View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Theseus

jamk.fi

Jatkuvan julkaisun mahdollistaminen ohjelmistokehityksessä

Markus Klinga

Opinnäytetyö
Toukokuu 2015

Tietojenkäsittelyn koulutusohjelma
Luonnontieteiden ala

JYVÄSKYLÄN AMMATTIKORKEAKOULU
JAMK UNIVERSITY OF APPLIED SCIENCES

Kuvailulehti

Tekijä(t) Klinga, Markus	Julkaisun laji Opinnäytetyö	Päivämäärä 7.5.2015
	Sivumäärä 63	Julkaisun kieli Suomi
		Verkkojulkaisulupa myönnetty: (x)

Työn nimi
Jatkuvan julkaisemisen mahdollistaminen ohjelmistokehityksessä

Koulutusohjelma
Tietojenkäsittelyn koulutusohjelma

Työn ohjaaja(t)
Jarkko Immonen

Toimeksiantaja(t)
SC5 Online Oy

Tiivistelmä

Ohjelmistokehityksen siirtyminen ketteriin menetelmin vaatii kulttuurinmuutoksen myös julkaisemisen prosesseihin. Tutkimuksessa tarkasteltiin ohjelmiston kehitystyön jatkuvaa julkaisemista sekä tämän vaatimia työkaluja ja ratkaisuja kehittämistutkimuksena.

Tutkimuksessa toteutettiin automatisoitu julkaisuputki SC5:n ohjelmistokehityksen yhteyteen ja osoitettiin tämän toimivuus sekä tästä saadut hyödyt niin asiakkaan kuin kehittäjienkin välillä. Toteutuksessa rakennettiin lisäksi kehittäjien työkoneilla toimiva kehitysympäristö automaatiota ja virtualisointia hyväksi käyttäen.

Tutkimuksessa havaittiin paljon etuja jatkuvan julkaisemisen menetelmien ja teknologioiden käyttöönottamisesta. Julkaisuprosessi pysyi läpinäkyvänä kehittäjien välillä ja julkaisumenetelmä tuki välittömän palautteen saamista kehitystyöstä.

Ohjelmistokehityksen siirtyessä yhä enemmän Internet-pohjaisiin ratkaisuihin on nopea julkaiseminen tärkeässä roolissa ohjelmiston saattamisessa asiakkaiden käytettäväksi. Palvelupohjaiset web-sovellukset mahdollistavat entistä lyhemmän matkan kehitystyöstä kuluttajien saataville. Tulevaisuuden haasteina ovatkin tämän matkan automatisointi niin kehitys- kuin palvelinpuolellakin.

Avainsanat
Automaatiojärjestelmät, verkko-ohjelmointi, prosessit, virtuaaliympäristö, tietotekniikka-arkkitehtuuri, palvelimet, ohjelmistosuunnittelu

Muut tiedot

JYVÄSKYLÄN AMMATTIKORKEAKOULU
JAMK UNIVERSITY OF APPLIED SCIENCES

Description

Author(s) Klinga, Markus	Type of publication Bachelor’s Thesis	Date 7.5.2015
	Pages 63	Language Finnish
		Permission for web publication: (X)

Title
Enabling continuous deployment in software development

Degree Programme
Business Information Systems

Tutor(s)
Immonen, Jarkko

Assigned by
SC5 Online Oy

Abstract

New, agile methods in software development requires also a cultural shift in the deployment processes. The Thesis describes the methods and tools that are essential to the continuous deployment of web software.

Within the thesis an automated deployment pipeline was built to continuously publish the developed project. The functionality and profit for both the customer and the developers was shown during the research. In the implementation also the developers' work environments were virtualized using proper automation.

In the thesis several benefits were found in the continuous deployment processes and technologies. The deployment process was transparent between the developers and supported instant feedback from the development.

While the software development is moving rapidly to Internet-based solutions quick deployment will play an important role in getting the software to the customers' reach. Service-based web-applications enable even shorter distance between the development and the customers. Challenges in the future will be automating this journey on both the development and the server side.

Keywords
Automated systems, web development, processes, virtual environment, IT-architecture, servers, software development

Miscellaneous

Sisältö

1 Johdanto 3

2 Tutkimusasetelma 4

2.1 Toimeksiantaja 4

2.2 Opinnäytetyön taustaa, aihe ja rajaukset 5

2.3 Tutkimusmenetelmä sekä tavoitteet 6

3 Devops-kulttuuri 8

3.1 Yleiskatsaus 8

3.2 Ohjelmisto 10

3.3 Automatisoinnin merkitys 13

3.4 Julkaisuputki 16

3.5 Palvelin-infrastruktuuri 19

4 Käytettävät työkalut 23

4.1 Työkalujen valinta 23

4.2 Vagrant 24

4.3 Ansible 27

4.4 Docker 30

4.5 Jenkins 36

5 Tutkimuksen toteutus 39

5.1 Julkaisuputki 39

5.2 Kehitysympäristö 45

6 Jatkuvan julkaisun mahdollistaminen 51

7 Pohdinta 54

Lähteet 57

Liitteet 58

Liite 1. Dockerfile 58

Liite 2. Vagrantfile 59
Liite 3. Ansible-käsikirja 60
Liite 4. Vagrant initfile 62

Taulukot

Taulukko 1. Esimerkki kuvailevien käskylauseiden tulkinnasta 22

1 Johdanto

Ohjelmistokehityksen päämääränä voidaan pitää jonkin idean valmistamista käytettäväksi tuotteeksi tai palveluksi. Tällöin varsinainen ohjelmakoodin kirjoittaminen, testaaminen, tuotantoympäristöön integroiminen tai vaikkapa ympäristön kapasiteetin varmistaminen ovat itse prosessin näkökulmasta vain senkaltaista välttämätöntä ajanhukkaa, jota on mahdotonta estää.

Ketterän kehityksen mallit ovat muuttaneet radikaalisti perinteisiä kehitysmenetelmiä, ja iteratiivinen ohjelmistokehitys onkin lyönyt itsensä läpi vauhdilla viime vuosikymmenten aikana. Etenkin nopeatempoisessa web-kehityksessä SCRUM, Kanban ja muut vastaavat työskentelymallit ovat voimakkaita ja tehokkaita. Nämä menetelmät kattavat usein kuitenkin vain pelkän ohjelmiston kehitystyön eivätkä ylety tuotantoketjussa tuotteen julkaisemisen tasolle.

Ohjelmistoja ketterin menetelmin kehittävät yritykset ovatkin yhä usein siilouttaneet ylläpitäjät ja kehittäjät toisistaan erillisiksi yksiköiksi, joiden välinen kommunikaatio saattaa rajoittua pelkän kehittäjien valmistaman tuotteen mukana ylläpitäjille luovutettavan README.txt-tiedoston tasolle.

Varsinainen ohjelmiston siirtäminen tuotantopalvelimelle tapahtuukin usein siten, että ylläpitäjä alkaa kehittäjältä saadun tuotteen pohjalta valmistelemaan mahdollisia asennus-skriptejä, muokkaamaan manuaalisesti asetustiedostoja vastaamaan tuotantoympäristöä ja asentelemaan tuotantopalvelimelle tarvittavia työkaluja ja kirjastoja. Parhaassakin tapauksessa ylläpitäjä vain toistaa saman työn, jonka kehittäjä on jo omaa ympäristöään varten suorittanut, pahimmassa tapauksessa ylläpitäjä luo uusia, korjaamista vaativia bugeja. (Edwards 2010.)

Ohjelmiston tuotantoympäristöön vientiä kaikkineen leimaavat kaoottisuus, ohjeistuksen puute sekä lopulta puutteelliseksi todetun, jo julkaistun tuotteen korjaaminen dokumentoimattomiksi jäävillä pikakorjauksilla suoraan tuotantoympäristössä. Kor-

jauksista jää pahimmillaan jälki vain ylläpitäjän muistiin tai "asennusohjeisiin", jotka sijaitsevat ylläpitäjien tietopankissa.

Devops (yhdistelmä englanninkielisistä sanoista "development", kehitys sekä "operations", ylläpito) on eräänlainen konsepti, jonka tarkoituksena lyhyesti kuvattuna on nivoa umpeen kehittäjien ja ylläpitäjien välistä tietorailoa ohjelmistokehityksen eri vaiheissa (Hüttermann 2012, 4). Yllä kuvattu julkaisun kertaluonteisuus on eräs kipeimmistä ongelmakohdista, mutta ei suinkaan ainoa: ylläpidettävän palvelun monitorointi, tiedonkeruu ja siihen reagoiminen, virheensietokyvyn varmistaminen ja jopa palvelimen arkkitehtuurimuutokset hyötyvät suuresti sekä kehittäjien että ylläpitäjien osallistumisesta yhteistuumin ohjelmistoa koskevien ongelmien ratkaisemiseen (Allspaw 2009).

Devops-kulttuuri on vasta löytämässä paikkaansa ohjelmistoyritysten sisällä. Vähitellen menetelmät, kuten jatkuva julkaiseminen tai automatiikan välttämättömyys kuitenkin vakiintunevat osaksi arkipäiväistä työntekoa. Mallin mukanaan tuomat hyödyt ovat perinteiseen kehittäjien ja ylläpitäjien rajanvetoon verraten monella tavalla edulliset. Tässä opinnäytetyössä käydään läpi esimerkinomaisesti tämän kaltaisen kulttuurin saattamista kiinteäksi osaksi modernia web-ohjelmistokehitystä ohjelmiston jatkuvan julkaisemisen mahdollistamiseksi toimeksiantajan puitteissa.

2 Tutkimusasetelma

2.1 Toimeksiantaja

Toimeksiantajana opinnäytetyölle toimii SC5 Online Oy, joka on ketterä, uudenaikainen ohjelmistotalo. SC5 työllistää nuoresta iästään huolimatta jo noin 80 henkeä ja on jatkuvasti kasvamassa yhä suuremmaksi toimijaksi web-pohjaisen ohjelmistokehityksen saralla.

Yhä digitalisoituvammassa maailmassa erityisosaaminen on kultaa, ja SC5 pyrkii muovaamaan tätä uutta, monelle asiakkaallekin vielä tuntematonta maastoa oman ammattitaitonsa sekä innostuksensa kautta. Yritys noudattaa sisäisesti matalaa hie-

rarkiaa, mikä mahdollistaa nopean kehittymisen ja keskittymisen uusimpiin teknologioihin sekä käytänteisiin myös työntekijöiden mielenkiinnon kohteiden mukaisesti.

SC5 on perinteisesti erikoistunut modernien web-tekniikoiden (kuten HTML5) tuomiseen ohjelmistokehityksen kärkeen, ja etsii jatkuvasti uusia alueita vallatakseen ne. Yritys suuntaa erityisesti loppukäyttäjä-keskeisten tuotteiden valmistamiseen, joissa pääpaino kehityksessä suunnataan etenkin käytettävyyteen ja toimivuuteen perinteisten ohjelmistoyritysten teknologiapainotteisen lähestymisen sijaan.

Loppukäyttäjänä tarkoitetaan sananmukaisesti ihmistä, joka valmista ohjelmistoa lopulta käyttää. SC5:n valmistamana tuotteena saattaakin olla vaikka teollisen Internetin mahdollistama työkoneen käyttöliittymä, Internet-pohjainen sanomalehti tai koukuttava online-peli.

2.2 Opinnäytetyön taustaa, aihe ja rajaukset

Opinnäytetyön aiheeksi valittiin jatkuvan julkaisemisen mahdollistaminen sekä ns. Devops-kulttuurin tuominen SC5:n ohjelmistokehitykseen. Käytännössä tämä tarkoittaa kehitteillä olevan ohjelmiston mahdollisimman sujuvaa, automatisoitua julkaisutapaa, jossa ylläpitäjien ja kehittäjien perinteisesti erilliset roolit limittyvät toisiinsa.

Yrityksen aiempi rooli ns. frontend-kehittäjänä ei ole vaatinut suurta panostusta julkaisuautomatiikan puolelle, mutta halu muutokseen Devops-kulttuurin, kokonaisvaltaisten ohjelmoijien (eng. full stack developer) ja erilaisten uusien teknologioiden myötä vaatii panostusta myös tällä osa-alueella ohjelmistokehitystä. Aiemmin yrityksessä valmistettujen ohjelmistojen siirtäminen tuotantopalvelimille sekä muu julkaisunjälkeinen toiminta on joitakin poikkeuksia lukuun ottamatta jäänyt asiakkaiden ylläpidon vastuulle, jolloin kehittäjien ja ylläpitäjien välinen matka on väistämättä ollut erittäin pitkä, eikä riittävää kommunikaatiota ole päässyt syntymään.

Opinnäytetyön tutkimus on rajattu kattamaan jatkuvan julkaisun käsite ja sen mahdollistaminen kehittäjälähtöisesti SC5:ssä. Lähdemme liikkeelle kehittäjien käyttämistä työkaluista ja muovaamme julkaisuputken kehyksen, jota kyetään tarpeiden mu-

kaan laajentamaan. Julkaisemiseen kiinteästi liittyvät palvelinratkaisut tullaan tämän työn puitteissa pääosin sivuuttamaan, sillä ne olisivat laajuutensa puolesta jo kokonaisen tutkimuksen aihe itsessään.

Tutkimuksessa käsitellään jatkuvaa julkaisua lähtökohtaisesti puolueellisesti SC5:ssä jo käytössä olevien tai "hyväksyttyjen" teknisten ratkaisujen kautta, mutta myös vaihtoehtoisia ohjelmistoja ja toimintatapoja pyritään esittelemään. Aiheen puitteissa ei liene yhtäkään osa-aluetta, jossa ei olisi mahdollista valita jotakin täysin toista työkalua tai ohjelmistoa ja saada aikaan tismalleen yhtä onnistunut lopputulos.

2.3 Tutkimusmenetelmä sekä tavoitteet

Opinnäytetyö toteutetaan kehittämistutkimuksena. Tässä yhdistyvät organisaatioissa luonnollisesti tapahtuva toimenpiteiden kehittäminen prosessien, tuotteiden tms. parissa sekä tämän kehittämistyön suorittaminen tieteellisen tutkimuksen mukaisesti. Kananen (2012, 20–21) kuvaa kehittämistyön muodostuvan tieteelliseksi dokumentoinnin kautta sekä käytettäessä tieteellisiä menetelmiä luotettavan, uuden tiedon tuottamiseen.

Kehittämistutkimuksen tarkoituksena on luoda muutos tai parannus johonkin käytännön toimintamalliin, joka ei ole välttämättä yleistettävissä (Kananen, 2012, 43). Tässä työssä on valittu kehitettäväksi toimintamalliksi ohjelmistokehityksen julkaisemisprosessi. Nykytilan, manuaalisen ohjelmiston julkaiseminen, pyritään muuttamaan automatisoiduksi ja luomaan toimintaedellytykset tämän kaltaisen julkaisuympäristön huomioon ottamiselle jo kehitystyön alkumetreiltä lähtien.

Tutkimuskysymys, johon työssä haetaan ratkaisua, on tämän työn puitteissa jaettu kahteen pääkysymykseen. Kysymyksistä ensimmäiselle on työtä tehtäessä annettu hieman enemmän painoarvoa:

1. Kuinka julkaisuputki saadaan toteutettua SC5:n ohjelmistokehityksen puitteissa? Minkälaisia työkaluja ja käytäntöjä tähän liittyy?

2. Miten kehitysympäristöt saadaan jo lähtökohtaisesti rakennettua niin, että niissä huomioidaan ohjelmiston lopullinen sijoitusympäristö? Minkälaisia menetelmiä tässä voidaan käyttää?

Nämä kaksi aluetta eivät toki kata kaikkea jatkuvaan julkaisemiseen tai Devops-kulttuuriin liitettäviä vaiheita vaan toimivat ensimmäisenä askeleena tämän kaltaisten menetelmien kehittämiseen SC5:n puitteissa.

Kananen (2012, 45) määrittelee kehitystutkimuksessa olevan kaksi prosessia. Ensimmäinen näistä on varsinainen kehittämistyö, joka kohdistuu esimerkiksi johonkin prosessiin, tuotteeseen, palveluun tai toimintaan, tässä tapauksessa ohjelmiston julkaisemisen automatisoimiseen. Toisena prosessina tämän rinnalla kulkee varsinainen tutkimus, jonka lopputuloksena tässä tapauksessa syntyy tämä opinnäytetyö. Tämän tärkeänä tehtävänä on paitsi dokumentoida se, mitä on tehty, myös perustella miksi näin on toimittu ja esitellä mahdollisia vaihtoehtoisia tapoja kehittämistyön suorittamiseen.

Luvussa 3 käsitellään yleiskuvan kaltaisesti niitä yksittäisiä osa-alueita, joita jatkuvan julkaisemisen käsitteeseen voidaan liittää. Luku toimii korkean tason katsauksena aiheeseen ja kehitystyön teoriakenttään, eikä sisällä vielä konkreettisia, teknisiä yksityiskohtia.

Luvussa 4 esitellään kehitystyössä käytettäviä työkaluja ja niiden perusominaisuuksia. Työssä esitellään lyhyin, teknisin esimerkein jokaisen työkalun yksinkertainen käyttötapaus, joka saattaa kiinnostuneet lukijat teoriasta käytännön tasolle apuvälineiden käytössä. Lisäksi käsitellään eri virtualisointitapoja sekä kerrotaan julkaisuputken käytännön toteutuksesta Jenkins-ohjelmistoa käyttäen.

Luvussa 5 käydään lävitse varsinainen kehitystyö lähtien julkaisuputken vaiheiden kuvauksesta sekä näiden käyttöönotosta. Tämän jälkeen laajennetaan ohjelmistokehityksen julkaisuputken alkupäätä, kehittäjien työ-ympäristöjä, valmistamalla nämä asianmukaista virtualisointimenetelmää hyväksi käyttäen

Luvut 6 ja 7 keskittyvät saavutettujen tulosten pohtimiseen ja vielä saavuttamattomienkin mahdollisuuksien hahmottamiseen niin SC5:n kehitystyössä kuin laajemmaltikin Devops-menetelmien kehityksen suhteen sekä varovaisesti arvioimaan mahdollista tulevaisuutta.

Aihepiirin terminologia on pääsääntöisesti englanninkielistä, eikä kaikille toiminnoille ole olemassa vielä vakiintunutta, suomenkielistä termiä. Tässä opinnäytetyössä on käytetty tällaisissa tilanteissa parasta vastaavaa käännöstä sekä mainittu selvyyden vuoksi englanninkielinen termi sulkujen sisällä tämän jälkeen. Tämä helpottanee etenkin aiheeseen perehtyneiden ymmärrystä käytännön tasolla terminologian pysyessä tuttuna.

3 Devops-kulttuuri

3.1 Yleiskatsaus

Termi "Devops" juontaa juurensa sarjasta konferensseja, jotka järjestettiin alun perin Belgian Ghentissä vuonna 2009. Siitä lähtien nämä "DevOpsDays"-nimellä tunnetut konferenssit ovat levinneet nopeasti läpi maailman, samoin kuin ajatus Devops-kulttuurista. Termi yhdistää kaksi perinteisesti toisilleen etäistä ryhmää, kehittäjät (eng. developers) ja ylläpitäjät (eng. operations) ja näiden välisen kommunikaation parantamisen pääsääntöisesti automatisoimalla ohjelmiston julkaisuun, ylläpitoon sekä kehitykseen liittyviä haasteita (Hüttermann 2012, 4).

Termi itsessään sisältää monia eri merkityksiä, ja siitä onkin tullut eräänlainen taisteluhuuto modernien ohjelmisto-yritysten viedessä tuotantotapojaan uuteen aikakauteen nopeuttamalla infrastruktuurin pystyttämistä, vakautta sekä automatisointia (Geerling 2015, 5).

Hütterman (2012) hahmottaa termin käsittämään sellaisia käytäntöjä, jotka:

- nopeuttavat ohjelmiston julkaisuprosessia

- edesauttavat virheistä oppimista palautteen kautta
- automatisoivat testaukseen, julkaisuun sekä tarkkailuun liittyviä vaiheita

Devopsin voi myös nähdä *ketteryyden* (eng. lean production) tuomisena ylläpito- ja julkaisupuolelle.

Goldratt (1984) esitteli kirjassaan The Goal teorian rajoitteista (eng. Theory of constraints), jonka mukaisesti tuotantolinjassa esiintyy väistämättä joitakin sellaisia rajoitteita, jotka määräävät kokonaistuotannon kyvyn. Devops-kulttuurin yhteydessä saatettaisiin ajatella ohjelmiston kehitystä perinteisenä tuotantolinjana, jonka alkupäässä on tuote-idea, joka myydään asiakkaalle, ja loppupäässä valmis tuote, joka työnnetään markkinoille. Rajoitusteorian mukaisesti tämän tuotantolinjan sisällä on varmasti löydettävissä joitakin pullonkauloja, jotka ratkaisemalla kokonaistuotanto – valmiin ohjelmiston saattaminen kuluttajien käyttöön saadaan tehokkaammaksi ja nopeammaksi.

Devopsin painopiste asettuu monesti varsinaisen ohjelmistokehityksen sijaan IT-ylläpidon puolelle kuitenkin kattaen ohjelmiston koko teknisen elinkaaren kehityksestä testaamiseen, julkaisuun, monitorointiin ja ylläpitoon (Edwards 2010). Tämä kenties johtuu siitä, että nimenomaan tällä alueella on vielä paljon kehitettäviä toimintatapoja, joita nykyiset ketterän kehityksen menetelmät eivät useinkaan huomioi. Devops ei kuitenkaan ota kantaa myynnin tai markkinoinnin mukaan tuomiseen ohjelmistokehitykseen, vaikka tästäkin saattaisi tietyissä tapauksissa olla lisäarvoa niin asiakkaalle kuin kehittävälle yrityksellekin.

Pohjimmiltaan Devopsissa on kuitenkin kyse rahasta. Uusi ohjelma tai vain uusi ominaisuus olemassa olevassa ohjelmistossa, olipa se kuinka hieno tahansa, ei tuota lisäarvoa asiakkaalle ennen kuin se on julkaistu ja otettu käyttöön (Hütterman 2012, 34–35).

3.2 Ohjelmisto

Ohjelmistolla tässä opinnäytetyössä tarkoitetaan palvelinohjelmiston kautta ajettavaa web-sovellusta, joka saattaa yksinkertaisimmillaan koostua yksittäisestä HTML-tiedostosta. Monimutkaisimmillaan ohjelmisto saattaa rakentua lukuisista, toisiinsa limittyneistä osasista, raskaasta taustajärjestelmästä, tietokannoista, ulkoisista palveluista ja muusta sellaisesta.

Web-ohjelmisto on väistämättömästi kytköksissä *ympäristöön*, jossa ohjelmistoa suoritetaan. Ohjelmisto kommunikoi ympäristönsä kanssa joko suoraan sen tiedostoja ja muita resursseja käyttäen tai ympäristöön asennettujen apuohjelmien sekä -kirjastojen välityksellä. Modernit web-ohjelmistot ovat harvoin itsessään binäärimuotoon käännettyjä tiedostoja, vaan niitä ajetaan erillisen ohjelmiston, kuten node.js:n tai Ruby:n, lävitse. Nämä suorittavat ohjelmistot toimivat web-sovelluksen *vaatimuksina*, jota ilman ohjelmistoa ei kyetä ajamaan.

Ohjelmistojen kehityksessä käytettävät apukirjastot saattavat olla niin ikään vaatimuksia, joiden tulee olla asennettuina. Kehitys- ja tuotantoympäristön vaatimukset tyypillisesti poikkeavat toisistaan: kehitysympäristössä on usein varsinaista kehitystyötä helpottavia työkaluja, joiden avulla lopullinen ohjelmisto valmistetaan tuotantomuotoiseksi. Nykyaikaisessa web-ohjelmistotuotannossa tällaisia ovat esimerkiksi Internet-selainten ymmärtämän Javascriptin tai CSS:n abstraktoiminen Coffeescript- tai LESS- kielien avulla.

Ohjelmistojen käyttämät apukirjastot asennetaan usein jotakin pakettienhallintaohjelmistoa käyttäen. Suosittuja ohjelmia etenkin modernissa web-kehityksessä ovat esim. npm, gem tai bower. Paketteja voidaan asentaa joko yksittäin komentoriviltä tai käyttäen tekstipohjaista asetustiedostoa, jonne määritellään kaikkien haluttavien pakettien nimet ja mahdollisesti versionumerot.

Esimerkiksi kaikki npm-paketinhallintaohjelmiston hallinnoimat ohjelmapaketit voidaan asentaa suoraan asetustiedostosta komennolla:

```
npm install
```

Tämä lukee hakemistossa olevan tiedoston nimeltä *package.json*, joka on tietynlaiseen JSON-muotoon kirjoitettu ohjetiedosto, ja asentaa kaikki tiedostoon määritellyt paketit. Usein on mahdollista määritellä tiedostoon erikseen kehitystyössä tarvittavat sekä varsinaisessa tuotantoympäristössä tarvittavat ohjelmistot.

Ohjelmiston vaatimusten määrittely on oleellinen osa Devops-tyyppistä kehitystyötä, jossa kiinnitetään suurta huomiota lopullisen ympäristön luomiseen ja tämän käyttäytymiseen (Hüttermann 2012, 29-30). Vaatimusten toteuttaminen on usein riippuvaista valitusta loppuympäristöstä, jossa ohjelmistoa suoritetaan, sekä kehittäjien käyttämästä kehitysympäristöstä.

Kehittäjät käyttävät valitettavasti usein kehitystapaa, jossa ohjelmiston käyttämien tietokannan, web-palvelimen sekä kääntäjien, apukirjastojen, testiympäristöjen yms. asentaminen suoritetaan suoraan kehittäjän omalle työkoneelle. Tämä johtaa hyvin moninaisiin, kehitteillä olevan projektin kannalta toisarvoisiin ongelmiin, joiden ratkaiseminen ei edesauta tekeillä olevan tuotteen kehitystä lainkaan, vaan on sille pahimmillaan jopa suureksi rasitteeksi (Hashimoto 2013, 3–4).

Kehittäjien koneilla saattaa olla monia eri projekteja, jolloin näiden yksittäiset kehitysvaatimukset esimerkiksi ohjelmistojen versionumeroiden yhteensovittamisen kanssa käyvät paitsi haastaviksi, myös erittäin aikaa vieväksi sekä alttiiksi virheille. Tämän lisäksi kehitysympäristö on usein aivan toisenlainen kuin ohjelmiston lopullinen palvelinympäristö, joka estää mahdollisten ongelmakohtien havaitsemista tässä vaiheessa kehitystyötä. (Hashimoto 2013, 2–3.)

Kehitettävän ohjelmiston ympäristö vaatii usein myös merkittävästi konfigurointia, erilaisia apuohjelmia sekä testidataa tai muita elementtejä, joiden pystyttäminen etenkin nopeissa kehitysprojekteissa haukkaa helposti suuren osan työajasta. Tämä ongelma korostuu senkaltaisissa projekteissa, joissa työntekijöiden vaihtuvuus on suurta. Hetkellisten apuvoimien pyytäminen vaikkapa yksinkertaisen korjauksen te-

kemiseen käy työlääksi, sillä jokaisen uuden kehittäjän pitää ensin manuaalisesti konfiguroida oma työkoneensa ohjelmiston käynnistämistä varten.

Ohjelmiston ympäristöä onkin hyödyllistä ajatella kokonaisuutena, joka ilmaistaan joitakin ohjelmointikieliä käyttäen. Näin myös ympäristöstä tulee eräs kehitysprojektin artefakteista, jonka valmistuskaava kulkee versionhallinnassa tasavertaisena muun ohjelmistokoodin rinnalla (Hüttermann 2012, 83).

Yksittäisten kehittäjien käyttämä kehitysympäristö saatetaan rakentaa tätä valmistuskaavaa hyväksi käyttäen, jolloin paitsi vältetään kehittäjien tekemään manuaalista työtä, myös rakennetaan kuin huomaamatta lopulliseen julkaisuun tähtäävää asennusohjetta, joka kulkee versionhallintaohjelmistossa muun ohjelmakoodin mukana. Tämän kaltainen lähestymistapa myös tukee tiiminsisäistä ymmärrystä siitä, minkälainen ohjelmiston ympäristö tulee lopulta olemaan, ja mahdollistaa potentiaalisten ongelmien havaitsemisen jo kehitysvaiheessa. (Hüttermann 2012, 82-83).

Nykyaikaista ohjelmistokehitystä onkin lähes mahdotonta kuvitella ilman versionhallintaohjelmia. Versionhallintaohjelmisto mahdollistaa usean henkilön yhtäaikaisen kehitystyön ja toimii itsessään läpinäkyvänä dokumentointina ohjelmiston historiaan. Se myös auttaa hallinnoimaan monimutkaisimpiakin kehityskohteita siten, että ohjelmiston ydinkehityslinja kyetään pitämään jatkuvasti vakaana ja julkaistavissa samanaikaisesti, kun siihen kehitetään raskaita muutoksia jossakin toisessa kehityslinjassa.

Devops-toimintakulttuurissa ohjelmiston versionhallinta nousee jopa vielä suurempaan arvoon kuin perinteisessä ohjelmistokehityksessä. Pelkän ohjelmistokoodin lisäksi Humble ja Farley (2011, 33) suosittelevat tallentamaan kaikki projektiin liittyvät tiedostot versionhallintaan, olivatpa nämä sitten testejä, yleistä dokumentaatiota, tietokanta-skriptejä, rakennus- ja julkaisutietoja, asetustiedostoja tai jotakin muuta sen kaltaista tietoa, jota ohjelmisto tarvitsee toimiakseen.

Siirryttäessä jatkuvaan julkaisemiseen ja automatisoituun palvelinympäristöjen luomiseen on versionhallinnassa luontevaa säilyttää myös kaikki tällaiset kehitys-, testi-

ja tuotantopalvelimen konfiguraatiot ja asennusohjeet. Projektin laajuudesta riippuen voidaan versionhallinnassa säilyttää myös projektia ympäröivän infrastruktuurin asetustiedostot, kuten DNS-tietueet sekä kuormantasaukseen, tietokantapalvelimiin tai toimintavarmuuden takaamiseen liittyvien palvelujen pystyttämiseen vaadittavat asetukset ja tiedot (Humble & Farley 2011, 33).

Koko ohjelmiston ja tämän suoritusympäristön saattaminen versionhallintaan saattaa äkkiseltään tuntua liioittelulta, mutta tarkoituksena tässä on saada kaikki sellainen informaatio, joka projektin kehityksen aikana saattaa muuttua, hallittuun säilytyspaikkaan. Tämä mahdollistaa mm. ohjelmiston palauttamisen määrätystä versiosta kokonaisuutena kehitys- ja tuotantopalvelinasetuksia myöten, eikä pelkästään lähdekoodina. Koska julkaisukaava kulkee versionhallinnassa, on minkä tahansa menneen version julkaiseminen mahdollista valitsemalla versionhallinnasta haluttu aika ja ajamalla julkaisutiedostot. (Humble & Farley 2011, 33)

Haasteita virtuaaliympäristöjen tuomisessa kehittäjien työpöydille tosin riittää. Kehittäjien ja ylläpitäjien toimintaympäristöt ovat perinteisesti olleet kaukana toisistaan, ja ohjelmiston suoritysympäristönä yleisesti käytetty Linux-käyttöjärjestelmä on yhä monelle kehittäjälle vieras, puhumattakaan erityisistä palvelinohjelmistoista tai tekstipohjaisesta asetusten konfiguroinnista. Osa vanhemmista kehittäjistä on saattanut kokeilla virtualisointityökaluja aikana, jolloin ne vielä kärsivät suuresti lastentaudeista, ja on näiden seurauksena hylännyt nämä ohjelmistot "tarpeettomina".

Kulttuurin muuttaminen ei käy tämänkään asian suhteen nopeasti, mutta kehitysympäristön virtualisoimiseen ja tämän konfiguraation automatisoimiseen on niin monta hyvää syytä, että sen tuominen kiinteäksi osaksi jokaista ohjelmistokehitysprojektiksi on perusteltua.

3.3 Automatisoinnin merkitys

Devops-kulttuurin keskeisimpiä käsitteitä on termi *jatkuva julkaisu*. Tällä tarkoitetaan ohjelmistokehityksen aikaista, hyvin useasti toistuvaa valmiin tuotteen koostamista

uusimpine muutoksineen ja tämän mahdollista asentamista joko testi- tai jopa tuotantoympäristöön automaattisesti. Tämän kaltaisesta toimintamallista saadaan merkittävästi hyötyä. Mitä pienempi muutos ohjelmistoon tehdään, sitä nopeammin ja tarkemmin mahdolliset ongelmat pystytään havaitsemaan, ja sitä pienempi on myös muutetun tuotteen loppukäyttäjille julkaisemisen riski. (Ries 2010, 49–52.)

Nopea julkaisutehokkuus mahdollistaa myös loppukäyttäjien paremman osallistumisen ohjelmiston kehitykseen ja lyhentää asiakkaan sekä kehittäjien välimatkaa. Asiakas pystyy välittömästi näkemään ne muutokset, joita ohjelmistoon on tehty, antamaan palautetta niistä, ja kehittäjä kykenee välittömästi ymmärtämään, mitä asiakas todella ohjelmistoon haluaa. Tämä edesauttaa senkaltaista kommunikaatiota, jota usein pidetään ketterän kehityksen edellytyksenä. (Fowler 2006.)

Englanninkielisessä termistössä jatkuvasta julkaisusta puhuttaessa törmää muutamaan samankaltaiseen käsitteeseen, jotka on syytä tässä erottaa toisistaan:

Jatkuva integrointi (eng. continuous integration) on termeistä vanhin ja tarkoittaa kehittäjien tekemien muutosten välitöntä integroimista ohjelmistokokonaisuuteen. Erillinen testipalvelin tarkkailee versionhallintaan saapuvia muutoksia ja suorittaa näiden muutosten testauksen annettujen yksikkö- sekä (automaattisten) hyväksyntätestien avulla antaen kehittäjille palautteena tiedon testien onnistumisesta mahdollisimman nopeasti. (Humble & Farley 2011, 55–56.)

Jatkuva tuotantovalmius (eng. continuous delivery) vie jatkuvaa integrointia askeleen pidemmälle ja asentaa onnistuneiden testien jälkeen ohjelmiston koekäyttöpalvelimelle (eng. staging server), josta laaduntarkkailijat tai muu rajoitettu käyttäjäryhmä pääsee konkreettisesti käyttämään tuotetta ja suorittamaan haluttuja hyväksyntätestejä. Jatkuva tuotantovalmius sananmukaisesti tarkoittaa, että jokainen testeistä läpi päässyt versio ohjelmistosta voidaan potentiaalisesti julkaista tuotantopalvelimella välittömästi, mikäli niin halutaan. (Humble & Farley 2011, 345.)

Jatkuvassa tuotantoon viennissä (eng. continuous deployment) edellinen malli viedään jälleen hieman pidemmälle, ja myös kaikki koekäyttöpalvelimella ajettavat tes-

tit automatisoidaan. Näiden onnistuessa testipalvelin toteaa kaiken olevan kunnossa ja asentaa ohjelmiston välittömästi myös tuotantopalvelimelle. (Humble & Farley 2011, 266–267.)

Ratkaisun tekninen toteutus jatkuvassa tuotantovalmiudessa ja jatkuvassa tuotantoon viennissä pysyy samankaltaisena, ja suurin poikkeavuus näiden välillä onkin poliittinen: ensimmäisessä mallissa päätösvalta julkaisusta luovutetaan liiketoiminnan harkinnalle, jälkimmäisessä se on kehittäjillä ja ylläpitäjillä (Smart 2011, 2).

Jatkuva tuotantoon vieminen ohjelmistokehityksen mallina vaatii suunnatonta luottamusta laaduntarkkailun onnistumiseen, ja onkin hyvä idea rakentaa nopea valmius palata takaisin edelliseen, toimivaan versioon mikäli tuotantoympäristössä havaitaan vakavia ongelmia (Fowler 2006). Hütterman (2012, 40) tosin huomauttaa että tehokkaan julkaisuputken myötä usein saattaa olla tehokkaampaa julkaista bugiin korjaus ja päivittää seuraava versio nopeasti (eng. forward rolling change) kuin punnertaa edellistä versiota uudelleen pystyyn.

Humble ja Farley (2011, 6) korostavat automatisoinnin merkitystä jatkuvan julkaisun edellytyksenä. Julkaisemisen automatisoinnilla saavutettavia etuja ovat mm:

- **Toistettavuus** ja **luotettavuus**: automatisoitu skripti suorittaa työvaiheet tismalleen samalla tavalla joka suorituskerralla, väsymättä ja tekemättä kirjoitusvirheitä
- **Automaattinen dokumentointi**: koska automatiikka pohjautuu eksplisiittisesti kirjoitettuihin välivaiheisiin, jotka ovat kaikkien kehittäjien saatavilla ja muokattavissa, pystyy jokainen yksittäinen kehittäjä ymmärtämään täsmällisesti minkälaisia toimenpiteitä tuotantoympäristöön on tehtävä.
- **Yhteistyöhön kannustaminen**: kaikki muutokset ohjelmiston julkaisemiseen tehdään julkisesti asetustiedostoja käyttäen, jolloin näiden muutokset sekä perustelut ovat kaikkien nähtävillä ja kommentoitavissa
- **Asennuksen voi suorittaa kuka tahansa**: tieto ohjelmiston julkaisemisesta ei lepää yhden ylläpitäjän vastuulla, joka kutsutaan vaikka kesälomaltakin työpaikalle kun jokin menee vikaan.

- **Ajansäästö**: julkaisu tapahtuu automaattisesti, jolloin kehittäjät ja ylläpitäjät voivat keskittyä tekemään muuta, tuottavaa työtä sen sijaan että kuluttaisivat tunteja, pahimmillaan jopa päiviä ohjelmiston julkaisemiseen manuaalisesti

Humble ja Farley (2011, 7) kehottavat käyttämään samaa julkaisuskriptiä niin kehitys- kuin tuotantoympäristönkin asentamiseen, jolloin ensimmäisen tuotantojulkaisun saapuessa julkaisuautomatiikka on suoritettu jo mahdollisesti satoja kertoja kehitysympäristöä vasten, ja tämän aikana esiin tulleet ongelmakohdat on ehditty havaita sekä korjata. Automatisoinnin todellinen päämäärä onkin luoda julkaisusta tylsää, yllätyksetöntä ja läpinäkyvää sen sijaan että uuden version julkaisu olisi yhteinen tapahtuma, jonka jälkimainingeissa, kenties aamuyön hiljaisina tunteina ohjelmisto on viimeinkin saatu toimimaan oikein.

3.4 Julkaisuputki

Jatkuvan julkaisemisen edellytyksenä on pitkälle automatisoitu julkaisuputki (eng. deployment pipeline), jossa ohjelmisto kuljetetaan useiden, erillisten askelten lävitse matkalla kohti lopullista sijoituspaikkaa. Jokainen muutos ohjelmiston konfiguraatioon, lähdekoodiin, ympäristöön tai dataan aloittaa uuden instanssin julkaisuputken läpi kulkemiseksi (Humble & Farley 2011, 4).

Julkaisuputken aikana ohjelmistolle luodaan annettujen ohjeiden mukaisesti virtuaalinen ympäristö, jonne haluttu versio ohjelmistosta asennetaan. Tästä virtuaaliympäristöstä pyritään luomaan mahdollisimman paljon lopullisen tuotantoympäristön kaltainen, jolloin mahdollisuus lopullisessa ympäristössä esiin tulevien ongelmien löytämiseksi kasvaa merkittävästi. (Humble & Farley 2011, 279.)

Kun ohjelmisto on asennettu virtuaaliympäristöön, suorittaa testipalvelin ohjelmistolle määritellyt yksikkö- sekä mahdollisesti myös automaattiset hyväksyntätestit. Välittömästi testien ajamisen jälkeen palvelinohjelmisto lähettää kehittäjille palautteena testien onnistumisesta. Näin palautekierros pysyy hyvin lyhyenä, eikä testien ajamiseen ja palautteen saamiseen optimaalisessa tilanteessa kulu kuin joitakin minuutteja siitä, kun kehittäjä on tallentanut muutoksensa versionhallintaan. Tämä

varmistaa että tehdyt muutokset ovat vielä tuoreena kehittäjän mielessä, jolloin niiden aiheuttamat ongelmat on mahdollisimman helppoa korjata. (Humble & Farley 2011, 13.)

Kaikkineen ohjelmisto kulkee julkaisuputkessa askel askeleelta erilaisten vaiheiden lävitse, joilla varmistetaan ja todennetaan lopullisen tuotteen julkaisukelpoisuus. Jokainen onnistuneesti läpi kuljettu testivaihe lisää kaikkien sidosryhmien luottoa siihen, että kyseinen ohjelmistokoodin, ympäristön, datan sekä asetusten yhdistelmä toimii moitteetta myös julkaisuympäristössä. (Humble & Farley 2011, 4.) Kaikista julkaisuputken vaiheista onkin syytä tehdä mahdollisimman läpinäkyviä, jotta jokainen kehitystyöhön osallistuja kykenee ymmärtää ohjelmiston kehitystilan.

Laadunvalvonnan tarkkuudesta riippuen halutut testit on mahdollista suorittaa jopa ennen kuin muutoksia varsinaisesti tallennetaan versionhallintaan. Tämä toimii "tuotantolinjan pysäytyksenä" ja lähettää kehittäjälle välittömän palautteen koodin sisältämistä ongelmista samalla varmistaen ettei rikkinäisiä muutoksia pääse kulkeutumaan eteenpäin. (Ries 2010, 57–58.)

Julkaisuputken lopullinen tavoite onkin kolmitahoinen (Humble & Farley 2011, 4):

- Julkaisuputki saattaa kaikki ohjelmiston rakennuksen, julkaisemisen sekä testaamisen osa-alueet näkyville kaikille kehitystyöhön osallistuville.
- Julkaisuputki parantaa palautteen saamista, sillä ongelmat havaitaan välittömästi ja ne kyetään korjaamaan mahdollisimman aikaisessa vaiheessa niiden ilmestymisestä.
- Julkaisuputki mahdollistaa kehitteillä olevan ohjelmiston julkaisemisen nopeasti ja automaattisesti mihin tahansa ympäristöön

Kun ohjelmisto on kuljetettu onnistuneesti julkaisuputken lävitse, on viimeisenä askeleena luontevasti tuotteen julkaiseminen. Tähän on useita eri tapoja, jotka riippuvat merkittävästi loppuympäristöstä ja tämän rajoituksista, vaatimuksista sekä halutuista infrastruktuurin elementeistä. Usein ohjelmistokehityksen tilaajalla on jo oma järjestelmänsä, johon kehitystiimin tulee tuote valmistaa.

Yksinkertaisimmillaan julkaiseminen saattaa tapahtua pelkällä verkkoyhteyden ylitse kopioidulla tiedostokansiolla esimerkiksi scp-ohjelmaa käyttäen. Julkaisuputken aikana valmistettu, testit läpi käynyt ohjelmiston lopputuotos kopioidaan suoraan palvelimelle, kenties versionumerolla varustettuun kansioon ja kerrotaan palvelinohjelmistolle missä haluttu versio sijaitsee. (Ries 2010, 58.) Tässä menetelmässä oletetaan palvelinympäristön olevan jo valmiina suoritusta varten sisältäen kaikki ohjelmiston vaatimukset valmiina käytettäväksi, eikä niitä asenneta enää uudestaan jokaisen julkaisun yhteydessä.

Rakennettaessa kestävämpää ja joustavampaa ohjelmiston julkaisujärjestelmää, on julkaisuympäristön käytössä olevien standardien hyödyntäminen usein hyvä ajatus. Tällöin hyödynnetään jo testattuja, toimivaksi havaittuja käytänteitä sen sijaan, että näitä ryhdyttäisiin projektikohtaisesti luomaan tyhjästä. (Hüttermann 2012, 126.)

Linux-järjestelmien eräs perustavimmista rakennuspalikoista on ohjelmistopakettien hallinta. Esimerkiksi Debian-pohjaisten jakeluiden ".deb"-paketit ja Redhat-jakeluiden ".rpm"-paketit ovat soveltuvia myös web-sovelluksen julkaisemiseen ja sisältävät itsessään tiedon mm. kaikista ohjelmiston tarvitsemista riippuvaisuuksista. (Hüttermann 2012, 125–126.)

Ohjelmistopaketin luominen kyetään helposti automatisoimaan julkaisuputken avulla. Versionhallintaan tullut muutos käynnistää jokaisella kerralla julkaisuputken uuden läpikulun. Yksittäisenä välivaiheena voidaan tällöin lisätä halutun muotoisen paketin rakentaminen ohjelmiston jokaisesta saapuvasta versiosta. Paketti lähetetään kääntämisen jälkeen binääri-repositorioon, josta se ladataan palvelimelle käyttöjärjestelmän omaa paketinhallintaohjelmistoa käyttäen. Myös paketin lataaminen palvelimelle voidaan liittää osaksi julkaisuputkea, joko manuaalisen hyväksymistestauksen taakse tai automatisoidusti. (Ries 2010, 56–57.)

Nykyaikaiset virtualisointimenetelmät ovat erinomaisen käytettäviä julkaisuputken aikana. Tällaisia virtuaalisia ympäristöjä saadaan nopeasti luotua sekä hävitettyä, mikä mahdollistaa ohjelmiston asentamisen lopullista sijoittamista vastaavaan ympäris-

töön, ohjelmiston testaamisen tässä ympäristössä sekä suorituksen loputtua ympäristön hävittämisen. Virtualisoidut ympäristöt voidaan luoda ohjelmiston versionhallintaan tallennetuilla "valmistusohjeilla", joita luodaan ja päivitetään samanaikaisesti kuin ohjelmistoakin kehitetään (Hüttermann 2012, 135–136).

Mikäli ohjelmiston loppusijoitus tapahtuu virtuaaliseen ympäristöön, joka on kehittäjien rajattavissa, saadaan jo testivaiheessa kosolti ymmärrystä tämän ympäristön tarpeista ja ongelmista, jolloin tuotantoympäristön mahdollisiin puutteisiin voidaan paneutua jo ennen kuin koko ohjelmistoa on edes julkaistu. Myös joitakin virtuaaliympäristöjä voidaan siirrellä binääri-repositorioiden kautta, jolloin palvelimen resursseja ei tarvitse käyttää virtuaaliympäristöjen rakentamiseen, vaan tämä voidaan suorittaa jo valmiiksi rakennuspalvelimella.

3.5 Palvelin-infrastruktuuri

Perinteisiin, manuaalisesti ylläpidettyihin palvelimiin kasvaa ajan myötä väistämättäkin hyvää tarkoittavia, mutta dokumentoimattomia korjauksia: kenties yksi muutos asetustiedostoon tuolla, kukaties yksi manuaalinen välimuistin tyhjennys täällä. Ehkä ohjelmisto ei toimikaan oikein enää uusimpien turvallisuuspäivitysten jälkeen, ja ongelma on ratkaistavissa yhdellä pienellä korjauksella – ylläpitäjä pelastaa jälleen päivän.

Tämän kaltaiset nopeat voitot ja pikakorjaukset sumentavat helposti ylläpitäjän ajatuksissa palvelin-infrastruktuurin laajemman kokonaiskuvan, josta lopulta seuraa palvelinten tilan karkaaminen tunnetusta tuntemattomaan (eng. configuration drift). Tähän ajautumiseen on syytä puuttua tuomalla palvelinten asennusten ja konfiguroinnin automatisointi kiinteäksi osaksi ympäristön ylläpitäjien rutiineja.

Palvelinympäristöt, joilla ohjelmistoja ajetaan, ovat nykypäivänä yhä useammin virtuaalisia, jolloin uusien palvelinten käyttöönotto ja "fyysisten" ominaisuuksien (kuten prosessoritehon tai keskusmuistin suuruuden) muokkaaminen on vaivatonta ja nopeaa. Nykyiset palveluntarjoajat mahdollistavat helposti juuri halutuilla ominaisuuksilla varustetun yksittäisen palvelimen pystyttämisen jopa minuuteissa. Tällöin voi-

daan varmistua siitä, että ohjelmisto asennetaan puhtaalle alustalle – palvelimelle, jossa ei ole minkäänlaisia "poikkeuksia" tai ylimääräisiä rajoitteita, jotka saattaisivat vaikuttaa ohjelmiston toimivuuteen.

Yksittäisten virtuaalipalvelinten jatkuvalle pystyttämiselle vaihtoehtona on pitkäaikaisen, pysyvän palvelimen käyttäminen, jonka sisällä ohjelmistot ajetaan virtuaalisissa ympäristöissä. Näin jokaisella ohjelmistolla on oma, eristetty tilansa palvelimen sisällä, jossa saattavat olla täysin erilaiset asetukset, jopa eri käyttöjärjestelmä kuin varsinaisella isäntäkoneella, jonka tärkeimpänä tehtävänä on huolehtia näiden virtuaaliympäristöjen käynnissä pysymisestä.

Valittiinpa kumpi tahansa yllä kuvatuista vaihtoehdoista palvelinarkkitehtuuriksi, on molempien perusajatus sama: ohjelmistolle tarjotaan tyhjä järjestelmä, johon tämä voidaan asentaa ilman huolta siitä, että jokin edellinen pikakorjaus tai inhimillinen erehdys haittaa ohjelmiston suorittamista. Tyhjällä järjestelmällä tässä tarkoitetaan senkaltaista palvelinympäristöä, jonne on asennettu vain tuoreen käyttöjärjestelmäasennuksen mukanaan tuomat ohjelmistot.

Virtualisoinnin myötä ylläpitopuolella käytetäänkin termiä *muuntumaton infrastruktuuri* (eng. immutable infrastructure), jolla tarkoitetaan sananmukaisesti ohjelmiston ympäristöä, joka luodaan kerran, ja jota käytetään kunnes se hylätään ja sen tilalle rakennetaan uusi. Näin esimerkiksi käyttöjärjestelmään tulevia turvallisuuspäivityksiä ei asenneta olemassa olevaan ympäristöön, vaan koko ympäristö luodaan tarvittaessa uudestaan. (Butler-Cole 2013.)

Asennettavan ohjelmiston ja sen tarvitsemien riippuvaisuuksien, asetustiedostojen ja käyttäjädatan saattaminen palvelimelle suoritetaan palvelimen (tai virtuaaliympäristöön) luonnin yhteydessä automaattisesti usein jotakin konfiguraationhallintaohjelmistoa apuna käyttäen (Humble & Farley 2011, 37–39).

Mikäli kehitys-, testi- ja tuotantopalvelimen kaikki asetukset, ohjelmistot, versiot ja muut tarvittavat määritelmät ovat kelvollisesti säilytettynä versionhallinnassa, tapahtuu uuden palvelimen asentaminen vain näiden tiedostojen avulla. Tällöin voidaan

varmistua siitä, että ympäristö on tismalleen halutun kaltainen ja että kuka tahansa kehittäjistä kykenee halutessaan luomaan uuden palvelinympäristön ilman ylläpitopuolen asiantuntemusta. (Humble ja Farley, 2011, 10.) Tämä myös mahdollistaa minkä tahansa aiemman version asentamisen vain versionhallinnan historiaa käyttämällä, sillä palvelinasennusten ohjeistus kehittyy muun ohjelmiston mukana ja on parhaimmillaan aina asennettavissa, kuljettiinpa versionhallinnan historiassa kuinka paljon takaisinpäin tahansa.

Adam Jacob (2010, 69) määrittelee konfiguraationhallinnan tarkoittamaan niiden toimintojen ylös kirjaamista, joilla tyhjästä palvelimesta (Jacob käyttää tässä englanninkielistä termiä "bare metal") saadaan palvelin, joka suorittaa jonkin työn. Hän kuvaa primitiivisimmän konfiguraationhallinnan ylläpitäjänä, joka asentaa ja konfiguroi palvelimelle halutut sovellukset manuaalisesti ja kirjoittaa työvaiheensa muistiin.

Motivaatio konfiguraationhallintaan on lähes yhtä vanha kuin ylläpitäjän ammatti. Vuosien saatossa manuaaliset terminaalikomennot ovat vaihtuneet milloin milläkin ohjelmointikielellä kirjoitetuiksi skripteiksi, jotka vähentävät inhimillisten virheiden lukumäärää, ja yhä edelleen korkeamman tason kuvaileviksi "ohjeiksi". Monet konfiguraationhallintaohjelmistot (kuten Chef tai Puppet) nojaavat Jacobin (2010, 73–74) mukaan yhä nykypäivänäkin Mark Burgessin (Cfenginen kehittäjä) 1990-luvun alussa määrittämiin konfiguraationhallintaohjelmiston peruspilareihin. Jacob (2010, 73-74) kuvaa Burgessin määrittelemät ohjelmiston vaatimukset seuraavasti:

Kuvaileva: Sen sijaan, että ohjelmalle kerrotaan kuinka jokin asia tulee tehdä, määritetään mitä halutaan tapahtuvaksi. Esimerkiksi sen sijaan, että annettaisiin suora asennuskomento ”apt-get install sudo”, kerrotaan ohjelmistolle korkean tason ohjelmointikielellä ”action=install, package=sudo”

Abstrakti: Ohjelmiston tulee itsenäisesti ratkaista korkean tason kuvailun perusteella, kuinka haluttu tila saavutetaan. Kuvaus tulkitaan järjestelmän puitteissa, kuten taulukossa 1 näytetään.

Taulukko 1. Esimerkki kuvailevien käskylauseiden tulkinnasta

Kuvailu	Järjestelmä	Tulkinta
install sudo	Debian	apt-get install sudo
Install sudo	RedHat	yum install sudo

Idempotentti: Ohjelmisto suorittaa toimenpiteitä vain siinä tapauksessa, ettei tila vielä ole halutun kaltainen. Mikäli esim. haluttu paketti on jo asennettu järjestelmään, sivuuttaa konfiguraationhallintaohjelmisto tämän lauseen.

Kokoava: Järjestelmän kokonaistila koostuu yksittäisistä, toisistaan riippumattomista tilanmuutoksista. Näitä voivat olla esimerkiksi määrätyn ohjelmiston asentaminen, konfiguraatiotiedoston muuttaminen tai uuden käyttäjän luominen.

Useimmat konfiguraationhallintaohjelmistot taipuvat mainiosti niin suuriin palvelinasennuksiin kuin yksittäisen kehittäjän työasemalla tapahtuvaan virtuaaliympäristön luomiseenkin. Tämän kaltaisten työkalujen käyttäminen onkin usein hyvin tehokasta jatkuvaan julkaisuun pyrkivää järjestelmää rakennettaessa. Ylläpidollisten komentojen abstraktoinnin vuoksi näiden työkalujen oppiminen ja käyttö on myös helpompaa kuin jokaisen yksittäisen palvelinympäristön komentojen opetteleminen erikseen.

Palvelinratkaisut, joissa ohjelmisto on jaettu virtuaalisiin, eristettyihin ympäristöihin tarjoaa runsaasti mahdollisuuksia niin ohjelmiston toiminnan hajauttamiseen kuin myös useiden toisiensa kanssa yhteen sopimattomien ohjelmistojen suorittamiseenkin samalla palvelimella. Tämä mahdollistaa myös niin kutsutun mikropalvelumallisen ohjelmistokehityksen. Tässä mallissa yksittäinen ohjelmisto koostuu useista erillisistä osista, komponenteista, jotka kommunikoivat keskenään vain verkkoprotokollia käyttäen. Tällöin yksittäiset komponentit saatetaan ohjelmoida täysin erilaisiin, keskenään yhteen sopimattomiin ympäristöihin, vaikkapa eri ohjelmointikielillä, eivätkä irralliset komponentit tiedä toistensa sisäisestä toiminnasta mitään. (Fowler & Lewis 2014.)

Eräs mielenkiintoinen kysymys palvelimien ylläpitäjien kannalta onkin monien erilaisten virtuaaliympäristöjen sekä palvelin-infrastruktuurin yhteen sovittaminen. Vaikka yksittäisten ohjelmistojen tai komponenttien konfigurointi sekä näiden suoritusympäristöt luodaan virtuaaliympäristöjen sisälle, eivät konfiguraationhallintatyökalut toki jää tarpeettomaksi: virtuaaliympäristöjen suorittamisen mahdollistavan ohjelmiston on silti oltava asennettuna isäntäkoneelle ja palvelinta on kaikesta huolimatta edelleen huollettava sekä virtuaaliympäristöjä käynnisteltävä ja päivitettävä.

4 Käytettävät työkalut

4.1 Työkalujen valinta

Mahdollistaaksemme jatkuvan julkaisemisen käytännössä joudumme alkajaisiksi valitsemaan työhön sopivimmat työkalut. Kuten jo luvussa 2 todettiin, on lähes jokaiseen ongelmakentän osa-alueeseen löydettävissä useita, toimivia vaihtoehtoja, jotka eroavat toisistaan lähinnä käyttötavan ja syntaksin perusteella.

Tässä työssä tullaan rakentamaan julkaisuputki avoimen lähdekoodin ohjelmistoja käyttäen ja kehityspalvelin-ohjelmistoksi on valittu SC5:ssä jo käytössä oleva Jenkins. Vaihtoehtoja ohjelmistolle olisi ollut hyvin paljon aina kolmannen osapuolen ylläpitämistä SaaS-palveluista (esim. Travis CI) kaupallisiin tuotteisiin, kuten TeamCity tai CruiseControl.

SC5:n tarpeiden mukaisesti julkaistavan ohjelmiston pakkausmenetelmäksi on valittu Docker-virtuaaliympäristöt muiden vaihtoehtojen sijaan, sillä tämä soveltuu erityisen hyvin web-ohjelmistojen nopeaan julkaisemiseen. Vaihtoehtoisia tapoja julkaista ohjelmisto on tarjolla hyvin monia, ja tyypillisimmillään jatkuvan julkaisun ympäristöissä saatettaisiin käyttää Linux-järjestelmäpaketteja tai kokonaisten palvelinympäristöjen pystyttämistä yksinomaan julkaistavan ohjelmiston käyttöön.

Tämän opinnäytetyön käytännön osuuden luonteesta johtuen ei oteta kantaa lopulliseen palvelinympäristöön, jossa ohjelmaa suoritetaan. Tämä vapauttaa lukkiutumas-

ta mihinkään julkaisumuotoon, mutta toisaalta myös estää aiheen syvemmän tarkastelemisen käytännön rajoittuessa kehityspalvelinten tasolle.

Kehittäjätiimistä ja käytettävistä työkaluista riippuen kehitysympäristön pystyttämisen voi tehdä monin eri tavoin, mutta tämän tutkimuksen lähtökohdaksi on valittu virtualisointi Oracle Virtualboxia sekä Vagrantia käyttäen. Virtualbox toimii tässä varsinaisena virtualisoinnin toteuttajana ja Vagrant nopeana käyttöliittymänä tälle. Muita mahdollisuuksia olisi ollut käyttää esim. suoraan Docker-virtuaalikontteja tai muita virtualisointitapoja ilman Vagrantin avustusta.

Opinnäytetyössä esitetään myös virtuaaliympäristön provisiointi eli vaatimusten sekä ohjelmiston automaattinen asentaminen virtualisoituun kehitysympäristöön. Tässä käytetään SC5:n puitteissa hyväksi havaittua Ansible-konfiguraationhallintaohjelmaa, mutta yhtä hyvin olisi voitu käyttää Puppetia, Chefiä, Saltstackia tai yksinkertaisimmillaan jopa Bash-kielellä kirjoitettua skriptiä. Kyse tämän työkalun valinnassa onkin pitkälti mielipiteestä sekä tottumuksesta, sillä kaikki edellä mainitut kykenevät suorittamaan tarvitut toiminnot helpohkosti.

4.2 Vagrant

Vagrant on eräs käytetyimpiä ratkaisuja virtuaaliympäristöjen pystyttämiseen. Se tarjoaa nopean ja automatisoidun tavan kokonaisten kehitysympäristöjen rakentamiseen omiin, eristettyihin virtuaaliympäristöihin. Vagrant vähentää merkittävästi kehittäjiltä kuluvaa aikaa kehitteillä olevan ohjelmiston pystyttämiseen, lisää kehitys- sekä tuotantoympäristöjen yhteneväisyyttä sekä tuo kertakäyttöisten ympäristöjen ajatuksen palvelimilta yksittäisen kehittäjän tasolle. (Hashimoto 2013, 1.)

Vagrantille ei oikeastaan ole tällä hetkellä varteenotettavaa kilpailijaa. Vaihtoehtona sen käytölle voidaan kuitenkin kuvitella esim. Docker-virtuaalikontti, joka valmistetaan Dockerfileä käyttäen.

Vagrantin kehittäjä Mitchell Hashimoto (2013, 4) kuvaa Vagrantin hengen upeasti:

> ”Vagrantin maailmassa kehittäjät voivat ladata minkä tahansa repositorion versionhallinnasta, ajaa `vagrant up` -komennon ja saada täydellisesti toimivan kehitysympäristön ilman minkäänlaista ihmisen kosketusta välissä. Kehittäjät jatkavat työskentelyä omilla työkoneillaan, omien tekstieditoriensa, selaintensa ja muiden työkalujensa tarjoamassa mukavuudessa. Vagrantin olemassaolo on näkymätöntä ja epäoleellista kehittäjän mielenmaisemassa. Vagrant on se hiljainen työntekijä, joka luo kehittäjälle säännönmukaisen ja vakaan kehitysympäristön.”

Vagrantin virtuaaliympäristön runkona toimii esivalmistettu virtuaalinen kuva, joka on pakattu Vagrantin omaan “.box”-päätteiseen formaattiin. Pohjakuvista puhutaankin Vagrantin terminologiassa usein laatikkoina (eng. box). Vagrant tarjoaa suoraan käytettäväksi joitakin käyttöjärjestelmäasennuksia, mutta myös omien ympäristöjen luominen on mahdollista. Tämä on hyödyllistä etenkin silloin, kun lopullinen palvelunympäristö poikkeaa suuresti valmiiksi tarjotuista.

Vagrant konfiguroidaan aina projektikohtaisesti, ja jokainen projekti sisältää vähintäänkin yhden virtualisoidun ympäristön. Vagrant-projektin tunnusomaisena piirteenä on kansiossa sijaitseva konfiguraatiotiedosto nimeltä Vagrantfile, yksinkertainen tekstitiedosto, jonka avulla Vagrant luo halutut ympäristöt ohjelmistolle. Vagrantfilessä voidaan kuvata haluttu virtuaalinen laatikko, virtuaaliympäristön "fyysiset" ominaisuudet (kuten annetun keskusmuistin määrä), mitä ohjelmistoja asennetaan yms. (Hashimoto 2013, 17.)

Vagrantfile on itseasiassa kooditiedosto, joka on kirjoitettu Ruby-kielellä. Tämä mahdollistaa monimutkaisempienkin konfiguraatioiden tekemisen, joskin käytännössä tämä on usein tarpeetonta. (Hashimoto 2013, 18.)

Vagrantfilessä virtuaalisen laatikon lataaminen tapahtuu seuraavasti:

```
config.vm.box = "ubuntu/trusty64"
```

Pohjakuvan lisäksi Vagrantfilessä on usein hyödyllistä jakaa ohjelmistokoodia sisältävä hakemisto isäntäkoneelta virtuaaliseen ympäristöön. Mikäli haluaisimme jakaa esimerkiksi isäntäkoneen hakemiston nimeltä “./src/” virtuaaliympäristön hakemistoksi “/app”, voimme kirjoittaa tiedostoon komentorivin:

```
config.vm.synced_folder "./src", "/app”
```

Jaettujen hakemistojen sisältämien tiedostojen lukeminen sekä kirjoittaminen on yhtäaikaisesti mahdollista niin isäntä- kuin virtuaalikoneeltakin, ja muutokset päivittyvät välittömästi molempiin järjestelmiin. Myös jaettujen kansioiden virtualisoituun tiedostojärjestelmään voi vaikuttaa määrittämällä sen esim. NFS-tyyppiseksi.

Pelkkä virtuaaliympäristön pystyttäminen on askel oikeaan suuntaan, mutta ollakseen kehitystyölle hyödyllinen on meidän asennettava kehitteillä oleva ohjelmisto sekä tämän tarvitsemat riippuvaisuudet virtuaaliympäristöön. Koska tavoitteemme on automatisoida virtuaaliympäristön pystyttäminen mahdollisimman pitkälle, emme hyväksy menetelmää jossa kehittäjät joutuvat suorittamaan manuaalisia komentoja pakettien asentamiseen tai konfiguroimiseen.

Vagrant tarjoaa mahdollisuuden leipoa tarvittavat riippuvaisuudet ja ohjelmistot kiinteästi suoraan pohjakuvana käytettävään "laatikkoon", mutta tämän kaltainen menetelmä käy helposti liian raskaaksi eri projektien vaatimusten ollessa toisistaan poikkeavia. Toinen tapa, jota tämän tutkimuksen valintana käytetään, on automaattinen ohjelmistojen asennus virtuaaliympäristöä pystytettäessä. Tätä kutsutaan provisioinniksi. (Hashimoto 2013, 35–36.)

Automaattinen provisiointi tuo Hashimoton (2013, 36) mukaan kolmenlaista etua: helppokäyttöisyyttä, toistettavuutta sekä tuotannon ja kehityksen välisen pariteetin lisäämistä. Kauhuskenaariot README-tiedostoista tai wiki-sivuista, joissa on merkittynä rivi riviltä asennuskomennot kullekin järjestelmälle voidaan provisioinnin myötä jättää historiaan.

Vagrant tukee virallisesti yleisimpiä konfiguraationhallintaohjelmistoja (Ansible, CFEngine, Chef, Puppet, Salt) sekä näiden lisäksi komentokehote-skriptejä (eng. shell scripts) ja hieman muista provisiointitavoista poiketen Dockeria oman syntaksinsa kautta. (Hashimoto 2013, 35.)

4.3 Ansible

Ansible on automatisointityökalu, joka soveltuu moniin IT:n tarpeisiin. Sen avulla voidaan konfiguroida järjestelmiä, julkaista ohjelmistoja ja orkestroida helposti monimutkaisiakin palvelinratkaisuja, kuten jatkuvaa julkaisua tai ohjelmistojen päivityksiä ilman, että palvelua ajetaan päivityksen ajaksi pois toiminnasta. Ansible on moduulipohjainen ohjelmisto, joka tarkoittaa, että kaikki suoritettavat konfiguraatiotoimenpiteet toteutetaan yksittäisten kirjastojen kautta. (Hochstein 2015, 2.)

Ansible on *idempotenssi* asetustenhallintatyökalu. Tämä tarkoittaa, että Ansiblen suorittamien komentojen jälkeen näiden tila on aina sama huolimatta siitä, ajetaanko asetus-skripti ensimäistä tai viidettä kertaa. (Hochstein 2015, 6.)

Ansible eroaa hieman luvussa 3.5. esitellyistä konfiguraationhallinnan peruspilareista filosofiallaan, jossa kaikkia komentoja ei abstraktoida täydellisesti, merkittävimpänä esimerkkinä pakettienhallinta. Ansible tarjoaa esim. apt- ja yum-moduulit, mutta ei näitä abstraktoivaa moduulia. Ansible toki mahdollistaa saman käsikirjan suorittamisen eri alustoilla ehtolauseita käyttäen, mutta tämä on hieman kömpelöä ja vaikeuttaa käsikirjan luettavuutta. Ansiblen vastaus konfiguraatiotiedostojen uudelleen käytettävyyteen ovat *roolit*, joista yksittäiset palvelinkonfiguraatiot (käsikirjat) koostuvat.

Ansible on eräs Vagrantin virallisesti tukemista provisiointityökaluista, joka helpottaa virtuaalisten ohjelmistoympäristöjen valmistamista tarjoamalla selkeän syntaksin ympäristön asetusten sekä ohjelmistojen asennukseen (Hashimoto 2013, 36).

Yksinkertainen Ansible-provisiointi Vagrantfilessä näyttää tältä:

```
config.vm.provision "ansible" do |ansible|
         ansible.playbook = "./provision_playbook.yml"

         # Run commands through sudo
         ansible.sudo = true
end
```

Ylläolevassa Ruby-kielellä kirjoitetussa koodipätkässä kerrotaan Vagrantille, että käytämme nimenomaan "ansible"-nimistä provisiointimenetelmää, sekä määrittelemme parametrina käytettävän käsikirjan, joka sisältää yksittäiset konfiguraatioaskeleet virtuaaliympäristön rakentamiseen.

Siinä missä Vagrantin sydän sykkii Vagrantfilen kautta, ovat Ansiblen keskiössä käsikirjat (eng. playbooks). Käsikirjassa määritellään, mitkä palvelimet sen avulla konfiguroidaan, ja listataan *tehtävät*, jotka käsitellään. Käsikirjat kirjoitetaan YAML-syntaksia käyttäen, ja ne ovat käytännöllisesti katsoen "suoritettava dokumentti" kaikista niistä välivaiheista, joita määriteltävä tehtävä tarvitsee. Jokainen näistä välivaiheista suoritetaan jonkin Ansiblen moduulin avulla. (Hochstein 2015, 28–29.)

Esimerkki yksinkertaisesta käsikirjasta:

```
---
- hosts: vagrant
  tasks:
    - name: Install nginx
      apt:
        pkg: "nginx"
        state: latest

    - name: Make sure nginx is running
      service:
        name: "nginx"
        state: "started"
```

Tässä:

- **hosts** kertoo millä palvelimella tai palvelimilla käsikirja on tarkoitettu suoritettavaksi.
- **tasks** listaa halutut tehtävät, jotka ympäristöä konfiguroitaessa suoritetaan

Esimerkissämme on kaksi tehtävää (eng. task): nginx-palvelinohjelmiston asennus sekä käynnistäminen.

- **name** on otsikko, joka näytetään tehtävää suoritettaessa. Vaikka tehtävien nimeäminen on vapaaehtoista, on se kuitenkin erittäin suositeltavaa. Otsikko toimii käsikirjassa eräänlaisena kommenttirivinä kullekin tehtävälle, jolloin konfiguraatiota lukevan henkilön on vaivatonta saada käsitys siitä, mitä mikäkin tehtävä tekee (Hochstein 2015, 29).
- **apt** on varsinaisesti suoritettavan *moduulin* nimi. Esimerkin moduuli hoitaa pakettienhallinnan Debian-pohjaisissa järjestelmissä. Ylläolevassa esimerkissä annamme parametreina moduulille paketin nimen "nginx" ja haluamamme tilan, jonka tahdomme paketille antaa, tässä "latest" (uusin versio). Muita mahdollisia tiloja ovat mm. "absent", poistettu tai "build-dep", joka varmistaa että paketin kääntämisessä vaadittavat apukirjastot ovat asennettuna.
- **service** on nimensä mukaisesti palveluihin erikoistunut moduuli, joka tukee useita eri käynnistysjärjestelmiä (kuten systemd tai upstart) abstraktoiden ne oman rajapintansa taakse. Esimerkissä annamme parametreina halutun palvelun nimen ("nginx") sekä tilan ("started"), ja Ansible huolehtii tämän tilan toteuttamisesta.

Vain kahdella yksinkertaisella tehtävällä saadaan näin määriteltyä toimiva järjestelmä, jolla on jo (pienimuotoinen) tehtävä. Mikäli tallennamme yllä olevan listauksen samaan kansioon aiemmin kuvatun Vagrantfilen kanssa ja suoritamme komennon "vagrant up", luo Vagrant "tyhjästä" virtuaalisen Ubuntu 14.04-ympäristön (ladaten pohjakuva Internetistä, mikäli tarpeen), johon on komennon suorittamisen jälkeen asennettuna sekä käyttövalmiina nginx-palvelinohjelmisto. Kyseinen lopputulos on toistettavissa ja identtinen jokaisella kehittäjän koneella.

4.4 Docker

Eräs suosituimmista uusimmista tulokkaista virtualisointitekniikoiden maailmassa on Docker, jonka versio 1.0 julkaistiin toukokuussa 2014 (Barbier 2014). Toisin kuin monet perinteiset virtualisointitekniikat, ei Docker emuloi koko käyttöjärjestelmää, vaan pelkästään "tarvittavat osat" käyttäen isäntäkoneen resursseja mahdollisimman paljon, mutta sisältäen mm. omat verkko- ja levyjärjestelmänsä.

Docker on "konttipohjainen" (eng. container) virtualisointimenetelmä. Näiden virtuaalisten konttien tarkoituksena on olla helposti siirrettävissä sekä olla suoritettavissa ympäröivästä isäntäkoneesta ja tämän ohjelmistoista riippumatta. Ainoina vaatimuksia Dockerin virtuaalikonttien suorittamiselle onkin vain yhteensopiva Linux-ydin sekä Dockerin ohjelmatiedosto. (Turnbull 2014, 32.)

Dockerin virtualisoinnin kulmakivenä on perinteisen alkuprosessin (eng. init) korvaaminen halutulla ohjelmistolla, jolle annetaan Linux-käyttöjärjestelmän ensimmäinen mahdollinen ohjelmatunnus (eng. process identifier, PID), "1". Docker kommunikoi suoraan vain tämän prosessin kanssa, ja tämän päättyessä myös koko muu ympäristö pysäytetään.

Virtuaaliympäristöön on mahdollista myös asentaa jokin prosessinhallintaohjelmisto (esim. Supervisor), joka sallii aliohjelmien käytön, mutta tätä ei varsinaisesti suositella. Dockerin parhaiden käytänteiden mukaisesti (Best practises for writing Dockerfiles 2015) yhdessä virtuaalikontissa tulisi olla vain yksi prosessi, jotta ns. horisontaalinen skaalaus (yksikköjen lukumääräinen lisääminen järjestelmään) sekä konttien uudelleenkäyttäminen olisi mahdollista ja helppoa. Erillisten palvelujen yhteen liittämiseksi Docker tarjoaa työkalut konttien - ja näin ollen yksittäisten prosessien - toisiinsa kytkemiseen. Dockerista puhuttaessa on tärkeätä erottaa toisistaan *kuva* (eng. image) ja *kontti* (eng. container).

Kuva tarkoittaa tässä spesifiä, ennalta määriteltyä virtuaaliympäristöä, joka voidaan rakentaa joko manuaalisesti komentoja suorittamalla tai erityisen asetustiedoston, Dockerfilen, avulla. Esimerkkejä Dockerin kuvista ovat vaikkapa virallinen *ubuntu-*

levykuva, joka sisältää puhtaan Ubuntu-käyttöjärjestelmän asennuksen tai SC5:n ylläpitäjien valmistama *ubuntu-sc5io*, joka pohjautuu *ubuntu*-levykuvaan, mutta sisältää mm. haluttujen ohjelmiston sekä asetusten asennuksen virtuaaliympäristön sisälle. (Turnbull 2014, 35–36.)

Dockerin levykuva rakennetaan komennolla ”docker build” tai ”docker commit”. Dockerin virtuaalikuvat ovat käytännössä aina rakennettuja jonkin olemassa olevan kuvan päälle, mutta myös "alkukuvan" voi rakentaa itse, jos näin haluaa.

Kontti tarkoittaa Dockerin terminologiassa "kuvaa, joka suoritetaan". Jokainen kontti on jonkin Docker-kuvan ajettava instanssi, joka on joko aktiivinen (parhaillaan käynnissä) tai epäaktiivinen (päättynyt suoritus). Samasta pohjakuvasta käynnistettyjä instansseja eli kontteja voi olla käynnissä useita samanaikaisesti ja ne ovat toisistaan riippumattomia. (Turnbull 2014, 37–38.)

Dockerin käyttö kehitysympäristönä vaihtoehtona Vagrantille ja Ansiblelle on hyvin mahdollista, ja tästä saattaa olla hyötyä etenkin siinä tapauksessa, että ohjelmiston loppusijoittaminen tapahtuu Docker-konttipohjaisena julkaisuna, sillä Dockerin virtuaaliympäristö on tässä tapauksessa kuitenkin määritettävänä. Docker-kontit ovat hyvin kevyitä ja toimivat pääsääntöisesti hyvin millä tahansa nykyisellä kehitystyökoneella. (Turnbull 2014, 33–34.)

Merkittävimpänä haittapuolena pelkän Dockerin käyttämiseen kehitysympäristön virtualisoinnissa voidaan mainita Ansiblen yksinkertaisen, luettavan asennussyntaksin kadottaminen Dockerfilen omaan, rajoittuneeseen ja melko vaikeaselkoiseen syntaksiin. Tämä saattaa kannustaa varsinkin Dockeriin perehtymättömiä kehittäjiä sivuuttamaan ympäristön kehittämisen, jolloin kehitystiimin yhtenäinen näkemys tuotantoympäristöstä jää puutteelliseksi.

Koska Docker (ainakin tätä kirjoitettaessa) vaatii Linux-järjestelmän toimiakseen, on kehittäjien avuksi kehitetty erillinen apuohjelma nimeltä boot2docker, joka toimii niin OS X- kuin Microsoft Windows-käyttöjärjestelmässä. Teknisesti ottaen boot2docker on hyvin pieni Linux-jakelu, joka pohjautuu Tiny Core Linuxiin ja joka

ajetaan Oracle Virtualbox -virtualisointiohjelmistoa käyttäen. Käyttäjän näkökulmasta tekninen toteutus on suhteellisen huomaamaton, ja voimmekin olettaa Dockerin käytön olevan suhteellisen riippumatonta kehittäjän käyttämästä käyttöjärjestelmästä.

Varsinainen Dockerin virtuaaliympäristö ("Docker-kuva") voidaan rakentaa manuaalisesti ajamalla kontin sisällä yksittäisiä komentoja ja julkaisemalla ne Dockerin komentoriviä käyttäen. Parempi tapa kuitenkin on käyttää tekstitiedostoa nimeltä *Dockerfile*, joka sisältää kaikki tarvittavat komentolauseet, joilla kontin sisäinen ympäristö rakennetaan. Nämä komentolauseet käyttävät Dockerin omaa syntaksia, ja ne suoritetaan järjestyksessä ylhäältä alas. (Turnbull 2014, 106.)

Yksinkertainen esimerkki Dockerfilestä voisi näyttää seuraavalta:

```
# Käytettävä järjestelmäkuva
FROM sc5io/ubuntu
MAINTAINER Markus Klinga

ENV NODE_ENV production

COPY ./dist /app
WORKDIR /app
RUN npm install

ENTRYPOINT [“node”, “app.js”]
```

Tässä:

- **FROM** kertoo Dockerille pohjakuvan, jonka päälle Dockerfilessä valmistettava kuva rakentuu. Parametrina "sc5io/ubuntu" tarkoittaa kuvaa nimeltä *ubuntu*, joka sijaitsee *sc5io*-repositoriossa. Halutessamme voimme määrätä pohjakuvan käytettävän version manuaalisesti kirjoittamalla version "merkki" (eng. tag) kaksoispistettä käyttäen parametrin perään. Esim:

FROM sc5io/ubuntu:14.04.1

- **MAINTAINER** kertoo Dockerfilen ylläpitäjän ja yhteystiedot. Tämän mainitseminen Dockerfilessä ei ole pakollista, mutta se helpottaa kommunikointia ja on hyvien käytänteiden mukaista.
- **ENV** määrittelee kuvan sisällä käytettävissä olevan ympäristömuuttujan. Näin ohjelmiston tärkeitä parametreja, kuten sen kuuntelemaa porttia tai mahdollisia konfiguraatiovaihtoehtoja, pystytään muokkaamaan suoraan Dockerfilestä muuttamalla.
- **COPY** lisää Dockerin virtuaaliympäristöön tiedostoja isäntäkoneelta. Esimerkissä lisäämme ”./dist” - kansion virtuaaliympäristön ”/app” -kansioksi. On huomattava, että tämän kaltainen lisääminen ei synkronoi tiedostoja kaksisuuntaisesti toisin kuin aiemmin esitetty Vagrantfilen kansioiden siirto isäntä- ja virtuaalikoneen välillä. Dockerin COPY yksinkertaisesti kopioi nykyiset tiedostot virtuaalikontin levyjärjestelmään.
- **WORKDIR** määrittelee kontille missä pohjahakemistossa seuraavat komennot tulee suorittaa.
- **RUN**-komennolla on kaksi vaihtoehtoista muotoa.
  - **RUN <komento>**

    Annettu komento suoritetaan virtuaalikontin sisällä käyttäen Bourne Shell -komentotulkkia (/bin/sh -c <komento>).

  - **RUN ["ohjelma", "parametri1", "parametri2", ...]**

    Laajennettu versio "RUN"-komennosta mahdollistaa minkä tahansa ohjelman suorittamisen käyttämättä komentotulkkia tai vaihtoehtoisen komentotulkin käyttämisen.
    - RUN ["/bin/bash", "-c", "echo", "$HOME"]
    - RUN ["/bin/nodejs", "/app/app.js"]

Jälkimmäisessä esimerkissä komentotulkin mahdollistamat prosessoinnit, kuten ympäristömuuttujat, eivät ole käytettävissä, sillä komentotulkki sivuutetaan tässä muodossa kokonaan.

- **ENTRYPOINT** määrittää kuvan pohjalta suoritettavien konttien pääprosessin sekä mahdolliset parametrit. ENTRYPOINT hyväksyy RUN-komennon tapaan kaksi erillistä formaattia.

Ylläannetun kuvan perusteella käynnistettävässä kontissa voidaan kuitenkin komentoparametreilla ohittaa suuri osa Dockerfilessä annetuista käskyistä (mm. ENTRYPOINT tai ENV-muuttujat), mikä saattaa vaikuttaa oleellisesti kontin sisältämien ohjelmistojen käyttäytymiseen.

Kun Dockerfile on kirjoitettu valmiiksi, voimme luoda ympäristön Dockerin komentorivikomennolla:

```
docker build -t projekti-kuva .
```

Tämä komento etsii Dockerfilen nykyisestä kansiosta ja suorittaa sen ylhäältä alas saakka komento kerrallaan. Komennolle annettava parametri ”-t” kertoo, minkä tunnuksen (eng. tag) haluamme luomallemme Docker-kuvalle antaa. Tätä tunnusta käytetään virtuaalikonttia suoritettaessa.

Kun haluamamme ympäristökuva on luotu, voimme suorittaa sen. Voimme ajaa virtuaaliympäristön komennolla:

```
docker run –ti projekti-kuva
```

Syntaksi on suhteellisen yksinkertainen: parametri ”-ti” kertoo dockerille, että haluamme ajaa ympäristön interaktiivisessa muodossa kyseistä terminaali-ikkunaa käyttäen (vastakohtana tälle toimisi parametri ”-d”, joka ajaa ympäristön piilotettuna taustaprosessina). Tämän jälkeen kerromme dockerille mitä levykuvaa haluamme käyttää (tässä "projekti-kuva", jonka aiemmin loimme "docker build"-komentoa käyttäen).

Kehitysympäristömme vaatii, että ohjelmiston sisältävä hakemisto on sekä virtuaaliympäristön että oman isäntäkoneemme käytettävissä, joten emme voi käyttää tä-

hän Dockerin omaa COPY-komentoa. Hakemistojen linkittäminen isäntä- ja virtuaalikoneen välillä on mahdollista Dockerin käynnistyskomennon ”-v”-parametrilla, jolle annetaan kaksoispisteellä erotettuna kansion sijainti isäntäkoneella sekä virtuaaliympäristössä.

Esimerkki komennosta, jossa ohjelmistokoodi sijaitsee kehittäjän työkoneella osoitteessa “/home/markus/projekti/src”, ja se synkronoidaan virtuaaliympäristön osoitteeseen “/app”.

```
docker run -ti -v /home/markus/projekti/src:/app projek-
ti-kuva <komento>
```

Hakemistojen synkronointi on kätevää tapauksissa, joissa haluamme työskennellä virtuaaliympäristössä olevien kohteiden kanssa. Lopullisissa Docker-julkaisutuotteissa hakemistojen synkronointia isäntäkoneen kanssa käytetään usein säilytettävän datan (eng. persistent data) tallentamiseen tai muokkaamiseen. Docker myös mahdollistaa pelkkien "datakonttien" luomisen, jolloin virtuaaliympäristön ainoana tehtävänä on toimia säilytettävän datan käyttöliittymänä. Nämä datakontit liitetään muihin virtuaalikontteihin Dockerin ”--volumes-from <kontin nimi>” -parametria käyttäen, jolloin niiden sisältämät hakemistot ovat myös muiden konttien käytettävissä.

Docker noudattaa "yhden prosessin" ideologiaa, joka tarkoittaa, että yhdessä virtuaaliympäristössä suoritetaan tismalleen yhtä prosessia (Best practices for writing Dockerfiles 2015). Käytännössä ohjelmistot kuitenkin monesti vaativat monia erillisiä osia toimiakseen, varsinaisen ohjelmiston lisäksi esimerkiksi tietokannan, jota ajetaan omassa virtuaaliympäristössään.

Docker tukee luontaisesti virtuaalikonttien linkittämistä toisiinsa, jolloin palvelimella sijaitsevat erilliset virtuaaliympäristöt kommunikoivat keskenään niiltä osin, kuin toiminnallisuuden saavuttamiseksi on tarpeen. Linkitettyjä kontteja voi olla useita, eivätkä kaikki näistä välttämättä "näe" toisiaan. Esimerkkinä ohjelmisto, jonka käyttöliittymä ja taustajärjestelmä on erotettu toisistaan, saattaa näyttää vaikkapa tämän kaltaiselta:

* 1. Kontti: HTML/Javascript - käyttöliittymä
* 2. Kontti: Taustajärjestelmänä toimiva sovellus
* 3. Kontti: Tietokanta

Tällöin jokaisesta kontista valmistetaan oma Dockerfile, ja suoritusvaiheessa kontit linkitetään toisiinsa siten, että ne voivat kommunikoida toistensa kanssa Dockerin hallinnoimaa sisäverkkoa hyväksi käyttäen. (Turnbull 2014, 172–174.)

Useasta kontista koostuvien ohjelmistojen hallinnointiin on kehitteillä useita apuohjelmia, mutta varsinaista standardia useiden konttien hallinnointiin ei vielä tätä kirjoitettaessa ole muodostunut. Dockerin omaan ekosysteemiin kuuluu ohjelma nimeltä docker-compose (tunnettiin aiemmin nimellä fig). Tämä käyttää omaa, tekstipohjaista asetustiedostoa koostamaan erillisten Docker-konttien käynnistysparametrit ja hallinnoi näistä koostuvaa kokonaisuutta oman käyttöliittymänsä avulla. (Turnbull 2014, 252–254.)

Tätä kirjoitettaessa kehitteillä on lukuisia web-käyttöliittymän avulla toimivia julkaisunhallintaohjelmistoja sekä kokonaisia palvelin-infrastruktuuriratkaisuja, jotka ovat keskittyneet yksinomaan Docker-konttien ajamiseen palvelimella, mutta aiheen nuoruudesta johtuen useimmat näistä ovat vielä joiltakin osin keskeneräisiä.

4.5 Jenkins

Integrointipalvelimen käyttöön ottaminen ja jatkuvan julkaisun menetelmiin siirtyminen muuttaa radikaalisti kehittäjien tapaa luoda uutta ohjelmistoa. Parhaimmillaan se virtaviivaistaa koko kehitysprosessia, auttaa löytämään bugeja nopeammin sekä tarjoaa niin kehittäjille kuin muillekin projektiin osallistuville läpinäkyvän katsauksen kehitystyöhön, sen ongelmakohtiin ja etenemiseen. Lopullinen tarkoitus onkin tuottaa business-arvoa yritykselle ammattimaisen, tehokkaan työskentelyn kautta. Smart (2011, 1) toteaakin, että jokaisen ammattimaisen kehitystiimin, olipa kuinka pieni tahansa, tulisi harjoittaa jatkuvan integroinnin kautta tapahtuvaa kehitystä.

Jatkuvan integroinnin ohjelmistona SC5:ssä käytössä on avoimen lähdekoodin ohjelmisto Jenkins, joka on luotu ohjelmistojen jatkuvaan rakentamisen sekä testaamisen helpottamiseen ja tarjoaa monipuolisen ja muokattavan repertuaarin erilaisia vaihtoehtoja lukuisien, vapaasti käytettävissä olevien lisäosien avulla. (Smart 2011, 3–4.)

Jenkinsin historia lähtee liikkeelle jo vuodesta 2004, jolloin Sun Microsystems alkoi kehittämään sitä "Hudson"-nimen alla. Vuonna 2011 Jenkins haarautui pääprojektista ennen kaikkea Oraclen (entinen Sun Microsystems) rekisteröityä yksinoikeuden "Hudson"-tavaramerkkiin. Oracle jatkaa edelleen Hudsonin kehittämistä, mutta projektit ovat ajautuneet varsin kauaksi toisistaan (Smart 2011, 4–5). Tässä opinnäytetyössä keskitytään pelkästään avoimen lähdekoodin versioon, Jenkinsiin.

Jenkinsin aloitusnäkymä tarjoaa kaikki ohjelmistoon konfiguroidut rakennustyöt (eng. build jobs). Rakennustöitä voidaan ajatella kokonaisen rakennusprosessin yksittäisinä askelina tai osasina: esimerkiksi lähdekoodin kääntäminen, teknisen dokumentoinnin generoiminen tai yksikkötestien suorittaminen voi olla yksittäinen rakennustyö. Projektit koostuvat usein erillisistä, toisiinsa liittyvistä töistä. (Smart 2011, 21)

Uuden rakennustyön luominen on nopeata ja vaivatonta Jenkinsin web-käyttöliittymää käyttäen. Työt ovat kaikkineen hyvin joustavia ja niiden parametrit sekä käytettävät lisäosat ovat varsin vapaasti muokattavissa. Yksinkertainen esimerkki työstä on ohjelmiston esityöstäminen (eng. preprocessing), jolloin työ vaatii varsinaisesti vain kaksi konfiguroitavaa osiota:

- Versionhallinnan osoite, josta ohjelmiston lähdekoodi haetaan.
- Välivaiheet ohjelmiston työstämiseksi.

Jenkins on helppo integroida käytettäviin viestintävälineisiin, kuten SC5:ssä käytettävään Slack-pikaviestinpalveluun. Halutessaan kehittäjät voivat saada tiedon niin onnistuneista kuin epäonnistuneistakin töistä suoraan jokapäiväisessä kommunikoinnissa käytettävälle projektikanavalle, jolloin tiedonkulku on varmaa ja tehokasta. Tä-

mä edesauttaa myös ongelmien tiedostamista sekä näihin puuttumista työryhmän välillä. (Humble & Farley 2011, 13–15.)

Jenkins tarjoaa web-käyttöliittymänsä avulla runsaasti tietoa siitä, mitä jokaisessa yksittäisessä rakennustyössä tapahtuu tallentamalla työn lokitiedostot, muuttuneet versionhallinnan tiedostot sekä muuta tämän kaltaista informaatiota jokaisen suorituskerran yhteydessä. (Smart 2011, 30–32.)

Jokainen yksittäinen rakennustyö jakautuu edelleen "askeliin", jotka täsmällisesti määräävät rakennustehtävän välivaiheet. Tällaisia välivaiheita voivat olla esimerkiksi Docker-kuvan rakentaminen, yksikkötestien suorittaminen tai jonkin toisen rakennustyön käynnistäminen. Projektiin liittyvät rakennustyöt voidaan käynnistää joko manuaalisesti tai halutuin väliajoin, esimerkiksi aina öisin. Jatkuvan julkaisun mahdollistamiseksi nämä vaihtoehdot kuitenkin ovat hitaita ja kankeita, ja onkin hyvä ajatus käyttää tässä kehitettävän ohjelmiston versionhallintaa apuna. (Humble & Farley 2011, 65.)

Yksinkertaisimmillaan tämä tapahtuu siten, että rakennuspalvelin lähettää aika ajoin versionhallintapalvelimelle pyynnön selvittääkseen onko jokin muuttunut edellisen pyynnön jälkeen (Smart 2011, 88). Mikäli näin on, lataa rakennuspalvelin automaattisesti uusimman version versionhallinnasta ja suorittaa tälle rakennustyössä määritellyt tehtävät. Tämä lähestymistapa on kuitenkin "tuhlaava", sillä pyyntöjä lähetetään usein tarpeettomasti. Parempi tapa onkin määrittää versionhallintapalvelimelle ns. koukku (eng. hook), joka suoritetaan jokaisen versionhallintaan ladatun muutoksen jälkeen. Tällöin rakennuspalvelin saa automaattisesti tiedon muutoksista vain silloin, kun jotakin todella on muuttunut. (Smart 2011, 100–101.)

Voimme myös määrittää rakennettavalle projektille sen versionhallinnan haaran, jonka haluamme rakentaa. Tämä mahdollistaa ohjelmiston kehityksen yleisesti "develop" ja "master"-nimisiksi nimetyissä päähaaroissa, joista "develop" ("kehitys") -haarassa saattaa olla ylimääräistä kehitysinformaatiota, laaduntarkkailua odottavia ominaisuuksia, ja "master" ("pää") -haarassa sijaitsee toimiva, julkaisuvalmis tuote.

Rakennuspalvelin suorittaa tismalleen sille määrätyt tehtävät, ja mikäli niin halutaan, voidaan myös varsinainen ohjelmiston julkaiseminen määrittää osaksi Jenkinsin rakennustöitä. Tällöin saavutetaan automaattinen julkaisuputki, joka hakee itsenäisesti versionhallinnasta uusimman version ohjelmistosta, valmistaa siitä julkaistavan tuotteen ja lähettää sen palvelimelle käytettäväksi.

5 Tutkimuksen toteutus

5.1 Julkaisuputki

Opinnäytetyön tutkimuksen kanssa rinnakkainen kehitystyö suoritettiin todellisen asiakasprojektin yhteydessä.

Tutkimuksen alkaessa kehitettävä projekti sisälsi jo Dockerfilen (liite 1), jonka avulla ohjelmisto saatettiin asentaa virtuaalikonttiin ja julkaista manuaalisesti käyttäen SC5:n sisäistä palvelua. Tämä on jo itsessään merkittävä edistysaskel aiempaan tilanteeseen verrattuna, jossa ohjelmistolle annettiin vain oma käyttäjätili joltakin SC5:n ylläpitämältä kehityspalvelimelta, ylläpitäjät asensivat tarvittavat puitteet ja kehittäjät kävivät palvelimella päivittämässä uusimman version manuaalisesti ssh-yhteyden ylitse.

Ensimmäiseksi kehityskohteeksi valittiin kokonaisen julkaisuputken pystyttäminen, sillä tämä tuntui tuovan eniten lisäarvoa ohjelmiston nopeaan kehitykseen. Kohdetilana (utopiana) toimi ajatus siitä, että jokaisesta versionhallintaan tallennettavasta muutoksesta rakennetaan julkaisuputken läpi kulkeva tuotos, joka on välittömästi niin asiakkaan kuin kehittäjienkin tai muiden projektiin osallistuvien henkilöiden nähtävissä halutussa internet-osoitteessa.

Koska projektin alkaessa eräs kehittäjistä oli jo valmistellut käytettävän Dockerfilen, suoritettiin ohjelmiston asentaminen virtuaaliympäristöön tämän pohjalta. Asennuskaava on suhteellisen yksinkertainen, eikä sen perustoiminnallisuutta ollut tarpeen tämän kehitystyön yhteydessä muuttaa.

Koko ohjelmisto sijaitsee Dockerfilen määritysten mukaisesti samassa virtuaaliympäristössä, jossa suoritetaan paitsi node.js-pohjainen ohjelmisto, myös tämän tarvitsema MongoDB-tietokanta. Tämä on mahdollista virtuaalikontin sisällä Supervisor-ohjelmaa käyttäen, joka hallinnoi molempien ohjelmistojen toimintaa. Ohjelmisto ei tarvitse kehitysvaiheessa vielä datan tallennusmahdollisuutta, vaan halutut tietokantataulut luodaan ohjelmistoa käynnistettäessä testidatasta, joka on kirjoitettu kiinteäksi osaksi ohjelmiston koodia.

Koska varsinaisen virtuaaliympäristön luomiseen ei tarvinnut tässä tapauksessa käyttää aikaa, saatettiin siirtyä suoraan julkaisuputken rakentamiseen. Tähän käytettiin SC5:n omaa, olemassa olevaa rakennuspalvelinta, jonne Jenkins oli jo asennettu ja konfiguroitu valmiiksi. Kehitystyö alkoi Jenkinsin web-käyttöliittymää käyttäen, jonne luotiin uusi rakennustyö. Työn rungoksi valittiin "Freestyle project", vapaan tyylin projekti, joka mahdollisti monipuolisen, täsmälleen halutun kaltaisen rakennustyön määrittämisen.

Tärkeimpinä konfiguraation kohteina rakennustyössä olivat versionhallinnan saattaminen Jenkinsin tietoisuuteen sekä virtuaalikontin rakentaminen ja sen julkaiseminen versionhallinnasta löytyneen version mukaisesti. Lisäbonuksena määriteltiin SC5:n kehittäjien käyttämälle pikaviestimelle Slackille ilmoitukset aina rakennuksen jälkeen, jolloin tieto rakennustyön onnistumisesta tai siinä mahdollisesti esiintyneistä virheistä saatiin nopeasti kehittäjien tietoon.

Jenkins sisältää tuen suosituimmille versionhallintatyökaluille, kuten Gitille tai Subversionille. Kehitettävän ohjelmiston versionhallinta sijaitsi asiakkaan yksityisessä Github-repositoriossa, joten tähän repositorioon asennettiin asiakkaan suostumuksella erillinen julkaisuavain (eng. deployment key). Tämä on tavanomaisen SSH-avainparin julkinen osa, ja yksityinen puoli avaimesta on Jenkinsin hallussa (tämän projektin kohdalla käytettiin Jenkinsille jo aiemmin konfiguroitua avainta, jonka julkinen osa löytyi rakennuspalvelimelta osoitteesta *~jenkins/.ssh/id_rsa.pub*).

Kun avain oli konfiguroitu asiakkaan repositorioon, saatettiin määrittää Jenkinsin rakennustyön versionhallintasovellukseksi Git ja repositorion osoitteeksi:

```
git@github.com:AsiakkaanNimi/Projekti.git
```

Jenkins tulkitsee tämän (aivan oikein) siten, että Githubin kanssa halutaan käyttää ssh-yhteyttä. Jenkinsille annettiin lisäksi ohjeeksi rakentaa “master”-haara versionhallinnasta löytyvästä ohjelmistosta kohdassa "Branches to build".

Toinen tärkeä kohta versionhallinnan määrityksen lisäksi on kertoa Jenkinsille, milloin rakennustyö varsinaisesti suoritetaan. Jenkinsin "Build trigger" (rakennustyön laukaisu) kohtaan määritettiin repositorioon halutuin väliajoin suoritettavat kyselyt mahdollisista muutoksista valitsemalla vaihtoehto "Poll SCM" ja ajastamalla se komennolla:

```
H/5 6-20 * * 1-5
```

Ajastuksen muoto on tuttu Linuxin "cron"-apuohjelmistoista. Vasemmalta oikealle luettuna Jenkinsille kerrotaan suorituksen ajankohdaksi määritetty minuutti, tunti, kuukaudenpäivä (1–31), kuukausi (1–12) ja viikonpäivä (0–7, jossa sekä 0 että 7 tarkoittavat sunnuntaita).

"*" -merkkiä käytetään merkitsemään kaikkia mahdollisia arvoja (0, 1, 2 jne.), kun taas "H"-merkkiä voidaan käyttää satunnaistamaan ajastus siten, että suoritettavat tehtävät ajetaan "joskus, kun mahdollista". Tämä helpottaa tasaamaan rakennuspalvelimelta lähteviä pyyntöjä. Versionhallintakyselyt ajastettiin suoritettavaksi noin viiden minuutin välein (H/5), kello 6–20 jokaisena arkipäivänä (1–5).

Versionhallinnasta löytyvä repositorio ladattiin Jenkinsin suoritusympäristöön, ja virtuaalikontin rakennus tapahtui täsmälleen versionhallintaan tallennetun Dockerfilen mukaisesti "Docker build and publish"-nimistä Jenkinsin lisäosaa käyttäen. Lisäosa on hyvin helppokäyttöinen ja vaatii lyhimmillään vain määrityksen siitä, millä nimellä haluamme kontin rakentaa. Käytimme SC5:n omaa, yksityistä Docker-repositoriota ja määritämme kontin nimeksi ”repositorio/projekti” sekä liitteeksi (eng. tag) ”latest”.

Käytännössä tämä tarkoittaa, että rakennusvaiheessa Jenkins suorittaa rakennettavan ohjelmiston sisältävässä hakemistossa seuraavat komennot:

```
docker build -t repositorio/projekti:latest .
docker push repositorio/projekti
```

Näiden komentojen tuottama lopputulos, virtuaalikontti, on Dockerin luonteesta johtuen identtinen yksittäisen kehittäjän omalla työkoneellaan suorittamaan virtuaalikontin valmistamisen kanssa. Jenkinsin rooli tässä on ennen kaikkea käännöstyön automatisointi sekä virtuaalikontin auktoriteettina toiminen. Useimmissa tilanteissa on hyvien käytänteiden mukaista estää yksittäisen kehittäjän oikeus suoraan työntää (eng. push) valmistettu virtuaalikontti Docker-repositorioon ja vaatia kaiken kehitystyön kulkevan julkaisuputken lävitse, jolloin kehitystyö pysyy läpinäkyvänä ja prosessi selvästi rajattuna. Kun Jenkins on valmistanut ja julkaissut virtuaalikuvan SC5:n repositorioon, on aika saattaa se palvelimelle. Tätä ennen kuitenkin jouduttiin luomaan uusi projekti, johon Docker-kuva kytkeytyy.

SC5:ssä käytetään virtuaalikonttien julkaisemiseen omaa, sisäisesti kehitettyä ohjelmistoa, joka helpottaa uusien projektien luomista nopealla web-käyttöliittymällä. Projektia luotaessa voidaan antaa asetuksina esim.:

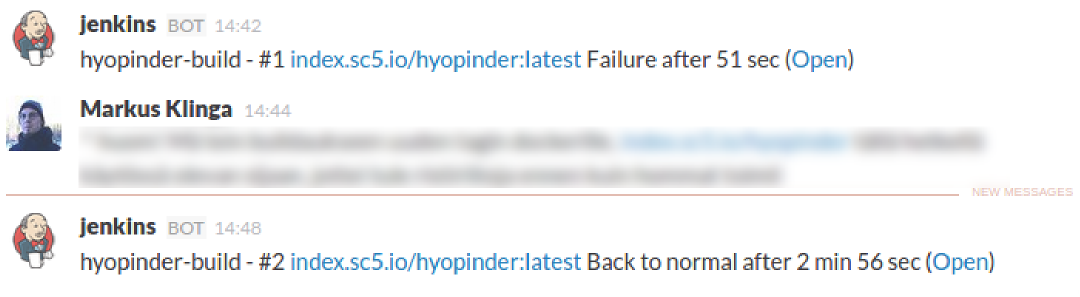
- haluttu osoite, josta ohjelmisto tulee löytyä
- portti, jota ohjelmiston kontti kuuntelee
- varattu levykansio säilytettävää dataa varten
- käyttäjätunnus ja salasana sivuston katselemiseen
- yms.

Kerroimme kehitteillä olevan projektimme löytyvän Docker-repositoriosta, kuuntelevan porttia 9000 ja määritämme osoitteen (kehitys-osoitteet ovat muotoa *.sc5.io, ja ne luodaan Amazon Web Servicesin Route 53:n rajapintaa käyttäen). Tämän jälkeen palvelu luo automaattisesti sijoituspaikan kehitteillä olevaa Docker-konttia varten, eikä kehittäjien tai ylläpitäjien tarvitse tämän myötä tehdä manuaalisesti mitään palvelinpuolella ohjelmiston julkaisemisen mahdollistamiseksi.

Kun projekti näin oli luotu, palattiin vielä Jenkinsin rakennustyöhön ja määritettiin viimeinen rakennusaskel. Tässä käytettiin tässä toista, Jenkinsiin jo aiemmin määriteltyä rakennustyötä nimeltä "awsdocker-restart-container", jonka keskeinen tehtävä on lähettää rajapinta-kutsu SC5:n Docker-hallintatyökalulle ja käynnistää haluttu virtuaalikontti uudelleen julkaistusta versiosta.

Tämän alityön suoritus määritettiin luomalla varsinaiseen rakennustyöhömme uusi rakennusaskel nimeltä "Trigger/call builds on other projects", ja antamalla parametrina alityön nimi. Työ määritettiin "estäväksi" (eng. blocking), eli projektin varsinainen rakennustyö keskeytetään, kunnes "awsdocker-restart-container" on suoritettu, joko onnistuneesti tai epäonnistuen, jolloin myös koko rakennusprojekti epäonnistuu. Tälle alityölle annettiin parametriksi ohjelmamme nimi ympäristömuuttujaa hyväksi käyttäen asetuskohdassa ”Predefined parameters”.

Näin olimme valmistaneet julkaisuputken ja voimme koettaa sitä.



Kuvassa näkyy Slack-viestintäohjelmaan saapuneita rakennusraportteja. Ensimmäinen yritys epäonnistui inhimillisen erehdyksen takia (tekijä unohti päivittää Dockerfilen muutokset versionhallintaan), toinen rakennusyritys meni puhtaasti lävitse ("Back to normal" on Jenkinsin tapa kertoa, että rakennus onnistui ensimmäistä kertaa epäonnistuneen rakennuksen jälkeen) ja päivitti ohjelmiston palvelimelle juuri niin kuin pitikin.

Näiden määritysten jälkeen Jenkinsin julkaisuputki toimi heti ensimmäisen version myötä juuri niin kuin lähtötilanteessa oli toivottu. Jokaisesta versionhallintapalveli-

melle tulleesta muutoksesta valmistetaan uusi virtuaalikontti, ja mikäli valmistusvaiheessa mikään ei tuota ongelmia, uusin versio julkaistaan välittömästi määrätylle kehityspalvelimelle. Mikäli ongelmia esiintyy, ilmoitetaan tästä kehittäjille pikaviestinohjelmistoa käyttäen, eikä palvelimelle julkaista uutta versiota ennen kuin nämä ongelmat on korjattu siten, että Jenkinsin rakennustyö suoritetaan onnistuneesti loppuun.

Jo olemassa olevan virtuaalikontin päivittäminen palvelimella tapahtuu Docker-repositoriota sekä SC5:n sisäisen hallintaohjelmiston rajapintakutsua hyväksi käyttäen. Jenkins lataa valmistuttuaan rakentamansa virtuaalikontin repositorioon ja kertoo hallinnointityökalulle, että uusin versio on saatavilla. Tämän jälkeen palvelimella sijaitseva virtuaalikontti poistetaan ja korvataan uudella, Jenkinsin valmistamalla versiolla. Julkaistavan tuotteen mahdollinen testaaminen tapahtuu julkaisuputken aikana Docker-kontin rakennusvaiheessa, mutta tässä vaiheessa kehitystyötä testaamista ei vielä koettu tarpeelliseksi viedä rakennusvaiheen pintapuolisia tarkasteluja pidemmälle.

Asiakas oli erittäin tyytyväinen järjestelyyn, jossa he saattoivat välittömästi nähdä uusimmat kehitykset suoraan omilla selaimillaan. Tämä paransi kaikkien osapuolien yhteistä ymmärrystä siitä, miltä tuote ja siihen kehitetyt ominaisuudet todellisuudessa näyttävät, ja vähensi tarpeettomien ominaisuuksien kehittämiseen kuluvaa aikaa.

Projektin alkaessa, kun julkaisukelpoista tuotetta ei vielä ollut olemassa, päädyttiin kehitystyössä valmistamaan vain yhtä versionhallinnan haaraa, jolloin kaikki kehitystyö tapahtuu suoraan julkaisuputken lävitse kulkevaan tuotteeseen. Koska kyseessä oli nopea prototyyppi, ei tästä koettu olevan ongelmia, mutta pidempi-ikäisissä, kenties jo yleisölle julkaistuissa tuotteissa tulee julkaisuputken valmistuksessa esiin myös useamman kehityslinjan kuljettaminen rinnakkain siten, että mahdollisesti joitakin ohjelmiston ominaisuuksia rikkova kehitystyö ei estä pika-korjausten yms. tekemistä valmiiseen tuotteeseen ja näiden testaamista.

Manuaalinen käyttäjätestaus jää tässä projektissa varsinaisesti asiakkaan tekemän tarkastelun tasolle, pitkälti projektin luonteesta riippuen. Julkaisuputkeen olisi mah-

dollista luoda erillinen "julkaisukandidaatti", joka vaatisi laadunvalvojien manuaalisen hyväksynnän julkaisulle, mutta tämä todettiin projektin luonteen johdosta tarpeettomaksi.

Ylläpitäjän näkökulmasta julkaisuputken valmistaminen projektille kevensi projektista aiheutuvaa työkuormaa. Suurena etuna palvelinpuolen ylläpitäjän ei tarvitse välttämättä tietää mitään kehitteillä olevan ohjelmiston tekniikoista, niiden rajoituksista tai päivittämisestä, sillä tämän kaltaiset asiat siirtyvät pääsääntöisesti kehittäjien harteille. Ylläpitäjien tehtävä SC5:ssä kääntyneekin ennen kaikkea fasilitoinnin puolelle, jolloin tärkeimpänä tehtävänä on valvoa virtuaalikonttien taustajärjestelmien tilaa ja ylläpidettävyyttä sekä mahdollistaa kehittäjille sopivat puitteet toimia rajatussa ympäristössä.

5.2 Kehitysympäristö

Julkaisuputken valmistuttua onnistuneesti suunnattiin katse seuraavaan kehityskohteeseen, joka oli kehitysympäristön virtualisoiminen. Kehitystyö tapahtuu tällä hetkellä kehittäjien omilla työkoneilla ilman kehitettävän ohjelmiston eristämistä muista ohjelmistoista. Ihannetilana tutkimuksen tässä vaiheessa pyrittiin kannustamaan Vagrantin käyttämiseen luomalla helppokäyttöiset ja kehittäjäystävälliset puitteet tätä varten. Tämä paitsi mahdollistaa kehittäjien työkoneiden pysymisen "puhtaina" tarpeettomista ohjelmista, myös uusien kehittäjien nopean liittämisen projektiin, sillä kehitysympäristön pystyttäminen ei vaadi vaivaa.

Kehitysympäristön virtualisointi aloitettiin varsin luonnollisesti jo olemassa olevan Dockerfilen pohjalta. Ensimmäisenä tehtävänä määriteltiin käytettävät riippuvuudet ja alettiin muokkaamaan näistä Vagrantfileä ja Ansiblen käsikirjaa. Dockerfile käyttää SC5:n julkista “sc5io/ubuntu14.04.1b” -Docker-kuvaa pohjanaan, joten luonnollisesti Vagrantin pohjaksi valittiin sama käyttöjärjestelmä ja versionumero, Ubuntu 14.04.1. Docker-kuva sisältää myös mukavuussyistä jo erinäisiä aputyökaluja, kuten SASS- ja Compass-apuohjelmat sekä uusimman node.js -julkaisuversion. Tämä on huomattava kehitteillä olevan ohjelmiston mahdollisina riippuvuuksina myös Ansiblen käsikirjaa luotaessa.

Käytetty Vagrantfile on varsin standardinomainen ja määrittää käytettävän järjestelmälaatikon, virtuaaliympäristöön jaetun kansion, virtuaaliympäristön IP-osoitteen sekä muut ominaisuudet. Kehittäjän kannalta oleellisinta tässä on huomata ympäristön IP-osoite, josta virtuaaliympäristöön saa yhteyden. Tässä tapauksessamme asetimme virtuaaliympäristölle varsin mielivaltaisesti osoitteen 192.168.2.2. Kokonainen Vagrantfile on nähtävillä liitteessä 2.

Ansiblen käsikirja koostuu peräkkäisistä, yksittäisistä askelista, jotka yhdessä suoritettuna saattavat virtuaaliympäristön haluttuun tilaan. Tällaisia askelia voivat olla esim. jonkin ohjelmiston asentaminen tai hakemiston luominen. Tässä käydään lävitse käsikirjasta vain pääkohdat sekä ns. ratkaisua vaatineet askeleet. Kokonainen käsikirja on nähtävillä liitteessä 3.

Ensimmäisenä vaiheena asennettiin järjestelmätason paketit Ubuntun käyttämällä apt-paketinhallintaohjelmalla. Kehittäjien valinta tässä projektissa oli käyttää MongoDB:n uusinta versiota, joka vaati esivalmisteluina MongoDB:n oman repositorion asentamista. Myös node.js asennettiin omasta lähteestään Ubuntuun valmiiksi asennettujen repositorioiden sijaan.

Varsinainen pakettien asentaminen Ansiblea käyttäen on yksinkertaista. Asentamisessa käytettiin Ansiblen tarjoamaa “with_items”-direktiiviä ja listattiin tälle parametreiksi halutut ohjelmistot.

```
- name: Install system packages
   apt:
     pkg: "{{ item }}"
     state: installed
     update_cache: yes
   with_items:
     - "mongodb-org"
     - "nodejs"
     - "git"
     - "ruby"
```

```
      - "ruby-dev"
```

Määrite ”update_cache” tässä suorittaa käytännössä paketinhallintaohjelmiston pakettilistauksen päivityksen ottaen huomioon aiemmissa askelissa asennetut repositoriot.

Koska ohjelmiston kehittäjät käyttävät SASS:a CSS-tyylitiedostojen kirjoittamiseen, oli tarpeen asentaa tämän mahdollistamat apuohjelmat. Nämä asennettiin Rubyn omaa gem-pakettienhallintaohjelmistoa käyttäen. Vaikka Ansible tarjoaa gem-moduulin, jouduttiin tähän käyttämään Ansiblen shell-moduulia, sillä gem-moduuli ei suoraan soveltunut yksittäisen käyttäjän tarvitsemien pakettien asentamiseen.

Kun järjestelmätason ohjelmistot oli asennettu virtuaaliympäristöön, voitiin keskittyä ohjelmiston erityisvaatimuksiin. Kuten luvussa 3.2. esitettiin, tapahtui tämä meidänkin tapauksessamme asentamalla vaatimukset käyttäen npm- ja bower-paketinhallintaohjelmia. Mielenkiintoisena yksityiskohtana tässä asennettiin bower-ohjelmisto käyttäen npm:ää.

Ohjelmistomme vaati kaksi järjestelmänlaajuista asennusta, grunt-cli:n (jota käytetään Grunt-ohjelmiston suorittamiseen komentoriviltä) sekä bowerin. Nämä voidaan asentaa Ansiblen npm-moduulia käyttäen siten, että komennolle annetaan yksinkertainen lisäparametri "global: yes".

```
- name: Install global dependencies
  npm:
         global: yes
         name: "{{ item }}"
  with_items:
         - "grunt-cli"
         - "bower"
```

Tämän jälkeen suoritettiin ohjelmiston versionhallinnasta löytyvien "ohjeiden" mukaisesti vaatimusten asentaminen ajamalla npm:n sekä bowerin asennuskomennot Ansiblen npm- ja command-moduuleita käyttäen. Ansibleen on kehitteillä myös eril-

linen bower-moduuli, mutta tätä työtä tehtäessä se ei ole vielä tuotantokäyttöön soveltuvassa versiossa.

Vaikka esitetty ratkaisu virtuaaliympäristöjen käyttämiseen on sinällään toimiva, havaittiin siinä eräs ongelmallinen seikka. Vagrant toimii kaikilla suurilla ympäristöillä (Linux, Windows, Mac), mutta Ansible ei ainakaan tätä kirjoitettaessa tue Windowsia alustanaan. Tämä ei ole SC5:n kehittäjien joukossa suuri ongelma, mutta jotta virtuaaliympäristön pystyttäminen toimisi varmasti kaikilla alustoilla, oli ratkaisuun muovattava pienoinen parannus.

Koska Vagrant ja Virtualbox ovat käyttöjärjestelmä-riippumattomia ja koska ohjelmiston tuotantoympäristöt ovat lähes poikkeuksetta Linux-järjestelmiä, tuntui luontevalta käyttää tätä virtuaaliympäristöä hyödyksi Ansiblen suorittamisessa. Päädyimme kirjoittamaan komentokehote-skriptin, joka asentaa Ansiblen virtuaaliympäristöön ja suorittaa annetun käsikirjan kokonaisuudessaan virtuaaliympäristön sisällä. Tämä ratkaisu oli myös siitä otollinen, ettei yksittäisen kehittäjän tarvitse asentaa Ansiblea omalle työkoneelleen.

Vagrant tukee komentokehote-skriptien kautta provisiointia suoraan ja Vagrantfilen provisiointi määritettiin seuraavasti:

```
config.vm.provision :shell, :keep_color => true,
         :inline => "export PYTHONUNBUFFERED=1 && export
ANSIBLE_FORCE_COLOR=1 && cd /app && ./vagrant_init.sh"
```

Tässä kerrottiin Vagrantille että käytämme provisiointi-menetelmänä shell:ä, ja annamme varsinaisen komennon ”:inline”-direktiiviä käyttäen. PYTHONUNBUFFERED ja ANSIBLE_FORCE_COLOR -ympäristömuuttujat mahdollistavat Ansiblen värikoodauksen ja välittömän tekstin tuottamisen. Oletuksena Python bufferoi ulostulevan tekstin ja julkistaa kaiken vasta, kun Vagrant on saanut provisioinnin suoritettua. Nämä asetukset eivät ole varsinaisesti tarpeen, mutta tekevät provisioinnin seuraamisesta miellyttävämpää. Tämän lisäksi siirryttiin virtuaaliympäristön sisällä “/app”-

hakemistoon ja ajettiin "vagrant_init.sh"-niminen Bash-skripti (liite 4), joka asentaa Ansiblen vastaluotuun ympäristöön.

Lopputuloksena saavutettava virtuaaliympäristö on siis aivan kuten aiempikin, mutta sisältää myös Ansiblen asennuksen. Kehitysympäristössä tämä lienee hyväksyttävä poikkeama täydellisen puhtaasta virtuaaliympäristöstä, joskin on huomattava että Ansiblen asennus lataa mukanaan joitakin python-kirjastoja. Mikäli tästä johtuvia ongelmia havaitaan, voidaan provisiointi-skriptiä muokata siten, että Ansiblen asennus poistetaan välittömästi virtuaaliympäristön onnistuneen luomisen jälkeen.

Kaikki näyttää tässä vaiheessa hyvältä, mutta on huomautettava että Bash-skriptin tuominen provisiointiin merkitsee entistä kiinteämpää sidonnaisuutta yhteen järjestelmään. Mikäli kehitetään ohjelmistoa, joka suoritetaan jotakin muuta kuin Ubuntu-käyttöjärjestelmää käyttäen, on “vagrant_init.sh” kirjoitettava uudestaan kutakin järjestelmää vastaavaksi (samoin kuin jokainen Ansiblen käsikirja on räätälöitävä kullekin ohjelmistolle erikseen).

Kehitysympäristöjen pystyttäminen on edellä annettujen skriptien jälkeen helppoa ja vaivatonta ja onnistuu jokaisella kehitysalustalla. Ainoana vaatimuksena kehitysympäristön pystyttämiseksi ovat Vagrant sekä Virtualbox, joiden tulee olla kehittäjän koneella asennettuina.

Pieni kompastuskivi tässä ratkaisussa huomattiin, sillä Linux-versiolla tehtynä Vagrant jakaa oletuksena työhakemiston virtuaaliympäristön “/vagrant”-hakemistoksi, mutta Mac-koneilla näin ei tapahtunut. Tämä oli kuitenkin helppoa kiertää muuttamalla Vagrantfilen provisiointikäskyn hakemistoksi manuaalisesti synkronoimamme “/app “aiemman “/vagrant”:n sijaan (molemmat kansiot osoittavat Linux-koneilla samaan isäntäkoneen juurihakemistoon).

Varsinainen kehitystyö onnistui virtualisoidussa ympäristössä oletetulla tavalla. Ympäristön käynnistäminen Vagrantin avulla tapahtuu komennolla:

```
vagrant up
```

Ja ympäristöön yhteyden ottaminen ssh:ta käyttäen komennolla

```
vagrant ssh
```

Tämän jälkeen kehittäjä voi siirtyä ohjelmiston hakemistoon ja käynnistää ohjelmiston kuten omalta koneeltansakin:

```
cd /app; grunt serve
```

Komento "grunt serve" tässä käynnistää kehityspalvelimen ja tarkkailee jatkuvasti ohjelmistokoodiin tulleita muutoksia. Muutoksen huomatessaan se rakentaa kehitysversion uudestaan ja päivittää selaimessa näkyvän ohjelmiston välittömästi.

Koska kehitysprojekti oli jo täydessä vauhdissa virtualisoitujen ympäristöjen luomisen aikana, eivät tästä saadut hyödyt olleet niin näkyviä tässä nimenomaisessa projektissa – kehittäjät olivat jo asentaneet tarvittavat ohjelmistot omille koneilleen projektin alkaessa. Edellä valmistettu Vagrant-ympäristö kuitenkin toimii hyvänä vertailupohjana myöhemmille projekteille ja todistaa konseptin toimivuuden käytännön tasolla.

Eräänä pohdintaa aiheuttaneena huomiona oli, että Vagrant ja Virtualbox toimivat ikään kuin ylimääräisinä riippuvuuksina ohjelmistolle, ja vaihtoehtona esitettiin Dockerin käyttäminen suoraan kehitystarkoitukseen. Kuten Dockeria käsitelleessä teorialuvussa todettiin, tämä on täysin mahdollista ja saattaa monesti olla helpompi kuin erillisen Ansible-käsikirjan luominen. Kääntöpuolena jokaisen projektiin liittyvän kehittäjän tulee osata vähintäänkin perusteet Docker-konttien käyttämisestä. Docker ei myöskään toimi suoraan Windows- tai Mac-koneilla, vaan vaatii jonkin ylimääräisen virtualisointitavan toimiakseen.

Toisinaan voi olla myös paikallaan korvata Ansible pelkällä yksinkertaisella Bash-skriptillä. Eräässä aiemmassa projektissa yksinkertainen, kymmenen rivin Bash-komentotiedosto havaittiin helpommaksi ymmärtää ja käyttää Vagrant-

kehitysympäristön pystyttämiseen kuin moneen paikkaan rönsyilevä, yli sata riviä sisältävä Ansiblen käsikirja, joka suoritti saman toiminnallisuuden.

Kehitysympäristön virtualisoimisen todettiinkin olevan hyvä käytäntö, mutta todellista hyötyä tästä saataneen vasta isommissa projekteissa ja etenkin sellaisissa joissa kehittäjät vaihtuvat taajaan tai heitä on paljon. Myös kehitysprojektit, joissa ympäristöllä on joitakin hyvin spesifisiä vaatimuksia, hyötyvät kehitysympäristön virtualisoinnista.

6 Jatkuvan julkaisun mahdollistaminen

Opinnäytetyön luonne määräytyi tutkimusta tehtäessä kaksijakoiseksi. Lähtökohtaisena ongelmana työssä lähdettiin tutkimaan kehittäjien sekä ylläpitäjien välistä kuilua ja tiedonkulun ongelmia. Tätä ongelmaa varten kehitettyjen Devops-menetelmien pohjalta lähdettiin muokkaamaan SC5:n arkipäivää koskevaa ratkaisua, jota lopulta käsiteltiin yksittäisen asiakasprojektin kautta.

Ongelmakenttänä jatkuvan julkaisemisen mahdollistaminen kaikissa tapauksissa on valtavan suuri, ja olennaisimmat osat tästä on kuvattu yleiskatsauksen omaisesti ohjelmistokehityksen näkökulmasta luvussa 3. Nopeatempoisessa web-kehityksessä yksittäiset projektit elävät usein omaa elämäänsä ja standardiratkaisujen löytäminen kaikkiin mahdollisiin tilanteisiin on haastavaa, mutta yleisiä puitteita ja toimintamalleja kyetään varsin hyvin kirjoittamaan.

Tämän työn puitteissa päädyttiin vain julkaisuputken mahdollistamista tukevien käytännön toimien kautta luomaan pohja laajemmalle jatkuvan julkaisemisen ja ohjelmiston ylläpitämisen, tarkkailun ja päivittämisen saattamiselle SC5:n ohjelmistokehitykseen. Teoksen merkittävin tulos liittynee Jenkinsin käyttöönottamiseen projektin julkaisuputken toteutuksessa. Tämä palvelinohjelmisto toimii linkkinä kehittäjien ja kehitteillä olevan tuotteen julkaisemisen välillä ilman ylläpitäjien näkyvää puuttumista prosessiin. Toteutuksen toimivuuden osoituksena kolmen viikon aikana julkaisuputken läpi oli kulkenut ohjelmistosta jo 186 erillistä, käyttövalmista versiota, jotka välittömästi julkaistiin asiakkaan saataville.

Merkittävänä apuna tämän prosessin luomisessa SC5:n näkökulmasta toimii vasta viime vuosina julkaistu Docker-virtualisointimenetelmä, joka tarjoaa helppokäyttöisen, nopean ja tehokkaan tavan luoda kertakäyttöisiä virtuaaliympäristöjä. Ohjelmiston pakkaaminen tämän kaltaiseen virtuaalikonttiin on automatisoitua ja suhteellisen yksinkertaista. Virtuaalikonttien hallinnointi palvelimilla aiheuttaa edelleen päänvaivaa, ja tähän tuleekin keskittyä tulevaisuudessa entistä paremmin.

Jatkuvasta julkaisusta saadut edut havaittiin käytännössä välittömästi projektin aikana. Ketterä ohjelmistokehitys, kuten SC5:ssä laajasti käytettävä SCRUM painottaa voimakkaasti kommunikaation ja läpinäkyvyyden merkitystä niin asiakkaan ja kehittäjien kuin muidenkin projektiin osallistuvien tahojen välillä. Valmistettavan ohjelmiston kehittäminen haluttuun suuntaan hyötyy suuresti siitä, että kulloinenkin kehitysversio on jatkuvasti kaikkien projektiin osallistuvien ihmisten saatavilla – tällöin jokainen voi helposti tarkastaa, miltä nykyinen versio ohjelmistosta näyttää.

Tutkimuksen aikaisessa projektissa asiakkaalta tuli positiivista palautetta nimenomaan läpinäkyvyyden ja kehitysprojektin välittömän saatavuuden johdosta. Tarpeettomaksi havaittujen ominaisuuksien kehittäminen voidaan pääsääntöisesti välttää ja keskittyä siihen, mikä kulloinkin näyttää tuovan eniten hyötyä asiakkaalle.

Jatkuva, useasti päivässä tapahtuva kehitystyön integroiminen lopputuotteeseen on välttämätöntä jatkuvan julkaisemisen onnistumisen kannalta. Mahdollisimman pitkälle viety automatisointi edesauttaa tätä, sillä kehittäjien ei tarvitse parhaassa tapauksessa muuttaa omaa ohjelmointitapaansa lainkaan. Automatisointi myös paitsi minimoi inhimillisten erehdysten riskit julkaisu- tai testausvaiheissa, myös antaa oikein konfiguroituna välittömästi palautetta epäonnistuneista koodin muutoksista. Opinnäytetyössä luotu yhteys Githubin versionhallinnan ja Jenkinsin välille toimi loistavasti eikä vaatinut minkäänlaista manuaalista kosketusta rakennustöiden käynnistämiseksi.

Projektissa havaittiin hyväksi SC5:ssä käytetyn Slack-pikaviestimen hyödyntäminen paitsi projektikohtaiseen kommunikaatioon kehittäjien välillä, myös Jenkinsin anta-

man informaation julkaisukanavana, jonne tieto onnistuneesta tai epäonnistuneesta rakennustyöstä kulkeutuu välittömästi.

Laajemmissa projekteissa tällaiseen pikaviestimeen saattaisivat tulla vaikkapa ohjelmistokoodiin tehdyt vetopyynnöt (eng. pull requests) tai palvelimella tapahtuneet poikkeustilanteet, kuten hälyttävän korkea resurssien käyttöaste tai suoranaiset ohjelmistossa tapahtuneet virheet. Valitettavasti nämä seikat jäivät tämän opinnäytetyön ulkopuolelle.

Vastauksena luvussa 2 esitetyistä tutkimuskysymyksistä ensimmäiseen, julkaisuputken rakentamiseen SC5:n puitteissa, löydettiin toimiva ratkaisu, jossa avainkohtina esiin nousivat Jenkinsin merkitys rakennuspalvelimena sekä Docker-pohjainen virtualisointitapa ohjelmistojen paketoimiseen ja julkaisemiseen.

Lienee tosin todennäköistä, että asiakkaat, joille ohjelmistoja kehitetään, eivät ole vielä lähivuosina siirtymässä laajamittaiseen Docker-virtuaalikonttien käyttöön omien palvelinratkaisujensa kanssa, jolloin lopullinen julkaisuaskel pitää ratkaista aina tapauskohtaisesti. Tässä esitetty ratkaisu on kuitenkin hyvä lähtökohta jatkuvan julkaisemisen mahdollistamiselle ja toimii todistetusti etenkin SC5:n omien kehitysympäristöjen puitteissa loistavasti. Viimeisen julkaisuaskeleen käsittely tarjoaa jatkotutkimukselle oivan mahdollisuuden.

Toiseen työssä esitettyyn tutkimuskysymykseen – kehitysympäristöjen muovaamiseksi sellaisiksi, että ne ottavat loppusijoituspaikan huomioon – löydettiin muutamiakin mahdollisia ratkaisuja, joista mikään ei tuntunut ylivoimaisesti toistaan paremmalta. Kehitysprojektissa päädyttiin rakentamaan kehitysympäristö Vagrantin ja Virtualboxin avulla, mutta kehittäjän omista preferensseistä ja projektin luonteesta riippuen myös puhtaasti Docker-pohjainen kehitysympäristö on aivan yhtä mahdollinen.

Tekijä onkin taipuvainen ajattelemaan, että ratkaisu kehitysympäristön virtualisoimiseen on asia, jonka kehitystiimi kykenee annetuista vaihtoehdoista valitsemaan itsenäisesti osaamistason ja käytettävien välineiden puitteissa. Tärkein seikka kuitenkin

on, että jokin virtualisointitapa on käytössä sen sijaan, että ohjelmistot asennettaisiin suoraan kehittäjien omille työkoneille.

7 Pohdinta

Tätä kirjoitettaessa sana "Devops" tuntuu olevan kuuma ja trendikäs ilmaisu, joka pyörii jokaisen teknologian kärjessä kulkevan markkinamiehen huulilla. Tähän on varmasti monta syytä, joista vähäisimpänä ei ainakaan kirjoittajan mielestä ole ohjelmistojen voimakas suuntautuminen verkkoympäristöihin. Perinteisten työpöytäkoneiden ja niille asennettavien ohjelmistojen kultakausi on väistämättömästi katkennut huippunopeiden verkkoyhteyksien myötä. Yhä useammin ohjelmistoja käytetään suoraan Internetin ylitse jotakin web-selainta käyttäen, ja on vaikeata kuvitella tämän kaltaisen kehityksen aivan lähitulevaisuudessa päättyvän.

Kun ohjelmistoja ei tarvitse enää julkaista fyysisesti tai lainkaan edes tallentaa asiakkaiden omille laitteille, pystytään luomaan uusia toimintamalleja, joissa ohjelmistojen ympäristöt ovat poikkeuksellisen tarkasti rajatut. Useat suuret ohjelmistotalot ovat jo huomanneet edut toiminnassa, jossa kokonaisten tuotteiden sijaan käyttäjät ostavat käyttöaikaa ja lisenssejä ohjelmistoihin. Tämän vastineeksi koko ohjelmisto säilyy valmistavan tahon hallinnassa ja käytettävissä vain verkkoyhteyden ylitse. Kuluttaja hyötyy tästä saamalla aina viimeisimmän version käyttöönsä ilman erillistä maksua.

Ohjelmistoja valmistavan tai julkaisevan tahon näkökulmasta asetelma on monella tavalla herkullinen. Nopeasta julkaisusta on hyötyä paitsi bugien ja tietoturvan korjaamiseen, myös kilpailuetuna ja käyttäjien koukuttamiseen uusilla, jatkuvasti lisääntyvillä ominaisuuksilla.

Tässä opinnäytetyössä on käyty lävitse perusperiaatteet jatkuvan julkaisemisen mahdollistamiseen yksinkertaisessa ohjelmistokehitysprojektissa, mutta periaatteiden laajentaminen ja skaalaaminen on usein välttämätöntä täysimittaisissa kehitysprojekteissa. Tämä on monella tavalla haastavaa, eikä esimerkiksi Docker-konttien laajamittaiseen, palvelintenväliseen julkaisemiseen ole vielä edes *de facto* - tasoista standar-

dia. Kilpailijoita esimerkiksi virtuaalikonttien orkestrointiin palvelimille löytyy jo muutamia, mutta tulevaisuus tulee näyttämään, mikä näistä saavuttaa kehittäjien keskuudessa suosikin aseman.

Pilvipalvelualustat, kuten Amazon Web Services tai Rackspace Cloud, tuovat oman mausteensa monimutkaisten palvelin-infrastruktuurien rakentamiseen ja hallintaan. Nämä tarjoavat tehokkaita ja monipuolisia työkaluja palvelinten ja näillä toimivien ohjelmistojen hallintaan, ja lienee mahdollista, että monet yritykset ulkoistavatkin omat palvelimensa tällaisien toimijoiden haltuun. Tärkeimpänä kehitystä jarruttavana seikkana on todennäköisesti kuitenkin luottamus siihen, että yrityksen toiminnalle kriittinen data ei päädy kenenkään muun hallintaan.

Ohjelmistokehityksen painopisteen voimakas siirtyminen Internetiin kannustaa ohjelmiston kehittäjiä myös siirtymään palvelukeskeisiin infrastruktuuriratkaisuihin, joissa ohjelmistot koostuvat pienistä, keskenään kommunikoivista palasista yksittäisen monoliittisen ratkaisun sijaan. Tällöin ohjelmiston yksittäisen osasen päivittäminen ja hallittavuus paranevat merkittävästi, sillä yksittäisiä ohjelmien osia voidaan julkaista, muokata ja skaalata paljon yksinkertaisemmin kuin koko suurta ohjelmistoa. Ohjelmiston osat keskustelevat toistensa kanssa sovittuja rajapintoja käyttäen, eikä erillisten osien tarvitse tietää muiden toiminnasta mitään muuta.

Tällaisessa mikropalveluksi kutsutussa mallissa ohjelmiston kehitys on merkittävästi erilaista kuin perinteisissä ohjelmistoissa ja sisältänee toki omat erityishaasteensa ratkaistaviksi, mutta se tarjoaa myös mielenkiintoisia mahdollisuuksia. Esimerkiksi minkä tahansa yksittäisen osan voi ohjelmoida millä tahansa kielellä ja suorittaa minkälaisessa (virtuaali)ympäristössä tahansa. Saattaa siis olla, että käyttäjän kirjautuminen hoidetaan node.js-pohjaisella järjestelmällä ja palvelun etusivu haetaan järjestelmästä, joka on koodattu Go:lla.

Opinnäytetyössä saavutettujen tulosten yleistäminen toimeksiantajan ulkopuoliseen ohjelmistokehitykseen lienee jossakin määrin perusteltua, vaikka osa käytetyistä tekniikoista onkin SC5:n sisäisen kehityksen tulosta. Perusperiaate julkaisuputken rakentamisesta ja sen sisältämistä askelista on kuitenkin suhteellisen teknologia-

riippumatonta ja soveltuu niin raskaiden työpöytäohjelmistojen kuin ketterien web-sivustojenkin valmistamiseen.

Julkaisuputkea voidaan ajatella laajennettuna työkaluketjuna (eng. tool chain), jonka lävitse käytännössä kaikki ohjelmistokehityksen tuotteet jossakin muodossa kulkevat. Käytettävissä olevista teknologioista, prosesseista ja toimintarajoituksista riippuen julkaisuputki voi tosin olla täydellisen erinäköinen kuin tässä työssä esitetty, mutta luvun 3 yleiskatsauksessa esitetyt osa-alueet löytynevät kustakin ratkaisusta aina jossakin muodossa.

Jatkuvan julkaisun toteuttaminen sisältää hyvin paljon suunnittelua ja haasteita. Vaikka ohjelmistoa julkaistaisiin jatkuvasti, on päivitysten saattaminen palvelimelle silti yhä paikoittain monimutkaista. Toiminnassa olevan ohjelmiston on usein säilytettävä päivityksenkin aikana toimintakykyisenä ja käytettävissä, jolloin julkaisuputken käyttämien menetelmien suunnittelu vaatii lukuisien yksityiskohtien huomioimista. Tähän on olemassa useita eri tapoja, ja nämä ovat käytännössä aina tapauskohtaisesti harkittavia, joskin joitakin yleistapauksia voidaan toki hahmottaa.

Pääsääntönä kuitenkin voidaan pitää periaatetta, jonka mukaan kaikki päivitykset ovat sitä helpompia toteuttaa mitä vähemmän koodipohja tai ympäristö on edellisen päivityksen jälkeen muuttunut. Tämä on jatkuvan julkaisemisen kulmakivi, joka nojaa vankasti automatisoinnin ja laadunvalvonnan vertauskuvallisiin jalkoihin ja varmistaa tuotteen onnellisen päätymisen loppukäyttäjän nautittavaksi.

Lähteet

Allspaw J. & Massie, M. 2010. Infrastructure and Application Metrics. Teoksessa Web Operations: Keeping the Data on Time. Toim. J. Allspaw ja J. Robbins. O'Reilly Media, Inc., 21–48.

Geerling, J. 2015. Ansible for DevOps. Leanpub.

Goldratt, E. M. 1984. The Goal. North River Press.

Hashimoto, M. 2013. Vagrant: Up and Running. O'Reilly Media, Inc.

Hochstein, L. 2015. Preview Edition of Ansible: Up and Running. O'Reilly Media, Inc.

Humble, J., Farley, D. 2011. Continuous Delivery. Pearson Education, Inc.

Hüttermann, M. 2012. Devops For Developers. Apress.

Jacob, A. 2010. Infrastructure As Code. Teoksessa Web Operations: Keeping the Data on Time. Toim. J. Allspaw ja J. Robbins. O'Reilly Media, Inc., 65–80.

Ries, E. 2010. Continuous Deployment. Teoksessa Web Operations: Keeping the Data on Time. Toim. J. Allspaw ja J. Robbins. O'Reilly Media, Inc., 49–64.

Smart, J. F. 2011. Jenkins: The Definitive Guide. O'Reilly Media, Inc.

Turnbull, J. 2014. The Docker Book: Containerization is the new Virtualization. James Turnbull.

Barbier, J. 2014. IT'S HERE: DOCKER 1.0. Viitattu 10.3.2015. http://blog.docker.com/2014/06/its-here-docker-1-0/

Butler-Cole, B. 2013. Rethinking building on the cloud: Part 4: Immutable Servers. Viitattu 28.4.2015. http://www.thoughtworks.com/insights/blog/rethinking-building-cloud-part-4-immutable-servers

Edwards, D. 2010. What is DevOps? Viitattu 11.11.2014. http://dev2ops.org/2010/02/what-is-devops/

Fowler, M. 2006. Continuous Integration. Viitattu 29.1.2015 http://martinfowler.com/articles/continuousIntegration.html

Fowler, M. & Lewis, J. 2014. Microservices. Viittattu 28.4.2015. http://martinfowler.com/articles/microservices.html

Best practices for writing Dockerfiles. N.d. Artikkeli Docker.com:in sivuilla. Viitattu 10.3.2015. https://docs.docker.com/articles/dockerfile_best-practices/

Liitteet

Liite 1. Dockerfile

```
FROM sc5io/ubuntu:14.04.1b
ENV BINDHOST 0.0.0.0
ENV PORT 9000
ENV NODE_ENV development

EXPOSE 9000

# Install MongoDB and Supervisor
# (latter in order to start both Node and Mongo)
RUN apt-key adv --keyserver \
hkp://keyserver.ubuntu.com:80 --recv 7F0CEB10
RUN echo "deb http://repo.mongodb.org/apt/ubuntu \
"$(lsb_release -sc)"/mongodb-org/3.0 multiverse" \
| tee /etc/apt/sources.list.d/mongodb-org-3.0.list
RUN apt-get update
RUN apt-get install -y mongodb-org supervisor
RUN mkdir -p /data/db

## Copy the repository files and install app dependencies
COPY . /app
WORKDIR /app
COPY supervisord.conf \
/etc/supervisor/conf.d/supervisord.conf

RUN rm -rf node_modules/
RUN npm install -g grunt-cli
RUN npm install -g bower
RUN npm install
RUN bower install --allow-root --config.interactive=false
```

```
# Build and run production version
RUN grunt build
ENV NODE_ENV production

WORKDIR /app/dist
CMD ["/usr/bin/supervisord"]
```

Liite 2. Vagrantfile

```
# Vagrantfile API/syntax version.
VAGRANTFILE_API_VERSION = "2"

MACHINE_HOSTNAME = "projekti"

Vagrant.configure(VAGRANTFILE_API_VERSION) do |config|

  # We use Ubuntu 14.04 as a box
  config.vm.box = "ubuntu/trusty64"

  # Set shared folder
  config.vm.synced_folder ".", "/app"

  # We set up private network address
  # we can use to test image
  config.vm.network :private_network, ip: "192.168.2.2"

  # Set machine "physical" attributes
  config.vm.provider :virtualbox do |v|
    v.customize ["modifyvm", :id, "--name", MA-
CHINE_HOSTNAME]
    v.customize ["modifyvm", :id, "--
natdnshostresolver1", "on"]
    v.customize ["modifyvm", :id, "--memory", 1024]
    v.customize ["modifyvm", :id, "--cpus", 2]
    v.customize ["modifyvm", :id, "--ioapic", "on"]
```

```
  end

  config.vm.provision :shell,
    :keep_color => true,
    :inline => "export PYTHONUNBUFFERED=1 && export ANSI-
BLE_FORCE_COLOR=1 && cd /app && ./vagrant_init.sh"

  # Set up hostname of the machine
  config.vm.define MACHINE_HOSTNAME do |machine|
    machine.vm.hostname = MACHINE_HOSTNAME
  end
end
```

Liite 3. Ansible-käsikirja

```
---
- hosts: projectname

  sudo: False

  tasks:

  - name: Get mongodb repository key
      apt_key:
        keyserver: "hkp://keyserver.ubuntu.com:80"
        id: "7F0CEB10"
      sudo: True

  - name: Add mongodb repository to apt
    apt_repository:
      repo: "deb http://repo.mongodb.org/apt/ubuntu
trusty/mongodb-org/3.0 multiverse"
    sudo: True

  - name: Setup node.js repository
```

```
    shell: "curl -sL https://deb.nodesource.com/setup |
bash -"
    sudo: True

  - name: Install system packages
    apt:
      pkg: "{{ item }}"
      state: installed
      update_cache: yes
    with_items:
      - "mongodb-org"
      - "nodejs"
      - "git"
      - "ruby"
      - "ruby-dev"
    sudo: True

  - name: Install gems
    shell: "sudo gem install compass"
    sudo: False
      - name: Install latest npm
    npm:
      name: npm
      global: yes
      state: latest
    sudo: True

  - name: Make sure data/db directory exists
    file:
      dest: /data/db
      state: directory
    sudo: True

  - name: Install global dependencies
    npm:
      global: yes
```

```
      name: "{{ item }}"
    with_items:
      - "grunt-cli"
      - "bower"
    sudo: True

  - name: Install application dependencies
    npm:
      path: "/app"

  - name: Run bower install
    command: "bower install --allow-root --
config.interactive=false"
    args:
      chdir: "/app"

  - name: Run grunt build
    command: "grunt build"
    args:
      chdir: "/app"
  - name: Make sure mongod is running
    service:
      name: mongod
      state: started
    sudo: True
```

Liite 4. Vagrant initfile

```
#!/usr/bin/env bash

# Note: this script is tested only on Ubuntu 14.04 system
if [ $(dpkg-query -W -f='${Status}' ansible 2>/dev/null |
grep -c "ok installed") -eq 0 ];
then
  echo "Add APT repositories"
```

```
  export DEBIAN_FRONTEND=noninteractive
  apt-get install -qq software-properties-common \
&> /dev/null || exit 1
  apt-add-repository ppa:ansible/ansible \
&> /dev/null || exit 1

  apt-get update -qq
  echo "Installing Ansible"
  apt-get install -qq ansible &> /dev/null || exit 1
  echo "Ansible installed"
fi

cd /app
ansible-playbook ansible_playbook.yml --connection=local
```