Primjena praksi i alata DevOps pri razvoju aplikacija za Android

Aleksić, Filip

Master's thesis / Diplomski rad

2019

Degree Grantor / Ustanova koja je dodijelila akademski / stručni stupanj: **University of Zagreb, Faculty of Organization and Informatics / Sveučilište u Zagrebu, Fakultet organizacije i informatike**

Permanent link / Trajna poveznica: https://urn.nsk.hr/urn:nbn:hr:211:361795

SVEUČILIŠTE U ZAGREBU
FAKULTET ORGANIZACIJE I INFORMATIKE
VARAŽDIN

Filip Aleksić

PRIMJENA PRAKSI I ALATA DEVOPS PRI RAZVOJU APLIKACIJA ZA ANDROID

DIPLOMSKI RAD

Varaždin, 2019.

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Filip Aleksić

JMBAG: 0016104188

Studij: Informacijsko i programsko inženjerstvo

PRIMJENA PRAKSI I ALATA DEVOPS PRI RAZVOJU APLIKACIJA ZA ANDROID

DIPLOMSKI RAD

Mentor :

Doc. dr. sc. Zlatko Stapić

Varaždin, srpanj 2019.

Filip Aleksić

Izjava o izvornosti

Izjavljujem da je moj diplomski rad izvorni rezultat mojeg rada te da se u izradi istoga nisam koristio drugim izvorima osim onima koji su u njemu navedeni. Za izradu rada su korištene etički prikladne i prihvatljive metode i tehnike rada.

Autor potvrdio prihvaćanjem odredbi u sustavu FOI-radovi

Sažetak

Rad opisuje što označava pojam DevOps, kakva je njegova primjena u razvoju programskih proizvoda i kako ga primijeniti prilikom razvoja Android aplikacije. Prvo poglavlje opisuje DevOps, od njegovih početaka, usporedbe s prijašnjim načinom rada, do danas. Prikazuje njegove karakteristike i temeljne principe iz kojih su izvedene DevOps prakse, te ih detaljno opisuje. Nakon što je jasno što je pojam DevOps jasan, u sljedećem poglavlju razmatramo njegovu primjenu u svijetu mobilnih aplikacija, specifično Android aplikacija. Opisuje koji se pristupi koriste kako bi implementirali DevOps ideje i koje alate koristimo za njih. Dijelimo razvoj i objavu Android aplikacija na procese i promatramo na koji način možemo poboljšati svaki od njih. Zadnje poglavlje prikazuje opisane metode i tehnike na stvarnom primjeru implementirane pomoću najpopularnijih i najkvalitetnijih dostupnih alata u trenutku pisanja rada. Dokumentiran je čitav proces kako bi ga čitatelj mogao sam rekreirati ili implementirati na nekom svom projektu.

Ključne riječi: Android; DevOps; razvoj; operacije; programsko inženjerstvo; razvoj programskih proizvoda; kontinuirana integracija; fastlane; CircleCI;

Sadržaj

1. Uvod

Razvoj programskih proizvoda je složen proces koji se sastoji od velikog broja aktivnosti i uključuje stručnjake raznih područja (programere, sigurnosne stručnjake, stručnjake IT operacija, itd.). Veliki broj projekata nikad ne dođe do završetka, a veliki dio onih koji i dođu do završetka nemaju implementirane sve funkcionalnosti koje su bile dogovorene u inicijalnoj fazi projekta i često su prekoračili dogovorene rokove. Kako bi se riješio taj problem dolazi do novog pokreta, koji zagovara da ne bi trebali stvarati podjele između odjela (razvojni, operacijski, sigurnosni, itd.) već poticati zajednički rad kako bi brže, bolje i jednostavnije ostvarili svoje zadatke i ciljeve organizacije u cjelokupnosti. Ne radi se naravno samo o brzini i broju implementiranih funkcionalnosti već i o radnom toku koji teče glatko, ne stvaraju se prekidi ili komplikacije dok posao teče kroz organizaciju. Ostali IT odjeli uvijek stvaraju načine kako bi tok mogao ići sa što manje prepreka kroz sustav i kreiraju sustave koji dozvoljavaju razvojnim programerima da njihov posao bude što jednostavniji, ustvari da se jedino moraju brinuti oko razvoja. Sustavi o kojima pričamo su automatizirani servisi i platforme koje omogućuju da manji timovi koriste podršku ostalih timova bez da su ovisni o njima (Kim, Debois, Willis, Humble, & Allspaw, 2016).

Iz gore navedenih razloga dolazi motivacija za ovaj rad. Želimo omogućiti da posao bude bez prepreka kako bi mogli maksimizirati produktivnost i na dnevnoj bazi izbacivati nove verzije proizvoda. Najčešće organizacije nisu ni blizu ovakvog sustava i na dnevnoj bazi se događaju prepreke zbog kojih dolazi do gomilanja poslova i na kraju do toga da proizvod kasni s isporukom. Rezultat toga je da čitava organizacija pati, ne dostiže ciljeve koje želi postići, ne ostvaruje profit koji bi mogla ostvarivati i sve skupa rezultira nezadovoljstvom svih uključenih u proces. Zbog toga je DevOps ključna tehnika koju mora savladati svaka IT organizacija danas (Kim et al., 2016). DevOps se prožima na sve vrste razvojnih projekata pa, tako i na mobilne aplikacije. Tržište mobilnih aplikacija je danas jako veliko i ako želimo stvoriti kompetentne proizvode koji se mogu probiti na tržištu moramo provoditi DevOps na dnevnoj bazi.

Ovaj rad se dijeli na dva velika dijela, prvi je DevOps gdje govorimo općenito o njemu, a drugi je Android i DevOps gdje promatramo kako primjenjujemo objašnjene prakse na Android projektima. Kroz prvi dio ćemo započeti upoznavati DevOps kroz njegovu povijest kako bi shvatili koji su razlozi doveli do njegova stvaranja. Zatim ćemo postepeno uvesti u DevOps i vidjeti kako se razlikuje taj pristup od tradicionalnog pristupa projektima. Kada smo shvatili razlike i vidjeli neke osnovne definicije, tada ćemo uvesti temeljne principe DevOpsa kako bi mogli shvatiti točno što je on i koji su njegovi ciljevi. Za kraj poglavlje ćemo zaokružiti zaključkom kako bi mogli vidjeti cijelu sliku. Drugi dio rada opisuje praktičnu primjenu DevOps praksi prilikom razvoja Android aplikacija. Prvo ćemo se upoznati s Androidom i vidjeti od kojih procesa se sastoji proces razvijanja Android aplikacije. Zatim ćemo svaki od tih procesa zasebno promotriti i navesti neke tehnike pomoću kojih možemo implementirati DevOps prakse u tim procesima. Kada smo to shvatili tada ćemo navesti specifične alate koji nam olakšavaju izvođenje navedenih tehnika te kako ih primijeniti na Android projektu. Poglavlje završava sa zaključkom koji daje osvrt na primjenu DevOps praksi tokom razvoja Android aplikacija. Za kraj slijedi zaključak koji zaključuje čitavi rad i sumira glavne ideje, te daje autorov osvrt i mišljenje na temu.

2. DevOps

Definiranje pojma DevOps nije jednostavna zadatak jer u sebi ne podrazumijeva samo alate ili tehnike, već i način na koji organiziramo timove i projekte. Bass, Weber i Zhu (2015) ga definiraju tako da se fokusiraju na ciljeve koje DevOps želi postići. Kažu da je to set praksi kojima je cilj smanjenje vremena između trenutka kad programer isprogramira neki dio koda do trenutka kad se taj programski kod izvršava u produkciji s osiguranjem visoke kvalitete. Visoka kvaliteta podrazumijeva više različitih segmenata. Prvo kvaliteta u smislu da je nova funkcionalnost uporabljiva onim osobama kojima je namijenjena. Također mora zadovoljavati dostupnost, sigurnost, pouzdanost, itd. Kako bi osigurali ovu razinu kvalitete moramo uključiti kvalitetno testiranje programskog koda prije objave u produkciji. Testiranje možemo osigurati tako da napišemo automatske testove koji moraju biti prolazni kako bi programski kod mogli objaviti, isto tako možemo objaviti novu funkcionalnost smanjenom setu korisnika prije nego ju objavimo svima, još jedna tehnika je detaljno pratiti novi dio koda određeni dio vremena kako bi bili sigurni da sve dobro funkcionira. Koju god metodu odabrali, ili čak ako sve koristimo, jako je važno da visoka kvaliteta mora biti zadovoljena. Osim kvalitete samog programskog koda, također mora postojati visoka kvaliteta sustava za objavu koda. Sustav mora biti pouzdan i mora moći objavljivati kod što je brže moguće. Ako sustav nije ispravan i dolazi do pogrešaka tijekom objave koda tada dolazi do usporavanja čitavog procesa i sve objave kasne s isporukom. Ovakva definicija je dobra jer ne inzistira na nikakvim tehnikama ni metodama već jednostavno kaže sve ono što će smanjiti vrijeme objave kreiranog programskog koda spada pod DevOps. Važno je razumjeti da ne utječu samo akcije koje se događaju u tom vremenskom rasponu na to vrijeme već i puno ranije recimo kod definiranja zahtjeva programskog rješenja i puno kasnije dok je programski kod već u produkciji. Cilj je zapravo osigurati visoku kvalitetu sustava za objavu programskog koda kroz njegov životni ciklus (Bass, Weber, & Zhu, 2015).

Drugačije gledište primjerice imaju Davis i Daniels (2016) koji definiraju DevOps kao kulturni pokret koji mijenja način na koji pojedinac razmišlja o poslu, potiče procese koji pomažu organizaciji da brže dođe do vrijednosti i mjeri utjecaj socijalne i tehničke promjene. Nazivaju ga kulturnim okvirom za dijeljenje priča i razvijanje empatije te tako omogućuje ljudima i timovima da rade svoj posao na efikasan i dugotrajan način. Naglašavaju kako mnogi gledaju na DevOps kao alate, ali to je pogrešno jer važno kako te alate koristimo, a ne alati sami za sebe. Važne su naše vrijednosti, norme i znanja. Možemo upasti u zamku da pretpostavimo da je DevOps nova metodologija za razvoj programskih proizvoda, iako je dosta vezana uz metodologije poput Agile ili XP i uključuje dosta praksi i svojstava kao i one, ona nije to jer uključuje među osobni aspekt i kulturu odnosa ljudi koji rade zajedno u timu (Davis & Daniels, 2016).

Vidimo da su navedene definicije različita gledišta i načine definiranja pojma DevOps, osobno smatram da je vrlo važan kulturni aspekt DevOps-a jer upravo on dovodi do najvećeg napretka unutar kompanije. Zaposlenici koji uvijek razmišljaju kako mogu poboljšati procese i zadatke koje rade na dnevnoj bazi, koji znaju kako raditi u timu i surađivati. Naravno alati i prakse nam uvelike pomažu kako bi došli što brže i jednostavnije do cilja, ali bez zaposlenika koji ih žele provoditi i sami doprinose s idejama, organizacija neće moći puno napredovati.

2.1. Povijest DevOps-a

Prije nego krenemo u detalje DevOps-a, proučit ćemo kako je došlo do ovog pokreta i koji koraci su doveli do konačnog cilja.

U početku razvijanja računala, programeri su bili operatori. Računala poput ENIAC-a (*eng. Electronic Numeric Itegrator and Computer*) su se programirala tako da su se pomicale vakuumske tube i mijenjali kabeli koji su ih pozivali kako bi se mogao izvesti drugačiji izračun s drugačijim podacima. Kako su računala napredovala tako su i njihovi zadatci postajali sve kompleksniji. Jedan od najvećih izazova računala kroz povijest je bilo pisanje programa za svemirsku letjelicu koji je trebao služio kao pomoć u misiji slijetanja na mjesec. Računalni programeri u NASA-i nisu imali prostora za pogreške i jako ozbiljno su pristupili projektu. Kreirali su pojam programsko inženjerstvo, listu zahtjeva i dodali osiguranje kvalitete kao jedan od problema programskog inženjerstva koje je uključivalo otklanjanje neispravnosti svih pojedinačnih komponenti, testiranje komponenti pojedinačno i integracijsko testiranje (Davis & Daniels, 2016).

Isprva programeri nisu smjeli međusobno dijeliti znanja jer su se smatrala tajnama kompanija i bilo je zabranjeno. Kako se internet razvijao tako su i ljudi postajali sve bliži jedni drugima, češće komunicirali i lakše razmjenjivali znanja. Osim interneta 1995. se rodila ideja rješenja otvorenog programskog koda koja je omogućila ljudima čitanje, modificiranje i distribuiranje tog koda. Znanje je postalo puno pristupačnije i kreiranje programa jednostavnije, organizacije su morale početi raditi brzo i fleksibilno kako bi ostale kompetentne. Zatim su se rodile agilne metodike razvoja programskih proizvoda, metode koje pomažu timovima kako bi mogli što brže i kvalitetnije isporučiti programsko rješenje. Marcel Weggermann je 2004. napisao esej koji je uzeo principe agilnih metodika i primijenio ih je na područje administratora sustava. Tijekom 2008. sve više ljudi je počelo razmišljati i samostalno primjenjivati agilne metode na operacije. Neki su podijelili na internetu priče kako su uvelike napredovali koristeći ovaj novi pristup, ali i na konferencijama. Primjerice Flickr, popularna stranica za fotografe, je u 2009. održala govor na temu "10+ Deploys per Day: Dev and Ops Cooperation at Flickr" gdje su naglasili revolucionarnu promjenu koja im je omogućila da vrlo brzo donose promjene na svom programskom proizvodu. Nakon toga je Patrick Debois odlučio kreirati prvu DevOps konferenciju u Ghentu. Konferencija je bila veliki uspjeh i DevOps se nezaustavljivo počeo širiti cijelim svijetom (Davis & Daniels, 2016).

U 2015. je objavljen članak pod nazivom stanje DevOps-a koji kaže da kompanije koje primjenjuju DevOps prestižu svoju konkurenciju i ostvaruju veći uspjeh na tržištu. Osim toga također češće objavljuju kod, imaju manje grešaka, brže se oporavljaju od pogrešaka i imaju sretnije i zadovoljnije zaposlenike. Važno je bilo vidjeti da nije važna samo učestalost objave programskog koda već i povećanje sreće, zadovoljstva i suradnje zaposlenika. DevOps je promijenio industriju na bolji način s fokusom na ljude, potičući kolaboraciju, ali i povećao efikasnost rada (Davis & Daniels, 2016).

Tokom 2015. se počela rađati ideja da se uključi čitavu organizaciju u DevOps procese. Ideju smo nazvali DevOps 2.0 ili BizDevOps (poslovanje (*eng. business*), razvoj (*eng.*

development) i operacije (*eng. operations*)). Cilj je proširiti moć povratnih informacija na sve odjele organizacije kako bi mogli donositi kvalitetnije odluke. Nove funkcionalnosti se objavljuju korisnicima u trenutku kad to odluče marketing i prodaja jer bi to ipak trebala biti poslovna odluka, a ne odluka operacija. Osim toga počinjemo koristiti praksu gdje ne objavljujemo svaku funkcionalnost odmah svim korisnicima već postupno objavljujemo određenom dijelu korisnika. Cilj ove prakse je maknuti stres objave i utjecati isprva samo na minimalan dio korisnika recimo 1%, pa onda 10%, pa 30%, itd. Također mora postojati mogućnost vraćanja na staru verziju u slučaju da postoji problem s novom funkcionalnošću. Također se događa tranzicija prema servisima gdje sve operacije prebacujemo na vanjske servise i pozivamo svu izgradnju, testiranja, itd. prema potrebi. Korištenje vanjskih servisa nam omogućava da ne moramo posjedovati previše opreme niti znanja za održavanje svih potrebnih servera. Sada koristimo vanjski servis koji nam garantira dostupnost i sigurnost. Također plaćamo servise onoliko koliko ih koristimo, recimo ako donesemo odluku da želimo testirati aplikaciju na 10 novih uređaja možemo to lako podesiti i imati ih odmah dostupne (Franco, 2017).

2.2. Uvod u DevOps

Ovo poglavlje će prikazati razliku tradicionalnog i agilnog pristupa razvoja programskih proizvoda. Vidjet ćemo gdje svaki od njih ima propuste i koje probleme rješava DevOps pristup.

Programski proizvod se sastoji od funkcionalnosti, a one funkcionalnosti koje završe u produkciji su dio programskog rješenja. Put do produkcijske okoline nije trivijalan i može biti dugotrajan. U tradicionalnom pristupu razvoja programski proizvod se na početku definira i programira se u fazama. Definirana specifikacija je često nedovoljna, neispravna ili nekonzistentna. Nekonzistentna je jer klijent često dodaje nove zahtjeve koji nisu bili definirani na početku te nastaje problem kako uskladiti specifikaciju s novim zahtjevima i osigurati da svi koriste posljednju verziju specifikacije. Veliki je problem također to što se prakse za osiguranje kvalitete koriste tek kad je programsko rješenje kreirano. Sve pronađene greške se dodaju u sustav za praćenje zadataka koji je najčešće zanemaren zbog visoke cijene ispravka. Programeri imaju negativan odnos prema testerima i smatraju ih krivim za pogreške u radu jer su oni ti koji ukazuju samo na njihove greške. Projekti su tradicionalno najčešće definirani s početnom i krajnjom točkom, članovima projekta, budžetom i funkcionalnostima koje moraju biti dostavljene. Voditelji projekta kreiraju projektne uloge koje su definirane procesima u projektu. Dio projekata je uspješan po definiranim standardima, ali se stvaraju mnogi drugi problemi. Problemi su sljedeći:

- Projekti postaju ovisni o pojedincima koji se smatraju eksperti jer samo oni znaju riješiti problem, a zapravo je rezultat loše dokumentiranog i napisanog koda koji nitko drugi ne može razumjeti
- Svaki zaposlenik ima titulu koja ne pomaže shvatiti uloge ili vještine, već je samo rezultat duljine rada zaposlenika unutar tvrtke
- Zaposlenici se ne drže svojih definiranih uloga već rade druge aktivnosti i izbjegavaju raditi vlastita zaduženja, kao rezultat imamo razliku između izvođenja definiranog i stvarnog

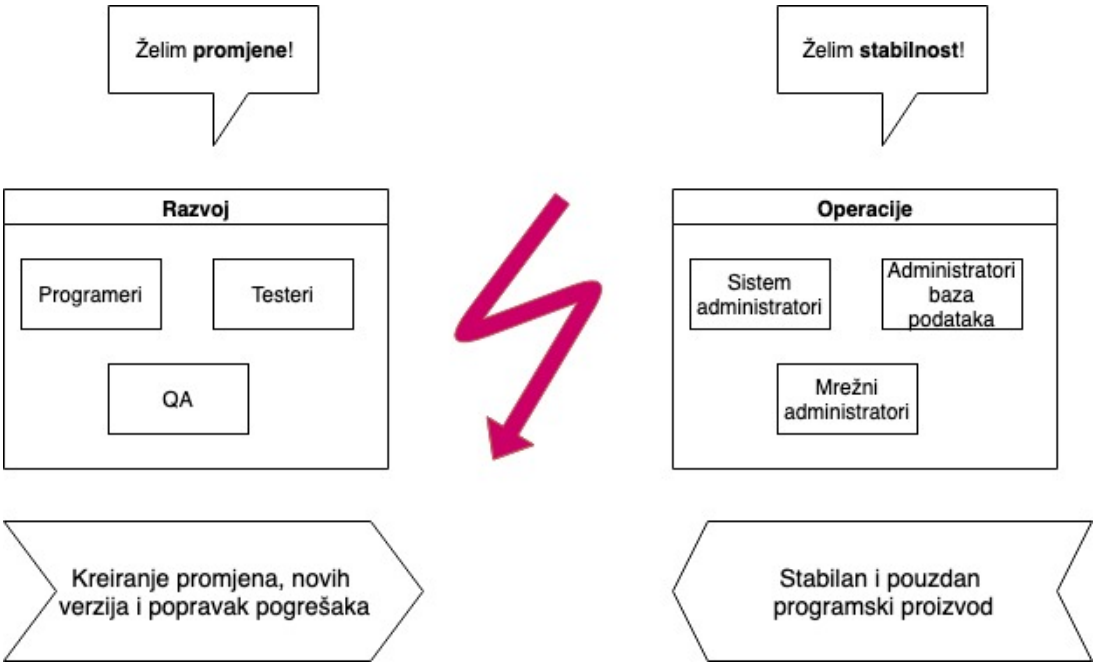
procesa

- Planiranje je rijetka aktivnost, jednom kad je definiran početni plan on postaje cilj projekta, umjesto da je cilj programsko rješenje koje donosi vrijednosti korisniku u što efikasnijem vremenu

Tradicionalno okruženje pod razvoj uključuje samo programere. Testeri i osiguranje kvalitete su odvojene uloge čije aktivnosti dolaze nakon što su programeri završili svoj dio posla. Operacije je posebno odjeljenje koje se sastoji od raznih administratora (baza podataka, sistemskih, mrežnih, itd.) i njihovo uključenje na projekt je tek posljednji korak u procesu završetka objave projekta. Često nisu uključeni u proces kreiranja i testiranja već primaju konačni rezultat programera. Ovakva organizacija timova je loša jer se stvaraju odvojeni timovi koji imaju vlastite interese i ciljeve umjesto da postoji jedan zajednički cilj prema kojem se krećemo. Svaki tim koristi vlastiti "pojmovnik", odnosno nema zajedničkog jezika već timovi koriste vlastite nazive koji su nejasni ostatku projektnog tima. Rezultat je loša komunikacija i ne mogućnost sudjelovanja čitavog tima u raspravama jer je žargon pre kompliciran da bi ga shvatili. Još jedna posljedica je strah od svega što rade drugi timovi jer bi sve moglo potencijalno utjecati na naš rad, te strah od gubitka moći i reputacije ako svi počnu dijeliti znanje i raditi zajedno. Upravo iz ovih razloga su se ljudi počeli odmicati od vodopadnog modela razvoja i okretati prema Agilnom (Hüttermann, 2012).

Agile promovira ideju zajedničkog tima, potiče komunikaciju i kolaboraciju među njenim članovima tako da tim dijeli jedan zajednički cilj. Programeri i testeri počinju blisko surađivati i definiraju čitavo osiguranje kvalitete. Svatko u timu provodi korake za osiguranje kvalitete i preuzima odgovornost da sve što kreira bude visoke kvalitete. U agilnom razvoju projektne uloge i odgovornosti se mijenjaju. Uloge nisu strogo definirane i jedna osoba može imati različite uloge na različitim projektima. Programeri rade druge zadatke osim pisanja programskog koda, a testeri rade više od samo testiranja. Kao rezultat dobivamo povišenu razinu kvalitete jer su svi odgovorni za nju, programer više ne pokušava sam razumjeti zahtjeve koje mora kreirati već mu pomaže čitav tim i projektne uloge više nisu samo titule i ne ograničavaju članove tima u njihovom radu. Iako je razvoj postao jedan tim, operacije su i dalje odvojeni tim s drugačijim ciljevima, smatra ga se posebnim odjeljenjem. Perspektiva i odnos prema njima je ostao isti, operacije primaju rezultat razvoja i očekuje se da ga naprave dostupnim korisnicima na produkcijskim uređajima s ne funkcionalnim zahtjevima koji možda uopće nisu ostvarivi s programom koji je dostavljen. Također odgovornost za stabilnost je samo na operacijama i svaki nedostatak poput sigurnosti, brzine odgovora, dostupnosti, itd. se smatra njihovom krivicom. S druge strane od razvoja se očekuje kreiranje proizvoda koji ispunjava korisničke zahtjeve, prolazi testove i da rade to u što efikasnijem vremenu. Znači razvoj želi što češće izbacivati nove verzije proizvoda kako bi klijent bio što zadovoljniji, dok operacije pokušavaju zaštiti svoje ciljeve poput sigurnosti i dostupnosti (Hüttermann, 2012).

Slika 1 vizualno prikazuje konflikt između operacija i razvoja. Konflikti često izbijaju zbog vremenskog pritiska na oba tima, razvoj ima pritisak da što brže kreira nove verzije sustava, dok operacije u slučaju pada moraju dići sustav na noge što je prije moguće. Jedna od čestih situacija u kojim dolazi do konflikta je prilikom objave novog koda. Razvoj kreira novu verziju



Slika 1: Raskorak između razvoja i operacija koji dovodi do konflikta (Prema: Hüttermann, 2012)

koja prolazi sve testove i funkcionira u testnom okruženju, predaje rješenje operacijama koji ne mogu pokrenuti novu verziju na produkcijskim serverima. Operacije traže pomoć od razvoja, a razvoj smatra da to nije njihov problem jer su oni zadovoljili testne slučajeve i jer radi u testnom okruženju, smatraju da operacije moraju same riješiti taj problem. Nakon nesuglasica nadređeni odredi da moraju zajedno raditi na rješenju problema. Rješenje najčešće bude sitna razlika između testne i produkcijske okoline za koju nisu znali ni operacije ni razvoj, ali oboje smatraju da je kriva druga strana za problem. Također su česti konflikti nakon razvoja kad nova funkcionalnost prestane raditi ili kad je jako usporena (Hüttermann, 2012).

Zbog ovih razloga Agile ne uspijeva postići veliko ubrzanje jer operacije postaju usko grlo koje koči ubrzanje koje je kreirala suradnja poslovnog i razvojnog dijela. Razvoj želi brze i česte promjene, a operacije dopuštaju promjene samo na određene datume u određeno vrijeme. Ako se verzija pokaže ne ispravna u trenutku objave sve se vraća na staru verziju i šalje se nazad razvoju na doradu. Razvoj je jako ne zadovoljan jer nemaju vremena pokušati odmah popraviti promjene već moraju čekati sljedeći vremenski trenutak koji odrede operacije za objavu programskog proizvoda. Operacije su nezadovoljne jer nova verzija nije spremna za produkciju i krive razvoj. Zapravo korijen problema je što nemaju zajednički cilj i loše surađuju (Hüttermann, 2012).

Kada su ljudi shvatili probleme koje smo naveli, pokušali su implementirati Agile na operacijama. Cilj ovih pokušaja je bio maknuti raskorak između razvoja i operacija, te stvoriti blisku suradnju. Nakon određenog vremena ovaj pokret se formirao u ono što danas zovemo DevOps. DevOps pristup pod razvojnu ulogu sadržava programere, testere, osiguravatelje kvalitete i stručnjake iz operacija. Svi zajedno rade kako bi razvili programski proizvod i dostavili ga korisniku. Kolaboraciju uključuje tako što kao preduvjete traži uzajamno poštivanje članova tima, predanost zajedničkom cilju, zajedničko vlasništvo programskog proizvoda i dijeljene vri-

jednosti. Ovakav pristup dovodi do timova koji međusobno dijele znanja i svi pridodaju onim u čemu su najbolji. U slučaju problema poput pada sustava, svi sudjeluju u rješavanju, a ne samo operacije. Način na koji izvodimo proces se također mijenja. Operacije su uključene u Agilne procese, operacije i razvoj sudjeluju u čitavom procesu dostavljanja programskog proizvoda i obje grupe su usredotočene na dostavljanje promjena korisniku u brzom vremenu s visokom kvalitetom. Počinje se uključivati puno alata kako bi se moglo automatizirati što više aktivnosti vezane uz objavu programskog koda. Također se svi alati čuvaju u sustavu za verzioniranje koda. Koristimo alate za izgradnju, testiranje, objavu i konfiguriranje aplikacija. DevOps je omogućio ubrzati proces razvoja i objave programskih proizvoda i uz to omogućiti sretnije zaposlenike zbog suradnje i uzajamnog poštenja među zaposlenicima. Sad kad smo se uveli u DevOps i shvatili na koji način je došlo do njega te koje probleme rješava nastavit ćemo s detaljnim opisom karakteristika.

2.3. Temeljni principi DevOps-a: 3 puta

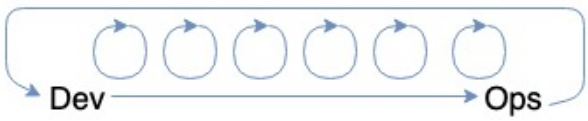
Kim et al. (2016) definiraju 3 puta iz kojih se mogu izvesti svi DevOps obrasci. Prvi put omogućava brz tok podataka od razvoja do operacija pa do klijenta. Rezultat korištenja praksi prvog puta je da se nikad neće dogoditi da se prepoznali defekt šalje dalje kroz tok, nikad se neće dopustiti da se nešto lokalno optimizira ako to uzrokuje globalno usporenje toka, uvijek se traže novi načini kako bi se povećao tok i kako bolje razumjeti sustav. Grafički prikaz prvog puta možemo vidjeti ispod na slici 2 (Kim, 2012).

Slika 2: Prvi put, sistemsko razmišljanje (Prema: Kim, 2012)

Drugi put se fokusira na kreiranje petlje povratnih informacija. Cilj svake inicijative u ovom putu je skraćivanje i pojačavanje povratne petlje kako bi se ispravke mogle stalno raditi. Rezultat drugog puta je razumijevanje i odgovaranje svih klijenata, skraćivanje i poboljšanje povratnih petlji i ugrađivanje znanja tamo gdje je to potrebno. Grafički prikaz drugog puta možemo vidjeti ispod na slici 3 (Kim, 2012).

Slika 3: Drugi put, poboljšanje petlje povratnih informacija (Prema: Kim, 2012)

Treći put se odnosi na kreiranje kulture koja potiče kontinuirano eksperimentiranje odnosno uzimanje rizika te učenja iz njihovih promašaja i razumijevanje da je vježba i ponavljanje preduvjet kako bi postali vješti. Rezultat korištenja navedenih praksi je izdvajanje vremena za poboljšanje svakodnevnog posla, kreiranje rituala koji nagrađuju timove koji uzimaju rizike i ukupno povećanje znanja zaposlenika kompanije. Grafički prikaz trećeg puta možemo vidjeti ispod na slici 4 (Kim, 2012).



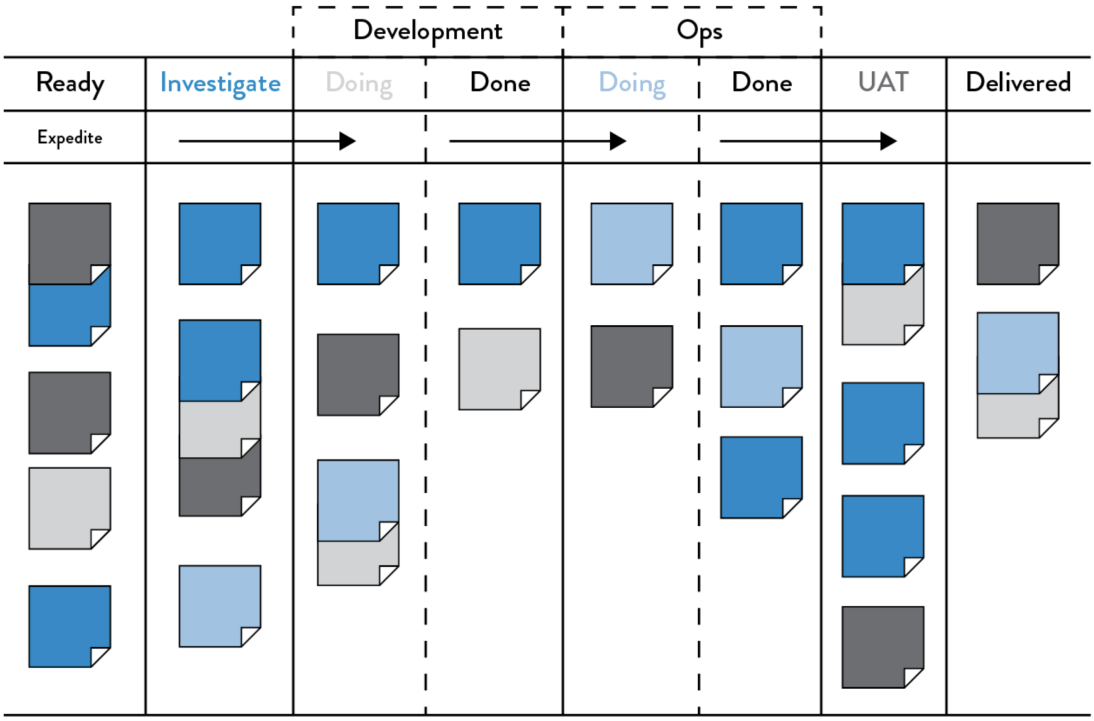
Slika 4: Treći put, kultura kontinuiranog eksperimentiranja i učenja (Prema: Kim, 2012)

2.3.1. Prvi put: tok od razvoja do operacija

Prvi put se fokusira na tok između razvoja i operacija, cilj mu je osigurati brz i jednostavan protok posla kako bi mogli isporučivati klijentu vrijednosti brže. Više nisu u fokusu lokalni ciljevi (razvoj želi što više zahtjeva isporučiti, dok operacije žele stabilnost i sigurnost) već se fokusiramo na zajednički globalni cilj (brza isporuka vrijednosti klijentu). Protok povećavamo tako što napravimo posao vidljivim, smanjimo veličinu posla koji pokušavamo napraviti i izgrađujemo kvalitetu kroz svaki korak i tako osiguravamo da nikakav defekt ne ide dalje kroz sustav. Posljedice ovog puta je smanjivanje vremena potrebno za izvršavanje klijentovih zahtjeva i povećanje kvalitete posla uz veću agilnost i mogućnost eksperimentiranja. Ideje kako ovo napraviti su velikim dijelom preuzete iz Lean principa primijenjenim u proizvodnom vrijednosnom toku (Kim et al., 2016).

2.3.1.1. Napravi posao vidljivim

Za razliku od proizvodnje u tehnologiji je posao nevidljiv. Kod fizičkih procesa se lako može vidjeti gdje se posao nagomilava i prijenos posla između poslovnih jedinica. Dok u tehnologiji prijenos posla se može obaviti u nekoliko klikova mišem i uopće nisu vidljive posljedice te akcije, također se često događa da se zadatak prebacuje između poslovnih jedinica jer je nevidljiv i lako ga je poslati nekom drugom. Kao rezultat kasnimo s dostavljanjem vrijednosti klijentu i usporavamo protok. Kako bi mogli vidjeti teče li posao kroz sustav ili stoji moramo ga napraviti vidljivim, jedan od najboljih načina za prikazivanje su vizualne ploče poput kanban ili sprint ploče. Vizualne ploče mogu biti stvarne ploče ili umjetne kao program na računalu.



Slika 5: Primjer kanban ploče (Izvor: Kim, Debois, Willis, Humble, and Allspaw, 2016)

Primjer kanban DevOps ploče možemo vidjeti na slici 5. Posao je vidljiv u obliku kartica i počinje s lijeve strane. Svaki stupac predstavlja radnu jedinicu i posao se pomiče s lijeva na desno. Kad kartica dođe do krajnjeg desnog stupca tada se taj zadatak smatra završenim ili u produkciji. Osim što posao postaje vidljiv sada nam postaje lakše upravljati poslom kako bi što prije došli s lijeve na desnu stranu. Također možemo lakše mjeriti vrijeme isporuke tako da mjerimo vrijeme od kad smo stavili karticu na ploču, do trena kad se stavlja u najdesniji stupac. Lakše je prioritizirati posao i odabrati koji zadatke prvo napraviti kad vizualno prikažemo sve moguće zadatke (Kim et al., 2016).

2.3.1.2. Ograniči posao u procesu

U proizvodnji postoji proizvodni plan koji jasno definira koji posao se događa u kojem trenutku i kada mora biti završen. Rijetko se događa da se proizvodni plan mijenja. U tehnološkom svijetu nažalost nije ista situacija. Okolina je puno dinamičnija i timovi često moraju zadovoljiti više različitih nadređenih. Rezultat ovakve organizacije je da se određuje posao koji je najprioritetniji, a najprioritetniji je onaj posao koji je hitan tog dana ili onaj koji je najhitniji prema nadređenima. Posao stiže iz svih mogućih komunikacijskih izvora, telefona, elektroničke pošte, sustava za prijavljivanje pogrešaka, itd. Prekidanje posla da bi se započeo novi je jako skupo i to je lako vidljivo u proizvodnji jer je potrebno maknuti sav ne dovršeni rad i napraviti pripremu za početak novog. Upravo zbog toga se promjena posla događa jako rijetko i pokušava se ako je moguće izbjeći. U tehnološkom svijetu zaposlenike je vrlo lakše prekinuti jer su posljedice nevidljive svima ostalima osim samom zaposleniku. Inženjer koji radi na više projekata mora mijenjati zadatke i svaki put iznova shvatiti kontekst zadatka i njegove ciljeve. Istraživanja su pokazala da čak i tijekom izvođenja najjednostavnijih zadataka, vrijeme potrebno da ih se izvrši je drastično duže kad ih se pokušava izvršiti paralelno. Ograničiti posao možemo tako da definiramo maksimalan broj kartica u svakom od kanban stupaca (osim prvog i posljednjeg). Tako kad dostignemo maksimalni broj kartica i želimo dodati novu moramo razmisliti koji posao ćemo izbaciti nazad u lijevi stupac ili ćemo možda shvatiti da zadatak mora pričekati dok ne riješimo jednu od kartica. Na ovaj način također možemo vidjeti koji poslovi zapinju u sustavu i kasne s isporukom. Na ovaj način umjesto da se bacimo na novi zadatak dok čekamo možemo vidjeti zašto trenutni zadatak kasni i kako možemo pomoći u izvršavanju (Kim et al., 2016).

2.3.1.3. Smanji količinu posla u ciklusu

Prije lean revolucije u proizvodnji, često su se uzimale velike količine posla u ciklusu, posebno ako je bilo teško obaviti mijenjanje zadataka. Zbog toga su umjesto da rade jedan po jedan proizvod od početka do kraja uzeli primjerice 10 istih proizvoda i svakom napravili prvi korak, pa svakom drugi, itd. Iako se čini kao dobra ideja ima svoje negativne strane. Prvo kosi se s našim prošlim savjetom gdje želimo imati manje posla u procesu, da bi ovo bilo isplativo morali bi uzeti veliki broj poslova. Također je problem što nisu svi poslovi jednaki pa kod svakih postoji razlika te ne ide jednostavno dalje. Ako se nađe nekakva pogreška u jednom od svih poslova tada se mora pretpostaviti da su svi pogrešni i odbaciti ih. Rezultat ovoga je dugo vrijeme is-

poruke i loša kvaliteta. Lead proizvodnja kaže da moramo smanjiti količinu posla u ciklusu ako želimo postići visoku kvalitetu i brže vrijeme isporuke. U svijetu tehnologije jedan proizvod ili posao možemo gledati kao jedan komad koda koji programer stavi na sustav za verzioniranje. Znamo iz poglavlja koji opisuje povijest koliki je problem ići vodopadnom metodom i tek na kraju pokušati isporučiti čitav kod. Mnogo bolje je koristiti tehniku kontinuirane objave koda gdje je svaka objava koda na sustav za verzioniranje integrirana, testirana i postavljena u produkciju. Tako odmah uočavamo problem i mnogo brže isporučujemo vrijednost klijentu što i je naš cilj (Kim et al., 2016).

2.3.1.4. Smanji broj prosljeđivanja zadatka kroz timove

Nakon što je neki zadatak isprogramiran i proslijeđen operacijama često se događa da mu treba još jako puno vremena da dođe u produkcijsko okruženje. Vrijeme može biti u mjesecima u nekim velikim firmama jer objava koda nije trivijalan zadatak. Primjerice potrebno je napraviti funkcionalno testiranje, integracijsko testiranje, kreiranje okoline, administracija servera, rezerviranje prostora, postavljanje mreže i namještanje sigurnosnih pravila. Kako bi ispunili navedene zadatke potrebno je uključiti puno različitih timova (testere, administratore servera, administratore mreže, sigurnosni tim). Svaki put kad se zadatak prosljeđuje između timova potrebna je komunikacija i planiranje. Primjerice potrebno je kreirati zahtjev kojeg netko treba pročitati i odobriti, nakon toga potrebno je pronaći osobu koja će napraviti zadatak i u kojem trenutku. Svaki od ovih koraka može uzrokovati čekanje jer se neki drugi zadatak možda natječe za taj isti resurs ili nema slobodnih ljudi da ispuni zadatak. Zbog toga vrijeme isporuke stalno raste i nezavršeni zadatci se gomilaju. Problem rješavamo tako da automatiziramo sve korake koje možemo tako da nismo ovisni o drugim ljudima. One zadatke koje nije moguće automatizirati riješimo tako da reorganiziramo timove tako da svaki tim ima članove sa svim potrebnim vještinama da riješi dani zadatak. Tako smo minimizirali broj potrebnih prosljeđivanja zadataka (Kim et al., 2016).

2.3.1.5. Kontinuirano identificirati ograničenja

Kako bismo ubrzali naš sustav i smanjili vrijeme potrebno za isporuku važno je ubrzati ne bilo koji dio sustava već onaj koji je ograničavajuć. Ako ubrzamo neki proces prije ograničenja tada ćemo još brže slati zadatke na usko grlo i stvarati još veći red zadataka. Ako ipak ubrzamo proces nakon ograničenja tada će proces brzo ostati bez posla jer ograničavajući dio ne šalje nove zadatke kroz tok dovoljno brzo. Zbog toga je važno kontinuirano ubrzavati ograničenja sustava (Kim et al., 2016). Goldratt (2005) definira pet koraka:

- Identificiraj ograničenja sustava
- Odluči kako riješiti ograničenje
- Podčini sve ostalo kako bi napravili odluku iz prošlog koraka
- Ponovno procijeni ograničenje

- Ako si riješio problem počni ponovno ispočetka, ako ne ponovno ga pokušaj riješiti

U DevOps transformaciji često identificiramo ista ograničenja:

- Kreiranje okoline: koči nas jer često trebamo slati zahtjeve kako bi kreirali novu okolinu i moramo čekati tjednima ili mjesecima da bi netko ispuni naš zahtjev. Rješenje ovog problema je implementacija servisa koji na zahtjev automatski kreira okolinu i uvijek je dostupan
- Isporuka programskog koda: ako nam isporuka koda traje u tjednima ili mjesecima jer zahtjeva tisuće koraka koje moraju ispuniti različiti timovi tada nam je to najveće usko grlo, kao i kod kreiranja okoline rješenje je automatiziranje svih zadataka i kreiranje servisa koji programer može pozivati bez ičije pomoći
- Postavljanje testova i njihovo izvršavanje: postavljanje testova (kreiranje okoline i testnih podataka) traje nekoliko tjedana i još nekoliko tjedana izvršavanje u većini tvrtki koje to rade bez automatizacije, zato je važno automatizirati postavljanje i omogućiti paralelno izvršavanje kako bi testovi mogli biti što brže izvedeni
- Preuska arhitektura: često tvrtke ne dopuštaju programerima da objave programski kod prije nego što to ne odobri netko od nadređenih, ovo je očiti problem jer se vrlo često mora dugo čekati prije nego nadređeni stigne pregledati i odobriti novi programski kod, problem riješimo tako da više vjerujemo programerima i damo im više slobode tako da sami mogu donositi odluke

Ako riješimo ove navedene probleme tada ćemo drastično smanjiti vrijeme isporuke i ograničenje više neće biti operacije već razvoj ili klijent. Potrebno je imati na umu da ova lista ne predstavlja sva ograničenja tvrtke, niti tvrtka mora imati sva ograničenja, lista prikazuje samo najčešće pronađena ograničenja.

2.3.1.6. Eliminiraj poteškoće i višak u vrijednosnom toku

Lean proizvodnja zagovara ideju eliminiranja svih viškova u proizvodnji (inventure, proizvođenje previše proizvoda, itd.). Moderna interpretacija Leana kaže da eliminiranje viškova može uzrokovati dehumaniziranje i napraviti da zaposlenici gube smisao, sad ga definiramo cilj kao eliminiranje poteškoća u našem svakodnevnom poslu kroz kontinuirano učenje kako bi postigli organizacijske ciljeve. Mi ćemo se fokusirati na modernu interpretaciju jer ona ima više smisla u kontekstu DevOpsa (Kim et al., 2016). Otpad u IT svijetu Poppendieck i Poppendieck (2007) definiraju kao bilo što što uzrokuje kašnjenje ispostave proizvoda klijentu. Također definiraju sljedeće kategorije poteškoća i viškova:

- Djelomično odrađen posao: Pod tim podrazumijevamo bilo koji posao koji nije dovršen i čeka u redu. Takav posao gubi na vrijednosti što duže stoji u redu čekanja.
- Višak procesa: Svaki dodatni zadatak u procesu koji ne donosi nikakvu dodanu vrijednost kupcu. To može biti primjerice dokumentacija koja se nikad ne koristi ili čekanje odobrenja koja uopće nisu potrebna. Svaki višak proces povećava vrijeme isporuke.

- Višak funkcionalnosti: Sve funkcionalnosti koje su dodane u programski proizvod, a ne donose vrijednost klijentu ni organizaciji. One stvaraju problem jer se gubi vrijeme na njihovo programiranje i testiranje.
- Mijenjanje zadataka: Zaposlenici koji su pridodijeljeni na više projekata trebaju svaki put prilikom promjene zadataka uložiti dodatan trud kako bi mogli promijeniti kontekst i shvatiti zadatak koji bi trebali napraviti.
- Čekanje: Ova kategorija je vrlo jednostavna, svako čekanje zadatka na neki resurs je gubitak jer produljuje vrijeme isporuke.
- Kretanje posla: Svako pomicanje posla od jednog radnog centra do drugog zahtjeva dodatni trud i suočavamo se s problemima u komunikaciji jer pošiljatelj treba objasniti zadatak primatelju
- Defekti: U ovu kategoriju spadaju sve neispravne informacije, zadatci ili proizvodi. Zahtijevaju napor da ih se prepozna i riješi. Što je defekt kasnije prepoznat to je teže riješiti ga.
- Nestandardni ili ručni posao: Oslanjanje na bilo kakav nestandardni ili ručni posao drugih je opasno. U tu kategoriju primjerice spadaju postavljanje testnog okruženja ili konfiguracije. Idealno bi svi ti zadatci trebali biti standardizirani i automatizirani.
- Junaštvo: Često se zna dogoditi da pojedinci u kompaniji moraju raditi ekstremno dugo ili ne razumne zadatke kako bi postigli organizacijske ciljeve. Jako je važno prepoznati ovakva junaštva i učiniti sve da se eliminiraju.

Eliminiranjem poteškoća iz svih kategorija ćemo povećati kvalitetu rada i smanjiti vrijeme isporuke. Nakon eliminiranja potrebno je kontinuirano raditi evaluaciju kako bi prepoznali nove poteškoće i viškove u vrijednosnom toku.

2.3.2. Drugi put: načelo povratnih informacija

Za razliku od prvog puta koji opisuje načela koja pomažu kreirati brz protok posla s lijeva na desno, drugi put opisuje načela koja pomažu ostvariti brz i stalan tok povratnih informacija s desno na lijevo u vrijednosnom toku. Cilj mu je osigurati da je sustav siguran i otporan. U svim kompleksnim sustavima je jako važno opaziti i ispraviti pogreške kako se ne bi ozlijedio zaposlenik, oštetili opremu ili napravili nekvalitetan proizvod. Tehnološki svijet spada u kompleksne sustave s visokim rizikom od kritičnih posljedica. Vrlo često se pogreške prepoznaju tek onda kad je prekasno i uzrokovale su velike probleme poput pada produkcijskog sustava ili sigurnosne pogreške koja je omogućila napadačima da ukradu podatke naših korisnika. Kvalitetni povratni tok informacija je jako važan jer tako činimo sustav sigurnijim. Prepoznajemo probleme dok su manji, jednostavniji za popraviti i nisi imali veliki utjecaj na sustav. Probleme ne smatramo kao uzrok za kažnjavanje zaposlenika već prilikom za učenje kako bi prepoznali uzroke problema i na koji način ih kvalitetno riješiti. U daljnjem tekstu ćemo promotriti kakvi su to kompleksni sustavi i kako ih napraviti sigurnijima (Kim et al., 2016).

2.3.2.1. Sigurnost u kompleksnim sustavima

Jedno od svojstava kompleksnih sustava je to da niti jedna osoba ne može sama vidjeti sustav kao cjelinu niti razumjeti kako sve funkcionira zajedno. U takvim sustavima pogreške su neizbježne, te je zbog toga važno napraviti sigurnosni sustav kako bi mogli raditi bez straha jer smo sigurni da će se sve pogreške prepoznati brzo, puno prije nego što budu imale neki veliki negativni utjecaj (Kim et al., 2016).

Spear u (2011) kaže kako je dizajniranje savršeno sigurnih sustava van naših mogućnosti, ali sustavi mogu biti sigurniji ako pratimo sljedeća 4 pravila:

- Kompleksnim radom se upravlja na način da se problemi otkrivaju u dizajnu i operacijama
- Problemi se prikupljaju i rješavaju tako da stvaraju novo znanje
- Novo lokalno znanje se koristi globalno kroz čitavu organizaciju
- Voditelji stvaraju voditelje koji kontinuirano razvijaju navedene osobnosti

Sve ove stavke su potrebne kako bi se posao mogao sigurno izvoditi u kompleksnim sustavima. Prve dvije točke ćemo opisati sada u daljnjem tekstu, a druge dvije kasnije kad budemo govorili o trećem putu.

2.3.2.2. Rano prepoznavanje pogrešaka

Kao što smo spomenuli u prvom pravilu prošlog poglavlja važno je kontinuirano tražiti pogreške u dizajnu i operacijama. Kako bi mogli prepoznati pogreške trebamo povećati informacijski tok u sustavu na što više područja. Želimo brz, jednostavan i jeftin tok jasnih informacija o izvoru i posljedicama procesa. Potrebno je imati što je manje moguće pretpostavki kako bi mogli sa sigurnošću znati što se događa. Sve ovo ostvarujemo pomoću petlje povratnih informacija kroz cijeli naš sustav. U proizvodnji nedostatak korisnih povratnih informacija uzrokuje velike probleme. U jednom slučaju tvornice General Motors Fremont imala problem što nije postojala procedura za prepoznavanje problema u procesu sastavljanja proizvoda, niti je postojala procedura koja opisuje kako se ponašati u slučaju problema tijekom sastavljanja. Posljedica toga je bila da su znali staviti motor naopako u auto, napraviti auto bez volana ili dovršiti auto, a da se on ne može pokrenuti. Tvornice koje imaju brz i čest tok informacija brzo prepoznaju defekte i djeluju na njih (Kim et al., 2016).

Najveći problem povratnih informacija u tehnološkom svijetu je ta što često nemamo brzu povratnu informaciju. Ako projekt organiziramo vodopadnom metodikom možemo čekati i godinu dana da bi dobili bilo kakvu povratnu informaciju o onom što smo razvili. Razlog tako dugom čekanju je taj što je testiranje jedna od posljednjih faza razvoja. Problem rješavamo tako što kreiramo brze povratne petlje na svim fazama vrijednosnog toka. Kreiranje povratne petlje uključuje kreiranje automatizirane izgradnje, integracije i testnih procesa kako bi odmah mogli prepoznati pogrešku koja nije funkcionalna ili spremna za produkcijsku fazu. Također je potrebno kreirati preventivni sustav na kojem se mogu vidjeti statusi svih komponenti kako bi mogli vidjeti ako se neka ne izvršava kako treba (Kim et al., 2016).

Osim pogrešaka sustav nam može pomoći da lakše možemo provjeriti postižemo li ciljeve koje smo si zadali i kako promjene utječu na sustav. Povratne petlje nam ne služe samo za prepoznavanje pogrešaka već nam omogućuju da učimo iz njih jer možemo saznati koji su uzroci i tako povećavamo kvalitetu i sigurnost našeg posla, te povećava naše znanje (Kim et al., 2016).

2.3.2.3. Rješavanje problema

Prepoznavanje problema nije dovoljno, potrebno je riješiti ga što je prije moguće i dovesti tko god je potreban da bi se problem riješio. U Toyota proizvodnji implementiraju ovu praksu pomoću nečeg što se zove Andon konopac (eng. *Andon cord*). Svaki radni centar ima obješen konopac kojeg zaposlenik poteže u slučaju pogreške, primjerice kad neki dio nije ispravan ili kad potreban dio nije dostupan. Nakon što je Andon konopac povučen voditelj tima je obaviješten i on odmah počinje raditi na rješavanju tog problema. Ako ne može riješiti problem u definiranom vremenskom rasponu (npr. jedna minuta) tada se čitava produkcijska linija zaustavlja i čitava organizacija se kreće kako bi riješila problem. Ovo je potpuno različito uobičajenom pristupu kada se radi oko problema i definira se popravak nekad kasnije kad se bude imalo više vremena. Ovakvo napadanje na problem je važno jer tako nije moguće da se problem pušta dalje kroz sustav gdje mu se cijena popravka samo povećava. Onemogućuje radnom centru pokretanje novog zadatka koji će vjerojatno uključiti nove probleme u sustav (Kim et al., 2016).

Pristup se može činiti ne produktivan jer lokalni problem uzrokuje obustavljanje operacija globalno, no na ovaj način omogućava učenje kakvo prije nije bilo moguće. Onemogućuje gubitak informacija jer se problem rješava odmah, a ne nakon nekog vremena. Ovo je vrlo važno u kompleksnim sustavima gdje je kasnije vrlo teško rekonstruirati događaje zbog kojih je došlo do problema te ih zbog toga ne možemo riješiti. Također je važno reagirati i na sitne probleme jer upravo oni kasnije postaju uzroci katastrofe, a kad dođe do katastrofe tada je prekasno za rješavanje (Kim et al., 2016).

Kako bi omogućili brzi povratni tok informacija u tehnološkom svijetu moramo implementirati svoju verziju Andon konopca i reagirati jednako kako u tvornici u Toyoti. Osim implementacije važno je kreirati kulturu koja daje sigurno okruženje i potiče povlačenje konopca kada se pronađe pogreška u vrijednosnom toku. Primjer pogreške u tehnološkom svijetu je pokušaj objave koda koji ne prolazi testove ili ga nije moguće izgraditi kako bi išao u produkciju. Kad se povuče konopac svi rade na rješavanju problema i ne kreće se dalje s poslom dok se ne riješi. Ovo daje brzu povratnu informaciju svima u sustavu i omogućuje nam brzu dijagnozu te zbog toga omogućuje prevenciju daljnjih komplikacija u vrijednosnom sustavu (Kim et al., 2016).

2.3.2.4. Približi kvalitetu izvoru

Mnogi sustavi pokušavaju uvesti više koraka provjere i odobrenja kako bi podigli razinu kvalitete, ali taj pristup je krivi i zapravo povećava vjerojatnost pojave pogrešaka. Efek-

tivnost procesa odobravanje se smanjuje što god je odluka za odobrenje dalje od mjesta gdje se izvršava posao. Ne samo da tako povećavamo vrijeme izvršavanja već i kvalitetu odluka, oslabljujemo povratnu informaciju o vezi između uzroka i posljedice i smanjujemo si mogućnost učenja iz naših uspjeha i promašaja. Kada metoda odozgo prema dole odnosno birokratski pristup zakaže tada je najčešće to zbog razlike tko treba učiniti nešto i tko to zapravo čini. Nekoliko primjera loše kontrole kvalitete (Kim et al., 2016) :

- Zahtijevanje ručno izvršavanja monotonog posla u kojem se često podvuku pogreške, a kojeg je lako automatizirati i onda pozivati kada je potrebno
- Zahtijevanje odobrenja od zaposlenih ljudi koji su udaljeni od posla koji se izvodi, te ih tako tjerajući da donose odluke o stvarima bez adekvatnog znanja i mogućih posljedica
- Kreirajući veliku dokumentaciju koja postaje nevažeća brzo nakon što je napisana
- Slanje velikog broja zadataka timovima i odborima na odobrenje i procesuiranje te čekanje na njihov odgovor

Umjesto ovog pristupa potrebno je imati sustav gdje svi sudionici svakodnevno pronalaze i rješavaju probleme u njihovoj domeni kontrole. Ovim pristupom dajemo odgovornost za sigurnost i kvalitetu onim osobama koje i obavljaju taj posao umjesto nekim distanciranim upraviteljima. Koristimo zajedničku provjeru rada kako bi osigurali da će se promjene ponašati kako je očekivano. Automatiziramo provjeru kvalitete što je više moguće kako bi razvojni programeri mogli napraviti reviziju u bilo kojem trenutku na zahtjev. Sljedeći ove korake kreiramo okruženje u kojem su svi zaduženi za kvalitetu, a ne samo jedan odjel poput operacija. Na ovaj način razvojni programeri uče puno bolje jer si međusobno čitaju kod i dobivaju brze povratne informacije (Kim et al., 2016).

2.3.2.5. Poboljšaj rad za daljnje sudionike vrijednosnog toka

Lean proizvodnja je također uvela nove ideje za dizajna dijelova i procesa tako da završni proizvod bude što jeftiniji, što veće kvalitete i da se brzo proizvodi. Primjer ove ideje je kreiranje dizajna dijelova koji su asimetrični kako bi onemogućili stavljanje dijela naopako ili kreiranje vijaka koje je nemoguće zavidati prečvrsto. Ovo je drugačiji pogled od dosadašnjeg jer smo prije u fazi dizajna gledali samo što klijent želi i kakav je njemu za koristiti ga, a sad gledamo i perspektivu radnika koji kreiraju proizvod. Ovo je omogućilo još bržu i bolju proizvodnju proizvoda. U tehnološkom vrijednosnom toku bi trebali optimizirati posao tako da bude što pogodniji operacijama, a da ne žrtvujemo ostale zahtjeve popust sigurnosti, performansi, arhitekture, itd. Koristeći ovu metodu osiguravamo kvalitetu već na samom izvoru koja će rezultirati u setu zahtjeva koje možemo poštivati kreirajući bilo kakav proizvod (Kim et al., 2016).

2.3.3. Treći put: načelo stalnog učenja i eksperimentiranja

Prvi i drugi put su se fokusirali na procesni tok, dok se treći put fokusira na kreiranje kulture koja potiče kontinuirano učenje i eksperimentiranje. Koristeći načela ovog puta kreirat ćemo organizaciju koja potiče konstantno kreiranje individualnog znanja koje će kasnije biti pretočeno u timsko i organizacijsko. Često u organizacijama posao je striktno definiran i zaposlenik ima malo prostora za poboljšavanje svog posla. Svaka sugestija koju bi zaposlenik poslao bi bila odbijena i posao se gotovo nikad ne bi mijenjao. U ovakvoj okolini zaposlenici osjećaju strah i ne povjerljivi su prema sustavu. Kažnjava ih se za sve pogreške koje kreiraju i oni koji pokušavaju promijeniti sustav se smatraju problematičnima. Voditelji aktivno zaustavljaju, čak kažnjavaju bilo kakva poboljšanja i zbog toga stalno ponavljaju iste probleme u kvaliteti i sigurnosti (Kim et al., 2016).

Ako želimo imati sustav s visokim performansama onda moramo aktivno promovirati učenje, posao ne smije biti rigorozan nego dinamičan i moramo poticati eksperimentiranje u svakodnevnom poslu. Sve nove promjene trebaju biti uvedene u standardnu proceduru i potrebno je dokumentirati rezultate. U tehnološkom svijetu cilj nam je kreirati okolinu s visokom razinom povjerenja koja naglašava da je učenje cjeloživotni proces i da trebamo svakodnevno uzimati rizike i eksperimentirati u našem poslu. Koristeći znanstveni pristup u poboljšanju procesa i proizvoda svakodnevno učimo što funkcionira, a što ne. Svako lokalno znanje brzo pretvaramo u globalno tako da su sve nove tehnike i prakse implementirane na razini čitave kompanije. Ne samo da potičemo poboljšanje već odvajamo dio radnog vremena za poboljšanje posla, radimo kako bi ga mogli ubrzati i osigurati daljnje učenje. Stalno vršimo pritisak na sustav i tjeramo ga na pad (naravno u kontroliranim uvjetima) kako bi vježbali oporavljanje od takvih situacija i povećali otpornost sustava. Implementirajući ovakvu kulturu u naš sustav omogućuje nam imati timove koji se brzo i lako prilagođavaju promjenama iz okoline što nam uvelike pomaže kako bi bili vodeći na tržištu (Kim et al., 2016).

2.3.3.1. Kreiranje organizacijske kulture koja daje osjećaj sigurnost i potiče učenje

U drugom putu smo opisivali kompleksni sustav i govorili kako je ne moguće preduvjeti sve moguće izlaze i mogućnosti u njemu. Zbog toga se znaju događati neočekivane stvari u našem svakodnevnom poslu čak i kada pazimo što radimo. Kada dođe do pogreške koja utječe na naše klijente tada tražimo koji je uzrok pogreške. Često je izvor pogreške ljudska greška i često menadžment odluči kazniti osobu koja je uzrokovala problem. Zatim kreiraju još procesa i odobrenja za prevenciju takvih pogrešaka za koje smo u prošlom poglavlju vidjeli da ne funkcioniraju. Ovakav slijed događaja je čest u tehnološkom svijetu i ovakva reakcija menadžmenta uzrokuje kulturu straha koja potiče zaposlenike da prikrivaju svoje pogreške i nikad ih ne prijavljuju. Rezultat toga su pogreške koje idu dalje kroz procesni tok i uzrokuju katastrofalne pogreške (Kim et al., 2016). Westrum (2014) je izučavao važnost organizacijske kulture na sigurnost i performanse.

Tokom izučavanja je definirao sljedeća tri tipa kulture:

- Patološke organizacije karakterizira velika količina straha i prijetnji. Ljudi većinom prikupljaju informacije i drže ih za sebe zbog političkih razloga ili da oni izgledaju bolje zbog toga. Pogreške se često skrivaju.
- Birokratske organizacije karakteriziraju pravila i procesi, svaki odjel se drži za sebe. Pogreške se gledaju kroz sustav osude i rezultiraju kažnjavanjem ili pravdom i milosti.
- Generativne organizacije karakterizira aktivno traženje i dijeljenje informacija kako bi organizacija lakše postigla svoju misiju. Odgovornosti su dijeljene kroz čitavi vrijednosni sustav i pogreške rezultiraju u refleksiji i iskrenoj istrazi.

Tablica 1: Kako organizacije procesiraju informacije

Patološka	Birokratska	Generativna
Orijentirana moći	**Orijentirana pravilima**	**Orijentirana performansama**
Informacije su skrivene	Informacije mogu biti ignorirane	Informacije se aktivno traže
Glasnici su "upucani"	Glasnici se toleriraju	Glasnici se treniraju
Odgovornosti se smanjuju	Odgovornosti su podijeljene	Odgovornosti su zajedničke
Povezivanje među timovima se obeshrabruje	Povezivanje među timovima je dozvoljeno, ali se obeshrabruje	Povezivanje među timovima se nagrađuje
Greške se skrivaju	Greške se sude i rezultiraju pravdom i milosti	Pogreške se istraživaju
Nove ideje su zabranjene	Nove ideje kreiraju probleme	Nove ideje su dobrodošle

(Prema izvoru: Westrum, 2004)

U tablici 1 možemo vidjeti razliku odnosa prema informacijama između Westrumovih tipova organizacija. Patološki tip organizacijske kulture je najgori tip jer zaposlenici osjećaju strah i čuvaju informacije za sebe. Birokratski rješava neke probleme, ali i dalje imaju odbojnost prema novim idejama i odgovornosti se ne dijele. Generativni tip je tip koji želimo postići u našoj organizaciji. Nove ideje su dobrodošle, informacije se dijele te stvaraju globalno znanje i pogreške se ne osuđuju već istražuju, cilj je riješiti problem i učiti iz njega. Pogreške nisu krivnja zaposlenika već gledamo kako možemo redizajnirati sustav kako bi izbjegli takve pogreške. Primjerice možemo nakon svake pogreške napraviti "obdukciju" gdje nam je cilj bolje shvatiti pogrešku i kako je došlo do nje te zajedno razmisliti koje su najbolje protumjere koje možemo napraviti kako se problem ne bi ponovio i kako ga možemo ranije primijetiti te se brže oporaviti od njega. Koristeći ove korake kreiramo organizacijsko znanje. Ako umjesto optužbe implementiramo organizacijsko znanje tada se organizacija kontinuirano poboljšava i postaje puno bolja u rješavanju problema (Kim et al., 2016).

2.3.3.2. Svakodnevno poboljšavanje posla

Timovi često nemaju vremena ili nisu voljni poboljšavati svakodnevne poslove. Važno je postaviti organizacijsku kulturu koja ih potiče na to jer zadatci koji se kontinuirano ne

poboljšavaju čak ne ostaju jednako dobri, već postaju sve lošiji zbog dinamične okoline. U tehnološkom vrijednosnom toku zapostavljanje problema i njihovo zaobilaženje dovodi do još većih problema i tehnološkog duga do trenutka kada dođe do katastrofe i više nije moguće produktivno izvoditi zadatke. Poboljšavanje zadataka izvodimo tako da odvojimo vremenski blok u radnom vremenu specifično za to i tada se vraća tehnički dug, popravljaju pogreške i refaktorira programski kod. Rezultat implementiranja ove prakse je da svi zaposlenici pronalaze i rješavaju probleme u svom poslu svakodnevno kao dio njihovog posla. Nakon što je prođe nekoliko mjeseci korištenja ove prakse, tada smo riješili sve problema za koje smo znali i zaobilazili. Zatim možemo početi rješavati manje očite probleme dok su jednostavniji i jeftiniji za popraviti. Tako dižemo kvalitetu čitavog vrijednosnog toka jer se problemi ne šalju dalje kroz sustav. Važno je početi s nekim najvećim problemima pa ćemo se postupno proširiti na druge probleme i rješavati ih. Primjerice započnemo samo s ne optužujućom obdukcijom problema koji su imali utjecaj na naše klijente. Nakon nekog vremena će se početi rješavati problemi koji utječu i na tim i ne samo rješavati problemi kad se dogode nego poboljšavati događaje kada se dogodi da smo samo bili blizu tome da dođe do pogreške (Kim et al., 2016).

2.3.3.3. Pretvaranje lokalnog znanja u globalno

Vrlo je važno svako lokalno znanje koje netko otkrije pretvoriti u globalno znanje jer tada ga čitava organizacija može iskoristiti. Gdje imamo timove ili individue koji su eksperti u nečemu, cilj nam je to znanje pretvoriti u znanje koje svi mogu iskoristiti. Znanje se može prenositi tako da se zapiše kako bi neka osoba mogla dobiti to znanje kroz proučavanje i vježbu. Ova praksa omogućuje da kad god neki zaposlenik radi neki zadatak ne koristi samo svoje vlastito znanje već znanje i iskustvo čitave organizacije. Primjer ove prakse možemo promotriti na primjeru američkog programa za pogon na nuklearnu snagu koji ima preko 5700 reaktor godina bez ijedne pogreške vezano za reaktor ili gubitak radijacije. Do ovog uspjeha su došli tako što imaju veliku predanost pisanim procedurama, standardiziranom poslu i dokumentiranju svih incidenata koji imalo odmiču od uobičajenog. Kontinuirano ažuriraju procese i dizajn sustava na osnovi ovog znanja. Rezultat je to da svaki tim posjeduje kolektivno znanje veličine 5700 reaktor godina bez ijedne pogreške. Da bi postigli ovakav uspjeh moramo i mi implementirati slične mehanizme u našem tehnološkom svijetu. Jedan primjer bi bila baza znanja koja bi sadržavala svaku obdukciju problema s kojom smo se do sad susreli. Također bi trebali imati dijeljeni repozitoriji izvornog koda gdje dijelimo naš kod, biblioteke i konfiguracije kako bi podijelili znanje koje svi zajedno možemo koristiti. Svi ovi alati pretvaraju individualnu ekspertizu u globalno znanje (Kim et al., 2016).

2.3.3.4. Povećavanje otpornosti svakodnevnog posla

Organizacije znaju dodavati viškove kako bi se zaštitili od zaustavljanja svakodnevnog posla. Primjerice kako bi se osigurali da uvijek imaju dovoljno materijala za rad naručuju više nego što im je potrebno ili recimo kako bi se osigurali od kvara strojeva kupuju dodatne strojeve i zapošljavaju više ljudi. Ranije smo detaljno opisali kako su viškovi loši i ovaj pristup povećava količinu posla u sustavu što smo također opisali kao jedan od glavnih problema. Ispravno

rješavanje problema je svakodnevno poboljšavanje svakodnevnog posla i namjerno dodavanje stresa u sustav kako bi povećali otpornost sustava i prepoznali slabe točke. U tehnološkom svijetu ovaj pristup često dovodi do smanjivanja vremena isporuke, povećanje pokrivenosti programskog koda s testovima i smanjenje vremena potrebno da se testovi izvrše. Neke firme imaju posebno izdvojene radne dane gdje ubacuju stres visoke razine u svoj sustav poput gašenja svih podatkovnih centara. Primjerice Netflix je napravio program nazvan "Chaos Monkey" koji nasumično ubija procese i servere u produkciji i tako stvara veliki izazov programerima da osiguraju da sustav ostane funkcionalan. Tako smo puno bolje zaštićeni od zaustavljanja svakodnevnog posla (Kim et al., 2016).

2.3.3.5. Voditelji trebaju poticati kulturu učenja

Tradicionalno voditelji su trebali biti odgovorni za postavljanje ciljeva, alociranje resursa za njihovo postizanje i postavljanje emocionalnog tona tvrtke koje vode. Smatralo se da su oni odgovorni i zasluženi za uspjeh ili ne uspjeh organizacije. Danas znamo da istina nije takva nego da voditelji trebaju kreirati uvijete koji dozvoljavaju njihovom timu da postignu uspjeh u svom svakodnevnom poslu. Drugim riječima za uspjeh su potrebni jedni i drugi te su zasluge i odgovornosti zajedničke. Važno je strateške ciljeve podijeliti u manje ciljeve kraćeg trajanja kojeg onda timovi mogu postići i mjeriti. Ciljeve možemo promatrati kao znanstveni eksperiment. Moraju imati jasno objašnjen problem koji pokušavamo riješiti, ideju kako taj problem riješiti, metode za testiranje tih ideja, interpretaciju rezultata i iskoristiti nova znanja u sljedećoj iteraciji. Voditelj uči zaposlenika voditi eksperimente pomoću sljedećih pitanja (Kim et al., 2016) :

- Koji je bio tvoj posljednji korak i što se dogodilo?
- Što si naučio?
- Kakvo je trenutno stanje?
- Kakvo je stanje sljedećeg cilja?
- S kojim preprekama se trenutno susrećeš?
- Koji ti je sljedeći korak?
- Kakav rezultata očekuješ?
- Kad možemo provjeriti rezultat?

Ovaj pristup rješavanja problema pomaže zaposlenicima riješiti probleme u svom svakodnevnom poslu i uči ih kako razmišljati i osmišljavati eksperimente u svom poslu. U tehnološkom svijetu ovaj pristup bi trebao voditi unaprjeđenje svih naših procesa i način na koji eksperimentiramo (Kim et al., 2016).

2.4. Zaključak

Kroz DevOps poglavlje smo upoznali što pojam predstavlja, odakle je došao i što stoji iza čitave ideje. Poglavlje je ključni dio rada jer je vrlo važno shvatiti što praksa predstavlja kako bi mogli razumjeti zašto ju je potrebno implementirati i kako će koraci koje ćemo napraviti ispunjavati navedene ciljeve. Tri puta koja su opisana predstavljaju temeljne principe DevOps filozofije i vrijede za primjenu DevOpsa s bilo kojom tehnologijom.

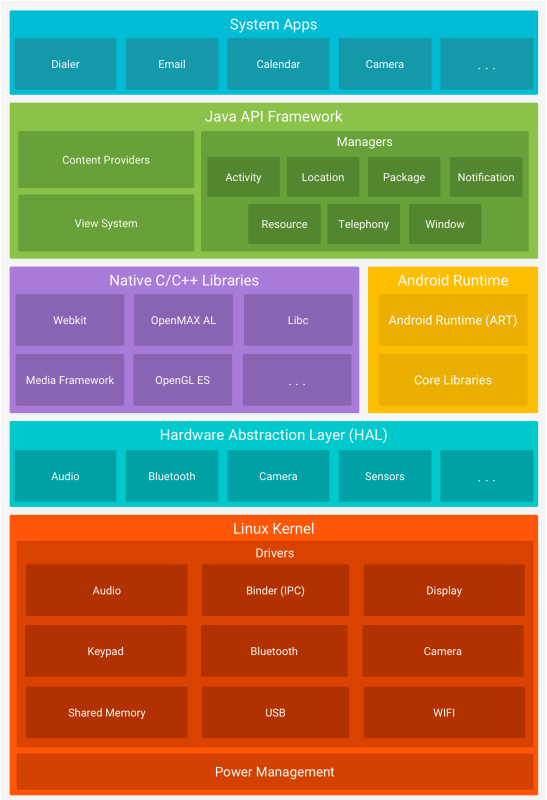
Osim objašnjenje pojma poglavlje nam daje motivaciju za implementaciju DevOpsa u svaki informacijski projekt kojeg trenutno radimo i koji ćemo u budućnosti raditi jer su posljedice jako pozitivne i uvelike nam pomažu u ostvarivanju svih projektnih ciljeva. Sad kad smo motivirani i razjasnili pojam možemo krenuti na sljedeće poglavlje koje će nam opisati odnos između Androida i DevOpsa. Prikazat ćemo kako primijeniti ideje u Android procesni tok i koji alati su nam raspolaganju.

3. Android i DevOps

U prošlom poglavlju smo detaljno opisali što je to DevOps i kako je došlo do njega. Nakon što imamo razumijevanje njegovih ciljeva pogledat ćemo kako primjenjujemo te ciljeve prilikom razvoja Android aplikacija. Upoznat ćemo razne alate koji nam olakšavaju dolazak do naših ciljeva. Potrebno je zapamtiti da su alati samo dio DevOps priče, drugi dio je organizacijska kultura koja treba biti postavljena unutar organizacije. Tek nakon cjelovite implementacije možemo drastično utjecati na naš procesni tok i zadovoljstvo zaposlenika i klijenata.

3.1. O Androidu

Android je operacijski sustav kreiran za veliki broj raznih uređaja. Uređaji se kreću od pametnih uređaja, tableta, televizija, satova, auta, itd. Otvorenog je koda i baziran na Linux operacijskom sustavu ("Platform Architecture | Android Developers", n.d.). Danas ima oko 2.5 milijardi aktivnih Android korisnika i 85% globalnog tržišta mobilnih uređaja koristi Android operacijski sustav ("95 Amazing Android Statistics", 2014). Trenutno postoji 2.1 milijuna različitih Android aplikacija na njihovoj trgovini aplikacijama ("App Stores", 2019). Možemo primijetiti da je tržište vrlo veliko i potrebna je visoka razina kvalitete proizvoda kako bi se mogli natjecati s konkurencijom.



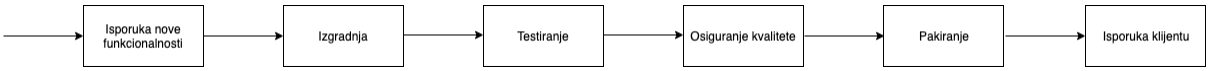
Slika 6: Android arhitektura (Izvor: ("Platform Architecture | Android Developers", n.d.))

Na slici 6 možemo vidjeti arhitekturu čitavog Android sustava. Na dnu se nalazi Linux kernel koji pruža određene funkcionalnosti na koje se Android oslanja poput dretvi, upravljanje memorijom, upravljanje energijom, itd. Sljedeći je sloj apstrakcije fizičkih komponenti (*eng. hardware abstraction layer*. On pruža sučelje za upravljanje fizičkim komponentama uređaja pomoću programskih jezika više razine. Sljedeću razinu dijele nativne C/C++ biblioteke (*eng. Native C/C++ libraries*) i Android izvršno okruženje (*eng. Android runtime*). Svaka aplikacija koja je pokrenuta na uređaju pokreće svoju instancu Android izvršnog okruženja i u njemu se izvršava. On pruža dodatne funkcionalnosti poput kompiliranje prije vremena i u pravom trenutku, optimizira skupljanje otpada, itd. C i C++ biblioteke služe kao podrška Android izvršnom okruženju i mnogim sustavskim procesima. Sljedeći sloj je Java API programski okvir koji koriste programeri koji razvijaju Android aplikacije. Sadrži u sebi čitav set funkcionalnosti koji pruža gradivne blokove pomoću kojih kreiramo aplikacije. Posljednji sloj su sustavske aplikacije

gdje se nalaze aplikacije koje dolaze s Androidom poput aplikacija za email, SMS, kalendar, internet preglednik, itd. Svrha im je pružanje ključnih funkcionalnosti korisniku čim pokrene Android uređaj. Set aplikacija može biti proširen instaliranjem aplikacija s Google trgovine aplikacijama Google Play Store ("Platform Architecture | Android Developers", n.d.).

3.2. Android procesni tok

Nakon što smo se upoznali s Android sustavom možemo vidjeti od kojih procesa se sastoji procesni tok. Pod procesni tok smatram čitav proces od početka razvoja funkcionalnosti do isporuke korisniku i na koji način možemo implementirati DevOps kako bi ga maksimalno optimizirali.



Slika 7: Android procesni tok (Prema izvoru: (Martinez, 2016))

Na slici 7 možemo vidjeti osnovne procese od kojih se sastoji Android procesni tok. Započinje s isporukom nove funkcionalnosti, u tom koraku programer isprogramira funkcionalnost i šalje ju na sustav za verzioniranje programskog koda (git operacija push). Korak izgradnje podrazumijeva izgradnju programskog koda, najčešće se radi pomoću Gradlea i tada možemo izgraditi više različitih verzija. Zatim testiramo programski kod, izvodimo jedinično i integracijsko testiranje. Sljedeći korak je osigurati kvalitetu. Najčešće to izvodimo kroz zahtjev za spajanjem koda (git operacija pull request). Također možemo koristiti lintere kako bi mogli automatski provjeravati imamo li nekih neispravnosti ili ne konzistentnosti u kodu. Zatim pakiramo programski kod, potrebno je filtrirati resurse koje ne koristimo, napraviti obfuskaciju i minifikaciju koda i zapakirati u APK datoteku koju Android može pročitati i instalirati aplikaciju. Za kraj je potrebno aplikaciju isporučiti klijentu što uključuje korištenje Google konzolu za razvojne inženjere pomoću koje objavljujemo aplikaciju na Google Play Store (Martinez, 2016).

3.2.1. Isporuka programskog koda

Razvojni programeri svakodnevno u svom poslu kreiraju nove dijelove programskog koda. Razlozi za promjenu su različiti, mogu biti rezultat kreiranja nove funkcionalnosti, popravljanja pogreške, refaktoriranja, itd. Kad programer kreira novi dio programskog koda on mora kod postaviti u nekakav zajednički repozitoriji kako bi programski kod bio dostupan svim ostalim članovima tima. Čak i ako je programer jedna osoba potrebno je koristiti repozitoriji kako bi mogli kodu pristupiti od bilo kuda i imati sigurnu kopiju na nekom udaljenom mjestu. Također bi taj repozitoriji trebamo imati nekakav sustav za verzioniranje koda u sebi kako bi mogli vidjeti tko je objavio koje promijene i da imamo mogućnost vratiti se na neku stariju verziju koda.

Sustav za verzioniranje je sustav koji sprema promjene nad datotekama tijekom vremena tako da možemo dohvatiti neku specifičnu verziju iz prošlosti. Omogućava povratak datoteke ili čak čitavog projekta na neko prijašnje stanje. Osim povratka možemo uspoređivati

različite verzije i povezuje osobe uz promjene tako da je moguće vidjeti koja osoba je napravila koje promjene i još mnogo toga. Ukratko omogućuje povratak datoteka na neku prijašnju verziju u slučaju pogreške ili gubitka podataka (Chacon & Straub, 2014).

Prvi pokušaji verzioniranja su bili takvi da bi ljudi jednostavno kopirali trenutno stanje projekta u neku odvojenu datoteku i nazvali ju nekako opisno kako bi znali koja je to verzija projekta. Ovaj pristup je jednostavan, ali je vrlo lako napraviti pogreške i pomiješati verzije ili slučajno obrisati krivu verziju. Osim toga sve verzije su samo na našem lokalnom računalu tako da u slučaju kvara možemo izgubiti sve podatke. Također je bio problem što su programeri radili u timovima i surađivali pa je bilo problematično međusobno usklađivati datoteke i verzije. Zatim je došlo do ideje da imamo jedan centralni sustav koji čuva verzije programskog koda na koji čitavi tim može pristupati i verzionirati. Kreirani su i sustavi koji su to olakšavali (CVS, Subversion i Perforce). Dugo godina je ovo bio način kako se radilo verzioniranje koda. Prednost ovog sustava je što su svi zapravo bili na istoj verziji sustava i imali pristupe svim podatcima. Velika mana je ta što ako server padne nitko ne može pristupiti sustavu. Također ako se disk pokvari možemo sve izgubiti ako se nisu radile sigurnosne kopije. Odgovor na navedene probleme je bio distribuirani sustav za verzioniranje. Umjesto da korisnici koji koriste sustav skinu samo posljednju verziju na svoje računalo oni skidaju čitavu povijest projekta i imaju doslovnu kopiju onoga što je na serveru. Tako da u slučaju da server ne radi svi korisnici imaju kopiju repozitorija sa svim verzijama i mogu ga lako ponovno podići. Postoji mnogo sustava danas koji to rade, ali najpopularniji je Git i mi ćemo njega koristiti tokom pokazivanja primjera. Nećemo detaljno opisivati Git jer je to van područja ovog rada već ćemo spomenuti koje metode koristimo kako bi smo implementirali DevOps prakse u ovom koraku (Chacon & Straub, 2014).

Jedna od vrlo korisnih funkcionalnosti sustava za verzioniranje je grananje. Ono nam omogućava da se odvojimo od glavne linije razvoja i radimo nekakve promjene bez da uništimo trenutno stanje. Preporučen model grananja unutar projekta je takozvani Git Flow koji je kreirao Vincent Driessen. Zasniva se na tome da imamo dvije glavne grane master i develop koje imaju beskonačan život trajanja. Master bi trebala biti glavna grana koja reflektira programski kod koji je spreman za produkciju. Develop grana treba imati programski kod koji reflektira posljednje stanje izvornog koda koji će biti u sljedećoj verziji našeg programa. Kada programski kod u develop grani postane stabilan i spreman za objavu tada ga spajamo s master granom i označavamo s oznakom koja predstavlja verziju sustava. Također imamo 3 tipa grana koje imaju ograničen vijek trajanja, a to su feature, release i hotfix. Feature koristimo kad želimo napraviti neku novu funkcionalnost u sustavu i tada se razgranamo iz develop grane i na njoj razvijamo funkcionalnost. Kada smo gotovi spajamo ju nazad u develop. Release grana služi kao potpora, odnosno za pripremu kada ćemo napraviti objavu nove verzije koda. Ovako možemo napraviti neke posljednje sitnice koje su ostale bez da kočimo develop granu u nastavku primanja nove verzije koda koje ćemo koristiti u sljedećoj verziji sustava. Kada release grana ima sve potrebne promjene tada ju spajamo s master granom i imamo novu verziju sustava i ne smijemo zaboraviti spojiti i sa develop granom. Time završava život te grane. Posljednja je hotfix grana i nju koristimo u slučaju da trenutna verzija u produkciji ima kritičnu pogrešku, a develop grana još nije stabilna. Tada kreiramo granu hotfix iz master grane, popravimo grešku i spojimo nazad u master s novom verzijom i naravno u develop granu. Svaka objava na mas-

ter granu je verzija koju šaljemo na produkciju. Ova organizacija sustava za verzioniranje se pokazala kao jako dobra praksa i često je korištena (Driessen, 2010).

Još jedna vrlo korisna praksa koju smo spominjali i u ranijim poglavljima je revizija koda. Svaki napisani komad programskog koda mora pregledati neki član tima koji nije napisao taj programski kod. Na taj način podižemo kvalitetu jer ćemo prije prepoznati pogreške i potičemo učenje jer osoba koja pregledava programski kod istovremeno uči čitati tuđe programske kodove i možda nauči nešto novo što nije znao od osobe koja je pisala taj programski kod. Također potiče zdravu raspravu jer će osoba koja pregledava primjerice komentirati "Zašto si X stvar napravio na Y način a ne na Z". Autor koda će se morati argumentirati i naučiti kolegu nešto novo ili shvatiti da je način Z možda bolji. Github (jedan od mnogih alata koji je cloud platforma sagrađena oko Git alata) nam olakšava čitav taj proces kroz nešto što se zove Pull request. Pull request je zapravo zahtjev za spajanje programskog koda iz jedne grane u drugu. Daje nam automatski prikaz razlika između dvije grane (koji dijelovi su dodani, a koji su obrisani) i sučelje nam omogućava komentiranje čitavog koda ili specifičnih linija. Mogu se dodati osobe koje će pregledati programski kod. Nakon revizije može se odobriti spajanje grana čime završava čitav taj proces.

3.2.2. Izgradnja

Android programski kod može biti pisan koristeći Kotlin, Java ili C++ programski jezik. Nakon što napišemo programski kod Android SDK alati izgrade taj kod zajedno sa svim podatcima i resursima u .apk datoteku. APK je skraćenica od riječi Android paket (*eng. Android package*). Datoteka sadrži sve podatke potrebne za aplikaciju i Android sustav pomoću nje instalira aplikaciju (“Application Fundamentals”, n.d.). Android SDK alati koriste Android sustav za izgradnju kako bi kompilirali resurse i izvorni kod te ih zapakirali u APK datoteku koja se može testirati i objaviti. Android sustav za izgradnju koristi Gradle i Android priključak (*eng. plugin*) za Gradle. Gradle je alat za automatiziranje izgradnje koji je otvorenog koda. Omogućava nam kreiranje konfiguracija kako bi izgradili kod onako kako mi želimo. Android priključak za Gradle nam omogućuje konfiguraciju izgradnje koja specifična za Android. Kroz konfiguraciju možemo mijenjati sljedeće aspekte izgradnje (“Configure Your Build”, n.d.) :

- Tipovi izgradnje: moguće je imati više tipova izgradnje i to je vrlo korisno jer često želimo imati različito konfigurirane testne i produkcijske aplikacije. Recimo da u testnoj aplikaciji želimo prikazivati sve pogreške u konzoli kako bi ih lakše mogli vidjeti i koristimo sve testne ključeve za vanjske servise, dok za produkcijsku aplikaciju to ne želimo već ju želimo maksimalno smanjiti i obfuscirati kod.

- Okusi proizvoda: (*eng. product flavors*) predstavljaju različite verzije aplikacija koje možda želimo dati korisnicima, recimo besplatna i plaćena verzija aplikacije. Omogućava nam da različite verzije koriste različite dijelove koda, a da opet mogu dijeliti jedan zajednički dio.

- Varijanta izgradnje: Ovo je kombinacija tipa i okusa iz gornjih dviju točki. Omogućava nam da recimo kreiramo testnu verziju besplatne aplikacije.

- Manifest zapisi: Android aplikacije sadrže jednu posebnu datoteku koja se zove manifest. Ona definira koja prava zahtjeva aplikacija, kako se zove, koja je minimalna verzija koju podržava, itd. U konfiguraciji možemo definirati drugačija pravila koja gaze ona definirana originalno i postavljaju se kao vrijedeća.
- Više APK verzija: omogućava nam automatsku izgradnju više različitih APK-ova gdje svaki sadrži kod i resurse potrebno samo specifično za recimo neku gustoću piksela.

Gradle je vrlo koristan jer s njim možemo automatizirati čitav proces izgradnje i na taj način izbjeći pogreške i ubrzati procesni tok. Bilo tko može izgraditi aplikaciju na zahtjev.

3.2.3. Testiranje

Testiranje je jedan od ključnih koraka u programskom razvoju. Stalnim testiranjem osiguravamo da nam programski kod funkcionira kako je očekivano. Također nam daje brzu povratnu informaciju, brže prepoznajemo pogreške, lakše nam je refaktorirat kod jer lako možemo provjeriti jesmo li možda slučajno promijenili kako funkcionira i pomaže nam minimizirat tehnološki dug ("Test Apps on Android", n.d.).

Da bi nam testovi donijeli navedene prednosti važno je napisati testove koji pokrivaju što više programskog koda i mogućih interakcija. Dobra je praksa prije pisanja nove funkcionalnosti napisati test za tu funkcionalnost pa zatim pisati programski kod koji će proći te testove. Važno je da pokrijemo sve moguće interakcije s programskim kodom, a ne samo očekivani put. Pisanje testova je iterativan proces jer svaki put kad idemo dodati novi programski kod, trebamo pisati i test za njega. Pisanje testova si možemo olakšati ako pišemo programski kod u smislu modula, gdje svaki modul predstavlja specifičan zadatak koji korisnik može napraviti unutar aplikacije. Primjerice ako radimo aplikaciju koja sadrži popis zadataka tada bi imali modul za kreiranje zadataka, za prikaz statistike riješenih zadataka, za prikaz popisa zadataka, itd. Ova arhitektura nam omogućuje da imamo odvojene klase i prirodnu strukturu za dodjelu vlasništva u timu. Važno je imati jasno definirane granice modula i kreirati nove module kako kompleksnost raste. Svaki modul mora imati jedan zadatak i komunikacija među modulima treba biti konzistentna ("Fundamentals of Testing | Android Developers", n.d.).

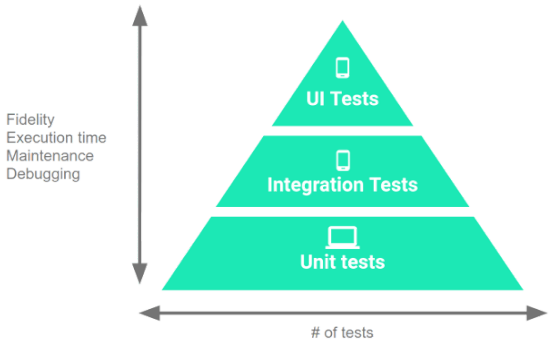
Osim organizacije programskog koda aplikacije potrebno je organizirati i testove. Najčešće testove odvajamo u 2 direktorija, jedan zovemo `androidTest` i on sadrži testove koji trebaju biti pokrenuti na stvarnom ili virtualnom uređaju, a drugi `test` i on sadrži kod koji može biti pokrenut na lokalnoj mašini. Ako testove moramo pokretati na uređaju tada možemo birati između testiranja na stvarnom, virtualnom ili simuliranom uređaju. Stvarni uređaji daju najveću širinu, ali im treba najviše vremena da se pokrenu testovi. Simulirani uređaji imaju bolju brzinu, ali manju širinu. Virtualni uređaji pružaju ravnotežu između širine i brzine. Mi ćemo koristiti simulirane uređaje kad god je moguće jer nam je cilj što veća brzina i mogućnost testiranja bez intervencije osobe ("Fundamentals of Testing | Android Developers", n.d.).

Prilikom kreiranja testova moramo odlučiti hoćemo li koristiti lažne podatke ili stvarne. Općenito je bolje ako možemo koristiti stvarne objekte jer testiramo aplikaciju u stvarnim uvjetima, ali ako recimo u testu pozivamo čitanje nekog velikog teksta koji dugo traje tada je bolje

koristiti lažan objekt (“Fundamentals of Testing | Android Developers”, n.d.).

Razlikujemo male, srednje i velike testove. Mali testovi su jedinični testovi *eng. unit test* koji validiraju ponašanje jedne po jedne klase. Srednji testovi su integracijski testovi koji validiraju interakcije među razinama unutar pojedinog modula i među različitim povezanim modulima. Veliki testovi su testovi grafičkog sučelja koji validiraju korisničko iskustvo preko više modula aplikacije.

Na slici 8 možemo vidjeti piramidu testova koja prikazuje navedene 3 kategorije testova. Kako idemo s dna prema vrhu povećava se širina testova, ali i vrijeme izvođenja, trud potreban za održavanje i pronalazak pogrešaka. S lijeva na desno se povećava količina koliko bi testova trebali imati. Dakle možemo zaključiti da bi najviše trebali pisati jedinične testove, pa integracijske i najmanje testove korisničkog sučelja. Preporuka je da se piše 70% malih, 20% srednjih i 10% velikih testova (“Fundamentals of Testing | Android Developers”, n.d.).



Slika 8: Piramida testova (Izvor: (“Fundamentals of Testing | Android Developers”, n.d.))

3.2.4. Osiguranje kvalitete

Osiguranje kvalitete je vrlo važan proces jer kvalitetan programski kod je lako čitljiv, održiv i većina pogreški je otkriveno prije nego što je programski kod uključen u produkcijsko okruženje. Već smo ranije spomenuli jednu metodu za osiguranje kvalitete, a to je revizija koda. Koristeći metodu zahtjeva za spajanje koda (*eng. pull request*) svaki se programski kod prije spajanja u razvojnu granu ili neku drugu provjerava i pregledava ima li neke funkcionalne ili bilo kakve druge pogreške. Osim što je odličan alat za osiguranje kvalitete on potiče proširivanja znanja među razvojnim programerima.

Dolazimo do problema ako se razvojni tim sastoji od samo jednog programera ili ako želimo uključiti i automatsku provjeru programskog koda. Tada moramo potražiti neke alate za analizu kvalitete koje možemo koristiti. Alati za analizu kvalitete pregledavaju izvorni kod, izvršne datoteke i ponekad čak i dokumentaciju kako bi pronašli probleme i prije nego će se dogoditi bez da pokreću programski kod. Alati nas tjeraju da poštujemo programske standarde, promatraju kompleksnost funkcija, klasa ili datoteka, također mogu promatrati dokumentaciju i mjeriti pokrivenost programskog koda s testovima. Ovo su samo neke od svih mogućnosti i možemo zaključiti da su vrlo korisni čak i ako imamo osobu koja nam može pregledati programski kod. Jednostavno ih možemo uključiti u naš razvojni proces i dodati u Gradle kako bi prilikom svake izgradnje napravili analizu (Ebel, 2018).

Jedan od najpopularnijih je `SonarQube` koji ima osnovnu verziju koja je otvorenog koda i besplatna. Radi provjeru kvalitete programskog koda koja uključuje pronalazak programskog koda koji bi mogao biti problematičan za održavanje (*eng. Maintainability Issue*), pronalazak

potencijalnih pogrešaka gdje bi se programski kod mogao ponašati neočekivano (*eng. Reliability Issue*) i pronalazak sigurnosnih problema (*eng. Security Issue*). Podržava integraciju s popularnim razvojnim okruženjima, alatima za automatsku izgradnju poput Gradlea i alatima za kontinuiranu integraciju. Zbog navedenih razloga ćemo ga i mi koristiti u kasnijem poglavlju koje će pokrivati praktični dio izrade ("Code Quality | SonarSource", n.d.).

Koristit ćemo i Android alat za statičku analizu koda koji se zove `Android Lint`. Pomaže u pronalaženju loše strukturiranog koda koji može utjecati na sigurnost, pouzdanost ili čitljivost koda. Svaki problem koji identificira prijavi s opisnom porukom i razinom opasnosti koju predstavlja. Vrlo je korisno to što ga možemo konfigurirati i dodavati vlastita pravila. Recimo možemo mu reći da ignorira određeno pravilo za ovaj projekt i recimo dodati pravilo da primjerice svaki atribut u resursima ovog projekta mora imati određeni prefiks ("Improve Your Code with Lint Checks | Android Developers", n.d.).

3.2.5. Pakiranje

Nakon što smo izgradili aplikaciju, testirali ju i analizirali možemo ju zapakirati kako bi ju mogli isporučiti klijentu. Prvi korak bi bio maknuti sve resurs koje ne koristimo i to možemo napraviti vrlo jednostavno koristeći `Android Lint`. Sljedeći korak je korištenje alata ProGuard koji maksimalno minimizira aplikaciju mičući sve dijelove koda, biblioteke i resurse koji se ne koriste. Također radi obfuskaciju koda kako bi otežali reverzni inženjering našeg koda i smanjili veličinu aplikacije skraćujući imena svih klasa, metoda i svojstava. Koristimo ga tako da ga uključimo u konfiguraciju Gradlea ("Shrink, Obfuscate, and Optimize Your App", n.d.).

Zatim je potrebno potpisati aplikaciju. Android zahtjeva da svaki APK bude potpisan s certifikatom prije nego što je instaliran na uređaj. Ako nemamo certifikat prvo je potrebno generirati ključ i spremnik za ključeva koji će se koristiti za potpisivanje aplikacije, a zatim potpisati aplikaciju s generiranim ključem ("Sign Your App", n.d.).

Za kraj je potrebno koristiti alat `zipalign` koji je alat za poravnanje arhiva. Daje važnu optimizaciju koja je potrebna APK datotekama. Poravnaje arhivu na način da je sve poravnato u odnosu na početak datoteke. Sada je datoteka spremna za objavu na Google Play trgovinu ("Zipalign | Android Developers", n.d.).

3.2.6. Isporuka klijentu

U kontekstu Android aplikacija isporuka klijentu je najčešće objava na Google Play trgovinu. Proces se sastoji od pripreme za objavu gdje smišljamo opis i slikamo zaslon s različitim ekranima aplikacije i same objave na na trgovinu. Ako aplikaciju objavljujemo prvi put tada je potrebno napisati opisni tekst aplikacije i uslikati ekrane s ključnim funkcionalnostima aplikacije, ako se radi o ažuriranju tada treba napisati opisni tekst koji govori što je novo u ažuriranju i slikati ekrane koji su se promijenili za razliku od prije.

Slikanje ekrana prilikom svake nove verzije može biti zamorno i puno je bolje ako možemo automatizirati taj proces. Možemo ga automatizirati tako da napišemo neki svoj pro-

gram koji će to raditi ili iskoristimo neki od postojećih poput `Spoon` ili `Screengrab`. Mi ćemo u primjeru koristiti Screengrab kako bi automatizirali taj proces (Martinez, 2016).

Proces objave također želimo automatizirati kako bi minimizirali broj pogrešaka i ubrzali taj proces. Automatiziramo koristeći Google Play Android Developer API. Pomoću njega možemo napraviti sve korake koje trebamo i ručno napraviti prilikom objave, poput pisanja opisa i postavljanje slika. Napišemo svoj program koji će zapravo popuniti sve potrebne elemente i zatim objavljujemo na Google Play trgovinu (Martinez, 2016).

3.2.7. Kontinuirana integracija, dostavljanje i isporuka

Kontinuirana integracija (*eng. Continuous integration*) je praksa u programskom razvoju gdje članovi tima svakodnevno integriraju svoj rad u zajednički proizvod. Svaka integracija je verificirana od strane automatskih sustava koji izgrađuju i testiraju programski kod (Fowler, 2006). Možemo primijetiti da procese koje smo do sada definirali nam omogućuju implementiranje kontinuirane integracije. Ova praksa nam omogućuje puno brže i ranije pronalaženje problema koji smo dodali prilikom kreiranja novog programskog koda i da je nam mogućnost da ju ispravimo. U Android sustavu gotovo sve korake možemo ispuniti kroz Gradle sustav za automatsku izgradnju. Proces kontinuirane integracije završava uspješnim integriranjem programskog koda u zajednički repozitoriji.

Proces pakiranja spada u proces kontinuiranog dostavljanja (*eng. Continuous delivery*) koji se direktno nastavlja na proces kontinuirane integracije. Ova praksa se odnosi na automatizirani sustav koji nam omogućuje pripremu izvornog koda za isporuku. Potrebno je omogućiti da za dostavljanje aplikacije bude potreban jedan klik na sustavu. Oduzima čitavu kompleksnost dostavljanja i omogućuje svakom članu tima da pripremi aplikaciju za isporuku.

Kontinuirana isporuka (*eng. Continuous deployment*) je proces koji se nastavlja na kontinuirano dostavljanje. Pomoću ove prakse se svaka promjena automatski isporučuje korisniku bez ikakve intervencije zaposlenika. Na ovaj način više nemamo određen datum kad objavljujemo programski kod već to radimo svakodnevno. U ovaj proces bi djelomično spada alat za automatsku objavu na Google Play koji smo ranije spominjali, ali u našem slučaju mi nećemo napraviti da se sam automatski nastavlja na kontinuirano dostavljanje jer kod Android aplikacija ne želimo svakodnevnu objavu koda. Ne želimo ju jer Android korisnici moraju instalirati svako ažuriranje na svoj uređaj i dobiju notifikaciju za njega. Ako stalno objavljujemo nove verzije to može dovesti do lošeg korisničkog iskustva i do negativnog pogleda korisnika na našu aplikaciju (Pittet, n.d.).

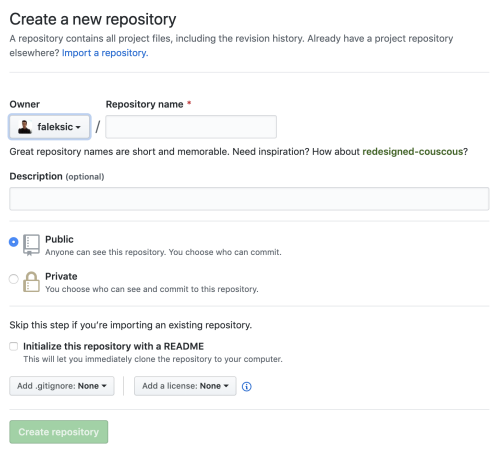
3.3. Primjena DevOps praksi i alata pri razvoju Android aplikacije

U ovom poglavlju ću prikazati korištenje navedenih praksi i alata u ranije spomenutim poglavljima. Napravit ću vrlo jednostavnu Android aplikaciju bez stvarne vrijednosti kojoj je svrha prikazati korištenje DevOpsa na primjeru.

3.3.1. Git

Odmah nakon kreiranja projekta postavljamo git repozitoriji i organiziramo grane prema ranije spomenutom Git Flowu. Repozitoriji držimo na GitHub platformi koja je git platforma u oblaku u kojoj možemo kreirati repozitorije i koristiti kao alat za kolaboraciju. Postoje još i mnoge druge poput BitBucket, GitLab, SourceForge, itd. GitHub sam odabrao jer ga najčešće koristim u svom radu pa mi je tako najjednostavnije. On je ujedno i najpopularniji.

Projekt kreiramo prema praznom predlošku koju pruža Android Studio (integrirana razvojna okolina koja služi za kreiranje Android aplikacija). Rezultat tog predloška je Android aplikacija koja se sastoji od jednog ekrana kojem se nalazi tekstualni okvir na sredini i u njemu piše Hello World. Zatim sam inicijalizirao git repozitoriji putem terminala. Važno je napomenuti da moramo prvo instalirati git na računalo. Git repozitoriji inicijaliziramo koristeći komandu `git init` dok smo pozicionirani u korijenskom direktoriju projekta. Zatim sam na GitHubu također kreirao prazan repozitoriji koristeći grafičko sučelje.



Slika 9: Inicijalizacija projekta na GitHub platformi

Na slici 9 možemo vidjeti kako izgleda forma za kreiranje novog projekta na GitHub platformi. Potrebno je odabrati vlasnika repozitorija (u ovom slučaju sam to ja, odnosno moj korisnički profil) i zatim upisati ime repozitorija. Ime može biti bilo koje dok se sastoji od engleske abecede ili brojeva, ne sadrži razmake i već ne postoji repozitoriji s takvim imenom na vlasnikovom profilu. Opis je opcionalan i u njemu opišemo svrhu repozitorija. Zatim moramo odabrati vidljivost repozitorija, možemo odabrati da je privatan (biramo tko ga može vidjeti) ili javan (svi ga mogu vidjeti). Posljednji korak preskačemo jer ćemo mi dodati postojeći repozitoriji u ovaj.

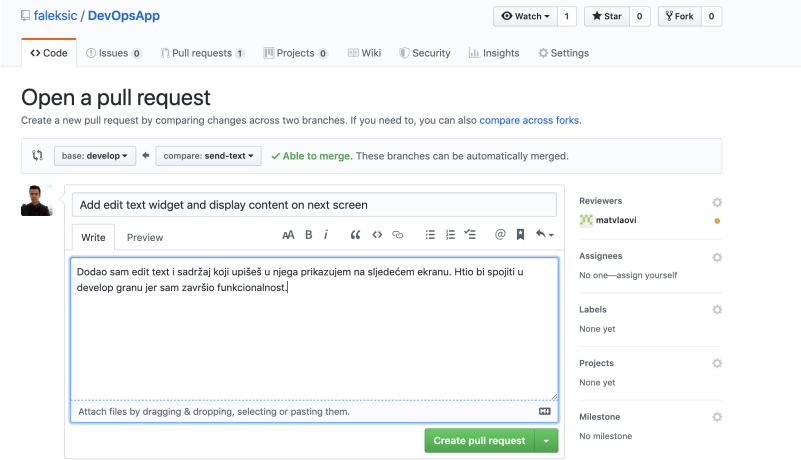
Zatim pomoću komande `git remote add origin git@github.com:faleksic/DevOpsApp.git` dodajemo lokaciju udaljenog repozitorija pod naziv origin (možemo imati više udaljenih repozitorija). Nakon toga dodajemo sve datoteke u projektu na pripremu za obvezivanje koda (*eng. commit*) pomoću komande `git add .`. Nakon toga obvezujemo se na dodane promjene i dokumentiramo ih s komandom `git commit -m "Prvi commit"`. Sada smo objavili kod u našem lokalnom repozi-

toriju i potrebno je napraviti objavu na udaljenom. Komanda koja to radi je `git push origin master`. U njoj kažemo pošalji promjene na udaljeni repozitoriji koji se zove origin na granu master (grana je kreirana automatski prilikom kreiranja repozitorija).

Sad kad smo kreirali lokalni i udaljeni repozitoriji možemo krenuti s organizacijom repozitorija prema Git Flowu. Rekli smo da uvijek imamo dvije grane, a to su master i develop. Master nam je automatski kreiran tokom inicijalizacije repozitorija, a develop kreiramo koristeći komandu `git checkout -b develop`. Pomoću ove komande ne samo da kreiramo granu već se i pozicioniramo na nju. Rekli smo da sve promjene trebaju biti na develop grani, a na masteru samo verzije koje su označene i puštamo u produkciju. Recimo sad da idemo dodati novu funkcionalnost gdje dodajemo gumb i tada otvaramo novi ekran. Prije početka pisanja koda kreiramo granu tipa feature na kojoj će se raditi sve promjene, komanda: `git checkout -b new-screen`. Nakon toga dodajemo sve promjene na tu granu i kad završimo potrebno je spojiti s develop granom i obrisati tu granu. Komande za izvođenje navedenih akcija možemo vidjeti u isječku koda ispod:

```
$ git checkout develop
$ git merge --no-ff new-screen
$ git branch -d new-screen
$ git push origin develop
```

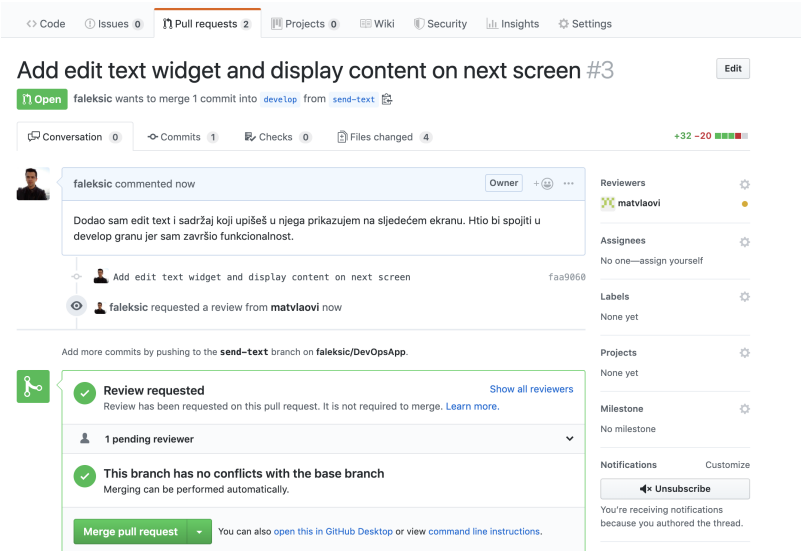
Prvo se pozicioniramo na develop granu i zatim ju spajamo s tom granom koristeći zastavicu koja kaže da želimo da se kreira commit prilikom spajanja programskog koda (ovim postižemo da imamo trag prijašnje grane u povijesti svih objava). Nakon što smo spojili grane možemo ju obrisati i to činimo pomoću treće komande. Za kraj sve to šaljemo na udaljeni repozitoriji kako bi ažurirali stanje. Ovaj pristup je uredu ako programski kod piše samo jedna osoba, ali ako programski kod piše više osoba tada bi bilo puno bolje implementirati praksu revizije koda. Sada ćemo kreirati novu granu, napraviti promjene, ali ovog puta nećemo automatski spajati već ćemo kreirati zahtjev za spajanje koji će netko trebati odobriti.



Slika 10: Kreiranje zahtjeva za spajanje koda na GitHubu

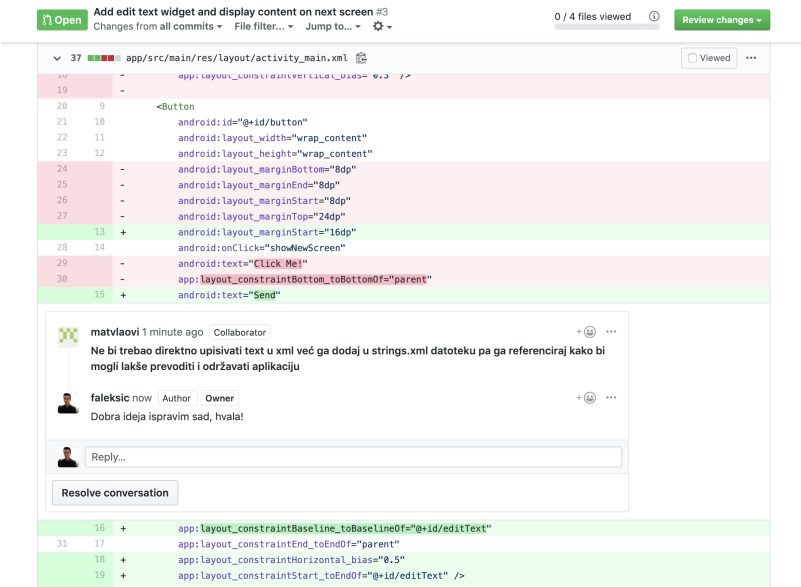
Slika 10 prikazuje izgled forme koja služi za kreiranja zahtjeva za spajanje unutar GitHub platforme. Kad odaberemo karticu zahtjev za spajanje (*eng. Pull Request*) i kliknemo gumb

novi zahtjev za spajanje (*eng. New Pull Request*) onda vidimo formu sa slike 10. Na njoj odabiremo koju granu želimo spojiti gdje, u našem slučaju granu send-text želimo spojiti u develop. Moramo napisati naslov i dodatni opis u kojem kažemo koji je razlog tog zahtjeva za spajanje i pod Reviewers dodajemo kolaboratora na projektu od kojeg želimo pregled. Kad smo definirali te opcije možemo kliknuti gumb ispod opisa s kojim kreiramo zahtjev za spajanje.



Slika 11: Kreirani zahtjev za spajanje koda

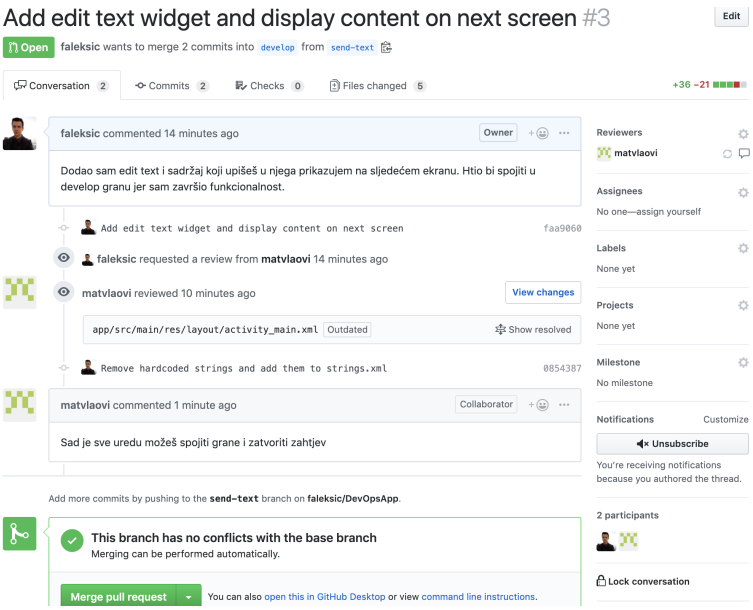
Na slici 11 vidimo kako izgleda zahtjev za spajanje nakon što je kreiran, vidimo naslov i opis, zatim ispod commit koji je dodan i da je zahtjevan pregled kolege. Sad čekamo na odgovor kolaboratora koji treba pregledati programski kod i dati odgovor. Može dati komentar odmah ispod koji se odnosi na čitavi zahtjev ili specifično komentirati liniju programskog koda.



Slika 12: Komentar recenzenta zahtjeva za spajanje

Slika 12 prikazuje komentar recenzenta gdje ukazuje na pogrešku na jednoj programskoj liniji u XML datoteci i daje prijedlog kako ju mogu ispraviti. Na ovaj način smo pronašli

pogrešku ranije i dogodila se razmjena znanja između zaposlenika na projektu.



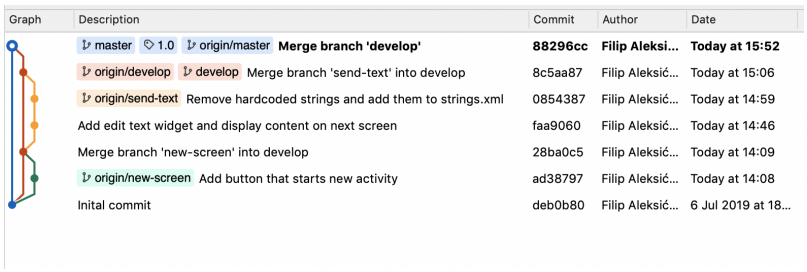
Slika 13: Komentar recenzenta zahtjeva za spajanje

Slika 13 prikazuje stanje nakon napravljenih promjena i objavljenih na toj grani. Recenzent odobrava promjene i potvrđuje spajanje grana. Grane možemo spojiti putem GitHub sučelja klikom na gumb ili na isti način kao u primjeru bez revizije programskog koda. Kad spojimo programski kod zahtjev se automatski zatvara i time taj proces završava.

Kreiranje hotfix i release grana izgleda jednako kao kreiranje feature grane samo što se hotfix kreira iz mastera. Kako bi prikazali jednu objavu koda sada ćemo trenutno stanje develop grane spojiti u master i označiti verzijom 1.0.

```
$ git checkout master
$ git merge --no-ff develop
$ git tag -a 1.0
$ git push origin master --follow-tags
```

Spajanje grana je zapravo jednak proces, samo u trećoj komandi označavamo objavu s oznakom 1.0 i pomoću zastavice –follow-tags objavljujemo oznaku i na udaljenom repozitoriju. Kroz GitHub sučelje pod karticom objave možemo vidjeti našu objavu s oznakom.



Slika 14: Grafički prikaz git grana

Slika 14 prikazuje grafički kako izgleda proces koji smo upravo napravili. Graf je prikazan u alatu `SourceTree` koji je grafički alat za rad s gitom. Ne moramo nužno koristiti

samo terminal već možemo i grafičke alate. Plava grana je master i ona se u početku odmah odvaja u develop (crvena boja) i new-screen (zelena boja) grane, zatim se new-screen spaja u develop i odvaja u send-text (narančasta boja) koja se na kraju isto spaja u develop i develop se spaja na kraju u master granu. Alat koji ćemo koristiti samo ovisi o preferenciji korisnika, ali svi oni zapravo na kraju šalju git naredbe.

3.3.2. Testiranje

Testiranje je jedna od najvažnijih tehnika za osiguranje kvalitete. Osim što nam daje brz način za provjeru programskog koda također nas prisiljava da pišemo programski kod koji je jednostavan za testiranje. U ovom poglavlju ćemo prikazati primjere nekih vrsta testova i opisati biblioteke koje koristimo kako bi si olakšali posao.

Započet ćemo s jediničnim testovima koji se mogu izvršavati na lokalnoj mašini bez stvarnog ili virtualnog Android uređaja. Jedinični testovi koji testiraju klase trebaju biti pisane koristeći `JUnit`. JUnit je najpopularniji i najrašireniji programski okvir za jedinično testiranje Java programskog koda. Kako bi ga mogli koristiti obavezno je u `build.gradle` datoteku dodati sljedeću liniju u `dependencies` blok `testImplementation 'junit:junit:4.12'`. Ako koristite Android Studio predložak za pisanje Android aplikacija tada je već on dodan tamo i postoji napisan jedan jedinični test kao primjer. ("Build Local Unit Tests | Android Developers", n.d.) Ne vezano za taj primjer prikazat ću vlastiti koji testira novu funkcionalnost aplikacije. Nova funkcionalnost se sastoji od tekstualnoga okvira u kojem možemo upisati bilo kakav tekst i klikom na gumb send pozivamo metodu klase StringManipulator koja ima metodu koja obrne tekst i napiše ga velikim slovima. Zatim se taj novi tekst šalje drugoj aktivnosti koja ga i prikaže.

Prvi test testira metodu za obrtanje teksta i povećavanje svih slova. Izgled mu je sljedeći:

```
public class StringManipulatorTest {
    @Test
    public void validateConvertToReverseUpperCase() {
        assertEquals("THIS␣IS␣A␣TEST", StringManipulator.convertToReverseUpperCase("
            tset␣a␣si␣siht"));
    }
}
```

Isječak koda iznad prikazuje vrlo jednostavan JUnit test. Započinje anotacijom `@Test` kojom označavamo testne metode i koristi metodu `assertEquals` koja testira jesu li 2 objekta ista. U ovom slučaju provjerava hoće li metoda StringManipulatora s ulazom "tset a si siht" vratiti rezultat "THIS IS A TEST". Ovo je vrlo jednostavan test koji nema nikakvih zavisnosti vezano za Android sustav i može biti pokrenut na bilo kojem uređaju koji ima Java virtualnu mašinu.

Sljedeći primjer testa provjerava hoće li se `NewScreenActivity` pokrenuti klikom na gumb Send. Ovaj test je ovisan o Android sustavu, ali umjesto da ga pokrećemo na stvarnom uređaju mi ćemo koristiti programski okvir `Robolectric` koji pokreće Android sustav unutar Java virtualne mašine tako da simulira ponašanje sustava i koristi lažne objekte. Prednost korištenja Robolectrica je u tome što ne moramo koristiti stvarni uređaj ili pokretati emula-

tor već možemo sve napraviti uz pomoć Java virtualne mašine. Također je izvođenje testova puno brže nego na stvarnom uređaju. ("Build Local Unit Tests | Android Developers", n.d.) Kako bi ga mogli koristiti trebamo dodati sljedeću liniju u `build.gradle` modul `app` pod blok `dependencies`:

`testImplementation 'org.robolectric:robolectric:4.3'`

```
@RunWith(RobolectricTestRunner.class)
public class MainActivityTest {
    private MainActivity activity;

    @Before
    public void setUp() throws Exception {
        activity = Robolectric.buildActivity(MainActivity.class)
                .create()
                .resume()
                .get();
    }

    @Test
    public void shouldNotBeNull() throws Exception {
        assertNotNull(activity);
    }

    @Test
    public void clickButtonAndTestStartingNextActivity() {
        Intent expectedIntent = new Intent(activity, NewScreenActivity.class);
        activity.findViewById(R.id.button).callOnClick();

        ShadowActivity shadowActivity = Shadows.shadowOf(activity);
        Intent actualIntent = shadowActivity.getNextStartedActivity();

        assertTrue(expectedIntent.filterEquals(actualIntent));
    }
}
```

Robolectric test možemo vidjeti u isječku koda iznad. Prvo je važno na početku označiti klasu anotacijom `@RunWith` i definirati da je potrebno pokretati s `RobolectricTestRunner` klasom. Zatim smo rekli da ćemo u ovom testu provjeravati MainActivity i imati interakciju s njenim sučeljem. Kako bi to mogli prvo ju trebamo nekako izgraditi i to radimo u metodi setUp koja je označena s `@Before` anotacijom koja kaže da pokrene tu metodu prije pokretanja testova. Zatim unutar metode u svojstvo `activity` pohranjujemo pokrenutu `MainActivity` aktivnost uz pomoć Robolectrica. Dodali smo i test `shouldNotBeNull` koji provjerava da aktivnost nije slučajno `null` pa da testovi ne bi padali zbog pogreške Robolectrica. Zatim dolazimo do testa koji ima interakciju s gumbom na kreiranoj aktivnosti i provjerava pokreće li se `NewScreenActivity` tada. Prva linija kreira `Intent` objekt koji očekujemo da će aktivnost kreirati. Sljedeća linija poziva `onClick` metodu gumba, zatim koristeći Robolectric dohvaćamo aktivnost i pomoću metode `getNextStartedActivity` dohvaćamo `Intent` objekt. na kraju ih uspoređujemo i na taj način provjeravamo pokreće li gumb aktivnost koju očekujemo.

Sljedeća vrsta testova su testovi koji zahtijevaju uređaj ili Android emulator kako

bi se mogli izvršavati. Prvi test koji ćemo prikazati je sličan prošlom jer također testira `MainActivity`, ali samo provjerava sadržaj `EditText` komponente. On ne koristi `Robolectric` već samo `JUnit`.

```
@RunWith(AndroidJUnit4.class)
public class MainActivityInstrumentedTest {

    private MainActivity activity;

    @Rule
    public ActivityTestRule< MainActivity > mActivityRule = new ActivityTestRule<>(
        MainActivity.class);

    @Before
    public void setUp() {
        activity = mActivityRule.getActivity();
    }

    @Test
    public void shouldNotBeNull() throws Exception {
        assertNotNull( activity );
    }

    @Test
    public void checkText() {
        EditText editText = activity.findViewById(R.id.editText);
        assertEquals("Enter␣Message", editText.getText().toString());
    }
}
```

Ponovno na početku koristimo @RunWith anotaciju, ali ovog puta s `AndroidJUnit4` klasom. Pomoću JUnit anotacije `@Rule` definiramo što se dogodi prije bilo kojeg testa ili `@Before` metoda. U ovom slučaju smo definirali da je potrebno pokrenuti `MainActivity`. U `setUp` metodi dohvatimo aktivnost u svojstvo `activity`. Test `shouldNotBeNull` provjerava isto kao u prijašnjem primjeru koda je li svojstvo`activity` različito od `null`. Test metoda `checkText` dohvaća `EditText` komponentu i provjerava je li joj sadržaj jednak vrijednosti `Enter Message`. Ako pokrenemo ovaj test Android Studio će odmah tražiti uređaj na kojem da pokrene test kao i prilikom pokretanja aplikacije tokom programiranja.

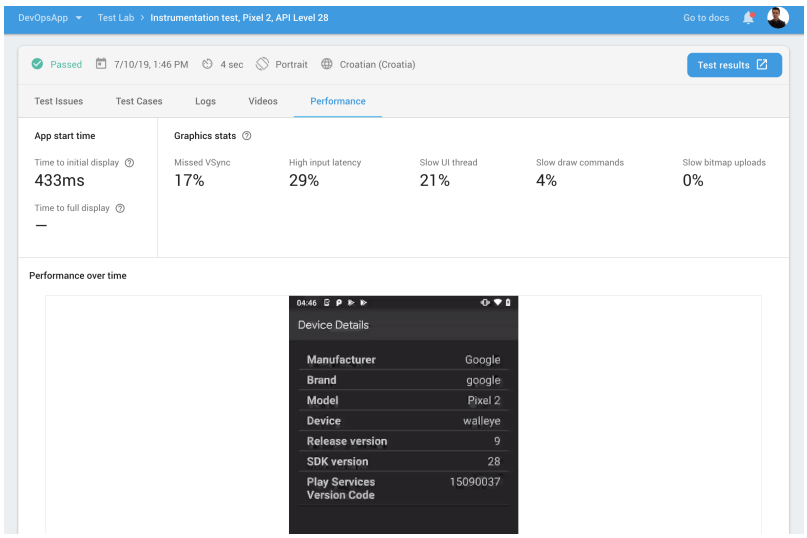
Posljednji primjer testa koristi `Espresso` programski okvir kako bi mogao raditi interakciju s aplikacijom i testirati korisničko sučelje. Ovaj tip testa bi spadao u kategoriju velikih testova i vremenski najduže traju. Test ne moramo nužno ručno pisati već je moguće uz pomoć Android Studia koristiti opciju snimanja Espresso testa i tada dobijemo sučelje koje automatski snima interakcije s aplikacijom i piše programski kod.

```
@RunWith(AndroidJUnit4.class)
public class ActivitesInstrumantedTest {

    @Rule
    public ActivityTestRule<MainActivity> mActivityTestRule = new ActivityTestRule
        <>(MainActivity.class);
```

```
    @Test
    public void activitesInstrumantedTest() {
        ViewInteraction appCompatEditText = onView(withId(R.id.editText));
        appCompatEditText.perform(click());
        appCompatEditText.perform(replaceText("Test"));

        ViewInteraction appCompatButton = onView(withId(R.id.button));
        appCompatButton.perform(click());

        ViewInteraction textView = onView(withId(R.id.textView));
        textView.check(matches(withText("TSET")));

    }
}
```

Na početku kao u prošlom testu pomoću `@Rule` anotacije pokrećemo `MainActivity`. Zatim u testu `activitesInstrumantedTest` prvo dohvaćamo `EditText` komponentu i pišemo unutra riječ Test. Zatim se pritišće gumb Send i dohvaća `TextView` te provjerava je li sadržaj TSET. Ovo je očekivano ponašanje jer aplikacija treba bilo kakav sadržaj iz početnog tekstualnog okvira obrnuti i postaviti sve na velika slova te prikazati na sljedećem ekranu.

Testovi koji zahtijevaju uređaj ne moraju nužno biti pokrenuti na našoj mašini. Postoje servisi koji nam omogućuju testiranje naše aplikacije na njihovim raznim uređajima s različitim verzijama, veličinama ekrana, uređajima, itd. Jedan takav servis je `Firebase Test Lab`. Omogućuje nam pokretanje naših vlastitih instrumentalnih testova na njihovim uređajima ili čak robo testove koji su njihovi testovi koji simuliraju akcije korisnika. Ovakav servis je vrlo koristan jer ga možemo povezati s Android Studiom ili Google Cloudom kako bi omogućili automatizirano izvođenje tih testova. Nakon što se testovi izvedu možemo vidjeti razne metrike na njihovom sustavu.



Slika 15: Rezultat testova pokrenutih na Firebase Test Labs

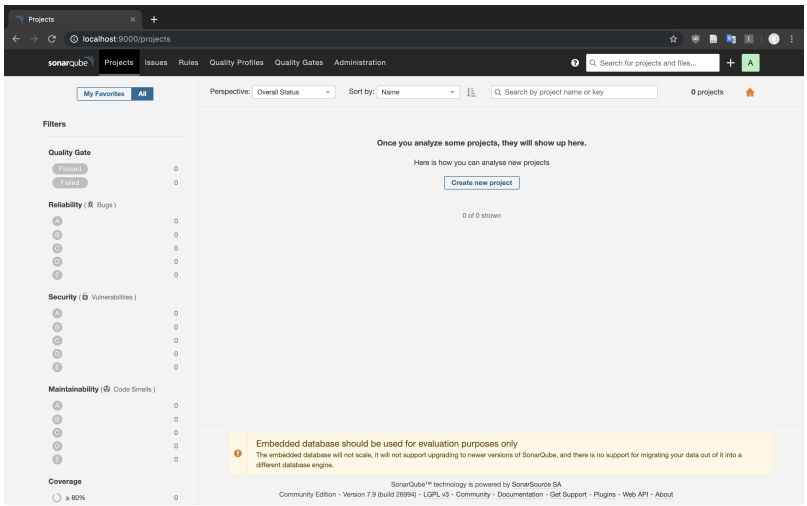
Na slici 15 možemo vidjeti kako izgleda pregled rezultata izvršavanje testova, specifično se prikazuje kartica performanse. Možemo vidjeti koliko je trebalo aplikaciji da se pokrene, koliko se usporavala dretva za korisničko sučelje, u kojem postotku se usporila komanda za

crtanje, itd. Osim toga možemo na ostalim karticama vidjeti video interakcije, čitavi dnevnik tokom izvršavanja, sve testne slučajeve i koji su prošli, a koji pali i koji problemi su se stvorili tokom izvođenja. Vrlo je korisno kako bi mogli vidjeti ponašanja na raznim uređajima bez da ih samostalno moramo kupiti.

3.3.3. Osiguranje kvalitete

U ovom dijelu ćemo prikazivati alate za osiguranje kvalitete, točnije Android Lint i SonarQube. Već smo opisali njihovu važnost i sada ćemo ih uključiti u stalni proces izgradnje pomoću Gradle alata.

SonarQube je alat za automatsku reviziju koda. Vrlo je koristan jer može raditi detaljne analize i sve probleme grafički opisati i objasniti. Koristit ćemo community verziju programa jer je besplatna i otvorenog koda. Na početku ga je potrebno preuzeti i zatim ga možemo pokrenuti. Nakon što smo preuzeli zip datoteku s njihove stranice možemo ju raspakirati, zatim otvoriti bin datoteku i verziju našeg sustava te zatim pokrenuti `sonar.sh` skriptu ako koristimo Linux ili Mac OS sustav s opcijom `console`. Tada ćemo pokrenuti SonarQube i putem web preglednika na lokaciji `http://localhost:9000` možemo vidjeti administratorsku ploču za pregled svih analiza i projekata nakon što se prijavimo s korisničkim podatcima admin / admin ("Documentation | SonarQube Docs", n.d.).



Slika 16: SonarQube administratorska ploča prilikom prvog pokretanja

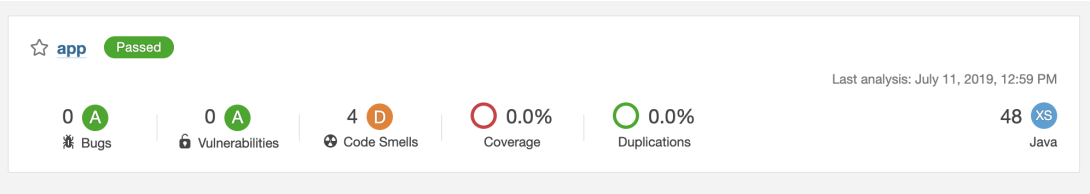
Slika 16 prikazuje kako izgleda nakon prve prijave na SonarQube. Možemo primijetiti da nemamo ni jedan projekt, možemo definirati pravila, profile kvalitete, vrata kvalitete i općenito administrirati alatom. Sad kad smo postavili SonarQube administrator potrebno ga je postaviti na našem Android projektu.

Prvo kako bi mogli uključiti SonarQube plugin u Gradle trebamo dodati u `build.gradle` blok `dependencies` na razini projekta sljedeću liniju `classpath "org.sonarsource.scanner.gradle:sonarqube-gradle-plugin:2.7.1"`. Zatim možemo u `build.gradle` u modulu `app` dodati liniju `apply plugin: 'org.sonarqube'`. Na taj način smo uključili SonarQube u naš projekt.

Možemo ga odmah izvršiti, ali mi ćemo napisati dodatna svojstva koja će mu omogućiti da bolje analizira naš projekt. Sljedeći isječak koda je potrebno dodati u `build.gradle` modul `app`.

```
sonarqube {
    properties {
        def libraries = project.android.sdkDirectory.getPath() + "/platforms/android
            -28/android.jar"/* + ", build/intermediates/exploded-aar*//**//*classes.
            jar"*/

        property "sonar.projectKey", "DevOpsApp"
        property "sonar.projectBaseDir", "."
        property "sonar.sources", "app/src/main/java"
        property "sonar.libraries", libraries
        property "sonar.java.test.libraries", libraries
        property "sonar.binaries", "build/intermediates/classes/debug"
        property "sonar.tests", "app/src/test/java,␣app/src/androidTest/java"
        property "sonar.java.test.binaries", "app/build/intermediates/classes/debug"
    }
}
```

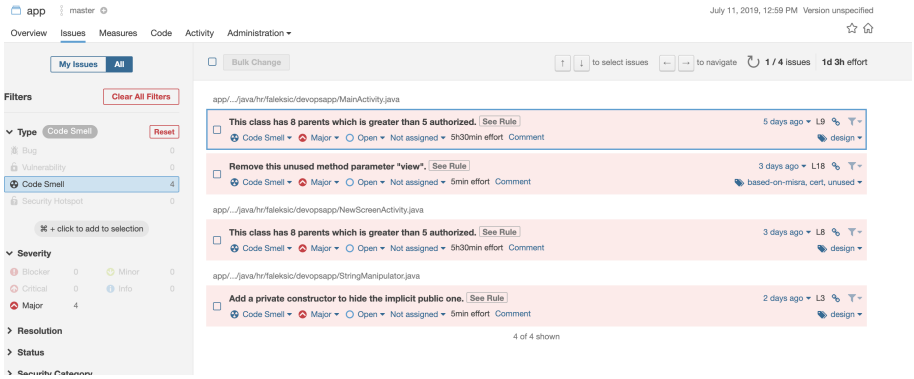
Možemo vidjeti da zapravo samo definiramo gdje nam se biblioteke nalaze, testovi, izvorni kod, itd. Lokacije ovih svojstava će biti različite ovisno o projektima, ali na uobičajenom projektu kreiranom prema Android Studio predlošku će izgledati kao iznad. Sada smo u potpunosti konfigurirali SonarQube i možemo krenuti s analizom projekta. Analizu možemo pokrenuti kroz Android Studio tako da odaberemo sonarqube s liste gradle zadataka ili kroz komandnu liniju tako da otvorimo korijenski direktoriji projekta i izvršimo `./gradlew sonarqube`. Nakon što se izvrši vratimo se na web preglednik gdje nam je otvorena SonarQube administratorska ploča. Među projektima možemo vidjeti projekt s nazivom app i glavne točke analize. Projekt je



Slika 17: SonarQube projekt

prošao vrata kvalitete odnosno zahtjeve kvalitete koje smo definirali. Na slici 17 možemo vidjeti da imamo nula pogrešaka i ranjivosti, ali imamo četiri potencijalno problematična programska koda. Klikom na naziv projekta otvaramo detalje analize i možemo vidjeti koji su to točno problematični isječci koda i zašto.

Slika 18 prikazuje listu problematičnih isječaka koda. U naslovu vidimo glavni opis problema, među detaljima vidimo u koju razinu problema spada, kolika je procjena vremena za ispravak i status. Gledanjem liste vidimo da su 2 problema ista samo se odnose na različite klase, a oni kažu da Activity klase imaju 8 roditelja što je više od 5 dozvoljenih. Ovo nije problem jer je Android sustav tako organiziran i mi nemamo utjecaja na to. Možemo označiti da nećemo rješavati taj problem ili u postavkama dozvoliti da klasa ima 8 roditelja. Sljedeći nam problem kaže da imamo parametar view na metodi `showNewScreen`, a nikad ga ne koristimo.

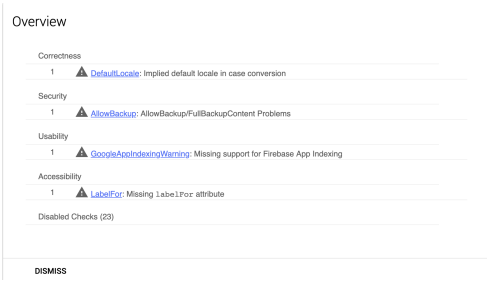


Slika 18: SonarQube problematični dijelovi koda

Ovo je inače dobar pokazatelj problema, ali Android zahtjeva da onClick metode imaju View objekt koji primaju inače neće funkcionirati. Znači ovo također možemo označiti da je riješeno i da ga nećemo popravljati. Posljednji problem je stvaran problem jer kaže da bi bilo bolje dodati privatni konstruktor na klasu `StringManipulator` kako ju netko ne bi mogao instancirati jer njena svrha je pozivanje statičke metode `convertToReverseUpperCase`. Nakon što dodamo privatni konstruktor možemo označiti kao riješeno i ponovno pokrenuti analizu. Sada možemo vidjeti da je sve kako treba i da smo riješili probleme.

Iz ovog jednostavnog prikaza rada SonarQube alata možemo vidjeti da je vrlo moćan i koristan jer nam pomaže da više razmišljamo o načinu na koji pišemo svoj programski kod i pomaže nam pronaći razne vrste problema. Kontinuiranim korištenjem ovog alata u našem razvoju možemo dići kvalitetu na visoku razinu.

Sljedeći alat za provjeru kvalitete je Android Lint koji provjerava je li kod strukturno ispravan i je li čitljiv. Alat je već dostupan unutar Android Studia i nije ga potrebno dodatno uključivati. Možemo ga pokrenuti odabirom iz liste Gradle zadatak Android Studia ili putem komandne linije izvršavajući naredbu `./gradlew lint` iz korijenskog direktorija projekta. Rezultat naredbe je broj problema koje je našao na debug i release verziji te lokacija gdje je napisao XML i HTML izvješća. Osobno preferiram HTML izvješća jer daju vizualno ugodan prikaz (“Improve Your Code with Lint Checks | Android Developers”, n.d.).



(a) Prikaz pogrešaka

(b) Detalji pogreške

Slika 19: Android Lint HTML izvještaj

Slika 19 nam prikazuje elemente HTML izvještaja. Pod a je prikaz svih pogrešaka s kratkim opisom i kategorijama problema u koje spadaju, a pod b prikaz detalja jedne od pogrešaka. Detaljni prikaz pogreške pokazuje točno na datoteku u kojoj se pogreška nalazi

i pod kojom linijom. Osim toga možemo vidjeti detaljan opis pogreške i ocjenu prioriteta te pogreške. Ako želimo još detaljnije objašnjenje možemo kliknuti na gumb objasni (*eng. explain*) i vidjeti potpuno objašnjenje s linkom na Android dokumentaciju gdje se nalazi najdetaljnije pojašnjenje, te oznaku pravila koje krši ta pogreška.

Nakon što smo vidjeli pogreške možemo ih riješiti ili označiti da ih lint ignorira. Definiranje toga što će lint ignorirati možemo koristeći: ("Improve Your Code with Lint Checks | Android Developers", n.d.)

- lint.xml datoteku : kreiramo datoteku `lint.xml` u korijenskom direktoriju i unutar nje definiramo koja pravila treba ignorirati
- Gradle konfiguraciju: na razini modula u `build.gradle` konfiguraciju unutar bloka `android` kreiramo blok `lintOptions` i tada definiramo koja pravila želimo ignorirati
- `@SuppressLint` anotaciju: koristimo ju iznad Java / Kotlin klasa ili metoda i unutar zagrada definiramo koja pravila je potrebno ignorirati
- XML atributa unutar resursa: dodajemo atribut `tools:ignore` unutar XML elementa i definiramo koje pravilo želimo ignorirati

Osim već definiranih pravila možemo kreirati naša vlastita. Potrebno je kreirati novi projekt koji će koristiti lint API kako bi definirali vlastita pravila. Nakon što kreiramo sve prema uputama potrebno je generirati jar datoteku koju ćemo smjestiti u `/.android/lint` putanju jer Android Studio automatski provjerava tu lokaciju za korisnički definirana pravila. Ovo je vrlo korisna mogućnost u slučaju ako definiramo nekakva pravila koja inače nisu uobičajena, a mi ih koristimo na našem projektu. Primjerice ako svakom XML elementu dajemo identifikator s nekim prefiksom kako bi znali da je to naš element. Ako kreiramo vlastito lint pravilo tada možemo biti sigurni da će sve članove tima upozoriti u slučaju korištenja pogrešnog nazivlja ("Writing Custom Lint Rules - Android Studio Project Site", n.d.).

3.3.4. Gradle, pakiranje i izgradnja

Gradle je vrlo važan alat za Android razvoj. Već smo ga spominjali kroz kontekst drugih alata i možemo vidjeti da pomoću njega alate uključujemo u projekt, konfiguriramo i pokrećemo. U ovom poglavlju ćemo opisati kako imati više tipova izgradnje, više okusa proizvoda, kako izgraditi, minimizirati i potpisati aplikaciju.

Gradle tipove izgradnje (*build types*) koristimo u gotovo svakom Android projektu bez da to možda i znamo jer nam Android Studio sam kreira na predložak dva tipa izgradnje, release (verzija koju bi objavili odnosno isporučili klijentu) i debug (verzija za testiranje). Mi možemo modificirati postojeće ili kreirati nove kroz `build.gradle` datoteku u modulu `app`. Potrebno je u blok `android` dodati blok `buildTypes`.

```
android {
    ...
    buildTypes {
```

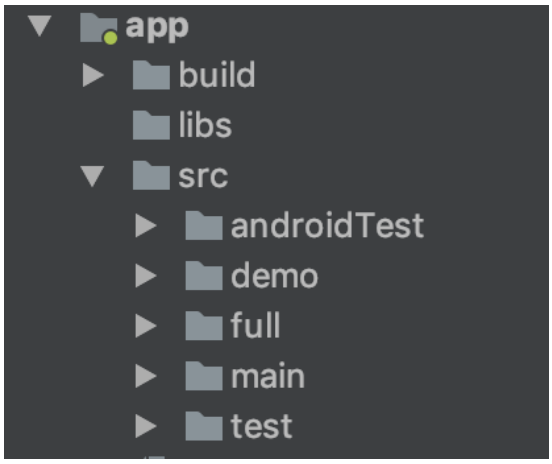
```
        release {
            debuggable false
            minifyEnabled true
            proguardFiles getDefaultProguardFile('proguard-android.txt'), 'proguard-
                rules.pro'
        }

        debug {
            applicationIdSuffix ".debug"
            debuggable true
        }
    }
    ...
}
```

U isječku koda iznad možemo vidjeti da nazive tipova kreiramo kao blokove i unutar njih njihova svojstva. Release tip ima isključenu opciju `debuggable` jer ne želimo da se generirana aplikacija izgradi u modu koji služi za razvojne programere. Također ima uključenu opciju `minifyEnabled` i definirane proguard datoteke jer želimo minimizirati i obfuscirati kod o čemu ćemo pričati malo kasnije. Debug verzija ima definiran sufiks na id aplikacije kako bi mogli lakše prepoznati o kojoj se verziji radi i `debuggable` mod joj je uključen. Osim tipova možemo definirati i okuse proizvoda (*eng. product flavors*).

```
android {
    ...
    flavorDimensions "myflavor"
    productFlavors {
        demo {
            applicationIdSuffix ".demo"
            versionNameSuffix "-demo"
        }
        full {
            applicationIdSuffix ".full"
            versionNameSuffix "-full"
        }
    }
    ...
}
```

Okuse proizvoda definiramo na istom mjestu kao i tipove samo je prvo potrebno dodati `flavorDimensions` jer svi okusi moraju pripadati nekoj dimenziji. U ovom slučaju mi imamo samo jednu dimenziju te zbog toga ne moramo na okusima specificirati dimenzije, ali da imamo morali bi dodati svojstvo `dimension` s nazivom jedne od definiranih dimenzija. U slučaju da nije definirano Gradle bi izbacivao pogrešku. U prikazanom primjeru smo kreirali okuse demo (verzija za prezentaciju proizvoda, djelomična funkcionalnost) i full (aplikacija sa svim mogućnostima). Ovo je čest slučaj korištenja kad koristimo demo verziju kako bi korisnici mogli besplatno skinuti našu aplikaciju i isprobati je, te ih potaknuti da kupe potpunu verziju aplikacije. Od svojstava definirali smo sufikse za id aplikacije i ime verzije. Kako bi imali različite funkcionalnosti u različitim okusima još smo u projektu kreirali direktorije za izvorni kod čiji su nazivi jednaki onima od okusa. Znači osim `main` izvornog direktorija koji sadrži sav naš

programski kod sada imamo i `demo` i `full`.



Slika 20: Android struktura s dodanim okusima proizvoda

Na slici 20 ih možemo vidjeti u `app` modulu unutar `src` direktorija. U sebi sadrže `java` direktoriji u koji stavljamo naš izvorni kod. Kao primjer sam kreirao da demo verzija u klasi `StringManipulator` samo pretvori u velika slova, dok puna verzija obrne i postavi velika slova. Način na koji sam to napravio je taj da sam klasu `StringManipulator` maknuo iz `main` direktorija i stavio u `demo` i `full`. Svaki od okusa ima svoju implementaciju kao što sam opisao iznad i prilikom pokretanja različitih okusa se izvršava različiti programski kod.

Kako bi pokretali različite verzije programa moramo odabrati varijantu izgradnje *eng. build variant*. Kroz Android Studio to možemo napraviti tako da u alatnoj traci odaberemo `Build` i zatim `Select Build Variant`. Tako ćemo dobiti dodatni prozor s padajućim izbornikom gdje možemo odabrati jednu od kombinacija tipa i okusa (demoDebug, demoRelease, fullDebug i fullRelease). Koristeći terminal možemo samo u komandi napisati dodatno koju verziju želimo recimo umjesto `./gradlew assemble` bi koristili `./gradlew assembleDemoDebug`.

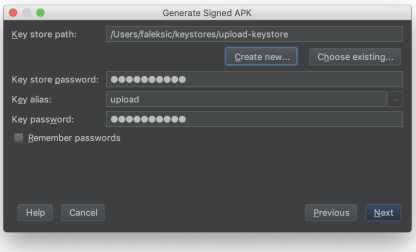
Osim navedenih promjena možemo konfigurirati i mijenjati jako puno elemenata. Možemo mijenjati pomoću kojeg ključa potpisujemo aplikaciju, kako ikona aplikacije izgleda, koje resurse koristi i još mnogo toga.

Nakon što smo razvili našu aplikaciju i odlučimo ju isporučiti potrebno je maksimalno ju optimizirati, minimizirati i obfuskirati. Optimizacija je važna jer želimo da korisnici imaju što bolje iskustvo koristeći našu aplikaciju tako da je brzina vrlo važna. Minimizacija je važna jer želimo što manje memorijskog prostora zauzimati korisniku kako bi on imao mjesta za našu aplikaciju i kako bi ju mogao brzo skinuti s Google Play trgovine. Obfuskacija koda otežava reverzni inženjering programskog koda pa time barem otežavamo proces krađe našeg programskog koda.

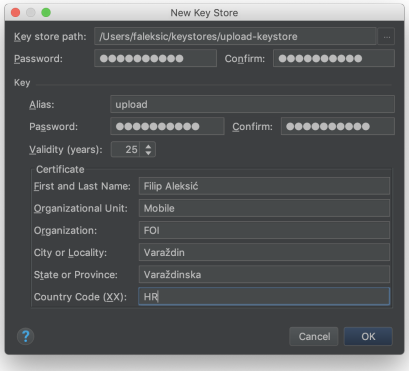
Navedene optimizacije možemo vrlo jednostavno napraviti tako da ih uključimo u `build.gradle` datoteku u modulu `app`. Već smo u isječku koda za prikaz tipova izgradnje pokazali opciju `minifyEnabled` koja je postavljena na `true`. Uključivanjem te opcije uključujemo minimizaciju, optimizaciju i obfuskaciju programskom koda. Dodavanjem opcije `shrinkResources` i postavljanjem na `true` uključujemo smanjivanje resursa, odnosno mičemo ne iskorištene resurse. Na kraju uključujemo `ProGuard` pravila uz pomoć linije `proguardFiles` koju smo također vidjeli u izresku koda koji je prikazivao tipove izgradnje. `ProGuard` je program koji minimizira, optimizira i obfuskira programski kod. Android Studio ga koristi kako bi napravio taj zadatak, iako ga trenutno mijenjaju s njihovim vlastitim programom R8. U svakom slučaju ProGuard i R8 možemo konfigurirati pomoću pravila,

česti razlog konfiguracije je ako želimo sačuvati neki dio programskog koda bez optimizacije tada to definiramo u pravilima. Zbog toga je važno definirati pravila i uključiti ih u Gradle konfiguraciji jer nam se inače može dogoditi da nam proces minimizacije obriše neki komad programskog koda koji nam je važan i aplikacija prestane raditi. Nakon obfuskacije nam se generiraju datoteke koje mapiraju naš trenutni kod u obfuscirani. Nalaze se na putanji `<imeModula>/build/outputs/mapping/<tipIzgradnje>/`. Kad objavimo kod na Google Play onda možemo objaviti i `mapping.txt` datoteku s navedene putanje i tad će nam Google Play konzola automatski dekodirati kod u čitljivi. Ako želimo to sami učiniti možemo koristiti `ReTrace` skriptu koja dođe zajedno s `ProGuard` alatom ("Shrink, Obfuscate, and Optimize Your App", n.d.).

Sad kad smo optimizirali aplikaciju možemo ju potpisati za objavu, kako bi ju mogli potpisati trebamo imati pohranu ključeva *eng. keystore* i ključ *eng. key*. Možemo ih generirati uz pomoć Android Studia.



(a) Generiranje potpisane aplikacije



(b) Kreiranje spremnika ključeva

Slika 21: Potpisivanje Android aplikacije

Na slici 21 vidimo kako izgleda dijalog za potpisivanje aplikacije. Pod a prikazujemo kako postavljamo spremnik ključeva pomoću kojeg želimo potpisati aplikaciju, a pod b možemo vidjeti prozor koji se prikaže klikom na kreiraj novi spremnik ključeva (*eng. Create new*. Potrebno je definirati putanju spremnika ključeva i njegovu lozinku, te alias za ključ, njegovu lozinku i još neke dodatne podatke. Nakon toga možemo generirati potpisanu aplikaciju. Ovaj proces možemo automatizirati tako da u Gradle dodamo opcije za potpisivanje tako da ih ne moramo ručno namještati.

```
android {
    signingConfigs {
        upload {
            keyAlias 'upload'
            keyPassword 'definiranaSifraKljuca'
            storeFile file('/Users/faleksic/keystores/upload-keystore')
            storePassword 'definiranaSifraSpremnika'
        }
    }
    ...
}
```

Na isječku koda iznad možemo vidjeti kako je potrebno postaviti opcije za potpisivanje u `build.gradle` datoteku u modulu `app`. Možemo postaviti različite ključeve ovisno o verzijama tako da unutar njihovih blokova postavimo prikazane postavke. Možemo namjestiti postavke tako da Google Play obavlja potpisivanje aplikacija, a možemo i sami raditi ovaj proces. Ono što je vrlo važno je ne izgubiti spremnik ključeva jer tada više nećemo moći objaviti ažuriranje pod istu aplikaciju već ćemo morati raditi novu objavu u potpunosti. Također je važno držati ključ na sigurnom jer ako nam ga netko ukrade tada on može objaviti bilo koju aplikaciju pod našu jer je potpisao našim ključem ("Sign Your App", n.d.).

3.3.5. Fastlane

`Fastlane` alata koji služi za olakšavanje i automatiziranje mnogo različitih zadataka s kojima se susrećemo prilikom razvoja Android aplikacija. Zadatke poput generiranje slika zaslona aplikacije, objava na Google Play, objava testnim korisnicima, itd. Uštedi nam vrijeme u satima prilikom svake objave koda. Potpuno je besplatan i otvorenog koda te je integriran sa jako mnogo alata i servisa. Vrlo lako ga je instalirati i spojiti s nekim alatom za kontinuirano integriranje što ćemo mi kasnije i napraviti. Kako bi ga postavili možemo koristiti `RubyGems` (`sudo gem install fastlane -NV`) ili `Homebrew` (`brew cask install fastlane`) na MacOS sustavu. Nakon što smo ga instalirali potrebno je izvršiti komandu `fastlane init` u korijenskom direktoriju projekta. Tada će se pokrenuti program za postavljanje i pitat će nas neke podatke za konfiguraciju koje možemo mijenjati i kasnije. Kada se program izvrši kreirat će nam poseban direktoriji `fastlane` s datotekama `Appfile` (globalna datoteka s konfiguracijama) i `Fastfile` (datoteka koja sadrži staze (*eng. lanes* koje definiraju ponašanje fastlanea, ovo je ujedno najvažnija datoteka) ("Setup - Fastlane Docs", n.d.).

Nakon što smo pripremili aplikaciju za objavu trebamo pripremiti i snimke zaslona naše aplikacije. Snimke zaslona su vrlo važan dio objave aplikacije jer su one prve vidljive na trgovini kad potencijalni korisnik ide pogledati informacije o aplikaciji. Jako mali dio korisnika čita opise, ali gotovo svi gledaju slike. Iako je važan ovaj proces može biti dosta vremenski zahtjevan, pogotovo ako podržavamo više različitih jezika. Zbog toga je pametno automatizirati taj proces. Već smo ranije spomenuli da ćemo koristiti `screengrab` alat koji generira snimke zaslona za različite jezike i tipove ekrana. Kreira ih koristeći emulator ili stvarni uređaj. Jednom kad se konfigurira bilo tko u timu ga može pokrenuti kako bi generirao nove slike. Uz njega je moguće generirati slike prilikom svake promjene sučelja kako bi prezentacija u trgovini što bolje prikazivala stvarno stanje aplikacije ("Screengrab - Fastlane Docs", n.d.).

Koristimo ga tako da ga kombiniramo s `Espresso` testovima koji se izvode na uređaju ili emulatoru. Ovakve testove smo već obradili u poglavlju testiranje tako da će `screengrab` biti samo malo proširenje. Potrebno je prvo na računalo instalirati alat tako da izvršimo naredbu `sudo gem install screengrab`. Nakon što smo ju instalirali potrebno je dodati zavisnosti u `build.gradle` modul `app`: `androidTestImplementation 'tools.fastlane:screengrab:1.0.0'`. Zatim je potrebno dodati posebne dozvole u `src/debug/AndroidManifest.xml`.

```
<uses-permission android:name="android.permission.DISABLE_KEYGUARD"/>
```

```
<uses-permission android:name="android.permission.WAKE_LOCK"/>
<uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE" />
<uses-permission android:name="android.permission.READ_EXTERNAL_STORAGE" />
<uses-permission android:name="android.permission.CHANGE_CONFIGURATION" />
```

Izrezak koda iznad prikazuje potrebne dozvole. Redom one služe za otključavanje uređaja, aktiviranje ekrana, zapisivanje snimki zaslona u memoriju, dohvaćanje snimki zaslona iz memorije i promjenu jezika. U našem primjeru nisam dao posljednju dozvolu jer je naša aplikacija samo na jednom jeziku. Kako bi uslikali sliku potrebno se pomoću `Espresso API-a` postaviti na željenu lokaciju i pozvati `Screengrab.screenshot("naziv_slike");`.

```
@Test
   public void testTakeScreenshot() {
       Screengrab.screenshot("main_activity_no_change");
       onView(withId(R.id.editText)).perform(click()).perform(replaceText("Test"));
       Screengrab.screenshot("main_activity_with_change");
       onView(withId(R.id.button)).perform(click());
       Screengrab.screenshot("new_screen_activity");
   }
```

Izrezak iznad prikazuje primjer testa koji prolazi kroz čitavu našu aplikaciju i slika snimke zaslona. Na početku slika `MainActivity`, zatim upisuje riječ Test u tekstualni okvir i slika novu sliku i za kraj pritišće gumb i slika `NewScreenActivity`. Nakon što izgradimo `debug` i `test` verziju aplikacije možemo pozvati naredbu `fastlane screengrab` u konzoli. Slike zaslona će se spremiti na putanji `fastlane/metadata/android`.

`Fastlane` nam omogućava kreiranje već ranije spomenutih staza unutar kojih možemo izvoditi razne zadatke. Vrlo je korisno što možemo pozivati gradle zadatke uz pomoć njega pa ćemo kreirati staze za pokretanje testove. Prvo ćemo dodati varijable kroz koje možemo konfigurirati tip izgradnje i okus proizvoda.

```
before_all do|lane, options|
   @build_type = "Debug"
   @build_flavor = "Full"
   if not options.empty?
     @build_type = options[:release] ? "Release" : "Debug"
     @build_flavor = options[:demo] ? "Demo" : "Full"
   end
  end
```

Izrezak koda iznad prikazuje definiranje varijabli `build_type` i `build_flavor` koje imaju predefinirane vrijednosti `"Debug"` i `"Full"` respektivno. Programski kod se nalazi unutar bloka `before_all` koji se kako mu ime kaže izvršava prije svih ostalih blokova. Ako su nam postavljene opcije `release` ili `demo` tada provjeravamo jesu li `true` ili `false` i ovisno o njihovim vrijednostima ih postavljamo pomoću ternarnog operatora. Ovo nam omogućava biranje okusa i tipova pomoću opcija, jer bi inače morali pisati staze za svaku mogućnost.

```
desc "Run␣unit␣tests"
lane :unit_tests do |options|
  gradle(task: "test", build_type: @build_type, flavor: @build_flavor)
end
```

Primjer staze prikazan iznad je jedan od najjednostavnijih staza u `Fastfile` datoteci. Zapravo poziva `gradle` zadatak `test` s varijablama koje smo maloprije prikazali. Nakon što smo definirali stazu pozovemo ju tako da napišemo naredbu `fastfile` pa zatim naziv staze u ovom slučaju bi to bilo `fastfile unit_test`. Sve takve naredbe moraju biti pozvane iz korijenskog direktorija projekta.

Razlog kreiranja staza je kasnija integracija sa servisom za kontinuiranu integraciju koji će moći sam pozivati staze i izvršavati zadatke. Cilj nam je izvršavati sve zadatke koristeći udaljene servise tako da se čitavi proces automatski odvija na udaljenoj infrastrukturi. Donošenjem te odluke se suočavamo s problemom kako izvršavati testove koji zahtijevaju uređaj ili emulator na udaljenom servisu. Već smo u poglavlju testiranje spomenuli servis koji se zove `Firebase Test Labs` kojemu je svrha upravo udaljeno izvršavanje testova na uređaju. Izvršava testove na bilo kojem uređaju, s bilo kojom verzijom Android sustava. Fastlane ima dodatak koji možemo instalirati kako bi kreirali stazu koja će izvršavati testove koristeći Firebase Test Labs. Prije nego što možemo pozivati servis prvo trebamo servisni račun od Googlea. Prvo je potrebno kreirati Firebase projekt ako već nismo, otići na `Google Cloud Console` i odabrati naš projekt, zatim možemo kreirati servisni račun i potrebno je dodijeliti mu ulogu vlasnik projekta (*eng. Project Owner*. Nakon što imamo račun kreiramo JSON ključ koji ćemo koristiti za autentifikaciju prilikom poziva servisa. Sad kad imamo ključ ćemo kreirati `.env` datoteku u korijenskom direktoriju projekta i definirati ključ s nazivom `GCLOUD_SERVICE_KEY` te pod njegovu vrijednost postaviti sadržaj JSON datoteke. Za kraj postavljanja je potrebno instalirati dodatak izvršavajući naredbu `fastlane add_plugin run_tests_firebase_testlab` i tada možemo napisati stazu. (Correia, 2018a).

```
desc "Run␣instrumentation␣tests␣in␣Firebase␣Test␣Lab"
lane :instrumentation_tests_testlab do
  assemble release:false, demo:false
  run_tests_firebase_testlab(
    project_id: "devopsapp-d45e5",
    devices: [
        {
          model: "Nexus6P",
          version: "27"
        }
    ],
    app_apk: "app/build/outputs/apk/full/debug/app-full-debug.apk",
    android_test_apk: "app/build/outputs/apk/androidTest/full/debug/app-full-debug-
        androidTest.apk",
    delete_firebase_files: true
  )
end
```

U stazi prikazanoj iznad prvo pozivamo stazu `assemble` s varijablama `release` i `demo` postavljenim na `false` što će zapravo izgraditi aplikaciju i testove u `debug` tipu i `full` okusu. Zatim pozivamo funkciju našeg dodatka `run_tests_firebase_testlab` sa sljedećim argumentima: naziv projekta, uređaj na kojem želimo izvršavati, putanja do aplikacije, putanja do testne aplikacije i da želimo obrisati firebase podatke nakon izvršavanja. U argument `devices` možemo definirati i više uređaja iako smo mi samo jedan. Izvršavanjem ove staze

će se izgraditi debug full verzija aplikacije i testa te početi izvršavati testiranje na Firebase Test Labs servisu na uređaju Nexus6P s verzijom sustava 27. Nakon što se zadatak izvrši obrisat će se generirani podatci (Correia, 2018a).

Često prije objave aplikacije na trgovinu želimo objaviti aplikaciju samo određenoj skupini korisnika koju mogu testirati neko vrijeme aplikaciju gdje ju mi nadziremo i provjeravamo da nije pošlo nešto po krivu. Članovi te skupine znaju da testiraju aplikaciju i ne očekuju u potpunosti stabilnu aplikaciju. Ovo je vrlo dobar način da dobijemo povratnu informaciju kako se aplikacija ponaša kad ju stvarni korisnici koriste na stvarnim uređajima. Postoji nekoliko servisa koji služe kako bi olakšali objavu aplikacije testnoj skupini, ali mi ćemo koristit `Crashlytics Beta` jer su također besplatni i imaju mnogo korisnih opcija. Također postoji dodatak za fastlane koji nam omogućava da kreiramo stazu koja će automatski objavljivati aplikaciju na Crashlytics.

Postavimo ga u našu aplikaciju tako da prvo dodamo zavisnosti u `build.gradle` modul `app`.

```
buildscript {
  repositories {
    maven { url 'https://maven.fabric.io/public' }
    ...
  }
  dependencies {
    classpath 'io.fabric.tools:gradle:1.+'
    ...
  }
}
apply plugin: 'io.fabric'

repositories {
  maven { url 'https://maven.fabric.io/public' }
  ...
}
dependencies {
  compile('com.crashlytics.sdk.android:crashlytics:2.10.1@aar') {
    transitive = true;
  }
  ...
}
```

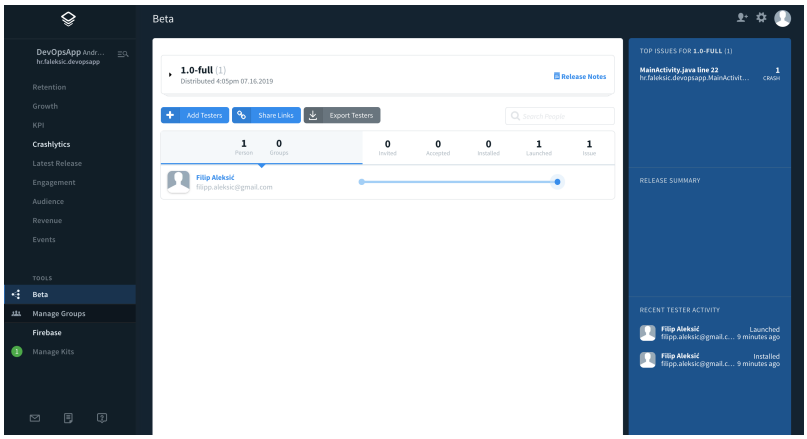
U primjeru iznad možemo vidjeti koje su to linije koje je potrebno dodati. Nakon što su dodane potrebno je dodati API ključ i dodati dozvolu za korištenje interneta u `AndroidManifest.xml`.

```
<application>
...
  <meta-data
      android:name="io.fabric.ApiKey"
      android:value="api_key"
  />
</application>
<uses-permission android:name="android.permission.INTERNET" />
```

Primjer iznad prikazuje koje elemente je potrebno dodati. Za kraj je potrebno inicijalizirati ga u početnoj aktivnosti prilikom kreiranja koristeći liniju `Fabric.with(this, new Crashlytics());`. Kad smo napravili sve navedene korake možemo pokrenuti aplikaciju i tada ćemo ju vidjeti na Crashlytics administratorskoj ploči. Nakon što smo postavili aplikaciju na servisu sada možemo napisati stazu za objavu aplikacije. Prvo je potrebno u `.env` datoteku dodati dvije nove varijable `CRASHLYTICS_API_TOKEN` i `CRASHLYTICS_BUILD_SECRET` s odgovorajućim ključevima koje možemo preuzeti s administratorske ploče pod organizacijskim postavkama. Sada je potrebno samo izgraditi aplikaciju i objaviti ju ("Crashlytics for Android - Fabric Install", n.d.).

```
desc "Submit␣a␣new␣Beta␣Build␣to␣Crashlytics"
lane :beta do
  assemble release:true, demo:false
  crashlytics
end
```

Izgradili smo full release verziju aplikacije i pozvali crashlytics dodatak. Nije ga potrebno instalirati jer je automatski uključen u fastlane. Izvršavanjem staze možemo vidjeti novu verziju u administratorskoj ploči pod Beta odabirom u izborniku.



Slika 22: Crashlytics Beta administratorska ploča

Na slici 22 možemo vidjeti Crashlytics Beta sučelje. Na vrhu je prikaz posljednje objavljenje verzije i ispod su testeri koji su dodani u skupinu. U desnom stupcu možemo vidjeti greške koje su se pojavile i nedavne aktivnosti testera. Za svakog testera možemo vidjeti uređaj koji koristi i kada je koristio aplikaciju. Alat je vrlo koristan jer na ovaj način dobivamo odličnu povratnu informaciju od strane testera umjesto da pružamo loše iskustvo našim stvarnim korisnicima.

3.3.6. CircleCI

CircleCI je sustav za kontinuiranu integraciju, dostavljanje i isporuku. Ono što on radi je integrira se sa sustavom za verzioniranje i prilikom svake objave koda kreira izgradnju. Izgradnja se izvršava u novom čistom `Docker` kontejneru gdje izvršava testove, izgradnju i sve ostalo što definiramo. Sustav obavještava tim ako izgradnja nije uspjela kako bi mogli brzo ispraviti

probleme. Uspješne izgradnje je moguće postaviti da se automatski postavljaju u produkciju. Koristit ćemo besplatan plan kako bi konfigurirali našu izgradnju. Prvo je potrebno prijaviti se na sustav uz pomoć Githuba, zatim odabrati repozitoriji koji želimo koristiti i kliknuti započni izgradnju (*eng. Start building*). Nakon toga je potrebno otići u korijenski direktoriji našeg projekta i kreirati direktoriji `.circleci` i unutar njega kreirati konfiguracijsku datoteku `config.yml`.

```
version: 2
references:
  # ...
jobs:
  test:
    # ...
workflows:
  version: 2
  workflow:
    jobs:
      # ...
```

Lijevo možemo vidjeti izrezak koda koji prikazuje sve glavne elemente CircleCI konfiguracije. Na početku je definirana verzija CircleCI-a koju koristimo, zatim definiramo reference koje se ponašaju zapravo kao varijable koje kasnije možemo pozivati, poslovi (*eng. jobs*) su kolekcija koraka koji definiraju kako napraviti nešto, tijek posla (*eng. workflows*) definira koji se poslovi mogu izvršavati paralelno, koji poslovi imaju preduvjete, itd. Konfiguracija koje ćemo mi kreirati će pozivati fastlane staze i izvršavati ih. Prvo ćemo kreirati poslove, zatim prikazati neke reference koje su koristili i zatim prikazati tokove posla (Correia, 2018b).

Svaki posao se sastoji od konfiguracije koja definira na čemu će se taj posao izvršavati i korake od kojih se taj posao sastoji. U nastavku ćemo prikazati kako izgleda posao koji izvršava jedinične testove.

```
test_unit:
  <<: *android_config
  steps:
    - checkout
    - *restore_gradle_cache
    - *restore_gems_cache
    - *ruby_dependencies
    - *android_dependencies
    - *save_gradle_cache
    - *save_gems_cache
    - run:
        name: Run unit tests
        command: bundle exec fastlane unit_tests
    - store_artifacts:
        path: app/build/reports/
        destination: /reports/
    - store_test_results:
        path: app/build/test-results/
        destination: /test-results/
```

Svaki dio konfiguracije koji započinje sa zvjezdicom predstavlja referencu, znači svaka od tih varijabli mora biti definirana među referencama. U isječku koda iznad možemo vidjeti da kreiramo posao koji se zove `test_unit` s konfiguracijom koja se referencira na `android_config` varijablu. Zatim nabrajamo korake u poslu, `checkout` označava preuzimanje izvornog koda u direktoriji posla, zatim imamo 2 koraka gdje preuzimamo zavisnosti za gradle i gems iz `cache` memorije ako ima dostupno, sljedeća 2 koraka preuzimaju gradle i gems zavisnosti koje nedostaju, zatim spremamo sve dobavljene zavisnosti u `cache` memoriju. Svi do sad navedeni koraci osim prvog su definirani u referencama. Zatim imamo korak

koji poziva fastlane stazu `unit_tests` koja poziva gradle zadatak za izvršavanje jediničnih zadataka. Na kraju spremamo izvještaje i rezultate u CircleCI. Osim ovog posla kreiramo poslove i za pozivanje `Firebase Test Labs` i `Crashlytics Beta` servisa. Njih nećemo prikazivati ovdje jer su vrlo slični navedenom pa nema razloga za detaljno razlaganje te ih čitatelj može pogledati u repozitoriju prikazanom na kraju rada.

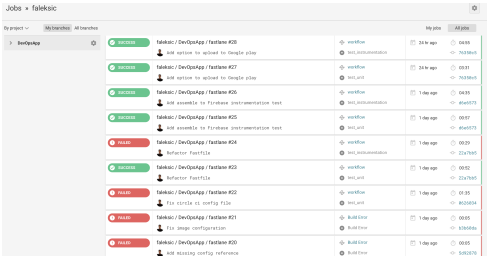
```
workspace: &workspace
    ~/src

android_config: &android_config
    working_directory: *workspace
    docker:
      - image: circleci/android:api-28
    environment:
      TERM: dumb
      _JAVA_OPTIONS: "-Xmx2048m -XX:+UnlockExperimentalVMOptions -XX:+
          UseCGroupMemoryLimitForHeap"
      GRADLE_OPTS: '-Dorg.gradle.jvmargs="-Xmx2048m"'
```

Izrezak koda iznad prikazuje primjer dviju referenci. Prva je `workspace`, a druga `android_config`. Druga je vrlo važna jer definira konfiguraciju za izvođenje svih naših projekata. Koristimo CircleCI Docker sliku koja koristi Android api verziju 28. Konfigurirana je za rad s Androidom i ima instalirane neke alate koje često koristimo dok gradimo Android aplikacije. Okruženje smo definirali kao "glupo" (*eng. dumb*) kako bi smanjili prikaz gradle ispisa u izlazni tok. Također smo definirali maksimalnu količinu raspoloživog prostora kako bi izbjegli situaciju gdje kontejner ostane bez raspoložive memorije.

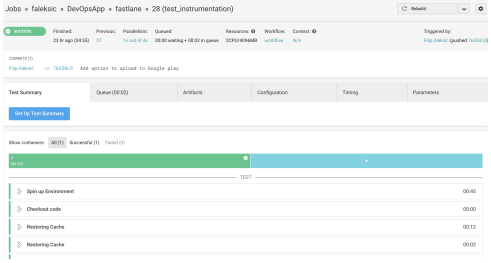
```
workflows:
  version: 2
  workflow:
    jobs:
      - test_unit
      - test_instrumentation:
          requires:
            - test_unit
      - deploy_crashlytics:
          filters:
            branches:
              only:
                - /release-.*/
          requires:
            - test_unit
            - test_instrumentation
      - deploy_play_store:
          filters:
            branches:
              only:
                - master
          requires:
            - test_unit
            - test_instrumentation
```

Primjer desno prikazuje sve definirane tokove posla. Definirani poslovi su `test_unit`, `test_instrumentation`, `deploy_crashlytics` i `deploy_play_store`. Posao `test_unit` nema preduvjeta i izvršava se na svakoj grani. Posao `test_instrumentation` se izvršava nakon što se izvrši `test_unit` što je definirano u `requires` svojstvu i izvršava se prilikom svake objave koda. Posao `deploy_crashlytics` se izvršava nakon što se izvrše prošla 2 test i samo ako se programski kod objavio na granu čiji naziv počinje s `release-`, primjerice `release-new-feature`. Posao `deploy_play_store` se izvršava isto nakon testova, ali samo ako se kod objavio na granu `master`. Postavka s granama je vrlo korisna jer ju možemo uskladiti s našom `GitFlow` tehnikom da objavljujemo u produkciju samo kod koji je na master grani.

Nakon što smo postavili konfiguraciju potrebno ju je objaviti na sustavu za verzioniranje i CircleCI će automatski započeti izgradnju.



(a) CircleCI upravljačka ploča



(b) CircleCI detalji izgradnje

Slika 23: CircleCI sučelje

Slika 23 prikazuje kako izgleda CircleCI sučelje za pregled izgradnji. Pod a prikazuje kako izgleda lista svih izvedenih izgradnji do sada. Možemo vidjeti da su neke uspješne, a neke ne. Također piše prilikom koje objave koda se pozvao test, na kojoj grani i koliko dugo je bilo potrebno da se izvede. Pod b možemo vidjeti kako izgledaju detalji izgradnje odnosno kako izgleda kad kliknemo na jedan od redaka sa slike a. Ovdje vidimo jako mnogo detalja poput koje je resurse koristio, koje je zadatke izvršavao, koje je vrijeme izvođenja bilo za pojedini zadatak, itd. CircleCI stvarno nudi odličnu integraciju, jednostavno postavljanje i vrlo je koristan DevOps alat.

Posao koji objavljuje aplikaciju na Google Play trgovinu nismo opisali pa ću ukratko navesti kako ga postaviti jer smatram da je to jedan od najvažnijih poslova. On se također izvršava uz pomoć fastlane alata i definiran je kao staza. Prvo je potrebno postaviti `supply` alat koji koristi fastane kako bi objavio aplikaciju. Postavljamo ga tako da otvorimo `Google Play` konzolu i u njoj kliknemo kreiraj servisni račun. Tada će nas preusmjeriti na `Google Developer` konzolu gdje možemo kreirati servisni račun. Kreiranje računa uključuje definiranje imena, odabir uloge (potrebno je odabrati korisnik servisnog račun (*eng. Service Account User*) i zatim generiramo JSON privatni ključ. Kad smo završili s tim vratimo se na `Google Play` konzolu i damo pristup novo kreiranom računu i damo mu ulogu voditelja projekta (*eng. Project lead*). Kad smo sve te korake napravili dodamo preuzeti JSON ključ u varijable okruženje pod ključem `GOOGLE_PLAY_KEY`. Kad smo ga dodali dodat ćemo referencu koju ćemo koristiti u poslu koja pretvara varijablu u JSON datoteku u datotečnom sustavu kontejnera (Correia, 2018c).

```
create_google_play_key: &create_google_play_key
  run:
    name: Create Google Play key
    command: echo $GOOGLE_PLAY_KEY > google-play-key.json
```

Nakon što smo kreirali referencu koju možemo vidjeti u isječku koda iznad potrebno je još na projektu u datoteci `fastlane/Appfile` postaviti varijablu `json_key_file` na vrijednost `google-play-key.json`.

Već smo spomenuli da je aplikaciju potrebno optimizirati, minimizirati, obfuskirati i pot-

pisati. Već smo vidjeli kako gradle zapravo sve to izvršava prilikom izgradnje release verzije aplikacije, ali je sada potrebno potpisati aplikaciju koja se nalazi na udaljenom serveru. Zbog toga je potrebno ažurirati stazu za izgradnju na sljedeći način:

```
desc "Assemble␣Build"
lane :assemble_build do |options|
  properties = {
    "android.injected.signing.store.file" => "keystore.jks",
    "android.injected.signing.store.password" => ENV['STORE_PASSWORD'],
    "android.injected.signing.key.alias" => ENV['KEY_ALIAS'],
    "android.injected.signing.key.password" => ENV['KEY_PASSWORD'],
  } if @build_type == "Release"

  gradle(task: "assemble", build_type: @build_type, properties: properties)
end
```

Iz isječka koda iznad vidimo da smo dodali svojstva u naše izvršavanje ako je verzija izgradnje release. Važno je primijetiti da se šifra spremišta, alias ključa i šifra ključa nalaze sve u varijablama okruženje koje možemo postaviti u CircleCI postavkama projekta. Ostaje nam jedino problem kako prenijeti samo spremiše ključeva na siguran način. Dobra praksa je enkodirati ga pomoću Base64 algoritma i dodati u varijable okoline, tako mu možemo pristupiti na siguran način kada je potrebno i dekodirati ga nazad u spremište ključeva (Correia, 2018c).

```
decode_android_key: &decode_android_key
  run:
    name: Decode Android key store
    command: echo $KEYSTORE | base64 -di | tee keystore.jks app/keystore.jks >/dev/
        null
```

U primjeru koda iznad vidimo referencu koja dekodira varijablu okruženja u spremište ključeva i dodaje postavke da se rezultat izvođenja ne ispisuje nigdje. Referencu ćemo koristiti prilikom pozivanja posla za objavu na Google Play pa zatim pozvati `assemble_build` zadatak. Sad kad smo postavili sve elemente možemo kreirati posao koji će objaviti aplikaciju na Google Play trgovinu.

```
deploy_play_store:
    <<: *android_config
    steps:
      - checkout
      - *restore_gradle_cache
      - *restore_gems_cache
      - *ruby_dependencies
      - *android_dependencies
      - *save_gradle_cache
      - *save_gems_cache
      - *decode_android_key
      - *create_google_play_key
      - run:
          name: Deploy to Play Store
          command: bundle exec fastlane deploy_to_play_store
      - store_artifacts:
          path: app/build/outputs/apk/
```

```
        destination: /apk/
    - store_artifacts:
        path: app/build/outputs/mapping/
        destination: /mapping/
```

Iznad možemo vidjeti definirani posao za objavu. Koristimo konfiguraciju istu kao i u svim ostalim poslovima. Koraci uključuju dohvat izvornog koda, dohvat spremljenih i novih zavisnosti, pa dekodiranje spremnika ključeva, kreiranje JSON ključa za objavu i zatim sam poziv staze za objavu na Google Play trgovinu (Correia, 2018c).

3.4. Zaključak

U ovom poglavlju smo se upoznali s Android sustavom, promatrali Android procesni tok od razvoja do isporuke i vidjeli razne alate koji su dostupni za implementiranje DevOps praksi tokom razvoja Android aplikacija. Promatrali smo svaki proces zasebno i naveli načine na koje možemo poboljšati svaki od njih.

Zatim smo počeli opisivati dostupne alate za primjenu DevOpsa. Započeli smo sa sustavom za verzioniranje i kako možemo poboljšati kvalitetu i znanje koristeći reviziju koda. Zatim smo opisali testove kao možda najbitniju praksu za osiguranje kvalitete jer osigurava brzu povratnu informaciju na izmjenu programskog koda i prisiljava programere da koriste dobre prakse za pisanje programskog koda kako bi ga mogli što jednostavnije testirati. Zatim smo se upoznali s alatima koji rade analizu programskog koda i prepoznaju većinu pogrešaka, te daju savijete programerima kako ih riješiti. Upoznali smo bolje Gradle i elemente njegovih konfiguracijskih datoteka. Naučili smo kako ga konfigurirati kako bi automatizirali što više zadataka i prilagodili izgradnju. Za kraj su došli alati fastlane i CircleCI koju sam omogućili maksimalno automatiziranje do sad kreiranih zadatak i automatiziranje nekih novih. Sada radimo sigurnosne provjere za svaku objavu programskog koda i tim će biti obaviješten ako nešto nije uredu i tada bi se svi trebali usmjeriti na rješavanje trenutnog problema. Smatram da bilo svaki tim koji implementira navedene prakse doživjeti veliki skok u kvaliteti, zadovoljstvu klijenata i u ukupno ušteđenom vremenu.

4. Zaključak

DevOps danas više nije praksa koju možemo implementirati u naš svakodnevni rad, već praksa koju **moramo** implementirati ako želimo biti kompetentni u današnjem svijetu. Osim visoke kvalitete imat ćemo sveukupno bolju organizacijsku kulturu, veću suradnju, povećanje organizacijskog znanja, sretnije zaposlenike i zadovoljnije klijente. Navodeći ove prednosti se može činiti da obećavamo ne realne stvari o kojima svaki član informacijske zajednice sanja, ali nakon detaljnog promatranja ideja i metoda rješavanja čestih problema u razvoju i objavi programskih proizvoda ne mogu ne pretpostaviti da će rezultat uvođenja ovih praksi u svakodnevni rad biti kao što sam opisao. Možemo samo pogledati koliku revoluciju je doživjela proizvodna industrija kada je prebacila fokus na kvalitetu i uvela Agilnu proizvodnju.

Tržište mobilnih aplikacija je sve veće i neprestano raste. Svake godine mobilni uređaji imaju sve veće sposobnosti i proširuju se njihove granice. Svake godine sve veći broj ljudi ima pristup pametnim uređajima što povećava broj potencijalnih korisnika. Isto tako i raste ponuda na trgovinama mobilnih aplikacija. Kako bi postali prepoznatljivi u masi potrebno je imati visoku razinu kvalitete i privlačnu prezentaciju aplikacije u trgovini. Upravo to možemo učiniti implementirajući prakse koje su opisane u ovom radu i koristeći alate za koje su dane upute kako ih koristiti.

Čak i nakon nekoliko godina kad se alati neizbježno promjene zbog dinamike informacijskog svijeta, prakse i ideje će ostati jednako važne. Važno je shvatiti da se DevOps ne radi o alatima, već o kulturi unutar organizacije. Važno je kakav je odnos prema pogreškama, kakva je suradnja, kakva je razmjena znanja i kakva je želja za poboljšanjem. U onom trenutku kad svaki zaposlenik unutar organizacije nema strah prilikom pronalaska pogreške već je iskreno zainteresiran kako je došlo do te pogreške, kako zaštiti sustav da se to više ne ponovi i kad dijeli novo dobiveno znanje sa svojim kolegama, tada ste uspjelu u DevOpsu. Alati će se pronaći i vaša organizacija će uspjeti!

Popis literature

95 Amazing Android Statistics. (2014). https://expandedramblings.com/index.php/android-statistics/.

App Stores: Number of Apps in Leading App Stores 2019. (2019). https://www.statista.com/statistics/276623/number-of-apps-available-in-leading-app-stores/.

Application Fundamentals. (n.d.). https://developer.android.com/guide/components/fundamentals.

Bass, L., Weber, I. M., & Zhu, L. (2015). *DevOps: A software architect's perspective*. The SEI Series in Software Engineering. New York: Addison-Wesley Professional.

Beyond the Goal: [Eliyahu Goldratt Speaks on the Theory of Constraints. (2005). OCLC: 1037184250.

Build Local Unit Tests | Android Developers. (n.d.). https://developer.android.com/training/testing/unit-testing/local-unit-tests.html.

Chacon, S. & Straub, B. (2014). *Pro Git: Everything you need to know about Git* (2. ed). The Expert's Voice. OCLC: 915532514. New York, NY: Apress/Springer.

Code Quality | SonarSource. (n.d.). https://www.sonarsource.com/why-us/code-quality/.

Configure Your Build. (n.d.). https://developer.android.com/studio/build.

Correia, B. (2018a). Android Continuous Integration using Fastlane and CircleCI 2.0 — Part I. https://medium.com/pink-room-club/android-continuous-integration-using-fastlane-and-circleci-2-0-part-i-7204e2e7b8b.

Correia, B. (2018b). Android Continuous Integration using Fastlane and CircleCI 2.0 — Part II. https://medium.com/pink-room-club/android-continuous-integration-using-fastlane-and-circleci-2-0-part-ii-7f8dd7265659.

Correia, B. (2018c). Android Continuous Integration using Fastlane and CircleCI 2.0 — Part III. https://medium.com/pink-room-club/android-continuous-integration-using-fastlane-and-circleci-2-0-part-iii-ccdf5b83d8f5.

Crashlytics for Android - Fabric Install. (n.d.). https://fabric.io/kits/android/crashlytics/install.

Davis, J. & Daniels, K. (2016). *Effective devOps: Building a culture of collaboration, affinity, and tooling at scale* (First edition). OCLC: ocn918591238. Beijing ; Boston: O'Reilly.

Documentation | SonarQube Docs. (n.d.).
https://docs.sonarqube.org/latest/.

Driessen, V. (2010). A successful Git branching model.
http://nvie.com/posts/a-successful-git-branching-model/.

Ebel, N. (2018). What are “static analysis” tools?
https://proandroiddev.com/what-are-static-analysis-tools-48ccff8135d4.

Fowler, M. (2006). Continuous Integration.
https://martinfowler.com/articles/continuousIntegration.html.

Franco, J. P. (2017). DevOps 2.0 for Digital Transformation - DevOps.com. https://devops.com/devops-2-0-digital-transformation/.

Fundamentals of Testing | Android Developers. (n.d.).
https://developer.android.com/training/testing/fundamentals.

Hüttermann, M. (2012). *DevOps for developers*. The Expert’s Voice in Web Development. OCLC: 792879828. New York, NY: Apress.

Improve Your Code with Lint Checks | Android Developers. (n.d.).
https://developer.android.com/studio/write/lint.

Kim, G. (2012). The Three Ways: The Principles Underpinning DevOps.

Kim, G., Debois, P., Willis, J., Humble, J., & Allspaw, J. (2016). *The DevOps handbook: How to create world-class agility, reliability, & security in technology organizations* (First edition). OCLC: ocn907166314. Portland, OR: IT Revolution Press, LLC.

Martinez, J. (2016). DevOps on Android: From one git push to a Play Store release by Jeremie Martinez - YouTube.
https://www.youtube.com/watch?v=O58yq8B3shc.

Pittet, S. (n.d.). Continuous integration vs. continuous delivery vs. continuous deployment.
https://www.atlassian.com/continuous-delivery/principles/continuous-integration-vs-delivery-vs-deployment.

Platform Architecture | Android Developers. (n.d.).
https://developer.android.com/guide/platform.

Poppendieck, M. & Poppendieck, T. D. (2007). *Implementing lean software development: From concept to cash*. The Addison-Wesley Signature Series. OCLC: ocm70114627. Upper Saddle River, NJ: Addison-Wesley.

Screengrab - Fastlane Docs. (n.d.).
https://docs.fastlane.tools/actions/screengrab/.

Setup - Fastlane Docs. (n.d.).
https://docs.fastlane.tools/getting-started/android/setup/.

Shrink, Obfuscate, and Optimize Your App. (n.d.).
https://developer.android.com/studio/build/shrink-code.

Sign Your App. (n.d.).
https://developer.android.com/studio/publish/app-signing.

Spear, S. J. & Spear, S. J. (2011). *The high-velocity edge: How market leaders leverage operational excellence to beat the competition*. OCLC: 966601956.

Test Apps on Android. (n.d.).
https://developer.android.com/training/testing.

Westrum, R. (2004). A typology of organisational cultures. *Quality and Safety in Health Care*, *13*(suppl_2), ii22–ii27. doi:10.1136/qshc.2003.009522

Westrum, R. [Ron]. (2014). The study of information flow: A personal journey. *Safety Science*, *67*, 58–63.

Writing Custom Lint Rules - Android Studio Project Site. (n.d.). http://tools.android.com/tips/lint-custom-rules.

Zipalign | Android Developers. (n.d.). https://developer.android.com/studio/command-line/zipalign.

Popis slika

Popis tablica

A. Prilog - Git repozitoriji projekta

Pristup javnom Git repozitoriju koji sadrži Android projekt kreiran u ovom radu:
https://github.com/faleksic/DevOpsApp