Testen einer App in der hybriden Welt

In der eCommerce-Welt von Thalia gibt es neben cross-funktionalen Teams, welche den Webshop u. a. unter thalia.de in bestimmten Domänen (z. B. Aftersales) betreuen und weiterentwickeln, auch sogenannte Touchpoint-Teams. Diese zeichnen sich dadurch aus, dass sie einen kompletten Touchpoint zum Kunden betreuen und weiterentwickeln. In unserem Fall handelt es sich um die Thalia App.

Während das App-Team als ein „geschlossener" Touchpoint zu den Kund*innen hin auftritt, haben wir mit allen cross-funktionalen Teams innerhalb der Organisation Schnittstellen. Diese lassen sich grob in API-Aufrufe und Webviews, d. h. die Anzeige von Webseiten innerhalb der App, aufteilen. Das führt zu verschiedenen Herausforderungen in Bezug auf die übergreifende Qualitätssicherung:

- Verantwortlichkeit: Wenn jedes Team seinen eigenen Code testet, wer (und wie) testet (man) dann über Teamgrenzen hinweg?
- Redundanz: In welchem Umfang ist Redundanz akzeptabel, erwünscht, notwendig?

Nimm du ihn, ich habe ihn sicher!

Der Webshop ist ein lebendes Objekt. In den letzten Jahren wurde vieles getan, um den alten Monolithen durch eine Self-Contained-Systems-

Architektur abzulösen. Insgesamt arbeiten derzeit sechs cross-funktionale Teams mit verschiedenen Schwerpunkten an ihren jeweiligen (oder übergreifenden) Themen, lösen die allerletzten Reste aus dem Monolithen heraus und entwickeln Features weiter oder neu, um das Kund*innenerlebnis besser zu machen. Mit gängigen CI/CD-Praktiken werden mehrmals täglich Änderungen produktiv genommen bzw. veröffentlicht. Diese können sich auch auf die App auswirken, wenn sie beispielsweise die eingebundenen Webviews betreffen. Auch im nativen Teil tut sich einiges: Wir entwickeln die App weiter und veröffentlichen in regelmäßigen Abständen neue Versionen.

Bei so viel Veränderung muss automatisiert – also durch automatisierte Tests – sichergestellt werden, dass die einzelnen Teile in sich funktionieren, aber auch ineinandergreifen.

Einfach: Testen, was man selbst entwickelt

Stark vereinfacht kann man sagen, dass der native Teil der App ein (vom Webshop) separates Frontend darstellt, welches auf REST-Schnittstellen der cross-funktionalen Teams aufsetzt. Um die Funktionalität dieses Frontends zu testen, verwenden wir zum einen Unit-Tests und zum anderen UI-Tests. Werden Änderungen im Code vorgenommen, führen wir diese beiden Arten von Tests aus. Bei den UI-Tests werden die Backend-Daten aus den REST-Schnittstellen gemockt, ähnlich wie z. B. bei WireMock. So nimmt man die potenzielle Fehlerquelle Backend aus der Gleichung und hat zusätzlich die Möglichkeit, Fehlerzustände durch HTTP-Fehlercodes zu simulieren.

Auf diese Weise stellen wir sicher, dass wir – das App-Team – mit unseren Änderungen nichts kaputt machen bzw. die Qualität beibehalten und wir bei Fehlern frühestmöglich reagieren können.

Fortgeschritten: Die REST-Schnittstellen anstöpseln aka den nativen Teil integrativ

testen

Um zu prüfen, ob das Zusammenspiel zwischen den REST-Schnittstellen und dem nativen App-Code so funktioniert, wie wir es vom Mockserver simulieren lassen, führen wir unsere UI-Tests auch gegen die tatsächlichen Schnittstellen aus. Diese Ausführung findet nächtlich statt und ist leider nicht ganz so stabil wie die Variante mit Mockserver.

Über die Zeit haben wir verschiedene potenzielle Fallstricke identifiziert, die diese Ausführungen fehlschlagen lassen, obwohl eigentlich nichts kaputt ist (ja, sicher…). Dazu gehören z. B. Timeouts durch Probleme mit dem Netzwerk oder sich ändernde Stammdaten im Testsystem. Durch Verbesserungen wie ein separates WLAN, in dem nur die Testgeräte hängen oder interne „Testdatentools“, über welche Stammdaten mit bestimmten Eigenschaften angefragt werden können, versuchen wir diese Risiken zu minimieren. Siehe auch Bereitstellung von Testdaten einmal smart.

Sollte ein Test fehlschlagen, welcher mit dem Mockserver erfolgreich ist, ist eine tiefergreifende Analyse nötig. Ergebnis kann sein, dass der Test aus irgendeinem Grund „flaky“ (unzuverlässig) ist oder der Aufruf einer REST-Schnittstelle zu einem Fehler führt. Für die Ursachenforschung und -behebung wird in einem solchen Fall häufig das Team eingebunden, welches die REST-Schnittstelle zur Verfügung stellt.

Die Königsdisziplin: Webviews testen

Richtig spannend wird es dann, wenn wir den nativen Teil der App verlassen. Die gegen den Mockserver ausgeführten Tests werden hier abgebrochen, da die Funktionalität der Webview kaum zu simulieren ist. Eine Webview ist aus Sicht der App gewissermaßen eine Blackbox. Wir können zwar mit ihr interagieren, haben aber keinen Einfluss auf ihr Verhalten. Auch bekommt die App Änderungen in einer Webview nicht ohne Weiteres mit, d. h. der Weg aus der Webview zurück in die App ist eine zusätzliche Schwachstelle. Erschwerend kommt hinzu, dass die Webviews, wie die REST-Schnittstellen, von anderen Teams entwickelt und betreut werden.

Austausch von Daten zwischen dem nativen Teil und der Webview

Wechseln wir also vom nativen Teil der App in eine Webview, übergibt die App gewissermaßen das Steuer – mittels automatisierten Tests stellen wir sicher, dass hierbei keine Fehler passieren.

Daten werden an Webview übergeben

So wird zum Beispiel der OAuth-Token zur Kund*innen-Authentifizierung oder eine Liste von Artikeln beim Aufruf des Warenkorbs an die Webview übergeben. Mittels UI-Test überprüfen wir dann, ob beim Wechsel in den Warenkorb die zuvor hineingelegten Artikel angezeigt werden. Die Darstellung dieser Artikel sowie die verschiedenen Interaktionen auf der Warenkorb-Webview selbst wiederum werden vom bereitstellenden cross-funktionalen Team getestet.

Um durch die Webview zu navigieren ist es wichtig, dass es eine teamübergreifende Absprache darüber gibt, wie bestimmte (HTML-)Elemente identifiziert werden. Die Vergangenheit hat gezeigt, dass sich das Label eines Buttons häufiger ändert, beispielsweise von „weiter“ auf „zur Kasse“, es aber immer einen Button mit gleichbleibender Funktionalität gibt. Deshalb verwenden wir eindeutige Selektoren wie „interaction“ oder das eigens für das Testen angelegte „data-test“-Attribut. Teilweise müssen wir auch auf „id“ zurückgreifen,

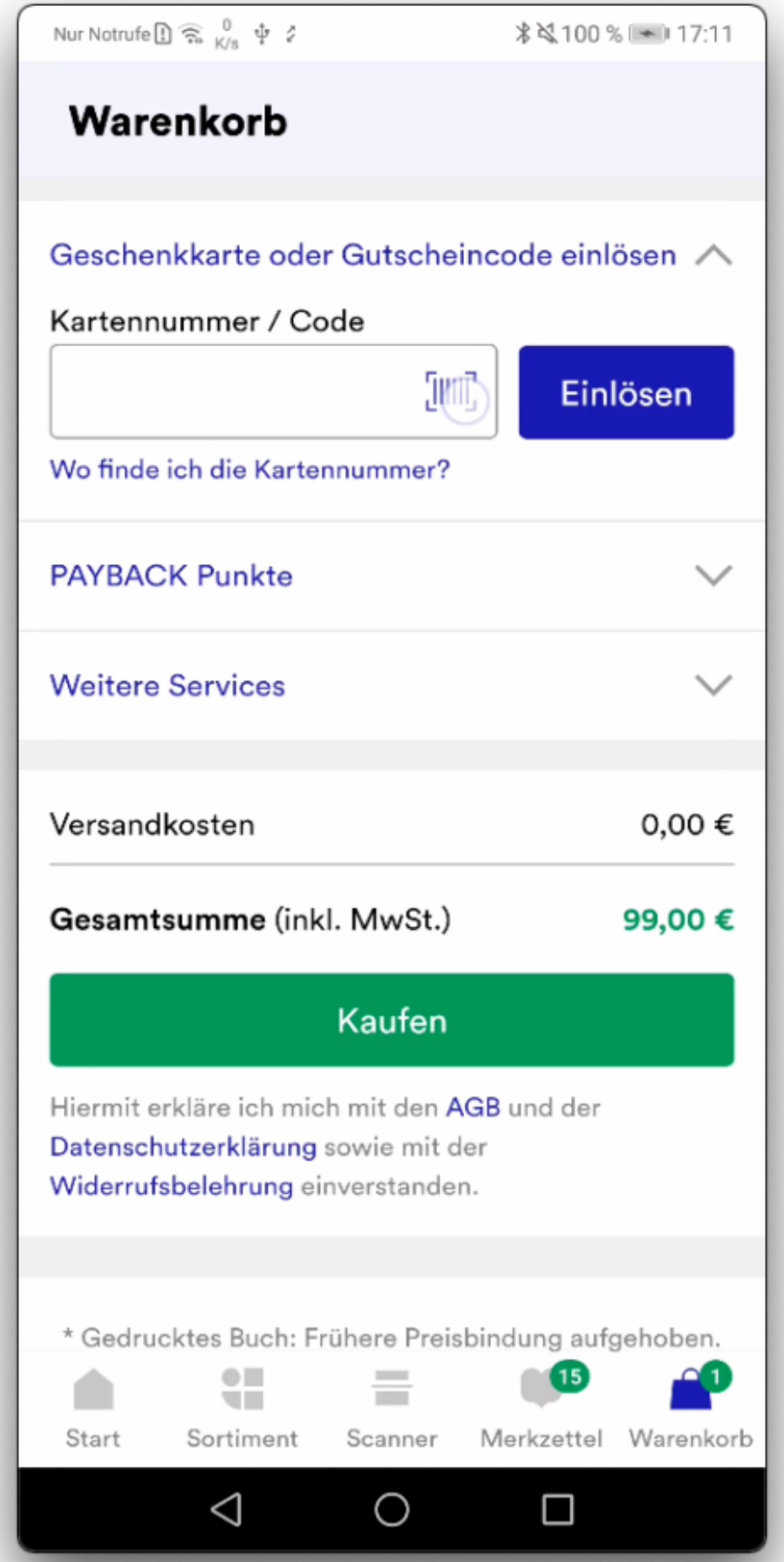
Stichwort „lebendes Objekt“. Auch haben sich einige Drittanbieter noch nicht bereit erklärt, durchgängig den Thalia-Standard zu verwenden. Schäme dich, großer, sehr bekannter Zahlungsdienstleister! Zum Vorteil von separaten HTML-Attributen: Robust locator strategy: custom attributes for test automation (Medium).

Über diese Selektoren hangelt sich der UI-Test über die Webview, füllt ein Kontaktformular aus oder klickt auf Buttons. Von all dem bekommt der native Teil der App allerdings nichts mit, d. h. alle vom UI-Test simulierten Kund*innen-Interaktionen spielen sich in besagter Blackbox ab.

Ist der*die Kund*in unangemeldet auf den Warenkorb navigiert und hat sich auf dem Weg in die Bestellübersicht angemeldet? Diese Information erhält der native Teil der App nur dann, wenn die Webview ihn darüber informiert. Dies geschieht über sogenannte Callbacks. Sie sind der einzige Weg aus einer Webview zurück in den nativen Teil der App, wenn nicht der gesamte Vorgang abgebrochen und die Webview selbst verlassen wird. Weiterführender Artikel zu diesem Thema: WebView- Love it or hate it, but you can’t ignore it (Medium).

Teamwork beim Scannen von Geschenkkarten

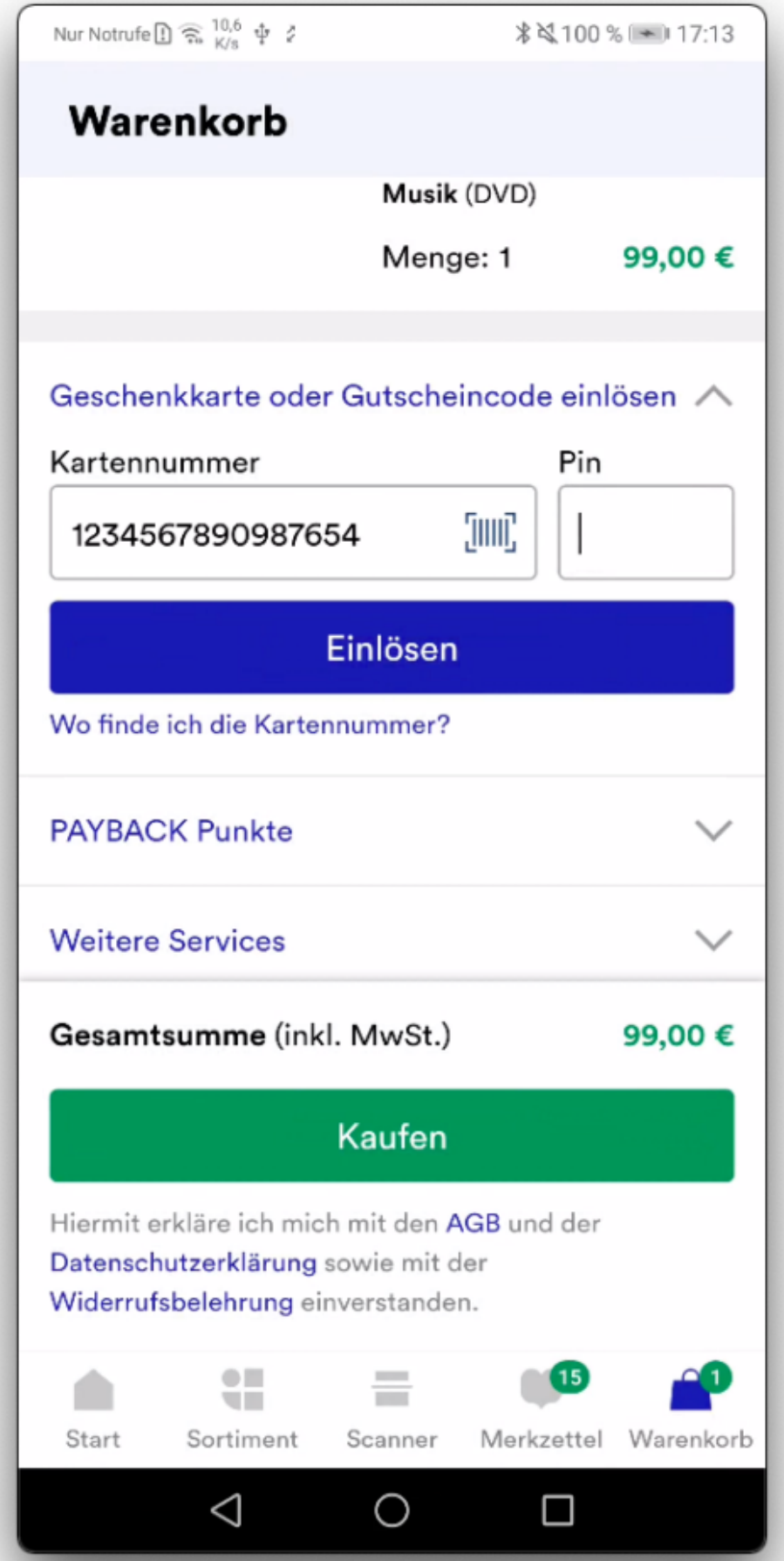
Ein anderes Beispiel für einen Callback ist der Geschenkkarten-Scanner. Dieser ist Teil der Bestellübersicht und bietet Kund*innen die Möglichkeit, den Barcode auf einer Thalia-Geschenkkarte einzuscannen, anstatt die vielen Zeichen selbst einzugeben.


Nur Notrufe
100 %
17:11
Warenkorb
Geschenkkarte oder Gutscheincode einlösen
Kartennummer / Code
Einlösen
Wo finde ich die Kartennummer?
PAYBACK Punkte
Weitere Services
Versandkosten
0,00 €
Gesamtsumme (inkl. MwSt.)
99,00 €
Kaufen
Hiermit erkläre ich mich mit den AGB und der Datenschutzerklärung sowie mit der Widerrufsbelehrung einverstanden.
* Gedrucktes Buch: Frühere Preisbindung aufgehoben.
15
1
Start
Sortiment
Scanner
Merkzettel
Warenkorb

Webview: Klick auf „Scannen“


Nur Notrufe
100 %
17:24
VB, .NET, Microsoft® ASP.NET, ASP, PHP, C/C
on (Nutzungsbedingungen) und erzeugen
arcodes, ISBN, etc.
1234567890987654
Barcode der Geschenkkarte im Rahmen platzieren.

Nativer Scanner: Geschenkkarte einscannen

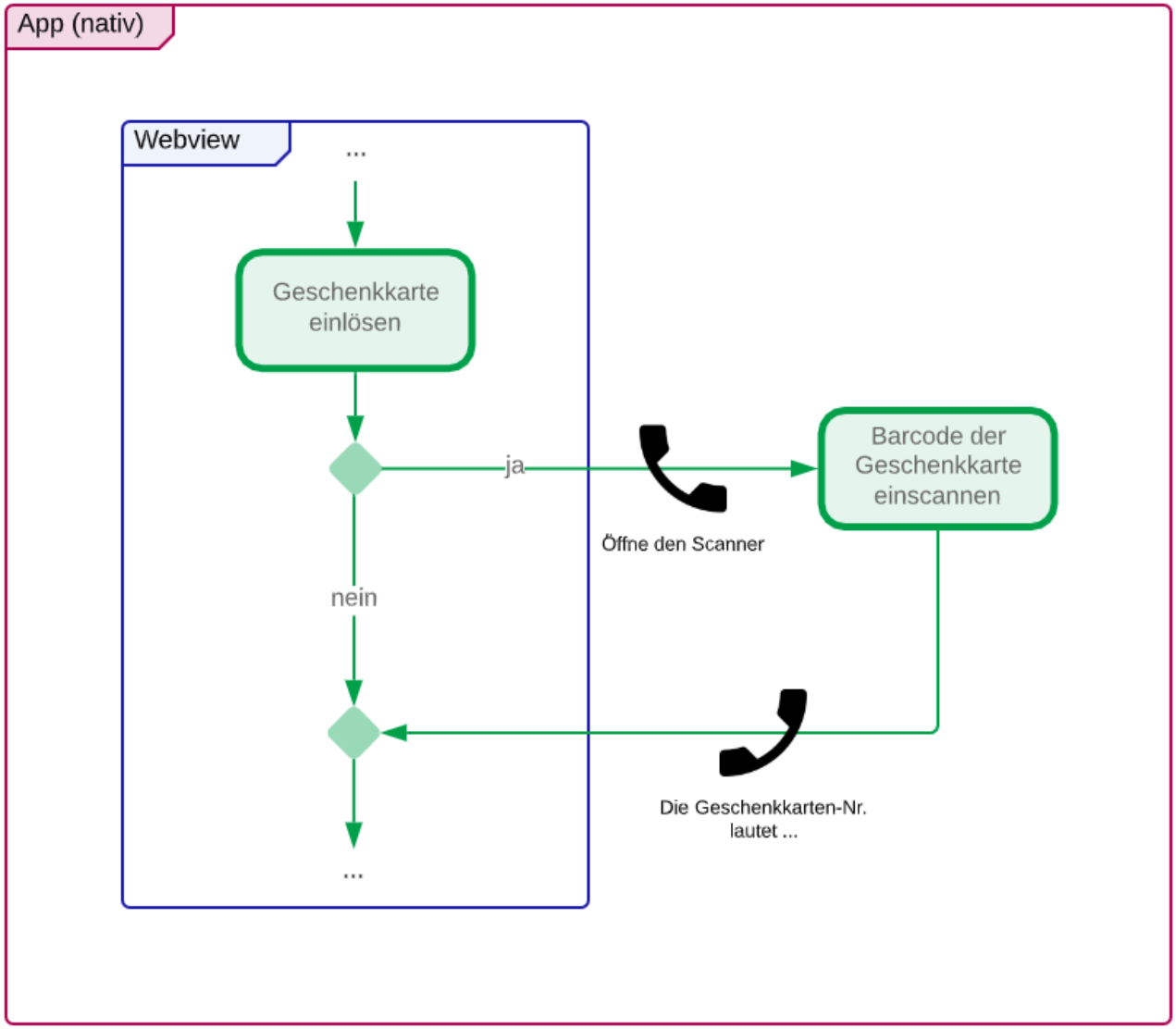

Nur Notrufe
10,6 K/s
100 %
17:13
Warenkorb
Musik (DVD)
Menge: 1
99,00 €
Geschenkkarte oder Gutscheincode einlösen
Kartennummer
Pin
1234567890987654
Einlösen
Wo finde ich die Kartennummer?
PAYBACK Punkte
Weitere Services
Gesamtsumme (inkl. MwSt.)
99,00 €
Kaufen
Hiermit erkläre ich mich mit den AGB und der Datenschutzerklärung sowie mit der Widerrufsbelehrung einverstanden.
15
1
Start
Sortiment
Scanner
Merkzettel
Warenkorb

Webview: Anzeige der gescannten Geschenkkarte

Bei dem Scanner handelt es sich um einen nativen Teil der App, welcher mittels Callback geöffnet wird, und der – wiederum mit einem Callback – die erkannte Zeichenfolge an den Webview übergibt. Hier gibt es also gleich mehrere mögliche Fehlerquellen:

1. Webview: Wird der Callback zum Öffnen des nativen Scanners (korrekt) gefeuert?
2. App: „Versteht" die App den Callback, also weiß sie, dass der Geschenkkarten-Scanner geöffnet werden muss?
3. App: Kann der Barcode vom nativen Scanner eingelesen werden?
4. App: Wird der Callback zur Rückgabe der erkannten Zeichenfolge (korrekt) gefeuert?
5. Webview: „Versteht" die Webview den Callback und schreibt die Zeichenfolge in das richtige Feld?

Während die Verantwortlichkeit für die Punkte 1 und 5 bei dem cross-funktionalen Team liegt, welches die Bestellübersicht betreut, liegt sie für Punkte 2 bis 4 bei der App.



Hier ist schön zu sehen, was ein UI-Test in der App alles implizit leistet. Denn auch wenn die Fragen alle isoliert (über bspw. Unit-Tests) getestet und mit „Ja“ beantwortet werden können, haben wir erst mit dem integrativen UI-Test eine Aussage darüber, ob das Zusammenspiel funktioniert. Nur ein in der App ausgeführter Test kann Sicherheit bringen und die fünf Prozessschritte zuverlässig zusammenstöpseln – ob automatisiert durch einen UI-Test oder manuell ausgeführt auf einem der zahlreichen mobilen Testgeräte, die über alle Teams verteilt sind. Dass jedes Team Zugriff auf ein Testgerät hat, ist vor allem wichtig und wertvoll, wenn an Komponenten gearbeitet wird. Die korrekte Funktionsweise, auch in der App, muss von allen Beteiligten kurzfristig sicherzustellen sein – Stichwort CI/CD der cross-funktionalen Teams.

Zusammenspiel der Webviews verschiedener Teams

Neben den Übergängen zwischen dem nativen Teil der App und den Webviews gibt es auch innerhalb der Webviews Kontextwechsel, welche teilweise bedeuten, dass die Domäne eines cross-funktionalen Teams verlassen und die eines anderen betreten wird. Navigieren nicht angemeldete Kund*innen z. B. vom Warenkorb auf die Bestellübersicht (beides Team A), wird zunächst auf die Anmelde-Seite weitergeleitet (Team B), um sich zu authentifizieren.

Erst danach kann er*sie mit der Wahl seiner*ihrer Versandoption und Zahlungsart in der Bestellübersicht fortfahren (zurück zu Team A). Hier gilt ähnliches wie beim Beispiel des Geschenkkarten-Scanners: Der durch die App navigierende UI-Test überprüft, ob alles wie gewünscht ineinandergreift.

Da diese Übergänge aber nicht nur in der App, sondern auch im Webshop existieren, gibt es ein weiteres, noch größeres Sicherheitsnetz, welches vom Team „zentrale QA“ über mehrere Browser für die Webvariante gespannt wird. Einen Eindruck über die Arbeit der zentralen QA gibt der Artikel True Grid oder wie wir unsere Testausführung durch Parallelisierung 6x schneller machten.

Die Herausforderungen des (App-)Testing bei Thalia

Das Testen in der App bringt viele Herausforderungen mit sich. Vor allem Abhängigkeiten zu anderen Teams und das Testen der unterschiedlichen integrativen Szenarien wie Verwendung von REST-Schnittstellen sowie die Einbindung der Webviews sind nicht nur technologisch, sondern auch organisatorisch anspruchsvoll. Deshalb schätze ich die Teamstruktur und -kultur bei Thalia sehr, bei der ich mit allen Kolleg*innen über Teststrategie oder ganz konkrete Probleme/Fehlerfälle sprechen kann. Denn für jedes Feature, das überarbeitet oder neu entwickelt wird, muss mit allen Beteiligten auch immer ein angemessenes Maß an Tests gefunden werden.

Um zu den Fragen vom Anfang zurückzukommen: Was liegt in wessen Verantwortung, wo ist Redundanz notwendig, überflüssig, wünschenswert? Da

das App-Testing so aufwendig und kompliziert ist, kann man nur sagen: Es gibt keine allgemeingültige Formel, kein Schema F, nur die Erfahrungen, die wir inzwischen gesammelt haben und an einigen Stellen erneut anwenden können.