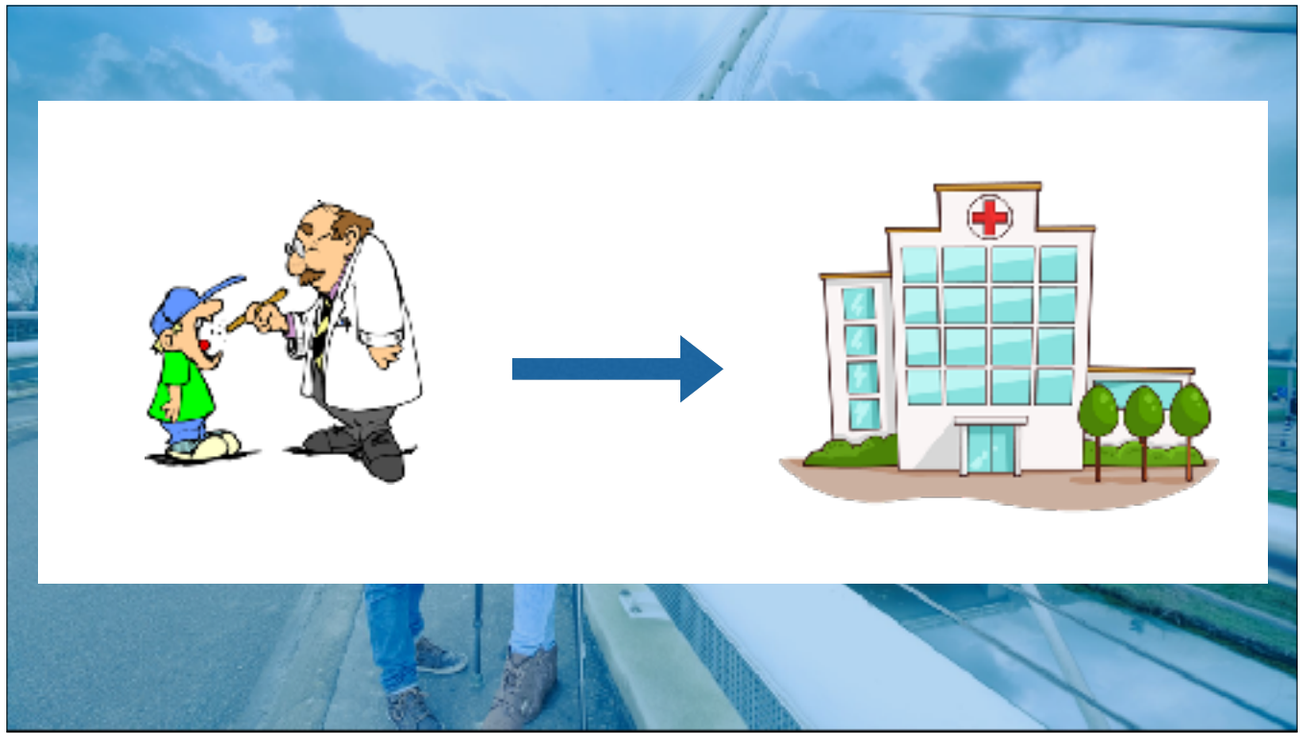
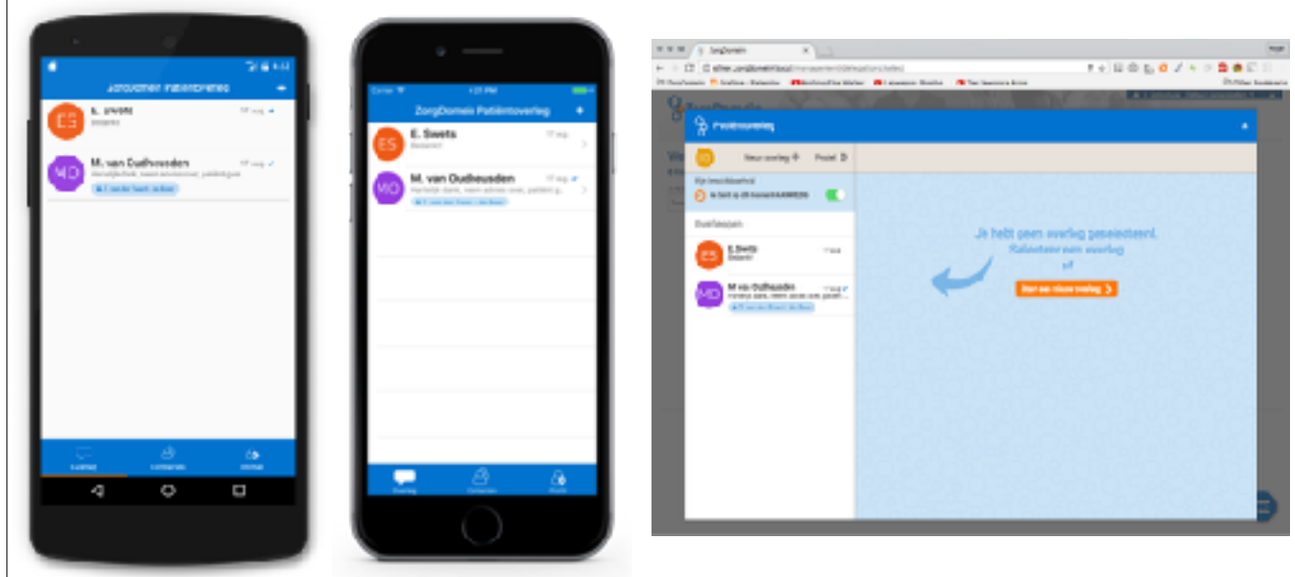

Continuous Deployment in the mobile domain
Arjan Pragt
ZorgDomein
verbetert zorg



ZorgDomain is as the name suspects active in the Healthcare domain. ZorgDomain is known for the referral application. This is a webapplication that handles the process of referring a patient from a general practitioner to a hospital.

But we are not going to talk about that. About a year ago we started a new initiative that should ease communication between different types of healthcare providers. For example between a general practitioner and a specialist.

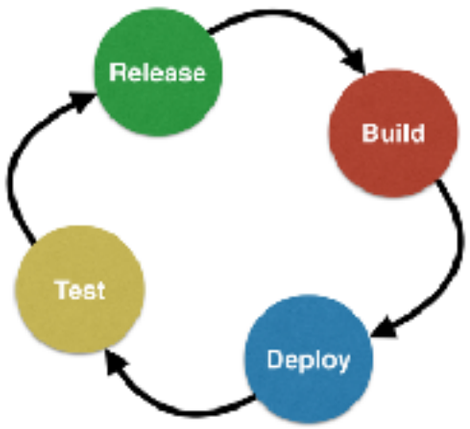


This product is called Patientoverleg. It consists out of mobile apps and a web version integrated into the referral application.

Patientoverleg is the first time we did mobile development and in this presentation I will share our experiences with the continuous deployment for the mobile apps.

We wanted to develop Patientoverleg really with out customers. So we formed a beta group and to involve them as much as possible we wanted to have a version live as soon as possible and frequently push small changes to them to get a short feedback loop.

Continuous Deployment



For our referral application we were already doing continous deployment. So we had experience how it works for web applications.

How we do this is we build a version, deploy it on a test environment run a set of automatic tests and when everything is ok we deploy it into production.

With this setup you will get fast feedback about new functionality, and since that is exactly what we wanted we set it up for Patientoverleg as well.

Mobile specific

- Testing
- Infrastructure
- Release

Like I said we had experience with web applications, but we had to rethink parts of the process.

So we had to think about what test tools there are available, where to deploy our apps and how to release them to our customers.

Testing

- Strangely enough automatic testing still not so common for mobile apps
- A lot of testing frameworks are available
- Unfortunately most of them are poor…

So first we looked at testing. Strangely enough a lot of apps are still build without any form of automatic testing. After development manual e2e testing is done.

There are however a lot of test frameworks, but in our experience, especially for IOS, a lot of them are slow, buggy or their support for new OS versions is late.

Then there is the concern of deploying your apps. A web application can be deployed almost everywhere but unfortunately for IOS development you will need Apple hardware.

You can either use a cloud provider or buy your own hardware. None these options are perfect. You don’t want to manage physical hardware these days, but looking at the pricing models of cloud provider they can become quite costly since you need a lot of running time from them for continuous deployment.

Deployments

- Have to play by the rules of Apple and Google
- Need to manage meta-data of apps
- Review process per app build
- Non-desirable experience for users

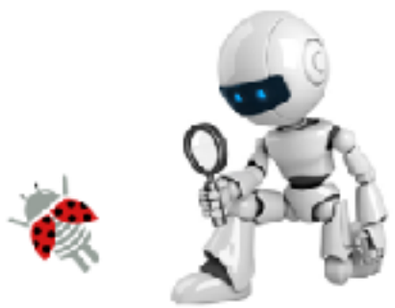
And there is the concern of deployments. A web version can be put in production instantly and in own control. Apps however need to be reviewed by either apple or google, and need to have proper metadata for every release.

Unlike web, apps need to be downloaded and installed. And also it raises the question how desirable real Continuous Deployment is for our users.

So, what did we choose

Testing

- Android: JUnit and Espresso
- IOS: XCTest and XCUITest



Thanks to it’s Java background the Android test frameworks are pretty mature and stable.

For IOS however that is a different story, we did not really find a mature framework and decided to use the apple provided frameworks. These have as main benefits that is is supported by apple and that the tests live in the same codebase as the app and are written in the same language. However especially with XCUITest you will experience the bugs and crashes.

Deployment

Tried OS X Server for iOS CI

- OS X Server deeply integrates with XCode
- Setup is rather effortless
- … but doesn't work all to well with XCUITest
- … and we want to automate more!

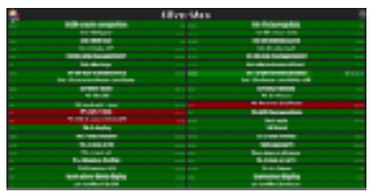
Just like with the test frameworks we decided to use the apple provided solutions. We found cloud providers to costly and slow and decided to buy a mac mini and install the tooling locally. So we installed osx server.

However it did not work nicely with XCUITests for our E2E tests and the integration with our existing build environment was difficult.

Deployment

Fastlane

- Tools ‘lanes’ for the most common tasks
- Does most of the configuration for you



We then decided to use something else. We found Fastlane. It actually contains tools for the most typical tasks.
It can run tests, generate metadata (like screenshtos) and release your app to the appstore.

And it integrated nicely into our build environment.

So with all this in place our progress we were confident and our progress was good.

However since we learned more and more about app development we started to refactor thing and implementing more and more features. And now everything we touched caused crashes.

So we fell back on manual testing. Before every release the app was tested for all the functionality.

Improve testing

- Refactor code for better unit testing
- Introduce visual regression tests

This was not what we wanted to started to improve our automated tests. We looked at design patterns to bring more parts of the code under unit tests.

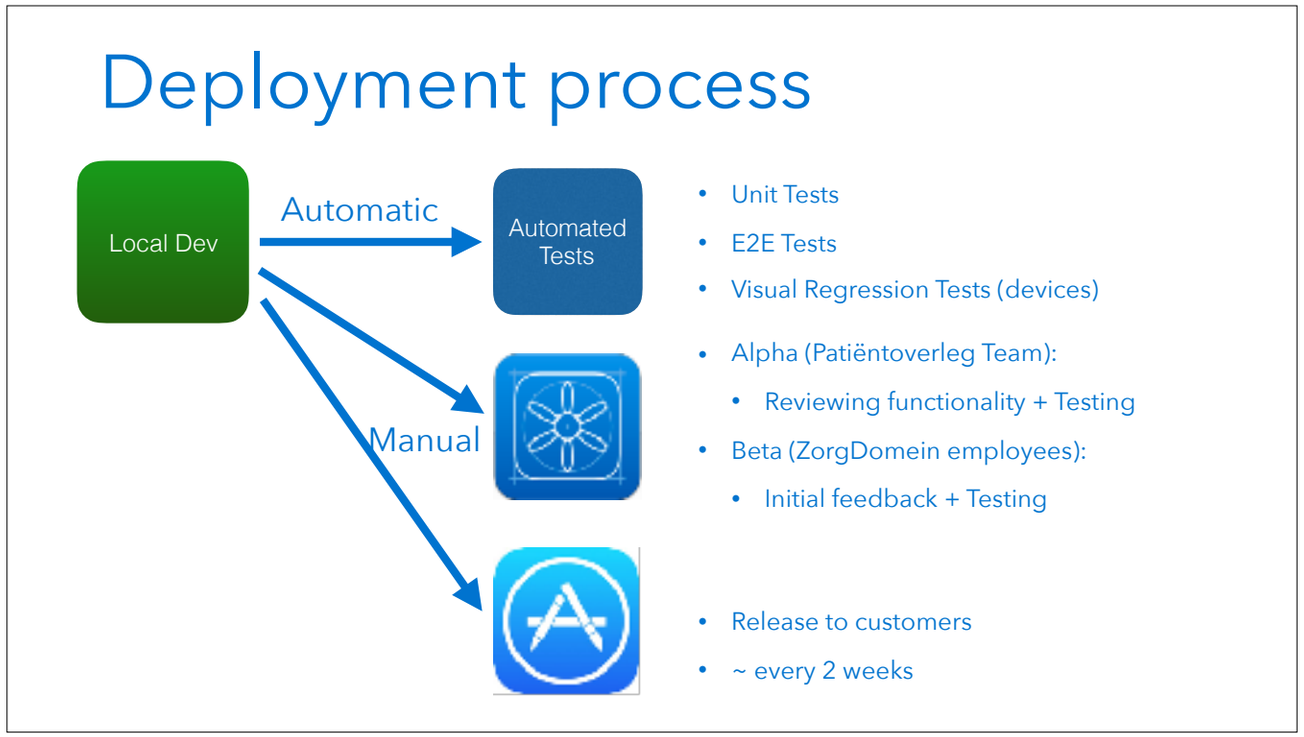
Besides that we introduced visual regression tests where we compare screenshots from older builds with new ones to capture even more glitches.

Manual testing

- Hard if not impossible to automate specific cases
- Important to be extremely critical (glitches, etc)
- Testflight is a godsent

However we are decided to keep the manual testing in place. Especially for IOS, that has a lot more animations and visual effects than android it is more likely to have glitches and unforeseen scenarios.

With the help of Testflight and the Play store we can release alpha and beta versions to a controlled group for testing.



This is how our current release deployment works. Whenever developers commit changes we deploy a new version on our mac mini and run a set of unit and e2e tests. After those tests we create screenshots of different parts of the application and check them against older builds on multiple devices.

Whenever a build passes the automated build we will manual release a alpha build to our team. We review new functionality and test it on our physical devices.

When we think the build is stable enough we release a beta version to a bigger group of employees within our company. They will give us again feedback and test it again on their devices.

As soon as we are satisfied and have reached trust in quality we will release the new version to the app stores and our customers.

Learning points & takeaways

- Continuous deployment?
- App development is a world apart
- Android more mature than ios
- Expect the occasional crashes and unexpected behavior
- Use of custom components causes troubles

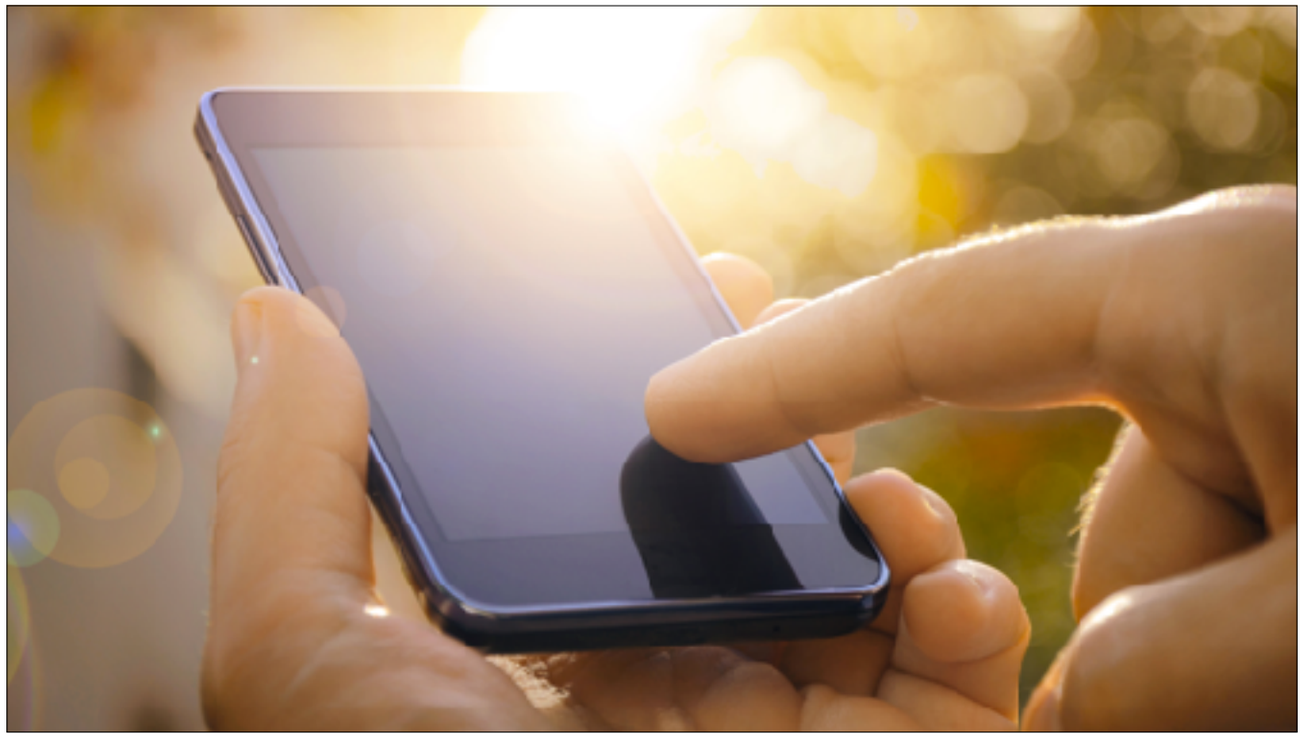
So what did we learn.

We are actually not doing continuous deployment at all. We are releasing versions in a quite traditional way. The only difference is that we are trying to continuous improve our product in small steps while we try to find a balance with releasing often and don’t annoy our customers with new versions too often.

Besides that we found that android as a development platform feels more mature. All the tooling for ios somethings crashes or does unexpected behavior.

Also we started our project with using a lot custom interface components. Because of our inexperience with the platform this caused a lot of effort to get stable.

App development is a complete new world with new tools patterns that you will have to get used to.



So with all this in place our progress we were confident and our progress was good.

Questions, remarks?