Pekka Astala

goMob CMS

Metropolia Ammattikorkeakoulu

Insinööri (AMK)

Tietotekniikka

Insinöörityö

11.6.2014

Metropolia

Tekijä(t) Otsikko Sivumäärä Aika	Pekka Astala goMob CMS 38 sivua 11.6.2014
Tutkinto	Insinööri (AMK)
Koulutusohjelma	Tietotekniikka
Suuntautumisvaihtoehto	Ohjelmistotekniikka
Ohjaaja(t)	Auvo Häkkinen, Yliopettaja Niklas Nylund, CTO

Insinöörityön tavoitteena oli kehittää sisällönhallintajärjestelmä startup-yrityksen käyttöön. Yrityksellä oli tarve saada erityisesti mobiilisivustojen muokkaamiseen soveltuva järjestelmä, joka olisi yhteensopiva erään kolmannen osapuolen ohjelmiston kanssa. Yksikään markkinoilla jo olleista tuotteista ei täyttänyt näitä vaatimuksia.

Yritykselle toteutettiin kolmivuotinen ohjelmistoprojekti, joka sisälsi sisällönhallintajärjestelmän suunnittelun, kehityksen ja ylläpitoa. Projektin aikana yrityksen liiketoiminta kehittyi, ja järjestelmän alkuperäiset määrittelyt todettiin riittämättömiksi. Järjestelmää uusittiin ja laajennettiin vastaamaan uusia liiketoiminnan asettamia vaatimuksia.

Järjestelmä kehitettiin MVC- ja REST-arkkitehtuurien mukaiseksi. Ohjelmointikielenä käytettiin aluksi PHP:tä ja Zend-ohjelmistokehystä, mutta järjestelmästä tehtiin myöhemmin uusi versio Javan ja Play!-sovelluskehyksen avulla. Järjestelmän kehitetyksessä käytettiin apuna TDD- ja SCRUM -prosesseja.

Projektin tuloksena yritykselle syntyi sisällönhallintajärjestelmä, jonka ympärille yritys pystyi rakentamaan liiketoimintansa. Järjestelmän avulla rakennettiin mobiilipalveluita useille suurille kotimaisille asiakkaille julkiselta, yksityiseltä ja kolmannelta sektorilta. Omalla ohjelmistotuotteella yritys saavutti kilpailuetua sekä asiakkaiden että sijoittajien silmissä.

Kehitetty järjestelmä jää yrityksen kaupalliseksi omaisuudeksi. Siitä tuotettu raportti sisältää hyödyllistä tietoa sisällönhallintajärjestelmän kehitykseen tai vastaavaan ohjelmistoprojektiin osallistuville kehittäjille.

Avainsanat	Sisällönhallintajärjestelmä, Palvelinohjelmointi, CMS, MVC, REST

Author(s) Title Number of Pages Date	Pekka Astala goMob CMS 38 pages 11 June 2014
Degree	Bachelor of Engineering
Degree Programme	Information Technology
Specialisation option	Software Engineering
Instructor(s)	Auvo Häkkinen Niklas Nylund

The goal of this Bachelor's thesis was to develop a content management system for a technology startup. The company needed a system that was well-suited for creating and editing web sites intended for mobile devices. The system also had to be compatible with a certain third party software component and none of the available products fit the criteria.

The software project included design, development and maintenance of the system and spanned a time period of three years. During this time the business requirements for the product evolved and it became clear that the original product specifications had been inadequate. The system was altered and expanded to match the new requirements.

The content management system was developed according to MVC and REST architectures. The work was started with PHP and Zend software framework but the product was later ported to Java and Play! framework. The development process was based on TDD and SCRUM processes.

As a result of the project the company received a content management system on which it was able to base its business. The system was used to build mobile services for several large Finnish clients from public, private and the non-profit sector. Having a software product became a competitive advantage for the company when attracting new clients and investors.

The resulting system will remain a commercial and intellectual property of the company it was developed for. The report of its development can be used as a research material by developers who are getting involved in similar projects.

Keywords	Content Management System, Web Development, CMS, MVC, REST

Sisällys

Lyhenteet

AJAX	Asynchronous Javascript and XML. Tekniikka päivittää verkkosivua tai välittää tietoa palvelimelle ilman erillistä sivulatausta.
CRUD	Create, Read, Update & Delete. Kokoelma objekteille tehtäviä toimintoja.
JVM	Java Virtual Machine. Ympäristö, jolle mm. Java- ja Scala-kielinen ohjelmakoodi käännetään.
MVC	Model View Controller. Lähestymistapa, jolla ohjelma jaetaan osiin ja osien välinen kommunikaatio määritellään.
ORM	Object Relational Mapping. Objektin muuttujat määritellään vastaamaan tietokannan muuttujia niin, että objektin muokkaaminen muuttaa tietokannan arvoja automaattisesti.
REST	Representational State Transfer. Arkkitehtuuri HTTP-rajapinnoille.
RSS	Really Simple Syndication. XML-pohjainen verkkosyötemuoto.
TDD	Test Driven Development. Ohjelmistokehityksen prosessi, jossa testit kirjoitetaan ensin ja sitten ohjelmalogiikkaa kehitetään läpäisemään sillä testeillä määritetyt edellytykset.
UA	User Agent. HTTP-pyyntöjen mukaan liitettävä tietue, joka kertoo palvelimelle yksityiskohtia pyynnön lähettäneestä asiakasohjelmasta.
WYSIWYG	What You See Is What You Get. Sisällön muokkaamiseen käytetty käyttöliittymä, joka ei vaadi käyttäjältä merkittävää teknistä osaamista.

1 Johdanto

Tämä opinnäytetyö on raportti goMob Finland Oy:lle toteutetusta verkkosovelluksesta, goMob CMS:stä.

goMob Finland Oy on alkuvuodesta 2011 perustettu yritys, joka tuottaa mobiililaitteille, eli pääasiassa älypuhelimille ja taulutietokoneille suunnattuja sovelluksia ja verkkosivustoja. Yrityksen asiakkaita ovat pääosin suomalaiset suuret ja keskisuuret organisaatiot, mm. Finnmatkat, Fortum, Museoliitto, YIT ja Vattenfall.

Vuosina 2011-2012 goMob Finland Oy toteutti verkkosivustoja käyttäen apuna erästä kolmannen osapuolen järjestelmää. Kyseinen järjestelmä toimi goMobin palvelimen ja asiakasohjelman välissä siten, että se luki sivustojen rakenteen ja sisällön goMobin palvelimilta XML-muodossa ja loi sen perusteella varsinaisen HTML-sivun, joka tarjottiin asiakasohjelmalle.

goMob toteutti sivustot aluksi staattisina XML-rakennetiedostoina tai PHP-ohjelmina, jotka tuottivat XML-koodia. Hyvin pian ilmeni tarve päästää yrityksen asiakkaat muokkaamaan heille toteutettuja verkkosivustoja, joten goMob lähti kehittämään sisällönhallintajärjestelmää, joka tuotti normaalin HTML:n sijasta XML-sivustoja. Loppukesästä 2011 julkaistiin goMob CMS:n ensimmäinen versio.

goMob CMS:n alkuperäinen toimeksianto oli jopa naiivin vaatimaton: Järjestelmän piti tarjota asiakkaille helppo tapa tehdä pieniä sisältömuutoksia jo valmiille sivustoille. Ajan myötä sekä yrityksen asiakaskunta että liiketoiminnan painopiste ovat muuttuneet huomattavasti. Järjestelmältä edellytetty toiminnallisuus on laajentunut rajusti ja samalla sen kehitykseen käytetyt työkalut ovat vaihtuneet useasti.

Tässä dokumentissa esitellään goMob CMS ja sen kehitys. Raportissa kuvataan järjestelmän taustoja ja nykytilaa, tärkeimpiä kehityksessä käytettyjä työkaluja ja prosesseja, sekä järjestelmän eri osa-alueiden toimintaa ohjelmistotekniikan näkökulmasta. Koko järjestelmän toiminnasta tarjotaan yleiskuva ja eräät kehityksessä tehdyt valinnat, niiden perustelut ja toteutustavat esitellään perusteellisemmin.

2 Ohjelmistoprojektin eteneminen

goMob Finland Oy:n ensimmäiset työntekijät ja päärahoittaja olivat aiemmin työskennelleet digitoimistoissa ja seuranneet mobiilipalveluiden kehitystä Suomessa. Vuoden 2010 lopulla he totesivat, että yritykset alkoivat olla valmiita panostamaan mobiilimarkkinointiin, mutta alaan keskittyviä toimijoita oli maassa hyvin rajoitetusti. He perustivat uuden yrityksen, jonka tavoitteena oli yhdistää markkinoinnin ja teknologian osaamista.

goMob tarjosi alusta asti myös muita palveluita, mutta huomasi pian, että helpoin yrityksille myytävä tuote oli mobiilisivusto. Vuonna 2011 mobiililaitteilla surffaavien ihmisten määrä kasvoi kovaa vauhtia, mutta perinteisten verkkosivustojen selailu sen hetkisillä puhelimilla oli vielä työlästä ja responsiiviset sivustot olivat suhteellisen uusi ilmiö.

Yritykset olivat jo tottuneet ostamaan verkkosivustoja ja ymmärsivät niiden hyödyt, joten oli helppoa myydä konsepti erillisestä verkkosivustosta, jonka käyttöliittymä ja sisältö oli suunniteltu älypuhelinten käyttäjille.

2.1 Mobiililaitteiden erot

Jo vuonna 2011 käytössä olleiden mobiililaitteiden kirjo oli hyvin laaja. Vanhat feature phonet olivat yhä käytössä, älypuhelimet vaihtelivat iPhoneista halpoihin aasialaisiin merkkeihin, ja ensimmäinen iPad taulutietokone oli ilmestynyt jo vuoden 2010 keväällä.

Tämä ympäristö asetti tiettyjä haasteita mobiilisivustojen myynnille. Erikseen mobiilikäyttäjille suunnattujen sivustojen odotettiin toimivan muillakin kuin uusimmilla iPhoneilla. Toisaalta vanhalla feature phonella helppokäyttöinen sivusto näytti aivan liian yksinkertaiselta iPadillä.

Yleisin tapa lähestyä edellä kuvattua ongelmaa on tarjota kaikille laitteille sama HTML-sivu, joka on suunniteltu näyttämään erilaiselta eri laitteissa. Sivusto voi olla mm. taitettu niin, että navigaatioelementtien sijainti ja koko riippuu näytön leveydestä. Tämä voidaan suorittaa useilla eri tavoilla käyttäen apuna mm. CSS-tyylittelykielen Media Queries-tekniikkaa tai dynaamisia JavaScript-ohjelmanpätkiä.

goMob koki lähestymistavan ongelmalliseksi useista eri syistä. Osa siihen liittyvistä tekniikoista ei ollut vielä 2011 keväällä kehittynyt nykyiselle tasolleen. Sivusto, jonka eri asiakasohjelmat piirtävät näytölle eri tavalla, vaatii huomattavasti enemmän testausta

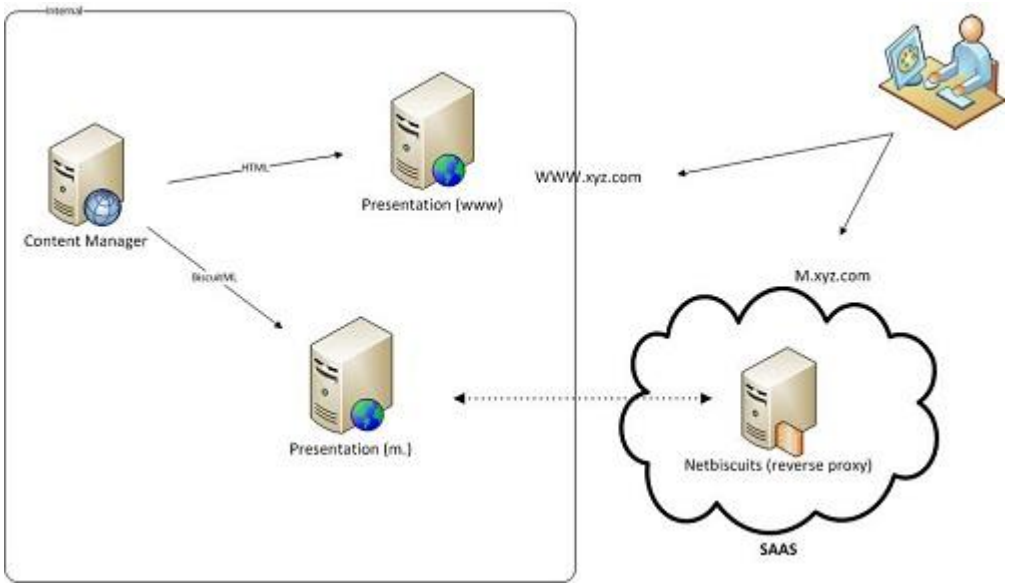
ja eri laitteilla ilmenevien ongelmien korjaamista. Yleisimmän lähestymistavan valitseminen olisi myös asettanut yrityksen samalle viivalle monen muun toimijan kanssa: kaikki tarjosivat samaan ongelmaan samaa ratkaisua.

2.2 Netbiscuits

goMob päätyi käyttämään Netbiscuits GmbH:n tarjoamaa Netbiscuits-verkkopalvelua. Kyseinen palvelu ei ole enää Netbiscuits GmbH:n päätuote, mutta sen jäänteet löytyvät kirjoitushetkellä yrityksen sivustolta tuotteistettuna nimellä Feature Phone Toolkit. [1.]

Netbiscuits tarjosi verkkosivuston toteuttajalle mahdollisuuden määrittää sivuston rakenteen ja sisällön yrityksen omalla, XML-pohjaisella BiscuitML-kielellä (kuva 1). Asiakasohjelman sivupyyntö ohjattiin Netbiscuitsin palvelimille, ja Netbiscuits luki pyydetyn sivun tiedot goMobin palvelimelta BiscuitML-muodossa. Palvelu päätteli asiakasohjelman User Agent -tiedosta mm. päätelaitteen tukemat ominaisuudet ja näytön koon ja loi sitten BiscuitML-määrityksen perusteella HTML-sivun.

Netbiscuitsin palvelu hoiti siis palvelinpuolella sivuston räätälöinnin päätelaitteen ominaisuuksien mukaan.



Kuva 1. Netbiscuitsin toimintatapa [2]

Tämä lähestymistapa käytännössä siirsi Netbiscuitsin henkilökunnan vastuulle varmistaa, että sivustot toimivat eri päätelaitteila. goMob sai kilpailuedun erilaisesta lähestymistavasta ja lupauksella, että sivustot toimivat tuhansilla erilaisilla mobiililaitteilla.

Netbiscuits oli goMobille muuten hyvä ratkaisu, mutta se ei tarjonnut goMobin asiakkaille mahdollisuutta muokata heille tuotettuja mobiilisivustoja. Yksinkertaisimmatkin sivustot olivat XML-tiedostoja, ja monimutkaisemmat olivat PHP-ohjelmakoodia, joka tuotti sivupyynnöstä riippuen erilaista XML:ää Netbiscuitsin BiscuitML-to-HTML tulkkia varten.

2.3 goMob CMS versio 1

Yritys totesi hyvin pian, että oli välttämätöntä tarjota asiakkaille tapa muokata heille toteutettuja mobiilisivustoja. Asiakkaille ei ollut välttämätöntä tarjota täysimittaista järjestelmää, jonka kautta he olisivat pystyneet mm. muuttamaan koko sivuston rakennetta tai ulkoasua. Kuitenkin yksinkertaisten sisältömuutoksien, kuten aukioloaikojen tai yhteystietojen päivitys, piti onnistua ilman tukipyyntöä goMobin suuntaan.

Järjestelmästä ei tehty alun perin yksityiskohtaista määrittelyä tai suunnitelmaa, vaan toteutus aloitettiin yllä kuvatulla tehtävänannolla. Työaika-arvio oli yhtä naiivi, kuin suunnitelmakin ja yhden henkilön oli tarkoitus käyttää järjestelmän kehitykseen noin kuukausi muun työn ohessa.

Jo parissa viikossa kävi ilmi, että alkuperäinen suunnitelma oli ollut aivan liian suppea ja järjestelmältä vaadittujen ominaisuuksien lista alkoi kasvaa kovaa vauhtia. Yrityksen johto ja myyjät päättivät, että palveluiden myyntiä – ja myöhemmin ylläpitoa - helpottaisi ominaisuuksien laajentaminen ainakin seuraavilla tavoilla:

- yksinkertainen web-analytiikka
- QR-koodien automaattinen generointi kaikista sivuista
- unohtuneen salasanan palautuksen automatisointi
- sivuston lomakkeiden datan koonti tietokantaan.

Järjestelmän ensimmäinen versio julkaistiin loppukesästä 2011, muutamia kuukausia kehityksen alkamisesta. Kun järjestelmän päällä alettiin julkaista yhä enemmän asiakassivustoja, ilmeni myös tarve integroida järjestelmä asiakkaiden tietokantoihin ja esim. RSS-muodossa saatuun dataan.

Yhdellä sivustolla haluttiin näyttää uusimmat uutiset asiakkaan RSS-syötteestä, toisella uutuustuotteet XML-rajapinnasta haettuna, kolmannella kaikki asiakkaan tuotteet

MySQL-tietokannasta ja neljännellä kartta, jossa näytettiin käyttäjän sijainti suhteessa XML-tiedostossa määriteltyihin toimipisteisiin.

Kaikki ominaisuudet saatiin toteutettua, mutta niitä lisättiin järjestelmään, jota ei oltu alun perin suunniteltu kasvamaan niin laajaksi. Staattisista HTML-sivuista koostuva käyttöliittymä alkoi muuttua raskaaksi käyttää, eivätkä prosessit mm. ohjelmakoodin dokumentoinnista tai muutosten viemisestä tuotantoon olleet riittävällä tasolla.

2.4 goMob CMS versio 2

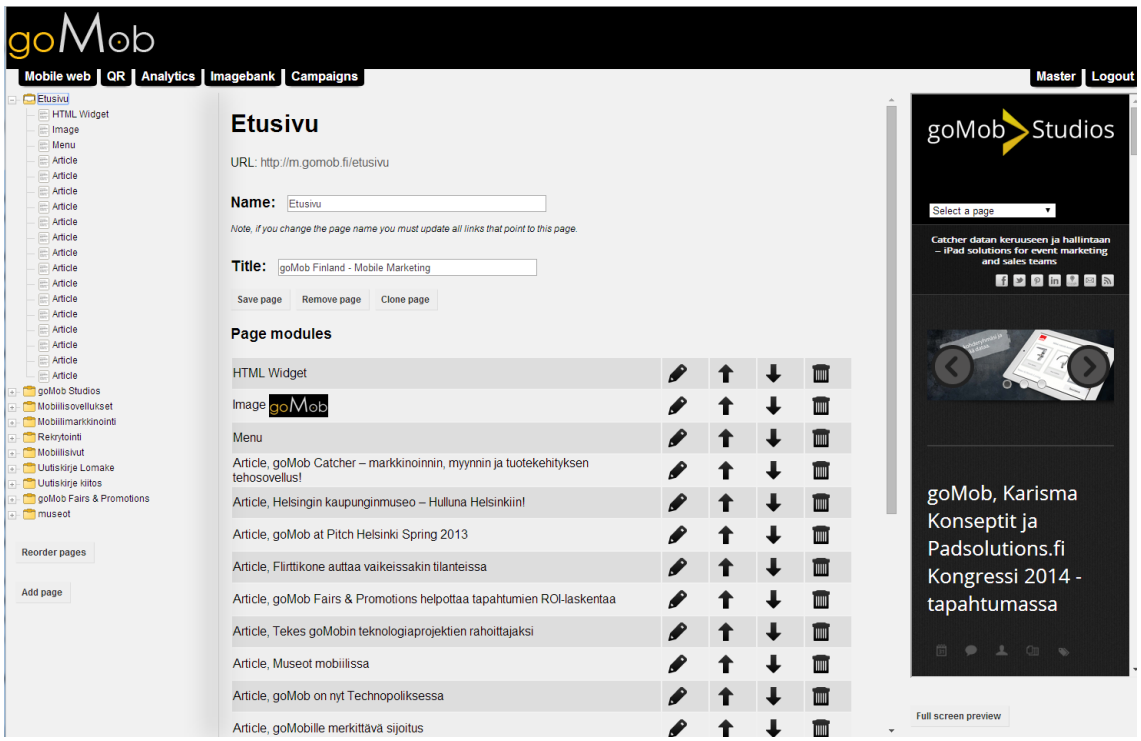
Keväällä 2012 päätettiin tehdä järjestelmästä uusi versio. Aiemmin yhden hengen projektiin liittyi uusi ohjelmistokehittäjä, ja tämä asetti uusia vaatimuksia ohjelmistotekniikan prosesseille. Samassa yhteydessä koettiin parhaaksi peruskorjata pois järjestelmään orgaanisesta kasvusta jääneet valuviat.

Alun perin PHP:n ja Zend-sovelluskehyksen päälle rakennettu järjestelmä toteutettiin uudelleen Javalla ja Play!-sovelluskehyksellä. Sovelluskehyksien MVC-toteutus oli samankaltainen, joten suurin osa ohjelmalogiikkaa pystyttiin siirtämään ohjelmointikielen vaihtumisesta huolimatta lähinnä pienillä syntaksimuutoksilla.

Ohjelmistotekniikan näkökulmasta uusi järjestelmä oli valtava edistysaskel:

- järjestelmässä aloitettiin laajamittainen yksikkötestaus
- Git-versiohallintatyökalu otettiin käyttöön
- tuotantopalvelimen ja -tietokannan varmuuskopiointi automatisoitiin
- tietokanta abstraktoitiin ORM:n taakse ja
- julkaisuprosessia parannettiin Makefile-pohjaisilla käskyillä.

Parempien prosessien lisäksi myös järjestelmän rakenteessa tapahtui muutoksia. Aiemmin käyttöliittymä oli koostunut PHP:llä luoduista HTML-sivuista ja kaikki tiedon lähetys palvelimelle oli tapahtunut perinteisten HTML-lomakkeiden avulla. Nyt järjestelmän näkymä jaettiin kahtia: palvelimelle luotiin kattava REST-rajapinta ja käyttöliittymästä tehtiin AJAX-pohjainen. Uutena työvälineenä otettiin käyttöön JavaScript-kehitystä helpottava Dojo-toolkit. Kuva 2 on kuvakaappaus täysin AJAX-pohjaisesta, mobiilisivustojen muokkaamiseen käytetystä goMob CMS:n käyttöliittymästä.



Kuva 2. AJAX-pohjainen goMob CMS v2 -käyttöliittymä

goMob CMS versio 2 julkaistiin loppukesestä 2012, tasan vuosi ensimmäisen version julkaisun jälkeen.

2.5 Netbiscuitsista luopuminen

Vuoden 2012 aikana alkoi käydä ilmi, että Netbsicuits ei enää vastannut yrityksen tarpeita. Yhä monipuolisemmat mobiilisivustot vaativat sellaisia rakenteita ja toiminnallisuuksia, joita oli erittäin työlästä luoda BiscuitML-määrityksen avulla. Lisäksi suurin järjestelmästä saatu hyöty – laitteella kuin laitteella varmasti hyvin toimiva sivusto – jouduttiin kyseenalaistamaan, kun goMobin asiakkaat raportoivat usein virhetilanteista, joista goMob joutui lähettämään korjauspyyntöjä Netbiscuitsille.

Netbiscuitsista päätettiin luopua, mutta goMobilla oli jo huomattava määrä asiakkaita, joiden sivustot oli rakennettu BiscuitML:llä ja jotka maksoivat yritykselle ylläpidosta ja pääsystä goMob CMS:ään. Nämä sivustot oli siis saatava pidettyä järjestelmän päällä ja muutettava BiscuitML-sivustoista normaaleiksi HTML-sivustoiksi ilman merkittäviä käyttökatkoja.

Asia otettiin huomioon versio 2:en toteutuksessa siten, että järjestelmään luotiin mahdollisuus luoda sisältöelementeille useita rakennemäärityksiä (käytännössä BiscuitML ja HTML-mallipohjat).

Teoriassa kaikista sisältöelementeistä pystyi luomaan HTML-versiot ja sen jälkeen yhden muuttujan arvoa vaihtamalla määrittämään, että samat sisältöelementit näytettäisiin samoilla sivuilla ja samassa järjestyksessä, mutta normaalina HTML:nä. Käytännössä jokainen sivusto vaati huomattavan määrän testausta, CSS-tyylittelyä ja muita toimenpiteitä.

Sivustoja muutettiin HTML-muotoon yksi kerrallaan, muun kehityksen ohella. Yrityksen alkuperäinen tavoite oli päästä eroon Netbiscuitsista vuoden 2013 ensimmäisen kvartaalin aikana, mutta käytännössä tämä onnistui vasta loppukesästä 2013.

Suurin osa suoraan Netbiscuitsiin viittaavasta ohjelmakoodista poistettiin järjestelmästä vuoden 2013 syksyllä. Teknisestä näkökulmasta BiscuitML-sivujen esittämiseen ja HTML-sivujen esittämiseen suunnitelluilla sisällönhallintajärjestelmillä ei ole merkittäviä eroja. Järjestelmän arkkitehtuurissa ei siis tapahtunut suuria muutoksia Netbiscuitsista luovuttaessa vaan valtaosa työstä liittyi sisältöelementtien mallipohjien muokkaamiseen.

2.6 goMob Catcher

Vuonna 2013 goMob Finland Oy julkaisi oman ohjelmistotuotteen, goMob Catcherin.

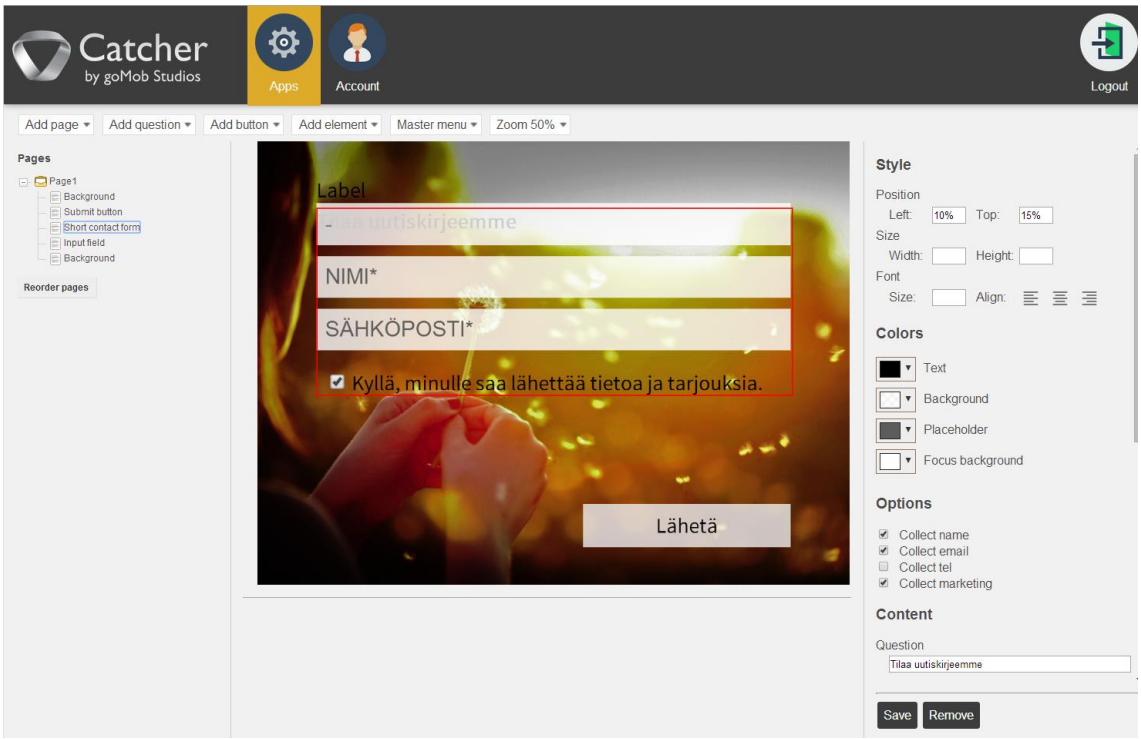
goMob Catcher antaa käyttäjän luoda goMob CMS:n avulla sivuston, jota voi selata erikseen App Storesta ladattavalla Catcher-sovelluksella. [3.] Latauksen jälkeen sivusto toimii, vaikka verkkoyhteys ei olisi käytettävissä ja mahdollisten interaktiivisten elementtien data kerätään laitteeseen odottamaan, että verkkoyhteys palautuu ja data voidaan lähettää goMobin järjestelmään.

goMobin järjestelmään kuuluu Remarketing-ominaisuus. Kun käyttäjä esimerkiksi jättää yhteystietonsa lomakkeeseen, joka on osa iPadille ladattua sivustoa, hänelle lähtee välittömästi etukäteen luotu sähköposti- tai tekstiviesti. goMobin järjestelmän kautta voi seurata, ketkä käyttäjät ovat avanneet kyseisen viestin.

goMob Catcher on tuotteistettu ensisijaisesti tapahtumamarkkinointiin. Kun yritys kerää messuilla kävijöiden yhteystietoja iPad-sovelluksella perinteisten paperilappujen sijaan, ohikulkijoiden aktivointi on helpompaa ja heille saadaan lähetettyä yhteydenotto heti, eikä vasta messujen jälkeen. Lisäksi yritys säästää aikaa saadessaan datan suoraan esimerkiksi MS Exceliin tai Salesforceen sen sijaan, että työntekijöiden täytyisi manuaalisesti kopioida järjestelmiinsä tiedot sadoista paperilapuista.

goMob Catcher -sivustoja hallitaan myös goMob CMS:n kautta. Koska kyseessä on iPadiin synkronoitavat verkkosivustot, niiden hallinta toimii sekä käyttäjän että järjestelmän näkökulmasta pääosin samalla tavalla kuin perinteisten mobiilisivustojen.

Sivujen muokkaukseen liittyvä käyttöliittymä on goMob Catcher -käyttäjätileillä erilainen kuin järjestelmän perinteisemmällä puolella. Sivuja muokataan WYSIWYG-editorilla (kuva 3), jossa käyttäjä saa raahata elementtejä haluamaansa kohtaan ruutua.



Kuva 3. Catcher-puolen WYSIWYG-editori

Tiettyjä perinteisten sivustojen puolella löytyviä osioita, kuten web-analytiikkaa, ei ole sellaisenaan näkyvissä goMob Catcher -tileillä. Eroista huolimatta Catcher-puolen käyt-

töliittymä käyttää samaa REST-rajapintaa kuin normaalien mobiilisivustojen muokkaukseen käytetty. Ainoa ero on selainohjelmoinnissa.

3 Teknisen kuvauksen rajaus ja esittely

goMob CMS -järjestelmän kehitykseen on kulunut useita henkilötyövuosia. Kyseinen kokonaisuus pitää sisällään pilvipalveluita ja kuorman testaus -mekanismeja hyödyntävän infrastruktuurin ja huomattavan määrän ohjelmakoodia sekä palvelimen että selaimen puolella. Raportille sallitun pituuden rajoissa ei ole mahdollista esitellä järjestelmän jokaista osa-aluetta yksityiskohtaisesti.

Tässä raportissa tarkastellaan tilannetta, jossa kävijä lataa verkkosivun, joka on rakennettu goMob CMS:n päälle. Sen lisäksi, että palvelinpuolen moduulit kuvataan yleisellä tasolla, raportissa tarkastellaan tapaustutkimuksena (eng. *case study*) mitä tapahtuu, kun erään tietyn asiakkaan sivu ladataan. Kyseessä on oikea asiakas, jonka nimi on jätetty mainitsematta, sillä sen käyttöön tässä raportissa ei ole saatu lupaa.

3.1 Tapaustutkimuksen kuvaus

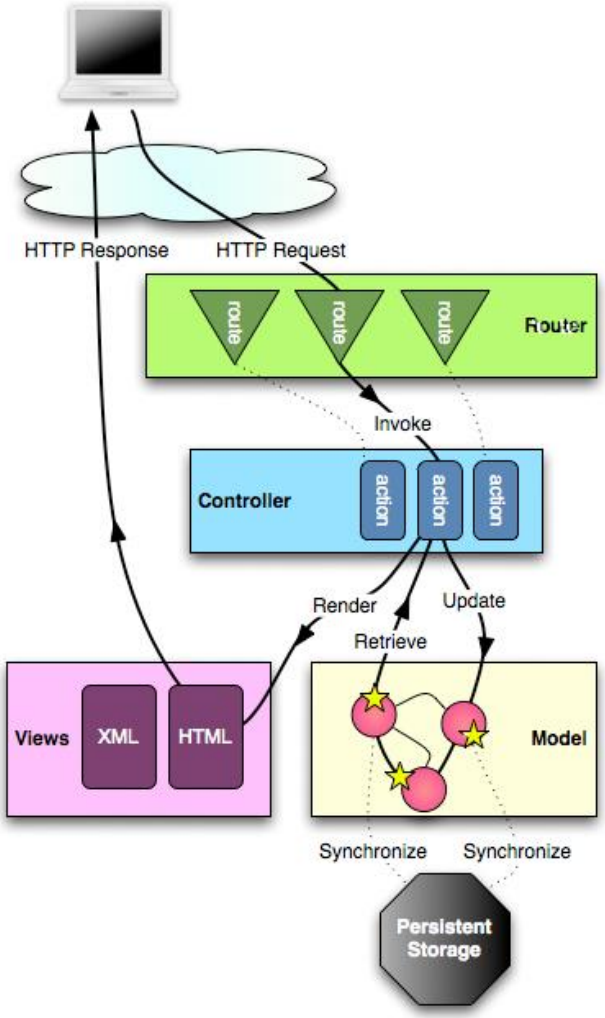
Asiakas on keskikokoinen, pääkaupunkiseudulla toimiva julkisen sektorin organisaatio. Heidän mobiilisivustonsa on toteutettu goMob CMS:n avulla, mutta heidän perinteinen verkkosivustonsa toimii suuren IT-talon toimittaman sisällönhallintajärjestelmän päällä.

Asiakkaan verkkosivustolla on listattu suuri määrä tapahtumia, jotka päivittyvät lähes päivittäin. Niiden manuaalinen päivittäminen sekä verkkosivuston, että mobiilisivuston sisällönhallintajärjestelmiin koettiin asiakkaan puolella kohtuuttoman työlääksi, joten goMob CMS integroitiin asiakkaan käytössä jo olleeseen sisällönhallintajärjestelmään.

Raportista selviää, miten goMob CMS näyttää sivun, jonka sisällöstä osa on syötetty goMob CMS:n kautta, ja osa haetaan dynaamisesti ulkoisesta järjestelmästä.

3.2 Sivulatauksen eri vaiheet

Kun kävijä saapuu sivulle, joka on rakennettu goMob CMS:n päälle, se aktivoi järjestelmän eri osa-alueita sovelluskehyksen määrittämässä järjestyksessä (kuva 4). Nämä osa-alueet on esitelty alla – ja myöhemmin omina lukuinaan – tämän järjestyksen mukaisesti.



Kuva 4. Sivupyynnön elinkaari Play!-sovelluskehyksessä [4]

Apache virtuaalipalvelin: Kun käyttäjä lataa sivun www.asiakasx.fi/ohjelmisto, sivupyyntö ohjautuu ensin Apache-palvelimelle, joka on Apachen ProxyPass -direktiivin avulla määritetty välittämään pyyntö toisessa portissa sijaitsevaan Play!-sovelluskehyksen palvelimeen, osoitteeseen /rest/prettyurl/asiakasx/ohjelmisto .

HTTP API & reititys: Kun Play!-sovelluskehys vastaanottaa sivupyynnön /rest/prettyurl/asiakasx/ohjelmisto, sen HTTP Routing -komponentti välittää Application-ohjaimen *prettyURL(userName, pageName)* -metodille. REST-arkkitehtuuria, goMob CMS:n rajapintaa ja eräitä näihin liittyviä ratkaisuja esitellään tässä raportissa tarkemmin luvussa 4.

Ohjain (engl. Controller): Kun Application-ohjaimen *prettyURL(userName, pageName)* vastaanottaa sivupyynnön, se tarkistaa käyttöoikeudet, päivittää analytiikkaa, hyödyntää välimuistitusta, hakee oikeat malli-objektit, oikeat näkymä-objektit, käskee niitä renderöimään itsensä ja luo tämän perusteella HTTP-vastauksen. Tätä toimintaketjua tarkastellaan tässä raportissa takemmin luvussa 5.

Malli (engl. Model): Tietokannan sisältöä vastaavat malli-luokat aktivoidaan ohjaimen pyynnöstä. Erityisesti asiakkaaseen, analytiikkaan ja sisältöelementteihin liittyviä malleja tarkastellaan tässä raportissa tarkemmin luvussa 6.

Tietokanta: Käytössä oleva MySQL-tietokanta on abstraktoitu Play!-sovelluskehykseen kuuluvan EBean ORM:n taakse. Tästä abstraktoinnista kerrotaan tarkemmin malleista puhuttaessa, mutta tästä syystä tietokannan tekninen tarkastelu erillisenä komponenttina on jätetty pois.

Näkymä (engl. View): Suuressa osassa sivulatauksista ohjain esittää sivun käyttäen apuna mm. Scala-mallipohjia, sekä ns. *static assetteja*, kuten CSS ja JavaScript -tiedostoja. Tapaustutkimukseksi valitun asiakkaan sivulatauksessa ei käytetä kumpiakaan, mutta tätä toiminnallisuutta esitellään silti yleisellä tasolla luvussa 7.

Ajastetut tehtävät: Ajastetuilla tehtävillä määritetään ohjelmakoodia suorittumaan esimerkiksi kerran vuorokaudessa. Suuri osa integraatioista muihin järjestelmiin on toteutettu niin, että goMob CMS päivittää esimerkiksi kerran vuorokaudessa sisältöä erillisestä rajapinnasta. Tähän toiminnallisuuteen on viittauksia muita komponentteja käsiteltäessä, mutta sen yksityiskohtaisempi tarkastelu on rajattu raportin ulkopuolelle.

Automatisoidut testit: Koko goMob CMS on toteutettu TDD:llä (engl. Test Driven Development), eli käytännössä kaikelle palvelinpuolen toiminnallisuudelle on olemassa automatisoituja funktionaalisia ja yksikkötestejä. TDD:tä ja Play!:n yhteistoimintaa JUnit- ja Selenium-testityökalujen kanssa tarkastellaan raportissa tarkemmin luvussa 9.

4 REST API

Kun goMob CMS v2:ta lähdettiin kehittämään, yksi selkeimmistä tavoitteista oli kehittää modernimpi, AJAX-pohjainen käyttöliittymä. Toisin sanoen vanha HTML-lomakkeisiin pohjannut tapa välittää tietoa palvelimelle korvattiin dynaamisilla JavaScript-moduuleilla, jotka suorittivat HTTP-kutsuja ilman perinteisiä sivulatauksia.

Edellä kuvatun toiminnan mahdollistamiseksi oli määriteltävä HTTP-rajapinta eli API. Katsottiin, että mikäli rajapinnasta tehdään tarpeeksi kattava ja tarpeeksi selkeä, käyttöliittymän (frontend) ja palvelinpuolen (backend) kehitys pystyttäisiin täysin irrottamaan toisistaan, mikä lisäisi järjestelmän modulaarisuutta.

Yrityksessä tehtiin päätös kehittää REST-rajapinta, jonka kautta olisi voitava käsitellä kaikkea järjestelmässä olevaa dataa.

4.1 REST-arkkitehtuurin kuvaus

REST (engl. Representational state transfer) on rajapinta-arkkitehtuuri, jonka W3C kehitti rinnakkain HTTP 1.1:en kanssa.

4.1.1 URL-rakenne

REST-arkkitehtuuri rakentuu *resurssi*-konseptin ympärille. Resurssilla tarkoitetaan itsenäistä kokonaisuutta, jollaisia halutaan lisätä, poistaa, hakea ja muokata rajapinnan kautta. goMob CMS:n tapauksessa tällaisia ovat esimerkiksi *sivu* ja *käyttäjä*.

REST-rajapinnassa jokaisella resurssikokoelmalla sekä jokaisella sen resurssilla on oma URL:nsä. Toisin sanoen, kun rajapinta löytyy osoitteesta /rest ja siihen liittyvät mm. resurssikokoelmat *users* ja *pages*, rajapinnasta voi odottaa löytyvän mm. taulukossa 1 esitellyt URL:t.

Taulukko 1. REST-arkkitehtuurin mukainen URL-rakenne

URL	**Selite**
/rest/users	URL, joka vastaa resurssikokoelmaa ”users”
/rest/users/[id]	URL, joka vastaa tiettyä users-kokoelman resurssia
/rest/pages	URL, joka vastaa resurssikokoelmaa ”pages
/rest/pages/[id]	URL, joka vastaa tiettyä pages-kokoelman resurssia

REST ei ota kantaa siihen mitkä asiat voivat olla resursseja.

4.1.2 HTTP-pyynnöt

REST-rajapinnassa hyödynnetään neljää HTTP-pyyntöjen metodia, jotka ovat: GET, POST, PUT ja DELETE.

HTTP GET -pyyntö resurssikokoelman URL-osoitteella palauttaa listan kaikista kokoelmaan kuuluvista resursseista. Vastaavasti pyyntö yksittäisen resurssin URL:ään palauttaa kyseisen resurssin. Arkkitehtuuri ei itsessään määritä, missä muodossa resurssikokoelma tai resurssi tulee palauttaa, vaan se riippuu rajapinnasta, resurssin tyypistä ja mahdollisista asiakkaan ja palvelimen välisistä sopimuksista. Kutsuun voidaan esimerkiksi liittää HTTP-otsake, jolla pyydetään palvelinta palauttamaan käyttäjälistaus XML- tai JSON-muodossa.

HTTP POST -pyyntö resurssikokoelman URL-osoitteella lisää uuden resurssin kokoelmaan. POST-metodi yksittäisen resurssin URL-osoitteella ei periaatteessa merkitse mitään, mutta käytännössä sitä kohdellaan usein PUT-pyyntönä.

HTTP PUT -pyyntö korvaa URL:ää vastaavan resurssin tai resurssikokoelman.

HTTP DELETE -pyyntö poistaa URL:ää vastaavan resrussin tai resurssikokoelman.

REST-arkkitehtuuriin kuuluu, että kaikkiin pyyntöihin vastataan asianmukaisilla HTTP-tilakoodeilla. Esimerkiksi GET-pyynnön resurssiin, jota ei löydy, tulee palauttaa HTTP-vastaus tilakoodilla 404 NOT FOUND mahdollisen *response body*:ssä olevan virheviestin lisäksi.

4.1.3 Tilattomuus

REST-arkkitehtuuri on tilaton (engl. stateless). Toisin sanoen jokaisessa pyynnössä on oltava mukana kaikki pyynnön suorittamiseen tarvittavat tiedot. Jos esimerkiksi POST-pyynnöstä puuttuu jokin tieto, joka tarvitaan resurssin lisäämiseen, palvelin ei pyydä tai jää odottamaan lisätietoa, vaan vastaa, että pyyntö ei onnistunut.

Puhdasoppisesti toteutetussa REST-rajapinnassa tilattomuus pätee periaatteessa myös tunnistautumiseen, eli palvelun edellyttämät käyttäjätunnukset liitetään osaksi jokaista sivupyyntöä esimerkiksi HTTP Basic Auth -menetelmällä. Tämä helpottaa esimerkiksi rajapintaa hyödyntävien ylläpitoskriptien kirjoittamista.

Käytännössä REST-rajapinnat, joita käytetään laajalti AJAX:in kautta, eivät ole täysin tilattomia vaan istuntoon liittyvää dataa tallennetaan palvelimen puolella sessioneihin. Tällä tavoin palvelinpuolta voidaan kehittää aivan, kuten perinteistä verkkopalvelua, vaikka data palvelimen ja selaimen välillä liikkuukin AJAX & REST -yhdistelmällä.

4.2 Play! HTTP -reititys

Play!-sovelluskehys tarjoaa moduulin nimeltä HTTP Routing (reititys). Käytännössä kaikista web-sovelluskehyksistä jossain muodossa löytyvä toiminnallisuus kartoittaa URL-osoitteisiin tulevat HTTP-sivupyynnöt ohjainluokkiin ja näiden metodeihin ottaen huomioon myös sivupyynnössä käytetyn HTTP-metodin.

Reitityksellä voidaan siis määritellä, että esimerkiksi osoitteeseen /rest/users tuleva GET-muotoinen pyyntö suorittaan UserController-luokan metodin listUsers() ja osoitteeseen /rest/users/14 tuleva DELETE-muotoinen pyyntö suorittaa UserController luokan metodin deleteUser(long userId) välittäen parametrina luvun 14.

Play!-sovelluskehyksessä reitit määritellään tiedostopäätteettömässä tiedostossa conf/routes hyvin yksinkertaisella syntaksilla. Yksi rivi vastaa aina yhtä reittiä niin, että rivi sisältää sivupyyntöön liittyvän HTTP-metodin, URL:n mahdollisine dynaamisine osineen ja tiedon ohjaimesta ja metodista, joiden käsiteltäväksi sivupyyntö annetaan. Rivi näyttää esimerkiksi tältä:

```
GET    /rest/users/:userId    controllers.UserController.showUser(userId: Long)
```

conf/routes-tiedostoa voi myös kommentoida, ja syntaksi tukee monimutkaisempiakin osoitemäärityksiä, jotka hyödyntävät säännöllisiä lausekkeita tai asettavat muuttujille oletusarvoja. Käytännössä näille toiminnallisuuksille ei ilmene tarvetta kovin usein: goMob CMS:n rajapinnassa on kokonaisuudessaan lähes sata reittiä, joista vain pari on edellä esiteltyä monimutkaisempia.

conf/routes käännetään samalla tavalla kuin muutkin lähdekoodin tiedostot. Kääntäjä varoittaa, jos jokin reitti on mahdotonta saavuttaa ja heittää virheen, mikäli osoite esimerkiksi viittaa sellaiseen ohjaimeen tai metodiin, jota ei ole olemassa.

4.3 goMob CMS API

Valtaosa goMob CMS:n rajapinnasta on CRUD:ia (engl. Create, Read, Update, Delete). Alla on esimerkki page-objekteihin liittyvästä CRUD:sta kokonaisuudessaan:

```
GET    /rest/users/:userId/pages            controllers.Page.listPages(userId:Long)
POST   /rest/users/:userId/pages            controllers.Page.addPage(userId:Long)
GET    /rest/users/:userId/pages/:pageId    controllers.Page.showPage(pageId:Long)
PUT    /rest/users/:userId/pages/:pageId    controllers.Page.updatePage(pageId:Long)
DELETE /rest/users/:userId/pages/:pageId    controllers.Page.deletePage(pageId:Long)
```

Page CRUD:iin liittyy joitain huomionarvoisia seikkoja. Kyseessä on pääosin REST-arkkitehtuurin mukainen rajapinta, mutta pages-resurssikokoelmaan liittyvät PUT- ja DELETE-metodit on jätetty toteuttamatta. Järjestelmässä ei ole missään vaiheessa ollut tarvetta esimerkiksi korvata käyttäjän kaikkia sivuja toisella kokoelmalla sivuja, joten kyseisen kaltaisen metodin lisäämiseen ei ole käytetty työaikaa.

Toinen merkittävä seikka on, että tiettyyn käyttäjään liittyvät sivut haetaan tässä rajapintamallissa hyvin hierarkisesti esim. …/users/13/pages. Tämä on sinänsä REST-arkkitehtuurin raamien mukainen, muttei ainoa niiden sallima tapa. Toinen yhtä hyvin arkkitehtuurin mukainen tapa olisi hakea sivut esimerkiksi osoitteesta /rest/pages ja välittää samalla query-parametrina tieto, että asiakasohjelma on kiinnostunut vain käyttäjän 13 sivuista.

goMob CMS:n käyttämä lähestymistapa valittiin, sillä se tuntui järjestelmän kehityksen alkuvaiheessa intuitiiviselta: käyttäjällä on kokoelma sivuja, joihin on loogista viitata kuin käyttäjän alahakemistoon. Järjestelmän kasvaessa tämä lähestymistapa osoittautui kuitenkin ongelmalliseksi.

Kun käyttäjällä on kasa sivuja, joilla on kasa sisältömoduuleita, joilla on taas kasa ominaisuuksia, rajapinnan osoiterakenne muodostuu raskaaksi ja monimutkaiseksi. Hierarkinen rakenne myös vaatii jatkuvasti mukaan tietoa, jota ei aina tarvita: esimerkiksi pääkäyttäjän halutessa poistaa tietyn sivun, ei pitäisi olla tarpeen välittää rajapinnalle sivun id:n lisäksi myös sen omistavan käyttäjän id:tä.

Järjestelmän laajetessa alkoi myös syntyä tilanteita, joissa samanlainen resurssi kuului useisiin eri kokoelmiin. Esimerkiksi käyttäjällä on sivuja, mutta käyttäjällä on myös kampanjoita ja kampanjoilla on sivuja. Tällöin hierarkinen rakenne URL:issä merkitsee,

että periaatteessa samanlaisten, Page-luokkaan pohjautuvien malliobjektien muokkaamiseen on useita eri reittejä rajapinnassa.

Ilmenneistä ongelmista otettiin opiksi ja uudemmissa CRUD:eissa vältetään hierarkisia reittejä, ellei niille ole oikeasti välttämätöntä tarvetta. Muutamissa ensimmäisissä CRUD:eissa ongelma on tiedostettu, mutta niiden refaktorointi on todettu niin työlääksi, että sille ei ole vielä löytynyt aikaa.

Edellä mainittuja asioita lukuunottamatta kaikki rajapinnan CRUD:t – Käyttäjät, sivut, kuvat, sisältömoduulit, jne. – seuraavat samaa REST-arkkitehtuuria ja niiden reittimääritykset ovat hyvin itseääntoistavia. Näistä koostuu noin 90 % goMob CMS:n rajapinnasta.

CRUD:ien lisäksi rajapinnasta löytyy muutama erityinen reitti esimerkiksi tilalliseen tunnistautumiseen liittyen. Tietyissä CRUD:eissa on myös hieman monimutkaisempia reittejä – esim. kuvatiedostojen CRUD:in GET-metodi hyväksyy määreitä, joiden perusteella kuvatiedostoa voidaan skaalata palvelimen puolella eri kokoiseksi.

4.4 Tapaustutkimus: Asiakkaan sivulatauksen REST-kutsut

Kun osoitteessa www.asiakasx.fi/ohjelmisto suoritetaan sivulataus, kyseinen lataus ohjataan Apachen virtuaalipalvelimeen asetettujen ProxyPass- ja ProxyPassReverse-määreiden avulla rajapinnan osoitteeseen /rest/prettyurl/asiakasx/ohjelmisto . Tähän liittyvä reittimääritys on esitetty alla (esitystilan rajallisuuden vuoksi jaettu kahdelle rivil-le):

```
GET     /rest/prettyurl/:userName/:pageName
        controllers.Application.prettyURL(userName:String,pageName:String)
```

Tällaisessa tilanteessa reititys siis ohjaa sivupyynnön käsittelyn controllers packagessa olevalle Page-luokalle ja kutsuu sen staattista metodia *renderHtml(String clientName, String pageName)* argumenteilla ”asiakasx” ja ”ohjelmisto”. Mikäli sivulataus olisi tullut osoitteeseen www.asiakasx.fi, Apache olisi ohjannut sen osoitteeseen /rest/prettyurl/asiakasx/ . Tällöin yllä oleva reittimääritys olisi aktivoitunut jälleen, mutta pageName paramterin arvona olisi ollut null, jonka metodi tulkitee pyynnöksi näyttää etusivu.

Järjestelmä renderöi siis sivut prettyurl-rajapintakutsun kautta. /rest/-alusta huolimatta prettyurl-osio ei ole varsinaisesti REST-arkkitehtuurin mukainen, sillä siihen ei liity esimerkiksi DELETE- tai PUT-metodeja ollenkaan. Nämä CRUD-tyyliset metodit eivät ole mielekkäitä sivujen näyttämiseen, vaan sivuille ja sisältöelementeille on erilliset CRUD:it.

Jos sivujen näyttäminen olisi koettu tarpeelliseksi tehdä puhdasoppisesti REST-arkkitehtuurin mukaisesti, tarkoitukseen olisi voitu käyttää esim. alla olevaa reittimääritystä, jota ei siis tällä hetkellä löydy järjestelmästä:

```
GET     /rest/users/:userId/pages/:pageId/_renderHTML
        controllers.Page.renderHtml(userId:Long,pageId:Long)
```

Tällöin Page-tyyppistä resurssia olisi voitu pyytää renderöimään itsensä samassa URL:ssä, jossa resurssia muutenkin käsitellään, mutta lisäämällä loppuun alaviivalla prefixattu nimi toiminnolle. Tämä on yleinen käytäntö palvelinpuolella suoritettavaa ohjelmakoodin kuvaamiseksi (sen sijasta, että resurssin tiedot haettaisiin esimerkiksi JSON-muodossa). Kolmas vaihtoehto sivun renderöimiseen olisi ollut normaali GET-kutsu resurssille, mutta esim. HTTP Content-Type -headerilla ilmoittaa, että vastausta toivotaan juuri HTML-muodossa.

Käytössä oleva prettyurl-rajapinta kuitenkin mm. yksinkertaistaa Apachen konfigurointitiedostojen kirjoittamista ja helpottaa sivujen testaamista silloinkin, kun sivusto on keskeneräisessä vaiheessa, eikä vielä löydy mistään domainista. Tämä koettiin tärkeämmäksi kuin REST-rajapinnan puhdasoppisuus.

5 Ohjain

Web-sovelluskehyksille tyypillisessä MVC-toteutuksessa ohjaimen rooli on käsitellä vastaanotettu sivupyyntö, tarkistaa käyttöoikeudet, analysoida muu pyyntöön liittyvä data ja sitten luoda mallia ja näkymää hyväksikäyttäen halutunkaltainen vastaus.

Esimerkiksi pyydettäessä näyttämään tietty sisällönhallintajärjestelmän sivu, ohjaimen tärkeimmät tehtävät ovat:

- tulkita sivupyyntö
    - mikä sivu halutaan näyttää, jne.

    - luoda virheilmoitus, mikäli sivupyynnöstä puuttuu oleellista dataa
- lähettää sivupyyntöä vastaava sivu välimuistista, jos mahdollista
- varmistaa, että pyydetty sivu on olemassa
    - luoda virheilmoitus, jos näin ei ole
- hakea tietokannasta sivua vastaava Page-malliobjekti
    - delegoida mm. sivun sisällön selvitys modelille
- valita käyttötarkoitusta vastaava näkymäobjekti
    - delegoida model objektin avulla HTTP-vastauksen luominen viewille
- hoitaa muut sivupyynnön käsittelyyn liittyvät tehtävät
    - esim. web-analytiikan päivittäminen siihen liittyvän modelin avulla
- lähettää mallin ja näkymän avulla luotu HTTP-vastaus.

Ohjaimen tehtävä on siis nimensä mukaisesti ohjata koko sivulatauksen käsittelyä puuttumatta kuitenkaan yksityiskohtiin esim. siitä, miten malliobjekti tietää, mitä sisältöä kyseisellä sivulla on.

5.1 play.mvc.Controller-luokka

Play!-sovelluskehys tarjoaa abstraktin Controller-nimisen luokan, josta kaikkien sovelluksen käyttämien ohjainten oletetaan periytyvän. Luokka tarjoaa kirjon kenttiä ja metodeja, jotka auttavat ohjaimille kuuluvien vastuiden hoitamisessa.

Controller -luokka määrittää kourallisen staattisia metodeita. Tärkeimmät näistä (*ctx(), request(), response(), session()* ja *lang()*) palauttavat objekteja, joiden avulla on mahdollista manipuloida valmisteilla olevaa HTTP-vastausta tai selvittää sivupyyntöön liittyviä tietoja. Näillä mm. tarkistetaan ja asetetaan HTTP-headereita, evästeitä, vastauksen HTTP-statusta ja niin edelleen. Luokka myös perii play.mvc.Results-luokalta kymmeniä vastauksen luomiseen liittyviä metodeja ja vakioita.

Luokka tarjoaa myös työkalut lomakedatan käsittelyyn. Esimerkiksi uuden käyttäjän rekisteröintiin voi määrittää erikseen luokan, jonka kentillä ja annotaatioilla on määritetty lomakkeeseen kuuluvan aina tietyt pakolliset kentät. Tämän jälkeen luokan *form()* -metodilla voidaan tarkistaa, onko nyt käsiteltävässä sivupyynnössä lähetetty kaikki kyseisen lomakkeen tarvitsemat kentät.

5.2 goMob CMS:n ohjaimet

goMob CMS:ssä on parikymmentä luokkaa, jotka periytyvät Play!:n Controller-luokasta ja hoitavat yllä kuvattuja vastuita. Metodien jaotteleminen eri ohjaimiin on lähes puhtaasti ohjelmakoodin luettavuuden ja ylläpidon helpottamista. Periaatteessa mikään ei estäisi pitämästä kaikkia paria sataa metodia yhdessä ohjaimessa.

Oleellisimmat ohjaimet on selitetty taulukossa 2.

Taulukko 2. goMob CMS:n sivulatausten oleellisimmat ohjaimet

Ohjain	**Selite**
Application	Yleisiä toimintoja, esim. login ja välimuistitus
Image	Kuvien lisääminen, poistaminen, skaalaus, ym.
Page	Sivujen CRUD ja renderöinti
User	Käyttäjien CRUD
Widget	Sisältöelementtien CRUD

Varsinaiseen sivun näyttämiseen liittyvä toiminnallisuus on pääasiassa Page-ohjaimessa ja sen muokkaamiseen liittyvä toiminnallisuus Widget-ohjaimessa. Poikkeuksen tekevät esim. sivun *title*-tietueen tai sivukohtaisen CSS-tyylitiedon asettaminen, jotka kuuluvat myös Page-ohjaimeen.

User-ohjain on välttämätön, koska sivut liittyvät aina johonkin käyttäjään ja näin ollen lähes kaikki järjestelmän toiminnallisuus on sidottu tavalla tai toisella käyttäjää vastaavaan malliin ja sen muokkaamiseen käytettyyn ohjaimeen.

Image-ohjainta käytetään, kun järjestelmään ladataan kuvia tai niitä näytetään. Yksinkertaisen CRUD:in lisäksi ohjain mahdollistaa myös esimerkiksi palvelinpuolella tehtävän kuvatiedostojen skaalauksen ja pakkaamisen perustuen URL:ään liitettyihin määreisiin. Ohjain ei siis aktivoidu, kun HTML-muotoinen sivu luodaan ja palautetaan, mutta jos sivulla on kuvia, joiden URL:t on upotettu osaksi lähdekoodia ja selain pyytää nämä kuvat välittömästi palvelimelta, Image-ohjain käsittelee kyseisen sivupyynnön.

5.3 Tapaustutkimus: Ohjainten toiminta asiakkaan sivulatauksessa

HTTP-reititys ohjaa sivupyynnön controllers.Application-luokasta löytyvälle metodille:

```
public static Result prettyURL(String username, String pageName)
```

Välitettyjen parametrien arvot ovat reitityksestä saadut *asiakasx* ja *ohjelmisto* . Application-ohjaimessa on kourallinen välimuistitukseen liittyviä metodeja. Jos haluttu sivu on edellisellä kyseiseen sivuun kohdistuvalla sivupyynnöllä tallennettu välimuistiin, se lähetetään sieltä sellaisenaan. Jos kyseistä sivua ei löydy välimuistista, pyyntö päätyy pienen pallottelun jälkeen Page-ohjaimesta löytyvälle metodille:

```
private static Content renderHtmlPageInternal(models.User user, models.Page page)
```

Tämä metodi tarkistaa ensin, että kyseiset mallit ovat olemassa – tässä vaiheessa ORM:ltä *name*-kentän perusteella kysytyt objektit saattaisivat olla vielä *null*-objekteja. Jos jompaakumpaa ei löydy tai löydetty sivu ei liity löydettyyn käyttäjään, ohjain palauttaa HTTP 404 NOT FOUND -statuksen.

Jos sivu löytyy ja kuuluu oikealle käyttäjälle, ohjain kutsuu sivun *render*-metodia. Kaikki varsinaisen HTML-merkkijonon luomiseen liittyvä tapahtuu tämän metodin sisällä ja on selitetty tarkemmin seuraavassa luvussa. Ohjain saa vastauksena merkkijonon, jossa on kaikkien sivun sisältöelementtien sisältö järjestyksessä ja kysyy vielä sivun mallilta sivun *title*-tiedon sekä sivuun mahdollisesti liitetyt omat CSS- ja JS-määreet.

Ohjain lähettää sivupyyntöön vastauksena HTTP 200 OK -statuksen, joka sisältää kaiken edellisen pohjalta luodun HTML-dokumentin.

Sivun näyttäminen tapahtuu *try-catch*-lohkon sisällä niin, että jos esim. HTML:n generoimiseen liittyvässä toiminnallisuudessa syntyy virhetilanne, ohjain osaa palauttaa virhesivun asiaankuuluvalla HTTP-statuksella.

6 Malli

Web-sovelluskehyksille tyypillisessä MVC-toteutuksessa malli on pääasiassa abstraktio tietokannasta. Mallin luokkia käytetään tietokannan sisällön kuvaamiseen, mutta niihin

on kapseloitu myös tiettyjä kyseisen sisällön käsittelemiseen ja esittämiseen tarvittavia toiminnallisuuksia.

Esimerkiksi Page-mallia käytetään tietokannan pages-taulun sisällön käsittelyyn. Datan säilömisen lisäksi mallissa löytyy mm. metodi, jolla sivua voi pyytää *renderöimään* itsensä, eli palauttamaan sisältönsä HTML-merkkijonona. Kyseistä merkkijonoa ei ole valmiina pages-taulussa, vaan se vaatii objektia hakemaan tietokannasta siihen liittyvät sisältöelementit ja luomaan merkkijonon niiden perusteella. Kyse on siis tietokannan sisällön kuvaamiseen liittyvästä toiminnallisuudesta, vaikkakin pelkkää tietokantahakua monimutkaisemmasta sellaisesta.

6.1 EBean ORM -kirjasto

Play!-sovelluskehyksessä on käytössä EBean ORM -kirjasto. ORM (engl. Object-relational mapping) on modernissa OOP:ssa (engl. Object-oriented programming) tyypillinen työkalu, jolla ohjelmakoodin luokat määritetään vastaamaan tietokannan tauluja. Tämän jälkeen ohjelmoija voi periaatteessa jopa unohtaa tietokannan täysin: ORM päivittää tietokantaa automaattisesti, kun objekteja luodaan, muokataan tai tuhotaan.

EBean ORM:ssä kartoitukset tehdään Javan annotaatioilla. Näillä määritetään aluksi luokat vastaamaan tauluja. Tämän jälkeen ORM tekee tiettyjä oletuksia tietokannan rakenteesta. Jos esimerkiksi luokassa on merkkijonomuuttuja nimeltä *name*, ORM olettaa löytävänsä tietokannasta samalla tavalla nimetyn varchar(255) -tyyppisen kentän. Tarvittaessa annotaatioita voidaan käyttää myös muuttujien tasolla.

Kuvassa 5 on esimerkki luokasta, joka on määritelty käyttämään EBean ORM:ää.

```
@Entity
public class Task extends Model {

  @Id
  @Constraints.Min(10)
  public Long id;

  @Constraints.Required
  public String name;

  public boolean done;

  @Formats.DateTime(pattern="dd/MM/yyyy")
  public Date dueDate = new Date();

  public static Finder<Long,Task> find = new Finder<Long,Task>(
    Long.class, Task.class
  );

}
```

Kuva 5. Esimerkki EBean ORM:ää käyttävästä luokasta

Osa kentistä – esimerkiksi *public boolean done* – on jätetty määrittelemättä tarkemmin, jolloin EBean olettaa niiden vastaavan saman nimisiä tietokannan kenttiä ilman erillisiä määreitä (esim. *not null*). Osassa kentistä on sallittuun arvoalueeseen tai datan formaattiin liittyviä annotaatioita. Jos objektin koettaa tallettaa niin, että määreitä ei noudata, aiheutuu *poikkeus*.

Kuva 6 on esimerkki siitä, miten edellisen luokan kuvaamia malliobjekteja voidaan hakea ja poistaa tietokannasta.

```
// Find all tasks
List<Task> tasks = Task.find.all();

// Find a task by ID
Task anyTask = Task.find.byId(34L);

// Delete a task by ID
Task.find.ref(34L).delete();
```

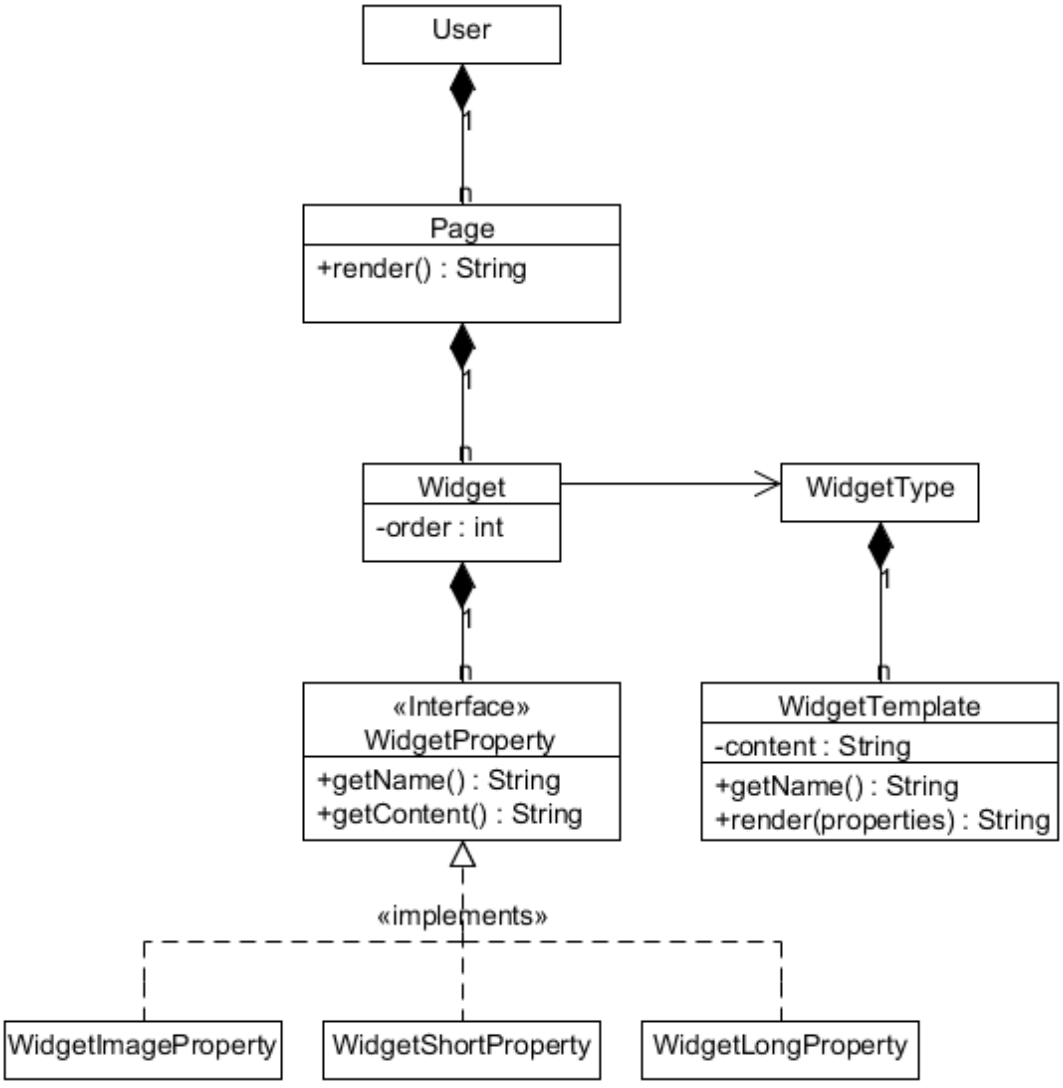
Kuva 6. Esimerkkejä Task-luokan käytöstä

Kun luokka on määritelty aieman esimerkin (kuva 5) mukaisesti, siitä löytyy staattinen, Finder-tyyppinen kenttä. Kyseisen objektin avulla voidaan suorittaa tietokantaoperaatioita suoraan malliobjektin ominaisuuksien mukaan. Tietokannasta voidaan esimerkiksi

hakea tai poistaa kaikki malliobjektit tietyn kentän arvon perusteella. Finder-objektia kutsuttaessa ei tarvitse tietää, millaiseen tietokantatauluun mallin data on tallennettu.

6.2 goMob CMS:n mallit

goMob CMS:ssä on useita kymmeniä malliluokkia, jotka liittyvät mm. web-analytiikkaan, käyttöoikeuksiin, maksutapahtumiin ja kaikkeen muuhun järjestelmän yleiseen toimintaan. Jokaisen sivulatauksen kannalta välttämättömimmät mallit on esitelty kuvassa 7.



Kuva 7. goMob CMS:n mallit, jotka liittyvät jokaiseen sivulataukseen

User vastaa yhtä CMS:n käyttäjätiliä, eli käytännössä yhtä goMobin asiakasta. Malli on tiukasti sidottu tunnistautumiseen, maksutapahtumiin ja laajaan kirjoon vastaavaa toiminnallisuutta. Kun tarkastellaan tämän raportin puitteissa hyvin kapea-alaisesti sivujen näyttämistä, mallissa ei ole kuitenkaan juuri muuta mielenkiintoista kuin lista yhteen käyttäjään liittyvistä sivuobjekteista.

Page on periaatteessa hyvin yksinkertainen malli, jonka tärkein tehtävä on ryhmittää kerralla näytettävät sisältöelementit yhteen. Malliin liittyy myös URL-rakenteeseen, sivun metatietoihin (esim. *title*) ja välimuistitukseen liittyviä toimintoja.

Widget on pääasiallinen sisältöelementti. Jokaisella sivulla on nolla tai useampi widget. Hieman yksinkertaistaen voidaan sanoa, että sivua näytettäessä yksinkertaisesti renderöidään järjestyksessä sivun widgetit, liitetään näistä saadut HTML-jonot peräkkäin ja palautetaan tulos.

Widget on periaatteessa kokoelma kahdesta eri mallista: *WidgetTypestä* ja *Widgetistä*. Yksi esimerkki WidgetType instanssista on ”Article”, eli järjestelmässä on article-tyyppinen widget, jolla on oma mallipohjansa. Kun tietylle sivulle halutaan lisätä uusi article, luodaan erillinen Widget-tyyppinen objekti, joka tietää, mihin WidgetTypeen (ja täten *WidgetTemplateen*) se viittaa, mihin sivuun se kuuluu, jne.

Käytännössä WidgetType on lähes näkymätön Widgetin ulkopuolella, eli useimmissa yhteyksissä voidaan puhua widgetistä ja tarkoittaa näiden kahden yhdistelmää.

WidgetTemplate jokaisella Widgetillä on nolla tai useampi mallipohja. Käytännössä lähes kaikilla on yksi mallipohja, jonka nimi on HTML. Kun järjestelmä tuki NetBiscuitsia ja HTML-sivustoja samaan aikaan, monilla widgeteillä oli yksi BiscuitML-mallipohja ja yksi HTML-mallipohja. Nykyään muutamalla widgetillä on XML- tai JSON-mallipohja harvinaislaatuisempia käyttötilanteita varten, mutta oikeastaan luokkien välinen suhde voisi olla yksi-yhteen.

Mallipohjalla on *content*-kenttä, jossa on merkkijonona widgetin rakenne Scala-templatekielellä. Kun widget näytetään, mallipohjan sisältö lähetetään *Scala template engine*:lle, joka suorittaa sen. Mallipohja saattaa sisältää ajettavaa ohjelmakoodia, joka voi esimerkiksi lukea WidgetPropertyjen tai evästeiden sisältöä ja muokata palauttamaansa merkkijonoa tämän perusteella.

WidgetProperty on abstraktio mistä tahansa yksittäiseen sisältöelementtiin liittyvästä arvosta. Esimerkiksi article-tyyppisellä widgetillä on otsikko, kuva ja tekstiä, eli kolme WidgetPropertyä. WidgetTemplate käyttää näitä näyttäessään Widgettiä. Ominaisuuksia on todellisuudessa useita erilaisia, lähinnä tietokantaan liittyvän optimoinnin takia, mutta käytännössä niitä käsitellään lähes ainoastaan yhteisen rajapinnan kautta.

6.3 Tapaustutkimus: Mallien roolit asiakkaan sivulatauksessa

Asiakkaan käyttötapaus, jossa sivulla näytetään asiakkaan toisesta järjestelmästä haettua sisältöä, alkaa kaikkien muiden sivulatausten tavoin. Käsiteltyään URL:stä löytyvät tiedot, ohjain hakee ORM:n avulla oikeat User- ja Page-mallit. Varmistettuaan, että ne ovat olemassa, ohjain kutsuu Page-mallin render-metodia, joka palauttaa sivun sisällön HTML-merkkijonona.

Page-malli listaa kaikki itseensä kuuluvat widgetit järjestyksessä, pyytää jokaista näistä renderöimään itsensä ja yhdistää näiden merkkijono-paluuarvot. Widgetit taas renderöivät itsensä käyttäen mallipohjaa (WidgetTemplate), jossa eri ominaisuuksia merkitsevät tunnisteet korvataan renderöitävään instanssiin kuuluvilla arvoilla.

Ero täysin goMob CMS:n kautta hallittavassa sisällössä ja ulkoisesta järjestelmästä haetussa sisällössä näkyy vasta tässä vaiheessa prosessia. WidgetTemplateissa käytetty Scala-koodikieli ei salli pelkästään muuttujien nimien korvaamista arvoilla, vaan myös Java-ohjelmakoodin upottamista mallipohjaan. Niissä widgeteissä, joiden sisältönä näytetään dataa muista järjestelmistä, templateen sisällytetään komennot esimerkiksi näyttää sisältöä välimuistista – ja tarvittaessa hakea sinne uutta sisältöä palvelinpuolen HTTP-kutsulla – kun mallipohjaa suoritetaan.

Asiakkaan tapauksessa ohjelmistoa kuvaavan widgetin mallipohjassa on Java-koodia, jolla tehdään MySQL-tietokantahaku. Sillä haetaan asiakkaan ohjelmisto paikallisesta tietokannasta ja luodaan tämän perusteella HTML:ää, kuin kyseessä olisi mikä tahansa sivu. Widget palauttaa tämän merkkijonon ja mistä tahansa muualta järjestelmästä katsoen se toimii kuin kaikki yksinkertaisemmatkin sisältöelementit.

Asiakkaan ohjelmisto päivitetään paikalliseen tietokantaan ajastetusti kerran vuorokaudessa. Tähän käytetään goMob CMS:n osana olevia ajastettuja tehtäviä – jotka on rajattu tämän raportin ulkopuolelle – mutta sivun näyttämisen kannalta päivitys voitaisiin yhtä hyvin tehdä vaikka cron jobilla.

Tietokannan sisältöä tai rakennetta ei ole sidottu goMob CMS:ään millään muulla tavalla kuin widgetin mallipohjan käyttämässä koodissa. Järjestelmään voi siis integroida käytännössä minkä tahansa dynaamisen sisältölähteen luomalla sitä varten uuden widgetin ja kirjoittamalla kyseiselle widgetille mallipohjan.

7 Näkymä

Web-sovelluskehyksissä näkymä tarkoittaa asiakasohjelmille näytettyä osaa sovelluksesta. Useimmiten tämä tarkoittaa käyttäjälle näytettyä HTML-sivua, mutta periaatteessa kaiken ulkopuolisen tahon tarkasteltavaksi luodun sisällön voidaan tulkita olevan osa näkymää. Tämä sisältää esim. suuren osan sivuvastauksista, joita REST-rajapinta luo GET-kutsujen perusteella.

Käytännössä kaikissa moderneissa web-sovelluskehyksissä on tuki sivupohjille, eli sivutason mallipohjille. Tällöin sovelluskehystä käyttävä ohjelmistokehittäjä voi määrittää monelle sivulle yhteisen rakenteen yhteen staattiseen tiedostoon. Tämä tiedosto on yleensä HTML-määritys, joka sisältää esim. navigaation ja viittaa yleisimpiin sivuston käyttämiin CSS- ja JS-tiedostoihin ym.

Niihin kohtiin määritystä, joiden sisältö vaihtelee sivukohtaisesti, asetetaan tunnisteet merkitsemään dynaamista sisältöä. Sivua luodessaan ohjain valitsee halutun sivupohjan ja määrittää, mitä sisältöä asetetaan mitäkin tunnistetta vastaavaan paikkaan.

Web-sovelluskehyksissä on hyvin tyypillistä tukea nk. *Two Step View* -suunnittelumallia. Tällöin on käytössä yksi sivupohja, jonka dynaamisena sisältönä näytetään muiden sivupohjien avulla luotua sisältöä. Esimerkiksi käyttäjän tietojen (profiilin) näyttämiseen voi olla yksi sivupohja. Tämä sivupohja ei sisällä esim. HTML-dokumentin alku- tai loppumäärityksiä, vaan osaa pelkästään renderöidä käyttäjän tiedot saadessaa ohjaimelta User-tyyppisen objektin. Tämän jälkeen kyseistä sivupohjaa voi käyttää apuna, kun geneerisempiin ja laajempiin sivupohjiin halutaan luoda sisältöä.

7.1 Näkymä goMob CMS:ssä

Template engine (myös *template processor* tai *template parser*) on sovelluskehityksen työkalu, joka mahdollistaa mallipohjan yhdistämisen dataan, joka on tiedossa vasta ajon aikana. Play! -sovelluskehyksessä yleisin käyttötapaus tälle on sivupohjat, jotka vastaavat MVC-mallin näkymää.

MVC-mallia voi lähestyä usealla eri tapaa, ja tämä valinta määrittää, miten sivupohjia käytetään. Periaatteessa täysin puristi lähestymistapa voisi olla siirtää kaikki näyttämiseen liittyvä toiminnallisuus pois malliluokista. Tällöin goMob CMS:n renderöidessä sivua ohjain hakisi instanssin sivupohjasta ja antaisi sen käyttöön ryhmän malliobjekteja, ja sivupohja osaisi luoda kaiken HTML:n sen perusteella. Malli ei tietäisi mitään sisällön näyttämiseen liittyviä yksityiskohtia.

goMob CMS:ssä olevien widgettien – ja niihin liittyvien mallipohjien – määrän ja muokkaustiheyden takia nähtiin tarpeelliseksi tallentaa myös widgettien mallipohjat tietokantaan ja täten osaksi tietokannan sisältöä kuvaavaa mallia. Tämän johdosta malli ja näkymä ovat kietoutuneet yhteen hieman puristi-lähestymistapaa tiukemmin, mutta sama yleisperiaate on yhä käytössä. Widgetin mallilla on oma templatekielellä kirjoitettu näkymänsä. Kun ne ovat kaikki renderöineet itsensä, niiden sisällön ja korkeampitasoisen, staattisen sivupohjan avulla näytetään itse sivu.

7.2 Scala-pohjainen templatekieli

Play!-sovelluskehyksen mukana tulee siihen integroitu, Scala-ohjelmointikieleen pohjaava mallipohjatulkki (engl. template engine). Scalaa ajetaan JVM-alustalla ja se on täysin yhteensopiva Javan kanssa: Java-koodista voidaan kutsua Scala-metodeja ja toisin päin. Play! on hyvin vahvasti sidottu Scalaan ja koko sovelluksen olisi voinut rakentaa kyseisellä kielellä, mutta goMob CMS käyttää Scalaa käytännössä vain mallipohjissa.

Kuvassa 8 on esitetty esimerkki Scala-mallipohjan käytöstä.

For example, here is a simple template:

```
@(customer: Customer, orders: List[Order])

<h1>Welcome @customer.name!</h1>

<ul>
@for(order <- orders) {
  <li>@order.getTitle()</li>
}
</ul>
```

You can then call this from any Java code as you would normally call a method on a class:

```
Content html = views.html.Application.index.render(customer, orders);
```

Kuva 8. Scala-mallipohjan käyttö Play!-sovelluskehyksen kanssa [8]

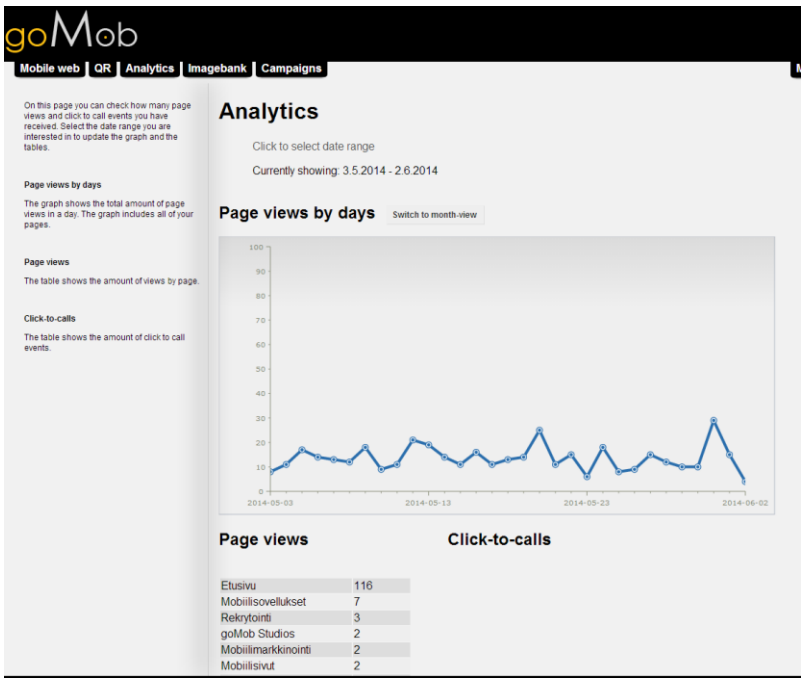
Mallipohjan alussa määritellään, minkä nimisten ja tyyppisten muuttujien kanssa mallipohjaa voi kutsua. Tämän jälkeen muuttujiin voi viitata ja – jos ne ovat objekteja – niiden metodeja voi kutsua mallipohjan sisältä. Nämä mallipohjat voidaan kääntää, jolloin kääntäjä varoittaa esimerkiksi kirjoitusvirheistä muuttujien nimissä.

Scala-mallipohjat voivat olla hyvin yksinkertaisia – käytännössä merkkijonon korvausta muuttujan arvolla – tai äärimmäisen monimutkaisia, sillä niihin voi upottaa käytännössä mitä tahansa ohjelmakoodia. goMob CMS:n tapauksessa valtaosa templateista on hyvin yksinkertaisia, mutta jotkut dynaamista sisältöä näyttävät widgetit sisältävät paljonkin suoritettavaa koodia, jossa mm. luetaan tietokantaa tai lähetetään HTTP-kutsu.

8 Web-analytiikka

Web-analytiikka on ominaisuus, joka puuttuu täysin tai lähes täysin monista sisällönhallintajärjestelmistä. Käytännössä kaikilla aktiivisesti ylläpidetyillä verkkosivustoilla on käytössä jokin web-analytiikkaan erikoistunut järjestelmä: noin 50 % sivustoista käyttää Google Analyticsia ja kun mukaan ottaa sen kaikki kilpailijat, päästään hyvin korkeisiin lukuihin. [9.] Myös goMob CMS:n päällä toimivista asiakassivuista lähes kaikilla on käytössä Google Analytics ja/tai suomalainen Snoobi.

Sisällönhallintajärjestelmien kehittäjät eivät useimmiten koe järkeväksi käyttää aikaansa tämän toiminnallisuuden toistamiseen. goMob Finland Oy päätti liittää web-analytiikan osaksi sisällönhallintajärjestelmää ensisijaisesti myyntiin liittyvistä syistä: suhteellisen pienellä työmäärällä oli mahdollista saada lisää esiteltävää myyntipalavereissa (kuva 9) ja kattavampi lista ominaisuuksia.



Kuva 9. goMob CMS:n web-analytiikan käyttöliittymä

Yrityksessä myös epäiltiin, että tavallista verkkosivustoa suppeampia mobiilisivustoja päätyisi usein päivittämään joku muu kuin yrityksen webmaster ja tällä henkilöllä voisi olla kiinnostusta tutkia mobiilipalvelun kävijämäärien kehittymistä, mutta ei välttämättä pääsyä tai osaamista yrityksen varsinaiseen web-analytiikkaan.

8.1 Kerätyt tiedot ja toimintatapa

Käytännössä kaikista web-analytiikkajärjestelmistä löytyy samat perustiedot:

- sivukohtainen sivulatausten määrä
- yksilöityjen kävijöiden määrä
- sivustolla vietetty aika tai sivulatausten määrä käyntiä kohden ja
- asiakasohjelmat (Selainten, älypuhelinten, ym. jakauma).

Näitä tietoja voidaan rajata vähintään päivämäärien ja yleensä myös kellonaikojen perusteella. Lisäksi edistyneet analytiikkajärjestelmät tarjoavat laajempia ominaisuuksia, joilla voi tarkastella esimerkiksi, miten suuri osa tietyltä sivustolta saapuneista käyttäjistä täyttää tietyn lomakkeen. goMobin tapauksessa näin edistyneitä toiminnallisuuksia ei kuitenkaan nähty tarkoituksenmukaiseksi toteuttaa vaan päätettiin tyytyä perustietojen keräämiseen ja näyttämiseen.

Kerätyn datan malli ja toimintalogiikka päätettiin pitää niin yksinkertaisena kuin mahdollista. Joka kerta, kun kävijä lataa sivun, ohjain tallentaa tietokantaan uuden uuden objektin PageView-mallista. Tämä objekti tietää mikä sivu ladattiin ja milloin, mutta ei mitään muuta. Ohjain luo myös ensimmäisellä sivulatauksella (perustuen istunnossa olevaan tietoon) Visitor-mallin perusteella objektin, joka tietää käynnin ajankohdan, kävijän IP:n ja käytetyn asiakasohjelman. PageView-objektit tietävät, mihin Visitor-objektiin ne kuuluvat.

Tämä datamalli – kahdella hyvin yksinkertaisella tietokantataululla ja muutamalla rivillä ohjelmakoodia ohjaimessa – mahdollistaa jo suhteellisen monipuolisten tilastojen haun ja esittämisen sivustoista, sivuista ja kävijöistä. Merkittävästi työtä jää tietenkin käyttöliittymän puolella diagrammien piirtämiseen, tietojen suodatukseen käyttäjän valintojen perusteella, ym., mutta kyseessä on silti varsin laaja toiminnallisuus suhteessa sen vaatimaan työmäärään.

8.2 Web-analytiikkaan liittyvien tietokantataulujen kasvu

Web-analytiikassa alkoi ajan myötä ilmetä suoritustehoon liittyviä ongelmia, joita ei ole osattu odottaa. Vaikka MySQL pyörittää kevyesti hyvin suuria datamääriä, goMob CMS on suunniteltu niin, että kaikki sen päällä olevat sivustot käyttävät samaa tietokantaa. Tuhat päivää, jonka aikana sadalla sivustolla tapahtuu jokaisella jokunen sata sivulatausta päivässä merkitsee, että aikaa myöten järjestelmän tietokanta kasvaa useisiin gigoihin, josta yli 90 % on analytiikkaan liittyvää dataa.

Tietokantahaut eivät vielä tälläkään määrällä hidastuneet niin, että se olisi merkittävästi haitannut järjestelmän normaalia toimintaa. Tietokannan koon kasvu alkoi kuitenkin tuntua varmuuskopioinnissa ja analytiikkaa näyttävän sivun latausnopeudessa.

pageviews oli ensimmäinen – ja toistaiseksi ainoa – tietokannan taulu, jossa on ollut tarpeen oikeasti hyödyntää monia MySQL:n tarjoamia ominaisuuksia, kuten taulujen osiointia.

Web-analytiikan luonteen takia tietokannan koko kasvaa erittäin tasaisella ja ennustettavalla vauhdilla, ja sen vaikutukset ovat hyvin tiedossa. Näistä syistä asian korjaaminen ei ole noussut korkeaksi prioriteetiksi, vaikka sen tarve on tiedostettu ja asiasta tehty alustavia suunnitelmia.

Ehkä helpoin goMobin käyttötarkoituksiin riittävän hyvä ratkaisutapa on yksinkertaisesti päättää olla säilömättä sivulatauskohtaista tietoa yli tietyn aikarajan (esim. vuoden). Kaikki tätä vanhemmat sivulataukset voidaan yhdistää eräänlaiseksi koonti-sivulataukseksi, jolla tieto päivämäärästä ja sivusta, johon se viittaa, mutta ei tarkkaa kellonaikaa tai viittausta Visitor-malliin.

Paljon maltillisemmin kasvava visitor-taulu voi jäädä nykymuotoonsa ja kävijämääriä tai sivukohtaisia latausmääriä tutkia jatkossakin. Tietyt analytiikan käyttötapaukset – kuten keskimääräinen sivulatausten määrä yli vuosi sitten sivustolla käynneillä kävijöillä – voidaan jättää edistyneempien ja erikoistuneempien analytiikkasovellusten vastuulle.

9 Kehitysympäristö

goMob CMS on jatkuvasti asiakaskäytössä oleva järjestelmä, jota silti kehitetään aktiivisesti. Järjestelmä vaatii uusia ominaisuuksia ja virhekorjauksia usein, joskus hyvin nopealla aikataululla. Tämän lisäksi järjestelmää kehittää useampi kuin yksi ohjelmisto-kehittäjä, joten kehitysympäristön – siihen liittyvine prosesseineen ja työkaluineen – on oltava kunnossa.

9.1 Automatisoidut testit

goMob CMS:ssä käytetään TDD:tä (engl. Test Driven Development). Kyseisessä ohjelmistokehityksen prosessissa kaiken toiminnallisuuden toteutus aloitetaan kirjoittamalla automatisoitu testi, joka suoritetaan onnistuneesti, jos toiminnallisuus on kunnossa. Tässä vaiheessa varmistetaan, että kun kyseistä toiminnallisuutta ei ole vielä olemassa, testiä ei voida suorittaa onnistuneesti. Varmistuksen jälkeen toiminnallisuutta kehitetään, kunnes se on siinä kunnossa, että aluksi kirjoitettu testi läpäistään.

Periaatteessa täydellinen TDD:n seuraus johtaa tilanteeseen, jossa mikään toiminallisuus ei voi mennä rikki ilman, että myös jokin testi menee rikki. Tällöin ohjelmistokehittäjä voi muokata ohjelman yhtä osaa ja sitten ajaa testit varmistaakseen, että muutokset eivät vahingossa aiheuttaneet virhetilanteita ohjelman missään muussa komponentissa. Testit myös toimivat dokumentaationa sille, miten eri komponentteja on tarkoitus käyttää ja mitä eri metodien odotetaan palauttavan missäkin tilanteessa.

9.1.1 TDD:n puutteet

Käytännössä TDD ei ole aivan yhtä aukoton kuin paperilla. Jos ohjelmistokehittäjä esimerkiksi muuttaa tapaa, jolla objekti renderöidään JSON-muotoon rajapinnassa, on mahdollista, että kyseistä rajapintaa testaavat testit menevät läpi myös uudella tavalla, mutta jokin rajapintaa käyttävä asiakasohjelma menee rikki muutoksista. Jos testit taas rakennetaan niin tarkasti, että jokainen muutos rikkoo ne, testejä pitää muokata joka kerta, kun toiminnallisuutta muokataan. Tällöin testin rikkoutumisella ei ole enää yhtä suurta informaatioarvoa kuin aiemmin.

Aivan kaikkea ei kannata testata. Esimerkiksi metodi, joka vain palauttaa muuttujan arvon, ei sisällä mitään sellaista toiminnallisuutta, jota varten kannattaisi kirjoittaa yksikkötesti. Tähän metodiin saatetaan kuitenkin tehdä myöhemmin muutoksia, jotka rikkovat toiminnallisuutta. Tästä syystä ns. puristi lähestymistapa testata aivan kaikkea, mutta usein siten saavutetut hyödyt eivät vastaa käytettyä aikaa. Useimmiten TDD on tasapainottelua sillä, miten paljon ja miten tarkkoja testejä kannattaa kirjoittaa.

TDD:n puutteista huolimatta totesimme sen olevan hyödyllinen prosessi ja olemme seuranneet sitä projektissa parhaamme mukaan vuoden 2012 suuresta refaktoroinnista lähtien.

9.1.2 Automatisoitujen testityökalujen käyttö Play!:n kanssa

Javan tunnetuin yksikkötestauskirjasto on JUnit. Play!-sovelluskehys on suunniteltu toimimaan hyvin yhteen kyseisen työkalun kanssa. JUnit on osana sovelluskehystä ja sen käytöstä yleisesti, sekä sovelluskehyksen konventioista sen käytöstä yksikkötesteihin ja funktionaalisiin testeihin, on paljon selkeää dokumentaatiota sovelluskehyksen kotisivuilla. [10.]

JUnit-integraation lisäksi sovelluskehyksen mukana tulee valmiiksi konfiguroituna Selenium WebDriver. Selenium on suunniteltu frontend-testaukseen, eli sillä voi simuloida käyttäjää varsinaisella verkkosivulla. Seleniumilla voi esimerkiksi määrittää testin, että tietty sivu ladataan, sillä aktivoidaan painike ja 2 s myöhemmin on näkyvissä tiettyä sisältöä.

Selenium mahdollistaa erittäin laajat toiminnallisuuden testaukset: Esimerkiksi koko järjestelmään rekisteröityminen, joka sisältää useita sivulatauksia, voidaan testata yhdellä ainoalla Selenium-testillä.

goMob CMS:n kehityksessä tehtiin useita yrityksiä ottaa Selenium laajamittaiseen käyttöön, sillä potentiaalinen hyöty on valtava. Yritykset kuitenkin epäonnistuivat, sillä erittäin suuri osa sisällöstä frontendissä luotiin dynaamisesti Dojo Toolkit -JavaScript-kirjastolla. Periaatteessa dynaamisesti luotua sisältöäkin olisi mahdollista testata, mutta esimerkiksi dynaamisesti luotujen navigaatioelementtien painallusten simuloiminen todettiin niin työlääksi, että se ei ollut tarvitun vaivan arvoista.

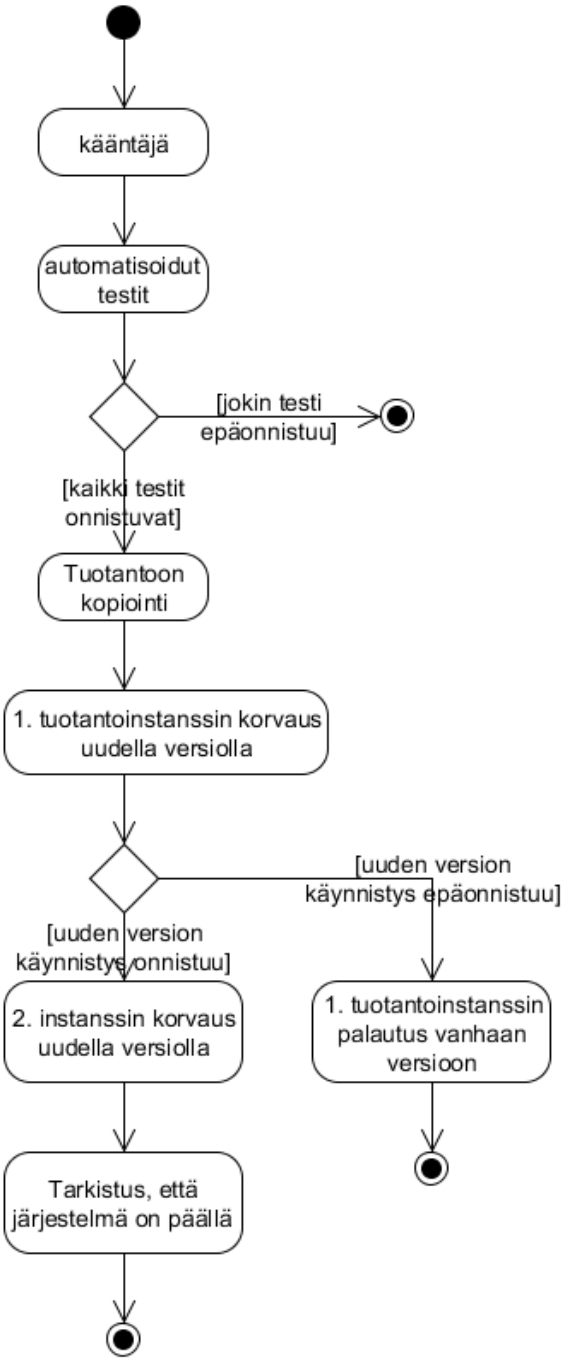
Projektissa kuitenkin päätettiin, että kun uusia elementtejä jatkossa luodaan tai vanhoja muokataan, yritetään ne muokata sellaisiksi, että Selenium-testaus on helpompaa. Näin ollen työkalu saatetaan ottaa käyttöön joskus tulevaisuudessa.

9.2 Makefile-komentosarjat

Kaikille usein tarvituille toimenpiteille on olemassa makefile-tyyppiset komentosarjat. Valtaosa käskyistä on hyvin yksinkertaisia, ympäristön hallintaan liittyviä toimintoja:

- *prod-to-dev* päivittää tietokannan vastaamaan tuotantopalvelinta
- *dev-to-prod* siirtää kehitysympäristön tietokantadatan tuotantopalvelimelle
- *clean* poistaa kaikki käännetyt paketit ja väliaikaistiedostot
- *eclipsify* luo tai päivittää Eclipse-ohjelmointiympäristön projektitiedostot
- *test* ajaa kaikki projektin automatisoidut testit
- *jne.*

Monipuolisin ja tärkein komento on kuitenkin *make deploy*, joka vastaa koko prosessista (kuva 10), joka tarvitaan, että järjestelmän uusin versio saadaan vietyä tuotantoon.



Kuva 10. make deploy -komentosarjan kulku

Make deploy kääntää järjestelmän lähdekoodin ja ajaa kaikki automatisoidut testit, joita on hieman vajaa 300 kappaletta. Käsky varmistaa testien tulostetta tarkkailemalla, että kaikki testit menevät läpi ja jos yksikin epäonnistuu, suoritus keskeytetään.

Kun kaikki testit on suoritettu, käsky siirtää käännetyn paketin tuotantopalvelimelle. Palvelimelle siirretään vain ne tiedostot, jotka ovat muuttuneet uudessa versiossa.

Keskeytymättömän palvelun varmistamiseksi tuotantopalvelimella on koko ajan käynnissä kaksi instanssia järjestelmästä. Make deploy sammuttaa niistä toisen ja koettaa käynnistää sen paikalle järjestelmän uuden version. Mikäli uusi versio ei lähde käyntiin – mikä voi tapahtua, jos esimerkiksi tietokannan rakenne ei vastaa sitä, mitä järjestelmä odottaa – se poistetaan ja make deploy korvaa sen kopiolla vanhasta, toimivasta versiosta. Kun uusi versio on saatu käynnistettyä, toinenkin vanhaa versiota käyttävä instanssi sammutetaan ja korvataan uudella.

Järjestelmän kääntämiseen, siirtämiseen ja käynnistämiseen kuluu merkittävästi aikaa (jopa useita minuutteja), mutta palvelu ei yleensä keskeydy uuden version julkaisun takia sekunniksikaan.

9.3 Palvelimet ja virtuaalikoneet

Projektin alussa – ennen kuin järjestelmän oli laajamittaisessa asiakaskäytössä – käytössä oli vain tuotantoympäristö, ja kaikki kehitys hoidettiin etäyhteyden avulla. Kun tuotannossa olevan järjestelmän muokkauksesta aiheutuneet virhetilanteet ja käyttökatkot eivät enää olleet hyväksyttäviä, kehitys siirrettiin VMWare-ohjelmistojen avulla käytetyille virtuaalikoneille.

Virtuaalikoneiden hyvä puoli oli, että ne pystyttiin pitämään käytännössä identtisenä kopiona tuotantoympäristöstä. goMobin sisäisessä käytössä olleessa wikissä oli lista komentoja, joilla tuoreen CentOS-asennuksen sai vastaamaan tuotantopalvelinta kaikkien koodikirjastojen, käyttöoikeuksien, ym. osalta. Vaikka projektin parissa työskenteli aktiivisesti kaksi ohjelmistokehittäjää ja molemmat usealla eri koneella, toiminnallisuus ja virhetilanteet ilmenivät samanlaisina jokaisessa ympäristössä.

Kehitykseen käytettyjen virtuaalikoneiden ongelmaksi ilmeni kuitenkin suorituskyky. Kun järjestelmän ensimmäiset versiot toimivat PHP:n päällä, sen pyörittämiseen ei vaadittu juuri tehoja niin kauan kuin käyttäjämäärät pysyivät pieninä. Kehityksen painopisteen siirtyessä Javalle tilanne muuttui. Paitsi että Java-virtuaaliympäristö vaatii muistia jo pyöriessään taustalla, jokaisen muutoksen jälkeen pyörivä Java-kääntäjä toimi hyvin hitaasti pyöriessään normaalin työaseman sisällä pyörivässä virtuaalikoneessa.

Jatkuva kääntäjän työn odottelu hidasti työnkulkua näkyvästi, joten kehitystyö oli siirrettävä virtuaalikoneista ja hoidettava työasemilla, jotka saattoivat poiketa toisistaan ja ennen kaikkea tuotantoympäristöstä. Kehityksestä tuotantoon siirretyn ohjelmakoodin ei voinut enää varmuudella luottaa toimivan odotetulla tavalla, joten uuden version toiminta piti testata tuotantoympäristössä – tai sitä identtisesti vastaavassa ympäristössä – ennen vanhan version korvaamista.

Laajasti käytössä oleva best practice jo julkaistujen verkkopalvelujen kehittämisessä on kolmivaiheinen *development – staging – production* -ympäristö. Staging on tuotantopalvelimen kanssa identtinen palvelin eikä mitään siirretä tuotantoon ennen, kuin se on todettu toimivaksi staging-palvelimella. Tähän ratkaisuun oltaisiin luultavasti päädytty goMob CMS:ssäkin, ellei projektissa olisi samaan aikaan kohdattu toinenkin ongelma.

Java-pohjaisen goMob CMS:n alkuvaiheessa järjestelmässä esiintyi muistivuotoja. Kyseessä oli sovelluskehyksen ja sen kanssa tulevan ORM-ympäristön integraatiossa tapahtunut bugi, jonka takia tietokantayhteyksiä ei suljettu oikein ja ajan myötä päällä pidetyn järjestelmän muistijälki kasvoi. goMob CMS:n omasta koodista johtumattoman virheen löytämisessä meni pitkään – yli vuosi sen huomaamisesta – ja siihen asti toimiva workaround oli yksinkertaisesti käynnistää järjestelmä uudelleen aina muistijäljen kasvaessa liian suureksi.

Edellä kuvatusta ongelmasta aiheutuvat käyttökatkot ja tarve staging-ympäristölle olivat täysin toisiinsa liittymättömiä, mutta ne sattuivat ilmenemään samaan aikaan ja päätettiin ratkaista yhdellä iskulla. Tuotantopalvelimella asetettiin pyörimään kaksi rinnakkaista instanssia järjestelmästä. Kun yksi uudelleenkäynnistettiin, toinen instanssi piti yllä palvelua ilman käyttökatkoja.

Tätä samaa voitiin käyttää myös uuden version julkaisussa: instanssit korvattiin uudella versiolla yksi kerrallaan. Jos uusi versio ei käynnistynyt tai osoitti muita merkkejä ongelmista, se voitiin poistaa ja toinen vanhaa versiota pyörittävistä instansseista jatkoi toimintaa koko prosessin ajan.

Periaatteessa ratkaisun varjopuoli on, että staging-palvelimen asiaa ajava instanssi käyttää tuotantopalvelinta ja sen tietokantaa. Jos järjestelmässä olisi kehitysympäristöstä siirrettäessä koko palvelimen lamauttava tai tietokannan sisältöä tuhoava virhe, se vaikuttaisi myös tuotantoympäristöön. Tällaiset virheet kuitenkin koettiin epätodennäköiseksi, eikä sellaisia ole toistaiseksi esiintynyt yhtäkään kappaletta.

10 Yhteenveto

goMob CMS -ohjelmistoprojektin tarkoituksena oli tuottaa startup-yrityksen käyttöön sisällönhallintajärjestelmä. Järjestelmän oli tarkoitus soveltua erityisesti mobiilisivustojen muokkaamiseen ja toimia hyvin yhteen kolmannen osapuolen järjestelmän (NetBiscuits) kanssa. Kolmivuotisen projektin aikana järjestelmän vaatimukset muuttuivat lähes täysin, sillä lopputuloksella piti voida muokata muutakin sisältöä kuin mobiilisivustoja, eikä sillä tarvinut olla enää mitään Netbiscuits-sidosta.

Muuttuneiden vaatimusten takia projektin onnistumisen arviointi alkuperäisten määritysten perusteella ei ole millään tavalla mielekästä. Sen sijaan projektia on tarkasteltava asiakasyrityksen ja sen liiketoiminnan kautta.

Ohjelmistoprojektissa toteutettiin järjestelmä, jonka varaan teknologia-alan startup on pystynyt rakentamaan liiketoimintansa viimeiset kolme vuotta ja jota on käytetty monen suuren suomalaisen mobiilipalvelun kehitykseen. Näiltä osin ohjelmiston voidaan sanoa täyttäneen tehtävänsä erinomaisesti.

Projektin alussa ei tehty riittävän kattavaa vaatimusmäärittelyä ja tästä syystä aikaarviot ja monet tehdyistä valinnoista sekä ohjelman arkkitehtuurin, että ohjelmistotekniikan prosessien hyödyntämisen suhteen, osoittautuivat virheellisiksi. Projektissa kävi hyvin ilmi liian nopeaan aloittamiseen liittyvät riskit sekä niistä myöhemmin kehityksessä maksettava hinta.

Toisaalta vuoden 2011 alussa oli mahdotonta tietää, mihin suuntaan aloittelevan yrityksen liiketoiminta kehittyisi ja täysimittainen vesiputousmallin soveltaminenkaan ei olisi vastannut liiketoimintaympäristön asettamia vaatimuksia. Parhaan mahdollisen tasapainon hakeminen näiden kahden välillä on yksi ohjelmistokehitykselle tyypillisistä ongelmista.

Täysimittaisen järjestelmän suunnittelu ja toteutus yrityskäyttöön oli ammatillisesti erittäin vaativa ja kehittävä kaikille osallistuneille. Tuloksena syntynyt sovellus jää kaupalliseen käyttöön ja raportilla on potentiaalia olla hyvää oppimateriaalia sekä projektinhallintaan että itse tekniseen toteutukseen liittyen.

Lähteet

1 NetBiscuits Feature Phone Product Sheet 2013. 2013. Verkkodokumentti. Netbiscuits. <http://www.netbiscuits.com/wp-content/uploads/2013/10/Feature-Phone-Product-Sheet.pdf>. Luettu 4.2.2014.

2 Go Mobile with SDL Tridion 2011 and Nebiscuits. 2011. Verkkodokumentti. SDL Tridion World. <http://sdltridionworld.com/articles/sdltridion2011/mobileTridion2011Netbiscuits.aspx>. Luettu 10.3.2014.

3 Catcher iTunes. Verkkodokumentti. goMob Finland Oy / Apple. <https://itunes.apple.com/us/app/catcher/id626138687?mt=8>. Luettu 1.6.2014.

4 The MVC application model. Verkkodokumentti. Play! community. <http://www.playframework.com/documentation/1.0/main>. Luettu 24.5.2014.

5 Architectural Styles and the Design of Network-based Software Architectures. Doctoral dissertation. Fielding, Roy Thomas (2000) / University of California, Irvine. <http://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm>

6 EBean ORM online documentation. Verkkodokumentti. Avaje. <http://www.avaje.org/ebean/documentation.html>. Luettu 18.5.2014.

7 Using the Ebean ORM. Verkkodokumentti. Play! community. <https://www.playframework.com/documentation/2.2.x/JavaEbean> Luettu 1.6.2014.

8 The template engine. Verkkodokumentti. Play! community. <https://www.playframework.com/documentation/2.2.x/JavaTemplates>. Luettu 1.6.2014.

9 Usage statistics and market share of Google Analytics for websites. Verkkodokumentti. W3 Technologies. <http://w3techs.com/technologies/details/ta-googleanalytics/all/all>. Luettu 1.6.2014.

10 Testing your application. Verkkodokumentti. Play! community. <http://www.playframework.com/documentation/2.1.1/JavaTest>. Luettu 10.6.2014.

11 Introduction to Test Driven Development (TDD). Verkkodokumentti. The Agile Data. <http://www.agiledata.org/essays/tdd.html>. Luettu 24.5.2014.