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

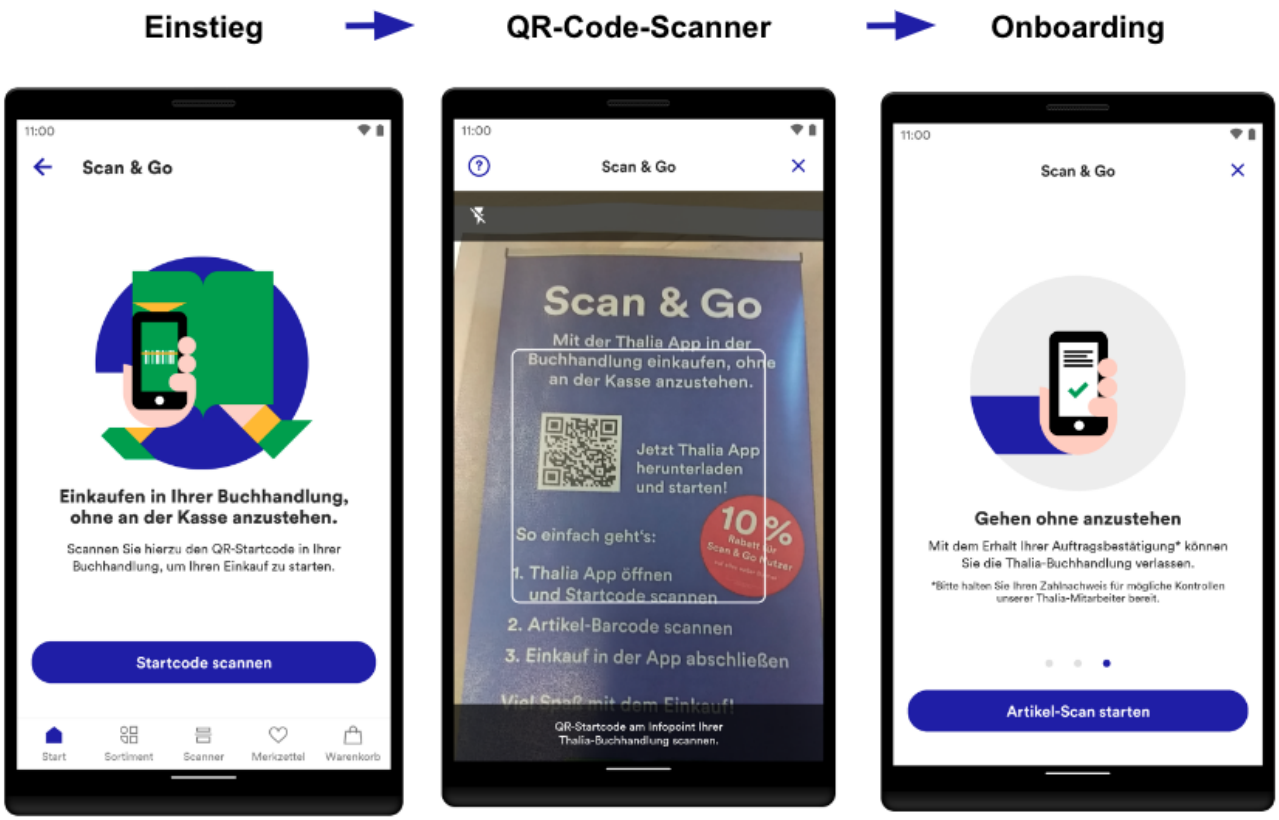
innerhalb der App bequem getätigt und die Buchhandlung, ohne an der Kasse anstehen zu müssen, sorgenfrei verlassen werden.

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).



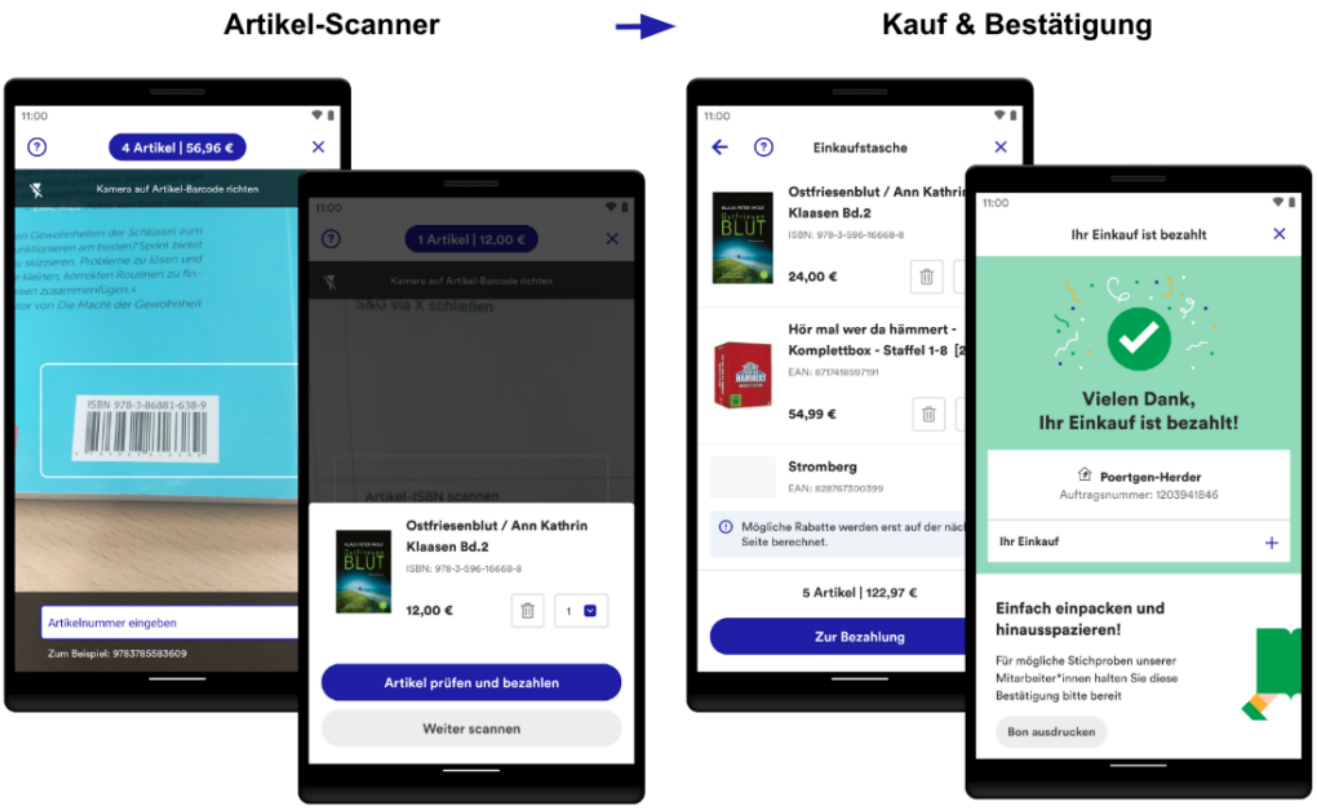


Abb. 3: Übersicht von Scan & Go

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):

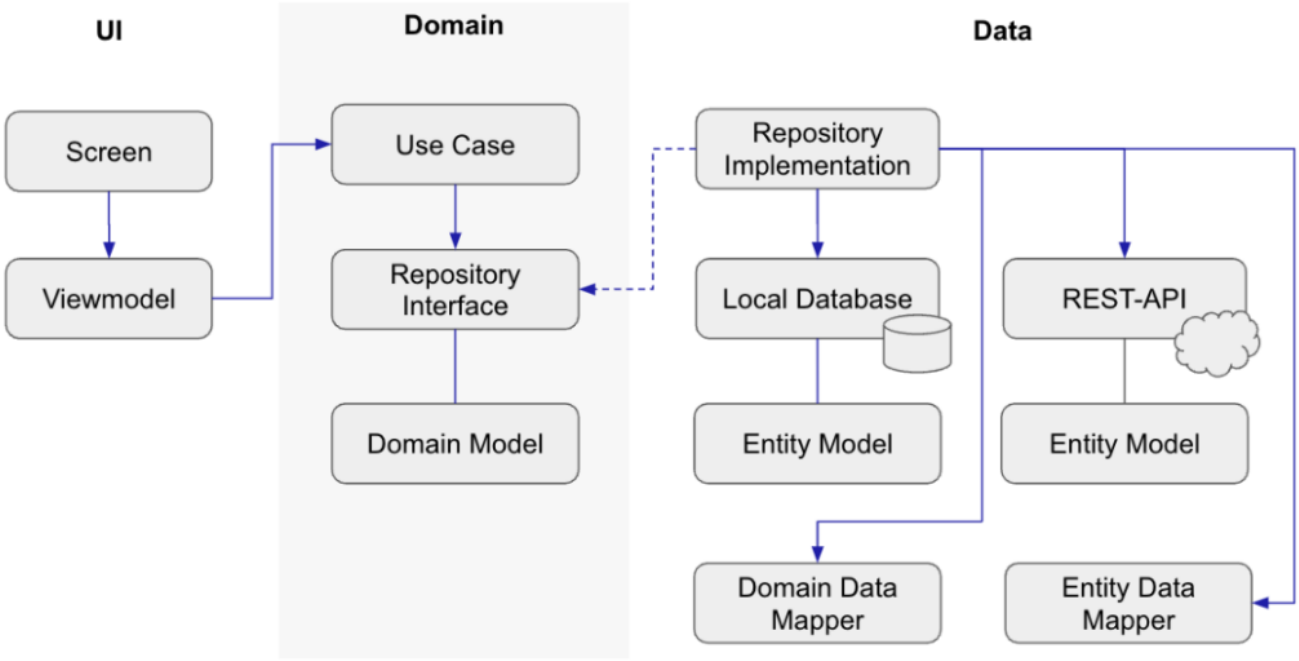


Abb. 4: Grober Aufbau einer UI-Klasse

- 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

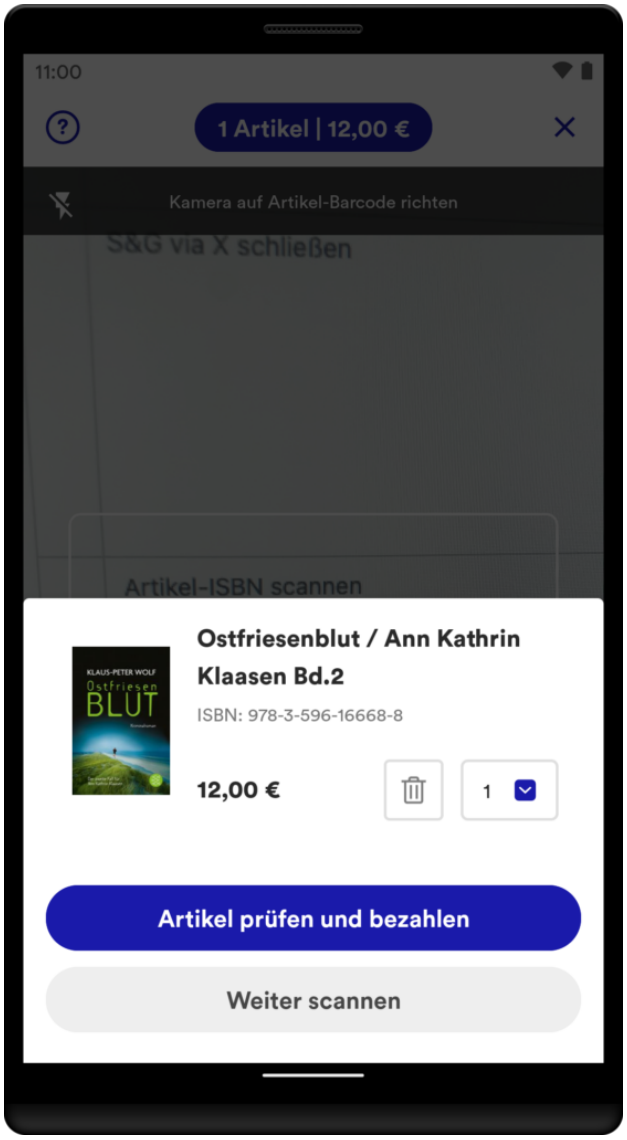


Abb. 7: Erfolgreich eingescannter Artikel

- (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:

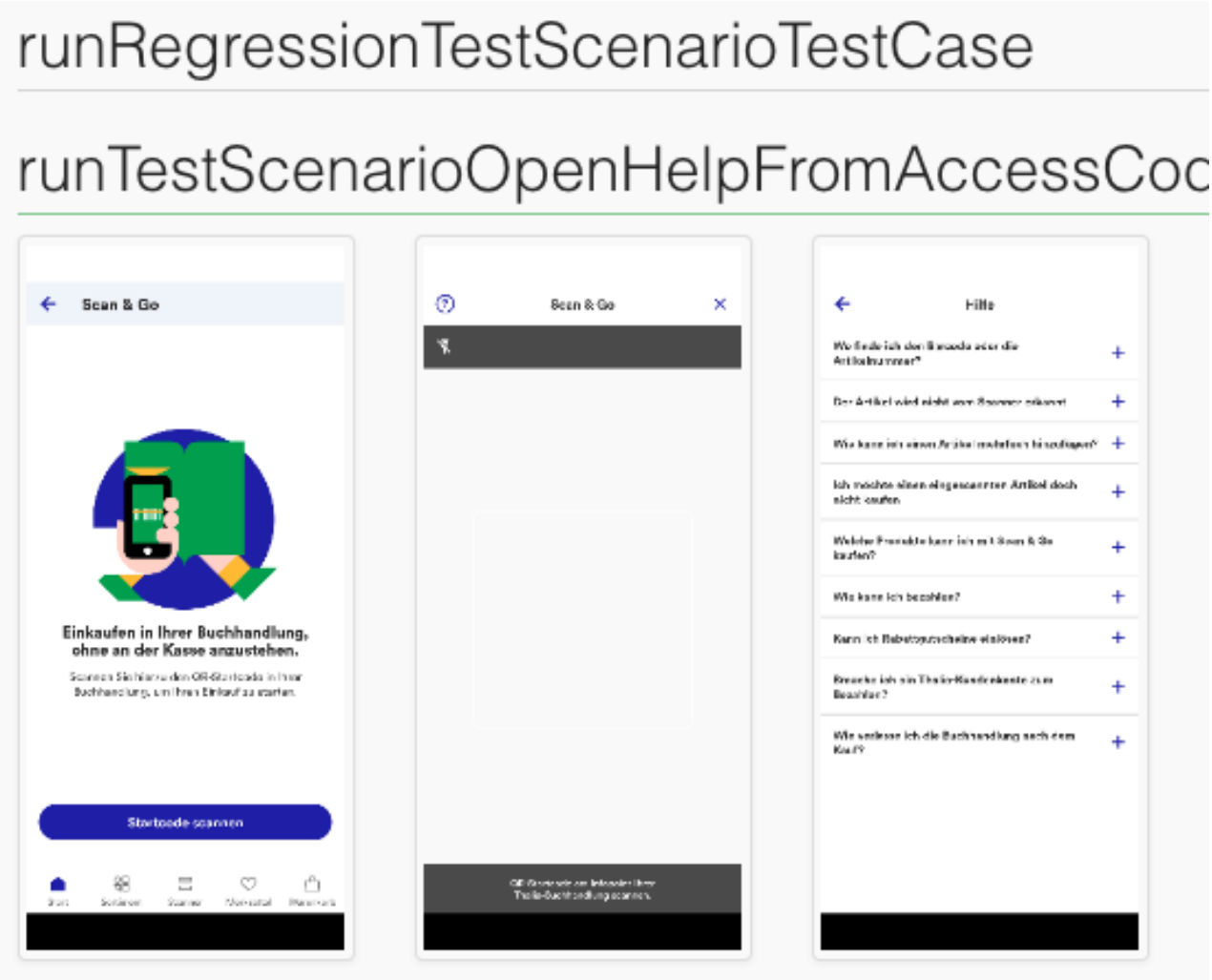


Abb. 12: Auszug aus dem Testreport (Android)

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

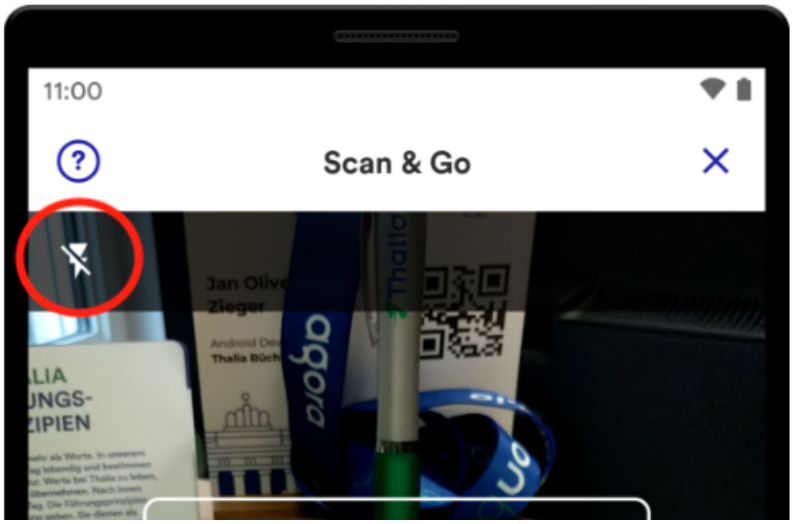


Abb. 13: Aktivierung des Lichts

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