Mobile CI/CD

Lukas Baronyai, 01326526, lukas.baronyai@tuwien.ac.at

Abstract—How does an ideal CI/CD pipeline in the mobile (Android) world look like, what are existing best practises, what are possible organisationalen impacts

Index Terms—Mobile Continuous Integration, Mobile Continuous Delivery, Android.

1 Introduction

Here comes the introduction - What's the motivation, what are the reasons of this

2 Classical CI/CD

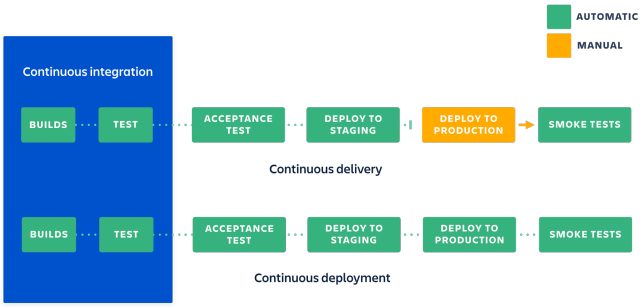

Fig. 1. CI & CD[1]

In order to look into mobile approaches to CI/CD, it is first necessary to have a quick look into the "classical" theory. The original concept was heavily shaped by Kent Beck and Ron Jeffries in the context of their concept of Extreme Programming [1], [2]

2.1 Continuous Integration

In his text about Continuous Integration Martin Fowler defined it as follows:

> *"A software development practice where members of a team integrate their work frequently, usually each person integrates at least daily - leading to multiple integration per day. Each integration is verified by an automated build (including test) to detect integration errors as quickly as possible." [3]*

According to this, the typical workflow in a CI setup looks like the following. During the whole process there is always a high emphasis on monitoring the code state, checking (and preventing) broken code as often as possible. This process replaced the formerly used first-implement-then-integrate approach by integrating the code on a regular (daily) base.

- Checkout master branch of repository
- Create working copy
- Add changes
- Check local build
- Verify integrity with local tests
- Downmerge with master branch
- Upmerge with master branch
- Automated build at server
- Automated testing at server

Additional to this, he gave in the same text 9 practices to use CI. This list will be used in the following as a tool to analyze existing practices and especially to apply it for mobile CI/CD.

1) Maintain a Single Source Repository
2) Automate the Build
3) Make Your Build Self-Testing[2]
4) Everyone Commits Every Day
5) Every Commit Should Build the Mainline on an Integration Machine
6) Keep the Build Fast
7) Test in a Clone of the Production Environment
8) Make it Easy for Anyone to Get the Latest Executable
9) Everyone can see what's happening

2.2 Continuous Delivery

Coming from CI, Continuous Delivery builds upon this and adds additionally a manual deployment step to the production environment. So after every test runs through positively, the setup is ready to deploy by a click on a button at any time. But it still has a final human instance in between the push to the repository and the release.

2.3 Continuous Deployment

Removing the last human component in this process then leads to Continuous Deployment, making the full process fully automated. Each push the the master branch then automatically leads to a release deployment if the whole pipeline does not detect any failed tests or broken builds.

1. https://wac-cdn.atlassian.com/dam/jcr:b2a6d1a7-1a60-4c77-aa30-f3eb675d6ad6/ci%20cd%20asset%20updates%20.007.png?cdnVersion=508

2. https://martinfowler.com/bliki/SelfTestingCode.html

Disregarding if the last step is now fully automated or only partially with a final human component, I extend the mentioned checklist with:

10) Automate Deployment

3 Mobile CI/CD

Coming now from the ”classical” CI/CD approach, it is not possible to instantly apply the known model to the mobile world, it needs some adoption. Before we do so, we have to consider the following points:

- **Mobile application have a high UI focus**
  UI tests are always more expensive to create than for ”standard” code - and this is even more valid for mobile applications. Additionally from the known challenges of desktop applications like clickable fields, listener states, dependencies between views & co, a mobile device introduces complexity

- **Mobile applications have per default a build tool**
  Both iOS (*XCode*) and Android (*AndroidStudio* & *gradle*) come along with their own build system and IDE, making the question if the application is build manually or automated with a tool pointless.

- **There is no standard production environment**
  Due to the nature of the mobile device world, there is no standard device towards a deployment can happen. Especially but not only with Android there is a huge variety of screen sizes and operating system versions to be handled. Therefore automated testing has to be a compromise between covered variations and invested efforts. There are of course attempts to test with assuming the maximum coverage of variations, but it can never be sufficient as for e.g. a server software.

- **Emulation is expensive**
  Another consequence of the variety of device types, testing with real hardware is a pain and lead to the usage of emulators. But emulation is expensive in terms of time and resources, especially since due to the high share of required UI tests, a majority of testing can not be done independent from the (emulated) mobile OS.

3.1 Adopted CI/CD model

As consequence to the mentioned points, we will adopt the practise checklist of Fowler by completely removing 2) (inherited to the build tools) and changing 7) to ”Test in a Emulated Environment”.

3.2 Googles CI/CD tools

Google provides many frameworks and guidelines which can be pretty overwhelming - this and the next section is therefore dedicated to give an overview of these and summarize which options there are to implement a proper CI/CD pipeline within the Android world.

3.2.1 Android Emulator

Bundled within the Android SDK comes the Android Emulator[3], which allows the emulation of a virtual device on the local machine. It supports various images, called Android Virtual Devices (AVD), with different configuration options. Once launched it is considered as a real device by `adb` and can therefore used as deployment target. The tool further provides the option to launch without an UI and to be used for testing on a server.

3.2.2 Firebase Test Lab

With Firebase Test Lab[4] Google offers a cloud-based device farm with virtual and real devices. It supports two different testing methods: Instrumentation (see section **??**) and Robo (basic test type that simulates real user interactions) Tests. Test results are provided with additional logs, videos and screenshots. Firebase Test Lab allows 10 tests per day for virtual devices and 5 for physical devices in it’s free version (”Spark Plan”) - but some CI/CD cloud services like Bitrise included the service within their contingents.

3.2.3 Google Play Deploy

Google offers via the Play Store console a REST API to deploy compiled APKs and AABs which can be used in an CD setup. The API is secured via the according Service account. Additionally to the (not prefered way) direct release track it is possible to deploy to the `alpha`, `beta` or `internal` tracks[5].

3.2.4 Available CI/CD Server

In order to properly utilize a mobile CI/CD pipeline there are many estabilshed server provider around - since it would exceed the scope of this paper to make a full comparison of these service providers, we limit it to a selection of the most commons:

- Bitrise[6]
- TeamCity[7]
- Travis CI[8]
- Jenkins[9]
- Bamboo[10]
- GitLab CI/CD[11]
- CircleCI[12]

3.3 Android testing

One of the most crucial steps within a CI/CD pipeline is the testing part. Again, there a many tools around for testing different levels of abstractions and granularity - this paper will foremost focus on the recommended toolchain by the Android project. [4]

3. https://developer.android.com/studio/run/emulator
4. https://firebase.google.com/docs/test-lab/android/overview
5. https://developers.google.com/android-publisher/tracks
6. https://www.bitrise.io/
7. https://www.jetbrains.com/teamcity/
8. https://travis-ci.org/
9. https://jenkins.io/
10. https://www.atlassian.com/software/bamboo
11. https://about.gitlab.com/product/continuous-integration/
12. https://circleci.com/

Similar to existing best practices in the ”classical” software world, there is a test-driven-development approach, but adopted for a mobile approach. As can be seen in figute 2 there are two cycles: unit and UI. Since UI testing usually consumes far more time than unit testing, these two are seperated from each other.

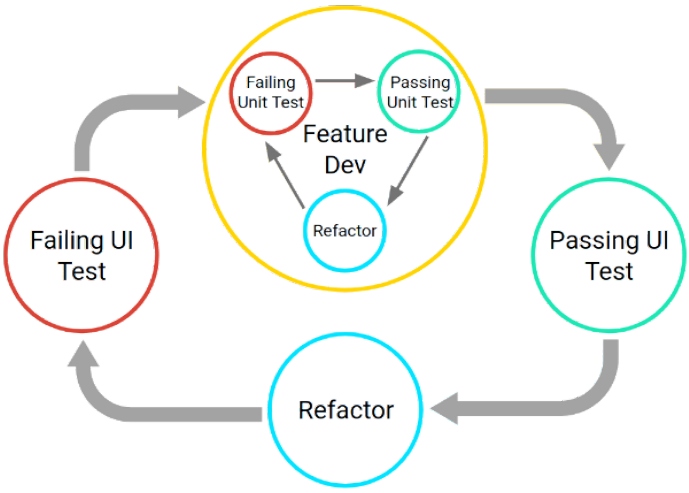

Fig. 2. The two cycles associated with iterative, test-driven development[14]

Google also defines three types of test categories: (see figure 3)

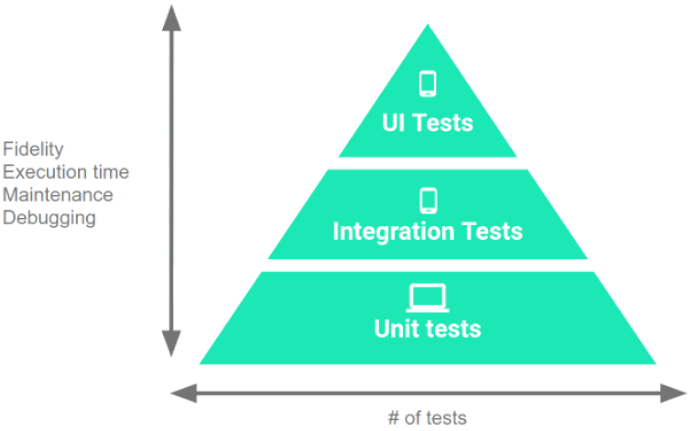

Fig. 3. The Testing Pyramid, showing the three categories of tests that should be included in an app's test suite[16]

- **Unit tests - Small tests**
  These kind of tests validate the behavior of a feature on an atomic level and are highly focues. Around 70% of the test base should consist of small test, typical frameworks are JUnit, Mockito and PowerMock.
- **Integration tests - Medium tests**
  The next level of testing are the medium/integration tests which are testing the integration of either different level of the the stack within a module or of different modules. The recommendation is to have around 20% of tests for integration, the toolchain provides Robolectric for this.
- **UI tests - Large tests**
  On top of the testing setup are the large tests, the UI tests. They are used to test end-to-end flows including the interaction of the user with the app through multiple stack level and modules. UI tests are usually very long running flows which have a high ressource impact. It is recommended to have 10% UI Test, Google provides Espresso and UI Automator on the tool side.

For a more detailed overview of the different test size see 4.

Feature	Small	Medium	Large
Network access	No	localhost only	Yes
Database	No	Yes	Yes
File system access	No	Yes	Yes
Use external systems	No	Discouraged	Yes
Multiple threads	No	Yes	Yes
Sleep statements	No	Yes	Yes
System properties	No	Yes	Yes
Time limit (seconds)	60	300	900+

Fig. 4. Test Sizes[17]

Additional to the different test types and sizes, there are also two types of tests based on the environment where they have to be executed:

- Normal **tests**
  are executed by the local machine and the JVM and do not require any dependencies to the Android OS. Test resources are located under `test`.
- **Instrumentation test**
  need for execution a mobile device and have a dependency to the Android OS. This device can be physical, virtual (by the Emulator) or simulated (by Roboelectric). Test resources are located under `androidTest`.

Based on this overview of testing classification, the following subsections will give an overview of the tools provided by Google to cover those different levels:

3.3.1 JUnit, Mockito & AndroidJUnit4

The standard JUnit[18] framework is the most obvious solution to build unit tests and is together with Mockito available for use in Android (with JUnit version 4). It is possible to access Android resources within an unit test by configuring gradle with `unitTests.includeAndroidResources`.

14. https://developer.android.com/images/training/testing/testing-workflow.png
16. https://developer.android.com/images/training/testing/pyramid.png
17. https://testing.googleblog.com/2010/12/test-sizes.html
18. https://junit.org/junit4/

It is further possible to write instrumented JUnit tests by using the `AndroidJUnit4` test runner (on which e.g. Espresso and UIAutomator relay on). In order to isolate the execution of these tests and to minimize possible shared states between those runs, there is the Android Test Orchestrator[19] available.

3.3.2 Robolectric

Robolectric[20] is a framework for unit and integration testing without relaying on a emulator or physical device - instead it uses a simulated device by wrapping the Android framework. This way it can provide far more faster test executions than with instrumentation tests.

3.3.3 Espresso

Espresso[21] provides instrumented UI tests with a relatively simple API to use. Next to standard situation with handling UI components, it is able to handle WebViews, idling resources, intents and even multi processes. Bundled with the *AndroidStudio* comes also the **Espresso Test Recorder** which simplifies the automatic creation by a great extend - by simply recording an use case.

3.3.4 UI Automator

The UI Automator[22] framework provides functionality for cross-app UI testing, including interaction between user apps and system apps (e.g. opening the setting menu or app launcher). It further lets tests access hardware functionality such as pressing the Back, Home or Menu buttons or using the volume buttons.

3.3.5 Android Jetpack Test

At the I/O18 it the new Android Jetpack component "Android Test"[23] was announced, which aims at unifying the different tools and framework together in one center API within the *AndroidX* package. At the time of writting this paper, the following tools are fully integrated: Espresso, UI Automator and AndroidJUnitRunner. Robolectric got a major update (v4) for the integration with AndroidX, but is no finished yet.

> **Note:** Google recently (I/O18) also announced "Project Nitrogen" which intents to remove any restrictions of tests to runtime environment - but there are no further information available at the time of writing this paper.

3.3.6 Outside of the Google ecosystem

Of course there are many more testing frameworks and tools out there which are not in a direct relation to Google and are not in the standard toolchain. Some well-known examples are:

- Appium[24]
- Detox[25]
- Calabash[26]
- screenshot-tests-for-android[27]

19. https://developer.android.com/training/testing/junit-runner
20. http://robolectric.org/
21. https://developer.android.com/training/testing/espresso
22. https://developer.android.com/training/testing/ui-automator
23. https://developer.android.com/training/testing/
24. http://appium.io/
25. https://github.com/wix/detox
26. http://calaba.sh/
27. https://facebook.github.io/screenshot-tests-for-android/

4 Mobile CI/CD example: AllAboutApps pipeline

After looking into what options there are to define a CI/CD pipeline for mobile applications, this section will now evaluate this with an existing pipeline at AllAboutApps, how a concrete implementation looks like and how it can be improved to fully apply as CI/CD pipeline.

4.1 Current state

The current setup is based on Bitrise and an automated Google Play Store deployment. The current workflow/pipeline looks as follows:

- Checkout `master` branch
- Create `feature` branch
- Implement feature, commit as atomic as possible
- Local builds
- Bitrise trigger for push: `debug` build
- If feature fulfills internal *Definition of Done*: pull request
- After pull request approval: merge to `master`
- Bitrise trigger for `master`: release & preview build
- If everything successful: upload to `internal` or `alpha` track of Google Play
- QA is testing last master build
- After QA & customer approval: promotion to `release` track in Google Play

As we can see, this workflow already covers most of the CI/CD steps, but misses some crucial points. In order to analyze how this can be improved to a fully implemented CI/CD pipeline, we will use the previously introduced adopted checklist of Fowler:

1) **Maintain a Single Source Repository**
   - ☑ `Git` is common practice at the company

2) **Make Your Build Self-Testing**
   - ☒ No automated testing
   - ☒ No testing culture
   - ☐ Bitrise would support it

3) **Everyone Commits Every Day**
   - ☑ Common practice at the company

4) **Every Commit Should Build the Mainline on an Integration Machine**
   - ☑ Yes by Bitrise push triggers

5) **Keep the Build Fast**
   - ☑ Inherited due to the `Gradle` build system for Android

6) **Test in an Emulated Environment**
   ☒ No automated testing
   ☐ Bitrise would support it (all of UI, integration and unit testing)

7) **Make it Easy for Anyone to Get the Latest Executable**
   ☑ Yes, by Bitrise build artifacts

8) **Everyone can see what's happening**
   ☑ Covered by Bitrise - everyone has access to the current build states of the projects

9) **Automate Deployment**
   ☑ Yes, automated deployment to Google Play `internal` or `alpha` track, manual deployment to `release`

4.2 Missing steps

As it can be seen in listening 4.1, the biggest issue is that there is currently **no automated testing** in the existing pipeline. Additionally a requirement analysis pointed out that next to testing there should be a **static code analysis** (`kklint`) introduced in the pipeline.

In order to fix this, the company developed a prototype pipeline for an existing project which covers all necessary steps of a full functional CI/CD pipeline. In order to do so, it was necessary to define limitations and a set of statements:

Statement 1. UI tests are very long running workflows on Bitrise

Statement 2. Unit tests are relatively short running workflows on Bitrise

Statement 3. Concurrent builds are limited by 3 due to the Bitrise billing limits.

Statement 4. Pull requests only have 1 trigger at Bitrise, there is (currently) no trigger when updating a pull requests.

Statement 5. Bitrise includes Firebase Test Lab runs, which are only limited by the maximal build duration.

Statement 6. Changing the pipeline of an existing project towards a full CI/CD pipeline shall not break the existing build without adding failing tests.

Statement 7. Bitrise supports triggered and scheduled builds.

Statement 8. The major advantage of Bitrise is the huge supply of Build Step Plugins.[28]

Statement 9. Android builds are done with several build variants. (debug/release, production/staging, etc).

Statement 10. UI requirements are usually changing very fast in projects.

TABLE 1
Workflow overview

Workflow	`debug`	`integration`	`release`	`preview`
Trigger	push to feature	scheduled daily	tag on master	push to master & pull requests
Duration	∼15min	> 1h	> 1h	∼10min
Testtype	unit & klint	integration & UI	full test suite	unit & klint
Build Variant	debug	debug	release	preview
Post build	-	-	Upload to Google Play `internal/alpha` track	-

4.3 Updated CI/CD pipeline

From the conclusion of these statements, there were these workflows defined:

1) **`debug`**
   Runs on every push on feature branches and triggers only unit tests and klint checks. Due to statement 6 the klint checks per default are configured to not fail the build on reporting errors.
2) **`integration`**
   Can only run each nights (using Bitrise's scheduling feature) due to its very long build duration (can even be over an hour). It uses Firebase Robo & UI testing, Espresso and Robolectrics.
3) **`release`**
   Triggered by a `git tag` and runs each available test suite, including unit, integration and UI tests, before building a release build and uploading it automated to the Google Play `internal/alpha` track. The final release is done manually.
4) **`preview`**
   On each pull request to the master branch and each push to the master itself the `preview` workflow is triggered, running all unit tests and klint checks.

The full configuration file of Bitrise is referred in the appendix A

4.3.1 Additional remarks & recommendations

Due to the nature of the company being an agency instead of doing product development, the outlined CI/CD pipeline is far less sophisticated than it could be. Projects in these kind of companies will never be long term enough to build up big test suites and investing resources into setting up big test cases are always in discrepancy with the project scope.

But there is always room for improvements, so here are some additional recommendations, how the pipeline could be enhanced:

- **Enhanced Pull Requests**
  The current state of pull requests reviews could be enhanced by providing the existing `klint/detekt` **reports** together with some other meta information (e.g. changed permission set, increased apk size) in order to

28. https://github.com/bitrise-io/bitrise-steplib

make proper use of the existing information and utilize a more informed decision about the pull request.
Further it would make sense to **analyze commits after opening pull requests** (which Bitrise currently does not support) in order to see the improvements in between these commits.

- **Pre-commit git hooks**
  Another quite useful tool are the `git hooks` which can be run on pre-commit (or pre-push) to trigger e.g. a lint analyses in order to already prevent faulty code in the repository. These hooks could block a push if the committed code does not reach a certain quality goal. And since `lint/detekt` are usually rather fast tools (which don't even require a full build to run), it would have not a big impact on the development process.

5 Summary

The summary goes here.

Appendix A
AllAboutApps Bitrise Configuration (new)

https://raw.githubusercontent.com/Rasakul/seminar_mobile_ci/master/bitrise/bitrise.yml

References

[1] K. Beck, "Extreme programming: A humanistic discipline of software development," in *International Conference on Fundamental Approaches to Software Engineering*. Springer, 1998, pp. 1–6.
[2] K. Beck and E. Gamma, *Extreme programming explained: embrace change*. addison-wesley professional, 2000.
[3] M. Fowler and M. Foemmel, "Continuous integration," *Thought-Works) http://www. thoughtworks. com/Continuous Integration. pdf*, vol. 122, p. 14, 2006.
[4] (2019, Aug.) Fundamentals of testing. [Online]. Available: https://developer.android.com/training/testing/fundamentals