissa. Yksinkertaisuus (ja yhtenäisyys) on myös yksi syy www:n suosioon [Richardson and Ruby, 2007]. Www:n olemassaolon aikana on muodostunut tavanomainen viittaustapa näyttää jokin URI vaikkapa televisiossa, ja jonka käyttäjä voi tämän jälkeen syöttää tietokoneensa www-selaimeen, ei olisi ollut ennen www:tä mahdollinen. Esimerkiksi jos televisiossa näytettiin jonkin kirjan yksilöivä tunniste, niin tämän jälkeen tuli ensin selvittää, missä kirjastossa kyseinen teos mahdollisesti on saatavilla, onko se lainassa jne. Poistamalla resurssiin viittaavat URI:t rikotaan myös www:n alkuperäisen suosion mahdollistanut yksinkertaisuus.

Tilattomuus on jo edellä käsitelty REST:n vaatimuksena ja toisaalta HTTP-protokollan ominaisuutena. Tilattomuus ROA:n mukaisessa www-sovelluspalvelussa tarkoittaa kuitenkin myös edellä esittämääni osoitteellisuuden vaatimuksesta seuraavaa tilannetta, jossa jokainen HTTP-pyyntö tapahtuu erillisenä suhteessa muihin pyyntöihin. Tällöin jokainen mahdollisesti tarvittava tieto tulee lähettää jokaisen pyynnön mukana erikseen. Tästä seuraa edelleen se, että mahdollinen sovelluksen vaatima tilatieto tulee säilyttää asiakassovelluksessa. Tämä tukee myös edelleen yksinkertaisuuden periaatetta: asiakassovelluksen ei tarvitse tietää missä järjestyksessä pyyntöjä tulee suorittaa, vaan ne voidaan suorittaa siinä järjestyksessä, kun on tilanteen kannalta tarpeellista.

Tilattomuudesta seuraa myös se, että palvelimen ei tarvitse huolehtia aikakatkaisusta, eikä muustakaan eri pyyntöjen yhdistämiseen liittyvistä seikoista. Tietoturvan näkökulmasta on tietenkin tarpeellista esimerkiksi salata verkkoliikenne, jos jonkinlainen tunnistautuminen on tarpeen. Joka tapauksessa palvelun käytön lopettaminen suoritetaan asiakassovelluksessa esimerkiksi sulkemalla selain tai kirjautumalla ulos, jonka jälkeen palveluun ei enää saada yhteyttä ilman asianmukaisia tunnistetietoja.

Ainoa palvelinsovelluksen tilatieto, joka ROA:ssa on sallittu, on yksittäisen resurssin tila. Esimerkiksi kirjoittaessani tätä tutkielmaa tallennettu versio siirtyy jokaisen tallennuksen yhteydessä Ubuntu One -pilvipalveluun.

Tällöin myös tiedostojen käsittelyyn käytetyn verkkosovelluksen tila edellä esitetyssä mielessä muuttuu. Sama URI viittaa edelleen samaan resurssiin, mutta representaatio resurssista on muuttunut. Samalla tavalla sovelluksen tila muuttuu poistettaessa vanhoja tai lisättäessä uusia resursseja.

Linkitettävyys tarkoittaa käytännössä sitä, että tarjotut representaatiot ovat hypermediasivuja, jotka sisältävät linkkejä toisiin sivuihin. Tämä tarkoittaa esimerkiksi sitä, että matkapuhelimen yleiset tuotetiedot sisältävällä sivulla voi olla hyperlinkki vaikkapa puhelimen tekniset yksityiskohdat sisältävälle sivulle. Tällöin palvelinsovellus antaa vihjeitä asiakassovellukselle siitä, mihin seuraavaksi on mahdollista siirtyä. Tavallisilla HTML-sivuilla tämä on nykyisin tietysti aivan arkinen tapa tehdä asioita. Esimerkiksi hakukonetta käytettäessä saadaan listaus hyperlinkkejä, joiden perusteella voidaan edelleen siirtyä hakukoneen mielestä hakua vastaaville sivuille [Richardson and Ruby, 2007].

Hyperlinkit eivät kuitenkaan mitenkään rajoita asiakassovelluksen valintoja. Esimerkiksi hakukoneen käyttäjä voi halutessaan siirtyä aivan jollekin muulle sivulle kuin hakukoneen ehdottamalle. Esimerkiksi gradun lähteitä hakiessa voidaan mahdollisesti todeta jo linkkien perusteella, että tulos ei ollut haluttu. Tällöin voidaan tehdä vaikkapa uusi haku tai käyttää toista hakukonetta.

Täysin ohjelmallisilla asiakassovelluksilla tilanne on aivan sama. Päättelyprosessi poikkeaa kuitenkin huomattavasti edellä esitetystä. Tällöin onkin käytännössä välttämätöntä ohjelmoida asiakassovellus tietyn palvelinsovelluksen tarjoamaa linkitysmallia silmällä pitäen. Mikään ei tietysti estä kehittämästä myös yleistä sovelluskehystä, jota voidaan käyttää useissa palvelinsovelluksissa. Tällöin asiakassovelluksia voidaan käyttää eri palvelinsovellusten kanssa pienemmillä muutoksilla. Vaarana tällaisissa sovelluskehyksissä piilee kuitenkin se, että ne voivat potentiaalisesti lisätä sovelluksen monimutkaisuutta ja vähentää näin yleisen rajapinnan tuomia etuja.

Ongelmatilanne syntyy vasta siinä vaiheessa, jos palvelinsovelluksen tarjoama linkitysjärjestelmä muuttuu. Tällöin asiakassovellusta täytyy muuttaa. Kuitenkin jos jokaiselle

resurssille määritellään edellä esitetyllä tavalla vähintään yksi oma URI, niin hierarkiaan tehtyjen muutosten muuttaminen myös asiakassovellukselle pitäisi olla tyypillisesti triviaali toimenpide. Tilanne on aivan toinen, jos käytetty rajapinta perustuu esimerkiksi johonkin aivan omaan protokollaan, kuten myöhemmin tarkemmin käsiteltävään SOAP:iin.

ROA:n vaatimuksiin ei kuulu se, että sovelluksen tulee käyttää HTTP-rajapintaa. Vaatimuksena on ainoastaan se, että rajapinnan tulee olla yhtenäinen siinä mielessä, että se on kaikkien tiedossa. Tästä syystä HTTP:n mahdollistama omien metodien lisääminen on ROA:ssa kielletty, koska tällöin käytetyt metodit eivät ole samat kaikkialla ja niiden käyttö vaatii erikseen hankittua tietoisuutta niiden toiminnasta. [Richardson and Ruby, 2007]

Toisaalta voidaan toki todeta, että vaikka metodien nimet olisivatkin samat, niiden varsinainen toiminta voi poiketa toisistaan hyvinkin paljon. Erityisesti tämä ongelma nousee POST-metodin yhteydessä esiin, jolloin HTTP-protokollaa voidaan käyttää ainoastaan jonkin muun protokollan kuorrutuksena.

Lisäksi on hyvä huomata, että ihmiselle tyypillinen suora www:n käyttötapa on käyttää jotakin selainohjelmistoa. Tämä asettaa rajapinnan yhtenäisyysvaatimukselle tiettyjä haasteita sen johdosta, että suurin osa selaimista tukee ainoastaan HTTP-metodeja GET ja POST. Myös esimerkiksi tulevan HTML 5.1 -määrittelyn, tätä kirjoitettaessa saatavilla olevan version, mukaan HTML-lomakkeen lähetysmetodeiksi hyväksytään ainoastaan GET ja POST [W3C, 2013]. Ohjelmallisissa asiakassovelluksilla kaikkien metodien käyttö on kuitenkin mahdollista.

2.3.4. ATOM

Resurssipohjaisen arkkitehtuurin mukaisen www-sovelluspalvelun toteutuksessa voidaan käyttää apuna erilaisia apuvälineitä. Tällaisia ovat esimerkiksi representaatioiden esittämiseen soveltuvat kuvauskielet, kuten XML ja JSON [Richardson and Ruby, 2007]. Lisäksi käsite Atom kokoaa yhteen kaksi itsenäistä määrittelyä: Atom Syndication Format [NWG, 2005] ja Atom Publishing Protocol [NWG, 2007].

```
<?xml version="1.0" encoding="utf-8"?>
 <feed xmlns="http://www.w3.org/2005/Atom">

   <title>Example Feed</title>
   <link href="http://example.org/"/>
     <updated>2003-12-13T18:30:02Z</updated>
     <author>
      <name>John Doe</name>
     </author>
    <id>urn:uuid:60a76c80-d399-11d9-b93C-0003939e0af6</id>

    <entry>
     <title>Atom-Powered Robots Run Amok</title>
       <link href="http://example.org/2003/12/13/atom03"/>
     <id>urn:uuid:1225c695-cfb8-4ebb-aaaa-80da344efa6a</id>
       <updated>2003-12-13T18:30:02Z</updated>
               <summary>Some text.</summary>
     </entry>

 </feed>
```

Taulukko 7: Esimerkki Atom Syndication Format -muotoisesta dokumentista [NWG, 2005].

Atom Syndication Format on XML -kuvauskieleen pohjautuva kieli, jonka avulla voidaan ryhmitellä web-sisältöä mm. blogeista ja muista uutisvirroista www-sivustoille [NWG, 2005].

Taulukossa 7 näemme esimerkin Atom Syndication Format -muotoisesta dokumentista. Se sisältää uutisvirran ominaisuuksista kertovia lohkoja. Näitä ovat esimerkiksi aikaleima (milloin uutisvirta on viimeksi päivitetty), kirjoittajan nimi ja artikkelin nimi. Esimerkin perusteella voidaan vaivattomasti päätellä, että kyseinen formaatti on käyttökelpoinen myös REST:in mukaisen www-sovelluspalvelun representaatioden esittämisen välineenä.

Atom Publishing Protocol on HTTP-protokollaa ja XML-versiota 1.0 käyttävä resurssien julkaisemiseen ja päivittämiseen tarkoitettu protokolla. Protokolla tukee resurssikokoelmia, palveluita sekä resurssien lisäämistä, muokkaamista ja poistamista. [NWG, 2007]

```
POST /edit/ HTTP/1.1
    Host: example.org
    User-Agent: Thingio/1.0
    Authorization: Basic ZGFmZnk6c2VjZXJldA==
    Content-Type: application/atom+xml;type=entry
    Content-Length: nnn
    Slug: First Post

    <?xml version="1.0"?>
    <entry xmlns="http://www.w3.org/2005/Atom">
      <title>Atom-Powered Robots Run Amok</title>
      <id>urn:uuid:1225c695-cfb8-4ebb-aaaa-80da344efa6a</id>
      <updated>2003-12-13T18:30:02Z</updated>
      <author><name>John Doe</name></author>
      <content>Some text.</content>
    </entry>

HTTP/1.1 201 Created
    Date: Fri, 7 Oct 2005 17:17:11 GMT
    Content-Length: nnn
    Content-Type: application/atom+xml;type=entry;charset="utf-8"
    Location: http://example.org/edit/first-post.atom
    ETag: "c180de84f991g8"
```

Taulukko 8: HTTP -pyyntö- ja vastaussanomat luotaessa uusi resurssi Atom Publishing Protocol -protokollaa käyttäen [NWG, 2007].

Taulukossa 8 nähdään uuden resurssin luonti Atom Publishing Protocol -protokollalla HTTP-metodia POST käyttäen. HTTP-otsikkotiedoissa nähdään avaimen 'Content-Type' arvona protokollan käyttämä tunniste: 'application/atom+xml;type=entry'. Protokolla soveltuu hyvin REST:in mukaisen www-sovelluspalvelun kehitystyön tarpeisiin.

Lisäksi protokollan määrittely käyttää HTTP-protokollan käytöstä johtuen samankaltaista terminologiaa kuin REST:in mukaisten www-sovelluspalveluiden yhteydessä käytetään. Tämä helpottaa protokollan soveltamista käytännön REST-toteutuksissa.

2.3.5. WADL

WADL-kuvauskieli on REST:in mukaisten www-sovelluspalvelujen kuvailemiseen käytetty XML-pohjainen kieli. Asiakassovellus voi ladata www-sovelluspalvelun

WADL-muodossa olevan kuvauksen ja ”omaksua” sen avulla palvelun ominaisuudet. Toimintaperiaate on yleisellä tasolla sama kuin esimerkiksi SOAP-tyyppisellä palvelulla, josta voidaan ladata palvelun kuvauksen sisältävä WSDL-muodossa oleva dokumentti ja generoida kuvauksen pohjalta palvelua käyttävät asiakaskomponentit. [Richardson and Ruby, 2007]

```
<?xml version="1.0"?>
<application xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://wadl.dev.java.net/2009/02 wadl.xsd"
xmlns:tns="urn:yahoo:yn"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:yn="urn:yahoo:yn"
xmlns:ya="urn:yahoo:api"
xmlns="http://wadl.dev.java.net/2009/02">

  <grammars>
   <include href="NewsSearchResponse.xsd"/>
   <include href="Error.xsd"/>
  </grammars>

  <resources base="http://api.search.yahoo.com/NewsSearchService/V1/">
   <resource path="newsSearch">
    <method name="GET" id="search">
     <request>
      <param name="appid" type="xsd:string" style="query" required="true"/>
      <param name="query" type="xsd:string" style="query" required="true"/>
      <param name="type" style="query" default="all">
         <option value="all"/>
         <option value="any"/>
         <option value="phrase"/>
      </param>
      <param name="results" style="query" type="xsd:int" default="10"/>
      <param name="start" style="query" type="xsd:int" default="1"/>
      <param name="sort" style="query" default="rank">
         <option value="rank"/>
         <option value="date"/>
      </param>
```

```
     <param name="language" style="query" type="xsd:string"/>
    </request>

    <response status="200">
      <representation mediaType="application/xml" element="yn:ResultSet"/>
    </response>
    <response status="400">
        <representation mediaType="application/xml" element="ya:Error"/>
    </response>

   </method>
  </resource>
 </resources>
</application>
```

Taulukko 9: Esimerkki WADL-kuvauskielestä [W3C, 2009].

Taulukossa 9 on esitetty esimerkki WADL-kuvauskielestä. Esimerkki on kuvaus Yahoon uutishakupalvelun hakutoiminnosta.

Esimerkissä määritellään aluksi käytetyt kieliopit 'grammars'-lohkossa. Tämän jälkeen määritellään resurssi ja sen polku. Lisäksi määritellään tuettu HTTP-metodi: GET. 'request'-lohkossa määritellään hyväksytyt URI-parametrit erilaisiin hakutulosten rajausoperaatioihin. Lopuksi 'response'-lohkossa esimääritellään HTTP-statuskoodit onnistuneelle ja epäonnistuneelle hakuoperaatiolle. Lisäksi määritellään palautettavan representaation formaatti.

REST:in mukaisten www-sovelluspalvelujen suhteellisesta yksinkertaisuudesta johtuen WADL:n kaltaisen kuvauskielen olemassaolo ei kuitenkaan ole aivan niin välttämätön kuin esimerkiksi WSDL on SOAP-tyyppisille palveluille [Richardson and Ruby, 2007].

2.4. RMM

Aikaisemmissa tämän luvun kohdissa olen esittänyt erilaisia vaatimuksia, jotka www-sovelluspalvelun tulee täyttää ollakseen REST:in mukainen. Nämä voidaan esittää yhteenvetona RMM:n avulla.

RMM on neliportainen malli, jonka jokaisella tasolla kasvatetaan vaatimusten määrää yhdellä edellä kuvatuista ominaisuuksista. Tasot on numeroitu välille 0-3.

Tasolla 0 käytännössä ainoa vaatimus on se, että palvelu toimii www:ssä. Tyypillisesti tämä tarkoittaa sitä, että käytössä on yksi URI ja yksi HTTP-metodi [Richardson, 2008]. Tyypillisesti käytetty HTTP-metodi on POST.

Tason 1 sovelluksessa jokaiselle resurssille on oma URI [Richardson, 2008]. Muilta osin sovellus toimii kuten tasolla 0. Käytännössä siis esimerkiksi www:ssä toimivan valokuvagallerian jokaisella valokuvalla voisi olla oma yksilöivä URI. Kaikki valokuvaan kohdistuvat operaatiot, esimerkiksi valokuvan poistaminen, toteutetaan HTTP-rajapinnan ulkopuolella.

Tason 2 www-sovelluspalvelussa jokainen URI tukee lisäksi useita HTTP-metodeja [Richardson, 2008]. Tällainen www-sovelluspalvelu täyttää jo monelta osin edellä kuvaamani REST:in mukaisen www-sovelluspalvelun vaatimukset. Kaikki kiinnostava tieto on saavutettavissa URI:en kautta ja toisaalta yleisen rajapinnan vaatimus täyttyy aidon HTTP-metodien hyödyntämisen kautta.

Kuitenkin vasta tasolla 3 www-sovelluspalvelun voidaan katsoa olevan aidosti REST:in mukainen. Tästä huomauttaa myös Fielding [2008]. Tason 3 www-sovelluspalvelu täyttää myös yhdistettävyyden kriteerin. Tällöin palautettu resurssi sisältää tiedon mahdollisista toiminnoista linkkeinä, siis URI:na.

```
POST /slots/1234 HTTP/1.1
[various other headers]

<appointmentRequest>
  <patient id = "jsmith"/>
</appointmentRequest>

HTTP/1.1 201 Created
Location: http://royalhope.nhs.uk/slots/1234/appointment
[various headers]

<appointment>
  <slot id = "1234" doctor = "mjones" start = "1400" end = "1450"/>
  <patient id = "jsmith"/>
  <link rel = "/linkrels/appointment/cancel"
        uri = "/slots/1234/appointment"/>
  <link rel = "/linkrels/appointment/addTest"
        uri = "/slots/1234/appointment/tests"/>
  <link rel = "self"
        uri = "/slots/1234/appointment"/>
  <link rel = "/linkrels/appointment/changeTime"
        uri = "/doctors/mjones/slots?date=20100104@status=open"/>
  <link rel = "/linkrels/appointment/updateContactInfo"
        uri = "/patients/jsmith/contactInfo"/>
  <link rel = "/linkrels/help"
        uri = "/help/appointment"/>
</appointment>
```

Taulukko 10: Esimerkki tason 3 www-sovelluspalvelun HTTP-pyyntö- ja -vastaussanomasta [Fowler, 2010].

Taulukossa 10 on esimerkki sairaalan ajanvarauspalvelun yksittäisestä toiminnosta. Ensin on lähetetty HTTP-pyyntö sovellukselle varata lääkärin vastaanottoaika asiakkaalle, jonka tunniste on 'jsmith'. Pyyntösanomaan saatuun vastaussanomaan sisältyy tiedot varatusta ajasta ja lisäksi mahdollisista jatko-operaatiosta, kuten ajan peruutus, (laboratorio)kokeen lisääminen, varauksen ajankohdan muuttaminen, yhteystietojen

muuttaminen ja ohjetoiminto. Lisäksi linkkien joukossa on linkki luotuun resurssiin tunnisteella 'self'.

Tällaisen rakenteen, ja siis REST:in mukaisten www-sovelluspalveluiden, etuna on se, että URI:t ja rajapinta ovat erotetut toisistaan. Käytännössä tämä tarkoittaa sitä, että URI:t voivat muuttua ilman, että varsinaiseen sovelluslogiikkaan täytyy asiakassovelluksessa tehdä muutoksia. Ainoastaan ns. aloituspisteen URI:n muuttuessa asiakassovellus vaatii muutoksia. Tämäkin tilanne voidaan tosin trivialisoida suunnittelemalla asiakassovellus siten, että aloituspisteen URI on konfiguroitavissa.

REST:in mukaisen www-sovelluspalvelun vahvuudet perustuvat sovelluslogiikan yksinkertaistamiseen ja palauttamiseen takaisin niihin periaatteisiin, jotka ovat tehneet www:stä suositun. Toisaalta ne myös sietävät muutoksia hyvin, joten palvelinsovelluksen muuttuessa ei kaikkia asiakassovelluksia tarvitse välttämättä muuttaa. Www:stä löytyy paljon www-sovelluspalveluita, jotka väittävät olevansa REST:in mukaisia, mutta eivät sitä aidosti ole. Aidosti REST:in mukaisen www-sovelluspalvelun tulee täyttää edellä kuvaamani ja perustelemani kriteerit.

3. SOAP:ista REST:in mukaiseen www-sovelluspalveluun

Richardson ja Ruby [2007] vetävät suuntaviivoja ROA:n avulla siitä, millä tavalla edellisissä luvuissa kuvattu REST:in mukainen www-sovelluspalvelu tulisi suunnitella. Lisäksi Laitkorpi ja muut [2009] ovat liittäneet näihin ohjenuoriin mallipohjaisuuden ja kehittäneet menetelmää edelleen mekaanisempaan suuntaan. Tarkoituksenani onkin tähän malliin nojautuen osoittaa, että sopivin lisäyksin mallia on mahdollista laajentaa tukemaan olemassa olevien SOAP:in mukaisten www-sovelluspalvelujen muuntamista REST:in mukaisiksi palveluiksi.

Alan teollisuudessa on tavallista, että asiakas vaatii omien liiketoiminnallisten tavoitteiden johdosta tiettyjen teknologioiden käyttöä. Tämän muunnosmallin tavoitteena onkin edistää ymmärrystä, toisaalta resurssipohjaisuuden eroista suhteessa palvelupohjaisuuteen, ja toisaalta vastata asiakastoiveiden täyttämisen asettamaan haasteeseen muunnettaessa olemassa olevia palveluita SOAP:ista REST:in mukaisiksi.

Www:stä löytyy joukko työkaluja, jotka väittävät tekevänsä muunnoksen SOAP:ista REST:in mukaiseksi automaattisesti. Tällainen on esimerkiksi WebServiceEngine [2013]. Lisäksi olemassa olevaa SOAP-pohjaista www-sovelluspalvelua voidaan täydentää esikäsittelijällä, joka muuntaa tulevat REST-muotoiset HTTP-pyynnöt taustalla olevan SOAP-palvelun ymmärtämään muotoon [O'Neill, 2008]. Kumpikaan näistä tavoista ei kuitenkaan tuota REST:in mukaista lopputulosta.

Ensimmäisessä jätetään huomiotta se, että SOAP-palvelu ei perustu resurssipohjaiseen arkkitehtuuriin. SOAP:in mukaisesti toteutetut www-sovelluspalvelut on usein suunniteltu samalla myös SOA:n mukaisiksi. Tällöin itse muunnosprosessi vaatii suunnittelijan tulkintaa ja päätöksentekoa vaiheessa jossa resurssimäärittely tehdään.

Jälkimmäisen lähestymistavan ongelma on se, että ainoastaan asiakassovellus omaa REST:in mukaisia piirteitä. Mitä luultavimmin myöskään varsinainen SOAP-palvelu ei ole suunniteltu linkitettävyyttä silmällä pitäen, joten edes asiakassovellus ei ole tällöin välttämättä aidosti REST:in mukainen, koska linkitettävyyden vaatimus ei täyty. Lisäksi yksi REST:in mukaisen sovelluksen suunnittelutavoitteista on yksinkertaisuus. Tässä lähestymistavassa kuitenkin ainoastaan lisätään monimutkaisuutta lisäämällä edelleen yksi ”ylimääräinen” kerros lisää palvelurajapintaan. Tämä myös hidastaa pyyntöjen prosessointia [Guinard et al., 2011].

Mallipohjaisessa suunnittelussa, kerättyjen vaatimusten pohjalta muodostetaan esimerkiksi UML -kaaviota käyttäen domain -malli [Chung et al., 2006]. Kuitenkin muunnettaessa jo olemassa olevaa palvelua vaatimuksia ei välttämättä ole saatavilla tai ne voivat

olla puutteelliset. Lähdenkin tässä yhteydessä siitä oletuksesta, että domain-mallin vaatimat tiedot ovat palautettavissa itse toteutuksesta.

Puhtaasti sovellussuunnittelun näkökulmasta olisi kuitenkin monessa tapauksessa järkevää vähintään iteroida olemassa olevat vaatimukset muunnostyön yhteydessä. Tätä on pohdittu esimerkiksi SOAP-pohjaisten RFID-palvelurajapintojen sovittamisessa REST:in mukaiseen muotoon [Guinard et al., 2011], mutta lopulta on kuitenkin päädytty, täysimuotoisen muunnoksen sijaan, hieman edellä kuvatun kaltaisen esikäsittelijän toteuttamiseen. Toteutettu malli kuitenkin aikaisemmasta mallista poiketen täyttää REST:in mukaisen www-sovelluspalvelun vaatimukset.

Käytännössä vaatimusten uudelleen iterointi ei kuitenkaan aina ole mahdollista, esimerkiksi aikataulullisista ja yritystaloudellisista seikoista johtuen. Menetelmäni tavoitteena onkin erityisesti yksinkertaistaa siirtymistä REST:in mukaiseen rajapintaan muuttamalla ainoastaan rajapintatoteutus ja olla mahdollisuuksien mukaan välittämättä taustalla olevasta toteutuksesta.

Rajapintojen muuttamisen yksinkertaisuus riippuu kuitenkin paljolti myös sovelluksen muusta arkkitehtuurista. Joka tapauksessa jos varsinaisen rajapinnan toteutus on eriytetty omaksi komponentikseen, pitäisi muutoksen olla useimmissa tapauksissa mahdollinen ilman koko sovelluksen muuttamista. Lisäksi muunnoksen jälkeen myös rajapinnan ulkopuolista toteutusta voidaan jatkokehittää REST:in mukaiseen suuntaan, mikäli se tuo tarpeellisiksi koettuja etuja. Tässä valossa menetelmäni voidaan nähdä siis myös yhtenä vaiheena laajemmassa siirtymäprosessissa SOAP-palvelusta REST:in mukaiseen palveluun.

Itse muunnos voidaan jakaa edelleen viiteen vaiheeseen: operaatioiden tunnistaminen, operaatioiden pilkkominen, domain-mallin luonti, resurssimallin luonti ja resurssimallista toteutukseen. Käsittelen näitä vaiheita tarkemmin jäljempänä.

3.1. SOAP ja WSDL

Www-sovelluspalveluiden toteuttamiseen on olemassa monia erilaisia teknologioita. Näitä ovat esimerkiksi XML-RPC, SOAP ja REST. Tämän tutkielman tavoitteena on kuvailla muunnos SOAP:ia käyttävästä palvelusta REST:in mukaiseen palveluun. Tästä johtuen en kiinnitä huomiota muihin teknologioihin tämän enempää. Tässä kohdassa kuvailen kuitenkin SOAP:in siltä osin kuin se on jäljempänä esitettävän muunnosprosessin ymmärtämisen kannalta tarpeellista.

SOAP on kirjekuorianalogian mukainen protokolla [Kumar et al., 2010]. Tämä tarkoittaa sitä, että sanomassa on eriytetty kuljetukseen liittyvät tiedot ja varsinainen sisältö. Myös HTTP-protokolla on tällainen.

Jokainen SOAP:ia käyttävä www-sovelluspalvelu voidaan kuvata XML-pohjaisella kuvauskielellä nimeltä WSDL. SOAP-palvelun WSDL-kuvauksen saa tavallisesti ladattua asettamalla palvelun URI:n kyselyosaan lisämäärittelyn 'WSDL'.

WSDL-kuvauksen pohjalta voidaan erilaisten työkalujen avulla generoida automaattisesti luokkia, jotka toteuttavat www-sovelluspalvelun rajapinnan. Tällainen on esimerkiksi Javassa JAX-WS -määrittelyn [Kumar et al., 2010] mukainen toteutus.

Nämä automaattisesti generoidut luokat ovat se osa sovellusta, joka mallini mukaisen muunnoksen jälkeen on toteutettava uudelleen. Esimerkiksi Javalle on kuitenkin jo olemassa REST-rajapintoja varten oma määrittely [Sun Microsystems, 2009]. Täten esimerkiksi Javalla toteutetussa sovelluksessa palvelurajapintojen muuttaminen pitäisi olla melko suoraviivaista. Kuitenkin muunnostyön vaativuus on voimakkaasti sidonnainen toteutuksen kanssa, joten yleisiä ohjeita tämän suhteen on mahdoton antaa. Muunnosta suunniteltaessa tulisikin kiinnittää huomiota erityisesti sovelluksen arkkitehtuurin yhteensopivuutta resurssipohjaisen arkkitehtuurin kanssa [Guinard et al., 2011].

3.1.1. WSDL-kuvauksen rakenne

Seuraavaksi käsittelen WSDL:n rakennetta ja muunnokseen liittyviä vaiheita säätietojen hakuun WebServiceX-palvelussa [2013] vapaasti tarjolla olevan www-sovelluspalvelun WSDL-tiedoston avulla.

```
<wsdl:definitions xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns:tm="http://microsoft.com/wsdl/mime/textMatching/"
xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:mime="http://schemas.xmlsoap.org/wsdl/mime/"
xmlns:tns="http://www.webserviceX.NET"
xmlns:s="http://www.w3.org/2001/XMLSchema"
xmlns:soap12="http://schemas.xmlsoap.org/wsdl/soap12/"
xmlns:http="http://schemas.xmlsoap.org/wsdl/http/"
targetNamespace="http://www.webserviceX.NET"
xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">

…

</wsdl:definitions>
```

Taulukko 11: Esimerkki 'wsdl:definitions'-lohkon määrittelystä.

Taulukossa 11 on esimerkki 'wsdl:definitions'-lohkon aloitustagista. Esimerkissä on määritelty käytettävä skeema SOAP:ia varten ja siihen liittyviä laajennoksia. Varsinaisen muunnoksen kannalta näiden merkitys on kuitenkin vähäinen.

```
<wsdl:types>

...

  <s:element name="GetWeather">
        <s:complexType>
          <s:sequence>
            <s:element minOccurs="0" maxOccurs="1" name="CityName" type="s:string"/>
            <s:element minOccurs="0" maxOccurs="1" name="CountryName"
type="s:string"/>
          </s:sequence>
        </s:complexType>
  </s:element>

  <s:element name="GetWeatherResponse">
        <s:complexType>
          <s:sequence>
            <s:element minOccurs="0" maxOccurs="1" name="GetWeatherResult"
type="s:string"/>
          </s:sequence>
        </s:complexType>
  </s:element>

  <s:element name="GetCitiesByCountry">
        <s:complexType>
          <s:sequence>
            <s:element minOccurs="0" maxOccurs="1" name="CountryName"
type="s:string"/>
          </s:sequence>
        </s:complexType>
  </s:element>

  <s:element name="GetCitiesByCountryResponse">
```

```
      <s:complexType>
        <s:sequence>
          <s:element minOccurs="0" maxOccurs="1" name="GetCitiesByCountryResult"
type="s:string"/>
        </s:sequence>
      </s:complexType>
  </s:element>

  <s:element name="string" nillable="true" type="s:string"/>

...

</wsdl:types>
```

Taulukko 12: Esimerkki 'wsdl:types'-lohkosta, jossa yksi 's:element' lohko.

'wsdl:types'-lohko sisältää palvelun tyyppimäärittelyt [W3C, 2001]. Taulukossa 12 'wsdl:types'-lohkon sisällä voidaan nähdä muun muassa 's:element'-lohkon aloitusmäärittely, jonka nimi on määritelty 'name'-attribuutilla muotoon 'GetCitiesByCountry'. 's:element'-lohkon sisällä on edelleen 's:complexType'-lohko. Tämä tyyppimäärittely kuvaa tietotyypin, joka muodostuu listasta, jonka alkiot ovat perustyyppiä 's:string' olevia tietueita nimeltä 'CountryName'. Elementtien nimien perusteella voidaan päätellä, että tietotyyppiä käytetään valtion nimen välittämiseen www-sovelluspalvelulle.

Kompleksityyppien lisäksi esimerkissä on määritelty lisäksi merkkijonomuotoinen tyyppi, jonka 'name'-attribuutin arvo on 'string'.

```
<wsdl:message name="GetCitiesByCountrySoapIn">
   <wsdl:part name="parameters" element="tns:GetCitiesByCountry"/>
</wsdl:message>

  <wsdl:message name="GetCitiesByCountrySoapOut">
   <wsdl:part name="parameters" element="tns:GetCitiesByCountryResponse"/>
</wsdl:message>

<wsdl:message name="GetCitiesByCountryHttpGetIn">
   <wsdl:part name="CountryName" type="s:string"/>
</wsdl:message>

<wsdl:message name="GetCitiesByCountryHttpGetOut">
   <wsdl:part name="Body" element="tns:string"/>
</wsdl:message>

<wsdl:message name="GetCitiesByCountryHttpPostIn">
   <wsdl:part name="CountryName" type="s:string"/>
</wsdl:message>

<wsdl:message name="GetCitiesByCountryHttpPostOut">
   <wsdl:part name="Body" element="tns:string"/>
</wsdl:message>
```

Taulukko 13: Esimerkki 'wsdl:message'-lohkoista.

Taulukosta 13 on esimerkki WSDL-kuvauksen 'wsdl:messages'-lohkoista. 'wsdl:message' määrittää sanoman sisällön [W3C, 2001]. Esimerkiksi siis ylimpänä esiintyvä sanoma 'GetCitiesByCountrySoapIn' koostuu aiemmin määritellystä 'tns:GetCitiesByCountry'-tietotyypistä. Lisäksi taulukosta voidaan nähdä myös HTTP-metodien GET ja POST mukaisten yhteystapojen sanomamäärittelyt.

```
<wsdl:portType name="GlobalWeatherSoap">

  <wsdl:operation name="GetWeather">
     <wsdl:documentation xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">Get weather
report for all major cities around the world.</wsdl:documentation>
     <wsdl:input message="tns:GetWeatherSoapIn"/>
     <wsdl:output message="tns:GetWeatherSoapOut"/>
  </wsdl:operation>

  <wsdl:operation name="GetCitiesByCountry">
     <wsdl:documentation xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">Get all major
cities by country name(full / part).</wsdl:documentation>
     <wsdl:input message="tns:GetCitiesByCountrySoapIn"/>
     <wsdl:output message="tns:GetCitiesByCountrySoapOut"/>
  </wsdl:operation>

</wsdl:portType>
```

Taulukko 14: Esimerkki 'wsdl:portType'-lohkosta.

'wsdl:portType'-lohko määrittää joukon abstrakteja operaatioita [W3C, 2001]. Kuten taulukosta 14 voidaan nähdä esimerkiksi 'GetCitiesByCountry' operaatiolle on määritelty sanomat 'tns:GetCitiesByCountrySoapIn' ja 'tns:GetCitiesByCountrySoapOut'. Tässä siis määritellään kyseiseen operaatioon liittyvät sanomat.

```
<wsdl:binding name="GlobalWeatherSoap" type="tns:GlobalWeatherSoap">
  <soap:binding transport="http://schemas.xmlsoap.org/soap/http"/>

  <wsdl:operation name="GetWeather">

     <soap:operation soapAction="http://www.webserviceX.NET/GetWeather"
style="document"/>

    <wsdl:input>
        <soap:body use="literal"/>
    </wsdl:input>

    <wsdl:output>
        <soap:body use="literal"/>
    </wsdl:output>

  </wsdl:operation>

  <wsdl:operation name="GetCitiesByCountry">

     <soap:operation soapAction="http://www.webserviceX.NET/GetCitiesByCountry"
style="document"/>

     <wsdl:input>
        <soap:body use="literal"/>
    </wsdl:input>

    <wsdl:output>
        <soap:body use="literal"/>
    </wsdl:output>

  </wsdl:operation>

</wsdl:binding>
```

Taulukko 15: Esimerkki 'wsdl:binding'-lohkosta.

Taulukossa 15 on esimerkkinä toimivan www-sovelluspalvelun yksi 'wsdl:binding'-lohko. 'wsdl:binding'-lohko määrittää varsinaisen toiminnallisuuden abstrakteille porteille [W3C, 2001]. Yhtä abstraktia porttityyppimääritystä kohden voi olla rajaton määrä 'wsdl:binding'-määrityksiä.

'wsdl:binding'-lohkosta löytyy myös viittaukset edellä määriteltyihin abstrakteihin operaatioihin. Tässä yhteydessä määritellään myös operaation kohteen tunniste attribuutin 'soapAction' arvona. Esimerkiksi siis operaation 'GetCitiesByCountry' kohde on 'http://www.webserviceX.NET/GetCitiesByCountry'. Tässä on hyvä huomata, että tunniste ei ole sama asia kuin www:ssä tiettyyn paikkaan osoittava URI. Kyseessä on SOAP-palvelun sisäinen tunniste.

```
<wsdl:service name="GlobalWeather">

   <wsdl:port name="GlobalWeatherSoap" binding="tns:GlobalWeatherSoap">
     <soap:address location="http://www.webservicex.net/globalweather.asmx"/>
   </wsdl:port>

   <wsdl:port name="GlobalWeatherSoap12" binding="tns:GlobalWeatherSoap12">
     <soap12:address location="http://www.webservicex.net/globalweather.asmx"/>
   </wsdl:port>

   <wsdl:port name="GlobalWeatherHttpGet" binding="tns:GlobalWeatherHttpGet">
     <http:address location="http://www.webservicex.net/globalweather.asmx"/>
   </wsdl:port>

   <wsdl:port name="GlobalWeatherHttpPost" binding="tns:GlobalWeatherHttpPost">
     <http:address location="http://www.webservicex.net/globalweather.asmx"/>
   </wsdl:port>

</wsdl:service>
```

Taulukko 16: Esimerkki 'wsdl:service'-lohkosta.

Taulukossa 16 on 'wsdl:service'-lohko, jonka sisällä määritellään varsinainen www-sovelluspalvelun URI 'http:address'-lohkon attribuuttina. 'wsdl:service'-lohko siis yhdistää toisiinsa liittyvät portit [W3C, 2001]. Esimerkissä kaikilla porttimäärityksillä on sama sijainti 'http://www.webservicex.net/globalweather.asmx'.

Voidaan siis todeta, että vaikka edellä esitetty SOAP-palvelu käyttääkin HTTP-protokollaa, niin se luo täysin omien määritystensä mukaisen rakenteen HTTP:n sisään. Lisäksi koko 'GlobalWeather'-palvelulla on ainoastaan yksi URI, joka toimii aidosti webin nimiavaruudessa.

3.1.2. Operaatioiden tunnistaminen

Muunnettaessa SOAP-palvelua REST:in mukaiseen muotoon ensimmäinen vaihe on operaatioiden tunnistaminen.

Käytännössä tämä tarkoittaa 'wsdl:portType'-lohkojen sisältämien 'wsdl:operation'-lohkojen 'name' -attribuuttien joukkoa. Jokainen täsmälleen sama 'name'-attribuutti voi siis esiintyä ainoastaan kerran riippumatta esiintymiskertojen lukumäärästä 'wsdl:portType'-lohkossa.

Taulukon 14 tapauksessa saadaan siis operaatiojoukko { 'GetWeather', 'GetCitiesByCountry' }.

3.1.3. Operaatioiden pilkkominen

Muunnoksen toinen vaihe on operaatioiden pilkkominen.

W3C:n [2007] SOAP-määrittely ei aseta rajoitteita operaatioiden nimeämiselle. Määrittelyn esimerkit noudattavat kuitenkin saman tyyppistä nimeämiskäytäntöä kuin edellä esitetystä esimerkkipalvelun WSDL-kuvauksesta voidaan nähdä.

On tietysti siis täysin mahdollista, että operaatiot on nimetty vaikkapa jonkin oman numerokoodijärjestelmän mukaan. Tällöin pilkkominen ei ole tietenkään aivan yhtä suoraviivainen toimenpide. Koska nimeämiselle ei ole yhtenäistä käytäntöä, tarvitaan pilkkomisvaiheessa kuitenkin joka tapauksessa ad hoc -päättelyä, joten erilaisten nimeämiskäytäntöjen tullessa vastaan on ne ensin määriteltävä.

Lähden tässä kuitenkin siitä olettamuksesta, että operaatiot on nimetty kuvailevasti jollakin systemaattisella menetelmällä.

Edellä saadun operaatiojoukon tapauksessa nimeämiskäytäntö voitaisiin määritellä vaikkapa ensin siten, että jokainen suuraakkosella alkava merkkijono on sana. Näin esimerkiksi operaatio 'GetCitiesByCountry' saadaan pilkottua sekvenssiksi merkkijonoja: ('Get', 'Cities', 'By', 'Country').

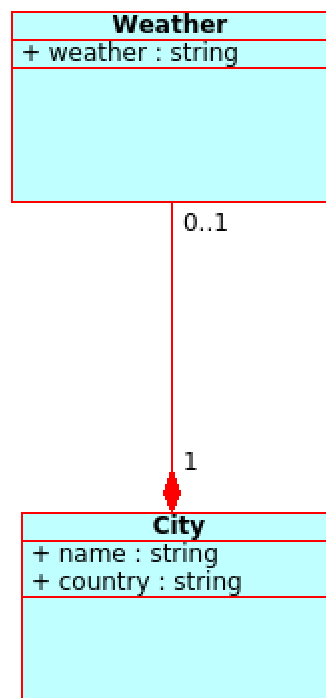
Edelleen voidaan esimerkkitapauksessa määritellä, että sekvenssin ensimmäinen alkio määrittää operaation luonteen, apusanat merkitsevät relaatiota, sekvenssin toinen sana merkitsee entiteettiä ja jäljelle jääneet sanat ovat sekvenssin määrittämän entiteetin attribuutteja. Esimerkissä siis operaation luonne on 'Get', entiteetti on 'Cities', entiteetin 'Cities'-attribuutti on 'Country' ja relaatiomerkki on 'By'.

Tavoitteena nimeämiskäytäntöjen määrittelyssä tulisi siis olla sen systematiikan tunnistaminen, jolla voidaan erotella operaation luonne, entiteetit ja relaatiomerkit toisistaan. Tällaista systematiikkaa ei tietysti välttämättä ole olemassa, joten tällöin jokainen operaatio on vain käsiteltävä erikseen.

3.1.4. Domain-mallin luonti

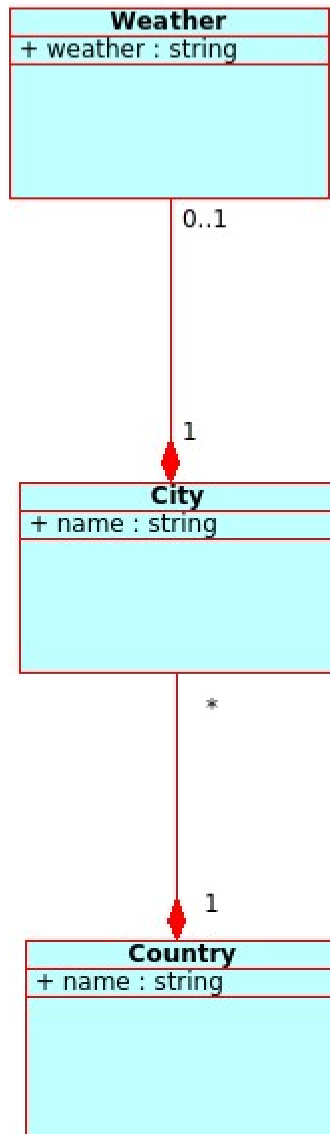
Sen jälkeen kun operaation luonne, entiteetit, atribuutit ja relaatiomerkit on saatu erotettua toisistaan, voidaan aloittaa domain -mallin luonti.

Aluksi operaation luonteen perusteella valitaan käytettävä HTTP-metodi. Esimerkissämme se on suoraviivaisesti GET. Toisaalta jos esimerkkioperaatiomme nimi olisi vaikkapa UpdateCitiesByCountry, saisimme operaation luonteeksi sanan 'Update'. Tässä tapauksessa olisi luonnollista valita käytettäväksi HTTP-metodiksi PUT. Käytännössä oikean metodin valinnassa onnistuminen lienee pitkälle kiinni nimeämiskäytännön selkeydestä ja toisaalta kyseessä olevan www-sovelluspalvelun tuntemuksesta. Päätettyjä HTTP-metodeja ei tarvita tässä muunnosvaiheessa, joten ne kirjataan ainoastaan ylös, niiden jäädessä odottamaan resurssimallin luontivaihetta.



Kuva 2: GlobalWeather -palvelun domain-malli.

Kuvassa 2 nähdään esimerkkipalvelumme domain-malli UML-kaaviona. Domain-mallin käsite noudattaa Selosen [2011] käyttämää tapaa. Entiteetti 'City' on muodostettu suoraan pilkkomisvaihe-esimerkissämme määritellyistä entiteeteistä.



Kuva 3: GlobalWeather -palvelun laajennettu domain-malli.

Tavoitteenani on pitää muunnettu www-sovelluspalvelu mahdollisimman lähellä alkuperäistä. Esimerkkipalvelumme ei kuitenkaan tarjoa mahdollisuutta esittää täydellistä esimerkkiä relaatiomerkin käytöstä, joten kuvaan 3 olen esityksen kattavuuden vuoksi hahmotellut mallin laajennetusta GlobalWeather-palvelusta.

Tähän malliin on luotu myös entiteetti 'Country'. Edellä kuvatun mekaanisen muunnoksen tuottamaan domain-malliin 'Country'-entiteetin mukaan saaminen olisi vaatinut olemassa olevien operaatioiden lisäksi operaation 'GetCountries'. Tällöin 'Countries' olisi ollut luonteeltaan atribuutin sijaan entiteetti. Tarvittaessa on siis myös mahdollista olla noudattamatta muunnosmekaniikkaa orjallisesti ja tehdä tämän kaltaisia lisäyksiä. Lisäykset aiheuttavat mahdollisesti kuitenkin lisätyötä itse toteutusvaiheessa, joten käsittelen tätä laajennettua mallia ainoastaan lyhyesti relaatiomerkin osalta.

Entiteettien välinen suhde on kuvan 3 domain-mallissa muodostettu määrittämällä ensin pilkkomisvaiheen tuotoksen perusteella relaatio R (City, Country). Relaatio on muodostettu mekaanisesti siten, että relaatiomerkkiä edeltävä alkio asettuu kaksipaikkaisen relaation ensimmäiseen paikkaan ja relaatiomerkin jälkeinen alkio jälkimmäiseen paikkaan. Relaatio on tämän jälkeen muunnettu entiteettien väliseksi suhteeksi.

Suhteen rajoittavuus, tässä tapauksessa kompositio, tulee määrätä tapauskohtaisesti. Lähtökohtaisesti kuitenkin relaation jälkimmäisessä paikassa oleva entiteetti riippuu ensimmäisessä paikassa olevasta entiteetistä.

Laajennetussa esimerkkitapauksessa on määrätty, että jokaisen kaupungin tulee sijaita jossakin valtiossa. Toisaalta kardinaliteettimäärityksellä on jätetty mahdollisuus, että yksittäisessä valtiossa ei sijaitse ainoatakaan kaupunkia. Käytännössä tämä tilanne voisi tulla kysymykseen esimerkiksi, jos jonkin valtion kaupunkeja ei ole vielä ehditty syöttää järjestelmään.

Jos pilkkomisvaiheen jälkeen muodostuneessa sekvenssissä ei ole yhtään relaatiomerkkiä (tai ei muuten voida tai haluta käyttää esitettyä tapaa), voidaan tarvittavat suhteet päätellä WSDL-kuvauksen operaation kompleksityyppimäärittelyn 's:element'-lohkojen perusteella. Näin on menetelty myös muodostettaessa kuvassa 2 esitettyä varsinaisen esimerkkimme domain-mallia.

Esimerkkikuvauksessa taulukossa 12, esimerkiksi 'GetWeather'-operaatio voidaan yhdistää suoraan 'wsdl:message'-lohkon avulla kompleksityyppiin 'GetWeather'. Kompleksityyppi sisältää kaksi 's:element'-lohkoa. Näiden perusteella voidaan päätellä relaatio entiteettiin 'City' ja sen atribuuttiin 'Country'. Relaatio tulee kuitenkin muodostaa päinvastaisessa järjestyksessä verrattuna relaatiomerkin yhteydessä käytettyyn tapaan suhteen rajoittavuuden määrittelyn helpottamiseksi.

Vaihtoehtoisesti voidaan kuitenkin hyödyntää myös relaatiomerkkiä, vaikka relaatiomerkin toisella puolella olisikin entiteetin sijaan attribuutti. Varsinaisessa esimerkkitapauksessa saataisiin siis muodostettua relaatio R (City, Country), sillä erotuksella, että relaation jälkimmäisessä paikassa oleva alkio on entiteetin 'City' -atribuutti. Tässä tapauksessa atribuutin tunniste on siis 'country'.

Entiteettien muut attribuutit on päätelty tyyppimäärittelyjen sellaisten 's:element'-lohkojen perusteella, jotka ovat tyypitykseltään jotakin perustyyppiä. Esimerkkitapauksessa kaikki attribuutit ovat merkkijonomuotoisia. Myös muut perustyypit ovat mahdollisia. Tässä vaiheessa voidaan myös lisätä muita tarpeellisiksi koettuja attribuutteja. Tämä on kuitenkin täysin suunnittelijan päätettävissä.

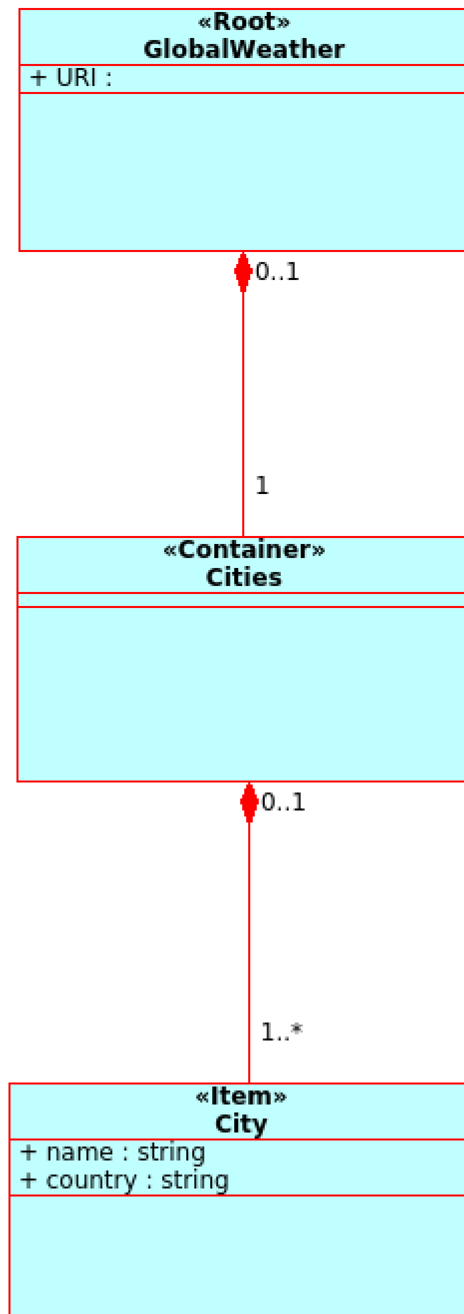
Domain-malli luodaan siis seuraavasti:

- luodaan oma entiteetti jokaista pilkkomisvaiheessa tunnistettua entiteettiä kohden
- määritellään relaatiot joko relaatiomerkkiä tai WSDL -kuvauksen tyyppimäärittelyä käyttäen
- muunnetaan relaatiot suoraan suhteiksi
- päätellään suhteiden rajoittavuus ja kardinaliteetti relaatioiden pohjalta
- asetetaan entiteettien loput attribuutit.

3.1.5. Resurssimallin luonti

Toiseksi viimeinen vaihe muunnosprosessissa on määritellä resurssimalli. Tämä voidaan luoda täysin itsenäisesti luodun domain-mallin pohjalta. Tällöin kuitenkin määrittelytietojen olemassaolo voi olla tarpeen tai jos niitä ei ole, niiden kerääminen uudelleen voi olla välttämätöntä.

Vaihtoehtoisesti voimme hyödyntää myös operaatioiden tunnistamisvaiheessa tunnistettuja operaatioita. Esimerkkitapauksessamme siis tunnistimme operaatiot 'GetWeather' ja 'GetCitiesByCountry'. Jotta pitäisimme www-sovelluspalvelumme mahdollisimman samanlaisena verrattuna alkuperäiseen SOAP-palveluun toteutamme siis ainoastaan näistä johdetut operaatiot. Voimme helposti päätellä, että vaadittu toiminnallisuus on siis tässä tapauksessa säätietojen haku ja kaupunkien haku attribuutin 'valtio' perusteella.

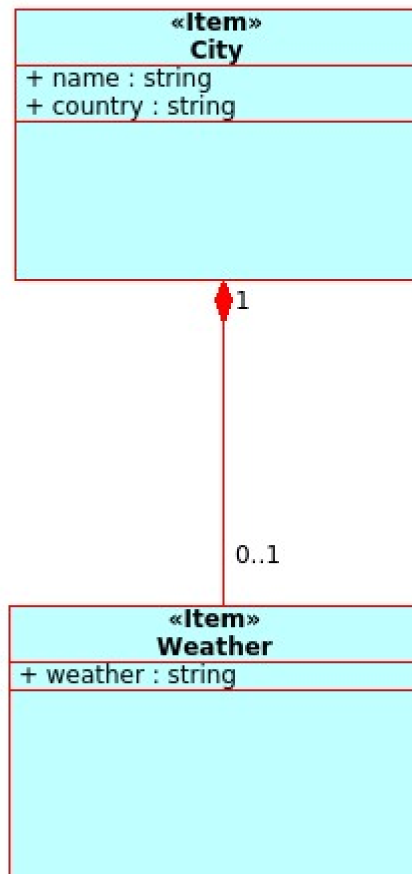


Kuva 4: Juurielementin resurssimalli.

Kuvassa 4 on esitetty domain-mallin pohjalta muodostettu juurielementin resurssimalli. Juurielementin resurssimalli on muodostettu seuraten sopivilta osin Selosen [2011] esittämää tapaa. Mallissa ylhäällä on varsinainen juurielementti, joka toteuttaa ste-

reotyypin 'Root'. Käytännössä tämä on siis GlobalWeather-palvelun aloituspiste. Aliresursseja ovat ainoastaan 'Container'-stereotyypin toteuttava 'Cities'.

'Container'-stereotyypin mukainen resurssi on joukko muita resursseja. Tässä tapauksessa siis joukko 'Item' -stereotyypin toteuttavia 'City'-resursseja. Käytännössä 'Container'-stereotyypin toteutus sisältää listauksen URI:a määritellyn tyyppisiin aliresursseihin.



Kuva 5: 'City'-elementin resurssimalli.

Kuvassa 5 näemme 'City'-elementin resurssimallin. Mallissa 'City'-elementille on määritelty 'Item'-stereotyypin toteuttava 'Weather'-aliresurssi. 'Weather'-aliresurssi sisältää yksittäisen kaupungin säätiedot merkkijonona atribuutin 'weather' arvona.

/cities?country={city.country}	GET
/cities/{ city.name }	GET
/cities/{ city.name }/weather	GET

Taulukko 17: Resurssimallin pohjalta luodut URI:t.

Taulukossa 17 on resurssimallin pohjalta luodut URI:t. Oikeanpuoleisessa sarakkeessa on tuetut HTTP -metodit. Nämä metodit on valittu pilkkomisvaiheessa esiin tulleen operaation luonteen perusteella.

Näillä URI:eilla saadaan toteutettua samankaltainen toiminnallisuus kuin alkuperäisen SOAP-palvelun operaatioilla. Taulukossa 17 ylimpänä esitetyn URI:n osalta täytyy huomata, että se on muodostettu projektio-stereotyyppiä hyödyntäen. Projektioiden avulla voidaan asettaa haettaville resursseille hakuehtoja [Selonen, 2011]. Tässä tapauksessa siis haetaan kaupunkeja 'country'-attribuutin perusteella.

```
<cities xml:base="http://globalweather.example/">
  <atom:link rel="self" href="cities?country=suomi"/>
  <city><atom:link rel="self" href="cities/tampere_suomi"/></city>
  <city><atom:link rel="self" href="cities/helsinki_suomi"/></city>
  ...
</cities>

<city xml:base="http://globalweather.example/">
  <atom:link rel="self" href="cities/tampere_suomi"/>
  <weather><atom:link rel="self" href="cities/tampere_suomi/weather"/></weather>
</city>

<weather xml:base="http://globalweather.example/">
  <atom:link rel="self" href="cities/tampere_suomi/weather"/>
</weather>
```

Taulukko 18: Esimerkki GlobalWeather-palvelun representaatioista XML-muodossa.

Taulukosta 18 voimme nähdä esimerkin kaikista resurssimallista johdetuista representaatioista. Huomionarvoinen yksityiskohta on erityisesti taulukossa 17 määritelty { city.name }-parametri. Kaupungin nimi ei välttämättä ole yksilöivä so. eri valtioissa voi olla täsmälleen saman nimisiä kaupunkeja. Olen kiertänyt tämän ongelman lisäämällä kaupungin nimen perään merkin '_' ja sen jälkeen valtion nimen.

Tapa vaikuttaa suoraviivaiselta, mutta aiheuttaa käytännössä ongelmia. Ongelmista ehkäpä selkein on se, että kaupungin nimeä ei voida laaditun mallin pohjalta esittää itsenäisenä tietona. Valittu tunniste riippuu kuitenkin täysin taustatoteutuksesta ja jos taustatoteutusta ei voida muuttaa, kuten tässä on oletetettu, niin täytyy valikoida jokin olemassaolevista tunnisteista. Tällainen voi olla esimerkiksi yksilöivä tietokannassa käytetty tunniste.

3.1.6. Resurssimallista toteutukseen

Aikaisemmissa muunnoksen vaiheissa on saatu aikaan yleinen resurssimalli ja sen pohjalta viitteitä siitä minkälaisia varsinaiset representaatiot voisivat olla. Lisäksi on havaittu, että SOAP-pohjaisen www-sovelluspalvelun muuntaminen REST:in mukaiseksi ei ole täysin suoraviivainen prosessi. Prosessin luonteeseen kuuluukin tehdä suunnitteluvalintoja olemassa olevan toteutuksen perusteella. Oma suunnitteluvalintani edellä on ollut pyrkiä toiminallisuuden osalta mahdollisimman suureen samankaltaisuuteen alkuperäisen www-sovelluspalvelun kanssa, lopputuloksen ollessa tästä huolimatta REST:in mukainen.

Lopulliseen toteutukseen voidaan käyttää jotakin valmista REST:in mukaisten www-sovelluspalvelujen toteuttamiseen tarkoitettua sovelluskehystä. Tällainen on esimerkiksi Java-kielelle saatavana oleva Restlet [Restlet, 2013]. Suuntaviivoja toteutukseen voi hakea myös esimerkiksi seuraamalla Selosen ja muiden [2010] viitoittamaa tietä.

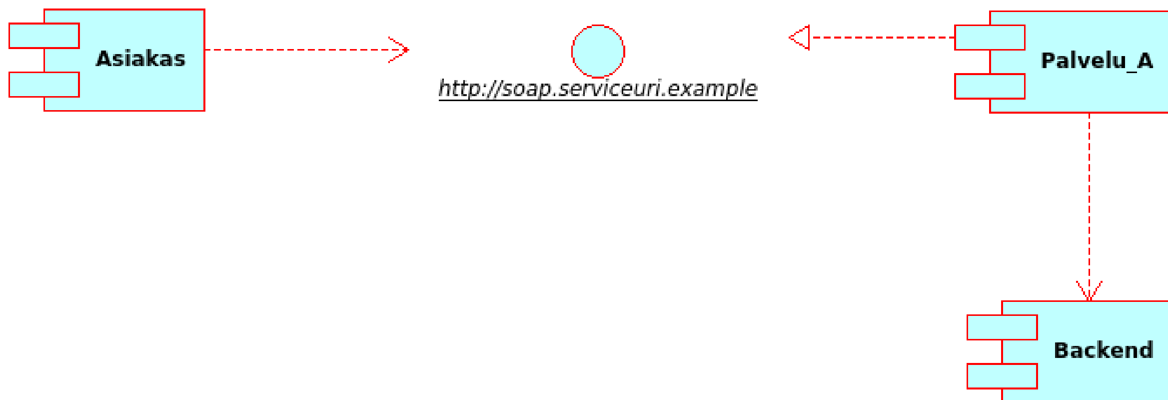
3.2. Sovellustason tarkastelua

Muunnettaessa aidossa ympäristössä jo toimivaa www-sovelluspalvelua on syytä kiinnittää laajemmin huomiota toimintaympäristöön. Erityisesti laajat SOA:n mukaiset ratkaisut voivat asettaa huomattavia rajoitteita yksittäisen palvelun muunnokselle.

Erityisesti palveluväylää hyödyntävät järjestelmät voivat olla haasteellisia, koska tällöin jouduttaisiin muuntamaan suuren sovellusjoukon lisäksi, näiden kaikkien jakama yhteinen arkkitehtuuri.

Palveluväylän tavoitteena on toimia asiakas- ja varsinaisen palvelusovelluksen välissä yhdistävänä komponenttina siten, että mahdolliset muutokset palveluihin on helpommin toteutettavissa, sekä palvelukokonaisuus paremmin hallittavissa [Flurry and Clark, 2011]. Käytännössä tavoitteet siis ovat samankaltaisia kuin REST:in mukaisella sovelluksella.

Palveluväylää hyödyntävissä ympäristöissä suoraviivainen muunnos, tai edes esikäsittelijä, tullee haastavuutensa johdosta harvoin kysymykseen. Tällöin yksi mahdollinen etenemistapa on kehittää olemassa olevan järjestelmän rinnalle kokonaan uusi kokonaisuus.



Kuva 6: Esimerkki palveluarkkitehtuurista.

Muunnosprosessin näkökulmasta mielekkäämpää onkin keskittää huomio pienimuotoisempiin järjestelmiin. Kuvassa 6 olen hahmotellut esimerkissä käyttämäni SOAP-palvelun arkkitehtuuria.

Pienin mahdollinen muutos järjestelmän REST:in mukaistamisessa olisi korvata rajapinnan 'http://soap.serviceuri.example' toteutus edellä luodulla REST:in mukaisella toteutuksella. Tämä vaatii muutoksia komponentteihin 'Palvelu_A' ja 'Asiakas'.

Jos oletetaan, että palvelua käytetään tavallisella www-selaimella voi parhaassa tapauksessa riittää ainoastaan SOAP-rajapintaan liittyvän toteutuksen poistaminen 'Asiakas'-komponentista. Muissa tapauksissa muutokset ovat kuitenkin tarpeen. Lisäksi resurssipohjaiseen malliin siirtyminen voi aiheuttaa muutoksia käyttöliittymän toteutukseen. Tätä muutospainetta pyrin muunnostyötä tehdessäni tietoisesti kuitenkin vähentämään pitäytymällä esimerkkimuunnoksessani mahdollisimman lähellä alkuperäistä toteutusta.

'Palvelu_A'-komponentin osalta rajapintatoteutuksen muutos tai rinnakkaisen rajapinnan toteuttaminen on välttämätöntä. Yksityiskohtaisella tasolla muutoksen suoraviivaisuus riippuu täysin muutostyötä vaativasta toteutuksesta. Jos toteutuksen arkkitehtuurissa rajapinta on eriytetty muista osista, muutostyö on helpompi.

Lisähaasteita muutostyöhön voivat asettaa myös 'Backend'-komponentin käyttämä tietomalli ja toisaalta useat tietovarastot. Yleisesti myös erilaiset valmiit SOA-kokonaisratkaisut voivat luoda ylimääräisiä haasteita toteutukselle.

4. Lopuksi

Olen hahmotellut REST-käsitteen luonteenpiirteitä ja sivunnut myös hieman siihen läheisesti liittyviä käsitteitä ja teknologioita. Tämän jälkeen olen esittänyt yhden tavan toteuttaa muunnos SOAP-palvelusta REST:in mukaiseen palveluun. Tehdessäni tämän olen osoittanut, että muunnos on mahdollinen ja toisaalta luonut hahmotelmia mekaanisista säännöistä helpottamaan muunnosprosessia. Lisäksi muunnosprosessin lomassa esittämäni pohdinnat toivottavasti helpottavat muunnettavuuden problematiikkaan liittyvien asioiden hahmottamista.

Tämän tutkimuksen valossa voidaan todeta, että vaikuttaisi siltä, että SOAP-palvelun muuntaminen REST:in mukaiseen palveluun ei ole täysin systematisoitavissa. Vaikuttaisi siis siltä, että muunnosprosessissa tarvitaan aina jossain määrin suunnittelijan tilannesidonnaista arviointia. Toisaalta vaikuttaisi kuitenkin siltä, että tätä arviointia ja itse muunnosprosessia tukevien yleisten systemaattisten menetelmien kehittäminen voi olla mahdollista.

Tämän tutkimuksen pohjalta ei voida kuitenkaan vetää yleisiä johtopäätöksiä palvelujen muunnettavuudesta. Voidaan ainoastaan todeta, että se on potentiaalisesti mahdollista. Myöskään esitettyjen menetelmien käyttökelpoisuutta ei voida arvioida ilman jatkotutkimuksia.

Mahdollisten jatkotutkimusten pohjalta on mahdollista tehdä tarkennuksia esitettyihin menetelmiin. Toisaalta laajemman tutkimuspohjan avulla voitaisiin tehdä myös arviointia menetelmien soveltuvuudesta käytännön tilanteisiin.

Suuremmat järjestelmäkokonaisuudet, kuten palveluväylää käyttävät järjestelmät on jätetty tässä tutkimuksessa hyvin vähälle huomiolle. Näiden osalta voikin olla mahdollista, että lähestyminen mahdolliseen muunnokseen tulee aloittaa tässä tutkielmassa esitettyä korkeammalta abstraktiotasolta.

Myös erilaiset SOA-kokonaisratkaisut ja toisaalta erilaiset REST:in mukaisten sovellusten toteutukseen tarkoitetut sovelluskehykset tulisi jatkossa huomioida tarkemmin suhteessa tässä esitettyihin menetelmiin. Voi olla mahdollista, että nämä asettavat joitakin tällä hetkellä tuntemattomia lisärajoitteita käytetyille menetelmille.

Laajempien tutkimusten valossa voitaisiin myös tehdä alustavaa arviointia siitä, miten hyvin olemassa olevien järjestelmien taustatoteutusten tietomallit tukevat resurssipohjaisuuden käyttöönottoa.

Viiteluettelo

[Berners-Lee, 1996] Tim Berners-Lee, Universal Resource Identifiers -- Axioms of Web Architecture, 19.12.1996. Available: *http://www.w3.org/DesignIssues/Axioms.*

[BOUTEL, 2006] BOUTEL.COM, WWW FAQs: What is the maximum length of a URL?, 13.10.2006. Available: *http://www.boutell.com/newfaq/misc/urllength.html.*

[Chung et al., 2006] Lawrence Chung, Weimin Ma and Kendra Cooper. Requirements elicitation through model-driven evaluation of software components. *IEEE International Conference on Commercial-off-the-Shelf (COTS)-Based Software Systems.* IEEE, 2006.

[Costa et al., 2012] Italo da Costa, Saurabh Chakradeo, Mustaque Ahamad, and Patrick Traynor, One-Time Cookies: Preventing Session Hijacking Attacks with Stateless Authentication Tokens. *ACM Transactions on Internet Technology.* **12** (1, 2012), article 1.

[Fielding, 2000] R. T. Fielding, Architectural Styles and the Design of Network-based Software Architectures. University of California, Irvine, Ph.D dissertation, 2000. Available: *http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm.*

[Fielding, 2008] Roy Fielding, REST APIs must be hypertext-driven, 20.8.2008. Available: *http://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven.*

[Flurry and Clark, 2011] Greg Flurry and Kim J. Clark, The Enterprise Service Bus, re-examined, 18.5.2011, IBM, 2011. Available: *http://www.ibm.com/developerworks/websphere/techjournal/1105_flurry/1105_flurry.html.*

[Fowler, 2010] Martin Fowler, Richardson Maturity Model, 18.3.2010. Available: *http://martinfowler.com/articles/richardsonMaturityModel.html.*

[Guinard et al., 2011] Dominique Guinard, Mathias Mueller, and Vlad Trifa, RESTifying Real-World Systems: A Practical Case Study in RFID. In: Erik Wilde and Cesare Pautasso (eds.), *REST: From Research to Practice.* Springer, 2011, pp. 359-379.

[Järvinen ja Järvinen, 2004] Pertti Järvinen ja Annikki Järvinen, *Tutkimustyön metodeista*. Opinpajan kirja, 2004.

[Koskimies, 2000] K. Koskimies, *Oliokirja*. Satku - kauppakaari, 2000.

[Kristol, 2001] David M. Kristol, HTTP Cookies: Standards, Privacy, and Politics. *ACM Transactions on Internet Technology.* **1** (2, 2001), pp. 151-198.

[Kumar et al., 2010] B. V. Kumar, Prakash Narayan and Tony Ng, *Implementing SOA Using Java EE*. Addison-Wesley, 2010.

[Laitkorpi et al., 2009] M. Laitkorpi, P. Selonen, and T. Systä. Towards a model-driven process for designing restful web services. *IEEE International Conference on Web Services, ICWS 2009.* IEEE, 2009, pp. 173–180.

[Li and Wu, 2011] Li Li and Chou Wu, Design and describe REST API without violating REST: a Petri net based approach. *Web Services (ICWS), IEEE International Conference*, 2011, pp. 508-515.

[NWG, 1994] Network working group, Universal Resource Identifiers in WWW, June 1994. Available: *https://www.ietf.org/rfc/rfc1630.txt.*

[NWG, 1999] Network working group, Hypertext Transfer Protocol – HTTP/1.1, June 1999. Available: *http://www.w3.org/Protocols/rfc2616/rfc2616.html.*

[NWG, 2005] Network working group, The Atom Syndication Format, December 2005. Available: *https://tools.ietf.org/html/rfc4287.*

[NWG, 2007] Network working group, The Atom Publishing Protocol, October 2007. Available: *https://tools.ietf.org/html/rfc5023.*

[O'Neill, 2008] Mark O'Neill, How to convert from REST to SOAP, 14.11.2008. Available: *http://www.soatothecloud.com/2008/11/how-to-convert-from-rest-to-soap.html.*

[Restlet, 2013] Restlet, Restlet 2.1 - Tutorial, 2013, Available: *http://restlet.org/learn/tutorial/2.1/.*

[Richardson, 2008] Leonard Richardsson, Justice Will Take Us Millions of Intricate Moves, Act Three: The Maturity Heuristic, *Qcon*, 20.11.2008. Available: *http://www.crummy.com/writing/speaking/2008-QCon/act3.html.*

[Richardson and Ruby, 2007] Leonard Richardson and Sam Ruby, *ReSTful Web Services*. O'Reilly Media, 2007.

[Richtr and Farana, 2011] Lukáš Richtr and Radim Farana, Remote Control the Robot using Web Service. IEEE *Carpathian Control Conference (ICCC),* 2011, pp. 326-330.

[Royal pingdom, 2010] Royal pingdom, REST in peace, SOAP. 15.10.2010. Available: *http://royal.pingdom.com/2010/10/15/rest-in-peace-soap/.*

[Selonen, 2011] Petri Selonen, From Requirements to a RESTful Web Service: Engineering Content Oriented Web Services with REST. In: Erik Wilde and Cesare Pautasso (eds.), *REST: From Research to Practice.* Springer, 2011, pp. 259-278.

[Selonen et al., 2010] Petri Selonen, Petros Belimpasakis and Yu You. Developing a ReSTful Mixed Reality Web Service Platform. *Proceeding WS-REST '10 Proceedings of the First International Workshop on RESTful Design*. ACM, 2010, pp. 54–61.

[Steiner and Algermissen, 2011] Thomas Steiner, Jan Algermissen, Fulfilling the hypermedia constraint via HTTP OPTIONS, the HTTP vocabulary in RDF, and link headers. *ACM Proceedings of the Second International Workshop on RESTful Design.* 2011, pp. 11-14.

[Sun Microsystems, 2009] Sun Microsystems, JAX-RS: Java API for RESTful Web Services, 17.9.2009. Available: *http://jcp.org/aboutJava/communityprocess/mrel/jsr311/.*

[W3C, 2001] W3C working group, Web Services Description Language (WSDL) 1.1, 15.3.2001. Available: *http://www.w3.org/TR/wsdl.*

[W3C, 2004a] W3C working group, Web services architecture, 11.2.2004. Available: *http://www.w3.org/TR/ws-arch.*

[W3C, 2004b] W3C working group, Architecture of the World Wide Web, Volume One, 15.12.2004. Available: *http://www.w3.org/TR/webarch/.*

[W3C, 2007] W3C working group, SOAP Version 1.2 Part 2: Adjuncts (Second Edition), 27.4.2007. Available: *http://www.w3.org/TR/2007/REC-soap12-part2-20070427/.*

[W3C, 2009] W3C Member Submission, Web Application Description Language, 31.8.2009. Available: *http://www.w3.org/Submission/wadl/.*

[W3C, 2013] W3C working group, HTML 5.1 Nightly, 26.2.2013. Available: *http://www.w3.org/html/wg/drafts/html/master/.*

[WebServiceEngine, 2013] WebServiceEngine, readme.md, 2013, Available: *https://github.com/Swiitch/WebServiceEngine.*

[WebServiceX, 2013] WebServiceX, wsdl, 2013, Available: *http://www.webservicex.net/WS/WSDetails.aspx?CATID=12&WSID=56.*

GlobalWeather WSDL-kuvaus


```
<wsdl:definitions xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns:tm="http://microsoft.com/wsdl/mime/textMatching/"
xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:mime="http://schemas.xmlsoap.org/wsdl/mime/"
xmlns:tns="http://www.webserviceX.NET"
xmlns:s="http://www.w3.org/2001/XMLSchema"
xmlns:soap12="http://schemas.xmlsoap.org/wsdl/soap12/"
xmlns:http="http://schemas.xmlsoap.org/wsdl/http/"
targetNamespace="http://www.webserviceX.NET"
xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
  <wsdl:types>
    <s:schema elementFormDefault="qualified"
targetNamespace="http://www.webserviceX.NET">
      <s:element name="GetWeather">
        <s:complexType>
          <s:sequence>
            <s:element minOccurs="0" maxOccurs="1" name="CityName" type="s:string"/>
            <s:element minOccurs="0" maxOccurs="1" name="CountryName"
type="s:string"/>
          </s:sequence>
        </s:complexType>
      </s:element>
      <s:element name="GetWeatherResponse">
        <s:complexType>
          <s:sequence>
            <s:element minOccurs="0" maxOccurs="1" name="GetWeatherResult"
type="s:string"/>
          </s:sequence>
        </s:complexType>
      </s:element>
      <s:element name="GetCitiesByCountry">
```

```
      <s:complexType>
        <s:sequence>
          <s:element minOccurs="0" maxOccurs="1" name="CountryName"
type="s:string"/>
        </s:sequence>
      </s:complexType>
     </s:element>
     <s:element name="GetCitiesByCountryResponse">
      <s:complexType>
        <s:sequence>
          <s:element minOccurs="0" maxOccurs="1" name="GetCitiesByCountryResult"
type="s:string"/>
        </s:sequence>
      </s:complexType>
     </s:element>
     <s:element name="string" nillable="true" type="s:string"/>
   </s:schema>
 </wsdl:types>
 <wsdl:message name="GetWeatherSoapIn">
   <wsdl:part name="parameters" element="tns:GetWeather"/>
 </wsdl:message>
 <wsdl:message name="GetWeatherSoapOut">
   <wsdl:part name="parameters" element="tns:GetWeatherResponse"/>
 </wsdl:message>
 <wsdl:message name="GetCitiesByCountrySoapIn">
   <wsdl:part name="parameters" element="tns:GetCitiesByCountry"/>
 </wsdl:message>
 <wsdl:message name="GetCitiesByCountrySoapOut">
   <wsdl:part name="parameters" element="tns:GetCitiesByCountryResponse"/>
 </wsdl:message>
 <wsdl:message name="GetWeatherHttpGetIn">
   <wsdl:part name="CityName" type="s:string"/>
   <wsdl:part name="CountryName" type="s:string"/>
 </wsdl:message>
 <wsdl:message name="GetWeatherHttpGetOut">
   <wsdl:part name="Body" element="tns:string"/>
```

```
  </wsdl:message>
  <wsdl:message name="GetCitiesByCountryHttpGetIn">
    <wsdl:part name="CountryName" type="s:string"/>
  </wsdl:message>
  <wsdl:message name="GetCitiesByCountryHttpGetOut">
    <wsdl:part name="Body" element="tns:string"/>
  </wsdl:message>
  <wsdl:message name="GetWeatherHttpPostIn">
    <wsdl:part name="CityName" type="s:string"/>
    <wsdl:part name="CountryName" type="s:string"/>
  </wsdl:message>
  <wsdl:message name="GetWeatherHttpPostOut">
    <wsdl:part name="Body" element="tns:string"/>
  </wsdl:message>
  <wsdl:message name="GetCitiesByCountryHttpPostIn">
    <wsdl:part name="CountryName" type="s:string"/>
  </wsdl:message>
  <wsdl:message name="GetCitiesByCountryHttpPostOut">
    <wsdl:part name="Body" element="tns:string"/>
  </wsdl:message>
  <wsdl:portType name="GlobalWeatherSoap">
    <wsdl:operation name="GetWeather">
      <wsdl:documentation xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">Get weather
report for all major cities around the world.</wsdl:documentation>
      <wsdl:input message="tns:GetWeatherSoapIn"/>
      <wsdl:output message="tns:GetWeatherSoapOut"/>
    </wsdl:operation>
    <wsdl:operation name="GetCitiesByCountry">
      <wsdl:documentation xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">Get all major
cities by country name(full / part).</wsdl:documentation>
      <wsdl:input message="tns:GetCitiesByCountrySoapIn"/>
      <wsdl:output message="tns:GetCitiesByCountrySoapOut"/>
    </wsdl:operation>
  </wsdl:portType>
  <wsdl:portType name="GlobalWeatherHttpGet">
    <wsdl:operation name="GetWeather">
```

```
      <wsdl:documentation xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">Get weather
report for all major cities around the world.</wsdl:documentation>
      <wsdl:input message="tns:GetWeatherHttpGetIn"/>
      <wsdl:output message="tns:GetWeatherHttpGetOut"/>
    </wsdl:operation>
    <wsdl:operation name="GetCitiesByCountry">
      <wsdl:documentation xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">Get all major
cities by country name(full / part).</wsdl:documentation>
      <wsdl:input message="tns:GetCitiesByCountryHttpGetIn"/>
      <wsdl:output message="tns:GetCitiesByCountryHttpGetOut"/>
    </wsdl:operation>
  </wsdl:portType>
  <wsdl:portType name="GlobalWeatherHttpPost">
    <wsdl:operation name="GetWeather">
      <wsdl:documentation xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">Get weather
report for all major cities around the world.</wsdl:documentation>
      <wsdl:input message="tns:GetWeatherHttpPostIn"/>
      <wsdl:output message="tns:GetWeatherHttpPostOut"/>
    </wsdl:operation>
    <wsdl:operation name="GetCitiesByCountry">
      <wsdl:documentation xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">Get all major
cities by country name(full / part).</wsdl:documentation>
      <wsdl:input message="tns:GetCitiesByCountryHttpPostIn"/>
      <wsdl:output message="tns:GetCitiesByCountryHttpPostOut"/>
    </wsdl:operation>
  </wsdl:portType>
  <wsdl:binding name="GlobalWeatherSoap" type="tns:GlobalWeatherSoap">
    <soap:binding transport="http://schemas.xmlsoap.org/soap/http"/>
    <wsdl:operation name="GetWeather">
      <soap:operation soapAction="http://www.webserviceX.NET/GetWeather"
style="document"/>
      <wsdl:input>
        <soap:body use="literal"/>
      </wsdl:input>
      <wsdl:output>
        <soap:body use="literal"/>
```

```
    </wsdl:output>
   </wsdl:operation>
   <wsdl:operation name="GetCitiesByCountry">
    <soap:operation soapAction="http://www.webserviceX.NET/GetCitiesByCountry"
style="document"/>
    <wsdl:input>
     <soap:body use="literal"/>
    </wsdl:input>
    <wsdl:output>
     <soap:body use="literal"/>
    </wsdl:output>
   </wsdl:operation>
  </wsdl:binding>
  <wsdl:binding name="GlobalWeatherSoap12" type="tns:GlobalWeatherSoap">
   <soap12:binding transport="http://schemas.xmlsoap.org/soap/http"/>
   <wsdl:operation name="GetWeather">
    <soap12:operation soapAction="http://www.webserviceX.NET/GetWeather"
style="document"/>
    <wsdl:input>
     <soap12:body use="literal"/>
    </wsdl:input>
    <wsdl:output>
     <soap12:body use="literal"/>
    </wsdl:output>
   </wsdl:operation>
   <wsdl:operation name="GetCitiesByCountry">
    <soap12:operation soapAction="http://www.webserviceX.NET/GetCitiesByCountry"
style="document"/>
    <wsdl:input>
     <soap12:body use="literal"/>
    </wsdl:input>
    <wsdl:output>
     <soap12:body use="literal"/>
    </wsdl:output>
   </wsdl:operation>
  </wsdl:binding>
```

```
<wsdl:binding name="GlobalWeatherHttpGet" type="tns:GlobalWeatherHttpGet">
  <http:binding verb="GET"/>
  <wsdl:operation name="GetWeather">
    <http:operation location="/GetWeather"/>
    <wsdl:input>
      <http:urlEncoded/>
    </wsdl:input>
    <wsdl:output>
      <mime:mimeXml part="Body"/>
    </wsdl:output>
  </wsdl:operation>
  <wsdl:operation name="GetCitiesByCountry">
    <http:operation location="/GetCitiesByCountry"/>
    <wsdl:input>
      <http:urlEncoded/>
    </wsdl:input>
    <wsdl:output>
      <mime:mimeXml part="Body"/>
    </wsdl:output>
  </wsdl:operation>
</wsdl:binding>
<wsdl:binding name="GlobalWeatherHttpPost" type="tns:GlobalWeatherHttpPost">
  <http:binding verb="POST"/>
  <wsdl:operation name="GetWeather">
    <http:operation location="/GetWeather"/>
    <wsdl:input>
      <mime:content type="application/x-www-form-urlencoded"/>
    </wsdl:input>
    <wsdl:output>
      <mime:mimeXml part="Body"/>
    </wsdl:output>
  </wsdl:operation>
  <wsdl:operation name="GetCitiesByCountry">
    <http:operation location="/GetCitiesByCountry"/>
    <wsdl:input>
      <mime:content type="application/x-www-form-urlencoded"/>
```

```
      </wsdl:input>
      <wsdl:output>
        <mime:mimeXml part="Body"/>
      </wsdl:output>
    </wsdl:operation>
  </wsdl:binding>
  <wsdl:service name="GlobalWeather">
    <wsdl:port name="GlobalWeatherSoap" binding="tns:GlobalWeatherSoap">
      <soap:address location="http://www.webservicex.net/globalweather.asmx"/>
    </wsdl:port>
    <wsdl:port name="GlobalWeatherSoap12" binding="tns:GlobalWeatherSoap12">
      <soap12:address location="http://www.webservicex.net/globalweather.asmx"/>
    </wsdl:port>
    <wsdl:port name="GlobalWeatherHttpGet" binding="tns:GlobalWeatherHttpGet">
      <http:address location="http://www.webservicex.net/globalweather.asmx"/>
    </wsdl:port>
    <wsdl:port name="GlobalWeatherHttpPost" binding="tns:GlobalWeatherHttpPost">
      <http:address location="http://www.webservicex.net/globalweather.asmx"/>
    </wsdl:port>
  </wsdl:service>
</wsdl:definitions>
```