Continous Integration in der App-Entwicklung

Am Standort Berlin entwickeln wir für unseren B2B-Partner Douglas unter anderem die Kunden-App [2], [3]. Einhergehend mit dem Ausbau der App-Entwicklungsaktivitäten haben wir in den letzten Monaten den CI-Ansatz überarbeitet. Ein zusätzliches Team sollte am selben Produkt – der KundenApp – mitarbeiten und die App sollte öfter veröffentlicht werden. Mit Methoden und Tools aus dem Bereich Continous Integration wollten wir dafür sorgen weiter zuverlässig und mit hoher Qualität zu liefern. Und das natürlich automatisiert. Neben der Technik geht es auch um die Teams. Wie sind sie vorgegangen und welche Hürden haben sie genommen.

Komplexität und Feedback

Je mehr Personen gleichzeitig an einem Produkt entwickeln, desto größer wird die Wahrscheinlichkeit, dass unbeabsichtigte Seiteneffekte auftreten. Gleichzeitig steigt der Umfang der App, da wir konstant neue Funktionen hinzufügen und bestehende Funktion ändern. Um die Komplexität weiter zu beherrschen, ist schnelles Feedback zu Änderungen ein entscheidender Faktor, um Probleme schnell zu korrigieren. Wie wäre es, automatisiert ein Feedback nach jedem Commit zu bekommen und darauf nur kurze Zeit warten zu müssen? Genau hier setzten wir an.

Schnelles Feedback erhalten wir durch den Einsatz von **Feature-Toggles** und durch die Ausführung von **automatischen Tests** im CI-Prozess.

Feature Toogles

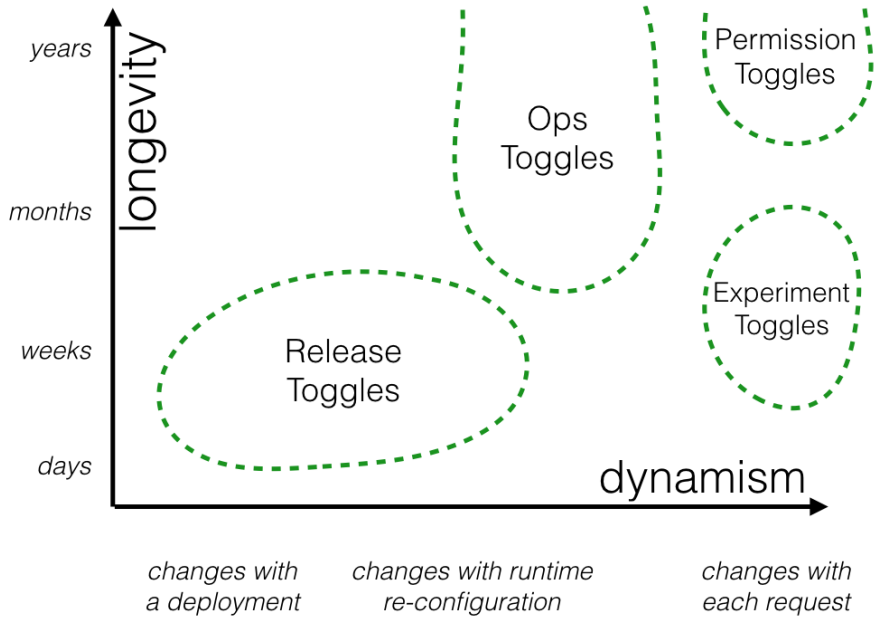
Feature Toogles ermöglichen es uns Codeänderungen aller Entwickler kontinuierlich in einen gemeinsamen Integration-Branch zusammenzuführen. Und das auch, wenn Features noch nicht fertig bzw. für den Kunden nicht sichtbar sein sollen. In der Vergangenheit haben wir solche Features für mehrere Tage, manchmal Wochen, in separaten Branches entwickelt und erst am Ende der Entwicklung in den Integration-Branch gemergt. Das Feedback kam entsprechend

spät. Traten Probleme auf, war es durch die Vielzahl der Änderungen mitunter schwer festzustellen, welche konkrete Änderung zum Bruch geführt hat. Die Integration hatte das Potential unsere Zeitplanung empfindlich zu stören. Diese Bing-Bang-Szenarien sollen durch Toggles und kontinuierliche Integration abgefedert werden.

Toggles und Diskussionen

Der Einsatz der Feature-Toggles wurde im Team intensiv diskutiert, denn die Einführung erhöht erstmal die Komplexität – und liebgewonnen Pfade, wie die isolierte Arbeit im Feature-Branch, standen auf einmal auf dem Prüfstand. Es gab diverse Pros und Cons. Auch musste ein gemeinsames Verständnis beim Thema Toggle-Mechanik erarbeitet werden. In Bezug auf Dynamik und Langlebigkeit der Toggles gab es unterschiedliche Auffassungen, da viele schon mal irgendetwas mit Toggels gemacht hatten.

Wir haben uns am Ende auf die Nutzung von Feature-Toggels zur Entwicklungszeit – auch Release-Toggles genannt – geeinigt. Sie werden für den Zeitraum weniger Tage/Wochen genutzt. Ist das Feature fertig entwickelt, wird der Toggle aus dem Code komplett entfernt. Der Artikel auf martinfowler.com [1] sei dem interessierten Leser an der Stelle empfohlen.



https://martinfowler.com/articles/feature-toggles.html
2 bis 3 Feature-Toggle sind im Durchschnitt parallel im Einsatz. In unserem

Jenkins haben wir durch einen manuellen Schritt in der Build-Pipeline die Möglichkeit geschaffen, einen einzelnen Toggle zu aktivieren und somit App-Artefakte für Features (apk, ipa) für das Testing zu bauen. Ist das Feature komplett entwickelt, wird der Toggle aus dem Code entfernt. Mit dem nächsten App-Release ist das Feature dann für den Kunden sichtbar.

Was ist ein Feature Toggle?

Ein Feature Toggle ist eine Progammiertechnik, die es erlaubt ein Feature oder eine Funktion vor Kunde ein- bzw. auszuschalten. Also die Sichtbarkeit zu ändern. Entwicklungsteams aktivieren Features beispielsweise um noch nicht fertige Funktionen integrieren und testen können. Ist ein Feature fertig, kann es ohne großen zusätzlichen Merge-Aufwand veröffentlicht werden, da die Arbeit in separaten Branches entfällt. Feature Toggles können auch dafür genutzt werden, die Sichtbarkeit von Funktionen zur Laufzeit der Anwendung zu ändern. Z.B. im Rahmen von A/B Tests oder wenn die Sichtbarkeit zu einer bestimmten Zeit geändert werden soll.

Toogles im Code

iOS

Unter iOS wird ein Feature in der App über eine Environment-Variable in der **Launch-Konfiguration** aktiviert (z.B.: USE_NATIVE_PRODUCT_LIST = 1). Im Code wird dann an relevanten Stellen über eine Abfrage entschieden, ob bestimmte Codestellen zur Ausführung kommen oder nicht.

```
if toggleRouter.isNativeProductListEnabled() {
  // Feature Code
}
```

Android

Es gibt ein Interface, in dem alle Toggles als Methoden definiert werden. Diese Methoden werden mit Java-Annotations annotiert und geben immer ein Boolean zurück – TRUE für Feature aktiv, FALSE für nicht aktiv.

```
@ReleaseToggle(BuildConfig.FEATURE_PRODUCT_LIST)
fun isProductListEnabled(): Boolean
```

Eine eigens dafür entwickelte Library mit einem **Annotation-Prozessor** wird während der Build-Phase ausgeführt: Dieser schaut in einer Konfigurations-Datei (json) nach, ob das jeweilige Feature getoggelt werden soll. Wenn das Feature eingeschaltet werden soll, muss der String, der sich in der Annotation befindet, hier eingetragen werden.

```
[
    "FEATURE_PRODUCT_LIST"
]
```

Der Prozessor baut dann jeweils die Implementation für das Interface zusammen. In diesem Fall würde die implementierte Methode TRUE zurück liefern. Wäre der String FEATURE_PRODUCT_LIST nicht in der Datei, wäre es FALSE.

So kann man auf jedem lokalen Rechner die Features beliebig ein- und ausschalten. Auf dem Jenkins kann man das genauso machen, hier editieren wir nicht manuell die Datei sondern sagen ihm über das **Blue Ocean Plugin**, welches Feature getoggelt werden soll.

Und die jeweiligen Code-Stellen togglen wir, in dem wir die Interface-Implementation prüfen:

```
if  (ReleaseToggleConfiguration_Impl().isProductListEnabled())
{
     // Mach was mit der Product list
}
```

Ein gemeinsames Traffic Light für Build und Test

Eine weitere zentrale Komponente im CI-Prozess stellt die Testautomatisierung dar. Das Feedback, dass Build und Test erfolgreich nach einem Commit auf Integration durchgelaufen sind, wird durch eine Ampel visualisiert. Diese ist für jedem im Team sichtbar. Ist sie rot, ist das **gesamte Team angehalten** den Grund zu ermitteln und die Ampel wieder auf „grün zu bekommen“. Also Fehler zu korrigieren, Tests oder die Automatisierung anzupassen.

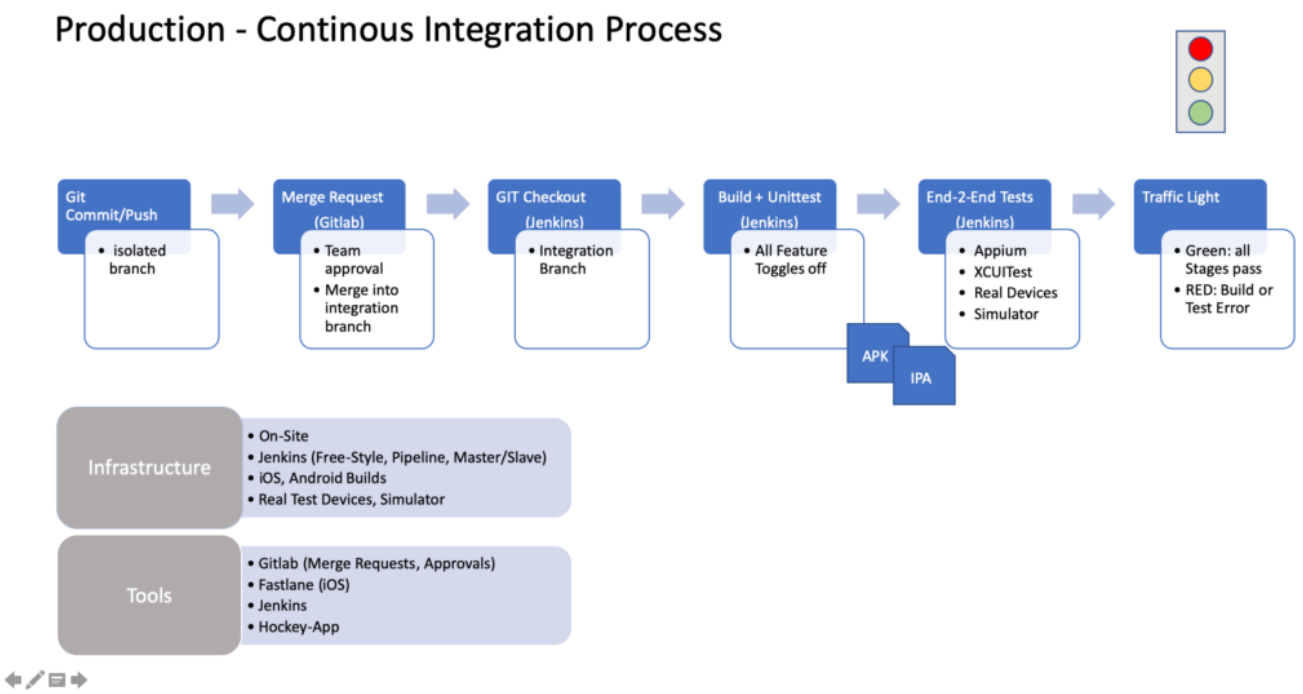
CI-Build-Status für Android und iOS

Die Tests sind eine Kombination aus Unit-Tests und End-2-End-Tests (Akzeptanztests). Die End-2-End-Tests laufen auf echten Geräten bzw. Simulatoren im Zusammenspiel mit dem Backend.

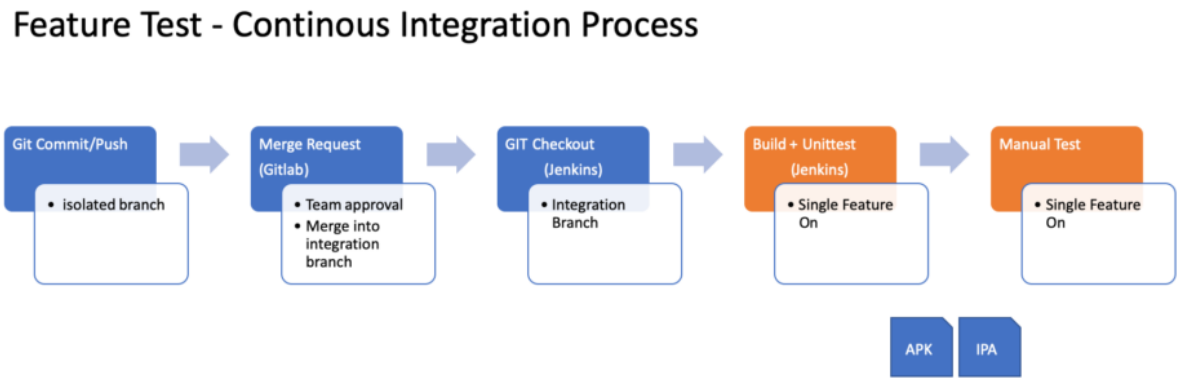
Continous Integration Process

Unser CI Prozess sieht wie folgt aus:

Nach dem Approval der Codeänderung in Gitlab und Integration in den Integrations-Branch baut der Jenkins das App-Artefakt, führt die Unittests aus und startet die End-2-End Tests. Das kombinierte Ergebnis aus Build/Unitests und End-2-End Test wird auf der Ampel dargestellt.



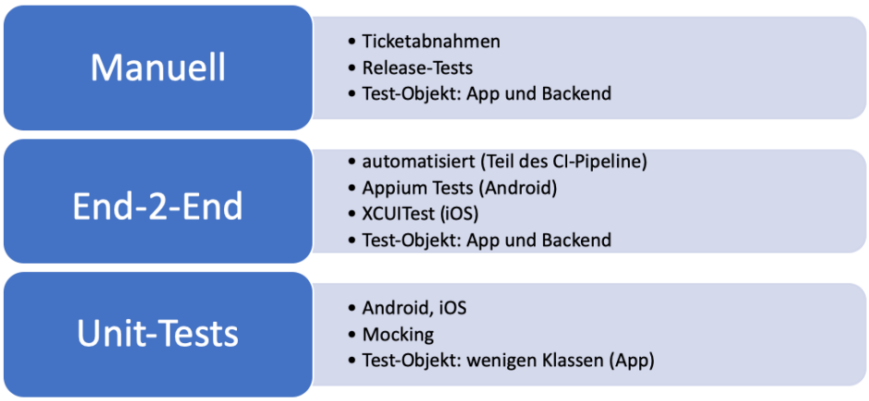
Für den Test eines Features, dass sich noch in der Entwicklung befindet, wird ein Feature-Toggle manuell im Jenkins aktiviert, die App gebaut und die Unittests ausgeführt. Die End-2-End Tests werden zum jetzigen Zeitpunkt noch nicht ausgeführt. Zum einen müssten die Tests für das Feature bereits angepasst und erweitert sein. Das ist noch nicht der Fall. Ein weiterer Grund sind die noch fehlenden Ressourcen in Form von Hardware und Testgeräten. Ein nächster Schritt.



Learnings zu Tools, Integrationslevel und Verantwortung

Auf drei Learnings möchte ich an der Stelle speziell eingehen.

Der Leser soll dazu wissen, dass unsere Entwicklungsteams Cross-funktional aufgebaut sind. Ein Entwicklungs-Team besteht aus iOS-Entwicklern, Android-Entwicklern, Backend-Entwicklern und einem Quality Engineer. Die Backend-Entwickler sitzen an einem anderen Standort und sind wie wir Dienstleister für den B2B-Partner. Folgende Tests führen wir zur Zeit durch:



Testtypen

Beim Tooling ging es vor allem um die Wahl des **Testautomatisierungstools**. Da die QA in der Vergangenheit auf Appium gesetzt hat, wollten wir die Technik auch für unsere CI-Tests im gesamtem Team nutzen (Dev+QA). Bisher wurden die Appium Test ausschließlich von der QA geschrieben und waren nicht in einem gemeinsamen CI-Prozess zusammen mit dem Build integriert. Es stellte sich heraus das die Akzeptanz des Tools für iOS unter den Entwicklern sehr gering war. Stabilität, Funktionsumfang, Integrierbarkeit und Ausführungsgeschwindigkeit überzeugten nicht. Unsere Teams haben sich daher entschieden, für iOS die End-2-End Tests auf Basis von XCUITest neu zu schreiben. Für den Android Bereich setzen wir vorerst weiter auf Appium.

Ein weiteres Learning gab es beim **Integrationslevel**. Unsere End-2-End Tests weisen ein hohes Integrationslevel auf: die App wird im Zusammenspiel mit dem Backend getestet. Fehler im Backend oder eine schlechte Verbindungsqualität können dazu führen, dass Tests fehlschlagen, obwohl die App „korrekt“ funktioniert. Die Ampel zeigt rot, obwohl „mit der App alles in Ordnung ist“. Flaky

Tests bzw. instabile Tests senken die Aussagekraft der Ampel deutlich und führen dazu, dass die Teams einer roten Ampel weniger Aufmerksamkeit schenken. Neben dem End-2-End Test planen wir daher einen zusätzlichen Testtyp einzuführen, der vom Integrationslevel zwischen Unittest und End-2-End Test liegt. Ziel ist es die App ohne Backend zusätzlich zu verifizieren. Dafür sollen Backend-Responses „gemockt“ und die Tests auf der UI-Ebene durchgeführt werden. Die Test sollen eine Ergänzung zu den End-2-End Tests werden.

Beim Thema Verantwortung gehen wir mit dem CI-Ansatz ebenfalls neue Wege. Das Ergebnis aus Build + Test in einem gemeinsamen Ampelstatus zu visualisieren und damit jeden im Entwicklungs-Team zu aktivieren, Probleme im **Test oder Build** zu analysieren, erfordert, dass sich Entwickler mehr mit dem Thema Testing und sich die QA mehr mit der Automatisierung auseinandersetzt. Dieser Veränderungsprozess benötigt Zeit und den Willen aller Beteiligten sich zu verändern. In unserem Produktteams ist diese Veränderung explizit gewünscht und alle im Team sind angehalten für den Prozess **Verantwortung** zu übernehmen und ihn aktiv weiterzuentwickeln.

Ausblick

Im Bereich Testing steht der Ausbau der Testautomatisierung für die End-2-End Tests und die Einführung der zusätzlichen Test-Verifikationsstufe für die Android-App mit Espresso als UI-Testing Tool an.

Um die Qualität zu steigern, möchten wir automatisiert statische Codeanalysen durchführen und Metriken wie beispielsweise die technische Schuld ermitteln.

Verweise

[1] https://martinfowler.com/articles/feature-toggles.html

[2] Douglas App iOS: https://itunes.apple.com/de/app/douglas-parfüm-kosmetik/id394685685?mt=8

[3] Douglas App Android: https://play.google.com/store/apps/details?id=com.douglas.main&hl=de