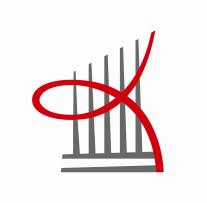


# TIKETTIJÄRJESTELMÄN PROTOTYYPIN KEHITYS GRAILS-SOVELLUSKEHYKSELLÄ

Joel Peltonen

Opinnäytetyö
Marraskuu 2011
Tietojenkäsittelyn koulutusohjelma
Ohjelmistotuotannon
suuntautumisvaihtoehto
Tampereen ammattikorkeakoulu

TAMPEREEN AMMATTIKORKEAKOULU
Tampere University of Applied Sciences

TIIVISTELMÄ

Tampereen ammattikorkeakoulu
Tietojenkäsittelyn koulutusohjelma
Ohjelmistotuotannon suuntautumisvaihtoehto

PELTONEN, JOEL: Tikettijärjestelmän prototyypin kehitys Grails-sovelluskehyksellä.

Opinnäytetyö 49 sivua, liitteet 1 sivu
Marraskuu 2011

Opinnäytetyö käsittelee moniin sovelluksiin lisättävän tikettijärjestelmän kehittämistä. Työ on kirjoitettu lukijalle, joka on kiinnostunut esimerkiksi Grailsin hyödyntämisestä omassa projektissaan ja jolle ohjelmistokehitys on ennalta tuttua. Työn toimeksiantajana oli Symmetria Software Oy. Tavoitteena oli kehittää prototyyppi tikettijärjestelmästä mahdollistamaan tikettijärjestelmän liittäminen mihin tahansa sovellukseen. Järjestelmäuudistuksen yksi kantavista ajatuksista oli eri toteutuksissa tehtävien ratkaisujen siirtäminen muihin ympäristöihin. Haluttu uudistus saavutettaisiin toteuttamalla järjestelmään joustavat ja tehokkaat, mutta kevyet rajapinnat, jotka mahdollistavat ratkaisujen siirtämisen. Aiempia järjestelmiä oli toteutettu yksittäisten ympäristöjen toiminnallisuuteen mukautetusti, mikä mahdollisti tehokkaat räätälöidyt ratkaisut, mutta joiden siirtäminen toisiin järjestelmiin ei olisi ollut nopeasti toteutettavissa. Ratkaisuna päätettiin kehittää kokonaan uusi räätälöitävissä oleva, mutta myös nopeasti siirrettävä ja keskitetysti kehitettävä järjestelmä. Opinnäytetyössä prototyypillä tarkoitetaan järjestelmän ensimmäistä versiota, jolla kokeillaan konseptin toimivuutta ja järjestelmän valmistettavuutta jatkokehityksen avulla.

Tikettijärjestelmä toteutettiin itsenäisenä projektina käyttäen Grails-sovelluskehystä. Grails on ohjelmoitu Groovy-ohjelmointikielellä, joka puolestaan pohjautuu Java-ohjelmointikieleen. Järjestelmä toteutettiin kahdelle palvelimelle, jotka keskustelevat keskenään käyttäen tiedonvälitykseen JavaScript Object Notation -kuvauskieltä ja Simple Object Access Protocol -tekniikkaa. Tikettijärjestelmä pystyy myös vastaanottamaan ja lähettämään sähköpostia.

Tikettijärjestelmän kehitys onnistui asetettujen vaatimusten mukaisesti ja kehityksen aikana saatiin koottua useita jatkokehitysehdotuksia. Toteutettu prototyyppi ei ole valmis tuotantokäyttöön, mutta jatkokehitykselle se on hyvä lähtökohta. Tärkein askel jatkokehityksessä on tikettijärjestelmän liittäminen todelliseen projektiin, jotta järjestelmän jokaista osa-aluetta saadaan testattua käytännössä.

Asiasanat: tikettijärjestelmä, tiketöinti, Grails

ABSTRACT

Tampere University of Applied Sciences
Degree Programme in Business Information Systems
Option of Software Development

PELTONEN, JOEL: Development of a Ticket System Prototype Using the Grails Framework.

Bachelor's thesis 49 pages, appedix 1 page
November 2011

The aim of this thesis is to describe the development of a ticketing system that can be integrated to multiple applications. The thesis is directed to a reader who is interested in utilizing the Grails framework on their own productions and who is already versed in software development. This thesis was commissioned by Symmetria Software Ltd. The objective of this thesis was to develop a prototype of a ticketing system to enable appending ticketing functionality to any application. One of the leading reasons for the ticketing system renewal was to enable transferring different ticketing solutions to other environments. The desired upgrade would be achieved by creating flexible and efficient, yet light interfaces that would enable a portable ticketing solution. Earlier ticketing systems were created specifically for the environments that they operated in, which did enable powerful customized solutions, but ones that would not be portable quickly. The solution to these demands was the creation of a new system that would be customizable, portable and centrally maintained. This thesis describes prototyping the system, meaning the creation of the first version of the system to test the feasibility the concept and how possible the system building would be with further development.

The ticket system development was done as an independent project using the Grails framework. Grails is a web framework for the Groovy programming language, which in turn has its roots in Java. The system was created for two servers, which communicate utilizing the JavaScript Object Notation language and Simple Object Access Protocol. The ticketing system is also able to send and receive email.

The system development succeeded in meeting the requirements set for it and several suggestions for further development were found during development. The prototype is not ready for production use, but it is a good basis for further development. The most important step in further developing the system is integrating the ticketing system to an real application project to test all aspects of the ticketing functionality with real data and real integration experience.

Index terms: ticket system, ticketing, Grails

# LYHENTEET JA TERMIT

| | |
|---|---|
| Etupalvelin | Engl. Front-end server. Tikettijärjestelmän palvelin, jolla käyttöliittymä. |
| GORM | Grails Object Relational Mapping. Grailsin tietokantarajapinta. |
| GSP | Groovy Server Pages. Grails-sovelluksen näkymiä muodostava tekniikka. |
| Huomautusmerkintä | Engl. Annotation. Ohjelman sisällä oleva metatietomerkintä, jota voidaan käyttää esimerkiksi tiedon tarjoamiseen. |
| JSON | JavaScript Object Notation. Standardoitu tapa esittää tietoa merkkijonona. |
| JVM | Java Virtual Machine. Java-ohjelmointikielen ja konekielen välinen virtuaalinen kone, joka suorittaa Javan tavukoodia. |
| Liitännäisvarasto | Engl. repository. Verkossa sijaitseva keskitetty varasto Grails-liitännäisille. |
| Malli | Engl. template. Runko, jonka sisään voidaan rakentaa käyttöliittymä. |
| Metaluokka | Engl. meta class. Luokka, jonka instanssit ovat toisia luokkia. Luokan attribuutti, jolla voidaan viitata luokkaan itseensä. |
| MIME | Multipurpose Internet Mail Extensions. Standardi, joka kuvaa sähköpostin sisältöä, mm. miten sähköpostissa on liitetiedostoja. |
| MVC | Model View Controller. Sovellusarkkitehtuuri, tapa miten sovelluksen toiminnot jaetaan kolmeen eri kategoriaan. |
| Otsikkotieto | Engl. header. Sähköpostin metatieto, joka kuvaa jotain sähköpostin ominaisuutta, kuten lähettäjän osoitteen. |

| | |
|---|---|
| Palvelutaso | Engl. service layer. Sovelluksen palveluluokat. |
| POGO | Plain Old Groovy Object. Grailsin ulkopuolinen Groovy-olio. |
| POJO | Plain Old Java Object. Grailsin ulkopuolinen Java-olio. |
| Rajoitus | Engl. constraint. Tiedon muotoa rajoittava määre, esimerkiksi minimipituus. |
| Riippuvuus | Engl. dependancy. Luokan tai toiminnallisuuden riippuvuussuhde. |
| Sidottu | Engl. coupled. Sovelluksen eri osien voimakas riippuvuussuhde. |
| SOAP | Simple Object Access Protocol. Verkkopalvelujen viestinvälitysprotokolla. |
| Tarkistaa | Engl. validate. Tiedon tarkistaminen ennalta asetetuilla määreillä. |
| Taustapalvelin | Engl. back-end server. Tikettijärjestelmän palvelin, jota käytetään ohjelmallisesti ilman käyttöliittymää. |
| Tavukoodi | Engl. bytecode. Ohjelmointikielestä käännetty koodi, mikä on Javan virtuaalikoneen suoritettavissa. |
| Tehdasmalli | Engl. factory pattern. Suunnittelumalli, missä jotakin rakennetta käytetään luomaan ilmentymiä luokasta. |
| Tunniste | Engl. tag. Merkintäkoodin osanen, GSP-tekniikassa myös metodikutsu, joka palauttaa merkintäkoodia tai käsittelee parametrina annettua tietoa. |
| Valtuus | Engl. token. Osoitus tilapäisestä valtuudesta tehdä jotakin. |
| Välityspalvelin | Engl. proxy. Etäkutsuja käsittelevä olio, joka välittää käskyjä taustapalvelimelle kuin kutsu olisi tapahtunut paikallisesti. |
| WSDL | Web Service Description Language. Verkkopalvelujen kuvauskieli. |

SISÄLLYS

1 JOHDANTO....7
2 KÄYTETYT TEKNIIKAT JA TYÖKALUT....9
2.1 Groovy....9
2.1.1 Esittely....9
2.1.2 Groovyn syntaksi....10
2.1.3 Groovy-luokat....11
2.2 Grails....11
2.2.1 Esittely....12
2.2.2 Grails-artifaktit....14
2.2.3 MVC-malli Grailsissa....14
2.2.4 Grails object relational mapping (GORM)....15
2.2.5 Liitännäiset....16
2.2.6 Palvelut....17
2.2.7 Testaus....18
2.2.8 Konfiguraatio....18
2.3 JavaScript Object Notation (JSON)....19
2.4 Simple Object Access Protocl (SOAP)....19
3 TIKETTIJÄRJESTELMÄ....20
3.1 Tikettijärjestelmän kuvaus ja esittely....20
3.2 Tarpeet ja vaatimukset....21
3.3 Sovelluksen rakenteen suunnittelu....22
3.4 Toteutunut sovellusarkkitehtuuri....23
4 TIKETTIJÄRJESTELMÄN TOTEUTUS....25
4.1 Työn toteutus yleisesti....25
4.2 Liitännäiset....26
4.3 Taustapalvelimen toteutus....27
4.3.1 Taustapalvelinliitännäinen....27
4.3.2 Tiedon tallennus....27
4.3.3 Ohjelmointirajapinta....29
4.3.4 Ohjelmointirajapinnan palautusarvot....29
4.3.5 Sähköposti....32
4.3.6 Tiketin luominen....34
4.4 Etupalvelimen toteutus....35
4.4.1 Etupalvelinliitännäinen....35
4.4.2 Tikettienhallintapalvelu....35
4.3.3 Käyttöliittymä....37
4.4.4 Käyttäjät....38
5 JATKOKEHITYS....39
5.1 Jatkokehityssuunnitelma....39
5.2 Taustapalvelimen jatkokehitys....40
5.3 Etupalvelimen jatkokehitys....41
5.4 Tiedonvälityksen jatkokehitys....41
5.5 Järjestelmäintegraation jatkokehitys....43
6 JOHTOPÄÄTÖKSET JA POHDINTA....45
LÄHTEET....46
LIITE....49

# 1 JOHDANTO

Tämä opinnäytetyöraportti kuvaa uuden tikettijärjestelmän toteutuksen Grails-sovelluskehyksellä. Työn toimeksiantaja on Symmetria Software Oy. Symmetrian nykyinen tikettienhallintajärjestelmä on tuotantokäytössä, mutta järjestelmää halutaan laajentaa ja uudistaa siten, että se ei olisi sidottuna yhteen sovellukseen kuten nykyinen järjestelmä, vaan liitettävissä moniin eri sovelluksiin. Liittäminen tapahtuu asentamalla kaksi liitännäistä ja mukauttamalla liitännäisten asetukset ja ulkoasu isäntäsovelluksen vaatimusten mukaisiksi. Järjestelmää ei suunniteltu tukemaan uuden järjestelmän tavoitteiden mukaisia ominaisuuksia, joten on luontevampaa rakentaa uusi järjestelmä kuin yrittää päivittää vanha tikettijärjestelmä. Vanhaa järjestelmää on ylläpidetty ja laajennettu rakentamalla useita pieniä päivityksiä ongelmakohtien korjaamiseksi ja uusien toiminnallisuuksien tuottamiseksi. Tämä päivitystapa on johtanut siihen, että järjestelmän rakenne on muuttunut sekavaksi, joten uusien päivityksien ja korjausten lisääminen on tullut aina vaikeammaksi.

Toteutin opinnäytetyön itsenäisesti työskennellen toimeksiantajan toimistolla. Työn aikataulu ja toteuttamistapa päätettiin toimeksiantajan kanssa palavereissa 8.3.2011, 15.3.2011 ja 23.3.2011 ja opinnäytetyön produktin luovutuspäivämääräksi asetettiin 30.6. Palavereissa sovittiin myös, että käyttäisin itse hyväksi katsomani ajan toteutustekniikoiden opiskeluun ennen itse työn alkamista. Kehitettävä sovellus päätettiin valmistaa prototyyppinä niin pitkälle kuin aikataulussa on mahdollista, minkä jälkeen kyseinen prototyyppi siirtyisi jatkokehitykseen toimeksiantajalle. Valmiin tikettijärjestelmän lopullisia käyttäjiä tulisivat olemaan toimeksiantajan asiakkaat.

Opinnäytetyön tavoitteena on rakentaa prototyyppi tikettijärjestelmäohjelmistosta. Keskeisimpinä tavoitteina on tehdä prototyypistä itsenäinen ja luontevasti ylläpidettävä kokonaisuus, joka voidaan liittää toimeksiantajan asiakkaiden ohjelmistoihin tuomaan lisätoiminnallisuutta. Tikettijärjestelmän tulee olla sovitettavissa toisiin web-sovelluksiin siten, että tikettijärjestelmää hyödyntävissä sovelluksissa voi olla erityyppisiä tikettejä ja että tikettijärjestelmän ulkoasu on yhdenmukainen muiden sovelluksien kanssa. Tikettijärjestelmän tulee myös hyödyntää jo käytössä olevia käyttäjätietokantoja.

Opinnäytetyöraportin tavoitteena on olla korkean tason dokumentaatio jatkokehityksen tueksi ja kertoa, mitä eri teknisiä ratkaisuja toteutetussa järjestelmässä on käytetty. Kuvaan järjestelmän toteutuksen vaihtelevasti sekä korkealta tasolta, kuten tiedon tallennusluokista kertovassa luvussa 6.2, että läheltä itse lähdekoodia, kuten taustapalvelimen palautusarvoja esittelevässä luvussa 6.4. Pyrin antamaan suunnan jatkokehitykselle järjestelmän kehityksen aikana tekemieni huomioiden perusteella. Raportin avulla voidaan myös arvioida tavoitteiden ja vaatimusten toteutumista opinnäytetyössä. Oletan, että opinnäytetyön lukijalle on ohjelmistokehityksen peruskäsitteet jo ennalta tuttuja, kuten termit olio, metodi, merkkijono, syntaksi, sovelluskehys, XML ja palvelin.

Opinnäytetyöraportin luvussa 2 käydään läpi tikettijärjestelmän määritelmä, tarpeet ja vaatimukset. Luvussa 3 käsitellään tikettijärjestelmän arkkitehtuuria korkealta tasolta tarkastellen suunniteltua ja toteutunutta arkkitehtuuria. Luku 4 kuvaa työssä käytetyt tekniikat, kuten Grails-ohjelmistokehyksen. Luvussa 5 kerrotaan tikettijärjestelmän teknisestä toteutuksesta yleisesti ja siitä, miten tikettijärjestelmä toteutettiin liitännäisiksi. Luvussa 6 käsitellään tikettijärjestelmän taustapalvelimen toteutus ja luvussa 7 taas etupalvelimen toteutusta. Luvut 6 ja 7 ovat lähimpänä teknisen toteutuksen yksityiskohtia ja niiden kappaleita on tarkoitus olla mahdollista käyttää yksittäisesti, esimerkiksi kun tikettijärjestelmän jostakin tietystä osa-alueesta halutaan lisätietoja. Toteutuskuvauksien jälkeen luvussa 8 kerrotaan, miten järjestelmää on suunniteltu jatkokehitettäväksi ja mitä eri jatkokehitysehdotuksia on noussut esiin kehityksen aikana. Viimeiseksi luvussa 9 käydään läpi opinnäytetyön tulokset: miten toteutettu tikettijärjestelmä ja opinnäytetyöraportti saavuttivat sille asetetut tavoitteet ja vaatimukset.

# 2 KÄYTETYT TEKNIIKAT JA TYÖKALUT

## 2.1 Groovy

Tässä luvussa esitellään Groovy-ohjelmointikieli lyhyesti. Luvussa vertaan Groovy-kieltä Javaan, koska kielet ovat läheisesti sukua toisilleen, koska Java on laajasti tunnettu ja koska Groovyn esitteleminen ilman vertailua vaatisi liian suuren tilan opinnäytetyöstä.

### 2.1.1 Esittely

Groovy on Javan virtuaalikoneelle (JVM, Java Virtual Machine) tehty ohjelmointikieli, joka on ottanut vaikutteita monista ohjelmointikielistä, kuten Pythonista, Rubystä ja Smalltalkista. Groovy on lähes täysin yhteensopivaa Javan kanssa. Groovy käännetään (compile) Javan tavukoodiksi (bytecode), joka voidaan suorittaa Javan virtuaalikoneella. (Groovy Home 2011.)

Groovy-kieleen tutustumista ja koodin testaamista helpottaakseen Groovyn kehittäjät ovat tehneet GroovyConsole-ohjelman, jolla voi kirjoittaa ja suorittaa koodia nopeasti, kuten monissa kehittyneissä ohjelmointiympäristöissäkin. Antamani esimerkit ovat kaikki tehty GroovyConsolessa.

Groovy on dynaaminen kieli, mikä tarkoittaa sitä, että ohjelmaa voidaan laajentaa ajonaikaisesti esimerkiksi muuttamalla luokkien toimintaa (Submaranian 2008, s.16). Esimerkiksi Javan String-luokkaan voidaan lisätä ajonaikaisesti metodeja, joita voidaan käyttää kaikissa String-luokan olioissa (Submaranian 2008, s.22). Groovyn muuttujat ovat myös dynaamisia, eli muuttujilla ei tarvitse Javan tapaan olla ennalta määriteltyä tietotyyppiä, vaan tietotyyppi päätellään kontekstista (Submaranian 2008, s.78). Groovyn muuttujat voidaan kuitenkin halutessa pakottaa staattisiin tietotyyppeihin. Lisäksi Groovy-luokilla on metaluokka (meta class), jota käytetään kielen dynaaminen luonteen hallinnassa (Sumerfield 2008).

### 2.1.2 Groovyn syntaksi

Groovyn syntaksi on lähes täysin yhteensopivaa Javan syntaksin kanssa (Groovy - Home). Groovy-ohjelmoinnin voi siis aloittaa kirjoittamalla puhdasta Javaa ja Groovylle ominaisia piirteitä voi lisätä myöhemmin.

Groovyssä ei ole lainkaan primitiivejä eli perustietotyyppejä. Javan perustietotyypit, kuten int, ovat Groovyssä aina oliota. Muuttujat, jotka ovat määriteltyjä int-tietotyypillä ovat Groovyssä aina java.lang.Integer-luokan ilmentymä (Submaranian 2008, s.87).

Seuraavat kolme esimerkkiä esittelevät eri tapoja toteuttaa toiminnallisuutta Groovyllä. Ensimmäinen on täysin yhteensopivaa Javan ja Groovyn kanssa, toinen sisältää hieman Groovylle ominaisia piirteitä ominaisuuksia ja kolmas on selvästi Groovyä eikä olisi helposti tunnistettavissa Javaksi. Esimerkin koodin voi suorittaa GroovyConsole-sovelluksessa. Java-koodia kokeillakseen on tehtävä uusi tiedosto, luokka ja päämetodi, joihin koodi sijoitetaan, minkä jälkeen vasta koodi voidaan kääntää ja suorittaa.

```
import java.util.regex.*;

// 1. Javaa, yhteensopivaa Groovyn kanssa
Pattern p1 = Pattern.compile("^AA.");
String[] stringset1 = new String[2];             // muuttuja
stringset1[0] = "AA1";
stringset1[1] = "1AA";
for (int i = 0; i < 2; i++) {                    // silmukka
   if (p1.matcher(stringset1[i]).matches()) {    // ehtolause
       System.out.println(stringset1[i]);        // tulostus
   }
}

// 2. Javan ja Groovyn sekoitusta
String[] stringset2 = ["AA2","2AA"];             // muuttuja
for (i in 0..1) {                                // silmukka
  if (Pattern.matches("^AA.", stringset2[i])) {  // ehtolause
      println(stringset2[i]);                    // tulostus
  }
}

// 3. Selkeästi Groovyä
def stringset3 = ["AA3","3AA"]   // muuttuja
stringset3.each {                // silmukka
  if (it ==~ /^AA./)  {          // ehtolause
      println it                 // tulostus
  }
}
```

### 2.1.3 Groovy-luokat

Luokat Groovyssä ovat samankaltaisia kuin Javassa, vaikka Groovyä voidaan kirjoittaa myös komentosarjakielen tapaan ilman luokkaviittauksia, sillä Groovy automaattisesti luo päämetodin koodin ympärille ennen koodin suorittamista. Groovy-koodia ilman luokkia kutsutaan komentosarjoiksi. Kaikki Groovy-luokat ovat Java-luokkia, jos tarkastellaan koodia Javan tavukoodiksi käännettynä. (Scripts and Classes 2011.)

Kuten Javassa, kaikki Groovyn luokat perivät java.lang.Object -luokan (Submaranian 2008, s.22). Groovy laajentaa Javan Object-luokkaa useilla lisämetodeilla. Groovyn luokkiin on mahdollista lisätä metodeja ja muuttujia ajonaikaisesti Groovyn metaluokan avulla. Tästä syystä monet Groovy-virheet on mahdollista huomata vasta ajonaikaisesti. (Dynamic Groovy 2011, Runtime vs Compile time, Static vs Dynamic 2011.)

Seuraavassa esimerkissä on määritelty kaksi esimerkkiluokkaa. Person-luokka kuvaa yksinkertaista kantaluokkaa ja Student-luokka kantaluokkaa laajentavaa luokkaa.

```
class Person {
   def birthday
}

class Student extends Person { }

def s = new Student(birthday:new Date())
assert(s.birthday instanceof Date)          // Varmistetaan syntymäpäivä
assert(s instanceof Person)                 // Varmistetaan, että olio on Henkilö
```

## 2.2 Grails

Opinnäytetyön toteutuksessa käytettiin Grailsin versiota 1.4.0.M1. Versiotunnisteen viimeinen osa "M1" tarkoittaa sitä, että versio on 1.4-sarjan ensimmäinen julkaistu versio, muttei kuitenkaan valmis 1.4.0-julkaisu. Näitä esijulkaistuja versioita suurista päivityksistä kutsutaan Milestone-versioiksi, josta tulee versiotunnisteen M-kirjain. Päätin toteuttaa järjestelmän yhteensopivaksi viimeisimmän julkaistun version ja viimeisimmän esijulkaistun version kanssa, jotta myöhemmin päivittäminen versioon seuraavaan vakaaseen versioon olisi mahdollisimman helppoa. Opinnäytetyötä toteutettaessa viimeisin julkaistu versio oli versio 1.3.7 ja viimeisin esijulkaistu versio oli käyttämäni 1.4.0.M1. Opinnäytetyöraporttia kirjoittaessani (29.8.2011) Grailsin uusin julkaistu versio oli 2.0.0.M1, jota kokeillessani tikettijärjestelmä näytti toimivalta, mutta en testan-

nut versioiden yhteensopivuutta kattavasti. Tämän kappaleen esimerkit ja dokumentaatioviittaukset koskevat versiota 1.4.0.M1.

### 2.2.1 Esittely

Grails on avoimen lähdekoodin web-sovelluskehys Groovy-kielelle (Documentation 2011). Grails on kehitetty monien modernien web-sovelluskehyksien konsepteja mukaillen. Grails käyttää hyväkseen Groovyn ohjelmointimahdollisuuksia ja Java-kielen hyväksi todettuja teknologioita, kuten Springiä ja Hibernatea. (Grails Reference Documentation 1. Introduction.)

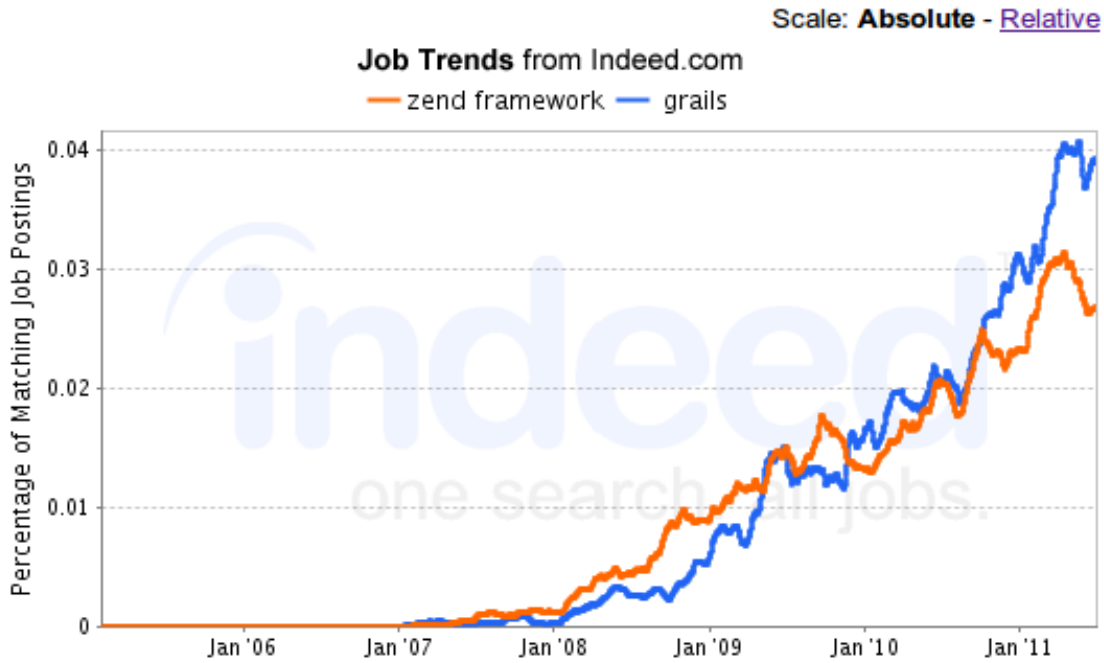
Grails noudattaa DRY-periaatetta (Don't Repeat Yourself, älä toista itseäsi), eli kehityksessä pyritään kirjoittamaan kaikki koodi vain kerran, ilman toistoa. Toinen Grailsin perusperiaatteista on Convention over Configuration eli tapojen suosiminen konfiguraation sijaan. Grailsia ei tarvitse konfiguroida aina uudelleen, vaan kehityksessä voidaan käyttää ennalta sovittuja tapoja ja järkeviä oletusarvoja, jotta kaikki toimii automaattisesti. (Grails Reference Documentation 1. Introduction, Grails Reference Documentation 3. Configuration.)

Grails-sovelluksen rakentamista aloittaessa ei tarvitse käyttää paljoa aikaa ulkopuolisten kirjastojen asentamiseen, vaan useimmiten käytetyt ja parhaiksi todetut kirjastot tulevat paketoituna Grailsin mukana. Grailsin version 2.0.0.M1 mukana on paketoituna Apache Tomcat, H2, jUnit, Groovy, Spring ja Hibernate. Toistaiseksi Grailsin dokumentaatiosta ei löydy listaa Grailsin mukana paketoiduista teknologioista, mutta kohtuullisen listauksen löytää selaamalla Grails-asennuksen lib-kansiota (tyypillisesti ~/grails/lib tai c:\grails\lib). Grails vaatii toimiakseen vain Java SDK -asennuksen, kahden ympäristömuuttujan asettamisen ja PATH-ympäristömuuttujan muokkauksen (Installation from Download 2011). Grailsin mukana tulee myös kattavat komentorivityökalut Grails-sovelluksen kehitykselle. Komentorivityökalujen avulla Grails-sovelluksen aloittaminen, kehittäminen ja suorittaminen onnistuu nopeasti (Grails Reference Documentation 4. The Command Line 2011).

Koska Grails toimii Javan virtuaalikoneella, on sillä toteutetut järjestelmät lähes alustariippumattomia, joten lopullista palvelinalustaa ei ole pakollista valita suunnitteluvai-

heessa ja kehitys voi tapahtua millä tahansa Javaa tukevalla käyttöjärjestelmällä. Myös käytetty tietokanta pitäisi olla helppo vaihtaa toiseen, joten tietokannan valinta ei ole kriittistä Grails-sovelluksen kehityksen kannalta.

Grailsin suosio työmarkkinoilla on voimakkaassa nousussa Indeed.com-sivuston trendityökalulla mitattuna (Kuvio 1). Valitsin vertailutyökaluiksi Indeed.com-työkalun, koska uskon työpaikkailmoitusten määrän olevan hyvä indikaattori sovelluskehyksien karkeasta suosiosta suhteessa toisiinsa. Harkitsin käyttäväni myös muita työkaluja vertailuun, mutta uskon työpaikkailmoitusten määrän olevan kehittäjille tärkeä vertailukohta. Valitsin Zend Frameworkin vertailukohdaksi Grailsille, koska Zend Framework on minulle ennalta tuttu, Grails ja Zend Framework on suunnilleen yhtä vanha, niiden absoluuttinen suosion ero ei ole suuri ja uskon ettei kuvaajaa vääristä asiaan liittymättömät hakutulokset.




*KUVIO 1. Indeed.com-työkalulla mitattuna grailsin ja zend frameworkin suosion kehitys.*

Grailsin suosiota uskoisin auttavan Grailsin on opettelun nopeus Java-osaajille ja luonnollinen yhdistäminen vanhoihin Java-sovelluksiin. Käytin itse työpäiväkirjani mukaan opinnäytetyöprosessin alussa 55 tuntia Grails in Action -kirjaan (Ledbrook & Smith 2009) tutustuen, jonka jälkeen aloitin tikettijärjestelmän toteuttamisen.

### 2.2.2 Grails-artifaktit

Lähes kaikkia Grails-sovelluksen luokkia kutsutaan Grails-artifakteiksi. Artifaktiluokkia eivät ole Groovy-luokat, jotka sijaitsevat Grails-sovelluksen kansiossa src/groovy/. Groovy-luokkia, jotka eivät ole Grails-artifakteja, kutsutaan usein termillä POGO (Plain Old Groovy Object). Artifaktit eroavat POGO-luokista siten, että ne sisältävät Grailsin automaattisten toimintojen antamaa toiminnallisuutta, kuten pääsyn Grails-palveluihin. Jos Grails-artifaktiin lisätään muuttuja, jonka nimi päättyy Service-sanaan, Grails yhdistää muuttujan palveluun ilman muita toimenpiteitä kehittäjältä.

### 2.2.3 MVC-malli Grailsissa

Grails toteuttaa osittain MVC-mallia (Model-View-Controller, Malli-Näkymä-Käsittelijä), eli Grails-sovelluksen toiminnallisuus on periaatteessa jaettu kolmeen eri kategoriaan. Grailsin MVC-mallin hyödyntäminen perustuu siihen, että Grails hyödyntää Spring MVC-sovelluskehystä (Spring MVC Integration). Osittaisella tuella tarkoitan sitä, että Grails-sovellukset usein sisältävät myös toiminnallisuutta, joita ei voida suoraan sijoittaa MVC-mallin tasoihin. Esimerkiksi Grailsin palvelutaso (service layer) ei sijoitu sopivasti mihinkään MVC-mallin tasoon, mutta on tärkeä osa Grails-sovellusta, johon kehotetaan sijoittamaan sovelluksen ydinlogiikkaa (Grails Reference Documentation 8. The Service Layer 2011). Grails-sovelluksessa saattaa olla myös Groovy- ja Java-luokkia (POGO, Plain Old Groovy Object ja POJO, Plain Old Java Object), liitännäisiä sekä konfiguraatioluokkia, joiden osuutta sovelluksesta on vaikea sijoittaa MVC-malliin. MVC-mallin hyödyntämistä ei painoteta Grailsin dokumentaatiossa tai teksteissä Grailista, esimerkiksi Ledbrook ja Smith mainitsevat termin vain neljä kertaa kirjassaan Grails In Action (2009): johdannon lisäksi termi mainitaan sivuilla 4, 7 ja 9 aina Spring-sovelluskehyksen yhteydessä.

MVC-mallin käsittelijät Grailsissa käsittelevät pyyntöjä ja luovat tai valmistelevat vastauksen http-pyyntöön. Käsittelijäluokat sisältävät metodeja, jotka usein vastaavat yksittäisiä sivuja web-sovelluksessa. Käsittelijät ovat myös mukana web-sovelluksen osoitteiden muodostamisessa. (Grails Reference Documentation, 6.1 Controllers 2011.)

MVC-mallin näkymää Grailissa vastaa GSP-tekniikka (Groovy Server Pages), joissa määritellään sovelluksen graafinen esitys. Jokaisella sivulla on yleensä oma näkymä, jota vastaa yksi GSP-tiedosto. GSP-tiedostot voivat sisältää Groovy-koodia, GSP-tun-

nisteita (GSP tag) ja perinteistä HTML-koodia. Grailsin mukana tulee laaja kirjasto GSP-tunnisteita ja tunnisteita voi myös määritellä itse. (Grails Reference Documentation 6.2 Groovy Server Pages 2011.)

Grailsin näkymät hyödyntävät Sitemesh-sovelluskehystä, mikä mahdollistaa, että näkymät voidaan muodostetaan malleista (template). Malleja hyödyntämällä ja yhdistelemällä on mahdollista tehdä yhdenmukaisia sivuja mallien päälle ilman koodin toistamista. (Grails Reference Documentation 6.2.4 Layouts with Sitemesh 2011.)

Grailsin käyttäjäohjeessa luvussa 6.2 Groovy Server Pages (Grails Reference Documentation) puhutaan "model"-muuttujaryhmästä, joka ei kuitenkaan vastaa MVC-mallin mallia siten, miten esimerkiksi Swing-sovelluskehyksen MVC-malli sitä käsittelee (Fowler 2011). Nimien yhteentörmäys voi aiheuttaa sekavuutta, mutta Grails-yhteisössä käytetään harvoin model-termiä viittaamaan kumpaankaan edellä mainituista. MVC-mallin mukaista mallia kutsutaan domain-sanalla tai domain model -sanaparilla ja usein mallin käsittelyn sijaan puhutaan suoraan seuraavassa luvussa esiteltävästä GORM-järjestelmästä. Vielä yleisempää tuntuu olevan, että MVC-mallia ei mainita lainkaan. (Ledbrook & Smith 2009, s. 65.)

### 2.2.4 Grails object relational mapping (GORM)

GORM (Grails object relational mapping) on Grailsin tiedon tallentamisen ydin. GORM on Hibernate 3 -sovelluskehyksen päälle rakennettu tiedon tallennusjärjestelmä. GORM-järjestelmän olennaisin toiminnallisuus on tietokannan yhdistäminen Groovy-luokkiin Hibernaten avulla. Yhdistäminen tarkoittaa sitä, että GORMin avulla ohjelmoijan on mahdollista käsitellä Groovy-olioita kuin relaatiotietokannan rivejä. Luokkiin määritellään ominaisuuksia ja metodeja, joiden avulla GORM osaa luoda tarvittavan tietokannan tauluineen. GORM lisää automaattisesti luokkiin myös dynaamisia metodeja tiedon hakuun, tallentamiseen, validointiin, muokkaamiseen ja poistamiseen liittyen, joten Grailsilla ohjelmoidessa ei välttämättä ole tarpeen kirjoittaa yhtäkään SQL-lausetta. (Grails Reference Documentation 5. Object Relational Mapping (GORM) 2011, What is Object Relational Mapping? 2011, Ledbrook & Smith 2009, s. 65.)

Tietokantaan tallentaessa GORM-luokasta luodaan ensin ilmentymä, minkä jälkeen kutsutaan ilmentymällä automaattisesti olevaa save-metodia, minkä jälkeen GORM tarkistaa (validate) ilmentymän tietojen oikeellisuuden käyttäen oletusarvoisia tai ohjelmoijan määrittelemiä tiedon muotoa rajoittavia määreitä (constraint). Jos tarkastus epäonnistuu, tallentaminen palauttaa null-arvon, mistä ohjelmoija voi päätellä tallentamisen epäonnistuneen. Tiedot voi tarkistaa myös kutsumalla validate-metodia. Jos tarkistus epäonnistuu, tapahtuneet virheet voidaan tarkistaa ilmentymälle ilmestyneen errors-muuttujan avulla. Koska GORM pohjautuu Hibernateen, tietoja ei välttämättä tallenneta tietokantaan välittömästi, sillä Hibernate luo tietokantaan tehtävät muutokset ja lisäykset väliaikaiseen istuntoon (session) ja vasta istunnon päättyessä suorittaa tietokantakutsun. Käytännössä Hibernaten istunnon aiheuttamaa viivettä ei huomaa. (Quick Reference: hasErrors 2011, Grails Reference Documentation 5.3.1 Saving and Updating 2011.)

Oletusarvoisesti tiedot tallentuvat Grailsin mukana tulevaan H2-relaatiotietokantaan Grailsin versiosta 1.4.0.M1 lähtien (Grails 1.4.0.M1 Release Notes 2011). H2 tallentaa tietoja oletusarvoisesti palvelimen muistiin, eikä kirjoita tietojaan palvelimen kiintolevylle. Grails-sovelluksen konfiguraatiotiedostoa grails-app/conf/DataSource.groovy muuttamalla GORM voidaan konfiguroida käyttämään tietokannan tallennuspaikkana tiedostoa. Konfiguraation avulla GORM voidaan myös yhdistää toiseen tietokantatoteutukseen, kuten MySQL-tietokantaan, jos tarvittavat ajurit asennetaan. (Grails Reference Documentation 3.3 The DataSource 2011.)

### 2.2.5 Liitännäiset

Grailsiin on saatavilla 632 liitännäistä tätä kirjoitettaessa (All Plugins 2011). Grailsin liitännäiset ovat samanlaisia kuin Grails-sovellukset, mutta ne voidaan liittää toisiin Grails-sovelluksiin tuomaan lisätoiminnallisuutta. Liitännäisien käyttäminen mahdollistaa modulaarisen kehityksen, missä sovelluksen toiminnallisuudet ovat toteutettu itsenäisinä kokonaisuuksinaan ja liitetty yhteen isoon sovellukseen. Osa Grailsin mukana tulevista toiminnallisuuksista ovat liitännäisiä, jotka ovat paketoitu tulemaan Grailsin perusasennuksen mukana. Versiossa 2.0.0.M1 Grailsin mukana on paketoitu 20 liitännäistä, mitkä voi nähdä uuden Grails-sovelluksen oletusetusivulla listattuna.

Liitännäisiä voidaan asentaa Grailsin virallisesta liitännäisvarastosta (repository) tai kiintolevyltä. Varastosta liitännäiset voi asentaa Grails-sovellukseen komentorivikomennolla grails install-plugin [liitännäisen nimi]. Liitännäistä voi käyttää myös viittaamalla Grailsin BuildConfig.groovy-konfiguraatiotiedostossa liitännäisen kansioon, mitä kutsutaan usein inline-asennukseksi. Viittaamalla liitännäisen kansioon on helppoa kehittää liitännäistä, sillä liitännäistä ei tarvitse paketoida ja asentaa aina, kun koodiin tekee muutoksia, vaan Grails huomaa muutokset liitännäisen tiedostoissa ja osaa ladata tarvittavat tiedostot uudelleen. (Grails Reference Documentation 12.2 Understanding repositories 2011.)

### 2.2.6 Palvelut

Grailsin palvelut ovat Grails-artifakteille yhteisiä luokkia. Palvelut toteuttavat singleton pattern-suunnittelumallia (Grails Reference Documentation 8.2 Scoped Services 2011), eli sovelluksen palveluluokista on olemassa sovelluksessa vain yksi ilmentymä. Suuri osa sovelluksen logiikasta suositellaan sijoitettavan palveluluokkiin. Palvelut yleensä kootaan loogisiksi toimintokokonaisuuksiksi, esimerkiksi kaikki valuuttamuunnoksiin liittyvät toimenpiteet voivat sijaita yhdessä palveluluokassa, josta niitä voidaan kutsua palvelua käyttävistä luokista. (Grails Reference Documentation 8. The Service Layer 2011.)

Palvelut voidaan ottaa käyttöön Grails-artifakteissa määrittelemällä artifaktille palvelun niminen muuttuja, esimerkiksi CurrencyConverterService-palvelun saa käyttöön määrittelemällä artifaktiin def-määreellä muuttuja currencyConverterService, jonka Grails automaattisesti yhdistää muuttujan ja palveluun (Quick Reference: Service Usage 2011). Grails-artifaktien ulkopuolella palveluluokkia ja GORM-järjestelmän luokkia voidaan hyödyntää ApplicationHolder-luokan avulla (Frequently Asked Questions: Misc 2011).

### 2.2.7 Testaus

Grailsin mukana tulee testaustyökalut yksikkö- integraatio- ja toimintatesteille. Käsittelijäluokkia ja GORM-luokkia luodessa Grails luo automaattisesti myös yksikkötestit uu-

sille luokille. Grailsin komennolla grails test-app Grails suorittaa kaikki määritellyt testit ja luo niistä testausraportit. Grailsin testaustyökaluja laajentamaan on olemassa myös monia liitännäisiä, kuten Spock-sovelluskehyksen toiminnallisuudet tuova liitännäinen. Opinnäytetyössäni en toteuttanut testausta, koska olen varma, etten olisi saanut toteutettua sovellusta halutulle tasolle, jos olisin opetellut testauksen. (Grails Reference Documentation 9. Testing 2011.)

### 2.2.8 Konfiguraatio

Vaikka Grails suosii ennalta sovittuja tapoja konfiguraation sijaan, pieni määrä konfiguraatiota on usein tarpeellista. Valtaosa Grailsin konfiguraatiosta määritellään Config.groovy-tiedostossa. Tiedostossa määritellään esimerkiksi Grailsin sisäänrakennetut konfiguraatiomuuttujat, globaalit konfiguraatiomuuttujat ja log4j-järjestelmän konfiguraatio. Log4j-järjestelmää käytetään Grails-sovelluksessa ajonaikaisien tietojen tulostamiseen komentoriville, tiedostoon tai molempiin. Edellä mainitun tiedoston lisäksi BuildConfig.groovy-tiedostossa voidaan konfiguroida sovellusta esimerkiksi määritellen riippuvuuksia (dependencies). Lisäksi GORM-järjestelmän tietokantavalintoihin voi vaikuttaa DataSource.groovy-tiedoston avulla. (Grails Reference Documentation 3. Configuration 2011.)

## 2.3 JavaScript Object Notation (JSON)

Tikettijärjestelmässä välitetään viestejä JSON-muodossa (JavaScript Object Notation). JSON on tekstin jäsennysmuoto, jonka avulla voidaan esittää tietoa standardoidussa muodossa. JSON-kuvauskieli on tiiviimpää ja helppolukuisempaa kuin suosittu XML-kuvauskieli (eXtensible Markup Language). (Introducing JSON 2011.)

Valitsin tiedon välitykseen JSON-muodon, koska JSON on mielestäni helpoimmin luettavaa kuin XML. Tämän lisäksi Grailsilla voi muuntaa minkä tahansa ilmentymän JSON- tai XML-muotoon ja muunnosprosessia on helppo muokata (Converters Reference 2011). Jos tarpeellista, JSON pystytään vaihtamaan XML-muunnokseen.

## 2.4 Simple Object Access Protocl (SOAP)

Tikettijärjestelmässä välitetään viestejä verkossa käyttäen SOAP-protokollaa (Simple Object Access Protocol). SOAP on XML-muotoisiin viesteihin perustuva tiedonvälitystekniikka (SOAP Introduction 2011). SOAP-viestit muodostuvat kirjekuoresta (envelope), otsakkeesta (header) ja kirjeestä (body). Kirjekuoren avulla dokumentti voidaan tunnistaa SOAP-viestiksi ja se on samalla SOAP-viestin juurielementti (SOAP Envelope Element 2011). SOAP-otsake sisältää sovellukselle ominaisia tietoja ja kirje sisältää varsinaisen lähetetyn viestin (SOAP Header Element 2011, SOAP Body Element 2011).

SOAP-palvelua käytetään opinnäytetyössä sitä kuvaavan WSDL-dokumentin (Web Services Description Language) avulla. WSDL on XML-pohjainen kuvauskieli, jolla voi kuvata web-palveluja, kuten SOAP-protokollaa. (Ogbuji 2000).

SOAP-palvelun pystyttämiseen on olemassa helppokäyttöinen Grails-liitännäinen, joka hyödyntää CXF-sovelluskehystä (Crum 2011), joten valitsin SOAPin tiedonvälitystekniikaksi. Harkitsin myös REST-tekniikan (Representational State Transfer) käyttöä tiedonvälityksen toteutuksessa, mutta SOAP oli toimeksiantajalla käytössä toisissa projekteissa, joten se olisi luonnollisempi osa toimeksiantajan sovelluksia. SOAP pystytään myös vaihtamaan toiseen tiedonvälitystekniikkaan, joten tiedonvälitys palvelimien välillä voidaan jatkossa toteuttaa jollain toisella tavalla.

# 3 TIKETTIJÄRJESTELMÄ

## 3.1 Tikettijärjestelmän kuvaus ja esittely

Toteuttamani tikettijärjestelmä on prototyyppi, jonka pohjalta voidaan kehittää moniin sovelluksiin sopiva tikettijärjestelmämalli. Tikettijärjestelmän ydintoiminnallisuudet ja ulkoasu voidaan erikoistaa erilaisiksi tikettijärjestelmiksi toimeksiantajan asiakkaille.

Tiketit voivat olla mitä tahansa asioita, joihin järjestelmän käyttäjän tulee reagoida tai jotka tulee kuitata nähdyksi. Kaikilla tiketeillä on aina otsikko, luomispäivämäärä, prioriteetti ja tietoa siitä onko tiketti suljettu vai avoin. Lisäksi kaikilla tiketeillä voi olla omistaja, kuvaus, sulkemisaika, muutosaika ja tieto siitä, onko tiketti odottamassa vastausta, mutta nämä tiedot eivät ole pakollisia. Tiketin määritelmä jätettiin avoimeksi, jotta järjestelmä pystyisi käsittelemään monia erilaisia tilanteita tulevaisuudessa. Järjestelmää saatetaan tulla käyttämään hyvin erilaisissa tilanteissa, esimerkiksi apuna palautteiden käsittelyssä tai työtehtävälistana.

Tiketit voivat olla ihmisen tai automaattisien prosessien tuottamia. Ihmisen tuottama tiketti voi olla luotu esimerkiksi yhteydenottolomakkeesta tai asiakkaan lähettämästä sähköpostista. Koneen tuottama tiketti voi taas olla esimerkiksi varoitus, että järjestelmässä on soittopyyntö, jota ei ole käsitelty loppuun kuukauden aikana, tai järjestelmän ilmoitus jonkin automaattisen prosessin epäonnistumisesta.

Tikettijärjestelmää tulisivat käyttämään asiakaspalvelijat ja teknikot, eli järjestelmälle tarvitaan vain kirjautuneiden käyttäjien käyttöliittymä. Asiakaspalvelijat voisivat käyttää järjestelmää esimerkiksi siten, että asiakkaan soittaessa yritykseen tilatakseen tarjouspyynnön, soitto ja tarjouspyyntö tallennettaisiin järjestelmään tehtäväksi. Teknikon käyttötapaus voisi olla, että järjestelmä on luonut automaattisesti huomautuksen, jossa ilmoitetaan tuotteiden käsittelyjärjestelmässä olleen käsittelemätön tilaus kuukauden, minkä takia tilaus pitää tarkistaa, käsittelyä tulee priorisoida tai asiakkaalle tulee ilmoittaa tilauksen viivästyksestä. Kirjautumattomilla käyttäjillä ei tule olemaan pääsyä tikettijärjestelmään.

## 3.2 Tarpeet ja vaatimukset

Tarve uuden tikettijärjestelmän rakentamiselle perustuu siihen, että vanhan järjestelmän päivittäminen erillisissä räätälöidyissä sovelluksissa johti työn toistoon ja järjestelmän epätehokkaaseen siirtämiseen. Tikettijärjestelmästä halutiin luoda itsenäinen sovellus, joka voidaan helposti sulauttaa eri projekteihin ilman tikettien käsittelyn ydintoiminnallisuuden rakentamista moneen kertaan. Ennen uudistusta järjestelmät oltaisiin toteutettu aina uudelleen yhteen sovellukseen räätälöitynä. Jotta tikettijärjestelmä voitaisiin sisällyttää räätälöidysti myös muihin sovelluksiin toistamatta työtä, oli tarpeellista toteuttaa järjestelmän ydin itsenäiseksi. Täten tikettijärjestelmää olisi mahdollist kehittää keskitetysti ja tikettijärjestelmä olisi mahdollista myös tuotteistaa.

Järjestelmän vaatimuksissa määriteltiin, että järjestelmän tulee tukea monia erityyppisiä tikettejä. Tikettityyppejä pitää olla mahdollista myös lisätä siten, että jokaisella tikettijärjestelmän sisältävällä sovelluksella voi olla omat tikettityyppinsä. Tikettityypit tulee eroamaan toisistaan siten, että eri tyyppisillä tiketeillä voi olla perustietojen lisäksi mitä tahansa lisätietoja, esimerkiksi tiedostoja tai luokitteluja. Tikettijärjestelmä luodaan itsenäiseksi moduulikseen, jotta toiminnallisuus voidaan liittää moniin sovelluksiin. Järjestelmän itsenäisyys saavutetaan toteuttamalla sovellus liitännäiseksi. Tikettijärjestelmän arkkitehtuuri tulee olla jaettu kahdelle palvelimelle, joista toinen toimii järjestelmän käyttöliittymänä ja toinen tikettien käsittelijänä. Kutsun opinnäytetyöraportissa järjestelmän käyttöliittymän sisältävää palvelinta etupalvelimeksi (front-end server) ja tikettejä käsittelevää palvelinta taustapalvelimeksi (back-end server).

Vaatimusmäärittelyssä päätettiin, että taustapalvelinta käytetään ainoastaan ohjelmointirajapinnan avulla, eikä taustapalvelimelle tule graafista käyttöliittymää. Taustapalvelin tallentaa tiketit tietokantaan ja luo tikettejä automaattisien prosessien kautta. Järjestelmä kehitetään rakenteeltaan selkeäksi, millä varmistetaan, että tulevat päivitykset tikettijärjestelmään eivät sekoita järjestelmän rakennetta. Järjestelmän pitää olla myös integroitavissa käyttäjätietokantoihin, eli järjestelmällä ei tule olemaan omaa yksityistä käyttäjätietokantaa, vaan se hyödyntää jotain ulkopuolista käyttäjätietokantaa.

Tuotantoon otettavassa järjestelmässä tullaan käyttämään sähköpostia kommunikaatioon, mutta sähköpostin hyödyntäminen järjestelmässä jätettiin palavereissa ideatasolle,

sillä en ollut varma, että kykenisinkö toteuttamaan sähköpostin käsittelyä järjestelmän kehittämiselle varatussa ajassa.

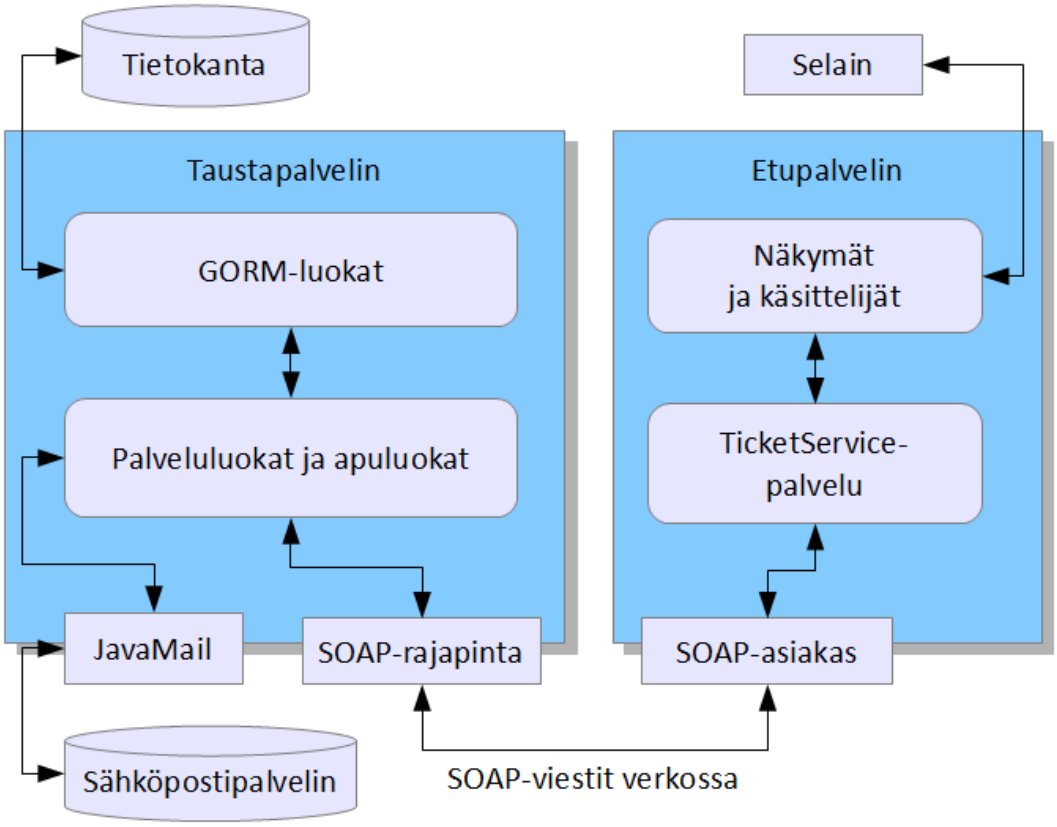
## 3.3 Sovelluksen rakenteen suunnittelu

Sovelluksen vaatimuksien pohjalta sovelluksen rakenteesta eli arkkitehtuurista tehtiin yksinkertainen luonnos palaverissa toimeksiantajan kanssa. Järjestelmästä päätettiin sisällyttää mahdollisimman paljon logiikkaa taustapalvelimelle, jotta tikettijärjestelmän lisääminen etupalvelimelle ei vaikuttaisi palvelimen suorituskykyyn. Taustapalvelimelle päätettiin tehdä tikettejä luova tehdasluokka, joka toteuttaa tehdasmallia (factory pattern). Tehdasluokka luo tikettiolioita annetun parametrin pohjalta käyttäen Oodesign.comin Procedural Solution -tyyppistä tehdasmallia, missä olio luodaan annetun parametrin avulla (Factory Pattern 2011).

Tiketit päätettiin toteuttaa periytyviksi, eli tiketeillä tulisi olemaan yksi kantaluokka, mistä periytettäisiin kaikki tikettityypit. Kantaluokka sisältäisi kaikille tiketeille yhteiset ominaisuudet, kuten luomisajan, kuvauksen ja tikettiin lisätyt kommentit ja sähköpostiviestit. Taustapalvelimen ja etupalvelimen tiedonsiirtoon suunniteltiin otettavan käyttöön SOAP-rajapinta (Simple Object Access Protocol), mutta rajapinnan toteutuksen yksityiskohdat jätettiin avoimeksi. Päätimme myös, että molemmat palvelimet tuntisivat sovelluksen luokat, jotta palvelimien välillä voitaisiin lähettää tikettejä todellisina olioina. SOAP-rajapinta päätettiin toteuttaa siten, että rajapinnan rinnalle voitaisiin myöhemmin lisätä muita rajapintoja ja siten, että SOAP olisi mahdollista korvata jollakin muulla vaihtoehdolla, kuten REST-rajapinnalla (Representational State Transfer).

## 3.4 Toteutunut sovellusarkkitehtuuri

Toteutettu järjestelmä vastaa suurimmalta osalta alkuperäistä korkean tason sovellusarkkitehtuurisuunnitelmaa. Taustapalvelin ja etupalvelin toteutettiin Grails-liitännäisiksi, eli kaikki tikettijärjestelmän toiminnallisuudet voidaan liittää Grails-sovellukseen asentamalla liitännäiset ja mukauttamalla toteutus isäntäsovelluksen vaatimuksien mukaan. Kuten suunniteltu, palvelimien väliseen keskusteluun otettiin käyttöön SOAP-rajapinta. Taustapalvelin paljastaa SOAP-rajapinnan WSDL-dokumentin (Web Service Description Language) avulla. Etupalvelin käyttää taustapalvelimen paljastamaa rajapintaa, eli etupalvelin toimii asiakkaana taustapalvelimelle. Kuviossa 2 on nähtävillä karkeasti miten järjestelmän eri osat ovat jaettu palvelimille ja miten osat keskustelevat toistensa kanssa.




*KUVIO 2. Tikettijärjestelmän eri osa-alueiden karkea kuvaus.*

Järjestelmän taustapalvelin on vastuussa tikettien käsittelystä ja tallentamisesta. Etupalvelin esittää ja välittää tietoja tiketeistä ilman tietoa tikettien todellisesta luokasta, tietokannasta tai taustapalvelimen toteutuksesta. Taustapalvelimella on tietokanta tikettien tallennukseen sekä rajapinta, jolla taustapalvelinta voi käyttää. Lisäksi taustapalvelimella on tikettien käsittelyyn liittyviä toiminnallisuuksia, kuten tikettiin tehtyjen muutosten automaattinen tallennus. Taustapalvelin pystyy luomaan tikettejä automaattisten prosessien avulla, esimerkiksi käsitellen sähköposteista automaattisesti tikettejä. Taustapalvelin käy tunnin välein noutamassa sähköpostitililtä viestit, joista tehdään kopiot ennalta luotuihin tiketteihin tai joista luodaan kokonaan uusia tikettejä.

Suurin ero järjestelmän suunniteltuun arkkitehtuuriin on se, että tieto palvelimien välillä lähetetään merkkijonoina käyttäen JSON-merkintää (JavaScript Object Notation), eikä olioita välitetä lainkaan. Tieto välitetään JSON-merkkijonoina kahdesta syytä. Ensinnäkin järjestelmää luodessa kokeiltiin todellisten olioiden liikuttamista verkossa, tämän käytännön toteutus osoittautui ongelmalliseksi. Käytössä olevalla viestinvälitysteknii-kalla olisi ollut mahdollista lähettää olioilmentymiä, mutta lähetetyt olioilmentymät menettivät kokeiluissa polymorfisuutensa, eli oliot eivät SOAP-kuljetuksen jälkeen sisältäneet tietoa perinnöllisyydestään. Olioiden siirto verkon yli on mahdollinen kehittää järjestelmään, mutta kehityksessä ilmeni monia ongelmia. Harkitsin tikettiluokkien toteuttamista liitännäisenä, joka voitaisiin asentaa takapalvelimen lisäksi etupalvelimelle, mutta tämä ratkaisu olisi kompastunut räätälöityjen tikettityyppiluokkien siirron toteuttamiseen. Toinen syy JSON-muodon valintaan, koska Grails tukee JSON-muunnoksia hyvin. JSON-muunnos oli helppo toteuttaa taustapalvelimelle ja etupalvelimen muunnos merkkijonosta JSON-objektiksi mahdollisti tiedon nopean ja yksinkertaisen hyödyntämisen. Vaihtoehto JSON-muodolle olisi lähinnä ollut XML-muotoisen tiedon käsittely, mutta JSON on mielestäni yksinkertaisesti helpompaa lukea ja hyödyntää. JSON-merkkijonojen hyödyntäminen johtaa myös siihen, että etupalvelin ei tunne taustapalvelimen tikettiluokkia kuten alkuperäisessä arkkitehtuurissa oli suunniteltu.

# 4 TIKETTIJÄRJESTELMÄN TOTEUTUS

## 4.1 Työn toteutus yleisesti

Keskityin tikettijärjestelmää toteutettaessa järjestelmän ydintoiminnallisuuksiin ja perusrakenteeseen, jotta saisin tehtyä mahdollisimman hyvän pohjan jatkokehitykselle. En toteuttanut kaikkia mahdollisia tikettityyppejä ja ominaisuuksia, joita tikettijärjestelmälle olisin keksinyt, vaan keskityin siihen, että toteutetusta järjestelmästä tulee mahdollisimman vankka pohja jatkokehitykselle. Eri tikettityypit voidaan määritellä muutamien toteuttamieni esimerkkien pohjalta.

Taustapalvelimen ja etupalvelimen toiminnot ovat molemmat paketoitu Grails-liitännäisiksi. Liitännäiset ovat suunniteltu liitettäväksi jo käytössä olevaan järjestelmään, joka samalla hoitaisi muitakin toimintoja. Taustapalvelinliitännäinen antaa sovellukselle tikettien ja sähköpostien tallentamisen mahdollistavat luokat, sähköpostia vastaanottavat ja lähettävät luokat, virheviestien generointipalvelun, yksilöidyn JSON-muuntimen tikettien tiedonvälitykseen, tiedostoja tallentavan ja lataavan palvelun sekä taustapalvelimen rajapinnan määrittelevät palvelut. Etupalvelinliitännäisen mukana tulee taustapalvelimen rajapinnan hyödyntämistä helpottava palvelu, esimerkkikäännöksiä virheviesteihin, tunnistekirjasto, käyttäjätilejä simuloiva luokka, käsittelijäluokka näkymineen tikettien, käyttäjien ja avainsanojen hallintaan sekä valmiit mallit sähköposteille.

Keskityin kehityksen aikana eniten taustapalvelimen toimintaan, koska halusin luoda taustapalvelimesta mahdollisimman vakaan pohjan jatkokehitykselle ja koska uskon, että etupalvelin tulee muuttumaan enemmän suhteessa taustapalvelimeen. Tiketinhallintaa toteutettaessa asiakkaalle määriteltäisiin asiakkaan taustapalvelimelle oma konfiguraatio ja luotaisiin omat tikettiluokat.

Työtä varten suunnittelin toteuttavani kaiken toteuttamani koodin käyttäen testilähtöistä menetelmää (TDD, Test Driven Development). Testilähtöisessä menetelmässä olisin kehittänyt järjestelmää siten, että olisin kirjoittanut jokaiselle ominaisuudelle testitapaukset ennen toteutusta. Sovelluksen koodimäärän kasvaessa päätin kuitenkin, että testilähtöisen kehityksen opiskelun ja toteutuksen urakka olisi liian suuri opinnäytetyölle varat-

tuun aikaan nähden. Kattavan testauksen saavuttaminen olisi ollut hyvin haastavaa ja suuri osa sovelluksen koodista kirjoitettaisiin useita kertoja uudelleen siten, että testaustapaa olisi jouduttu muuttamaan, joten päätin, että liian suuri osa ajastani olisi kulunut testauksen opiskeluun ja toteutukseen.

Tikettijärjestelmä toimii vain web-käyttöliittymällä, joten järjestelmän toimintaa on helppoa testata http-pyynnöillä. Käytin http-pyyntöjen testaamisessa selainta ja cURL-ohjelmaa. cURL on komentorivityökalu, jolla on mahdollista tehdä monimutkaisia pyyntöjä käyttäen monia eri protokollia (cURL 2011). Toistuvat pyynnöt on mahdollista sisällyttää myös komentoriviskripteihin, yksinkertaistaen esimerkiksi monimutkaisien lomakkeiden lähettämistä.

## 4.2 Liitännäiset

Koska toteutin taustapalvelimen ja etupalvelimen toiminnallisuudet liitännäisinä, tiketti-toiminnallisuudet voi asentaa mihin tahansa Grails-sovellukseen. Liitännäistä konfiguroimalla tikettienhallinnan saa käyttöön omilla asetuksilla ilman liitännäisen koodin muokkausta. Tämä mahdollistaa myös sen, että tikettijärjestelmää päivitetään erillään sen toteutuksista, ja että liitännäistä hyödyntävät palvelimet voivat päivittää liitännäisen päivittämättä koko sovellusta.

Hyödynnän Grailsin mukana paketoitujen liitännäisten lisäksi Quartz-liitännäistä, joka mahdollistaa ajastettujen tehtävien lisäämisen, CXF-liitännäistä, joka mahdollistaa taustapalvelimen toiminnallisuuden julkaisemisen web-palveluna ja WSClient -liitännäistä, joka mahdollistaa taustapalvelimen SOAP-rajapinnan hyödyntämisen etupalvelimelta.

Liitännäisten asennusta testattiin lisäämällä tyhjän Grails-projektin konfiguraatiotiedostoon (BuildConfig.groovy) viittaus paketoimattoman liitännäisen projektikansioon. Myös tuotantokäytössä tämänkaltainen asennus on mahdollinen, mutta liitännäiset voidaan paketoida myös yhdeksi tiedostoksi, joka voidaan asentaa kuin mikä tahansa muukin Grails-liitännäinen. Asennuksen jälkeen isäntäsovelluksen konfiguraatiotiedostoon voidaan lisätä tikettijärjestelmän asetuksia ottamalla mallia liitännäisten mukana tulevasta konfiguraatiotiedostosta. Taustapalvelimelle lisätään asennuksen jälkeen omat ti-

kettityyppejä kuvaavat GORM-luokat, jonka jälkeen taustapalvelin pitäisi olla käyttövalmis. Etupalvelinliitännäisen mukana tuleviin sivuihin lisätään linkki jonnekin isäntäsovellukseen, jotta tikettijärjestelmään pääsee käsiksi ja etupalvelimen sivut muokataan sopimaan uuteen isäntäsovellukseen. Todennäköisesti etupalvelimen koko käyttöliittymä pitää rakentaa hyvin laajasti uudelleen isäntäsovelluksen mukaan, koska tikettityyppien listaukset ja esitykset tulevat todennäköisesti vaihtelemaan paljon.

## 4.3 Taustapalvelimen toteutus

### 4.3.1 Taustapalvelinliitännäinen

Taustapalvelin vaatii toimiakseen isäntäsovellukseltaan CXF-liitännäisen. Ajastetut toiminnot taustapalvelimelle voi lisätä käyttäen Quartz-liitännäistä. Lisäksi sähköpostin toimintaan vaaditaan JavaMail-kirjasto, paitsi jos sähköpostin käsittely ei ole tarpeellista. Taustapalvelinliitännäisen mukana tulee tiiviisti listattuna tikettien kantaluokka, kaksi tikettityyppiesimerkkiä, palvelu SOAP-rajapinnalle, rajapinnan toimintojen logiikan toteuttava palvelu, sähköpostipalvelu, virheviestien käsittelypalvelu, tiedostojen käsittelypalvelu ja luokka tikettien muuntamiseen JSON-muotoon.

### 4.3.2 Tiedon tallennus

Tikettijärjestelmän tiedot tallennetaan luvussa 4.2.4 esiteltyyn GORM-järjestelmään. Järjestelmän kaikki GORM-luokat joko periytyvät yhdestä kantaluokasta tai ovat olemassa ainoastaan kantaluokan yhteydessä. Kaikille yhteinen kantaluokka määrittelee ne ominaisuudet ja tiedot, joita voidaan olettaa olevan kaikilla tiketeillä. Esimerkkinä kantaluokkaa laajentavasta tiketistä on palautetiketti, jota varten toteutin kantaluokkaa perivän palautetikettiluokan, joka sisältää tiketin perustietojen lisäksi tiedon, onko lähetetty palaute positiivinen, negatiivinen tai jotain muuta. Periyttämällä tiketit kantaluokasta voidaan varmistaa, että kaikilla tiketeillä tulee aina olemaan tietyt perustiedot tunnetussa muodossa. Tätä tietoa kaikkien tikettien yhteisistä tiedoista voidaan käyttää esimerkiksi hakutoiminnossa ja tikettilistauksissa, missä näytetään tietoja useista eri tikettityypeistä.

Tikettien muutoshistoria tallennetaan muutoskokoelmaluokan (Changeset) avulla. Muutoskokoelma voi sisältää yhden tai useita muutosluokan (Change) ilmentymiä. Muutoskokoelma kuvaa mitä muutoksia tikettiin on tehty yhdellä kertaa ja jokainen yksittäinen muutosluokan ilmentymä kuvaa yhtä muutosta. Tiketillä voi olla esimerkiksi muutoskokoelma, missä on kaksi muutosta, joista toinen ilmaisee sitä, että tikettiin on lähetetty sähköpostivastaus ja toinen sitä, että tiketti on samalla merkitty odottamaan vastausta. Muutoskokoelma voi sisältää myös kommentteja, jotka liittyvät tiketin historiaan.

Millä tahansa tiketillä voi olla sähköpostiviestejä liitettyinä. Lähetetyt ja vastaanotetut viestit tallennetaan EmailMessage-luokan ilmentyminä. Sähköpostin liitetiedostot tallennetaan tiedostoina palvelimen kiintolevylle siten, että tietokantaan tallennetaan tiedoston MIME-standardin (Multipurpose Internet Mail Extensions) mukainen tietotyyppi (Wikipedia MIME Content-Type 2011), tiedoston nimi ja tiedoston sisällöstä generoitu merkkijono, jota käytetään tiedostonimenä tallennettaessa ja hakiessa tiedostoa palvelimen kiintolevyltä. Merkkijono luodaan muodostamalla tiiviste (hash) tiedoston tavuista SHA-1-algoritmin avulla. Taustapalvelin ottaa oletusasetuksilla yhdeltä sähköpostiviestiltä vastaan maksimissaan 9 liitetiedostoa, jotka saavat olla kukin maksimissaan 10 megatavun kokoisia. Asetuksia voidaan muuttaa toteutuskohtaisesti. Jos sähköpostissa on liikaa tai liian isoja liitetiedostoja, sähköposti hylätään hiljaisesti. Vaihtoehtoisesti hylkäyksestä voitaisiin lähettää ilmoitusviesti vastauksena sähköpostiin, mutta ilmoitusviestiä ei toteutettu prototyyppiin.

### 4.3.3 Ohjelmointirajapinta

Taustapalvelimella ei ole graafista käyttöliittymää, vaan taustapalvelinta käytetään ainoastaan ohjelmointirajapinnan kautta. Taustapalvelimen ohjelmointirajapinnan toiminnot on määritelty TicketAPIService-palvelussa ja toimintojen toteutus on määritelty TicketHandlerService-palvelussa. Uloin palvelu, eli TicketAPIService, ei prosessoi tietoa eikä sisällä mitään logiikkaa, vaan se kutsuu sisemmän TicketHandlerService-palvelun metodeja. Tämä kahtiajako helpottaa pitämään rajapinnan muuttumattomana, sillä sen avulla toteutuksen ja rajapinnan välissä on looginen raja. Rajapinnan uloimpaa palvelua ei tulisi muokata usein, jotta sitä käyttävät etupalvelimet pystyvät luottamaan, että rajapinta pysyy samanlaisena, vaikka rajapinnan taustatoteutus muuttuisikin.

TicketAPIService voidaan myös korvata tai se muokata käyttämään jotain muuta tiedonsiirtotekniikkaa. TicketAPIService-palvelun rinnalle voidaan myös lisätä toisia tiedonsiirtotekniikoita tukevia palveluja siten, että uudet palvelut kutsuvat TicketHandlerService-palvelun metodeja.

Ohjelmointirajapinta muodostetaan Grailsin CXF-liitännäisen avulla. CXF luo TicketAPIService-palvelun metodeista WSDL-dokumentin. CXF myös käsittelee rajapinnalle tulevat SOAP-pyynnöt metodikutsuiksi ja muodostaa metodin palautusarvosta SOAP-vastauksen. CXF:n voi vaihtaa omaan toteutukseen tai sen rinnalle voi lisätä muita ohjelmointirajapintoja sen vaikuttamatta mitenkään tikettienkäsittelyyn vaikuttavaan koodiin. Jos CXF-liitännäinen vaihdetaan toiseen toteutukseen, on syytä poistaa myös huomautusmerkinnät (annotation) taustapalvelimen GORM-luokista, joiden avulla olisi mahdollista paljastaa luokat rajapinnan kautta.

### 4.3.4 Ohjelmointirajapinnan palautusarvot

Lähes kaikkien TicketHandlerService-palvelun metodit ja täten myös ohjelmointirajapinnan metodit palauttavat merkkijonoja. Merkkijonot ovat JSON-muodossa, jotta etupalvelin pystyisi jäsentämään merkkijonoista JSONObject-luokan ilmentymän ja täten käsittelemään palautettua tietoa helposti. Syy JSON-muodon käyttöön eritellään tarkemmin luvussa 3.4, tiivistetysti verkossa ei välitetä todellisia tikettiluokkien ilmentymiä,

koska ilmentymien välityksessä ongelmaksi nousi ilmentymien polymorfismin menettäminen.

JSON-muotoisen merkkijonot muodostetaan kolmella eri tavalla. Yksinkertaisimmat JSON-viestit muodostetaan suoraan merkkijonoiksi, esimerkiksi totuusarvot palautetaan valmiina JSON-merkkijonona ja virheistä muodostetaan merkkijonot ErrorService-palvelun avulla. Listat, jotka palauttavat aina samat tiedot ennalta tunnetussa muodossa, muodostavat ensin moniulotteisen taulukon halutuista tiedoista. Luotu taulukko muunnetaan JSON-esitykseen Grailsin JSON-muuntajalla (luokka grails.converters.JSON).

Edellä mainittujen JSON-viestien muodostamistekniikkojen lisäksi käytössä on CustomDomainMarshaller-luokkaan perustuva muunnos. CustomDomainMarshaller muuntaa minkä tahansa tiketin ilmentymästä JSON-muotoisen esityksen muuntaen samalla tikettien sisäiset omistussuhteet JSON-muotoon. CustomDomainMarshaller perustuu Siegfried Puchbauerin samannimiseen luokkaan, jonka Puchbauer (2009) esitteli vastatessaan Grailsin JSON-muunnoksiin liittyvään kysymykseen. Puchbauer ollut kehittämässä Grailsin muunnosjärjestelmää, joka on ollut osana Grailsin ydintoiminnallisuutta versiosta 0.6 (Puchbauer 2011b). Puchbauer on myös merkitty Grailsin dokumentaatiossa muunnosluokkien kirjoittajaksi (Puchbauer 2011a).

CustomDomainMarshaller on tarpeellinen, koska se pystyy muuntamaan minkä tahansa tiketti-ilmentymän ominaisuuksineen dynaamisesti. Ero Grailsin tavalliseen JSON-muunnokseen on, että CustomDomainMarshaller muuntaa tikettien sisäiset tietokantaviittaukset rekursiivisesti. Tikettiä muunnettaessa myös tiketin omistamat muutoskokoelmat ja niiden omistamat muutokset muuttuvat rekursiivisesti JSON-muotoon. Samalla tiketin luokkanimet voidaan muuttaa helpommin käsiteltävään muotoon, esimerkiksi luokkanimi fi.symmetria.ticket.EmailTicket muuttuu muotoon email.

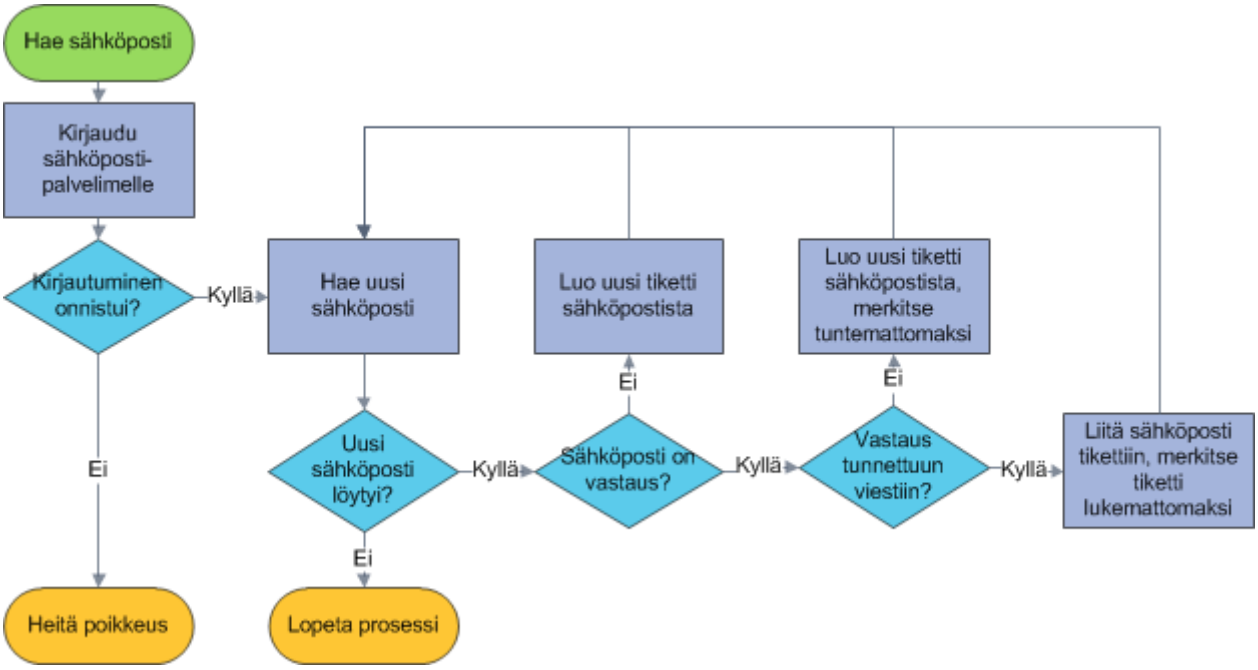
Liitteessä 1 on esimerkkejä JSON-palautusarvoista. Ensimmäinen esimerkki on yksinkertainen muuttuja ja arvo -pari, joka sisältää muuttujalla result arvon true. Esimerkki 2 on virheviesti, jossa on ainoastaan viestimuuttuja message. Message-virheviestit on suunniteltu käytettäväksi kaikissa tilanteissa, jossa kyse ei ole kyseessä virheestä, joka liittyy web-lomakkeen kenttään. Jos esimerkiksi sähköpostin lähettäminen epäonnistuu, message-muuttujassa on ilmoitus tapahtuneesta virhetilanteesta. Esimerkissä 3 on mu-

kana viittaus virheen sisältäneeseen kenttään field-muuttujassa. Tämä virhetilanne voisi olla lähtöisin esimerkiksi lähetetystä web-lomakkeelta, jonka jokin kenttä ei läpäise GORM-tarkastusta. Esimerkissä 3 field-muuttujan arvo email osoittaa virheen olevan sähköpostikentässä. Code-kentässä on virheviesti, mistä voi päätellä miten annettu arvo on virheellinen ja rejectedValue-kentässä on annettu virheellinen arvo tai null-arvo, jos arvo puuttuu. Esimerkin 4 tikettilistauksessa tickets-muuttuja pitää sisällään taulukon, jossa on tiketeistä ennalta päätetyt tiedot. Viimeinen esimerkki kuvaa yhden tiketin JSON-muotoista esitystä. Esimerkeistä on karsittu tietoja, jotta esimerkit eivät veisi isoa tilaa. Etenkin tikettien JSON-esitysten, kuten esimerkin 5, tiedot riippuvat tikettitoteutuksesta ja voivat olla hyvin monimutkaisia.

Taustapalvelinta voidaan muokata siten, että SOAP-rajapinta palauttaa JSON-muotoisten merkkijonojen sijaan todellisia olioilmentymiä, kuten sähköpostitikettiluokan ilmentymän. Jos taustapalvelin palauttaisi ilmentymän rajapinnan kautta, olisi palautettavien ilmentymän luokan määrittelyyn tarpeellista lisätä huomautusmerkintä @XmlAccessorType(XmlAccessType.FIELD), jonka avulla CXF-liitännäinen osaa paljastaa luokan (Grails CXF Plugin). Kun taustapalvelin palauttaa esimerkiksi sähköpostitikettiluokan ilmentymän, etupalvelin ei saa tietoa ilmentymän perinnästä. Jos taustapalvelimen WSDL-dokumentissa olisi määritelty jonkin metodin palautusarvoksi sähköpostitikettiluokka, joka on paljastettu huomautusmerkinnän avulla, etupalvelin tunnistaisi palautetun ilmentymän sähköpostitiketiksi, muttei tietäisi, että tiketti perii tikettien kantaluokkaa. Samoin jos palautusarvo olisi merkitty kantaluokan ilmentymäksi ja palautettu ilmentymä olisi luotu sähköpostitikettiluokasta, etupalvelin ei kuitenkaan tunnistaisi palautettua ilmentymää sähköpostitiketiksi. Palautetuilta ilmentymältä puuttuisi myös GORM-järjestelmän mukaan tuomat metodit.

### 4.3.5 Sähköposti

Monimutkaisin osa taustapalvelinta on sähköpostin käsittelyyn liittyvä toiminnallisuus. En avaa sähköpostijärjestelmän toiminnallisuutta yksityiskohtaisesti, koska se ei liity tikettijärjestelmän ydintoiminnallisuuteen. Kerran tunnissa tai pyydettäessä taustapalvelin noutaa ja käsittelee sähköpostipalvelimelta haetut viestit. Jos viesteissä on vastaus järjestelmässä jo olevaan sähköpostiviestiin, lisätään sähköposti tikettiin ja tiketti merkitään lukemattomaksi sekä avataan jos se oli suljettu. Jos saapunut sähköposti on tuntematon, sähköpostista luodaan uusi tiketti. Järjestelmän toiminta on kuvattu kuviossa 3.




*KUVIO 3: Sähköpostin käsittelyprosessin vuokaavio.*

Sähköpostijärjestelmä hyödyntää Oraclen JavaMail-kirjastoa, joka mahdollistaa sähköpostiviestien lähettämisen ja hakemisen käyttäen monia eri protokollia (JavaMail 2011). JavaMail asennetaan lisäämällä JavaMail 1.4.4-julkaisun mukana tulleet jar-paketit mailapi.jar ja imap.jar Grails-sovelluksen lib-kansioon. Järjestelmän pitäisi toimia myös asentamalla ainoastaan mail.jar-paketti, sillä se Oraclen julkaisutietojen (JavaMail(TM) API 1.4.4 release NOTES: Protocol Providers 2011) mukaan sisältää mailapi.jar-paketin ja imap.jar-paketin luokat.

Sähköpostijärjestelmä on kehitetty tukemaan toistaiseksi ainoastaan IMAP-sähköpostitilejä. Kun järjestelmästä lähetetään vastaus sähköpostiin, merkitään lähetetyn sähköpos-

tin inReplyTo-otsikkotietoon (header) vastatun viestin IMAP-tunniste. Lähetetty sähköposti tallennetaan järjestelmään ja sähköpostin tunniste tallennetaan. Lähetetyn sähköpostin tunnistetta käytetään tunnistamaan sähköpostiin tulevat vastaukset, eli jos tikettijärjestelmästä lähetettyyn viestiin vastaa, viesti tunnistetaan inReplyTo-otsikkotiedosta ja lisätään automaattisesti oikeaan sähköpostitikettiin. Jos käyttäjä luo uuden sähköpostin eikä vastaa järjestelmästä lähetettyyn sähköpostiin, käyttäjän sähköpostiviestistä luodaan uusi sähköpostitiketti. Järjestelmän toimivuutta on vaikea arvioida ilman erillistä arviota siitä, että miten ihmiset yleensä toimivat vastatessaan sähköpostiin. Jos monet tekevät vastauksesta uuden sähköpostin, jossa ei ole viitteitä alkuperäiseen sähköpostikeskusteluun, tikettijärjestelmän lopullinen etupalvelin tulee kehittää siten, että sähköpostitikettejä pystyy selaamaan luontevasti ilman inReplyTo-kenttään luottavia tietoja. Tämä mahdollistaisi pitkiä sähköpostikeskusteluja sisältävien tikettien käsittelyn myös ilman sähköpostin otsaketietoihin luottamista.

Järjestelmässä käytettiin sähköpostiviestien lähettämiseen kehityksen aikana Googlen SMTP-palvelua. Järjestelmä voidaan konfiguroida käyttämään jotain toista sähköpostipalvelinta, mutta muita palvelimia ei testattu lukuun ottamatta paikallista sähköpostia. Kehityksen alussa käytin paikallista postfix-sähköpostipalvelinta. Paikalliset sähköpostit tulkittiin säännöllisten lausekkeiden (regular expression) avulla tiedostosta. Jätin paikallisen postin lukemiseen kehittämäni koodin järjestelmään kommentiksi jos tulevaisuudessa olisi tarpeellista tulkita paikallisen palvelimen sähköpostia. Jos jokin ohjelma samalla palvelimella lähettää sähköpostiviestejä, voidaan viesteistä luoda automaattisesti tikettejä. Esimerkiksi lokitiedostojen tulkintaan tehty komentosarja Logwatch voi lähettää paikallisella sähköpostilla palvelimen pääkäyttäjälle ilmoituksia (Logwatch 2011). Logwatchin viesteistä voitaisiin esimerkiksi luoda järjestelmään tikettejä ilmoitusten käsittelyyn.

### 4.3.6 Tiketin luominen

Tiketit luodaan joko liittämällä etupalvelimelta lähetetty parametritaulukko tikettityypin ilmentymään tai automaattisien prosessien kautta luomalla ilmentymä halutusta tiketti-luokasta. Taustapalvelimen tikettejä käsittelevä TicketHandlerService-palvelu vaatii, että parametritaulukossa on type-niminen parametri, joka määrittelee luotavan tiketin tyypin. Tämä tuo sen edun, että kaikki tiketit voidaan luoda käyttäen tehdasmallia. Tehdasmalli on toteutettu omaksi tikettejä luovaksi tehdasluokaksi, jota hyödyntämällä voidaan luoda tikettejä seuraavalla hyvin lyhyellä koodipätkällä, joka soveltuu kaikkiin tikettityyppeihin ja parametreihin sillä oletuksella, että parametritaulukon muuttujanimet vastaavat tikettiluokan nimiä.

```
TicketThread ticket = ticketFactoryService.getInstance(params.type)
                                          .getClass()
                                          .newInstance(params)
```

Type-parametrin pohjalta taustapalvelin luo tehdasluokan avulla uuden ilmentymän tyypin määrittelemästä tikettiluokasta, jolle annetaan etupalvelimen parametritaulukko. Parametritaulukon parametrien tulee olla nimetty täsmälleen samoin kuin luotavan tiketinluokan muuttujat, joten etupalvelimen tulee tuntea tikettityypit ja tikettityyppien muuttujat nimeltä.

Taulukko 1 on esimerkki etupalvelimelta tulleesta parametritaulukosta, jonka pohjalta taustapalvelin voisi luoda palautetiketin, jos palautetikettiluokan parametrien nimet vastaavat parametritaulukon parametrien nimiä. Palautetiketillä voi olla myös muita muuttujia, joita ei käytetä jokaisessa tiketissä, kunhan aina vaaditut muuttujat ovat asetettuina ja läpäisevät tarkastuksen.

*TAULUKKO 1. Esimerkki parametritaulukosta.*

| **Parametrin nimi** | **Parametrin arvo** |
|---|---|
| type | feedback |
| content | Palautelomakkeen leipäteksti |
| email | asiakas@esimerkki.xyz |
| wantsReply | true |

Mikäli uuden tiketin luominen ei onnistu, taustapalvelin lähettää virheet etupalvelimelle errorService-palveluluokan avulla. Virheitä voi tulla monista syistä, esimerkiksi jos jokin vaadittu lomakekenttä jätetty tyhjäksi tai jos jokin tekstikenttä sisältää liikaa tietoa.

Tiketit voivat saada alkunsa myös automaattisista prosesseista. Automaattinen prosessi voi olla aikaan tai tapahtumaan sidottu. Aikaan sidottu prosessi suoritetaan tiettyinä ajankohtina, esimerkiksi kerran tunnissa. Automaattisiin toimintoihin ei ole taustapalvelimella mitään erityistä toiminnallisuutta, koska Grailsille saatavilla olevalla Quartz-liitännäisellä (Nebolsin 2011) voidaan toteuttaa tarvittavat ajastetut tehtävät ja koska tapahtumaan sidotut prosessit vaativat aina manuaalisen lisäyksen. Toteutin esimerkkinä ajastetusta tehtävästä kerran tunnissa suoritettavan sähköpostien haun.

## 4.4 Etupalvelimen toteutus

### 4.4.1 Etupalvelinliitännäinen

Tuotantoympäristössä tikettijärjestelmä olisi integroituna toisiin moduuleihin, kuten käyttäjähallintaan ja mahdollisesti tiedostohallintaan, joten etupalvelimen toteutukset tulevat muuttumaan paljon sovelluksesta toiseen. Etupalvelinliitännäisen mukana tulee graafinen käyttöliittymä, taustapalvelimen käyttöä helpottava palvelu, esimerkkinäkymät tikettien luomiseen, etsimiseen ja hallintaan, esimerkkikäännöksiä tikettijärjestelmän automaattisista viesteistä ja pieni kirjasto tikettien esittämiseen ja muokkaamiseen liittyviä tunnisteita. Etupalvelinliitännäinen vaatii toimiakseen ainoastaan WSClient-liitännäisen.

### 4.4.2 Tikettienhallintapalvelu

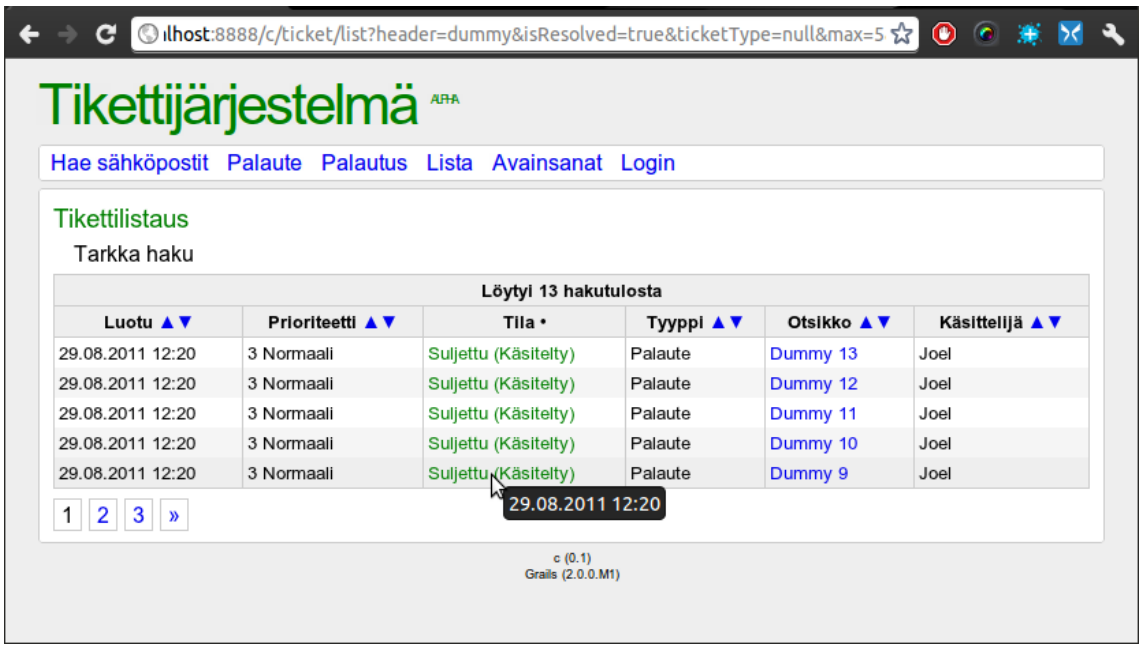
Etupalvelin keskustelee taustapalvelimen kanssa käyttäen kehittämääni TicketService-palveluluokkaa. Palvelu kokoaa tikettien hallintaan liittyvät toiminnot yhteen kokonaisuuteen, samalla piilottaen tiketinhallinnan tiedonvälitystavan ja WSClientin hyödyntämisen muilta sovelluksen osilta.

TicketService hyödyntää WSClient -liitännäistä, joka mahdollistaa taustapalvelimen kanssa kommunikoinnin näkymättömästi. WSClient-liitännäinen luo taustapalvelimen WSDL-dokumentin perusteella välityspalvelimen (proxy) SOAP-kutsujen käyttämiseen. Välityspalvelimella on taustapalvelimen ohjelmointirajapinnan toiminnot käytettävissään siten, että toimintoja voi kutsua kuin ne olisivat välityspalvelimen metodeja. WSClient muodostaa metodikutsusta SOAP-pyynnön, joka lähetetään taustapalvelimelle. Taustapalvelin vastaa pyyntöön ja lähettää SOAP-muodossa vastauksen etupalvelimelle, minkä jälkeen WSClient purkaa vastaanotetun vastauksen SOAP-kirjekuoresta ja palauttaa vastauksen välityspalvelimelle tehdyn metodikutsun vastauksena. Koko tapahtumaketju toimii CXF- ja WSClient-liitännäisien avulla automaattisesti.

TicketServicen metodit ovat yksinkertaisia, eivätkä sisällä paljoa logiikkaa välityspalvelinkutsujen lisäksi. Kutsujen lisäksi TicketService muuntaa taustapalvelimelta vastaanotetut JSON-muotoiset merkkijonot Grailsin oman JSON-muunnostyökalun avulla JSONObject-luokan ilmentymiksi. Ilmentymien avulla on helppo hyödyntää taustapalvelimen lähettämiä viestejä riippumatta viestien sisällöistä. Taustapalvelimen vastauksesta muodostetusta JSONObjectista voidaan esimerkiksi tarkistaa lähettikö taustapalvelin virheitä kutsumalla JSONObjectin Errors-muuttujaa.

### 4.3.3 Käyttöliittymä

Etupalvelinta käytetään graafiselta käyttöliittymältä, jota voi tarkastella web-selaimessa. Käyttöliittymä määritellään kolmessa eri osassa; sivumalleissa, GSP-tiedostoissa ja kehittämissäni tunnisteissa. Mallit määrittelevät sovelluksessa jokaisen sivun perusrakenteen ja jokaiselle sivulle yhteiset elementit, usein malli määrittelee esimerkiksi otsakkeen, navigaatiolinkit, yhteisen JavaScript-koodin ja sivun alaosan. Käyttöliittymän avulla järjestelmää voidaan esitellä ja käyttöliittymän toimivuuden perusteella voidaan olla varmoja, että tikettijärjestelmästä on mahdollista luoda graafisesti tyydyttävä toteutus. Käyttöliittymä voi toimia myös pohjana tikettijärjestelmän tunnisteiden kehittämiselle ja erikoistettujen käyttöliittymien pohjana. Käyttöliittymä on nähtävillä kuviossa 4, jossa avoinna on tikettilistausnäkymä.




*KUVIO 4. Tikettijärjestelmän käyttöliittymä.*

Etupalvelimen käyttöliittymä on toteutettu siten, että toiminta simuloisi arvioni mukaan todennäköisimmin käytettyjä toimintoja. Kehitin käyttöliittymään myös joukon GSP-tunnisteita helpottamaan tulevaa kehitystä. Tunnisteiden avulla voi muuttaa päivämäärämerkintöjä helpommin luettaviin muotoihin, lyhentää merkkijonoja, tulostaa virheitä ja lisätä sivulle tiketinmuokkaustyökaluja.

### 4.4.4 Käyttäjät

Etupalvelinliitännäisen mukana tulee järjestelmän käyttäjää simuloiva käyttäjäluokka User. Toteutin luokan, koska en käyttänyt liitännäisen kehityksessä mitään tuotantokäytössä ollutta käyttäjätietokantaa, mutta tarvitsin jonkin tavan simuloida järjestelmään kirjautunutta käyttäjää. Tuotantojärjestelmään käyttäjäluokkaa ei sisällytettäisi.

Prototyyppi tukee ainoastaan admin-käyttäjän sisäänkirjautumista. Sisäänkirjautumisen avulla voi nähdä tiketeistä tietoja, jotka muuten olisivat piilotettuina, muokata tikettiä ja lisätä tiketteihin kommentteja. Toisin kuin tuotantokäytössä olisi, osaa etupalvelimen toiminnoista voi käyttää kirjautumatta kehitysversiossa. Tikettien listaus, tiketin tietojen katselu ja avainsanojen perusteella selaaminen on mahdollista myös kirjautumattomalle käyttäjälle, vaikka lopullisessa järjestelmässä tämä ei luultavasti olisikaan mahdollista. Yritin julkisten ja ei-julkisten toimintojen erolla esittää sen mahdollisuuden, että järjestelmässä voi olla eri tasoisia käyttäjiä, joista jotkut voisivat esimerkiksi vain selata, katsoa, ottaa omitukseen ja sulkea tikettejä ja joista toiset voisivat näiden lisäksi kommentoida, ottaa toisten tikettejä itselleen, määrätä tikettejä toisille ja sulkea toisten tikettejä.

Käyttäjäviittaukset järjestelmässä ovat toteutettu hyvin yksinkertaisesti merkkijonomuuttujiksi tietokantaluokkiin. Esimerkiksi muutoskokoelmaluokassa on omistajakenttä, johon on mahdollista tallentaa käyttäjäviittaukseksi tunnistenumero, käyttäjänimi tai sähköpostiosoite. Järjestelmään on mahdollista lisätä esimerkkitikettejä ja -kommentteja, joissa on erimuotoisia käyttäjäviittauksia, jotta voidaan varmistaa, että järjestelmä tukee kaikkia tarpeellisia käyttäjäntunnistuskeinoja. Etupalvelimen tai taustapalvelimen täytyisi aina erikseen yhdistää käyttäjäviittaukset käyttäjätiliin, jotta todellinen käyttäjäintegraatio saataisiin aikaiseksi. Tiliin yhdistämistä ei toteutettu, koska käyttäjäviittauksien lopullinen muoto päätettiin jättää lopullisen sovelluksen päätettäväksi.

# 5 JATKOKEHITYS

## 5.1 Jatkokehityssuunnitelma

Uusi tikettijärjestelmä ei ole vielä valmis tuotantokäyttöön. Järjestelmän rakenne tulisi tarkastaa ongelmakohtien varalta ja sovelluksen koodi olisi hyvä käydä läpi jonkin yleisen parannuslistan kanssa. Parannuslista voisi olla lista sääntöjä, ohjenuoria ja toimintatapoja, mikä oltaisiin koottu aikaisempien kokemusten pohjalta. Listassa voisi olla sääntöjä kommentoinnista, koodaustyylistä ja erilaisista refaktoroinneista, jotka ovat yleisiä Groovy- ja Grails-sovelluksissa.

Mielestäni tärkein askel tikettijärjestelmän kehitykselle on järjestelmäintegraation testaus. Tikettijärjestelmä tulee aina toimimaan jonkun isäntäsovelluksen sisällä, joten on tärkeää, että järjestelmästä löydetään kaikki mahdolliset ongelmat integraatioon liittyen mahdollisimman aikaisin. Jos esimerkiksi käyttäjien integraatio tuleekin toteuttaa jollakin erilaisella tavalla kuin olin suunnitellut, suuret muutokset tulisi tehdä mahdollisimman aikaisin kehitysprosessissa.

On myös mahdollista, että olen arvioinut jonkin järjestelmältä vaaditun toiminnallisuuden väärin ja tämän takia järjestelmästä puuttuu jokin toiminnallisuus tai jokin toiminnallisuus voi olla vajavainen. Uskon, että myös nämä mahdolliset ongelmakohdat löytyvät helpoiten järjestelmäintegraatiota testatessa.

Jokin sovelluksen arkkitehtuuria koskeva päätökseni saattaakin tulla olemaan jatkokehityksen tai sovelluksen hyödyntämisen kannalta ongelmallinen. Näihin ennalta arvaamattomiin ongelmiin olen varautunut toteuttamalla järjestelmän mahdollisimman modulaariseksi siten, että järjestelmän eri osat ovat mahdollista vaihtaa toisiin muutoksen vaikuttamatta järjestelmän kokonaistoimintaan.

## 5.2 Taustapalvelimen jatkokehitys

Taustapalvelimen sähköpostijärjestelmä on järjestelmän monimutkaisin osa ja oletettavasti myös herkin virheille. Ensimmäinen sähköpostijärjestelmän testauskohde on sähköpostipalvelimen vaihto. Toteutin sähköpostin hakemisen tukemaan kehityksen aikana käyttämääni Gmail-tiliä, joten joltakin toiselta sähköpostipalvelimelta tulevat viestit voivat esimerkiksi käyttää erilaisia otsikkotietoja, vaikkakin otsikkotietojen pitäisi olla pitkälti standardisoituja. Sähköpostiviestit haetaan IMAP-palvelimelta (Internet Message Access Protocol ), joten jos jatkossa käytössä onkin esimerkiksi POP- (Post Office Protocol) tai Exchange-palvelin, sähköpostin hakeminen saattaa vaatia muutoksia. Ennen järjestelmän käyttöönottoa olisi tärkeää myös testata sähköpostin käsittelyä otoksella oikeita sähköposteja vanhasta tikettijärjestelmästä. Samalla olisi otollinen tilaisuus kerätä sähköpostin otsaketietoja ja analysoida tunnettujen ja tuntemattomien otsakkeiden yleisyyttä. Erityisen tarkasti tulisi tarkastella pitkien sähköpostikeskustelujen käsittelyä, jotta voidaan varmistua, että saapuneet ja lähetetyt sähköpostit voidaan varmasti yhdistää oikeaan tikettiin. Sähköpostin lähettäminen ja vastaanottaminen voitaisiin toteuttaa myös omaksi Grails-liitännäisekseen.

Tällä hetkellä taustapalvelinta voidaan käskeä hakemaan uudet sähköpostit etupalvelimen lähettämällä komennolla. Samanaikaiset komennot voivat kuitenkin johtaa virhetilanteeseen, eli jos automattinen sähköpostien haku on käynnissä ja käyttäjä yrittää käynnistää sähköpostin haun manuaalisesti, tuloksena voi olla virhetilanne. Samoin monen käyttäjän yrittäessä käynnistää sähköpostihaku samaan aikaan on mahdollista päätyä virhetilanteeseen. Kokeilin kymmenen samanaikaisen sähköpostihaun aloittamista, mistä ei tullut virhetilannetta, mutta valtava osa taustapalvelimen tehosta meni täysin turhaan. Tämä voidaan korjata lisäämällä sähköpostia hakevaan luokkaan jokin luokan tilaa ilmaiseva muuttuja, jonka avulla käyttäjälle voidaan ilmoittaa, jos sähköpostin käsittely on kesken. Näin vältetään turhat yritykset ja mahdolliset virhetilanteet monien kutsujen samanaikaisessa prosessoinnissa.

Tulevan tiedon käsittelyä olisi mahdollista parantaa erottamalla taustapalvelimen rajapinta ja logiikka yhdellä ylimääräisellä abstraktiokerroksella. Taustapalvelimen TicketAPIService-palvelun ja TicketHandlerService-palvelun väliin voisi luoda uuden palvelun, joka käsittelisi etupalvelimen pyyntöjä. Uuden palvelun tarkoituksena olisi, että

TicketHandlerServicestä voitaisiin erottaa JSON-muunnos, muuntaen ThicketHandler-Service agnostiseksi välitetyn tiedon muotoon. TicketAPIService prosessoisi pyyntöjä uuden palvelun kautta. Tämä mahdollistaisi uusien viestinvälitysformaattien käyttöön ottamisen siten, että taustapalvelimelle voisi lisätä rajapintoja jokaisellle haluttavalle tiedonvälitystavalle. Yksi rajapinta voisi käsitellä XML-muotoista tietoa välittäviä SOAP-pyyntöjä, toinen JSON-muotoista tietoa välittäviä ja kolmas esimerkiksi REST-pyyntöjä. Pyynnöt välitettäisiin uudelle käsittelevälle palvelulle, joka muuttaisi pyynnön TicketHandlerService-kutsuksi. Uutta tasoa ei toteutettu järjestelmään, koska tarvetta eri rajapintaformaateille ei ole kartoitettu.

## 5.3 Etupalvelimen jatkokehitys

Järjestelmän käyttöliittymänä etupalvelin ei ole läheskään valmis, tekemäni käyttöliittymä soveltuu järjestelmän esittelemiseen, muttei varsinaiseen hyödyntämiseen. Käyttöliittymää voisi kehittää käyttämään enemmän malleja (template) ja viitteitä (tag) sivujen koostamiseksi. Tällä tavoin helpotetaan järjestelmän käyttöönottoa sovelluksissa, jotka vaativat, että tikettijärjestelmä seuraa tarkalleen jotain tiettyä ulkoasulinjausta. Tämän lisäksi kehitystä vaativat etupalvelimen osa-alueet löytyvät varmasti järjestelmäintegraatiota testausta. Esimerkiksi käyttäjäintegraation toteutusta on vaikeaa kehittää ilman järjestelmäintegraation testausta.

## 5.4 Tiedonvälityksen jatkokehitys

Saattaa olla, että järjestelmien välillä olisi järkevämpää siirtää oikeita olioita eikä niiden JSON-muotoisia esityksiä. Tämä toiminnallisuus voi olla mahdollista saada esimerkiksi siten, että taustapalvelimen GORM-luokat eriytettäisiin omaksi liitännäisekseen, joka voitaisiin lisätä myös etupalvelimelle. Täten voidaan varmistua, että molemmilla on varmasti sama versio luokkamäärityksistä ja luultavasti myös kehitys on helpompaa kun luokista ei tarvitse muodostaa jar-pakettia tai kopioida käsin luokkatiedostoja. Ongelmaksi GORM-luokkien eriyttämisessä nousee isäntäsovellukselle räätälöidyt tikettityypit. Jos järjestelmässä olisi räätälöityjä tikettityyppejä, määritelmät pitäisi saada jotenkin

molemmille palvelimille. Tämän ongelman takia en toteuttanut GORM-luokkia itsenäiseksi liitännäiseksi.

JSON-muotoisien tekstijonojen lähettäminen XML-muotoisessa SOAP-kirjekuoressa vaikuttaa turhalta tekniikoiden sekoitukselta, mutta toteutuksen nopeus ja helppous lievittää tätä ongelmaa. Vaihtoehtojen tarjoaminen palvelinten välisessä tiedonsiirrossa toisivat järjestelmää käyttäville etupalvelimille joustavuutta toteutukseen ja tekniikoita kehitettäessä järjestelmästä todennäköisesti löytyisi muitakin parannettavia kohtia.

Palvelinten välillä välitetyn tiedon muoto on myös yksi mahdollinen kehityskohta. Virheet ja osa olioista muodostetaan JSON-esitykseksi itse kehittämieni sääntöjen mukaan, jotka on tarpeellista tuntea ennen tietojen hyödyntämistä. Tämä rajoittaa taustapalvelimen kehitystä, koska JSON-muoto on tiukasti sidottu (coupled) taustapalvelimen toimintaan, joten tiedonpalautusmuotojen kehittäminen ilman JSONia tulisi olemaan työlästä. Jos järjestelmää laajennetaan tukemaan vaihtoehtoisia viestinvälitystekniikoita, myös ErrorService-luokka tulee luoda uudelleen. ErrorServicen avulla GORM-virheistä voidaan luoda automaattisesti JSON-muotoiltuja virheviestejä. Jos järjestelmää laajennetaan käsittelemään esimerkiksi XML-viestejä, myös virheviestien tulisi olla XML-muodossa ja ErrorService pitää päivittää. Virhepalvelu pitää joko jakaa useampaan virheitä muuntavaan luokkaan tai sen toimintaa pitää laajentaa tukemaan haluttuja virheilmoitusmuotoja.

Tällä hetkellä etu- ja takapalvelimen välissä ei ole mitään varmennusta, koska järjestelmä on suunniteltu toimimaan vain turvallisissa sisäverkoissa. Järjestelmän luotettavuutta voitaisiin lisätä ja järjestelmä voitaisiin avata julkiseksi, jos palvelimille toteutettaisiin jokin varmennusjärjestelmä viestinvälitykseen. Viestien varmennus voisi toimia esimerkiksi siten, että taustapalvelimelta saisi tunnusta ja salasanaa vastaan tilapäisen valtuuden (token), joka lähetettäisiin tunnistautumisen vaativassa kutsussa ja tarkastettaisiin taustapalvelimella. Taustapalvelimen rajapinnan julkistaminen vaatisi järjestelmän testauksen tietoturva-aukkojen ja mahdollisen korkean liikenteen aiheuttaman rasituksen varalta.

## 5.5 Järjestelmäintegraation jatkokehitys

Järjestelmäintegraation kehityksellä tarkoitan niiden osien kehitystä, jotka mahdollistavat tikettijärjestelmän sisällyttämisen isäntäsovellukseen. Sisällyttäminen isäntäsovellukseen on järjestelmän jatkokehityksen tärkein kohta, sillä tikettijärjestelmää integroitaessa isäntäsovellukseen tulee varmasti ilmi tekijöitä, joita tikettijärjestelmän kehityksessä ei otettu huomioon. Taustapalvelimella järjestelmäintegraation ei pitäisi tuottaa ongelmia, koska taustapalvelimella tikettiliitännäinen ei toimi minkään isäntäsovelluksen toiminnallisuuden kanssa. Taustapalvelimella näkisin, että integraation mukana tulevista ongelmista todennäköisin on nimeämisongelmat, jos liitännäisen mukana tulee esimerkiksi saman nimisiä palveluluokkia kuin isäntäsovelluksella. Etupalvelimella lähes varmasti ainakin käyttäjien käsitteleminen järjestelmässä ja järjestelmän grafiikan mukauttaminen isäntäsovelluksen ulkoasuun tuovat haasteita integroitaessa liitännäistä isäntäsovellukseen. Etupalvelimen vaatimat muutokset saattavat vaatia myös taustapalvelimen kehitystä jos esimerkiksi käyttäjäviittaukset tulevat muuttumaan.

Käyttäjäintegraatiota kehitettäessä järjestelmä tulee luultavasti muuttumaan eniten jatkokehityksen aikana. Taustapalvelimen käyttäjäviittauksien pitäisi soveltua lähes minkälaiseen viittaustapaan tahansa, mutta etupalvelimen käyttäjätilit eivät vastaa lainkaan todellisia käyttäjiä. Tällä hetkellä suunnittelin järjestelmän käyttäjäintegraation siten, että taustapalvelin tallentaisi käyttäjäviittauksia ja etupalvelin yhdistäisi käyttäjäviittaukset käyttäjien varsinaisiin tietoihin. Taustapalvelimelta saaduista tiedoista pitäisi siis erottaa käyttäjäviittaus, oletettavasti jokin tunnistenumero, jonka avulla voitaisiin hakea käyttäjän nimi, sähköpostiosoite tai muita tietoja.

Tikettijärjestelmän graafinen integraatio isäntäsovellukseen tuo mukanaan monia haasteita. Nykyisellään tikettijärjestelmä tuo mukanaan isäntäsovelluksesta täysin erillisen käyttöliittymän näkymineen. Ihannetilanteessa etupalvelin osaisi etsiä automaattisesti isäntäsovelluksen oletusmallin ja hyödyntäisi sitä pohjana tikettijärjestelmän graafiselle esitykselle. Sovelluksen pohja voisi olla myös määriteltävissä konfiguraatiotiedostossa. Tämä integraatio tosin vaatisi sen, että isäntäsovellukset aina käyttäisivät sivupohjaa, jossa ei olisi samoja nimiä kuin tikettijärjestelmän tunnistetiedoissa. Ongelmia saattavat aiheuttaa esimerkiksi jos tikettilistauksissa on käytössä jokin sama tunnistenimi (id), kuin isäntäsovelluksen oletuspohjassa. Nimitörmäykset voisivat johtaa rikkinäisiin Ja-

vaScript-komentosarjoihin ja CSS-tyylien (Cascading Style Sheets) sekoittumiseen. En toteuttanut graafista integraatiota isäntäsovellukseen, koska käytössäni ei ollut todellista isäntäpalvelinta ja koska tulevista isäntäsovelluksista olisi tehtävä tutkimus miten tiket-tijärjestelmän olisi järkevää viitata isäntäsovelluksien resursseihin ja sivupohjiin.

Järjestelmä tulisi myös luultavasti käyttöön isäntäsovelluksen joihinkin Internet-lomakkeisiin siten, että lomakkeen pohjalta luotaisiin uusi tiketti. Tälle toiminnallisuudelle on olemassa jo hyvä tuki, mutta integraatiota tulisi kokeilla todellisessa isäntäsovelluksessa ennen kuin järjestelmää voidaan kehittää.

# 6 JOHTOPÄÄTÖKSET JA POHDINTA

Toteutetusta järjestelmästä tuli itsenäinen kokonaisuus, joka voidaan liittää toisiin Grails-sovelluksiin. Tikettijärjestelmän prototyyppi luovutettiin toimeksiantajalle 30.6, kuten oli sovittu. Tikettijärjestelmän ydintoiminnallisuuksien ylläpitäminen on erillään isäntäsovelluksesta ja eri toteutuksilla voi olla erityyppisiä tikettejä. Paranneltavaa olisi ollut tikettijärjestelmän ulkoasun sovitettavuudessa isäntäsovellukseen sopivaksi. Tällä hetkellä tikettijärjestelmän sivujen ulkoasu täytyy tehdä uudelleen lähes kokonaan, jotta se saadaan sopimaan erilaisiin ulkoasuihin, mutta tikettijärjestelmän nykyistä ulkoasua ja sen rakennustapaa voidaan hyödyntää missä tahansa sovelluksessa.

Käyttäjätietokannan integraation onnistumista on vaikea arvioida ilman tietoa käyttäjäviittausten käytännön toteutuksesta. Esimerkiksi tikettijärjestelmä voi tehdä tikettilistauksen tiketin omistavan käyttäjän mukaan. Käyttäjän loput tiedot voisi hakea listaukseen käyttäjätietokannasta nimen mukaan, joten tuki integraatiolle on olemassa.

Kaikki järjestelmän vaatimukset saatiin täytettyä mielestäni tyydyttävällä tavalla. Järjestelmä on itsenäinen moduuli, joka tukee monia erityyppisiä tikettejä. Järjestelmän isäntäsovellukseen pystyy lisäämään tikettityyppejä, järjestelmä on jaettu kahdelle palvelimelle joista toinen käsittelee tikettejä ja toinen sisältää järjestelmän käyttöliittymän. Järjestelmää voidaan ylläpitää ja päivittää erillään muista sovelluksista ja järjestelmä tukee käyttäjäviittauksia. Järjestelmä saatiin myös luomaan tikettejä sähköpostista ja lähettämään sähköpostia vastauksena tiketin sähköpostiviesteihin.

Opinnäytetyöraportti onnistuu mielestäni tehtävässään korkean tason dokumentaationa järjestelmän jatkokehityksen tueksi. Esittelen käyttämäni tekniset ratkaisut ja kuvaan järjestelmän korkealta tasolta ja niin parhaaksi katsoessani myös läheltä toteutusta. Tikettijärjestelmä osoittautui monimutkaisemmaksi ja laajemmaksi kokonaisuudeksi, kuin olin kuvitellut ja aiheen olisi voinut jakaa useiksi itsenäisiksi opinnäytetöiksi, esimerkiksi palvelinten viestinvälityksestä ja sähköpostin käsittelemisestä olisi saanut hyvät aiheet. Jos aihetta olisi kuitenkin rajattu pienempään järjestelmän osa-alueeseen järjestelmää ei olisi saatu kehitettyä yhtä pitkälle kuin saatiin.

LÄHTEET

All Plugins. 2011. Luettu 19.7.2011. http://www.grails.org/plugin/category/all.

Converters Reference. 2011. Tarkastettu 5.9.2011. http://grails.org/Converters+Reference.

Crum, R. 2011. Grails CXF plugin. Tarkastettu 19.7.2011. http://grails.org/plugin/cxf/.

cURL. 2011. Tarkastettu 19.7.2011. http://curl.haxx.se/.

Documentation. 2011. Tarkastettu 5.9.2011. http://grails.org/Documentation.

Dynamic Groovy. 2011. Tarkastettu 19.7.2011. http://groovy.codehaus.org/Dynamic+Groovy.

Factory Pattern. 2011. Tarkastettu 30.8.2011 http://www.oodesign.com/factory-pattern.html.

Fowler, A. 2011. A Swing Architecture Overview: Roots in MVC. Tarkastettu 5.9.2011. http://java.sun.com/products/jfc/tsc/articles/architecture/#roots.

Frequently Asked Questions: Misc. 2011. Tarkastettu 5.9.2011. http://grails.org/FAQ#Misc.

Grails 1.4.0.M1 Release Notes. 2011. Tarkastettu 5.9.2011. http://www.grails.org/1.4.0.M1+Release+Notes

Grails Reference Documentation. 2011. Tarkastettu 19.7.2011. http://www.grails.org/doc/latest/guide/single.html.

Groovy Home. 2011. Tarkastettu 30.8.2011. http://groovy.codehaus.org/.

Installation from Download. 2011. Tarkastettu 5.9.2011. http://grails.org/Installation.

Introducing JSON. 2011. Tarkastettu 19.7.2011. http://www.json.org/.

JavaMail(TM) API 1.4.4 release NOTES. 2011. Tarkastettu 19.7.2011. http://www.oracle.com/technetwork/java/javaee/notes-220865.txt.

JavaMail. 2011. Tarkastettu 30.8.2011. http://www.oracle.com/technetwork/java/javamail/index.html.

Ledbrook, P & Smith, G. 2009. Grails In Action. Greenwich: Manning Publications Co.

Logwatch. 2011. Tarkastettu 19.7.2011. https://help.ubuntu.com/community/Logwatch.

Nebolsin. S. 2011. Quartz Plugin. Tarkastettu 5.9.2011. http://grails.org/plugin/quartz.

Ogbuji, U. 2000. Using WSDL in SOAP applications. Tarkastettu 19.7.2011. http://www.ibm.com/developerworks/library/ws-soap/?dwzone=ws.

Puchbauer. S. 2009. Grails: JSONP callback without id and class in JSON file. Tarkastettu 5.9.2011. http://stackoverflow.com/questions/1700668/grails-jsonp-callback-without-id-and-class-in-json-file/1701258#1701258.

Puchbauer, S. 2011a. DomainClassMarshaller. Tarkastettu 5.9.2011. http://grails.org/doc/latest/api/org/codehaus/groovy/grails/web/converters/marshaller/xml/DomainClassMarshaller.html.

Puchbauer, S. 2011b. Grails converters plugin. Tarkastettu 5.9.2011. http://grails.org/plugin/converters.

Quick Reference: hasErrors. 2011. Tarkastettu 19.7.2011. http://grails.org/doc/latest/ref/Domain%20Classes/hasErrors.html.

Quick Reference: Service Usage. 2011. Tarkastettu 19.7.2011. http://grails.org/doc/latest/ref/Services/Usage.html.

Runtime vs Compile time, Static vs Dynamic. 2011. Tarkastettu 19.7.2011. http://groovy.codehaus.org/Runtime+vs+Compile+time,+Static+vs+Dynamic.

Scripts and Classes. 2011. Tarkastettu 19.7.2011. http://groovy.codehaus.org/Scripts+and+Classes

SOAP Body Element. 2011. Tarkastettu 19.7.2011. http://www.w3schools.com/soap/soap_body.asp.

SOAP Envelope Element. 2011. Tarkastettu 19.7.2011. http://www.w3schools.com/soap/soap_envelope.asp.

SOAP Header Element. 2011. Tarkastettu 19.7.2011. http://www.w3schools.com/soap/soap_header.asp.

SOAP Introduction. 2011. Tarkastettu 19.7.2011. http://www.w3schools.com/soap/soap_intro.asp.

Spring MVC Integration. 2011. Tarkastettu 5.9.2011. http://www.grails.org/Developer+-+Spring+MVC+Integration.

Submaranian, V. 2008. Programming Groovy: Dynamic Productivity for the Java Developer. Raleigh: Pragmatic Booshelf.

Sumerfield, E. 2008. Using the Delegating Meta Class. Tarkastettu 19.7.2011. http://docs.codehaus.org/display/GROOVY/Using+the+Delegating+Meta+Class.

What is Object Relational Mapping? 2011. Tarkastettu 19.7.2011. http://www.hibernate.org/about/orm.

Wikipedia MIME Content-type. 2011. Tarkastettu 19.7.2011. http://en.wikipedia.org/wiki/MIME#Content-Type.

LIITE 1

Esimerkki ohjelmointipinnan palautusarvoista.

1. JSON-esitys totuusarvosta

```
{"result":"true"}
```

2. JSON-esitys virheviestistä ilman kenttäviittausta

```
{
   "Errors":[
       {"message":"Lorem ipsum dolor sit amet."}
   ]
}
```

3. JSON-esitys virheviestistä, jossa eritellään virheen sisältänyt kenttä

```
{
   "Errors":[
       {"field":"email","rejectedValue":null,"code":"nullable"}
   ]
}
```

4. JSON-esitys kahden tiketin listasta (tietoja karsittu)

```
{
   "tickets":[
       {"id":15,"owner":"username","header":"Esimerkkitiketti 1"},
       {"id":22,"owner":"username","header":"Esimerkkitiketti 2"}
   ]
}
```

5. JSON-esitys tiketistä (tietoja karsittu)

```
{
   "ticket": {
       "id":2276,
       "class":"esimerkkitiketti",
       "dateCreated":"2011-12-24T080000Z",
       "keywords":{
           "content":"avainsana"
       },
       "priorities":[
           "1 Triviaali",
           "2 Alhainen",
           "3 Normaali",
           "4 Korkea",
           "5 Kriittinen"
       ],
       "priority":"2 Alhainen",
       "header":"Tiketin otsikko",
       "owner":"admin",
       "changesets":{
            "changeset": {
               "dateCreated":"2011-12-24T081312Z",
               "changes":[
                   {"comment":"Vaihdoin nämä arvot esimerkkiä varten"},
                   {
                        "field":"priority",
                        "oldValue":"3 Normaali",
                        "newValue":"2 Alhainen"
                   },
                   {"field":"owner", "oldValue":"", newValue:"admin"}
               ],
               "owner":"admin"
           }
       }
   }
}
```