Elias Törmänen

JATKUVA INTEGRAATIO, JULKAISU JA TOIMITUS MOBIILISOVELLUSKEHITYKSESSÄ

CASE OBSERVIS OY

Opinnäytetyö
Tietojenkäsittely

2018

XAMK
Kaakkois-Suomen ammattikorkeakoulu

Tekijä/Tekijät	**Tutkinto**	**Aika**
Elias Törmänen	Tradenomi	Marraskuu 2018
Opinnäytetyön nimi		
Jatkuva integraatio, julkaisu ja toimitus mobiilisovelluskehityksessä: Case Observis Oy		33 sivua
Toimeksiantaja		
Observis Oy		
Ohjaaja		
Janne Turunen		

Tiivistelmä

Tämän opinnäytetyön tavoitteena on jatkuvan integraation, julkaisun ja toimituksen (lyhennettynä CI/CD, eli continuous integration, delivery and deployment) liittäminen osaksi Observis Oy:n mobiilisovelluskehitystä. Tarkoituksena on automatisoida sovelluksen julkaisuun menevä aika mahdollisimman lyhyeksi. Opinnäytetyön tuotoksena syntyy esimerkkimalli toimivasta CI/CD-mallista yhdelle mobiilisovellukselle, jota voidaan käyttää myöhemmin hyödyksi tulevissa projekteissa.

Raportti alkaa teoriaosuudella, jossa tutustutaan yleisiin jatkuvan integraation, julkaisun ja toimituksen periaatteisiin sekä niihin oleellisesti liittyviin käsitteisiin, kuten erilaisiin CI-palveluihin, testaamiseen, versionhallintaan ja kooditarkistukseen. Lisäksi teoriaosuudessa käydään läpi yleistä teoriaa erilaisista toimitus- ja testiympäristöistä sekä mobiilisovelluskauppojen ominaisuuksista.

Teoriaosuuden jälkeen siirrytään opinnäytetyön käytännön osuuteen, jossa kerrotaan yhden käytössä olevan sovelluksen pääasiallinen arkkitehtuuri sekä sen muutosprosessi jatkuvan integraation ja toimituksen malliin sopivaksi. Tässä osuudessa selitetään myös valmiin prosessin vaiheet ja toiminta sovelluksen virheenkorjauksesta kauppaan julkaisemiseen asti. Lisäksi tässä osuudessa käydään läpi jatkuvan integraation ja toimituksen käyttöönottamisessa tulleita ongelmia, sovelluskehityksen automaation nykytilaa yrityksellä sekä jatkokehitysmahdollisuuksia.

Opinnäytetyön lopputoteutus täyttää toimeksiantajan asettamat tavoitteet. Lopputuotoksena syntynyttä esimerkkimallia tullaan hyödyntämään tulevien mobiilisovellusten integroimisessa osaksi CI/CD:tä.

Asiasanat

integrointi, mobiilisovellukset, automaatio, jatkuva julkaisu

XAMK
South-Eastern Finland University of Applied Sciences

Author (authors)	Degree	Time
Elias Törmänen	Bachelor of Business Administration	November 2018

Thesis title

Continuous integration, deployment and delivery in mobile application development: Case Observis Oy

33 pages

Commissioned by

Observis Oy

Supervisor

Janne Turunen

Abstract

The main objective of this thesis was to provide an example of continuous integration, deployment and delivery (later CI/CD) practices as a part of the mobile application development process in Observis Oy. The aim was to the automate amount of time spent on software delivery to be as little as possible. This thesis provided a working example of continuous integration and deployment for a single mobile application that could be later utilized in other projects.

The report began with a theory part where core principals of CI/CD were introduced, as well as many related topics, such as continuous integration services, testing, version control and code inspection. Some theory about deployment environments and mobile application store features were also summarized.

A basic layout of one mobile application was explained in the practical part, along with the transferring process to the continuous integration, deployment and delivery practices. In this part the phases of a working process and its functionality from a bugfix to a release were also discussed. Problems with the process and adopting the continuous integration, delivery and deployment practices were described as well as the current stages of CI/CD in the company practices, and future insights and improvements.

The practical part of the thesis met the specifications given by the contractor. The produced example will be used in adopting CI/CD for the development process of future mobile applications.

Keywords

integration, mobile applications, automation, continuous delivery

SISÄLLYS

1 JOHDANTO 5
2 YLEISTÄ TUOTANTOPROSESSIN AUTOMAATIOSTA 6
2.1 Jatkuva integraatio 7
2.2 Jatkuva julkaiseminen ja jatkuva toimitus 9
3 CI -TYÖKALUT JA TOIMITUSYMPÄRISTÖT 11
3.1 CI-palvelimet 11
3.2 Fastlane 13
3.3 Versionhallinta 14
3.4 Jatkuva kooditarkistus 16
3.5 Testaus 17
3.6 Toimitusympäristöt 20
3.7 Sovelluskaupat 22
4 CASE OBSERVIS 24
4.1 Prosessi 24
4.2 Esimerkkitoteutus 27
4.3 Ongelmat 31
5 PÄÄTÄNTÖ 32

1 JOHDANTO

Tämän opinnäytetyön aiheena oli kehittää jatkuvan integraation ja julkaisun malli mobiilisovelluksien kehitykselle. Toimeksiantajana opinnäytetyössä oli Observis Oy, ja opinnäytetyön käytännön osuus on toteutettu erityisesti Observis Oy:n mobiilikehitystä ja julkaisua varten. Opinnäytetyön lopputuloksesta tulee esimerkkimalli yrityksen mobiilisovellusten jatkuvaa integraatiota ja julkaisua varten, ja sitä hyödynnetään myös muiden kuin esimerkissä käytetyn sovelluksen julkaisussa.

Luvussa kaksi käydään läpi jatkuvan integraation, julkaisun ja toimituksen yleistä teoriaa. Luvussa määritellään käsitteet ja niiden erot sekä kerrotaan myös tämän prosessin hyötyjä ja haittoja sekä prosessin käyttöönotossa mahdollisesti ilmentyviä vaikeuksia tai ongelmia.

Kolmannessa luvussa käydään läpi erilaisia työkaluja, joita tämän prosessin käyttöönottamisessa voidaan hyödyntää. Tässä luvussa kerrotaan sekä yleisistä, lähes kaikkiin projekteihin sopivista työkaluista, kuten jatkuvan integraation palvelimista, että myös mobiilisovelluksiin keskittyvistä työkaluista. Luvussa kerrotaan myös teoriaa testaamisesta ja kooditarkistuksesta sekä versionhallinnasta, jotka ovat melko oleellisia asioita jatkuvan integraation ja julkaisun kannalta. Viimeisenä luvussa käydään läpi myös toimitusympäristömallit sekä teoriaa mobiilisovellusten julkaisualustoista, eli sovelluskaupoista ja niiden ominaisuuksista.

Neljäs luku keskittyy esimerkkitoteutukseen. Luvussa käydään läpi jatkuvan integraation ja toimituksen työkalujen asennus, itse prosessin yksityiskohtainen esimerkkitoteutus sekä prosessissa ilmenneet vaikeudet ja ongelmat. Lisäksi luvussa kuvaillaan Observis Oy:n automaation nykytilaa ja tulevaisuudennäkymiä.

2 YLEISTÄ TUOTANTOPROSESSIN AUTOMAATIOSTA

Jatkuvalla integraatiolla (continuous integration, myöhemmin CI) tarkoitetaan ohjelman koostamisen (build) ja testien automaatiota kehittäjän tehdessä muutoksia (commit) versionhallintaan. CI rohkaisee kehittäjiä yhdistämään koodimuutokset ja yksikkötestit yhteiseen koodivarastoon aina, kun jokin uusi sovelluksen osa tai korjaus on valmis. Koodin siirto koodivarastoon laukaisee automaattisen järjestelmän, joka nappaa viimeisimmän version koodista ja kääntää, koostaa, testaa sekä tarkistaa sen (Guckenheimer 2017).

Jatkuva julkaisu (continuous delivery) tarkoittaa puolestaan muutosten julkaisua tuotantoversioon. Jatkuva julkaisu ei ole täysin automaattinen, sillä sitä hyödynnettäessä muutokset julkaistaan ensin testi- tai hyväksyntäympäristöön, jossa testaajat ja asiakas tarkistavat ja hyväksyvät muutokset ja testaavat toiminnallisuudet manuaalisesti (Rossel 2017, 18-19).

Jatkuva Integraatio	Jatkuva Julkaisu	Jatkuva Toimitus
- Koodin integrointi tiheään tahtiin - Master-haara lähdekoodivarastossa - Toimivan käännöksen ylläpito koko ajan - Tiimit kommunikoivat integraatiosta - Käännökset ovat automatisoitu - Integraatiopalvelinta käytetään koodin kääntämiseen ja testaamiseen	Jatkuva integrointi sekä - Automatisoitu julkaisu jokaiseen ympäristöön milloin tahansa - Jokainen muutos on mahdollista päivittää suoraan tuotantoon - Automaattinen julkaisuprosessi hoitaa käännökset - Tuotantojulkaisut käynnistetään manuaalisesti	Jatkuva julkaisu sekä - Automaattinen julkaisu tuotantoon - Kaikki muutokset julkaistaan tuotantoon

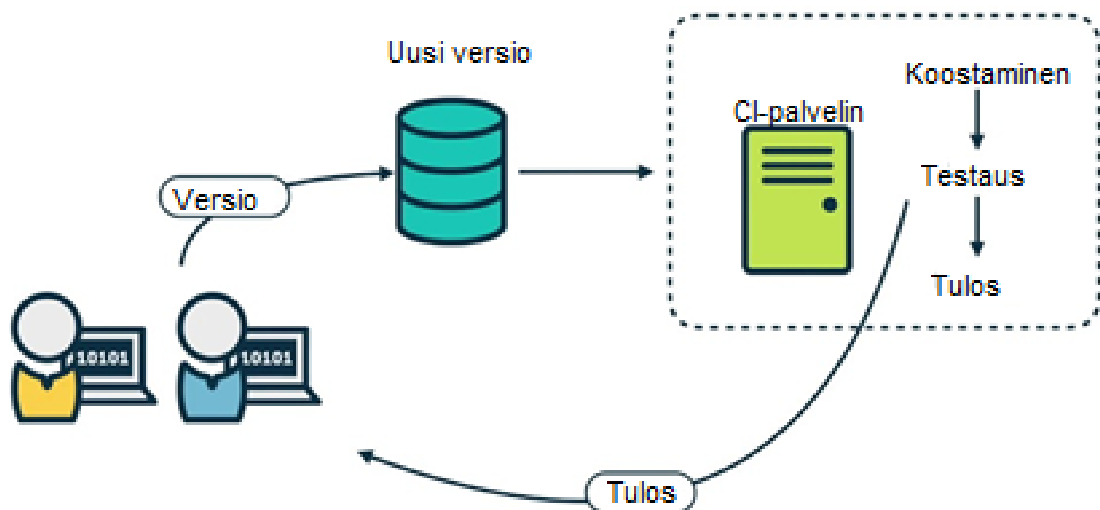
Kuva 1. Jatkuva integrointi, julkaisu ja toimitus

Jatkuvaa toimitusta (continuous deployment) hyödynnetään silloin, kun asiakkaalla ei ole tarvetta tarkistaa jokaista muutosta itse, vaan muutokset voidaan suoraan ottaa käyttöön tuotannossa onnistuneen koostamisen jälkeen. Kuvasta 1 voidaan havaita jatkuvan integroinnin, julkaisun ja toimituksen perusperiaatteet. Jatkuvaa julkaisua ja toimitusta kutsutaan myöhemmin myös lyhenteellä CD, ja koko prosessia CI/CD.

2.1 Jatkuva integraatio

Perinteisissä kehitysmalleissa integraatio on yleensä projektin loppupuolella kerralla tehtävä suoritus, josta tulee helposti pitkä ja vaikea prosessi, erityisesti jos integraatio ei ole ollut mielessä jo heti projektin alusta asti (Poimala, ym). Ohjelmistokehitysprosessin suurimmat riskit tulevatkin erillään kehiteltyjen komponenttien yhdistämisyrityksistä, jossa ne eivät toimikaan yhteen halutulla tavalla. Pahimmillaan integraatio-ongelmat voivat johtaa jopa koko projektin peruuntumisen erityisesti, jos ohjelmisto on monimutkainen ja kriittiset osat kehitetään ja testataan erillään muista osista (Vilkman 2010, 4).

Fowlerin (2006) mukaan CI tarkoittaa ohjelmistokehityksen käytäntöä, jossa ohjelmistoa integroidaan jatkuvasti kehityksen aikana, ja kehittäjät tekevät useita muutoksia versioon päivittäin. Jokaisen muutoksen jälkeen projekti tulisi myös koostaa ja testata, jotta varmistutaan ohjelmiston olevan käyttökelpoinen ja varmistetaan ympäristön toiminta ennen uusien muutosten kehitystä (kuva 2).



Kuva 2. Jatkuva integraatio (Prince 2016, mukaillen)

Projektin koostaminen tulisi aina automatisoida mahdollisimman pitkälle, sillä se tavallisesti sisältää monta askelta, kuten koodin kääntäminen (compile), jolla koodi yleensä muutetaan kevyempään, tietokoneen luettavaan muotoon, tiedostojen siirtelyä, tietokannan pystyttämistä jne. Kummallisilta tuntuvien komentorivikomentojen kirjoittelu sekä dialogien klikkailu on yleensä vain ajan haaskuuta, ja voi johtaa turhiin virheisiin. Yksi yleinen virhe on automaation

jättäminen puolitiehen, esimerkiksi tietokannan pystyttämisen automatisointi unohtuu täysin (Fowler 2006).

CI on suosiota kasvattava tekniikka, koska varsinainen kehitystyö tapahtuu yleensä yksin. Kehittäjien tulisi integroida muutoksensa osaksi tiimin koodipohjaa (codebase) säännöllisesti useita kertoja päivässä. Jotkin CI:tä hyödyntämättömät projektit saattavat integroida kehittäjien tai kehitystiimien tekemät muutokset jopa viikkojen tai kuukausien kehityksen jälkeen (Laukkanen, ym. 2015, 56). Jos muutoksia aletaan odottamaan pitkiä aikoja, tuloksena on helposti paljon yhdistämiskonflikteja (merge conflict). Lisäksi mahdollisten bugien löytäminen on huomattavasti haastavampaa, sillä koodia joudutaan tarkastelemaan käsin jopa viikkoja taaksepäin. Tuloksena on vain tuplasti työtä, ja paljon hukattua työaikaa (Guckenheimer 2017).

CI:tä hyödyntävissä yrityksissä työntekijöitä rohkaistaan myös lisäämään yhteistyötä, sillä jokainen kehitystiimin jäsenen tekemä muutos tulee näkyviin muille (Guckenheimer 2017). CI palvelimen (esim. Jenkins tai cruise) verkkosivuilta löytyy tiedot koostamisen onnistumisesta, muutosten tekijästä, tiedot muutoksista sekä muutoshistoria (Timofeev 2018).

Jatkuvaan integraatioon ei ole yhtä ennalta määriteltyä toteutustapaa, vaan sen kattavuus voidaan tehdä yrityksen jo olemassa olevien toimintatapojen mukaisesti ja sitä voidaan alkaa käyttämään askel kerrallaan. Fowler (2006) suosittelee aloittamaan koodin koostamisen automaatiolla, eli yhdellä komennolla voidaan kääntää koko koodi alusta loppuun. Tämä voi olla suurikin työ projektista riippuen, mutta se on kriittinen osa jatkuvaa integraatiota. Toinen tärkeä osa on automaattiset testit, jotka edistävät toimivan ohjelmiston koostamista. Ohjelmistoon pitäisi saada ainakin muutama automaattinen testi testaamaan kriittisimpiä ja virhealttiimpia osia (Fowler 2006).

Ohjelmiston koostamisen ei tulisi kestää muutamaa kymmentä minuuttia kauempaa, mutta suuressa, vanhassa projektissa muutama tunti on jo hyvä alku, jos koostaminen on ennen kestänyt pitkään. Uusien projektien parissa kymmenen minuuttia on hyvä tavoite, ja jos koostaminen kestää pidempään, koodin uudelleenkirjoitus joiltain osin on hyvä idea (Fowler 2006).

2.2 Jatkuva julkaisu ja jatkuva toimitus

Jatkuvan julkaisun pääideana on vähentää aikaa ja vaivaa sovelluksen saamisessa tuotantoon. Ilman jatkuvan julkaisun periaatteiden hyödyntämistä julkaisuprosessiin voi mennä pahimmillaan jopa tunteja turhaa työaikaa, minkä vuoksi uusien ominaisuuksien viemistä tuotantoon voi helposti lykätä ikuisuuksia, mikä lisää vuorostaan uusien virheiden määrää ja heikentää laatua (Rossel 2017, 18-19).

Laukkanen ym. (2015, 55) määrittelee jatkuvan julkaisun ohjelmistokehityksen menetelmäksi, jossa ohjelmisto pidetään siinä tilassa, että se voitaisiin julkaista milloin tahansa käyttäjille. Se vähentää myös julkaisun riskejä, mahdollistaa uskottavan edistymisen seurannan sekä nopean palautteen käyttäjiltä.

Laukkanen ym. (2015, 57) määrittelee jatkuvaan julkaisuun siirtymisen yrityksessä siihen liittyvien vaiheiden mukaan, sillä itse prosessi ja sen käyttöön ottaminen vaihtelee yrityksestä toiseen niissä jo olemassa olevien menetelmien mukaisesti. Siihen liittyviä käsitteitä ja ominaisuuksia Laukkanen ym. (2015) listaa tutkimuksessaan: Jatkuvan julkaisun ominaisuudet, jatkuvan julkaisun käyttöönottoon liittyvät ongelmat (vaihtelupisteet), käyttöönoton vaiheet, jatkuvan julkaisun aiheuttamat ongelmat sekä jatkuvan julkaisun hyödyt (taulukko 1).

Jatkuvan julkaisun periaatteita hyödyntämällä tuotantoon julkaisuun kuluva aika saadaan lyhennettyä minuutteihin, ja kehittäjällä on merkittävästi vähemmän prosessiin liittyviä asioita muistettavana. Parhaimmillaan koko prosessi saadaan käynnistettyä yhden napin painalluksella tai tiedoston ajamisella, kunhan jatkuvan julkaisun ympäristö on konfiguroitu oikein (Rossel 2017, 18-19).

Taulukko 1. Jatkuvan julkaisun ominaisuudet (Laukkanen, ym. 2015, mukaillen)

Hyödyt	Automaattiset hyväksyntä- ja yksikkötestit, parempi kommunikointi, lisääntynyt tuottavuus sekä projektin ennustettavuus (ongelmien löytäminen aiemmin), lisääntynyt asiakastyytyväisyys, lyhyempi aika kauppoihin, pienempi testialue sekä parantunut julkaisujen luotettavuus ja laatu
Käyttöönoton onnistumisen määrittely	Koostamisen kesto, tiheys ja käynnistys, vikatilojen kesto ja käsittely, integroinnin tiheys, integroinnin kohde, modularisaatio, integraatiota edeltävät prosessit, tilapäivitykset, testien erittely sekä uusien ominaisuuksien testaus
Käyttöönoton ominaisuuksien vaiheet	Testien epäonnistumisten välitön hoitaminen, CD:n asennus monimutkaisille järjestelmille, työnjaon selkeytys, CD jaetuissa ympäristöissä, testivetoisen kehityksen hallitseminen, CD:n aloitus uusissa projekteissa, testiviiveiden pienentäminen, ketterien menetelmien toimitus, useiden julkaisujen rinnakkainen kehitys, automaattinen testaus, infrastruktuuri tehokkaalle koostamiselle, testaukselle sekä julkaisulle
Ongelmat, joihin haetaan ratkaisua	Tekninen velka, matalampi luotettavuus ja testikattavuus, matalampi käyttäjätyytyväisyys, ajanhallintapaineet, vaiva laadunvarmistuksessa
Ominaisuudet	Nopea ja tiheä julkaisu, joustava tuotesuunnittelu ja arkkitehtuuri, jatkuva testaus ja laadunvarmistus, automaatio, konfiguroinninhallinta, toimituksen jälkeiset aktiviteetit, haaroitus ja yhdistäminen, koostaminen ja testaaminen, koostamisjärjestelmä, infrastruktuuri koodina, julkaisu ja toimitus

Rosselin (2017) mukaan jatkuva toimitus puolestaan on viimeinen vaihe ohjelmistokehitysprosessin automaatiossa, sillä muutokset ajetaan automaattisesti ja ohjelma rakennetaan tuotanto- tai sitä vastaavaan testiympäristöön joka kerta kun koodivarastoon lisätään uusi versio. Mitä aikaisemmin tämä hoidetaan, sitä nopeammin voidaan korjata mahdolliset virheet. On paljon helpompi muistaa mahdolliset virheen lähteet eiliseltä kuin kaksi kuukautta sitten tehdyistä muutoksista. Jatkuvaa toimitusta hyödyntämällä jokainen tehty muutos ajetaan siis suoraan tuotantoympäristöön automaattisesti, jos ohjelma koostetaan onnistuneesti ja testit menevät läpi.

Jatkuvassa toimituksessa on myös omat ongelmansa. Vaikka automaattinen tuotantoon ajaminen on hyvin konfiguroidun järjestelmän myötä jopa vähemmän virhealtis kuin manuaalinen, sen toteuttaminen ei aina ole mahdollista. Monissa tilanteissa asiakas tai tuoteomistaja suhtautuu epäluuloisesti automaattiseen järjestelmään (Rossel 2017, 21-22), ja monen projektin parissa tulee vastaan myös fyysiset ongelmat jatkuvan toimituksen kannalta, sillä jotkin projektit vaativat manuaalista käyttöönottoa, esimerkiksi jotkin suljetut projektit, tai sovellukset, jotka täytyy asentaa ilman internetyhteyttä paikan päällä.

3 CI -TYÖKALUT JA TOIMITUSYMPÄRISTÖT

Tässä luvussa käydään läpi mahdollisia CI-palvelimia ja -palveluita julkaisun sekä laatuvarmistuksen (quality control) osalta. Aiheen osalta tutustutaan erityisesti jo Observis Oy:llä käytössä olevaan Jenkinsiin, lupaavaan mobiilisovellusehdokkaaseen Fastlaneen sekä käydään läpi myös muita mahdollisia vaihtoehtoja. Luvussa käsitellään myös hieman versionhallintatyökaluista, joista isoimpana Git:iä. Jatkuvaan integraatioon kuuluu oleellisena osana myös testit sekä koodin ulkoasun ja oikeellisuuden tarkistus (lint).

3.1 CI-palvelimet

CI-palvelin on tavallaan kuin näkymä koodivarastoon. Joka kerta kun uusi koodi varastoidaan, palvelin tarkistaa koodin lähteet integrointikoneelle, käynnistää ohjelman koostamisen ja ilmoittaa koosteen tuloksesta, usein sähköpostitse, mutta myös esimerkiksi Slack- tai Discord-ilmoituksia käytetään. Koodin varastoija ei ole valmis ennen ilmoituksen vastaanottoa. CI-palvelin ei ole välttämätön väline CI:tä hyödynnettäessä, mutta usein se on erittäin hyödyllinen (Fowler 2006).

Erilaisia CI-palvelimia ovat muun muassa CircleCI, Travis, Bamboo ja Jenkins. Pääasiassa CI-palvelimet suoriutuvat tehtävistä yhtä hyvin, ja yrityksen omat toimintavavat ovat suurin vaikuttava tekijä niitä valittaessa ja yrityksen tulisi valita itselleen sopivin vaihtoehto. Jotkin palvelimet tukevat esimerkiksi paremmin joitain ohjelmointikieliä tai koodivarastoja, ja jotkin saattavat olla maksullisia ja jostain puolestaan löytyy parempi tuki lisäosille (Taulukko 2)

Taulukko 2. CI-palvelinten ominaisuudet (Allen 2016, mukaillen)

	Jenkins	Bamboo	Travis	CircleCI
Avoin lähdekoodi	Kyllä	Ei	Kyllä	Ei
Lisäpalikat	Kyllä (monia)	Kalliita	Muutamia	Ei
Tekninen tuki	Kyllä	Vähän (pieni yhteisö)	Ei	Kyllä
Helppokäyttöinen	Ei (lisäpalikat auttavat)	Kyllä	Kyllä	Kyllä
Koodivarastotuki	Kaikki	Kaikki	GitHub	GitHub
Ilmainen yrityskäytössä	Kyllä	Ei	Ei	Ei

Jenkins on Javalla rakennettu itsenäinen, avoimen lähdekoodin automaatiopalvelu, jonka avulla voidaan automatisoida kaikenlaisia tehtäviä koodin koostamiseen, kääntämiseen, testaamiseen, julkaisuun ja käyttöönottoon liittyen (Jenkins s.a). Jenkins itsessään ei tee kauhean paljoa, mutta kaikkeen kuviteltavissa olevaan löytyy lisäosa (plugin), jolla tästä työkalusta saa monipuolisemman (Rossel 2017).

CircleCI on toinen automaatiopalvelu, joka ei ole niinkään sidottu lisäosien käyttöön, vaan kaikki yleisimmät toiminnot ovat sisäänrakennettuna. Jenkinsiin verrattuna CircleCI on nopeampi, helppokäyttöisempi ja rakennettu modernimmilla työkaluilla. CircleCI mahdollistaa sen käyttämisen heidän omilta palvelimiltaan, joten yrityksen ei tarvitse välttämättä hankkia omaa palvelinta CI-käyttöön (circleci.com)

3.2 Fastlane

Fastlanen on kehittänyt Felix Krause, jolta Google osti sen myöhemmin. Kyseessä on työkalu, jolla voidaan automatisoida mobiilisovelluksen julkaisu Applen App Storeen sekä Googlen Play-kauppaan. Fastlane hoitaa sovelluksen kuvakaappaukset, koosteen, koodin allekirjoituksen sekä kauppoihin päivittämisen helposti ja nopeasti yhdellä rivillä koodia (Kuva 3). Fastlane voidaan integroida yli 400 olemassa olevan työkalun tai palvelun kautta, mukaan lukien Jenkins. Fastlane tukee tällä hetkellä vain MacOS:ää xcoden sekä muiden Applelle julkaisussa tarvittavien työkalujen rajoituksista johtuen, mutta ainakin osittainen Windows- ja Linux-tuki on tulossa (Fastlane, s.a).

Kuva 3. Fastlane-julkaisun vaiheet (Fastlane s.a, mukaillen)

Fastlane on tällä hetkellä suosituimpia mobiilisovellusten työkaluja jatkuvan julkaisun kannalta, mutta siinä on useita ongelmia. Yksi suurimpia ongelmia Fastlanen kanssa on se, että se hyödyntää Rubya ja Bundleria, joista monella ”peruskoodarilla” ei ole kovin suurta tietämystä. Fastlanen asennus ja ylläpito on myös melko työlästä sekä aikaa vievää (Jagtap 2018).

Fastlanen asennetaan MacOS:lle hyödyntämällä paketinhallintaohjelmistoa (package manager), joita MacOS:llä on esimerkiksi RubyGems ja HomeBrew.

Paketinhallintaohjelmisto täytyy mahdollisesti päivittää ennen Fastlanen asennusta, sillä siitä saattaa olla asennettuna vanha versio, joka ei tue kaikkia lisäosia. Itse paketinhallinnan lisäksi joudutaan todennäköisesti päivittämään jotain paketteja tai komponentteja. Komentorivi kertoo tavallisesti melko hyvin puuttuvista tai vanhentuneista paketeista, joita tarvitsee lisätä tai päivittää.

3.3 Versionhallinta

Versionhallintatyökalujen avulla pidetään kirjaa muutoksista lähdekoodiin. Versionhallintaohjelmisto pitää kirjaa jokaisesta muutoksesta koodiin erityisessä tietokannassa, ja jos kehittäjä tekee virheen, voidaan verrata nykyistä koodia helposti aiempiin versioihin, ja korjata ongelma häiritsemättä koko tiimin työskentelyä.

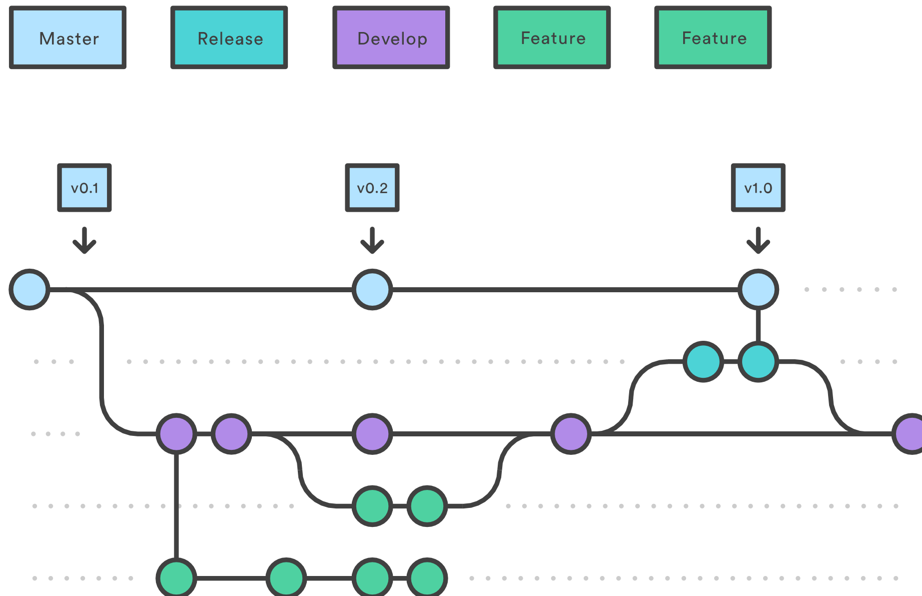
Modernit versionhallintaohjelmat hyödyntävät niin kutsuttua puurakennetta, jossa koodin viimeisin versio on yleensä master-haarassa (branch), ja uusia ominaisuuksia tai bugikorjauksia tehdään sivuhaaroissa, jotka myöhemmin yhdistetään master-haaraan. Ideaalisessa tilanteessa master-haaraa ei suoraan muokata, vaan jokainen kehittäjä tekee muutoksensa omassa haarassaan (What is version control? s.a).

Versionhallintatyökalut ratkaisevat jo yksistään monia koodipohjaan liittyviä muutoksia isommissa tiimeissä, sillä niiden avulla voidaan helposti pitää eri versiot ohjelmistosta järjestyksessä. Versionhallintatyökaluilla pidetään myös huoli siitä, että kaikki tehty koodi on keskenään yhteen sopivaa ja kaikesta on aina varmuuskopio tallessa (What is version control? s.a).

Tämän hetken suosituin versionhallintatyökalu on ilmainen ja avoimeen lähdekoodiin pohjautuva Git, joka hyödyntää ”puurakennetta”, jolloin kehitys tapahtuu aina uudessa haarassa, joka yhdistetään master-haaraan kehitettävän ohjelmiston osan valmistuessa.

Gitin kanssa yleisimpiä työskentelymenetelmiä on ns. Git flow (kuva 4), jossa uudet ominaisuudet kehitetään omissa haaroissaan, jotka yhdistetään develop-haaraan. Julkaisut toteutetaan release-haarasta, joka on johdettu viimeiset

muutokset sisältävästä develop-haarasta. Release-haara yhdistetään sen jälkeen myös master-haaraan, joka nimetään versionumeron mukaisesti. Uusien ominaisuuksien kehitystä voidaan jatkaa develop-haarasssa, ja julkaisun hiomista release-haarassa, jos siellä on vielä jotain ongelmia (Gitflow workflow s.a).



Kuva 4. Git flow (Gitflow workflow s.a)

Toinen esimerkki versionhallintatyökalusta on SVN (Subversion), joka puurakenteen sijasta hyödyntää runkorakennetta (trunk). Runkorakenteisessa versionhallinnassa ei työskennellä haaroissa, vaan kaikki muutokset tehdään paikallisessa työasemassa ja integroidaan osaksi samaa versiota.

Erilaisia Git-palveluja on saatavilla verkossa monia. Näitä palveluja tarjoavat esimerkiksi BitBucket, GitHub sekä GitLab. Harrastelijoiden ja avoimien lähdekoodien suosiossa on yleensä GitHub, mutta monet yritykset hyödyntävät usein yksityisissä projekteissaan omalle palvelimelleen konfiguroitua GitLabia tai BitBucketia.

Versionhallinnalla on suuri merkitys jatkuvan integroinnin ja julkaisun kanssa. Pelkän koodin lisäksi myös lähes kaikki konfigurointitiedostojen, käännösskriptien (build script) ja testien tulisi kuulua osaksi versiota. Lisäksi esimerkiksi kehitysympäristöjen (IDE) asetustiedostojen tulisi kuulua osaksi versiota, jos kehitystiimillä on sama IDE käytössä, jotta samat asetukset olisi helppo ottaa muillakin myöhemmin käyttöön (Fowler 2006).

Fowler (2006) määrittelee version sisällön hyväksi nyrkkisäännöksi sen, että pitäisi olla mahdollista mennä täysin konfiguroimattomalle tietokoneelle, ottaa versionhallintatyökalulla uusin versio, jonka avulla projektin saisi täysin ajettua. Jotkin asiat pitää tosin olla tietokoneessa valmiina, kuten käyttöjärjestelmä, tietokanta sekä kehitysympäristö, sillä niiden asennus scriptejä hyödyntämällä olisi turhan työlästä.

3.4 Jatkuva kooditarkistus

Koodin tarkistamisella (inspection) tarkoitetaan muun muassa koodin ulkoasun, luettavuuden sekä oikeellisuuden tarkistusta. Useissa ohjelmointikielissä, esimerkiksi Javassa, on jo itsessään omat yleiset kirjoituskäytännöt, kuten esimerkiksi niin kutsutun camelcasen käyttö metodeissa ja muuttujissa (pieni alkukirjain, yhdyssanojen myöhemmät sanat aloitetaan isolla sanan keskellä), sisennyskäytännöt, tai kommenttien rakenne (Java Code Conventions 1997). Myös yrityksissä on yleensä omat käytäntönsä nimeämisiä ja sisennyksiä varten, sillä ne helpottavat koodin lukemista ja ymmärtämistä muille tiimin jäsenille.

Kehitysympäristöissä on yleensä jo mukana ns. lint-tiedosto, joka kertoo kehitysympäristölle halutut ohjelmointitavat. Lint-tiedosto varoittaa esimerkiksi syntaksivirheistä, alustamattomien muuttujien käytöstä, vanhentuneiden metodien kutsuista sekä muotoilukäytännöistä (Arcanist User Guide: Lint s.a). Lint-tiedostoon konfiguroidaan yleensä myös yrityksen tai projektin yhteiset käytännöt, ja se lisätään koodivarastoon muiden käyttöön, jottei jokaisen kehittäjän tarvitse tehdä omaa linttiä, ja jotta se olisi kaikilla kehittäjillä yhdenmukainen. Lint-tiedostoon voidaan myös joissain tapauksissa lisätä automaattinen korjaus, jolloin yleisimmät tyylivirheet, kuten puuttuvat pilkut, voidaan korjata automaattisesti (Arcanist User Guide: Lint s.a).

Esimerkiksi TypeScriptin (ohjelmointikieli, jolla voidaan kirjoittaa JavaScriptiä vahvalla tyypityksellä) mukana tulee tslint-tiedosto, joka saa kehitysympäristön ehdottamaan muun muassa mahdollisia tyyppejä ja muuttujia (kuva 5). Tslint-

tiedostossa voidaan myös määritellä yleisiä hyviä ohjelmointikäytäntöjä ja koodin ulkoasuun liittyviä tyylittelyjä, jotta kehitysympäristö ilmoittaa virheestä, vaikka koodi olisikin muuten toimivaa.

```
1  {
2    "rules": {
3      "adjacent-overload-signatures": true,
4      "ban-comma-operator": true,
5      "no-namespace": true,
6      "no-parameter-reassignment": true,
7      "no-reference": true,
8      "no-unnecessary-type-assertion": true,
9      "label-position": true,
10     "no-conditional-assignment": true,
11     "no-construct": true,
12     "no-duplicate-super": true,
13     "no-duplicate-switch-case": true,
14     "no-duplicate-variable": [true, "check-parameters"],
15     "no-shadowed-variable": true,
16     "no-empty": [true, "allow-empty-catch"],
17     "no-floating-promises": true,
18     "no-implicit-dependencies": true,
19     "no-invalid-this": true,
20     "no-unsafe-finally": true,
21     "no-use-before-declare": true,
22     "no-void-expression": [true, "ignore-arrow-function-shorthand"],
23     "no-duplicate-imports": true,
24     "no-empty-interface": {"severity": "warning"},
25     "no-import-side-effect": {"severity": "warning"},
26     "no-var-keyword": {"severity": "warning"},
27     "triple-equals": {"severity": "warning"},
28     "deprecation": {"severity": "warning"},
29     "prefer-for-of": {"severity": "warning"},
30     "unified-signatures": {"severity": "warning"},
31     "no-unused-variable": {"severity": "warning"},
32     "prefer-const": {"severity": "warning"},
33     "trailing-comma": {"severity": "warning"}
34   },
35
36   "defaultSeverity": "error"
37 }
```

Kuva 5. tslint.json

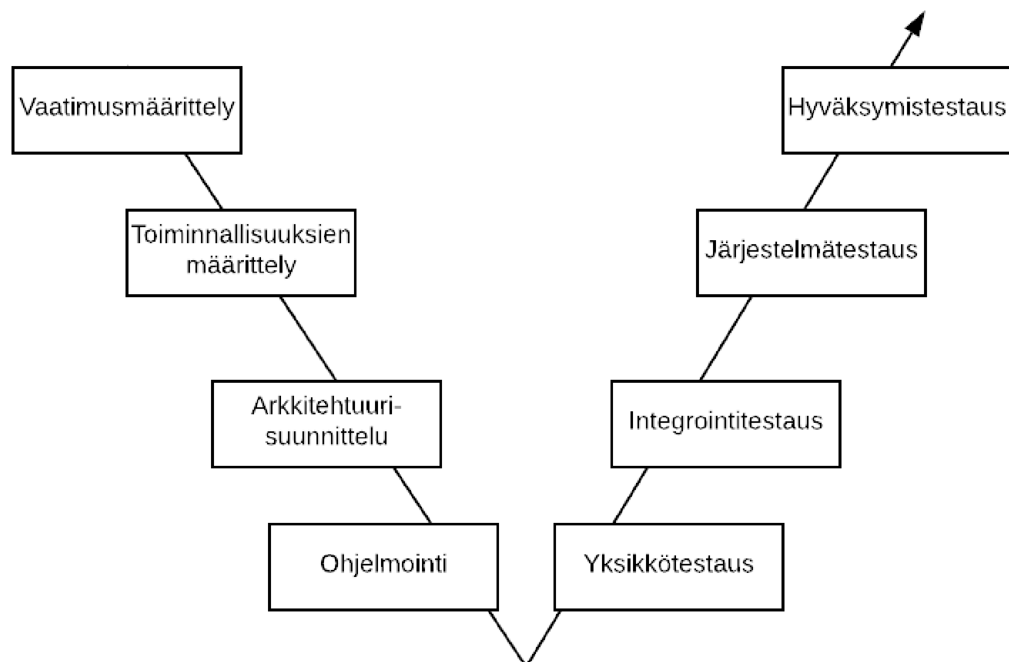
Jatkuvassa tarkistuksessa voidaan hyödyntää ulkoisia palveluita, kuten esimerkiksi SonarLintiä, jotka tarkistavat koodista virheitä, koodihajuja, testikattavuuden, teknisen velan sekä haavoittuvuuksia automaattisesti joka kerta kun uusi versio säilötään koodivarastoon.

3.5 Testaus

Testauksella varmistutaan siitä, että kirjoitettu koodi varmasti (tai todennäköisesti) toimii. Ideaalisessa tilanteessa ohjelmiston testikattavuus olisi ainakin 80 %, jotta lähes kaikki toiminnallinen koodi tulisi testattua ohjelman koostamisen yhteydessä. Testien määrää tärkeämpää on kuitenkin laatu, sillä niiden pitäisi oikeasti toimia ollakseen hyödyllisiä. (Fowler 2012).

Jatkuvan integraation ja julkaisun kannalta tärkeimpiä testimuotoja ovat yksikkötestit (unit tests), integraatiotestit (integration tests) sekä hyväksyntätestit (acceptance tests).

Testivetoisessa ohjelmoinnissa suunnitellaan ensin ominaisuudet, ja niille kirjoitetaan testit ennen varsinaisen ohjelmistokoodin kirjoittamista. Testin kirjoittamisen jälkeen varsinaista koodia tehdään juuri sen verran, että testi menee läpi. Testivetoisen ohjelmistokehityksen hyötynä on suunnitelmallisuus, sillä tuotantoon päätyy vähemmän helposti turhia tai monimutkaisia osia, koodin laatu on hyvää ja koodi on suoraan valmista tulevia muutoksia varten (Pietiläinen 2018, 7). Ohjelmistokehityksessä testauksen ja varsinaisen kehitystyön suhdetta voidaan havainnollistaa V-mallilla, jonka mukaan suunnittelu tapahtuu testaustasoa vastaavalla suunnittelutasolla. Erillisiä testaustasoja ovat yksikkötestaus, integrointitestaus sekä järjestelmätestaus (Haikala & Mikkonen 2011, 206-207). CI/CD- mallissa osaksi järjestelmätestausta lisätään myös hyväksyntätestaus (Kuva 6).



Kuva 6. Testauksen tasot

Yksikkötestit ovat nimensä mukaisesti yksikköjen testausta varten luotuja testejä. Yksiköillä tässä tapauksessa tarkoitetaan erilaisia menetelmiä, metodeja tai luokkia. Yksikkötestit kertovat meille toiminnallisuuden toimivuudesta siten, kuin testaaja haluaisi niiden toimivan (Pietiläinen 2018, 9). Pietiläisen (2018,

10) mukaan yksikkötestin perimmäinen tarkoitus on pitää huoli siitä, ettei tuotantoon eksy rikkinäisiä tai vajaita luokkia, ja jos testi epäonnistuu, sen tulisi kertoa testaajalle epäonnistumisen syy ja sijainti. Kuvassa 7 näkyy esimerkki Jestillä kirjoitetusta React Native- ohjelmointikielen yksikkötestistä.

```
1  import 'react-native';
2  import React from 'react';
3  import Overview from '../js/screens/Overview';
4  // Note: test renderer must be required after react-native.
5  import renderer from 'react-test-renderer';
6
7  jest.useFakeTimers();
8
9  const DATE_TO_USE = new Date('2016');
10 const _Date = Date;
11 global.Date = jest.fn(() => DATE_TO_USE);
12 global.Date.UTC = _Date.UTC;
13 global.Date.parse = _Date.parse;
14 global.Date.now = _Date.now;
15
16 test('String generation', () => {
17   const props = {
18     navigation: {
19       state: {
20         params: {
21           reportData:{
22             damages: ["Steering", "Front wheel", "Front light"]
23           }
24         }
25       }
26     },
27   };
28
29   let overview = new Overview(props);
30
31   it('should generate the correct string', () => {
32     expect(overview.generateDamagePreview()).toBe("Steering, Front wheel, Front light");
33   });
34 });
```

Kuva 7. Esimerkki yksikkötestistä Jestillä

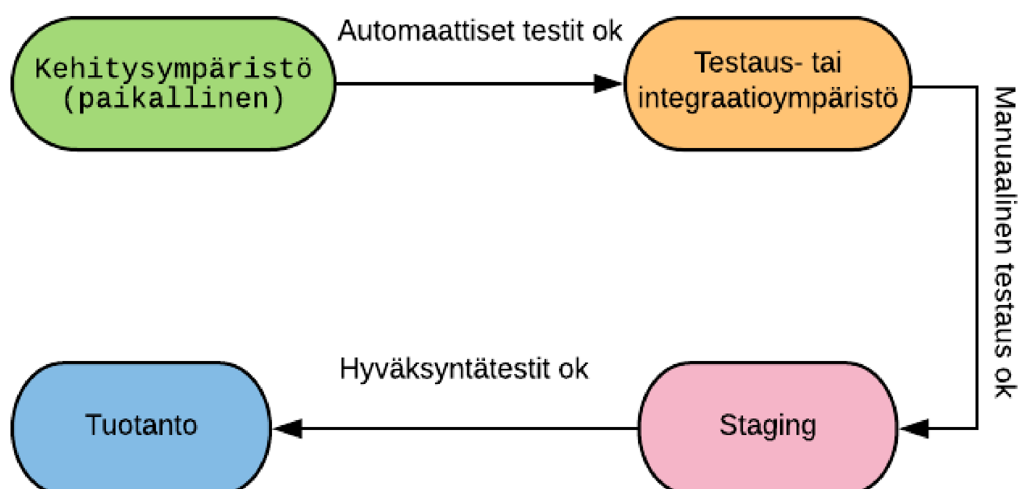
Integraatiotesteillä testataan koko järjestelmän toiminnallisuutta kokonaisuutena. Niillä varmistutaan esimerkiksi tietokantayhteyksien toiminnasta, ulkoisten palvelujen yhteyksien toiminnallisuuksista sekä tietojen kirjoittamisesta tiedostoihin. Ylipäätään integraatiotesteillä testataan käyttöliittymän yhteensopivuutta palvelinpuolen kanssa. Hyväksyntätesteihin on kaksi päälähtökohtaa: Big Bang-testit sekä Incremental-testit. Big bang-testit odottavat kaikkien systeemin komponenttien olevan valmiina ennen testausta ja testaa sitten koko järjestelmän. Incremental-testit testaavat komponentteja sitä mukaa kun ne ovat valmiina, ja niille komponenteille, jotka eivät ole vielä valmiita luodaan jonkinlaisia tilantäyttäjiä (placeholder) (Rossel 2017, 12-13).

Yksikkö- ja integraatiotestien läpäisyn jälkeen voidaan olettaa järjestelmän toimivan kokonaisuutena ja laadun olevan vähintään kohtalaisen hyvä, mutta se ei vielä välttämättä tarkoita ohjelmiston tekevän haluttuja asioita. Tässä tilanteessa hyväksyntätestit tulevat mukaan kuvioon. Niillä testataan tiettyjen toiminnallisuuksien toimimista halutulla tavalla. Esimerkiksi sovellukseen voidaan ottaa yhteyttä ulkoisesta palvelusta, kirjautua sisään, luoda ja päivittää käyttäjä, ja lopulta epäaktivoida sen. Hyväksyntätestien pääidea on siis tehdä automaattisesti suurin piirtein sama, mitä loppukäyttäjä tulee ohjelmistolla tekemään (Rossel 2017, 14).

Rosselin (2017, 15) mielestä tärkein idea testejä kirjoittaessa on käyttää järkeä: Jos tunnet tarvitsevasi testin toiminnallisuuteen, kirjoita se, ja jos testi tuntuu turhalta, jätä se kirjoittamatta. Testin tulisi tukea ohjelmointiprosessia eikä olla itseisarvo.

3.6 Toimitusympäristöt

Jatkuvan integraation periaatteiden mukaan sovelluksen tulisi pyöriä ainakin kolmessa tai neljässä eri ympäristössä samalla koodipohjalla: Kehitys (development), testaus, staging sekä tuotanto (production). Kehitysversiota hyödynnetään sovelluksen kehittämisessä, ja uuden ominaisuuden valmistuessa koodi ajetaan testausympäristöön, jossa voidaan kokeilla kaikkien ominaisuuksien toiminta halutulla tavalla. Kun testausversioon ollaan tyytyväisiä, voidaan uudet ominaisuudet päivittää staging-ympäristöön (Murray 2006) ja sen toiminnallisuuksiin tyydyttäessä se päivitetään tuotantoympäristöön (Kuva 8).



Kuva 8. Toimitusympäristöt

Kehitysympäristössä kehitetään uudet ominaisuudet, jotka valmistuvat kehittäjien paikallisesta ympäristöstä. Kehitysympäristössä on siis yleensä aina uusimmat ominaisuudet, ja siellä ajetaan aina uusinta versiota koodista (Ellison 2016). Kehitysympäristön tarkoituksena on mahdollistaa uusien ja mahdollisesti jopa radikaalien ideoiden testaaminen vaikuttamatta muiden kehittäjien työhön (Murray 2006). Kehitysympäristö voi olla myös vain kehittäjän paikallisella tietokoneella oleva ympäristö, tai pienen kehitystiimin yhteisessä käytössä. Kehitysympäristössä on tavallisesti vain mock- tai testausdataa, joka on samankaltaista oikean datan kanssa, mutta sitä yleensä lisätään vain sen verran kuin on tarpeellista kehitystyön kannalta. Kehitysympäristön data tavallisesti tulisi myös siivota säännöllisesti. (Murray 2006)

Kehitysympäristöstä voidaan eritellä vielä integraatio- tai testiympäristö, jossa yhdistetään ja testataan koko projektitiimin työ sekä voidaan hyödyntää erillistä testaustiimiä manuaalisessa testauksessa (Ellison 2016; Murray 2006).

Staging-ympäristön tulisi olla mahdollisimman samankaltainen tuotannon kanssa, mutta yleensä ilman oikeaa tuotantodataa. Sitä voidaan käyttää esimerkiksi uusien ominaisuuksien demoamiseen asiakkaalle, markkinoinnissa tai uusien henkilöiden perehdyttämisessä (Ellison 2016; Murray 2006). Pienemmässä projektissa staging-ympäristö voi myös hoitaa testi- tai integraatioympäristön virkaa, jolloin kolme erillistä ympäristöä voi olla riittävästi kehitystyön kannalta.

Tämän opinnäytetyön kannalta staging-ympäristön virkaa hoitaa mobiilisovelluksille sovelluskauppojen testiympäristöt, joihin on mahdollista määritellä sovellus julkaistavaksi joko sisäistä suljettua testausta, tai ulkoista ja avointa testausta varten. Tuotantoympäristöllä tarkoitetaan sovelluskauppoihin vapaasti ladattavaksi julkaistuja sovelluksia, jotka eivät ole enää kehitys- tai testausvaiheessa, vaan sisältävät vain toimivaksi todettuja ominaisuuksia.

3.7 Sovelluskaupat

Sovelluskaupoilla tarkoitetaan sovellusten julkaisualustoja, kuten Applen App Storea tai Googlen Play -kauppaa. Tässä opinnäytetyössä keskitytään mobiilisovelluskauppoihin. Kaupoissa on sekä ilmaisia, että maksullisia sovelluksia. Sovellusten julkaisemiseksi kauppoihin tulee niihin rekisteröityä kehittäjäksi. Molemmissa vaihtoehdoissa on yleiset ohjeistukset, joita julkaistavissa sovelluksissa tulee noudattaa.

App Storen ohjeistuksissa painotetaan erityisesti ulkoasua, tietoturvaa, suorituskykyä sekä laillisuutta, ja Apple on usein hyvinkin tarkka näistä asioista ja voi helposti hylätä sovelluksen, jos se rikkoo vähänkin näitä. Sovellusten tulee olla hyvin toimivia, ilmoittaa käyttäjälle selkeästi esimerkiksi kameran tai sijainnin käytön syistä sekä niiden sisällön tulee olla laillista ja lapsiystävällistä (App Store review guidelines 2018). Sovellusten ikäraja voi helposti nousta yli 18 vuoden, jos ne sisältävät esimerkiksi rumaa kielenkäyttöä tai kilpailuja.

Google painottaa myös paljon samoja asioita Play-kaupassa kuin Apple. Sovellusten tulee olla tietoturvallisia, laillisia sekä originaaleja. Kiellettyjä aiheita Play-kaupassa ovat muun muassa sovellukset, jotka lähettävät tai sisältävät paljon roskapostia, sisältävät sopimattomia tai laittomia asioita tai uhkapelejä sekä tekijänoikeuksia rikkovat sovellukset. Lisäksi sovelluksia, jotka kaatuvat yhtenään tai sisältävät tietoturvauhkia saatetaan poistaa tai kieltää kaupasta. Jos sovellusten rikkeet ovat vakavia tai toistuvia, saatetaan kehittäjältä poistaa lisenssi sovellusten julkaisua varten (Developer policy center s.a).

Kuva 9. Sovelluksen lataaminen App Storeen

Ennen varsinaisen sovelluksen lataamista tulee molemmissa kaupoissa täyttää tiedot kauppasivulle, kuten kuvakaappaukset, muutosloki, tietosuojakäytäntö, yrityksen yhteystiedot sekä hinnoittelu (kuva 9) (App Review s.a; Publishing an app s.a).

Kauppaan ladatut sovellukset tarkistetaan aina ennen niiden hyväksymistä kauppoihin. Play-kaupassa tämä prosessi on pääasiassa automaattinen, mutta joissain tapauksissa sovellukset tarkistetaan myös käsin erityisesti silloin, jos sovelluksessa on aihetta epäillä jonkin ohjeistuksen rikkomista. App Storessa kaikki sovellukset tarkistetaan käsin, mikä lisää yleensä myös tarkistamiseen kuluvaa aikaa: App Storessa sovelluksen tarkistukseen menee yleensä 1-3 vuorokautta. Noin 50% sovelluksista tarkistetaan kuitenkin 24 tunnin sisään, ja 90% sovelluksista 48 tunnin sisään (App Review, s.a). Googlen Play-kaupassa sovellus on yleensä valmiina ladattavaksi jo muutaman tunnin päästä sen julkaisun jälkeen (Publishing an app s.a).

4 CASE OBSERVIS

Tässä luvussa selitetään sovellusten julkaisuprosessi Jenkinsin ja Fastlanen avulla. Lisäksi kerrotaan prosessin konfiguroinnista ja käydään läpi sen toiminnallisuus. Viimeisenä kerrotaan myös prosessin integroimiseen liittyvistä ongelmista sekä jatkokehitysmahdollisuuksista.

4.1 Prosessi

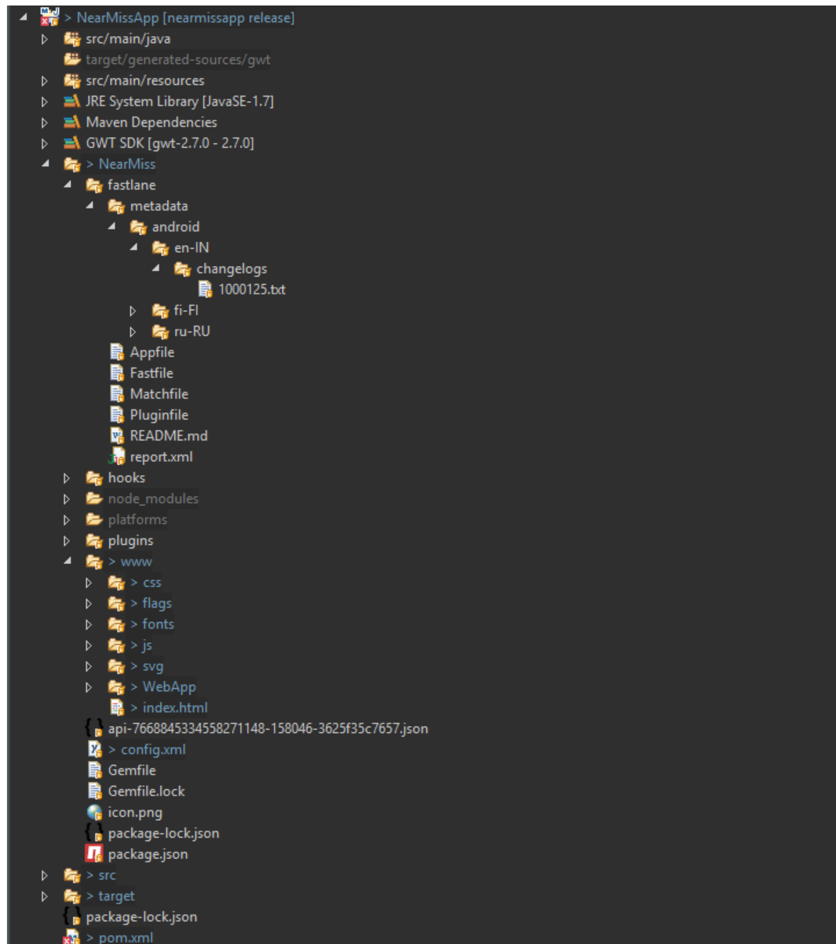
Esimerkkisovelluksena toimii NearMiss, joka on tapaturmien tai läheltä piti -tilanteiden havainnointiin ja ilmoittamiseen tarkoitettu sovellus. NearMiss on kirjoitettu Javalla ja GWT:llä, ja Android- ja iOS -käännöksiin käytetään Cordovaa. Tavoitteena oli saada jatkuva integraatio tiettyyn pisteeseen asti sekä jatkuva julkaisu kokonaisuudessaan konfiguroitua projektille. Projektin versionhallinta hyödynsi alun perin SVN:ää, ja osana CI/CD-mallia siirrettiin projekti hyödyntämään GitLabia, sillä puurakenteinen versionhallintatyökalu sopii käytettyyn malliin paremmin.

Jatkuvan julkaisun prosessi on konfiguroitu Jenkinsin ja Fastlanen avulla. Aina kun versionhallinnan kehityshaara yhdistetään julkaisuhaaraan, Jenkins koostaa Maven-sovelluksen. Tämän jälkeen Cordova lisää alustat Androidille sekä iOS:lle ja luo niille sopivat ikonit. Cordovan jälkeen käynnistyy Fastlane, joka hoitaa sovelluksen suoraan sekä Googlen Play-kauppaan uutena betaversiona sekä Applen TestFlightiin sisäistä testausta varten. Kun uuteen testiversioon ollaan tyytyväisiä, voidaan julkaisuhaara yhdistää master-haaraan, joka käynnistää vastaavan prosessin, mutta julkaisee testiversion sijasta tuotantoversion kauppoihin.

Maven on Apachen ylläpitämä koosteautomaatiotyökalu. Sen avulla voidaan helposti koostaa sovellukset, siivota vanhat koosteet sekä helpottaa siirtymistä uusiin ominaisuuksiin. Maven hyödyntää POM (project object model, projektiobjektimalli) -tiedostoa, johon määritellään sen käyttämät asetukset.

Cordova on myös Apachen ylläpitämä työkalu, joka kääntää ohjelman koodit mobiilialustojen natiivikielelle, joka iOS:llä on Swift, ja Androidilla Java. Cordovalla voidaan kääntää myös esimerkiksi Windows Phone- ja BlackBerry-alus-

toille, mutta näiden käyttö on nykyään jäänyt niin pieneksi, ettei NearMiss-sovelluksesta ole tehty näille versiota. Cordova hyödyntää natiivikoodien käytössä lisäpalikoita, joita löytyy yhteensä satoja erilaisia tarkoituksia varten.



Kuva 10. NearMiss projektin kansiorakenne

NearMiss oli jo ennestään Maven-projekti, mutta automaatio oli jäänyt aika puolitiehen, ja kootut tiedostot täytyi kopioida käsin Cordova-kansion sisälle. POM- muutoksilla saatiin Mavenin kopioimaan tiedostot suoraan oikeaan kansioon Cordova-kansion sisälle. Kuva 10 havainnoi projektin kansiorakennetta. NearMiss-kansion alla näkyy Cordovan sekä Fastlanen automaattisesti luomat tiedostot.

Fastlanen asennus macOS:lle osoittautui hieman monimutkaiseksi, sillä kokemukseni macOS:stä oli aika pientä, etenkin sen erilaisista bundlereista ja pakettihallintaohjelmistoista. Fastlanen asennuksen jälkeen tuli konfiguroida Fastfile, jossa määritellään erilaiset kaistat eri julkaisuja, kuten Android tai iOS beta- tai tuotantojulkaisuja varten (kuva 11). Fastfile on oleellisin osa Fastla-

nea, sillä siinä määritellään kaikki tarvittavat komennot julkaisuja varten. Fastfilen lisäksi Fastlane sisältää Pluginfilen sekä Appfilen, joissa voidaan määritellä mahdolliset lisäpalikat tai Fastfilen käyttämät muuttujat, esimerkiksi tiimin tai sovelluksen nimet.

```
14  update_fastlane
15  before_all do
16  unlock_keychain(path: "/Users/Shared/Jenkins/Home/workspace/NearMissApp_Staging/observis.keychain",
17  password:ENV['FASTLANE_PASSWORD'])
18  end
19
20  platform :ios do
21    desc "deploy iOS releases"
22    lane :beta do
23    match(type: "appstore")
24    cordova(platform:'ios', build_flag: ['-UseModernBuildSystem=0'])
25    gym(scheme: "NearMiss",
26      workspace: "platforms/ios/NearMiss.xcworkspace")
27    changelog_from_git_commits(commits_count: "1", merge_commit_filtering: "include_merges")
28    testflight(username: 'support@observis.fi')
29    end
30    lane :release do
31    match(type: "appstore")
32    cordova(platform:'ios', build_flag: ['-UseModernBuildSystem=0'])
33    gym(scheme: "NearMiss",
34      workspace: "platforms/ios/NearMiss.xcworkspace")
35    set_changelog(changelog: "Bugfixes")
36    deliver(username: 'support@observis.fi',
37      submit_for_review: "true",
38      force: "true",
39      skip_screenshots: "true",
40      skip_metadata: "true",
41      precheck_include_in_app_purchases: "false",
42      automatic_release: "true")
43    end
44  end
45
46  platform :android do
47    desc "Deploy android releases"
48    lane :beta do
49    cordova(
50        platform: 'android',
51        keystore_alias: 'nearmiss',
52        keystore_password: '         ',
53        key_password: '          ',
54        release: "false"
55      )
56      changelog_from_git_commits(commits_count: "1", merge_commit_filtering: "include_merges")
57      supply(track: 'internal', apk: ENV['CORDOVA_ANDROID_DEBUG_BUILD_PATH'], package_name: 'fi.observis.mobile.nearmiss'
58    end
59      lane :release do
60    cordova(
61        platform: 'android',
62        keystore_alias: 'nearmiss',
63        keystore_password: '         ',
64        key_password: '          '
65      )
66      supply(track: 'production', apk: ENV['CORDOVA_ANDROID_RELEASE_BUILD_PATH'], package_name: 'fi.observis.mobile.nearmiss'
67    end
68  end
```

Kuva 11. Fastfile- esimerkki

Sovellusten allekirjoittamiseen Fastlane tarjoaa useita vaihtoehtoja, joita ovat muun muassa Match jossa allekirjoitusavaimet tallennetaan Git- koodivarastoon, Cert ja Sigh jotka luovat uudet sertifikaatit sekä provisioprofiilit (provisioning profile) ja lisäävät ne Applen avainten ja sertifikaattien säilytysohjelmaan Keychainiin. Lisäksi vaihtoehtona on manuaalinen allekirjoitus.

CI-palvelimelle järkevin vaihtoehto iOS-julkaisujen kanssa oli hyödyntää Matchia, sillä avaimet pysyvät varmasti tallessa yhdessä paikassa, ja niiden löytäminen on helpompaa kuin CI-palvelimen tallennusmuistista. Android-avaimen käyttö on helpompaa, sillä sen pystyy kirjoittamaan suoraan Fastfileen, tai

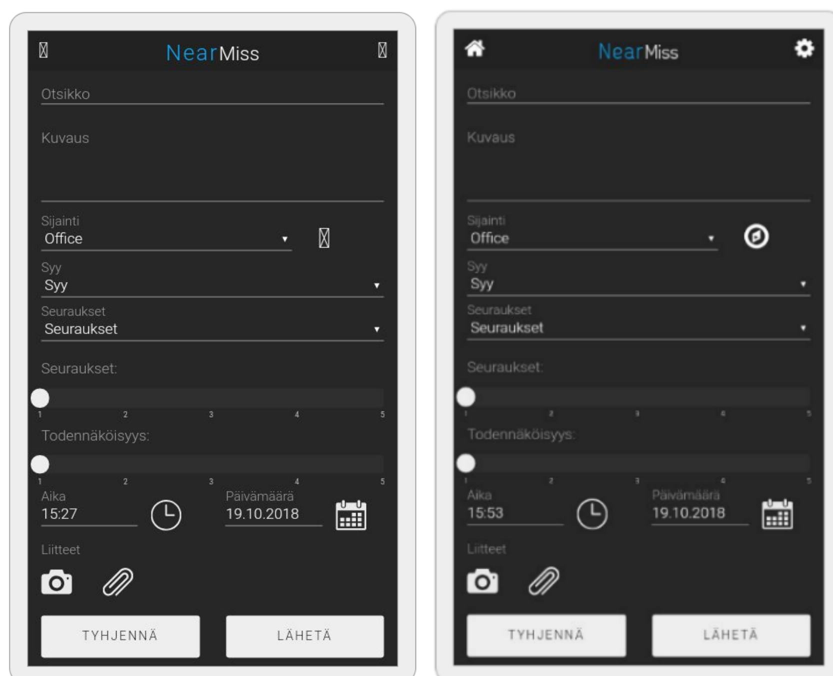
Jenkinsiin ympäristömuuttujina. Match-konfiguroinnit, kuten Git-osoitteet ja tunnukset kirjattiin erilliseen Matchfileen, jonka pohjan Match loi automaattisesti.

Fastlane pystyy käytännössä hoitamaan aivan uudenkin sovelluksen julkaisun kauppasivun muokkaamista myöten alusta alkaen tarvittaessa, mutta etenkin vähemmän kokeneelle tiimille kauppasivun ja jopa ensimmäisen julkaisun tekeminen käsin voi olla jopa vähemmän aikaa vievää ja helpompaa. Sovelluspäivitykset Fastlane hoitaa kuitenkin helposti, ja jos kauppasivuille ei juuri tule muutoksia sovellusversioiden mukana, ei välttämättä kannata nähdä vaivaa ylimääräiseen Fastlane-tiedostojen kirjoitteluun.

Prosessin lopputuotoksena sain Git-haarat yhdistämällä tai muutoksen lähettämällä release-haaran varastoon täysin automaattisen julkaisun suoritettua Google Play -kaupassa sekä beta- että varsinaiseen julkaisuversioon, ja App Storessa julkaisuversioon.

4.2 Esimerkkitoteutus

Skenaario: NearMiss-sovellukseen tehty bugikorjaus, ja automaattinen korjauksen julkaisu kauppoihin. Sovellukseen oli jossain vaiheessa päätynyt bugi fonttien lataamisen kanssa, joka aiheutti väärien fonttivalintojen lisäksi myös FontAwesome-ikonien rikkoutumisen (kuva 12).



Kuva 12. Rikkoutuneet ikonit ja väärät fontit sovelluksessa (vasen), korjattu sovellus (oikea)

Vika johtui väärin konfiguroidusta pom.xml-tiedostosta. Kopioitaviin tiedostoihin oli vahingossa lisätty filtteröinti päälle, joka korruptoi kuva- ja fonttitiedostot. Bugin sai korjattua lisäämällä poikkeukset filtteröintiin (nonFilteredFileExtensions) kuva- ja fonttitiedostopäätteille (kuva 13).

```
<plugin>
  <artifactId>maven-resources-plugin</artifactId>
  <version>2.6</version>
  <executions>
    <execution>
      <id>copy-resources-1</id>
      <phase>validate</phase>
      <goals>
        <goal>copy-resources</goal>
      </goals>
      <configuration>
        <outputDirectory>${basedir}/NearMiss/www/</outputDirectory>
            <nonFilteredFileExtensions>
                <nonFilteredFileExtension>svg</nonFilteredFileExtension>
                <nonFilteredFileExtension>woff</nonFilteredFileExtension>
                <nonFilteredFileExtension>woff2</nonFilteredFileExtension>
                <nonFilteredFileExtension>ttf</nonFilteredFileExtension>
                <nonFilteredFileExtension>eot</nonFilteredFileExtension>
                <nonFilteredFileExtension>otf</nonFilteredFileExtension>
            </nonFilteredFileExtensions>
        <resources>
          <resource>
            <directory>${basedir}/src/main/webapp/</directory>
            <includes>
                <include>css/**</include>
                <include>flags/**</include>
                <include>fonts/**</include>
                <include>js/**</include>
                <include>svg/**</include>
                <include>index.html</include>
                <include>WebApp/*</include>
            </includes>
            <excludes>
                <exclude>appicon/**</exclude>
                <exclude>WEB-INF</exclude>
                <exclude>META-INF</exclude>
            </excludes>
            <filtering>true</filtering>
          </resource>

        </resources>
      </configuration>
    </execution>
```

Kuva 13. Pom.xml filteröintipoikkeukset

Bugin korjaamisen jälkeen muutokset voidaan siirtää koodivarastoon ”git commit”-komennolla, tai käyttämällä kehitysympäristön graafista käyttöliittymää. Seuraavaksi yhdistetään develop- eli kehityshaara release- eli julkaisuhaaraan joko GitLabin kautta, komentoriviltä tai kehitysympäristön kautta. Kuvassa 14 tehdään yhdistämispyyntö (Merge request) GitLabin kautta.

Title Develop
Start the title with WIP: to prevent a **Work In Progress** merge request from being merged before it's ready.
Add description templates to help your contributors communicate effectively!
Description Write Preview
Fixed fonts and icons
Markdown and quick actions are supported Attach a file
Assignee Assignee Assign to me
Milestone Milestone
Labels Labels
Source branch develop
Target branch release Change branches
Remove source branch when merge request is accepted.
Squash commits when merge request is accepted. About this feature
Submit merge request Cancel

Kuva 14. GitLab yhdistämispyynnön luominen

Kun yhdistämispyyntö hyväksytään, käynnistyy Jenkins-tehtävä automaattisesti (kuva 15), joka hoitaa loput automaattisesti, kunhan kaikki on oikein määritelty.

Jenkins
Jenkins › NearMissApp_Staging ›
Back to Dashboard
Status
Changes
Workspace
Build Now
Delete Maven project
Configure
Modules
Dependency Graph
Purge Build History
Build History trend
find
#176 Oct 19, 2018 4:51 PM
Triggered by GitLab Merge Request #9: eliast/develop => release
Maven project NearMissApp_Staging
Staging environment for NearMiss application. Successful build will create a new version for Google Play beta and iOS TestFlight
Workspace
Recent Changes
Permalinks
- Last build (#175), 9 min 48 sec ago
- Last failed build (#175), 9 min 48 sec ago
- Last unsuccessful build (#175), 9 min 48 sec ago
- Last completed build (#175), 9 min 48 sec ago

Kuva 15. Jenkins -tehtävä käynnistyy automaattisesti

Jenkins koostaa Maven-sovelluksen, lisää alustat cordovaa varten sekä luo sovelluksen ikonin molemmille alustoille oikean kokoisina, ja lopuksi ajaa Fastlane-komennot, jotka hoitavat sovelluksen cordova-käännökset loppuun, allekirjoittavat sovelluksen sekä lähettävät sen suoraan Play-kauppaan (kuva 16).

```
[17:45:08]: ▸ :app:validateSigningDebug
[17:45:10]: ▸ :app:packageDebug
[17:45:10]: ▸ :app:assembleDebug
[17:45:10]: ▸ :app:cdvBuildDebug
[17:45:10]: ▸ BUILD SUCCESSFUL in 51s
[17:45:10]: ▸ 46 actionable tasks: 46 executed
[17:45:11]: ▸ Built the following apk(s):
[17:45:11]: ▸ /Users/Shared/Jenkins/Home/workspace/NearMissApp_Staging/NearMiss/platforms/android/app/build/outputs/apk/debug/app-debug.apk
[17:45:11]: --------------------
[17:45:11]: --- Step: supply ---
[17:45:11]: --------------------
+------------------------+--------------------------------------------------+
|                      Summary for supply 2.105.2                           |
+------------------------+--------------------------------------------------+
| track                  | alpha                                            |
| skip_upload_metadata   | true                                             |
| package_name           | fi.observis.mobile.nearmiss                      |
| json_key               | api-7668845334558271148-158046-3625f35c7657.json |
| metadata_path          | ./fastlane/metadata/android                      |
| skip_upload_apk        | false                                            |
| skip_upload_aab        | false                                            |
| skip_upload_images     | false                                            |
| skip_upload_screenshots | false                                           |
| validate_only          | false                                            |
| check_superseded_tracks | false                                           |
| timeout                | 300                                              |
| deactivate_on_promote  | true                                             |
+------------------------+--------------------------------------------------+

[17:45:15]: Preparing to upload for language 'en-US'...
[17:45:15]: Preparing to upload for language 'fi-FI'...
[17:45:15]: Preparing to upload for language 'ru-RU'...
[17:45:15]: Uploading all changes to Google Play...
[17:45:18]: Successfully finished the upload to Google Play

+------+----------------+------------+
|          fastlane summary          |
+------+----------------+------------+
| Step | Action         | Time (in s) |
+------+----------------+------------+
| 1    | update_fastlane | 0          |
| 2    | unlock_keychain | 0          |
| 3    | cordova        | 65         |
| 4    | supply         | 6          |
+------+----------------+------------+
```

Kuva 16. Fastlanen automaattinen sovelluksen koostaminen sekä julkaisu Play-kauppaan testiversiona

Jos Play-kauppaan lähettäminen onnistuu, jatkaa Fastlane vielä sovelluksen koostamisen, muutoslokien sekä latauksen ja lähetyksen tarkistusta varten myös App Storeen (kuva 17).

```
[17:46:46]: ▸   adding: NearMiss.app.dSYM/ (stored 0%)
[17:46:46]: ▸   adding: NearMiss.app.dSYM/Contents/ (stored 0%)
[17:46:46]: ▸   adding: NearMiss.app.dSYM/Contents/Resources/ (stored 0%)
[17:46:46]: ▸   adding: NearMiss.app.dSYM/Contents/Resources/DWARF/ (stored 0%)
[17:46:46]: ▸   adding: NearMiss.app.dSYM/Contents/Resources/DWARF/NearMiss (deflated 63%)
[17:46:46]: ▸   adding: NearMiss.app.dSYM/Contents/Info.plist (deflated 52%)

[17:46:46]: Successfully exported and compressed dSYM file
[17:46:46]: Successfully exported and signed the ipa file:
[17:46:46]: /Users/Shared/Jenkins/Home/workspace/NearMissApp_Staging/NearMiss/NearMiss.ipa
[17:46:46]: ----------------------------------------
[17:46:46]: --- Step: changelog_from_git_commits ---
[17:46:46]: ----------------------------------------
[17:46:46]: Collecting the last 1 Git commits

added changelogs

[17:46:46]: ------------------------
[17:46:46]: --- Step: testflight ---
[17:46:46]: ------------------------
[17:46:46]: Login to App Store Connect (support@observis.fi)
[17:46:49]: Login successful
[17:46:50]: Ready to upload new build to TestFlight (App: 1162136864)...
[17:46:50]: Fetching password for transporter from environment variable named `FASTLANE_APPLE_APPLICATION_SPECIFIC_PASSWORD`
[17:46:50]: Going to upload updated app to App Store Connect
[17:46:50]: This might take a few minutes. Please don't interrupt the script.
[17:48:33]: iTunes Transporter successfully finished its job
[17:48:33]: -------------------------------------------------------------------
[17:48:33]: Successfully uploaded package to App Store Connect. It might take a few minutes until it's visible online.
[17:48:33]: -------------------------------------------------------------------
[17:48:33]: Successfully uploaded the new binary to App Store Connect
[17:48:33]: If you want to skip waiting for the processing to be finished, use the `skip_waiting_for_build_processing` option
[17:48:41]: Waiting for App Store Connect to finish processing the new build (1.2.7 - 1.2.7)
[17:54:18]: Successfully finished processing the build 1.2.7 - 1.2.7
[17:54:22]: Successfully set the changelog for build
[17:54:22]: Distributing new build to testers: 1.2.7 - 1.2.7
[17:54:22]: Successfully distributed build to Internal testers 🚀
```

Kuva 17. Fastlane-julkaisu App Storen TestFlightiin

Jenkins ilmoittaa vielä kehitystiimille Discordissa tehtävän onnistumisesta tai epäonnistumisesta. Jos kaikki meni kuten pitää, uusi versio sovelluksesta löytyy kaupoista muutaman tunnin tai päivän sisällä, kunhan Google ja Apple saavat uuden version tarkistettua.

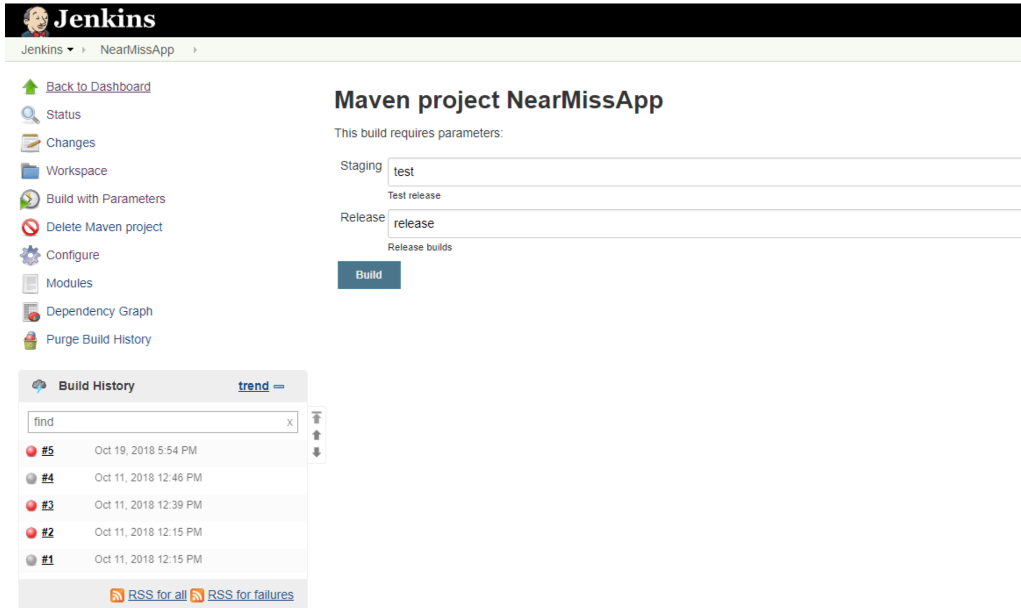
4.3 Ongelmat

Fastlanen kanssa yhdeksi isoksi ongelmaksi tuli Cordova, joka luo automaattisesti alustat Androidia sekä iOS:ää varten. Näistä alustoista usein puuttuu oletuksena viimeisimmät versiot esimerkiksi Androidin Gradlesta sekä Applen xcode-projekteista, ja joitain tiedostoja joutuu mahdollisesti joka tapauksessa avaamaan Android Studioon tai Xcodeen ja lisäämään puuttuvia tietoja. Apple erityisesti on tullut erityisen tarkaksi tietoturva- asioissa, eikä hyväksy sovelluksia kauppaansa, ellei lähes kaikkien puhelimen ominaisuuksien käyttöoikeuksia selitetä auki. Esimerkiksi sijaintia tai kameraa käytettäessä tulee tarkentaa, että mihin niitä käytetään sovelluksessa.

Suurimmat ongelmat tulivat vastaan iOS-sovellusten allerkirjoittamisessa Jenkins-palvelimen kautta, sillä Applen avainhallintaohjelmistoihin pääsy oli melko haastavaa. Fastlane tarjoaa tähän useampaa eri lähestymistapaa, kuten uusien allekirjoitusavainten luominen lennossa, avainten säilytys Git-varastossa sekä manuaalinen allekirjoitus, mutta kaikkien lähestymistapojen kanssa Jenkinsissä tuli mutkia matkaan. Poistamalla olemassa olevat sertifikaatit sekä proviisioprofiilit ja luomalla uudet onnistui allekirjoitus lopulta julkaisuversiolle. Fastlanen Match-ominaisuus mahdollistaa sekä niiden poistamisen, että luomisen melko yksinkertaisesti, mutta tässäkin syntyi ongelmia. Isoimpana ongelmana oli kirjoitusoikeuksien saaminen Git-varastoon suoraan CI-palvelimelta, ja jouduin alustamaan Matchin suoraan tätä palvelinta pyörittävältä macbookilta.

Xcode aiheutti myös muita ongelmia, sillä tätä opinnäytetyötä tehdessä siihen tuli iso päivitys, joka vaikutti Cordovan toimintaan, johon ei ehtinyt tulla yhteensopivaa päivitystä. Cordovaan ja Fastlaneen ehti kuitenkin tulla tuki taaksepäin yhteensopivuudelle, jonka avulla sovellus saatiin koostettua onnistuneesti.

Yksi iso ongelma koko projektin kannalta on lähes täydellinen testien puute. Sovelluksessa ei ole ainoatakaan yksikkö-, integraatio- tai hyväksyntätestiä, joten kauppajulkaisujen toimivuudesta ei ole mitään automaattista varmistusta, ja sen kanssa ollaan manuaalisen testiversion testauksen varassa. Observis Oy:llä erityisesti vanhempien projektien kanssa testikattavuus on jäänyt vähiin.



Kuva 18. Esimerkki parametrisoidusta Jenkins -tehtävästä

Prosessiin jäi myös asioita jatkokehitettäväksi. Versionumeroiden automaattiseen kasvattamiseen config.xml:stä antaa sekä Maven, Cordova, että Fastlane omat mahdolliset vaihtoehtonsa, mutta mikään niistä ei vielä toiminut kunnolla. Staging- sekä tuotantoprojektit Jenkinsissä voisi yhdistää samaan projektiin, ja halutun projektin ajaminen määritellä parametrien avulla (Kuva 18).

Tulevaisuudessa Cordova ja Jenkins tullaan eliminoimaan Observis Oy:n sovelluskehityksestä ja julkaisusta, ja niiden sijaan otetaan käyttöön uudemmat tekniikat, kuten React Native ja CircleCI. Tämä tulee helpottamaan CI/CD-mallin lisäämistä uusille projekteille.

5 PÄÄTÄNTÖ

Opinnäytetyön tavoitteena oli toteuttaa jatkuvan integroinnin, toimituksen ja julkaisun esimerkkimalli mobiilisovelluskehitykselle Observis Oy:llä. Esimerkki-

malli onnistui ihan hyvin, ja matkan varrella opin paljon uutta jatkuvan integraation ja julkaisun periaatteista, mobiilisovellusten julkaisemisesta sekä erityisesti Fastlanesta. Opinnäytetyön tekemisen aikana tutustuin myös syvemmin Jenkinsiin, jota olin jo vähän käyttänyt ennen opinnäytetyön aloittamista.

Työn lopputuotoksena syntyi toimiva jatkuvan julkaisun prosessi mobiilisovellusten betajulkaisua varten. Sain myös tuotantojulkaisut toimimaan, mutta ihan koko prosessia versiomuutoksesta tuotantoon en saanut vielä toimintaan, sillä puutteeksi jäi sovellusversion automaattinen kasvattaminen betajulkaisun jälkeen. Lisäksi hyödyntämäni sovellus ei ole tällä hetkellä eikä välttämättä tulevaisuudessakaan varsinaisessa tuotantokäytössä. Prosessista saadaan kuitenkin hyvä esimerkki tulevien projektien jatkuvan integraation ja julkaisun kannalta. Observis Oy:llä otetaan tulevissa projekteissa käyttöön tämä jatkuvan integraation ja automaattisen julkaisun malli.

Suurimmat ongelmat toteutuksen kanssa tulivat vastaan iOS -sovellusten allekirjoituksessa sekä Applen ja Androidin päivityksistä, jotka erityisesti yhdessä Cordovan kanssa aiheuttivat aina uusia muutoksia projektin koostamiseen, ja näitä uusia vikoja oli ajoittain vaikea selvittää.

Monet sovellusjulkaisut Observis Oy:llä ovat perinteisillä tekniikoilla todella hitaita prosesseja, joihin saa kulumaan koko päivän. Tämän CI/CD-mallin tavoitteena on tulevaisuudessa lyhentää suurimmilla projekteilla tähän julkaisuun menevää vaivaa minuutteihin, ja automatisoida se mahdollisimman pitkälle. Olen tyytyväinen suoritukseeni, ja siitä tulee tulevaisuudessa olemaan hyötyä sekä itselleni, että Observis Oy:lle.

Observis Oy:llä ollaan siirtymässä modernimpiin tekniikoihin sekä jatkuvassa integraatiossa, että sovelluskehityksessä. Tulevat sovellukset rakennetaan React Nativella Javan ja Cordovan sijasta, ja CI-palvelu vaihtuu todennäköisesti CircleCI:hin. Tämä tulee vähentämään ongelmia sekä kehityksessä, että julkaisussa.

LÄHTEET

Allen, S. 2016. Which is the most widely used continuous integration tool, Jenkins, Teamcity? Quora. Verkkopalvelu. Saatavissa: https://www.quora.com/Which-is-the-most-widely-used-continuous-integration-tool-Jenkins-Teamcity [viitattu 6.9.2018]

App Store Review Guidelines 2018. Apple. WWW-dokumentti. Saatavissa: https://developer.apple.com/app-store/review/guidelines/ [viitattu 25.10.2018]

Arcanist User Guide: Lint s.a. Phabricator. Saatavissa: https://secure.phabricator.com/book/phabricator/article/arcanist_lint/ [viitattu 22.10.2018]

CircleCI s.a. Verkkopalvelu. Saatavissa: https://circleci.com/ [viitattu 7.11.2018]

Ellison, R. 2016. Software Testing Environments Best Practices. WWW-dokumentti. Saatavissa: http://www.softwaretestingmagazine.com/knowledge/software-testing-environments-best-practices/ [viitattu 4.9.2018]

Fastlane s.a. Verkkopalvelu. Saatavissa: http://fastlane.tools/ [viitattu 4.9.2018]

Fowler, M. 2006. Continuous Integration. WWW-dokumentti. Saatavissa: https://martinfowler.com/articles/continuousIntegration.html [viitattu 6.9.2018]

Fowler, M. 2012. Test Coverage. WWW-dokumentti. Saatavissa: https://martinfowler.com/bliki/TestCoverage.html [viitattu 11.9.2018]

Gitflow workflow s.a. Atlassian. WWW-dokumentti. Saatavissa: https://www.atlassian.com/git/tutorials/comparing-workflows/gitflow-workflow [viitattu 10.9.2018]

Developer policy center s.a. Google. WWW-dokumentti. Saatavissa: https://play.google.com/about/developer-content-policy/ [viitattu 25.10.2018]

Guckenheimer, S. 2017. What is continuous integration? s.a Microsoft. WWW-dokumentti. Saatavissa: https://docs.microsoft.com/en-us/azure/devops/learn/what-is-continuous-integration [viitattu 20.8.2018]

Haikala I, Mikkonen T. 2011. Ohjelmistotuotannon käytännöt. 11. Painos. Talentum Media Oy. [viitattu 25.9.2018]

Jagtap S. 2018. Medium. Five options for continuous delivery without Fastlane. WWW-dokumentti. Saatavissa: https://medium.com/xcblog/five-options-for-ios-continuous-delivery-without-fastlane-2a32e05ddf3d [viitattu 25.9.2018]

Java Coding Conventions, 1997. Sun Microsystems Inc. PDF-dokumentti. Saatavissa: https://www.oracle.com/technetwork/java/codeconventions-150003.pdf [viitattu 22.10.2018]

Jenkins, s.a. Verkkopalvelu. Saatavissa: http://jenkins.io [viitattu 4.9.2018]

Laukkanen E, Itkonen J, Lassenius C. 2015. Problems, causes and solutions when adopting continuous delivery—A systematic literature review. 1. Painos. Aalto-yliopisto. [viitattu 5.9.2018]

Prince, S. 2016. The Product Managers' Guide to Continuous Delivery and DevOps. Mindtheproduct. Blogi. Saatavissa: https://www.mindtheproduct.com/2016/02/what-the-hell-are-ci-cd-and-devops-a-cheatsheet-for-the-rest-of-us/ [viitattu 6.9.2018]

Murray, R. 2006. Traditional Development/Integration/Staging/Production Practice for Software Development. WWW-dokumentti. Saatavissa: https://dltj.org/article/software-development-practice/ [viitattu 13.9.2018]

Myyry, J. 2002. V-mallin mukainen testausmenetelmä. PDF-dokumentti. Confuse. Saatavissa: http://www.soberit.hut.fi/T-76.115/01-02/palautukset/groups/Confuse/t3/docs/vmalli/vmalli.pdf [viitattu 25.9.2018]

Pietiläinen, N. 2018. Yksikkötestausohjeisto. Kaakkois-Suomen Ammattikorkeakoulu. Opinnäytetyö. [viitattu 10.9.2018]

Poimala S, Tolvanen P. 2013. Ketteryys haltuun – yleisimmät ketterät käytännöt. WWW-dokumentti. Saatavissa: https://www.meteoriitti.com/2013/06/06/ketteryys-haltuun-yleisimmat-ketterat-kaytannot/ [viitattu 6.9.2018]

Publishing an app s.a. Google. WWW-dokumentti. Saatavissa: https://support.google.com/googleplay/android-developer/answer/6334282 [viitattu 25.10.2018]

Rossel, S. 2017. Continuous Integration, Delivery and Deployment. 1. painos. Birmingham: Packt Publishing Ltd. [viitattu 20.8.2018]

Timofeev, M. 2018. Continuous Integration – The enterprise strategy. Kaakkois-Suomen Ammattikorkeakoulu. Opinnäytetyö. Saatavissa: http://www.theseus.fi/bitstream/handle/10024/149668/Timofeev%20Mikhail%20-%20Continuous%20Integration%20-%20Thesis.pdf?sequence=1&isAllowed=y [viitattu 20.8.2018]

What is Version Control? s.a. Atlassian. WWW-dokumentti. Saatavissa: https://www.atlassian.com/git/tutorials/what-is-version-control [viitattu 27.8.2018]