Matias Hirvonen

KOMPOSIITTISOVELLUKSEN MUODOSTAMINEN PALVELUKESKEISEN ARKKITEHTUURIN WEB-PALVELUISTA

JYVÄSKYLÄN YLIOPISTO
TIETOJENKÄSITTELYTIETEIDEN LAITOS
2014

TIIVISTELMÄ

Hirvonen, Matias Juhani
Komposiittisovelluksen muodostaminen palvelukeskeisen arkkitehtuurin web-palveluista
Jyväskylä: Jyväskylän yliopisto, 2014, 108 sivua
Tietojärjestelmätiede, pro gradu-tutkielma
Ohjaajat: Hirvonen, Pertti. Puuronen, Seppo.

Palvelukeskeisten arkkitehtuurien nauttiessa kasvavaa huomiota tietojärjestelmien suunnittelussa, komposiittisovelluksien tehokas käyttö on noussut avainasemaan liiketoimintasovellusten toteuttamisessa. Tutkielmassa selvitetään, mitä palvelukeskeinen arkkitehtuuri, web-palvelut ja komposiittisovellukset ovat ja toteutetaan palveluita käyttävä komposiittisovellus.

Tutkielmassa selvitetään kolmen eri määritelmän mukaiset palvelukeskeisen arkkitehtuurin perusperiaatteet. Perusperiaatteet ristiintaulukoidaan, mikä helpottaa komposiittisovelluksen kannalta olennaisimpien periaatteiden tunnistamista. Ristiintaulukointia käytetään apuna sovelluksen suunnittelussa. Sovelluksen toteuttamisen jälkeen selvitetään, toteutuivatko periaatteet käytännön sovelluksessa.

Toteutettua sovellusta arvioidaan palvelukeskeisen arkkitehtuurin, ohjelmoijan ja ohjelmistosuunnittelijan näkökulmasta. Komposiittisovelluksen todetaan soveltuvan hyvin palvelukeskeiseen arkkitehtuuriin ja tehostavan sovelluskehitystä. Lähestymistavan todetaan aiheuttavan myös uusia haasteita, mutta niiden arvioidaan jäävän hyötyjä vähäisemmäksi.

Avainsanat: palvelukeskeinen arkkitehtuuri, komposiittisovellus, web-palvelu

ABSTRACT

Hirvonen, Matias Juhani
Building a composite application from service-oriented architecture's web services
Jyväskylä: University of Jyväskylä, 2014, 108 pages
Information Systems Science, Master's Thesis
Supervisors: Hirvonen, Pertti. Puuronen, Seppo.

As the interest in SOA is increasing, effective implementation and use of composite applications has become crucial for business application development. This study will examine what service oriented architecture, web service and composite application are. A composite application, which uses web-services, is also implemented.

This study explains the basic principles of service-oriented architecture based on three different definitions. Principles are cross tabulated, which assist in identifying the most important principles for composite applications. Results of cross tabulation are used when designing the composite application. After implementing the composite application, the results are used for checking if the principles were followed.

Implemented application is evaluated from service-oriented architecture's, programmer's and software designer's perspective. The composite application is found to suit well to service-oriented architecture and to decrease application development time. The approach is found to cause some new challenges, but the benefits are estimated to be greater than the shortcomings.

Keywords: service oriented architecture, composite software, web service, service design

KUVIOT

TAULUKOT

SISÄLLYS

1 JOHDANTO

Ohjelmistoarkkitehdeille ja yritysten johtajille suunnatussa kirjassaan Agile-Path-yrityksen toimitusjohtaja Eric Marks esittää, että laajoja tietojärjestelmäarkkitehtuureja toteutettaessa palvelukeskeisten arkkitehtuurien (Service-oriented Architecture, SOA) asema on muuttunut vaihtoehdosta enemmänkin välttämättömyydeksi. Syyksi hän näkee erityisesti Internetin kasvaneen roolin yritysten sisäisen ja ulkoisen tiedonvälityksen kanavana. Marksin mukaan kaupalliset tarpeet ovat luoneet pohjaa verkon yli käytettävien palveluiden jatkuvaan kehittämiseen. (Marks, 2003, luku 1)

Tutkielmassa tarkastellaan palvelukeskeistä arkkitehtuuria yleisesti hyväksytyn, muun muassa IBM:n esimerkkiarkkitehtuurina käyttämän viisitasoisen perusmallin pohjalta. (IBM, 2004) Viisitasoisen jaon perusteella selvitetään, mitä ominaispiirteitä kunkin arkkitehtuuritason palveluilla ja sovelluksilla on. Komposiittisovelluksia tutkitaan tunnistamalla eri sovellustyyppejä seuraavien laajasti käytettyjen määritelmien perusteella:

- Valtion teknillisen tutkimuskeskuksen tutkijat Julia Kantorovitch ja Eila Niemelä (2009).
- Stevens Institute of Technology-yliopiston professorit Haim Kilov ja Ira Sack (2009).
- Zayedin yliopiston professori Zakaria Maamar (2009).
- Ohjelmistoarkkitehti Jean-Jacques Dubray (2007).

Palvelukeskeisen arkkitehtuurin määritelmistä puolestaan selvitetään, mitä perusperiaatteita määritelmän mukaisen palvelun tulee täyttää. Perusperiaatteita tutkitaan seuraavien palvelukeskeisen arkkitehtuurin määritelmien perusteella:

- Surreyn ylipiston professori Alan W. Brown sekä Rational Softwaren arkkitehdit Simon Johnston ja Kevin Kelly (2002).
- Kaikkien aikojen suosituimpien SOA-kirjojen kirjoittaja Thomas Erl (2007).
- InnoQ-konsulttiyrityksen perustaja ja InfoQ:n kirjoittaja Stefan Tilkov (2007).

Komposiittisovelluksen kannalta keskeisimpien periaatteiden tunnistamista helpotetaan ristiintaulukoimalla eri määritelmien perusperiaatteet.

Palvelukeskeisessä arkkitehtuurissa sovellukset tarjoavat toimintojaan useimmiten web-palveluina, jolloin arkkitehtuurin tasojen välinen kommunikaatio tapahtuu alemman kerroksen palveluita kutsuen. Ensisijaisena tutkimusongelmana selvitetään, soveltuvatko palvelukeskeisen arkkitehtuurin web-palvelut komposiittisovelluksen muodostamiseen. Tältä osin tutkimusmenetelmä on konstruktiivinen. Soveltuvuutta testataan toteuttamalla palvelukeskeisen lähestymistavan mukaisia palveluita sekä niitä käyttävä komposiittisovellus. Teknisen toteutuksen pohjaksi selvitetään, millainen rooli web-palveluilla on komposiittisovelluksessa ja jaotellaan palvelutyyppejä arkkitehtuurin ja toteutustavan perusteella. Toteutusta arvioidaan evaluoivalla tutkimusmenetelmällä ja selvitetään, noudattaako esimerkkisovellus palvelukeskeisten arkkitehtuurien perusperiaatteita.

Komposiittisovelluksia ja niiden soveltuvuutta palvelukeskeiseen arkkitehtuuriin on tutkittu etenkin viime vuosina enenevissä määrin. Tutkielman näkökulmaan soveltuvaa aineistoa on saatavilla paljon artikkelien ja kirjojen muodossa. Määritelmiä vertaillessa pyritään valitsemaan parhaiten tutkimusongelman selvittämistä tukevat määritelmät. Palvelukeskeistä arkkitehtuuria käsitel-

lään yleisellä tasolla hyvin esimerkiksi Josuttisin (2007) kirjassa ”SOA in Practice”. Palvelukeskeisen lähestymistavan vuoksi teknologiset valinnat eivät ole suuressa roolissa tutkimusongelman selvittämisessä, joten toteutusteknologiat voidaan valita koodin yksinkertaisuutta ja luettavuutta korostaen. Useita sovellustason teknisiä ominaisuuksia on selvitetty muun muassa Goncalvesin (2013) kirjassa ”Beginning Java EE 7”. Tutkimusongelmaan paneudutaan syvällisesti arvioimalla toteutettua sovellusta sekä palvelukeskeisen arkkitehtuurin teorian että käytännön ohjelmistokehityksen näkökulmista. Arvioinnissa hyödynnetään palvelukeskeisen arkkitehtuurin perusperiaatteiden ristiintaulukointia sekä kirjoittajan vuosien käytännön työkokemusta web-sovellusten kehittämisestä. Kirjoittaja on työskennellyt vuodesta 2005 alkaen ohjelmoijan ja ohjelmistoarkkitehdin rooleissa. Vaikka arviointi on subjektiivista, pyrkii se myös kriittisyyteen.

Tutkielma jakautuu karkeasti kahteen osaan. Luvuissa 2-4 esitetään konstruktiivisen osan taustaksi tarvittavaa teoriaa. Luvuissa 5-6 esitellään toteutettu sovellus ja arvioidaan sovellusta esitetyn teorian näkökulmasta. Tutkielman tuloksia on hyödynnetty menestyksekkäästi käytännön sovelluskehitystyössä keskisuuressa suomalaisessa ohjelmistoalan yrityksessä.

2 KOMPOSIITTISOVELLUKSIIN LIITTYVÄÄ PERUSKÄSITTEISTÖÄ

Tässä luvussa kerrotaan, mitä komposiittisovellukset ovat ja luodaan katsaus niiden historiaan. Tärkeimpien termien määrittelemisen jälkeen komposiittisovellukset jaetaan alityyppeihin viisitasoista palvelukeskeistä esimerkkiarkkitehtuuria hyväksikäyttäen.

2.1 Historiaa

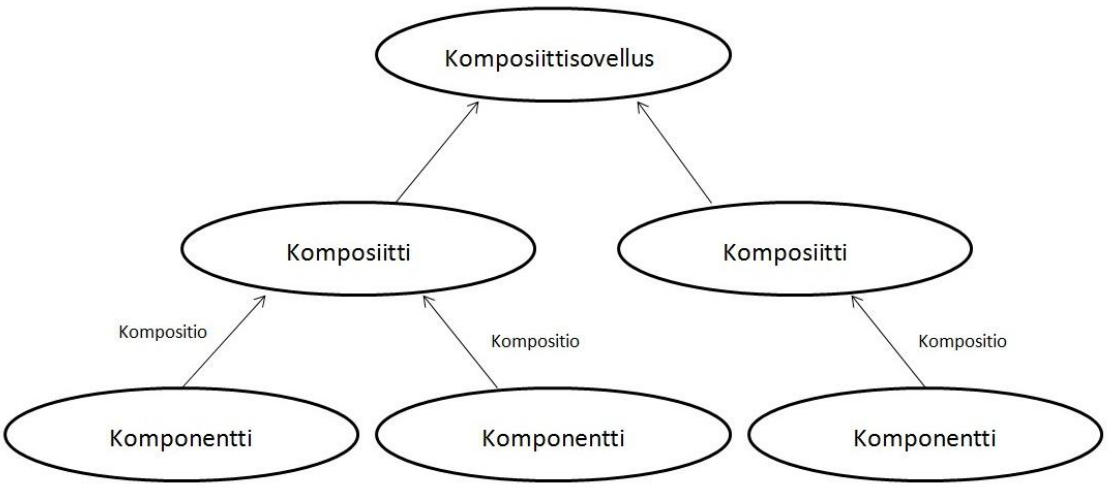
Komposiittisovellusten historian voidaan katsoa ulottuvan komponenttikeskeisen ohjelmistosuunnittelun juurille vuosikymmenien taakse. Muun muassa Unix-järjestelmän putkitustoiminnon toteuttamisesta tunnettu tohtori Malcolm Douglas McIlroy esitti NATO:n vuonna 1968 järjestämässä ohjelmistokonferenssissa ajatuksiaan ohjelmistotuotannon teollistamisesta. Hän ehdotti toteutettavien toimintojen paketoimista uudelleenkäytettäviksi ohjelmistokomponenteiksi. Komposiittisovelluksissa toteutetaankin McIlroyn yli neljäkymmentä vuotta sitten esittämää ja jo muissa ohjelmistotuotannon konteksteissa toteutettua ideaa. (McIlroy, 1968, 138–150)

Komposiittisovellukset voidaan nähdä myös ohjelmistoarkkitehtuurin suunnittelumallien kehittymisen tuloksena. Pitkälle kehittyneet suunnittelumallit ja ohjelmistoarkkitehtuurit, kuten MVC (model-view-controller), tarjoavat erinomaiset lähtökohdat ohjelmistosuunnittelijan ja ohjelmoijan näkökulmasta tietojär-

jestelmien toteuttamiseen. Useimmiten tietojärjestelmä tulee kuitenkin tiettyyn toimintaympäristön määrittämään tarpeeseen, jonka muuttuessa myös järjestelmää tulee muuttaa. Nämä perinteiset sovelluskeskeiset arkkitehtuurit (application-centric architecture) taas eivät tue parhaalla mahdollisella tavalla tietojärjestelmien muuttamista esimerkiksi jatkuvasti muuttuvien kaupallisten tarpeiden mukaan. (Dubray, 2007, 34–45)

2.2 Määritelmä

Komposiittisovelluksen tavoitteet määritellään eri näkökulmista katsottuna useilla eri tavoilla. Valitettavan usein myös aihealueen termien käyttö on sekavaa ja monimuotoista. Yleisesti voidaan sanoa komponentin (component) olevan komposiittia (composite) pienempi osa. Kompositio (composition) nähdään oliomallinnuksesta tutun UML-kielen (Unified Modeling Language) tapaan komponentin ja komposiitin suhdetta kuvaavana käsitteenä. Kuvassa alla (kuvio 1) esitetään tutkielman näkökulma termien välisistä suhteista. Nuolet kuvaavat komposiitin koostamista. (Object Management Group, 2009, 113–114)



KUVIO 1 Komposiitin, komponentin ja komposition keskinäiset suhteet

Seuraavaksi esitellään neljä määritelmää keskeisille termeille ja pyritään tunnistamaan niistä tutkielman kannalta olennaiset osat.

1. Valtion teknillisen tutkimuskeskuksen (VTT) tutkijat Julia Kantorovitch ja Eila Niemelä määrittelevät web-palveluista muodostetun komposiitin seuraavasti:

   Palvelukomposiitti on rypäs alhaalta ylös lähestymistavalla (ks. luku 3.4.2) toisiinsa yhdistettyjä palveluita, jotka yhteen liitettynä tarjoavat toiminnallisuutta, jota yksikään ryppään palvelu ei pysty itsenäisesti tarjoamaan. Komposiitteja voidaan luoda proaktiivisesti suunnitteluvaiheessa (design-time) tai reaktiivisesti ajovaiheessa (run-time). (Kantorovitch & Niemelä, 2009)

2. Stevens Institute of Technology-yliopiston professorit Haim Kilov ja Ira Sack määrittelevät komposition sellaiseksi komposiitin ja komponenttien väliseksi suhteeksi, jossa komposiitin jotkin ominaisuudet määräytyvät siihen liitettyjen komponenttien ja liitostavan mukaan. (Kilov & Sack, 2009, 686–690)

3. Zayedin yliopiston professorin Zakaria Maamarin mukaan sellaista web-palvelua, joka muita web-palveluita hyväksikäyttäen täyttää jonkin sellaisen tarpeen, jota mikään käytetyistä web-palveluista ei pysty yksin täyttämään, voidaan sanoa komposiitiksi. (Maamar, 2009, 1024–1029)

4. Kirjassa Composite Software Construction ohjelmistoarkkitehti Jean-Jacques Dubray puolestaan esittää teknisen näkemyksen komposiittisovelluksiin liittyvistä termeistä. Hän luokittelee komposiittiratkaisun vain yleisellä tasolla:

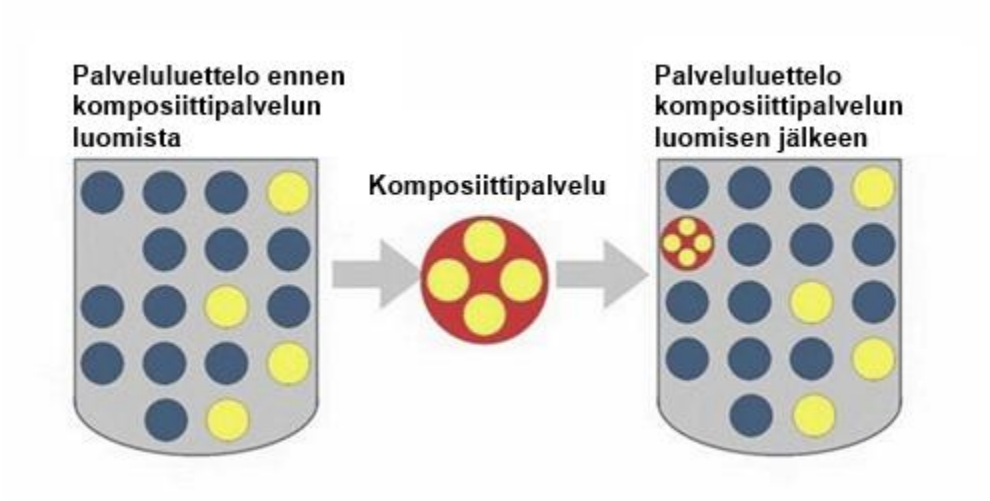
   ”Ratkaisu, joka rakennetaan olemassa olevista resursseista.” (Dubray, 2007, 19)

> Kantrovitchin ja Niemelän tapaan myös Dubray määrittelee komposiitin resurssien käyttöönoton ajankohdalle eri vaihtoehtoja. Suunnitteluvaiheen ja ajonaikaisen vaiheen lisäksi koostamista voi Dubrayn mukaan tapahtua sovelluksen käyttöönottovaiheessa (deployment-time). Komponentti-termiä Dubray käyttää vain suunnitteluvaiheessa käyttöön otetuista resursseista puhuttaessa. Muun muassa web-palvelut eivät täytä tätä komponentin määritelmää, koska ne otetaan käyttöön ajoaikaisesti. Web-palveluita Dubray kutsuukin vain palveluiksi. (Dubray, 2007, 19–23)

Edellä esitellyt määritelmät vahvistavat luvun alussa esitetyn luokittelun, jossa komponentti nähdään näistä komposiittisovelluksen osista pienimpänä. Dubray (2007) käyttää komponentin sijasta palvelu-termiä web-palveluista puhuessaan. Komposiittisovelluksen näkökulmasta määritelmät ovat kuitenkin samassa linjassa muiden määritelmien kanssa. Kolme ensimmäistä määritelmää on valittu IGI Globalin vuoden 2009 informaatioteknologian tietosanakirjaan. (IGI Global, 2009, List of Contributors)

Tutkielman näkökulmasta ei ole oleellista löytää tarkinta mahdollista määritelmää kullekin termille, vaan hahmottaa web-palveluiden rooli komposiittisovellusten näkökulmasta. Web-palveluilta vaadittavien ominaisuuksien voidaan olettaa olevan riippuvaisia yrityksen arkkitehtuurista. Jos arkkitehtuurin palvelut on suunniteltu yksinkertaisiksi, ovat ne komposiittisovellukselle ainoastaan luku- ja kirjoitusoperaatioita tarjoavia tietolähteitä. Toisaalta, jos palvelut ovat monimutkaisia, voivat ne itsessään olla komposiittisovelluksen komponentteja tai kokonainen komposiittisovellus voidaan tarjota yhtenä web-palveluna. Kuvassa alla (kuvio 2) esitetään edellä kuvaillut tilanteet palveluluettelon (ks. kohta 3.5.4) näkökulmasta. Kuviossa vasemmalla esitetyssä tilanteessa luettelo sisältää vain yksinkertaisia palveluita. Kuvion keskellä muodostetaan komposiittipalvelu, jossa käytetään yksinkertaisia web-palveluita. Oikeanpuoleisessa tilanteessa palveluluetteloon on lisätty juuri muodostettu komposiittipalvelu.

Palveluluettelon näkökulmasta komposiittipalvelu on tasavertainen yksinkertaisempien web-palveluiden kanssa. (Davis, 2009, 14–15)



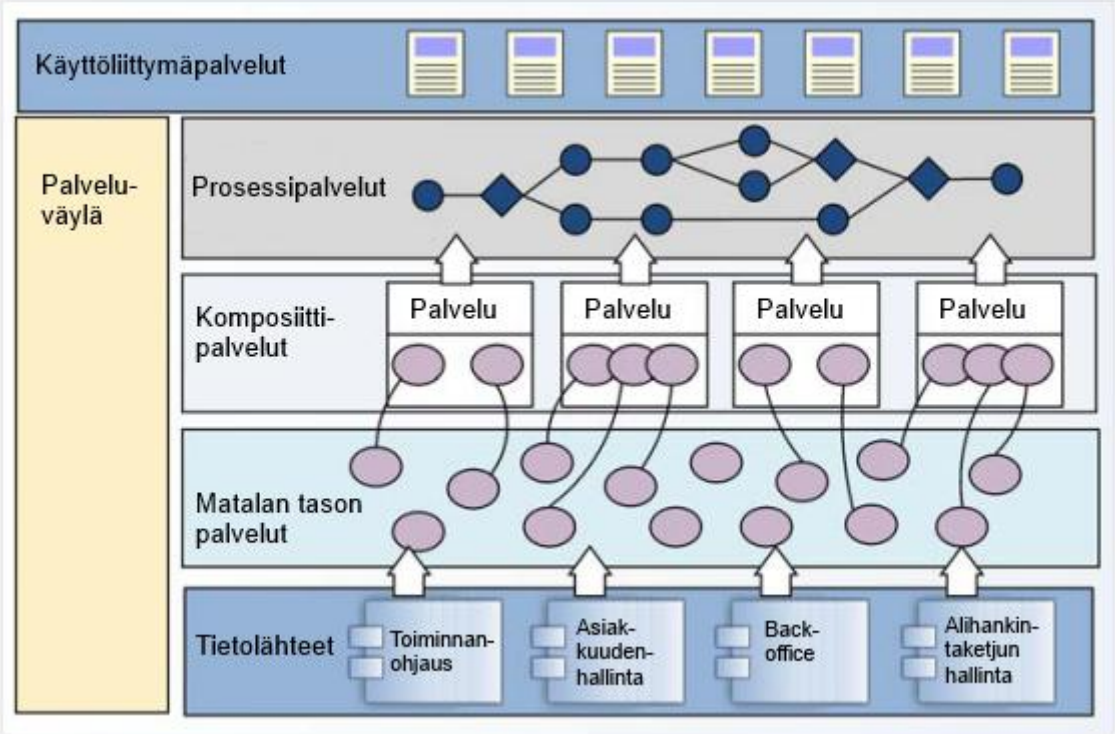
KUVIO 2 Komposiittipalvelu palveluluettelossa (Davis, 2009, sivu 15)

Termien sekalaisen käytön ja ristiriitaisten määrittelyjen vuoksi tutkielmassa käytetään vain termejä komponentti, komposiitti ja komposiittisovellus. Komponentilla tarkoitetaan mitä tahansa komposiitissa käytettävää palvelua. Aiheen rajaus kuitenkin rajoittaa esimerkkisovelluksen komponenttien olevan palvelukeskeisen arkkitehtuurin mukaisia web-palveluita. Komposiitille käytetään Dubrayn (2007) määritelmää, jolloin komposiitin sijainti ympäröivässä arkkitehtuurissa jätetään tapauskohtaisesti määriteltäväksi. Komposiittisovelluksella tarkoitetaan jonkin liiketoiminnallisen tarpeen tyydyttävää komposiittijoukkoa.

2.3 Karkea alajaottelu sovellustyypin perusteella

Kuvassa alla (kuvio 3) esitetään Oraclen tuotepäällikön ja SOA asiantuntijan Jeff Davisin näkemys yleisestä palvelukeskeisestä arkkitehtuurista. Kuvion alimmassa kerroksessa on esitetty arkkitehtuurin tietolähteet, joihin pääsyn tarjoavat toisen kerroksen yksinkertaiset web-palvelut. Matalan tason palveluiden toiminnot voivat olla hyvin suoraviivaisia tietolähdepalveluita käyttäviä toteu-

tuksia, mutta muuntavat tavallisesti tietolähteestä tulevan raakadatan liiketoimintaolioiksi. Arkkitehtuurin kolmanteen kerrokseen on sijoitettu web-palveluita hyväkseen käyttäviä komposiittipalveluita, jotka tarjoavat monipuolisempaa toiminnallisuutta jälleen ylemmän kerroksen liiketoimintaprosesseille. Ylin kerros on rajapinta käyttäjille. (Davis, 2009, 8–9)



KUVIO 3 Viisitasoinen palvelukeskeinen arkkitehtuuri (Davis, 2009)

Komposiittisovelluksen määritelmässä alaluvussa 2.2 todettiin, että komposiitin komponentit voivat itsekin olla komposiitteja. Myös niistä muodostettavan uuden komposiitin palvelut voidaan tarjota eteenpäin web-palveluna, jolloin ylemmällä tasolla toimivan komposiitin näkökulmasta juuri muodostettu komposiitti onkin vain yksi komponentti. Komposiitit voivat siis muodostaa rekursiivisia rakenteita ja kuvassa yllä (kuvio 3) esitettyjen arkkitehtuurin tasojen määrä voi tapauskohtaisesti vaihdella. Viisitasoinen arkkitehtuuri on yleisesti hyväksytty perusmalli ja on käytössä muun muassa IBM:n esimerkkiarkkitehtuureissa. (Davis, 2009, 64–70; Arsanjani, Zhang, Ellis, Allam & Channabasavaiah, 2007, 10–17)

Seuraavaksi selvitetään, mitä komposiittisovellustyyppejä on olemassa ja mihin tarkoitukseen ne ovat suunniteltu. Sovellustyyppejä luokitellaan palvelukeskeisen arkkitehtuurin näkökulmasta arvioimalla, millä arkkitehtuuritasolla tietyntyyppinen sovellus voisi toimia. Kunkin alaluvun lopussa tehdään lyhyt katsaus tekniikoihin, joilla kyseisentyyppisiä komposiittisovelluksia voi toteuttaa.

2.3.1 Mashup

Mashup-tyyppiset verkkosovellukset (esimerkiksi http://woozor.com/ ja http://www.tilannehuone.fi/), joille on ominaista usean eri tietolähteen tietojen yhdistäminen kootusti käyttäjän näkyville, täyttävät alaluvussa 2.2 esitetyn komposiittisovelluksen määritelmän. Ohjelmistoarkkitehti ja Kalifornian yliopiston alaisuudessa toimivan School of Informationin luennoitsija Raymond Yee (2008) esittää kirjassaan mashupien noudattavan yleensä yksinkertaista kolmen askeleen suunnittelumallia (Yee, 2008, 3):

1. Tieto kerätään lähdesivuilta.
2. Kerätty tieto muokataan kohdesivun ymmärtämään muotoon.
3. Muokattu tieto lähetetään kohdesivulle.

Mashup-termiä käytetään siis kuvaamaan monia erilaisia verkossa olevaa tietoa yhdistäviä sovelluksia. Vaikka tarkkaa ja yleisesti hyväksyttyä määritelmää ei ole vakiintunut, voidaan yleisiä ohjelmistosuunnittelussa ja siten myös mashup-sovelluksissa huomioitavia seikkoja tunnistaa. Jo määrittelyvaiheessa on olennaista selvittää, mitä tietoja halutaan yhdistää ja minkä takia yhdistäminen on tarpeen. Teknisestä näkökulmasta on hyvä päättää, missä sovelluksen osassa yhdistäminen tapahtuu ja millä teknologioilla se toteutetaan. Ohjelmiston elinkaaren pituus ja monipuoliset käyttömahdollisuudet voidaan huomioida suunnittelemalla riittävät laajennusmahdollisuudet sovellukseen. (Yee, 2008, 5–13)

Cybersoft Information Technologies yrityksen perustaja Semih Cetin (2007) kollegoineen lähestyvät mashup-tyyppisiä sovelluksia palvelukeskeisen arkkitehtuurin näkökulmasta. Pitkään käytössä olleiden järjestelmien integroiminen osaksi yrityksen palvelukeskeistä arkkitehtuuria on usein haasteellista. Uusia järjestelmiä kehittäessä ympäröivä arkkitehtuuri voidaan ottaa jo suunnitteluvaiheessa huomioon, mutta olemassa olevien sovellusten osalta tilanne on vaikeampi. Ratkaisuksi ehdotetaan sovellusten palveluiden tarjoamista mashup-sovellukseen sopivan rajapinnan kautta, jolloin uuden arkkitehtuurin vaatimat operaatiot suoritettaisiin mashupissa. Tämän kaltaiset yrityksen sisäiset mashup-sovellukset sijoittuisivat viisitasoisessa arkkitehtuurissa (kuvio 3) ylimmälle eli esitystasolle. (Cetin, Altintas, Oguztuzun, Dogru, Tufekci & Suloglu, 2007)

Mashupien toteuttamiseen on tarjolla useita ilmaisia ja maksullisia työkaluja. Isoista vaikuttajista muun muassa Microsoft, IBM, Oracle ja SAP tarjoavat integroituja kehitysympäristöjä mashup-sovellusten suunnitteluun ja toteutukseen. Maksuttomia ja kevyempiä vaihtoehtoja ovat muun muassa Yahoo Pipes sekä pilviteknologiaa toteuttava Google App-Engine, johon on siirretty jo käytöstä poistetun Google Mashup Editorin keskeisimpiä ominaisuuksia. Jokainen tuote tekee tiedon keräämisen ja käsittelyn omalla tavallaan keskinäistä yhteensopivuutta juurikaan korostamatta. Tuottoa tavoittelematon Open Mashup Alliance-järjestö (OMA) kehittää EMML-kieltä (Enterprise Mashup Markup Language), jonka tavoitteena on parantaa mashup-sovellusten uudelleenkäytettävyyttä ja yhteentoimivuutta. EMML saavutti version 1.0, mutta kielen tulevaisuudennäkymiä heikentää kappaleen alussa mainittujen ohjelmistoalan suurimpien tekijöiden puuttuminen OMA-järjestön jäsenistöstä. (Cetin ym., 2007; Galpin ym., 2008; Open Mashup Alliance, 2009)

2.3.2 Prosessikomposiitit

Yrityksen tuotannollisen toiminnan näkökulmasta tietojärjestelmien pääasiallinen tehtävä on liiketoimintaprosessien tukeminen. Prosesseja on muutettava

jatkuvasti liiketoimintaympäristön ja innovaatioiden myötä, jolloin myös tietojärjestelmän muutokset prosessia vastaaviksi ovat välttämättömiä. Perinteiset arkkitehtuurin usealla tasolla toimivat sovellukset vaativat useimmiten muutoksen myötä suunnittelun, toteutuksen, koko sovelluksen testauksen sekä muutetun järjestelmän käyttöönoton tuotantoympäristössä. Prosessikomposiittien käytöllä pyritään minimoimaan muutostarpeen tunnistamisen ja muutetun sovelluksen tuotantokäyttöön ottamisen välistä aikaa automatisoimalla prosesseja, jotka voivat tietojärjestelmän toimintojen lisäksi sisältää myös ihmisen suorittamia toimenpiteitä. (Juric, 2006, 5–8)

Nopeiden prosessin muutosten mahdollistamiseksi prosessikomposiiteissa eriytetään prosessissa käytettävä data ja liiketoiminnan tarpeita vastaavat prosessin toimenpiteet toisistaan. Viisitasoisessa arkkitehtuurissa (kuvio 3) prosessikomposiitit sijoittuvat toiseksi ylimmälle tasolle. Prosessin eriyttäminen arkkitehtuurin alempien kerrosten tietojärjestelmistä on kuvattu yhteisellä rajapinnalla palvelukomposiittien kanssa. Palvelukomposiitit, jotka itsekin yhdistävät useiden alempien kerrosten toimintoja yhteen, ovat prosessin näkökulmasta atomisia operaatioita tarjoavia palveluita. Palvelukomposiittien tarjoamia operaatioita voidaan kutsua prosessin eri vaiheissa, mutta liiketoimintasäännöt ja eteneminen vaiheesta toiseen määritellään prosessikomposiittiin. (Juric, 2006, 16–21)

Esimerkkinä prosessikomposiitista voisi olla rekisteröityminen verkkopalvelun käyttäjäksi. Rekisteröintiprosessin tapahtumat tallennetaan vasta, kun käyttäjä on läpäissyt koko prosessin. Monivaiheisen rekisteröitymisen viimeisenä vaiheena voisi olla esimerkiksi yhteydenottopyyntöjä hallinnoiva palvelukomposiitti (ks. kohta 2.3.3). Muissa vaiheissa käytettäisiin muita komposiitteja sekä mahdollisesti ihmisen suorittamaa toimintoa.

Prosessien automatisointiin on tarjolla paljon etenkin kaupallisia, mutta myös joitain maksuttomia vaihtoehtoja. Prosessien kuvauskielistä suosituimmaksi on valikoitunut lähes kolmensadan yrityksen ja yhteisön yhteenliittymän OASIS:n (Organization for the Advancement of Structured Information Standards) yllä-

pitämä WS-BPEL (Web Services Business Process Execution Language), joka usein lyhennetään BPEL. Kyseistä prosessinkuvauskieltä tukevista kaupallisista vaihtoehdoista suosittuja ovat muun muassa Oracle BPEL Process Manager, IBM WebSphere Process Server ja Microsoft BizTalk Server. Avoimen lähdekoodin vaihtoehtoja tarjoaa muun muassa RedHat-yrityksen alaisuudessa toimiva JBoss Process Server- ja RiftSaw-projekteillaan. (OASIS, 2009b; OASIS, 2009a; Vasiliev, 2007; Brown & Stam 2009)

2.3.3 Palvelukomposiitit

Palvelukeskeisen arkkitehtuurin lähtökohta on yrityksen liiketoiminnan vaatimien palveluiden tunnistaminen ja mallintaminen arkkitehtuuriin. Palvelukomposiitin tunnusmerkkejä on muun muassa kutsujen saapuminen useasta eri lähteestä. Komposiitin suorittama prosessi on liiketoiminnan näkökulmasta yksinkertainen, mutta sen suorittamiseen käytetään alemman kerroksen palveluita. Esimerkkitapaus voisi olla yhteydenottopyynnön lähettäminen asiakkaalta yritykselle. Pyyntö voidaan tehdä yhtä lailla julkisilta verkkosivuilta kuin yrityksen tarjoaman sovelluksen sisältä. Asiakas valitsee listasta yksikön, jota yhteydenottopyyntö koskee. Lista yksiköistä haetaan arkkitehtuurin alemmalta web-palvelulta. Kaikki yritykseen tulleet pyynnöt halutaan kuitenkin lähteestä ja valitusta yksiköstä riippumatta käsitellä samassa paikassa. Yhteydenottopyynnön luomisoperaatio olisi luonnollista sijoittaa viisitasoisessa arkkitehtuurissa (kuvio 3) alhaalta lukien kolmannelle kerrokselle. (Davis, 2009, 61–64)

Alemman kerroksen palveluita yhdistävät palvelukomposiitit voivat olla myös esimerkkiä monimutkaisempia, useita palveluita yhdistäviä komposiittisovelluksia, joiden operaatiot tarjotaan web-palveluna. Palveluiden koostamiseen on kehitetty useita määrittelyjä, mutta suurten kaupallisten toimijoiden pitkään jatkunut keskinäinen kilpailu määrittelyiden paremmuudesta on hidastanut niiden käyttöönottoa. Web-palveluiden koostamista varten kehitettiin 2000-luvun alkupuolella WS-CAF (Web Services Composite Application Framework), jonka ominaisuuksia on sisällytetty saman organisaation myöhemmin

kehittämään WS-TX-määritykseen (Web Services Transaction). (Little, 2003; OASIS, 2005; OASIS, 2009d)

Hieman toisenlaisen lähestymistavan palvelukomposiittien koostamiseen tarjoaa alun perin OSOA-yhteisön (Open Service Oriented Architecture) julkaisema, nykyään OASIS:n valvonnassa kehitettävä SCA (Service Component Architecture). Liiketoiminnan näkökulmasta koko yrityksen ja tarvittavien sidosryhmien tiedon kulku voidaan mallintaa yhdessä SCA-kohdealueessa (domain), jossa määritetään alustariippumattomasti XML-dokumenteilla kohdealueessa tarvittavat komponenttikokoelmat eli komposiitit. SCA-komposiitin komponentti voi olla web-palvelu, joten SCA soveltuu myös palvelukomposiittien muodostamiseen. (OSOA, 2007; OASIS, 2009c; Chappell, 2007, 3–7; Davis, 2009, 64–76)

2.3.4 ESB-palveluväylä

Viisitasoisessa esimerkkiarkkitehtuurissa (kuvio 3) esitetään vasemmassa laidassa kaikille käyttöliittymäkerrosta alemmille tasoille ulottuva Enterprise Service Bus (ESB). ESB juontaa juurensa vuosikymmeniä sitten esiteltyyn EAI-käsitteeseen (Enterprise Application Integration), mutta palvelukeskeisten arkkitehtuurien nauttiessa kasvavaa huomiota ESB sisällytetään usein myös niiden kuvauksiin. ESB tarjoaa muun muassa sovittimia (adapter) eri protokollilla toimiviin, vanhoihin ja uusiin järjestelmiin. Tiedonvälitys hoidetaan yleensä viestikeskeisesti (message-oriented middleware) ja alustariippumattomasti XML-dokumentteja liikutellen. Viestien välityksen lisäksi voidaan tarjota monipuolisia tiedon reititys- ja muunnostoimintoja, valvontaa sekä ajastettuja tehtäviä. (Juric, 2006, 10–11; Davis, 2009, 254–268)

Laajan protokollatuen sekä viestikeskeisen tiedonvälitystavan ansiosta ESB-toteutukset tarjoavat mahdollisuuksia myös komposiittisovelluksen tekoon. Palvelukeskeisen arkkitehtuurin näkökulmasta se ei kuitenkaan ole suositeltavaa. ESB:n avulla koostettaessa ESB:n ja komponentin rajapintaan tulisi proto-

kollaan ja mahdollisesti alustaan sidottua toteutusta, jolloin komponentin uudelleenkäytettävyys ja koostettavuus kärsii. (Davis, 2009, 266–268)

Pitkän historian sekä vapaan määrittelyn vuoksi ESB-toteutuksia löytyy kaikilta ohjelmistoalan suurilta vaikuttajilta. Microsoftin ESB-tuote on BizTalk-palvelimen lisäosa. Javalla toteutettu IBM:n ESB puolestaan toimii WebSphere Application Server-palvelimen päällä. Java-yhteisössä ESB:tä varten on tehty JBI-määritys (Java Business Integration). Maksullisten tuotteiden lisäksi JBI:n toteuttaa myös moni avoimen lähdekoodin projekti kuten OpenESB ja Apache ServiceMix. (Microsoft, 2009; IBM, 2009; Java Community Process, 2009; Open ESB, 2009; The Apache Software Foundation, 2009)

2.4 Yhteenveto

Toisessa pääluvussa selvitettiin, mitä komposiittisovellukset ovat ja luotiin katsaus niiden historiaan. Komposiittisovelluksen olennaisimmalle osalle, komposiitille, esitettiin neljä toisistaan poikkeavaa määritelmää ja selvitettiin niiden perusteella web-palvelun ja komposiitin välistä suhdetta. Web-palvelun todettiin voivan olla yksinkertainen tietolähdekomponentti, monimutkaisempi komposiitti tai jopa kokonainen komposiittisovellus.

Olennaisimpien termien määrittelyn jälkeen komposiittisovellukset luokiteltiin alityyppeihin viisikerroksisen palvelukeskeisen referenssiarkkitehtuurin perusteella. Lopuksi selvitettiin, millaisia komposiitteja tai komposiittisovelluksia tietyltä arkkitehtuurikerrokselta tyypillisesti löytyy.

3 WEB-PALVELUIHIN LIITTYVÄÄ PERUSKÄSITTEISTÖÄ

Yksi komposiittisovellusten tavoitteista on ohjelmistojen yhteentoimivuuden ja uudelleenkäytön parantaminen. Palvelukeskeisessä arkkitehtuurissa näiden toteuttaminen vaatii web-palveluiden tehokasta käyttöä. Arkkitehtuuria suunnitellessa päätetään muun muassa, mitä rajapintoja arkkitehtuurin web-palvelut tukevat. Tämä taas toimii pohjana toteutusmenetelmien ja teknologioiden valinnalle.

Tässä luvussa kerrotaan lyhyesti, mitä web-palvelut ovat ja luodaan katsaus niiden historiaan. Web-palvelun käsite yhdistetään tekniseen toteutukseen jaottelemalla palvelut karkeasti arkkitehtuurin ja toteutustavan perusteella. Aihealueen laajasta termistöstä selvitetään tärkeimmät termit, niiden merkitys ja keskinäiset suhteet.

3.1 Historiaa

Hajautettujen tietojärjestelmien historian voidaan katsoa ulottuvan ainakin 70-luvulle saakka, jolloin IETF (Internet Engineering Task Force) julkaisi ehdotelman standardista tiedon liikuttamiseksi maantieteellisesti hajautuneiden koneiden, sovellusten ja ihmisten välillä (IETF, 1976). Vaikka kolme vuosikymmentä sitten suunniteltujen hajautettujen järjestelmien toteutukset olivat yksinkertaisia ohjelmistojen etäkutsuja (Remote Procedure Call, RPC) yhdestä UNIX-

käyttöjärjestelmästä toiseen, teknisestä näkökulmasta samoihin haasteisiin pyritään vastaamaan myös tämän päivän uusimmissa web-palvelujen toteutuksissa (Rosen, Lublisnky, Smith & Balcer, 2008, 3–10). Web-palveluiden ilmestymiseen johtanutta kehitystä ei voidakaan ohjelmistoalalla pitää kiistatta mullistavana poikkeuksena, vaan pikemminkin luontaisena teknologisena evoluutiona.

Onnistuneita, Tuxedo-ohjelmistokehystä (Transactions for Unix, Extended for Distributed Operations) hyväksi käyttäneitä toteutuksia hajautetuista järjestelmistä nähtiin 80-luvun loppupuolella (Rosen ym., 2008, 3–7). Alustariippumattomuuteen pyrittiin 90-luvulla muun muassa CORBA-standardin (Common Object Request Broker Architecture) mukaisilla toteutuksilla (Rosen ym., 2008, 7–10). Kyseisen vuosikymmenen Microsoft kulki omaa polkuaan tiukasti omaan alustaansa sidotulla DCOM-spesifikaatiolla (Distributed Component Object Model). (Kalin, 2009, luku 7.1)

Vaikka tekniset tavoitteet ovat osittain samoja myös uudemmissa toteutuksissa, englannin kielen sanat ”web service” yhdistettiin teknologiseksi käsitteeksi vuonna 2000 Microsoftin toimesta. Yrityksen perustama työryhmä ehdotti useita standardeja yhdistettynä otettavaksi käyttöön toteutettaessa koneiden välistä, verkon yli tapahtuvaa kommunikaatiota. Ehdotettuja standardeja olivat muun muassa jo aiemmin paljon käytetyt XML (Extensible Markup Language) ja HTTP (Hypertext Transfer Protocol). (Josuttis, 2007, 209–211)

3.2 Määritelmä

Määritellessään termiä web-palvelu Josuttis (2007, 209–211) lainaa alaluvussa 3.1 mainitun alkuperäisen web-palvelutyöryhmän jäsentä Adam Bosworthia, joka määrittelee web-palvelun olevan mikä tahansa ohjelmistojen välisen kommunikaation mahdollistava arkkitehtuuri. Hän kuitenkin tarkentaa tätä määritelmää viittaamalla web-palvelulla ohjelmistojen yhteentoimivuuden mahdollistavaan standardikokoelmaan. Kokoelmassa tulee olla normit alustariippumattomalle formaatille, jolla siirrettävä data esitetään, sekä protokollalle, jolla

ohjelmistojen välinen kommunikaatio tapahtuu. Josuttis esittää myös web-palveluiden viestien olevan itseään kuvailevia ja helppoja liikuttaa eteenpäin tietoverkossa. Alaluvussa 3.1 mainitussa, Microsoftin ensimmäisessä standardikokoelmassa, HTTP olisi kommunikaatioon käytettävä standardi ja XML datan esittämiseen käytettävä standardi. (Josuttis, 2007, 209–211)

World Wide Web Consortium (W3C) puolestaan määrittelee web-palvelun seuraavasti:

> *"Web-palvelu on ohjelmisto, joka on suunniteltu tukemaan koneidenvälistä vuorovaikutusta tietoverkon välityksellä. Sillä on koneen luettavassa muodossa määritelty rajapinta (tarkemmin WSDL). Vuorovaikutus muiden järjestelmien kanssa toteutetaan SOAP-viesteillä, jotka siirretään tyypillisesti sarjallistettuina XML-viesteinä http-protokollaa käyttäen."* (World Wide Web Consortium, 2004)

Ohjelmistoalalle tuttuun termistön sekavaan käyttöön ei voi olla törmäämättä myöskään web-palvelun määrittelyjä vertaillessa. Josuttisin (2007) käyttämän määritelmän mukaan web-palvelun lähettämien viestien täytyy olla itseään kuvailevia, kun taas W3C:n määritelmässä ei vielä vaadita semanttisia viestejä. W3C:n määritelmää ei myöskään ole julkaistu standardina, vaan työryhmän huomautuksena (W3C Working Group Note). (World Wide Web Consortium, 2004)

Semantiikan vaatiminen web-palvelulta erottaa web-palvelut aiemmin käytetyistä teknologioista. Organisaation tiedonsiirron lisäksi niillä voidaan toteuttaa semanttinen tietoverkko, jossa tietyn tyyppisiä palveluita etsivä sovellus voi suorittaa hakuja tarjolla olevien palveluiden metatietoihin. (de Bruijn, Fensel, Kerrigan, Keller, Lausen & Scicluna, 2008, 17–20)

3.3 Karkea alajaottelu arkkitehtuurin perusteella

Web-palveluina tarjottavien toimintojen monimutkaisuus ja laajuus vaihtelevat tarpeen mukaan yksinkertaisesta resurssin tarjoamisesta monimutkaiseen ohjelmistologiikkaan. Palvelut voidaan jakaa toteutusarkkitehtuurin mukaan kahteen SOA-arkkitehtuuria noudattavaan alaryhmään, joita seuraavaksi kuvaillaan. (Richardson & Ruby, 2007, 13–21)

3.3.1 Tilaton REST-arkkitehtuuri

Yksinkertaisia web-palveluja voidaan toteuttaa REST (Representational State Transfer)-arkkitehtuuria noudattaen. REST-palveluiden sisäinen tila on pääteltävissä suoraan kutsuttavasta osoitteesta ja yksi palvelu toteuttaa vain yhden toiminnon. Palvelun kutsu on tyypillisesti deklaratiivinen, sisältäen kutsuttavan metodin ja kaikki metodille välitettävät tiedot ihmisen ymmärtämässä muodossa. (Richardson & Ruby, 2007, 13–21; Fielding, 2000)

HTTP-siirtoprotokollaa käyttävää REST-arkkitehtuuria noudattavaa web-palvelua voitaisiin kutsua esimerkiksi seuraavalla osoitteella:

http://www.web-palveluntarjoaja.org/kuvat/etsi?tagi=koira

Javalla web-sovelluksia tehneet huomaavat, että esimerkin kaltainen toteutus olisi yksinkertaisimmillaan *web-palveluntarjoaja.org*-palvelimella */kuvat/etsi*-polusta vastaava Javan HttpServlet-rajapinnan toteuttava palvelinsovelma, jonka GET-metodille välitetään *tagi*-nimisessä parametrissa arvo *koira*. Voidaanko tällaista yksinkertaista toteutusta pitää edes web-palveluna, riippuu määrittelystä. Esimerkiksi Richardson ja Ruby (2007, 13), jotka käsittelevät kirjassaan REST-arkkitehtuuria noudattavien palveluiden toteuttamista, esittävät kaikkien staattisten web-sivujen olevan REST-arkkitehtuurin mukaisia web-palveluita. Alaluvussa 3.2 mainitun Adam Boswortin määritelmässä vaadittu web-palveluiden koneellinen löydettävyys voidaan toteuttaa REST-palveluille esi-

merkiksi tarjoamalla niitä UDDI-rekisterin (ks. kohta 3.5.4) kautta. (Battle & Benson, 2007, 63–66)

3.3.2 Tilallinen RPC-arkkitehtuuri

Monimutkaista ohjelmistologiikkaa sisältävät web-palvelut perustuvat tavallisesti RPC-arkkitehtuuriin. Esimerkiksi kaikki SOAP-viesteillä (ks. kohta 2.4.2) kommunikoivat web-palvelut ovat RPC-arkkitehtuurin mukaisia. REST-tyyppisistä palveluista poiketen RPC-palveluilla voi olla käyttäjälle näkymätön sisäinen tila, jolloin erilliset kutsut palveluun voivat riippua toisistaan. Myöskään palvelukutsu ei ole REST-palveluiden tapaan itseään kuvaileva, vaan sekä parametrit että tieto palvelulta pyydettävästä toiminnosta, esimerkiksi metodista, lähetetään käyttäjältä piilossa. HTTP-protokollaa käyttävässä palvelussa tämä tarkoittaa tietojen lähettämistä POST-metodilla. (Richardson & Ruby, 2007, 13–21)

Edellisessä kohdassa esitetty *koira*-tagattujen valokuvien etsiminen voisi RPC-arkkitehtuurin mukaista palvelua kutsuttaessa näyttää tältä:

http://www.web-palveluntarjoaja.org/SOAPViestinKasittelija

Kutsussa käy ilmi vain, miltä palvelimelta palvelu löytyy, mutta kaikki muu piilotetaan HTTP-viestin POST-parametreihin. Kuvassa alla (kuvio 4) on esimerkki yksinkertaisesta SOAP-viestistä, joka voitaisiin lähettää RPC-arkkitehtuurin mukaiseen web-palveluun. Viestin sisältö on vastaavanlainen kutsu kuviteltuun kuvahakupalveluun, joka tehtiin myös REST-arkkitehtuuria käsittelevässä kohdassa (ks. kohta 3.3.1 Tilaton REST). SOAP-viestiformaatti esitellään tarkemmin kohdassa 3.5.2.

```
<?xml version="1.0"?>
<soap:Envelope xmlns:soap="http://www.w3.org/2001/12/soap-envelope"
    soap:encodingStyle="http://www.w3.org/2001/12/soap-encoding">

    <soap:Body xmlns:p="http://www.web-palveluntarjoaja.org/haku">
        <p:SuoritaOperaatio>
            <p:Kategoria>kuvat</p:Kategoria>
            <p:Metodi>etsi</p:Metodi>
            <p:Tagi>koira</p:Tagi>
        </p:SuoritaOperaatio>
    </soap:Body>

</soap:Envelope>
```

KUVIO 4 Esimerkki yksinkertaisesta SOAP-viestistä

3.4 Karkea alajaottelu toteutustavan perusteella

Web-palveluiden käyttökohteet vaihtelevat aina laajoista palvelukeskeisen arkkitehtuurin mukaisista toteutuksista yksittäisen olemassa olevan sovelluksen logiikan tarjoamiseen web-palveluna. Vaikka palvelua käyttävän osapuolen näkökulmasta sen sisäinen toteutustapa on merkityksetön, voidaan sillä ratkaisevasti vaikuttaa käsiteltävän tapauksen ratkaisuun.

3.4.1 Dokumenttikeskeinen lähestymistapa

Kehitettäessä web-palveluita dokumenttikeskeisen eli ylhäältä alas lähestymistavan mukaisesti on suunnittelun painopiste normaalisti palvelua kuvaavalla WSDL-dokumentilla ja sitä vastaavalla XML-skeemalla. WSDL-dokumentissa kuvataan tekniseen toteutukseen tarvittavat tiedot, kuten tuetut viestityypit ja saatavilla olevat operaatiot. Dokumentti on riippumaton ohjelmointikielestä ja palvelurungon generointi toteutuskielelle jätetään ohjelmistokehyksen tehtäväksi. (Jayasinghe, 2008, 137–139)

Dokumenttikeskeinen lähestyminen mahdollistaa web-palveluiden suunnittelun ilman minkään ohjelmointikielen tuntemusta ja auttaa myös tiettyjen teknisten haasteiden ratkaisemisessa. Esimerkiksi Java-kielelle ominaisten tietotyyppien sekä rekursiivisten oliorakenteiden esitteleminen rakenteisessa

WSDL-dokumentissa on varsin haastavaa, mutta XML-rakennetta vastaavan oliomallin generointi ei vaadi kompromisseja. (Poutsma, Evans & Rabbo, 2007, 4–6)

Dokumenttikeskeisen lähestymistavan eduista huolimatta tietyt käyttötapaukset, kuten olemassa olevan järjestelmän operaatioiden tarjoaminen web-palveluina, soveltuvat ratkaistaviksi koodikeskeisellä lähestymistavalla. (Jayasinghe, 2008, 137)

3.4.2 Koodikeskeinen lähestymistapa

Koodikeskeisessä eli alhaalta ylös lähestymistavassa tehdään ensin palvelun rungon toteutus kohdekielellä. Java-kieltä käyttäessä kirjoitetaan ensin palvelulogiikan toteuttavat luokat ja luokkiin operaatiot toteuttavat metodit. Ohjelmistokehykselle voidaan kertoa annotaatioilla tai erillisellä XML-dokumentilla, minkä operaation metodi toteuttaa. Merkityn ohjelmakoodin pohjalta web-palvelun toteuttava ohjelmistokehys luo palvelun rajapinnan määrittävän WSDL-dokumentin. (Jayasinghe, 2008, 137–139)

Dokumenttikeskeiseen lähestymistapaan verrattuna koodikeskeisen tavan selkeisiin etuihin lukeutuu mahdollisuus toteuttaa web-palveluita ilman syvällisempää WSDL-tuntemusta. Myös toistuvien muutosten toteuttaminen, esimerkiksi protoiluvaiheessa, on koodikeskeisellä lähestymistavalla helpompaa. (Jayasinghe, 2008, 137–138)

3.5 Keskeistä termistöä

Koska kyseessä on uusi aihealue ja yleisesti hyväksytyt perusteokset ovat vasta vakiintumassa myös termien määritelmät vaihtelevat lähteestä riippuen. Seuraava taulukko (taulukko 1) esittää tärkeimpiä web-palveluiden yhteydessä käytettäviä termejä ja niiden rooleja teknisestä näkökulmasta. Kutakin taulukossa esiteltävää termiä käsitellään myöhemmin omassa kohdassaan.

TAULUKKO 1 Web-palveluiden keskeisiä termejä ja rooleja

Termi	Rooli
JSON – XML:ää tiiviimpi tekstimuotoisen datan merkintätapa.	**Tilaa säästävä viestiformaatti, jonka tuki on toteutettu useille ohjelmointikielille.**
SOAP – XML-muotoinen viestiformaatti rakenteisen tiedon välittämiseen hajautetussa ympäristössä.	**Välittää dataa operaatiolta ja operaatiolle WSDL:ssä määriteltyjen porttien kautta.**
WSDL - Määrittää palvelun käyttämät portit, tarjoamat operaatiot, tuetut viestinvälityskanavat ja viestiformaatit.	**Palvelun tarkka kuvaus -** Mitä operaatiota ja mistä portista tarjotaan? Millaisia viestejä voidaan välittää? Mitkä ovat operaatiolle mahdollisesti vietävien parametrien tietotyypit?
UDDI - Alustariippumaton tapa löytää ja kuvailla web-palveluita ja niiden tarjoajia.	**Palveluiden ja niiden korkean tason kuvausten löytäminen** - Mitä palveluita on saatavilla?
SOA - Lähestymistapa organisaation rakenteen palvelukeskeiseen mallintamiseen.	**Arkkitehtuuri, joka voidaan toteuttaa mm. web-palveluilla.**

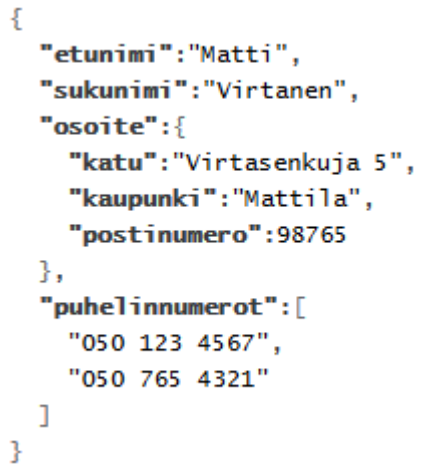
3.5.1 JSON-viestiformaatti

JSON eli JavaScript Object Notation on JavaScript-kielessä käytetty tapa olioiden sarjallistamiseen. Notaatio on kuvattu ECMAScript-ohjelmointikielen määrittelyissä, joihin myös kaikki tunnetut JavaScript-kielen toteutukset perustuvat. XML:ää tiiviimpänä, mutta helposti ihmisen sekä koneen luettavissa olevana merkintätapana, JSON on otettu laajasti käyttöön myös tiedonsiirrossa verkon yli. (IETF, 2006; Ecma International, 2011)

Web-palveluiden kannalta olennaisinta on ohjelmistokehysten laaja tuki JSON–muotoiseen tiedonvälitykseen. JSON on tekstimuotoista dataa, jossa aaltosulkeilla ilmaistaan oliota, kaksoispisteellä olion parametreja ja hakasulkeilla taulukkotietorakennetta. (Pehlivanian & Nguyen, 2013)

Kuvassa alla (kuvio 5) on henkilötietoja sisältävä esimerkki JSON-muotoisesta oliosta. Oliolla on merkkijonoattribuutit etunimi ja sukunimi, olioattribuutti

osoite sekä taulukkoattribuutti puhelinnumerot. Kuvan esimerkin voisi asettaa suoraan JavaScript-kielen muuttujaan, jonka jälkeen kyseinen muuttuja sisältäisi viittauksen kyseiseen olioon. Kuviossa esitetyn JSON-vasteen voisi saada esimerkiksi henkilötietoja hakevasta REST-palvelusta (ks. kohta 3.3.1 Tilaton REST).



```
{
  "etunimi":"Matti",
  "sukunimi":"Virtanen",
  "osoite":{
    "katu":"Virtasenkuja 5",
    "kaupunki":"Mattila",
    "postinumero":98765
  },
  "puhelinnumerot":[
    "050 123 4567",
    "050 765 4321"
  ]
}
```


KUVIO 5 JSON-muodossa esitetty olio

3.5.2 SOAP-viestiformaatti

Web-palveluiden näkökulmasta SOAP on standardoitu tapa paketoida palveluiden välittämiä viestejä. Viestin kaikki data on XML-muotoista. Teknisesti SOAP-spesifikaatio koostuu yhteisesti sovituista säännöistä, joilla alusta- ja kieliriippuvaiset tietotyypit sekä ohjelmistologiikan tila, esimerkiksi virhetilanteen sattuessa, esitetään alustariippumattomassa XML-tiedostossa. (Tidwell, Snell & Kulchenko, 2002, 15–24)

Ohjelmistotalo MindStream Softwaren pääarkkitehti ja johtaja Robert Englander (2002, 47–52) muistuttaa kirjassaan, ettei spesifikaatio sido viestien käyttömahdollisuuksia mihinkään tiettyyn teknologiaan eikä edes web-palveluihin. Standardin mukaisia viestejä käytetään muun muassa dokumenttien liikuttamiseen EDI-järjestelemissä (Electronic Document Interchange). Tässä tutkielmassa

SOAP-viestit ovat kuitenkin RPC-arkkitehtuurin mukaisia metodikutsuja, jolloin viestin sisältämässä XML-tiedostossa välitetään myös metodien parametreja ja palautusarvoja. Viestejä voidaan kuljettaa esimerkiksi HTTP-protokollalla tavallisten nettisivujen tapaan, mutta vastaanottajana on selaimen asemesta SOAP-viestin lukija (Monson-Haefel, 2003, Luku 4).

Alun perin SOAP oli lyhenne sanoista Simple Object Access Protocol. SOAP ei kuitenkaan ole kovin yksinkertainen eikä sillä varsinaisesti ole olioiden kanssa mitään tekemistä. Määritelmän versiosta 1.2 alkaen SOAP onkin ollut itsenäinen termi. (Josuttis, 2007, 217–218)

SOAP:n kaltaisia, mutta huomattavasti yksinkertaisempia ja helppokäyttöisempiä viestiformaatteja ovat esimerkiksi XML-RPC ja JSON-RPC (JavaScript Object Notation RPC). Kaikki ovat RPC-arkkitehtuuriin soveltuvia viestiformaatteja, joista sopivin tulee valita tapauskohtaisesti.

3.5.3 WSDL- ja WADL-kuvauskielet

WSDL (Web Service Description Language) on alun perin Microsoftin, IBM:n ja Ariban ehdottama standardi web-palveluiden liittymien määrittämiseen. Versio 2.0 hyväksyttiin W3C:n standardiksi vuonna 2007 (Taylor & Harrison, 2009, 275). Selkeyden vuoksi mainittakoon, että versio 1.2 uudelleennimettiin suurten muutosten takia versioksi 2.0. Kyseessä on kuitenkin sama määrittely. (World Wide Web Consortium, 2003; World Wide Web Consortium, 2007)

WSDL-dokumentin voidaan katsoa koostuvan kahdesta osakokonaisuudesta: palvelun kuvauksesta sekä toteutuksen määrittelystä. Näistä ensimmäistä tarvitaan, kun asiakas (esimerkiksi toinen web-palvelu) on löytänyt palvelun ja haluaa tietää, millä tekniikoilla löydetty palvelu on valmis kommunikoimaan. Jälkimmäisessä, toteutuksen määrittelyssä, puolestaan kerrotaan palvelua tarjoavalle ohjelmistokehykselle, mihin ja miten tähän palveluun osoitetut viestit tulee ohjata. (Englander, 2002, 170–179; Taylor & Harrison, 2009, 275–281)

Kehittäjän näkökulmasta web-palvelujen määritteleminen WSDL-dokumentissa parantaa koneiden keskinäisen tiedonvaihdon lisäksi myös palveluiden virheensietokykyä. WSDL-kuvauskieli on vahvasti tyypitetty ja palvelun käyttöön ottava asiakas saa ilmoituksen mahdollisesta virhetilanteesta jo käyttäjäksi rekisteröityessään. (Pautasso, Zimmermann & Leymann, 2008)

Versioon 2.0 saakka WSDL-dokumenteilla ei voitu määritellä REST-arkkitehtuurin mukaisia web-palveluita. Puutteen korjatakseen Sun Microsystems julkaisi vuonna 2006 WADL-spesifikaation (Web Application Description Language), jolla REST-palveluiden liittymät voidaan määritellä. Seuraavana vuonna julkaistuun WSDL 2.0-spesifikaatioon lisättiin tuki REST-palveluiden määrittelylle. (Hadley, 2006; Pautasso ym., 2008)

3.5.4 UDDI-palvelurekisteri

UDDI (Universal Description, Discovery and Integration) on rekisteri, jossa palveluiden tarjoajat voivat julkaista tietoja itsestään ja tarjoamistaan palveluista. UDDI on itsessään web-palvelu, joka mahdollistaa rekisterissä esiteltyjen palvelujen koneellisen löydettävyyden, tarjoten muun muassa hakutoimintoja palveluiden etsimiseen metatietojen perusteella. Rekisterissä tarjottava palvelu esitellään UDDI-palvelimelle määrityksen mukaisella XML-dokumentilla. Dokumentin määrittely ei rajoita rekisterin käyttöä vain web-palveluiden esittelyyn, mutta tässä tutkielmassa käsitellään UDDI:a vain web-palveluiden rekisterinä. (Cerami, 2002, 135–144)

Kuten edellisessä kohdassa (3.5.3) todettiin, voidaan web-palveluiden tekniset metatiedot esittää WSDL-dokumentissa. Palvelujen etsijäkonetta kiinnostaa kuitenkin teknisten tietojen asemesta etsittävän palvelun semantiikka. Tällaisen palvelujen luokittelun UDDI-rekisteri pyrkii toteuttamaan rekisteröidyiltä palveluilta keräämillään metatiedoilla. Rekisterin tarkoitus onkin auttaa etsijäkonetta löytämään oikeanlainen palvelu ja antaa etsijälle tiedot, mistä löydetyn palvelun tekniset metatiedot, esimerkiksi WSDL-dokumentti, löytyvät. Tekni-

sen (WSDL) ja semanttisen (UDDI) metatiedon eriyttäminen mahdollistaa myös palvelun teknisten rajapintojen muuttamisen ilman palvelun uudelleenrekisteröintiä. (McGovern, Tyagi, Stevens & Matthew, 2003b, luku 6)

Microsoftin, IBM:n ja Ariban yhteistyössä kehittämä UDDI-spesifikaatio julkaistiin vuonna 2000 (Cerami, 2002, 135–136). Monesta muusta web-palveluteknologiasta poiketen UDDI ei ole W3C:n standardi, vaan spesifikaatiosta vastaa nykyään kohdassa 2.3.2 mainittu OASIS.

3.6 Yhteenveto

Kolmannessa pääluvussa selvitettiin, mitä web-palvelut ovat ja luotiin katsaus niiden historiaan. Web-palvelulle esitettiin kaksi toisistaan poikkeavaa määritelmää ja todettiin, etteivät termistö ja määritelmät ole aihepiirin nuoruuden takia täysin vakiintuneita. Web-palvelut jaoteltiin arkkitehtuurin perusteella REST- tai RPC-tyyppisiin palveluihin ja toteutustavan perusteella dokumenttikeskeisiin tai koodikeskeisiin palveluihin.

Arkkitehtuurityyppi- ja toteutustapajaottelun lisäksi web-palveluiden toteuttamista lähestyttiin selvittämällä teknisen toteutuksen kannalta olennaisimmat termit, niiden merkitykset ja keskinäiset suhteet. Tärkeimmiksi termeiksi tunnistettiin suosituimmat viestinvälitykseen ja palveluiden löydettävyyteen liittyvät teknologiat ja protokollat.

4 PALVELUKESKEISEN ARKKITEHTUURIN PERUSPERIAATTEET

Ensimmäisessä luvussa luokiteltiin komposiittisovellustyyppejä ja sijoitettiin ne palvelukeskeisen arkkitehtuurin tasoille. Toisessa luvussa paneuduttiin syvemmin komposiittisovelluksen ulkoiseen rajapintaan eli web-palveluihin. Tässä luvussa selvitetään, millaisia yleisesti hyväksyttyjä periaatteita palvelukeskeisessä arkkitehtuurissa toimivien sovellusten tulisi noudattaa.

Tarkempaan käsittelyyn valikoitiin kolmen tunnustusta saaneen asiantuntijan Brownin ym. (2002), Erlin (2007) ja Tilkovin (2007) perusperiaatteiden määrittely. Luvun lopussa selvitetään ristiintaulukoimalla, mitkä eri määritelmien perusperiaatteista vastaavat toisiaan.

4.1 Palvelukeskeisen arkkitehtuurin perusperiaatteet Brownin mukaan

Akateemista ja kaupallista tunnustusta saanut tohtori Alan W. Brown (2002) kollegoineen määrittelee lyhyesti ja ytimekkäästi palvelukeskeisen arkkitehtuurin palvelulle neljä perusperiaatetta:

> *"Palvelu on karkeajakoinen ja löydettävissä oleva ohjelmistokomponentti, joka kommunikoi viestikeskeisesti löyhän sidoksen periaatetta noudattavan rajapinnan kautta." (Brown ym., 2002)*

4.2 Palvelukeskeisen arkkitehtuurin perusperiaatteet Erlin mukaan

Useiden kaupallisten tahojen, muun muassa Microsoftin ja IBM:n SOA-asiantuntijaksi tunnustama ja yli 100 000 SOA-aiheista kirjaa myynyt Thomas Erl (2007) puolestaan esittää kahdeksan perusperiaatetta, joita palvelukeskeisessä arkkitehtuurissa tulee noudattaa.

1. *Sopimukset (Service contracts)* ovat standardeja tapoja, joilla palvelut ilmoittavat tarjoamansa operaatiot ja parametrit, mitä kullekin operaatiolle tulee kutsuttaessa toimittaa. Teknisten määrittelyjen, kuten WSDL-tiedostojen (kohta 3.5.3) lisäksi sopimukset voivat kattaa myös palveluun liittyviä ei-toiminnallisia määritteitä. (Erl, 2007, 125–132)
2. *Löyhä sidos (Service coupling)*-periaatteen mukaan keskenään kommunikoivien palveluiden ei tule olla riippuvaisia toistensa toteutustavoista. Löyhän sidoksen toteutumista voidaan tarkastella useista eri näkökulmista, eikä kaikkien sidosten löyhä toteuttaminen ole yleensä järkevää. Erlin mukaan sidoksia muodostetaan ainakin seuraavien näkökulmien väleille. Kunkin välin yhteydessä mainitaan esimerkki, millaisia käytännön asioita tulee huomioida toteutuksessa. (Erl, 2007, 163–181):
   - Logiikka ja sopimus: Koodikeskeinen lähestymistapa (kohta 3.4.2) auttaa tämän toteuttamisessa.
   - Sopimus ja logiikka: Dokumenttikeskeinen lähestymistapa (kohta 3.4.1) auttaa tämän toteuttamisessa.
   - Sopimus ja teknologia: Muille tarjottava rajapinta ei saa vaatia käyttämään esimerkiksi Java- tai .NET-teknologiaa.
   - Sopimus ja implementaatio: Rajapintaa ei tule määritellä palvelun taustalla olevan tietokantarakenteen mukaiseksi.
   - Sopimus ja toiminnallisuus: Palvelu tulee toteuttaa yleiskäyttöiseksi, eikä ainoastaan yhteen tiettyyn tarkoitukseen.
3. *Abstraktio (Service abstraction)*-periaatteen mukaan käyttäjille kerrotun tiedon määrä on käänteisesti verrannollinen abstraktiotasoon. Alhainen abstraktiotaso puolestaan estää löyhien sidosten toteuttamisen. Suunnit-

telun tueksi Erl luokittelee abstraktion neljään eri tyyppiin. Jokaisen abstraktiotyypin kohdalla tulisi miettiä, mitä metatietoja palvelusta tarvitaan muiden käyttöön, koska jaetun tiedon määrä laskee abstraktiotasoa (Erl, 2007, 211–235):

- Teknologinen abstraktio
- Funktionaalinen abstraktio
- Sisäisen logiikan abstraktio
- Palvelun laadun abstraktio.

4. *Uudelleenkäytettävyys (Service reusability)*-periaate korostaa palveluiden yleiskäyttöisyyttä. Erl luokittelee uudelleenkäytettävyyden kolmeen eri toteutustasoon (Erl, 2007, 253–269):
   - Taktinen (tactical): Huomioidaan uudelleenkäytettävyys ja mahdollistetaan tulevat laajennukset, mutta toteutetaan ensimmäisessä vaiheessa vain vaaditut toiminnallisuudet.
   - Kohdennettu (targeted): Toteutetaan välittömien vaatimusten lisäksi ominaisuuksia, joita arkkitehtuurissa tullaan lähitulevaisuudessa tarvitsemaan.
   - Täydellinen (complete): Toteutetaan kaikki suunnitellut toiminnallisuudet valmiiksi, vaikkei välttämättä tiedetä, milloin niitä tarvitaan. Täydellisen uudelleenkäytön toteuttaminen on järkevää vain, jos arkkitehtuuria suunniteltaessa on tehty syvällinen palvelukeskeinen analyysi, jonka perusteella tiedetään arkkitehtuurin pitkän tähtäimen vaatimukset.
5. *Autonomia (Service autonomy)*-periaatteella pyritään parantamaan palvelun luotettavuutta ja monikäyttöisyyttä. Erl jakaa autonomian kahteen eri tyyppiin (Erl, 2007, 293–310):
   - Ajonaikainen autonomia: Palvelu ei ole ajonaikana autonominen, jos se käyttää samoja tietolähteitä tai osatoiminnallisuuksia muiden palveluiden kanssa. Tietolähde voi olla esimerkiksi suora yhteys tietokantaan. Ajonaikainen autonomia kuitenkin toteutuu, jos yhteisesti käytössä olevat resurssit tarjotaan palveluina.

- Suunnitteluaikainen autonomia: Suunnitteluaikainen autonomia keskittyy palvelun riippuvuuteen muista arkkitehtuurin rajapinnoista. Korkea suunnitteluaikainen autonomia mahdollistaa yleensä korkean ajonaikaisen autonomian. Korkeampi autonomian taso puolestaan mahdollistaa palvelun mukautumisen esimerkiksi kasvaneisiin suorituskykyvaatimuksiin ilman arkkitehtuurimuutoksia.

6. *Tilattomuus (Service statelessness)*-periaatteen mukaan palvelulla ei ole lainkaan sisäisiä tiloja, vaan jokainen kutsu palveluun käsitellään saman prosessin mukaan. Tilattomuus auttaa pitämään palvelun muistinkäytön kohtuullisena ja vähentämään verkkoliikennettä, koska palvelun kutsukohtaista tilaa ei varastoida palvelimen muistiin tai verkon yli kutsuttavaan tilapalvelimeen. Palvelun tilattomuus voidaan toteuttaa tekemällä liiketoimintaprosessien näkökulmasta yksinkertaisia palveluita (ks. kohta 3.3.1 Tilaton REST) tai kuljettamalla tilatieto kommunikointiin käytettävien viestien mukana (ks. kohta 3.3.2 Tilallinen RPC). Monimutkainen tilatiedon jäsentäminen, esimerkiksi XML-viestin sisällöstä, voi kuitenkin aiheuttaa uusia suorituskykyhaasteita palvelulle. (Erl, 2007, 325–350)
7. *Löydettävyys (Service discoverability)* on yksi laajimmista ja monipuolisimmista periaatteista. Erlin mukaan palvelukeskeisessä ympäristössä löydettävyyteen liittyy keskeisesti myös palvelun tarjoamien toiminnallisuuksien ymmärrettävyys. Ennen uuden palvelun toteuttamista arkkitehtuurista tulisi etsiä jo olemassa olevaa saman toiminnallisuuden tai osan siitä toteuttavaa palvelua. Tämän mahdollistamiseksi palveluiden metatiedot tulee säilyttää keskitetysti metatietorekisterissä, joka toimii arkkitehtuurin palveluluettelona. Tiettyä toiminnallisuutta etsittäessä tarvitaan fyysisen löydettävyyden, kuten verkko-osoitteiden ja porttien lisäksi tieto kunkin palvelun tarjoamista toiminnallisuuksista sekä niiden rajoitteista. Metatiedon keräämistä, luokittelua ja hakua varten kehitettiin muun muassa UDDI-rekisteri, jota käsiteltiin tarkemmin kohdassa 3.5.4. (Erl, 2007, 361–375)

8. *Koostettavuus (Service composability)*-periaatteella tutkitaan, onko palveluilla valmius toimia komposiitin komponenttina tai itse muita palveluita koostavana komposiittina. Koostettavuuden näkökulma on muita palveluita korkeammalla tasolla ja täydellisesti toteutuakseen se vaatii kaikkien muiden periaatteiden osittaista täyttämistä. Koostettavuus vaatii muun muassa samoja ominaisuuksia kuin uudelleenkäytettävyys, mutta koostettavuus-periaatteen mukaan palveluita luodessa tulee huomioida palvelun tarpeellisuus arkkitehtuurin näkökulmasta sekä soveltuvuus arkkitehtuurin palveluluetteloon (ks. kohta 3.5.4). (Erl, 2007, 387–411)

4.3 Palvelukeskeisen arkkitehtuurin perusperiaatteet Tilkovin mukaan

Saksalaisen InnoQ-konsulttiyrityksen perustaja ja InfoQ:n kirjoittaja Stefan Tilkov (2007) listaa artikkelissaan peräti kymmenen palvelukeskeisen arkkitehtuurin perusperiaatetta.

1. *Selkeät rajat (Explicit boundaries)*-periaate toteutuu, kun palvelua kutsuttaessa sille välitetään kaikki tarvittavat tiedot. Palvelua kutsutaan ainoastaan julkisen rajapinnan kautta, jolloin palvelulla ja kutsujalla ei ole jaettua tietoa toisistaan tai ympäristöstä. (Tilkov, 2007)
2. *Yhteiset määrittelyt, palvelukohtaiset toteutukset (Shared Contract and Schema, not Class)*-periaatteen mukaan palvelun kutsujalla on riittävästi tietoja myös palvelun toteuttamiseen. Tällä varmistetaan alustariippumattomuus, mutta väistämättä myös rajoitetaan toteutusteknologioita. Esimerkkisovelluksen osalta (luvut 5 ja 6) alustariippumattomuus on jo varmistettu valitsemalla web-palvelut rajapintojen toteutukseen. (Tilkov, 2007)
3. *Menettelytapakeskeisyydellä (Policy-driven)* tarkoitetaan palvelun kutsujan ja tarjoajan toiminnallisten ja ei-toiminnallisten määrittelyjen yhteensopivuutta. Osa periaatteen vaatimuksista täyttyy valitsemalla web-palvelut arkkitehtuurin toteutukseen, mutta avoimeksi jäävät määrittelyt

tulee sopia yhteisillä menettelytavoilla, joita palveluille voidaan määrätä. (Tilkov, 2007)

4. *Autonomia (Autonomy)*-periaate liittyy ensimmäiseen periaatteeseen. Autonomian mukaan palvelu kommunikoi ympäristönsä kanssa vain ulkoisen rajapintansa kautta. Vaatimuksen täyttyessä palvelun ulkoinen rajapinta säilyy muuttumattomana, vaikka sisäinen toteutus vaihdettaisiin. (Tilkov, 2007)
5. *Viestiformaattien määrittämisen periaate (Wire formats, not Programming Language APIs)* liittyy kahteen ensimmäiseen periaatteeseen, mutta tarjoaa hieman erilaisen näkökulman. Periaatteen mukaan palvelua tulee voida kutsua mistä tahansa järjestelmästä, joka tukee rajapinnassa määritettyä viestiformaattia ja palvelulle määrättyjä menettelytapoja. (Tilkov, 2007)
6. *Dokumenttikeskeisyydellä (Document-orientated)* korostetaan järkevän viestiformaatin valintaa. Jos rajapinnassa määritellään valmiiksi viestin rakenne, voidaan ei-semanttisia viestejä käyttämällä säästyä XML-pohjaisia RPC-palveluita vaivaavalta tiedonsiirtokuormitukselta. Semanttisia viestejä käyttämällä voidaan puolestaan rajapinta jättää RPC-palveluita avoimemmaksi. Ideaalitilanteessa palvelut voisivat lähettää ja vastaanottaa kohdealueessa käytettyjä paperidokumentteja vastaavia viestejä. Periaatteen mukaan myös tiedon päällekkäisyys palveluiden välillä sallitaan, jos sillä voidaan edesauttaa seuraavan, löyhän sidoksen periaatteen toteutumista. (Tilkov, 2007)
7. *Löyhä sidos (Loose coupling)*-periaatteen päämäärä on sama kuin Erlin (2007) kuvailema. Tilkov esittää kuitenkin löyhään sidokseen erilaisia näkökulmia. Hänen mukaansa palveluiden välinen sidos voi olla löyhä ajan, sijainnin, tyyppien, versioiden, kardinaalisuuden, löydettävyyden ja rajapintojen suhteen. (Tilkov, 2007)
8. *Standardienmukaisuus (Standards-compliant)*-periaatteen mukaan arkkitehtuurissa tulisi käyttää avoimia ja yleisesti hyväksyttyjä standardeja aina, kun mahdollista. (Tilkov, 2007)

9. *Toimittajariippumattomuus (Vendor independent)*-periaatteella pyritään pitämään arkkitehtuurin suunnittelu mahdollisimman pitkään riippumattomana minkään kaupallisen toimijan tarjoamasta teknologiasta. Lopulta voidaan päätyä joidenkin tuotteiden valintaan, mutta tällä ei pitäisi olla vaikutusta arkkitehtuuriin. (Tilkov, 2007)
10. *Metatietokeskeisyys (Metadata-driven)*-periaatteen mukaan kaikki arkkitehtuurin metatieto tulee olla saatavilla suunnitteluvaiheessa (design-time) ja ajonaikana (run-time). Kaikkialla arkkitehtuurissa tarvittaviin metatietoihin sisältyy muun muassa palveluiden rajapintojen kuvauksia ja rooleja organisaation näkökulmasta, liikuteltavien dokumenttien tyyppejä ja rakenteita sekä tietoja palveluiden keskinäisistä riippuvuuksista. (Tilkov, 2007)

4.4 Yhteenveto ja perusperiaatteiden ristiintaulukointi

Tässä luvussa selvitettiin palvelukeskeisen arkkitehtuurin perusperiaatteet kolmen eri määritelmän mukaan. Alla olevassa taulukossa (taulukko 2) tehdään yhteenveto eri määritelmien perusperiaatteiden keskinäisistä vastaavuuksista.

Periaatteiden ristiintaulukoinnilla esitetään, mitkä Tilkovin periaatteet liittyvät mihinkin Erlin periaatteeseen. Erlin periaatteet tarkentavat Brownin ym. neljää periaatetta, joten erillistä taulukointia Brownin ym. ja Erlin periaatteiden suhteen ei tehdä. Brownin ym. periaatteet esitetään samalla akselilla Erlin periaatteiden kanssa, jotta nähdään, mitkä Erlin periaatteista sisältyvät mihinkin Brownin ym. periaatteeseen. Palvelukeskeinen arkkitehtuuri on aina yksi yhtenäinen kokonaisuus, joten kaikkien perusperiaatteiden väliltä on löydettävissä selvä yhteys. Taulukon jokaisen solun rastimiselle olisi siis löydettävissä peruste, ainakin näkökulmaa vaihtamalla.

Alla oleva taulukointi on tehty ohjelmistosuunnittelijan näkökulmasta helpottamaan komposiittisovelluksen suunnittelua. Taulukon tarkoituksena on muistuttaa suunnittelijaa mitkä periaatteet kuuluvat läheisesti toisiinsa. Esimerkiksi

TAULUKKO 2 Palvelukeskeisten perusperiaatteiden vastaavuudet

Brown ym.	Erl	Menettelytapakeskeisyys	Löyhä sidos	Standardienmukaisuus	Autonomia	Toimittajariippumattomuus	Selkeät rajat	Yhteiset määrittelyt palvelukohtaiset toteutukset	Metatietokeskeisyys	Viestiformaattien määrittäminen	Dokumenttikeskeisyys	Tilkov
Löyhä sidos	Löyhä sidos		X						X	X		
	Sopimukset	X		X						X		
Itsenäisyys	Autonomia				X		X					
Löydettävyys	Löydettävyys								X			
	Koostettavuus					X				X		
	Uudelleenkäytettävyys			X					X		X	
Karkeajakoisuus	Abstraktio						X	X				
	Tilattomuus						X					

Tilkovin Toimittajariippumattomuus- ja Viestiformaattien määrittämisen periaate tulisi ottaa huomioon samalla, kun suunnitellaan, miten Erlin Koostettavuusperiaate toteutuu. Ristiintaulukointia käytetään apuna luvussa 6, kun arvioidaan kuinka hyvin palvelukeskeisen arkkitehtuurin perusperiaatteet toteutuvat luvun 5 esimerkkisovelluksessa.

Akateemisella ja kaupallisella puolella suurta arvostusta nauttivan Alan W. Brownin ym. lyhyt SOA-määritelmä on aikajärjestyksessä ensimmäinen kolmesta periaatelistauksesta. Thomas Erl on yksi menestyneimmistä SOA-kirjailijoista ja lähestyy aihealuetta verrattain teoreettisesta näkökulmasta. Erlin määrittelemät periaatteet tarkentavat ja laajentavat Brownin ym. periaatteita, jonka vuoksi ne sijoitettiin ristiintaulukoinnissa samalle akselille Brownin ym. periaatteiden kanssa. Stefan Tilkovin määritelmissä näkyy kiinnostuneisuus palvelukeskeisten arkkitehtuurien toteuttamiseen sekä teknisessä että kaupallisessa mielessä.

Tilkovin periaatteissa näkökulma on hieman erilainen, mutta periaatteet eivät kuitenkaan ole ristiriidassa Brownin ym. tai Erlin määrittelyn kanssa. Tämä näkökulma otettiin mukaan perusperiaatteiden tutkimiseen, jotta tutkielman käytännön osiolle saadaan saumaton linkitys käsiteltyyn teoriaosuuteen. Tilkovin perusperiaatteet sijoitettiin erilaisen näkökulman vuoksi ristiintaulukoinnissa eri akselille Brownin ym. ja Erlin kanssa.

5 ESIMERKKISOVELLUKSEN KUVAUS JA TOTEUTUS

Edellisessä luvussa käsiteltiin komposiittisovelluksen toteutuksessa olennaisia palvelukeskeisen lähestymistavan perusperiaatteita. Tässä luvussa toteutetaan yksinkertainen komposiittisovellus ja selvitetään, miten periaatteet toteutuvat sovelluksen komponenteissa ja komposiiteissa.

Sovelluksen toteuttaminen aloitetaan määrittelemällä sovelluksen toiminnot (alaluku 5.3). Alaluvussa 5.4 suunnitellaan tarvittavat komponentit määriteltyjen toimintojen toteuttamiseksi. Komponenttien suunnittelussa käytetään alaluvussa 2.3 esitettyä Davisin 5-kerroksista palvelukerrosjaottelua. Komponenttien ja komposiittien suunnittelun jälkeen arvioidaan, mitkä teknologiat soveltuvat parhaiten suunnitellun sovelluksen toteuttamiseen (alaluku 5.5).

5.1 Taustat ja motivaatio

Toimeksiannon ja kirjoittajan käytännön työkokemuksen vuoksi tutkielmaan haluttiin sisällyttää vahvasti teoriaosuuteen perustuva käytännön toteutus. Tutkielmassa käsiteltyjen periaatteiden toteutuminen ei vaadi valtavan suurta sovellusta, mutta joidenkin periaatteiden käytännön hyöty on suurimmillaan vasta isoissa palvelukeskeisissä arkkitehtuureissa. Sovelluksen haluttiin sekä tuottavan että käyttävän palveluita, jotta periaatteiden toteutuvuutta voidaan

arvioida kummastakin näkökulmasta. Sovellus ei kuitenkaan saisi kasvaa liian suureksi, jotta arviointia voitaisiin tehdä mahdollisimman tarkalla tasolla.

Sovellukseen päätettiin rakentaa kokoelma palveluita ja toteuttaa käyttäjälle näkyvät osat mashup-tyyppisenä komposiittisovelluksena (kohta 2.3.1), joka käyttää hyväkseen sovelluksen oman palvelukokoelman palveluita. Sovellusta ei toteutettu suoraan käytännön tarpeeseen, mutta tutkielmassa kokeiltua lähestymistapaa on käytetty menestyksekkäästi myös kirjoittajan päivätyönään tekemissä keskisuuren ohjelmistoalan yrityksen asiakasprojekteissa.

Eräässä sovelluskokonaisuudessa ilmeni vaatimus, että käyttöliittymäosia haluttaisiin ajaa eri palvelimella kuin sovelluksen muita osia. Perinteisessä websovelluksessa vaatimuksen täyttäminen olisi ollut erittäin vaikeaa ja suuritöistä, mutta esimerkkisovelluksen mukaisten teknologiavalintojen myötä palveluiden siirtäminen ei vaatinut lainkaan muutoksia toteutukseen. Tutkielmasta on saatu ja tullaan tulevaisuudessakin saamaan myös huomattavaa kaupallista hyötyä käytännön sovelluskehityksessä. Palvelukeskeisten arkkitehtuurien, avointen palvelurajapintojen ja big data-järjestelmien yleistymisen myötä esimerkkisovelluksen lähestymistapa tulee todennäköisesti yhä suositummaksi vaihtoehdoksi sovellusten toteuttamiseen.

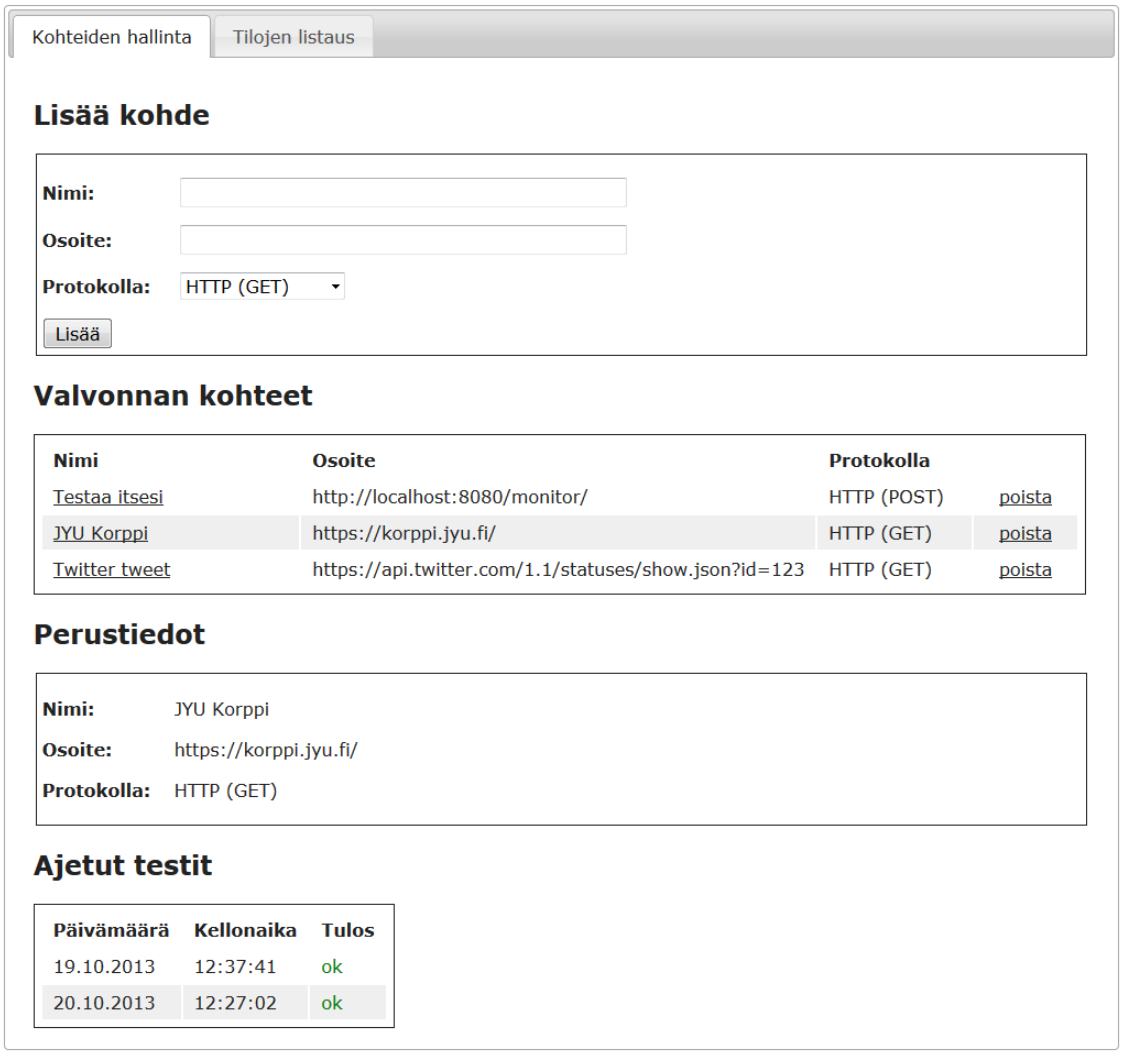
5.2 Sovelluksen yleiskuvaus

Esimerkkisovelluksena toteutetaan valvontatyökalu, jolla voidaan testata, ovatko valvotut palvelut, esimerkiksi nettisivut, käytettävissä. Sovelluksella voidaan valvoa HTTP-protokollan GET- ja POST-metodien mukaisilla rajapinnoilla kommunikoivia palveluita. Toteutuksen tulee olla laajennettavissa tukemaan myös muita protokollia.

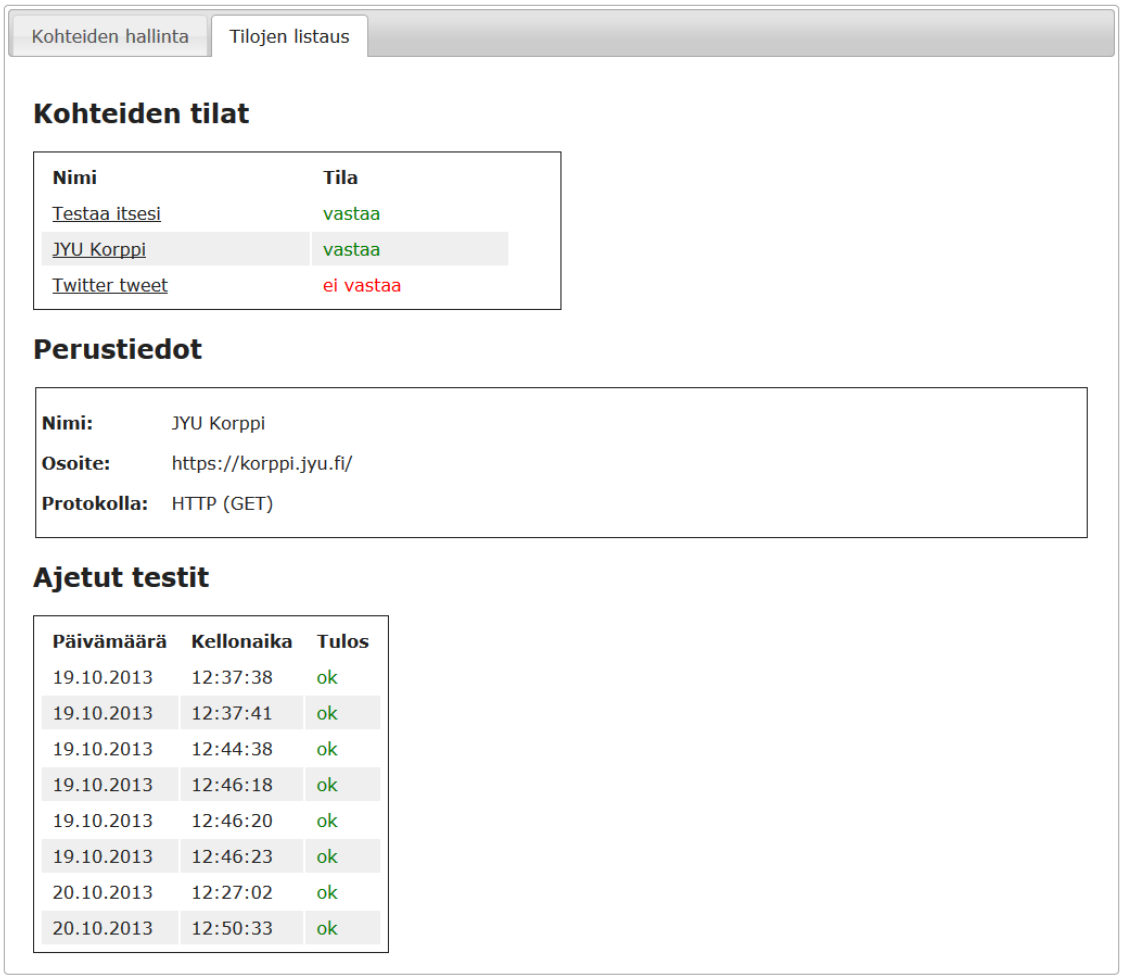
Sovellus palvelee kahta käyttäjäkuntaa; pääkäyttäjiä ja tukihenkilöitä. Pääkäyttäjät voivat lisätä ja poistaa valvottavia palveluita sekä listata palvelut. Tukihenkilöt voivat testata, ovatko palvelut käytettävissä sekä listata ajettujen testi-

en tulokset tietyn palvelun osalta. Kummatkin käyttäjäryhmät voivat myös katsella valvottavien palveluiden perustietoja.

Seuraavissa kuvissa (kuvio 6 ja kuvio 7) havainnollistetaan, miltä valmis valvontasovellus näyttää käyttäjälle. Sovellusta kutsutaan web-selaimella eikä sillä ole sisäisiä tiloja, joten sovellusta itseään voidaan pitää yksinkertaisena REST-arkkitehtuurin mukaisena web-palveluna (kohta 3.3.1). Sovelluksella voidaan valvoa web-palveluita ja näin toteutuksen yhdeksi päämääräksi oli luontevaa asettaa tavoite, että sovellus valvoo itse itseään.



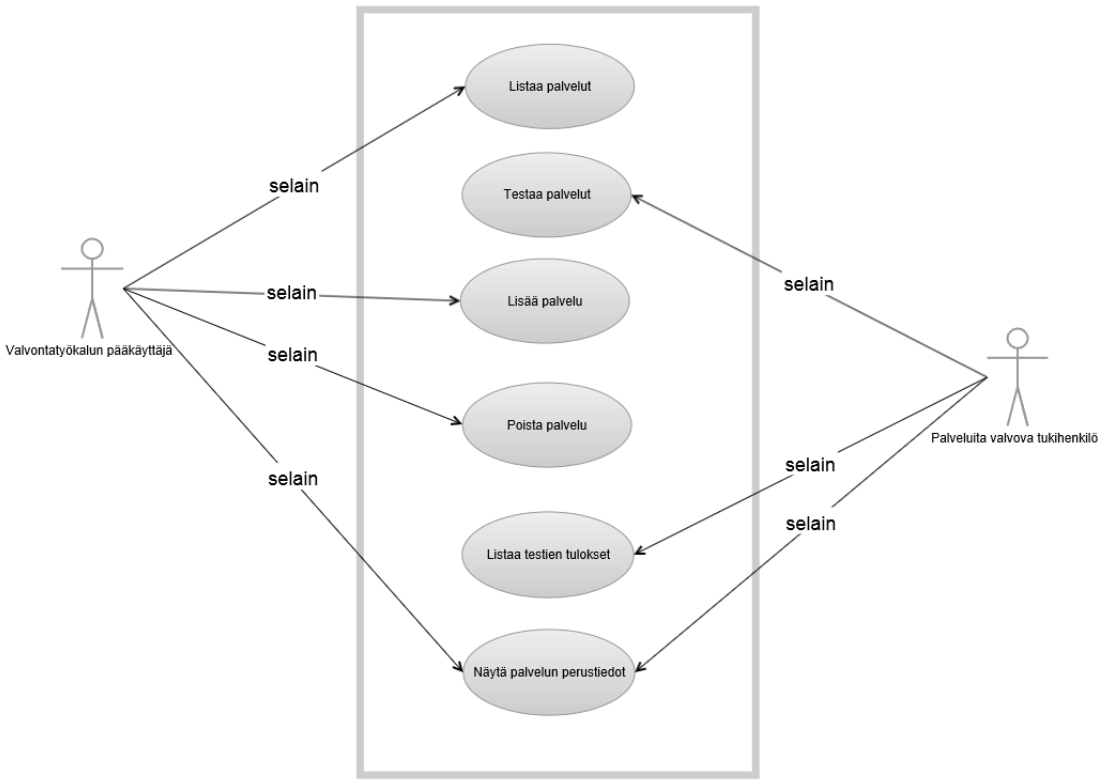
KUVIO 6 Käyttöliittymäkuva valvontatyökalun pääkäyttäjien näkymästä



KUVIO 7 Käyttöliittymäkuva valvontatyökalun tukihenkilöiden näkymästä

5.3 Käyttötapausten määrittely

Sovelluksen suunnittelu aloitettiin käyttötapausten tarkennetulla määrittelyllä. Seuraavassa kuvassa (kuvio 8) esitetään sovelluksen toiminnot käyttötapauskaaviossa. Alaluvuissa kuvataan jokaisen käyttötapauksen toiminnot ja esitetään käyttötapausta vastaava osa käyttöliittymästä. Käyttötapausten määrittely tehtiin ennen kuin lopullista käyttöliittymän ulkoasua oli päätetty, mutta tutkielman luettavuuden vuoksi käyttötapauskuvauksiin on lisätty käyttöliittymäkuvat valmiista sovelluksesta.



KUVIO 8 Valvontatyökalun käyttötapauskaavio

5.3.1 Lisää palvelu

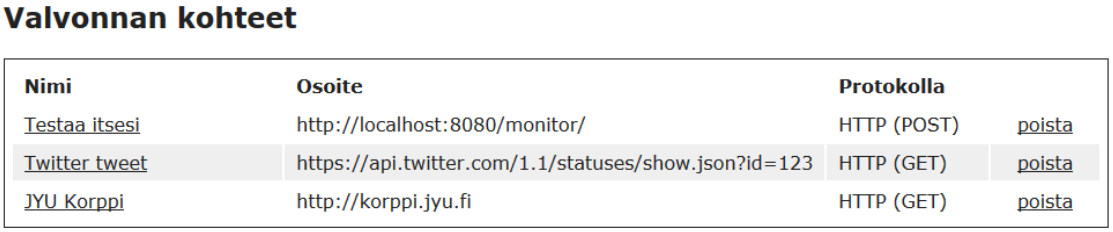
Valvontatyökalun pääkäyttäjä voi lisätä valvottavan palvelun järjestelmään. Seurantaa varten järjestelmälle täytyy antaa palvelun nimi sekä syöttää verkko-osoite, josta palvelun oletetaan vastaavan. Palvelua lisättäessä on valittava myös yhteystapa, jolla palvelun testaus suoritetaan. Valittavana ovat HTTP-protokollan GET- ja POST-metodit. Alla on kuva tämän käyttötapauksen täyttävästä käyttöliittymän osasta (kuvio 9).

5.3.2 Listaa palvelut

Sovelluksen pääkäyttäjä näkee listauksen palveluista, joita sovelluksella valvotaan. Listauksessa näytetään palvelun perustiedot eli nimi, yhteysosoite ja yhteyden käyttämä protokolla. Listauksesta on mahdollista poistaa palvelu, joka

vastaa palvelun poistamisen käyttötapausta (kohta 5.3.3). Alla on kuva tämän käyttötapauksen täyttävästä käyttöliittymän osasta (kuvio 10).

KUVIO 9 Käyttöliittymän osa, jolla pääkäyttäjä voi lisätä seurattavan kohteen



KUVIO 10 Käyttöliittymän osa, jossa listataan valvonnan kohteet

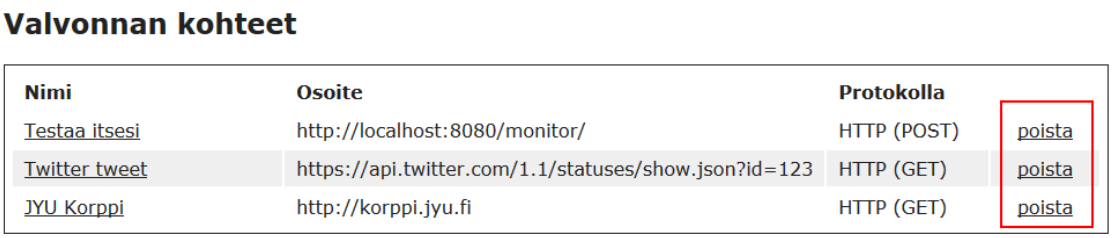
5.3.3 Poista palvelu

Sovelluksen pääkäyttäjä voi poistaa palvelun seurannasta. Kaikki palveluun liittyvät tiedot, mukaan luettuna sovelluksen tallentama testiajojen historia poistetaan. Alla on kuva tämän käyttötapauksen täyttävästä käyttöliittymän osasta (kuvio 11).

5.3.4 Testaa palvelut

Sovellus tarjoaa tukihenkilöille näkymän, jossa listataan seurattavat palvelut ja testataan kunkin palvelun senhetkinen tila. Listauksessa näytetään palvelun nimi ja palvelun tila. Listauksesta voi navigoida kohdan 5.3.5 käyttötapausta

vastaavaan toimintoon. Alla on kuva tämän käyttötapauksen täyttävästä käyttöliittymän osasta (kuvio 12).



KUVIO 11 Käyttöliittymäkuva kohteen poistamistoiminnosta

KUVIO 12 Käyttöliittymäkuva valvottavan palvelun senhetkisestä tilasta

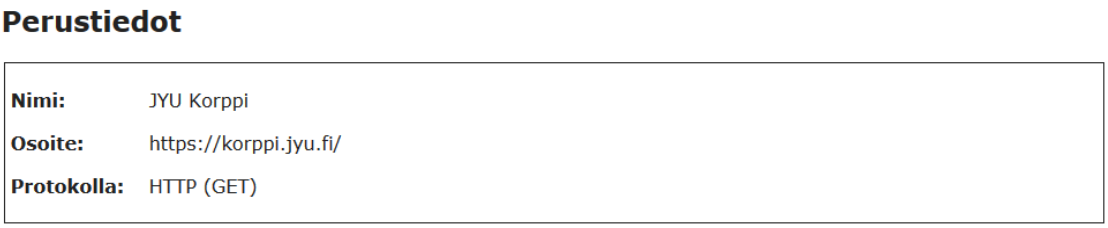
5.3.5 Listaa palvelulle ajettujen testien tulokset

Tukihenkilöt voivat tarkastella ajettujen testien historiaa valitsemansa palvelun osalta. Listauksessa näytetään kyseiselle palvelulle ajettujen testien päivämäärä, kellonaika sekä testin tulos. Virheeseen päättyneiden testien kohdalla näytetään tuloksena virhetilanteen syy. Alla on kuva tämän käyttötapauksen täyttävästä käyttöliittymän osasta (kuvio 13).

5.3.6 Näytä palvelun perustiedot

Tässä näkymässä näytetään palvelun perustiedot eli nimi, yhteysosoite ja yhteyden käyttämä protokolla. Käyttötapaus on yhteinen pääkäyttäjälle ja tukihenkilölle. Alla on kuva tämän käyttötapauksen täyttävästä käyttöliittymän osasta (kuvio 14).

KUVIO 13 Käyttöliittymäkuva, jossa näkyy valitun kohteen testihistoria



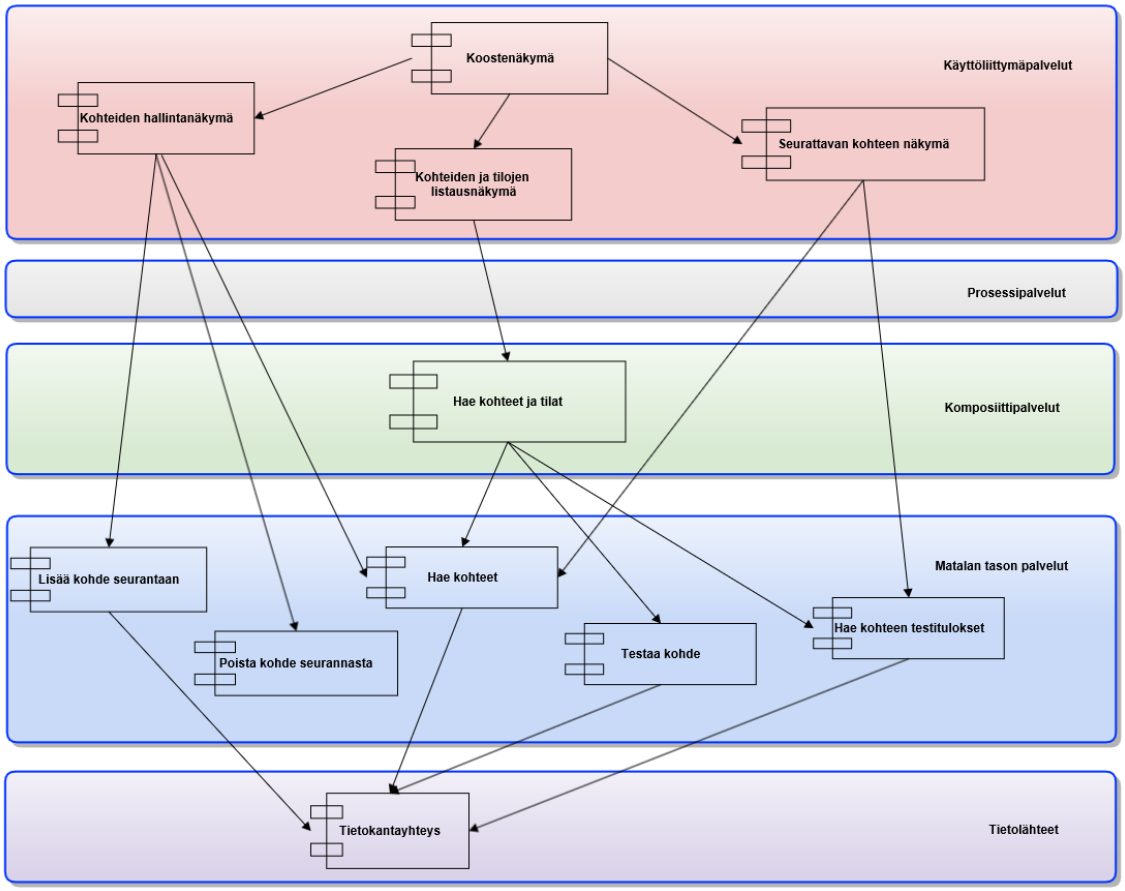
KUVIO 14 Käyttöliittymäkuva, jossa näkyy valitun kohteen perustiedot

5.4 Komponenttien ja komposiittien suunnittelu

Käyttötapauskuvausten perusteella voidaan suunnitella käyttötapausten toteuttamista varten tarvittavat komponentit ja komposiitit. Komponenttien ja komposiittien suunnittelu tehdään Davisin viisitasoisen palvelukerrosajattelun mukaisesti (alaluku 2.3).

Sovelluksen käyttöliittymälle ei ole ulkopuolelta määrättyjä vaatimuksia tai käyttöliittymäkuvia. Toteutuksen suunnittelu voidaan näin ollen aloittaa suunnittelemalla alemman tason palveluita teknisestä näkökulmasta tai käyttäjälähtöisesti hahmottelemalla käyttöliittymäkuvia ja käyttöliittymäprototyyppiä. Suunnittelun lähestymistavaksi valikoitui käyttöliittymälähtöinen suunnittelu, minkä johdosta suunnittelu aloitetaan käyttöliittymäpalveluista ja edetään palvelukerroksittain ylhäältä alaspäin.

Alla olevassa kuvassa (kuvio 15) esitetään kokonaiskuva sovelluksen komponenteista ja niiden keskinäisistä riippuvuuksista. Suunnitellut palvelut käydään läpi yksi kerros kerrallaan seuraavissa alaluvuissa. Jokaista kerrosta vastaavan alaluvun loppuun on lisätty havainnollistavia käyttöliittymäkuvia, joista käy ilmi, miten palvelun palauttama tieto näkyy käyttäjälle. Ainoastaan käyttöliittymäpalvelut palauttavat tiedon käyttäjälle näytettävässä muodossa, mutta alempienkin kerrosten palveluiden osalta tuodaan ilmi, missä kohtaa käyttöliittymää niiden palauttamaa tietoa esitetään.



KUVIO 15 Valvontatyökalun komponentit ja komposiitit

Palveluiden lähdekoodit ovat tutkielman liitteinä. Kunkin palvelukerroksen lähdekoodit ovat omissa liitteissään (liitteet 1-4). Palvelinpään toteutuskielenä esimerkkisovelluksessa on Java, mutta käyttöliittymäpalvelut sisältävät myös html-merkkausta, sekä JavaScript-koodia (liite 1). Sovelluksen tietomalliluokki-

en lähdekoodit ovat liitteessä 5 ja sovellusta varten toteutettujen yleisten työkaluluokkien lähdekoodit ovat liitteessä 6.

5.4.1 Käyttöliittymäpalvelut

Viisitasoisen palvelukerrosjaottelun ylimmällä kerroksella ovat käyttöliittymäpalvelut. Palvelukerrosajattelu ei ota kantaa palveluiden rajapintoihin, joten kaikkien kerrosten palveluita voitaisiin kutsua vaikkapa Internet-selaimella (olettaen, että palveluita tarjotaan HTTP-protokollan mukaisilla rajapinnoilla). Kuitenkin vain ylimmän tason palveluiden vaste on ihmisen luettavaksi suunniteltu, esimerkkisovelluksen tapauksessa selaimessa näkyvä sivu tai osa sivusta.

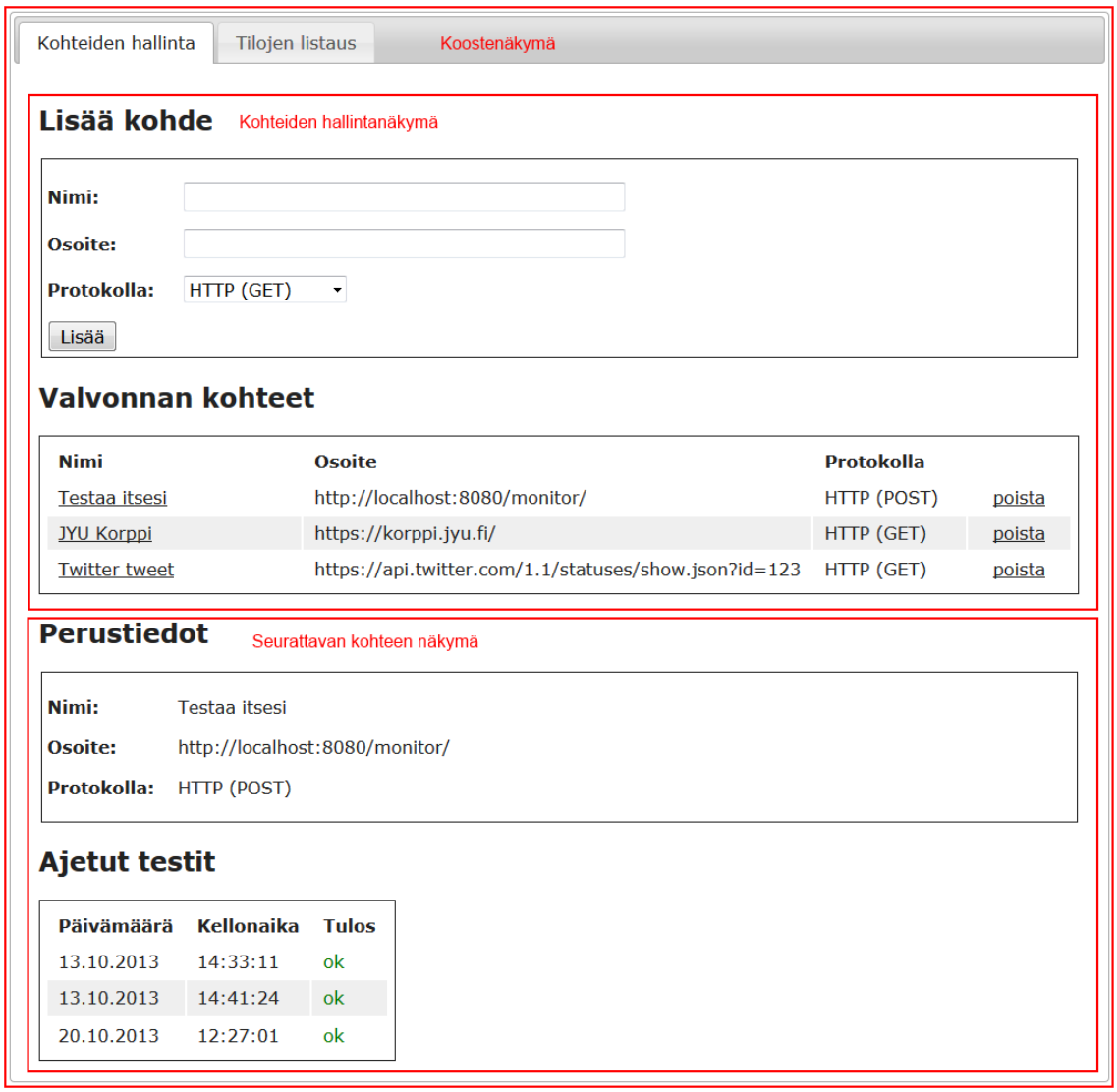
Yksi käyttöliittymäpalvelu voi olla laaja, vaikkapa koko sovelluksen kerrallaan näyttävä kokonaisuus tai pienen käyttöliittymäosan, esimerkiksi tuloslistauksen näyttävä pieni osakokonaisuus. Esimerkkisovelluksen tapauksessa tuntuu luontevalta jakaa myös käyttöliittymäpalvelut pieniin osiin ja koostaa niistä kohdassa 2.3.1 käsitelty mashup-tyyppinen kokonaisuus.

Koostenäkymää havainnollistavissa käyttöliittymäkuvissa käy ilmi, miltä käyttöliittymäpalvelut näyttävät lopullisessa sovelluksessa. Ensimmäisessä käyttöliittymäkuvassa (kuvio 16) ovat koostenäkymän pääkäyttäjälle tarkoitetut osat ja toisessa käyttöliittymäkuvassa (kuvio 17) ovat palveluita valvovalle tukihenkilölle tarkoitetut osat. Kuvaukset lyhyesti ovat seuraavat:

- Koostenäkymä

  Koostenäkymä on mashup-tyylinen (kohta 2.3.1) muita käyttöliittymäpalveluja kutsuva ja niiden käyttöliittymänäkymiä koostava kokonaisuus. Koostenäkymässä eritellään käyttötapauskuvauksien mukaan (kuvio 8) ylläpitäjän käyttötapaukset erilleen tukihenkilöille tarjottavista toiminnoista. Molemmille käyttäjäryhmille yhteiset toiminnot, kuten seurattavan kohteen pe-

rustietojen ja testihistorian näyttäminen, voidaan toteuttaa kyseisen käyttöliittymäkomponentin uudelleenkäytöllä.



KUVIO 16 Sovelluksen pääkäyttäjän käyttöliittymäpalvelut

- Kohteiden hallintanäkymä

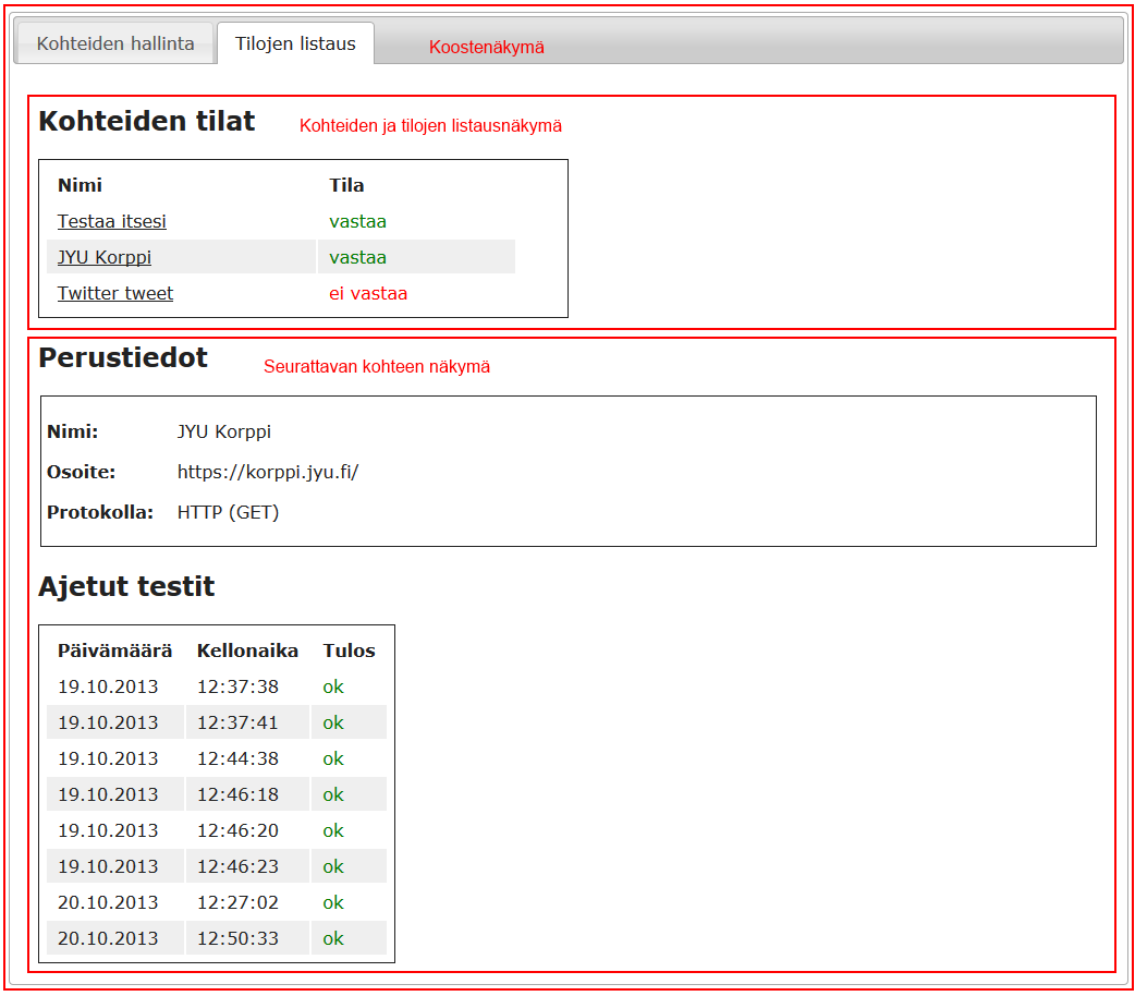
Kohteiden hallintanäkymän kautta voidaan lisätä seurattavia palveluita sovellukseen. Käyttäjän täytyy antaa palvelulle nimi ja kertoa osoite sekä yhteystapa, jolla palvelun toimivuutta testataan. Hallintanäkymässä myös listataan kaikki palvelut ja tarjotaan poistomahdollisuus.

- Seurattavan kohteen näkymä

Valitun kohteen näkymässä käyttäjälle esitetään palvelun perustiedot sekä koko palvelun seuranta-ajan historia. Ajettujen testien historialistauksessa näytetään päivämäärä ja kellonaika, jolloin palvelua on testattu sekä testin tulos.

- Kohteiden ja tilojen listausnäkymä

Seurattavien kohteiden listausnäkymässä käyttäjälle tarjotaan yhdellä silmäyksellä tieto palveluiden saavutettavuudesta. Näkymässä listataan kaikki palvelut sekä jokaiselle palvelulle juuri ajetun testin tulos.



KUVIO 17 Sovelluksen tukihenkilön käyttöliittymäpalvelut

Käyttöliittymäpalveluiden JavaScript- ja Java-lähdekoodit ovat liitteessä 1.

5.4.2 Prosessipalvelut

Toiseksi ylimmällä palvelutasokerroksella ovat prosessipalvelut. Kohdan 2.3.2 mukaan prosessipalvelut kuvaavat tyypillisesti hyvin tarkalla tasolla sovellusta käyttävän osapuolen liiketoimintaprosessit. Prosessipalveluiden kuvauksista voisi nähdä esimerkiksi teollisen prosessin etenemisen tietojärjestelmän näkökulmasta. Yksinkertaisessa esimerkkisovelluksessa tälle tasolle sopivaa pitkäkestoisempaa prosessia ei ole tunnistettavissa, joten vastaavia palveluja ei myöskään kannata suunnitella. Palvelutasoajattelu on pohjimmiltaan vain suunnittelun havainnollistava apuväline, joten pelkästään kerroksien täyttämisen vuoksi ei ole järkevää luoda sovellukselle epäolennaisia palveluita.

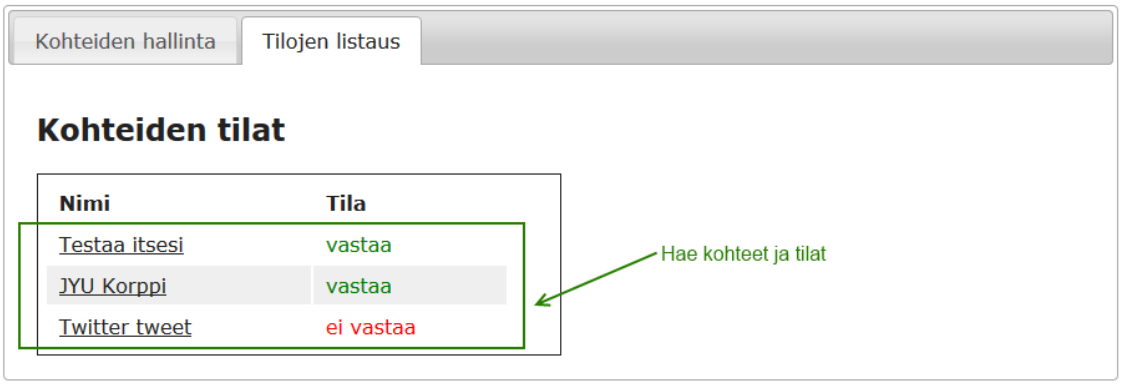
5.4.3 Komposiittipalvelut

Viisikerroksisen palvelutasojaottelun keskimmäisellä kerroksella on komposiittipalvelut. Tämän kerroksen palvelut käyttävät tyypillisesti useaa alemman kerroksen palvelua ja yhdistävät niistä saatuja tietoja suorittaakseen oman tehtävänsä.

Komposiittipalvelut eivät palauta suoraan käyttäjälle näytettäviä käyttöliittymän osia, mutta alla olevassa käyttöliittymäkuvassa (kuvio 18) havainnollistetaan, miten komposiittipalvelun koostama tieto näytetään sovelluksen käyttöliittymässä.

- Hae kohteet ja tilat

  Tämä osa käyttää hyväkseen alemmalta tasolta kohteiden listauspalvelua ja kohteen testauspalvelua. Sovellus listaa kaikki seurattavat palvelut ja ajaa jokaiselle yhteystestin. Tiedoista koostetaan lista tulosolioita ja palautetaan tuloslista kutsujalle. Kutsujalta ei vaadita parametreja.



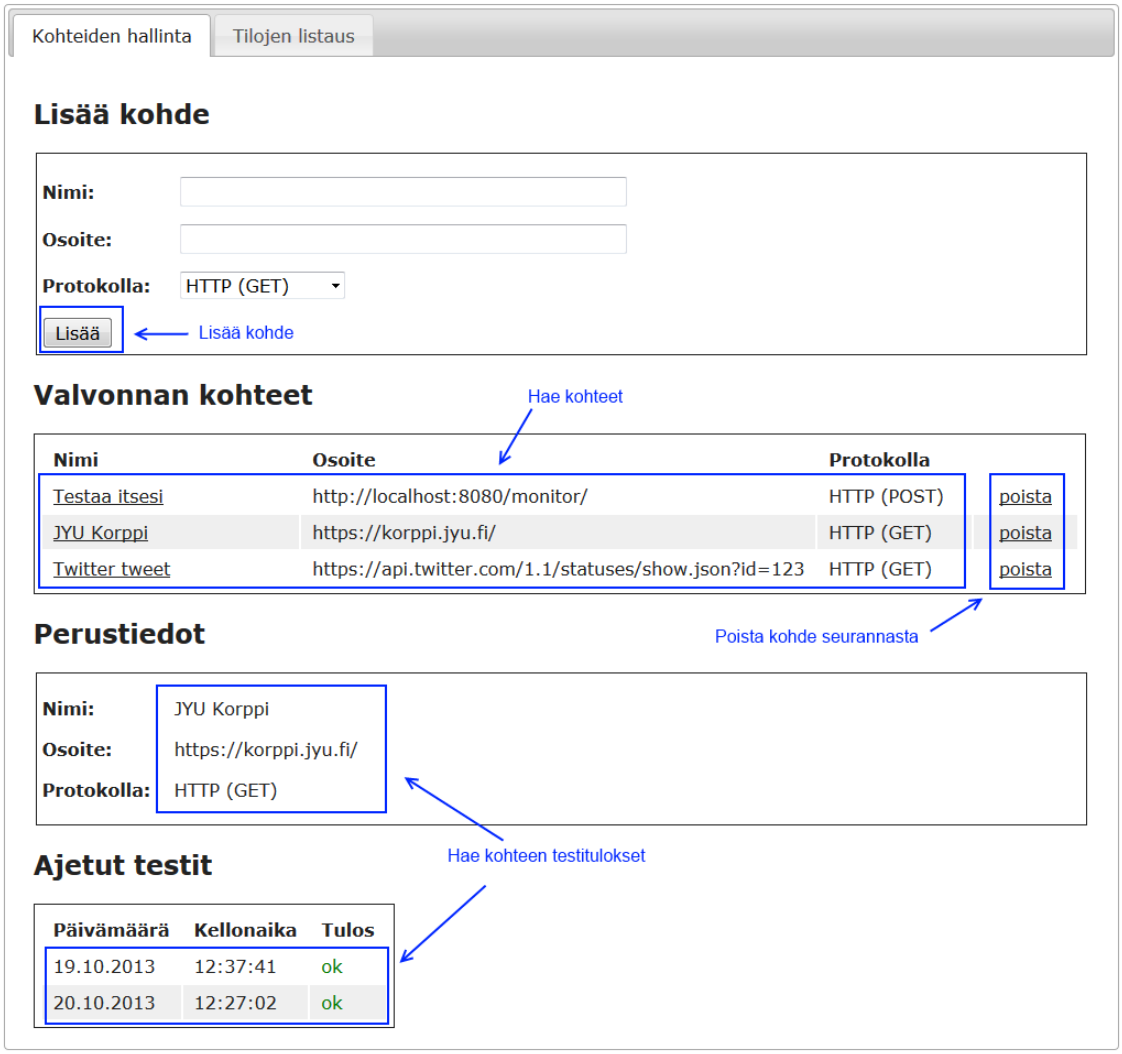
KUVIO 18 Komposiittipalvelun palauttamat tiedot käyttöliittymässä

Komposiittipalveluiden Java-lähdekoodit ovat liitteessä 2.

5.4.4 Matalan tason palvelut

Toiseksi alimmalla palvelukerroksien tasolla on alaluvussa 2.3 käsitellyt, liiketoimintalogiikaltaan verrattain yksinkertaiset matalan tason palvelut. Esimerkkisovelluksen tapauksessa matalan tason palveluiden tavoitteena on muuttaa tietolähteestä saatu raakadata sovelluksen liiketoimintatarpeet täyttäviksi olioiksi.

Matalan tason palveluiden roolina ei ole palauttaa tietoa suoraan käyttöliittymäkerroksen, kuten selaimen, ymmärtämässä muodossa. Käyttöliittymäkuvalla (kuvio 19) havainnollistetaan, mistä osista sovelluksen käyttöliittymäkerrosta kutsut saavat alkunsa. Havainnollistaminen suoraan käyttöliittymätasolle on mahdollista vain niiden kutsujen osalta, jotka menevät suoraan ylimmästä kerroksesta matalan tason palveluille. Monimutkaisemman kutsulogiikan osalta käyttäjän on mahdotonta ja myös tarpeetonta tietää, mitä käyttöliittymäkerroksen jälkeen tapahtuu.



KUVIO 19 Matalan tason palveluiden palauttamat tiedot käyttöliittymässä

- Lisää kohde seurantaan

Lisää annettujen tietojen mukaisen palvelun tietokantaan. Vaatii kutsun parametrina palvelun perustiedot, kuten nimen, osoitteen ja käytettävän yhteysprotokollan.

- Poista kohde seurannasta

Poistaa kyseisen kohteen sekä kohteelle ajettujen testien tiedot tietokannasta. Vaatii kutsun parametrina palvelun yksilöivän teknisen tunnisteen.

- Hae kohteet

Listaa kaikki sovelluksen seuraamat kohteet tietokannasta. Ei vaadi parametreja kutsun mukana.

- Testaa kohde

Suorittaa yksinkertaisen yhteystestin yhdelle kohteelle. Kutsuu kohdetta määriteltyä yhteysprotokollaa käyttäen. Vaatii kutsun parametrina testattavan kohteen teknisen tunnisteen. Tallentaa testin tuloksen tietokantaan. Jos testi päättyy virhetilanteeseen, tallentaa myös virheen syyn tietokantaan.

- Hae kohteen testitulokset

Hakee yhden kohteen perustiedot sekä testihistorian tietokannasta. Vaatii kutsun parametrina seurattavan kohteen teknisen tunnisteen.

Matalan tason palveluiden Java-lähdekoodit ovat liitteessä 3.

5.4.5 Tietolähteet

Viisitasoisen palvelukerrosajattelun mukaan alimmalla tasolla ovat palvelut, jotka tarjoavat rajapinnan erilaisiin tietolähteisiin tai -varastoihin. Relaatiotietokanta soveltuu hyvin esimerkkisovelluksen tietovarastoksi.

- Tietokantayhteys

Tarjoaa palveluita, joilla tallennetaan, muokataan ja haetaan tietovaraston, tässä tapauksessa tietokannan sisältöä. Tietolähdepalveluiden Java-lähdekoodit ovat liitteessä 4.

5.5 Toteutuksen teknologiavalinnat

Tutkielman kirjoittajan mieltymyksien ja aiemman kokemuksen perusteella esimerkkisovelluksen toteutuskieleksi valikoitui Java. Java-termiä käytettäessä voidaan tarkoittaa muun muassa Java-ajoympäristöä, Java-ohjelmointikieltä tai jotakin Java-ohjelmointikielen ohjelmistoalustaa. Ajoympäristö tarjoaa ohjelmistolle pääsyn alustajärjestelmän resursseihin, kuten keskusmuistiin ja oheislaitteisiin. Ohjelmistoalusta sisältää valmiita kirjastoja ohjelmistojen kehittäjille. Perinteisiä työpöytäohjelmia kehittäessä voidaan käyttää Java SE (Java Standard Edition)-standardin mukaista ohjelmistoalustaa, joka tarjoaa yleiskäyttöisiä kirjastoja muun muassa tiedostonhallintaan, laskentaan, tekstinkäsittelyyn sekä verkon käyttämiseen. (McGovern ym., 2003a, 3–5; Oracle, 2004)

Laajojen web-sovelluksien kehittämistä varten on rakennettu Java SE:n päälle laajempia ohjelmistokehyksiä. Java-standardi tarjoaa mahdollisuuden web-sovelluksien kehittämiseen Java EE:n määrittelyn osana. Javan ajoympäristöön on kehitetty valtava määrä erityyppisten sovellusten toteuttamiseksi tarkoitettuja ohjelmistokehyksiä. Web-sovelluksia Javalla tehneet ovat todennäköisesti huomanneet tyrmistyttävän laajan ohjelmistokehysvalikoiman; esimerkiksi Java-Source.net listaa pelkästään avoimen lähdekoodin puolelta yli 60 ohjelmistokehystä web-sovellusten tekemiseen Java-alustalle. (Java-Source.net, 2013)

5.5.1 Java EE

Java EE (Java Enterprise Edition) on Java SE:tä laajempi ohjelmistokehys, joka tarjoaa valmiita kirjastoja laajojen, moduulirakenteisten järjestelmien kehittämiseen. Ohjelmistokehittäjälle Java EE tarjoaa normaalien työpöytäsovellusten lisäksi mahdollisuuksia web-sovellusten rakentamiseen sekä monia vaihtoehtoja sovellusten integroimiseen. Web-palveluita voidaan käyttää sekä Java EE-sovellusten keskinäiseen integraatioon että ulkoisten sovellusten kanssa kommunikoimiseen. (Jendrock, Ball, Carson, Evans, Fordin & Haase, 2008, 51–63)

Java EE on kehittynyt ja laajentunut huomattavasti vuonna 1998 julkaistusta ensimmäisestä versiosta nykyiseen Java EE 7 versioon. Suurimpana kehitysaskeleena voidaan pitää vuonna 2006 julkaistua Java EE 5 versiota. Viidennessä versiossa mahdollistettiin tavallisten Java-olioiden (POJO, Plain Old Java Object) käyttö EJB-olioina (Enterprise Java Bean) sekä yksinkertaistettiin EJB-olioiden keskinäisten viittauksien hallinnointia annotaatioperustaisella riippuvuuksien injektoinnilla (DI, Dependency Injection). (Goncalves, 2013, 23–25)

Riippuvuuksien injektointi on keskeisessä osassa Spring-ohjelmistokehyksessä (kohta 5.5.2), jonka suosion myötä periaatteet ovat päätyneet myös Java EE:n standardiin.

5.5.2 Spring Framework

Monipuolista ja paljon suosiotakin saanutta Java EE-standardia pidettiin vuosituhannen taitteessa verrattain monimutkaisena ja raskaana ohjelmistokehyksenä. Tästä motivoituneena australialainen ohjelmoija Rod Johnson kehitti yhdessä itävaltalaisen kollegansa Juergen Hoellerin kanssa kilpailevan Spring Framework-ohjelmistokehyksen, jonka ensimmäinen versio julkaistiin vuonna 2002 heidän kirjassaan *Expert One-on-One J2EE™ Development without EJB™*. (Johnson & Hoeller, 2004)

Spring-ohjelmistokehys on yli kymmenvuotisen historiansa aikana kasvanut erittäin monipuoliseksi kokonaisuudeksi, mutta alkuperäiset perusperiaatteet ovat pääosassa myös uusimmissa versioissa (kirjoitushetkellä uusin versio on 3.2). Ohjelmoinnin kannalta olennaisimmat niistä ovat tavallisten Java-olioiden käyttö ohjelmistokehyksestä perittyjen olioiden sijasta, löyhän sidonnan toteuttaminen riippuvuuksien injektoinnilla (DI) sekä deklaratiivisen ohjelmointityylin korostaminen aspektilähtöisellä ohjelmoinnilla (AOP, Aspect Oriented Programming). (Craig, 2011, luku 1)

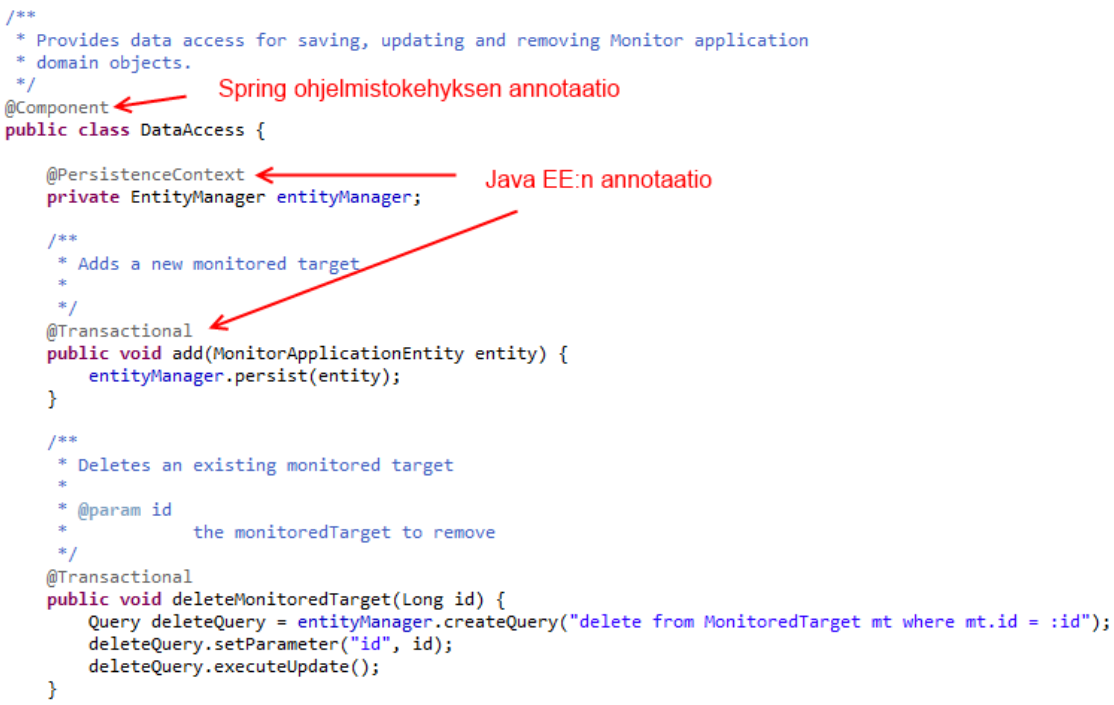
Spring Frameworkin perusperiaatteet ovat hyvin olennainen osa ohjelmistokehyksen käyttöä ja yksinkertaisimmassakin sovelluksessa käytetään riippuvuuksien injektointia sekä tavallisten Java-olioiden muunnosta ohjelmistoalustan olioiksi annotaatioiden avulla. Tutkielman kannalta on olennaista toteuttaa web-palveluita mahdollisimman yksinkertaisella tavalla, jonka DI- ja annotaatioperustaiset tekniikat mahdollistavat.

5.5.3 Ohjelmistokehyksen valinta

Edellisissä luvuissa esitellyt ohjelmistokehykset tarjoavat hyvän pohja-arkkitehtuurin sekä runsaasti valmiita ratkaisuja ohjelmistokehitykseen. Kummatkin sopisivat yhtä hyvin esimerkkisovelluksen toteutukseen, mutta tällä kertaa pääasialliseksi toteutusalustaksi valikoitui Spring Framework. Spring Framework ja Java EE sopivat keskenään myös hyvin yhteen. Esimerkkisovelluksessa oliot tallennetaan tietokantaan käyttäen Java EE:n standardikokonaisuuteen sisältyvän JPA:n (Java Persistence API) referenssitoteutusta.

Toteutuksen luettavuuden ja kokonaisuuden hahmottamisen kannalta riippuvuuksien injektointi on tärkein yksittäinen osa-alue. Esimerkkisovelluksessa injektointi tehdään annotaatioiden avulla. Seuraavat kuvat (kuvio 20 ja kuvio 21) ovat kuvakaappauksia valvontasovelluksen lähdekoodista. Kuviin on merkitty punaisilla nuolilla esimerkkikohtia, joissa tehdään riippuvuuksien injektointia annotaatioita käyttäen.

Ensimmäisessä kuvassa (kuvio 20) on valvontasovelluksen tietolähdepalveluita tarjoava luokka, jossa käytetään Springin ja Java EE:n annotaatioita. Luokalle on määritelty Springin @Component annotaatio, jonka perusteella sovelluksen mukana kulkeva Spring-ohjelmistokehys tekee yhden olioesiintymän DataAccess-luokasta ja ympäröi esiintymän omalla säiliöoliollaan (framework bean). Oletuksena Spring nimeää pavut luokan nimellä eli tässä tapauksessa Springin papulistaan tulee DataAccess-niminen papu.

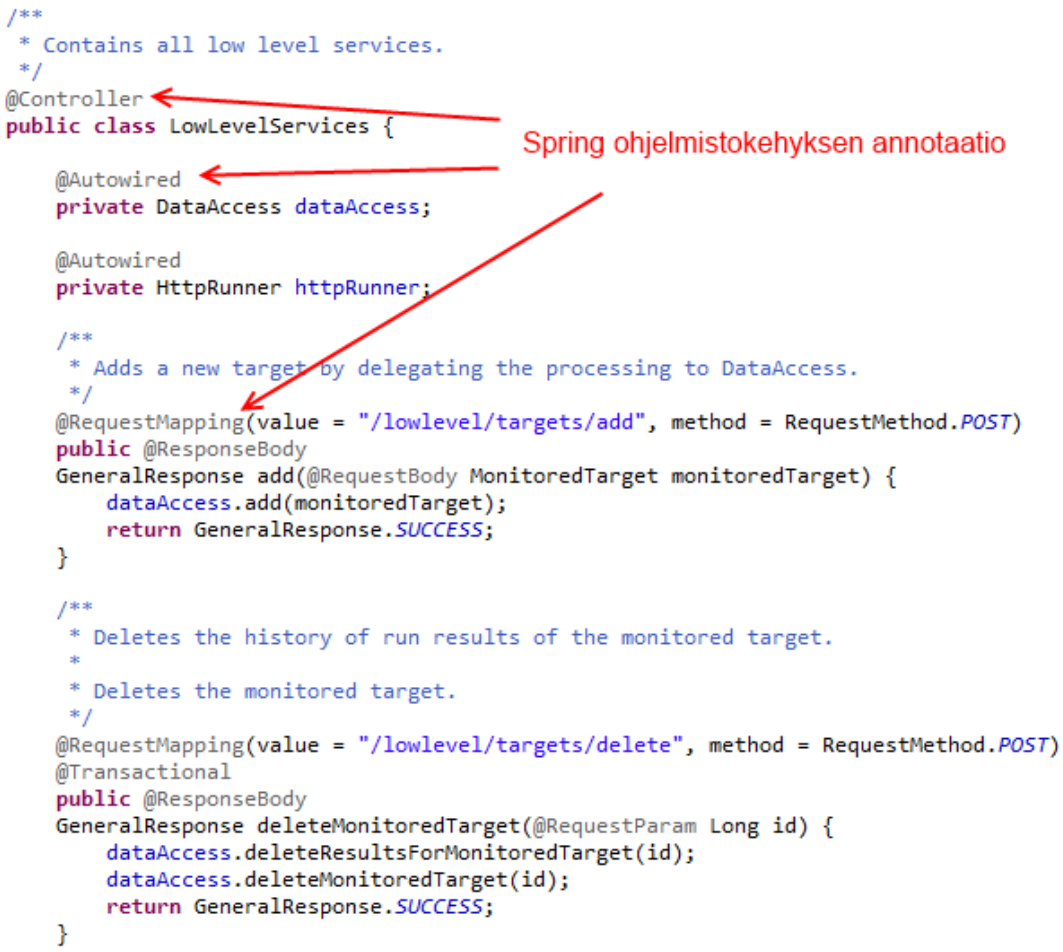



```java
/**
 * Provides data access for saving, updating and removing Monitor application
 * domain objects.
 */
@Component
public class DataAccess {

    @PersistenceContext
    private EntityManager entityManager;

    /**
     * Adds a new monitored target
     *
     */
    @Transactional
    public void add(MonitorApplicationEntity entity) {
        entityManager.persist(entity);
    }

    /**
     * Deletes an existing monitored target
     *
     * @param id
     *            the monitoredTarget to remove
     */
    @Transactional
    public void deleteMonitoredTarget(Long id) {
        Query deleteQuery = entityManager.createQuery("delete from MonitoredTarget mt where mt.id = :id");
        deleteQuery.setParameter("id", id);
        deleteQuery.executeUpdate();
    }
```

KUVIO 20 Tietolähdepalveluita tarjoava Java-luokka

Jälkimmäisessä kuvassa (kuvio 21) DataAccess-tyyppinen muuttuja on merkitty Springin @Autowired-annotaatiolla. Spring etsii papulistastaan oletuksena muuttujan tyypin nimistä papua, löytää edellisessä kuvassa (kuvio 20) esitellyn DataAccess-nimisen pavun ja asettaa, toisin sanoen injektoi, sen muuttujaan. Käytetyt annotaatiot ovat ajonaikaisia annotaatioita, joten ne kulkevat käännetyn luokan mukana.

Spring lukee annotaatiot sovelluksen käynnistyessä, joten päätös siitä, mikä papu injektoidaan muuttujaan, tapahtuu vasta käynnistyksen aikana. Springin pavut ovat oletuksena ainokaisia (singleton), joten sovelluksen kaikkien olioiden DataAccess-tyyppisiin muuttujiin, joilla on @Autowired-annotaatio, injektoidaan sama olioesiintymä. Ainokainen on olion luontiin liittyvä suunnittelumalli, joka takaa, että luokalla on vain yksi ilmentymä. (Gamma, Helm, Johnson & Vlissides, 1995, sivu 127)




```java
/**
 * Contains all low level services.
 */
@Controller
public class LowLevelServices {

    @Autowired
    private DataAccess dataAccess;

    @Autowired
    private HttpRunner httpRunner;

    /**
     * Adds a new target by delegating the processing to DataAccess.
     */
    @RequestMapping(value = "/lowlevel/targets/add", method = RequestMethod.POST)
    public @ResponseBody
    GeneralResponse add(@RequestBody MonitoredTarget monitoredTarget) {
        dataAccess.add(monitoredTarget);
        return GeneralResponse.SUCCESS;
    }

    /**
     * Deletes the history of run results of the monitored target.
     *
     * Deletes the monitored target.
     */
    @RequestMapping(value = "/lowlevel/targets/delete", method = RequestMethod.POST)
    @Transactional
    public @ResponseBody
    GeneralResponse deleteMonitoredTarget(@RequestParam Long id) {
        dataAccess.deleteResultsForMonitoredTarget(id);
        dataAccess.deleteMonitoredTarget(id);
        return GeneralResponse.SUCCESS;
    }
```

KUVIO 21 Matalan tason palveluita tarjoava Java-luokka

Springin injektoinnin lisäksi DataAccess-luokassa (kuvio 20) käytetään Java EE:n tietokantarajapintaa, jolla sovelluksen oliot voidaan tallentaa relaatiotietokantaan. Tietokantayhteys injektoidaan @PersistenceContext-annotaatiolla, jonka perusteella Java EE:n annotaatioiden käsittelijä tekee tietolähteen injektoinnin. Metoditasolla käytetään @Transactional-annotaatiota, jolla kerrotaan Java EE:n tietokantayhteyden käsittelijälle, että metodin aikana tehtävät tietokantaoperaatiot tulee suorittaa tietokantatransaktion sisällä. Kyseiset Java EE:n annotaatiot ovat myös ajonaikaisia annotaatioita, joten ne kulkevat käännettyjen luokkien mukana.

Edellisessä kohdassa (5.5.2 Spring Framework) todettiin Spring-ohjelmistokehyksen kasvaneen varsin monipuoliseksi ohjelmistojen toteutus-

työkaluksi. Esimerkkinä monipuolisuudesta mainittakoon LowLevelServices-luokassa (kuvio 21) käytetty @RequestMapping-annotaatio. Kyseisellä annotaatiolla merkitty metodi tarjotaan HTTP-rajapinnan palveluna sovelluspalvelimelta ulospäin.

Parametrin välityksessä käytetty @RequestBody-annotaatio puolestaan kertoo, että HTTP-kutsun mukana tulee olioparametri. Oletuksena olioparametrin välitys tapahtuu JSON-formaatilla (kohta 3.5.1 JSON), jonka Spring muuttaa automaattisesti Java-olioksi ja välittää parametrina annotoidulle metodille. Jos oletetaan, että valvontasovellus olisi saatavilla osoitteesta

http://www.valvontasovellus.fi

niin LowLevelServices-luokan (kuvio 21) add-metodia kutsuttaisiin osoitteella

http://www.valvontasovellus.fi/lowlevel/targets/add

Vastaavan metodin vaatima MonitoredTarget-olio välitettäisiin HTTP-kutsun POST-parametrina JSON-muodossa, joka alkaisi esimerkiksi näin

{"name":"JYU Korppi", "smokeTestUrl":{…

5.5.4 jQuery

Rakennettaessa sovellusta palvelukeskeiseen arkkitehtuuriin, jossa palveluiden viestiformaatti on JSON, käyttöliittymäkerrokseen kirjoitetaan verrattain paljon ohjelmakoodia. Esimerkkisovelluksen tapauksessa käyttöliittymäkerros toteutettiin HTML- ja JavaScript-teknologioilla, minkä ansiosta palvelut ja käyttöliittymätoteutus säilyivät teknisesti autonomisina (alaluku 4.2).

Useiden muiden ohjelmointikielien tapaan myös JavaScript-kielellä on toteutettu erittäin tehokkaita työkalukirjastoja ja ohjelmistokehyksiä. Sovelluskehittäjän kannattaa useimmiten valita käyttöön vähintäänkin hyvä kirjasto. Esimerkkiso-

velluksen käyttöliittymäkerroksen toteuttamiseen harkittiin AngularJS-, Backbone.js- ja jQuery-kirjastoja, joista lopulta käyttöön valittiin jQuery. AngularJS ja Backbone.js ovat jo laajoja ohjelmistokehyksiä, jotka toteuttavat tuen useille suunnittelumalleille. Esimerkkisovelluksen avulla tutkitaan ensisijaisesti ohjelmiston toteuttamista palvelukeskeisen arkkitehtuurin palveluiden päälle, joten käyttöliittymäkerroksen koodi on syytä pitää mahdollisimman yksinkertaisena ja luettavana, jotka puolestaan ovat jQueryn vahvuuksia. (Franklin, 2013, 15–20)

jQueryn kehitys aloitettiin vuonna 2005 ohjelmoijan ja JavaScript-kirjailijan John Resigin ideoiden pohjalta. Nykyään jQueryn tukijoihin lukeutuu muun muassa jQuery-järjestön perustanut ja kirjoittamishetkellä (2014) maailman suosituimman julkaisualustan tarjoava WordPress sekä useita muita alan suuria toimijoita, kuten Mozilla, Adobe ja Intel. (Franklin, 2013, 15; jQuery, 2014)

5.6 Yhteenveto

Tässä luvussa määriteltiin, suunniteltiin ja toteutettiin komposiittisovellus. Suunnittelu- ja toteutusvaiheiden tulokset esitettiin luettavuuden nimissä kuvakaappauksina määrittelyjen yhteydessä. Käyttötapausten vaatimat palvelut suunniteltiin viisitasoisen esimerkkiarkkitehtuurin mukaisesti ja sijoitettiin arkkitehtuuriin alaluvun 2.3 jaottelua noudattaen. Palvelut käytiin läpi tasoittain havainnollistaen samalla palvelun rooli loppukäyttäjän näkökulmasta kuvakaappauksia hyödyntäen.

Sovelluksen läpikäynnin jälkeen paneuduttiin toteutuksessa käytettyihin teknologioihin koodiesimerkkien ja ohjelmistotuotannossa yleisesti käytettyjen suunnitteluperiaatteiden kautta. Teknologioista esiteltiin sovelluspalvelimella pyörivät palvelinpään ohjelmistokehykset sekä loppukäyttäjän selaimessa ajettava käyttöliittymäkerroksen ohjelmistokehys.

6 ESIMERKKISOVELLUKSEN ARVIOINTI ESITETYN TEORIAN NÄKÖKULMASTA

Tässä luvussa tutkitaan ja arvioidaan edellisessä luvussa esiteltyä komposiittisovellusta palvelukeskeisen arkkitehtuurin sekä toteutuksen helppouden näkökulmasta. Alaluvussa 6.1 selvitetään, noudattaako esimerkkisovellus luvussa 4 esitettyjä palvelukeskeisen arkkitehtuurin perusperiaatteita.

Web-palveluista koostetun komposiittisovelluksen soveltuvuutta web-ohjelmistojen toteuttamiseen sekä toteutuksen aikana havaittuja riskejä käsitellään alaluvussa 6.2. Arviointi tehdään subjektiivisesti kirjoittajan käytännön työelämän kokemukseen sekä alaluvun 6.1 havaintoihin perustuen.

6.1 Palvelukeskeisen arkkitehtuurin perusperiaatteiden toteutuminen

Esimerkkisovelluksen toteutuksen katselmointi jaetaan alalukuihin Brownin ym. neljän pääperiaatteen mukaan ja tarkennetaan Erlin ja Tilkovin periaatteilla luvun 4 lopussa esitettyä ristiintaulukointia noudattaen.

6.1.1 Löyhä sidos

Sovelluksen toteutus perustuu palveluihin, joista kaikkien rajapintoja voidaan käyttää verkon yli yleisesti käytössä olevien protokollien ja viestiformaattien avulla. Sovellus käyttää omia palveluitaan Javan sisäisillä oliokutsuilla niissä tapauksissa, joissa palvelun käyttäjä ja tarjoaja on Java-olio, mutta palvelut tar-

jotaan myös HTTP-protokollan yli JSON-viestiformaattia (kohta 3.5.1) käyttäen. Erlin mukaan löyhä sidos toteutuu, kun palvelut eivät ole riippuvaisia toistensa toteutustavoista. Esimerkkisovelluksen palveluita voitaisiin kutsua yhtä hyvin vaikkapa .NET-alustan sovelluksesta, joten sovelluksen voidaan katsoa täyttävän riippumattomuusehdon.

Ristiintaulukoinnin mukaan Brownin ym. löyhän sidoksen määritelmä sisältää Erlin sopimukset-periaatteen, jonka mukaan palvelun täytyy esittää standardilla tavalla tieto tarjoamistaan operaatioista sekä kunkin operaation vaatimista parametreista. Esimerkkisovelluksessa palvelut tarjotaan suoraan HTTP-protokollan yli JSON-viestiformaattia käyttäen, jonka vuoksi palveluista ei ole mahdollista luoda esimerkiksi WSDL-määrityksiä (kohta 3.5.3). Sopimukset-periaatteen toteutuminen vaatisi viestiformaatin vaihtamisen JSON:sta (kohta 3.5.1) esimerkiksi XML:ään, jolloin viestit voitaisiin tehdä SOAP-standardin mukaisesti ja julkistaa rajapinnat WSDL:n mukaan.

Tilkovin määritelmässä löyhä sidos voi toteutua ajan, sijainnin, tyyppien, versioiden, kardinaalisuuden, löydettävyyden tai rajapintojen suhteen. Verrattain yksinkertaisen esimerkkisovelluksen toteutus ei sulje näitä pois, mutta periaatteiden toteutumisesta tulisi huolehtia jatkokehityksen aikana.

Ristiintaulukoinnin perusteella myös Tilkovin menettelytapakeskeisyysperiaate sisältyy Brownin ym. määritelmään. Menettelytapakeskeisyyden toteutuminen vaatii palveluiden tarjoajien ja kutsujien noudattavan samoja määrittelyperiaatteita. Esimerkkisovelluksen tapauksessa palveluiden tarjoaja ja käyttäjä on saman sovelluksen sisällä, joten ristiriitaa ei pääse syntymään. Hyvin vähän rajoittavien toteutusteknologioiden myötä on kuitenkin mahdollista, että jatkokehityksen aikana palveluiden määrittelyperiaatteet eroavat toisistaan. Esimerkkinä voisi olla vaikkapa virhetilanne operaatiota suorittaessa. Palvelun tarjoajan määritelmien mukaan voitaisiin lähettää tyhjä vastaus ja palvelun kutsujan määritelmien mukaan vasteena pitäisi tulla virheestä kertova viesti.

6.1.2 Itsenäisyys

Itsenäisyys- eli autonomiaperiaate on omana perusperiaatteena kaikissa kolmessa määritelmässä. Erlin mukaan autonomian toteutumista voidaan tarkastella ajonaikaisten ja suunnitteluaikaisten ominaisuuksien mukaan.

Ajonaikainen autonomia toteutuu, jos palvelut eivät käytä muiden palvelujen kanssa jaettuja tietolähteitä tai osatoiminnallisuuksia muuten kuin palvelurajapintojen kautta. Esimerkkisovelluksen tapauksessa kaikki palvelut pyörivät teknisesti samassa sovelluksessa, kutsuvat toisiaan Java-olioviitteillä ja käyttävät samaa tietokantayhteyttä, joten ajonaikainen autonomia ei nykytilanteessa toteudu. Teknisesti palvelut voitaisiin kuitenkin jakaa omiin sovelluksiinsa ja laittaa pyörimään vaikkapa eri sovelluspalvelimille tekemättä mitään muutoksia palveluiden varsinaiseen toteutukseen. Jokainen palvelu on tarjolla myös HTTP-protokollalla, jolloin keskinäisistä Java-riippuvuuksista päästään eroon. Ajonaikaisen autonomiaperiaatteen toteutuminen olisi siis teknisesti mahdollista, mutta siitä olisi tässä tapauksessa enemmän haittaa kuin hyötyä.

Erlin määritelmän mukainen suunnitteluaikainen autonomia toteutuu, jos palvelu ei ole riippuvainen muista arkkitehtuurin osista. Yksittäisen palvelun osalta pyrkimys suunnitteluaikaiseen autonomiaan voi olla järkevää, mutta komposiittipalveluiden ja -sovelluksen ideana on luoda riippuvuuksia muihin palveluihin. Riippuvuuksien tunnistamisella ja riskianalyysillä voidaan selvittää miten yhden palvelun kaatuminen vaikuttaa komposiittisovelluksen toimintaan, joten tämäkin periaate on varsin olennainen komposiittisovellusta muodostettaessa, vaikkei sen täydelliseen toteutumiseen pyrittäisikään.

Tilkovin määritelmän mukaan autonomiaperiaate toteutuu, jos palvelu kommunikoi ympäristönsä kanssa vain ulkoisen rajapintansa kautta. Esimerkkisovelluksen osalta Tilkovin määritelmä vastaa Erlin ajonaikaisen autonomian määritelmää ja olisi kokonaan toteutettavissa, jos se nähtäisiin tarpeelliseksi.

6.1.3 Löydettävyys

Palvelukeskeisen arkkitehtuurin näkökulmasta löydettävyys on yksi laajimmista ja haastavimmista perusperiaatteista. Komposiittisovelluksen näkökulmasta löydettävyys on pääasiassa suunnitteluvaiheessa huomioitava periaate, jota tarvitaan, kun kartoitetaan, mitä palveluita komposiittisovelluksen määrittelyjen täyttämiseksi tarvitaan ja mitä palveluita haluttaisiin mahdollisesti antaa muiden käyttöön palveluluettelon kautta. Esimerkkisovelluksen käyttämät palvelut sijaitsevat sovelluksen sisällä, eikä sovelluksen arkkitehtuurissa käytetä palveluluetteloa.

Ristiintaulukoinnissa Brownin ym. löydettävyysperiaatteeseen yhdistettiin Erlin periaatteista lisäksi koostettavuus ja uudelleenkäytettävyys, jotka puolestaan ovat toisiaan täydentäviä ja komposiittisovelluksen kannalta hyvin keskeisiä periaatteita. Komposiitin muodostaminen on palveluiden koostamista ja samalla myös uudelleenkäyttöä. Uudelleenkäytettävyyden Erl jakaa kolmelle eri tasolle, mutta yksinkertaisessa esimerkkisovelluksessa ei ole järkevää ajatella muuta kuin ensimmäistä eli taktista tasoa. Sovelluksessa uudelleenkäytetään kaikkia siinä toteutettuja palveluita, joten Erlin taktisen uudelleenkäyttötason voidaan sanoa toteutuneen täydellisesti.

Tilkovin perusperiaatteista löydettävyyteen yhdistettiin ristiintaulukoinnissa metatietokeskeisyys ja dokumenttikeskeisyys. Metatietokeskeisyys liittyy pääasiassa palveluiden löytämiseen palvelukeskeisessä arkkitehtuurissa. Dokumenttikeskeisyysperiaatteella Tilkov korostaa järkevän viestiformaatin valintaa. Esimerkkisovelluksen viestiformaatille ei ollut ulkopuolisia vaatimuksia, joten formaatiksi valikoitiin tiedonsiirroltaan minimaalinen, mutta ihmisen luettavissa oleva JSON (kohta 3.5.1 JSON).

6.1.4 Karkeajakoisuus

Karkeajakoisuusperiaatteen voisi yhdistää ohjelmistokomponenttiin sen tyypistä riippumatta. Palveluita ja komposiitteja tutkittaessa Brownin ym. karkeajakoisuusperiaatteeseen yhdistettiin Erlin abstraktio- ja tilattomuusperiaatteet sekä Tilkovin selkeät rajat periaatteeseen. Myös Tilkovin yhteiset määrittelyt, palvelukohtaiset toteutukset periaate yhdistettiin karkeajakoisuuteen, mutta sen toteutumista olisi järkevää tutkia vasta laajemmassa palvelukeskeisessä arkkitehtuurissa.

Abstraktioperiaatteen Erl jakaa tyypeittäin teknologiseen, funktionaaliseen, sisäisen logiikan ja palvelun laadun abstraktioon. Esimerkkisovelluksessa abstraktio on huomioitu teknologiavalinnassa sekä palveluiden rajapintojen ja toteutuksen suunnittelussa. Kaikki sovelluksen palvelut käsittelevät samoja liiketoimintaolioita, joten funktionaalista tai palvelun laadun abstraktiotasoa on vaikea määrittää. Erlin mukaan korkea abstraktio mahdollistaa löyhien sidosten toteuttamisen, mutta liiallinen abstraktio tekee käytännön toteutuksesta vaikeasti luettavan ja ylläpidettävän. Esimerkkisovelluksen osalta voidaan todeta, että abstraktiotaso on sopiva, koska löyhien sidosten periaate toteutui hyvin ja toteutus on säilynyt selkeänä.

Tilattomuusperiaatteen mukaan palvelussa ei saa olla sisäistä tilaa, vaan kaikki tilaan liittyvä tieto on välitettävä parametreina palvelulle. Esimerkkisovelluksen palvelut on toteutettu REST-periaatteita (kohta 3.3.1) noudattaen, joten ne ovat tilattomia. Myös Tilkovin selkeät rajat periaate toteutuu REST-arkkitehtuurin noudattamisen myötä.

6.2 Johtopäätökset ja pohdintaa

Puhtaasti ohjelmistoteknisestä näkökulmasta komposiittisovelluksen toteutus vastaa monilta osin normaalin web-sovelluksen toteuttamista. Käyttöliittymäpalveluiden osalta selvä ero perinteisiin, palvelinsovelmilla (Servlet) ja palveli-

mella dynaamisesti generoiduilla sivuilla (Java Server Pages, JSP) toteutettuihin sovelluksiin verrattuna on käyttöliittymän ja palvelinpään selkeä rajanveto. Käyttöliittymäkerrokseen ja sovelluspalvelimella pyörivään toteutukseen liittyviä asioita pohditaan ohjelmoijan näkökulmasta kohdissa 6.2.1 ja 6.2.2. Kohdassa 6.2.3 asiaa lähestytään ohjelmistosuunnittelijan näkökulmasta.

6.2.1 Käyttöliittymäkerroksen toteutuksen arviointi

Esimerkkisovelluksessa palvelimella pyörivät palvelut vastaanottavat ja tarjoavat dataa JSON-muodossa (kohta 3.5.1), jolloin käyttäjän selaimessa näkyvän HTML-rakenteen päivittäminen jää selaimessa pyörivän JavaScript-kielellä kirjoitetun toteutuksen tehtäväksi.

Lähestymistapa soveltuu hyvin prototyyppikeskeiseen ohjelmistokehitykseen, joka mahdollistaa sovelluksen käyttötapausten suunnittelun yksinkertaisen käyttöliittymäprototyypin avulla. Käyttöliittymäprototyyppiin voidaan toteuttaa dynaamisuutta JavaScriptillä ja simuloida sovelluksen todellista käyttöä, vaikka ei edes tiedettäisi, mille ohjelmistoalustalle ja millä kielellä palvelinpään toteutus tullaan toteuttamaan. Käyttöliittymän ja palvelinpuolen selvä erottelu voi helpottaa myös tehtävien jakamista käyttöliittymä- ja palvelintoteutuksen asiantuntijoiden välillä. Tuottavuus luonnollisesti paranee, jos jokainen projektiin osallistuva voi keskittyä siihen osa-alueeseen, jonka tuntee parhaiten.

Perinteisiin palvelinsovelmiin perustuviin sovelluksiin verrattuna palvelimen ja selaimen välinen JSON-rajapinta aiheuttaa myös uudenlaisia haasteita kehittäjille. Arkkitehtuurin suunnittelussa täytyy päättää, tuleeko käyttöliittymätason palveluista selaimessa näkyvä sivun rakenne HTML-merkkauksena vai pysytäänkö puhtaasti JSON-rajapintojen takana, jolloin HTML-osuus luodaan dynaamisesti selaimessa. Esimerkkisovelluksessa päädyttiin ensimmäiseen vaihtoehtoon ja sallittiin korkeimman tason palveluiden palauttaa HTML-merkkausta.

Sivun rakenteen lisäksi olennainen osa käyttöliittymätoteutusta on CSS-tyylit ja selaimessa pyörivä JavaScript-ohjelmistologiikka. Tavallisessa web-sovelluksessa tyylit tulevat sovelluksen mukana, mutta palvelukeskeisessä ympäristössä sidos ei välttämättä ole mahdollinen, koska tyylimääritykset voisivat tulla esimerkiksi erillisestä palvelusta. Tyylien erottelu käyttöliittymän rakenteesta ei ole kuitenkaan teknisen toteutuksen kannalta ongelmallista, koska CSS-tyyleihin viitataan tyypillisesti aina REST-tyylisillä osoitteilla (kohta 3.3.1).

Perinteisessä web-sovelluksessa lisätään usein palvelimella generoituun HTML:ään jonkin verran selaimessa pyörivää JavaScriptiä. Esimerkkisovelluksen tapaan toteutetussa komposiittisovelluksessa selaimessa pyörivällä koodilla toteutetaan huomattava osa sovelluksen liiketoimintalogiikasta, joka luo uudenlaisia haasteita sovelluksen testaukselle ja laadunvarmistukselle. Olennaista osaa toteutuksessa näyttelevätkin nykyaikaiset JavaScript-ohjelmistokehykset, joista esimerkkisovelluksen käyttöön valikoitui jQuery (kohta 5.5.4).

6.2.2 Palvelinpään toteutuksen arviointi

Esimerkkisovelluksessa palvelimella pyörivien palveluiden toteutus näyttää perinteisiä web-sovelluksia Javalla toteuttaneelle hyvinkin tutulta: sovelluksessa on palvelinsovelmia, jotka kutsuvat toisen sovelluskerroksen logiikkaa ja noutavat tarvittavaa tietoa tietolähteistä. Palvelukeskeisten periaatteiden toteutuminen asettaa kuitenkin toteutukselle omat suuntaviivansa.

Yksittäinen löyhästi sidoksissa oleva ja itsenäinen palvelu on toteutukseltaan kohtalaisen yksinkertainen kokonaisuus. Pienten osakokonaisuuksien toteuttaminen yleisesti hyväksyttyjä periaatteita ja teknologioita käyttäen osoittautui erittäin nopeaksi ja tehokkaaksi. Esimerkkisovelluksessa tietolähteen rajapinnaksi valikoitiin Java Persistence Api (kohta 5.5.1) ja tiedonvälitysformaatiksi JSON (kohta 3.5.1), jotka molemmat ovat hyvin suosittuja nykyohjelmistoissa.

Yleisesti käytössä olevien teknologioiden ja periaatteiden noudattaminen mahdollistaa parhaimmillaan koko teknologiapaletin vaihtamisen toiseen. Esimerkkisovelluksessa vaihdettiinkin JavaEE:n oliosäiliöt Spring Frameworkin vastaaviin sekä Hibernaten olio-relaatio-muunnin EclipseLink-muuntimeen. Tietolähteenä käytettiin aluksi ObjectDB-oliotietokantaa, joka vaihdettiin myöhemmin HSQLDB-relaatiotietokannaksi. Mikään edellä mainituista vaihdoksista ei aiheuttanut muutoksia palveluiden varsinaiseen liiketoimintalogiikkaan. Teknologisia vaihdoksia ei ole kuvattu toteutusta käsittelevässä luvussa tutkielman näkökulman rajauksen vuoksi, vaan vaihdokset toteutettiin ainoastaan kirjoittajan omasta mielenkiinnosta.

Toteutusvauhdin osalta palvelulähtöinen pieniin osiin jakaminen osoittautui erinomaiseksi valinnaksi. Suorituskyvyn osalta pienessä esimerkkisovelluksessa ei kohdattu yllätyksiä, mutta suuremmissa järjestelmissä palvelukeskeisyys voi aiheuttaa uudenlaisia suorituskykyhaasteita. Käytetään esimerkkitapauksena alaluvussa 2.3 esiteltyä viisitasoista arkkitehtuuria. Oletetaan, että kutsumme yhtä käyttöliittymäpalvelua, joka kutsuu kahta alemman tason palvelua, joista kumpikin kutsuu edelleen kahta seuraavan tason palvelua ja niin edelleen. Palvelukutsuja tulee viisitasoisessa arkkitehtuurissa 16 kappaletta ja lisäksi saman verran palveluiden vasteita. Ylemmän kerroksen palvelun voisi myös olla tarpeen kutsua silmukan sisällä alemman kerroksen palvelua, jolloin kutsujen määrä voi kasvaa huimasti. Jokaista kutsua ja vastetta varten täytyy tehdä arkkitehtuurista riippuen tiedon muunnosta. Esimerkkisovelluksen palveluissa tehtäisiin tiedon muunnos JSON-tekstistä ohjelmointikielen käyttämään muotoon.

Suorituskykyhaasteiden lisäksi toimintalogiikan toteuttaminen palveluina voi tuoda myös suorituskyvyn parannuksia. Yksinkertaisten palveluiden rajapintoja on helppo testata suurilla kuormilla ja havaita pullonkaulat, jolloin rajallista ja kallista ohjelmointiaikaa voidaan käyttää juuri olennaisimpien palveluiden optimointiin. Löyhästi sidottuja ja itsenäisiä palveluita voidaan myös sijoittaa

suorituskyvyn ja verkkorakenteen mukaan keskeisille palvelimille, joka vähentää pullonkaulojen muodostumista. Jo pitkään yleisessä käytössä olleiden teknologioiden, kuten HTTP-protokollan käyttö mahdollistaa erittäin tehokkaiden välimuistiratkaisujen käyttöönoton. Myös tietolähteen tarjoavassa palvelussa voidaan käyttää välimuistiratkaisuita, jotka helpottavat tietolähteen kuormaa huomattavasti.

6.2.3 Komposiittisovelluksen suunnittelun arviointi

Ohjelmistosuunnittelun hyvät perusperiaatteet, kuten ohjelmiston jakaminen pieniin komponentteihin sekä komponenttien uudelleenkäyttö säilyvät hyvin samanlaisina myös komposiittisovelluksen suunnittelussa. Palvelulähtöinen lähestymistapa kuitenkin nostaa komponenttien ja uudelleenkäytön suunnittelun perinteistä web-sovellusta korkeammalle arkkitehtuurin tasolle. Komposiittisovellusta suunniteltaessa selvitetään perinteisen web-sovelluksen tapaan ensin liiketoiminnalliset tarpeet ja kuvataan ne käyttötapauksina suunnittelun pohjaksi.

Käyttötapausten perusteella suunnitellaan liiketoimintatarpeen täyttämiseksi vaaditut palvelut ja selvitetään, mitä valmiita palveluita ympäröivässä arkkitehtuurissa on saatavilla. Komposiittisovelluksen suunnittelu vaatii hieman perinteisestä web-sovelluksesta poikkeavaa lähestymistapaa, koska tavoitteena on uudelleenkäyttää olemassa olevia palveluita ja toteuttaa ainoastaan ne palvelut, joita ei löydy valmiina. Lähestymistapa kannustaa palveluiden uudelleenkäyttöön ja voi parantaa ohjelmiston suunnitteluun osallistuvien organisaation osien keskinäistä viestintää, koska palvelut voidaan määritellä ja suunnitella pienissä osissa irrallaan sovelluksen varsinaisesta toteutuksesta.

Palveluiden tunnistamisen ja uusien palveluiden suunnittelun myötä sovellusta käyttävässä organisaatiossa nähdään, millaisia palveluita ympäröivässä arkkitehtuurissa on saatavilla ja millaisia uusia palveluita sinne sovelluksen myötä syntyy. Tämä puolestaan mahdollistaa uusien käyttötarkoitusten ideoinnin

olemassa oleville palveluille, jota voidaan ohjelmistosuunnittelun näkökulmasta pitää yhtenä palvelukeskeisen arkkitehtuurin olennaisimmista hyödyistä. Komposiittisovellukset soveltuvatkin hyvin ympäristöön, jossa on paljon verkon yli kutsuttavia rajapintoja.

Palveluiden tunnistaminen on mahdollisesti tärkein yksittäinen vaihe komposiittisovelluksen suunnittelussa, mutta esimerkkisovelluksen osalta se oli myös vaikein vaihe. Tavalliseen web-ohjelmistokehitykseen tottuneelle on haastavaa suunnitella mahdollisimman pieni ja yksinkertainen palvelu, jonka on kuitenkin oltava sen verran monipuolinen, että se täyttää järkevästi jonkin sovelluksen vaatimuksista. Palveluiden yksinkertaisuuden lisäksi varsinkin yksinkertaisten palveluiden ja komposiittipalveluiden luokittelu eri kerroksille oli haastavaa (alaluku 5.4, kuvio 15). Esimerkkisovelluksen osalta ei saatu todellista hyötyä palveluiden kerrosluokittelusta, mutta luokittelu voi olla huomattavasti olennaisempaa laajassa palvelukeskeisessä arkkitehtuurissa.

6.3 Yhteenveto

Tässä luvussa tutkittiin ja arvioitiin esimerkkisovellusta palvelukeskeisen arkkitehtuurin sekä toteutuksen helppouden näkökulmasta. Alaluvussa 6.1 todettiin, että esimerkkisovellus noudattaa pääpiirteissään luvussa 4 esitettyjä palvelukeskeisen arkkitehtuurin perusperiaatteita. Perusperiaatteita käsiteltiin Brownin ym. jaottelun mukaisesti, täydentäen arviointia ristiintaulukoinnin mukaisesti vastaavilla Tilkovin ja Erlin periaatteilla. Joidenkin periaatteiden todettiin olevan olennaisia vasta suuremmassa sovelluksessa, mutta esimerkkisovelluksen toteutus ei estä niiden noudattamista jatkossa.

Alaluvussa 6.2 arvioitiin web-palveluista koostetun komposiittisovelluksen soveltuvuutta web-ohjelmistojen toteuttamiseen ja toteutuksen aikana havaittuja riskejä. Arviointi oli subjektiivista ja perustui kirjoittajan käytännön työelämän kokemukseen sekä alaluvun 6.1 havaintoihin. Käytännön toteutuksen tasolla, ohjelmoijan näkökulmasta, komposiittisovelluksen toteuttamisen ei havaittu

poikkeavan merkittävästi perinteisen web-sovelluksen toteuttamisesta. Muutamia suorituskykyyn sekä toteutuksen ylläpidettävyyteen liittyviä riskejä tunnistettiin, mutta niiden todettiin olevan kohtuullisesti hallittavissa teknisin keinoin.

Merkittävimmät eroavaisuudet perinteiseen web-sovellukseen verrattuna tunnistettiin ohjelmiston suunnitteluvaiheen tehtävissä. Palvelukeskeisen näkökulman todettiin nostavan suunnittelun abstraktiotasoa kauemmaksi teknisestä toteutuksesta ja samalla lähemmäksi sovelluksen käyttötarkoitusta loppukäyttäjän näkökulmasta. Abstraktiotason noston arvioitiin parhaimmillaan avaavan uusia viestintäkanavia sekä parantavan uudelleenkäyttömahdollisuuksia, mutta toisaalta vaativan enemmän suunnitteluaikaa myös loppukäyttäjien organisaatiolta.

7 YHTEENVETO

Tutkielman tavoitteena oli selvittää, mitä palvelukeskeinen arkkitehtuuri, komposiittisovellus ja web-palvelut tarkoittavat sekä toteuttaa palvelukeskeisen arkkitehtuurin palveluita käyttävä komposiittisovellus.

Komposiittisovellusta tutkittiin tunnistamalla eri sovellus- ja palvelutyyppejä ja asemoimalla ne viisitasoiseen palvelukeskeiseen arkkitehtuuriin (luvut 2 ja 3). Palvelukeskeisen arkkitehtuurin teoriaan perehdyttiin tutkimalla kolmen eri määritelmän mukaisia perusperiaatteita (luku 4). Komposiittisovelluksen kannalta olennaisimpia seikkoja selvitettiin ristiintaulukoimalla määritelmien perusperiaatteita, joka auttoi sovelluksen tarvitsemien palveluiden määrittelyssä ja suunnittelussa. Soveltavassa osuudessa (luku 5) kokeiltiin, voidaanko komposiittisovellus toteuttaa ainoastaan palvelukeskeisen arkkitehtuurin tarjoamia palveluita käyttäen.

Toteutusteknologiat valittiin suosituimpien Java-teknologioiden joukosta kirjoittajan käytännön kokemuksen perusteella. Teknologioiden todettiin soveltuvan hyvin palvelukeskeisen arkkitehtuurin mukaisten palveluiden sekä varsinaisen komposiittisovelluksen toteuttamiseen. Komposiittisovelluksen käyttämät palvelut toteutettiin liiallisen monimutkaisuuden välttämiseksi komposiittisovelluksen kanssa samaan sovelluspakettiin. Palveluiden todettiin olevan REST-arkkitehtuurin mukaisia ja käyttävän viestinvälitykseen JSON-

viestiformaattia HTTP-protokollan yli, jonka vuoksi yksittäinen palvelu voitaisiin siirtää eri sovelluspakettiin tai kokonaan toiselle palvelimelle.

Esimerkkisovellusta arvioitiin palvelukeskeisen arkkitehtuurin teorian näkökulmasta (alaluku 6.1) sekä subjektiivisesti suunnittelun ja teknisen toteutuksen osalta (alaluku 6.2). Subjektiivinen arviointi tehtiin kirjoittajan aiemman kokemuksen perusteella. Sovelluksen palveluiden osalta selvitettiin, kuinka hyvin ne noudattavat palvelukeskeisen arkkitehtuurin perusperiaatteita. Lähes kaikkien perusperiaatteiden huomattiin toteutuneen, mutta sovelluksen yksinkertaisuuden vuoksi jotkin perusperiaatteet eivät olleet relevantteja. Epäolennaistenkin perusperiaatteiden osalta todettiin, että valittu toteutustapa ei estä yhdenkään periaatteen noudattamista, vaan vaatii lisäpohdintaa, jos sovellusta tai palveluita laajennetaan myöhemmin.

Teknisestä näkökulmasta komposiittisovelluksen toteutus palveluista koostamalla vaikutti lupaavalta vaihtoehdolta perinteisen web-sovelluksen rinnalle. Toteutusvauhdin arvioitiin olevan vähintään yhtä hyvä, eikä kooditason toteutuksessa havaittu merkittävää eroa perinteisen ja palvelulähtöisen lähestymistavan välillä. Palveluiden itsenäisyydestä johtuva selvä erottelu palvelimella ja selaimessa pyörivän koodin välillä havaittiin selkeimpänä erona perinteiseen web-sovellukseen verrattuna: normaalissa web-sovelluksessa palvelin lähettää selaimelle aina HTML-muotoisen vasteen, mutta esimerkkisovelluksen palveluista vain käyttöliittymäpalvelut palauttavat HTML-merkkausta.

Myös sovelluksen suunnitteluvaiheessa huomattiin selvä ajattelutavan muutos perinteiseen web-sovellukseen verrattuna. Palvelukeskeisen lähestymistavan vuoksi käyttötapauksien vaatimat palvelut suunnitellaan erillään komposiittisovelluksen toteutuksesta, jonka todettiin mahdollistavan entistä asiakaslähtöisemmän suunnitteluvaiheen. Perinteisessä sovelluksessa määritellään vaatimukset ja käyttötapaukset tilaajan kanssa, mutta tekninen suunnittelu on pääasiassa toteuttavan osapuolen tehtävä. Palveluiden suunnittelun todettiin sen sijaan kannustavan vuoropuheluun tilaajan ja toteuttajan välillä, koska sovel-

luksen tarvitsemat palvelut tulevat samalla myös muiden tilaajan omistamien palvelukeskeisessä arkkitehtuurissa toimivien sovellusten käyttöön.

Tutun teknologian käyttö totutusta poikkeavalla tavalla tuo aina haasteita mukanaan, mutta komposiittisovelluksen edut palvelukeskeisessä ympäristössä voidaan arvioida jo esimerkkisovelluksen perusteella haittoja suuremmiksi. Esimerkkisovelluksen tapaisen, erittäin yksinkertaisen sovelluksen toteuttamisen täytyy toki olla helppoa, mutta palveluista koostaminen soveltuu myös laajemman kokonaisuuden toteuttamiseen. Tutkielmassa esitettyä lähestymistapaa on käytetty menestyksekkäästi myös usean sadan henkilötyöpäivän kokoisissa sovelluskokonaisuuksissa. Alaluvussa 6.2 esitetyt johtopäätökset lähestymistavan hyvistä ja huonoista puolista vastaavat myös isommassa sovelluksessa tehtyjä havaintoja.

Esimerkkinä lähestymistavan skaalautuvuudesta erittäin suurta kuormaa käsitteleviin järjestelmiin voisi pitää esimerkiksi IBM:n Sterling Order Management-tilaustenhallintajärjestelmää, jonka parissa kirjoittaja on työskennellyt päivätyössään. Järjestelmä koostuu yli tuhannesta web-palvelusta, sekä niiden päälle rakennetuista web- ja työpöytäkäyttöliittymiä tarjoavista komposiittisovelluksista. Järjestelmällä ohjataan ja seurataan tilausten etenemistä varastonhallinnasta aina tavaroiden luovutushetkeen saakka muun muassa maailman toiseksi suurimman vähittäiskauppaketjun kivijalkaliikkeissä ja verkkokaupassa. (IBM, 2013)

Palvelukeskeisten arkkitehtuurien käyttöönotto kaupallisissa sovellutuksissa jatkuu ja mielenkiintoisia jatkotutkimusaiheita olisi sekä komposiittisovelluksen että palveluiden saralla. Yksi keskeisimpiä ja mielenkiintoisimpia palveluihin liittyviä aiheita olisi palvelun elinkaaren hallinta ja versiointi palvelukeskeisessä arkkitehtuurissa. Komposiittisovelluksen osalta olisi erittäin mielenkiintoista tutkia, soveltuuko lähestymistapa tiedon koostamiseen ja esittämiseen jatkuvasti suosiotaan kasvattavista Big Data-järjestelmistä. Big Data-järjestelmät tarjoavat useimmiten ulkoisen rajapinnan jollain yleisesti käytössä olevalla protokol-

lalla ja viestiformaatilla, jolloin niitä voidaan pitää web-palveluina ja käyttää myös komposiittien muodostamiseen.

LÄHTEET

Arsanjani, A., Zhang, L.-J., Ellis, M., Allam, A., Channabasavaiah, K. 2007. S3: A Service-Oriented Reference Architecture. IEEE IT Professional 9 (3), 10–17.

Battle, R., Benson, E. 2007. Bridging the semantic Web and Web 2.0 with Representational State Transfer (REST). 2008. Web Semantics: Science, Services and Agents on the World Wide Web 6 (1), 61–69.

Brown, A., Johnston, S., Kelly, K. 2002. Using Service-Oriented Architecture and Component-Based Development to Build Web Service Applications. Santa Clara, CA: Rational Software Corporation. <http://www.ibm.com/developerworks/rational/library/content/03July/2000/2169/2169.pdf>

Brown, G., Stam, K. 2009. RiftSaw 2.0-M1 Getting Started Guide. [viitattu 5.3.2013] <http://docs.jboss.com/riftsaw/publish/en-US/pdf/GettingStartedGuide.pdf>

Cerami, E. 2002. Web Services Essentials. Sebastobol: O'Reilly & Associates, Inc.

Cetin, S., Altintas, N. I., Oguztuzun H., Dogru A. H., Tufekci O., Suloglu, S. 2007. Legacy Migration to Service-Oriented Computing with Mashups. Teoksessa The Second International Conference on Software Engineering Advances (ICSEA 2007). IEEE Computer Society, 21.

Chappell, D. 2007. Introducing SCA. Chappell & Associates. [viitattu 4.1.2013] <http://www.davidchappell.com/articles/Introducing_SCA.pdf>

Craig, W. 2011. Spring in Action. Shelter Island: Manning Publications Co.

Davis, J. 2009. Open Source SOA. Greenwich: Manning Publications Co.

de Bruijn, J., Fensel, D., Kerrigan, M., Keller, U., Lausen, H., Scicluna, J. 2008. Modeling Semantic Web Services. Berlin: Springer.

Dubray, J. 2007. Composite software construction. C4Media.

Ecma International. 2011. ECMAScript Language Specification 5.1 Edition. Geneva.

Englander, R. 2002. Java and SOAP. Sebastobol: O'Reilly & Associates, Inc.

Erl, T. 2007. SOA: Principles of Service Design. Upper Saddle River: Prentice Hall.

Fielding, R. 2000. Architectural Styles and the Design of Network-based Software Architectures. Irvine: University of California.

Franklin, J. 2013. Beginning jQuery. New York: Apress.

Gamma, E., Helm, R., Johnson, R., Vlissides, J.1995. Design Patterns: Elements of Reusable Object-Oriented Software. Boston: Addison-Wesley.

Galpin, M. 2008. Creating mashups on the Google App Engine using Eclipse. [viitattu 26.6.2013] <http://www.ibm.com/developerworks/opensource/library/os-eclipse-mashup-google-pt1/>

Goncalves, A. 2013. Beginning Java EE 7. New York: Apress.

Hadley, M. 2006. Web Application Description Language (WADL). Santa Clara: Sun Microsystems, Inc. [viitattu 5.6.2013] <https://wadl.java.net/wadl20061109.pdf>

IBM. 2004. Service-oriented modeling and architecture. [viitattu 5.6.2013] <http://www.ibm.com/developerworks/webservices/library/ws-soa-design1/>

IBM. 2009. WebSphere Enterprise Service Bus. [viitattu 10.7.2013] <http://www-01.ibm.com/software/integration/wsesb/v6/faqs.html>

IBM. 2013. Showcasing IBM client stories: El Corte Ingles. [viitattu 22.4.2014] <http://www-03.ibm.com/software/businesscasestudies/us/en?synkey=P556747H99543H67>

IGI Global. 2009. Encyclopedia of Information Science and Technology. 2009. Hershey: IGI Global.

IETF. 1976. A High-Level Framework for Network-Based Resource Sharing. [viitattu 2.8.2013] <http://tools.ietf.org/html/rfc707>

IETF. 2006. The application/json Media Type for JavaScript Object Notation (JSON) [viitattu 27.10.2013] <http://tools.ietf.org/html/rfc4627>

Java Community Process. 2009. JSR-000208 Java Business Integration 1.0. [viitattu 17.6.2013] <http://jcp.org/aboutJava/communityprocess/final/jsr208/>

Java-Source.net. 2013. Open Source Web Frameworks in Java. [viitattu 1.12.2013] <http://java-source.net/open-source/web-frameworks>

Jayasinghe, D. 2008. Quickstart Apache Axis2. Birmingham: Packt Publishing Ltd.

Jendrock, E. Ball, J. Carson, D. Evans, I. Fordin, S. Haase, K. 2003. The Java EE 5 Tutorial. 2008. Sun Microsystems.

Johnson, R. Hoeller, J. 2004. Expert One-on-One J2EE™ Development without EJB™. Indianapolis: Wiley Publishing, Inc.

Josuttis, N. M. 2007. SOA in Practice. Sebastopol: O'Reilly Media, Inc.

jQuery. 2014. The jQuery Foundation Members. [viitattu 6.2.2014] <https://jquery.org/members/>

Juric, M. B. 2006. Business Process Execution Language for Web Services. Birmingham: Packt Publishing.

Kalin, M. 2009. Java Web Services: Up and Running. Sebastopol: O'Reilly Media, Inc.

Kantorovitch, J., Niemelä, E. 2009. Service Description Ontologies. Teoksessa Encyclopedia of Information Science and Technology, 3445–3451. Hershey: IGI Global.

Kilov, H., Sack, I. 2009. Conceptual Commonalities in Modeling of Business and IT Artifacts. Teoksessa Encyclopedia of Information Science and Technology, 686–690. 2009. Hershey: IGI Global.

Little, M. 2003. Web services transactions: past, present and future. Teoksessa Proceedings of the XML Conference and Exposition 2003. Philadephia.

Maamar, Z. 2009. Design and Development of Communities of Web Services. Teoksessa Encyclopedia of Information Science and Technology, 1024–1029. 2009. Hershey: IGI Global.

Marks, E. A. 2003. Executives Guide To Web Services. Hoboken: John Wiley & Sons, Inc.

McGovern, J., Adatia, R., Fain, Y., Gordon, J., Henry, E., Hurst, W., Jain, A., Little, M., Nagarajan, V., Oak, H., Phillips, L. A. 2003a. Java™ 2 Enterprise Edition 1.4 Bible. Indianapolis: Wiley Publishing, Inc.

McGovern, J., Tyagi, S., Stevens, M., Matthew, S. 2003b. Java Web Services Architecture. San Francisco: Morgan Kaufmann Publishers.

McIlroy, D. 1968. Mass produced software components. Teoksessa Report on a conference sponsored by the NATO Science Committee, Garmisch, Germany, 7th to 11th October 1968, 138–151.

Microsoft. 2009. BizTalk Server ESB Toolkit. [viitattu 10.8.2013] <http://www.microsoft.com/biztalk/en/us/esb-guidance.aspx>

Monson-Haefel, R. 2003. J2EE(TM) Web Services. Boston: Addison-Wesley.

OASIS. 2005. Web Services Coordination Framework Specification (WS-CF) 1.0. <http://www.oasis-open.org/committees/download.php/19658/WS-CF.zip>

OASIS. 2009a. OASIS Contributors. [viitattu 16.10.2013] <http://www.oasis-open.org/about/contributors.php>

OASIS. 2009b. OASIS Sponsors. [viitattu 16.10.2013] <http://www.oasis-open.org/about/index.php>

OASIS. 2009c. Service Component Architecture. [viitattu 16.10.2013] <http://www.oasis-opencsa.org/sca>

OASIS. 2009d. Web Services Coordination (WS-Coordination) Version 1.2. [viitattu 20.10.2013] <http://www.oasis-open.org/committees/download.php/35161/wstx-wscoor-1%202-spec-os.pdf>

Object Management Group. 2009. Unified Modeling Language (OMG UML) Infrastructure 2.2. [viitattu 12.9.2013]<http://www.omg.org/spec/UML/2.2/Infrastructure>

OSOA. 2007. JAX-WS Services Integration. [viitattu 12.11.2013] <http://www.osoa.org/display/Main/JAX-WS+Services+Integration>

Open ESB. 2009. [viitattu 8.10.2013] <https://open-esb.dev.java.net/>

Open Mashup Alliance. 2009. OMA EMML 1.0 Documentation [viitattu 26.8.2013] <http://www.openmashup.org/omadocs/v1.0/index.html>

Oracle. 2004. JavaTM 2 Platform Standard Ed. 5.0. [viitattu 15.12.2013] <http://docs.oracle.com/javase/1.5.0/docs/api/>

Pautasso, C., Zimmermann, O., Leymann, F. 2008. RESTful Web Services vs. "Big" Web Services: Making the Right Architectural Decision. Teoksessa Proceedings of the 17th international conference on World Wide Web, 805–814. New York: ACM.

Pehlivanian, A., Nguyen, D. 2013. Jump Start JavaScript. Collingwood: SitePoint Pty. Ltd.

Poutsma, A. Evans, R. Rabbo, A. T. 2007. Spring Web Services - Reference Documentation 1.5.6 [Viitattu 4.12.2013] <http://static.springframework.org/spring-ws/sites/1.5/reference/pdf/spring-ws-reference.pdf>

Richardson, L., Ruby, S. 2007. RESTful Web Services. Sebastopol: O'Reilly Media, Inc.

Rosen, M., Lublisnky, B., Smith, K. T., Balcer, M. J. 2008. Applied SOA: Service-Oriented Architecture and Design Strategies. Indianapolis: Wiley Publishing, Inc.

Taylor, I., Harrison, A. 2009. From P2P and Grids to Services on the Web: Evolving Distributed Communities. London: Springer-Verlag.

The Apache Software Foundation. 2009. Apache ServiceMix. [viitattu 8.12.2012] <http://servicemix.apache.org/>

Tidwell, D. Snell, J., Kulchenko, P. 2001. Programming Web Services with SOAP. Sebastopol: O'Reilly & Associates, Inc.

Tilkov, S. 2007. 10 Principles of SOA. [viitattu 7.9.2013] <http://www.infoq.com/articles/tilkov-10-soa-principles>

Vasiliev, Y. 2007. SOA and WS-BPEL. Birmingham: Packt Publishing.

World Wide Web Consortium. 2003. Web Services Description Language (WSDL) Version 1.2. [viitattu 9.11.2013] <http://www.w3.org/TR/wsdl12>

World Wide Web Consortium. 2004. Web Services Architecture. [viitattu 10.11.2012] <http://www.w3.org/TR/2004/NOTE-ws-arch-20040211/>

World Wide Web Consortium. 2007. Web Services Description Language (WSDL) Version 2.0. [viitattu 12.11.2013] <http://www.w3.org/TR/2007/REC-wsdl20-20070626>

Yee, R. 2008. Pro Web 2.0 Mashups: Remixing Data and Web Services. New York: Springer-Verlag.

LIITE 1 KÄYTTÖLIITTYMÄPALVELUIDEN LÄHDEKOODI

Java-lähdekoodi: UiServices.java

```
package monitor.services;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.RequestParam;

/**
 * Contains all UI services.
 */
@Controller
@RequestMapping("/ui")
public class UiServices {

    /**
     * Show current statuses
     *
     * @return the name of the JSP page
     */
    @RequestMapping(value = "/statuses", method = RequestMethod.GET)
    public String currentStatuses() {
        return "statuses";
    }

    /**
     * Show manage page
     *
     * @return the name of the JSP page
     */
    @RequestMapping(value = "/manage", method = RequestMethod.GET)
    public String dashboard() {
        return "manage";
    }

    /**
     * Show tabbedInterface
     *
     * @return the name of the JSP page
     */
    @RequestMapping(value = "/tabbed", method = RequestMethod.GET)
    public String tabbedInterface() {
        return "tabbed";
    }

    /**
     * Show target
     *
```

```
     * @return the name of the JSP page
     */
    @RequestMapping(value = "/target", method = RequestMethod.GET)
    public String target(@RequestParam(value = "id") Integer targetId) {
        return "target";
    }
}
```

JSP-lähdekoodi: tabbed.jsp

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
  <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
  <title>Monitorointisovellus</title>
  <!-- Javascript resources -->
  <script type="text/javascript" src="/monitor/resources/js/jquery-
1.9.1.js"></script>
  <script type="text/javascript" src="/monitor/resources/js/jquery-ui-
1.10.2.custom.js"></script>
  <script type="text/javascript"
src="/monitor/resources/js/form2js/form2js.js"></script>
  <script type="text/javascript"
src="/monitor/resources/js/form2js/js2form.js"></script>
  <script type="text/javascript"
src="/monitor/resources/js/form2js/jquery.toObject.js"></script>
  <script type="text/javascript"
src="/monitor/resources/js/monitor.js"></script>
  <!-- Stylesheets -->
  <link rel="stylesheet" href="/monitor/resources/css/jquery-ui-
1.10.2.custom.css" />
  <link rel="stylesheet" href="/monitor/resources/css/monitor.css" />
</head>
<body>
  <div id="tabs">
    <ul>
      <li id="tab1"><a href="#tabs-1">Kohteiden hallinta</a></li>
      <li id="tab2"><a href="#tabs-2">Tilojen listaus</a></li>
    </ul>
    <div id="tabs-1"></div>
    <div id="tabs-2"></div>
  </div>
  <div id="selectedTarget" data-selected-target-id=""></div>
</body>
<script type="text/javascript">
  $(function() {
    $("#tabs").tabs();

    $.get("/monitor/ui/manage", function (response) {
          $("#tabs-1").html(response);
        });

    $("#tab1").click(function(){
          $("#tabs-2").html("");
```

```
      $.get("/monitor/ui/manage", function (response) {
            $("#tabs-1").html(response);
          });
    });

    $("#tab2").click(function(){
          $("#tabs-1").html("");
      $.get("/monitor/ui/statuses", function (response) {
            $("#tabs-2").html(response);
          });
    });

  });
</script>
</html>
```

JSP-lähdekoodi: manage.jsp

```
<h2>Lisää kohde</h2>
<form id="newTarget">
  <fieldset class="infobox">
    <p>
      <label for="targetName">Nimi: </label>
      <input id="targetName" type="text" name="name" />
    </p>
    <p>
      <label for="smokeUrl">Osoite: </label>
      <input id="smokeUrl" type="text" name="smokeTestUrl.url" />
    </p>
    <p>
      <label for="testProtocol">Protokolla: </label>
      <select id="testProtocol" name="smokeTestUrl.testProtocol">
        <option value="HTTP_GET">HTTP (GET)</option>
        <option value="HTTP_POST">HTTP (POST)</option>
      </select>
    </p>
    <button id="addTarget">Lisää</button>
  </fieldset>
</form>

<h2>Valvonnan kohteet</h2>
<table class="contentList" id="targetsList">
  <thead>
    <tr><th>Nimi</th><th>Osoite</th><th>Protokolla</th><th></th></tr>
  </thead>
  <tbody>
    <!-- ajax filled -->
  </tbody>
</table>

<div id="manageTargetInfoPlaceholder"></div>



<script type="text/javascript">
  $(function() {
    // Declared functions
```

```
    function refreshTargetsList() {
      var $tbody = $("#targetsList").find("tbody");
      $tbody.empty();
      $.get("/monitor/lowlevel/targets/list", function(response) {
        for ( var i = 0; i < response.length; i++) {
          var monitoredTarget = response[i];
          var $row = $("<tr>");
          var $nameCol = $("<td>");
          var $nameLink = $("<a>").attr("href",
                      "/monitor/ui/target?id=" + monitoredTarget.id)
                      .attr("data-selected-target-id", monitoredTarget.id)
                      .html(monitoredTarget.name);
          $nameLink.on("click", function(event) {
            event.preventDefault();
            $("#selectedTarget").attr("data-selected-target-id",
                    $(this).attr("data-selected-target-id"));
            $.get($(this).attr("href"), function(response) {
              $("#manageTargetInfoPlaceholder").html(response);
            });
          });
          var $smokeCol = $("<td>").html(
                      monitoredTarget.smokeTestUrl.url);
          var $protocolCol = $("<td>").html(
                      monitoredTarget.smokeTestUrl.testProtocol ==
                            "HTTP_GET" ? "HTTP (GET)" : "HTTP (POST)");
          var $delCol = $("<td>");
          var $delLink = $("<a>").attr("href",
                      "/monitor/lowlevel/targets/delete?id=" +
                                    monitoredTarget.id);
          $delLink.html("poista").click(function(event) {
            event.preventDefault();
            $.post($(this).attr("href"), function() {
              refreshTargetsList();
            });
          });
          $tbody.append($row.append($nameCol.append($nameLink))
                      .append($smokeCol)
                      .append($protocolCol)
                      .append($delCol.append($delLink)));
        }
      });
    }

    // Event handlers as anonymous functions
    $("#addTarget").click(function(event) {
      event.preventDefault();
      var newTargetJson = $("#newTarget").toObject();
      $.postJSON("lowlevel/targets/add", newTargetJson, function(response) {
        $.get("/monitor/ui/manage", function (response) {
              $("#tabs-1").html(response);
            });
      });
    });

    // On load function calls
    refreshTargetsList();
  });
</script>
```

JSP-lähdekoodi: statuses.jsp

```
<h2>Kohteiden tilat</h2>
<table class="contentList" id="compositeList">
  <thead>
    <tr><th>Nimi</th><th>Tila</th><th></th></tr>
  </thead>
  <tbody>
    <!-- ajax filled -->
  </tbody>
</table>

<div id="statusTargetInfoPlaceholder"></div>

<script type="text/javascript">
  $(function() {
    var $tbody = $("#compositeList").find("tbody");
    $.get("/monitor/composite/targets/statuses", function(response) {
      $tbody.empty();
      for ( var i = 0; i < response.length; i++) {
        var monitoredTargetStatus = response[i];
        var $row = $("<tr>");
        var $nameCol = $("<td>");
        var $nameLink = $("<a>").attr("href", "/monitor/ui/target?id=" +
                    monitoredTargetStatus.monitoredTarget.id)
                    .attr("data-selected-target-id", monitoredTargetSta-
tus.monitoredTarget.id)
                    .html(monitoredTargetStatus.monitoredTarget.name);
                $nameLink.on("click", function(event) {
              event.preventDefault();
            $("#selectedTarget").attr("data-selected-target-id",
$(this).attr("data-selected-target-id"));
                $.get($(this).attr("href"), function(response) {
                      $("#statusTargetInfoPlaceholder").html(response);
                });
                });
        var $smokeCol = $("<td>").html(monitoredTargetStatus.isResponsive ?
"vastaa" : "ei vastaa");
        monitoredTargetStatus.isResponsive ? $smokeCol.addClass("success") :
$smokeCol.addClass("fail");
        $tbo-
dy.append($row.append($nameCol.append($nameLink)).append($smokeCol));
      }
    });
  });
</script>
```

JSP-lähdekoodi: target.jsp

```
<h2>Perustiedot</h2>
<fieldset class="infobox">
  <p><span>Nimi: </span><span id="targetNameInfo"></span></p>
```

```
  <p><span>Osoite: </span><span id="smokeUrlInfo"></span></p>
  <p><span>Protokolla: </span><span id="testProtocolInfo"></span></p>
</fieldset>

<h2>Ajetut testit</h2>
<table class="contentList" id="testResultsList">
  <thead>
    <tr><th>Päivämäärä</th><th>Kellonaika</th><th>Tulos</th></tr>
  </thead>
  <tbody>
    <!-- ajax filled -->
  </tbody>
</table>

<script type="text/javascript">

  $(function() {

    var targetId = $("#selectedTarget").attr("data-selected-target-id");

    $.get("/monitor/lowlevel/target?id=" + targetId, functi-
on(monitoredTarget) {
      $("#targetNameInfo").html(monitoredTarget.name);
      $("#smokeUrlInfo").html(monitoredTarget.smokeTestUrl.url);
      $("#testProtocolInfo").html(monitoredTarget.smokeTestUrl.testProtocol
== "HTTP_GET" ? "HTTP (GET)" : "HTTP (POST)");

      var $tbody = $("#testResultsList").find("tbody");
      $.get("/monitor/lowlevel/target/results/list?id=" + monitoredTarget.id,
function(response) {
        $tbody.empty();
        if (response.length == 0) {
          $tbody.append($("<tr>").append($("<td>").html("Ei ajettuja teste-
jä").addClass("success")));
        } else {
          for ( var i = 0; i < response.length; i++) {
            var result = response[i];
            var resultTimestamp = new Date(result.created);
            var $row = $("<tr>");
            var $dateCol = $("<td>").html($.datepicker.formatDate('dd.mm.yy',
resultTimestamp));
            var $timeCol =
$("<td>").html($.formatZero(resultTimestamp.getHours()) + ":" +
                  $.formatZero(resultTimestamp.getMinutes()) + ":" +
                  $.formatZero(resultTimestamp.getSeconds()));
            var $resultCol = $("<td>");
            if (result.error) {
              $resultCol.html(result.error).addClass("fail");
            } else {
              $resultCol.html("ok").addClass("success");
            }
            $tbo-
dy.append($row.append($dateCol).append($timeCol).append($resultCol));
          }
        }
      });
    });
  });
</script>
```

LIITE 2 KOMPOSIITTIPALVELUIDEN LÄHDEKOODI

Java-lähdekoodi: CompositeServices.java

```
package monitor.services;

import java.util.LinkedList;
import java.util.List;

import monitor.entity.MonitoredTarget;
import monitor.response.GeneralResponse;
import monitor.response.MonitoredTargetStatus;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.ResponseBody;

/**
 * Contains all composite services.
 */
@Controller
public class CompositeServices {

    @Autowired
    private LowLevelServices lowLevelServices;

    /**
     * Uses multiple low level services and combines results from them.
     *
     * @return List of statuses of all monitored targets in the application.
     */
    @RequestMapping(value = "/composite/targets/statuses", method = Request-
Method.GET)
    public @ResponseBody
    List<MonitoredTargetStatus> statuses() {
        List<MonitoredTargetStatus> monitoredTargetStatuses = new Linked-
List<MonitoredTargetStatus>();
        List<MonitoredTarget> monitoredTargets = lowLevelServi-
ces.listMonitoredTargets();
        for (MonitoredTarget monitoredTarget : monitoredTargets) {
            MonitoredTargetStatus monitoredTargetStatus = new MonitoredTar-
getStatus();
            monitoredTargetSta-
tus.setTestResults(lowLevelServices.listTestResultsForTarget(monitoredTarget.
getId()));
            monitoredTargetStatus

.setIsResponsive(lowLevelServices.runSmokeTest(monitoredTarget) == General-
Response.SUCCESS);
            monitoredTargetStatus.setMonitoredTarget(monitoredTarget);
            monitoredTargetStatuses.add(monitoredTargetStatus);
```

```
            }
            return monitoredTargetStatuses;
        }
    }
```

LIITE 3 MATALAN TASON PALVELUIDEN LÄHDEKOODI

Java-lähdekoodi: LowLevelServices.java

```
package monitor.services;

import java.util.List;

import monitor.entity.MonitoredTarget;
import monitor.entity.TestResult;
import monitor.helper.HttpRunner;
import monitor.response.GeneralResponse;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.transaction.annotation.Transactional;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.ResponseBody;

/**
 * Contains all low level services.
 */
@Controller
public class LowLevelServices {

    @Autowired
    private DataAccess dataAccess;

    @Autowired
    private HttpRunner httpRunner;

    /**
     * Adds a new target by delegating the processing to DataAccess.
     */
    @RequestMapping(value = "/lowlevel/targets/add", method = RequestMet-
hod.POST)
    public @ResponseBody
    GeneralResponse add(@RequestBody MonitoredTarget monitoredTarget) {
        dataAccess.add(monitoredTarget);
        return GeneralResponse.SUCCESS;
    }

    /**
     * Deletes the history of run results of the monitored target.
     *
     * Deletes the monitored target.
     */
```

```
    @RequestMapping(value = "/lowlevel/targets/delete", method = RequestMet-
hod.POST)
    @Transactional
    public @ResponseBody
    GeneralResponse deleteMonitoredTarget(@RequestParam Long id) {
        dataAccess.deleteResultsForMonitoredTarget(id);
        dataAccess.deleteMonitoredTarget(id);
        return GeneralResponse.SUCCESS;
    }

    /**
     * Retrieves all targets.
     */
    @RequestMapping(value = "/lowlevel/targets/list", method = RequestMet-
hod.GET)
    public @ResponseBody
    List<MonitoredTarget> listMonitoredTargets() {
        List<MonitoredTarget> monitoredTargets = dataAc-
cess.listMonitoredTargets();
        return monitoredTargets;
    }

    /**
     * Lists results for target.
     */
    @RequestMapping(value = "/lowlevel/target/results/list", method = Re-
questMethod.GET)
    public @ResponseBody
    List<TestResult> listTestResultsForTarget(@RequestParam Long id) {
        MonitoredTarget target = dataAccess.loadMonitoredTarget(id);
        List<TestResult> results = dataAccess.listTestResults(target);
        return results;
    }

    /**
     * Retrieves all targets.
     */
    @RequestMapping(value = "/lowlevel/target", method = RequestMethod.GET)
    public @ResponseBody
    MonitoredTarget loadMonitoredTarget(@RequestParam Long id) {
        MonitoredTarget monitoredTarget = dataAccess.loadMonitoredTarget(id);
        return monitoredTarget;
    }

    /**
     * Runs smoke test for target.
     */
    @RequestMapping(value = "/lowlevel/targets/smoke", method = RequestMet-
hod.GET)
    public @ResponseBody
    GeneralResponse runSmokeTest(MonitoredTarget monitoredTarget) {
        GeneralResponse status = GeneralResponse.SUCCESS;
        String error = httpRunner.runSafe(monitoredTarget.getSmokeTestUrl());
        TestResult testResult = new TestResult();
        testResult.setMonitoredTarget(monitoredTarget);
        if (error == null) {
            status = GeneralResponse.SUCCESS;
        } else {
            status = GeneralResponse.FAIL;
```

```
            testResult.setError(error);
        }
        dataAccess.add(testResult);
        return status;
    }
}
```

LIITE 4 TIETOLÄHDEPALVELUIDEN LÄHDEKOODI

Java-lähdekoodi: DataAccess.java

```
package monitor.services;

import java.util.List;

import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;
import javax.persistence.Query;
import javax.persistence.TypedQuery;

import monitor.entity.MonitorApplicationEntity;
import monitor.entity.MonitoredTarget;
import monitor.entity.TestResult;

import org.springframework.stereotype.Component;
import org.springframework.transaction.annotation.Transactional;

/**
 * Provides data access for saving, updating and removing Monitor application
 * domain objects.
 */
@Component
public class DataAccess {

    @PersistenceContext
    private EntityManager entityManager;

    /**
     * Adds a new monitored target
     *
     */
    @Transactional
    public void add(MonitorApplicationEntity entity) {
        entityManager.persist(entity);
    }

    /**
     * Deletes an existing monitored target
     *
     * @param id
     *            the monitoredTarget to remove
     */
    @Transactional
    public void deleteMonitoredTarget(Long id) {
        Query deleteQuery = entityManager.createQuery("delete from Monito-
redTarget mt where mt.id = :id");
        deleteQuery.setParameter("id", id);
        deleteQuery.executeUpdate();
    }

    /**
```

```
     * Deletes all test results of one monitored target
     *
     * @param id
     *            the monitoredTarget which tests results are to be removed
     */
    public void deleteResultsForMonitoredTarget(Long id) {
        Query deleteQuery = entityManager.createQuery("delete from TestResult
tr where tr.monitoredTarget.id = :id");
        deleteQuery.setParameter("id", id);
        deleteQuery.executeUpdate();
    }

    /**
     * Retrieves all monitored targets
     *
     * @return a list of targets
     */
    public List<MonitoredTarget> listMonitoredTargets() {
        TypedQuery<MonitoredTarget> query = entityManager.createQuery("select
mt from MonitoredTarget mt",
                MonitoredTarget.class);
        return query.getResultList();
    }

    /**
     * Lists results of monitored target
     *
     * @param monitoredTarget
     * @return the results for targer
     */
    public List<TestResult> listTestResults(MonitoredTarget monitoredTarget)
{
        TypedQuery<TestResult> query = entityManager.createQuery(
                "select testResult from TestResult testResult where test-
Result.monitoredTarget = :monitoredTarget",
                TestResult.class);
        query.setParameter("monitoredTarget", monitoredTarget);
        return query.getResultList();
    }

    /**
     * Loads monitored target from database
     *
     * @param id
     *            the id of the monitored target
     * @return the target
     */
    public MonitoredTarget loadMonitoredTarget(Long id) {
        return entityManager.find(MonitoredTarget.class, id);
    }
}
```

LIITE 5 TIETOMALLILUOKKIEN LÄHDEKOODI

Java-lähdekoodi: MonitorApplicationEntity.java

```
package monitor.entity;

import java.io.Serializable;
import java.util.Date;

import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.MappedSuperclass;
import javax.persistence.PrePersist;
import javax.persistence.PreUpdate;
import javax.persistence.Temporal;
import javax.persistence.TemporalType;

/**
 * Superclass for all JPA-entities in the application.
 *
 */
@MappedSuperclass
public class MonitorApplicationEntity implements Serializable {

    private static final long serialVersionUID = 1L;

    @Temporal(TemporalType.TIMESTAMP)
    private Date created;

    @Id
    @GeneratedValue
    private Long id;

    @Temporal(TemporalType.TIMESTAMP)
    private Date updated;

    public Date getCreated() {
        return created;
    }

    public Long getId() {
        return id;
    }

    public Date getUpdated() {
        return updated;
    }

    @PrePersist
    protected void onPrePersist() {
        created = new Date();
    }
```

```
    @PreUpdate
    protected void onPreUpdate() {
        updated = new Date();
    }

    public void setId(Long id) {
        this.id = id;
    }

}
```

Java-lähdekoodi: MonitoredTarget.java

```
package monitor.entity;

import javax.persistence.CascadeType;
import javax.persistence.Entity;
import javax.persistence.OneToOne;

/**
 * JPA entity, which represents one monitored target.
 */
@Entity
public class MonitoredTarget extends MonitorApplicationEntity {

    private static final long serialVersionUID = 1L;

    private String name;

    @OneToOne(cascade = CascadeType.ALL)
    private MonitoredTargetUrl smokeTestUrl;

    public String getName() {
        return name;
    }

    public MonitoredTargetUrl getSmokeTestUrl() {
        return smokeTestUrl;
    }

    public void setName(String name) {
        this.name = name;
    }

    public void setSmokeTestUrl(MonitoredTargetUrl smokeTestUrl) {
        this.smokeTestUrl = smokeTestUrl;
    }

}
```

Java-lähdekoodi: MonitoredTargetUrl.java

```
package monitor.entity;

import javax.persistence.Entity;
import javax.persistence.EnumType;
import javax.persistence.Enumerated;

/**
 * JPA entity, which represents one url used to monitor target.
 */
@Entity
public class MonitoredTargetUrl extends MonitorApplicationEntity {

    private static final long serialVersionUID = 1L;

    @Enumerated(EnumType.STRING)
    private TestProtocol testProtocol;

    private String url;

    public TestProtocol getTestProtocol() {
        return testProtocol;
    }

    public String getUrl() {
        return url;
    }

    public void setTestProtocol(TestProtocol testProtocol) {
        this.testProtocol = testProtocol;
    }

    public void setUrl(String url) {
        this.url = url;
    }

}
```

Java-lähdekoodi: TestProtocol.java

```
package monitor.entity;

/**
 * Defines the supported protocols.
 */
public enum TestProtocol {

    HTTP_GET("HTTP (GET)"), HTTP_POST("HTTP (POST)");

    private String label;

    private TestProtocol(String label) {
        this.label = label;
    }

    public String getLabel() {
```

```
        return label;
    }
}
```

Java-lähdekoodi: TestResult.java

```
package monitor.entity;

import javax.persistence.Entity;
import javax.persistence.JoinColumn;
import javax.persistence.ManyToOne;

/**
 * JPA entity which represents one test result of monitored target test.
 */
@Entity
public class TestResult extends MonitorApplicationEntity {

    private static final long serialVersionUID = 1L;

    private String error;

    @ManyToOne
    @JoinColumn(name = "MONITORED_TARGET_ID")
    private MonitoredTarget monitoredTarget;

    public String getError() {
        return error;
    }

    public MonitoredTarget getMonitoredTarget() {
        return monitoredTarget;
    }

    public void setError(String error) {
        this.error = error;
    }

    public void setMonitoredTarget(MonitoredTarget monitoredTarget) {
        this.monitoredTarget = monitoredTarget;
    }
}
```

LIITE 6 TYÖKALULUOKKIEN LÄHDEKOODI

Java-lähdekoodi: MonitorRuntimeException.java

```
package monitor.exception;

/**
 * General runtime exception for all errors in application.
 *
 */
public class MonitorRuntimeException extends RuntimeException {

    private static final long serialVersionUID = 1L;

}
```

Java-lähdekoodi: HttpRunner.java

```
package monitor.helper;

import monitor.entity.MonitoredTargetUrl;
import monitor.entity.TestProtocol;

import org.apache.http.client.HttpClient;
import org.apache.http.client.ResponseHandler;
import org.apache.http.client.methods.HttpGet;
import org.apache.http.client.methods.HttpPost;
import org.apache.http.client.methods.HttpUriRequest;
import org.apache.http.impl.client.BasicResponseHandler;
import org.apache.http.impl.client.DefaultHttpClient;
import org.springframework.stereotype.Component;

/**
 * Helper class which creates one http request.
 */
@Component
public class HttpRunner {

    /**
     * Runs http request in safe manner (does not throw exceptions).
     *
     * @param monitoredTargetUrl the url for http request.
     * @return If call is errorenous, returns the error string. Otherwise re-
turns null.
     */
    public String runSafe(MonitoredTargetUrl monitoredTargetUrl) {
        String error = null;
        HttpClient httpclient = new DefaultHttpClient();
        try {
```

```
            HttpUriRequest httpRequest;
            if (monitoredTargetUrl.getTestProtocol() == TestProto-
col.HTTP_GET) {
                httpRequest = new HttpGet(monitoredTargetUrl.getUrl());
            } else {
                httpRequest = new HttpPost(monitoredTargetUrl.getUrl());

            }
            ResponseHandler<String> responseHandler = new BasicResponseHan-
dler();
            httpclient.execute(httpRequest, responseHandler);
        } catch (Exception e) {
            error = e.getMessage();
        } finally {
            httpclient.getConnectionManager().shutdown();
        }
        return error;
    }
}
```

Java-lähdekoodi: GeneralResponse.java

```
package monitor.response;

/**
 * General response.
 */
public enum GeneralResponse {
    FAIL, SUCCESS
}
```

Java-lähdekoodi: MonitoredTargetStatus.java

```
package monitor.response;

import java.util.List;

import monitor.entity.MonitoredTarget;
import monitor.entity.TestResult;

/**
 * Composite response object
 */
public class MonitoredTargetStatus {

    private Boolean isResponsive;

    private MonitoredTarget monitoredTarget;

    private List<TestResult> testResults;

    public Boolean getIsResponsive() {
```

```
        return isResponsive;
    }

    public MonitoredTarget getMonitoredTarget() {
        return monitoredTarget;
    }

    public List<TestResult> getTestResults() {
        return testResults;
    }

    public void setIsResponsive(Boolean isResponsive) {
        this.isResponsive = isResponsive;
    }

    public void setMonitoredTarget(MonitoredTarget monitoredTarget) {
        this.monitoredTarget = monitoredTarget;
    }

    public void setTestResults(List<TestResult> testResults) {
        this.testResults = testResults;
    }

}
```