REST-POHJAISEN WEB SERVICEN KEHITTÄMINEN

Case oldtimerTimer

Ammattikorkeakoulun opinnäytetyö

Tietojenkäsittelyn koulutusohjelma

Visamäki, syksy 2015

Mikko Jussilainen

HAMK
HÄMEEN AMMATTIKORKEAKOULU

TIIVISTELMÄ

Visamäki
Tietojenkäsittelyn koulutusohjelma
Systeemityö

Tekijä Mikko Jussilainen **Vuosi** 2015

Työn nimi REST-pohjaisen Web Servicen kehittäminen
Case oldtimerTimer

TIIVISTELMÄ

Opinnäytetyön toimeksiantajan toimi Hämeen ammattikorkeakoulu. Sen aikana suunniteltiin ja toteutettiin REST-arkkitehtuurin mukainen Web Service. Web Servicen resursseina toimivat oldtimerTimer-järjestelmän käyttäjät tietoineen sekä aktiviteetit tapahtumineen.

Työn tavoitteena oli kehittää oldtimerTimer-järjestelmän tietokannan ja mobiilisovelluksen väliin palvelu, jonka kautta dataa voitaisiin lähettää mobiilisovelluksesta tietovarastoon ja päinvastoin. Olemassa olevat keinot tähän olivat vajavaisia eivätkä sisältäneet tietoturvaa.

Web Service toteutettiin Java-ohjelmointikielellä REST-arkkitehtuurin mukaisesti ja se pyörii Glassfish-sovelluspalvelimella. Varsinaisena tietovarastona puolestaan toimii Apachen alla pyörivä MySQL-tietokanta. Web Service tukee viestiensä osalta XML-merkintäkieltä, sillä JSON-merkintäkielen tukemista ei todettu oldtimerTimerin kannalta elintärkeäksi.

Kehitystyössä käytettiin NetBeans-sovelluskehitintä, johon ladattiin RESTful Web Services-lisäosa. Lisäosa mahdollisti REST-pohjaisen Web Servicen kehityksen sovelluskehittimellä.

Työn lähdemateriaalina käytettiin pääasiassa internetistä haettuja englanninkielisiä oppaita, luentomateriaaleja sekä videoita. Myös kirjallisuutta käytettiin jonkin verran, mutta teokset käsittelivät lähinnä REST-arkkitehtuuriin lainattuja tekniikoita eivätkä suoranaisesti itse arkkitehtuuria.

Palvelua testattiin jatkuvasti kehitystyön aikana, käyttämällä erilaisia selaimille suunniteltuja lisäosia kuten RESTClient sekä Postman. Näiden lisäosien avulla mahdollistettiin pyyntöjen lähettäminen Web Serviceen ilman varsinaista asiakasohjelmaa.

Avainsanat REST-arkkitehtuuri, Web Service, oldtimerTimer

Sivut 31 s.

HAMK
UNIVERSITY OF APPLIED SCIENCES

ABSTRACT

Visamäki
Degree Programme in Business Information Technology
Software Engineering

Author Mikko Jussilainen **Year** 2015

Subject of Bachelor's thesis Development of RESTful Web Service
Case oldtimerTimer

ABSTRACT

The commissioner for this thesis was Häme University of Applied Sciences (HAMK). It consisted of planning and developing a Web Service, based on the REST-architecture. The users (including their details) and the activities (including their events) related to oldtimerTimer act as the resources for this Web Service.

The objective of this thesis was to build a layer between the mobile application and the database, through which we could easily and safely send data from one to the other. Existing methods for doing this were obscure, and lacked any kind of security or authentication.

The Web Service was implemented in Java using the REST-architecture, and it is currently running under the Glassfish Application Server. The actual data-storage-layer is running under Apache, in the form of MySQL-database. The Web Service supports the use of XML-based messages on its requests, since JSON was deemed unnecessary addition regarding oldtimer-Timer at this point in time.

The development work was done using The NetBeans Software Development Platform, which had RESTful Web Services-plugin installed to it. This plugin allowed the development of RESTful Web Services using Java.

The source material for this thesis was mainly found from various different websites, including some academic ones, in the form of lecture-materials, guides and video tutorials. These materials were mainly written in English, as Finnish material was quite hard to come by. Some literature was also used, but these pieces mainly dealt with the various architectures from which the REST architecture borrows parts from, and not the actual REST architecture itself.

The service was being continuously tested during the development, using different kinds of browser-plugins (such as RESTClient and Postman) that allowed posting requests and receiving responses from the Web Service.

Keywords REST-architecture, Web Service, OldtimerTimer

Pages 31 p.

SISÄLLYS

1 JOHDANTO ... 1

2 OLDTIMERTIMER ... 2

3 OHJELMISTOARKKITEHTUURI ... 4

3.1 Asiakas-palvelin-arkkitehtuuri ... 4
3.2 Kolmikerrosarkkitehtuuri ... 5

4 WEB SERVICE ... 6

4.1 Määritelmä ... 6
4.2 Toimintamalli ... 7

5 WEB SERVICE -TEKNOLOGIAT ... 9

5.1 REST-arkkitehtuuri ... 9
5.2 REST-arkkitehtuurin Rajoitteet ... 10
5.2.1 Asiakas-palvelin ... 10
5.2.2 Tilaton ... 10
5.2.3 Yhtenäinen rajapinta ... 11
5.2.4 Mahdollisuus käyttää välimuistia ... 11
5.2.5 Kerroksittainen järjestelmä ... 12
5.2.6 Code on Demand ... 12
5.3 Resurssipohjaisuus REST-arkkitehtuurissa ... 12
5.4 Simple Object Access Protocol ... 13
5.4.1 Toimintamalli ... 14

6 HTTP-PROTOKOLLA ... 16

6.1 Unique Resource Identifier ... 16
6.2 Metodit ... 16
6.3 Header ja Body ... 17
6.4 HTTP-statuskoodit ... 18

7 REST-ARKKITEHTUURIIN SISÄLTYVÄT TEKNOLOGIAT ... 20

7.1 JAX-RS ... 20
7.2 Java Persistence API ... 20
7.2.1 JPA-annotaatiot ... 21
7.3 Java Architecture for XML Binding ... 22

8 KÄYTÄNNÖN SOVELLUS ... 23

8.1 REST-pohjainen Web Service ... 23
8.2 Kirjoitettuja luokkia ... 24
8.2.1 Resurssiluokat ... 24
8.2.2 Persistence Unit ... 25
8.2.3 Palveluluokat ... 25
8.2.4 Virheenkäsittelijäluokat ... 26
8.2.5 Muut luokat ... 27

9 TULOKSET & POHDINTAA ... 28

10 LÄHTEET 30

KÄSITELUETTELO

Suomi	English	Selitys
HTTP	Hypertext Transfer Protocol	Siirtoprotokolla, jota käytetään muun muassa selainten ja palvelinten väli-seen tiedonsiirtoon.
URI	Unique Resource Identifier	Merkkijono, jonka avulla viitataan tiettyyn resurssiin HTTP-kutsun yh-teydessä.
HTTP-metodi	HTTP-method	HTTP-pyynnön osa, jolla kerrotaan, mikä toiminto resurssille halutaan suorittaa. Kutsutaan myös verbiksi.
HTTP-statuskoodi	HTTP-statuscode	Palvelimen HTTP-pyynnön yhtey-dessä palauttama kolminumeroinen koodi, jonka avulla kerrotaan suoritet-tavana olevan operaation tila.
JAXB	Java Architecture for XML Binding	Arkkitehtuuri, joka sallii ohjelmoijan käsitellä XML-muotoista dataa tunte-matta XML-merkintäkieltä laajem-min.
JPA	Java Persistence API	Spesifikaatio, joka mahdollistaa taval-listen Java-luokkien tallentamisen suoraan tietokantaan ja päinvastoin.
JSON	JavaScript Object Notation	Tiedostomuoto, jota käytetään usein tiedon siirtämiseen järjestelmästä toi-seen. Dataa voidaan tallentaa hyvin organisoidussa, helppolukuisessa muodossa.
Resurssi	Resource	REST-arkkitehtuurissa käytetty termi tietolähteestä, jonka kanssa käyttäjä voi olla vuorovaikutuksessa.
XML	Extensible Markup Langu-age	Alustariippumaton, rakenteisen tiedon esittämiseen kehitetty metakieli, jota kirjoitetaan varsinaisen datan sekaan. Käytetään usein tiedon siirtämiseksi järjestelmien välillä.
XML jäsennin	XML parser	Ohjelman osa, jolla tarkistetaan XML-muotoisen dokumentin rakenne. Sen avulla dokumenttia voidaan käsitellä, pilkkoa tai muuntaa toiseen muotoon.

1 JOHDANTO

Opinnäytetyön idea muodostui pääasiassa kesällä 2014, jolloin tekijä osallistui Digital Service Development -kurssin yhteydessä oldtimerTimer-järjestelmän esikartoitustyöhön, jatkaen tämän jälkeen sovelluksen ensimmäisen prototyypin kehityspuolelle ja myöhemmässä vaiheessa sen jatkokehitykseen ICT-projektin aikana.

OldtimerTimer on Hämeen ammattikorkeakoulussa kehityksen alla oleva, pääasiassa opiskelijoiden kehittämä järjestelmä. Sen tarkoituksena on tukea ikäihmisten pidempiaikaista itsenäistä asumista ja arkielämää, tarjoamalla tietoa erilaisista alueella sijaitsevista aktiviteeteista sekä palveluista. Tämä puolestaan edesauttaa laitoshuollon tarpeen poistamista ja antaa omaisille mahdollisuuden osallistua käyttäjän elämään entistä helpommin. Järjestelmä koostuu verkkosivustosta ja mobiilisovelluksesta, joita käyttävät sekä ikäihmiset että heidän omaisensa.

Opinnäytetyössä kehitettiin oldtimerTimer-järjestelmää eteenpäin suunnittelemalla ja toteuttamalla REST-arkkitehtuurin mukainen Web Service, jonka kautta hoidetaan järjestelmän mobiilisovelluksen ja www-palvelimella sijaitsevan tietokannan välinen kommunikointi (CRUD-toiminnot käyttäjien sekä aktiviteettien osalta).

Web Servicen resursseina toimivat tällä hetkellä oldtimerTimer-järjestelmän käyttäjät sekä aktiviteetit, mutta järjestelmää on mahdollista laajentaa tulevaisuudessa kattamaan myös muitakin suunniteltuja toimintoja. Näihin kuuluvat esimerkiksi erilaiset palvelupyynnöt sekä mobiilisovelluksen näkymien muokkaus käyttäjän valitseman aktiviteetin perusteella. Mobiilisovelluksen muokkaaminen Web Serviceä käyttäväksi jätettiin kuitenkin tämän opinnäytetyön ulkopuolelle, eli työssä keskityttiin pääasiassa Web Servicen toteuttamiseen REST-arkkitehtuurin mukaisesti.

Opinnäytetyössä oli kaksi päätutkimuskysymystä, joista toinen sisälsi muutaman selventävän alikysymyksen:

Mikä on Web Service?

- Millä eri tavoilla Web Service voidaan rakentaa? Miten menetelmät eroavat toisistaan?
- Mitä menetelmää kannattaa soveltaa OTT:n kanssa?

Miten Web Service toteutetaan käytännössä?

2 OLDTIMERTIMER

OldtimerTimer on Hämeen ammattikorkeakoulussa kehityksen alla oleva, pääasiassa opiskelijoiden kehittämä järjestelmä. Sen tarkoituksena on tukea ikäihmisten pidempiaikaista itsenäistä asumista ja arkielämää, tarjoamalla helpon tavan saada tietoa erilaisista asuinalueensa aktiviteeteista sekä palveluista. Tämä puolestaan edesauttaa laitoshuollon tarpeen poistamista ja antaa omaisille mahdollisuuden osallistua ikäihmisten elämään entistä helpommin. Tällä hetkellä järjestelmä on jaettu kahteen osaan: verkkosivustoon sekä mobiilisovellukseen.

Järjestelmää on kehitetty kahdessa osassa. Vuoden 2014 kesällä järjestelmästä toteutettiin prototyyppi, jotta nähtäisiin miten se tulisi toimimaan käytännössä. Kesän aikana syntyi järjestelmän verkkosivusto sekä ensimmäinen prototyyppi mobiilisovelluksesta. Mobiilisovellusta kehitettiin eteenpäin seuraavana lukuvuonna alkaneen ICT-projektin aikana, jolloin käytiin läpi kesänaikaiset tuotokset ja tehtiin mobiilisovellukseen useita korjauksia ja uusia ominaisuuksia.

Mobiilisovellus on tarkoitettu ikäihmisten käytettäväksi heidän omilla mobiililaitteillansa (pääasiassa tablet tai puhelin). Sen avulla käyttäjät pystyvät tarkastelemaan sekä ilmoittautumaan erilaisiin järjestelmään lisättyihin tapahtumiin tai muihin aktiviteetteihin. Kun käyttäjä on merkinnyt osallistuvansa tapahtumaan, hänelle luodaan tapahtumasta muistutus. Tämä muistutus aktivoituu hieman ennen tapahtuman alkamisajankohtaa. Halutessaan käyttäjä voi myös tehdä muistutuksesta toistuvan, jolloin se aktivoituu ennen jokaista tapahtumakertaa.

Tapahtumien päättymisen jälkeen mobiilisovellus tiedustelee, kävikö käyttäjä kyseisessä tapahtumassa. Mikäli käyttäjä ei osallistunut tapahtumaan, tiedustellaan syytä siihen. Edellämainittujen lisäksi sovellus kerää aika ajoin dataa myös sen käytöstä, kuten montako kertaa aktiviteetteja on selattu, tai kuinka monta minuuttia päivässä sovellusta on yhteensä käytetty. Tämä data lähetetään aika ajoin verkkopalvelimelle tallennettavaksi, jotta sitä voidaan myöhemmin tarkastella verkkosivuston kautta.

Verkkosivustoa käyttävät ikäihmisten omaiset. Luotuaan tunnuksen järjestelmään he voivat linkittää ikäihmisen omaan tiliinsä. Linkityksen jälkeen he saavat tietoa ikäihmisen tilanteesta sekä päivittäisistä toimista, kuten tapahtumaan osallistumisista ja sovelluksen käytöstä. Kun ikäihminen osallistuu tapahtumaan, häneltä kysytään myöhemmin osallistumisesta sekä tapahtuman mielekkyydestä. Omaiset voivat verkkosivuston kautta lukea näitä tietoja. Tulevaisuudessa verkkosivustoon saatetaan lisätä ominaisuus, jonka avulla omaiset voivat ehdottaa mobiilisovelluksen käyttäjälle erilaisia tapahtumia. Ehdotetut tapahtumat näkyisivät esimerkiksi mobiilisovelluksen alkusivulla, josta käyttäjä voisi niitä halutessaan tarkastella.

Järjestelmää tullaan myös luultavimmin tulevaisuudessa laajentamaan niin, että käyttäjät voivat sen kautta myös tilata itselleen erilaisia palveluita, kuten esimerkiksi koti- ja siivousapua, kauppakäyntejä tai kuljetuspalveluita.

Verkkosivuston kautta on myös tulevaisuudessa mahdollista luoda sekä hallinnoida aktiviteetteja, joita mobiilisovelluksen kautta selataan. Vastaava ominaisuus lisätään myös mahdollisesti mobiilisovellukseen.

3 OHJELMISTOARKKITEHTUURI

Ohjelmistoarkkitehtuurin avulla pyritään täyttämään tietynlaiset vaatimukset ohjelmistossa. Siinä määritellään joukko rajoitteita, joiden avulla tähdätään tietynlaiseen ohjelmistorakenteeseen. Se toimii pohjana suunniteltavalle järjestelmälle, ja sen avulla saadaan kokonaiskuva järjestelmään kuuluvista osista sekä toiminnasta. Arkkitehtuuri luo pohjan järjestelmälle, mutta se saattaa myös muuttua järjestelmän kehittyessä eteenpäin. *(Kotilainen 2009, 4-5.)*

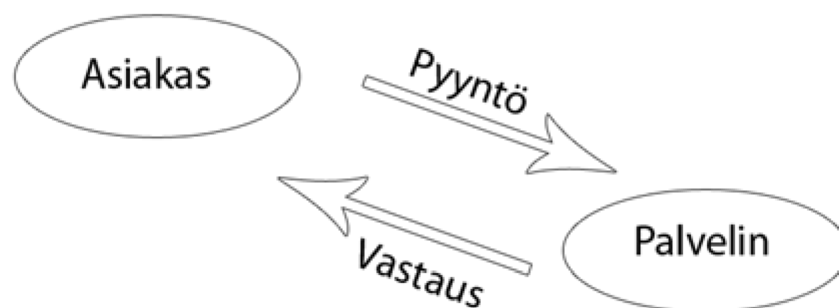
On olemassa paljon erilaisia arkkitehtuurimalleja eri tarkoitusperiä varten. Opinnäytetyössä käsiteltävä REST on myöskin arkkitehtuurimalli, tai pikemminkin hybridi-arkkitehtuurimalli. Se lainaa hyväksi todettuja, yhteensopivia osia eri arkkitehtuurimalleista, yhdistäen ne yhdeksi toimivaksi kokonaisuudeksi. Tässä luvussa käsitellään yleistasolla muutamaa arkkitehtuurimallia, jotka on sisällytetty REST-arkkitehtuuriin. Varsinainen REST-arkkitehtuuri käsitellään myöhemmässä luvussa.

3.1 Asiakas-palvelin-arkkitehtuuri

Asiakas-palvelin-arkkitehtuuri on laajalti käytössä oleva arkkitehtuurimalli, jota käytetään usein tietoverkkojen tai verkkopalvelujen yhteydessä. Arkkitehtuurissa on olennaista se, että siinä on aina vähintään kaksi osapuolta: asiakas sekä palvelin.

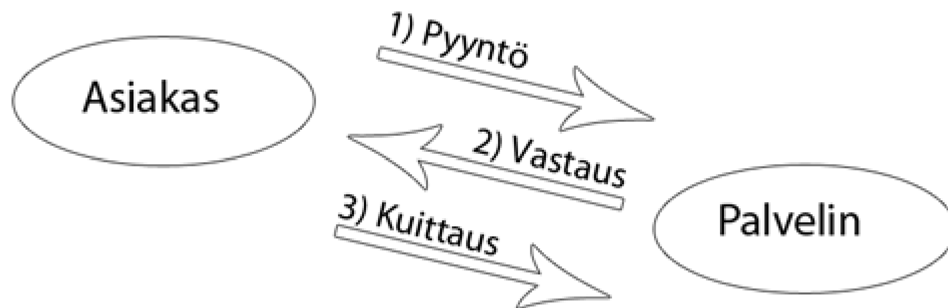
Arkkitehtuurissa asiakkaalla tarkoitetaan sellaista osapuolta, jolla on jokin tietty tehtävä, jonka se haluaa saada suoritettua. Asiakas ei tiedä tai kykene tehtävää itse suorittamaan, joten sen tarvitsee löytää joku, joka tähän kykenee. Palvelin puolestaan on sellainen osapuoli, joka palvelee asiakkaita. Se kykenee suorittamaan vaaditut tehtävät, mutta vaatii yleensä jonkinlaista perustietoa suoritettavasta tehtävästä kyetäkseen suorittamaan sen.

Asiakas ottaa yhteyden palvelimeen ja pyytää sitä suorittamaan halutun tehtävän. Palvelin ottaa pyynnön vastaan, käsittelee sen ja palauttaa asiakkaalle vastauksen suorittamastaan operaatiosta. Tämän jälkeen asiakas katkaisee yhteyden palvelimeen *(Rouse 2008)*.



Kuva 1: Asiakas-palvelin-arkkitehtuurin periaate

Arkkitehtuurin yhteydessä käytetään joskus myös niin sanottua kolmen viestin protokollaa. Tällöin asiakas lähettää palvelimelle pyynnön, palvelin palauttaa vastauksen ja asiakas kuittaa vastauksen saamisen takaisin palvelimelle. Pyyntöä lähetettäessä asiakas käynnistää eräänlaisen ajastimen. Jollei palvelin palauta viestiä tietyn aikavälin sisällä, lähetetään pyyntö uudelleen. Palvelin toimii yleensä samalla periaatteella ja poistaa pyyntöön liitetyt tiedot välimuistista vasta kuittauksen saatuaan. *(Crichlow 2001, 50.)*



Kuva 2: Kolmen viestin protokollan toimintaperiaate

3.2 Kolmikerrosarkkitehtuuri

Järjestelmästä ja koodikielestä riippuen ei aina ole mahdollisuutta avata suoraa yhteyttä sovelluksen ja palveluntarjoajan välillä. Tällaista tilannetta varten on kehitetty kolmikerrosarkkitehtuuri.

Kolmikerrosarkkitehtuuri on nimensä mukaisesti askel eteenpäin asiakas-palvelin-arkkitehtuurista. Siinä missä edellä mainitussa osapuolia oli vain kaksi (asiakas ja palvelin), on kolmikerrosarkkitehtuurissa väliin lisätty vielä kolmas osapuoli. Jokaista siihen kuuluvaa osapuolta kutsutaan termillä kerros. Näitä kerroksia löytyy kolme: ”esityskerros, toimintalogiikkakerros ja tietolähdekerros” *(Vuorenmaa 2011, 73)*.

Esityskerroksella viitataan asiakas-palvelin-arkkitehtuurista tuttuun asiakkaaseen, mutta palvelin on sen sijaan jaettu kahteen eri osaan: tietolähdekerrokseen sekä toimintalogiikkakerrokseen. Tietolähdekerros toimii eräänlaisena tietovarastona (esimerkiksi tietokanta), jonne kaikki haettava tieto on varastoitu.

Toimintalogiikkakerros sijaitsee tietolähdekerroksen ja asiakkaan välissä, ja sen tehtävänä on toimia ”asiakkaana tietokantapalvelimelle ja palvelimena asiakkaalle” *(Rantala 2005, 254)*.

Esimerkkitapauksessa asiakas ottaa yhteyden välikerrokseen ja välittää tälle tarvittaessa lisätietoa suoritettavasta operaatiosta, jonka jälkeen välikerros välittää pyynnön eteenpäin tietovarastolle. Tietovarasto prosessoi halutut tehtävät ja palauttaa vastauksen välikerrokselle, joka puolestaan palauttaa sen asiakkaalle.

4 WEB SERVICE

Ennen Web Service -käsitteen avaamista on hyvä hieman perehtyä rajapinnan käsitteeseen. Rajapinnalla tarkoitetaan ohjelmoinnissa yleisesti Ohjelmointirajapintaa (Application Programming Interface, tästä eteenpäin API). API on palvelu tai ohjelman osa, jonka avulla eri arkkitehtuurilla ja koodikielellä rakennetut järjestelmät tai ohjelmat pystyvät välittämään viestejä toisilleen, keskustelemaan keskenään käyttämällä standardisoituja pyyntöjä.

Ohjelmoijan kannalta API on eräänlainen musta laatikko. Tärkeintä ei ole tietää millä tavalla API hoitaa halutun toiminnon, vaan mitä API tarvitsee hoitaakseen toiminnon. Ohjelmoijan ei tarvitse tietää mitään toisen osapuolen rakenteesta, vaan pelkästään, minkälaista dataa hänen tulee syöttää ja missä muodossa vastaus hänelle palautetaan.

Yksinkertaisimmillaan API:a voisi kuvailla viestinviejäksi, joka toimittaa viestejä kahden osapuolen välillä. Viestinviejällä ei ole tiedossaan tarkkoja tietoja viestin toimitustavasta, tärkeintä on että viesti toimitetaan perille. Viestinviejä voi pyytää asiakasta tarkentamaan pyyntöään tai kirjoittamaan sen tietyllä kielellä. Viestin välittämisen jälkeen viestinviejä palaa takaisin lähettäjälle raportoimaan tilanteesta, usein vastauksen kera. Yksinkertaistettuna API siis määrittelee oikeanlaisen tavan lähettää pyyntöjä järjestelmään. *(Orenstein 2000.)*

4.1 Määritelmä

Web Serviceksi kutsutaan sellaisia palveluita, jotka mahdollistavat tiedon vaihtamisen erilaisten järjestelmien välillä tietoverkon yli. Järjestelmillä ei usein ole keskenään mitään yhteistä ja ne on useasti rakennettu eri tahojen toimesta, täysin eri alustoille käyttämällä eri arkkitehtuurimalleja tai koodikieliä *(W3C Web Services Architecture Working Group 2004, 1.3).*

Web Service -käsitteelle ei ole suomen kielessä suoraa vastinetta. Sanastokeskuksen mukaan termi ”www-sovelluspalvelu” on kuitenkin olemassa, vaikka työn tekijä ei termiin opinnäytetyön aikana muualla törmännytkään *(Sanastokeskus TSK: Termipankki).* Tekijä ei ylipäätään löytänyt paljon suomeksi kirjoitettua materiaalia, vaan pääasiassa kaikki tieto oli englanniksi.

Yleisesti tällaisiin palveluihin viitataan suoraan englanninkielisellä termillä ”Web Service”. Tämän vuoksi Web Servicet sekoitetaan valitettavan usein verkkopalvelun kanssa, sillä käännös on englanniksi sama. Tästä johtuen työn tekijä koki tarpeelliseksi selventää kyseisten palvelujen väliset erot lyhyesti.

Kun otetaan yhteyttä verkkopalveluun, palauttaa palvelin vastauksen yleensä yhdistelmänä HTML:ää ja CSS:ää, sillä palvelua käyttävät ihmiset ja sen avulla visualisoidaan sisältö. Käyttäjälle tarjotaan graafinen käyttö-

liittymä, jonka avulla tämä voi nähdä palvelun sisällön järkevästi jäsennettynä. Käyttöliittymän avulla käyttäjä voi myös suorittaa erilaisia toimintoja palvelussa.

Web Service taas on tarkoitettu pääasiassa järjestelmien tai ohjelmien väliseen vuorovaikutukseen. Palvelun palauttama vastaus on usein esitetty joko XML tai JSON-muotoisena, sillä ne ovat alustariippumattomia merkintäkieliä.

Palvelun palauttama vastaus jäsennetään myöhemmin (usein asiakkaan toimesta) oikeaan muotoon, jotta asiakas voi sitä käyttää tarkoitusperiensä mukaisesti. Käyttäjälle ei siis tarjota graafista käyttöliittymää toimintojen suorittamiseksi, sillä interaktio on puhtaasti laitteiden tai ohjelmistojen välistä *(Beal, 2014)*

4.2 Toimintamalli

Web Servicen perimmäinen tarkoitus on toimia yhdyshenkilönä itsensä ja sitä käyttävän asiakasohjelman välillä. Tämän yhdyshenkilön kautta asiakasohjelma pystyy käyttämään Web Serviceen linkitettyjä resursseja järjestelmän määrittämällä tavalla. Tämän opinnäytetyön käytännön osuudessa Web Serviceen linkitettyjä resursseja ovat tietokannan tiedot, ja niitä käytetään HTTP 1.1 spesifikaation mukaisten kutsujen avulla.

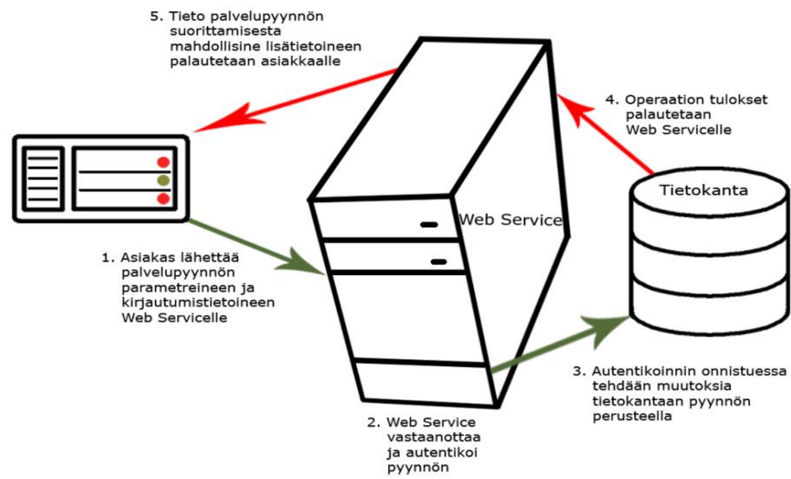
Web Servicet käyttävät hyödykseen asiakas-palvelin-arkkitehtuuria, jossa asiakas esittää pyynnön, Web Service suorittaa pyynnön edellyttämät toiminnot ja palauttaa asiakkaalle viestin tuloksista. Pyyntöjen välittämiseen käytetään usein (muttei aina) HTTP-protokollaa.

Asiakas ei tiedä miten Web Service halutun operaation hoitaa, eikä tällä usein ole asiakkaalle merkitystä. Asiakkaan täytyy kuitenkin kertoa Web Servicelle tarvittavat operaatioon liittyvät tiedot oikeassa muodossa, jotta operaatio voidaan suorittaa. Tätä varten Web Servicen yhteyteen laaditaan yleensä koneen luettavissa oleva dokumentti (XML tai JSON), jossa selitetään sen keskeisimmät toiminnot, niiden käyttötarkoitus, parametrit ja vastauksen paluumuoto. *(W3C Web Services Architechture Working Group 2004, 1.4)*

Riippuen Web Servicen rakentamiseen käytetystä arkkitehtuurista ja menetelmistä (REST/SOAP) saattaa pyynnön lähettämisestä palvelulle muodostua varsin monimutkainenkin tapahtuma. Esimerkiksi SOAP-pohjaiset Web Servicet käyttävät ennalta määriteltyä standardia pyyntöjen muodosta ja käsittelystä, kun taas REST-pohjaisilla palveluilla tällaista käytäntöä ei ole, mikä helpottaa kommunikointia ja vähentäen turhaa työtä.

Käytännön esimerkkinä Web Servicestä mainittakoon tässä opinnäytetyössä toteutettu Web Service, jonka kautta oldtimerTimer-mobiilisovelluksen ja tietokannan välinen tiedonsiirto tapahtuu. Asiakassovellus lähettää Web Servicelle tarpeelliset tiedot, jonka jälkeen Web Service hoitaa vaaditut tehtävät. Lopuksi mobiilisovellukselle palautetaan operaatiosta vastaus

tietoineen. Palvelin myös tallentaa resursseja välimuistiin tiedonhaun nopeuttamiseksi, ja vaatii tiettyihin resursseihin pyrkiviltä tunnistautumista.



Kuva 3: Web-Servicen toimintaperiaate

5 WEB SERVICE -TEKNOLOGIAT

Web Service voidaan toteuttaa monella eri tavalla, eikä toteutukseen ole olemassa pelkästään yhtä oikeaa tapaa. Yleisesti valitaan se toteutustapa, joka sopii valitun järjestelmän yhteyteen ja vastaa parhaiten asetettuja määrityksiä. Web Service -teknologiat voidaan rajata kahteen pääluokkaan: REST-pohjaisiin toteutuksiin, joissa tietolähteet jaotellaan resursseihin ja joita muokataan representaatioiden kautta, sekä hieman mielivaltaisempiin toteutuksiin joissa palvelu paljastaa käyttäjälle joukon suoritettavissa olevia operaatioita *(W3C Web Services Architechture Working Group 2004, 3.1.3).*

Tässä luvussa keskitytään pääasiassa REST-arkkitehtuuriin ja sen rajoitteisiin, mutta käydään läpi vertailun vuoksi myös SOAP-pohjaista toteutustapaa.

5.1 REST-arkkitehtuuri

Representational State Transfer (tästä eteenpäin REST) on arkkitehtuurimalli, jonka kulmakiviä ovat keveys, ylläpidettävyys ja skaalautuvuus. Se on kokoelma sääntöjä jotka toimivat hyvin yhdessä, eräänlainen koostettu ohje, jota seuraamalla luodaan REST-pohjainen palvelu. Nämä säännöt on lainattu eri arkkitehtuurityyleiltä, eli tietyllä tapaa REST on sekoitus useaa eri tyyliä. REST muun muassa käyttää hyödykseen asiakas-palvelin-arkkitehtuuria, jossa se tarjoaa asiakkaalle pääsyn resursseihin ja asiakas määrittää resursseille suoritettavat operaatiot lähettämällä HTTP-pyynnön, joka sisältää tarpeelliset tiedot pyynnön suorittamiseksi.

Perusideana on, että vaikka erilaiset järjestelmän osat muuttuisivat itsenäisesti, säilytetään silti yhteentoimivuus niiden välillä. Asiakas, palvelin sekä tietovarasto voivat näin kehittyä omaa tahtiaan, sillä ne pystyvät aina suorittamaan eri osapuolille tarkoitetut pyyntönsä samalla tavalla kuin ennenkin, vaikka osapuolet muuttuisivatkin.

REST-arkkitehtuurin perusideana on tilan (eli datan) muuttaminen, oli se sitten asiakkaan tai palvelimen päässä. Tulkitaksemme sanoja hieman tarkemmin, Representational viittaa käsiteltävän datan, eli resurssin muotoon. Datan nykyisestä muodosta luodaan representaatio, ilmentymä, jossa data esitetään uudessa muodossa (esimerkiksi XML tai JSON). State puolestaan viittaa juurikin tähän datan muotoon, sillä sekä tietovarasto (palvelin) että asiakassovellus sisältävät tietyn version resurssista, eli resurssin tämänhetkisen tilan. Transfer viittaa tämän tilan siirtämiseen palvelimelta asiakassovellukseen ja toisinpäin käyttämällä järjestelmässä määriteltyjä kutsuja. Tähän sisältyvät niin resurssin tilan osittainen ja kokonainen päivittäminen, uuden resurssin luominen sekä poistaminen *(Tampereen Teknillinen Yliopisto 2010a, 3).*

Arkkitehtuurimallin määritteli vuonna 2001 tietokoneasiantuntija Roy Fielding osana tohtorintutkintoaan. Hän on myös aikanaan osallistunut mm. HTTP-spesifikaation kehitystyöhön sekä Apache HTTP Server -projektin

pohjatyöhön. Tästä johtuen ei olekaan kumma, että REST ja HTTP sopivat hyvin yhteen.

REST-arkkitehtuurissa määritellään kuusi ”ehdotonta rajoitetta”, joita REST-pohjaisten järjestelmien tulee noudattaa. Kaikki muut näiden ohella määritellyt säännöt eivät ole ehdottomia määräyksiä vaan pikemminkin ohjenuoria ja hyväksi todettuja käytäntöjä paremman REST-palvelun luomiseen. SOAP-pohjaisista toteutuksista poiketen tarkoituksena ei ole seurata näitä sääntöjä orjallisesti, vaan niin tarkasti kuin vain on teknisesti ja käytännöllisesti kannattavaa. Tästä syystä REST-pohjaiset palvelut suunnitellaan yleensä asiakkaan näkökulmasta helppokäyttöisiksi ja kehittäjän näkökulmasta vaivattomasti ylläpidettäviksi. *(Fredrich 2013, 6.)*

Syitä RESTin suosion kasvuun varsinkin Web Service-kehityksen yhteydessä on useita. Siinä missä REST-pohjaiset palvelut saadaan toteutettua nopeasti, ovat SOAP ja muut WSDL-pohjaiset toteutustavat ovat usein aikaa vaativia. Lisäksi ne sisältävät paljon erilaisia sääntöjä, joita ohjelmoijan on pakko noudattaa palvelua rakentaessaan. REST-pohjaisen toteutuksen oppimiskynnys on huomattavasti matalampi, ja töihin päästään kiinni nopeammin. Rajoitteidensa vuoksi se on myös alusta- sekä kieliriippumaton, eli tietoa voidaan syöttää ja pyytää monessa eri muodossa. Keveytensä vuoksi se sopii myös erinomaisesti mobiilipohjaisien järjestelmien yhteydessä käytettäväksi.

5.2 REST-arkkitehtuurin Rajoitteet

REST-arkkitehtuuri määrittelee kuusi rajoitetta, joita REST-pohjaisen palvelun tulee noudattaa. Rajoitteet on lainattu useasta eri arkkitehtuurimallista ja ne on asetettu takaamaan toimivuus eri resurssien välillä *(Fielding 2000, 5.1)*. Tässä alaluvussa käydään läpi arkkitehtuurissa määriteltyjä rajoitteita.

5.2.1 Asiakas-palvelin

REST-pohjainen järjestelmä on jaettu selkeästi kahteen osapuoleen, asiakkaaseen ja palvelimeen. Palvelin on vastuussa asiakkaan lähettämien viestien vastaanottamisesta ja käsittelystä (hylkäys, hyväksyminen, vastaus). Asiakas puolestaan on vastuussa viestien lähettämisestä oikeanlaisessa muodossa ja vastausten käsittelemisestä tarpeidensa vaatimalla tavalla.

”Jakamalla järjestelmä selkeästi kahteen osaan, erottamalla käyttöliittymä tietovarastosta, parannetaan käyttöliittymän siirrettävyyttä erilaisille alustoille sekä koko palvelun skaalautuvuutta yksinkertaistamalla serveripuolen komponentteja” *(Fielding 2000, 5.1.2).*

5.2.2 Tilaton

REST-pohjainen Web Service on tilaton, eli palvelimelle ei tallenneta tietoja suoritetuista kyselyistä. Kyselyiden tuloksia voidaan tosin tallentaa välimuistiin, mikäli turhaa liikennöintiä palvelun ja tietokannan välillä halu-

taan välttää. Käytännössä tämä tarkoittaa sitä, että jokaista kutsua kohdellaan yksilönä. Uudet kyselyt eivät saa olla riippuvaisia aiemmin suoritettujen kyselyiden tuloksista, vaan jokaisen kyselyn tulee sisältää tarpeelliset tiedot operaation suorittamiseen tavalla, jolla palvelin voi suorittaa kyselyn *(Fielding 2000, 5.1.3).*

5.2.3 Yhtenäinen rajapinta

SOAP-pohjaisessa toteutuksessa palveluntarjoaja yleensä määrittelee itse rajapintansa eli ne operaatiot joita voidaan kutsua. Tästä poiketen kaikilla REST-pohjaisen järjestelmän resursseilla tulee olla yhtenäinen rajapinta, eli operaatiot suoritetaan samalla tavalla resurssista riippumatta. Tämä saavutetaan Web Servicen yhteydessä yleensä käyttämällä HTTP-protokollaa ja siinä määriteltyjä verbejä operaatioiden suorittamiseen, joista kerrotaan enemmän tämän opinnäytetyön loppupuolella. Yhtenäisellä rajapinnalla taataan se, että kaikkia resursseja on mahdollista käsitellä samalla tavalla, eikä jokaiseen resurssiin tarvita täysin omia komentoja.

Edellä mainittuun rajoitteeseen kuuluu myös muutamia alirajoitteita. Nämä rajoitteet määräävät, että tietovaraston tiedot on jaettu resursseihin ja niihin viitataan yksilöivällä tavalla, esimerkiksi HTTP-protokollan tapauksessa käyttämällä Unique Resource Identifieriä (tästä eteenpäin URI). Näitä resursseja ei koskaan palauteta kutsujalle suoraan, vaan niistä luodaan tietyn muotoinen representaatio (esimerkiksi XML muodossa), joka palautetaan varsinaisen tuloksen sijaan.

Edellä mainittuja resursseja tulee hallita niiden representaatioiden kautta. Tällä tarkoitetaan sitä, että representaation vastaanotettuaan asiakkaalla on tarpeeksi tietoa suorittaakseen operaatioita kyseiseen resurssiin. Tähän sisältyvät myös header-tiedot sisällön muodosta ja siitä onko resurssi varastoitu välimuistiin.

Viimeiseksi nämä alirajoitteet kuvailevat HATEOAS-käsitettä. HATEOAS tulee sanoista Hypermedia as the Engine of Application State, ja sillä määritetään miten tila tulee siirtää asiakkaan ja palvelimen välillä.

Palvelin siirtää tilan asiakkaalle sisällyttämällä varsinaisen sisällön body-osioon ja metadatan header-osioon. Asiakas puolestaan siirtää tilan kohdistamalla pyynnön tiettyyn resurssiin käyttämällä URIa, sisällyttämällä halutut metatiedot header-osioon sekä resurssin tilan body-osioon. Asiakas voi myös lisätä pyyntöön rajoitteita query-parametrien avulla.

Palvelimen palauttamat vastaukset tulisi myös koota sellaisiksi, että ne sisältävät tietoa resurssille suoritettavista operaatioista esimerkiksi linkkien muodossa. *(*Fredrich *2013, 7.)*

5.2.4 Mahdollisuus käyttää välimuistia

Pyörää ei kannata keksiä uudelleen. Mikäli samaa kyselyä suoritetaan monta kertaa uudelleen, ei tietoa kannata aina hakea uudestaan tietovarastosta. Resursseihin merkitään, saako niitä siirtää välimuistiin vai ei. Näin

kyselyn saapuessa voidaan resurssi lähettää saman tien takaisin ilman yhteyttä tietovarastoon, tai vastaavasti, jos välimuisti sijaitsee asiakkaalla, ei tietoa tarvitse hakea palvelimelta ollenkaan.

Tällä tavoin vähennetään, tai parhaassa tapauksessa poistetaan kokonaan turhia, aikaa vieviä interaktioita järjestelmästä ja parannetaan näin koko palvelun keveyttä ja skaalautuvuutta. Servicen lisäksi myös asiakassovellukseen on mahdollista (ja usein jopa järkevää) toteuttaa jonkinlainen välimuisti-ominaisuus.

5.2.5 Kerroksittainen järjestelmä

Palvelun tulee olla koottu siten, että osapuolten välille on mahdollista asettaa esim. palomuureja tai proxyja, ilman että asiakkaan ja palvelimen välisiä rajapintoja täytyy muokata muutosten yhteydessä. Asiakas ei tiedä, onko se suorassa yhteydessä varsinaiseen tietovarastoon, vaiko palvelimeen. Käytännössä tämä tarkoittaa sitä, että esimerkiksi tietokerros ei aseta palautettavaan dataan muotoiluja, vain välittää pelkästään raakadatan seuraavalle kerrokselle. Seuraava kerros puolestaan lisää siihen muotoilut ja lähettää sen taas seuraavalle kerrokselle. *(Fielding 2000, 5.1.6.)*

5.2.6 Code on Demand

Kuudes rajoite on itse asiassa ehdollinen. Sen mahdollistaa REST-palvelun asiakassovellusten laajentamisen ilman, että kaikkia ominaisuuksia olisi kehitetty etukäteen. Se määrittelee koodin jakamisen sallittavaksi REST-palvelun kautta (appletit, skriptit). Tällöin asiakassovellukset pystyvät pyytämään palvelulta tietyn toiminnon suorittavia koodeja, jotka ne voivat ladata ja suorittaa paikallisesti. Tämä kuitenkin vähentää näkyvyyttä, joten se on määritelty ehdolliseksi rajoitteeksi. *(Fielding 2000, 5.1.7.)*

5.3 Resurssipohjaisuus REST-arkkitehtuurissa

REST-pohjaiset Web Servicet keskittyvät varsinaiseen sisältöön ja sen esittämiseen, sillä päätarkoituksena on tuoda palveluun linkitetty sisältö asiakassovelluksen käyttöön. REST-arkkitehtuurissa tämä sisältö jaetaan resursseiksi, jotka nimetään usein substantiiveilla (esimerkiksi käyttäjät). Resurssit eivät välttämättä sijaitse samalla palvelimella REST-palvelun kanssa, vaan niihin päästään käsiksi juuri REST-palvelua käyttämällä (vertauskuvallisesti REST-palveli toimii ikkunana resursseihin).

RESTin rajoitteissa määrätään, että resursseja täytyy hallita yhtenäisen rajapinnan kautta. Tämän lisäksi jokainen resurssi tulee voida yksilöidä, ja sitä tulee käsitellä representaationsa kautta *(Fielding 2000, 5.1.5)*. Nämä rajoitteet voidaan toteuttaa käyttämällä hyödyksi HTTP-protokollaa, joka sisältää valmiit keinot tätä varten. Jokaiselle resurssille määrätään yksilöivä Unique Resource Identifier (URI), jonka kautta sitä käsitellään HTTP-metodien välityksellä. HTTP-protokollaa käsitellään enemmän myöhemmässä luvussa.

REST-arkkitehtuurissa määritellään myös, ettei asiakkaalle koskaan palauteta tiedon varsinaista lähdettä, vaan esimerkiksi tekstityyppinen representaatio lähteestä *(Fielding 2000, 5.2.1)*. Resursseja on mahdollista luoda melkein mistä tahansa tietolähteestä, ja ne voivat myös koostua kokoelmasta toisia resursseja (esimerkiksi resurssi, joka sisältää kaikki käyttäjä-resurssit).

Yleisesti on hyvä paljastaa resurssin sisältö osa kerrallaan, varsinkin jos dataa on paljon tai käsitellään yhden resurssin sijasta kokoelmaresurssia. Tällaisissa tapauksissa resurssikokoelman representaatio voi sisältää kaiken tiedon sijasta jokaisen resurssin URI:n. Yksittäiseen resurssiin päästään myöhemmin käsiksi tämän URI:n avulla, eikä kokoelman tulosten läpikäyminen ole enää niin sekavaa.

Resursseja käsitellään pyyntöjen avulla. Kuten työssä aiemmin mainittiin, REST-pohjaiset Web Servicet käyttävät usein hyödykseen HTTP-protokollaa näiden pyyntöjen välittämiseksi. Protokollassa on määritelty rajattu joukko erilaisia operaatioita, jotka kohdistetaan tietyn URI:n avulla haluttuun resurssiin. Käyttämällä hyödyksi näitä metodeja, voidaan jokaista resurssia hallita samalla tavalla, ja toimitaan yhtenäinen rajapinta -rajoitteen mukaisesti.

SOAP-pohjaisissa Web Serviceissä lähetettävät pyynnöt sisältävät pääasiassa XML-muotoisen operaatiokutsun, kun taas REST-pohjaisessa toteutuksessa keskitytään operaatiokutsun sijaan varsinaiseen dataan, lähettämällä varsinainen data kutsun mukana *(Tampereen Teknillinen Yliopisto 2010a, 4)*.

Toisin kuin SOAP-pohjaisessa toteutuksessa, RESTissä pyynnöillä ei ole ennalta määriteltyä muotoa (vrt. SOAP message ja envelope), vaan muoto voi olla mikä tahansa, niin kauan kunhan sekä palvelu että asiakas ymmärtävät sen. Servicen palauttama vastaus voi myös olla useassa muodossa, joista pyynnön lähettänyt asiakas valitsee yhden käyttämällä HTTP-protokollassa määriteltyä accept-headeria (josta kerrotaan enemmän seuraavassa kappaleessa). Yleisimmät pyynnöissä ja vastauksissa käytetyt paluumuodot ovat XML ja JSON, joskin tämän opinnäytetyön käytännön osuudessa tuetaan pelkästään XML:ää.

5.4 Simple Object Access Protocol

Simple Object Access Protocol (tästä eteenpäin SOAP) on REST-arkkitehtuurin ohella yksi suosituimmista tavoista rakentaa Web Service. Toisin kuin REST-palvelut, jotka rakennetaan usein asiakassovelluksen tarpeiden mukaan, SOAP pohjautuu pitkälti sääntöihin ja standardeihin ja seuraa tarkasti Service Specification-määritystä, jossa kerrotaan mitä kaikkea SOAP-pohjaisen palvelun pitää tehdä. Toisin kuin REST-arkkitehtuurissa, jossa säännöt ovat pikemminkin ohjenuoria, SOAP-pohjaisessa toteutuksessa säännöt ovat ehdottomia, eikä niistä saa poiketa. Jos yksikin näistä ehdoista ei toteudu, palvelu ei ole SOAP-palvelu. Siinä määritellään tarkasti, minkälaisessa muodossa palvelupyynnöt tulee sille lähettää.

5.4.1 Toimintamalli

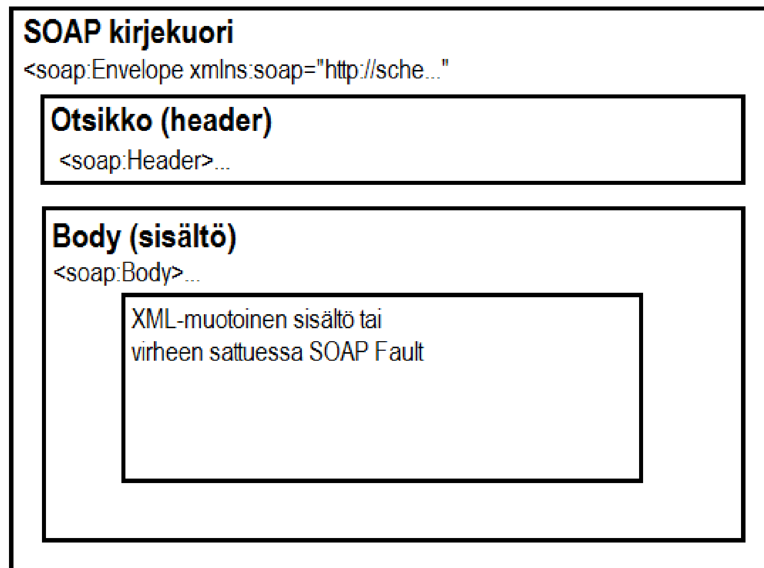
REST- sekä SOAP-pohjainen Web Service eroavat toisistaan konseptiensa kannalta huomattavasti. Siinä missä REST jaottelee tietolähteet resursseiksi ja keskittyy resurssin tilan siirtämiseen, SOAP-pyynnöt sisältävät tarkkaan määritellyn operaatiokutsun, joka aiheuttaa tietyn operaation suorittamisen palvelussa. Lisäksi SOAP-toteutuksen yhteydessä joudutaan laatimaan erillinen Web Service Description Language -dokumentti (tästä eteenpäin WSDL), jossa kerrotaan Web Servicestä ja sen toiminnasta kaikki oleellinen pyyntöjen muodostamista varten. Tämä dokumentti on kirjoitettu XML-muotoiseksi ja se jaetaan palvelun käyttäjille.

WSDL-dokumenttia käytetään SOAP-pohjaisissa palveluissa, eikä REST-pohjaisissa toteutuksissa vastaavaa dokumenttia yleensä ole. Jos REST-pohjaiset palvelut sisältävät vastaavanlaisen dokumentoinnin, on se yleensä esitetty jotenkin ihmisen luettavissa olevassa, visuaalisessa muodossa, esimerkiksi listauksena toiminnoista verkkosivuna. REST-arkkitehtuurin yhteyteen on kuitenkin myös kehitetty WSDL:n kaltainen dokumentointitapa nimeltään WADL, mutta se ei ole kovinkaan suuressa suosiossa. Pääsyynä tähän lienee se, että REST-palvelu rakennetaan yleensä itseään kuvaavaksi, jolloin dokumenttia ei pitäisi tarvita lainkaan.

Kuten REST-pohjaisessa palvelussa, myös SOAP-palvelussa asiakkaan täytyy täsmentää pyyntöään ottaessaan yhteyttä Web Serviceen. Palvelupyynnön täytyy olla sellaisessa muodossa, jonka molemmat osapuolet varmasti ymmärtävät. Tätä muotoa kutsutaan protokollaksi.

SOAP-pohjaiset Web Servicet käyttävät niin sanottua standardisoitua protokollaa (SOAP-protokolla) tiedon välittämiseen. Se koostuu kolmesta osasta: kirjekuori, otsikko ja sisältöosio. Kirjekuori muodostaa kehyksen varsinaiselle viestille, sisältäen otsikon sekä sisällön. Sisältöosio sisältää varsinaisen sisällön, eli pyynnön joka halutaan suorittaa. Tämä pyyntö on kirjoitettu XML-muotoiseksi seuraamalla WSDL-dokumentissa lueteltuja sääntöjä. Sisältösioon merkitään myös mahdolliset virhetiedot, joita kutsutaan SOAP Faultiksi.

Viimeinen osa, eli otsikkotieto, on vapaaehtoinen. Sen avulla voidaan kuitenkin välittää lisätietoa kutsuun tai esimerkiksi kutsujaan liittyen. *(Tampereen Teknillinen Yliopisto 2010b, 28.)*



Kuva 4: SOAP-viestin rakenne

Tiivistettynä voidaan siis todeta, että varsinainen kommunikointi järjestelmien välillä hoidetaan SOAP-protokollan avulla, ja viestien sisältö ja rakenne määritellään WSDL-dokumentissa. Käsitteen avaamiseksi paremmin kuvataan SOAPin toimintaa esimerkin avulla:

Palvelun A täytyy kutsua palvelussa B sijaitsevaa toimintoa. Molemmat palvelut on koodattu eri kielillä ja ne pyörivät eri alustoilla. Palvelu B:n yhteyteen on kehitetty SOAP-pohjainen Web Service, joka mahdollistaa toimintojen suorittamisen kutsumalla palvelua. Palvelu A:n kehittäjä pyytää palvelu B:ltä tämän WSDL-dokumentin, luo sen pohjalta oikeanlaisen pyynnön ja lähettää pyynnön palvelulle B.

Tästä huomataan kuinka tärkeä WSDL-dokumentti on SOAP-pohjaisen toteutuksen kannalta, sillä ilman sitä ei asiakassovelluksen laatija pysty lähettämään viestejä palvelulle oikeassa muodossa.

6 HTTP-PROTOKOLLA

Yksi REST-arkkitehtuurin peruspilareista on yhtenäisyys. Tästä johtuen REST-pohjaiset Web Servicet käyttävät usein HTTP-protokollaa resurssien käsittelemiseen (eli pyyntöjen ja vastausten lähettämiseksi), sillä sen avulla voidaan toteuttaa yhtenäinen rajapinta -rajoite.

SOAP-pohjaisissa Web Serviceissä lähetettävät viestit sisältävät pääasiassa tarkasti määrätyn XML-muotoisen operaatiokutsun SOAP Envelopen sisällä. Tästä poiketen REST-pohjainen Web Service keskittyy operaatiokutsun sijaan varsinaiseen dataan lähettämällä varsinaisen datan pyynnön mukana. Datalle suoritettava operaatio puolestaan määräytyy pyyntöön sisältyneen verbin, eli metodin avulla. *(Tampereen Teknillinen Yliopisto 2010a, 4).*

6.1 Unique Resource Identifier

Resurssit identifioidaan URI:n avulla. URI on yleensä substantiivi joka kuvastaa resurssia, esimerkiksi resurssin nimi. Jokapäiväisessä internet-käytössä linkkejä muodostetaan antamalla ensin protokolla, palvelun osoite, URI sekä parametrit. Näin on myös REST-arkkitehtuurissa, sillä WWW on itse asiassa suurin esimerkki REST-arkkitehtuuria noudattavasta palvelusta.

REST-arkkitehtuurin rajoitteissa määritellään, että jokaisella resurssilla täytyy olla ainakin yksi URI. Sen avulla etsitään pyynnölle oikea resurssi. Se, mitä kyseiselle resurssille tehdään, määräytyy pyyntöön syötetyn HTTP-metodin perusteella. Perussääntönä URI:lle määritellään, ettei siitä saa käydä ilmi suoritettavaa operaatiota. Tästä syystä seuraava osoite olisi epäkelpo, sillä siitä käy selvästi ilmi suoritettavan operaation luonne:
`http://localhost/REST/Persons/Edit.`

WWW-maailmassa käytetään myös usein erilaisia query-paremetreja pyynnön selventämiseksi. Parametreina voidaan esimerkiksi syöttää numero, jonka perusteella pyynnön palauttamaa tulosjoukkoa rajoitetaan. REST-arkkitehtuuri ei varsinaisesti kiellä query-parametrien käyttöä osana URI:a. Tavallisesti niiden käyttöä kuitenkin vältetään, sillä parametrien määrän kasvaessa ohjelmakoodista tulee vaikeaselkoisempaa ja mutkikkaampaa.
(Vaqqas 2014, 1.)

6.2 Metodit

URI:n lisäksi HTTP-pyyntö sisältää myös HTTP-metodin, jota kutsutaan toisinaan myös verbiksi. Verbi on se pyynnön osa, joka määrittelee resurssille suoritettavan toiminnon. HTTP-protokolla määrittelee joukon metodeja, joista jokainen on hyödyllinen REST-pohjaisen Web Servicen kannalta. Tällaisia metodeja ovat GET, POST, PUT, DELETE, OPTIONS sekä HEAD.

GET-metodia käytetään resurssin lukemiseen. Resurssin tilaa ei tällöin muuteta ollenkaan, vaan palvelu pelkästään palauttaa resurssin tämänhetkisen tilan tietovarastosta. Resurssin tilan muuttamiseksi on olemassa POST-

metodi, jota käytetään uuden resurssin luomiseen. Jos resurssi on jo olemassa, metodi yksinkertaisesti pyrkii päivittämään sen tiedot syötetyillä tiedoilla.

POST-metodin lisäksi on olemassa PUT-metodi. Se on toiminnaltaan pitkälti samantapainen POST:in kanssa, paitsi että sitä käytetään pääasiassa kokonaisen resurssin päivittämiseen. POST-metodista poiketen päivitys vaatii kuitenkin täyden URI:n, eli sitä ei voida kohdistaa esimerkiksi resurssikokoelmaan. Sen avulla on mahdollista myös luoda uusi resurssi, mutta tätä ei REST-arkkitehtuurissa suositella, kuten ei myöskään resurssin osittaista päivittämistä. Resurssin osittaiseen päivitykseen suositellaan PATCH-metodia, vaikkei se olekaan "virallinen" metodi.

Lopuksi on olemassa DELETE-metodi, joka nimensä mukaisesti poistaa valitun resurssin. Näiden lisäksi on olemassa kaksi vaihtoehtoista metodia, joita ei välttämättä käytetä kovin usein, mutta jotka ovat hyödyllisiä REST-arkkitehtuurin kannalta: OPTIONS ja HEAD.

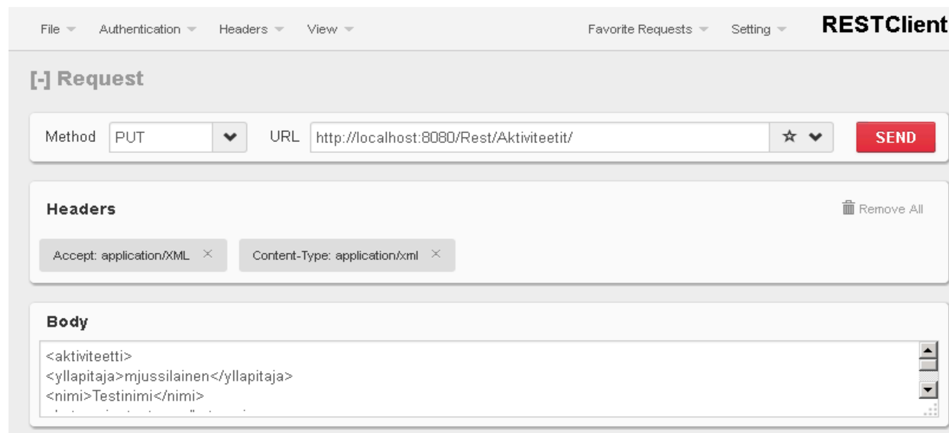
OPTIONS on erittäin hyödyllinen metodi tilanteesta riippuen, sillä sen avulla saadaan selville, mitkä operaatiot ovat sallittuja millekin resurssille. HEAD puolestaan on hyvin samankaltainen GET-metodin kanssa, sillä erotuksella että vastauksena palautetaan vain viestin HEADER-osio, ei ollenkaan BODY:a Tämä verbi on hyödyllinen pienten tarkistusoperaatioiden, kuten resurssin olemassa olemisen tarkistamiseksi. Edellä mainituista metodeista GET ja HEAD ovat turvallisia, eli niiden avulla ei ole mahdollista muokata tai lisätä resursseja.

6.3 Header ja Body

Kuten aiemmin todettiin, URI:lla kerrotaan mihin resurssiin pyyntö halutaan kohdistaa, ja metodilla määritetään sille suoritettava operaatio. Näiden lisäksi pyyntöön kuuluvat headerit sekä body.

Header pitää sisällään kaiken pyyntöön liittyvän metadatan, kuten esimerkiksi asiakkaan tukemat ja pyytämä vastaustyyppi. Metadata esitetään avain-arvo-pareina, joista kuuluisimmat ovat ehkä Content-Type ja Accept, joita käytetään resurssin muodon neuvotteluun. *(Vaqqas, 1-2.)*

Body puolestaan on viestin varsinainen sisältöosa, joka sisältää varsinaisen lähetetyn tai pyydetyn datan. Siihen sijoitetaan esimerkiksi POST-metodia käyttävän pyynnön resurssi, joka halutaan luoda. Vastaavasti palvelu sijoittaa siihen GET-metodia käyttäneen pyynnön yhteydessä haetun resurssin nykyisen tilan, esimerkiksi XML tai JSON-muodossa, riippuen asiakkaan Accept-headeriin kirjoittamasta arvosta.



Kuva 5: Esimerkki HTTP-pyynnöstä RESTClient-sovelluksessa esitettynä

6.4 HTTP-statuskoodit

Kun Web Service palauttaa asiakkaalle vastauksen suorittamastaan operaatiosta, olisi asiakkaan kannalta oleellista tietää, miten operaatiossa kävi. Vastaus tähän ongelmaan löytyy HTTP-statuskoodeista. Pyynnön epäonnistuessa asiakassovellus pystyy koodien avulla selvittämään, mikä meni vikaan, ja mahdollisesti korjaamaan pyyntöä ennen seuraavan lähettämistä. Vastaavasti pyynnön onnistuessa tiedetään, että enempää pyyntöjä ei enää tarvita, sillä operaatio on jo suoritettu.

Statuskoodit ovat jokaisen kutsun yhteydessä palautettavia kolminumeroisia koodeja, jotka kertovat pyynnön tilan ja auttavat asiakasta selvittämään nopeasti, onnistuiko pyyntö odotetusti. REST-arkkitehtuurin yhteydessä useimmiten käytetyt koodit voidaan jakaa karkeasti kolmeen ryhmään: 20x, 40x sekä 50x.

Virhekoodit jotka kuuluvat ryhmään 20x kuvastavat sitä, että pyyntö onnistui. Pyyntö siis vastaanotettiin onnistuneesti, palvelin ymmärsi pyynnön ja hyväksyi sen. Tärkeimpiä tähän ryhmään kuuluvia koodeja ovat 200 eli OK, 201 eli Created sekä 204 eli No Content.

Ryhmään 40x kuuluvat virhekoodit puolestaan tarkoittavat sitä, että pyynnön aikana tapahtui virhe joka johtui asiakkaasta. Asiakas saattoi esimerkiksi syöttää tiedon virheellisessä muodossa, tai pyynnöstä puuttui jokin osa. Tärkeimpiä tähän ryhmään kuuluvia koodeja ovat 400 eli Bad Request, 401 Unauthorized, 403 Forbidden, 404 Not Found sekä 409 Conflict. REST-pohjaisessa palvelussa kannattaa myös käyttää hyödyksi koodeja 405 Not Allowed, sekä 415 Unsupported Media Type.

Viimeinen tärkeä ryhmä on 50x eli Server Error. Tämän luokan virheet tarkoittavat myöskin virhettä pyynnön aikana, mutta tällä kertaa aiheuttajana on palvelin eikä asiakas kuten 40x ryhmän virheissä. Palvelin tietää kohdanneensa virheen, jota se ei pysty käsittelemään. Ryhmän tärkeimmät koodit ovat 500 Internal Server Error sekä 503 Service Unavailable.
(***The Internet Society 1999, 10.1-10.4***)

[-] Response

Response Headers | Response Body (Raw) | Response Body (Highlight) | Response Body (Preview)

```
1.  Status Code       : 200 OK
2.  Content-Length    : 246
3.  Content-Type      : application/xml
4.  Date              : Fri, 02 Oct 2015 18:53:15 GMT
5.  Link              :
    <http://localhost:8080/Rest/Kayttajat/msalo/>; title="URI"; rel="self", <http://localhost:8080/Rest/Kayttajat/msalo/das>;
6.  Server            : GlassFish Server Open Source Edition  4.1
7.  X-Powered-By      : Servlet/3.1 JSP/2.3 (GlassFish Server Open Source Edition  4.1  Java/Oracle Corporation/1.7)
```

Kuva 6: Palvelun palauttama HTTP-statuskoodi ja Headerit

7 REST-ARKKITEHTUURIIN SISÄLTYVÄT TEKNOLOGIAT

7.1 JAX-RS

Java API for RESTful Web Services (tästä eteenpäin JAX-RS) on spesifikaatio, jossa määritellään miten REST-pohjaisia Web Servicejä voidaan luoda Java-ohjelmointikielellä. Se sisältää vain määrityksen, eli ohjelmoijan täytyy tehdä varsinainen toteutus. Aikaa voidaan kuitenkin säästää käyttämällä valmista kehystä (framework), joka tukee JAX-RS määrittelyä toteuttamalla tarvittavat luokat. Tällä tavalla säästyy aikaa, kun ohjelmoijan ei tarvitse koodata kaikkea itse *(Kothagal 2015).*

Kehyksen valinnalla ei ole sen suurempaa käytännön merkitystä. Kaikki REST-kehityksen tukemiseen valmistetut kehykset on luotu JAX-RS spesifikaation pohjalta, eli niiden perusominaisuudet ja -toiminnot ovat aina samat. Kun opettelee yhden kehyksen, opettelee samalla suuren osan muistakin. Jokainen kehys sisältää kuitenkin myös kehittäjän omia toimintoja, joihin tarvitsee tutustua erikseen.

Esimerkkejä näistä kehyksistä ovat muun muassa Restlet, Spring ja Jersey, joista Jersey on mallitoteutus (reference implementation). Sen kehittämiseen osallistuivat monet niistä ihmisistä, jotka kehittivät varsinaisen JAX-RS-spesifikaation. Opinnäytetyön tekijä valitsikin Jerseyn kehyksekseen, sillä sen on todettu olevan aloittelijaystävällisempi uusille REST-palvelun kehittäjille.

7.2 Java Persistence API

Tiedon tallentamisen tueksi tietokantaan Javaan on kehitetty Java Persistence Api (tästä eteenpäin JPA). Se mahdollistaa tavallisten POJO (Plain Old Java Object)-luokkien tallentamisen suoraan tietokantaan. Opinnäytetyön yhteydessä käytettiin JPA:n EclipseLink-toteutusta, joka on JPA:n mallitoteutus.

JPA:ta hyödyntävät luokat ovat tavallisia olio-luokkia, jotka edustavat tauluja tietokannassa. Näiden luokkien avulla tietoa voidaan hakea, päivittää tai lisätä tietokannasta luokaksi tai luokasta tietokantaan. Tietokannan ja luokan välinen relaatio tapahtuu joko lisäämällä luokkiin annotaatioita, tai luomalla erillinen XML-tiedosto, joka sisältää määritykset. *(Mukesh 2013, JPA Introduction.)*

REST-pohjaisessa toteutuksessa JPA vaatii kolme asiaa: Entity luokan, Persistence Unitin sekä JAX-RS-luokan. Näistä Entity-luokka on aivan tavallinen Java-luokka, joka kuvastaa tietokannan rakennetta siihen merkityillä JPA-annotaatioilla. Näiden annotaatioiden avulla määritellään, miten luokan sisältämät elementit liittyvät tietokantaan.

Persistence Unit puolestaan on se osapuoli, joka hoitaa tiedon hakemisen ja tallentamisen varsinaisen tietolähdekerroksen (tietokanta) puolelta. Sen tärkein osa on persistence.xml-tiedosto, jossa Persistence Unitin varsinainen

määrittely tapahtuu. Tähän tiedostoon merkitään Persistence Unitiin kuuluvat JPA Entity-luokat, käytettävä yhteystyyppi sekä tietolähde (DataSource) ja mahdollisesti sen tiedot, kuten salasana ja tunnus.
(Mukesh 2013, JPA Introduction)

JAX-RS-luokka (tai Jersey-luokka) puolestaan määrittelee, minkälaisia operaatioita resursseille voidaan suorittaa, mitä ne vastaanottavat, palauttavat ja missä muodossa**.** Näitä luokkia on aina yksi yhtä REST-resurssia kohden, ja ne määritellään yleensä käyttämään omia Persistence Unitejaan. Se kutsuu varsinaisia JPA:n tietokantaoperaatioita suorittavia metodeja, jotka sijaitsevat usein toisessa luokassa.

7.2.1 JPA-annotaatiot

Jotta tieto voidaan tallentaa tietokantaan, täytyy taululla ja luokalla olla yhtenäinen rakenne. Yhtenäisyys toteutetaan sijoittamalla JPA-annotaatioita taulukon kenttiä vastaavien muuttujien yhteyteen.

Luokka merkitään tietokantaan tallennettavaksi luokaksi lisäämällä annotaatio @Entity ennen luokan määrittelyä. Tietokannan sarakkeita vastaavat muuttujat merkitään annotaatioilla @column, ja taulun perusavainta kuvaava muuttuja tämän lisäksi annotaatiolla @Id. *(Mukesh 2013, JPA Introduction.)*

Oletuksena JPA antaa taululle ja sarakkeille muuttujia vastaavat nimet.
Taulun nimen voi määritellä itse tekemällä seuraavat muutokset @Entity-annotaation perään:

```
41    @Entity
42    @Table(name = "aktiviteetit")
```

Kuva 7: Taulun nimen määrittäminen JPA:ssa

Vastaavasti sarakkeiden nimet voidaan määrittää tekemällä seuraavat muutokset @Column-annotaatioon:

```
80        @Column(name = "id")
81        private Integer id;
```

Kuva 8: Taulun sarakkeiden nimeäminen JPA:ssa

Mikäli tiettyä kenttää ei koskaan haluta tallentaa tietokantaan, voidaan se merkitä annotaatiolla @Transient.

JPA:ssa on lisäksi olemassa oma kyselykielensä, Java Persistence Query Language, jonka avulla pystytään hoitamaan halutut kyselyt resursseille. Se on syntaksiltaan SQL-kieltä muistuttava, mutta kyselyistä tulee huomattavasti lyhempiä. Ohjelmoijan ei tarvitse erikseen syöttää SQL-komentoja järjestelmään, vaan hän voi työskennellä suoraan olioluokkien kanssa.

7.3 Java Architecture for XML Binding

Java Architecture for XML Binding (tästä eteenpäin JAXB) on spesifikaatio olio-luokkien muuntamiseksi XML-muotoon sekä päinvastoin. Käytännössä sen toiminta on pitkälti samankaltaista JPA:n kanssa, sillä molemmat sijoittavat annotaatioita suoraan Java-luokkiin. Niitä käytetäänkin usein hyödyksi samassa luokassa tiedon muuntamiseen ja tallentamiseen samanaikaisesti.

JPA:n tapaan myös JAXB pohjautuu annotaatioihin, jotka kirjoitetaan Java-luokkiin. Näiden annotaatioiden perusteella määritellään XML-dokumentin rakenne. JAXB-implementaatio käy ajon aikana läpi nämä annotaatiot, ja muodostaa niiden perusteella XML-rakenteen. Rakenteen avulla voidaan XML-muotoinen data muuntaa olio-luokaksi (unmarshalling), tai olio-luokka XML-muotoiseksi dataksi (marshalling). Riippuen suoritetusta operaatiosta voidaan luotu XML-muotoinen vastaus palauttaa käyttäjälle tai tallentaa Java-luokan tiedot tietokantaan käyttämällä JPA:ta. *(Ort &Mehta 2003.)*

8 KÄYTÄNNÖN SOVELLUS

Opinnäytetyön käytännön työnä toteutettiin REST-arkkitehtuuria noudattava Web Service oldtimerTimer-järjestelmälle. Opinnäytetyön puitteissa resursseina toimivat mobiilisovelluksen käyttäjät tietoineen, sekä sovelluksen kautta tarkasteltavat aktiviteetit tapahtumineen.

Web Servicen kautta on mahdollista listata, luoda, muokata tai poistaa siihen linkitettyjä resursseja. Pääkäyttötarkoituksena on Web Servicen myöhempi hyödyntäminen oldtimerTimer-järjestelmän mobiilisovelluksessa sekä tulevaisuudessa myös www-sivuston puolella, esimerkiksi aktiviteettien luomisessa. Nämä vaativat kuitenkin oman asiakassovelluksen tekemisen tai koodaamisen suoraan mobiilisovellukseen, joten ne jätettiin opinnäytetyön ulkopuolelle.

Web Service toteutettiin REST-arkkitehtuurin mukaisesti ja se pyörii Glassfish-sovelluspalvelimella, joka puolestaan sijaitsee virtuaalikoneella. Toteutus on helppo siirtää myöhemmin halutessa toiselle sovelluspalvelimelle, mutta uuden palvelimen tulee tukea Javan EE-ominaisuuksia (esim. TomctEE ja EJB lisäosa). Se on yhteydessä toisella virtuaalikoneella, Apachen alla pyörivään MySQL-tietokantaan, jonka sisältö toimii Web Servicen resursseina.

Kehitystyöhön käytettiin NetBeans-sovelluskehitintä, johon ladattiin Restful Web Services-lisäosa. Tämä lisäosa lisäsi Jersey 2.1-kehyksen (JAX-RS:n referenssi implementaatio) luokat projektiin.

Suurin osa kehitystyössä käytetystä opiskelumateriaalista löytyi verkosta, ja se oli kirjoitettu englanniksi. Suomenkielisiä lähteitä ei juurikaan löytynyt. Kirjallisuutta käytettiin myös jonkin verran, mutta lähteet sisälsivät enemmänkin REST-arkkitehtuurissa lainattuja menetelmiä eivätkä käsitelleet suoranaisesti REST-arkkitehtuuria.

8.1 REST-pohjainen Web Service

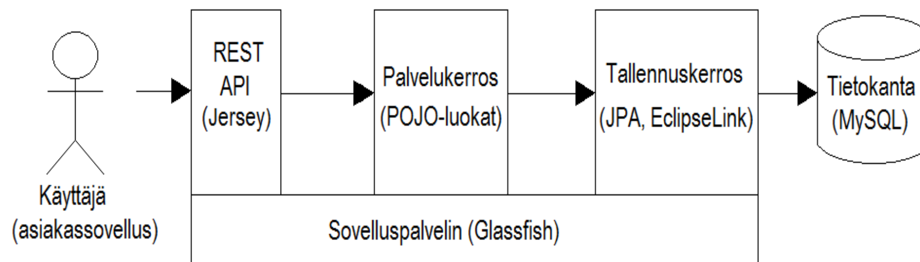
Web Serviceä, joka seuraa REST-arkkitehtuurissa määriteltyjä ohjeita, kutsutaan nimellä ”Restful Web Service”. Nimitystä käytetään tästä eteenpäin.

Työn alkuvaiheessa asennettiin työympäristönä toimineelle virtuaalikoneelle NetBeans-sovelluskehitin, jonka avulla varsinainen kehitystyö tapahtui. Sovelluskehittimeen ladattiin myöhemmin myös Restful Web Services -lisäosa, joka mahdollisti Web Service-kehityksen sovelluskehittimessä.

Tietokantayhteys MySQL-palvelimen ja sovelluskehittimen välillä osoittautui odotettua vaativammaksi tehtäväksi, johtuen sovelluskehittimen vähäisestä käyttökokemuksesta sekä eri ohjelmaversioille kirjoitetuista ohjeista. Ongelmia aiheutui myös työympäristön virtuaalisuudesta, sillä palvelin ja Web Service eivät aluksi pystyneet kommunikoimaan keskenään ollenkaan.

Kuten aiemmin mainittu, opinnäytetyössä Web Servicen resursseina toimivat MySQL-tietokannassa sijaitsevat tiedot. Näihin tietoihin päästiin käsiksi JPA:n kautta, jota sovelluskehitin hallinnoi pääsääntöisesti itse. Tietokantarakenteet ja operaatioiden logiikka tuli kuitenkin kirjoittaa itse; sovelluskehitin loi tätä varten vain suuntaa antavan rungon.

Seuraava kuva kuvastaa työn rakennetta. REST-arkkitehtuurin mukaisesti logiikka on jaettu eri osastoihin (kolmikerrosarkkitehtuuri). Varsinaiseen REST-rajapintaan (JAX-RS / Jersey), POJO-luokkiin (tiedot) sekä Persistence Layeriin (JPA/eclipselink). Nämä osastot puolestaan pyörivät Glassfish-sovelluspalvelimella.



Kuva 9: RESTful Web Servicen rakenne

8.2 Kirjoitettuja luokkia

8.2.1 Resurssiluokat

Opinnäytetyössä kirjoitettiin seuraavat JPA:ta hyödyntävät Entity- eli resurssiluokat: Aktiviteetti, Kayttaja sekä Tapahtuma. Nämä luokat ovat toiminnaltaan hyvin samankaltaisia (muutamine erotuksineen) ja niiden päätarkoituksena on mallintaa tietokannan rakenne Java-tiedostoihin. Yksi luokka mallintaa aina yhtä tietokannan taulua.

JPA-annotaatiot kirjoitettiin suoraan luokkiin, sillä erillisten tiedostojen tekemisestä ei todettu tässä tapauksessa olevan paljoa etua. Luokissa on käytetty hyödyksi myös JAXB-määritystä, joka toimii JPA:n lailla kirjoittamalla luokkiin annotaatioita. Siinä missä JPA-annotaatiot mallintavat tietokannan rakennetta, JAXB-annotaatiot mallintavat XML-dokumentin rakennetta luokissa. JAXB-annotaatioiden avulla luokasta voidaan luoda XML-muotoisia vastauksia.

Aktiviteetti-luokka mallintaa tietokannan Aktiviteetit-taulun rakennetta. Taulukossa listataan käyttäjien luomat aktiviteetit kuvauksineen ja osoitetietoineen. Lisäksi sinne on merkitty aktiviteetin ylläpitäjä, joka on yksi käyttäjistä. Aktiviteetti-luokka sisältää myös listan Tapahtuma-luokista, jotka kuuluvat kyseiseen aktiviteettiin.

Tapahtuma-luokka mallintaa Aktiviteetin tapaan tietokannan Tapahtumat-taulun rakennetta. Taulukkoon on listattu kaikkien aktiviteettien tapahtumat

tietoineen, kuten Aktiviteetti johon tapahtuma kuuluu, sen alku- ja loppuajankohdat sekä kuvaus. Tapahtumat yksilöidään id-numeron perusteella. Luokka ei eroa ulkoasultansa kovinkaan paljoa Aktiviteetti-luokasta, pois lukien erilaiset hienosäädöt tulostettavien XML-tietojen puolella.

Kayttaja-luokka seuraa kahden edellisen luokan jalanjälkiä. Sen avulla mallinnetaan tietokannan käyttäjät-taulua, jossa listataan jokainen rekisteröitynyt käyttäjä. Tapahtuma-luokan tapaan tämäkin luokka poikkeaa muista lähinnä pelkästään tulostettavien XML-tietojen ja tietokantaan tallentamattomien kenttien kohdalla.

```
59  public class Kayttaja implements Serializable {
60
61      //kayttajatunnus-sarake tietokannassa
62      @Id
63      @Basic(optional = false)
64      @NotNull
65      @Size(min = 1, max = 20)
66      @Column(name = "kayttajatunnus_V")
67      private String kayttajatunnus;
68
69      //Set
70      public void setKayttajatunnusV(String kayttajatunnusV)
71      {this.kayttajatunnus = kayttajatunnusV;}
72
73      //Get
74      @XmlElement(name = "kayttajatunnus")
75      public String getKayttajatunnusV()
76      {return kayttajatunnus;}
```

Kuva 10:Pieni esimerkki Kayttaja-luokan rakenteesta

8.2.2 Persistence Unit

Projektin Persistence Unitit määritellään Persistence.xml-tiedostossa. Siinä määritellään jokainen Persistence Unitin käyttämä yhteysmuoto tietoineen, sekä kaikki ne entity-luokat joita Persistence Unitin kautta käsitellään. Tämän tiedoston pohjalta luodaan ja konfiguroidaan Entitymanager-luokat, joiden avulla tietokannan tietoja voidaan muokata. Työssä on määritelty kaksi Persistence Unitia: yksi käyttäjille ja toinen Aktiviteeteille sekä tapahtumille.

8.2.3 Palveluluokat

Nämä luokat ovat vastuussa Web Serviceen saapuvien pyyntöjen vastaanottamisesta ja välittämisestä seuraavalle kerrokselle. Niitä kirjoitettiin opinnäytetyössä kolme: AktiviteettiFacadeREST, tapahtumaFacadeREST sekä KayttajaFacadeREST. Kaikki edellä mainitut luokat toteuttavat AbstractFacadeREST–rajapintaluokan, joka sisältää tarvittavat metodit tiedon siirtämiseen tietokannan ja Web Servicen välillä. Näin peruslogiikkaa ei tarvitse koodata erikseen jokaisen luokan kohdalla, vaan voidaan kutsua yläluokan metodeja.

Pyynnön saapuessa tarkistetaan ensin sen polku sekä HTTP-metodi. Näiden avulla luokasta valitaan oikea metodi, joka käsittelee kyseisen pyynnön.

Valittu metodi sisältää tiettyjä tarkistuksia, jotka se tekee ennen varsinaista käsittelemistä. Näihin voivat kuulua XML-datan oikeellisuus, tietyn parametrin olemassaolo tai kutsun header-arvojen tarkistaminen. Näiden tarkistusten jälkeen tehdään vielä kutsukohtaisia tarkistuksia, kuten onko lisättävä resurssi jo olemassa vai ei. Virheiden sattuessa kutsutaan virheen tyypille sopivaa virheenkäsittelijäluokkaa, joka käsittelee virheen ja hoitaa vastauksen palauttamisen käyttäjälle. Mikäli virheitä ei ilmene, kutsutaan AbstractFacadeREST-luokan metodeja, jotka kykenevät suorittamaan vaaditut tietokantaoperaatiot.

```
@POST
@Path("{id}/tapahtumat")
public Response addTapahtuma(@PathParam("id") Integer id, @Valid Tapahtuma tapahtuma) {

    //Etsitään aktiviteetti
    Aktiviteetti activity = super.find(id);

    //Jos aktiviteettia ei löydy, palautetaan virhe
    if(activity == null)
        throw new NotFoundException("Aktiviteettia " + id + " ei löytynyt.");

    //Asetetaan tapahtuman aktiviteetiksi nykyinen aktiviteetti
    tapahtuma.setAktiviteetti(activity);

    //Lisätään tapahtuma aktiviteetin tapahtumalistaan (jolloin se tallennetaan tietokantaan)
    activity.getTapahtumaCollection().add(tapahtuma);

    em.flush();

    // Otetaan linkki talteen
    URI uri = info.getAbsolutePathBuilder().path("/"+tapahtuma.getId()).build();
    return Response.created(uri).build();
}
```

Kuva 11: Esimerkki service-luokan metodin rakenteesta

8.2.4 Virheenkäsittelijäluokat

Opinnäytetyössä kirjoitettiin muutamia virheenkäsittelijäluokkia. Nämä luokat ovat kukin määritelty vastaanottamaan ja käsittelemään tietyntyyppiset virheet, joita palvelun käyttämisen aikana voi ilmetä. Yleisin käyttötapaus lienee virheellinen URL tai puutteelliset tiedot.

Luokkien käsittelemiä virheitä ovat ConstraintViolationException, EntityExistsException, JAXBException, NotFoundException, NumberFormatException sekä WebApplicationException. Virheitä ei käsitellä tässä sen laajemmin.

Kirjoitettuja virheenkäsittelijäluokkia on 6 kappaletta, joista kukin käsittelee aina tietyn tyyppiset virheet. Luokkia kutsuessa ne vastaanottavat virheen tiedot ja luovat virheen tyypistä riippuen kuvauksen, josta ilmenee virheen syy. Esimerkiksi WebApplicationExceptionMapper-luokka tarkistaa ensin virheen HTTP-statuskoodin ja valitsee sen pohjalta virheeseen sopivan kuvauksen.

```java
@Provider
public class WebApplicationExceptionMapper implements ExceptionMapper<WebApplicationException>{
    @Override
    public Response toResponse(WebApplicationException exception)
    {
        //Haetaan alkuperäinen error response
        Response r = exception.getResponse();

        switch(r.getStatus()){
            //Method not allowed
            case 405:{
                return Response.status(r.getStatus()).entity(
                    Development.Tools.toXML(
                        new ErrorMessage(r.getStatus(),r.getStatusInfo().toString(),"HTTP-metodin käyttö tähän resurssiin ei ole sallittu.
                    )
                ).type(MediaType.APPLICATION_XML).build();
            }
```

Kuva 12: Esimerkki virheenkäsittelijäluokan rakenteesta (WebApplicationException)

Käyttäjälle palautetaan XML-muotoinen vastausviesti, josta käy ilmi virheen koodi, syy sekä kuvaus. Kuvauksen perusteella käyttäjän tulisi kyetä jatkamaan itsenäisesti eteenpäin, ellei kyseessä ole palvelintason virhe, johon käyttäjä ei voi vaikuttaa.

Tämä virheviesti on generoitu ErrorMessage-apuluokan avulla. Se vastaanottaa virheen tiedot, joiden perusteella luokasta voidaan luoda XML-versio, joka puolestaan palautetaan käyttäjälle virheen sattuessa. Kyseessä on tavallinen Java-luokka, johon on upotettu JAXB-annotaatioita XML-muotoilua varten. Lähes kaikki virheenkäsittelijäluokat käyttävät tätä luokkaa hyödykseen lähettäessään virhetietoja käyttäjälle.

[-] Response

Response Headers | Response Body (Raw) | Response Body (Highlight) | Response Body (Preview)

```xml
1. <?xml version="1.0" encoding="UTF-8" standalone="yes"?>
2. <Error>
3.     <code>405</code>
4.     <status>Method Not Allowed</status>
5.     <message>HTTP-metodin käyttö tähän resurssiin ei ole sallittu.</message>
6. </Error>
```

Kuva 13: Käyttäjälle palautettu virheviesti, joka on generoitu ErrorMessage-luokan avulla

8.2.5 Muut luokat

Muita luotuja luokkia ovat erilaiset apuluokat. PACTH-luokka mahdollistaa PATCH-pyyntöjen (yksittäisen kentän päivitys koko resurssin sijaan) tukemisen Web Servicessä, sillä kyseinen metodi ei kuulu yleisimmin käytettyihin HTTP-metodeihin, eikä sovelluskehitin tukenut tätä oletus asetuksilla.

Tools-luokka sisältää joukon erilaisia kehitystyötä ja debuggausta auttavia metodeja, kuten URI:n rakentamisen osista, viestien kirjaaminen lokiin tai tiedon muuntaminen XML-muotoon.

9 TULOKSET & POHDINTAA

Opinnäytetyön päätutkimuskysymykset käsittelivät Web Service-käsitettä, Web Servicen toteuttamista REST-arkkitehtuuria hyödyntäen sekä sen sopivuutta OldTimerTimerin yhteyteen. Opinnäytetyön aikana todettiin, että REST-arkkitehtuurilla toteutettu Web Service vastaa hyvin OldTimerTimerin tarpeisiin sekä vaatimuksiin, korvaten aiemmin toteutetut kehnommat ja hitaammat menetelmät. Tietolähteet on jaettu resursseihin, joita pystytään muokkaamaan helposti ja nopeasti HTTP-yhteyden yli käyttämällä URL:eja sekä tarvittaessa XML-muotoista dataa.

Palvelu ilmoittaa asiakassovellukselle onnistuneista operaatioista sekä niiden aikana mahdollisesti tapahtuneista virheistä, käyttämällä HTTP-statuskoodeja sekä selkokielisiä kuvauksia. Näin oldtimerTimer-järjestelmän mobiilisovellus, sekä mahdollisesti tulevaisuudessa myös WWW-palvelu pystyvät reagoimaan tapahtumiin oikeanlaisella tavalla pelkästään tarkastelemalla Web Servicen palauttamaa HTTP-statuskoodia. HTTP-pyynnöt eivät myöskään vie mobiilisovellukselta turhan paljoa resursseja, verrattuna nykyiseen toteutukseen jossa jokainen resurssi joudutaan hakemaan palvelimelta yksi kerrallaan. Aktiviteettien puolella dataa pystytään filtteröimään syöttämällä pyynnön mukana esimerkiksi halutun kaupungin nimi. Näitä filttereitä voidaan myöhemmin rakentaa lisää sitä mukaa, minkälaisia rajoitusvaihtoehtoja käyttäjät sattuvat tulevaisuudessa tarvitsemaan.

Resurssien sisältämät tiedot saadaan tarpeen vaatiessa ulos eri muodoissa, ja järjestelmä varmistaa tiedon ajantasaisuuden sekä oikeamuotoisuuden. Näin ollen mobiilisovelluksen puolella tiedon esitysmuotoa ei ole enää rajattu vain XML-muotoon, vaan se voidaan tilanteen vaatiessa esittää esimerkiksi JSON-muodossa.

Käyttäjätietojen muuttamisen yhteydessä Web Service vaatii käyttäjätunnusta sekä salasanaa, jotka lähetetään header-tietoina kutsun yhteydessä. Tähän yhteyteen voitaisiin tulevaisuudessa kehitellä API key -toiminto, jolloin sovelluksille jaettaisiin tiettyjä käyttöoikeuksia sisältävä avain, jonka avulla pyyntöjä voitaisiin suorittaa tiedustelematta tunnuksia. Nykyinen toteutus toimii, kunhan palvelua käytetään HTTPS:n yli. Toinen vaihtoehto olisi korvata nykyinen autentikointitoiminto käyttämällä kehittyneempää OAuth-autentikointia. Tutkismusprosessin aikana jouduttiin kuitenkin toteamaan, että opinnäytetyöhön käytettävissä oleva aika ei enää tulisi riittämään OAuth-pohjaiseen toteutukseen. Vastaavanlaista autentikointitoimintoa ei myöskään ole vielä kehitetty aktiviteettien yhteyteen.

Tulevaisuudessa Web Serviceen voidaan lisätä uusia ominaisuuksia, kuten esimerkiksi toiminto mobiilisovelluksen aktiviteettinäkymän elementtien järjestelemiseksi. Näkymiä varten voitaisiin luoda resurssi, josta elementtien järjestelytiedot haetaan XML-muodossa näkymän luonnin yhteydessä. Näin jokaisesta näkymästä tai vaihtoehtoisesti näkymäryhmästä saataisiin tarkoitusperiin sopiva näkymä.

OldtimerTimerin yhteyteen on aiemmissa projekteissa ollut suunnitteilla myös tilaustoiminto, jonka avulla asiakkaat kykenisivät tilaamaan itselleen

erilaisia palveluja, kuten siivous-, kauppa tai kuljetuspalveluja. Mikäli ominaisuus päätetään myöhemmin lisätä, voidaan se toteuttaa helposti käyttämällä jo olemassa olevia toimintoja mallina. Tämän toiminnon kanssa kannattaa kuitenkin kiinnittää entistä enemmän huomiota kunnolliseen tietoturvaratkaisuun.

Mikäli oldtimerTimer-järjestelmään kaivataan tulevaisuudessa tiukempia määrittelyjä, tai jos REST-pohjainen toteutus todetaan uusien ominaisuuksien kehityksen aikana kokonaisuuteen sopimattomaksi, voi SOAP-pohjaisen Web Servicen kehittäminen olla ajankohtainen asia. Tässä tapauksessa kannattaa kuitenkin ottaa huomioon se, että samalla menetetään tiettyjä REST-pohjaisen toteutuksen mukanaan tuomia etuja, kuten JSON-tuki. SOAP-pohjainen toteutus on lisäksi paljon REST-pohjaista toteutusta raskaampi.

10 LÄHTEET

Beal, V. 2014. Web Services. Viitattu 25.5.2015.
http://www.webopedia.com/TERM/W/Web_Services.html

Chrichlow, J. 2001. Hajautetut tietojärjestelmät. Helsinki: Edita.

Fielding, R. 2000. Architectural Styles and the Design of Network-based Software Architectures. Väitöskirja. University of Carolina, Information and Computer Science. Viitattu 28.9.2015.
http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm

Fredrich, T. 2013. RESTful Service Best Practices. Best-practices document. Viitattu 15.7.2015.
https://github.com/tfredrich/RestApiTutorial.com/raw/master/media/RESTful%20Best%20Practices-v1_2.pdf

Kothagal, K. 2015. REST Web Services 10 – What is JAX-RS?
Viitattu 17.9.2015.
https://www.youtube.com/watch?v=BuYivu9ZjDw

Kotilainen, O. 2009. 58309102 Seminaari: Palvelusuuntautuneet järjestelmät. Pdf-dokumentti. Viitattu 17.8.2015.
http://www.cs.helsinki.fi/group/cinco/teaching/2009/soc-seminaari/papers/kotilainen_paper.pdf

Mukesh, K. 2013. JPA Introduction. Viitattu 24.9.2015.
http://www.javawebtutor.com/articles/jpa/jpaintro.html

Orenstein, D. 2000. Application Programming Interface.
Viitattu 25.5.2015.
http://www.computerworld.com/article/2593623/app-development/application-programming-interface.html

Ort, E. & Mehta, B. 2003. Java Architecture for XML Binding.
Viitattu 12.11.2015.
http://www.oracle.com/technetwork/articles/javase/index-140168.html

Rantala, A. 2005. Web Ohjelmointi. Jyväskylä: Docendo.

Rouse, M. Client/Server Definition. 2008. Viitattu 25.5.2015.
http://searchnetworking.techtarget.com/definition/client-server

Sanastokeskus TSK:n termipankki. N.d. Viitattu 8.7.2015.
http://www.tsk.fi/cgi-bin/netmot.exe?UI=figr&height=161&qfind=verkkopalvelu

Tampereen Teknillinen Yliopisto. 2010a. OHJ-5201 Web-palveluiden toteutustekniikat. Pdf-dokumentti. Viitattu 17.7.2015.
http://www.cs.tut.fi/kurssit/OHJ-5201/materiaali/9.pdf

Tampereen Teknillinen Yliopisto. 2010b. OHJ-5201 Web-palveluiden toteutustekniikat. Pdf-dokumentti. Viitattu 9.10.2015.
http://www.cs.tut.fi/kurssit/OHJ-5201/materiaali/4.pdf

The Internet Society. 1999. HyperText Transfer Protocol.
Viitattu 1.10.2015.
http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html

Vaqqas, M. 2014. Restful Web Services: A Tutorial. Viitattu 13.7.2015.
http://www.drdobbs.com/web-development/restful-web-services-a-tutorial/240169069

Vuorenmaa, P. 2011. Digitaalisen Median Perusteet. Luentomoniste.
Viitattu 28.5.2015.
https://noppa.aalto.fi/noppa/kurssi/t-111.2400/materiaali/T-111_2400_luentomoniste_0.2.pdf

W3C Web Services Architecture Working Group. 2004. Web Service Architecture. Viitattu 28.5.2015. http://www.w3.org/TR/ws-arch/