App-Technik, die begeistert! Über App-Architektur und Testing

Technische Einblicke am Beispiel von Scan & Go – Die mobile Einkaufslösung für die Thalia-App.

Einleitung

Wir, bei der Thalia Bücher GmbH, möchten unseren Kund*innen ein optimales Einkaufserlebnis anbieten. Aus diesem Grund stellen wir verschiedene Produktlösungen bereit, welche wir kontinuierlich verbessern.

Eines dieser Produkte ist die App „Thalia – Bücher entdecken“ für das Smartphone & Tablet (Android, iOS): https://www.thalia.de/vorteile/thalia-app.

Abb. 1: QR-Code zur App-Installation (Quelle: Thalia Bücher GmbH)

Um für unseren Kund*innen den Kauf zu vereinfachen, haben wir bei Thalia das Feature Scan & Go entwickelt. Ziel ist der kontaktlose, schnelle und mobile Einkauf in unseren Buchhandlungen mit der App. In Zeiten von Pandemie oder, um lange Schlangen an den Kassen in der Weihnachtszeit zu vermeiden, ein Mehrwert für unsere Kund*innen.

Allgemeine Informationen sind unter folgenden Webseiten zu finden: https://www.thalia.de/vorteile/scan-go oder https://www.youtube.com/watch?v=jCHEASuHVAc.

Alle Abbildungen stammen vom Autor, sofern keine Quelle angegeben ist.

Abb. 2: Scan & Go-Aufsteller mit QR-Startcode

So funktioniert Scan & Go

Nach der Installation bzw. nach dem Start der App kann in der Buchhandlung der QR-Startcode auf den hierfür bereitgestellten Aufsteller eingescannt werden (siehe Abbildung 2). Anschließend können Artikel anhand des Barcodes – z.B. auf der Buchrückseite – mit der App erfasst werden. Zum Schluss kann der Kauf

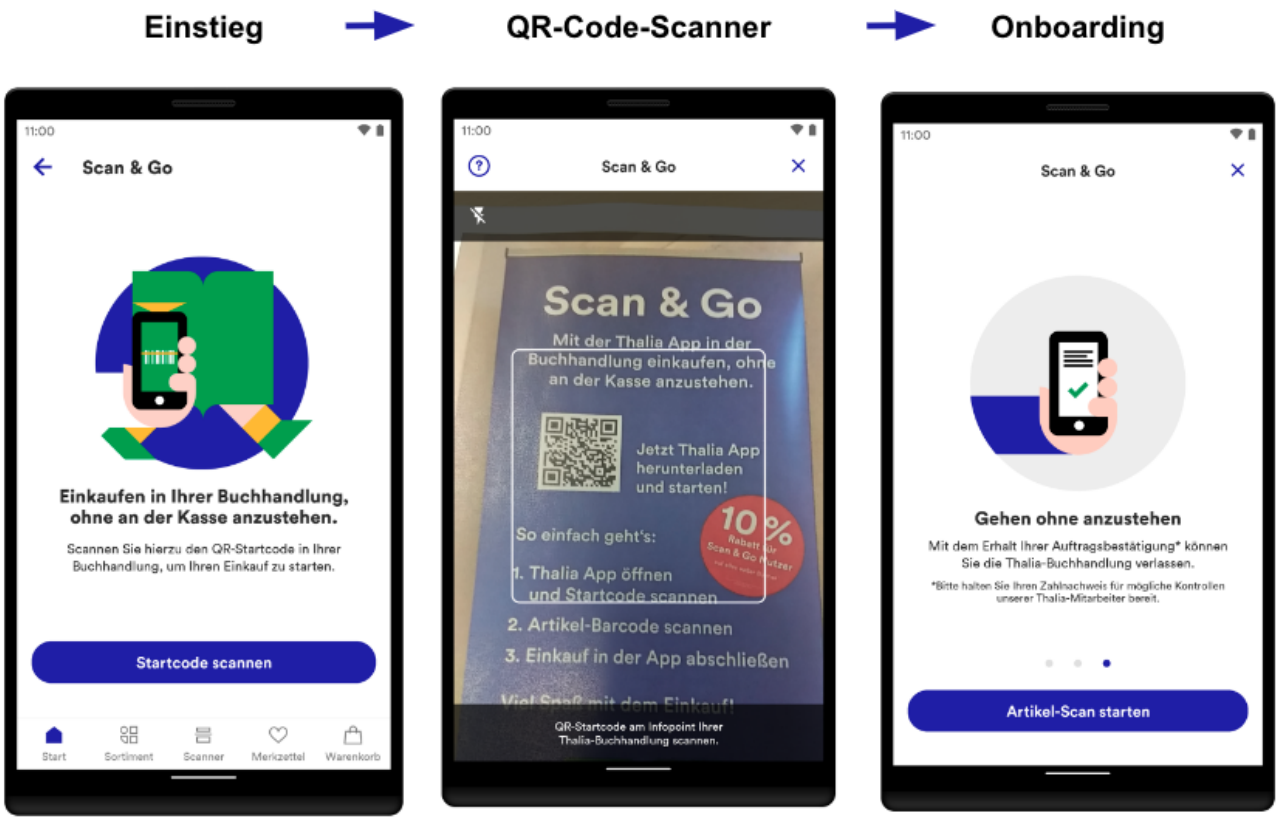
innerhalb der App bequem getätigt und die Buchhandlung, ohne an der Kasse anstehen zu müssen, sorgenfrei verlassen werden.

Ziel

Mit diesem Tech-Blog-Artikel möchten wir aufzeigen, wie wir aus technischer Sicht Scan & Go entwickelt haben und welche Hürden wir zu bewältigen hatten. Des Weiteren wird ein Überblick über unser automatisches Testen von Scan & Go beschrieben. Der Artikel richtet sich an alle App-Entwickler*innen und technisch versierten Leser*innen.

Überblick

Die wesentlichen Bestandteile von Scan & Go werden mit Hilfe der folgenden Screenshots analog der User Journey dargestellt. Aufgrund des Umfangs werden nicht alle Screens vorgestellt (z.B. Hilfe oder Login).



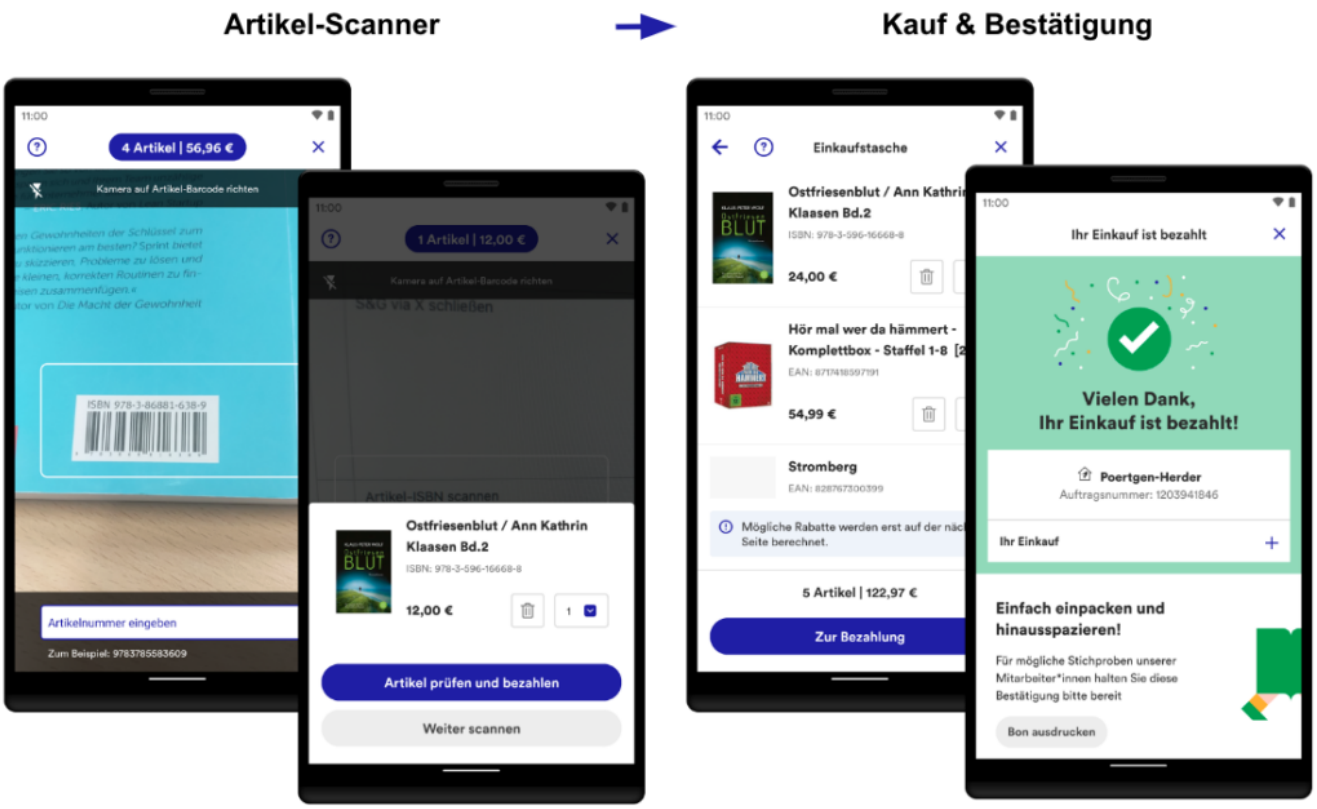


Abb. 3: Übersicht von Scan & Go

Einstieg: Scan & Go kann über diverse Wege angesteuert werden: Über die Startseite der App, den internen Scanner in der App oder über die Kamera des Gerätes.

QR-Code-Scanner: Der Zugang zu Scan & Go wird durch das Einscannen von auf diversen Werbemitteln gedruckten QR-Codes in der Buchhandlung gewährt. Der Zugang ist zwei Stunden gültig.

Onboarding: Nach dem Einscannen des QR-Codes werden mit Hilfe eines optionalen, dreistufigen Onboardings die wichtigsten Funktionen und Informationen erklärt. Sie werden erst wieder angezeigt, nachdem der Zugang abgelaufen ist.

Artikel-Scanner: Das Herzstück: Im nachfolgenden Scanner werden die Artikel anhand des Barcodes erfasst. Im Bestätigungsdialog kann der Artikel begutachtet und auf Wunsch bearbeitet oder entfernt werden. Alle Artikel werden automatisch in der digitalen Einkaufstasche gespeichert.

Kauf & Bestätigung: In der Einkaufstasche werden alle Artikel in einer Liste zusammengefasst. Einzelne Artikel können geändert werden. Abschließend erfolgt nach der Bezahlung eine Bestätigung auf der Dankeseite. Der Einkauf ist damit erfolgreich abgeschlossen.

Implementierung

Für die Implementierung haben wir einen modernen Technologie-Stack gewählt, damit Scan & Go stabil läuft, einfach erweiterbar ist und sich über einen langen Zeitraum bewahren kann. Im Folgenden wird das technische Konzept erläutert und anhand von ausgewählten Beispielen vorgestellt. Scan & Go haben wir für iOS und Android entwickelt. Eingesetzte Technologien befinden sich im Anhang.

Architektur

Als übergreifendes Architekturkonzept haben wir die sog. “Clean Architecture” herangezogen (vgl. https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html, Buchempfehlung: https://www.thalia.de/shop/home/artikeldetails/A1039840971). Ziel des Konzeptes ist, eine skalierbare, einheitliche und wartbare Implementierung anzustreben. Des Weiteren ist einer der wichtigsten Merkmale: Wir können verständlichen und gut testbaren Code entwickeln. Erst dadurch sind wir in der Lage, eine unerlässliche Testautomatisierung umzusetzen (siehe “Abschnitt Sicherstellung der Qualität & Automatisiertes Testen”).

Unsere Architektur ist in drei Schichten gegliedert: **UI**, **Data** und **Domain** (z.B. https://developer.android.com/topic/architecture).

UI: Die UI beinhaltet die Logik zur Präsentation von aufbereiteten Daten. Sie ist abhängig von Data und Domain und sollte keine Business-Logik beinhalten. Die UI ist eng an das Betriebssystem (z.B. Android) gekoppelt bzw. interagiert mit diesem.

Data: Beinhaltet die Business-Logik der App und implementiert die Datenhaltung und -beschaffung z.B. über REST-Services. Data ist lediglich von Domain abhängig.

Domain: Domain ist technisch das Bindeglied zwischen UI und Data. Ziel ist die Vermeidung von Redundanzen bzw. Bereitstellung von wiederverwendbaren Funktionalitäten (Use Cases) und weitaus komplexeren Business-Logiken. Dadurch wird die Interaktion zwischen UI und Data stark vereinfacht. Domain ist komplett unabhängig von UI und Data und kennt keine umgebende Infrastruktur (z.B. das Betriebssystem Android). Aus diesem Grund ist es sehr gut automatisch

testbar.

Für jeden Screen in Scan & Go wurde eine separate UI-Klasse implementiert und das übliche Entwurfsmuster „Model View Viewmodel“ verwendet. Der Unterbau für jede UI-Klasse setzt sich wie folgt zusammen (vereinfacht):

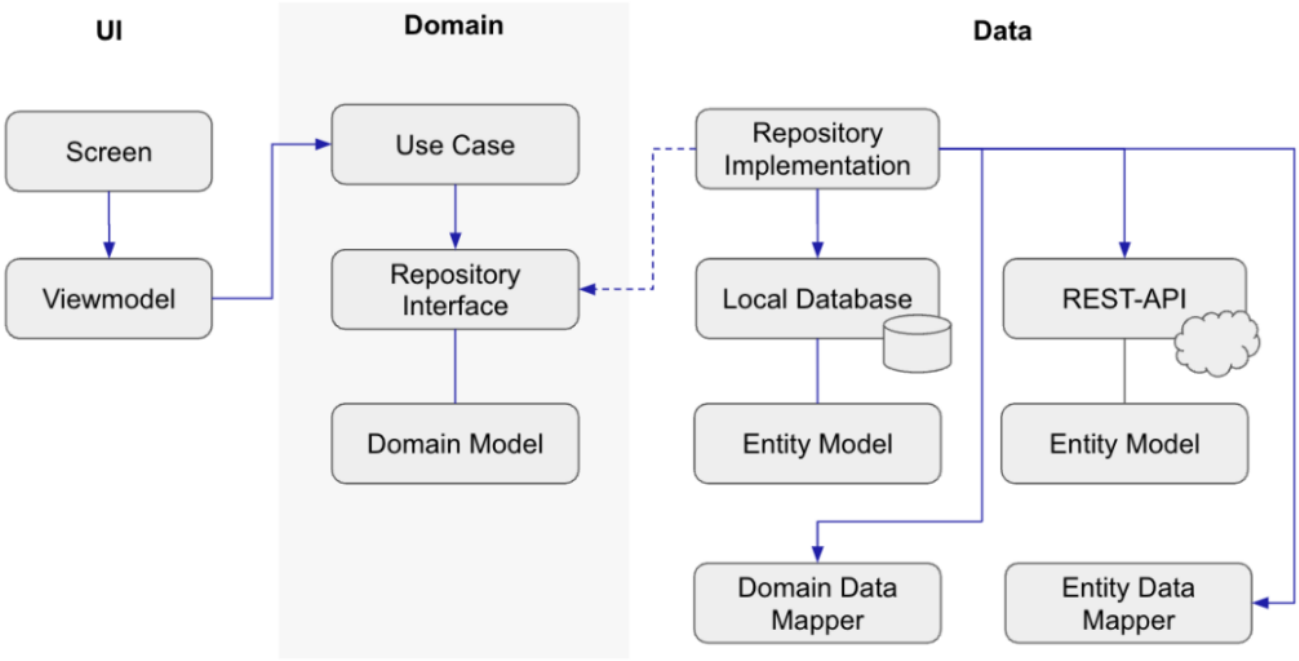


Abb. 4: Grober Aufbau einer UI-Klasse

- Die UI-Klasse *Screen* arbeitet nur mit einem *Viewmodel* (durchgezogener Pfeil). Sie observiert das *Viewmodel* und präsentiert bei jeder Änderung eins zu eins die Daten.
- Ein Viewmodel operiert mit *Use Cases* und verwendet lediglich die Daten aus dem Domain Model.
- Ein *Use Case* greift ausschließlich auf abstrakte *Repositories* (Interfaces) zu und bereitet gemäß der gewünschten Business-Logik die Daten auf.
- Data implementiert die Interfaces aus Domain (gestrichelte Linie), regelt den lokalen und externen Datenzugriff und bildet über *Mapper*-Klassen die entsprechende Datenstruktur ab. Dadurch hat beispielsweise eine Änderung von Attributen der *Entity*-Klasse in der Regel keinen Einfluss auf Domain und UI.

Beispiel: Implementierung des Artikel-Scanners

(Android)

Nachfolgend werden die oben beschriebenen Zusammenhänge exemplarisch für den Artikel-Scanner-Screen mit Android-Code illustriert. Aufgrund der Komplexität und Schutz des Urheberrechts sind nicht alle Details abgebildet.

UI + Domain (Fragment, Viewmodel und Use Case): Für den Artikel-Scanner haben wir eine Klasse *StorePaymentScannerFragment* implementiert. Die Verdrahtung der Klassen erfolgt mittels Dependency Injection auf App-Ebene mit Dagger.

Abb. 5: Zusammenhang zwischen Fragment, Viewmodel und Use Case

- (A) Mit Hilfe der Kamera des Gerätes kann der Barcode des Artikels erfasst werden. Die Erkennung des Barcodes erfolgt beispielsweise in Android mit dem ML Kit und mit CameraX von Google.
- (B) Alternativ kann die EAN bzw. ISBN oberhalb des Barcodes manuell eingetragen werden.
- (1) Nach Erkennung der EAN-Nummer wird diese an *loadArticle* übergeben.
- (2) Der übergebene Code wird an den *HandleScannedEanUseCase* weitergereicht, welcher asynchron ausgeführt wird und die Beschaffung der Artikeldaten sowie die Speicherung kapselt.
- (3) *HandleScannedEanUseCase* übergibt intern den Code an das Interface (siehe nächsten Abschnitt).

Data (Repository, Mapper und API): In der nächsten Abbildung wird anhand einer eingescannten EAN-Nummer der passende Artikel aus dem Backend besorgt.

Abb. 6: Zusammenhang zwischen Use Case, Repository, Mapper und API

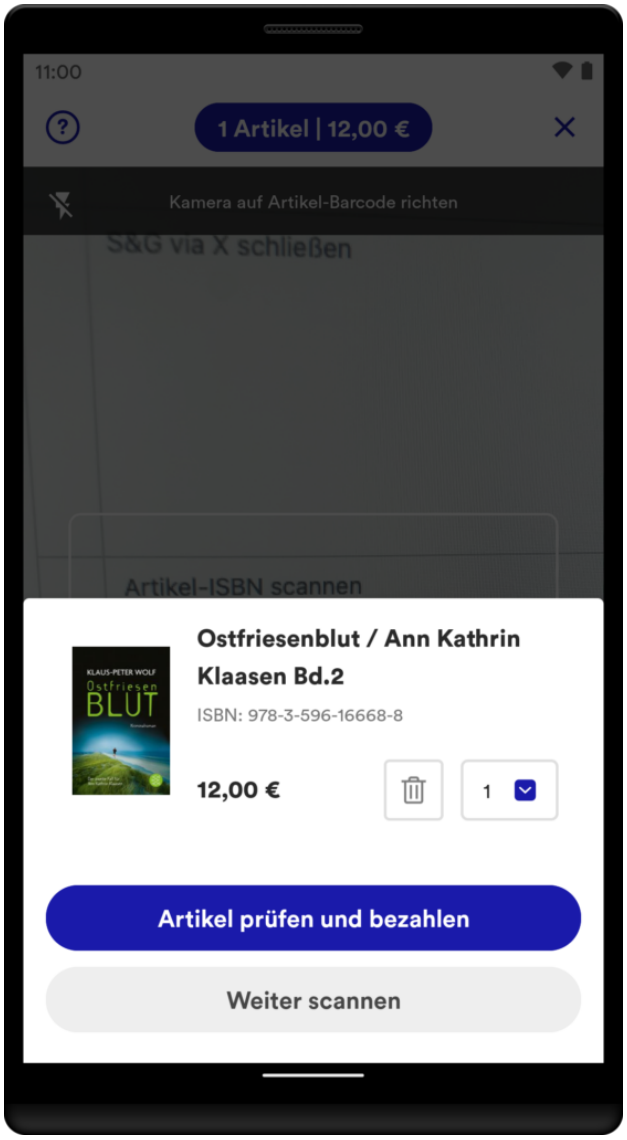


Abb. 7: Erfolgreich eingescannter Artikel

- (1) Der *HandleScannedEanUseCase* greift auf das ihm bekannte Interface *getArticle* zu und übergibt u.a. die eingescannte EAN-Nummer. Implementierungsdetails sind dem Use Case nicht bekannt.
- (2) Die Klasse *StorePaymentArticleRepositoryImpl* implementiert das Interface und greift auf die externe Datenquelle mit *getStoreArticle* zu. Die Implementierung unserer Schnittstelle *StoreArticleAPI* wird generisch mit Retrofit realisiert.
- (3) Im letzten Schritt wird die Entität *ArticleEntity* von der API in die domänenspezifische Datenklasse *StorePaymentArticle* mit einem Subset notwendiger Daten abgebildet und an das Use Case zurückgeliefert.

Final wird im Fragment der Artikel angezeigt. Es können weitere Artikel eingescannt werden. Dann werden erneut die drei beschrieben Schichten

durchlaufen.

Sicherstellung der Qualität & Automatisiertes Testen

Um dauerhaft die Qualität, Stabilität und Zukunftssicherheit von Scan & Go zu gewährleisten, haben wir umfangreiche Unit- und UI-Tests implementiert. Die Tests sollen rechtzeitig Fehler aufdecken bzw. die fachlichen Anforderungen bewahren. Zudem wird ganz erheblich der manuelle Abnahmeprozess für ein neues App-Release reduziert. Somit sind wir in der Lage, schneller zu agieren und neue Features oder Bugfixes zu releasen.

Das Schreiben von Unit-Tests ist komplett in unserem Scrum-Prozess verankert. Es ist ein bedeutender Teil unserer Definition of Done (DoD). UI-Tests hingegen sind komplexerer Natur und werden in separaten Tickets entwickelt. Diese werden plattformübergreifend gemeinsam mit der QA spezifiziert und anschließend realisiert.

Unit-Tests

Aufgrund der gewählten Architektur und der strikten Trennung in einzelne Klassen ist es leicht, Unit-Tests zu schreiben. Der große Vorteil ist, dass diese Art von Tests sehr schnell direkt in einer JVM ausführbar sind. Ein Unit-Test ist deterministisch, wird isoliert ausgeführt und testet einen kleinen Bereich des Codes (eine „Unit“) ab.

Während der Entwicklung werden bei jedem Commit einer Codeänderung auf den Entwicklungsbranch (vgl. Gitflow) automatisch alle Unit-Tests auf unserem Jenkins ausgeführt. Dadurch erhalten wir als Entwickler*innen schnelles Feedback, falls der Code Regressionsfehler enthält oder wichtige fachliche Anforderungen ausgehebelt wurden. Dementsprechend können wir rechtzeitig korrigierende Schritte einleiten.

Wir haben für Scan & Go alle Nicht-UI-Bereiche mit Unit-Tests abgedeckt. Im folgenden Codesnippet ist ein in Kotlin entwickelter Test exemplarisch abgebildet. Der Test prüft, ob falsch ausgezeichnete Barcodes (EAN, ISBN, …)

identifiziert werden.

```
class EanDigitCheckerTest {

    private val eanDigitChecker = EanDigitChecker()

    @Test
    fun givenInvalidEanCodes_whenCodeIsVerified_thenCodeIsDetectedAsInvalid() {
        assertInvalidEanCode(eanDigitChecker.verifyEan(""))
        assertInvalidEanCode(eanDigitChecker.verifyEan("-"))
        assertInvalidEanCode(eanDigitChecker.verifyEan("T"))
        assertInvalidEanCode(eanDigitChecker.verifyEan("544900009624T"))
        assertInvalidEanCode(eanDigitChecker.verifyEan("test"))
        assertInvalidEanCode(eanDigitChecker.verifyEan("9783406628178"))
    }

    private fun assertInvalidEanCode(result: Result) {
        Assert.assertFalse("Expect invalid code.", result.isValid)
        Assert.assertNotNull("Expect not empty code.", result.outputEan)
    }
}
```

Abb. 8: Unit-Test zur Prüfung von invaliden EAN-Codes

UI-Tests

Technisch anspruchsvoller ist es, die Scan & Go-Screens mittels UI-Tests automatisiert zu testen. Grund ist nicht der zu schreibende Code, sondern vielmehr, dass die Tests auf physischen Geräten bzw. Emulatoren ausgeführt werden und eine deutlich längere Ausführungszeit brauchen. Zudem wird eine stabile Infrastruktur benötigt, wie z.B. Mockserver, Testmaschinen und ein dediziertes WLAN, um nur einige Punkte zu nennen.

Aufgrund der Infrastruktur und der externen Einflüsse und Abhängigkeiten können Tests scheitern, obwohl der Code nicht geändert wurde. Diese Tests sind sog. „flaky Tests“ und sollten vermieden werden. Für das Scheitern der UI-Tests sind folgende Einflussfaktoren maßgeblich:

- Ausgelastetes Test-WLAN/Timeouts in den Service-Requests
- Stabilität von REST-Services in der Testumgebung
- Änderungen von REST-Services oder Webseiten von anderen Teams

- App-Animationen (z.B. Einblendung von Bannern)
- Dialoge (z.B. das BottomSheetDialogFragment von der Android-API)
- (Ungeplante) Systemdialoge vom Betriebssystem
- Zugriff auf die GPS-Position (z.B. Auffinden der nächsten Buchhandlung)

Wir sind die genannten Einflussfaktoren über einen längeren Zeitraum angegangen und haben für beide Plattformen Android und iOS inzwischen eine gute Erfolgsquote an positiven Tests (ca. 98 %, siehe Abbildung 9). Unser Ziel ist dennoch 100 %, um eine hohe Zuverlässigkeit mit den UI-Tests zu gewährleisten. Tests sollten ausschließlich scheitern, wenn der Code Regressionsfehler enthält oder wichtige fachliche Anforderungen ausgehebelt wurden.

Abb. 9: Testreport für unsere Kundenapp (Android)

Auf Basis dieser Vorarbeiten haben wir für jeden Scan & Go-Screen einen Test entwickelt und häufig genutzte Funktionalitäten getestet („happy path“). Aufgrund der langen Ausführungszeit haben wir auf Edge-Cases verzichtet (z.B. testen, ob ein Artikel 100 Mal in die Einkaufstasche gelegt werden kann).

Damit wir für beide Plattformen Android und iOS identische Testfälle haben, haben wir die zu testenden Schritte und deren Reihenfolge in gemeinsame Interfaces ausgelagert (siehe Abbildung 10). Ein Interface ist als Contract zwischen der Qualitätssicherung (QA) und der Entwicklung zu verstehen.

```
public protocol InstorePaymentManageArticleOnScannerFeatureInterface {
    func givenIAmAtTheInstoreArticleScannerWith(storeId: Int)
    // changed parameter eanString to ean
    func whenIAddArticleWith(ean: String)
    func thenTheArticleInformationIsDisplayedWith(displayLabel: String)
    func andTheAmountAndTotalAreUpdated()

    func thenTheFirstTimeUnknownArticleAlertIsDisplayedWith(displayLabel: String)
    func whenITapEnterManually()
    func thenTheManualInputIsSelected()
    func thenTheUnknownArticleAlertIsDisplayed()
}

public extension InstorePaymentManageArticleOnScannerFeatureInterface {

    func runTestScenarioAddKnownArticle() {
        // Is article information displayed correctly when I add a known article?
        givenIAmAtTheInstoreArticleScannerWith(storeId: 5198)
        whenIAddArticleWith(ean: "9783596166688")
        thenTheArticleInformationIsDisplayedWith(displayLabel: "ISBN: 978-3-596-16668-8")
        andTheAmountAndTotalAreUpdated()
    }

    func runTestScenarioAddKnownArticleWithHyphens() {
        // Is article information displayed correctly when I add a known article with hyphens?
        givenIAmAtTheInstoreArticleScannerWith(storeId: 5198)
        whenIAddArticleWith(ean: "978-3-596-16668-8")
        thenTheArticleInformationIsDisplayedWith(displayLabel: "ISBN: 978-3-596-16668-8")
        andTheAmountAndTotalAreUpdated()
    }
```

Abb. 10: Interface für einen gemeinsamen Testfall

Die jeweilige Plattform (Android, iOS) implementiert das Interface als normalen UI-Test (siehe Abbildung 11). Dabei müssen lediglich die atomaren Schritte implementiert werden. Die Reihenfolge der Schritte gibt das Interface in Form einer Methode vor (z.B. *runTestScenarioAddKnownArticle*). Diese Methode muss in der Testklasse für das entsprechende Testframework referenziert werden.

Im folgenden Codesnippet ist eine Implementierung mit Kotlin exemplarisch abgebildet.

```
class StorePaymentManageArticleOnScannerTest
    : StorePaymentBaseTest(), InstorePaymentManageArticleOnScannerFeatureInterface {

    @Test
    fun runTestScenarioAddKnownArticleTestCase() {
        runTestScenarioAddKnownArticle()
    }

    @Test
    fun runTestScenarioAddKnownArticleWithHyphensTestCase() {
        runTestScenarioAddKnownArticleWithHyphens()
    }

    override fun givenIAmAtTheInstoreArticleScannerWith(storeId: Int) {
        navigateToArticleScanner(storeId)
    }

    override fun whenIAddArticleWith(ean: String) {
        waitHandler.waitForView(R.id.enter_ean_search_input)
        BaristaEditTextInteractions.writeTo(R.id.enter_ean_search_input, ean)
        BaristaKeyboardInteractions.pressImeActionButton(R.id.enter_ean_search_input)
    }

    override fun thenTheArticleInformationIsDisplayedWith(displayLabel: String) {
        waitHandler.waitForText(displayLabel)
        // close dialog
        BaristaClickInteractions.clickBack()
    }

    override fun andTheAmountAndTotalAreUpdated() {
        BaristaVisibilityAssertions.assertDisplayed(RegexViewMatcher(articleSummaryRegex))
        screenshotProvider.screenshot("amount_and_total")
    }
```

Abb. 11: Implementierung eines gemeinsamen Testfalls (Android)

Für Scan & Go haben wir 24 Tests konzipiert und in fachliche Bereiche gruppiert, z.B.:

- Funktioniert der Einsprung nach Scan & Go?
- Kann zur Hilfe navigiert werden und zurück?
- Funktioniert der komplette Kaufprozess?
- Wie verhält sich das Einscannen von bekannten/unbekannten Artikeln?
- Funktioniert das Löschen eines Artikel ordnungsgemäß?

Insgesamt wurden in etwa 1200 Codezeilen implementiert. Die Tests werden

täglich nachts ausgeführt und mit einem Report begutachtet:

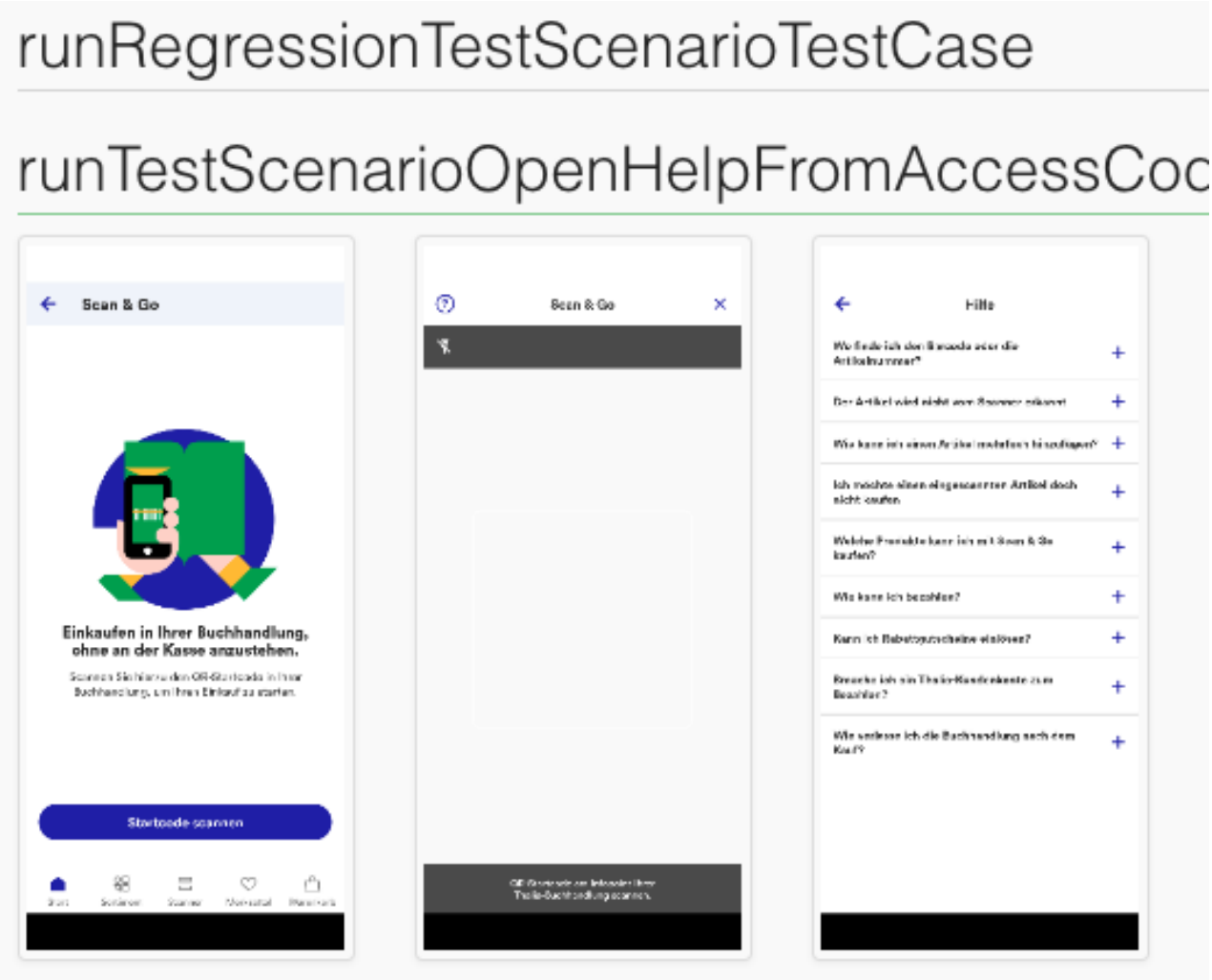


Abb. 12: Auszug aus dem Testreport (Android)

Um das Schreiben von Tests für unsere QA einfach zu halten, haben wir uns entschieden, die Testinterfaces in Swift zu implementieren. Alle Testinterfaces werden als Library in einem Nexus-Repository deployed und in Android und iOS als übliche Dependency eingebunden. Für Android wird der Swift-Code automatisch im Buildprozess nach Kotlin konvertiert.

Weiterführende Informationen rund um unsere allgemeine Teststrategie haben wir in einem separaten Blog beschrieben: https://tech.thalia.de/testen-einer-app-in-der-hybriden-welt/.

Probleme & Lösungen während der Implementierung

Lichtverhältnisse

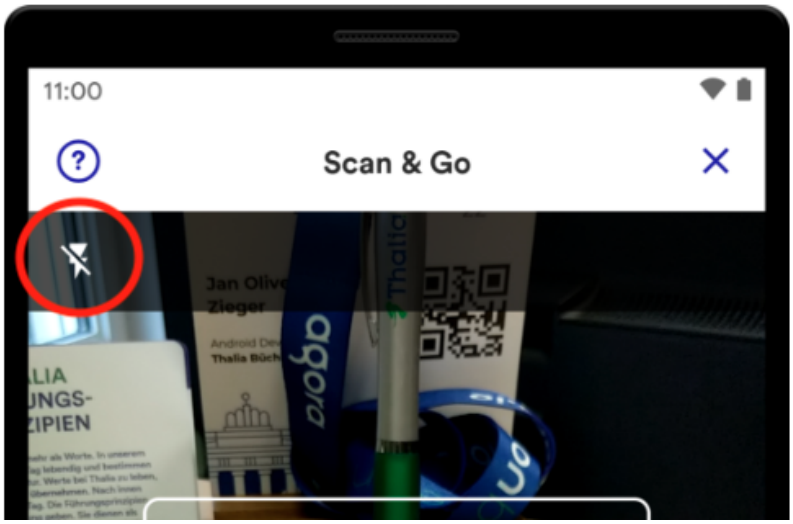


Abb. 13: Aktivierung des Lichts

Die Lichtverhältnisse in den verschiedenen Buchhandlungen variieren und sind aus technischer Sicht nicht immer optimal. Die Kamera des Gerätes hat unter Umständen Schwierigkeiten, den Barcode auf den Artikeln zu erkennen. Um dieses Problem zu lösen, bieten wir den User*innen eine einfache Möglichkeit, direkt beim Einscannen des Artikels das Licht des Gerätes zu aktivieren. Das technische Aktivieren des Lichts ist mit der Standard-API des jeweiligen Betriebssystems gelöst (z.B. *androidx.camera.coreCameraControl*).

Reduzierung der Serverlast

Um den User*innen ein bequemes und flüssiges Einscannen zu ermöglichen, werden während des Scanvorgangs die Aufnahmen der Kamera fortwährend automatisch analysiert und versucht, den Barcode des Artikels zu erfassen. Für jede erfolgreiche Erfassung eines Barcodes wird dieser an unser REST-Backend gesendet. Bei einer erfolgreichen Suchanfrage werden die Artikeldaten instantan geliefert und in der UI präsentiert.

Die Optimierung besteht darin, nicht valide Barcodes zu erkennen und dem Backend vorzuenthalten. Das mindert nicht nur die Serverlast, sondern spart auch Datenvolumen der Kund*innen ein. Die Prüfung erfolgt direkt in der App und verifiziert den Barcode anhand der letzten Zahl im Barcode, der sog. Prüfziffer (vgl. https://de.wikipedia.org/wiki/European_Article_Number).

Für Android haben wir den Algorithmus für die Prüfung beispielhaft in Kotlin wie

folgt implementiert:

```
fun verifyEan(eanCode: String): Result {
    val cleanedDigits = eanCode.replace(" ", "").replace("-", "")

    if (cleanedDigits.toLongOrNull() == null) {
        return Result(outputEan = eanCode, isValid = false)
    }

    val eanDigits = toEan(cleanedDigits)
    val digitsToCheck = eanDigits.dropLast(n = 1).reversed()
    val detectedChecksum = Character.getNumericValue(eanDigits.last())

    val computedChecksum = computeChecksum(digitsToCheck)
    val isValid = computedChecksum == detectedChecksum

    return Result(outputEan = eanDigits, isValid = isValid)
}
```

Abb. 14: Implementierung der Prüfung von Barcodes

Die App wertet das Ergebnis aus und sendet entsprechend die Ausgabe an das REST-Backend oder bricht ab und setzt den Scanvorgang fort.

WLAN

Sofern das Gerät nicht mit einem WLAN verbunden ist, kann eine App träge wirken und zu einer geringeren Zufriedenheit führen. Insbesondere ist die Netzabdeckung innerhalb der Buchhandlungen und der Shoppingcenter oft unzureichend. Aus diesem Grund bieten wir in den Buchhandlungen kostenlose WLAN-Zugänge an. Einmal mit dem WLAN verbunden, kann unsere App optimal performen, da die Artikeldaten signifikant schneller geladen werden können. Zudem wird kein mobiles Datenvolumen verbraucht.

Nach der Entwicklung der ersten Version für Scan & Go haben wir das Feedback erhalten, dass das Einscannen schneller sein könnte. Des Weiteren war den User*innen nicht bekannt, dass ein WLAN-Zugang in den Buchhandlungen vorhanden ist. Diese Probleme sind wir zügig angegangen. Ziel war in einer nächsten App-Version den Scan & Go-Vorgang flüssiger von der Hand gehen zu lassen und den User*innen eine verbesserte User Experience anzubieten.

Abb. 15: Einblendung Hinweisbanner für das WLAN

Wir haben in der App einen Hinweisbanner entwickelt, der angezeigt wird, wenn der QR-Code- oder der Artikel-Scanner geöffnet wird. Der Hinweisbanner wird nach ein paar Sekunden von oben eingeblendet. Er macht darauf aufmerksam, das WLAN in der Buchhandlung zu nutzen. Für den Absprung in die WLAN-Einstellungen wird die Standard-API des jeweiligen Betriebssystems verwendet.

Fazit & Ausblick

Aufgrund der gewählten technischen Architektur und der Teststrategie können wir mit dem Feature Scan & Go unseren Kund*innen einen Mehrwert anbieten, um schnell und intuitiv in einer Buchhandlung einkaufen zu können. Wir arbeiten weiter fokussiert an der Thematik und möchten aus Sicht des Kunden Scan & Go kontinuierlich erweitern und verbessern.

Langfristig wollen wir den Scan & Go-Prozess verschlanken und noch einfacher für unser Kund*innen gestalten. Beispielsweise möchten wir zukünftig das initiale Einscannen des QR-Startcodes obsolet machen. Damit wir wissen, in welcher Buchhandlung Scan & Go verwendet wird, müsste eine Ortungstechnologie herangezogen werden (WLAN-Ortung, Bluetooth Beacon, GPS, Geofencing, …). Beim Start von Scan & Go wird automatisch die Buchhandlung ermittelt und das Einscannen des QR-Startcodes entfällt. Das ist zudem nachhaltiger, da die Werbemittel zum Bedrucken des QR-Codes (Aufsteller, Flyer, …) eingespart werden können.

Weitere Ideen werden gesammelt und priorisiert umgesetzt. Im Entwicklungsprozess halten wir uns an Scrum und durchlaufen vereint alle Fachdisziplinen (UX, PO, QA, DEV), um ein stimmiges Endprodukt zu erreichen.

Anhang

	Android	**iOS**
Sprache	Kotlin, teilweise Java (legacy Code)	Swift, SwiftUI, teilweise Objective C (legacy Code)
OS-Version	>= Android 8	>= iOS 14
IDE	Android Studio	Xcode
CI/CD	Jenkins, git, GitLab, gradle, Bash, Pipeline-Scripting	Jenkins, git, GitLab, xcodebuild, Bash, Fastlane
Netzwerk, Datenbank	Retrofit, Room	Alamofire, CoreData
QR-Code- und Text-Erkennung	Google ML Kit, CameraX	AVFoundation
Unterstützte Scan-Codes	FORMAT_EAN_13, FORMAT_EAN_8 FORMAT_UPC_E, FORMAT_QR_CODE FORMAT_CODE_128	EAN13Code, EAN8Code, UPCECode, QRCode, Code128Code
Test	Espresso, Roboletric, Mockito, JUnit Barista	XCTestCase, XCTest
Animation	Lottie	Lottie

- https://www.thalia.de/vorteile/thalia-app
- https://www.thalia.de/vorteile/scan-go
- https://tech.thalia.de/
- https://tech.thalia.de/testen-einer-app-in-der-hybriden-welt/
- https://lottiefiles.com/what-is-lottie

- https://developer.android.com/training/testing/espresso
- https://github.com/AdevintaSpain/Barista
- https://robolectric.org/
- https://site.mockito.org/
- https://developers.google.com/ml-kit
- https://developer.android.com/training/camerax
- https://developers.google.com/ml-kit/reference/android
- https://developer.apple.com/av-foundation/
- https://developer.apple.com/documentation/avfoundation/avmetadataobject/objecttype
- https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html
- https://developer.android.com/topic/architecture
- https://de.wikipedia.org/wiki/European_Article_Number
- https://www.atlassian.com/git/tutorials/comparing-workflows/gitflow-workflow
- https://www.youtube.com/watch?v=jCHEASuHVAc
- https://www.thalia.de/shop/home/artikeldetails/A1039840971
- https://www.esri.com/en-us/arcgis/products/arcgis-ips/overview

---

Testen einer App in der hybriden Welt

Die Thalia-App konsumiert von anderen Teams entwickelte Endpunkte. Wir beleuchten die technologischen und organisatorischen Herausforderungen beim Testen.

Stressfrei ins neue Schuljahr dank Machine Learning

Im Zuge der Weiterentwicklung der Thalia App entstehen regelmäßig neue Features. Dabei versuchen wir bereits bestehenden Code durch Refactorings für uns, aber auch für den Kunden, zu verbessern. Nach der Pfadfinderregel hinterlässt jeder im Team die Codebasis ein bisschen besser als vorgefunden.

Problemstellung und bisheriger Prozess

Beim Aufräumen des Android Manifest bin ich auf eine Activity (repräsentiert eine Ansicht innerhalb einer App) gestoßen, die sich mit Schulbüchern befasst. Thalia hat jedes Jahr zwischen Juni und September Aktionen für Schulartikel. Dort können Hefte, Kalender oder auch Schulbücher für den Start in das neue Schuljahr gekauft werden. Die Schüler erhalten von ihren Lehrern in der Regel eine Liste mit den benötigten Materialien am Anfang des neuen oder zum Ende des vergangenen Schuljahrs. Eine solche Liste beinhaltet gewöhnlich eine European Article Number (kurz: EAN), einen Titel und einen Preis.

Die Liste kann mithilfe der Kamera über die Thalia App fotografiert und im Anschluss per Mail an Thalia übermittelt werden. Das Foto wird mit weiteren Informationen von der App an einen E-Mail-Client auf dem Gerät übergeben und für den Bestellprozess an schulbuch@thalia.de verschickt. Mit wenigen Klicks und dem Verfassen einer E-Mail waren die benötigten Materialien für das neue Schuljahr bestellt. Der Prozess ist funktional – aber kann dieser für den Kunden noch einfacher gemacht werden?

Idee zur Verbesserung

Gemäß eines unserer Schlüsselverhalten bei Thalia versuchen wir unsere Lösungen spielerisch einfach zu gestalten. Der vorher beschriebene Prozess funktioniert, lässt aber einige Fragen offen: Können die Artikel nur per Rechnung bezahlt werden oder lässt die Bestellung per Mail auch andere Zahlungsmöglichkeiten zu? Wie wird am einfachsten kenntlich gemacht, wenn ein Artikel nicht bestellt werden soll? Wie wird verfahren, wenn das Bild unkenntlich

oder abgeschnitten ist? Grundsätzlich könnten die Bücher auch vom Kunden selbst gesucht und bestellt werden. Dadurch wären die vorherigen Fragen obsolet, aber das Abtippen einer längeren Liste erscheint weniger praktisch als die Aufnahme dieser mit der Kamera. An der Stelle kam die neue Idee ins Spiel: Machine Learning.

Machine Learning oder maschinelles Lernen ist ein Teilgebiet der Informatik und erkennt durch Muster und Training von Algorithmen Muster in Datensätzen. Auf unseren Anwendungsfall übertragen ist der Text auf dem Zettel, der über das Foto aufgenommen wird, unser Datensatz. Der Kern der Idee besteht darin, den Text auf dem Zettel für eine Suche in unserem Sortiment auszulesen. Prädestiniert im Datensatz sind die EANs. Diese sind eindeutig und würden, im Gegensatz zur manuellen Suche (oder Suche über eine Schnittstelle) nach einem Titel, mit Sicherheit den gewünschten Artikel auffinden.

Für die Erkennung von Text auf Bildern stellt Google mit dem Machine Learning Kit (ML Kit) eine Bibliothek für Android und iOS zur Verfügung[1]. Das versprochene Ergebnis nach Analyse eines Bildes ist ein Objekt vom Typ FirebaseVisionText[2]. Dieses ist entweder leer, wenn kein Text erkannt wurde, oder enthält je nach Präsenz des Textes auf dem Bild Textblöcke, die wiederum in Textzeilen aufgeteilt sind, oder einen einzelnen Satz.

Auf den Schulzettel übertragen würden alle EANs, Titel und Preise erkannt und in einem FirebaseVisionText-Objekt bereitgestellt werden. Durch eine Filterung ließen sich dann die EANs über einen regulären Ausdruck auslesen. Diese könnten dann via API-Call gesucht und das Ergebnis in einer Liste dargestellt werden. Mit einem Klick wandern die gefundenen Artikel in den Warenkorb und der Nutzer kann selbstständig die benötigten Schulmaterialien bestellen. Die Umsetzung der beschriebenen Theorie erfolgt im nächsten Kapitel.

Implementierung Android

Die Aufnahme des Bildes wird von der Kamera App des Android OS gehandelt. Wir geben dem entsprechenden Intent eine Uniform Resource Identifier (URI) mit, unter der das Bild nach der Aufnahme gespeichert werden soll.

```
val values = ContentValues()
values.put(MediaStore.Images.Media.TITLE, "titel_des_bildes")
outputFileUri = activity?.contentResolver?.insert(MediaStore.Images.Media.EXTERNAL_CONTENT_URI, values)

takePhotoIntent.putExtra(MediaStore.EXTRA_OUTPUT, outputFileUri)
startActivityForResult(takePhotoIntent, REQUEST_SCHOOLBOOK_IMAGE)
```

Abbildung 1: Intent für die Aufnahme durch die Kamera App

Das aufgenommene Bild wird in unserer Analyzer Klasse als Bitmap erstellt und anschließend unter Zuhilfenahme der ML-Kit-Bibliothek zu einem FirebaseVisionImage konvertiert.

```
private fun prepareImage(imageURI: Uri, rotation: Float): FirebaseVisionImage {
    val matrix = Matrix()
    matrix.postRotate(rotation)
    val original : Bitmap! = Media.getBitmap(context.contentResolver, imageURI)
    val bitmap : Bitmap! = Bitmap.createBitmap(original, x: 0, y: 0, original.width, original.height, matrix, filter: true)
    return FirebaseVisionImage.fromBitmap(bitmap)
}
```

Abbildung 2: Konvertierung in FirebaseVisionImage

Dabei wird auch die ursprüngliche Rotation zum Bild hinzugefügt. Die Konvertierung ist erforderlich, damit die Erkennung des Textes anschließend stattfinden kann. Das vorbereitete Bild wird daraufhin mittels eines FirebaseVisionTextRecognizer analysiert und gibt uns das im Theorieteil erwähnte FirebaseVisionText-Objekt zurück.

```
private fun decodeText(firebaseVisionImage: FirebaseVisionImage, result: ScanCallback) {
    val detector : FirebaseVisionTextRecognizer = FirebaseVision.getInstance().onDeviceTextRecognizer
    detector.processImage(firebaseVisionImage)
        .addOnSuccessListener { firebaseVisionText : FirebaseVisionText! ->
            // replace all hyphens and whitespaces
            val scannedText : String = firebaseVisionText.text.replace( oldValue: "-", newValue: "")
            val results : List<String> = regex.findAll(scannedText).toMutableList().map { it.value }
            if (results.isNullOrEmpty()) {
                result.onScanNextRotation()
            } else {
                result.onSuccess(SchoolbookResultObject(cleanScanningResults(results)))
            }
        }
}
```

Abbildung 3: Analyse des aufgenommenen Bildes

Aus dem gescannten Text löschen wir die Bindestriche. Teilweise werden diese auf den Schullisten mit dargestellt, sind für die Suche aber unwichtig. Der bereinigte Text wird durch den regulären Ausdruck überprüft. Dabei werden nur Strings herausgefiltert, die Zahlen zwischen null und neun enthalten und eine Länge von zehn bis 13 besitzen. Die Längen ergeben sich aus der Länge der EAN mit 13 Stellen und der Internationalen Standardbuchnummer (kurz ISBN), die

zehn (ISBN-10) oder 13 (ISBN-13) Stellen besitzt. Diese kann analog zur EAN für unsere Suche genutzt werden und wird teilweise auch auf den Schullisten abgedruckt. Die für die Suche unwichtigen Bindestriche stellen den Unterschied zwischen der EAN und einer ISBN-13 dar.

Anpassungen nach Rollout Android

Nachdem das erste Minimum Viable Product fertiggestellt und bereits in einer neuen App Version veröffentlicht wurde, haben wir noch weitere Verbesserungen vorgenommen.

Eine Maßnahme bestand darin, die ausgelesenen EANs (oder ISBNs) auf ihre Länge hin zu untersuchen.

```
when (current.length) {
    13 -> cleanedList.add(current)
    10 -> cleanedList.add(current)
    11 -> if (current.first() == '8') {
        cleanedList.add(current.removePrefix( prefix: "8"))
    } else if (current.first() == '9') {
        cleanedList.add(current.removePrefix( prefix: "9"))
    }
    12 -> if (current.substring(0, 2) == "78") {
        cleanedList.add(current.removePrefix( prefix: "78"))
    } else if (current.substring(0, 2) == "79") {
        cleanedList.add(current.removePrefix( prefix: "79"))
    }
}
```

Abbildung 4: Bereinigung der gefundenen Ergebnisse

Bei den Längen zehn und 13 können wir uns sicher sein, dass die Nummer korrekt vom Bild ausgelesen wurde. Tauchen im Ergebnis Strings mit den Längen elf oder zwölf auf, wurden entweder ein oder zwei Zeichen zu wenig erkannt (für eine ISBN-13 oder eine EAN) oder es wurden ein oder zwei Zeichen zu viel erkannt (für eine ISBN-10). Die ersten drei Stellen bei einer ISBN-13 (oder EAN) im Kontext der Schulbuchzettel bestehen aus den Zahlenfolgen „978“ (oder „979“). Dabei handelt es sich um Präfixe für das Medium Buch. Da auf den

Schulbuchzetteln in der Regel nur Bücher zu finden sind und wir nicht sicher sein können, ob zu viel oder zu wenig erkannt wurde, löschen wir das Präfix im untersuchten Text. Allerdings auch nur, wenn das Präfix darauf hindeutet, dass eine oder zwei Stellen der ISBN nicht erkannt wurden. Innerhalb der drei möglichen Ziffernfolgen (ISBN-13, EAN und ISBN-10) handelt es sich bei der letzten Stelle um eine Prüfziffer. Durch ein Streichen des vorangestellten Präfixes ist die Prüfziffer nicht korrekt. Die mit dieser fehlerhaften Nummer angesprochene Schnittstelle übernimmt die Fehlerkorrektur, sodass wir keine Prüfziffer neu berechnen müssen. Bei Nummern mit weniger als zehn oder mehr als 13 Stellen hat die Analyse zu sehr gestreut und es wird keine Bereinigung vorgenommen.

Eine weitere Verbesserung haben wir bezüglich der Rotation entwickelt. Das Problem bestand darin, dass die Analyse des Textes auf dem Bild weniger Erfolg hat, wenn dieses z. B. um 90 Grad gedreht war. Das heißt, das Bild wurde z. B. im Landscape-Modus aufgenommen und die Analyse hat im Portrait-Modus ohne Rotation in diesem stattgefunden.

```
if (rotation >= 360) {
    emitter.onError(Exception("No EAN/ISBN numbers found."))
} else {
    rotation += 90f
    decodeText(prepareImage(imageURI, rotation), result: this)
}
```

Abbildung 5: Rotation des Bildes für weitere Analyse

Die Lösung für das Problem besteht darin, das Bild solange zu rotieren, bis Ergebnisse gefunden werden. Wenn die Analyse ohne Ergebnis bleibt findet eine erneute Analyse mit einem um 90 Grad rotierten Bild statt. Dieser Vorgang wird so oft wiederholt, bis die Rotation das Bild einmal komplett um 360 Grad gedreht hat oder bis ein Ergebnis vorliegt.

Scannen des Schulzettels in der App

Nach der Theorie und der Beschreibung des Codes soll in Form von Screenshots das Feature in der App gezeigt werden. Als erstes erfolgt die Aufnahme des Bildes.

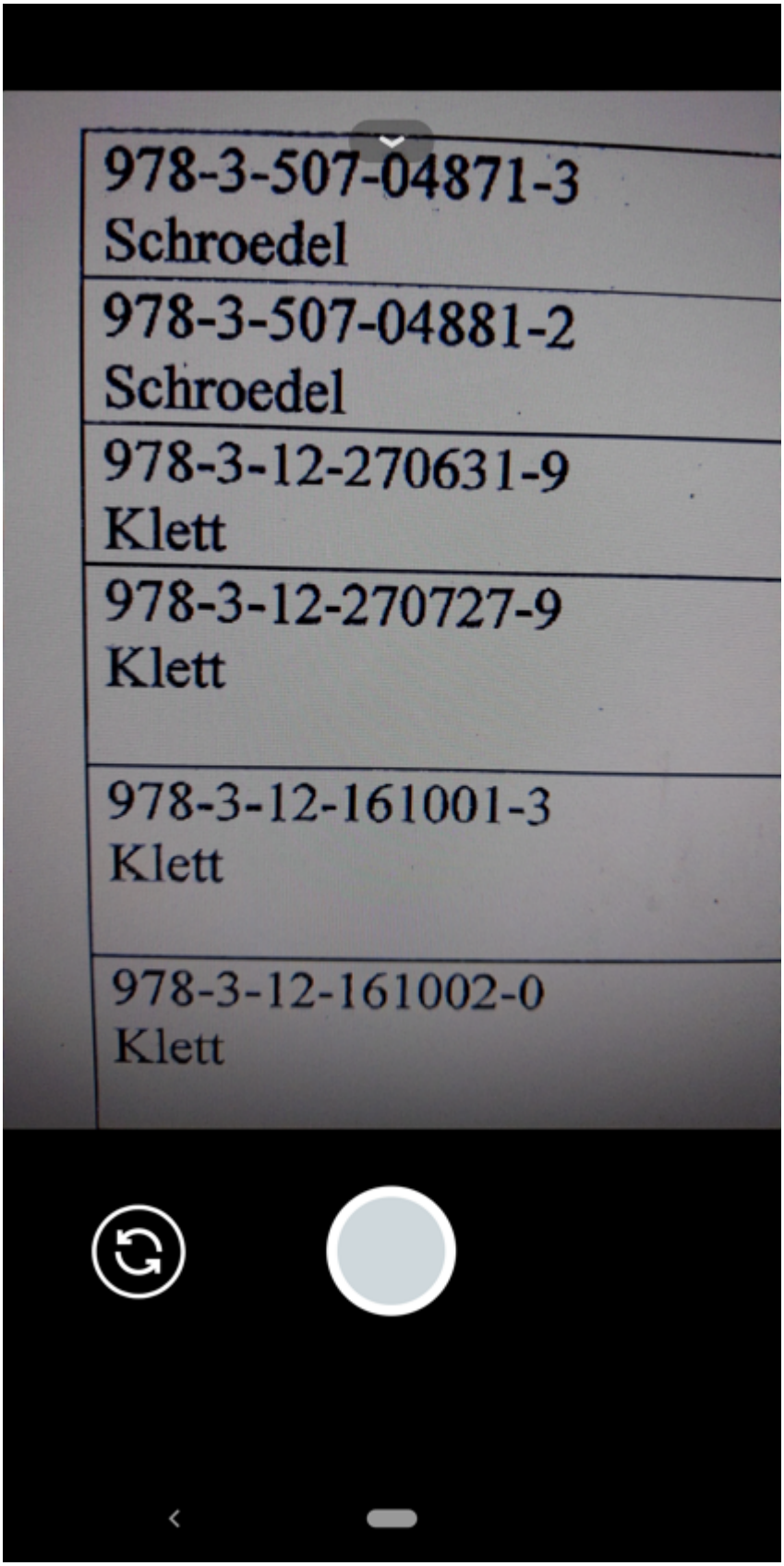


Abbildung 6: Aufnahme des Bildes mit der Kamera App

Anschließend wird das Bild analysiert und die Ergebnisse in einer Liste dargestellt.

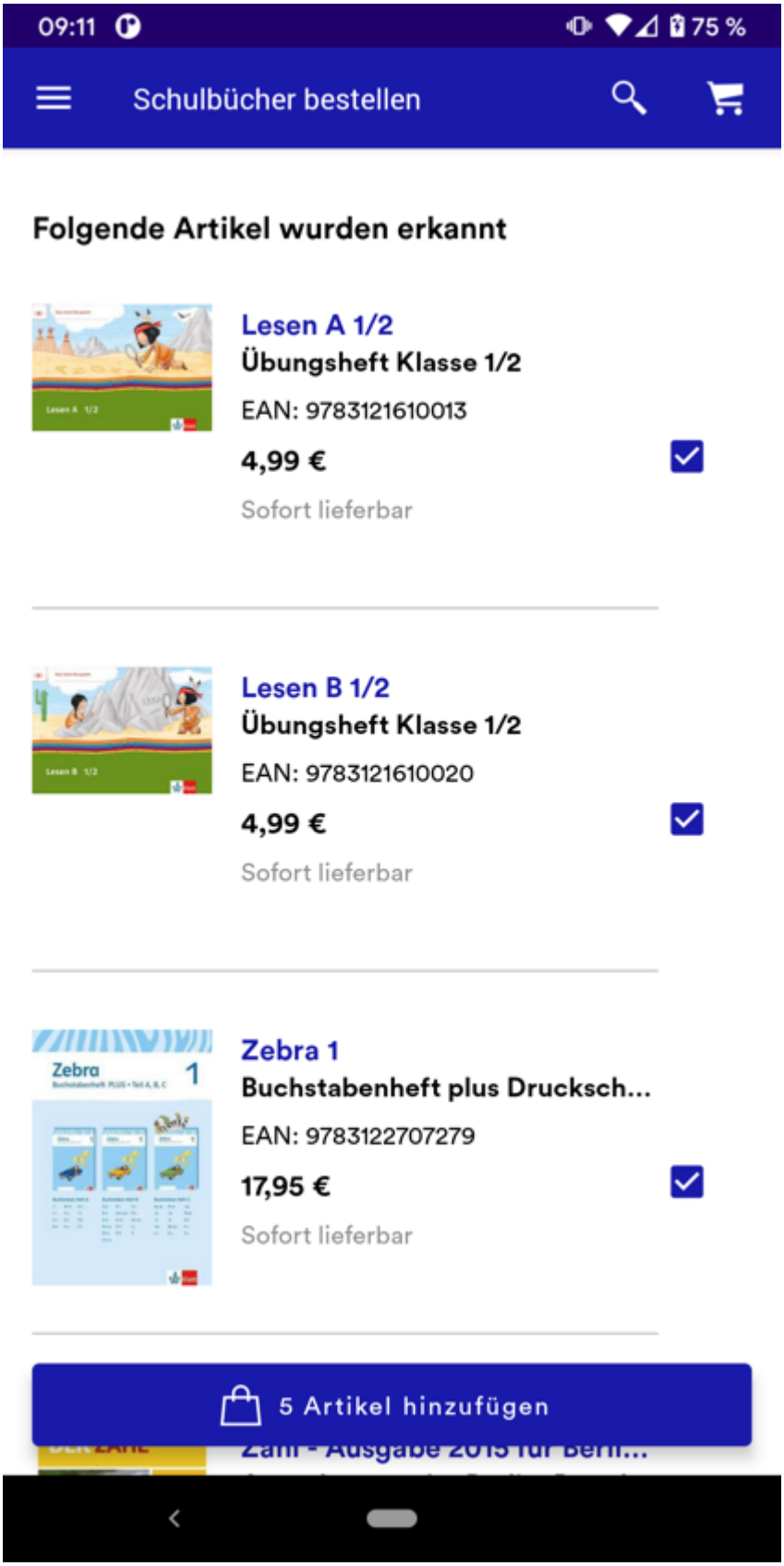


Abbildung 7: Ergebnisliste nach der Analyse des aufgenommenen Bildes

Der Kunde hat die Möglichkeit, die benötigten Materialien in den Warenkorb zu legen.

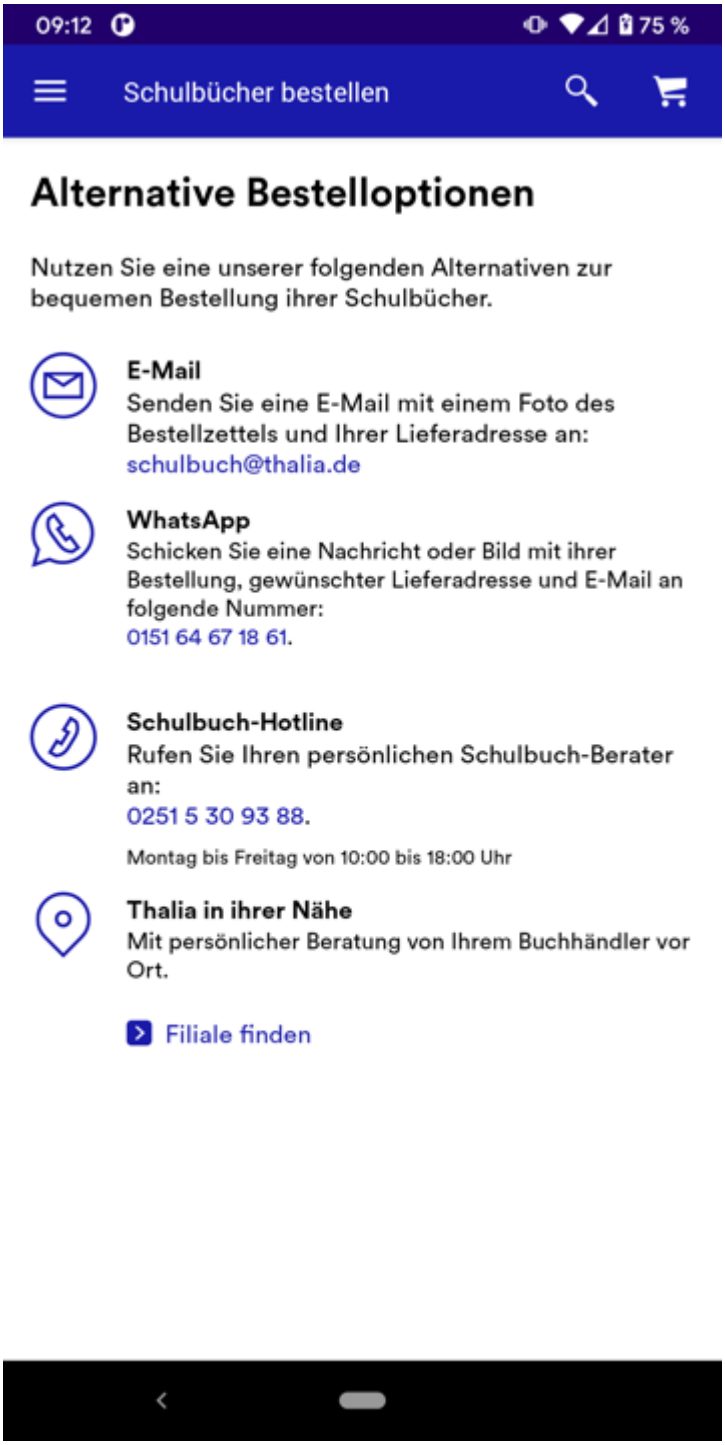


Abbildung 8: Alternativoptionen bei fehlerhafter Analyse

Falls bei einem Scan etwas schiefläuft und keine Ergebnisse gefunden werden, sind Alternativen verfügbar. Eine davon ist der eingangs beschriebene Prozess mit der Versendung des Bildes per E-Mail.

Fazit und Ausblick

Der Schulbuchscanner hat den Bestellprozess für Schulmaterialien anhand einer Liste deutlich erleichtert. Für den Kunden ist kein mühsames Eingeben der EANs

oder das Versenden eines Bildes erforderlich. Mit dem Scanner und der Analyse eines Fotos können alle benötigten Materialien unter Nutzung der gewünschten Bezahlmethode direkt an die Wunschadresse bestellt werden. Auch die Mitarbeiter bei Thalia profitieren. Dank des Scanners kommen deutlich weniger Anfragen bei schulbuch@thalia.de an, wodurch eine geringere Anzahl an Bestellungen für die Kunden manuell vorgenommen werden muss.

Künftig könnte die generelle Scanning-Funktionalität auch für andere Use-Cases verwendet werden. Die Erkennung von Text auf Bildern ist, wie der Schulbuchscanner gezeigt hat, möglich. Allerdings wird der Anwendungsfall hier dadurch erleichtert, dass ein Großteil des Textes durch einen regulären Ausdruck herausfilterbar ist.

---

[1] https://developers.google.com/ml-kit/vision/text-recognition

[2] https://firebase.google.com/docs/reference/android/com/google/firebase/ml/vision/text/FirebaseVisionText

---

SonarQube Integration in einem Android Projekt

Um eine bessere Code Qualität an unseren Softwareprodukten zu gewährleisten haben wir uns im Unternehmen entschlossen eine statische Code Analyse einzuführen.
Ein besonderer Augenmerk liegt hierbei auf der Code Coverage und das Einhalten von zuvor festgelegten Programmier-Richtlinien.

In diesem Artikel geht es um das Erstellen+Anzeigen von Testreports auf einem SonarQube Server mittels Jenkins, um das Herunterladen+Anzeigen dieser in Android Studio und um das Anzeigen von lokalen, neuen SonarQube Issues.

Desweiteren wird noch kurz auf die SonarQube-Benutzer-Oberfläche eingegangen.

SonarQube ist ein Tool für die statische Code Analyse. Hierfür werden zuvor erstellte Test Reports von SonarQube eingelesen und nach bestimmten Richtlinien und Regeln ausgewertet.

Vorbereitung des lokalen Projekts

Zuerst muss das Android-Projekt so konfiguriert werden, dass es Jacoco Test Reports erstellt. Dafür wird eine jacoco.gradle Datei erstellt:

```
apply plugin: 'jacoco'

ext {
    coverageSourceDirs = 'src/test/java'
}

jacoco {
    toolVersion = "0.8.3"
    reportsDir = file("$buildDir/reports")
}

task jacocoTestReport(type: JacocoReport, dependsOn: "testDebugUnitTest") {
    group = "Reporting"
    description = "Generate Jacoco coverage reports for Debug build"

    reports {
        xml.enabled = true
        html.enabled = true
    }

    def excludes = ['**/R.class',
                    '**/R$*.class',
                    '**/build/generated/**',
                    '**/*$ViewBinder*.*',
                    '**/*$InjectAdapter*.*',
                    '**/*Injector*.*',
                    '**/BuildConfig.*',
                    '**/Manifest*.*',
                    '**/*Test*.*',
                    '**/CiMattersApplication*.*',
                    'android/**/*.*']

    def debugTree = fileTree(
            dir: "$buildDir/intermediates/javac/debug/compileDebugJavaWithJavac/classes",
            excludes: excludes)
    def kotlinDebugTree = fileTree(
            dir: "$buildDir/tmp/kotlin-classes/debug",
            excludes: excludes)
    def mainSrc = "${project.projectDir}/src/main/java"

    classDirectories = files([debugTree], [kotlinDebugTree])
    executionData = files("$buildDir/jacoco/testDebugUnitTest.exec")
    sourceDirectories = files([mainSrc])
}
```

In dieser Datei wird der Gradle-Task zur Erstellung des Test-Reports angelegt und die entsprechenden Pfade für die Source- und Class Dateien für die Java- und Kotlin Klassen angegeben.

In der build.gradle Datei des Moduls muss dann noch die jacoco.gradle Datei hinzugefügt werden.

```
apply from: '../jacoco.gradle'

android {
```

Zum Schluss müssen noch einige Konfigurations-Einstellungen für Sonar in den gradle.properties angegeben werden.

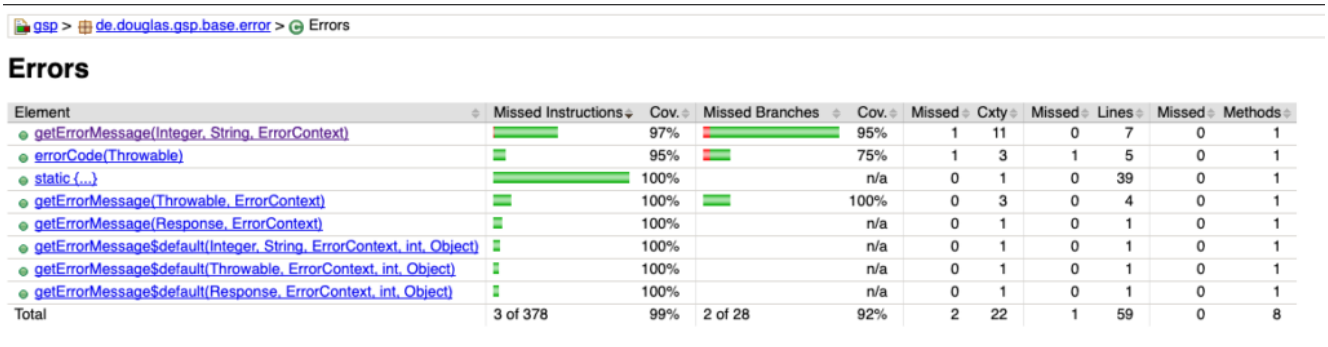
```
sonar.projectVersion=3.2.2.0.1
# ------------------
sonar.projectKey=gsp
sonar.projectName=Douglas Beauty Tab (Android)
sonar.modules=gsp
sonar.sources=src/main/java/de/douglas/gsp
sonar.scm.provider=git
sonar.java.binaries=build/intermediates/javac/debug/compileDebugJavaWithJavac/classes/de/douglas/gsp
sonar.coverage.jacoco.xmlReportPaths=build/reports/jacocoTestReport/jacocoTestReport.xml
```

Nun können mit den folgenden Gradle-Tasks die Test-Reports erstellt werden.

1. **Build**
   – baut das Projekt und generiert die benötigten Class-Datein
2. **testDebugUnitTest**
   – führt die Tests aus und erstellt einen jacoco Test-Report (modul/build/jacoco/testDebugUnitTest.exec)
3. **jacocoTestReport**
   – erstellt die Test-Reports im HTML Format (modul/build/reports/jacocoTestReport/)

Es
kann auch nur der jacocoTestReport-Task ausgeführt werden, da dieser eine Abhängigkeit zu den anderen Tasks beinhaltet. Bei einem cleanen Projekt benötigt dies aber mehr Zeit.

Nach dem Ausführen der Gradle-Tasks liegen die Reports im Modul-Ordner unter build/reports/jacoco/TestReports im HTML-Format vor und können im Browser angezeigt werden.


gsp > de.douglas.gsp.base.error > Errors

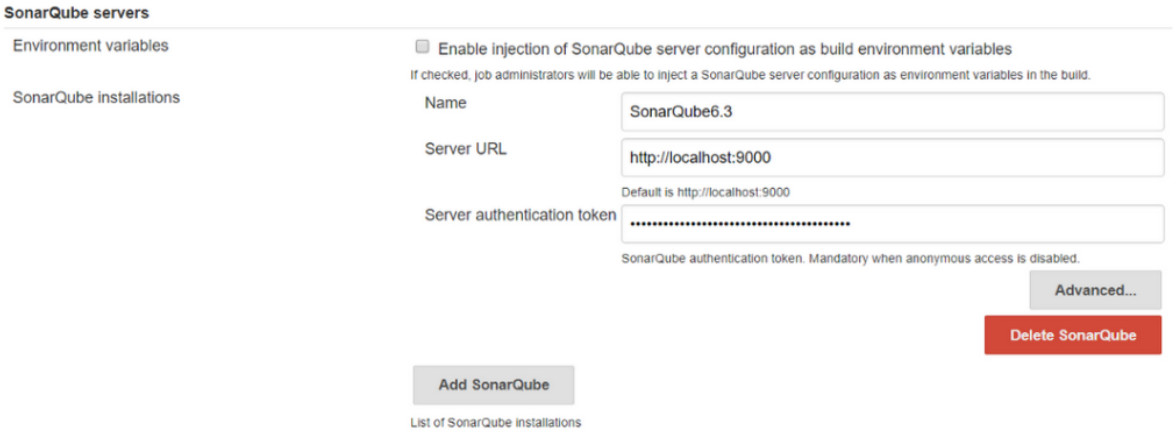
Errors

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
getErrorMessage(Integer, String, ErrorContext)		97%		95%	1	11	0	7	0	1
errorCode(Throwable)		95%		75%	1	3	1	5	0	1
static {...}		100%		n/a	0	1	0	39	0	1
getErrorMessage(Throwable, ErrorContext)		100%		100%	0	3	0	4	0	1
getErrorMessage(Response, ErrorContext)		100%		n/a	0	1	0	1	0	1
getErrorMessage$default(Integer, String, ErrorContext, int, Object)		100%		n/a	0	1	0	1	0	1
getErrorMessage$default(Throwable, ErrorContext, int, Object)		100%		n/a	0	1	0	1	0	1
getErrorMessage$default(Response, ErrorContext, int, Object)		100%		n/a	0	1	0	1	0	1
Total	3 of 378	99%	2 of 28	92%	2	22	1	59	0	8

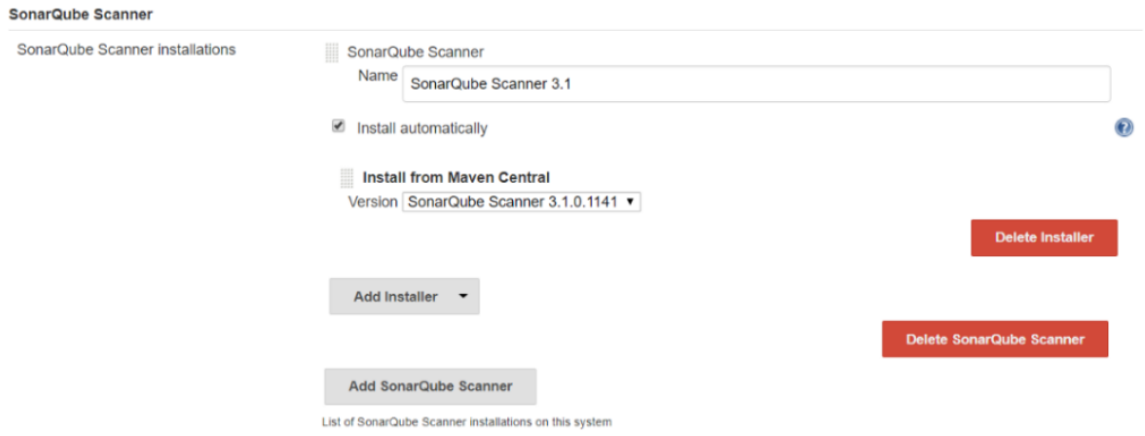
Vorbereitung des Jenkins-Servers

Um sich die Reports im Sonar an zu schauen, kann man diese über den Jenkins-Server zu Sonar übertragen.
Im Jenkins muss zuerst das Sonar-Plugin installiert werden, bei dem die URL des Servers eingetragen wird und der Sonar Runner ausgewählt werden.

Die Einstellung für das Sonar-Plugin finder man unter:
Manage Jenkins -> Configure System -> SonarQube

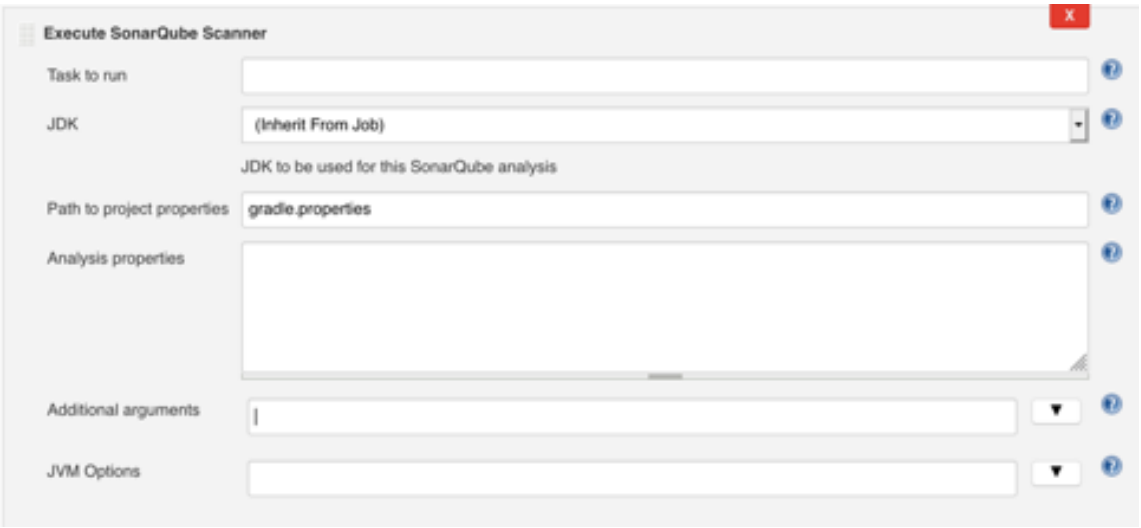


Die Einstellung für den Sonar-Runnar befindet sich unter:
Manage Jenkins -> Global Tool Configure -> SonarQube Scanner



In dem Jenkins-Job müssen dann noch die folgenden Gradle-Tasks eingetragen werden:

Anschließend wird der SonarQube Scanner konfiguriert. Hier muss die Datei mit den Properties für Sonar angegeben werden.

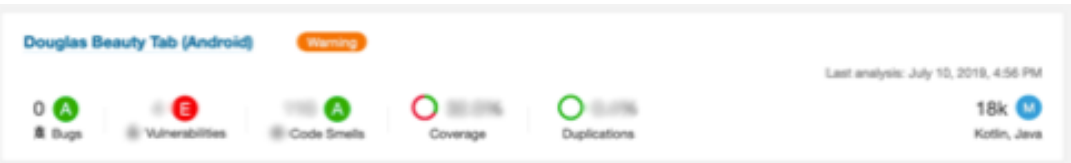


Nun wird jedes Mal, wenn der Jenksins-Job für das Projekt ausgeführt wird, auch der Sonarqube Scanner ausgeführt und veröffentlicht seinen Report auf dem SonarQube-Server.

Benutzeroberfläche des Sonar-Servers

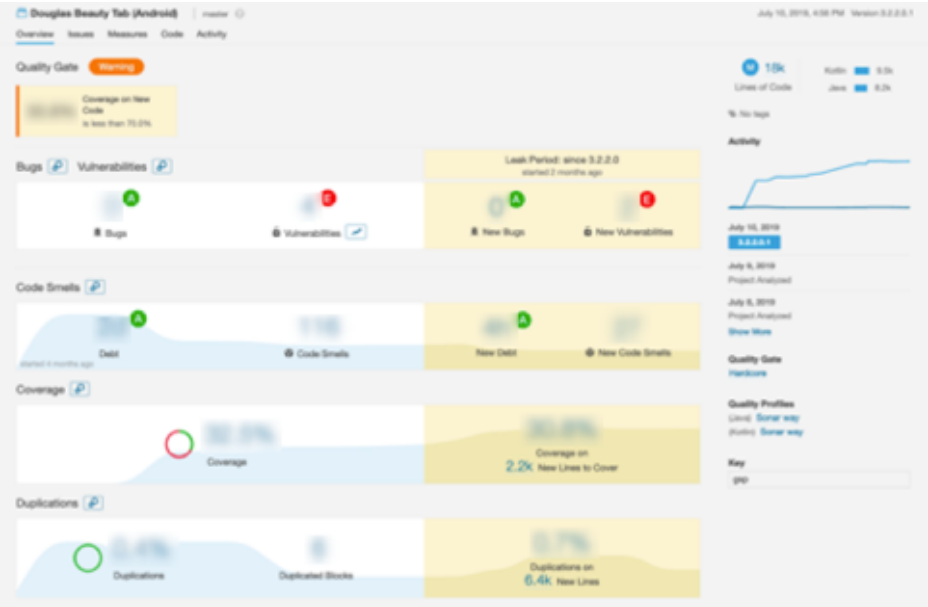
Damit Sonar die Testreports empfangen kann und diese korrekt angezeigt werden, müssen hier noch das SonarKotlin-, das SonarJava- und das Git-Plugin installiert werden.

Auf der Startseite von Sonar werden alle Projekte aufgelistet und eine Übersicht der Analyse angezeigt.

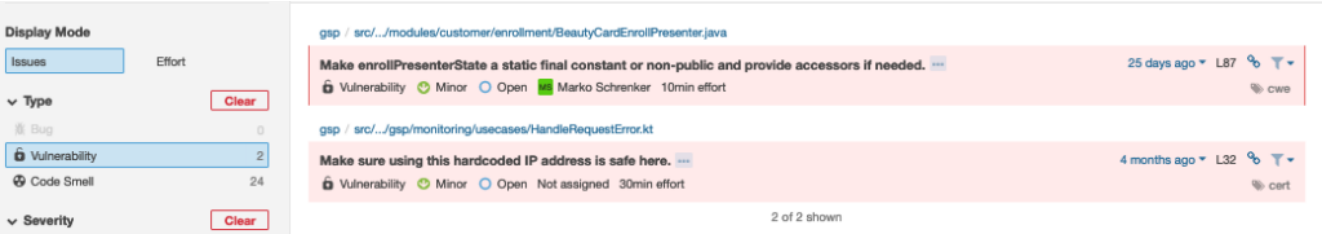


Klickt man auf ein Projekt, werden detailliertere und zusätzlich Informationen angezeigt.

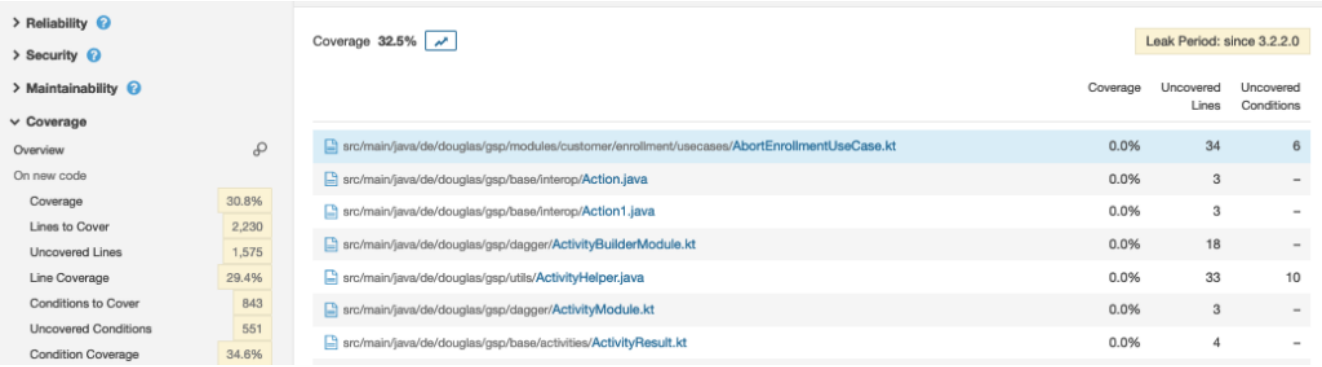
Man kann die Analyse in 2 Kategorien einteilen. Das sind zum Einen Messungen (Code Coverage, Duplication) die SonarQube anhand des Codes durchführt und zum Anderen Issues (Bugs, Vulnerabilities, Code Smells), die durch Verletzung von zuvor festgelegten Code Richtlinien auftreten. Eine zusätzliche Richtlinie kann man für ein sogenanntes Quality Gate festlegen. Dies kann zum Beispiel eine Mindestanforderung von der Code Coverage des neu entwickelten Codes sein.



Über die einzelnen Analyse-Kategorien gelangt man zu einer detaillierten Ansicht der einzelnen Issues oder Messungen.



Isseus (Bugs, Vulnerabilities, Code Smells)



Messungen (Code Coverage, Duplications)

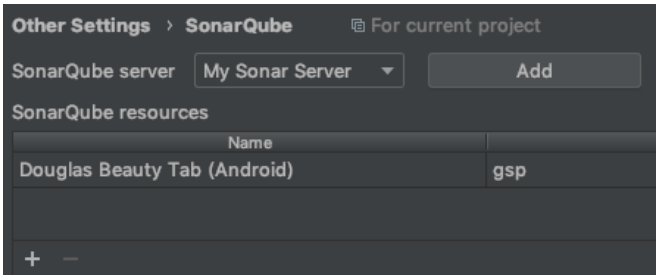
SonarQube Community Plugin

Für
Android-Studio gibt es das SonarQube Community Plugin, welches 2 Funktionen mit sich bringt:

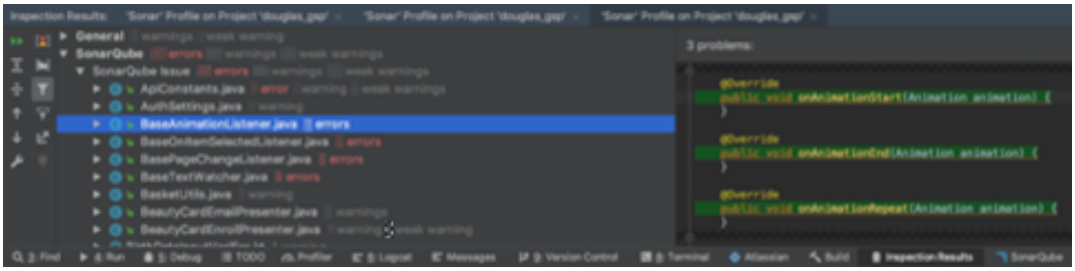
1. Anzeigen von vorhandenen Issues vom Sonar-Server
2. Anzeigen von neuen Issues mittels einer lokalen Projekt-Analyse

Vorhandene Issues vom Sonar-Server

Um sich die vorhandenen Issues vom Server herunterzuladen, muss das Plugin zuerst konfiguriert werden. Hierzu ruft man dieses in den Einstellungen von Android-Studio auf und trägt den Sonar-Server ein. Anschließend wählt man das Projekt in dem Abschnitt Resourcen aus.

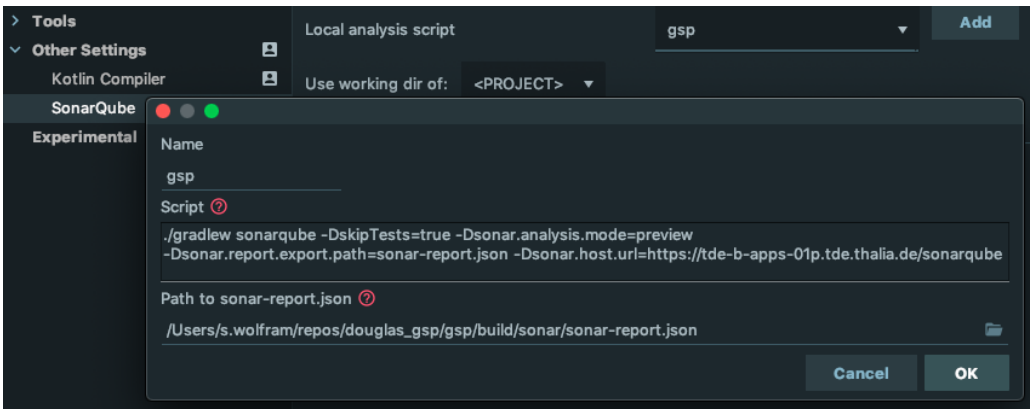


Unter dem Reiter Analyse -> Inspect Code muss in dem Inspection Profile SonarQube ausgewählt werden. Nun werden bei der Code Inspection die Sonar-Issues vom Server heruntergeladen und in der IDE angezeigt.



Lokale Projekt-Analyse mit dem Sonar-Gradle-Plugin

Um das lokale Projekt nach Sonar-Issues zu analysieren muss zunächst das SonarQube-Script konfiguriert werden. Hier muss der Sonar-Gradle-Task und der Pfad zum sonar-report eingestellt werden. Der Sonar-Report ist vor dem ersten Ausführen noch nicht vorhanden, da er erst durch den Gradle-Task erstellt wird.



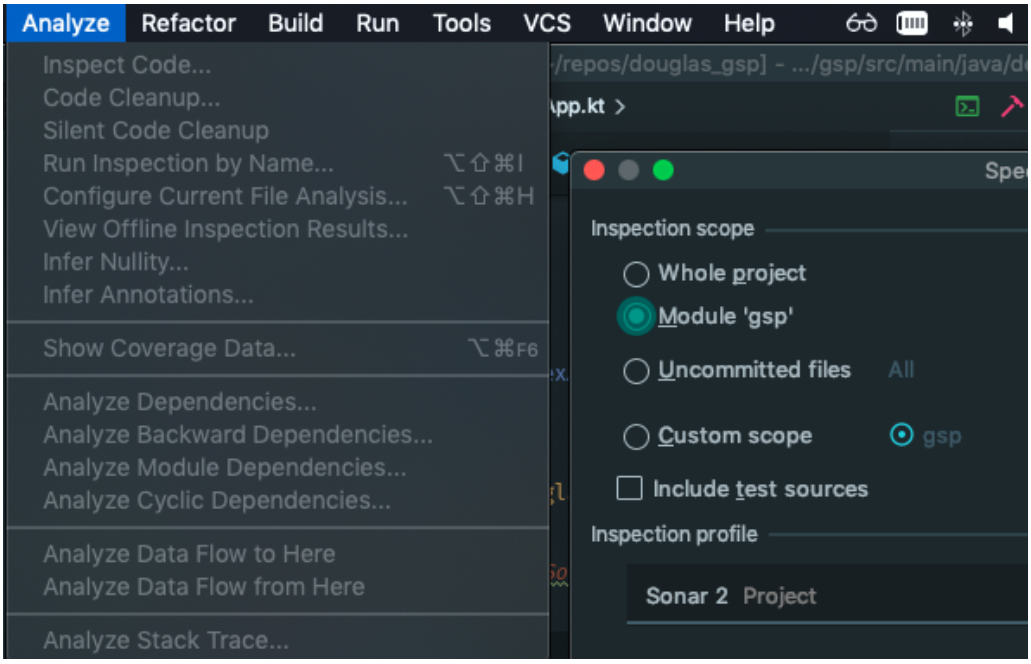
Als nächster Schritt muss lokal SonarQube konfiguriert werden. Hierzu wird zuerst in der build.gradle Datei des Projekts sonarQube als Dependencie hinzugefügt.

```
dependencies {
    classpath "org.sonarsource.scanner.gradle:sonarqube-gradle-plugin:2.7.1"
```

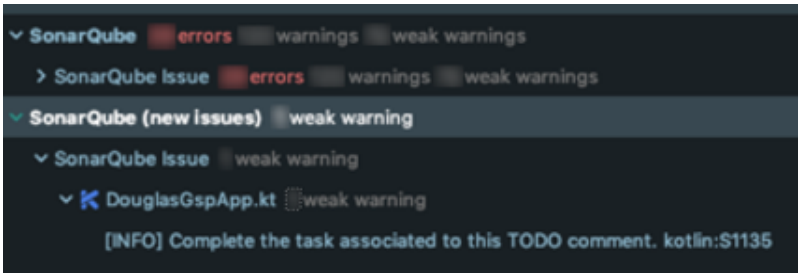
In der build.gradle Datei des Moduls wird SonarQube dann applied und konfiguriert.

```
apply plugin: 'org.sonarqube'
sonarqube {
    properties {
        property 'sonar.projectName', 'Douglas Beauty Tab (Android) - From Local'
        property "sonar.host.url", "https://tde-b-apps-01p.tde.thalia.de/sonarqube"
        property "sonar.jacoco.reportPaths", "build/jacoco/testDebugUnitTest.exec"
        property "sonar.projectVersion", version
        property "sonar.scm.provider", "git"
    }
}
```

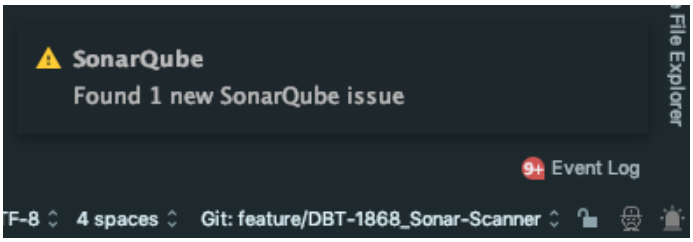
Nachdem SonarQube konfiguriert wurde, kann das Projekt unter Analyze -> Inspect Code analysiert werden. Hierzu muss nur noch das Sonar-Profile (mit Sonar Issues + new Sonar Issues) ausgewählt und mit OK bestätigt werden.



Nach der Analyse werden die Sonar-Issues vom Server und die aus der lokalen Analyse unten im Reiter Inspection Results angezeigt.



Neu gefundenen lokalen SonarQube Issues werden zusätzlich noch in einem kleinen Popup unten rechts von Android Studio angezeigt.



Nach der Einführung von SonarQube in unser Android-Projekt haben wir eine zentrale Anlaufstelle für unsere Code Qualität und kontrollieren regelmäßig diese in der SonarQube-Benutzer-Oberfläche.
Desweiteren bekommen wir über Jenkins ein schnelles Feedback ob unser Quality Gate eingehalten wurde.
Mit dem Community-Plugin laden wir uns die SonarQube-Issues vom Server herunter und beheben diese direkt in Android Studio. Bevor wir ein Feature pushen, kontrollieren wir mittels des Plugins im Vorfeld das Einhalten der Programmier Richtlinien und können neu aufgetretene Issues direkt in Android

Studio beheben.

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.

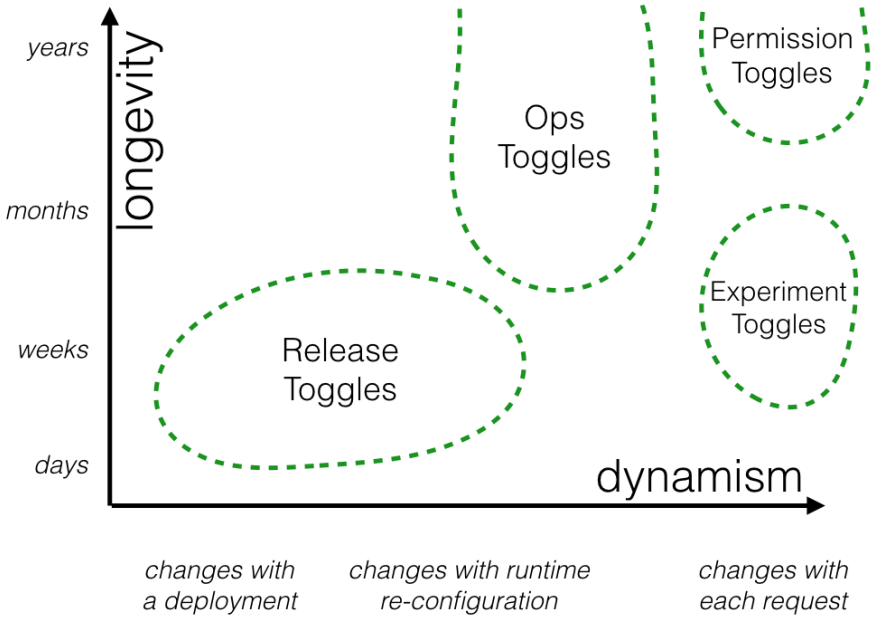
Feature Toogles

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.

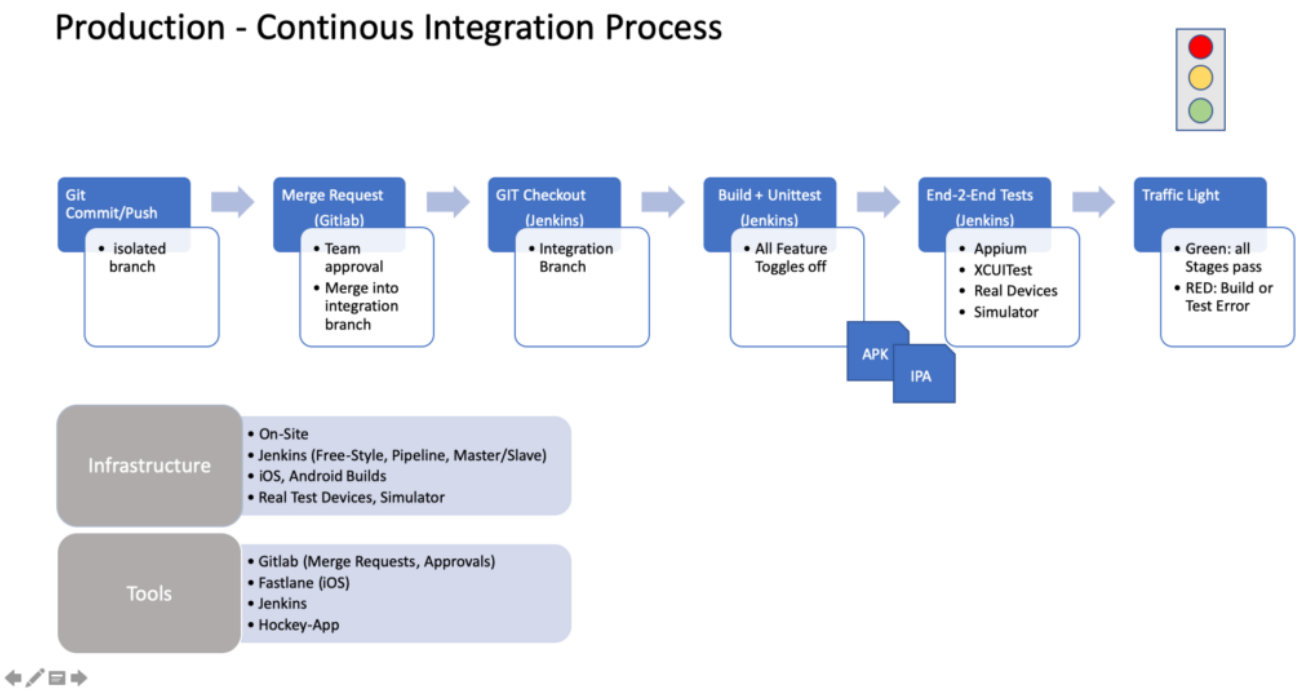
CI-Build-Status für Android und iOS

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.

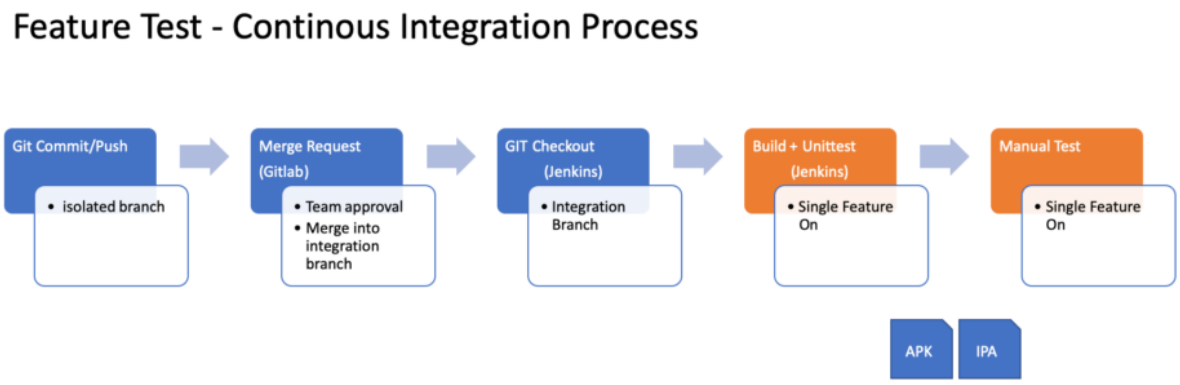
Continous Integration Process

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.



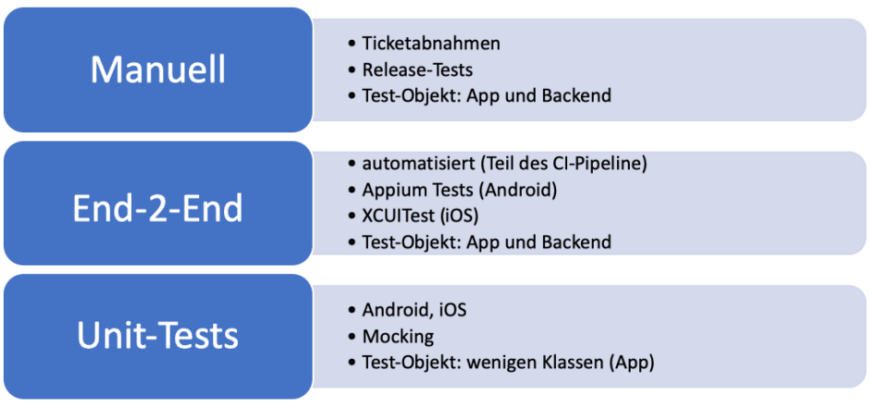
Für den Test eines Features, dass sich noch in der Entwicklung befindet, wird ein Feature-Toggle manuell im Jenkins aktiviert, die App gebaut und die Unittests ausgeführt. Die End-2-End Tests werden zum jetzigen Zeitpunkt noch nicht ausgeführt. Zum einen müssten die Tests für das Feature bereits angepasst 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