TAMPEREEN
AMMATTIKORKEAKOULU

KOONNIN KONFIGUROINTI JA JULKAISUN AUTOMATISOINTI ANDROID-ALUSTALLE

Taru Peltola

Opinnäytetyö
Huhtikuu 2018
Tietojenkäsittely
Ohjelmistotuotanto

TIIVISTELMÄ

Tampereen ammattikorkeakoulu
Tietojenkäsittely
Ohjelmistotuotanto

PELTOLA, TARU:
Koonnin konfigurointi ja julkaisun automatisointi Android-alustalle

Opinnäytetyö 43 sivua, joista liitteitä 5 sivua
Huhtikuu 2018

Opinnäytetyössä kehitettiin ja konfiguroitiin varianttipohjainen Android-testisovellus. Sovelluksella testattiin sovelluspäivitysten julkaisun automatisointia fastlane-työkalun avulla. Projektissa huomioitiin työn toimeksiantajan, Futurice Oy:n tarpeet asiakasprojektin julkaisutyön helpottamiseksi.

Opinnäytetyön tavoitteena oli tutustua tarkemmin Android-sovelluksen konfiguraatiotiedostoihin ja dokumentoida vaiheet fastlanen käyttöönottoon. Fastlanella haluttiin automatisoida sovelluspäivitysten koonti, julkaisu Google Play -sovelluskauppaan sekä lisääminen HockeyApp-sovelluksenhallintatyökaluun. Testisovelluksen ja hyvän dokumentaation avulla saatettiin tarpeellinen tieto ja osaaminen siirtää myöhemmin asiakasprojektiin. Opinnäytetyön tarkoitus on edustaa tilannetta, jossa julkaisuprosessin automatisointi on kannattavaa sekä käytännössä testata, kuinka paljon manuaalinen julkaisutyö helpottuu fastlanen avulla.

Projektin aluksi luotiin SpiceCabinet-niminen sovellus jäljittelemään asiakasprojektia. Koontitiedostot konfiguroitiin mahdollistamaan sovellusten variointi. Valmiit sovellusversiot lisättiin Google Play -kauppaan ja HockeyAppiin testaamista varten. Haasteelliseksi automaation tekivät sovellusvariantit, jotka piti pystyä erottamaan ja huomioimaan kaistan ajon yhteydessä. Varianttikohtaisilla ympäristömuuttujilla saatiin aikaan dynaaminen, laajentumisen mahdollistava ratkaisu.

Opinnäytetyön tuloksena oli vaatimusten mukainen automaatioprosessi sovelluspäivitysten julkaisuun, jossa huomioidaan sovellusvarianttien käsittely. Fastlane-automaatiotyökalun käyttöönotosta ja varioinnin käsittelystä syntyi monipuoliset ohjeet, joiden avulla fastlanen käyttöönotto asiakasprojektiin toteutettiin. Lisäksi luotiin Android-testisovellus, jota voidaan jatkossa käyttää uusien toimintojen testaamiseen, kun asiakasprojektin automaatiota jatkokehitetään.

Asiasanat: android, julkaisu, automaatio, koonti, konfiguraatio, fastlane, HockeyApp

ABSTRACT

Tampereen ammattikorkeakoulu
Tampere University of Applied Sciences
Business Information Systems
Software Development

PELTOLA, TARU:
Build Configuration and Release Automation for Android

Bachelor's thesis 43 pages, appendices 5 pages
April 2018

The purpose of this thesis was to test automating release updates in variant based Android application. The study is to be utilized in one of the customer projects of the client, Futurice. A variant based Android test application was developed and configured during this thesis work. With the application it was possible to try out release automation in practice using a fastlane app automation tool.

The objective of this thesis was getting a better understanding about configuration files in Android project and creating a report about fastlane setup and usage. The aim of the fastlane was to automate build updates and releases for Google play and HockeyApp platforms. By utilizing the test project and documentations about fastlane it was easy to take into use in the customer project. The aim was to recreate the situation of the customer project and by testing in practice in order to find a solution from fastlane to ease manual releasing.

SpiceCabinet application was created to represent the customer project. It was configured to use different build variants. The application variants were uploaded into Google Play and HockeyApp. The challenge was to have an automation that supports build variants. The variants were supposed to be able to separate from one another during a lane run. By using variant specific environment variables, a dynamic solution was created, and new variables were easy to add.

The result of this thesis was an automation process for release updates which takes into account support for build variants. Versatile instructions were created about the setup and variant usage of the fastlane tool. Fastlane was easy to setup in the customer project as well. In the future, when the automation process needs to be improved in a customer project, it is possible to take advantage of the test application.

Key words: android, release, automation, build, configuration, fastlane, HockeyApp

ESIPUHE

Opinnäytetyöprojekti sai alkunsa, kun työn toimeksiantajan asiakasprojektissa nähtiin tarve julkaisuautomaatiolle ja koin hyödylliseksi sekä mielekkääksi testata automaatiotyökalun käyttöönottoa käytännössä. Lisäksi halusin oppia ymmärtämään Android-projektin konfigurointia, jota automaation testaaminenkin vaati.

Testisovellus työstettiin osana toimeksiantajayrityksen, Futuricen, Spice Programia. Kyseessä on sponsoroitu avoimen lähdekoodin ja sosiaalisen vaikuttamisen ohjelma (Spice Program n.d.). Se kannustaa kehittämään vapaa-ajalla ohjelmointiosaamista, lisäämään vapaa-ajan projekteja julkiseen versionhallintaan ja jakamaan osaamista muiden ihmisten kesken. Spice Programin idea on oppia ja opettaa.

Haluan kiittää koko projektitiimiäni, joka on ollut tukena projektin edetessä ja lopullisen työn esitarkastuksessa. Erityisesti kiitos Ville Keski-Nikkola, kun olet toiminut mentorinani ja opettanut minulle Androidin saloja. Olet ollut jälleen varsinainen Android-Yoda. Iso kiitos myös Joose Fjällström, kun olet ollut niin timanttinen tiimikaveri, auttanut ratkaisemaan minunkin pulmiani ja opetellut kanssani paremmin ymmärtämään fastlanen sielunmaisemaa. Lopuksi kiitän kaikkia niitä Tammerforcelaisia, joilta olen saanut apua ja kannustusta tämän opinnäytetyön toteutukseen.

Tampereella 24.4.2018

Taru Peltola

SISÄLLYS

LYHENTEET JA TERMIT

APK	(Android Application Package) Android-sovellusten tiedostotyyppi.
Fastfile	Fastlanen alustama tiedosto, johon määritellään käytössä olevat kaistat.
Fastlane	Avoimen lähdekoodin työkalu applikaatioautomaatioon.
Groovy	Dynaaminen skriptikieli java-alustalle.
HockeyApp	Sovelluksen julkaisualusta ja kehitystyökalu.
JSON	(JavaScript Object Notation) Tekstipohjainen, selväkielinen ja yksinkertainen avoimen standardin tiedostomuoto tiedonvälitykseen.
Ruby	Oliopohjainen avoimen lähdekoodin ohjelmointikieli.
SDK	(Software Development Kit) Kokoelma ohjelmistokehitystyökaluja, joiden avulla voidaan kehittää ohjelmistoja tietyille ohjelmistoalustoille.

1 JOHDANTO

Android-sovelluspäivitysten lisääminen Google Play -sovelluskauppaan on suoraviivainen ja suhteellisen nopea prosessi. Uusi sovellus ladataan palveluun edellisen tilalle, kirjoitetaan muutosloki päivityksen uusista ominaisuuksista ja määritellään, kuinka suurelle käyttäjämäärälle uusi versio tarjotaan. Tämä prosessi voi kuitenkin olla työläs, mikäli sovelluksia päivitetään kerralla useita ja mukaan lasketaan sovellusten koonti sekä lisääminen HockeyApp-työkaluun tai vastaavaan palveluun sovelluksenhallintaa varten. Pahimmassa tapauksessa päivitys tuo esiin uusia virheitä, joista seuraa uusi sovelluspäivityskierros.

Opinnäytetyön toimeksiantajana toimii Futurice Oy, Helsingissä vuonna 2000 perustettu web- ja mobiilikehitykseen keskittyvä ohjelmistoyritys. Se on kasvanut nopeasti eurooppalaiseksi, kansainväliseksi it-alan tekijäksi, joka ratkaisee asiakkaidensa liiketoimintaongelmia teknologian, designin ja datan keinoin sekä kehittää yhdessä yritysten kulttuuria ja toimintamalleja (ContactForum, Futurice n.d.).

Tarve opinnäytetyölle syntyi toimeksiantajayrityksen asiakasprojektista, jossa kohdattiin manuaalisen julkaisun ongelmat. Projekti sisältää useita eri sovellusvariantteja. Jokainen erillinen sovellusvariantti pohjautuu samaan koodiin, mutta koska jokaisella on yksilöllinen konfiguraatio, niille määräytyy erilainen sisältö. Kun päivityksiä tehdään, kaikki sovellusvariantit halutaan julkaista kerralla Google Playssa. Lisäksi ne päivitetään myös HockeyApp-sovelluksenhallintatyökaluun, johon käyttäjillä ilmenneet virhetilanteet tallentuvat. Sovellusvariantit menevät keskenään helposti sekaisin, sillä niiden kansiorakenne on lähes identtinen. Julkaisu vaatii sekä HockeyAppin, että sovelluskaupan puolella toimintoja, joissa liikutellaan samannimisiä tiedostoja drag and drop -tekniikalla. Jo muutaman sovellusvariantin kohdalla voidaan sanoa, että julkaisutyö vie paljon aikaa kehittäjältä. Sovellusvarianttien määrä on pysynyt pitkään alle viidessä ja manuaalinen julkaisu on ollut vielä hallittavissa. Seuraavan vuoden aikana uusia variantteja on tulossa noin nelinkertainen määrä ja sitä ennen julkaisuprosessi halutaan saada automatisoitua.

Opinnäytetyön tavoitteena on tutkia, miten konfiguroidaan ja kehitetään sovellusvariantteihin perustuva sovellus Android-alustalle sekä testataan ja dokumentoidaan vaiheet fastlane-automaatiotyökalun käyttöönottoon. Lopputuotosten avulla fastlanen lisääminen

osaksi asiakasprojektia on mahdollisimman suoraviivainen ja nopea työ. Koska asiakasprojekti on pitkälle edennyt ja julkaisua täytyy voida testata tuotannon ulkopuolella, testaamista varten luodaan testisovellus.

Opinnäytetyön tarkoitus on testata käytännössä, kuinka manuaalista julkaisutyötä voitaisiin helpottaa fastlanen avulla. Samalla tarkoituksena on tutustua Android-sovelluksen koonnin konfigurointiin ja erityisesti tuotevarianttien luomiseen. Fastlanea varten testisovellukset julkaistaan Google Play -sovelluskaupassa ja ne lisätään HockeyAppiin, jotta lähtökohdat ovat samat kuin asiakasprojektissa.

Tämän opinnäytetyön ensimmäinen luku käsittelee testisovelluksen luomista ja kehitystä. Esittelen lyhyesti, millainen sovelluksesta tuli ja miltä eri sovellusvariantit näyttävät. Seuraavaksi käyn läpi tarkemmin koonnin konfigurointitiedostoja ja -sääntöjä, jotka ovat oleellisia yleisellä tasolla tai testisovelluksen osalta. Koska testattavat sovellusvariantit julkaistaan Google Playssa ja lisätään HockeyAppiin, tuon esille joitakin niihin liittyviä huomioita. Erityisesti HockeyAppista esitän lyhyen katsauksen: mitä hyötyä siitä on ja miten se otetaan käyttöön. Kuudes luku käsittelee fastlanea, sen käyttöönottoa ja testisovelluksen osalta tärkeitä toimintoja. Siinä selvitetään, kuinka sovellusvariantteja voi hallita fastlanessa ja miten ympäristömuuttujia käytetään.

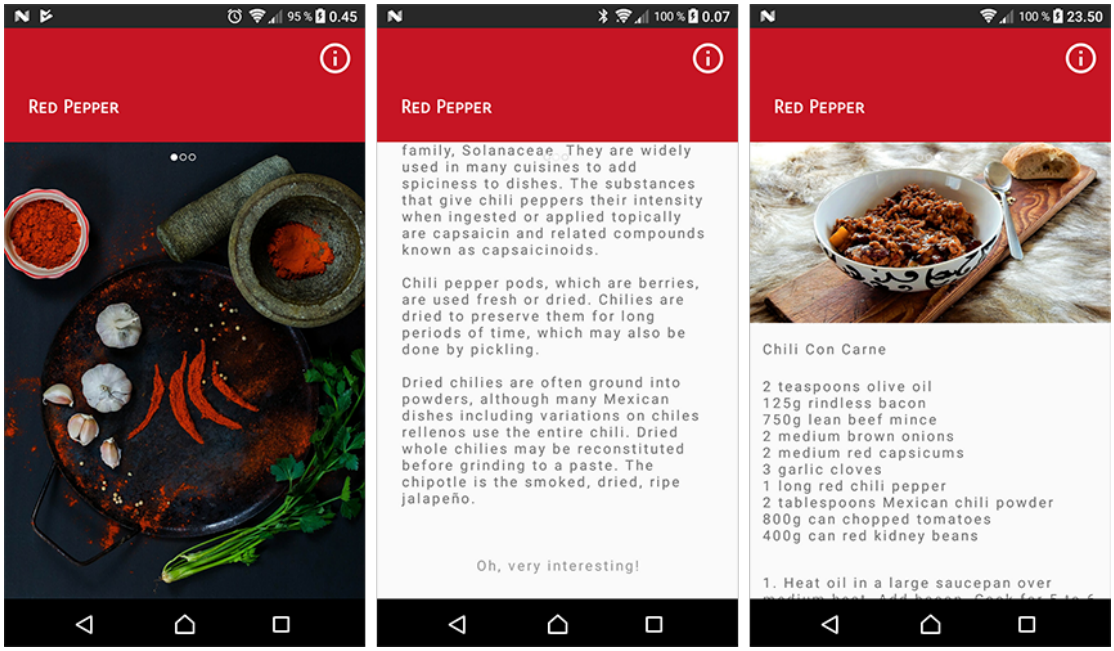
Tässä opinnäytetyössä kursiivia käytetään terminaalikomentojen erotteluun sekä suoriin lainauksiin. Lihavoinnilla viitataan joko tiedostonimeen tai toimintaohjeiden kannalta tärkeisiin termeihin, linkkeihin sekä painikkeisiin.

2 SPICE CABINET -TESTISOVELLUS

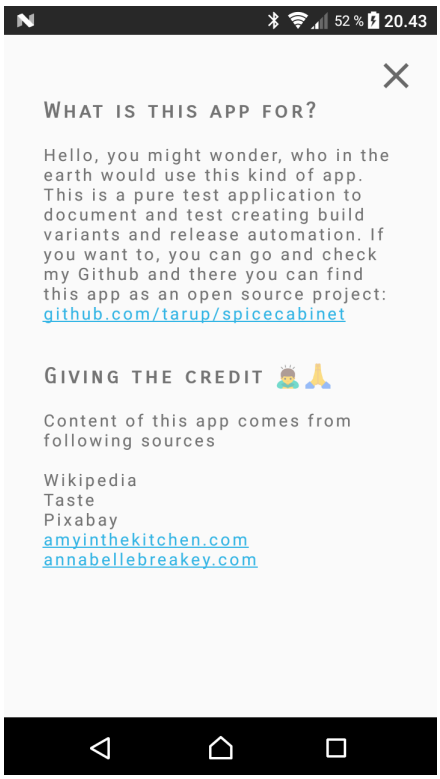
Julkaisuautomaation testaukseen luotu projekti, SpiceCabinet koostuu eri sovellusvarianteista, joita luotiin tämän opinnäytetyön puitteissa kolme: Red Pepper, Ginger ja Vanilla. Projekti on kehitetty Android Studion 3.0 -versiolla ja sovellusten alhaisin tuettu SDK-versio on 21. Kaikki kolme sovellusvarianttia on nähtävissä ja ladattavissa Google Play -kaupassa. Koko SpiceCabinet projekti on saatavilla avoimena lähdekoodina Githubista osoitteesta https://github.com/tarup/spicecabinet.

Sovellusprojektia kehitettäessä oli tärkeä huomioida, ettei toiminnallisuuksilla tai ulkoasulla ollut merkittävästi väliä. Tarkoitus ei ollut tehdä liian monimutkaista sovellusta, kun opinnäytetyöprojektin lopputavoite oli ainoastaan ottaa käyttöön ja testata automaation tuomat ominaisuudet fastlanen avulla. Toki projektin julkaisu avoimena lähdekoodina toi oman painoarvonsa osaamisen esilletuomiselle ja siististi kirjoitetulle koodille.

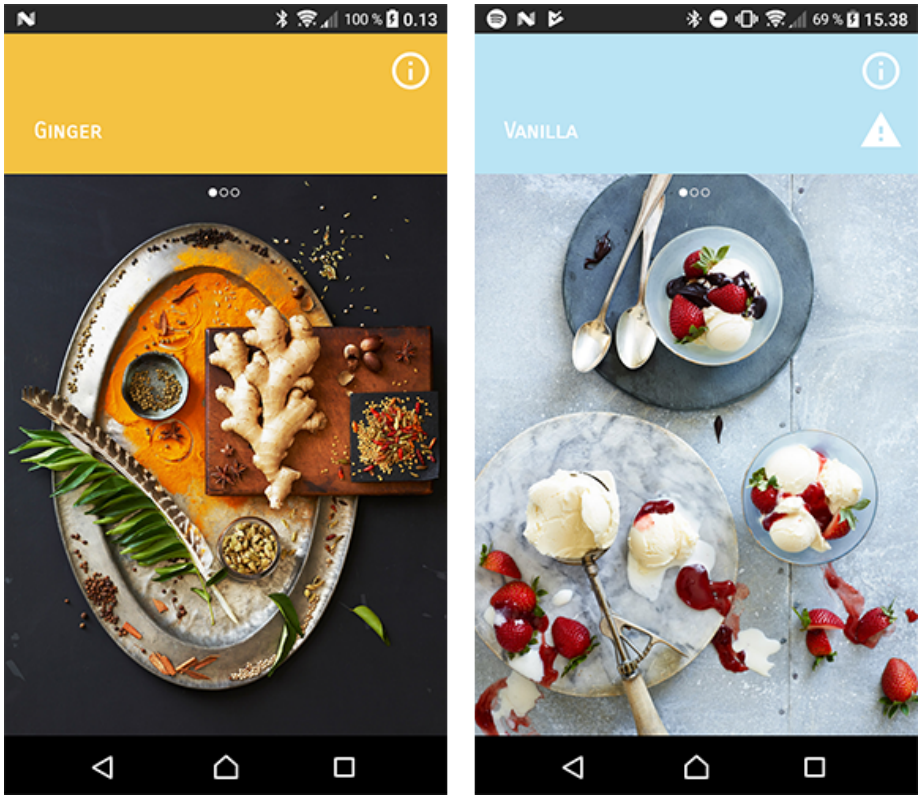
Projektin suunnittelun alkuvaiheilla selvitettiin, millaista sisältöä sovelluksiin voisi tuoda, jotta ne erottuisivat selkeästi toisistaan. Päädyttiin esittämään variaatiokohtaista kuva- ja tekstisisältöä helppouden vuoksi. Erilaisia sivunäkymiä syntyi kaiken kaikkiaan kolme (ks. Kuva 1). Testisovelluksessa on kovakoodattu näkymien määrä yksinkertaisuuden vuoksi. Toki sovelluksella olisi voinut olla oma tiedostonsa konfiguraatioon, joka määrittelee eri määrän sivunäkymiä varianttikohtaisesti. Kuva- ja tekstisisällön ohelle sovellukseen oli tarve luoda jokin informoiva näkymä mahdollista käyttäjää varten. Infonäkymästä avautuu sovelluksen käyttötarkoitus ja sisältölähteet (ks. Kuva 2). Lopuksi sovellusvariantit jaoteltiin vielä väriteemoihin, joka erottelee sovellukset selväpiirteisesti (ks. Kuva 3).



KUVA 1. Sivunäkymät Red Pepper –sovellusversiossa



KUVA 2. Infonäkymä



KUVA 3. Ginger ja Vanilla –sovellusversiot

Teknisesti testisovellus on yksinkertainen. Siinä hyödynnettiin Androidin omaa ViewPager-komponenttia. ViewPager tarjoaa nopean ja helpon tavan luoda selattavia sivunäkymiä, jotka voivat merkittävästi erota toisistaan. Lopputulos on ikään kuin kirja, jota selataan laidasta toiseen. ViewPager ei mahdollista sivujen jatkuvaa ympäri selaamista, mutta tähän projektiin se soveltui oikein hyvin. Kuten Androidin oma dokumentaatiokin ohjeistaa, komponentille tulee antaa PagerAdapter (ks. Koodiesimerkki 1), joka reagoi ViewPager-komponentissa tapahtuviin muutoksiin. Sivua vaihdettaessa PagerAdapterin metodi getItem käsittelee saamansa position. Jokaiselle positiolle määritellään, mikä fragmentti sen kohdalla kuuluu näyttää.

Jotta väriteema pysyi johdonmukaisensa, lisättiin ViewPager pienemmän FrameLayoutin sisään. Jokainen sivunäkymä sisältää oman fragmenttinsa, mutta niiden kaikkien tehtävä on käytännössä näyttää tietynlaista dataa, eikä mihinkään ole lisätty toiminnallisuutta. Tekstisisällöt haetaan varianttikohtaisesti asset-hakemistosta, JSON-tiedostosta, joka on kirjoitettu jokaista tuotetta varten erikseen. Kuvat on lisätty drawable-kansioihin. Nimeämiskäytännöissä on otettu huomioon, että ne täsmäävät kaikkien sovellusvarianttien kesken. (Tarkemmin kansiorakenteiden luonnista aliluvussa 3.5 Lähdehakemistot / Source sets.)

```
private ViewPager mPager;

@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_pager);

    mPager = findViewById(R.id.pager);
    PagerAdapter mPagerAdapter =
                            new PagerAdapter(getSupportFragmentManager());
    mPager.setAdapter(mPagerAdapter);
}


private class PagerAdapter extends FragmentPagerAdapter {
    private PagerAdapter(FragmentManager fm) {
        super(fm);
    }

    @Override
    public Fragment getItem(int position) {
        switch(position) {
            case 0: return PageFragment.newInstance();
            case 1: return WikiFragment.newInstance();
            default: return RecipeFragment.newInstance();
        }
    }
```

KOODIESIMERKKI 1. PagerAdapterin käyttö

3 SOVELLUSVARIANTIT JA PROJEKTIN KONFIGUROINTI

Tässä luvussa tutustutaan Android-projektin konfigurointiin liittyviin vaiheisiin ja sääntöihin. Samalla selviää, millaisia hyötyjä konfigurointi tuo projektille. Projektin konfigurointia tarkastellaan ensisijaisesti varianttipohjaisen testisovelluksen näkökulmasta.

3.1 Gradle

Gradle-tiedostot ovat se välttämätön, joskin pieni suo, jonka läpi tarpomista moni Android-kehittäjä karttaa viimeiseen asti. Jopa julkaisu Google Play -sovelluskauppaan onnistuu muuttamatta kyseisiä tiedostoja manuaalisesti. Android Studio on hyvä esimerkki siitä, miten käyttäjäystävällinen kehitysympäristö voi olla, kun projektin luonti, testaaminen ja paketoiminen tapahtuvat parilla klikkauksella.

Gradle on koontityökalu ja liitännäinen, joka toimii erillään Android Studiosta. Projektin koonti onnistuu siis myös ilman Android Studiota esimerkiksi komentoriviltä. (Android Studio User Guide n.d.a.) Android-projektia voi toki kehittää muussakin ohjelmointiympäristössä ja niiden kesken voi vaihdella sekin, onko koontityökaluna käytössä Gradle, Maven vai jokin muu. Mavenin tapaan Gradle perustuu käytäntöihin ja sillä on oletuskäytännöt, joista poikkeaminen kerrotaan koontitiedostoissa (Mikkola 2016). Gradle käyttää Groovy-skriptikieltä, jolla käytäntöjä konfiguroidaan. Kun Android-projektia luodaan, käyttäjältä kysytään SDK-version alaraja ja applicationId. Gradle käyttää kysyttyjä alkutietoja luodakseen oletuskonfiguraation. Konfigurointia paremmin ymmärtääkseen tulee tutustua seuraaviin tiedostoihin: projektin nimi/build.gradle, settings.gradle sekä app/build.gradle.

3.2 Koontityypit ja Proguard

Kun uusi sovellusprojekti aloitetaan, Android Studio luo automaattisesti debug- ja release-koontityypit. Vaikka debug ei oletuksena näy koonnin konfigurointitiedostossa, se on konfiguroitu ja sovelluksen voi kääntää debuggerilla. (Android Studio User Guide

n.d.a.) Näiden koontityyppien erottuvin tekijä on proguard-tiedosto, joka määritellään release-koonnissa oletuksena ja debug-koonnissa se jätetään pois (ks. Koodiesimerkki 2). Google Play Console (n.d.) tiivistää termin hyvin: *"ProGuard pienentää, optimoi ja obfuskoi koodiasi poistamalla käyttämätöntä koodia ja nimeämällä uudelleen luokkia, kenttiä ja menetelmiä semanttisesti hämäävillä nimillä."* Proguard-sääntöjen tarkoituksena on siis kutistaa koonnin tuloksena syntyvästä APK-paketista mahdollisimman pieni. Tarkoitus on myös määritellä, mikä kohta koodista on ylimääräistä ja jätetään huomiotta. Obfuskointi on yksi proguardin tärkeimmistä ominaisuuksista, sillä se suojaa ohjelmaa hankaloittamalla koodin takaisin kääntämistä lähdekoodiksi. Lisäksi obfuskointi on myös yksi tapa tiivistää koodia. Esimerkiksi muuttujien ja metodien nimet käännetään satunnaisiksi ja mahdollisimman lyhyiksi merkkijonoiksi. Kun proguard määritellään release-koonnissa, koodia myös optimoidaan, jolloin käyttämätöntä eli turhaa koodia siivotaan pois. Proguard ennen kaikkea suojaa tiedostoa, mutta se tarjoaa myös mahdollisimman pienen ja nopeasti ladattavan tiedoston käyttäjille. Sen sijaan debug-koonnin halutaan näyttävän kommentteja myöten kaikki mahdollinen, mitä ajo aikana tapahtuu.

```
buildTypes {
    debug {
        versionNameSuffix "-Debug"
    }
    release {
        minifyEnabled true
        proguardFiles getDefaultProguardFile('proguard-android.txt'),
                                                    'proguard-rules.pro'
        signingConfig signingConfigs.config
    }
}
```

KOODIESIMERKKI 2. Koontiversioiden määrittely.

Kuten koodiesimerkki 2 esittää, release-koonnissa määritellään lisäksi **minifyEnabled**. Kyseinen arvo on oletuksena epätosi, joka tulee vaihtaa todeksi, jotta proguard-säännöt otetaan käyttöön. MinifyEnabled käytännössä sallii tai estää proguardfilen käytön (Android Studio User Guide n.d.a). Koontityyppejä voi tarvittaessa lisätä ja konfiguroida omiin tarpeisiin. Uutta koontityyppiä pystytään kääntämään debuggerilla ilman muutoksia projektin rakenteeseen (project structure), sillä uusissa koontiversioissa on oletuksena määritelty "debuggable" todeksi. Release-koonnissa tämä on tietysti oletuksena epätosi.

3.3 Product flavors

Product flavors tarkoittaa tuotevarianttia ja juuri nämä flavorit mahdollistavat eri sovellusvarianttien lisäämisen. Tuotevarianttien määrittelystä on kyse myös testisovelluksessa (ks. Koodiesimerkki 3). Varianttien lisääminen on hyvin tavallista esimerkiksi silloin, jos halutaan tehdä sekä ilmainen, että maksullinen versio samasta sovelluksesta. Tuotevariantilla ei ole tarkoitus luoda täysin erilaista sovellusta, vaan muuttaa hiukan alkuperäisen sisältöä. Maksullinen versio voisi esimerkiksi poistaa mainokset tai paljastaa lisäsisältöä käyttäjälle.

```
flavorDimensions "flavorApp"
productFlavors {
    redpepper {
        applicationId "fi.tarup.spicecabinet.redpepper"
        dimension "flavorApp"
    }
    ginger {
        applicationId "fi.tarup.spicecabinet.ginger"
        dimension "flavorApp"
    }
    vanilla {
        applicationId "fi.tarup.spicecabinet.vanilla"
        dimension "flavorApp"
    }
}
```

KOODIESIMERKKI 3. Product flavoreiden ja Flavor Dimensioiden määrittely.

Jokaiselle uudelle product flavor -arvolle määritellään oma applicationId. Tässä kohtaa defaultConfigin luomaa applicationId:tä ei tarvita. Id toimii sovelluksen tunnuksena sovelluskaupassa ja se ei siis voi olla sama product flavor -arvojen kesken. Kun product flavorit on luotu ja konfiguroitu sekä projekti synkronoitu, Gradle automaattisesti luo koontiversiot pohjautuen määriteltyihin koontityyppeihin ja product flavoreihin (Android Studio User Guide n.d.b). Se nimeää ne <product-flavor><koontityyppi> -mallin mukaisesti. Mahdollisten koontiversioiden määrä on siis koontityyppien määrä kerrottuna product flavoreiden määrällä.

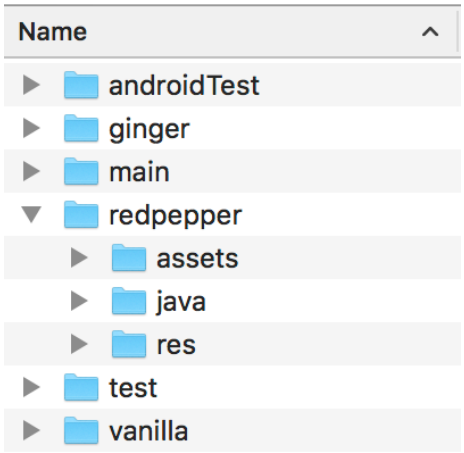
Dimension on arvo, jonka avulla määritellään, mihin varianttidimensioon (flavor dimension) kyseinen variantti kuuluu. Testisovelluksen tapauksessa product flavorit kuuluvat kaikki samaan dimensioon kuten koodiesimerkissä 3. Flavor dimensions -kohdassa voi listata useita eri dimensioita. Jos esimerkiksi halutaan testisovelluksen variantit konfiguroida vielä eri API-tasoille, flavor dimension mahdollistaa tämän. (Android Studio User

Guide n.d.b.) Käytännössä lisäämällä toinen varianttidimensio Gradle kokoaa koontiversiot <product-flavor><product-flavor><koontityyppi> -mallin mukaan.

3.4 Lähdehakemistot / Source sets

Source sets tarkoittaa lähdehakemistoja, jotka voidaan määritellä jokaiselle koonti- tai tuoteversiolle erikseen. Lähdehakemistot kuuluvat app/src-polun alle ja ne sisältävät tarvittavat lähdekoodit ja resurssit kyseiselle versiolle. Android Studio ei tee lähdehakemistoja automaattisesti, kun koonti- tai tuoteversio lisätään. Jokainen niistä käyttää siis oletuksena app/src/main-hakemistoa. Luomalla esimerkiksi tuoteversiolle oman kansion, joka on nimetty sen mukaisesti, Gradle osaa koostaa sovelluksen käyttäen ensisijaisesti kyseisen variantin hakemistosta löytyvää koodia ja resursseja. Vasta sen jälkeen se etsii main-hakemistosta tarvitsemaansa. Tämä helpottaa huomattavasti tiedostojen ja lähteiden organisointia, mutta mahdollistaa myös sisällön vaihtumisen eri koonti- tai tuoteversioiden välillä. Myös yhteisiä resursseja voidaan määrittää yhdistelemällä variantteja keskenään (esim. src/gingervanilla) tai koontiversioiden kanssa (esim. src/vanillarelease). (Android Studio User Guide n.d.a.)

Testisovelluksen app/src hakemistorakenteeseen lisättiin ainoastaan sovellusvariantit (ks. Kuva 4). Koontityyppejä varten ei luotu poikkeavia lähdehakemistoja, mutta product flavoreille lisättiin /assets-, /res- ja /java-kansiot. Niihin säilöttiin JSON-data, kuvat ja varianttikohtaiset lokalisointitiedostot. Ainoa Java-luokka ja koodi, joka määriteltiin varianttikohtaisesti testisovelluksessa, oli VariantConstants. Kyseinen tiedosto lisättiin jokaiselle flavorille ja siihen säilöttiin sovelluskohtainen HockeyApp id. Varianttien sisältöä voisi konfiguroida JSON-datassa ja näin saataisiin hyvinkin erinäköisiä sovelluksia.



KUVA 4. Testisovelluksen app/src hakemistorakenne.

Hakemistorakenteita muutettaessa Gradle ei aina välttämättä pysy ajan tasalla. Kun jotain on lisätty tai poistettu voi olla, että seuraavan kerran ohjelma ei käännykään. **Clean project** on ensimmäinen asia, jota kannattaa kokeilla, mikäli koonti epäonnistuu source set -muutosten jälkeen.

3.5 Allekirjoituskonfiguraatio ja keystore

Sovelluksen julkaisu Google Playhin vaatii, että käyttäjällä on sovelluksen allekirjoitusavain (app signing key). Kun APK varmennetaan tällä avaimella, APK:hon kiinnitetään public-key sertifikaatti. Se toimii ikään kuin sormenjälkenä, joka voidaan yhdistää kehittäjään ja hänen omistamaansa yksityiseen avaimeen (private key). Tällä varmistetaan, että tulevaisuudessa sovelluspäivitykset ovat autenttisia ja tulevat oikealta julkaisijalta. Android Studiolla voi manuaalisesti generoida allekirjoitetun APK:n julkaisua varten, joko yksitellen tai eri sovellusvarianteille. Allekirjoitus voidaan myös automatisoida konfiguroimalla Gradle käsittelemään se koonnin yhteydessä. (Android Studio User Guide n.d.c.)

Allekirjoitusavain on ehdottoman tärkeä suojata sekä kehittäjän, että käyttäjien vuoksi ja se tulisi pitää salassa. Kun allekirjoituskonfiguraatio lisätään (joko muuttamalla Android Studion projektirakennetta tai kirjoittamalla signingConfigs-tiedot suoraan build.gradleen), kaikki allekirjoitukseen tarvittava tieto on nähtävillä. Allekirjoituskonfiguraatio koostuu keystore-lokaatiosta, keystore salasanasta sekä avaimen aliaksesta ja salasanasta.

Jos työskentelee tiimissä tai open source -projektin parissa, tätä sensitiivistä informaatiota ei tulisi missään nimessä jättää näkyville. (Android Studio User Guide n.d.c.)

Allekirjoituskonfiguraatiota varten luodaan Android-dokumentaation mukainen **keystore.properties**-tiedosto, joka sisältää tarvittavat avain-arvo parit. Tiedosto lisätään projektin juurihakemistoon ja siihen kirjataan allekirjoitusta varten seuraavat tiedot.

```
keyAlias=myKeyAlias
keyPassword=myKeyPassword
storeFile=myStoreFileLocation
storePassword=myStorePassword
```

Koodiesimerkissä 4 näkyy, kuinka keystorea käytetään app/build.gradle-tiedostossa. Aluksi luodaan keystorePropertiesFile-niminen muuttuja ja alustetaan se juurihakemiston keystore.properties-tiedostolla. Seuraavaksi luodaan keystoreProperties muuttuja, joka viittaa Properties-objektiin. Se lataa keystorePropertiesFilen avain-arvo-parit ja viittaamalla kyseiseen objektiin voidaan hakea haluttu avain-arvo.

```
def keystorePropertiesFile = rootProject.file("keystore.properties")
def keystoreProperties = new Properties()
keystoreProperties.load(new FileInputStream(keystorePropertiesFile))

android {
    signingConfigs {
        config {
            keyAlias keystoreProperties['keyAlias']
            keyPassword keystoreProperties['keyPassword']
            storeFile file(keystoreProperties['storeFile'])
            storePassword keystoreProperties['storePassword']
        }
    }
```

KOODIESIMERKKI 4. Allekirjoituskonfiguraation määrittely keystoren avulla.

Keystore.properties-tiedosto jätetään lopuksi pois versionhallinnasta kirjaamalla se gitignore-tiedostoon. Näin voidaan välttää allekirjoituskonfiguraation luvaton käyttö.

4 PROJEKTIN JULKAISU JA HALLINTA

Tässä luvussa on tuotu esiin asioita, jotka kannattaa huomioida Google Play -sovelluskauppajulkaisussa, kun sovellusta on konfiguroitu. Lisäksi toteutetaan HockeyApp-työkalun käyttöönotto. Jotta fastlane-automatisointia voidaan hyödyntää, sovellus tulee julkaista ensimmäisen kerran manuaalisesti sekä Google Play -kauppaan, että alustaa HockeyAppiin.

4.1 Google Play - ennen julkaisua

Kun gradle-tiedostoja on konfiguroitu ja päätetään tehdä uusi julkaisu, sovelluksen kääntämistä ja testaamista release-versiona ei kannata unohtaa. Kehitettäessä applikaatiota debug-tilassa vältytään ongelmilta, jotka liittyvät luokkien tai kirjastojen linkitykseen. Julkaisuversiossa kuitenkin on tapana käyttää proguard-sääntöjä, jolloin obfuskoinnissa voidaan nimetä uudelleen muuttuja tai metodi siten, että linkitys rikkoutuu. Oli kyse sitten kolmannen osapuolen kirjastoista tai omista DTO-luokista (Data Transfer Object), release-versio saattaa muuttua siten, ettei sovellus enää käänny. Tällöin tulee varmistaa, että proguard-säännöt ovat kunnossa.

Generoimisen jälkeen tulisi APK-paketti vielä testata. Testauksessa varmistetaan, että käännös on tapahtunut ilman häiriöitä, APK-paketti asentuu laitteelle ja sovellus edelleen toimii. Hyvä työkalu tähän on Android Debug Bridge eli adb. Tavallisesti se asennetaan Android SDK:n yhteydessä automaattisesti, ja adb toimii komentorivillä android_sdk/platform-tools -hakemistossa. Testilaite liitetään kiinni koneeseen ja varmistetaan, että adb löytää laitteen komennolla *adb devices*. Seuraavaksi APK-paketti asennetaan komennolla *adb install -r <paketin sijainti hakemistossa>*. Komento korvaa aiemmin ladatun samannimisen tiedoston ja näin voidaan varmistaa, että sovellus päivittyy myös oikein.

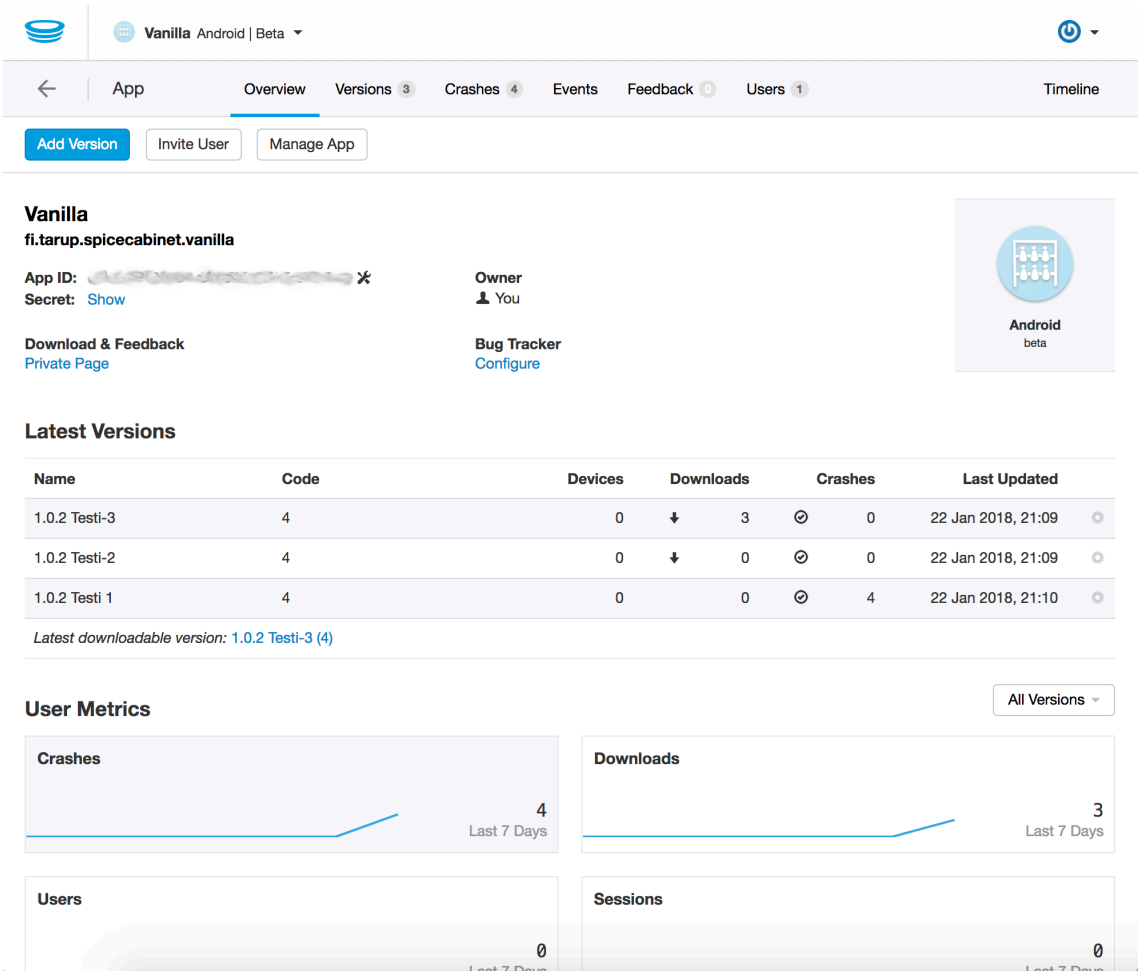
Google Play Console osaa luoda stack tracen eli raportin sovelluksen ajonaikana tapahtuneesta virheestä. Stacktrace listaa, mitä kutsuja oli käynnissä virhetilanteen syntyessä. (Sanasto n.d.) Kyseistä raporttia on kuitenkin hankala tulkita, mikäli proguard on muuttanut luokkien tai metodien nimiä. Jotta nämä raportit olisivat luettavassa muodossa, ne

tulee deobfuskoida eli kääntää obfuskoitu data alkuperäiseksi. Tämä onnistuu **mapping.txt**-tiedoston avulla, joka tarjoaa käännöksen alkuperäisen ja obfuskoidun version välillä (Android Studio User Guide n.d.d). Mapping-tiedosto löytyy app/build/outputs/mapping -lähteestä ja se tulee lisätä Google Play Consoleen. Oikea paikka tiedostolle löytyy sovellusnäkymän Android Vitals -listan pohjimmaisesta linkistä **Tietojen selventämistiedostot** (Deobfuscation files).

4.2 HockeyApp-työkalun käyttöönotto

HockeyApp on korkean luokan devops-työkalu eli sovelluksenhallintaohjelma. Sen tarkoitus on auttaa kehittäjää ymmärtämään sovelluksen kehitystarpeet ja helpottaa ylläpitotyötä. HockeyApp voi toimia käyttäjän omana henkilökohtaisena sovellusjulkaisualustana, johon voi julkaista applikaation ja päättää, kenellä on valtuudet nähdä ja asentaa kyseinen applikaatio. Toisin sanoen se toimii kanavana kehittäjien sekä testaajien välillä, ja sillä ohitetaan oikeat sovelluskaupat. HockeyApp kerää testikäyttäjiltä ja julkaisuversion sovelluskauppakäyttäjiltä sovelluksen kaatumisista raportit yhdistäen ne samaan paikkaan. Ohjelma luo myös erilaisia kaavioita kaatumisista, lataus- ja käyttäjämääristä sekä sessioista. (HockeyApp n.d.)

HockeyAppin käyttöönotto etenee niin, että rekisteröitymisen jälkeen (esim. Google-tilillä) kehittäjää ohjataan lisäämään ensimmäinen sovellus pudottamalla APK-paketti etusivulle, dashboard-näkymään. APK-paketti latautuu HockeyAppiin, ja kehittäjä voi kirjata ylös muistiinpanot julkaisusta selventäen, mitä muutoksia kyseinen versio sisältää. Seuraavaksi määritellään asetuksia: onko sovellus ladattavissa HockeyAppin kautta, onko se kaikille ladattavissa vai määritelläänkö erikseen, mille ryhmälle lataaminen sallitaan. Käyttäjä valitsee vielä, lähetetäänkö ilmoitus ryhmän sisällä uudesta sovellusversiosta. Sama prosessi toistuu, kun uusi sovellusversio halutaan taas lisätä. Se tapahtuu klikkaamalla **Add Version** -painiketta sovellusnäkymän vasemmassa yläkulmassa (ks. Kuva 5). Uuden sovelluksen voi halutessaan alustaa pelkän sovellusnimen ja paketin nimen perusteella. Tällöin sovellus saa oman HockeyApp id:n, jonka avulla se saatetaan ensimmäisen kerran julkaista käyttämällä fastlanea.



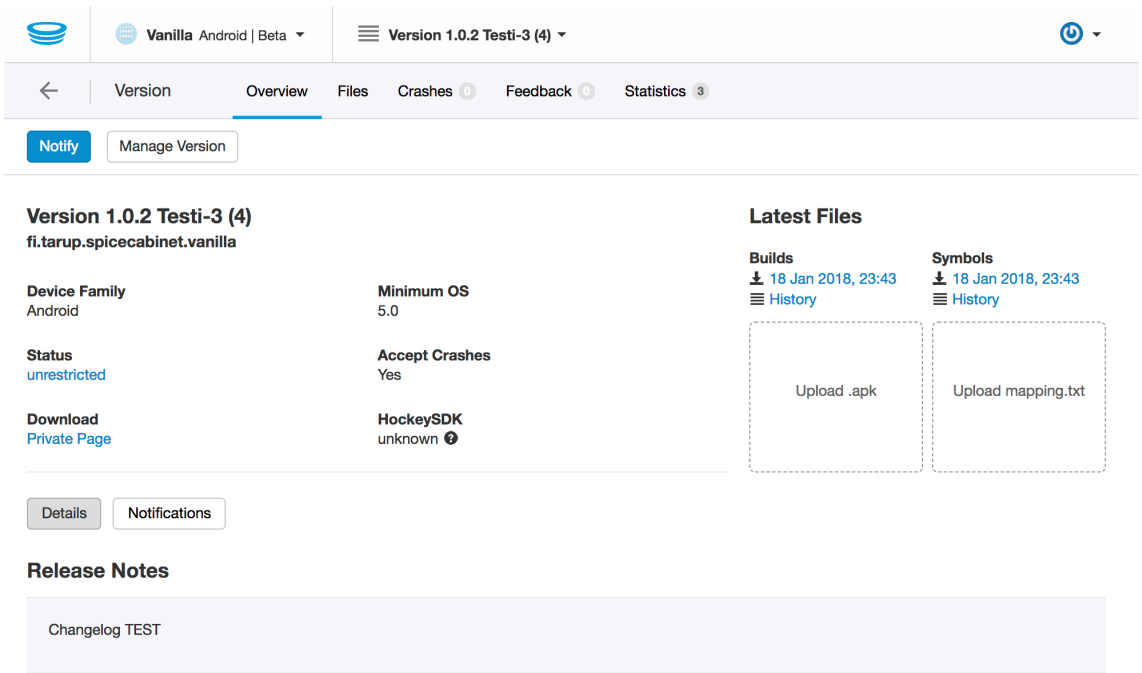
KUVA 5. Sovellusnäkymä, HockeyApp

HockeyAppia voi käyttää keräämään raportit vain rajatulta testiryhmältä, mutta siihen voidaan linkittää myös Google Play -käyttäjät. Jotta tieto siirtyisi käyttäjien sovelluksista HockeyAppiin, tarvitaan hiukan muutoksia myös projektin koodiin. Implementoidaan ensin viimeisin HockeySDK osaksi projektia ja rekisteröidään CrashManager MainActivityssa (ks. Koodiesimerkki 5). Sille tarjotaan konteksti, sovellustunnisteena HockeyApp id ja CrashManagerListener, joka kuuntelee sovelluksen kaatumisia.

```
@Override
protected void onResume() {
    super.onResume();

    CrashManager.register(this, VariantConstants.HOCKEY_APP_ID,
                                        new MyCrashManagerListener());

    Intent intent = new Intent(this, PagerActivity.class);
    startActivity(intent);
}

private static class MyCrashManagerListener extends CrashManagerListener {
    public boolean shouldAutoUploadCrashes() {
        return true;
    }
}
```

KOODIESIMERKKI 5. CrashManagerin lisääminen

Sovelluksen kommunikaatio HockeyAppin kanssa ei silti ole vielä aivan täydellinen. Jotta raporteissa varmasti viitattaisiin oikeisiin luokkiin ja mikään ei menisi rikki raportin luonnissa, tarvitaan myös HockeyAppiin mapping.txt-tiedosto. HockeyAppin versionäkymässä (ks. Kuva 6) oikeassa laidassa näkyy **Builds** ja **Symbols**. Builds-kohtaan ladataan sovelluksen APK-paketti ja symbols-kohdan alle pudotetaan mapping.txt-tiedosto.



KUVA 6. Versionäkymä, HockeyApp

5 ANDROID JULKAISUN AUTOMATISOINTI – FASTLANE

Tässä luvussa kerrotaan fastlane-työkalun käyttöönotosta. Lisäksi otetaan selvää, kuinka perustoiminnallisuuksien, koonnin ja julkaisun suorittaminen onnistuvat varianttipohjaisessa Android-sovelluksessa.

5.1 Fastlane

Fastlane on avoimena lähdekoodina työstetty työkalu julkaisun automatisointiin iOS- ja Android-alustoille (Fastlane n.d.a). Avoin lähdekoodi lisää aktiivisuutta kehitysyhteisössä ja siksi fastlane kehittyy nopealla tahdilla. Fastlane perustuu Ruby-ohjelmointikieleen. Sen tarkoitus on ennen kaikkea helpottaa sovelluksen koonti- ja julkaisutyötä (About Felix Krause n.d.).

Fastlanen idea perustuu kaistoihin. Kaistat (lanes) ovat skriptejä, joita luodaan eri toiminnallisuuksiin Rubyn avulla. (Fastlane n.d.b.) Fastfile-niminen tiedosto sisältää kaistat, jotka on tarkoitus suorittaa komentoriviltä käsin. Aikaisempaa Ruby-osaamista ei välttämättä tarvita, sillä kaistojen luonti on yksinkertaista ja fastlanen omat action-metodit auttavat alkuun. **Action** on fastlane-termi ja tarkoittaa tehtävää tai toimintaa, joka määritellään kaistan sisällä ja toteutetaan, kun kaista suoritetaan. Kaikki käytössä olevat actionit saa näkyviin komennolla *fastlane actions*. Toinen hyödyllinen komento on *fastlane action [action_name]*, joka listaa yksittäisen actionin tiedot ja käytössä olevat parametrit sekä niiden käyttötarkoituksen. Sama informaatio löytyy myös fastlanen omasta dokumentaatiosta, joka sisältää lisäksi joitakin esimerkkejä.

Koska fastlane kehittyy jatkuvasti, myös dokumentaatio muuttuu aika ajoin. Käyttöönoton vaiheet pohjautuvat fastlanen dokumentaatioon kirjoitushetkellä.

5.2 Alustus

Fastlanen ainut virallisesti tuettu käyttöjärjestelmä on macOS. Tuki muille käyttöjärjestelmille on vielä vajavainen ja testaamaton, mutta sitä kehitetään (Fastlane n.d.a). Fastlane käyttää sellaisia käyttöjärjestelmärajapintoja, joita ei ole välttämättä tuettu muilla alustoilla. Eli vaikka Rubya voi suorittaa Windowsilla, fastlane käyttää toistaiseksi sellaisia luokkia tai metodeja, jotka taas eivät ole yhteensopivia Windows-käyttöjärjestelmän kanssa.

Fastlanen lataaminen tapahtuu komennolla *[sudo] gem install fastlane -NV* tai mikäli Homebrew-pakettienhallintaohjelmisto on käytössä: *brew cask install fastlane*. (Fastlane n.d.b.)

Fastlanen alustus tapahtuu navigoimalla projektihakemistoon ja antamalla komento *fastlane init*. Ohjelma luo fastlane-kansion ja nopeuttaakseen projektin alustamista, se kysyy kolme asiaa: paketin nimi, json secret -tiedoston polku sekä haluaako käyttäjä ladata sovelluksen metadatan Google Play:stä. Käytettäessä fastlanea ensimmäistä kertaa kannattaa antaa ainoastaan paketin nimi ja ohittaa muut kohdat. Tämän vaiheen jälkeen fastlane-kansion alta löytyvät Appfile- ja Fastfile-tiedostot. Appfilen avulla fastlane määrittelee konfiguraation koko sovelluksen tasolle. Fastfileen määritellään ainoastaan kaistat. (Fastlane n.d.b.)

Fastlane huomauttaa saatavilla olevista päivityksistä ja ehdottaa todennäköisesti jo alustuksen jälkeen päivittämään uuteen versioon. Uusimman version lataamista suositellaan aina. Se onnistuu komennolla *fastlane update_fastlane.*

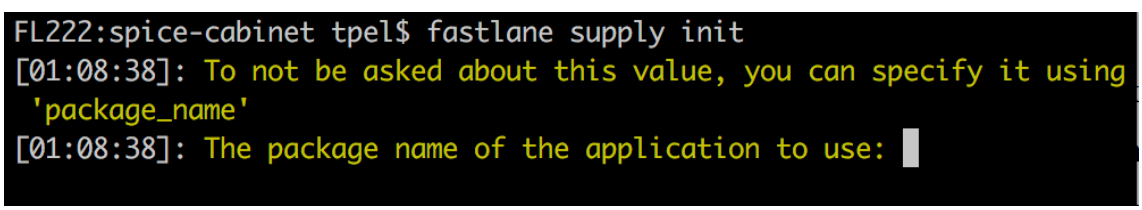
Supply on fastlane-työkalu, jonka avulla saadaan ladattua sovelluksen metadata, kuvakaappaukset ja binäärit Google Play -kauppaan. Käytännössä se hoitaa tarvittavan kommunikaation sovelluskaupan välillä, kun julkaisua tehdään. Supplyn käyttöönotto vaatii, että Googlen kehittäjäkonsolin palvelutililtä ladataan käyttöoikeustiedosto.

1. Avaa Google Play Console.
2. Avaa **Asetukset**-valikko ja Kehittäjätilin alta **Sovellusliittymän käyttöoikeus**.
3. Palvelutilit -otsikon alta klikkaa **Luo palvelutili**.

4. Seuraa dialogin Google API Console -linkkiä englanninkieliseen Google-kehittäjäkonsoliin.
5. Klikkaa **Create Service Account** -painiketta sivun yläpalkissa.
6. Anna palvelutilille nimi.
7. Avaa **Role**-pudotusvalikko ja valitse **Service Accounts** > **Service Account User**
8. Merkitse **Furnish a new private key** -valintaruutu.
9. Valitse avaimen tyypiksi JSON.
10. Klikkaa **Create** sulkeaksesi dialogin.
11. Ota ladattu JSON-tiedosto hyvään talteen, tarvitset sitä pian.
12. Palaa Google Play konsoliin ja klikkaa **Valmis** sulkeaksesi dialogin.
13. Klikkaa **Myönnä käyttöoikeus** -painiketta uuden palvelutilin kohdalla.
14. Valitse **Rooli**-pudotusvalikosta **Julkaisupäällikkö**
15. Klikkaa **Lisää käyttäjä** sulkeaksesi dialogin.

Seuraavaksi voidaan editoida Appfile-tiedostoa ja muuttaa oletuksena näkyvään **json_key_file** kohtaan polku, johon käyttöoikeustiedosto tallennettiin (json_key_file "/path/to/your/downloaded/key.json"). Mikäli paketin nimi on lisätty jo alustusvaiheessa, supply voidaan seuraavaksi määritellä projektihakemistoon komennolla *fastlane supply init*. Näin fastlane-kansion sisään luodaan metadata-niminen kansio, johon ladataan Google Play -kaupasta löytynyt metadata. Käytännössä kyse on sovellusikonista, kuvausteksteistä ja viimeisimmistä muutosloki-tiedostoista. Fastlane jaottelee metadatan kielikohtaisesti. Google API:n rajoituksista johtuen supply ei vielä pysty lataamaan olemassa olevia kuvakaappauksia tai videoita. (Fastlane n.d.b.)

Koska testisovellus koostettiin varianteista, alustus tuli hoitaa hiukan eri tavalla. Appfile-tiedostoon lisättiin lopulta vain käyttöoikeustiedoston polku- ja package_name-kohdat jätettiin tyhjiksi. Tämä siksi, että ei haluttu viitata ainoastaan yhteen sovellukseen. Jos paketin nimeä ei ole määritelty supply init -komennon yhteydessä, seuraa virhetilanne (ks. Kuva 7).



KUVA 7. Supply init -komennon suorittaminen, kun paketin nimi puuttuu.

Testisovelluksessa metadatan alustaminen ratkaistiin Fastfilessa (ks. Liite 1). Metadatan alustukselle lisättiin kaista, jossa suoritetaan shell-komento. Samalla määriteltiin package_name ja polku, johon metadata haluttiin ladata. Fastfilea muokattaessa kannattaa huomioida, että shell komennot eivät saa jakautua usealle riville. Rivitys kannattaa tarkistaa, jos komentoriville ilmestyy FastlaneShellError.

Fastlane vaatii muutamia ympäristömuuttujia toimiakseen oikein. Jos käyttäjän localea ei ole asetettu UTF-8:ksi, seuraa ongelmia koonnin ja lataamisen yhteydessä. Lisätään shell profileen seuraavat rivit:

```
export LC_ALL=en_US.UTF-8
export LANG=en_US.UTF-8
```

Käyttöjärjestelmästä riippuen shell profile löytyy ~/.bashrc-, ~/.bash_profile- tai ~/.zshrc-polusta. (Fastlane n.d.b.) Muutosten jälkeen tulee komentorivi sulkea ja avata uudelleen, jotta tehdyt muutokset tallentuvat.

5.3 Gemfile

Gemfile määrittelee tarvittavat riippuvuudet ja nopeuttaa fastlanen käyttöä (Fastlane n.d.b.). Käyttö vaatii Ruby-version 2.1 tai uudemman. Kuvassa 8 fastlane huomauttaa tarvittavasta päivityksestä. Varoituksen ohessa löytyvä linkki ohjaa bundler dokumentaatioon, jossa Rubyn päivittämiseen neuvotaan käyttämään komentoa *gem update –system*, jolla se päivittyy uusimpaan versioon.

```
FL735:spice-cabinet tpel$ bundle update
Warning: Your Ruby version is compiled against a copy of OpenSSL that is very
old. Starting in January 2018, RubyGems.org will refuse connection requests from
these very old versions of OpenSSL. If you will need to continue installing gems
after January 2018, please follow this guide to upgrade:
http://ruby.to/tls-outdated.
Fetching gem metadata from https://rubygems.org/.............
Fetching gem metadata from https://rubygems.org/..
Resolving dependencies...................
RubyGems 2.0.14.1 is not threadsafe, so your gems will be installed one at a tim
e. Upgrade to RubyGems 2.1.0 or higher to enable parallel gem installation.
Fetching CFPropertyList 2.3.6
```

KUVA 8. Ruby päivitettävä -varoitus komentorivillä.

Gemfilen lisääminen (Fastlane n.d.b):

1. Varmista, että olet projektihakemistossa (*pwd*) ja lataa bundler komennolla *[sudo] gem install bundler*
2. Projektin juurihakemistosta tulisi nyt löytyä ./Gemfile-tiedosto. Lisää siihen seuraavat rivit:

   source https://rubygems.org
   gem "fastlane”

   Nämä myös generoituvat automaattisesti, jos lisäät uuden riippuvuuden, eikä niitä ole vielä lisätty.
3. Suorita *[sudo] bundle update,* joka luo myös Gemfile.lock-tiedoston.
4. Lisää Gemfile ja Gemfile.lock osaksi versionhallintaasi.

Erityisesti Gemfile.lock-tiedoston kanssa saattaa ilmaantua tiedoston luku-/kirjoitusongelmia ja tarvitaan oikeudet tiedoston ylikirjoittamista varten. Komennolla *sudo chown $(whoami) <hakemisto, johon ei ole oikeuksia>* voidaan asettaa oikeudet tiettyyn tiedostoon, joka on tässä tapauksessa Gemfile.lock. (StackOverflow 2016b.)

Aina kun fastlane-komento suoritetaan, tulee käyttää muotoa:

bundle exec fastlane [lane]

Bundlerin **bundle exec** huolehtii, että käytössä olevat riippuvuudet sekä liitännäiset ovat käyttökelpoisia ja ne on mahdollista edellyttää (require) osaksi projektia. Joissakin tapauksissa ympäristömuuttujat ovat valmiiksi oikein ja riippuvuuksien edellytys toimii ilman Bundlerin apua. Silloinkaan ei ole haitaksi käyttää bundle exec -liitettä ja varmistaa, että käytössä olevat riippuvuudet ovat samat kuin Gemfilessa. (Danger 2015.) Jatkossa tässä opinnäytetyössä mainittavista fastlane-komentoesimerkeistä on jätetty bundle exec -alku kirjaamatta.

5.4 Käyttö

Fastlane perustuu kaistoihin, joiden luontia käsitellään tässä luvussa. Kun fastlane alustetaan, Fastfile sisältää oletuksena kaistat koontiin ja Google Play -julkaisuun. Niitä ei tulla hyödyntämään tämän työn yhteydessä, vaan Fastfile luodaan tyhjästä.

Ensimmäisen kaistanajon yhteydessä saattaa esiintyä seuraavan tyyppisiä virheitä:

> No java runtime present, requesting install (Fastlane) /
> Could not determine java version from “9.0.1” (Fastlane) /
> Error:Could not initialize class com.android.sdklib.repositoryv2.AndroidSdkHandle (Android Studio)

Varmista, että Java Developer Kit on asennettu ja se on yhteensopiva Android Studion kanssa. Testisovelluksessa on käytetty JDK 8 (1.8.0_151). Vaikka JDK 9 on saavuttanut yleisen saatavuuden ja julkaistu tuotantokäyttöön 21. marraskuuta 2017 (JDK 9 2017), Android Studio ei vielä tue sitä. Mikäli asennettuna on useampi eri JDK, Android Studio käyttää oletuksena uusinta versiota, joka voi aiheuttaa virhetilanteen. Tähän löytyy apu projektirakenteen asetuksista (File > Project Structure) (StackOverflow 2016a). **JDK location** -kohtaan lisätään hakemisto, josta oikea JDK löytyy.

Asennetun JDK:n löytämiseksi voi käyttää *which java* -komentoa (ks. Kuva 9). Jos which-komento palauttaa: /usr/bin/java, kyseessä on symbolinen linkki todelliseen sijaintiin. Suorita komento *ls -l `which java`*. Seuraavaksi se osoittaa etsittyyn hakemistoon. Mikäli /usr/bin/java osoittaa toiseen symboliseen linkkiin, sen voi hakea uudelleen: *ls -l <mihin usr/bin/java osoittaa>*. (Topolnik 2015.)

```
FL222:~ tpel$ which java
/usr/bin/java
FL222:~ tpel$ ls -l `which java`
lrwxr-xr-x  1 root  wheel  74 Nov 22 01:22 /usr/bin/java -> /System/
Library/Frameworks/JavaVM.framework/Versions/Current/Commands/java
FL222:~ tpel$
```

KUVA 9. JDK-polun hakeminen.

5.4.1 Sovellusten koonti

Koontia varten lisätään uusi kaista (lane). Toiminto, jota halutaan kutsua kaistan sisällä, on tässä tapauksessa ”gradle” (ks. Koodiesimerkki 6). Gradle-actionin koontityypiksi määritellään ”Release” ja tehtäväksi ”assemble”. Variantin nimi määritellään kohdassa ”flavor”.

```
lane :build do
  gradle(
    task: "assemble",
    flavor: "Redpepper",
    build_type: "Release"
  )
end
```

KOODIESIMERKKI 6. Koontikaista.

Gradle käyttää Android Studiossa vastaavia parametreja ja tietoja toimittaessaan koontia. Esimerkiksi käännettäessä applikaatiota Android Studiolla Event log -ikkunassa esiintyy ”Executing tasks: [:app:assembleRedpepperRelease]”. Koodiesimerkki 7 esittää fastlane koonnin määrittelyn yhdellä rivillä yhdistämällä kaiken task-määreen perään. Kaista suoritetaan komennolla *fastlane build*, jonka jälkeen kootut APK-paketit löytyvät standardin mukaisesti **outputs**-kansiosta app/builds -hakemiston alta.

```
lane :build do
  gradle(task: "assembleRedpepperRelease")
end
```

KOODIESIMERKKI 7. Koontikaista, yksinkertaistettu.

5.4.2 Dotenv

Dotenv mahdollistaa ympäristömuuttujien tallentamisen projektikohtaisiin env-tiedostoihin. Env on piilotettu tiedosto, johon kehittäjä lisää käytössä olevia ympäristömuuttujia. (Fastlane n.d.d.) Niitä voidaan luoda konfiguraatiokohtaisesti niin monta kuin tarvitaan. Dotenviä kannattaa hyödyntää, vaikka projektissa ei olisikaan product flavoreita. Ympäristömuuttujien käyttö selkeyttää Fastfilen ulkoasua merkittävästi, mahdollistaa konfiguroinnin sekä tietojen salaamisen. HockeyAppin API token ja muut salassa pidettävät

muuttujat voidaan tallentaa omaan tiedostoon, joka erotetaan julkisesta versionhallinnasta lisäämällä se gitignoreen. Jos Dotenv ei jo löydy Gemfile-tiedostosta, se voidaan lisätä projektiin komennolla *[sudo] gem install dotenv*. Kaikki asennetut gemit on mahdollista nähdä suorittamalla komento *bundle*.

Kun Dotenv on asennettu, voidaan luoda .env-alkuisia tiedostoja. Tiedostot luodaan fastlane-kansioon. Terminaalilla se onnistuu *touch .env* komennolla ja tiedostoa voi muokata tekstieditorilla, esimerkiksi nanolla (*nano .env*). Tähän tosin kelpaa mikä tahansa ohjelma, joka mahdollistaa piilotettujen tiedostojen käsittelyn. Seuraavaksi tiedostoon listataan käytettävät ympäristömuuttujat (ks. Liite 2). Jotta Dotenv osaisi etsiä kyseisiä ympäristömuuttujia eri env-tiedostoista, lisätään *fastlane_require 'dotenv'* -rivi Fastfilen alkuun (ks. Liite 1). Ympäristömuuttujaan viitataan Fastfile:ssa syntaksilla **#{ENV['avainarvo']}** (ks. Koodiesimerkki 8). Kun kaista suoritetaan komentoriviltä, env-viittaus lisätään komennon päätteeksi *--env [konfiguraatio]*.

```
desc "Upload application to HockeyApp"
lane :hockeyapp do
  hockey(
    apk: "#{ENV['PATH_APK_RELEASE']}",
    api_token: "#{ENV['API_TOKEN_HOCKEY']}",
    dsym: "#{ENV['PATH_MAPPING']}"
    notify: "2",
    status: "1",
    notes: File.read("changelog.txt")
  )
end
```

KOODIESIMERKKI 8. Ympäristömuuttujien käyttö Fastfilessa

Fastfilen alussa voidaan määritellä ylikirjoittava env-tiedosto. **before_all** komennon sisään lisätään **Dotenv.overload 'tiedostonimi'**. Overload-tiedosto ylikirjoittaa kaikki muut projektin .env -tiedostot, sillä siitä haetaan muuttujaa ensimmäisenä ja jos se löytyy, sitä käytetään. Seuraavaksi muuttujaa haetaan konfiguraationimellä, jolla --env komento on annettu (.env.app_name). Viimeisenä tarkistetaan löytyykö kyseinen muuttuja .env tiedostosta. (Holtz 2017.) SpiceCabinet-projektissa overloadin sijaan käytetään **Dotenv.load** komentoa, joka lataa tiedoston ympäristömuuttujat, mutta ei ylikirjoita mitään. Ladattu tiedosto sisältää muuttujat, jotka ovat uniikkeja, eikä tarvetta ylikirjoittamiselle ole.

5.4.3 HockeyApp-julkaisu

HockeyApp-julkaisuja varten fastlane tarjoaa actionin nimeltään **hockey**. Parametreistä oleellisimmat ovat **api_token**, **apk**, **dsym** ja **status** (ks. Koodiesimerkki 9). APK-parametriin lisätään APK-paketin hakemistopolku. Dsym sisältää mapping-tiedoston polun. Statuksella määritellään, onko applikaatio HockeyAppin kautta ladattavissa (2) vai ei (1). API token sen sijaan on tunnus, jolla fastlanelle annetaan applikaatio-oikeudet HockeyAppissa. API token luodaan siirtymällä HockeyApp-tilin asetuksiin (Account settings) ja klikkaamalla valikossa kohtaan **API Tokens**. Tokenia luotaessa määritellään ainoastaan, onko tokenilla oikeus yhteen applikaatioon vai kaikkiin ja onko sillä täydet vai rajatut käyttöoikeudet. Luotu API token on käyttäjäkohtainen. Testisovelluksessa luotiin API token kaikille applikaatioille ja täysillä käyttöoikeuksilla.

```
hockey(
      apk: "#{ENV['PATH_APK_RELEASE']}",
      api_token: "#{ENV['API_TOKEN_HOCKEY']}",
      dsym: "#{ENV['PATH_MAPPING']}"
      notify: "2",
      status: "1",
      notes: File.read("changelog.txt")
)
```

KOODIESIMERKKI 9. Action, hockey.

5.4.4 Google Play Store -julkaisu

Supply on action, jota käytetään APK-pakettien lataamiseksi Google Playhin (ks. Koodiesimerkki 10). Koska supply on jo alustettu lataamalla keyfile.json, riittää, että määritellään sovelluksen APK-tiedoston ja metadatan polku sekä paketin nimi. Supply mahdollistaa APK-paketin lataamisen suoraan tuotantoon, mutta asiakasprojektin vaatimusten valossa tehdään vain beta-julkaisuja. Sovelluspäivitykset pystytään valmistelemaan sen avulla paremmin.

```
supply(
      track: "beta",
      package_name: "#{ENV['PACKAGE']}",
      metadata_path: "#{ENV['PATH_METADATA']}",
      apk: "#{ENV['PATH_APK_RELEASE']}",
      mapping: "#{ENV['PATH_MAPPING']}"
)
```

KOODIESIMERKKI 10. Action, supply.

Kuten Google Play -konsoli, myös Fastlane huomautaa, jos julkaisu yritetään tehdä nostamatta versiokoodia: "*Google Api Error: apkUpgradeVersionConflict: APK specifies a version code that has already been used.*" Myös versionumeron korotus on mahdollista automatisoida. Fastlanella on action nimeltään **increment_version_number**, mutta se on tuettuna ainoastaan iOS:ia. Android-alustalle versionumeron korottamiseen fastlanella löytyy jonkin verran valmiita esimerkkejä.

5.4.5 Variantit fastlanessa

Ainoa ohje fastlanelta varianttien käyttöön on dotenvin soveltaminen. Jotta yhdellä komennolla mahdollistettaisiin useiden eri varianttien koonti tai julkaisu samanaikaisesti, pitää hyödyntää hiukan Rubyn tarjoamia ominaisuuksia. Koodiesimerkissä 11 esitetään, kuinka koonti lopulta toteutettiin testisovelluksen osalta. Komento *fastlane build_all release:true* suorittaa release-version koonnin jokaiselle applications-listassa nimetylle variantille. Varianttien tulee olla saman nimisiä kuin dotenv-tiedostot, sillä nimiä käytetään env-tiedoston lataamiseen. Release:true -komentorivivipu lisätään, kun halutaan luoda release-versio. Komentorivivipuja voi olla useita ja ne toimivat parametrien tavoin. Kun kaistaan lisätään |**options**|-liite, on mahdollista käyttää saatuja parametreja (Fastlane n.d.e).

```
@@applications = [
  'redpepper',
  'ginger',
  'vanilla'
]


  desc "Build all release versions"
  lane :build_all do |options|
    isRelease = options[:release] == true
    @@applications.each do |app_name|
      build(env: app_name, release: isRelease)
    end
  end

  desc "Build application"
  lane :build do |options|
    Dotenv.overload ".env.#{options[:env]}"
    scheme = (options[:release] ? "Release" : "Debug")

    gradle(
      task: "assemble",
      flavor: "#{ENV['FLAVOR_NAME']}",
      build_type: scheme
    )
  end
```

KOODIESIMERKKI 11. Koonti sovellusvarianteilla.

Kaistaa voidaan kutsua myös toisen kaistan sisällä, joko parametrien kanssa tai ilman. Parametrit määritellään tällöin suluissa (Fastlane n.d.e). Kaistoja voi kutsua myös shell-komennolla kirjoittamalla *sh(fastlane [kaistan nimi] [parametrit])*. Koodiesimerkki 11 mahdollistaa, että applications-listaa voidaan laajentaa. Uusi sovellusvariantti lisätään listan jatkoksi ja sitä varten luodaan oma env-tiedosto.

Asiakasprojektissa nähtiin hyödylliseksi lisätä uusi variantti osaksi fastlanea jo ennen kuin siitä olisi tarve luoda release-versiota. Toisin sanoen staging-versioita haluttiin päivittää hockeyappiin. Jotta listaa ei tarvitsisi manuaalisesti muokata joka kerta, kun mukana on julkaisematon variantti, lisättiin koodiesimerkki 12 mukainen vertailu ennen varsinaisen toiminnon suorittamista. Env-tiedostosta jätetään tällaisten varianttien kohdalla release- ja metadata-polut tyhjäksi. Jos halutaan ajaa release-koonti ja kyseisen variantin release-polku on tyhjä, koonti ohitetaan.

```
if scheme == "Release" and "#{ENV['PATH_APK_RELEASE']}" == ''
  puts "SET RELEASE APK PATH."
else
  gradle(
    ...
  )
end
```

KOODIESIMRKKI 12. Release-koonnin ohittaminen julkaisemattomille varianteille.

5.5 Yhteenveto

Kehittäjän työn havaittiin helpottuvan fastlanen myötä. Monivaiheinen ja aikaa vievä prosessi saadaan yhden komennon taakse. Se on selkeä apu erityisesti projekteissa, jotka sisältävät variantteja. Karkeasti arvioituna yli tunnin työ voidaan tiivistää 15 minuuttiin. Fastlanen oma dokumentaatio käyttöönottoon ei ota huomioon yleisimpiä virhetilanteita, mutta on kuitenkin selkeä ja suoraviivainen.

Ongelmaksi kuitenkin muodostui se, että fastlane on tehty ensi sijassa yhden sovelluksen julkaisun automatisointiin. Androidin supply-action ei toimi sovellusvarianttien kanssa automaattisesti yhteen. Tähän fastlanen oma dokumentaatio ei ota kantaa kuin dotenvin osalta. Kehittäjän vastuulle jää toteuttaa omaan käyttötapaukseen liittyvä ratkaisu, jolla perustoiminnot, kuten varianttien koonti tai julkaisu, saadaan ajettua yhdellä komennolla.

6 POHDINTA

Fastlanen käytännön osaamista ei löytynyt projektitiimistä ennestään. Sen tiedettiin kuitenkin olevan julkaisun automatisointityökalu, josta löytyy tuki sekä iOS-, että Android-alustalle, se mahdollistaa sovelluskauppajulkaisujen lisäksi HockeyApp-julkaisut ilman erillistä liitännäistä ja sovellusvarianttien koonti sekä julkaisu saadaan suoritettua yhdellä komennolla. Näistä syistä fastlane koettiin testaamisen arvoiseksi työkaluksi, ja testisovelluksen luonti nähtiin kannattavana.

Opinnäytetyön yhtenä tavoitteena oli konfiguroida ja kehittää sovellusvariantteihin perustuva Android sovellus. SpiceCabinet-testisovellus konfiguroitiin jäljittelemään asiakasprojektia ja lopputuloksesta oli merkittävä apu. Kun fastlane alustettiin ensimmäisen kerran, nähtiin, millaisia virhetilanteita se saattoi aiheuttaa ja näihin pystyttiin myöhemmin varautumaan. Tarvittavat asennukset ja ympäristövaatimukset saatettiin valmistella jo testisovelluksen yhteydessä. Testisovelluksen merkittävin tehtävä oli lopulta toimia apuna HockeyApp- ja sovelluskauppajulkaisun automaatiotestauksessa. Uusia kaistoja oli mahdollista kokeilla käytännössä ja päätellä vasta jälkeenpäin, mitä tehtäviä halutaan jättää automaation hoidettavaksi.

Opinnäytetyön tuloksena oli tarpeita vastaava automaatioprosessi sovelluspäivitysten julkaisuun. Automaatioprosessi kehitettiin tukemaan sovellusvariantteja. Fastlanen käyttöönotosta ja varioinnin käsittelystä syntyi ohjeet, joiden avulla toteutettiin fastlanen käyttöönotto onnistuneesti myös asiakasprojektissa. Ohessa luotiin varianttipohjainen testisovellus, jota voidaan jatkossa käyttää uusien toimintojen testaamiseen sitä mukaan, kun asiakasprojektin automaatiota jatkokehitetään.

Asiakasprojektin julkaisuautomaation jatkokehityssuunnitelmat selkeytyvät, kun fastlane on ollut jonkin aikaa käytössä ja uudet tarpeet ymmärretään paremmin. Seuraavassa vaiheessa on tarkoitus selvittää, miten nykyiset kaistat palvelevat työskentelytapojamme ja miten niitä tulisi muokata vastaamaan paremmin projektin olemassa oleviin ja uusiin tarpeisiin. Kun sovellusvarianttien määrä kasvaa, julkaisuautomaation tarpeet kasvavat. Sovellusvarianttien koonti halutaan tällöin siirtää suoritettavaksi erilliselle työasemalle.

Opinnäytetyön aikataulu oli joustava. Uusia sovellusvariantteja ei ollut tiedossa silloin, kun projekti aloitettiin. Työn suoritus ajoittui odotettua laajemmalle aikavälille, joka mahdollisti laajemman ja kattavamman ymmärryksen fastlanesta. Myös sovellusvarianttien käsittelyssä alettiin hyödyntää enemmän Ruby-ohjelmointikieltä vasta projektin lopulla. Fastfilessa oli toistensa kaltaisia kaistoja ja haluttiin välttää toistoa sekä tiedoston kasvamista liian suureksi. Listojen ja for each -lausekkeiden avulla tiedostosta saatiin selkeämpi ja helpommin luettava.

Työn onnistumista voidaan arvioida pohtimalla, millainen tilanne olisi, jos automatisointia ei olisi toteutettu ja variantteja olisi asiakasprojektissa esimerkiksi 20. Uudet sovelluspäivitykset tehtäisiin manuaalisesti jokaiselle 20:lle sovellusvariantille. Sovelluksen koonti olisi Android Studion avulla yhtä yksinkertaista kuin aiemminkin, sillä Android Studiolla on mahdollista suorittaa koonti kaikille valituille sovellusvarianteille samalla kertaa. Ainoastaan koonnin suorittamisen kesto kasvaisi. Julkaisu HockeyAppiin veisi selvästi kauemmin aikaa, mutta myös mahdollisuudet virheisiin olisivat suuremmat. Uuden sovellusversion manuaalinen lisääminen on selitetty luvussa ”4.3 HockeyAppin käyttöönotto” ja jokaisen sovellusvariantin kohdalla käytäisiin kyseinen prosessi läpi. Mapping.txt -tiedosto on kaikilla varianteilla samanniminen, joten sen siirtämisessä tulisi olla huolellinen sekä HockeyAppin, että myös Google Play -konsolin puolella. Google Play -konsolissa pystyy valmistelemaan sovellusjulkaisut etukäteen ja ne voi jälkeenpäin käydä vain julkaisemassa. Sovellusten valmisteluun käytettäisiin huomattavasti kauemmin aikaa, jos julkaisut tehtäisiin manuaalisesti. Automaatiolla beta-julkaisut käytännössä korvaavat valmistelutyön, sillä ne lataavat valmiiksi APK-paketin ja mapping-tiedostot Google Play -konsoliin.

Voidaan siis todeta, että opinnäytetyöprojektin tuloksista oli suuri hyöty asiakasprojektille ei pelkästään tällä hetkellä, vaan myös tulevaisuutta ajatellen. Fastlanen asennus Androidille ja käyttöönotto antoivat suuntaviivoja siihen, kuinka sama toteutetaan iOS:lle. Jatkokehityksen myötä automaatioprosessi voi kehittyä merkittävästi ja sen tuomista oivalluksista on apua varmasti muissakin mobiilialustoihin keskittyvissä asiakasprojekteissa.

LÄHTEET

About Felix Krause. N.d. Luettu 22.12.2017. https://krausefx.com/about

Android Studio User Guide. N.d.a. Configure your build. Luettu 4.3.2018. https://developer.android.com/studio/build/index.html

Android Studio User Guide. N.d.b. Configure build variants. Luettu 4.3.2018. https://developer.android.com/studio/build/build-variants.html

Android Studio User Guide. N.d.c. Sign your app. Luettu 11.2.2018 https://developer.android.com/studio/publish/app-signing.html

Android Studio User Guide. N.d.d. Shrink your code and recources. Luettu 1.4.2018. https://developer.android.com/studio/build/shrink-code.html

ContactForum, Futurice. N.d. Luettu 21.4.2018. http://www.contactforum.fi/en/yritykset/futurice/

Danger, J. 2015. What does 'bundle exec' do? Luettu 18.3.2018. https://jdanger.com/what-does-bundle-exec-do.html

Google Play Console. N.d. Android Vitals. Tietojen selventämistiedostot. Luettu 1.4.2018. Google Play -konsolin sovelluskohtainen näkymä. Pääsy vain Google Play Console -tunnuksilla. Viitattu 11.4.2018.

Fastlane. N.d.a. System requirements. Luettu 19.1.2018. https://docs.fastlane.tools/

Fastlane. N.d.b. Getting started with Fastlane for Android. Luettu 22.12.2017. https://docs.fastlane.tools/getting-started/android/setup/

Fastlane. N.d.c. Fastlane actions. Luettu 18.1.2018. https://docs.fastlane.tools/actions/

Fastlane. N.d.d. Fastlane keys. Luettu 5.1.2018. https://docs.fastlane.tools/best-practices/keys/

Fastlane. N.d.e. Advanced. Luettu 1.4.2018 https://docs.fastlane.tools/advanced/

HockeyApp. N.d. Luettu 4.12.2017. https://hockeyapp.net/

HockeyApp SDK. N.d. Luettu 7.12.2017. https://support.hockeyapp.net/kb/client-integration-android/hockeyapp-for-android-sdk

Holtz, J. 2017. Episode2: Using dotenv and environment variables with fastlane. Vimeo. Katsottu 18.1.2018. https://vimeo.com/228243145

JDK 9. 2017. Luettu 28.1.2018. http://openjdk.java.net/projects/jdk9/#Feature_Extension_Complete

Mikkola, V. 2016. Ohjelmistojen testaus ja hallinta, Gradle. Luettu 2.4.2018. http://docplayer.fi/272801-Ohjelmistojen-testaus-ja-hallinta-gradle.html

Sanasto. N.d. Stack trace. Luettu 1.4.2018 https://plus.cs.hut.fi/o1/2017/yleista/sanasto/

Spice Program. N.d. Luettu 29.1.2018. https://spiceprogram.org/

StackOverflow. 2016.a. Error:Could not initialize class com.android.sdklib.repository.v2.AndroidSdkHandle. Mihir Shah. Luettu 27.12.2017. https://stackoverflow.com/questions/37513651/errorcould-not-initialize-class-com-android-sdklib-repositoryv2-androidsdkhandl

StackOverflow. 2016.b. What does "You should probably `chown' them" mean? Luettu 2.1.2018. https://stackoverflow.com/questions/17223427/what-does-you-should-probably-chown-them-mean/17223847

Topolnik, M. 2015. StackOverflow. What is path of JDK on Mac? Luettu 27.12.2017. https://stackoverflow.com/questions/18144660/what-is-path-of-jdk-on-mac

LIITTEET

Liite 1. SpiceCabinet, Fastfile

1 (4)

```
# Customize this file, documentation can be found here:
# https://docs.fastlane.tools/actions/
# All available actions: https://docs.fastlane.tools/actions
# can also be listed using the `fastlane actions` command
fastlane_require 'dotenv'

default_platform(:android)

@@applications = [
  'redpepper',
  'ginger',
  'vanilla'
]

platform :android do
  before_all do
    Dotenv.load '.env.secret'
  end

  lane :test do
    puts "APPLICATION #{ENV['API_TOKEN_HOCKEY']}"
  end

# ----------------------------------------------------------
# Initializing

# Initialize all metadata whenever needed using: init_metadata_all

# Initialize new application metadata with supply using:
#   init_metadata --env 'app name'

  desc "Initialize metadata for all"
  lane :init_metadata_all do
    @@applications.each do |app_name|
      init_metadata(env: app_name)
    end
  end

  desc "Initialize metadata for a specific app"
  lane :init_metadata do |options|
      Dotenv.overload ".env.#{options[:env]}"

      sh("fastlane supply init --metadata_path ./metadata/#{ENV['APP_NAME']} -
-package_name #{ENV['PACKAGE']}")
  end
```

```
# ------------------------------------------------------------
# Build lanes

# Build all applications using: build_all
# Build a specific application using: buid --env 'app name'
# Define release version with: release:true

  desc "Build all release versions"
  lane :build_all do |options|
    isRelease = options[:release] == true
    @@applications.each do |app_name|
      build(env: app_name, release: isRelease)
    end
  end

  desc "Build application"
  lane :build do |options|
    Dotenv.overload ".env.#{options[:env]}"
    scheme = (options[:release] ? "Release" : "Debug")

    gradle(
      task: "assemble",
      flavor: "#{ENV['FLAVOR_NAME']}",
      build_type: scheme
    )
  end

  desc "Clean outputs"
  lane :clean do
    gradle(task: "clean")
  end

# ------------------------------------------------------------
# HockeyApp lanes

# Upload all applications to HockeyApp using: hockeyapp_all
# Upload specific application to HockeyApp using: hockeyapp --env 'app name'
# Define release version with: release:true

  desc "Upload all applications to HockeyApp"
  lane :hockeyapp_all do |options|
    isRelease = options[:release] == true
    @@applications.each do |app_name|
       hockeyapp(env: app_name, release: isRelease)
    end
  end
```

3 (4)

```
# (notify: yes = 1, no = 2 | status: downloadable = 2, disabled = 1)
  desc "Upload application to HockeyApp"
  lane :hockeyapp do |options|
    Dotenv.overload ".env.#{options[:env]}"

    hockey(
      apk: (options[:release] ? "#{ENV['PATH_APK_RELEASE']}"
                              : "#{ENV['PATH_APK_DEBUG']}"),
      api_token: "#{ENV['API_TOKEN_HOCKEY']}",
      dsym: (options[:release] ? "#{ENV['PATH_MAPPING']}" : nil),
      notify: (options[:release] ? "2" : "1"),
      status: (options[:release] ? "1" : "2"),
      notes: File.read("changelog.txt")
    )
  end

# ---------------------------------------------------
# Google Play lanes

# Upload all release versions to PlayStore (Beta) using: playstore_all
# Upload specific release version to PlayStore (Beta) using:
#   playstore --env 'app name'

  desc "Build all application releases & Deploy to Google Play"
  lane :playstore_all do
    @@applications.each do |app_name|
       playstore(env: app_name)
    end
  end

  desc "Build application release & Deploy Beta to Google Play"
  lane :playstore do |options|
    Dotenv.overload ".env.#{options[:env]}"

    supply(
      track: "beta",
      package_name: "#{ENV['PACKAGE']}",
      metadata_path: "#{ENV['PATH_METADATA']}",
      apk: "#{ENV['PATH_APK_RELEASE']}",
      mapping: "#{ENV['PATH_MAPPING']}"
    )
  end
```

```
  desc "Validate changes with Google Play"
  lane :validate do
    supply(
      track: "beta",
      package_name: "#{ENV['PACKAGE']}",
      metadata_path: "#{ENV['PATH_METADATA']}",
      apk: "#{ENV['PATH_APK_RELEASE']}",
      mapping: "#{ENV['PATH_MAPPING']}",
      skip_upload_screenshots: "true",
      skip_upload_images: "true",
      validate_only: "true"
    )
  end

# ------------------------------------------------------------
# Combined deploy lanes

  desc "Build & Deploy single release to HockeyApp + Google Play"
  lane :deploy_release do
    clean()
    build(env: "#{ENV['APP_NAME']}", release:true)
    hockeyapp(env: "#{ENV['APP_NAME']}", release:true)
    playstore(env: "#{ENV['APP_NAME']}")
  end

# ------------------------------------------------------------

  after_all do |lane|
    # This block is called, only if the executed lane was successful
  end

  error do |lane, exception|
    # This block is called when error occured.
  end
end
```

Liite 2. SpiceCabinet, .env.redpepper

```
APP_NAME = redpepper
PACKAGE = fi.tarup.spicecabinet.redpepper

FLAVOR_NAME = Redpepper

PATH_APK_RELEASE = ./app/build/outputs/apk/redpepper/
release/app-redpepper-release.apk
PATH_APK_DEBUG = ./app/build/outputs/apk/redpepper/debug/
app-redpepper-debug.apk

PATH_MAPPING = ./app/build/outputs/mapping/redpepper/
release/mapping.txt

PATH_METADATA = ./fastlane/metadata/redpepper
```