INTERNETSOVELLUKSEN TUOTANTOPROSESSI

Kehityksen ja julkaisun hallinta esimerkkisovelluksen ohjaamana

Jussi Rajala

Opinnäytetyö
Joulukuu 2011

Ohjelmistotekniikka
Tekniikan ja liikenteen ala

JYVÄSKYLÄN AMMATTIKORKEAKOULU
JAMK UNIVERSITY OF APPLIED SCIENCES

JYVÄSKYLÄN AMMATTIKORKEAKOULU
JAMK UNIVERSITY OF APPLIED SCIENCES

OPINNÄYTETYÖN
KUVAILULEHTI

Tekijä(t) RAJALA, Jussi	Julkaisun laji Opinnäytetyö	Päivämäärä 9.12.2011
	Sivumäärä 59	Julkaisun kieli Suomi
	Luottamuksellisuus ( ) saakka	Verkkojulkaisulupa myönnetty (X)

Työn nimi
INTERNETSOVELLUKSEN TUOTANTOPROSESSI

Koulutusohjelma

Ohjelmistotekniikka

Työn ohjaaja(t)
MIESKOLAINEN, Matti

Toimeksiantaja(t)
POVELAINEN, Tuomo, Kulttuurituotanto-osuuskunta Raymond

Tiivistelmä

Opinnäytetyössä tutkittiin internetsovelluksen tuotantoprosessin eri vaiheiden kehitystä ja automatisointia käyttäen vertailukohtana kokemuksia ja ongelmakohtia aikaisemmista yhteistyöprojekteista toimeksiantajan kanssa. Työssä tutkittiin myös uusien teknologioiden, kuten palvelinpuolella toimivan JavaScriptin ja dokumenttitietokannan, soveltuvuutta internetsovellusten toteutukseen.

Tutkimustyötä vietiin eteenpäin työn aikana toteutetun tapahtumakalenterisovelluksen avulla. Työ eteni vaiheittain kattaen internetsovelluksen tuotannon eri vaiheet suunnittelusta julkaisuun.

Työn tärkeimpinä tuloksina opittiin automatisoinnin ja sovelluksen elinkaaren sekä infrastruktuurin hallintaan keskittyvien työkalujen ja toimintamallien lukuisat hyödyt. Työssä käytetyt ratkaisut todettiin yleiskäyttöisiksi ja toimiviksi myös aikaisemmin käytetyissä teknologioissa, ja niiden käyttöönotto nähtiin tarpeelliseksi tulevaisuudessa.

Toteutuksessa käytettyjen uusien teknologioiden todettiin soveltuvan tietyntyyppisten internetsovellusten toteutukseen aikaisemmin käytettyjä teknologioita paremmin. Tietoa voidaan käyttää jatkossa hyväksi uusien sovellusprojektien teknologiavalintojen kohdalla.

Avainsanat (asiasanat)
Ohjelmistokehitys, internetsovellus, versiohallinta, Git, Node.js, JavaScript, dokumenttitietokanta, CouchDB, Puppet, julkaisu

Muut tiedot

JYVÄSKYLÄN AMMATTIKORKEAKOULU
JAMK UNIVERSITY OF APPLIED SCIENCES

DESCRIPTION

Author(s)	Type of publication	Date
RAJALA, Jussi	Bachelor's Thesis	9.12.2011
	Pages 59	Language Finnish
	Confidential ( ) Until	Permission for web publication (X)

Title
INTERNET APPLICATION DEVELOPMENT

Degree Programme
Software Engineering

Tutor(s)
MIESKOLAINEN, Matti

Assigned by
POVELAINEN, Tuomo, Kulttuurituotanto-osuuskunta Raymond

Abstract

The purpose of the Bachelor's Thesis was to study the development process and lifetime management of an Internet application in its early stages. The study focused on using automated processes in tasks that have previously required complicated and error-prone manual interactions by developers.

The work also took advantage of many new technologies from the assignee's point of view. These new technologies included document database and server-side JavaScript, and the choices were evaluated in the process.

The process was driven by an example application that was developed from ground up, and the choices were made according to its needs. Most of the work in the thesis consisted of the preparation, planning, testing and implementing of the more automated deployment pipeline for the ease of the application's future development.

The most important finding of the thesis was the process of automated server provisioning that can be used to spawn new development and production environments with small effort. The method will be used by the assignee in future projects with technologies not limited to the ones used in this thesis. Rest of the examined technologies can also be taken into consideration for future projects.

Keywords
Internet application, version control, Git, Node.js, JavaScript, server-side, document database, CouchDB, provisioning, Puppet, deployment

Miscellaneous

SISÄLTÖ

KUVIOT

KÄYTETYT TERMIT JA LYHENTEET

Acceptance test, hyväksymistesti

Korkean tason testitapaus, joka on usein kirjoitettu käyttäjän näkökulmasta. Testi kattaa yksittäisen toteutettavan ominaisuuden tai tehtäväkokonaisuuden, ja testin läpäisy toimii usein indikaattorina ominaisuuden valmistumiselle. Yksittäinen asiakasvaatimus saattaa kääntyä suoraan acceptance-testitapaukseksi.

ACID; Atomic, Consistency, Isolation, Durability

ACID (atomisuus, eheys, eristyneisyys, pysyvyys) tarkoittaa tietokantajärjestelmien periaatetta, jota noudattava järjestelmä lupaa taata eheytensä kaikissa tilanteissa. Atomisuus tarkoittaa transaktioiden suorittamista kokonaan onnistuneesti tai ei ollenkaan, eheys sallii järjestelmän siirtymisen ainoastaan eheästä tilasta toiseen eheään tilaan, eristyneisyys määrittää transaktioiden olevan riippumattomuuden toisistaan, ja pysyvyys takaa suoritettujen transaktioiden säilyvyyden.

Bash

Bash on POSIX-yhteensopiva komentotulkki, joka on esimerkiksi useimpien Linux-jakeluiden oletuskomentotulkki.

BDD, Behavior Driven Development

BDD on testauskäytäntö, joka laajentaa TDD:tä tuomalla projektin vähemmän tekniset osapuolet, kuten testaajat ja asiakkaan, mukaan automatisoidun testauksen prosessiin. BDD käsittää korkean tason testejä, jotka usein kääntyvät suoraan asiakasvaatimuksiksi. Automaattisesti ajettavat testitapaukset kirjoitetaan käyttäjän näkökulmasta, ja ne kuvaavat lähtötilanteen, suoritetun toimenpiteen ja toivotun lopputuloksen. BDD-työkalut mahdollistavat testitapausten kirjoituksen lähes luonnollisella kielellä.

Bootstrap, Bootstrapping, Booting

Bootstrap on monimerkityksellinen metafora omavaraisesta itsenäisesti tapahtuvasta, useimmiten käynnistykseen tai lataukseen liittyvästä, prosessista. Bootstrap voi tarkoittaa esimerkiksi ensimmäistä asennusta tai käynnistysajoa ympäristöön, joka ei vielä sisällä tavanomaisesti käytettyjä lataus-, asennus- tai päivitystyökaluja.

Branch (Git-versiohallinnassa)

Versiohallinnan kehityshaara, jolla on oma työtilansa (workspace). Versiohallinnan kehitystyö tehdään kehityshaarassa, jolla on oma lineaarinen muutoshistoriansa. Uusi haara luodaan kopiona olemassa olevasta haarasta, ja kaksi haaraa voidaan yhdistää takaisin yhdeksi (merge). Kehitystyö on mahdollista erillään useassa haarassa samaan aikaan, ja Git-versiohallinnassa kehityksen suositellaan tapahtuman haaroissa.

CRUD (Create Read Update Delete)

CRUD on esimerkiksi internetsovellukselle tyypillinen toimintamalli, jossa ohjelmankulku koostuu neljästä resurssille suoritettavasta perusoperaatiosta: luonti, luku, päivitys ja poisto.

DSL, Domain-specific Language

DSL, eli täsmäkieli tai aihekohtainen kieli, on ohjelmointikieli, joka on suunnattu tarkoin määrätylle sovellusalueelle. Täsmäkielen tarkoitus on käyttää toimialalle ominaisia termejä ja rakenteita rajatun ongelma-alueen ratkaisuun. Täsmäkieli voidaan toteuttaa geneerisen ohjelmointikielen tietorakenteilla, joissa käytetään ongelma-alueen termejä. Toinen vaihtoehto on luoda oma tulkattava täsmäkieli, joka rajoittuu ongelma-alueen keskeisiin termeihin, eikä sisällä kaikkia geneerisen ohjelmointikielen ominaisuuksia.

I/O, Input / Output, siirräntä

I/O on yleistermi tiedon siirtämiselle esimerkiksi laitteiden, sovellusten, sovelluksen ja käyttöjärjestelmän tai käyttöjärjestelmän ja laitteiston välillä. Tiedon siirto voi tapahtua esimerkiksi ohjelmamuistin, fyysisten porttiväylien tai verkon välityksellä.

JSON, JavaScript Object Notation

JSON on kompakti tiedon sarjallistamismuoto, joka pohjautuu JavaScript-kielen objektinotaatioon. JSON-formaatti on kuitenkin kieliriippumaton, ja sitä käytetään usein HTML-formaatin rinnalla tai sijaan HTTP-sovelluksissa palvelimen ja asiakkaan välillä tapahtuvaan tiedonsiirtoon.

Make

Make on varsinkin Unix-järjestelmissä yleisesti käytetty työkalu tiedostojen luonnin, käännöstyön ja tehtävien automatisointiin. Työkalu suorittaa Makefile-sääntötiedostoon määriteltyjä tehtäviä. Make on luonteeltaan deklaratiivinen, ja tehtävistä suoritetaan lähtötilanteen mukaan ainoastaan tarvittavaksi havaitut. Tehtävillä voi olla riippuvuussuhteita toisiinsa.

MVCC, MCC, Multiversion concurrency control

MVCC on menetelmä, jota useat tietokantaohjelmistot käyttävät yhtäaikaisten operaatioiden hallintaan. Termi kuvaa tapaa, jossa merkinnän päivitys tai luonti ei muuta vanhaa tietoa, vaan uusi tieto kirjoitetaan uuteen sijaintiin ja kirjoituksen jälkeen vanha merkintä merkitään vanhentuneeksi ja nykyisen version osoitin asetetaan osoittamaan uutta versiota. Merkinnät saavat oman juoksevan versionumeronsa, ja toteutuksesta riippuen vanhoja versioita voidaan säilyttää jonkin aikaa. Menetelmää käyttävä järjestelmä on jatkuvasti eheässä tilassa ja tarjoaa siten korkean virhesietoisuuden, sekä mahdollisuuden lukea tietokantaa myös kirjoituksen tapahtuessa.

POSIX, Portable Operating System Interface for Unix

POSIX IEEE -standardiperhe määrittää yhteiset rajapinnat ja käyttäjätason ohjelmat sekä järjestelmäkutsut Unix-perheen käyttöjärjestelmille. Määrityksen tarkoitus on varmistaa yhteensopivuus ja estää pirstaloitumista käyttöjärjestelmien välillä.

Repository

Versiohallinnan koodiarkisto, joka pitää sisällään versiohallinnassa säilytettävät tiedostot, niiden muutoshistorian kokonaisuudessaan sekä versiohallintaohjelmasta riippuvaa metadataa, kuten haarat (branch) ja merkinnät (tag).

REST, Representational state transfer

Ohjelmistoarkkitehtuuri hajautetuille hypermediasovelluksille. Arkkitehtuuria käytetään laajasti esimerkiksi www-ympäristössä. Termi on lähtöisin Roy Fieldingin väitöskirjasta vuodelta 2000. Arkkitehtuuri perustuu vahvasti HTTP-protokollaan, jonka kehityksessä Fielding oli osallisena. Arkkitehtuurin olennaisia osia ovat hyperlinkitys ja palvelimilla säilytettävät resurssit, joiden kanssa asiakas toimii HTTP-protokollan avulla siten, että kaikki käytönaikainen tilatieto paketoidaan HTTP-kutsuihin ja vastauksiin.

RSA

RSA on epäsymmetrinen alkulukuihin perustuva julkisen ja yksityisen avaimen salausalgoritmi. Tässä projektissa RSA-avainparin rooli oli mahdollistaa salasanavapaa kirjautuminen yksityisen avaimen haltijalle siten, että kirjautumisen kohteessa säilytettiin avainparin julkista avainta.

Smoke test

Smoke test on kevyehkö testi, jonka tarkoitus on osoittaa selkeimmät ja räikeimmät ongelmatapaukset, jotka estävät ohjelman toiminnan ja joiden esiintyessä testausta tai operaation suoritusta ei ole tarpeellista jatkaa, sillä se on varmasti epäonnistunut. Nimi juontuu selkeästi viallisesta sähkölaitteesta, joka päästää savupilven, kun laitteeseen kytketään virta.

Tag (Git-versiohallinnassa)

Git-versiohallinnassa tag tarkoittaa valittuun haaraan (branch) tehtyä merkintää, joka määrittää jonkin merkittävän hetken haaran aikahistoriassa. Yleensä tagia käytetään merkitsemään julkaisuun tarkoitettu versio aikahistoriassa, jolloin tagi voidaan nimetä muotoon v1.2.3. Versiohallintaohjelma tarjoaa komennot, joilla projektissa merkitsemishetkellä ollut sisältö voidaan noutaa myöhemmin.

TDD, Test Driven Development

TDD tarkoitaa testauslähtöistä kehitystä, jossa järjestelmää tarkastellaan ulkopuolelta siten, että jokaiselle toteutettavalle toiminnalle, ns. tuotantokoodille, kirjoitetaan aluksi epäonnistuva yksikkötesti. Testin luonnin ja ajon jälkeen to-

teutetaan ainoastaan tarvittava määrä tuotantokoodia testin läpäisyksi. Ulkopuolelta tarkastelulla tarkoitetaan sitä, että aluksi kirjoitettava testikoodi kutsuu uuden ominaisuuden rajapintaa ominaisuuden käyttäjän näkökulmasta, ja siten TDD-kehityksessä ominaisuuden rajapinta määritetään testin kirjoituksen aikana käyttäjän, eli testin, näkökulmasta.

1 TYÖN LÄHTÖKOHTA

1.1 Toimeksiantaja

Työn tilaajana toimi Kulttuurituotanto-osuuskunta Raymondin riveissä vuodesta 2008 työkennellyt graafinen suunnittelija Tuomo Povelainen. Raymond on Jyväskylässä 2005 perustettu vahvojen kulttuuri- ja media-alan osaajien freelance-kollektiivi. (Kulttuurituotanto-osuuskunta Raymond tietosivu 2011.)

Tuomo Povelaisen työnkuva ja ydinosaaminen ovat graafisessa suunnittelussa. Toteutetut projektit vaihtelevat erilaisista painotuotteista internetsivustojen ulkoasun ja toiminnan suunnitteluun. Toiminnallisesti laajempaa toimintalogiikkaa sisältävät internetsivustot ja sovellukset toteutetaan tarvittaessa yhteistyökumppaneista löytyvien asiantuntijoiden kanssa, ja toisaalta useat internetpalveluiden tuottajat tilaavat Povelaiselta suunnittelutyön sekä projektin graafisen ilmeen. Asiakaskunta on kokoluokaltaan hyvin vaihtelevaa. Opinnäytetyön tilaaja ja tekijä ovat tehneet aikaisemmin yhteistyötä erilaisten internetsivustojen ja palveluiden toteutuksessa.

Opinnäytetyön toimeksiannon lähtökohtana oli toteuttaa asiakkaan henkilökohtaisena projektina yksinkertainen internetsivusto, joka listaa Jyväskylän kulttuuritapahtumia keskittyen aluksi elävän musiikin tapahtumatarjontaan. Sivuston kävijän tulisi saada sivustolta vaivattomasti ja nopeasti ajantasainen kuva kaupungin tulevista tapahtumista. Työn aloituksen hetkellä tulevista tapahtumista kiinnostuneen käyttäjän täytyi etsiä tulevat ohjelmatiedot kunkin yksittäisen tapahtumapaikan internetsivulta.

Tapahtumakalenterisivuston ensimmäiseksi ongelmaksi todettiin sisällön ajantasaisuus sekä kattavuus. Ongelmaa päätettiin lähestyä siten, että sivustolle tuotettaisiin alkuvaiheessa mahdollisimman vähän omaa sisältöä, jolloin ylläpitotyö ja sisällön ajantasaisuus pysyisivät kohtuullisina. Tapahtumatiedot päätettiin hakea ohjelmallisesti tunnettujen tapahtumapaikkojen internetsivuilta ja tarjota ylläpitäjälle mahdollisuus päivittää automaattisesti tuotuja tietoja tarvittaessa.

1.2 Ongelmat käytetyissä toimintamalleissa

Yhteisinä projekteina aikaisemmin tuotetut sivustot ja palvelut on toteutettu PHP-kielellä itse tuotettujen ja kolmannen osapuolen kirjastojen päälle. Projektit on toteutettu siten, että kehittäjä on työstänyt paikallisella koneellaan Apache Subversion -versiohallinnassa säilytettävää projektia käyttäen apunaan Windows-käyttöjärjestelmään asennettua sovellusympäristöä ja palvelinalustaa. Kehittäjä on asentanut ja konfiguroinut kehitysympäristönsä käsin, ja alustan asetukset on pyritty saattamaan mahdollisimman lähelle tuotantopalvelimen asetuksia. Käytännössä paikallisen koneen alusta ja konfiguraatiot ovat kuitenkin eronneet julkisesta ajoympäristöstä, josta on aiheutunut lukuisia aikaa kuluttavia ongelmatilanteita. Tuotantopalvelimen konfiguraatioita on muutettu suoraan palvelimella, jolloin operaatiosta on usein ollut ainoastaan suullinen tieto kehittäjillä.

Julkaisukuntoiseksi todetun internetsovelluksen toimitus on tapahtunut siten, että sovelluksen kehittäjä on kopioinut sovelluksen ohjelmakoodit ja staattiset resurssit, kuten kuva- ja tyylitiedostot, manuaalisesti tiedostonsiirto-ohjelmalla tuotantopalvelimelle. Kopioinnin jälkeen, tai tapauksesta riippuen kopiointia ennen, tuotantopalvelimella sijaitsevat muut resurssit, kuten mahdollinen tietokanta ja sovelluksen ympäristökohtaiset konfiguraatiot, on saatettu käsin ajan tasalle. Esimerkiksi edellisen julkaisuversion jälkeen kehityksessä tapahtuneet tietokantamuutokset on toistettu palvelimen tietokantaan komentorivin tai tietokannan hallintapaneelin kautta.

Manuaaliset vaiheet ovat riski ohjelmistoprojektissa, sillä inhimillisen erehdyksen mahdollisuus on niissä suuri. Automatisoimattomat vaiheet eivät myöskään ole varmuudella toistettavissa täysin samalla tavalla, ja toimintavaiheista ei usein jää historiamerkintää lokiin myöhempää tarkastelua varten. (Humble & Farley 2011, 19.)

1.3 Tavoitteeksi toimintamallien kehitys

Toimeksiantajan ja työn tekijän aikaisemmat projektit ovat olleen projektin asiakkaan budjettiin ja aikatauluun sidottuja, minkä vuoksi projektit on toteutettu tutuilla ja toimivaksi todetuilla ratkaisuilla. Aikaisemmat projektit eivät siis

pääsääntöisesti ole tarjonneet mahdollisuutta kehittyä ja kokeilla uusia tekniikoita ja käytäntöjä. Joustavuutensa ja rahoitustekijöistä riippumattomuutensa vuoksi tapahtumakalenteriprojektille annettiin tavoitteeksi toimia tutkimusprojektina uusien teknologioiden ja toimintamallien käyttöönotossa.

Vanhoista käytännöistä paljastuneiden manuaalisesti suoritettujen automatisoitavissa olevien työvaiheiden automatisointi ja siitä saatujen oppien sekä työkalujen hyödyntäminen koettiin ensiarvoisen tärkeäksi tavoitteeksi pidemmällä tähtäimellä. Automatisointi on ensiarvoisen tärkeässä roolissa kaikilla ohjelmistoteknologian eri saroilla (Humble & Farley 2011, 17 - 24). Julkaisustrategian kehityksessä todettiin tarve kiinnittää huomiota vähintäänkin seuraaviin kohtiin:

- Julkaisuun osallistuvat osapuolet ja oikeuksienhallinta
- Resurssien ja konfiguraatioiden hallinta, tietokannan ja mahdollisten muiden tietovarastojen migraatioiden hallinta
- Tieto julkaisuversioista kehityshistoriassa, versiohallinnassa ja tuotantoympäristössä
- Tietoisuus käytössä olevista ja hallittavista käyttöjärjestelmä- ja sovellustason ympäristöistä, sekä hallintasuunnitelma
- Jatkuva testaus
- Yhtenäinen automatisoitu julkaisukäytäntö
- Varasuunnitelma virhetilanteiden varalle julkaisutilanteessa.

(Humble & Farley 2011; Freeman & Pryce 2010.)

1.4 Tutkimus uusista teknologioista

Toimintamallien tehostustarpeiden lisäksi työlle asetettiin toinen tutkimusluontoinen tavoite. Ennestään tuttu PHP-ympäristö todettiin soveltuvuudeltaan epäoptimaaliseksi ratkaisuksi tapahtumakalenterin automaattisen tapahtuma-

keruun toteuttamiseen. Lisäksi uudesta alustasta kokemusten saanti nähtiin hyödylliseksi.

Työn toteutukseen valituille uusille teknologioille, kuten dokumenttitietokannalle ja Node.js-ohjelmistoalustalle, arvioitiin useita todennäköisiä käyttötarpeita lähitulevaisuudessa, mikäli niiden todettaisiin vastaavan ennakko-odotuksia. Tulevaisuudessa nähdyt tarpeet vaihtelivat toisista tiedonkeruuprojekteista selainpeleihin. Dokumenttitietokanta nähtiin etukäteen mielenkiintoisena alustana monimuotoisen tiedon tallentamiseen.

Kehitysalustaksi valittiin myös palvelinpuolella käytetty Ubuntu Linux, sillä homogeeninen ympäristö vähentää yhteensopivuusongelmia, ja lisäksi useimmat projektissa tarkasteltavista teknologioista ja työkaluista olivat lähtökohtaisesti POSIX-yhteensopivalle järjestelmälle kehitettyjä ja komentorivipohjaisia.

2 VERSIOHALLINTA

2.1 Yleistä

Ohjelmistoprojektien lähdekoodit säilytetään lähes aina versiohallinnassa. Versiohallinnan tärkeimpiä tehtäviä on mahdollistaa projektin kehitys usean kehittäjän kesken samaan aikaan ja tarjota ratkaisut yksittäisten kehittäjien tekemien muutosten yhdistämiseen ristiriitoja välttäen. Mikäli ristiriitatilanteita ilmenee, versiohallintatyökalu avustaa niiden ratkaisussa. Versiohallintatyökalu tarjoaa projektille myös kommentoidun ja selkeän historian, josta tulee selvitä projektin tila sen luonnista nykyiseen hetkeen asti, projektin jokainen versio, sekä vastaus kysymykseen kuka yksittäisen muutoksen teki, miksi ja koska. (Sink 2011.)

2.2 Kokemukset keskitetystä mallista

Työn osapuolien aikaisemmissa projekteissa on käytetty Apache Subversion -versiohallintaa, joka on luonteeltaan keskitetty. Keskitetyssä versiohallinnassa on yksi yhteinen repository, jonka kanssa asiakkaat synkronisoivat paikallisen työversionsa. Subversion-asiakkaat eli yksittäiset kehityskoneet, joissa työn teko tapahtuu, sisältävät vain ajankohtaisen tai hakutapahtumassa määrätyn version projektista, ja historian tarkastelu tapahtuu keskitetyn repositoryn kautta.

Keskitetty versiohallintamalli on siis täysin riippuvainen keskusrepositorysta. Keskuspalvelimen varmuuskopiointi on siten ensiarvoisen tärkeää. Toimintatapa ei myöskään mahdollista versiohallinnan hyödyntämistä tilanteissa, jolloin repositoryyn ei saada yhteyttä.

Aikaisemmissa keskitettyä versiohallintaa käyttävissä projekteissa repository oli sijoitettu toimiympäristön sisäiselle palvelimelle, jonka käyttöoikeudet ovat olleet rajoitettuja. Keskitetty malli on palvellut muutaman kehittäjän kattavaa ympäristöä hyvin. Tilanteet, joissa kehittäjä ei ole ollut samassa verkossa kes-

kuspalvelimen kanssa, on hoidettu joko salatulla VPN-yhteydellä, tai internetyhteyden puuttuessa paikallisia muutoksia on tullut tavallista enemmän yhdellä kertaa, koska tekijä ei ole päässyt synkronisoimaan töitään.

Viivästyneistä tai suureksi kasvaneista synkronisoimattomista muutoksista on aiheutunut lukuisia pieniä ja muutamia suurempia työaikaa kuluttavia ongelmia. Viivästynyt synkronisaatioväli on kasvattanut riskiä ristiriitojen syntymisille. Ristiriitoja syntyy, kun kaksi osapuolta tekee yhtäaikaisia muutoksia lähdekoodin samaan osaan.

Keskuspalvelimen ulottumattomissa olleen kehittäjän näkökulmasta suurin ongelma on aiheutunut siitä, ettei tekijä ole päässyt hyödyntämään versiohallinnan tarjoamaa välitallennuksen turvaa. Versiohallinnan avulla kehittäjä voisi jakaa tekemänsä muutokset pieniin loogisiin kokonaisuuksiin, jotka valmistuessaan synkronisoidaan keskitetyn versiohallinnan palvelimelle. Mikäli kehittäjä toteaa seuraavan muutoksen epäonnistuneen lähtökohtaisesti tai rikkoneen jotain niin pahasti, ettei korjaaminen ole järkevää, hän on voinut helposti palata edellisen synkronisaation aikaiseen puhtaaseen tilanteeseen. Käytännössä tätä ongelmaa on ratkaistu versiohallintaa edeltävässä maailmassa käytetyllä ratkaisulla, jossa muokattavasta tiedostosta on otettu nimettyjä välikopioita.

Etätyöskentelyn jälkeen suoritetut liian suureksi kasvaneet, monta loogisesti toisiinsa liittymätöntä muutosta sisältävät, synkronisaatiot eivät ole näkyneet projektin historiassa selkeinä kokonaisuuksina. Joissain tapauksissa tällaisia isoja muutoskokonaisuuksia on jouduttu myös viemään sisarprojekteihin, ja muutosten suuri määrä on taannut lähes poikkeuksetta työläästi ratkottavan ristiriitatilanteen syntymisen.

2.3 Keskitetystä hajautettuun malliin

Keskitetty Subversion-versiohallinta on toiminut rajoitteidensa puitteissa hyvin. Viime vuosina keskitetyn mallin suosiota on kuitenkin syrjäyttänyt hajautettujen versiohallintaratkaisujen yleistyminen. Kirjoitushetkellä tunnetuimpia avoimen lähdekoodin hajautettuja versiohallintatyökaluja ovat Git, Mercurial ja Bazaar. Kullakin ohjelmalla on vankka käyttäjäkuntansa, ja ainakin Git sekä Mer-

curial ovat teknisesti suhteellisen lähellä toisiaan, joten valinta näiden väliltä määräytyy usein projektin osallisten henkilökohtaisten mieltymysten mukaan. Kummallakin edellä mainituista on käyttäjinään suuria nimiä. Gitin käyttäjiin kuuluvat mm. Linux Kernel, Perl, Fedora ja Android. Mercurialin käyttäjäkuntaan puolestaan lukeutuvat esimerkiksi Mozilla, OpenJDK, Symbian ja OpenSolaris. (Git vs. Mercurial n.d.)

Projektin versiohallinnaksi valittiin Git. Merkittävin syy valintaan oli sen laaja suosio avoimen lähdekoodin projekteissa. Git oli myös ylivoimaisesti suosituin versiohallintajärjestelmä projektiin valitun Node.js -alustan aktiivisen yhteisön keskuudessa. Valtaosaa moduuleista kehitettiin avoimesti Github-sivustolla. Työskentely kolmannen osapuolen moduulien kanssa oli helpompaa, kun käytössä oli sama versiohallintajärjestelmä. Myös osa työssä käytetyistä aputyökaluista sijaitsivat Github-sivustolla, jonka kautta apuohjelmien kehitys ja levitys tapahtuu käyttäjälle läpinäkyvästi. Aputyökalut tuotiin kehityskoneelle kloonaamalla Github-sivustolla sijaitseva repository paikalliselle koneelle.

Git on Linux-ytimen kehittäjän Linus Torvaldsin alun perin vuonna 2005 kehittämä versiohallintatyökalu, tai oikeammin kokoelma työkaluja, POSIX-yhteensopiville käyttöjärjestelmille. Gitin tärkeimpinä suunnitteluperiaatteina ovat nopeus, hajautettu toimintamalli sekä datan eheyden varmistaminen. (Chacon 2010.)

Git ei käytä juoksevaa versionumerointia versiohistoriassaan, kuten perinteiset keskitetyt järjestelmät tekevät, sillä hajautetun versiohallinnan historia ei ole välttämättä lineaarinen. Jokainen versioitu koodimuutospaketti (commit) on yksittäinen kokonaisuutensa, jonka tunnisteena toimii SHA1-algoritmilla paketista laskettu tarkistussumma. Tarkistussumma takaa myös datan eheyden, sillä yksittäisen kokonaisuuden eheys voidaan aina tarkistaa laskemalla sen tarkistussumma uudelleen ja vertaamalla sitä ilmoitettuun arvoon.

Hajautetussa versiohallinnassa ei ole keskitettyä repositorya teknisessä mielessä, vaan jokainen, esimerkiksi kehittäjän koneella sijaitseva, instanssi on itsenäisesti toimiva yksikkönsä. Käytännössä projektia työstävä henkilö sai siis koneelleen täydellisen kopion projektista versiohistorian kanssa. Projektia voitiin kehittää paikallisesti omaa repositorya vasten siten, että koodimuutokset

voitiin tallentaa lähettämällä ne paikalliseen repositryyn (commit), jolloin tallennuksesta jäi merkintä versiohallinnan historiaan ja merkinnät mahdollistivat projektissa merkintähetkellä vallinneeseen tilaan palaamisen jälkikäteen. Ominaisuus ratkaisi aikaisemmin todetun keskitettyä versiohallintaa vaivanneen ongelman tilanteissa, joissa kehittäjällä ei ollut yhteyttä keskitettyyn palvelimeen.

Erilliset Git-repositoryt voitiin synkronisoida toistensa kanssa siten, että edellisen synkronisoinnin jälkeen tapahtuneet paikallista repositorya vasten suoritetut muutokset vietiin toiseen repositoryyn kattaen koko muutoshistorian ja yksittäiset muutokset. Käytännössä monen käyttäjän ympäristössä synkronisaatio tulisi suorittaa mahdollisimman usein, vähintäänkin useita kertoja päivässä, ristiriitatilanteiden minimoimiseksi. (Humble & Farley 2011, 396.)

Git-repositoryn synkronisaatio tapahtuu kaksisuuntaisesti: käyttäjä voi joko hakea etärepositoryn muutokset omaan versiohallintaansa (pull) tai työntää paikalliset muutoksensa etärepositoryyn (push). Synkronisaatiot määritetään haarakohtaisesti, jolloin yksittäinen kehittäjä voi luoda tarvittaessa omia henkilökohtaisia kehityshaarojaan. Git salliii työntöoperaation oletusarvoisesti ainoastaan bare-tyyppiselle repositorylle, joka ei sisällä aktiivista työskentelytilaa. Synkronisaatiopisteeksi tarkoitetut Git-repositoryt luodaan bare-tyyppisinä.

Hajautetulla versiohallinnalla voidaan toteuttaa sopimusluontoisesti samankaltainen toimintamalli kuin keskitetyllä versiohallinnalla. Tällöin sovitaan yksi yhteinen keskuspiste, jonka kanssa kaikki projektissa työskentelevät synkronisoivat paikalliset repositorynsa säännöllisesti. Synkronisaatio tapahtuu hakemalla aluksi muualta keskusrepositoryyn tulleet muutokset ja työntämällä omat muutokset sinne. Tämä onkin usein selkeytensä vuoksi järkevä toimintamalli, ja sitä sovellettiin myös tässä projektissa.

Keskitetysti käytetty hajautettu versiohallinta tarjosi lukuisia hyötyjä, sillä paikalliset itsenäiset repositoryt mahdollistivat versiohallinnan monipuolisen hyödyntämisen paikallisesti, jolloin oli helppo kokeilla erilaisia riskejä sisältäviä muutoksia. Epävarmat ja kokeiluluontoiset muutokset voitiin toteuttaa omassa paikallisessa kokeiluhaarassa, jota ei synkronisoitu keskuspalvelimen versio-

hallintaan. Tällöin tekijä saattoi kuitenkin suorittaa välitallennuksia ja palata tarvittaessa aikaisempiin muutoksiinsa. Mikäli kokeilu todettiin huonoksi, koko kokeiluhaara voitiin poistaa siten, ettei se jättänyt jälkeä projektiin. Jos ominaisuus todettiin toimivaksi, voitiin kokeiluhaara ja koko sen muutoshistoria synkronisoida aktiiviseen kehityspuuhun tai synkronisoida haara myös keskuspalvelimelle jatkokehitystä varten.

Git ja hajautetut malli tarjosivat myös erään käytännössä erityisen hyväksi havaitun ominaisuuden: mikä tahansa projekti tai hakemisto oli helppo asettaa versiohallinnan alaisuuteen, jos siihen nähtiin pientäkään aihetta. Koska paikallinen repository on täysiverinen, voitiin mikä tahansa hakemisto alustaa Git-repositoryksi yhdellä komennolla ja hyödyntää versiohallinnan tarjoamaa muutoshistoriaa, vaikka repository jäisi paikalliseksi.

Keskitetyllä mallilla toimivan versiohallinnan hyödyntäminen satunnaisille kohteille olisi vaatinut joko uuden hakemiston luomista olemassa olevan yhteisen repositoryn sisään tai kokonaan uuden repositoryn luontia, mikä olisi vaatinut komentojen suoritusta keskuspalvelimella. Olemassa olevaan repositoryyn luotu hakemisto jättäisi merkintänsä ja painolastinsa repositoryn historiaan, vaikka kohde todettaisiin myöhemmin hyödyttömäksi ja poistettaisiin.

Hajautetulla versiohallinnalla hallittu paikallinen projekti voitiin lähettää yhteisesti sovitulle keskuspalvelimelle vasta, kun kohde oli todettu kokeiluvaiheen myötä säilyttämisen arvoiseksi. Mikäli kokeilu osoitti kohteen tarpeettomaksi, voitiin se yksinkertaisesti tuhota tai jättää paikalliseksi kopioksi, jolloin keskitetty repository säilyi siistinä. Kokeilun kehityksen aikana voitiin kuitenkin saada täysimittainen hyöty versiohallinnan käytöstä.

2.4 Toimintamalli ja Git Flow

Hajautetun versiohallinnan kehityshaara muodostaa oman lineaarisen historiansa projektin kokonaishistoriaan. Versiohallinnalla hallittava projekti voi sisältää useita yhtäaikaisia aktiivisia haaroja, jotka syntyvät toisista haaroista ja joiden elinkaari voidaan päättää yhdistämällä ne takaisin toisiin haaroihin. Yhtäaikaiset haarat muodostavat projektille kokonaisuutena puumaisen historian. (Sink 2011.)

Eräs Gitin vahvuuksista olivat nopeat ja halvat kehityshaarat. Kehityshaaran luonti tapahtui käytännössä välittömästi, ja haarat voitiin synkronisoinnin yhteydessä joko lähettää keskuspalvelimelle tai säilyttää paikallisina ennen takaisin pääkehityshaaraan yhdistämistä tai haaran hylkäämistä. Työversion vaihto kehityshaarasta toiseen oli erittäin nopeaa ja vaivatonta, jolloin yhdellä kehityskoneella saatettiin yhtä aikaa työstää jotain suurempaa muutosta tai ominaisuutta omassa haarassaan ja vaihtaa tarvittaessa esimerkiksi virheenkorjaushaaraan, jossa korjattiin jokin virhe tai haavoittuvuus. Valmiit haarat voitiin yhdistää takaisin pääkehityshaaraan, jolloin myös haaran kehityshistoria säilyi projektin yhteisessä historiassa. Git mahollisti myös haaran historian siistimisen ennen sen takaisintuontia, mutta kyseiselle ominaisuudelle ei koettu tarvetta tässä projektissa. (Git Tutorial 2011.)

Git mukautuu hyvin erilaisiin kehitysmalleihin, mutta käytännössä on erityisen hyödyllistä sopia projekti- ja tiimikohtaisesti yhteiset käytännöt. Yhteiseksi malliksi tapahtumakalenteriprojektille valittiin Git Flow -termillä esitelty toimintakulku, joka perustuu kahteen jatkuvasti säilytettävään kehityshaaraan: aktiiviseen kehityshaaraan ja vakaaseen julkaisuhaaraan. Näiden lisäksi käytetään paljon tilapäisiä haaroja.

Develop-nimellä kulkeva aktiivinen kehityshaara sisälsi aina tuoreimman kehitysversion projektista. Aktiivinen kehitystyö tehtiin tähän haaraan. Kehityksen apuna käytettiin kertakäyttöisiä ominaisuushaaroja (feature branch), jotka nimettiin haarassa kehitettävän toiminnon mukaan. Jokainen sovelluksen yksittäinen ominaisuus tai toiminto pyrittiin kehittämään erillään tällaisessa ominaisuushaarassa. Kun ominaisuus todettiin valmiiksi, se yhdistettiin historioineen takaisin aktiiviseen kehityshaaraan ja erillinen ominaisuushaara tuhottiin. Git mahdollisti helposti usean toiminnon samanaikaisen kehittämisen aktiivista kehityshaaraa vaihtelemalla.

Kun aktiivisen kehityshaaran todettiin lähestyvän julkaisukelpoisuutta, tehtiin siitä erillinen tulevan versionumeron mukaan nimetty julkaisuhaara (release branch). Julkaisuhaara jäädytettiin siten, että siihen ei enää lisätty uusia ominaisuuksia, ja aktiivista kehitystä saatettiin jatkaa keskeytyksettä aktiiviseen kehityshaaraan. Julkaisuhaara olisi voitu tarvittaessa viedä esimerkiksi rajoitetun käyttäjäkunnan käyttöön staging-palvelimelle, jossa tulevaa julkaisuversio-

ta olisi testattu oikeassa käytössä mutta rajatulla käyttäjäryhmällä. Mikäli kyseessä olisi ollut ulkopuolisen asiakkaan projekti, olisi tuotteen omistaja voitu ohjata palvelimelle testaamaan tulevaa julkaisuversiota ennen sen julkistusta.

Ominaisuuksilta jäädytettyyn julkaisuhaaraan sallittiin ainoastaan korjausten teko, mikäli siinä todettiin testauksen aikana ongelmia. Mahdolliset korjaukset vietiin myös aktiiviseen kehityshaaraan. Kun julkaisuhaara todettiin vakaaksi, yhdistettiin julkaisuhaara vakaaseen päähaaraan (master) ja julkaisuhaara tuhottiin. Yhdistämisen yhteydessä vakaaseen päähaaraan tehtiin julkaisun versionumeron mukainen historiamerkintä (tag).

Git Flow -mallin mukaan vakaaseen päähaaraan ei koskaan tehdä muutoksia suoraan. Päähaara sisältää aina erillisen julkaisuhaaran kautta tuodun vakaan ja versionumeroidun julkaisun. Ainoastaan vakavat ja kiireelliset virhekorjaukset voidaan tuoda erillisen korjaushaaran avulla päähaaraan. Päähaaran tulee siis sisältää aina ohjelman uusin vakaa versio. Jos kyseessä on internetsovellus, voitaisiin päähaaran uusin versio viedä automaattisesti julkiselle palvelimelle. Mikäli kyseessä olisi erillinen paketoitava ohjelma, päähaaraan tuodusta uudesta versiosta voitaisiin rakentaa automaattisesti ohjelmapaketti, joka julkaistaisiin paketoinnin jälkeen automaattisesti esimerkiksi internetsivuilla tai pakettienhallintajärjestelmissä. (Driessen 2010.)

2.5 Kaikki versiohallintaan

Perinteisesti versiohallinnan alla on säilytetty pääasiassa projektin ohjelmakoodeja. Työn suunnittelussa todettiin kuitenkin tarpeelliseksi säilyttää kaikki projektiin edes välillisesti liittyvä versiohallinnassa. Myös useat lähteet, esimerkiksi työn ensisijaisena tietolähteenä toiminut Continuous Delivery (Humble & Farley 2011), painottivat yksiselitteisesti kaiken säilyttämistä versiohallinnan alaisuudessa. Termillä ”kaikki” tarkoitetaan kaikkea jolla on ollut osuutensa, tai jota tarvitaan tyhjästä lähtötilanteesta valmiin toimitetun tuotteen tuottamiseen. Lähdekoodien lisäksi määritykseen lukeutuvat siis esimerkiksi alustakonfiguraatiot, tietokantamuutokset ja mediatiedostot, asennus- tai käännösprosessin aikana käytettävät työkalut, testit, dokumentaatiot ja kirjastot. (Humble & Farley 2011, 33.)

Git tuki erityisen hyvin kaiken versiohallinnointia, sillä myös kokeilulähtöiset kohteet voitiin asettaa alusta lähtien versiohallintaan muutamalla komennolla ja jättää kohteen säilytys keskuspalvelimella myöhemmin päätettäväksi. Projektin osakokonaisuuksille luotiin erilliset Git-repositoryt. Tässä työssä osakokonaisuuksiin, ja siten erillisiin repositoryihin, lukeutuivat:

- palvelimien, käyttöjärjestelmien ja infrastruktuurin hallintaan liittyvät kohteet
- dokumentaatio ja suunnitteludokumentit
- sovelluksen luontiin ja ajoon vaadittavat kohteet, kuten sovelluksen lähdekoodit, artefaktien luontiin vaadittavat resurssit ja sovelluksen rakentamisen sekä ajon aikana käytetyt resurssit, kuten tietokantapohjat ja mediatiedostot.

Sovelluksen moduuliriippuvuudet päätettiin asentaa erillisen pakettienhallintatyökalun avulla. Tässä tapauksessa riippuvuudet tallennettiin versiohallintaan konfiguraatiotiedostona, jossa riippuvuudet määritettiin yksiselitteisesti siten, että ne voitiin asentaa toistettavasti ja automaattisesti. Sovelluksen ajobinäärin versiohallinnointi tapahtui repositoryyn tallennetun asennusskriptin avulla. Ajobinäärin asennusskripti haki sovelluksen riippuvuuksiin määritetyn version mukaiset lähdekoodit julkaisijan palvelimelta, suoritti lähdekoodien kääntämisen ja asetti käännöksen tuotoksena syntyneen ajobinäärin sovelluksen paikalliseen hakemistorakenteeseen. Moduulien tapaan myös ajobinäärin asennus oli siis toistettavissa oleva automaattinen prosessi. Sovelluksen yhteyteen paketoidut käynnistys- ja testiskriptit ajettiin tällä paikallisesti hallitulla ajobinäärillä.

Myös moduulien ja ajoympäristön tallentaminen sellaisenaan suoraan sovelluksen yhteyteen olisi ollut mahdollista, mutta tässä työssä päädyttiin käyttämään edellä kuvattua mallia. Riippuvuuksien suoraa paketointia saatetaan soveltaa tulevissa projekteissa, sillä tällöin vältyttäisiin riippuvuussuhteilta kolmannen osapuolen palveluihin. Kriittisemmissä sovelluksissa ongelmaan täytyisi kiinnittää erityistä huomiota, mikä tulikin työn aikana todistetuksi, sillä käytettyyn julkiseen pakettivarastoon ei muutamaan otteeseen saatu useaan tuntiin yhteyttä kolmannen osapuolen ongelmien vuoksi. Sovelluksen julkaisun

hetkellä tapahtuessaan tämä tilanne olisi voinut aiheuttaa merkittävää haittaa ja palvelukatkoksia. Huomion arvoista on myös se, että artefaktien luonti kohdejärjestelmässä parantaa alustayhteensopivuutta esikäännettyjen binääritiedostojen tallentamiseen verrattuna. Riippumattomuuden ja laajan alustayhteensopivuuden saavuttamisen vuoksi paras ratkaisu saattaisi olla artefaktien luonti kohdealustalla omassa hallinnassa olevia paikallisia paketti- ja lähdekoodivarastoja käyttäen.

Koska internetsovellusten kehitys ja julkaisu tapahtui itse hallinnoidulla palvelimilla, kuului projektin toteuttamiseen myös palvelinalustojen konfigurointi ja erilaisten käyttäjätilien hallinta. Kaikki palvelinten konfiguraatiot, tuotteen ajoa varten luodut käyttäjätunnukset sekä tunnusten kotihakemistot ympäristöasetuksineen päätettiin säilyttää alusta lähtien versiohallinnassa.

3 OHJELMISTOALUSTA

3.1 JavaScript ja Node.js

Tapahtumakalenterisovellus päätettiin toteuttaa suhteellisen tuoreen Node.js -alustan päälle. Node.js on vuonna 2009 alkunsa saanut palvelinpuolella ajettava avoimen lähdekoodin asynkroninen tapahtumapohjainen JavaScript-ympäristö, joka toimii Googlen kehittämän avoimen lähdekoodin V8 JavaScript-moottorin päällä.

Node.js -alustaa markkinoidaan tehokkaana ratkaisuna sovelluksille, joiden toiminta on vahvasti I/O-painotteista. Alustan toimintamallin erottaa tavanomaisista teknologioista sen asynkroninen luonne. Node.js -sovellus suoritetaan yhtenä prosessina, ja alusta ei sisällä tukea säikeistykselle. Toiminta perustuu asynkronisiin rajapintakutsuihin. Rajapinnan tarjoamat funktiot toimivat (muutamaa poikkeusta lukuun ottamatta) asynkronisesti siten, että funktion kutsu ei itsessään suorita mitään raskasta toimintaa itse, vaan välittää toimintapyynnön eteenpäin, jonka jälkeen funktiokutsusta palataan välittömästi. Sovelluksen suoritus ei jää odottamaan operaatioiden valmistumista. Funktiokutsuille annetaan kutsuhetkellä oma tapahtumankäsittelijä (callback), jota funktion suorittanut osapuoli kutsuu, kun operaatio on valmis. (Joyent, Inc, 2010.)

Teknologia sopii huonosti raskasta laskentaan sisältäviin sovelluksiin, sillä pitkän ajan viettäminen yhdessä ohjelman osassa pysäyttäisi suorituksen muissa osissa. Malli soveltuu kuitenkin hyvin esimerkiksi verkkoliikennettä käsitteleviin, suhteellisen vähän laskentaa sisältäviin sovelluksiin. Perinteisesti ongelmaa suuresta määrästä yhtäaikaisia pyyntöjä on ratkaistu moniajolla ja säikeistyksellä. Säikeistyksestä aiheutuu kuitenkin ylimääräistä painolastia (overhead), ja varmatoimisen monisäikeisen sovelluksen toteuttaminen on huomattavasti yksisäikeistä vaikeampaa, monimutkaisempaa ja virheille alttiimpaa. (Martin 2009, 178.)

Perintesesti JavaScript-ohjelmointikieltä on käytetty ainoastaan internetsovellusten asiakaspäässä, eli internetselaimessa, tuomaan staattisille internetsi-

vuille ohjelmoitavuutta, lisätoiminnallisuutta ja interaktiivisuutta. JavaScript-kielen käyttöön palvelinpuolella on olemassa myös lukuisia muita ratkaisuja, kuten esimerkiksi Mozilla Rhino -projekti, mutta Node.js on kerännyt pienessä ajassa suuren mediahuomion ja käyttäjäkunnan.

3.2 Eroavaisuudet PHP-alustaan

Tapahtumakalenteriprojekti päätettiin toteuttaa Node.js -alustalla osittain puhtaasta mielenkiinnosta tutkimusmielessä, sillä ennakkotietojen perusteella teknologia mahdollistaisi uudentyyppisten sovellusten rakentamisen verrattuna aikaisemmin käytettyyn PHP-kieleen. PHP-sovelluksen ajo tapahtuu palvelimella jatkuvassa ajossa olevan erillisen HTTP-palvelinsovelluksen, esim. Apache HTTP Server, Nginx tai Lighttpd, toimesta. Palvelinsovellus vastaanottaa sivuston käyttäjän lähettämän pyynnön. Palvelinsovellus voi vastata asiakkaan pyyntöön lähettämällä staattista sisältöä, kuten palvelimella sijaitsevia kuva- tai HTML-tiedostoja, tai ajaa PHP-sovelluksen, joka luo vastauksessa lähetettävän sisällön ohjelmallisesti suoritushetkellä.

Tyypillisesti PHP-sovelluksen suoritus päättyy vastauksen lähetykseen. Suoritusmalli toimii hyvin perinteisten internetsivustojen ja resurssikeskeisten REST-periaatteen mukaan toimivien CRUD-sovellusten tekoon, missä suoritus tapahtuu kysymys-vastaus -periaatteen mukaisesti käyttäjän siirtyessä sivulta toiselle. Näin toimivat HTTP-sovellukset eivät tyypillisesti säilytä käyttäjäkohtaista tilatietoa palvelimella sivulatausten välillä. Käyttäjäkohtainen tilatieto pyritään paketoimaan jokaiseen asiakasohjelman palvelimelle lähettämään palvelupyyntöön (request). REST-sovelluksessa jokainen palvelupyyntö on siis itsenäinen tapahtumansa. Pitkäikäinen tieto, eli sisältö josta sivusto koostuu, tallennetaan tavanomaisesti palvelimella sijaitsevaan tiedostoon tai tietokantaan. (Webber, Parastatidis & Robinson 2010.)

Node.js eroaa PHP-toteutuksesta siten, että suoritettava sovellus käynnistetään palvelimelle siirron jälkeen ajoon, ja sovelluksen on tarkoitus säilyä ajossa jatkuvasti. Node.js -sovellus ei siis vaadi erillistä HTTP-palvelinta sovelluksen suoritukseen, vaan sovellus sisältää myös verkkoliikennettä käsittelevän palvelintoiminnallisuuden asiakkaalta tulleiden pyyntöjen käsittelyyn. Jatkuva ajossa olo mahdollistaa interaktiivisten palvelimella käyttäjäkohtaista tilaa säi-

lyttävien sovellusten, kuten selaimessa toimivien moninpelien tai reaaliaikaisten keskustelusovellusten, helpon toteutuksen.

Tavanomaisen internetsovelluksen tapauksessa tilan säilyttämistä palvelimella pyritään kuitenkin välttämään, sillä tilaa säilyttävän sovelluksen testaus ja sovelluksen toimivuuden takaaminen erilaisien toimien jälkeen on vaikeampaa ja siten kalliimpaa. Tilattoman sovelluksen kuormitusta voidaan keventää käyttämällä erilaisia välimuistipalvelinratkaisuja (cache). (Webber, Parastatidis & Robinson 2010, 158.)

Tapahtumakalenterin julkisivun toteutus olisi onnistunut hyvin tutuilla PHP-ratkaisuilla. Koska kyseessä kuitenkin oli tutkimusluonteinen projekti, ja toisaalta kalenterin taustalle toimivaksi suunniteltuja tapahtumakerääjiä ajastetusti ajava monitorointi- ja hallintasovellus oli luontevasti toteutettavissa jatkuvasti ajossa olevaksi, päätettiin koko projekti toteuttaa Node.js -alustan päälle. Alustan käytöstä saatuja kokemuksia oli tarkoitus hyödyntää myöhemmissä välitöntä interaktiivisuutta vaativissa projekteissa, kuten selainpeleissä. Työn toteutuksessa selvisi että alusta sopii hyvin myös tilattomien sivustojen toteutukseen, pääasiassa tarjolla olevien kattavien ja hyvin suunniteltujen kirjastojen vuoksi.

3.3 Lukuisten HTTP-palvelimien ongelma

Koska jokainen HTTP-protokollaa käyttävä Node.js-sovellus sisältää oman HTTP-palvelintoiminnallisuutensa, jokainen sovellus käynnistyy kuuntelemaan omaa TCP-porttiaan. HTTP-asiakasohjelmat yhdistävät oletusarvoisesti palvelimen TCP-porttiin 80 (salatun yhteyden tapauksessa porttiin 443). Porttia kuunteleva sovellus varaa portin omaan käyttöönsä, eivätkä useat sovellukset voi kuunnella samaa porttia yhtäaikaisesti. Tämä muodostuu ongelmaksi kun samalla palvelimella ajetaan useita HTTP-sovelluksia. Ongelma ratkaistaan yleensä siten, että erilliset sovellukset asetetaan kuuntelemaan kukin omaa vapaata porttiaan, ja palvelimelle asennetaan ns. reverse proxy -sovellus, joka asetetaan kuuntelemaan yleistä porttia. Reverse proxy sisältää logiikan, jonka avulla saapuneet palvelupyynnöt voidaan delegoida esimerkiksi toiseen palvelimen TCP-porttiin, jota pyyntöä palveleva sovellus kuuntelee.

Reverse proxyna tässä työssä päädyttiin käyttämään Nginx-palvelinsovellusta. Nginx on Igor Sysoevin kirjoittama sovellus, joka toimii sekä HTTP-palvelimena että reverse proxyna. Saatavilla olisi ollut myös Node.js-alustalla kirjoitettuja ratkaisuja, mutta Nginx-palvelimeen päädyttiin ensisijaisesti sen maineen, keveyden ja suurilla sivustoilla käytännössä testatun toimivuuden vuoksi. (About Nginx n.d.)

Nginx konfiguroitiin Node.js-sovellusten näkökulmasta siten, että se ohjasi saapuvat palvelupyynnöt oikealle käsittelijälle pyynnön kohteena olevan verkkotunnuksen perusteella. Koska Nginx on itsessään tavanomainen HTTP-palvelin, jonka avulla voidaan tehokkaasti palvella staattisia tiedostoja tai jopa PHP-sovelluksia, on samalla palvelimella mahdollista pyörittää tarvittaessa myös perinteisiä sivustoja. Tapahtumakalenterin kehityksen ajaksi Nginx asetettiin näyttämään verkkotunnukseen saapuvalle kävijälle epätoiminnallista ilmoitussivua, josta kävi ilmi että verkkotunnuksen sivustoa oltiin vasta rakentamassa, ja varsinainen sisältö ilmestyisi myöhemmin.

3.4 Modulaarinen rakenne ja paikalliset riippuvuudet

Node.js-alustaan tutustumisen yhteydessä erittäin positiivisen yllätyksen toi järjestelmän modulaarinen luonne. Moduulin tehtävä oli ratkaista tai tarjota aputyökalu yksittäiseen keskitettyyn ongelmaan. Moduulit olivat luonteeltaan itsenäisiä ja yleiskäyttöisiä, ja niiden käyttö oli moduulin käyttäjän näkökulmasta yksisuuntaista, jolloin tarpeettomia kaksisuuntaisia riippuvuussuhteita ei päässyt syntymään. Moduuli saattoi käyttää apunaan toisia moduuleja, jolloin riippuvuussuhteet voitiin havainnollistaa puurakenteella, jossa yksittäisen moduulin riippuvuusvaatimukset näkyvät sen oksana (ks. listaus 1). Monimutkainenkin sovellus on toteutettavissa pienellä koodimäärällä omia ja kolmannen osapuolen toteuttamia avoimen lähdekoodin moduuleja yhdistelemällä.

LISTAUS 1. Sovelluksen käyttämät moduulit riippuvuuksineen puunäkymässä

```
$ npm ls
fuzz-pub@0.0.1 /home/zemm/fuzz/fuzz-pub
├── colors@0.5.1
├─┬ cradle@0.5.7
│ ├── vargs@0.1.0
│ └─┬ vows@0.5.11
│   └── eyes@0.1.6
├── ejs@0.4.3
```

```
├─┬ express@2.4.7
│ ├── connect@1.7.1
│ ├── mime@1.2.4
│ ├── mkdirp@0.0.7
│ └── qs@0.3.1
├── express-messages@0.0.2
├── migrate@0.0.4
├── nodemon@0.5.5
├── nodeunit@0.5.5
└── underscore@1.2.0
```

Aktiivinen kehitysyhteisö oli tuottanut muutamassa vuodessa vaikuttavan kokoelman itsenäisiä paketoituja avoimen lähdekoodin moduuleja. Moduulien hallintaan käytettiin npm-pakettienhallintatyökalua. Hallintatyökaluun tutustuminen oli eräs projektin positiivisimpia yllätyksiä.

Sovelluksen moduuliriippuvuudet määriteltiin projektin juuressa sijaitsevaan konfiguraatiotiedostoon, jota pakettienhallintatyökalu tulkitsi. Yksittäinen riippuvuusmääritys koostui vaaditun moduulin nimestä ja halutulla tarkkuudella määritetystä versionumerosta.

Moduulit käyttivät pääasiassa internetissä julkaistun Semantic Versioning (SemVer) ehdotuksen mukaista versionumerointia. Määrityksen mukaiseen versionumeroon sisältyi kolme pisteellä erotettua versio-osaa, esimerkiksi 2.3.12, jossa ensimmäinen numero määritti pää- (major), toinen ala- (minor) ja kolmas paikkausversion (patch). SemVer määritys asetti vaatimukset sovelluksen tai moduulin rajapintayhteensopivuuden säilyttämiseen aikaisempien versioiden suhteen. Pääversiotaan kasvattavan sovelluksen sallittiin rikkovan yhteensopivuudet aikaisempiin versioihin nähden, alaversiota kasvattava julkaisu saattoi toteuttaa lisäyksiä rajapintaansa, säilyttäen kuitenkin yhteensopivuuden taaksepäin, ja paikkausversionumeroa kasvattavan julkaisun ei odotettu tekevän muutoksia näkyvään rajapintaansa. (Preston-Werner, Semantic Versioning.)

Yhteisesti sovitun versiokäytänteen avulla sovelluksen riippuvuudet voitiin siis määrittää siten, että pakettienhallintatyökalu asensi käytössä oleviin moduuleihin julkaistut uudet tietoturvapäivitykset, mutta ei yhteensopivuutta rikkovia suurempia päivitysversioita. Yhteinen dokumentoitu versionumero koettiin turvalliseksi, ja sitä pyrittiin noudattamaan jatkossa myös omien sovellusten ja moduulien suhteen.

Riippuvuussuhteet määrittävään konfiguraatiotiedostoon sijoitettiin myös laajempi yleisluontoinen koneellisesti käsiteltävä tieto toteutetusta sovelluksesta. Jokaiselta julkaistulta moduulilta edellytettiin samaa konfiguraatiotiedostoa. Oman julkisen moduulin teko ja julkaisu onnistuisi helposti konfiguraatiotiedoston ja pakettienhallintatyökalun avulla. Myös oman sovelluksen paketointi olisi ollut mahdollista, sillä tuotettu sovellus on itsessään käytännössä korkeimman tason moduuli. Teknisesti sovellus erosi moduulista lähinnä siten, että sovellus on luotu ratkaisemaan niin kapean alan ongelma (esimerkiksi Jyväskylän Tapahtumakalenteri), ettei se olisi järkevästi uudelleenkäytettävissä, ja siten sen levitys järkevää.

Npm-pakettienhallintatyökalu nouti asennettavat moduulit oletusarvoisesti työkalua kehittävän tahon ylläpitämästä pakettivarastosta, jossa moduuleista säilytettiin myös vanhemmat versiot. Riippuvuuksiin määritettyjen versioiden asennus oli siis mahdollista, vaikka moduuleista oli jo ehditty julkaista tuoreempia versioita. Kolmannen osapuolen pakettivaraston käytöstä tunnistettiin riskeiksi pakettivaraston saatavuuden taattavuus, ja moduuleihin päivityksessä sisällytettävän haittakoodin mahdollisuus. Moduulit sijaitsivat CouchDB-dokumenttitietokannassa, ja pakettivarasto olisi ollut mahdollista kloonata paikalliseksi. Kalenterisovelluksen yhteydessä riskit arvioitiin kuitenkin suhteellisen vähäisiksi, ja oman paikallisen pakettivaraston ylläpidon hyötyjen ei todettu oikeuttavan oman pakettivaraston ylläpidosta aiheutuvaa lisätyötä, joten työssä käytettiin pakettienhallinnan kehittäjän julkista pakettivarastoa.

Node.js moduulit oli mahdollista asentaa niin haluttaessa sovelluskohtaisesti paikallisina. Mahdollisuus erosi aikaisemmin tutuksi tulleiden muiden alustojen, kuten PHP, Ruby tai Perl, tavasta, jossa paketteja hallittiin palvelimen laajuisesti. Node.js-alustaa käytettäessä saman palvelimen sisällä oli siis luontevasti mahdollista ajaa sovelluksia, jotka riippuivat samojen moduulien eri versioista. Koska sovelluksen JavaScript-koodin suorittava Node.js-alusta oli yksi suhteellisen pienikokoinen binääritiedostosto, oli koko sovelluksen ympäristö paketoitavissa paikallisesti itsenäiseksi paketikseen, joka ei ollut käytännössä riippuvainen käyttöjärjestelmään asennetuista paketeista.

Mahdollisuus palvelinympäristöön asennetuista paketeista riippumattomien sovellusten toteutukseen vähensi palvelimen ylläpidosta aiheutuvaa taakkaa

totutusta, sillä ylläpitotehtäviksi jäi sovelluksen näkökulmasta vain käyttäjätilien hallinta, joilla sovelluksia suoritettiin. Toisaalta päivitysten hallinnointi ympäristössä, jossa paketit olivat sovelluskohtaisia, todettiin tulevaisuudessa kasvavaksi ongelmaksi, joka vaatii päivitysstrategian tarkempaa suunnittelua viimeistään siinä vaiheessa, kun sovellusten määrä on kasvanut suureksi.

Tapahtumakalenteriprojekti päätettiin toteuttaa mahdollisimman pitkälle paikallisella riippuvuuksienhallinnalla. Sovelluksen käyttämät moduulit asennettiin paikallisesti, ja sovelluksen ajoon käytetyn Node.js alustan binääritiedoston versio 0.4.12 asennettiin sovelluskohtaisen käyttäjätunnuksen kotihakemistoon. Käyttäjätunnuksen ympäristömuuttujassa määritettyyn ajopolkuun (PATH) asetettiin ensisijaiseksi etsintäpoluksi kotihakemistoon määritetty binhakemisto, jolloin sovellus ajettiin aina paikallisella Node.js versiolla. Ajobinäärin asennukseen ja hallintaan käytettiin n-apuohjelmaa, joka automatisoi ajobinäärin tietyn version lähdekoodeista kääntämisen ja asentamisen käyttäjätunnuksen paikalliseen ajohakemistoon (github.com/visionmedia/n 2011).

3.5 Express-ohjelmistokehys

Tapahtumakalenterin julkinen HTTP-rajapinta ja käyttöliittymä toteutettiin moduulina asennettavalla Express-ohjelmistokehyksen ja sen liitännäisten avulla. Ohjelmistokehyksellä tarkoitetaan runkoa, jonka päälle oma sovellus toteutetaan. Rungon toimenkuvaan kuuluu tarjota ohjelmistoalustan rajapinnan päälle valmiiksi toteutettuja korkeamman tason toimintoja ja apuvälineitä eri projekteissa toistuviin yleisiin ongelmiin. Internetsovelluksen tapauksessa tällaisia ongelmia ovat esimerkiksi HTTP-tietueiden käsittely, näkymälogiikan erotus ja sovelluksen sisäinen reititys.

Express oli lokakuussa suosituin Github-sivustolla kehitettävä Node.js-moduuli projektia seuraavien kehittäjien määrällä mitattuna (Nipster 2011). Yhteisön käyttäjäkokemuksia, blogikirjoituksia ja IRC-keskusteluja seuraamalla saatiin mielikuva siitä, että Express oli suosituin Node.js-alustan web-ohjelmistokehys myös käyttäjämäärällä ja aktiivisuudella mitattuna.

Express oli rakennettu Connect-ohjelmistokehyksen päälle. Connect oli Expressin kehittäjän tuottama palvelupyyntöjen käsittelyyn tarkoitettu matalan ta-

son ohjelmistokirjasto. Connect mahdollisti sille luotujen irtonaisten, yleiskäyttöisten ja toisistaan riippumattomien väliohjelmistojen (middleware) liittämisen toisiinsa. Väliohjelmistoista luotiin palvelupyyntöjä käsittelevä ketju, jossa raaka palvelupyyntö lähetettiin ketjun läpi, ja jokainen ketjuun liitetty väliohjelmisto saattoi tehdä vuorollaan oman ongelma-alueensa mukaisen toiminnon tai muokkauksen palvelupyynnölle. Väliohjelmistojen ketjutus mahdollisti koodin uudelleenkäytettävyyden, ja monimutkaistenkin toimintojen toteuttamisen pienellä tapauskohtaisella koodimäärällä. Väliohjelmistoja löytyi valmiina kymmenittäin sekä Connectin kehittäjän että ulkopuolisten tahojen tuottamana, ja aktiivisen yhteisön myötä määrä kasvoi tasaisesti. Yksittäinen väliohjelmistom tehtävä oli esimerkiksi POST- tai COOKIE-tietueen parsimisen raa'asta HTTP-palvelupyynnöstä ja liittää se pyyntöobjektiin ketjun myöhempien osien tai lopullisen sovelluskoodin käytettäväksi.

Expressin päälle rakennettu sovellus saattoi luoda tarpeen mukaisen Connect-väliohjelmistojen ketjun, jolla raa'asta pyynnöstä muodostettiin siisti korkeamman tason pyyntöobjekti. Express tarjosi palvelupyyntöjen käsittelyn päälle joukon HTTP-protokollalle keskeisiä yleisesti tarvittuja toimintoja. Expressin tarjoamiin HTTP-keskeisiin palveluihin ja työkaluihin kuuluivat:

- sovelluksen reitityksen hallinta, jonka avulla saapuva palvelupyyntö voitiin ohjata oikealle käsitttelijälle
- korkean tason rajapinta HTTP-protokollan ominaisuuksille kuten uudelleenohjauksille, tilakoodeille ja sisältötyypin kättelylle
- näkymäkirjastojen hallinta, joka mahdollisti lukuisten ulkopuolisten näkymäkirjastojen käytön vastauksen muodostamisessa
- ajoympäristöstä tietoinen ehdollinen konfiguraationhallinta.

(Express 2011.)

Ohjelmistokehyksen tason konfiguraatiohallinnassa sijaitsevia alustakohtaisesti vaihtuvia asetuksia olivat mm. tietokantayhteyden määritys ja virheilmoitusten käsittely. Tuotantopalvelimella virheet haluttiin kirjata hiljaisesti lokitiedostoihin ja näyttää sivuston käyttäjälle muotoiltu virheilmoitus joka ei sisältä-

nyt tarpeettomia teknisiä yksityiskohtia. Kehityspalvelimella virheet sen sijaan haluttiin välittömästi näkyviin asiakasohjelmaan, jotta ongelmatapausten toteaminen olisi mahdollisimman nopeaa ja selkeää. Tietokantayhteyksien yksityiskohdat, kuten tietokannan osoite ja käyttäjätunnukset, saattoivat sisältää alustakohtaisia eroja, joten myös niiden asetukseen vaadittiin ympäristöstä tietoinen sovellustason konfiguraatiohallinta. Express tarjosi ongelmaan yksinkertaisen ja toimivan ratkaisun, jossa käytössä oleva ympäristö tunnistettiin NODE_ENV -ympäristömuuttujasta. Muuttuja oli helppo asettaa eri ympäristöille joko sovelluskäyttäjän ympäristömuuttujiin tai vaihtoehtoisesti sovelluksen asennuksessa luotavaan ympäristökohtaiseen käynnistystiedostoon, jota kutsumalla sovelluksen automaattinen käynnistys suoritettiin.

PHP-ympäristön suuriin ohjelmistokehyksiin verrattuna Express oli varsin pieni ja ohut, eikä se tarkoituksellisesti asettanut juurikaan rajoitteita tai tarjonnut ennalta määritettyä muotoa toteutettavalle sovellukselle. Työn aikana seurattujen ohjelmistokehyksen kehittäjien välillä käytyjen useiden IRC-keskustelujen mukaan Express-sovelluksen kehittäjälle tarjottu vapaus on ollut tietoinen valinta ja lähtökohta ohjelmistokehityksen suunnittelussa.

Aikaisemmat sovelluskehityskokemukset ovat osoittaneet että liian suuren vapauden hintana voi usein olla suuriakin ongelmia loppukehittäjän tekemien väärien rakenteellisten valintojen myötä. Jyrkemmin rajattujen ohjelmistokehysten rajoitukset pakottavat tuotetulle sovellukselle yhtenäisen rakenteen. Laadukkaan ohjelmistokehyksen pakottama muotti saattaa usein olla parempi kuin mihin loppukehittäjä ilman rajoituksia päätyisi. Toimivan Express-sovelluksen rakenteen löytäminen saattaakin vaatia lukuisten testisovellusten luontia tai sovelluksen rakenteen uudelleenjärjestelyä. Näin kävi myös tapahtumakalenterin suhteen, sillä sen rakennetta muutettiin useita kertoja kehityksen aikana, eikä siihen voitu edelleenkään olla täysin tyytyväisiä.

Connectin tapaan Expressin suuri vahvuus olivat irralliset moduulit, joita yhdistelemällä ja ketjuttamalla monimutkainenkin sovellus oli toteutettavissa suhteellisen pienellä rivimäärällä. Moduuleita löytyi valmiina lähes kaikkiin ongelma-alueisiin joihin toteutuksessa törmättiin. Esimerkiksi käyttäjäautentikaatioon oli tarjolla useita testattuja ja laajassa käytössä olevia moduuleja, jotka tarjosivat korkean tason rajapinnan lukuisille yleisimmille autentikaatiometo-

deille, jolloin sovelluskehittäjän tehtäväksi jäi yhdistää autentikaatiomoduuli, roolienhallinta ja tiedon tallennukseen käytetty datavarasto toisiinsa toivotun lopputuloksen mukaiseksi kokonaisuudeksi.

4 DOKUMENTTITIETOKANTA

4.1 Tarve skeemavapaalle tietovarastolle

Työn osapuolilla oli ennen työn toteutusta kokemusta ainoastaan perinteisten relaatiotietokantojen käytöstä. Relaatiotietokantojen laajasta kirjosta käytössä ovat olleet MySQL ja PostgreSQL. Ensimmäiset relaatiomallin tietokannat kehitettiin jo 70-luvulla, ja malli on ollut siten äärimmäisen testattu ja hallitseva ohjelmistoprojektien tietovarastona useita vuosikymmeniä. Relaatiotietokanta tallentaa tiedon ennalta määritetyn skeeman mukaiseen taulurakenteeseen, joissa yksittäinen taulurivi tai relaatioilla toisiinsa kytkettyjen rivien joukko kuvaa tallennettua tietoa.

Viime aikoina relaatiotietokantojen rinnalla on ollut enenevissä määrin esillä erilaisia dokumentti- ja avain-arvo -tietokantoja. Eräs näiden yhteinen piirre on skeemavapaus, eli tietokantaan tallennettavan tiedon muotoa ei pakoteta suunnitteluvaiheessa luotuun muottiin. Esimerkkejä kirjoitushetkellä suosituista dokumenttitietokannoista ovat Apache CouchDB ja 10gen MongoDB. Avain-arvo -tietokannoista suosittuja vaikuttavat olevan Redis ja Riak. Yhteistä eri tyyppisille uuden sukupolven tietovarastoille on relaatioiden ja etukäteen määritetyn tietorakenteen puuttuminen. Tietokannat eroavat toisistaan tekniseltä toteutukseltaan ja käyttölogiikaltaan huomattavasti enemmän kuin SQL-standardia noudattamaan pyrkivät relaatiotietokannat.

Suunnitteluvaiheessa tapahtumakalenterin tapahtumatiedon yhteyteen sisällytettävän erilaisen tiedon määrän todettiin kasvavan vauhdikkaasti lukuisten tulevaisuuteen suunniteltujen lisätoiminnallisuuksien myötä. Eri tapahtumien kesken todettiin myös huomattavaa vaihtelua tapahtumaan liitetyn tiedon määrässä ja laadussa. Alkuvaiheessa säilytettävän tiedon muoto ja laatu oli myöskin hieman epävarmaa. Yksittäinen tapahtumatiedon todettiin konseptitasolla vastaavan erittäin hyvin dokumenttitietokantoihin tallennettavaa JSON-dokumenttia. Tapahtumatietojen ei myöskään nähty sisältävän tapahtumapaikkaa ja käyttäjätietoja lukuun ottamatta vahvoja relaatioita, jotka olisivat pakot-

taneet perinteisen relaatiotietokannan käyttöön. Sovelluksen tapahtumatietovarastona päädyttiin käyttämään dokumenttitietokantaa.

Tapahtumaa kuvaavaan dokumenttiin sisätyi numero- ja tekstimuotoista tietoa, kuten tapahtuman esittelykuvaus, tiedot esiintyjistä, tapahtuma-ajasta ja -paikasta sekä lipuista. (Katso listaus 2, joka havainnollistaa projektin aikana CouchDB-tietokantaan tallennetun yksittäisen tapahtumadokumentin.) Lisäksi tapahtumaan todettiin tarpeelliseksi liittää loppukäyttäjälle näkymätöntä metatietoa, jonka perusteella tiedonhakijat erottavat järjestelmään ennestään lisätyt tapahtumat uusista. Myöhemmissä versioissa nähtiin mahdolliseksi myös erilaisten mediatiedostojen, kuten kuvien ja videoiden, sisällyttäminen tapahtumadokumenttiin.

LISTAUS 2: Esimerkki tapahtumaa kuvaavasta dokumentista

```
{ ”_id”: ”f31bcdf52b79c97193a9ab2b113c4c4ed500f761”,
  ”_rev”: ”1-2442b7b53b8a1b1e9ab7346888b319d2”,
  ”title”: ”Vegfest tukikeikka”,
  ”description: ”Pitsaa ja sipsejä tarjolla nopeimmille ja keikkojen
jälkeen (ja ennenkin) Djt soittaa jorausmusaa!”,
  "artist": ["Delta Force 2", "Presley Bastards", "Vapaa Maa"],
  "location": "Lutakko",
  ”start_time”: ”2011-11-26T21:00:00”,
  ”tickets”: ”8€ / 6€”,
  ”url”: ”http://www.jelmu.net/keikat/102”
}
```

4.2 Dokumenttitietokannan valinta

Työn yhteydessä käytettäväksi tutkittiin kahta dokumenttitietokantaratkaisua: Apache CouchDB ja MongoDB. Molemmat säilyttävät tallennetut dokumentit JSON-objekteina, ja tarjoavat mahdollisuuden luoda map/reduce -funktioita tallennettujen tietojen hakuun, tiivistämiseen ja muotoiluun. MongoDB tarjoaa myös etäisesti relaatiotietokantojen SQL-kieltä muistuttavan tavan ajonaikaisten kyselyiden käyttämiseen tiedonhaussa, kun taas CouchDB käyttää tiedonhakuun ainoastaan etukäteen luotuja map/reduce-funktioita.

Kumpikin tietokantaratkaisu tarjoaa oman mallinsa tietokannan replikointiin, synkronisointiin ja skaalaukseen. Tapahtumakalenterin tapauksessa käyttäjäkunta arvioitiin kuitenkin niin pieneksi, ettei skaalautuvuuteen nähty tarvetta kiinnittää huomiota tietokannan valinnassa.

Ensimmäinen valintaan vaikuttava ero löytyi tietokantojen käyttötavoista. MongoDB käyttää perinteisten relaatiotietokantojen tapaan alustakohtaista binääriajuria. Ratkaisua perustellaan tehokkuudella sekä nopeudella (Merriman 2011). CouchDB:n käyttö tapahtuu REST-mallia mukailevan HTTP-protokollan välityksellä tarjottavan rajapinnan kautta, jolloin tietokantaa käyttävältä alustalta ei vaadita erillistä ajuria, vaan käyttäjä voi olla mikä tahansa alusta, joka tukee HTTP-pyyntöjen lähettämistä ja vastaanottamista.

Käyttöliittymän lisäksi tietokantaratkaisut erosivat toisistaan painopisteissä, joilla ne markkinoivat itseään. MongoDB keskittyy ensisijaisesti nopeuteen ja skaalautuvuuteen. Nopeus saavutetaan osittain luotettavuuden kustannuksella, sillä tiedon tallennus tapahtuu vanhan tiedon päälle, ja tietoa saatetaan säilyttää jonkin aikaa muistissa ennen sen kirjoittamista pysyvään tietovarastoon levylle. (Merriman 2011.)

CouchDB markkinoi itseään ensisijaisesti luotettavuuden näkökulmasta. Tietokanta toimii ACID-periaatteen mukaan, ja lupaa järjestelmän olevan jokaisella hetkellä eheässä tilassa. Periaatteen mukaisesti transaktiot suoritetaan joko onnistuneesti kokonaisina ja pysyvinä, tai ei lainkaan. CouchDB ei tallenna uutta tietoa vanhan päälle kuten MongoDB. Dokumentaatio kuvaa CouchDB:n suunnittelua ”crash-only” -termillä, jolla tarkoitetaan sitä, että koska järjestelmä takaa olevansa jokaisella hetkellä eheässä tilassa, erillistä alasajoprosessia ei tarvita, vaan tietokantasovellus voidaan sulkea satunnaisella hetkellä.

Skaalautuvuusongelmaan CouchDB tarjoaa ratkaisuksi replikointia. Tietokanta voidaan monistaa kokonaisuudessaan, ja jokainen kopio on toisiin kopioihin nähden tasavertainen itsenäinen instanssinsa. Eri kopioihin suoritetut muutokset voidaan myöhemmin synkronisoida kopioiden kesken kertaluontoisena tai jatkuvana prosessina. Tietokanta tunnistaa ristiriitatilanteet, ja jättää niiden ratkaisemisen käyttäjätoteutuksen ratkaistavaksi. Eheysperiaatteen mukaisesti ratkaisemattomat ristiriitatilanteet estävät synkronisaatioprosessin onnistumisen. (CouchDB Technical Overview 2011.)

Projektissa päädyttiin tutustumaan ja käyttämään CouchDB-tietokantaa. Valintaan vaikuttivat ensisijaisesti tietokannan luotettavuuteen painottuvat ratkaisut, ja toisaalta HTTP-rajapinnan tarjoamat lukuisat mahdollisuudet monissa eri

teknologioita käyttävissä projekteissa. Tietokannan nähtiin soveltuvan tapahtumakalenterin lisäksi erinomaisesti erilaisiin tulevaisuudessa toteutettaviin internetsovelluksiin perinteisistä kotisivuista blogeihin ja tietopankkeihin. Dokumenttitietokantaa voidaan myös käyttää relaatiotietokannan rinnalla, eivätkä teknologiat ole välttämättä toisiaan poissulkevia.

HTTP-rajapinta antaa vapauden sovelluksen toteutukseen käytettävän teknologian tapauskohtaiselle valinnalle ja tietokannan sijainnille. HTTP-rajapinta toimii sovelluksen näkökulmasta samoin, sijaitsipa tietokanta sitten sovelluksen kanssa samalla palvelimella, tai täysin toisessa verkossa ja maantieteellisessä sijainnissa. REST-periaatteen mukaisesti toimivan HTTP-rajapinnan ja asiakassovelluksen välissä on myös mahdollista käyttää lukuoperaatioiden nopeutukseen erilaisia välimuistiratkaisuja, mutta tämä lisäisi järjestelmän komponenttien ja monimutkaisuuden määrää, mistä johtuen ratkaisua tulee harkita vain jo sille ilmenee todellista tarvetta.

CouchDB-tietokantamoottorista on olemassa myös mobiiliversioita esimerkiksi Android-älypuhelinalustalle. Mobiilialustalla toimiva tietokantamoottori ja replikointiominaisuus mahdollistavat esimerkiksi internetyhteydestä riippumattoman mobiilisovelluksen toteutuksen siten, että sovelluksen sisältämä data synkronisoidaan silloin, kun internetyhteys on saatavilla.

5 LAADUNVARMISTUS

5.1 Testauskäytäntö

Projektin alusta alkaen oli selvää että järjestelmästä tulee löytyä automatisoidut testit vähintäänkin kriittisille osille. Aikaisempien toteutettujen järjestelmien testaus on tapahtunut pääasiassa kehityksen yhteydessä kehittäjän toimesta manuaalisesti kokeilulähtöisellä menetelmällä. Toiminnot on onnistuttu toteuttamaan toimivina, mutta monimuotoisempien projektien yhteydessä toistuvaksi ongelmaksi ovat muodostuneet uusien muutoksien aiheuttamat rikkinäisyydet aikaisemmissa toiminnoissa, ja epävarmuus projektin terveydentilasta kokonaisuudessaan.

Automatisoitu testaus on suuressa roolissa ohjelmistotuotannossa. Useat tahot ovat jo pitkään käyttäneet onnistuneesti useita läheisesti toisiinsa liittyviä testauslähtöisiä käytäntöjä, kuten TDD, ATTD ja BDD. Kattavan testimäärän ja testauslähtöisen kehityksen on todettu parantavan sovelluksen laatua useilla mittareilla (Siniaalto 2006).

Projektin kehityksen aikana luoduille ominaisuuksille kirjoitettiin välittömästi ominaisuuden luonnin yhteydessä ohjelmallisesti ajettavia testejä. Toteutuksessa ei noudatettu orjallisesti testauslähtöistä menetelmää, vaan testejä kirjoitettiin tapauskohtaisen harkinnan perusteella sekä ennen ominaisuuden toteutusta, että sen jälkeen. Testit suunnattiin todennäköiseksi määritettyihin ongelmakohtiin. Testeillä varmistuttiin siitä, että uusi ominaisuus toimii tarkoitetulla tavalla, ja toisaalta että lisätty toiminnallisuus ei aiheuttanut ongelmia aikaisemmin toteutetuissa toiminnoissa.

Projekti sisälsi kolmeen ryhmään jaoteltuja testejä, jotka kaikki ajettiin erillisellä testisuorittajalla. Suorittaja oli pakettienhallinnan avulla asennettava apuohjelma, jolle määritettiin hakemisto jossa suoritettavat testit sijaitsivat. Suorittaja ajoi testit, ja tuotti ajon jälkeen selkokielisesti luettavan tai koneellisesti käsiteltävän yhteenvetoraportin testien onnistumisesta. Testien toteutus oli riippuvainen käytössä olleesta testikirjastosta, sillä jokainen testisuorittaja odotti oman

määrityksensä mukaisesti kirjoitettuja testejä, ja kukin kirjasto tarjosi omat ratkaisunsa esimerkiksi asynkronisen ohjelmakoodin testauksen ongelmiin.

Asynkronisen ohjelman testauksessa oli useita sudenkuoppia, sillä testin kohteena oleva asynkronisen ohjelmakutsun lopputulos saattoi olla useiden viiveellä suoritettavien järjestelmäkutsujen päässä. Koska pääohjelman suoritus jatkui välittömästi kutsun jälkeen, saattoi testifunktiokyseiset tilanteet vaativat testikirjaston tarjoamaa erillistä testin valmistumisen ilmaisevaa takaisinkutsua.

Testit ajettiin kehittäjän toimesta ominaisuuden toteuttamisen yhteydessä paikallisessa kehitysympäristössä ennen ominaisuuden synkronisointia keskuspalvelimen versiohallintaan.

Testien ajoon käytettiin aluksi pakettienhallinnan avulla asennettua Expresso-testisuorittajaa, mutta työkalu todettiin nopeasti sekä tarpeettoman raskaaksi pieneen käyttöön, että huonosti soveltuvaksi asynkronisen ohjelmakoodin testaukseen. Lukuisista npm-pakettienhallinnan kautta saatavilla olevista testisuorittajista päädyttiin projektissa käyttämään suosittua nodeunit-testisuorittajaa.

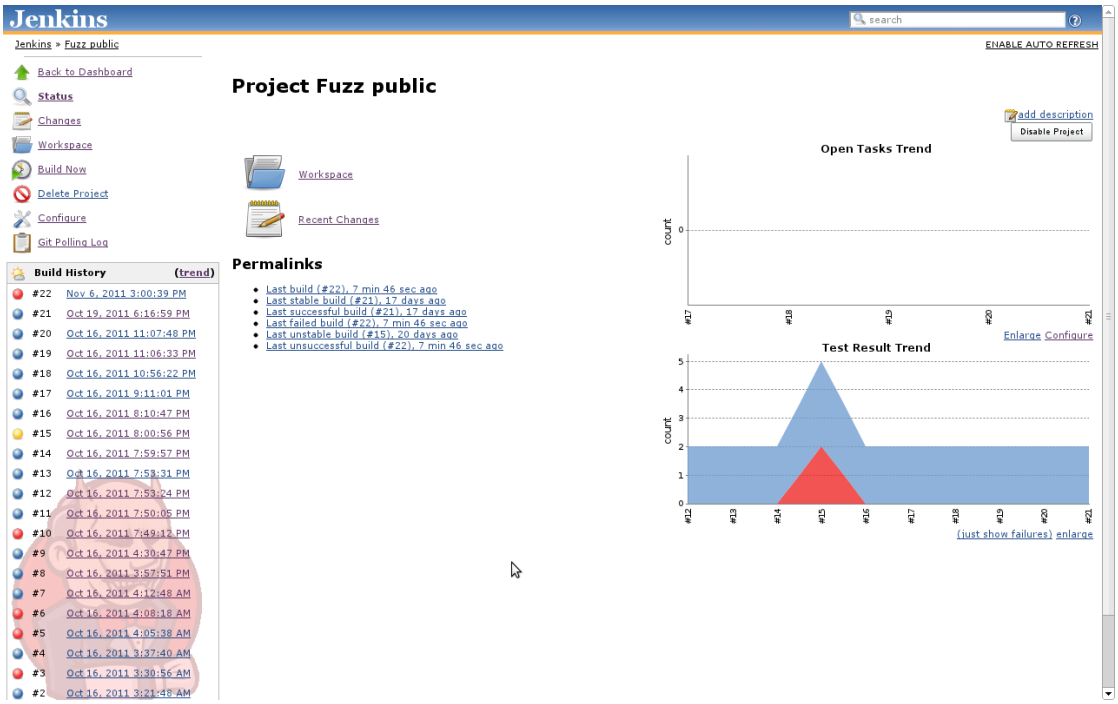
5.2 Välitön palaute ja Jenkins

Projektin laatua tarkkailtiin alusta lähtien jatkuvan integraation (continous integration) periaatetta noudattaen. Keskuspalvelimelle, jossa keskitetyt Git-repository sijaitsivat, asennettiin tausta-ajoon lukuisista saatavilla olevista avoimen lähdekoodin jatkuvan integraation palvelimista Jenkins CI, josta työn toteuttajalla oli ennestään pieni määrä kokemusta. Jenkins on aktiivisesti kehitetty tunnetun Hudson-integraatiopalvelimen sisarversio. Hudsonilla ja Jenkinsillä on suuri käyttäjäjoukko, ja sen vahvuuksiin kuuluu erittäin laaja liitännäisvalikoima, ja helppo konfiguroitavuus sisäänrakennetun selaimella toimivan graafisen hallintakäyttöliittymän avulla.

Jenkisiin luotiin tapahtumakalenterille ja sen taustalla toimivalle tapahtumakerääjälle omat projektit, jotka asetettiin seuraamaan kohteensa Git-repositoryn muutoksia. Aina kun kehittäjä päivitti paikalliset muutoksensa keskuspalvelimen kehitys- tai julkaisuhaaraan, Jenkins nouti versiohallinnasta tuoreen ver-

sion projektista omaan työympäristöönsä. Koska sovelluksen alustariippuvuudet olivat vähäisiä, ja ajoympäristö oli paketoitu sovelluksen mukaan, ja toisaalta koska keskuspalvelimella ajettiin samaa versiota Ubuntu Linux -käyttöjärjestelmästä kuin kohdepalvelimillakin, onnistui sovelluksen testien ajo Jenkinsin työympäristössä ilman erillistä konfigurointia.

Uuden ohjelmaversion noudon jälkeen Jenkins alusti projektin samoilla komennoilla, joilla sovellus alustettiin kaikkiin ympäristöihin. Alustukseen kuului ympäristömuuttujien asetus, moduulipäivitykset pakettienhallintatyökalun avulla ja migraatioskriptien ajo. Alustuksen jälkeen Jenkins ajoi projektin testit, ja tallensi testiajon tulokset. Testien lisäksi koodipohjalle ajettiin myös joitakin yleisiä koodin laatua mittaavia liitännäisiä, kuten kopioidun ja toistuvan koodin määrää sovelluksen laajuudella. Testien ja liitännäisten tulokset Jenkins esitti taulukkojen ja graafien muodossa, joista projektin laatu oli todettavissa erilaisilla kriteereillä mitattuna.



KUVIO 1: Jenkinsin projektisivu

Projektin ajantasainen terveys oli jatkuvasti nähtävillä Jenkinsin projektisivulla, ja mikäli jokin projektin testeistä ei mennyt läpi, lähetti Jenkins välittömästi sähköpostilla tiedotteen henkilölle jonka toimista ongelmat aiheutuivat. Tämän jälkeen kyseinen henkilö oli velvollinen korjaamaan välittömästi aiheuttamansa ongelmat.

6 PALVELIMIEN JA YMPÄRISTÖJEN KARTOITUS

6.1 Ongelmat entisessä ympäristönhallinnassa

Aikaisemmissa projekteissa palvelinympäristöt oli asennettu käsin. Julkiset palvelimet olivat kolmannelta osapuolelta tilattuja virtuaalipalvelimia Ubuntu Linux -käyttöjärjestelmällä. Virtuaalipalvelimet toimitettiin esiasennettuna puhtaalla minimalistisella konfiguraatiolla, jonka päälle asennettiin tarpeen mukaiset ohjelmistot, PHP-sovellusten tapauksessa Apache HTTP-palvelin, MySQL-relaatiotietokanta ja sovellusten tarvitsemat PHP-moduulit. Ohjelmistot asennettiin käyttöjärjestelmän natiivilla pakettienhallintatyökalulla.

Ohjelmistopakettien asennusten jälkeen asennettujen ohjelmistojen konfiguraatiot tehtiin suoraan palvelimelle tarpeesta ja palvelimen roolista riippuen. Konfiguraatioihin kuului esimerkiksi virtuaalisten sivustojen määritys Apachen asetustiedostoihin, joilla määrätty internetosoite ohjattiin osoitteessa näytettävän sivuston tuottavan PHP-sovelluksen hakemistoon. Käyttöjärjestelmään ja MySQL-tietokantaan luotiin sivustokohtaiset tunnukset.

Manuaaliseen konfiguraatioon sisältyi lukuisia ongelmia. Palvelimen konfiguraatioprosessi vei usein paljon aikaa, ja mikäli saman roolin palvelimia on useita, niiden konfigurointi toisiaan vastaavaksi manuaaliprosessilla oli hankalaa ja virhealtista. Prosessi saatettiin dokumentoida, mutta tämä jätti ongelmiksi inhimillisen virheen mahdollisuudet dokumentaation tulkinnassa tai toteutuksessa, puutteellisen tai vanhentuneen dokumentaation ja prosessin kuluttaman ajan. Palvelimen projisointi ei siis ollut luotettavasti ja helposti toistettavissa oleva prosessi. Ongelma tulee esille viimeistään kun ympäristö pitäisi monistaa esimerkiksi testi- tai kehitysympäristöksi, tai uuden palveluntarjoajan virtuaalipalvelimelle. Ohjelmistokehityksen tulisi tapahtua mahdollisimman paljon tuotantoympäristöä vastaavassa kehitysympäristössä (Humble & Farley 2011).

Tapahtumakalenteriprojektissa näihin ongelmiin päätettiin etsiä ratkaisuja, sillä oli selvää että parempia tapoja on olemassa, ja manuaalinen prosessi toimii

korkeintaan yhdellä ympäristöllä. Automaation tarve tulee tulevaisuudessa vain lisääntymään erilaisten pilvipalveluiden myötä, jolloin saatetaan esimerkiksi rasituksen ja käyttäjäkunnan kasvaessa hankkia pilvipalvelusta uusia virtuaalipalvelimia tai palveluita nopealla aikataululla.

6.2 Ympäristöt ja niiden merkitys

Tapahtumakalenteriprojektin ja tulevien muiden Node.js-projektien julkaisua varten hankittiin Ubuntu Linux -virtuaalipalvelimella. Palvelinta kutsuttiin tuotantopalvelimeksi, sillä sen tehtävä oli tarjota projektin uusinta vakaata versiota sivuston loppukäyttäjille HTTP- tai HTTPS-protokollan yli. Projektin tuotantopalvelin oli kolmannelta osapuolelta vuokrattu virtuaalipalvelin, jota hallinnoitiin itse.

Tuotantopalvelimen lisäksi kehitystyöhön kuului kehityspalvelin, jossa sivustoa ajettiin sen kehityksen aikana, ja jossa sovellukseen tehdyt muutokset voitiin todeta välittömästi. Kehityspalvelimena käytettiin paikalliselle kehityskoneelle asennettua virtuaalikonetta, jonka käyttöjärjestelmänä oli tuotantopalvelinta vastaava versio Ubuntu Linux -jakelusta. Osan ajasta kehitysalustana toimi suoraan paikallisen kehityskoneen oma käyttöjärjestelmä, joka sekin oli versioltaan tuotantopalvelimen kanssa yhtenevä Ubuntu Linux. Paikallisille virtuaalikoneille käytettiin Oraclen Virtualbox -virtuaalikonealustaa.

Kehitysalustat olivat ainoastaan kehittäjien henkilökohtaisessa käytössä, ja tuotantopalvelimella sijaitsi aina sovelluksen vakaa julkaisuversio. Näiden lisäksi tunnistettiin vielä satunnainen tarve kolmannelle alustatyypille, staging-palvelimelle. Staging-palvelimelle voitiin antaa julkinen osoite rajoitetulla pääsyllä. Staging-roolin palvelin mahdollisti epävakaassa vaiheessa olevan sovellusversion kokeilun laajemmalla mittakaavalla paikalliseen kehitysalustaan verrattuna. Tarvittaessa osa sivuston käyttäjistä voitaisiin ohjata läpinäkyvästi staging-palvelimelle, jolloin ohjelmaversiota voitaisiin testata oikeassa käyttäjäympäristössä rajatulla käyttäjäjoukolla.

Kehitys- ja tuotantoympäristön lisäksi projektin aikana käytettiin rajattua sisäistä keskuspalvelinta, joka toimi Git-repositoryjen yhteisenä synkronisointipisteenä. Keskuspalvelin sisälsi myös projektin dokumentaation, työkalut ja me-

diatiedostot. Keskuspalvelimella ajettiin jatkuvan integraation palvelinsovellusta, sekä satunnaisesti myös staging-roolin virtuaalipalvelinta.

7 YMPÄRISTÖJEN HALLINTA

7.1 Infrastruktuurin hallinta ja Puppet

Koska projektissa oli mukana useampi kuin yksi samankaltainen palvelinympäristö, lisättiin työn tavoitteisiin tutkimus erilaisista metodeista palvelimien projisoinnin ja konfiguraatiohallinnan automatisointiin. Lyhyen tutkimustyön tuloksena löydettiin kaksi varteenotettavaa tuotetta, Opscode Chef ja Puppet Labs Puppet, joilla molemmilla oli historiaa suuren luokan tuotantokäytössä, suuri yhteisö ja aktiivinen kehitystyö. (Opscode Customers 2011; Organizations using Puppet 2011.)

Kahdesta vaihtoehdosta projektille valittiin Puppet. Valinta suoritettiin pääasiassa käyttäjäkokemuksia lukemalla ja käyttöastetta arvioimalla. Teknisessä mielessä kumpikin vaihtoehdoista vaikutti toimivalta ratkaisulta infrastruktuurin automaattiseen hallintaan. Valinnassa ei suoritettu käytännön testausta, vaan Puppet asetettiin tositoimiin heti valintansa jälkeen.

Palvelimien hallinta Puppet-työkalulla alkoi tarvekartoituksella, jossa tutkittiin projektissa tarvittavat palvelinroolit. Koska tapahtumakalenteriprojekti oli suuruusluokaltaan varsin pieni, palvelimen kuormitus arvioitiin toistaiseksi pieneksi, ja toisaalta koska paikallisesti asennettavilla riippuvuuksilla toteutettu Node.js-sovellus todettiin varsin itsenäiseksi ja palvelimen palveluilta erittäin vähän vaativaksi kokonaisuudeksi, tarvittiin projektissa vain yksi yleisluontoinen palvelinrooli. Koska testi- ja integraatiopalvelimien haluttiin vastaavan tuotantopalvelinta, sama rooli sopi kaikille ympäristöille.

7.2 Palvelinkonfiguraatiot Puppetin avulla

Puppet-työkalun tehtävä oli saattaa käyttöjärjestelmän tila automatisoidusti toivotun kaltaiseksi. Puppet on luonteeltaan deklaratiivinen, eli ylläpitäjän tehtävä oli esittää Puppetille tila, jota palvelimen tuli vastata. Puppet suoritti tämän jälkeen tarvittavat toiminnot, joilla palvelimen tila pyrittiin saattamaan määritysten

mukaiseksi. Puppetin deklaratiivinen luonne tarkoitti sitä, että projisoinnin tapahtuessa ylläpitäjän ei tarvinnut ottaa huomioon palvelimen tilaa projisointihetkellä. Projisointikomento voitiin toistaa turvallisesti useaan kertaan, ja mikäli järjestelmä oli jo toivotussa tilassa, mitään toimenpiteitä ei suoritettu. Myöhempien konfiguraatiomuutosten tapauksessa Puppet suoritti ainoastaan toimenpiteet joilla edellisen projisoinnin jälkeiset muutokset tuotiin voimaan. Ohjelman todettiin täyttävän tehtävänsä projisoinnin osalta erinomaisesti.

Tilamääritykset kirjoitettiin Puppetin omalla DSL-kielellä resepteiksi, jotka kuvasivat järjestelmän toivottua tilaa melko abstraktilla ja alustariippumattomalla tasolla. Konfiguraatiokokoelmaan luotiin aihekohtaisia moduuleja, jossa yksittäinen aihe oli esimerkiksi käyttöjärjestelmän palomuuri, tai jokin sovellus, kuten Nginx-palvelinsovellus. Yksittäinen moduuli paketoi aihepiiriinsä liittyviä reseptejä sekä tarpeen mukaisia resurssitiedostoja, kuten konfiguraatiotiedostojen sapluunoita. Moduuli, tai sen sisältämä yksittäinen resepti, voitiin tapauksesta riippuen käyttää sellaisenaan, tai sille saatettiin antaa käyttöönoton yhteydessä erilaisia konfiguraatioparametreja. Suoraan käytettävä resepti saattoi esimerkiksi varmistaa että haluttu taustapalvelu löytyy palvelimelta asennettuna ja ajossa olevana (ks. listaus3). Konfiguroitava kohde oli esimerkiksi järjestelmän käyttäjätili, jonka oikeuksilla tapahtumakalenteri käynnistettiin, ja käyttäjätiliin liitetyt kehittäjien julkiset SSH-avaimet, joiden avulla sovellus voitiin julkaista kohdepalvelimella ilman salasanojen hallinnasta aiheutuvaa lisätyötä.

LISTAUS 3: Nginx-palvelinsovelluksen saatavuudesta huolehtiva resepti

```
class nginx {
    package { 'nginx':
        ensure => latest,
    }
    service { 'nginx':
        ensure     => running,
        enable     => true,
        hasstatus  => false,
        hasrestart => true,
        require    => Package['nginx'],
    }
}
```

Resepti saattoi käyttää moduuliin erillisenä tiedostona pakattua konfiguraatiotiedoston sapluunaa, tai jopa kokonaista valmista konfiguraatiotiedostoa sellaisenaan. Puppet tarjosi yksinkertaisia työkaluja sapluunoiden muokkaukseen, sekä riippuvuussuhteiden ja tapahtumaketjujen määritykseen, joilla voitiin esi-

merkiksi varmistaa että palvelu, jonka konfiguraatiotiedostoa muutettiin, käynnistettiin uudelleen. Kokonaisia konfiguraatiotiedostoja käytettiin projektissa muutamassa tilanteessa, esimerkiksi Nginx-palvelinsovelluksen sivustojen määrityksiin. Kokonaisten konfiguraatiotiedostojen säilytys Puppetin moduuleissa vähentää alustariippumattomuutta, mutta projektin palvelinalustat olivat niin homogeenisiä että tämä voitiin toistaiseksi hyväksyä.

Moduuleista ja niiden resepteistä koostettiin ylemmän tason palvelinrooleja. Roolit olivat itsessään reseptejä, joiden tehtävä oli paketoida tapauskohtainen tarkemmin määritelty konfiguraatiokokonaisuus moduuleissa sijaitsevien abstraktien ja yleiskäyttöisten reseptien avulla. Roolin tarkoituksena oli välttää samojen määritysten kopiointia palvelinkohtaisten määritysten välillä tilanteissa, joissa useampi palvelin toteutti samaa roolia. Esimerkkinä yksittäisestä roolista oli jokaiselle palvelimelle asetettu pohjarooli, jossa määritettiin palvelimen palomuuri sallimaan liikennöinnin SSH-porttiin, ja estämään kaikki sisäänpäin suuntautuvan liikenne, jota ei erikseen muissa määrityksissä sallittu. Työssä päädyttiin paketoimaan myös sovelluskohtaiset konfiguraatiot roolia vastaavaksi kokonaisuudeksi.

Lopuksi luotiin varsinaiset palvelinkonfiguraatiot, joissa yksittäiselle palvelimelle annettiin halutut roolit ja mahdolliset palvelinkohtaiset yksittäiset tarkemmat reseptit. Listaus 4 kuvaa erään paikallisen testipalvelimen konfiguraation kokonaisuudessaan. Palvelinkohtaiset konfiguraatiot pysyivät roolien avulla rivimäärällisesti pienenä. Testipalvelin projisoitiin listauksen konfiguraatiolla lähes jokaisen kehityssession alussa, ja projisoinnissa testipalvelin saatettiin puhtaasta asennuksen jälkeisestä tilasta kohdesovellusta palvelevaan tilaan ilman yhdenkään manuaalisen komennon suorittamista kohdejärjestelmässä.

LISTAUS 4: Alpha-nimisen testipalvelimen Puppet-konfiguraatio

```
node 'alpha.gear.dazeconf.net' {
    include role_base
    include role_couchdb
    include role_nodejs
    include role_nginx
    include app_fuzz
    include app_fuzz::nginx-stage
    include access_fuzz_zemm
}
```

7.3 Palvelinprojisointi Gitin avulla

Puppetin dokumentaatio tarjosi yleiseksi käyttötavaksi mallin jossa määritetään keskitetty Puppetmaster-reseptipalvelin, jolta kullakin hallittavalla palvelimella taustaprosessina pyörivä Puppet-asiakas hakee itseään koskevat reseptit, ja tekee tarvittavat toimenpiteet mikäli määritykset eroavat sen nykyisestä tilasta. Tämä malli kuitenkin hylättiin, sillä jatkuvasti ajossa oleva Puppet-asiakasprosessi todettiin tarpeettomaksi resurssien tuhlaukseksi tilanteessa, jossa palvelinkonfiguraatiot muuttuvat harvoin, ja hallittavien kohteiden määrä ei ole suuri. Konfiguraatioprosessi päätettiin toteuttaa siten, että konfiguraatiomuutokset työnnettiin kehittäjän toimesta kohdepalvelimelle, jossa muutokset ajettiin automaattisesti voimaan. Palvelinkonfiguraatiot, kuten muutkin projektiin liittyvät käsitteet, säilytettiin alusta lähtien Git-versiohallinnassa keskuspalvelimella.

Konfiguraatiokokoonpanolle annettiin työnimi Puppmin. Konfiguraatioprosessi suunniteltiin ajettavaksi siten, että hallittavaan järjestelmään luotiin puppmin-niminen käyttäjätunnus. Käyttäjätunnukselle ei asetettu erillistä salasanaa, ja käyttäjätunnuksella kirjautuminen oli mahdollista ainoastaan keskuspalvelimen RSA-avaimen avulla. Käyttäjätunnus lisättiin järjestelmän sudoers-listaan, joka sallii komentojen ajamisen pääkäyttäjän oikeuksin sudo-komennon avulla. Puppmin-käyttäjän sallittiin ajavan pääkäyttäjän oikeuksin ainoastaan puppet-komentoa, jonka avulla puppet-konfiguraatiot realisoitiin järjestelmän tilaksi.

Kohdepalvelimen tuontin Puppmin-hallintaprosessin alle toteutettiin omalla bootstrap-skriptillä (ks. listaus 5). Skriptin kehityksessä kiinnitettiin huomiota kohdepalvelimen vaatimusten minimoimiseen, jotta se voitiin suorittaa täysin muokkaamattomalle vasta-asennetulle kohdepalvelimelle. Skripti suoritti kohdepalvelimelle seuraavat operaatiot:

- Puppet-sovelluksen asennus järjestelmän pakettienhallintatyökalun avulla
- Puppmin-käyttäjätunnuksen luonti järjestelmään
- Ylläpitäjien julkisten RSA-avainten lisäys puppmin-käyttäjätunnuksen autentikaatiokonfiguraatioihin

- Sudo-oikeuksien salliminen puppmin-käyttäjätunnukselle puppet-komennon ajoon
- Kohdepalvelimella sijaitsevan Puppmin-työkalun Git-repositoryn luonti ja konfigurointi siten, että siihen työnnetyt alustakonfiguraatiot ajetaan voimaan välittömästi
- Kohdepalvelimen osoitteen lisäys keskuspalvelimella sijaitsevaan hallittavien palvelinten listaan.

LISTAUS 5: Bootstrap-skriptin ajo testipalvelimelle

```
$ ./deploy_bootstrap johndoe@alpha.gear.dazeconf.net
johndoe@alpha.gear.dazeconf.net's password:
 -- Copying bootstrap* files to johndoe@alpha.gear.dazeconf.net:~/
bootstrap                          837  0.8KB/s 00:00
bootstrap.hooks.post-receive       376  0.4KB/s 00:00
bootstrap.pp                      3232  3.2KB/s 00:00
 -- Copy-operation done.

Should we now execute the bootstrap script on remote server? [y/n] y

 -- Running bootstrap on remote host
Bootstarpping this machine to be controlled with puppmin
Puppet found installed.
 -- Running bootstrap manifest
Group[puppmin]/ensure: created
User[puppmin]/ensure: created
File[/var/puppmin]/ensure: created
Puppmin::Rsa_auth[puppminhub]/Ssh_authorized_key[puppminhub]/ensure:
created
Puppmin::Rsa_auth[zemm@ziranha]/Ssh_authorized_key[zemm@ziranha]/ensu
re: created
File[/var/puppmin/puppmin.git]/ensure: created
Package[git]/ensure: ensure changed 'purged' to 'present'
Exec[puppmin-repository]/returns: executed successfully
File[puppmin-repository-post-receive-hook]/ensure: defined content as
'{md5}b46f31dd100e7e2691f940021183977f'
Exec[Make puppet command password-less for puppmin user with
/etc/sudoers.d/puppmin]/returns: executed successfully
 -- Bootstrap returned successfully, removing bootstrap folder.
All done!
```

Bootstrap-skripti asetti kohdepalvelimelle luodun tyhjän Git-repositoryn keskuspalvelimella sijaitsevan Puppmin Git-repositoryn etäkohteeksi (Git remote). Tämän jälkeen konfiguraatiot voitiin työntää kohdepalvelimelle yhdellä komennolla, jonka jälkeen etäpalvelimen Git-hook ajoi konfiguraatiot välittömästi voimaan. Operaation tuottama tuloste nähtiin välittömästi työntökomennon yhteydessä, ja tuloste voitiin tarvittaessa myös tallentaa lokitiedostoon. Listaus 6

havainnollistaa konfiguraatiotapahtuman kokonaisuudessaan paikalliselta koneelta katsottuna.

LISTAUS 6: Päivitettyjen konfiguraatioiden työntö hallittavalle palvelimelle

```
~/puppmin$ git push alpha
Counting objects: 7, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (4/4), done.
Writing objects: 100% (4/4), 356 bytes, done.
Total 4 (delta 3), reused 0 (delta 0)
remote: Running puppet for commit...
remote: notice: /Stage[main]/Nodejs/Package[libssl-dev]/ensure:
ensure changed 'purged' to 'latest'
remote: notice: /Stage[main]/Appuser/Group[appuser]/ensure: created
remote: notice: /Stage[main]/Role_couchdb/Package[couchdb]/ensure:
ensure changed 'purged' to 'latest'
remote: notice: /Stage[main]/Nginx/Package[nginx]/ensure: ensure
changed 'purged' to 'latest'
remote: notice: /Stage[main]/Nginx/Service[nginx]/ensure: ensure
changed 'stopped' to 'running'
remote: Cleaning up...
remote: All done.
To ssh://puppmin@alpha.geardazeconf.net/var/puppmin/puppmin.git
   60d137c..2dc2018  master -> master
```

Automaattisen projisoinnin testauksen ja ongelmien havaitsemisen vuoksi kehityksen aikana käytetyt virtuaalikoneet palautettiin päivittäin käyttöjärjestelmän asennuksen jälkeiseen puhtaaseen tilaan virtuaalikonealustan tarjoamien tallennettavien tilannekuvien avulla. Käytännössä siis uusi kehityssessio alkoi aina kehitysvirtuaalikoneen projisoinnilla. Käytäntö osoittautui hyväksi, sillä toistuva suoritus pakotti automatisoimaan prosessin mahdollisimman kivuttomaksi, vain kahteen suoritettavaan komentoon. Puhtaan asennuksen jälkeisessä tilassa olleen palvelimen projisointi tuotantopalvelinta vastaavaan tilaan vei asennettavien pakettien määrästä riippuen korkeintaan muutamia minuutteja, ja samaa prosessia käytettiin myös tuotantoalustalle. Käytännössä tämä tarkoitti sitä, että projektialustaksi olisi voitu ostaa uusi palvelin esimerkiksi pilvipalveluntarjoaja Amazonilta, ja palvelin olisi ollut täysin käyttövalmiina korkeintaan kymmenissä minuuteissa.

8 SOVELLUKSEN KEHITYS JA JULKAISU

8.1 Lähtökohdat

Aikaisemmissa kappaleissa voitiin todeta Node.js-sovelluksen toteutuksen olevan mahdollista erittäin pienillä järjestelmäriippuvuuksilla. Sama alustariippuvuuksien minimointiin tähtäävä lähtökohta pyrittiin ottamaan myös sovelluksen julkaisun ja ylläpidon automatisointiin käytetyille ratkaisuille.

Sovelluksen ylläpitoon ja julkaisuun kuuluu erilaisia tehtäviä jotka voidaan suorittaa käyttäjän toimesta tai automatisoidusti. Työn keskeisenä tavoitteena oli automatisoida tällaiset toistuvasti suoritettavat tehtävät yksinkertaisten komentojen taakse. Koska sovelluksen elinkaaren kaikki vaiheet tapahtuivat Linux-käyttöjärjestelmässä, tehtävät oli luontevinta ja tehokkainta suorittaa komentorivityökalujen avulla. Sovelluksen elinkaaren aikana tapahtuvista toimenpiteistä tunnistettiin esimerkiksi:

- riippuvuuksien asennus ja päivitys
- testien ajo ja raportointi
- ympäristökohtaisten asetusten ja ajoskriptien luonti tai linkitys
- mahdollinen staattisten resurssitiedostojen, kuten tyylitiedostojen tai asiakaspään skriptien pakkaaminen pienempään tilaan
- sovelluksen julkaisu palvelimella
- migraatioiden ajo palvelimella ajossa olevan sovelluksen päivityksen yhteydessä.

Työn aikana tutkittiin erilaisia tehtävien suoritukseen tarkoitettuja työkaluja. Perinteinen valinta kuvatun kaltaisten tehtävien ajoon on Unix-alustoilta tuttu Make, jonka toimintamallia myös useat uudemmat työkalut mukailevat. Java-ohjelmointikielen parista löytyi useita paljon käytettyjä monipuolisia työkaluja

kuten Ant tai Maven, mutta näiden käyttö hylättiin Java-keskeisyytensä vuoksi. Samalla periaatteella ohitettiin laajasti käytetty Rake, joka on Ruby-kielellä kehitetty nykyaikainen versio Make:sta. Lopuksi myös Node.js-alustalta löydettiin Make-työkalun JavaScrip-versio, nimeltään Jake. Jaken käyttöä pidettiin aluksi luontevana valintana JavaScript-kieltä käyttävässä projektissa, mutta sen käytössä havaittiin looginen ongelma siinä vaiheessa kun tehtävätyökalun tehtävänä olisi ollut asentaa ajobinääri, jolla sitä itseään ajettiin.

Erilaisten kokeilujen jälkeen projektissa päädyttiin käyttämään perinteistä Make-työkalua sekä Bash-skriptejä, joiden ajo onnistui käytännössä kaikilla Linux-alustoilla ilman esivaatimuksia. Make-työkalun ja sen käyttämän Makefile-tehtävätiedoston yksityiskohtien opettelu ja sen kompastuskivet veivät projektin aikaa, sillä aikaisemmat kokemukset siitä ovat rajoittuneet hyvin yksinkertaisiin operaatioihin käännettävän C-kielen parissa. Make osoittautui nopeasti yhdessä Bash-skriptien kanssa tehokkaaksi työkaluksi hyvin vaihtelevan tyyppisiin tehtäviin, ja yhdistelmän toimivuus lähes millä tahansa Unix-pohjaisella alustalla teki siitä ensisijaisen valinnan myös tulevissa projekteissa.

8.2 Tapahtumakalenterin kehitys

Tapahtumakalenterisivuston tekninen toteutus aloitettiin toimeksiantajan toimittaman kuviossa 2 esitetyn graafisen suunnitelman pohjalta. Kuvatiedostona toimitettu graafinen ilme toteutettiin aluksi epätoiminnallisena HTML-sivuna kiinnittäen huomiota yhdenmukaiseen ja semanttiseen rakenteeseen, joka olisi helppo myöhemmin pilkkoa toiminnallisiin osiin. Tuotettu sivupohja koostui käsin kirjoitetusta rakenteellisesti jaotellusta HTML-tiedostosta, suunnitelmasta poimituista staattisista kuvatiedostoista kuten sivuston logosta sekä css-tiedostoista, joiden avulla HTML-koodi muotoiltiin ulkoasultaan pikselintarkasti suunnitelmaa vastaavaksi. Lopputulos testattiin yleisimmillä käytetyillä selaimilla Windows- ja Linux-käyttöjärjestelmillä.

KUVIO 2: Tapahtumakalenterin graafinen suunnitelma

Seuraavaksi kuviossa 2 esitetystä graafisesta suunnitelmasta tunnistettiin ja määriteltiin seuraavat kuvioon 2 punaisilla numeroilla merkityt toiminnalliset osa-alueet:

1. Sisältöalueella näytetään tulevat tapahtumat listan muodossa. Tapahtumista näytetään yleiset tiedot otsikkorivinä ja käyttäjä voi valita listauksesta haluamansa tapahtuman, jolloin tapahtumasta näytetään tarkempaa lisätietoa.

2. Sivuston yläreunasta löytyy alue, jonne voidaan tehdä nostoja tulevien tapahtumien joukosta. Nostettua tapahtumaa pyritään ensisijaisesti ko-

rostamaan tapahtumassa esiintyvän artistin kuvalla, mikäli sellainen on käytettävissä.

3. Sivupalkin alue varataan myöhemmin määriteltävälle sisällölle, joka alustavien suunnitelmien mukaan koostuisi ylläpidon tuottamasta vapaasta sisällöstä ja mahdollisista mainoksista, mikäli sellaisia päädyttäisiin sivustolle asettamaan.

4. Sivuston alapalkki varataan ylläpitäjien tuottamalle vapaamuotoiselle HTML-sisällölle.

Seuraavaksi sovellukselle luotiin Express-ohjelmistokehyksen päällä toimiva runko. Aluksi sovellus asetettiin vastaamaan selaimelta tulevaan palvelupyyntöön suunnitelmaa vastaavalla staattisella sivupohjaksi luodulla HTML-tiedostolla. Sovellukseen liitettiin Express-moduuli, jonka avulla sivuston staattiset resurssit, kuten ulkoasun kuvat ja css-tyylitiedostot voitiin toimittaa asiakkaalle suoraan tiedostojärjestelmästä. Tämän jälkeen nostettujen tapahtumien palkki piilotettiin ulkoasusta, sillä se toteutettaisiin vasta myöhemmässä vaiheessa.

Express tuki useiden erillisten näkymämoottorien käyttöä. Moottorin tehtävä oli näkymän renderöinti esimerkiksi käsittelemällä ja yhdistämällä sovelluskehittäjän kirjoittama näkymäkoodi ja erilliset näkymien sapluunatiedostot. Työssä päädyttiin käyttämään ejs-moottoria, jossa ohut näkymäkoodi kirjoitettiin sapluunatiedoston sisään. HTML-näkymän tapauksessa sapluunatiedosto sisälsi esikirjoitetun HTML-koodin ja siihen upotetun dynaamisesta sisällöstä vastaavan näkymäkoodin, jonka ejs-moottori käsitteli ja ajoi renderöinnin yhteydessä. Kyseinen toimintamalli oli suhteellisen tehokasta, mutta se vaati kehittäjältä HTML-koodin oheen kirjoitetun näkymäkoodin määrän pitämisen mahdollisimman pienenä.

Express tarjosi toiminnot joiden avulla näkymäriippumattomasta sovelluksen toimintalogiikasta vastaavasta koodista voitiin lähettää tietueita näkymäkoodille käytössä olevasta renderöintimoottorista riippumattomalla tavalla. Sovelluksen toiminta- ja näkymälogiikka voitiin näin erottaa toisistaan. Express-sovellukseen kirjoitettut toimintalogiikasta vastaavat ohjelmakoodit saivat aikaisemmin kuvatuista Connect-esikäsittelijöistä koostetun ketjun läpi ajetun käsitellyn

palvelupyyntöobjektin, josta logiikkakoodi saattoi tarvittaessa poimia asiakassovelluksen lähettämiä parametreja.

Tapahtumakalenterin sisältöalueen tapauksessa sovelluksen toimintalogiikasta vastaava ohjelmakoodi suoritti tietokantahaun dokumenttitietokantaan, jonka map/reduce-funktiona luodut näkymätoiminnot palauttivat hakukyselyssä annettujen parametrien mukaisen listauksen tietokantaan tallennetuista tapahtumatietueista. Sovelluksen logiikkakoodi välitti tietokantanäkymälle palvelupyyntöobjektiin sisällytetyt parametrit, jotka tapahtumalistauksen tapauksessa liittyivät listauksen rajaukseen ja tulosten sivutukseen.

Sovelluslogiikka toimitti dokumenttitietokannan haun palauttamat tapahtumatietueet näkymämoottorille, näkymäkoodin käyttöön. Kun sovelluslogiikan ajo päättyi, välitettiin Express-ohjelmistokehykselle tieto sivupohjista jotka sen tulisi ajaa näkymämoottorin läpi. Tapahtumakalenterin etusivun HTML-näkymän tapauksessa sivuston runko toimi pohjanäkymänä, johon sisältyi kuviossa 2 numeroidut neljä alinäkymää, jotka oli erotettu omiksi alinäkymikseen. Näkymämoottori ajoi pohjanäkymän ja määritetyt alinäkymät, ja lopuksi kaikkien ajojen tulokset yhdistettiin ja lopputuloksena syntynyt renderöity näkymä sisällytettiin asiakasohjelmalle lähetettävään palvelupyynnön vastaukseen. Myös kuvioon 2 merkittyjen HTML-alueiden 3 ja 4 sisältö säilytettiin dokumenttitietokannassa, josta alueiden tapahtumakäsittelijät hakivat ne.

Sivustoon liitettiin myöhemmässä vaiheessa Ajax-toiminnallisuutta, jolloin asiakasohjelmassa ajettava JavaScript-ohjelmakoodi saattoi hakea palvelimelta sivun tietyn osan, ja päivittää sen selaimen näkymään ilman perinteistä sivun uudelleenlatausta. Tässä tapauksessa asiakasohjelma lähetti palvelimelle HTTP-protokollan mukaisen Accept-tietueen, joka indikoi asiakasohjelman haluavan vain osan sivusta esimerkiksi JSON-objektina. Express ja väliohjelmistot tarjosivat toiminnallisuuden, jonka avulla sovelluskoodi pystyi havaitsemaan palvelupyyntöön sisällytetyn toiveen vastauksena palautettavasta resurssista ja sen esitysmuodosta. Koska sivusto oli jaettu toiminnallisiin osuuksiin sekä toimintalogiikan että näkymälogiikan tasolla, voitiin pyynnön käsittely rajata helposti ainoastaan pyydetyn osa-alueen käsittelyyn ja renderöintiin.

8.3 Julkaisutapahtuma

Työn lähtökohdissa asetettujen tavoitteiden mukaan sivuston julkaisun tuli tapahtua ilman kohdepalvelimella kehittäjän toimesta suoritettavia toimintoja. Paras tapa sovelluksen julkaisuun olisi käyttöjärjestelmän natiivin pakettienhallintajärjestelmän käyttö (Humble & Farley 2011), mutta koska käytettyjen Ubuntu-palvelimien Debian-pohjainen paketointikäytäntö ei ollut ennestään tuttua, ja toisaalta koska projekti oli jo täynnä uusia teknologioita, päätettiin natiivipaketoinnin käyttö jättää tulevaisuuden kehitystavoitteeksi.

Ratkaisu jätti jäljelle julkaisustrategian, joka koostui kahdesta osa-alueesta: julkaistavan sovelluksen toimittaminen kohdepalvelimelle ja julkaisuun liittyvien toimintojen (asennus, migraatiot, smoke-testit ja sovelluksen käynnistys) suoritus kohdejärjestelmässä. Erilaisia julkaisustrategioita kokeiltiin pienien testisovelluksien ja omien Bash-skriptien avulla. Lopuksi työssä päädyttiin käyttämään Express-ohjelmistokehyksen kehittäjän julkaisemaan minimalistista Bash-skriptinä toteutettua julkaisutyökalua nimeltä Deploy (Deploy 2011).

Deploy oli toteutettu yksinkertaisena Bash-skriptinä, joka oli kooltaan pieni, eikä se tuonut projektiin lisäriippuvuuksia Gitiä lukuun ottamatta. Työkalun käyttö koostui muutamasta julkaisuun liittyvästä komennosta, ja siihen sisältyi tuki usealle julkaisukohteelle sekä kohdekohtaisille konfiguraatioille. Julkaisutyökalulle kirjoitettiin konfiguraatiotiedosto, jonne julkaisuympäristöt asetuksineen määriteltiin. Työkalun avulla sovelluksen määrätty versio (käytännössä siis jokin Gitin julkaisuhaaran numeroitu versio) voitiin toimittaa etäpalvelimelle sekä ajaa tämän jälkeen asetustiedostoon määritellyt komennot etäpalvelimella.

Deploy käytti tiedostojen siirtoon Gitiä siten, että etäpalvelimelle luotiin kopio sovelluksen Git-repositorysta, jonka jälkeen toistuva päivitysten siirto palvelimelle tapahtui Gitin työntöoperaation avulla. Toimintatavan etuihin lukeutui tehokas verkkoliikenteen käyttö, sillä versiohallinnan avulla suoritetussa tiedonsiirrossa siirrettiin ainoastaan muutokset kokonaisen sovelluspaketin sijaan. Haittapuoleksi voitiin lukea kohdepalvelimelta vaadittu Git, mutta automatisoidun palvelinhallinnan myötä sen lisääminen ei ollut ongelma.

Julkaisu tapahtui kehityskoneella, jonka julkisella RSA-avaimelle voitiin kirjautua julkaisun kohteena olevalla palvelimelle julkaistavaa sovellusta ajavan sovelluskäyttäjän tunnuksella. Sallittujen RSA-avainten hallinta tapahtui Puppetin avulla sovelluskäyttäjätilin määrittelyn yhteydessä.

Yksi julkaisutapahtuman vaatimuksista oli, että kehittäjän ei tarvitsisi kirjautua kohdepalvelimelle ja suorittaa siellä komentoja. Deploy-työkalun, itse luotujen Bash-skriptien ja niitä ehdollisesti suorittavan Make:n avulla tämä tavoite saavutettiin. Julkaisutapahtuma tapahtui kehittäjän näkökulmasta yhdellä komennolla, ja sivuston valittu versio julkaistiin kehityksen aikana toistuvasti sekä testipalvelimelle että julkiselle palvelimelle. Tuotantopalvelimella julkaistun keskeneräisen sovelluksen näkyvyys oli kuitenkin työn ajan rajoitettu sovelluksen ollessa vielä keskeneräinen.

Deploy-työkalulle voitiin määrittää komentoja, joita se ajoi julkaisutapahtuman eri vaiheissa. Onnistuneen tiedostojensiirron jälkeen työkalu asetettiin ajamaan riippuvuuksien päivityksestä ja ajoympäristökohtaisen käynnistysskriptin linkityksestä huolehtivat Make-komennot. Operaatioiden jälkeen työkalu päivitti palvelimella sijaitsevan työkopion ja ajoi sille konfiguraatioon määritellyt testikomennot. Sovellukselle oli kirjoitettu yksikkötestien oheen muutama korkean tason smoke-test, joiden tarkoituksena oli havaita ympäristön olennaisimmat ongelmat, kuten tietokannan saatavuus. Mikäli asetuskomento tai testien ajo epäonnistui, Deploy-työkalu palasi automaattisesti aikaisempaan toimivaksi todettuun versioon. Myös kyseinen toiminnallisuus testattiin käytännössä koittamalla rikkinäisen version julkaisua.

Kun asetusvaihe ja testit oli suoritettu onnistuneest, Deploy-työkalu ajoi mahdolliset päivityksessä tulleet uudet migraatiot. Migraatiot luotiin kronologisessa järjestyksessä numeroituina ajettavina tiedostoina. Kohdealustalla säilytettiin paikallisesti tieto siitä, minkä numeron omaava migraatio oli viimeksi ajettu. Julkaisun yhteydessä voitiin näin etsiä mahdolliset uudet migraatiot ja ajaa ne. Migraatioskriptejä käytettiin ensisijaisesti tietokannan hallintaan, ja se ratkaisi ongelman kehityksen välillä tuotettujen tietokantamuutosten toistamisesta päivityksen kohteessa. Uudelle alustalle julkaisun yhteydessä kaikki olemassa olevat migraatiot ajettiin järjestyksessä, ja ensimmäiseen migraatioon sisältyi sovelluksen käyttämän tietokannan luonti.

Viimeisenä Deploy-työkalu suoritti skriptin, jolla päivitetty sovellus käynnistettiin ajoon. Sovelluksen käynnistys toteutettiin ajoympäristön tiedostavalla Bash-skriptillä, sillä sovellusta ajettiin eri tavalla eri ympäristöissä. Kehityspalvelimella sovelluksen ajoon käytettiin apuohjelmaa, joka tarkkaili lähdekoodiin tehtyjä muutoksia, ja käynnisti sovelluksen automaattisesti uudelleen mikäli sen lähdekoodit muuttuivat. Tämä nopeutti kehitystä, sillä muutoin lähdekoodimuutosten ja testiajon välissä sovellus olisi jouduttu käynnistämään uudelleen kehittäjän syöttämillä komennoilla.

Tuotantopalvelimella sovelluksen ajoon käytettiin Forever-nimistä apuohjelmaa, joka asettui tausta-ajoon tarkkailemaan sovellusta sen ajon aikana. Mikäli sovelluksen suorituksessa tapahtui virhe ja sen suoritus loppui, Forever yritti sen uudelleenkäynnistämistä. Palvelinympäristön käynnistysskripti otti myös huomioon mahdollisen ennestään ajossa olevan sovelluksen ja sen sulkemisen ennen päivitetyn version käynnistystä.

9 PÄÄTELMÄT JA YHTEENVETO

9.1 Tärkeimpänä tuloksena prosessit

Työssä keskityttiin Node.js-sovelluksen julkaisuprosessiin, mutta käytetyt metodit soveltuvat ja ovat tarpeellisia myös muilla teknologioilla toteutettujen internetsovellusten kehityksessä. Työ osoitti automatisoitujen vaiheiden hyödyt aikaisemmin käytettyihin metodeihin verrattuna.

Erityisesti palvelimen automaattinen projisointi voitiin todeta tarpeelliseksi missä tahansa ympäristössä jossa hallitaan useampaa kuin yhtä palvelinta. Käytännössä jokainen projekti kattaa useamman palvelinalustan kun mukaan lasketaan kehitysympäristöt. Nopeasti ja varmasti toistettava projisointi, joka ei vaadi konfiguraatiotiedostojen käsin muuttelua, toi kehitykseen varmuutta ja mielikuvan siitä, että käytäntö täytyy ottaa mukaan myös tuleviin projekteihin.

Toisaalta projisointiprosessin kehitys, testaus ja Puppet-konfiguraatioiden luonti muodostivat ajallisesti työn suurimman osuuden. Puppet-konfiguraatioiden kirjoitusta ei koettu helpoksi, ja myöskään tuotettuihin resepteihin ei usein oltu tyytyväisiä. Puppetin oma DSL-kieli unohtui nopeasti ja kaikkiin sen helpottaviin ominaisuuksiin ei ehditty työn aikana tutustua. Osa resepteistä päädyttiin toteuttamaan omilla Bash-skripteillä, joka ei ole suositeltava ratkaisu, sillä tällöin menetetään osa Puppetin tarjoaman abstaktion hyötyistä, ja konfiguraatioista tulee alustariippuvaisia. Omat pitkät Bash-skriptit tekivät myös ongelmista vaikeasti havaittavia ja olivat hitaita kirjoittaa. Puppetin tarkoitus olisi nimenomaan korvata vastaavan kaltaiset yleiskäyttöisellä kielellä kirjoitetut ylläpitoskriptit.

Ongelmat eivät välttämättä olleet niinkään Puppetissa kuin sen käyttäjissä ja kokemattomuudessa. Lisäkokemuksen myötä ongelmat tulevat todennäköisesti vähentymään ja moduulien laatu parantumaan. Projisointityökalun käyttö voitiin todeta ehdottoman tarpeelliseksi kaikissa projekteissa, joten hankaluuksista huolimatta sen käyttöä tullaan jatkamaan myös seuraavissa projekteissa.

Tarjolla on myös projisointityökaluja joita ei tässä työssä nimetty, ja niiden väliseen vertailevaan tutkimukseen päätettiin varata sopivalla hetkellä aikaa.

Työn tärkein tulos oli lukuisia kertoja testattu toistettava prosessi, jota käyttämällä kohdesovellus voitiin julkaista oletusasetuksilla asennetulle muokkaamattomalle palvelimelle siten, että kehittäjä kutsui ainoastaan muutamia työssä tuotettuja asennusskriptejä ja komentoja. Uuden sovellusversion julkaisu onnistui vastaavasti yhden komennon avulla ilman esivalmisteluja. Julkaisutoimenpiteeseen sisältyi tietoisuus virhetilanteista ja niiden käsittely aikaisempaan toimivaan versioon paluun muodossa. Prosessin avulla voitiin suurimmaksi osaksi eliminoida inhimillisen erehdyksen tekijä ja monimutkainen puutteellisesti dokumentoitu manuaalinen julkaisuprosessi.

9.2 Kasvanut teknologiavalikoima

Lähes kaikki työssä käytetyt teknologiat olivat työn osapuolille ennestään tuntemattomia, joka näkyi hitaan edistymisen muodossa. Työstä saatiin kuitenkin kokemusta ja tietoisuutta monista uusista työkaluista, joille nähtiin käyttöä tulevaisuudessa.

Dokumenttitietokannan käyttöönotto esimerkiksi PHP-projekteissa nähtiin todennäköisenä. Dokumenttitietokannan käyttöä suunniteltiin myös erilaisten sisäisten projektien kuten tehtävienhallinta- ja asiakastietojärjestelmien tietovarastona. Node.js-alustaa ei toistaiseksi aiottu käyttää PHP-alustan korvaamiseen tavanomaisissa projekteissa, mutta ympäristön tuntemus mahdollistaa aikaisempaa interaktiivisempien ja reaaliaikaisempien sovellusten toteutuksen tarvittaessa.

Make-työkalun ja Bash-skriptien käyttöä päätettiin lisätä tulevissa projekteissa. Yhdistelmän käyttöönotto ei vaadi käytännössä mitään esivalmisteluja, ja yksinkertaistien toistettavien tehtävien, kuten vaikkapa projektin resurssien pakkaamisen vieminen yhden make-komennon taakse vapauttaa pitkällä aikavälillä paljon aikaa tärkeämpien ongelmien ratkaisuun ja itse sovelluksen kehitykseen.

9.3 Tapahtumakalenterin jatkokehitys

Koska valtaosa työhön käytetystä ajasta kului prosessien hiomiseen yritysten, erehdysten ja kokeilujen muodossa, sekä uusien sovellusten ja työkalujen käytön opetteluun, ei esimerkkisovelluksena toimineen varauskalenterisivuston taustalla toimivia tiedonkerääjiä ehditty toteuttaa työn aikana toimiviksi. Tapahtumakalenterisivustoa ei siten asetettu vielä kirjoitushetkellä julkiseksi. Projektin valmiiksi saattamisesta ja jatkokehityksestä sovittiin toimeksiantajan kanssa, ja sivusto tullaan saattamaan julkaisukuntoiseksi aluksi yksinkertaisena versiona, jonka jälkeen siihen on tarkoitus lisätä vaiheittain lukuisia lisäominaisuuksia.

Käytetty aika kului kohteisiin, joiden kuntoon laitto oli tärkeää heti projektin alussa ennen aktiivisen kehitystyön aloittamista. Lisäksi aikaa vieneiden kohteiden pääasiallinen tarkoitus oli tähdätä ajan säästämiseen pitkällä aikavälillä vähentämällä toistettujen manuaalivaiheiden määrää ja niistä johtuvia ongelmia. Pohjatyö saatettiin työn aikana siihen vaiheeseen, että jatkossa käytetty aika voidaan suunnata täysipainotteisesti itse sovelluksen kehitykseen. Sivuston uudet versiot voidaan julkaista vaivattomasti heti kun ne todetaan vakaiksi.

Uuden Node.js-projektin luonti onnistuu tuotetun pohjatyön päälle huomattavan pienellä vaivalla. Käytännössä uuden projektin aloitus vaatii sovelluskäyttäjän ja mahdollisten alustariippuvuuksien, esimerkiksi Nginx-määritysten (mikäli sen käyttöä jatketaan) lisäämisen projisointikonfiguraatioihin ja uuden sovellusrepositoryn luonnin. Näiden vaiheiden jälkeen uusi sovellus voidaan kehittää ja julkaista käyttämällä samoja julkaisu- ja riippuvuuksienhallintatyökaluja joita tässä työssä tutkittiin.

LÄHTEET

About Nginx, n.d. Nginx-palvelinsovelluksen tietosivu. Viitattu 17.11.2011. nginx.org/en/.

Chacon, S. 2010. Pro Git. Julkaisun verkkoversio. Viitattu 29.10.2011. progit.org/book/.

CouchDB Technical Overview. n.d. CouchDB-tietokannan dokumentaatio. Viitattu 5.11.2011. couchdb.apache.org/docs/overview.html.

Deploy, 27.7.2011. Deploy-sovelluksen sivu Github-projektisivustolla. Viitattu 20.11.2011. github.com/visionmedia/deploy.

Driessen, V. 5.1.2010. A successful Git branching model -blogikirjoitus. Viitattu 16.10.2011. nvie.com/posts/a-successful-git-branching-model/.

Express. n.d. Express-ohjelmistokehyksen kotisivu. Viitattu 14.11.2011. expressjs.com/.

Freeman, S. & Pryce, N. 2010. Growing Object-Oriented Software, Guided by Tests. USA: Addison-Wesley.

Gittutorial. 2011. Gitin sisäänrakennettu ohjekirja (Linux Man page). Viitattu 25.10.2011.

Git vs. Mercurial. n.d. Viitattu 30.10.2011. gitvsmercurial.com/

Humble, J. & Farley, D. 2011. Continuous Delivery: Reliable Software Releases through Build, Test and Deployment Automation. USA: Addison-Wesley.

Joyent, Inc. 2010. About Node.js -tietosivu. Viitattu 25.10.2011. nodejs.org/#about.

Kulttuurituotanto-osuuskunta Raymond tietosivu. n.d. Viitattu 16.10.2011. www.raymond.fi/osuuskunta.php

Martin, R. 2009. Clean Code: A Handbook of Agile Software Craftsmanship. USA: Prentice Hall.

Merriman, D. 2011. Comparing Mongo DB and Couch DB -artikkeli. Viitattu 28.10.2011. www.mongodb.org/display/DOCS/Comparing+Mongo+DB+and+Couch+DB.

Nipster. 2011. Npm-moduulien suosiota kartoittava lista. Viitattu 3.11.2011. eirikb.github.com/nipster/.

Opscode Customers. 2011. Tietosivu Opscode Chef -sovelluksen tunnetuista asiakkaista. Viitattu 15.11.2011. www.opscode.com/customers/.

Organizations using Puppet. 2011. Tietosivu Puppet Labs Puppet -sovelluksen tunnetuista asiakkaista. Viitattu 15.11.2011. projects.puppetlabs.com/projects/puppet/wiki/Whos_Using_Puppet.

Preston-Werner, T. n.d. Semantic Versioning -versionumeroinnin määritys. Viitattu 14.10.2011. semver.org/.

Siniaalto, K. 2006. Test driven development: empirical body of evidence. Viitattu 30.10.2011. www.agile-itea.org/public/deliverables/ITEA-AGILE-D2.7_v1.0.pdf

Sink, E. 2011. Version Control by Example. USA: Pyrenean Gold Press.

Webber J., Parastatidis, S. & Robinson, I. 2010. REST in Practice: Hypermedia and System Architecture. USA: O'Reilly.