Aelitta Ezugbaya

Continuous Deployment For Cross-Platform Mobile Application

Metropolia University of Applied Sciences
Bachelor of Engineering
Information Technology
Bachelor’s Thesis
14 April 2019
The goals of this thesis are to figure out the best modern technologies and services for creating a Continuous Deployment pipeline for cross-platform mobile application, and to create a pipeline, which will build, test, sign and deliver the app to "Apple Store" and "Play market". The pipeline was created for a ReactNative application. CD pipeline for cross-platform mobile application consists of two CD pipelines for Android and iOS application that are run in parallel. The CD pipeline also a common part for both mobile OS which runs tests for them.

The CD pipeline deploys the app to three release environments: Alpha, Beta, and Production. Both iOS and Android apps are deployed to different release platforms because the Android can be deployed to different release environment within one native Play Market store, while Apple does not provide such services, so third-party solutions have to be used.

The thesis describes modern tools and services for creating and improving the CD pipeline. The biggest part of the paper is about the implementation process of the CD. It describes in detail iOS and Android signing process, the alpha, beta and production deployment, iOS and Android app building and aspects of using tools and services for CD pipeline.

<table>
<thead>
<tr>
<th>Author</th>
<th>Aelitta Ezugbaya</th>
</tr>
</thead>
<tbody>
<tr>
<td>Title</td>
<td>Continuous Deployment For Cross-Platform Mobile Application</td>
</tr>
<tr>
<td>Number of Pages</td>
<td>46 pages + 3 appendices</td>
</tr>
<tr>
<td>Date</td>
<td>14 April 2019</td>
</tr>
<tr>
<td>Degree</td>
<td>Bachelor of Engineering</td>
</tr>
<tr>
<td>Degree Programme</td>
<td>Information Technology</td>
</tr>
<tr>
<td>Professional Major</td>
<td>Software Engineering</td>
</tr>
<tr>
<td>Instructors</td>
<td>Marko Klemetti, Lead Architect</td>
</tr>
<tr>
<td></td>
<td>Janne Sallonen, Head of School, School of ICT</td>
</tr>
</tbody>
</table>

Keywords

Continuous Deployment, Continuous Integration, Cross-platform mobile application
Contents

List of Abbreviations

1 Introduction 1

2 Literature Review 1

  2.1 Continuous Integration and Continuous Delivery and Continuous Deployment 1
  2.1.1 Continuous Integration. 1
  2.1.2 Continuous Delivery. 2
  2.1.3 Continuous Deployment. 2
  2.1.4 Conclusion 2

  2.2 CircleCI 3

  2.3 ReactNative 4
  2.3.1 History 4
  2.3.2 Advantages 4
  2.3.3 Disadvantages 5
  2.3.4 Conclusion 6

  2.4 Fastlane 6
  2.4.1 Capture screenshots automatically 7
  2.4.2 Releasing the application 8
  2.4.3 Match – Automatic code signing for iOS 11

3 Implementation 12

  3.1 CD description 12
  3.1.1 Basics of CD pipeline structure 13
  3.1.2 Branches 14

  3.2 CircleCI settings 15
  3.2.1 config.yml and its structure 15
  3.2.2 Common part for iOS and Android 16

  3.3 iOS 18
  3.3.1 Fastlane set up 19
  3.3.2 Signing 19
  3.3.3 Building 22
  3.3.4 Alpha version 23
3.3.5 Beta 25
3.3.6 AppStore 26
3.3.7 config.yml and CircleCI works 27

3.4 Android 29
3.4.1 Fastlane 29
3.4.2 Signing 31
3.4.3 Building 34
3.4.4 Delivery Android app to different environment 35

4 Results and Discussions 37
4.1 Summary of Results and Evaluations of Results 37
4.2 Future Improvements 38

5 Conclusion 40

References 42

Appendices
Appendix 1. ~/ios/fastlane/Fastfile – Fastlane for iOS
Appendix 2. ~/android/fastlane/Fastfile – Fastlane for Android
Appendix 3. ~/.circleci/config.yml
List of Abbreviations

CI  Continuous Integration
CD  Continuous Deployment
CDE Continuous Delivery
OS  Operating System
DevOps Development and Operations
VM  Virtual Machine
VCS Version Code System
QA  Quality Assurance
APK Android Package
IPA iOS App Store Package
UI  User Interface
1 Introduction

Mobile technologies become more and more popular nowadays. One of the problems is that mobile application development needs a lot of recourses. A company needs to find iOS and Android developers and develop applications for different mobile operating systems (OS) separately. Using a mobile cross-platform framework such as React Native can solve this problem. React Native is a framework for creating a cross-platform mobile applications; this means the application can run on devices with different operating systems: Android and iOS. If it is planned to update and maintain the app continuously, then Continuous Deployment (CD) pipeline has to be done for comfortable app development and deployment.

The CD pipeline in this thesis should be done for a small self-made mobile application for creating a shopping list. The mobile application is developed with React Native framework. The CD pipeline should handle unit testing, mobile application building, code signing, and deploying. Also, the pipeline should handle app deployment to different release environments: Alpha testing, Beta testing, and Production. All these pipeline processes should be run on different types of machines with different operating systems. The code signing processes are absolutely different between platforms, the same as apps building and deploying. The goal of this thesis is to create the CD pipeline for cross-platform mobile application, which will handle the build and deployment for two different platforms and consider the aspects of both ecosystems.

I have never had such experience, so the thesis aims to find the best ways and technologies for creating a continuous deployment pipeline and to create one using those tools.

2 Literature Review

2.1 Continuous Integration vs. Continuous Delivery vs. Continuous Deployment

The beginner in the Development and Operations (DevOps) world can be confused with all these abbreviation and names, such as Continuous Integration (CI), Continuous Delivery (CDE), and Continuous Deployment (CD). What is the difference between them and when these or that principles should be used? These concepts will be explained in detail in the next several paragraphs.

2.1.1 Continuous Integration.

Continuous Integration (CI) – is a widely established development practice in the software development industry, when developers merge their changes to the main branch as
often as possible. It helps to improve software quality. CI includes automated software building and testing, which helps to validate developer’s new changes. This practice helps developers to avoid the merge hell that can happen if they are merging their changes by release day. CI is a great solution to be confident that the product is not broken whenever new changes coming to the main branch.

2.1.2 Continuous Delivery.

Continuous Delivery (CDE) is a software engineering approach in which team always keeps software at production-ready-state. CDE is the extension of Continuous Integration, where automated build and testing is complemented with the automated release process. CDE can include automated deployment process to the production-like environment or have on top of automated testing and building the automated release process that can deploy the product to production at any time by clicking a button. Such an approach has several benefits, such as reducing deployments risks and faster delivery to users.

Continuous Delivery is about giving the release power to the hands of the business team, not to the hands of development team. Implementing Continuous Delivery is making sure the software is always production-ready, and any build could potentially be released to users at the click of a button using a fully automated process.

2.1.3 Continuous Deployment.

Continuous deployment (CD) is an approach that goes a little bit further than CDE and implements the continuous deployment to production or customer environment. It means that every product changes that pass all stages of the production pipeline will be released to the customers automatically; only the fall of tests or any other stages can stop the deployment.

There is a debate about the differences between CDE and CD. Some people are not distinguishing CDE and CD. CD employs CDE practices, but the final deployment in CDE is a manual step, while CD should not contain any manual steps, that means as developers commit its changes into production branch, the process of delivering the product changes to customers is started automatically. CDE approach is pull-based where business part decides what and when to deploy. CD is a push-based approach. Also, one of the most significant differences is that while CDE practice can be applied for all types of systems and organizations, CD practice may only be suitable for certain types of organizations or systems.

2.1.4 Conclusion

Learned all approaches detailed, continuous deployment is the most efficient way for the current project because all release processes are automated and developers can focus
more on developing the product than doing the routine job to deploy the app. Figure 1 shows in scheme how CI, CD, and CDE work and what are the differences.

Figure 1. The relationship between continuous integration, delivery and deployment [16], [17].

2.2 CircleCI

There is a wide variety of Continuous Integration and Delivery (CI/CD) Platforms that can be used for automating integration and delivery of the mobile application. Which one should be chosen? The platform should be chosen according to several criteria:

- Price
- Aim for mobile application
- Possibility of using Mac machines to build apps for iOS
- Possibility of deployment to app stores
- Popularity
- Possibility of signing the app

The final choice was CircleCI. CircleCI was founded in 2011 and is headquartered in San Francisco with a global remote workforce. CircleCI is venture-backed by Scale Venture Partners, DFJ, Baseline Ventures, Top Tier Capital, Industry Ventures, Heavybit, and Harrison Metal Capital. [3]

CircleCI is a modern Continuous Integration and Continuous Delivery (CI/CD) platform. It enables automating the build, testing, and delivery processes. It promises the ability to build an automated pipeline from committing to deploy.
This platform is free for first "container"; each additional container will cost 50$, but this price is only for Linux Machines. Packages with Mac machine starts from 39$ per month. Therefore, CircleCI is very popular, the most popular tool for mobile applications, it is even used by ReactNative developers [4]. This fact gives the confidence that this platform will be working smoothly with my React Native application.

CircleCI supports integration with free tool – Fastlane. This tool does automated code signing and delivery.

CircleCI orchestrates the entire delivery process from build to deploy through a single file called config.yml, which is located in a “.circleci” folder at the top of the project. [5] The whole set up of the pipeline is stored there.

CircleCI can run commands in one of the three environments: docker images, Linux, or macOS virtual machines. [6] It allows building iOS and Android applications in the CircleCI cloud because iOS apps need to be built in macOS system, while integrated android docker image can be used to build an android app. [7]

2.3 ReactNative

ReactNative is a JavaScript-based framework created by Facebook that allows development of native mobile apps for iOS and Android. [8]

2.3.1 History

The story of ReactNative is quite exciting. Firstly, Facebook launched the React.js, the minimalistic JavaScript framework, which became popular. Facebook realized that the same idea could be implemented for developing mobile apps, and the first version of ReactNative was created in the summer of 2013 within an internal hackathon. The First public presentation happened two years later in January 2015 during React.js Conference. [9]

The official release happened in May 2015 when the company also open-sourced ReactNative. Initially ReactNative supported only iOS, but Facebook released Android support in the September of 2015. [9]

2.3.2 Advantages

ReactNative became quite popular as its elder brother React. There are several benefits of using React Native to develop mobile applications.
2.3.2.1 It is not web-app inside a mobile application, it is a native app

One of the most significant ReactNative advantages is that it provides truly native user experience, unlike other hybrid tools that merely provide a native-style wrapper for browser-based apps. If the user is unaware of the technology used to build an app, he would not be able to differentiate between a simple app built with React Native or an Android programming language or iOS.[10]

2.3.2.2 Cross-platform

The ability to create one app to run in for multiple OS is excellent. It does not matter that it is not possible to have all the features and functionalities that native apps have. A simple app can be implemented the as well as with native tools, and an ordinary user would never notice that it is not native. ReactNative lets to have applications for two platforms with the time, effort and resources for creating one.

2.3.2.3 Short Team Size, Save Money

When one developer can create an app for iOS and Android, it reduces the size of the team and the cost of the work. Developers even do not need to study two different languages and platforms for them. It is better to have one team than having two different teams because two teams potentially can have communicational issues between each other. JavaScript developers, preferably with React background, would be the best option for developing ReactNative application.

2.3.2.4 Ready-made Components Increase Development Speed

Another great feature is that React Native has several built-in components. It is beneficial because developers do not need to write everything from scratch; they can use ready components. It not only makes the development process simple but also faster.

Also, developers are not limited by ReactNative built-in components, React community is so big, that there are third party libraries which can provide something more interesting and specific. [11]

2.3.3 Disadvantages

2.3.3.4 Native Code Required

Developers on ReactNative might still need to write a minimal amount of native code when developing apps.[10] It can become a problem for teams, that started developing
using React Native to avoid having to do native code for iOS and Android. It is something that should be considered beforehand, especially if the app will need access to device camera and other onboard hardware. In a rare case, this con can outweigh all pros.

2.3.3.5 Low Security

ReactNative is a JavaScript-based library. JavaScript is famous for its number of potential vulnerabilities. That is why ReactNative should be used very accurately for apps which require high level of security, like mobile banking apps or personal finance apps. Developers will need to pay extra attention to details because JavaScript is famous for its fragility. Extra attention should be applied to code details and any third-party libraries.

2.3.3.6 ReactNative is still beta

React Native is already in use by numbers of big companies, Facebook, Instagram, DanskeBank, but React Native is still in beta version. It still has issues with debugging, hot reloading problems, and it has some breaking changes from version to version.

2.3.4 Conclusion

There are several problems with React Native that can scary newcomers, but it is still the easiest way to start the mobile application for web-developers. It is a good option for small teams and when there are limited recourses.

2.4 Fastlane

Fastlane is an open-source tool that helps to make Android and iOS deployment much more comfortable. Fastlane gives the possibility to automate every aspect of deployment and release. It allows automating such tedious tasks as taking screenshots, code signing, beta, and app store deployment. [12]

Fastfile is a file which contains the configuration of Fastlane commands for the app. It is located under the “fastlane” folder on the root of android or iOS folder. The Fastfile gives the possibility to use the integrated Fastlane commands (build_app, upload_to_testflight, etc.), which help to make the deployment process much easier. Each new Fastlane command is a new public lane (command, function) inside the Fastfile. It is easy to call any of them by typing "fastlane [lane name]" in the terminal, from the root folder of iOS or Android app.
2.4.1 Capture screenshots automatically

Taking screenshots does not sound complicated and time-consuming task, but in fact, it is. Screenshots should give an understanding of app content and motivate users to download it, so the screenshots should be well-prepared.

Several aspects should be considered, for example, different languages, if the app should be localized in many languages. Some companies choose to create screenshots in one language and use them for all locales. While this might seem okay to developers, many potential users are out because they cannot read the text on the app screenshots. It will not convince them to download the app. Also, screenshots should be consistent with the latest version of the app. Screenshots should not contain loading indicators, and they should be in the right format for the app stores. Manual taking screenshot approach has one positive side is that these screenshots will all be crisp and correctly sized, with readable text.

Fastlane tools can automate this process and make it fast and consistent while giving beautiful results. [13] Fastlane’s screenshot feature works to automate this process, and it gives such benefits as:

- A developer can take a break or spend the time doing something valuable while his computer takes screenshots.
- It can make screenshots on all necessary simulators in multiple languages.
- Fastlane can make take screenshots on different devices in parallel, that will cut down the execution time of the whole process.
- As all Fastlane features, once this process is configured, any one of the team can run it.
- Screenshots will never have loading indicator because Fastlane is intelligently waiting for network requests to be finished.
- Fastlane can generate the web page with all screenshots [Figure 2], which can be used for many purposes (for example: translation verification)
2.4.2 Releasing the application

One of the key features of Fastlane is the possibility to deploy an app to beta or production environments. It includes such tasks as incrementing the build version, code signing, preparing build release, setting a changelog, and uploading it to the chosen release service.

Manually making all these tasks is very time-consuming, so the automation of these processes saves developers’ time and mental health. Once the automation for these tasks is configured, it will work for all consecutive times.

Fastlane supports out of the box over 15 beta testing services including: Google Play, TestFlight, Crashlytics Beta and Hockey and also it includes uploading to AppStore. It is not only helping to publish new builds but also can send the new build to review for AppStore approval, which means that even with iOS Fastlane can automate the whole release process.[12]

2.4.2.1 Incrementing the build number

Most release services ask to increment the build number each time to upload a new build. It is a requirement for TestFlight, for example.[14] Build number should always
be bigger than the previous one; it is a requirement of the most beta test platforms and AppStore and Google Play. Fastlane offers several solutions to automate this process.

1. Fetching the latest build number from TestFlight or AppStore and increase its number to one. It works only with these two platforms. [14]

2. Committing the build number to version control – so keeping the build number in git and commit the increased number during each build. [14]

3. Use the number of commits – it is the easiest way, and it is universal for iOS and Android, but build numbers can be not consistent.[14]

2.4.2.2 Building the app

Something should be uploaded to platforms to be shared with testers and customers, Android Package (APK) file should be built for Android and iOS App Store Package (IPA) file for iOS. Fastlane provides the functionality to build such files.

1. gradle – is the Fastlane function to execute all gradle related actions, including building and testing of the Android app. Gradle can take parameters that will configure and set up its job. These are the most critical properties to build APK file: task, build_type, and properties where build number should be defined.[15]

```plaintext
gradle(
    task: 'assemble',
    build_type: 'Release',
    properties: {'VERSION_CODE' => build_number}
)
```

Figure 3. The code to build android release build

2. gym -is a Fastlane function to build and archive the iOS app. It takes care of all the heavy lifting and makes it super easy to generate a signed IPA file. Gym as Gradle has configuration properties. There is a list of the minimum set of rules to prepare release build with gym: scheme, configuration, and export_method.[16]

```plaintext
gym(
    scheme: "ShoppingList",
    configuration: "Release",
    export_method: "app-store",
)
```
Figure 4. The code to build IPA release file

The build number does not have to be passed to the gym because Fastlane has a function to define new build number for iOS

```
increment_build_number( build_number: build_number)
```

Figure 5. The code to increase iOS build number

### 2.4.2.3 Changelog

Changelog – is summary of new changes in the product for a new release. Fastlane offers several solutions to automate it; there is a list of the easiest approaches.

1. Changelog based on git commits
   - This approach is the fastest way, where developers do not spend time to write the summarize of the changes. Fastlane takes the latest commit and formats them to changelog. [17]

   ```
   changelog_from_git_commits # this will
   ```

   Figure 6. Fastlane command to generate changelog based on last commit [17]

   The only problem here is that developers always should consider that all their commit texts will be seen to customers. This problem can be solved by passing the option to Fastlane function, which can filter what commits will be used to generate changelog.

   ```
   changelog_from_git_commits(
       between: ['7b092b3', 'HEAD'], # Optional, lets you set the
       merge_commit_filtering: exclude_merges # Optional, let
   )
   ```

   Figure 7. Option to filter commits for changelog [17]

2. Fetching the changelog from the file

   This approach lets preparing professional and good structured changelog. It means that the changelog should be prepared in advance and can be stored in the project folder or somewhere in the cloud, and Fastlane can take changelog from this file and pass it to release function. [17]
2.4.2.4 Upload to app stores and testers

Fastlane tool supports over 15 platforms where app builds can be released, [12] and these platforms can be reached from Fastlane with specific Fastlane command for each of them. For example, there is a command “upload_to_testflight” to upload a build to TestFlight or “upload_to_play_store” is used to upload the build to Play Market.

```ruby
lane :beta do
  # Variant 1: Read from file system
  # note the `..`, since fastlane runs in the _fastlane_ directory
  changelog = File.read("../Changelog.txt")

  # Variant 2: Fetch data from a remote web server
  changelog = download(url: "https://lookatmycms.com/changelog.txt")

  sync_code_signing
  build_app
  upload_to_testflight(changelog: changelog)
end
```

Figure 8. Approaches to take changelog from file [17]

```ruby
upload_to_testflight(
  username: "aelittae@gmail.com",
  changelog: changelog
)
```

Figure 9. Block of code to upload iOS build to TestFlight

2.4.3 Match – Automatic code signing for iOS

Fastlane “match” is the easiest way and tool to manage the code signing of mobile application in iOS.

2.4.3.1 Code Signing without Match

Deploying the app to AppStore is not that easy; the most important and challenging step here is code signing, especially for the team. The iOS app signing needs that the certificate should be generated on the developer’s machine and then be uploaded to Apple developer console; based on it a provision profile is created. There are different types of
certificates for different types of builds; this means that one developer can have several certificates and provision profiles. [18]

Development teams have to separate code signing identities for every member. It results in numbers of profiles, including many duplicates. Also, developers have to manually renew and download the latest set of provisioning profiles every time they add a new device, or a certificate expires. Additionally, this requires spending much time when setting up a new machine that will build the app and making this set up on a cloud-based platform as CircleCI seems to be a big problem. [18]

2.4.3.2 Code Signing with Match

The solution was found; it is to share one code signing identity across the development team to simplify the setup and prevent code signing issues. The goal is that every developer is using the same certificates and profiles and regenerating them if necessary. The certificates and provision profiles can be shared through a private Git repository. Match-file - is a configuration file for Fastlane "match" tool where should be stored the git URL with certificates and username. [19]

"match" tool creates all required certificates & provisioning profiles and stores them in a separate git repository. Every team member with access to the repository can use those credentials for code signing. "match" tool also automatically repairs broken and expired credentials. It is the easiest way to share signing credentials across teams.[19]

3 Implementation

This block describes the implementation of Continuous Deployment pipeline for cross-platform mobile application.

3.1 CD description

There are requirements for the CD pipeline for cross-platform mobile application.

1. Create a different set of rules for different git branches.

2. Have different release versions and deliver changes to different environments: Alpha, Beta, Production

3. Every build should pass all tests and steps before being delivered
4. CD pipeline should build an application for both mobile OSs on every pushed change.

3.1.1 Basics of CD pipeline structure

The structure of the continuous deployment pipeline should be clear and consistent.

![Diagram of the CD pipeline for cross-platform mobile application](image)

Figure 10. UML diagram of the CD pipeline for cross-platform mobile application

The CD pipeline for cross-platform mobile application is separated to iOS and Android part; they should be separated because applications should use different environments to be built: macOS virtual machine (VM) and Android Docker Image. These two parts have the same sets of jobs. Job is a collection of executable commands. All commands in the job are executed in a single unit. Each job has its own aim, like deploy to production, prepare development build, etc. The iOS and Android parts have such jobs as a development build, alpha deploy, beta deploy, and production deploy. Also, there will be one common job that will run tests for both platforms. Figure 10 represents the structure of the CD pipeline, where “Common part” is the common job which runs unit tests for both OSs. It is the start of the process, after Android and iOS part run in parallel by choosing the correct type of job.

CircleCI connects to the version control system (VCS) as many other CI/CD platforms to track all code changes. CircleCI can trigger the pipeline build on every change.
pushed to a remote repository or only trigger the pipeline build on Pull Requests. Pull Request lets to tell collaborators about changes that have been pushed to a branch in a repository on VCS. Once a pull request is opened, the changes can be discussed and reviewed by collaborators and add follow-up commits before changes will be merged into the base branch. CircleCI can run the pipeline on one branch or on the several at the same time. CircleCI differentiates git branches and runs a different set of jobs for each of them.

![UML Diagram of the app release lifecycle](image)

There are three release environments to be supported by the CD pipeline: Alpha, Beta, and Production. Alpha and Beta are the test environments, but they are different. Alpha is usually the first test step for the product. Alpha testers are usually a limited group of people. Quality assurance (QA) engineers, developers, and people who are tightly bound to the development process usually have access to the alpha version of the app. Alpha version of the app is a private test environment.

Beta testing usually available for a much bigger group of people than for Alpha testing, the beta version of the app can be even public, and any user of the product can install it and give feedback about its new features and changes. Benefits of the beta testing are to check how the new features will be accepted and used by customers.

The production environment is the last step; it is the moment when all customers of the product should get new changes, fixes, and features.

### 3.1.2 Branches

The continuous deployment for cross-platform mobile application contains four different build pipeline plans to deal with different branches and different environments. The list of build plans contains development, alpha, beta, and production types. Each build plan has to pass a list of required steps to deliver new changes. The required steps are code signing, success build, and success tests pass. It is a minimum requirement which is included in all pipeline build plans. Development build plan is the simplest one; it contains only the minimum required steps. The development pipeline runs for all git branches by default, except the main branches: master, beta, and production.

Alpha, beta, and production are pipelines build plans to deliver the changes to corresponding environments. They contain the minimum required steps as a development build and extended by deployment to the necessary environment.
Three main branches were mentioned above; these branches contain the different versions of the mobile app. Master branch is the first step where the feature branch will be merged. Master branch contains the alpha app version. It means that all changes that would be merged to master will come to the alpha environment and alpha testers.

Master branch can be merged to Beta branch when Alpha version would pass all check stages; then new changes will be available for Beta testing. Production branch and production environment is the last step, where customers already can fully enjoy the new changes and functionalities of the app.

![Diagram of deployment stages based on branch](image)

Figure 12. Diagram of deployment stages based on branch

Figure 12 shows the communication between Git branches and the deployment build plan for changes on different branches. It visually represents the deployment lifecycle of changes, how changes come from development step to production.

3.2 CircleCI settings

It is necessary to sign up to the CircleCI via GitHub or Bitbucket account depends which VCS service is in use. It lets CircleCI service get access to remote git repositories. The necessary repository should be chosen to start creating a build pipeline for it. The next step is to create a configuration for the CD pipeline build in `~/.circleci/config.yml`.

3.2.1 config.yml and its structure

config.yml is a file which will orchestrate the whole CD pipeline process. It uses the YAML syntax for config. YAML is a readable data serialization language. It is a strict superset of JSON and another data serialization language. It means that it can do what JSON can and even more. CircleCI configuration is stored in a single YAML file located at `.circleci/config.yml`, in the root of the project’s directory. [21]
config.yml file starts with defining the version of CircleCI platform. Version 2.0 is the most recent one. Then the file is divided into two parts: jobs and workflow. Jobs are a collection of steps, and each job must declare an executor that is either docker, Linux, or macOS virtual machine. Each job should have the unique name; if a workflow is in use in config.yml, otherwise there should be a job with name "build" because the “build” job is the default entry-point for a run that is triggered by a push to your VCS. Each job should contain two parts: steps and an executor. Steps are actions that need to be taken to perform the job. Steps are usually a collection of executable commands. [22]

Workflows is a list of jobs and the rules when they should be executed. There different ways how jobs can run, they can run in parallel, sequentially, on some schedule or run by manual approval. [22] Workflow is used for orchestrating all jobs. Each workflow contains its name as a key and a map as a value. A name should be unique within one config.yml.[23]

![Diagram of CircleCI workflow](image)

Figure 13. Example of workflow [22]

### 3.2.2 Common part for iOS and Android

config.yml file should be committed and pushed to VCS to trigger first build in CircleCI. config.yml is the CD deployment orchestrator file. It contains different jobs for android and iOS builds as was described before, but there is a job that should be run for both types of builds. Figure14 contains the common job “run_tests” which should be run with any build, for any branch and any environment. Also, this figure represents the basic structure of the config.yml with jobs and workflow configurations.
The latest version of CircleCI was used for config.yml; the version 2.0 is the latest one for the moment. There is only one job; it is aimed to run the unit test for the app. NPM libraries should be installed on the virtual machine to run specific "npm" command for test execution. The first line of the job defines where in which directory the upcoming steps should be executed. Then the name of the docker image is defined. The docker image with stable node.js version is the perfect choice as "npm" commands should be run. Steps part is defined next. First of all, the "checkout" command should be run. The checkout step checks out the source code for a job over SSH. [22] Next step is to restore cache of package.json. “save_cache” method generates and stores a cache of a file or directory such as dependencies or source code in the object storage. The cache will be
regenerated every time something is changed in the cached file. (https://circleci.com/docs/2.0/configuration-reference/#save_cache). Two simple "npm" commands should be run, “npm install” – which installs all external libraries for the project and “npm test” - is a command to run unit tests for the app. Then the current version of package.json is saved to cache, to speed up next pipeline build. Next step is to persist “node_modules” directory for other jobs, this will help to improve the time execution of next jobs where “node_modules” also need to be used. “persist_to_workspace” is a special method used to persist a temporary file to be used by another job in the workflow. The last step of “run_tests” job is to define the path where the results of unit tests should be saved, this can be used to figure out what is the reason of tests fail.

This job is the start points for all other jobs in the CD deployment pipeline. The application cannot be built, signed, or delivered to any platform if this job would ever fail. This is a quality basis for the code changes. This initial step can be improved; it can be extended with “audit” step to check that any third-party library does not have any vulnerabilities or with “lint” check to test the code quality. It is the first part, which is general for both platform iOS or Android.

3.3 iOS

Auto deployment of iOS app is important for developers, who are planning to update apps often, because manual deliver process for iOS app is very annoying, time-consuming and contains many steps, that fortunately can be automated with Fastlane. This process is so tricky because Apple tries to secure its store and to prevent of containing apps with vulnerabilities from no-name developers.

Apple Development account lets creating apps for distribution, not only in AppStore, but it also gives the possibility to upload apps to beta releases platforms, including TestFlight from Apple. The Apple Developer Membership is cost 99 USD per year. [24] Developers get access to the tools needed to test and distribute apps.

The first thing that should be done for creating the CD pipeline for iOS application is to create the Apple Developer account; this is enough to start work on the pipeline from the local machine, when all infrastructure is set up locally, it makes sense to buy the macOS plan in CircleCI and start running the deployment pipeline there.

The most critical and challenging part for iOS app deployment is the code signing because Apple has a different type of certificates, provision profiles and the code signing needs two steps verifications with the apple account to secure that nobody steals the app signing credentials. Also, it is necessary to create several build configurations for different app-building; by default, there is “Debug” and “Release”, but “Alpha” configuration is also needed. CD pipeline consists of development, alpha, beta and production builds, “Debug” configuration can be used for development build, “Alpha” build configuration is for alpha testing, but “Release” configuration build can be used for beta and production builds and deployments, as TestFlight platform from Apple is used for beta testing.
TestFlight requires that app should be signed as “release” app and build as for release.
iOS app is delivered to three different app distribution platforms. Alpha is delivered to Crashlitycs, Beta to TestFlight, and Production to AppStore.

The app should pass Apple review before it can be accessed by AppStore users or beta testers in TestFlight. The app review usually takes 24 hours. The app should be reviewed to be sure that there are no crashes, bugs, broken link, unnecessary permissions request to users’ data and everything else that can bring issues for AppStore customers. It is how apple tries to secure its users and provide only high-quality apps. [25]

3.3.1 Fastlane set up

Fastlane should be installed separately for iOS and Android; it means that the install command for Fastlane should be run inside ~/android and ~/ios, where ~ is the root directory of the project. Fastlane can be installed by the command “sudo gem install fastlane -NV”. [26]

Appendix 1 represents the Fastfile for iOS part of React Native app. The first line of the Fastfile is the line which defines that the default platform for this Fastfile is iOS. The platform also defines at the beginning of the configuration block of code. Fastfile consists of four different lanes. Fastfile lane is a set of commands to run together. Each lane is supposed to build the specific version of the application. iOS Fastfile contains development lane, alpha lane, beta lane, and production lane. Each lane contains a function to increment a build number, "match" function to sign the build and command to build the application. Alpha, beta, and production also contain a function to deploy built iOS application to different release platforms and get a changelog. The number of commits defines the build number. The changelog is taking the text from the last commits. It is suitable for the current architecture of CI to have a build number as a number of commits because there is a hierarchical structure of branches and higher branch in the structure always will have one commit more than the lower branch.

Fastfile contains integrated function “before_all” which runs before any lane. This “before_all” contains one function “setup_circle_ci”. This action helps with CircleCI integration. “setup_circle_ci” is Fastlane integrated method to set up the keychain and match to work with CircleCI. What it actually does? It creates a new temporary keychain for use with match, switches” match” to “readonly” mode to prevent creating new provision profiles and certificates on the remote machine, sets up log and test result paths to be easily collectible. [27]

3.3.2 Signing

The app code signing assures users that the app is from a known source, and the app has not been modified since it was last signed. Before the app can be integrated into app services, be installed on a device, or be submitted to the App Store, it must be signed with a certificate issued by Apple. [28]
Fastlane match takes care of everything that belongs to iOS app signing. "match" will automatically create necessary certificates and provision profile. iOS Certificate is a public/private key-pair, which identifies who developed the app. [29] A corresponding provision profile should be created for each certificate. Provisioning profile is a set of digital information that uniquely ties developers and devices to an authorized iPhone Development Team and enables a device to be used for testing. [30] A provision profile contains a certificate, apple id, and all other digital information, which allows to install the app on local devices or distribute it.

There are two options of the certificate that can be created: a development certificate and a distribution certificate. iOS App Development certificate is for developers who want to test the app on a physical device while developing it. Distribution certificate can be AdHoc and AppStore one. These certificates are to distribute the app to other people. AdHoc is for testing to non-TestFlight testers and AppStore is for general distribution via TestFlight or AppStore. [29] There are also different types of provision profiles: iOS App Development, AdHoc, and AppStore. The purpose of them absolutely the same as for certificates, and this is logical because provision profile and certificate work together.

Fastlane "match" is responsible for automatically generating the necessary type of certificate and provision profile. A new private git repository for "match" signing credentials should be created to start signing the application with "match". The Matchfile should be added to ~/ios/fastlane directory; this file contains a git URL to a private git repository with credentials and Apple ID user name. Also, in environment variable should be saved: “FASTLANE_PASSWORD” and “MATCH_PASSWORD”. “FASTLANE_PASSWORD” – is the Apple Developer Portal password. “MATCH_PASSWORD” – is the password of the match encryption. [31] When the match is running for the first time on a new machine, it asks for the password for the Git repository, It is an extra security layer, each credential file is encrypted with “openssl”. MATCH_PASSWORD is used to decrypt the profiles. [31] Then “match” function can be created in Fastlane file.

```ruby
match(
  app_identifier: "com.shoppinglist.rn",
  team_id: "ZY6HGU2XYF",
  type: "appstore",
)
```

Figure 15. Match function to sign release app

Figure 15 contains the “match” method of Fastlane, which can be called from ~/ios/fastlane/Fastfile. This “match” function is to sign the release version of iOS app which can be delivered to such platforms as AppStore and TestFlight. The “match”
action contains three parameters to make it work: “app_identifier”, “team_id”, “type”. App identifier is a unique identifier of the application in Apple’s ecosystem. [32] Team ID is provided by Apple and is unique to a specific development team or developer. “type” defines the profile type. “appstore” type is a type for signing a build for production, for TestFlight and Appstore, where the app will be reviewed. There is a match action for production release; there are other "match" functions in Fastfile; they are absolutely the same except the type of signing.

For the first type, the “match” commands can be called locally; it can be called directly from the terminal to generate certificates and provision profiles of all necessary types. It gives the possibility to answer on all question from the “match” during the generation process and also it necessary because the right provision profile and the certificate should be assigned to specific build configuration before creating a signed build.

It is important to connect devices to Apple developer account, which should be included for development and alpha testing before generating signing credentials; otherwise, development and alpha versions will not run on these devices. The application signed with AdHoc certificate can be launched on a device which is not connected to the apple developer account.

The “match” process of generating new signing credentials can be described in several sentences. Firstly, “match” summarize value from the “Matchfile” and passing as parameters to the "match" action. Then it clones the remote git repository with credentials and decrypting data from it. "match" should verify that certificates and profiles exist and still valid. If it could not find the valid certificate with the profile, “match” logins to apple developer profile with two-step verification to generate a new profile and certificate. When provision profile and certificates were successfully created and downloaded, “match” encrypts them and push to the remote git repository. If remote repository already has valid certificates, "match" will download them and install locally to use.

As was mentioned above the correct provision profile should be connected to the build signing configuration before creating a build. The signing configuration by default is set as “Automatically manage signing” in Xcode. This option should be unchecked, and the provision profiles should be manually connected to the right build configurations. The signing configuration for each build type contains the list of created provision profiles for the specific app identifier. Xcode is connected to Apple Account and download provision profiles automatically. One of the provision profiles from the list should be chosen. Figure 16 represents how the created provision profile can be chosen from the list.
Two-factor authentication is required for apple account to be able to get access to certificates, identifiers, and provision profiles. This extra layer of security for the Apple ID helps to ensure that the owner of the account is the only person who can access its account. [33] So, this can become a problem when new certificate generates with "match" on CircleCI, as the process is automated. How can two-factor verification be passed there? Two environment variables should be added to CircleCI: FASTLANE_APPLE_APPLICATION_SPECIFIC_PASSWORD and FASTLANE_SESSION.

The CircleCI machine is not able to pass the two-step verification; a login session for Apple ID should be generated in advance. It can be generated on the local machine by running “fastlane spaceauth -u user@email.com” command. The generated value should be stored inside the FASTLANE_SESSION environment variable. This session will be reused instead of triggering a new login each time Fastlane communicates with Apple’s APIs. An Apple ID session is only valid for up to a month; this means to generate a new session have to be generated every month. [34] FASTLANE_APPLE_APPLICATION_SPECIFIC_PASSWORD is an environment variable which contains the application-specific password, which can be generated on account setting of Apple Account. This application-specific password should be generated in case if the two-factor verification is enabled and Fastlane is going to communicate with Apple Store Connect. [31]

3.3.3 Building

Fastlane can build iOS applications as was already mentioned above. There is a “gym” fastlane action for this purpose which accepts parameters. The parameters tell “gym” how iOS app should be built and for what purposes. Figure 17 represents the block of code from Fastfile. The block of code contains the” gym” action to prepare iOS application build for Beta testing or Production release.

```
gym(
    verbose: true,
    scheme: "ShoppingList",
    configuration: "Release",
    export_method: "app-store",
    clean: true,
)
```

Figure 17. Gym method for Beta release.
The first parameter for “gym” action is “verbose”. “verbose” parameter is responsible for the switching on or off the verbose mode for building. The verbose mode gives more detailed information about the build and makes it easier to find an error if build would fail. The second parameter defines the project scheme, which should be used for the build. Configuration parameter should contain the name of the build configuration. “export_method” parameter specifies the method which is used to export the archive of the application build. “clean” parameter means that the project should be cleaned before to start the building process. It aims to delete the build folders from previous builds and empty cash. This preparation step helps to start the build from scratch and not be interrupted with old data or errors. iOS Fastfile has four “gym” calls for different build configurations. They all have the same parameters, but different values of them. “verbose”, “clean” and “schema” parameter stay the same for all “gym” actions, but “configuration” and “export_method” are different. Development build has a “Debug” build configuration and “development” export method. Development export method means that it cannot be shared with anyone within Alpha or Beta or Production environments. The Alpha build has the export method “ad_hoc” which means that it can be run on resisted devices and the build configuration is “Alpha”. Beta and Production builds have absolutely the same “gym” actions, which was already described above. They are the same because the TestFlight platform is used for Beta testing, which needs “appstore” certificate and provision profile. “Release” build can be run on any devices. The “gym” action makes the same job as the build job in XCode, but also “gym” archives the build automatically when it is necessary, while the archive step should be done manually in XCode.

3.3.4 Alpha version

Alpha version of the app is supposed to be for initial app testing, before app release to TestFlight where the app will go through the review process. Alpha version should be available for a limited group of people. The new version of “Alpha” app should become available for testers immediately when a new build file is uploaded to the Alpha test platform. The alpha app has the AdHoc distribution certificate. An AdHoc Distribution Provisioning Profile lets test the built apps on devices which have been configured in Apple Developer Account. AdHoc distribution is the in-house distribution, and not registered devices cannot run the Alpha version of the app. Crashlytics is one of the platforms which supports the app distribution with AdHoc access.

3.3.4.1 Crashlytics

Crashlytics or Fabric Beta is chosen for Alpha testing. Crashlytics is a platform for distributing apps that gives a single, cross-platform toolset for iOS and Android, and easy onboarding for the testers. [35] The installation and uploading build to Crashlytics is not tricky. Crashlytics is a Fabric product. Firstly, it is necessary to sign up to Fabric, when a confirmation email is received it would have a link to the page where the platform of the application can be chosen, and Fabric desktop application for the specific project can be downloaded. The Fabric for XCode should be installed for iOS app distribution. Fabric desktop application shows the list of existing XCode project where the right one
should be chosen. The next step is that the Crashlytics service should be chosen from the list of services when the XCode project was chosen. Then the screen shows the steps that should be done to upload app to Crashlytics platform, but before it, the Crashlytics libraries should be manually downloaded and integrated to the XCode project. The libraries file should be moved to the root directory of the XCode project. The new build script should be added to the XCode project when the libraries are added to it. The new build script is generated by Fabric app and exists in “the list with steps”. The new build script is look like:

```
./Fabric.framework/run <crashlytics_api_key_here> <build_secret>.
```

Then API key should be added to info.plist file. The info.plist file is a structured text file that contains essential configuration information for bundled executable files. The configuration information should be added for Crashlytics libraries. Listing 1 represents the block of configuration code for Crashlytics in info.plist file.

```xml
<key>Fabric</key>
<dict>
    <key>APIKey</key>
    <string>{crashlytics_api_key_here}</string>
    <key>Kits</key>
    <array>
        <dict>
            <key>KitInfo</key>
            <dict/>
            <key>KitName</key>
            <string>Crashlytics</string>
        </dict>
    </array>
</dict>
```

Listing 1. Crashlytics configuration in info.plist [36]

Then Crashlytics should be initialized in AppDelegate.m file, by importing “Fabric” and “Crashlytics” libraries and add this line of code: “Fabric.with([Crashlytics.self])” inside function application. After all these steps, the app should be built with XCode in Alpha configuration, and Fabric will determine the app and will add it to the platform. The app can be distributed among testers after these steps. The testers can be added in the dashboard of Fabric page. Then testers will get invitations to install the app.

### 3.3.4.2 Fastfile

Fastlane has the Crashlytics integration, which means that there is a Fastlane action to upload ready iOS app build to Crashlytics platform.
Figure 18. Crashlytics upload

```ruby
crashlytics(path: './Crashlytics.framework/',
            api_token: ENV['CRASHLYTICS_API_TOKEN'],
            build_secret: ENV['CRASHLYTICS_BUILD_SECRET'],
            notes: '#{changelog.to_s}',
            notifications: true,
            groups: ['Group'])
```

Figure 18 represents the “crashlytics” Fastlane action to upload the Alpha iOS app build. “crashlytics” accept parameters to configure how the app should be distributed. “crashlytics_path” is the path to the Crashlytics library file. “api_token” and “build_secret” should be defined to upload the build to Crashlytics platform. These values are stored in environment variables: CRASHLYTICS_API_TOKEN and CRASHLYTICS_BUILD_SECRET. “notes” is the release notes or changelog; it is a description of changes for this build which are taken from the commits. “notifications” means that all testers should get notifications via email about new uploaded build. The group is the array of groups that should get a new build. The distribution group is named “Group” here.

3.3.5 Beta

The beta test version is an app version that should be available for a bigger group of people than Alpha. Quite often, companies have a link to their beta version of the app on its official website. It helps developers to get a response from real users about new changes. TestFlight is the apple product to distribute beta versions of iOS apps. iOS applications should also go through apple review process there to become available for the tester as in AppStore to become available for general users. It helps developers to check that the app passes apple validation and be sure that app release will not have any delays because of the failed review result.

The Apple Developer Account is needed to start working with TestFlight. A new project with bundle id of iOS application should be created in TestFlight. This is the only set up needed to start uploading a beta version of the app to TestFlight. Upload to TestFlight should happens only when new changes come to “beta” git branch.

Fastlane have integrated method to upload builds to TestFlight.
Figure 19. Fastlane upload to TestFlight method

Figure 19 represents the upload action to TestFlight for iOS app. “upload_to_testflight” is a Fastlane action to upload iOS app build to TestFlight platform. “upload_to_testflight” accepts different parameters as other Fastlane actions. It accepts four parameters here: “username”, “groups”, “distribute external” and “reject_build_waiting_for_review”. This information is enough to upload iOS app to TestFlight and submit for an app review; all other necessary information is available to TestFlight from the provision profile. There is provided the “username” of the apple developer account. “distribute external” is the parameter that defines should the build be distributed to testers outside of the development team or not. Also, “distribute external” parameter defines would the app be submitted to app review, because otherwise the app cannot be distributed to external testers within TestFlight platform.[37] “groups” is a parameter that specifies which group of external testers should get the new beta app version, if the “groups” parameter is not defined, then new beta app version would not be submitted to beta app review. “reject_build_waiting_for_review” is a parameter which expires previous build if it's 'waiting for review'. [38]

TestFlight is like a repetition stage for iOS app before AppStore, that is why the provision profile, certificate, and build configuration is the same as for production AppStore. Also, TestFlight needs AppStore certificate to distribute the app, because the application should be available for launching on multiple devices of stranger people, which devices cannot be registered in the apple certificate management system.

3.3.6 AppStore

AppStore deployment is the last deployment of new iOS app changes. It delivers the updated app to all customers. The app becomes publicly available. The deployment to AppStore is done with Fastlane tool. “delivery” is a Fastlane action to submit the app build to AppStore and AppStore review. [39]
Figure 20 represents the block of code from ~/ios/Fastlane/Fastfile, which is responsible for deploying the iOS app builds to AppStore. “submit_for_review” and “automatic_release” are self-descriptive parameters. “submit_for_review” parameter defines that build is sent automatically to review when the build is uploaded to AppStore. “automatic_release” specifies that new build should be automatically released when it will pass the Apple review. “force” parameter lets to skip the HTML preview of the app metadata and screenshots before uploading to AppStore. [40] “release_notes” is the object type of parameter where should be given the release notes for different locales. Figure 20 contains the “release_note” value as an object with one key. “default” key means that this text is used as a release note for all locales where the app is available. The “metadata_path” is the path to the folder with metadata and screenshots of the app that should be uploaded to AppStore with the new build. Also, there are such parameters as “username” and “copyright”. “username” is already a common parameter, while “copyright” is a new requirement of AppStore to upload an app. “copyright” consists of the name of the person or entity that owns the exclusive rights to the app, preceded by the year the rights were obtained.

3.3.7 config.yml and CircleCI works

config.yml contains three jobs to build and deliver the iOS app to different release platforms. There are jobs for an alpha, beta, and production deployments. There is also one job which is signing and building the app without delivery; it is used for feature-branches build. All these jobs contain the same steps, there is only one difference, is the last step in the job, which call specific build lane from ~/ios/fastlane/Fastfile.
Figure 21. CircleCI job to build and deliver iOS app to beta platform

Figure 20 represents the typical iOS job in ~/.circleci/config.yml. This specific job is aimed to build and deliver the iOS app to the beta distribution platform – TestFlight. The structure of the job is readable and understandable. Firstly, the executor machine is defined; it is macOS with a suitable version of XCode. The first step in the job is the command to increase the number of Ruby. The Ruby should be the same version, or higher than Gemfile is specified. The second step is the third-party libraries installation. The workspace with installed libraries cannot persist to iOS jobs because Node and Yarn version differ between CircleCI macOS executor and the docker container in the common job – “run_tests”. The next step is “Configure Bundler” command to update the bundler version to download necessary gems later. The gems install command is calling right after the step where the bundler was increased. The “bundle install” is the command to install gems.

The next step which is running the “npm” command – “npm run build:ios”. It is a “npm” script which solves the React Native issue. [41] React Native is not creating a
main.jsbundle for iOS automatically. The “npm” command is generating this file manually. The “npm” command contains the script:

```
"react-native bundle --entry-file ./index.js --platform ios --bundle-output ios/main.jsbundle"
```

The last step is the call of the lane from ~/ios/fastlane/Fastfile to build and deliver the iOS app. There is a command “bundle exec fastlane beta”, it is the Fastfile lane to build a beta build and distribute the build to TestFlight platform.

### 3.4 Android

Google Play Console is a developers’ part of the Play Market ecosystem. Play Market is a Google app store for Android applications. It is not necessary to publish an android app to any market to give other people the possibility to download and use the android app. The developer can share anywhere the APK file of its application, and anyone can install it. APK – is a file of build android application. If an application should be installed by the maximum number of users or the owner wants to make money on it, the app should be placed to the famous app market. Play Market is the integrated app market on the most phones with Android OS, which makes it the best place to distribute any android app.

Google Play Console is a developer-friendly platform to upload a new version of the app and distribute it to different groups of people or upload it to production. Google Play Console gives a possibility to have Alpha, Beta, and Production distribution within one native platform. There are four different tracks to distribute app: Production, Beta, Alpha, and Internal test. Internal testing, Beta, and production distribution tracks are chosen to the current CD pipeline. Also, the Play Market does not have any review stage in any deployment process. The application becomes available for customers in several seconds after success upload of a new APK file, which makes the process of app distribution much more comfortable.

#### 3.4.1 Fastlane

Fastlane should be set up before the work on android deployment would start. Fastlane helps to prepare a build and deliver the app to Play Market. First of all, the Fastlane should be installed. Figure 21 contains the command that should be run in the android directory to install Fastlane tool there.

```
sudo gem install fastlane -NV
```

Figure 22. Command to install Fastlane

The Fastlane has a specific command initialize Fastlane tools for android app. Figure 22 contains this command.
fastlane init

Figure 23. Command to set up Fastlane

The setup process will ask several questions, the package name of the application, path to JSON secret credential file and set up upload to Google Play. The package name should be provided, while the path to JSON secret credential file and setup deployment to Google play steps should be skipped, these stages would be set up manually.[42] The “fastlane init” action automatically generates a configuration based on information provided during set up. It creates the Fastlane directory where Appfile and Fastfile are created. Appfile defines global configuration information for the app. Fastfile defines the lanes that orchestrate the behavior of Fastlane. The most interesting file is ~/android/fastlane/Fastfile, which contains all necessary information to distribute the app. [42]

The primary Fastlane tool for android app is “supply”. “supply” is a tool that uploads app metadata, screenshots, and binaries to Google Play. It can also create builds for different tracks and promote builds to different release stages! [42] Setting it up requires to download a secret JSON credentials file from the Google Developers Service Account.

Service Account can be created on Google Play Console, and secret file would be generated and should be saved as a JSON file. Then Appfile should be edited, the path to this secret JSON file should be provided there. Figure 23 shows the content of edited Appfile; it contains two values: “json_key_file” and “package_name”.

```ruby
json_key_file("google-play-key.json") # Path to the json secret file
package_name("com.shoppinglist.rn") # e.g. com.krausefx.app
```

Figure 24. Android Appfile content

json_key_file – a path to google play key file should be provided which was generated after the service account was created. It allows Fastlane access to Google Play Market methods.

package_name – The package name of the application, which is going to be built and uploaded to Play Market.
3.4.2 Signing

3.4.2.1 Theory

The Play Market is the securest place to get an Android application because the app cannot be uploaded there without developer sign. The APK file should be signed by the developer to be distributed through the Play Market platform.

The signing key and Service Account should be prepared to sign the android app from Continuous Integration system. Service Account lets to access Google Play Developer Publishing API without providing developer’s credentials, which make this option perfect for using it on CI/CD platforms. [43] Upload key or Signing key is the private key that in use to sign the app bundle or APK before APK would be uploaded to Google Play. The upload key must be kept in secret.

It can be generated by this command:

```
keytool -genkey -v -keystore my-release-key.keystore -alias my-key -alias -keyalg RSA -keysize 2048 -validity 10000
```

Figure 25. Command to generate key

This command prompts for passwords for the keystore and key; it also asks information about the development team and a company who owns the app. It generates the keystore as a file called my-release-key.keystore.

Keystore is a binary file that works as a repository of certificates and private keys. An identity certificate contains the public key of a public/private key pair and other metadata identifying the owner who holds the corresponding private key. [44] The generated keystore contains a single key. The key is valid for 10000 days. The alias is a name that will be used later for signing the app. [45]

![Diagram](image)

Figure 26. Signing an app with app signing by Google Play [46]

Figure 26 represents the scheme which illustrate the work of app signing with Google Play. Google manages and protects the app's signing key and uses it to sign the APKs.
for distribution. [46] When the app is ready to be published, it should be signed. The
signing process with the Google Play in brief looks like this. The key which was used to
sign the app becomes the app’s upload key. Google verifies the developer identity by
using the upload certificate and signs the APK with the app signing key for distribution.
The signing key is the key that is used to sign APKs that should be installed on a user’s
device. [47] The upload certificate is the public certificate for the upload key that should
be held privately. The upload key should be used to sign each release so that Google
Play knows the release comes from the specific developer. [46]

3.4.2.2 Practical part

It is necessary to generate a signed APK file before uploading the APK file to Google
Play. The private signing/upload key can be generated by keytool. [45] Keytool is a
command-line tool which can generate public key / private key pairs and store them in a
Java KeyStore.

There is a special command to create a keystore. The command is represented on Figure
25. Firstly, the command asks to create a keystore password to secure the keys. The ter-

tinal will ask for some information for an identity certificate. Also, keytool asks a
password for a key; it can be the same as for keystore. This command creates keystore
as my-release-key.keystore. It would have a single key with alias my-key-alias.

Then my-release-keystore should be paced under ~/android/app folder in the project.
~/android/gradle.properties should be edited with the new info

```java
MYAPP_RELEASE_STORE_FILE=my-release-key.keystore
MYAPP_RELEASE_KEY_ALIAS=my-key-alias
MYAPP_RELEASE_STORE_PASSWORD=${MYAPP_RELEASE_STORE_PASSWORD}
MYAPP_RELEASE_KEY_PASSWORD=${MYAPP_RELEASE_KEY_PASSWORD}
```

Figure 27. Define global gradle variables

Figure 27 shows the content of ~/android/gradle.properties file, where global Gradle
variables are defined, which can be used in the gradle config to sign the app. The Gradle
config file defines build configurations that apply to all modules in the project. [48] The
last two properties should be moved to environment variables, for better security and
used direct form environment variable for Gradle config file. Next step is adding sign-
ing config to the Gradle config of the app.

~/android/app/build.gradle should be edited with next lines to sign the app during APK
build.
Figure 28. Signing config in app’s gradle build config.

Figure 28 contains the block of code from `~/android/app/build.gradle`. This block of code is responsible for APK file signing. The created global Gradle variable are in use there to define the keystore file and the key alias. The passwords for keystore and key are stored in environment variables. The preparation for creating the signed app is finished, now the signed APK file can be generated locally.

Next steps should be done to generate the signed APK file remotely. First of all, Fastlane should be set up, that already was described. The keystore is generated locally, but it can be stored in the git repository because of security issues, especially in public repository. The second problem is that once the application was signed with one key, it should always be used for all next build and upgrades. So, the copy of keystore should be saved in some secure place.

The decision is to store the keystore in the private dropbox repository; it can be also stored in any other secure cloud platforms. The keystore should be a private file in dropbox, but it should be accessed through special URL. It needs for CircleCI; then CircleCI would be able to access the keystore and download it on its virtual machine. The bash script file was created to download the keystore file and put it to the right place.

Figure 29. Bash script to download and place keystore file

Figure 29 contains the bash script to download the keystore file from dropbox and place it to the right path. `KEYSTORE` is an environment variable which contains the right path to the place where the keystore file should be saved. `SIGNING_KEY_URI` is a URL where the keystore file can be accessed and downloaded from dropbox.

There is a list of all environment variable in CircleCI which are used for the android app signing process.
• KEYSTORE – path to the keystore file.

• SIGNING_KEY_URI – URL to download the keystore file from dropbox.

• MYAPP_RELEASE_STORE_PASSWORD – password for keystore

• MYAPP_RELEASE_KEY_PASSWORD – password for the key in the keystore

3.4.3 Building

Fastlane can build an android app to signed APK file. “gradle” is the Fastlane tool which is responsible for the android build. “gradle” tool can perform all Gradle related actions, including building and testing the Android app. [15] As Android does not need to have a different type of certificate for a different distribution, one signed APK can be distributed between internal users and be delivered to production release track of Play Market. It means that build process will be the same for all distribution lanes in Fastfile. The private built lane was created to be run from the delivery-specific lanes.

Figure 30. Build lane for android app / Build lane to create a signed APK

Figure 30 contains the private build lane from ~/android/fastlane/Fastfile. This build lane is used for preparing APK file for distribution to different release tracks. The build lane is the private lane. Private lane can be called only inside Fastfile, not from the terminal or any other place by Fastlane command. [49] So “fastlane android build” command cannot be called.

The first line of the build lane is “gradle(task: ‘clean’)”. This command deletes previous build directory and artifacts from the previous build job, which can cause a problem during the upcoming build. [15] The second line is already known as the function, which takes the number of commits and assigns this number to the variable, which will define the build number.

The last line is the line where the magic happens and signed APK should be created. Fastlane “gradle” action builds the APK file of the app, several parameters are passed to this action: “task”, “build_type” and “properties”. “task” property is defined as “assemble”. “assemble” task is the aggregate task that assembles all archives in the project. So, “assemble” is the task which assembles code to create the app. “build_type” is defines which build type should be executed. As this build function prepares the APK file for distribution, the “build_type” should be “Release”. “properties” is an object where
Gradle properties can be set to be exposed to the Gradle script. The version number of the build is passed only to “properties”.

### 3.4.4 Delivery Android app to different environment

The next step is to deliver the signed APK to necessary test or production environment. “supply” is a Fastlane tool which helps to communicate with Google Play Console.

“supply” can:

- Upload new builds
- Retrieve and edit metadata, such as title and description, for multiple languages
- Upload the app icon, promo graphics and screenshots for multiple languages
- Have a local copy of the metadata in your git repository
- Retrieve version code numbers from existing Google Play tracks

“supply” needs somehow get access to the Google Play Console to do all these actions from the developer's face. The service account was created for these purposes, and its secret key file should be given to Fastlane. It is not secure to save the secret file in the Git repository. The problem was solved by saving the content of secret JSON file to an environment variable and run the script in CircleCI job which will extract the content of the environment variable to the specific file.

Google Play Console already handles different release tracks for the app. It makes developers’ life easier because different versions of the app should be uploaded to the same platform but to different tracks. Google Play Console offers to upload app to 4 different release tracks “Internal test”, “Alpha”, “Beta” and “Production”.

~/android/fastlane/Fastfile consists of 3 public lanes which deploy signed APK file of the app to different environments. There is “build” private lane which prepares a signed APK file which should be uploaded to some specific release track.

Figure 31 represents the lane from Fastfile to upload the signed APK file to Beta release track of Google Play Console. The first line of the “beta” lane runs the private lane “build” to prepare the signed APK file which can be delivered to the release platform. The second line is the Fastlane “supply” action which uploads the prepared APK file to specific release track. The APK file is uploaded to the “beta” track in the current case as it is defined in the parameters of the “supply” action. Other available options for this parameter are “production”, “alpha”, “internal” and “rollout”.

metropolia.fi/en
Other lanes in ~/android/fastlane/Fastfile are absolutely the same, the “track” parameter for “supply” method is the only difference.

config.yml has three jobs for building and deploying the android app to different platforms. Figure 32 performs the typical android job in the config.yml. This figure represents the deploy process for the beta platform, also config.yml contains internal/alpha test, and production deploying jobs. The difference between these jobs is the last line of job, it is the “lane” command from ~/android/fastlane/Fastfile. The “fastlane android internal” is the command for internal test deployment and “fastlane android deploy” is the command for production deploy of the android app.

```yaml
android_beta_deploy:
  working_directory: ~/project/android
  docker:
    - image: circleci/android:api-28-node8-alpha
  steps:
    - checkout:
      - path: ~/project
    - attach_workspace:
      - at: ~/project
    - restore_cache:
      - key: bundle-v1-{{ checksum "Gemfile.lock" }}-{{ arch }}
    - run:
      - name: Create Google Play key
        command: echo $GOOGLE_PLAY_KEY > google-play-key.json
    - run: ./download_keystore.sh
    - run:
      - name: Configure Bundler
        command: |
          echo 'export BUNDLER_VERSION=$(cat Gemfile.lock | tail -1 | tr -d ")")' >> $BASH_ENV
        source $BASH_ENV
        gem install bundler
    - run: bundle install
    - run: fastlane supply init
    - run: fastlane android beta
```

Figure 32. CircleCI job to build and deliver android app to beta platform

The job on Figure 32 runs in the docker container “circleci/android:api-28-node8-alpha”. The steps of android jobs are quite familiar with the steps of iOS jobs. The first step in the job is the code checkout to get the last changes. Then the workspace that was
persisted in “run_tests” job is attached to this job. The “Create Google Play key” command runs the script to get the google-secret key for the Service Account. It takes the value of the $GOOGLE_PLAY_KEY environment variable and copies its content to the google-play-key.json file. This JSON file is used in Appfile to provide the connection with Google Play Console to Fastlane. “./download_keystore.sh” step is the script to download the signing key from the dropbox. The signing key is necessary to generate the signed APK file.

The next step is significant; it checks what is the version of bundler is used for Gemfile.lock and installs this version on the machine. Otherwise, there is a big risk that docker image and Gemfile can have different bundler versions and gems cannot be installed successfully.

Bundler creates a consistent environment for Ruby projects by tracking and installing the right gems and versions that are needed. [50] Gem is a ruby external library. Next step installs all gems for the project, here is only a Fastlane.

“fastlane supply init” is a command which gets all app’s metadata from Google Play store, these data would be downloaded to ~/android/fastlane/metadata/android. The app’s metadata would be downloaded if the app has been already created in the Google Play developer console.

Fastlane tool can be used with apps which have been already created in Google Play Console, and the first signed APK file of the app has been uploaded to one of the released tracks. The process of creating an app in Google Play Console includes answering on questions about the app, uploading the screenshots of the app, its icon and the signed APK file.

4 Results and Discussions

This section describes the result of the create CD pipeline for cross-platform mobile application. The second subsection describes how the CD can be improved and could be changed potentially to enhance the usability of the deployment process.

4.1 Summary of Results and Evaluations of Results

The aim of this thesis is to create the Continue Deployment pipeline for cross-platform mobile application and describe the technologies for implementing such pipeline. The working version of such pipeline was created, and it contains such functionalities as:

- running unit test for the app before it would be compiled to iOS or Android native app.
• Android and iOS apps have Alpha, Beta and Production automated deployments

• Android and iOS apps can be built/compiled during pipeline execution

• Android and iOS apps are signed during the build process automatically

• Deployment to release platform executes automatically if all previous steps are passed

The created pipeline contains several build plans, but all of them have the same scheme: firstly, run unit tests, secondly, set up the environment for app building, installing third-party libraries and downloading additional secret files, thirdly, build and sign the app, fourthly, deploy the app to release platform. Figure 33 visually describes this process.

![UML Diagram for the build plan scheme](image)

Figure 33. UML Diagram for the build plan scheme

The whole set up of the pipeline consists of three files: ~/.circleci/config.yml, ~/ios/Fastlane/Fastfile and ~/android/fastlane/Fastlane. The configurations of these files were described in the Implementation part of the document. The full version of configuration files can be found in appendixes: Appendix 1, Appendix 2 and Appendix 3.

The result of the CD pipeline satisfies the requirements which has been defined in the beginning of the thesis. It works, it builds and deploys apps by committing the changes to the specific branch.

### 4.2 Future Improvements

The CD pipeline works and fulfills the minimum requirements for the continuous-deployment pipeline, but some aspects were discovered during the process of development that should be added or improved for the fully comfortable experience of developing and deployment the app. The first thing is icon badges for different release platforms.
Figure 34. App Icons from Alpha and Beta test platforms.

Figure 34 represents two app icons of the same app, but one from the TestFlight (Beta) and another one from the Crashlytics (Alpha). They look absolutely the same, and the only difference is the red circle in the beginning of the app name under one of the icons. The red circle means that this app from TestFlight, but this is not as obvious and easy to recognize. It would be much more helpful to have a specific badge for each test app with the version and name of the environment as in Figure 35.

Figure 35. Badges for App Icons (https://github.com/solinor/react-native-ci/blob/master/docs/ReactFin-land-RN-CICD.pdf)
The second improvement is to find and implement the process of giving a version number to the app automatically. At this moment, the app version is always the 1.0, and this is okay at the stage of developing the app, but the app should have the different version number for the second production release. The new version number can be taken from the specific file and be assigned everywhere when it is necessary or to find some other way. The laziest way is to change the version manually in all config files for iOS and Android, but there is a big risk that some file or someplace would be forgotten to be changed manually. It is safer to make it automatically.

The release notes in the current version of pipelines is taken from last commits since the last release. It is the fastest and easiest way, but the commits are not always explicit and can adequately describe the new changes. The release from commits can be kept for internal or Alpha testing, but the release note should be proper descriptive and clear for customers in Beta and Production environments. It would be nice to add the functionality that release notes for Production and Beta would be taken from the specific file where release changes were appropriately described.

The current version of the pipeline cannot take screenshots automatically. Taking screenshots of the app is a time-consuming task. Fortunately, screenshots should not be updated for each new release; it is necessary only when the User Interface (UI) would be changed significantly. The Fastlane tool has the functionality to take screenshots automatically, so the process can be automated to reduce manual work for app deployment.

5 Conclusion

The main objective of this thesis was to investigate the methods and technologies to implement a Continuous Deployment pipeline for cross-platform mobile applications and create it. The technologies, its details, and the aspects of its implementation were detailed described in this thesis. The most important is that the signing process for Android and iOS were explained. In practice, to create CD for cross-platform mobile application is to create two continuous deployment pipelines for Android app and iOS app and make them run together.

The goal of the thesis was reached; the created pipeline performs all necessary steps: test, build, and deploy of the app. The process of learning and creating the pipeline was not easy. However, the result is more than satisfying. The quite interesting detail was noticed during the thesis work, the difficulty level of Android and iOS application deployment are on absolute different levels. Android methods to sign and deploy the app are much easier than iOS. When the investigation of Android app signing could take one to two days, the research of iOS app signing could take a week or even more. Such miscalculation of time-consuming tasks made the implementation period of the pipeline much longer than it was expected. Otherwise, I am satisfied with the gained result.
The current mobile application and its CD/CI configuration files can be found in GitHub repository - https://github.com/aelittaezugbayashopping-list-react-native. The Android version of the app can be downloaded by this link https://play.google.com/store/apps/details?id=com.shoppinglist.rn. The iOS version of the app can be downloaded as Beta in TestFlight by the link https://testflight.apple.com/join/oRqManL9.
References


14 Google Inc. Fastlane; iOS Beta deployment using Fastlane; Best Practices; Incrementing the build number. [online] 2019. [cited date 2019 July 20]; Available from: https://docs.fastlane.tools/getting-started/ios/beta-deployment/


17 Google Inc. Fastlane; iOS Beta deployment using Fastlane; Release Notes; Prompt for changelog. [online] 2019. [cited date 2019 July 20]; Available from: https://docs.fastlane.tools/getting-started/ios/beta-deployment/


21 CircleCI. Writing YAML. Overview. [online]. [cited date 2019 July 20]; Available from: https://circleci.com/docs/2.0/writing-yaml/


What is a provisioning profile & code signing in iOS? [online] 10 April 2018. [cited date 2019 July 20]; Available from: https://medium.com/@abhimuralidharan/what-is-a-provisioning-profile-in-ios-77987a7c54c2


Apple Inc. Continuous Integration. Application specific passwords. [online] [cited date 2019 July 20]; Available from: https://docs.fastlane.tools/best-practices/continuous-integration/#application-specific-passwords


Google Inc. Fastlane. Upload to TestFlight. Parameters. [online] [cited date 2019 July 20]; Available from: https://docs.fastlane.tools/actions/upload_to_test-flight/#parameters

Google Inc. Fastlane. Deliver. [online] [cited date 2019 July 20]; Available from: https://docs.fastlane.tools/actions/upload_to_app_store/

Google Inc. Fastlane. Deliver. Parameters. [online] [cited date 2019 July 20]; Available from: https://docs.fastlane.tools/actions/deliver/#parameters


49 Google Inc. Fastlane. Lanes. Private Lanes. [online] [cited date 2019 July 20]; Available from: https://docs.fastlane.tools/advanced/lanes/#private-lanes

50 Bundler. [online] [cited date 2019 July 20]; Available from: https://bundler.io/

default_platform(:ios)

platform :ios do
  before_all do
    setup_circle_ci
  end

  desc "Development build"
  lane :dev do |options|
    build_number=number_of_commits()
    increment_build_number(build_number: build_number)
    match(
      app_identifier: "com.shoppinglist.rn",
      team_id: "ZY6HGUX2XYF",
      type: "development",
    )
    gym(
      scheme: "ShoppingList",
      configuration: "Debug",
      export_method: "development",
      clean: true,
    )
  end

  desc "Submit a new build to iTunes Connect Testflight"
  lane :beta do |options|
    changelog_from_git_commits
    match(
      app_identifier: "com.shoppinglist.rn",
      team_id: "ZY6HGUX2XYF",
      type: "appstore",
    )
    increment_build_number(build_number: latest_testflight_build_number + 1)
    gym(
      scheme: "ShoppingList",
      configuration: "Release",
      export_method: "app-store",
      clean: true,
    )
    upload_to_testflight(
      username: "aelittae@gmail.com",
      distribute_external: true,
      groups: "group",
      reject_build_waiting_for_review: true,
    )
  end

  desc "Submit a new Alpha Build to Fabric"
  lane :alpha do |options|
    changelog = sh("git log --pretty=format:'* %s <%an>%n' --first-parent --
                   abbrev-commit #{ENV['GIT_PREVIOUS_SUCCESSFUL_COMMIT'] || 'HEAD^^^^^'}..HEAD")
    build_number=number_of_commits()
    increment_build_number(build_number: build_number)
    match(
      app_identifier: "com.shoppinglist.rn",
      team_id: "ZY6HGUX2XYF",
      type: "Fabric",
    )
    gym(
      scheme: "ShoppingList",
      configuration: "Release",
      export_method: "Fabric",
      clean: true,
    )
    # Fabric specific
    # upload_to_fabric(
    #   token: "your_fabric_token",
    #   groups: "group",
    #   reject_build_waiting_for_review: true,
    # )
  end
```ruby
app_identifier: "com.shoppinglist.rn-alpha",
team_id: "ZY6HGU2XYF",
type: "adhoc",
}
gym(
  scheme: "ShoppingList",
  configuration: "Alpha",
  export_method: "ad-hoc",
  clean: true,
}
crashlytics(
  crashlytics_path: './Crashlytics.framework/',
  api_token: ENV['CRASHLYTICS_API_TOKEN'],
  build_secret: ENV['CRASHLYTICS_BUILD_SECRET'],
  notes: "#{changelog.to_s}",
  notifications: true,
  groups: ['Group']
}
end

desc "Push a new release build to the App Store"
lane :release do
  increment_build_number(xcodeproj: "ShoppingList.xcodeproj")
  match(
    app_identifier: "com.shoppinglist.rn",
    team_id: "ZY6HGU2XYF",
    type: "appstore",
  }
  gym(
    verbose: true,
    scheme: "ShoppingList",
    configuration: "Release",
    export_method: "app-store",
    clean: true,
  }
  deliver(
    submit_for_review: true,
    automatic_release: true,
    force: true,
    metadata_path: "./metadata",
    username: "aelittae@gmail.com",
    release_notes: {'default': "Simple ShoppingList app"},
    copyright: "2019 Aelitta Ezugbaya"
  }
end
end
```
default_platform(:android)

platform :android do
  desc "Runs all the tests"
  lane :test do
    gradle(task: "test")
  end

  desc 'Build the Android application.'
  private_lane :build do
    gradle(task: 'clean')
    build_number=number_of_commits()
    gradle(task: 'assemble', build_type: 'Release', properties: {"VERSION_CODE" => build_number})
  end

  desc "Just build an app"
  lane :development do
    build
  end

  desc "Submit a new Internal Test Build to Crashlytics Internal Test"
  lane :internal do
    build
    supply(track: 'internal')
  end

  desc "Submit a new Beta Build"
  lane :beta do
    build
    supply(track: 'beta')
  end

  desc "Deploy a new version to the Google Play"
  lane :deploy do
    gradle(task: "clean assembleRelease")
    build
    supply(track: 'production')
  end
end
~/.circleci/config.yml

version: 2.1
jobs:
  build:
    working_directory: ~/project
docker:
    - image: circleci/node:10
steps:
  - checkout
  - restore_cache:
    key: node-v1-{{ checksum "package.json" }}-{{ arch }}
  - run: npm install
  - run: npm run test
  - save_cache:
    key: node-v1-{{ checksum "package.json" }}-{{ arch }}
    paths:
    - node_modules
  - persist_to_workspace:
    root: ~/project
    paths:
    - node_modules
  - store_test_results:
    path: ~/project/junit.xml

android:
  working_directory: ~/project/android
docker:
    - image: circleci/android:api-28-node8-alpha
steps:
  - checkout:
    path: ~/project
  - attach_workspace:
    at: ~/project
  - restore_cache:
    key: bundle-v1-{{ checksum "Gemfile.lock" }}-{{ arch }}
  - run:
    name: Configure Bundler
    command: |
      echo 'export BUNDLER_VERSION=$(cat Gemfile.lock | tail -1 | tr -d "")' >> $BASH_ENV
      source $BASH_ENV
      gem install bundler
  - run: bundle install
  - save_cache:
    key: bundle-v1-{{ checksum "Gemfile.lock" }}-{{ arch }}
    paths:
    - vendor/bundle
  - run:
    name: Create Google Play key
    command: echo $GOOGLE_PLAY_KEY > google-play-key.json
  - run: ./download_keystore.sh
  - run: fastlane supply init
  - run: fastlane android development

android_internal_deploy:
  working_directory: ~/project/android
docker:
- image: circleci/android:api-28-node8-alpha
  steps:
  - checkout:
    path: ~/project
  - attach_workspace:
    at: ~/project
  - restore_cache:
    key: bundle-v1-{{ checksum "Gemfile.lock" }}-{{ arch }}
  - run:
    name: Create Google Play key
    command: echo $GOOGLE_PLAY_KEY > google-play-key.json
  - run:
    name: Configure Bundler
    command: |
      echo 'export BUNDLER_VERSION=$(cat Gemfile.lock | tail -1 | tr -d "")' >> $BASH_ENV
      source $BASH_ENV
      gem install bundler
  - run: bundle install
  - run: fastlane supply init
  - run: fastlane android internal

android_beta_deploy:
  working_directory: ~/project/android
  docker:
    - image: circleci/android:api-28-node8-alpha
  steps:
  - checkout:
    path: ~/project
  - attach_workspace:
    at: ~/project
  - restore_cache:
    key: bundle-v1-{{ checksum "Gemfile.lock" }}-{{ arch }}
  - run:
    name: Create Google Play key
    command: echo $GOOGLE_PLAY_KEY > google-play-key.json
  - run:
    name: Configure Bundler
    command: |
      echo 'export BUNDLER_VERSION=$(cat Gemfile.lock | tail -1 | tr -d "")' >> $BASH_ENV
      source $BASH_ENV
      gem install bundler
  - run: bundle install
  - run: fastlane supply init
  - run: fastlane android internal

android_production_deploy:
  working_directory: ~/project/android
  docker:
    - image: circleci/android:api-28-node8-alpha
  steps:
  - checkout:
    path: ~/project
  - attach_workspace:
    at: ~/project
  - restore_cache:
    key: bundle-v1-{{ checksum "Gemfile.lock" }}-{{ arch }}
  - run:
    name: Create Google Play key
    command: echo $GOOGLE_PLAY_KEY > google-play-key.json
  - run:
    name: Configure Bundler
    command: |
echo 'export BUNDLER_VERSION=$(cat Gemfile.lock | tail -1 | tr -d"")' >> $BASH_ENV
  source $BASH_ENV
  gem install bundler
  - run: bundle install
  - run: fastlane supply init
  - run: fastlane android deploy

ios_development:
  macos:
    xcode: "10.2.1"
  working_directory: ~/project

shell: /bin/bash --login -o pipefail

steps:
  - checkout
  - run:
    name: set Ruby version
    command: echo "ruby-2.4" > ~/.ruby-version
  - restore_cache:
    key: node-v1-{{ checksum "package.json" }}-{{ arch }}
  - run: yarn
  - save_cache:
    key: node-v1-{{ checksum "package.json" }}-{{ arch }}
    paths:
      - node_modules
  - restore_cache:
    key: bundle-v1-{{ checksum "ios/Gemfile.lock" }}-{{ arch }}
  - run:
    name: Configure Bundler
    command: |
      echo 'export BUNDLER_VERSION=$(cat Gemfile.lock | tail -1 | tr -d"")' >> $BASH_ENV
      source $BASH_ENV
      gem install bundler
    working_directory: ios
  - run:
    command: bundle install
    working_directory: ios
  - save_cache:
    key: bundle-v1-{{ checksum "ios/Gemfile.lock" }}-{{ arch }}
    paths:
      - vendor/bundle
  - run:
    command: bundle exec fastlane dev
    working_directory: ios

ios_beta:
  macos:
    xcode: "10.2.1"
  working_directory: ~/project

shell: /bin/bash --login -o pipefail

steps:
  - checkout
  - run:
    name: set Ruby version
    command: echo "ruby-2.4" > ~/.ruby-version
- run: yarn install

- run:
  name: Configure Bundler
  command: |
    echo 'export BUNDLER_VERSION=$(cat Gemfile.lock | tail -1 | tr -d "\n")' >> $BASH_ENV
    source $BASH_ENV
    gem install bundler
  working_directory: ios

- run:
  command: bundle install
  working_directory: ios

- save_cache:
  key: bundle-v1-{: (checksum "ios/Gemfile.lock")}-{: (arch) }
  paths:
    - vendor/bundle
- run:
  command: npm run build:ios

- run:
  command: bundle exec fastlane beta
  working_directory: ios

ios_alpha:
  macos:
    xcode: "10.2.1"
  working_directory: ~/project

shell: /bin/bash --login -o pipefail

steps:
  - checkout
  - run:
    name: set Ruby version
    command: echo "ruby-2.4" > ~/.ruby-version

  - run: yarn install

  - restore_cache:
    key: bundle-v1-{: (checksum "ios/Gemfile.lock")}-{: (arch) }

    - run:
      name: Configure Bundler
      command: |
        echo 'export BUNDLER_VERSION=$(cat Gemfile.lock | tail -1 | tr -d "\n")' >> $BASH_ENV
        source $BASH_ENV
        gem install bundler
      working_directory: ios

    - run:
      command: bundle install
      working_directory: ios

    - save_cache:
      key: bundle-v1-{: (checksum "ios/Gemfile.lock")}-{: (arch) }
      paths:
        - vendor/bundle

    - run:
      command: npm run build:ios

    - run:
      command: bundle exec fastlane alpha
working_directory: ios

ios_production:
  macos:
    xcode: "10.2.1"
    working_directory: ~/project

shell: /bin/bash --login -o pipefail

steps:
- checkout
- run:
  name: set Ruby version
  command: echo "ruby-2.4" > ~/.ruby-version
- run: yarn install
- restore_cache:
  key: bundle-v1-{{ checksum "ios/Gemfile.lock" }}-{{ arch }}
- run:
  name: Configure Bundler
  command: |
    echo 'export BUNDLER_VERSION=$(cat Gemfile.lock | tail -1 | tr -d " ")' >> $BASH_ENV
    source $BASH_ENV
gem install bundler
    working_directory: ios
- run:
  command: bundle install
  working_directory: ios
- save_cache:
  key: bundle-v1-{{ checksum "ios/Gemfile.lock" }}-{{ arch }}
  paths:
  - vendor/bundle
- run:
  command: npm run build:ios
- run:
  command: bundle exec fastlane release
  working_directory: ios

workflows:
  version: 2
  node-android-ios:
    jobs:
    - build
    - ios_development:
      requires:
      - build
      filters:
        branches:
        ignore:
        - master
        - beta
        - production
    - ios_alpha:
      requires:
      - build
      filters:
        branches:
        only:
        - master
    - ios_beta:
requires:
  - build
filters:
  branches:
    only:
      - beta
      - production
  - ios_production:
      requires:
        - build
      filters:
        branches:
          only:
            - production
        - production
  - android:
      requires:
        - build
      filters:
        branches:
          ignore:
            - master
            - beta
            - production
      - android_internal_deploy:
          requires:
            - build
          filters:
            branches:
              only:
                - master
          - android_beta_deploy:
              requires:
                - build
              filters:
                branches:
                  only:
                    - beta
          - android_production_deploy:
              requires:
                - build
              filters:
                branches:
                  only:
                    - production