1, 2, 3 – Build!

Continuous Integration for Mobile Applications

Continuous Integration is a well-established process for supporting the development, test and release of applications. However, when developing mobile applications, many companies still rely on manual processes that are expensive and error-prone, because a few years ago, the tools were not ready or required specialists for setting them up.

This article presents tools, frameworks and services that have been made available recently to solve this issue. By making them easy to use, developers have no more excuse to not do Continuous Integration, even while facing some unique challenges of mobile app development such as testing on real devices.

Continuous Integration

Continuous Integration (CI) is the practice of merging all developer copies into a shared repository, possibly several times a day. The main idea is to prevent integration problems that originate from multiple developers working on separated copies of the same source-code for some time. Especially the merging can be simplified significantly if it happens on a small scale instead of merging largely diverged branches of the same source base. Typically a Continuous Integration server is in the loop and automatically checks the source-code that has been pushed into the repository by building it, exercising the tests and sending feedback to the developers within a short period of time. This short feedback-loop is critical, if something broke down, because the developer still knows the places that changed most recently, which limits the area, where bugs might have been introduced.

Using Continuous Integration has a wide range of benefits, namely (1) the early detection of integration bugs, (2) constant availability of a "current" build for testing and release, (3) reduced dependencies on individuals and (4) small losses if reverting to an earlier state is required.

Continuous Integration does not come for free. Especially setting up the infrastructure and getting the Continuous Integration server up and running can be tricky and takes some time. Such tools often cost money, but in my experience the costs for not using Continuous Integration always exceeded those costs by magnitudes.

That said, Continuous Integration is being used nowadays by almost all professional software developers, although the details on how to work with branches and how to merge individual changes to ensure that the master branch of the repository always stays green are still begin disputed.

Continuous Delivery

Taking the idea of Continuous Integration one step further, leads to the practice of Continuous Delivery, where not only the building and testing of the application is automated, but also the actual rollout of an application. This does not mean, that every change in the source-code repository necessarily leads to a new version of your app in the App store. But all the required steps like signing and uploading a binary package or updating screenshots are being performed automatically.

Tools like Visual Studio Team Services (VSTS) allow to define release definitions with several stages, e.g. automatic rollout to beta-users, staged rollout to 20% of all users and full rollout to all users. Each stage can have certain quality gates or require the approval from specific authorities.

As Continuous Delivery is based on Continuous Integration, it has the same advantages but adds the following benefits, namely (1) avoiding last-minute chaos before releasing a new version, (2) reproducible releases with version-controlled definitions and automated workflows, therefore (3) independence from individuals, (4) reduced costs for releasing and (5) dropped inhibition threshold for performing a release, as an actual release is not much more effort than a few clicks on a website.

So is possible, we should always strive for Continuous Delivery. But still, many companies do not have a Continuous Delivery process and heavily rely on manual steps being taken, whenever a new version of their software is released.

Continuous Delivery in practice

Most developers know the benefits of a Continuous Delivery process, but use some of the following excuses to justify that many steps of the release process are still done manually:

"This is just a small project. I can quickly test and upload it manually"

True, many projects start small – but eventually they grow bigger if they are successful. And even for really tiny projects with a single developer, the repetitive steps involved

in the release process are just boring and some steps might have been forgotten along the way.

"We have no time or budget to set up such services"

The time for setting up a cloud-service for building and testing a mobile application has been reduced dramatically by providers like Greenhouse that offer a streamlined process for setting up the server within minutes. And especially if setting those things up at the beginning of the project, a lot of time can be saved later on.

Cloud-services do cost some money, but most cloud-services offer free plans within certain limits. Only when exceeding those limits, additional packages can be purchased with prices that are still ridiculously low compared to the hourly rate of a developer.

"We don't have a server to run a CI on"

That is not a problem at all, because there are many commercial providers that provide the entire infrastructure for building mobile applications as a service. You don't have to do it on your own and I would not recommend doing so.

"Those services sound great, but we have strict NDAs and are not allowed to use cloud-services"

Strict NDAs do pose a problem, but very often exceptions can be negotiated with clients. And regarding the security concerns – remember that keeping your data safe and private is an essential core competence for any cloud-service provider. A break into their system could have devastating consequences for the company. If all that does not work out, you can still install on-premise versions of various CIs like Travis or VSTS.

"We only release twice a year, so the effort of setting up the Continuous Delivery process is not worth it"

Even if you release only twice a year, you still have the effort of setting up the process just once. And when you start automating this process, you might end up with shorter release cycles which gives you more flexibility and eventually happier customers because you can fix issues faster.

"I don't know how to set up Continuous Delivery for mobile application because there are some steps that cannot be automated."

Really? Just read on.

1 - Building your app automatically

Building mobile applications for the three mobile platforms Android, iOS and Windows unfortunately requires that you run more than one (virtual) machine, since iOS apps can only be built on Apple hardware and Windows UWP apps require Windows (which at least can be virtualized). If you can restrict yourself on the two major platforms Android and iOS, one Apple machine should suffice. Otherwise, a solution that uses multiple agents on different machines can be used. For example, Microsoft offers a platform-independent agent for VSTS.

Setting up an automated build with Greenhouse-CI is a matter of minutes. A straight-forward wizard guides the user through each step, automatically inspects the repository and executes all found tests, including Android Instrumentation Tests that require an emulator (see figure 1).

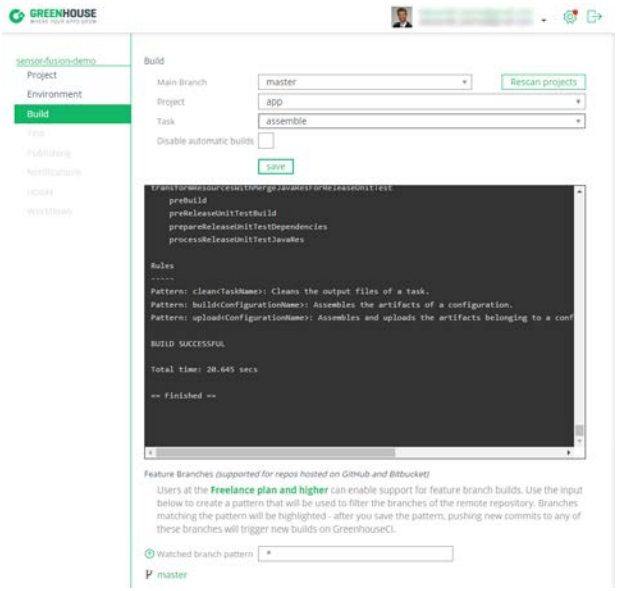

Figure 1: Greenhouse-CI wizard for setting up an automated build

Note that you should automatically set the version number of your application during the build to enable traceability. This can be achieved with a few lines inside a gradle file or with a separate build-step that runs a script.

Signing your app automatically

Signing an application is the process of applying a cryptographic signature to a binary package to enable users to verify that the source of an application is equal to its claimed source.

Signing Android applications is a simple process that can be included into the gradle script which assembles the app. All that is required is a keystore that can be generated with a single command from the shell containing a private key, a public key and some metadata. As there is no verification of the physical entity that created a keystore or the metadata specified within, everything that Google can guarantee, is that an updated version of an application has the same vendor as the originally installed application. Be aware of the fact that everybody could claim to be a member of a major company and –until reported to Google – could distribute malicious software over the Google Play store using that name.

Apple on the other side has very strict rules that have to be followed, before an application is granted permission to the App store. Basically Apple, and only Apple, can allow an application to run on an iOS device. Signing the application with a developer certificate (which gets issued by Apple) is only the first step of the entire procedure. After submitting an application, it is being reviewed and re-signed with Apple's security certificate. Only then it

can run on any iOS device. This leads to the necessity of special profiles – called provisioning profiles – that allow developers to run applications on their own devices. At least, Apple and Xamarin provide step-by-step tutorials on how to create the required certificates and provisioning profiles.

Another great tool for signing iOS applications is called *match*. It is accompanied by a guide of best practices on performing code signing when working as a team of developers.

2 - Testing your app automatically

That you write automated tests for your application should be a natural thing. Writing unit-tests for mobile applications is to no extend any different from writing unit-tests for desktop- or web-applications. The first difference occurs when you start creating more extensive integration-tests that instrument the lifecycle of your application and eventually call the native API. There are basically two approaches to handle this scenario: hide all such calls behind a mockable interface that is initialized accordingly in the tests to yield the desired results or use tools such as Robolectric that are able to handle native calls and process them without the need of an emulator.

The most challenging tests however are full-blown user-interface tests that mirror complete user-scenarios. The tools for writing such tests have seen tremendous improvements in the last few years. Tools like Espresso, Robotium, Xamarin.UITest or XCTest allow to write tests in that resemble the following pattern: *On a given view X, find the element that matches criteria Y and perform the action Z*. More concrete, this could be *onView(withText("OK")).perform(click())*. Note that the actual syntax might deviate slightly for each of the previously mentioned frameworks, but the general idea is the same for all of them.

Note that you can create those UI-tests even easier by using UI-test recorders that are available for Android, iOS and Xamarin.

Once a few tests have been created, a step can be added on the CI-server to automatically execute them, whenever a change is committed. Most CI-systems have predefined steps for unit-testing, firing up an emulator and executing the tests on a virtual or a remote devices. Bitrise, for example offers a beautiful, visual website, where build steps such as *Gradle Unit Test*, *Xcode Test for iOS*, *Xamarin Test Cloud*, *TestFairy* or *Calabash UI tests* can be added and configured (see figure 2).

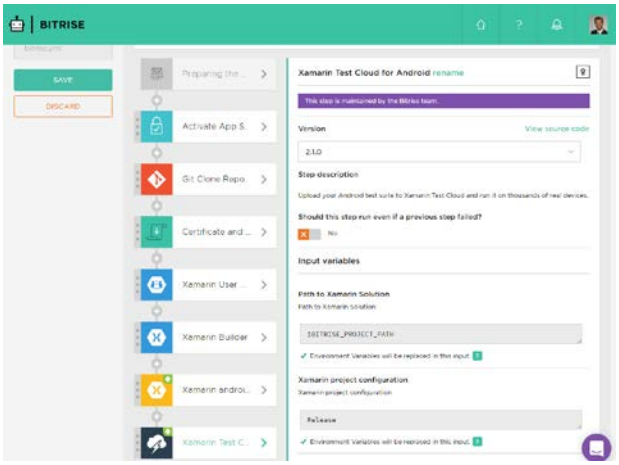

Figure 2: Bitrise website to configure the individual steps of the build

Testing on real devices

No application should ever be released to a larger group of users without being tested at least once by a real human being. But that does not mean that all the tests have to be performed manually, especially not if you consider the wide range of target devices that your application might run on if you release it publicly into an app store. So the testers should primarily focus on the changed bits and verify that the overall application runs smoothly. For all other parts, automated UI-tests should dig deep into the application and verify that all existing workflows produce the expected results on all devices.

If you just want to verify that all visual elements are visible on the screen and the interactions behave as expected when using different locales with various resolutions and platform versions, you can setup your CI to start a couple of different emulators with the specific configurations and run the tests on all of them. This leaves the question open, why one should test on real devices at all? The answer is that some combinations of vendor, model and operating system can show unexpected behavior such as different image rotations when accessing the camera. You can only be sure that your app runs fine, if you have tested it on a wide range of devices.

For testing an application on real devices, a couple of cloud-services are available that are usually payed per usage. Examples of such services are Bitbar Testdroid, Google Firebase, Amazon Web Service Device Farm, Xamarin Test Cloud or TestObject. These services run your application, execute your test-code on the selected number of real devices and usually provide extensive information on the results, such as detailed diagnostics, screenshots or entire videos of the test-run (see figure 3).

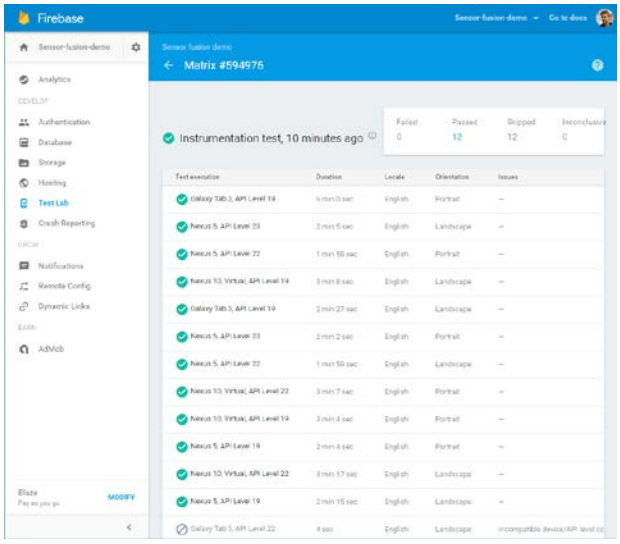

Figure 3: Summary page of a test run on various physical devices with Google Firebase TestLab

One might feel tempted to setup his own farm of physical devices when running an on-premise Continuous Integration server. I would highly discourage this, as setting up such a farm is non-trivial, way more expensive than using cloud-services and one will have to deal with operational issues such as overheated batteries or system-updates that distract you from the actual testing of your application. Note that some cloud services offer a better integration into other tools but for simply automating UI-tests, they are pretty much the same and switching between providers is very easy.

3 – Deploy your app automatically

Once you are happy with you app and gained some confidence from your automated and manual tests, it is time to release your application into an actual app store.

For publishing apps into the Google Play store, Google offers an API that allows you to upload builds (apk-files) to the Google Developer Console, publish it to beta-users or roll-out a new version of your application to a specified percentage of your user-base.

Having this API, one is able to specify release environments (e.g. alpha, beta, production) inside of your CI and push binaries through those environments with a few clicks. Visual Studio Team Services for example offers release definitions that request the consent of dedicated authorities before advancing from one environment to the next one (see figure 4).

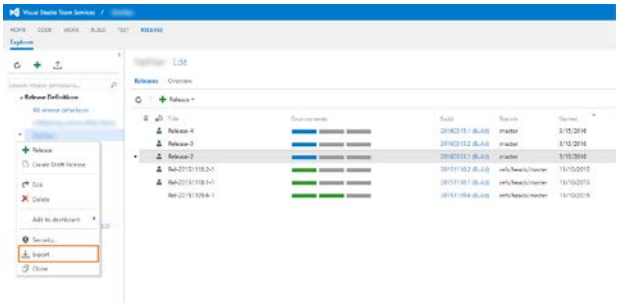

Figure 4: Visual Studio Team Service with multiple release definitions for different environments.

For publishing iOS applications, the process is a bit more elaborate, as Apple demands updated screenshots for each version amongst other information. But Apple provides an API too that allows to upload builds (ipa/pkg-files) to iTunes Connect. A great tool for automatically preparing and uploading the application is called *fastlane*. It is a set of command-line tools that automate the whole process into a few commands that can run automatically on the CI-server.

With these tools, one could even automate the entire process in a way, that each commit onto the master-branch triggers a full release of the application into production if all previous steps completed successfully such as automated tests. This makes sense, if development follows the *GitFlow* branching model, initially proposed by Vincent Driessen and nowadays adopted by many Git clients, where the primary development is done on the develop-branch and each commit onto the master-branch is equivalent to a release.

Monitoring your app

Having an app finally released to the public is usually not where development stops. New features need to be implemented and reported bugs have to be fixed as soon as possible. But how to know that your application has bugs, apart from the fact that *there is no bug-free software*? The answer is by monitoring your application.

A well-known tool for monitoring a mobile application is HockeyApp that was acquired by Microsoft in 2014. HockeApp provides an SDK that can be integrated into any mobile application to automatically send usage statistics and crash-reports. It also makes it easy to distribute beta versions to selected testers and allows them to send feedback to the developers.

Crash reports with a full stack-trace can be vital when tracing bugs in your application, as they contain more insights into the application than any user could formulate verbally (see figure 5).

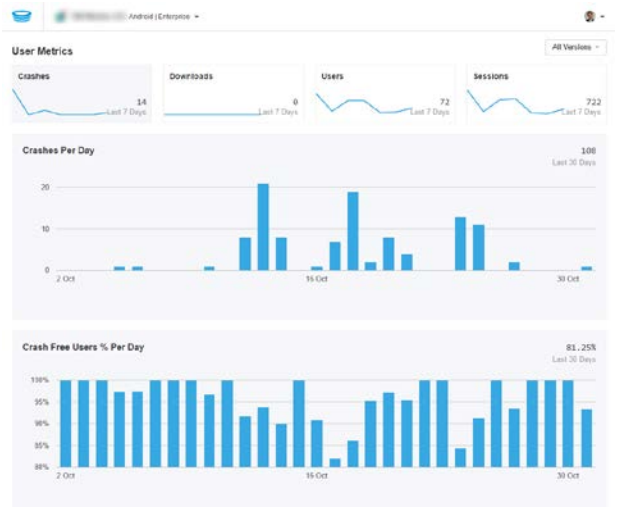

Figure 5: Crash statistics collected by HockeyApp

Conclusion

Continuous Integration and Continuous Delivery are powerful processes that should be used in all software development projects. There are some unique challenges when applying the process to mobile development. But recent advances in the tooling and with cloud-services, offering CI platforms as a service, have made it really easy for all developers to do Continuous Delivery. Even for difficult operations such as signing or testing an application on real devices, there are production-ready solutions that are affordable and work well.

Bibliography

Duvall, Paul et. al, *Continuous Integration: Improving Software Quality and Reducing Risk*, Addison-Wesley, 2007

Humble, Jez et. al, *Continuous Delivery: Reliable Software Releases through Build, Test and Deployment Automation*, Addison-Wesley, 2010

Driessen, Vincent, A successful Git branching model, 2010:

http://nvie.com/posts/a-successful-git-branching-model/

Cloud-services that facilitate **Continuous Integration and Continuous Delivery** for mobile app:

https://www.bitrise.io/

https://greenhouseci.com/

https://www.visualstudio.com/team-services/

https://travis-ci.org/

An extensive guide to **signing iOS applications**:

https://codesigning.guide/

Samples for **automated Android testing**:

https://github.com/googlesamples/android-testing

Build-automation toolset for iOS and Android:

https://fastlane.tools/

Author

Alexander Pacha

Alexander Pacha is a Software Engineer at Zühlke Engineering and responsible for the development of mobile and C# enterprise applications. He is also a trainer for Clean Code.

alexander.pacha@zuehlke.com, @PachaAlexander