Vergleich zwischen React Native und Progressive Web Apps für die Verwendung auf mobilen Endgeräten

Florian Maximilian Dörr
florian.maximilian.doerr@mni.thm.de

ABSTRACT

To target as many mobile users as possible, you need two apps: One for iOS and one for Android. But the development of two separate apps means double the effort. Another approach would be to develop a hybrid app which is able to run on both iOS and Android. But how performant are hybrid apps, especially in comparison to native apps? This paper compares React Native, Ionic and native apps by running automated tests. The test results are then taken in order to draw conclusions about how the frameworks performance compare.

Inhaltsverzeichnis

1 EINLEITUNG

Die Smartphone-Landschaft ist in zwei Betriebssystem-Lager fragmentiert: Googles Android und Apples iOS. Android bleibt auch im Jahr 2022 weltweit an der Spitze der mobilen Betriebssysteme mit 71,52% Marktanteil und iOS mit immerhin 27,83%. [1] Als Unternehmen hat man nun die Qual der Wahl: Fokussiert man sich auf Android oder iOS oder versucht man zwei separate Apps zu entwickeln, um beide Plattformen bedienen zu können?

Eine weitere Möglichkeit wäre es, eine hybride App zu entwickeln, welche auf beiden Plattformen lauffähig ist. Doch welche Kompromisse muss man hierbei eingehen?
Häufig wird damit argumentiert, dass hybride Ansätze eine unzureichende Performanz aufweisen.
Ob das naive Urteil über die Performanz von hybriden Ansätzen tatsächlich stimmt, wird in diese Ausarbeitung anhand von React Native im Vergleich zu einer progressiven Web-App unter Zuhilfenahme des Ionic Frameworks untersucht.

Diese Ausarbeitung unterstützt dabei zugleich meine aktuelle Aufgabe meiner Arbeitsstätte *Cursor Software AG* aus Gießen. Dabei gilt es die Evaluation einer Neuentwicklung mit hybriden Ansätzen für die aktuellen, nativen Apps durchzuführen. Da im Unternehmen bereits das React Framework für die Infoboard 2.0 Technologie[1] eingesetzt wird, liegt es im Interesse des Unternehmens, auch einen möglichen hybriden-App-Ansatz mit dem React-Framework zu gestalten.

Da sowohl React Native als auch Ionic die Möglichkeit bieten, React zu verwenden, fiel die Vorauswahl auf diese beiden Frameworks. Zusätzlich ist so ein möglichst fairer Vergleich möglich: So kann der Performanceaspekt vollends auf die verwendete Render-Engine des Frameworks reduziert werden. Für Testanwendungen kann (fast) identischer Code verwendet werden.

Zunächst werden die verwendeten Frameworks React, React Native und Ionic vorgestellt. Dabei werden vor allem Aspekte hervorgehoben, welche aus Sicht des Autors besonders relevant für die Performance der jeweiligen Frameworks sind. Im Folgenden wird ein Testaufbau definiert, der die Performance einer Listenansicht quantifiziert. Anschließend werden Beispiel-Apps mit einer Listenansicht in React Native, Ionic sowie natives Android und iOS anhand der vorher definierten Metriken verglichen und ausgewertet. Im Anschluss wird ein weiterer Testaufbau definiert, der es ermöglicht, das Startverhalten der Frameworks und der nativen Ansätze zu vergleichen und auszuwerten. Zuletzt werden die Ergebnisse zusammengetragen, die Aussagekraft der gewonnenen Daten bewertet und ein Ausblick geliefert.

[1] Kleine, eigenständige Info-Karten, die in das hauseigene CRM-System *Cursor-CRM* integriert werden können.

2 VORSTELLUNG DER FRAMEWORKS

Im Folgenden werden die zu vergleichenden Hybrid-Frameworks React Native und Ionic sowie das Frontend-Framework React vorgestellt. Die Vorstellungen sind hierbei bewusst recht oberflächlich gehalten. Grund hierfür ist, dass alle Frameworks extrem umfangreich sind und eine genauere Betrachtung sich nicht für den begrenzten Umfang dieser Ausarbeitung eignet. Sehr wohl werden aber insbesondere Aspekte und Designentscheidungen der Frameworks beleuchtet, die – aus Sicht des Autors – einen Einfluss auf die allgemeine Performance haben, auf die später im Performancevergleich zurückgegriffen werden kann.

2.1 React

React ist ein quelloffenes JavaScript-Paket für die Entwicklung von Webanwendungen. React wird dabei federführend von Meta[2] entwickelt und für Metas Webanwendungen *Facebook* sowie *Instagram* verwendet.

2.1.1 JSX. Im Gegensatz zu klassischen Webseiten verwendet React kein HTML, sondern JSX[3]. JSX ist dabei syntaktisch sehr ähnlich zu HTML sowie XML und lässt sich durch JavaScript Ausdrücke anreichern. Eine Reactanwendung wird durch die Verwendung von *Komponenten* aufgebaut. Eine React-*Komponente* besteht dabei aus einem Teil, der die Logik der *Komponente* abbildet und einen Teil, der die Darstellung dieser *Komponente* übernimmt. Besonders hervorzuheben ist bei React, dass eine Zustandsänderung einer *Komponente* immer in der Aktualisierung der Darstellung resultiert. Ändert sich zum Beispiel der Wert einer Variable einer *Komponente*, so *reagiert* die Darstellung auf eben diese Änderung. Folglich wird sich am Programmierparadigma der reaktiven Programmierung bedient, wodurch sich auch der Name *React* ableiten lässt. Bei der Erstellung solcher Komponenten gilt es zu beachten, dass ein bestimmter Zustand auch immer dieselbe Darstellung liefern muss. So bildet eine Komponente eine Schablone, in die der Zustand eingebettet wird. Ein großer Vorteil dieser Eigenschaft ist die einfache Testbarkeit: Einzelne Komponenten können mit verschiedenen Zuständen aufgerufen werden und müssen immer ein definiertes Ergebnis liefern. Einzelne Komponenten werden anschließend so ineinander verschachtelt, bis die gewünschte Gesamtanwendung aufgebaut ist. [2]

2.1.2 JSX im Browser. Da klassische Browser nur HTML, CSS und JavaScript verarbeiten können, muss das JSX übersetzt werden. Diese Umwandlung kann entweder auf Seite des Clients – also z. B. im Webbrowser an sich – oder aber auch auf dem Server passieren. Klassischerweise[4] erfolgt die Umwandlung im Client. Bei jeder Zustandsänderung **einer** Komponente muss auch eine Aktualisierung des DOMs[5] durchgeführt werden, damit auch dort die Änderungen der Darstellung durchschlagen. React geht dabei jedoch hocheffizient vor und aktualisiert nur DOM-Elemente, welche auch notwendigerweise angepasst werden müssen. [2, 3]

2.1.3 DOM vs. VDOM. Ein weiterer Vorteil von React ist, dass das Framework eine interne Darstellung (*virtual DOM*, bzw. VDOM) der Komponenten orchestriert und erst anhand dieser Darstellung Veränderungen im DOM vornimmt. Dadurch ist es dem Framework möglich, andere Darstellungsformen als das DOM zu verwenden, wie z. B. native Android und iOS Anwendungen. Dies wird im folgenden anhand des React Native Frameworks genauer beschrieben. [2, 4]

2.2 React Native

React Native erlaubt das Erstellen von hybriden Apps in Java- bzw. TypeScript. Für das Deklarieren von UI-Elementen wird hierbei das populäre, reaktive Framework React (vgl. Abschnitt 2.1) von Meta verwendet.

Der Java- bzw. TypeScript Code wird anschließend – nicht wie gewöhnlich – in einer Browserumgebung interpretiert und in DOM-Nodes, sondern direkt in native Komponenten der verschiedenen Plattformen umgewandelt. So wird zum Beispiel eine React Komponente wie `<Text />` in ein Android `<TextView>`, unter iOS in ein `<UITextView>` und im Web in ein enfaches `<p>`-Tag umgewandelt. Dies hat zur Folge, dass die Komponenten immer zu "echten" nativen Komponenten respektive deren Plattformen korrelieren. [5]

2.2.1 Host View Tree. Diese Komponenten werden in dem *Host View Tree* angeordnet und beinhalten je nach Plattform iOS oder Android spezifische UI-Komponenten. [6] Dieser *Host View Tree* wird dabei von der plattformunabhängigen Layout-Engine *Yoga* generiert. *Yoga* ermöglicht es, CSS[6]-Flexbox-Layouts zu verwenden, welche klassischerweise nur von Browsern interpretiert werden können. Da Yoga in C implementiert wurde, ist es sowohl unter Android als auch iOS lauffähig. [7]

2.2.2 Fabric. Mit Version `0.68.0` wurde eine neue Architektur in React Native implementiert. [8] Als größte Änderung setzt diese auf eine neue Render-Engine namens *Fabric*. Laut der Dokumentation bringt diese neben besserer Wartbarkeit und Codequalität auch Performanceverbesserungen mit sich. [9]

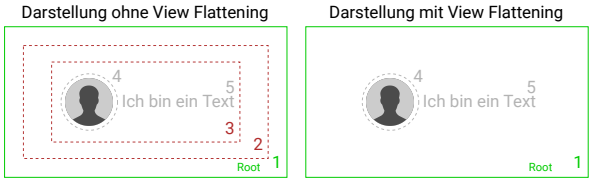
2.2.3 View Flattening. Eine dieser Verbesserung ist dabei ein plattformübergreifendes *View Flattening*. Bei der Entwicklung einer React-Komponente werden häufig Layout-Komponenten ineinander geschachtelt (siehe Listing 1). Diese zeigen dabei keine konkreten Informationen an, sondern definieren lediglich das Aussehen der Komponente (siehe Abbildung 1). Das Resultat vieler, ineinander verschachtelten Layout-Komponenten ist ein tiefer *Host View Tree* an Komponenten. Das *View Flattening* komprimiert hierbei aufeinander folgende Layout-Komponenten, indem es das Styling der einzelnen Layout-Komponenten in eine Layout-Komponente zusammenfasst. Folglich muss der Renderer in Teilen des *Host View Tree* nicht mehr viele Layout-Komponenten darstellen, sondern nur noch eine (siehe Abbildung 2). [10]

[2] Ehemals Facebook
[3] JSX = JavaScript XML
[4] Jedenfalls so lange, bis die Anwendungen zu komplex werden
[5] DOM = Document Object Model, beschreibt die Darstellung von HTML-Tags im Browser
[6] CSS = Cascading Style Sheets, legt z. B. Farben, Abstände, Schriftarten, … einer Webseite fest

Listing 1 Beispiel-Komponente mit mehreren Layout-Komponenten [10]

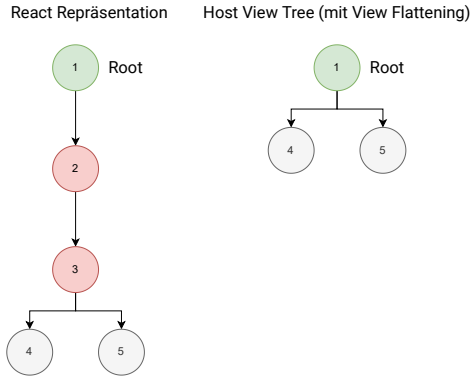
```
function ListViewItem() {
  return (
    <View id="1">
      <View id="2" style={{ display: "flex" }}>
        <View id="3" style={{ gap: 8}}>
          <Image id="4" />
          <Text id="5">Ich bin ein Text</Text>
        </View>
      </View>
    </View>
  );
}
```

Abbildung 1: Schematische Darstellung des View Flattenings [10]



Die Ziffern 1 – 5 sind identisch zu den IDs aus Listing 1.
Gestrichelte Linien sind Layout-Komponenten, die der Nutzer nicht ohne weiteres sieht.

Abbildung 2: Schematische Darstellung des View Flattenings [10]



Die Ziffern 1 – 5 sind identisch zu den IDs aus Listing 1.
Linker Baum: interne React Repräsentation (vgl. Abschnitt 2.1.3)
Rechter Baum: Host View Tree (also Darstellung auf Gerät, vgl. Abschnitt 2.2.1)

2.2.4 JavaScript Interfaces (JSI). Benötigt eine native Komponente oder ein natives Plugin Daten von React – z.B. für das Anzeigen eines Textes – so werden unter Verwendung der alten Architektur diese Daten zunächst in die JavaScript Object Notation (JSON) serialisiert[7]. Das serialisierte JSON-Objekt wird anschließend an die Komponente oder das Plugin geschickt. Um in der Komponente nun an die Daten des JSON-Objektes zu gelangen, muss dieses zunächst wieder deserialisiert werden. Diese beiden Schritte benötigen evident Rechenleistung.

Aus diesem Grund greift Fabric direkt auf die Werte über *JavaScript Interfaces* (JSI) zu und spart somit die beiden Umwandlungsschritte. [9, 11]

2.2.5 Lazy Loading. Bei der Verwendung von Fabric werden native Plugins erst dann geladen, wenn sie auch tatsächlich benötigt werden. Unter der Verwendung der alten Architektur werden diese alle beim Starten der App geladen, wodurch der Start womöglich verzögert wird. Durch das Verwenden des sogenannten *Lazy Loading*s verspricht Fabric schnellere Startzeiten. [9, 11]

2.3 PWAs und Ionic

Um die Funktionsweise von Ionic verstehen zu können, ist es notwendig die Eigenschaften einer progressiven Web-App (PWA) zu kennen. PWAs sind zunächst normale Webanwendungen – also Webseiten, welche gewisse Charakteristika aufweisen.

2.3.1 Progressive. Zunächst sind PWAs *Progressive*, das Ausschließen älterer Browser sollte vermieden werden; so viele Nutzer wie möglich sollten in der Lage sein, die PWA nutzen zu können. Nutzer mit aktuelleren Browsern könnten unter Umständen aber einen erweiterten Funktionsumfang genießen.

2.3.2 Responsive. Eine extrem wichtige Eigenschaft ist die *Responsiveness*. Jedes Gerät, dass die PWA verwendet hat potenziell andere Displaydimensionen. Es ist also praktisch unmöglich, die App für alle Displaygrößen manuell anzupassen. Aus diesem Grund muss die Anwendung in der Lage sein, sich an einen der App zur Verfügung stehenden Rahmen automatisch anzupassen.

2.3.3 Connectivity und Freshness. Mobile Endgeräte sind stets in Bewegung und schlechte Konnektivität oder gar der sämtliche Verlust der Netzwerkverbindung sind keine Seltenheit. Folglich müssen PWAs auch offlinefähig sein. Dennoch müssen die Daten – ungeachtet der Offlinefähigkeit – stets aktuell gehalten werden.

2.3.4 App-Like und Discoverable. Eine PWA sollte möglichst das Look-And-Feel der Plattform widerspiegeln. Diese Eigenschaft hilft zusätzlich dabei, eine PWA von einer klassischen Webseite zu unterscheiden.

2.3.5 Safe. Eine PWA muss *sicher* sein. Konkret bedeutet dies, dass die Verbindung zum Backend[8] verschlüsselt sein muss[9].

2.3.6 Re-engageable. Eine PWA sollte mittels Push-Benachrichtigungen Mitteilungen an den Nutzer senden können.

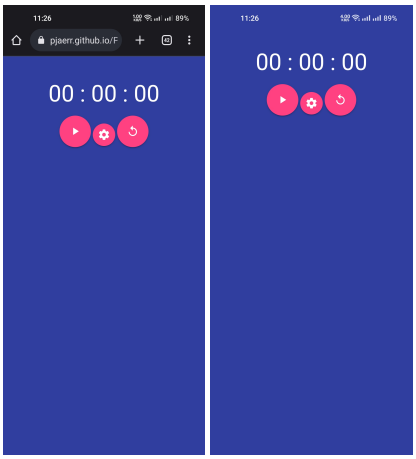
2.3.7 Installable. Die PWA muss installierbar sein. Unter iOS oder Android erscheint z. B. eine installierte PWA als Icon auf dem Homescreen und ist somit auf den ersten Blick nicht von einer nativen App unterscheidbar. Öffnet man diese PWA nun, wird

[7] Vereinfacht ausgedrückt, werden Objekte in eine Zeichenkette transformiert, die den Zustand des Objekts widerspiegelt
[8] Serverlogik, z. B.: Login, Abfragen von Daten; läuft nicht innerhalb der PWA
[9] z. B. über HTTPS

der systemeigene Webbrowser verwendet, dabei werden jedoch sämtliche Interaktionselemente des Browsers versteckt, sodass nur die darinliegende Webseite zu sehen ist (vgl. Abbildung 3).

2.3.8 Linkable. Eine PWA sollte über einen Link aufrufbar sein. Ebenfalls sollte es möglich sein, über *Deep Links* direkt auf bestimmte Datensätze zugreifen zu können. Dies ermöglicht z. B. das Teilen eines bestimmten Datensatzes über einen *Deep Link* mit anderen Nutzern. [12]

Abbildung 3: Beispiel-PWA

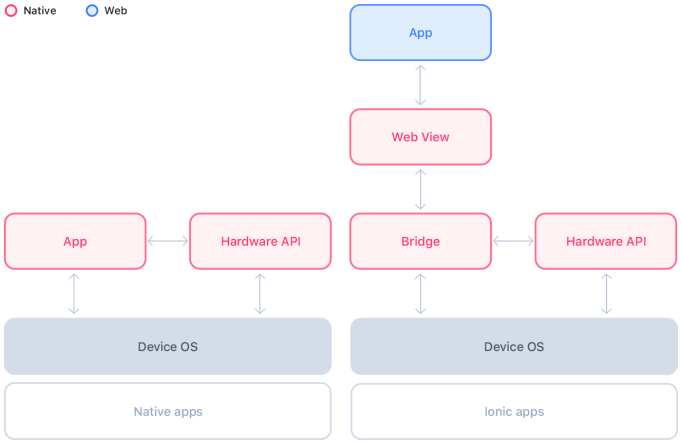


Links: PWA wird im normalen Webbrowser aufgerufen (Bedienelemente alle vorhanden)
Rechts: Installierte PWA wird aufgerufen (Bedienelemente versteckt, sieht aus wie eine App)
siehe https://pjaerr.github.io/PWA-Timer

Ionic ist eine Open Source JavaScript-Bibliothek, die das Entwickeln von PWAs konform der Charakteristika aus Abschnitt 2.3 für iOS, Android und den Desktop[10] ermöglicht. Ionic erweitert dabei eine PWA mittels *Cordova* oder *Capacitor* Plugins um native Funktionalitäten. Diese Plugins implementieren dabei Web-, iOS und/oder Android-spezifische Funktionalitäten, die durch klassische PWAs nicht umsetzbar sind. [12] So ist es zum Beispiel möglich, direkt über ein solches Plugin SMS zu versenden. [13] Ionic verwaltet hierbei neben der Schnittstelle zu Plugins ebenfalls noch einen in einer nativen App eingebetteten Webbrowser[11] (vgl. *Web View* in Abbildung 4). Wird eine native Funktionalität bereits von der HTML5 API unterstützt, wird diese Anfrage direkt an das *Device OS* oder über ein Plugin – die *Hardware API* – an das *Device OS* weitergeleitet (vgl. Abbildung 4; die *Bridge* entscheidet, welcher der beiden Wege genommen werden muss). [14] Hervorzuheben ist, dass Ionic keine Vorgaben an das verwendete Frontend-Framework macht. Somit ist es möglich, eine beliebige PWA innerhalb der *Web-View* anzuzeigen. Dennoch wird seitens Ionic ab Version 4 offiziell Angular, React (vgl. Abschnitt 2.1) und Vue.JS unterstützt, jedoch ist es auch möglich, keines dieser Frameworks zu verwenden. [12]

[10] Unter Zuhilfenahme des *Electron*-Frameworks
[11] in dem Fall auch *Web View* genannt

Abbildung 4: Überblick der Architektur Ionics



Linker Teil: Native App, ohne Web View.

Rechter Teil: Ionic App, mit Web View.
Mit *App* ist hierbei die PWA gemeint, die innerhalb der Web View angezeigt wird.
[14]

3 PERFORMANCEVERGLEICH - LISTENANSICHTEN

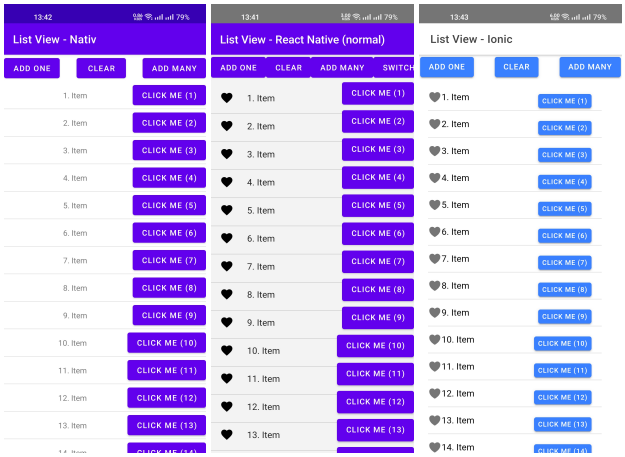
Im Folgenden wird die Performance der Frameworks React Native und Ionic (vgl. Abschnitt 2.2 bzw. Abschnitt 2.3) sowie einer nativen Android bzw. iOS App verglichen. Eine Anforderung ist hierbei, dass die gewählten Tests möglichst realitätsnah und wenigst möglich instrumentiert sind: Die für den Nutzer spür-/sichtbare Performance soll hierbei getestet werden und nicht, wie schnell das jeweilige Framework x-Nachkommastellen von π berechnen kann. Ziel sind also Usability Benchmarks und weniger synthetische Benchmarks.

Die erstellten Testanwendungen und die Testapplikationen sind unter https://git.thm.de/fmdr54/list-view-performance zu finden.

3.1 Testaufbau

Listenansichten kommen in vielen Apps – wenn auch in abgewandelten Formen – vor. Als kritische Aspekte einer solchen Listenansicht werden hierbei das Scrollverhalten sowie die Fähigkeit um dynamische Erweiterung festgelegt.
Aufgrund der genannten Aspekte wurde folgender Versuchsaufbau für die Apps gewählt: Eine Liste startet mit fünf Elementen. Jedes Element besitzt dabei ein Bild, einen Text und einen Button. Jedes Listenelement ist klickbar. Zusätzlich existieren drei Buttons oberhalb der Liste, die das Hinzufügen eines Listenelements, das Löschen aller Elemente der Liste und das Hinzufügen von 50 Listenelementen ermöglichen (vgl. Abbildung 5).

Abbildung 5: Übersicht aller drei Testanwendungen unter Android.



Von links nach rechts: Nativ, React Native, Ionic.

3.1.1 Testablauf. Der Test ist dabei folgendermaßen konzeptioniert:

(1) Starten der App.
(2) Finden der drei Buttons oberhalb der Liste.
(3) Finden und klicken des ersten Elements der Liste.
(4) Fünf einzelne Elemente über den entsprechenden Button hinzufügen.
(5) Zehn Mal 50 Elemente über den entsprechenden Button hinzufügen.
(6) Schnellstmöglich an das Ende der Liste (mit nun 510 Elementen) scrollen.
(7) Das 500. Element der Liste anklicken.

3.1.2 Bewertungskriterien. Als Bewertungskriterien wird sich im Folgenden auf verworfene Bilder, der CPU sowie Arbeitsspeicher Belastung festgelegt. Die CPU-Auslastung wie auch die Arbeitsspeicherauslastung spiegeln hierbei gut die Gesamtlast des Systems wider und erlauben einen einfachen Vergleich: Ist die CPU-Auslastung einer App höher, so muss diese für die gleichen Aufgabe mehr Arbeit verrichten. Ist die Arbeitsspeicherauslastung höher, muss die App für die gleiche Aufgabe mehr Daten zwischenspeichern.

3.1.3 Verworfene Bilder. Die meisten Smartphones sind in der Lage 60, gar 120 oder noch mehr Bilder pro Sekunde[12] anzuzeigen. Die Bildschirme der Smartphones besitzen folglich eine Bildwiederholrate von 60 bzw. 120Hz. Im Umkehrschluss bedeutet dies; um ein Bild bei einer Bildwiederholraten von 60Hz zu berechnen, bleiben nur $\frac{1\text{s}}{60\text{Hz}} = 0{,}01\overline{6}\text{s} = 1\overline{6}\text{ms}$ und bei 120Hz gerade mal $\frac{1\text{s}}{120\text{Hz}} = 0{,}008\overline{3}\text{s} = 8{,}\overline{3}\text{ms}$. Schafft es das System nicht, innerhalb der sogenannten *Deadline* von ≈ 17ms bzw. ≈ 8ms ein Bild zu berechnen, wird dieses Bild als *verworfenes Bild*[13] bezeichnet. Die Häufung solcher *skipped frame* erscheinen dem Benutzer als "ruckelndes" Bild. Die Android-Dokumentationen sprechen hierbei von *Janky Frames*. iOS nennt diese *Animation Hitches*. Beide stehen jedoch für das nicht rechtzeitige Berechnen von Bildern, weswegen sie im Folgenden einfachheitshalber synonym verwendet werden. Verworfene Bilder sind somit eine gute Möglichkeit, eine schlechte bzw. gute Benutzererfahrung zu quantifizieren. [15–17]

Um die bestmögliche Performance der Apps zu vergleichen, wurde stets der Production-Build der jeweiligen Frameworks verwendet. Insbesondere bei der Verwendung von React Native ist dies erforderlich. [18][14]

3.2 Android

Aufgrund der genannten Anforderungen (vgl. Abschnitt 3) wurde sich für das UI-Automator Framework entschieden. Dieses ermöglicht das Schreiben von automatisierten Oberflächentests für Android. Ein großer Vorteil dieses Tools ist es hierbei, dass Elemente anhand deren Anzeigetexte selektiert werden können. Konkret bedeutet dies, dass der Test in verschiedenen Apps Elemente mit dem gleichen Text selektieren kann, ohne deren interne IDs zu kennen. Das Vergeben von identischen IDs für Komponenten der unterschiedlichen Apps ist nicht grundsätzlich nicht trivial. Weiterhin simuliert UI-Automator echte Nutzereingaben und arbeitet nicht *headless*[15].[18]

Die Tests werden dabei auf zwei physischen Geräten (Oneplus 8T und Samsung Galaxy S8+) getestet (siehe Tabelle 1). Dabei werden die Tests 50 Mal ausgeführt und die Ergebnisse arithmetisch gemittelt, um eventuell auftretende Fehler zu minimieren. Auf allen Testgeräten wurden systemseitige Animationen deaktiviert. Dies wird empfohlen, da Animationen die Testausführung negativ beeinflussen können. [19]

Für React Native wurden zwei Tests durchgeführt. Dabei verwendet der erste Ansatz einen klassischen Ansatz, wohingegen der zweite eine Implementierung mittels *virtualized lists* umsetzt. Grund hierfür ist, dass dieser Ansatz explizit in der React Native Dokumentation erwähnt wird. *Virtualized lists* rendern – im Gegensatz zu dem klassischem Ansatz – nur den Teil der Liste, der auch auf dem Bildschirm des Nutzers angezeigt werden kann. Normalerweise werden alle Elemente gerendert, auch wenn dieses akuell außerhalb des Bildschirms liegt. Elemente werden in einer *Virtualized list* bei Bedarf – beim Scrollen – nachgeladen oder auch entladen. [20, 21]

3.2.1 Testergebnisse. Die im Folgenden dargestellten Ergebnisse beinhalten den prozentualen Anteil der verworfenen Bilder, die Arbeitsspeicherauslastung über den gesamten Test sowie die CPU-Auslastung aller zur Verfügung stehenden CPUs.

3.2.2 Datenquelle - verworfene Bilder. Der prozentuale Anteil der verworfenen Bilder wird mittels `adb shell dumpsys gfxinfo "$appId"` vom Testgerät abgerufen. `$appId` entspricht dabei der in Android hinterlegten `applicationId`[16]. Um die verworfenen Bilder aus der Ausgabe des Kommandos zu extrahieren, werden einige

[12] Auch als *Frames Per Second* oder kurz *FPS* bezeichnet
[13] oder *skipped frame*
[14] Der Production Build deaktiviert den Developer Modus automatisch
[15] *headless* = ohne sichtbare Oberfläche, nur theoretischeÄnzeige von Elementen
[16] Jede Android-App besitzt eine eindeutige ID. Z. B. `de.flodoerr.listperformance`

Tabelle 1: Technische Daten der Android-Testgeräte

Testgerät	**Oneplus 8T** [22]	**Samsung Galaxy S8+ (G955F)** [23]
Veröffentlichungsdatum	Oktober 2020	April 2017
Android Version	12	13
CPU	Qualcomm® Snapdragon™ 865 8-Kerne @ 2,84 GHz (max)	Qualcomm® Snapdragon™ 835 8-Kerne @ 2,45 GHz (max)
CPU-Architektur	ARM-64	ARM-64
GPU	Adreno 650	Mali-G71 MP20
Arbeitsspeicher	12 GB	4 GB
Bildwiederholrate (max.)	120Hz	60Hz

Pipes verwendet: `adb shell dumpsys gfxinfo "$appId"|grep -m 1 "Janky frames"|grep -Eio "\w+\.\w2"`.

3.2.3 Datenquelle - Arbeitsspeicherauslastung. Die Arbeitsspeichernutzung wird anhand `adb shell dumpsys meminfo "$appId"` ermittelt. Die Angabe ist dabei in MB. Um lediglich die Arbeitsspeichernutzung aus der Ausgabe zu extrahieren, wird das folgende Kommando verwendet: `adb shell dumpsys meminfo "$appId"|grep -m 1 TOTAL |awk -v col=3 'print $col'`.

3.2.4 Datenquelle - CPU-Auslastung. Die CPU-Auslastung wird durch `adb shell ps -p "$appPido %cpu` ermittelt. `$appPid` ist hierbei die PID[17], welche anhand `adb shell pidof "$appId"` ermittelt wird. Der Prozentwert der CPU kann dabei über 100% liegen, da die Auslastung aller CPUs akkumuliert wird.[18] Zur Extrahierung der Ausgabe werden ebenfalls wieder einige Kommandos mithilfe von Pipes verknüpft: `adb shell ps -p "$appPido %cpu |tail +2`

Die Ergebnistabellen sind stets aufsteigend nach prozentualem Anteil der verworfenen Bilder sortiert. In allen Bewertungsaspekten bedeuten kleinere Werte ein besseres Ergebnis.

Tabelle 2: Testergebnisse der Listenansicht des Testgerätes *OnePlus 8T* bei $120Hz$

Framework (n=50)	"Janky Frames" Ø	Arbeitsspeicher Ø (gesamt)	CPU-Auslastung Ø (gesamt)
Nativ	1,71 %	29,57 MB	10,21 %
Ionic	4,23 %	81,91 MB	10,56 %
React Native (virtualized list)	7,43 %	186,36 MB	103,19 %
React Native	11,70 %	232,80 MB	37,46 %

Tabelle 3: Testergebnisse der Listenansicht des Testgerätes *OnePlus 8T* bei $60Hz$

Framework (n=50)	"Janky Frames" Ø	Arbeitsspeicher Ø (gesamt)	CPU-Auslastung Ø (gesamt)
Nativ	3,03 %	33,28 MB	7,98 %
React Native (virtualized list)	3,50 %	175,67 MB	87,80 %
React Native	4,08 %	229,88 MB	31,25 %
Ionic	5,39 %	86,61 MB	10,42 %

Tabelle 4: Testergebnisse der Listenansicht des Testgerätes *Samsung Galaxy S8+*

Framework (n=50)	"Janky Frames"	Arbeitsspeicher (gesamt)	CPU-Auslastung (gesamt)
React Native (virtualized list)	9,03 %	206,21 MB	108,94 %
Ionic	36,57 %	101,30 MB	28,06 %
Nativ	41,83 %	36,31 MB	23,91 %
React Native	78,19 %	318,80 MB	75,09 %

[17] PID = Process Identifier, Jeder Prozess besitzt einen PID. Zwei unterschiedliche Prozesse können nicht die gleiche PID besitzen.
[18] CPU_1 ist zu 65% ausgelastet, CPU_2 ist zu 55% ausgelastet. Die Gesamtauslastung liegt somit bei 65% + 55% = 120%.

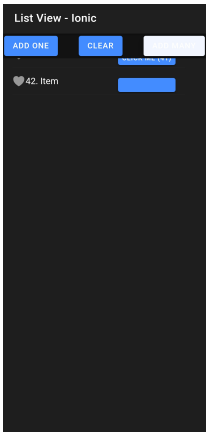
3.3 Auswertung: Native Android App

In beiden Tests auf dem *OnePlus 8T* (vgl. Tabelle 2 und Tabelle 3) liegt die native Implementation in allen Bewertungskriterien vorne. Besonders hervorzuheben ist, dass nicht nur weniger Bilder verworfen wurden, sondern zusätzlich die CPU- und Arbeitsspeicherauslastung im Vergleich zu React Native und Ionic geringer ist. Ausgehend von den festgelegten Bewertungskriterien (vgl. Abschnitt 3.1) ist die native Android-Implementation somit die beste in diesem Vergleich.

3.4 Auswertung: React Native und Ionic - Android

Leider sind die Anzahl der *Janky Frames* bei der Verwendung von Ionic nur bedingt aussagekräftig. Grund hierfür ist, dass Ionic eine PWA innerhalb einer WebView anzeigt (vgl. Abschnitt 2.3). Bei Tests der Listenansicht (vgl. Abschnitt 3.1) konnte häufig beobachtet werden, dass die WebView Schwierigkeiten hatte, die Listenelemente nachzuladen, obwohl das Scrollen an sich flüssig ablief (vgl. Abbildung 6). Das Fehlen der Listenelemente wirkt sich dabei jedoch nicht zwangsläufig in den verworfenen Bildern aus. Dennoch wird es möglich sein, *Janky Frames* in der Anwendung zu messen, sobald die Last der Webview zu stark ist.

Abbildung 6: Ionic schafft es beim Scrollen nicht, die Elemente rechtzeitig anzuzeigen



Bei Tests von React Native auf einem Gerät mit einer Bildwiederholrate von $120Hz$ ist aufgefallen, dass der Anteil an verworfenen Bildern häufig nahezu 50% betrug (vgl. Abbildung 7). Somit wurden

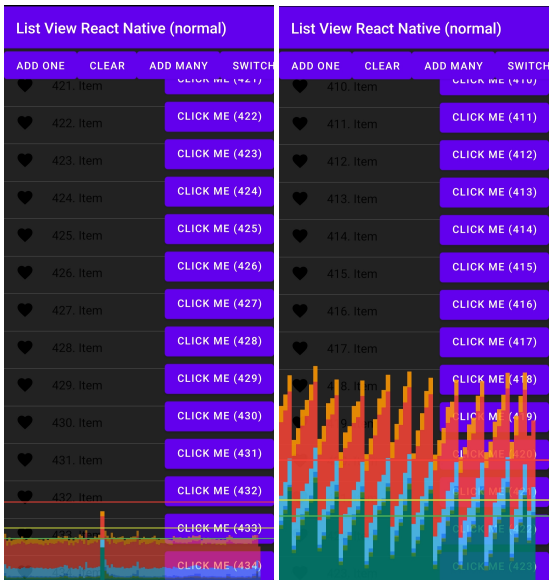
stets ungefähr nur 60*FPS* angezeigt. Nach einer kurzen Recherche zeigte sich, dass React Native von sich aus leider nur 60*FPS* unterstützt. [24] Aus diesem Grund wurde für React Native das Package *react-native-reanimated* verwendet, welches eine 120*Hz* Unterstützung nachliefert. [25] Da 120*Hz* dennoch nicht offiziell unterstützt werden, sollten die Ergebnisse mit 120*Hz* mit Vorsicht genossen werden.

Die Implementation unter React Native, welche *virtualized lists* verwendet, liegt unter Betrachtung der verworfenen Bilder stets vor dem React Native Ansatz mit der klassischen Listenansicht. Zusätzlich benötigt dieser Ansatz auch immer weniger Arbeitsspeicher. Bei der Verwendung von *virtualized lists* muss jedoch mit einer erhöhter CPU-Auslastung gerechnet werden. Diese liegt liegt bei allen Testgeräten über der CPU-Auslastung des klassischen Ansatzes (vgl. Tabellen 2, 3 und 4 *React Native (virtualized list)* im Vergleich zu *React Native*).

Im direkten Vergleich bei der Betrachtung der verworfenen Bilder zu Ionic schneidet React Native bei 60*Hz* (vgl. Tabellen 3 und 4) ähnlich oder besser ab. Dafür verbraucht Ionic jedoch stets weniger CPU und Arbeitsspeicher als die beiden React Native Ansätze (vgl. Tabellen 2, 3 und 4).

Da React Native und Ionic in allen Bewertungskriterien nicht klar besser bzw. schlechter abschneiden, lässt sich keine pauschale Aussage treffen, ob Ionic eine bessere Performance bietet als React Native oder anders herum. Ebenso sind die Ergebnisse der beiden Testgeräte durchaus unterschiedlich.

Abbildung 7: React Native Frame Times mit 60 FPS (links) und 120 FPS (rechts)



Am unteren Bildschirmrand wird angezeigt, wie lange die Bilder gebraucht haben, um angezeigt zu werden. Ist einer der Balken über der dünnen, horizontalen Linie, konnte das Bild nicht rechtzeitig berechnet werden (= verworfenes Bild)

3.5 iOS

Leider bietet iOS keine Möglichkeit zum Abrufen der verworfenen Bilder, Arbeitsspeicherauslastung und CPU-Auslastung über ein CLI[19]. Somit ist das Zusammentragen der Daten mittels Skripten wie unter Android nicht möglich.

Um dennoch die Performance unter iOS bewerten zu können, werden im Folgenden die Werte aus dem Profiler *Instruments* für iOS Anwendungen verwendet. Die Graphen für CPU und Arbeitsspeicherauslastung aus *Instruments* werden hierbei als Bewertungsgrundlage herangezogen.

Für das Bewertungskriterium der verworfenen Bilder wird unter iOS die Anzahl der *Animation Hitches* aufgelistet und nicht der prozentuale Anteil. Dies liegt daran, dass *Instruments* leider keine Informationen über die Gesamtzahl der berechneten Bilder liefert.

Weiterhin ist das Schreiben von automatisierten Oberflächentests nicht so komfortabel wie unter Android. Die zum Scrollen in Listen bereitgestellte Funktion scrollt viel zu schnell durch die Liste. Somit ist ein benutzernaher Test nicht mehr möglich. Deswegen wurde sich dazu entschieden, die Eingaben auf dem Testgeräte manuell durchzuführen. Der Ablauf ist jedoch identisch zum automatischen Test unter Android.

Wegen der inkonsistenten Reproduzierbarkeit der Nutzereingaben und der geringen Testmenge, kann es demnach zu deutlich größeren Abweichungen der Ergebnisse als unter Android kommen. Zusätzlich wurde unter Ionic langsamer gescrollt, da die Elemente der Liste deutlich länger benötigten, um nachzuladen. Aus diesem Grund besitzen die folgenden Werte nicht die Aussagekraft wie die Daten unter Android. Für eine grobe Einordnung der Performance tauchen diese dennoch in dieser Ausarbeitung auf.

Als Testgeräte wurden ein iPhone 7+ und ein iPhone 6s verwendet (vgl. Tabelle 5). Die beiden Geräte sind älter als die unter Android verwendeten Testgeräte (vgl. Tabelle 1).

Tabelle 6: Testergebnisse der Listenansicht des Testgerätes *iPhone 7+*

Framework (n=5)	"Animation Hitches" ∑	Arbeitsspeicher (max)	CPU-Auslastung (max)
Ionic	3,2	1090,00 MiB	168,10 %
React Native	14,4	997,80 MiB	140,60 %
React Native (virtualized list)	15,4	946,98 MiB	172,10 %
Nativ	18,8	843,88 MiB	100,90 %

Tabelle 7: Testergebnisse der Listenansicht des Testgerätes *iPhone 6s*

Framework (n=5)	"Animation Hitches" ∑	Arbeitsspeicher (max)	CPU-Auslastung (max)
Ionic	8,60	876,67 MiB	186,40 %
Nativ	18,00	746,94 MiB	110,80 %
React Native	24,00	773,75 MiB	200,00 %
React Native (virtualized list)	30,60	705,02 MiB	188,40 %

3.6 Auswertung: Ionic und React Native - iOS

Ionic schneidet unter Betrachtung der *Animation Hitches* auf beiden Testgeräten deutlich am besten ab – sogar besser als die native App.

[19] CLI = Command Line Interface

Tabelle 5: Technische Daten der iOS-Testgeräte

Testgerät	**iPhone 6s** [26]	**iPhone 7 Plus** [27]
Veröffentlichungsdatum	September 2015	September 2016
iOS Version	15.7	15.7
CPU	A9 2-Kerne @ 1,84 GHz (max)	A10 4-Kerne @ 2,34 GHz (max)
CPU-Architektur	ARM-64	ARM-64
GPU	PowerVR GT7600	PowerVR Series7XT Plus
Arbeitsspeicher	2 GB	3 GB
Bildwiederholrate (max.)	60Hz	60Hz

Gleichzeitig benötigt Ionic dafür dennoch stets im Maximum am meisten Arbeitsspeicher. (vgl. Tabellen 6 und 7). Auch die maximale CPU-Auslastung ist recht hoch. Diese liegt mit 168.1% auf dem *iPhone 7+* nur knapp vor React Native mit *virtualized list*s und somit auf dem vorletzten Platz (vgl. Tabelle 6). Auf dem *iPhone 6s* liegt Ionic bei Betrachtung der maximalen CPU-Last vor den beiden React Native Implementierungen auf dem zweiten Platz, hinter der nativen Implementierung (vgl. Tabelle 7).

Auf dem *iPhone 7+* liegt React Native unter der Betrachtung der *Animation Hitches* vor der nativen Implementierung. Eine Verbesserung bei der Verwendung von *virtualized list*s ist nicht zu erkennen. Im Gegenteil: Die Implementierung mit *virtualized list*s produziert sogar einen *Animation Hitch* zusätzlich (vgl. Tabelle 6). Analog zu dem Test unter Android (vgl. Abschnitt 3.2.1), ist der Ansatz mit *virtualized list*s CPU-intensiver, dafür aber Speicherressourcen schonender. Jedoch ist das Ergebnis hier weniger deutlich als im Vergleich zu Android (vgl. Tabelle 6 und z. B. Tabelle 2).
Auf dem *iPhone 6s* schneidet React Native im Vergleich allgemein schlechter ab als auf dem *iPhone 7+* (vgl. Tabellen 7 und 6). Auch hier produziert der Ansatz bei der Verwendung von *virtualized list*s mehr *Animation Hitches*. Analog zu den Testergebnissen des *iPhone 7+*s verbraucht der Ansatz mit *virtualized list*s weniger Arbeitsspeicher, jedoch auch im Gegensatz zu allen anderen Tests zusätzlich weniger CPU-Last (vgl. Tabellen 6, 7 sowie Tabellen 2, 3 und 4).

3.7 Auswertung: Native iOS App

Die native App scheidet analog zur Android-Implementierung unter Betrachtung der Arbeitsspeicher- und CPU-Last größtenteils am besten ab (vgl. Tabellen 6, 7 sowie Tabellen 2, 3 und 4). Lediglich React Native kann bei der Verwendung von *virtualized list*s die Arbeitsspeicherauslastung der nativen App auf dem *iPhone 6s* untertreffen (vgl. Tabelle 7). In puncto *Animation Hitches* schneidet die native App sowohl am schlechtesten (vgl. Tabelle 6), aber auch am zweitbesten ab (vgl. Tabelle 7).

4 PERFORMANCEVERGLEICH - STARTZEITEN

Ein schnelles Startverhalten ist extrem wichtig für mobile Anwendungen. Dauert das Laden einer Anwendung zu lange, verlieren Nutzer die Geduld und springen ab. Für Webanwendungen wird häufig der *Largest Contentful Paint* LCP als Metrik verwendet. Dieser beschreibt den Zeitpunkt, an dem das größte Element[20] einer Seite zu Ende geladen wurde. So gilt eine Startzeit bis 2,5*s* als gut, alles zwischen 2,5s und 4s als verbesserungswürdig sowie alles über 4s als schlecht. [28] So steigerte sich z. B. die Konversionsrate um 7%, die Anzahl der Sitzungen um 10% und zusätzlich senkte sich die Absprungrate um 7% als das Unternehmen *COOK*[21] die Ladezeiten ihrer Website um gerade mal 0,85s verbessert hat. [29]

Dieses Konzept aus der Web-Welt wird im Folgenden auf die hybriden Apps übetragen. Die Testapps aus Abschnitt 3 werden auch hier verwendet. Diese zeigen nach dem Start fünf Listenelemente an.

Die erstellten Testanwendungen und die Testapplikationen stehen ebenfalls unter https://git.thm.de/fmdr54/list-view-performance zur Verfügung.

4.1 Android

Einen wirklich fairen Vergleich der Startzeiten der Frameworks experimentell zu untersuchen, ist schwierig. Android bietet zwar die Funktion `Activity#reportFullyDrawn()`[30], jedoch genügt es nicht, diese in den nativen React Native und Ionic Apps aufzurufen, da bis zu dem Zeitpunkt des Aufrufs der Hauptaktivität der LCP noch nicht erreicht sein muss.

Aus diesem Grund wurden für React Native und Ionic native Erweiterungen verwendet bzw. implementiert, um `Activity#reportFullyDrawn()` an der richtigen Stellen im React Code aufrufen zu können. Dieser Aufruf wird dabei in einem `React.useEffect`[22] mit leerem *Dependency-Array*[23] in der Komponente der Listenansicht aufgerufen. Dies bewirkt, dass `Activity#reportFullyDrawn()` einmalig nach Anzeigen der Listenansicht aufgerufen wird.

Für die konkrete Testausführung sind zwei unterschiedliche Testfälle denkbar: Das Ausführen der Apps nach einer Neuinstallation und das erneute Öffnen der bereits installierten Apps.

In beiden Tests werden zunächst alle zu testenden Apps geschlossen. Innerhalb des Neuinstallationstest werden zusätzlich die internen Daten der Apps gelöscht (inkl. Cache). Dadurch wird die App in ihren Ausgangszustand zurückgesetzt. Dieser Art von Test wird im Folgenden deswegen auch als *Kaltstart* bezeichnet.

Weiterhin, gilt es einen Testfall zu definieren, der das wiederholte Öffnen eines wiederkehrenden Nutzers abbildet. Für diesen

[20] z. B. ein Bild oder ein langer Text

[21] https://www.cookfood.net/

[22] Code innerhalb dieser Funktion wird nur ausgeführt, wenn sich ein bestimmter Teil des Zustandes ändert. Der *Dependency-Array* definiert, die Teile des Zustandes, bei der der Code ausgeführt werden soll.

[23] Ist der *Dependency-Array* leer, wird der Code eines `React.useEffect` nur beim initialen Anzeigen der Komponente ausgeführt.

Test wird zwar auch die zu testende App geschlossen, dessen Cache jedoch nicht gelöscht. Sollten die Apps also den Cache zur Beschleunigung des wiederholten Starts verwenden, wäre dies in den Ergebnissen abzulesen.

Die Ergebnistabellen sind aufsteigend nach Kaltstartzeit sortiert. Kleinere Werte bei Kalt- und Warmstart sind auch hierbei ein Indikator für ein besseres Ergebnis. Bei der Warmstartverbesserung hingegen, sind höhere Prozentzahlen besser.

Tabelle 8: Testergebnisse der Startzeit des Testgerätes *OnePlus 8T*

Framework (n=50)	Startzeit Ø "kalt"	Startzeit Ø "warm"	Warmstart Verbesserung
Nativ	205,00 ms	189,34 ms	7,64%
React Native	235,30 ms	226,58 ms	3,71%
Ionic	499,12 ms	493,04 ms	1,22%

Tabelle 9: Testergebnisse der Startzeit des Testgerätes *Samsung Galaxy S8+*

Framework (n=50)	Startzeit Ø "kalt"	Startzeit Ø "warm"	Warmstart Verbesserung
Nativ	555,98 ms	474,46 ms	14,66%
React Native	661,64 ms	652,32 ms	1,41%
Ionic	1069,20 ms	1099,22 ms	-2,81%

4.2 Auswertung der Testergebnisse - Android

Über alle Tests startet die native Android-Implementierung am schnellsten. Dabei jedoch dicht von der React Native Implementation gefolgt (vgl. Tabellen 8 und 9). Auf dem Testgerät *OnePlus 8T* liegen weniger als 40*ms* (vgl. Tabelle 8), auf dem Testgerät *Samsung Galaxy S8+* weniger als 180*ms* zwischen nativ und hybrid in den Kalt- und Warmstart Tests (vgl. Tabelle 9). Ionic ist hierbei deutlich weiter abgeschlagen. In Relation zur nativen App benötigt die Implementation im Kaltstart je nach Testgerät $\approx 294ms$ bis zu $\approx 513ms$ länger. Der Warmstart ist mit $\approx 304ms$ bis zu $\approx 625ms$ ebenfalls deutlich schlechter als das Startverhalten von React Native in Relation zur nativen Implementierung (vgl. Tabellen 8 und 9).

Die Verbesserung durch den Warmstart gegenüber dem Kaltstart ist ebenfalls im nativen Ansatz am höchsten. Hierbei ist jedoch der Unterschied zwischen Ionic und React Native nicht mehr allzu deutlich. Dennoch liegt React Native auch hier vor Ionic, aber hinter der nativen App (vgl. Tabellen 8 und 9). Auf dem Testgerät *Samsung Galaxy S8+* war der Warmstart sogar schlechter als der Kaltstart (vgl. Tabelle 9).

4.3 iOS

Unter iOS werden alle Werte aus den Ergebnissen des *Instruments* Profilings ausgelesen. Konkret wird hierbei die Zeit verwendet, sobald die App im Vordergrund angezeigt wird.[24]

Aufgrund der fehlenden Möglichkeit zum Löschen des Caches über ein CLI repräsentieren die Zeiten lediglich den Warmstart der Apps.

[24] Abgelesen wurde die Startzeit des Lifecycle-States *Foreground - Active*

Tabelle 10: Testergebnisse der Startzeit des Testgerätes *iPhone 6s*

Framework (n=5)	Startzeit Ø
Nativ	783,20 ms
React Native	784,80 ms
Ionic	971,20 ms

Tabelle 11: Testergebnisse der Startzeit des Testgerätes *iPhone 7+*

Framework (n=5)	Startzeit Ø
Nativ	666,81 ms
React Native	688,97 ms
Ionic	877,64 ms

4.4 Auswertung der Testergebnisse - iOS

Unter iOS zeichnet sich ein ähnliches, jedoch nicht ganz so eindeutiges Bild ab wie unter Android (vgl. Tabellen 8 und 9). Der native Ansatz liegt auch hier vorne, jedoch nur dicht gefolgt von React Native mit einem zeitlichen Unterschied von gerade mal $1,6ms$ auf dem iPhone 6s. Ionic liegt auf dem iPhone 6s 188*ms* hinter der nativen App, also kürzer als im Vergleich zu Android. In Relation zu dem Unterschied React Natives von $1,6ms$ jedoch auch unter iOS deutlich langsamer (vgl. Tabelle 10).

Die Testergebnisse des *iPhone 7+*s sind hierbei sehr ähnlich zu denen des iPhone 6s'. Der größte Unterschied ist hierbei, dass alle Ansätze auf dem im Vergleich zu dem iPhone 6s neuerem Gerät schneller sind (vgl. Tabelle 11).

5 FAZIT

5.1 Zusammenfassung

Zunächst wurden die zu vergleichenden Frameworks React (vgl. Abschnitt 2.1), React Native (vgl. Abschnitt 2.2) sowie Ionic (vgl. Abschnitt 2.3) beschrieben. Dabei wurden insbesondere Teile der Architekturen genauer betrachtet, welche aus Sicht des Autors maßgeblich für die Performance der Frameworks relevant sind.
Anschließend wurde ein Testaufbau definiert, der darauf abzielen sollte, die für einen Nutzer merkbare Performanceaspekte bei der Verwendung einer Listenansicht mess- und quantifizierbar zu machen. Dabei wurde sich auf die Verwendung von verworfenen Bildern, die CPU sowie Arbeitsspeicherauslastung festgelegt (vgl. Abschnitt 3).
Weiter wurde erläutert, wie unter Android und iOS die definierten Performanceaspekte aggregiert werden können (vgl. Abschnitt 3.2 bzw. Abschnitt 3.5). Die erlangten Testdaten wurden darauf folgend in Tabellen aufgetragen und anschließend textuell ausgewertet (vgl. Abschnitt 3.3 bzw. Abschnitt 3.6).

5.2 Ergebnis

Aus Sicht des Autors lassen die Testergebnisse nicht klar erschließen, welches Framework am besten geeignet ist. Beide vorgestellten Frameworks hegen dabei Stärken und Schwächen. So ist es zum Beispiel unter Ionic möglich, Teile einer bereits bestehenden Web-Anwendung einzubetten[25]. React Native bietet auf der anderen Seite die Möglichkeit, eine React Native Anwendung in eine bestehende native App zu integrieren. [32]

Trifft man die Auswahl jedoch *nur* anhand der Performance der Frameworks, lassen sich bereits konkretere Aussagen treffen. Dennoch muss man hier kompromissbereit sein. Ausgehend von den Testergebnissen und den Auswertungen der Listenansicht lässt sich sagen, dass React Native ausgehend der verworfenen Bilder eine gute Nutzererfahrung präsentiert, solange man kein Gerät mit $120Hz$ explizit forcieren möchte und die Nutzer der App genügend Rechenleistung vorzuweisen haben.
Ionic hingegen bringt deutlich weniger CPU und Arbeitsspeicherlast mit sich und schafft es, 120 Bilder pro Sekunde zu produzieren. Dennoch war es Ionic häufig nicht möglich, Elemente der Liste schnell genug zu rendern. Anhand der Tatsache, dass JavaScript Single-Threaded ist [33, Seite 2], lässt sich erklären, warum die CPU-Auslastung im Vergleich zu React Native so gering ist. React Natives Fabric-Render-Engine hingegen ist Multi-Threaded und kann somit alle Threads der CPU verwenden, auch für das Rendering. [34]
Der native Ansatz war dabei in Summe am besten. Er benötigte stets wenig CPU wie auch Arbeitsspeicher und erzielte darüber hinaus meistens die wenigsten verworfenen Bilder. Da ein nativer Ansatz stets auf des darunterliegende Betriebssystem optimiert werden kann, ist dieses Ergebnis keine Überraschung.

5.3 Bewertung der Methodik

Unter Betrachtung der Ergebnisse zum Startverhalten sind die Aussagen ähnlich, jedoch viel eindeutiger. Der native Ansatz ist auch hier der beste. React Native überrascht dennoch durch nahezu identische Zeiten unter iOS und ähnliche unter Android. Ionic ist dabei stets weit abgeschlagen. React Natives neue Render Engine *Fabric* hält also – in Relation zu Ionic – das Versprechen über ein stark optimiertes Startverhalten dank *Lazy Loading* (vgl. Abschnitt 2.2.5).

Bei den Testdurchläufen wurde auch stets die Performance der Apps beobachtet und hier stellte sich heraus, dass Ionic häufig Probleme hatte, Elemente der Liste nachzuladen. Dieses Verhalten spiegelte sich jedoch nicht in den verworfenen Bildern wider – es konnte zwar ein Bild geliefert werden, nur war auf diesem die Listenansicht nicht zu sehen. Folglich hat die Quantifizierung dieses Performanceaspektes unter Ionic leider nicht so gut funktioniert wie erhofft.
Ebenso war das Messen der Werte unter iOS problematisch. Ohne die Möglichkeit, die Werte in eine maschinenlesbare Form zu bekommen, war das Durchführen von größeren Testmengen nur mit erheblichem Aufwand möglich, die den Rahmen dieser Arbeit und die vorhandene Zeit übersteigen würden.

5.4 Ausblick

Die geschaffene Grundlage des Performancevergleiches über verschieden Frameworks bietet ein großes Erweiterungs- sowie Verbesserungspotential. Wie bereits in Abschnitt 5.3 beschrieben, weist der aktuelle Testaufbau vereinzelt Lücken auf. Diese könnten in Zukunft behoben werden.
Ebenso könnten weitere Metriken verwendet werden. Z. B. könnte es interessant sein – neben den verworfenen Bildern – zusätzlich das 99-Perzentil der Bilder pro Sekunde bzw. der *frame times*[26] aufzutragen. Dieser Ansatz wird häufig in der Bewertung von Videospielen verwendet, welche ebenfalls jede Sekunde $>\approx 30$-Bilder liefern müssen, um flüssig zu wirken. [35]
Weiterhin ist es denkbar, mehrere Arten von Listenansichten oder gar Grids miteinander zu vergleichen, um Stärken bzw. Schwächen eines Frameworks herauszuarbeiten[27].
Zusätzlich kann geprüft werden, ob weitere Hybrid-Frameworks, wie z. B. Googles *Flutter*[28] oder Microsofts *Xamarin*[29], in die Tests integrierbar sind.

[25] Unter React Native ist dies natürlich prinzipiell auch möglich, nur können nicht alle HTML-Tags verwendet werden. Beispiel: Nur `<View>`-Tags statt `<div>`-Tags. [31] Somit wäre in den meisten Fällen eine größere Anpassung notwendig.

[26] Wie lange benötigt ein Bild, um angezeigt werden zu können?

[27] z. B. Grid mit vielen Bildern, Liste mit aufklappbaren Elementen, sortierbare Listen, etc.

[28] https://flutter.dev

[29] https://dotnet.microsoft.com/en-us/apps/xamarin

Tabellenverzeichnis

Abbildungsverzeichnis

LITERATUR

[1] "Mobile Operating System Market Share Worldwide." [Online]. Available: https://gs.statcounter.com/os-market-share/mobile/worldwide/

[2] Nils Hartmann and Oliver Zeigermann, *React : Grundlagen, fortgeschrittene Techniken und Praxistipps – mit TypeScript und Redux*. Heidelberg: dpunkt.verlag, 2019, vol. 2nd ed. [Online]. Available: https://search.ebscohost.com/login.aspx?direct=true&db=nlebk&AN=2325266&site=ehost-live

[3] "Introducing JSX – React." [Online]. Available: https://reactjs.org/docs/introducing-jsx.html

[4] "Virtual DOM and Internals – React." [Online]. Available: https://reactjs.org/docs/faq-internals.html

[5] "Core Components and Native Components · React Native." [Online]. Available: https://reactnative.dev/docs/intro-react-native-components

[6] "Glossary · React Native - Host View Tree (and Host View)." [Online]. Available: https://reactnative.dev/architecture/glossary

[7] E. Sjölander, "Yoga: A cross-platform layout engine," Dec. 2016. [Online]. Available: https://engineering.fb.com/2016/12/07/android/yoga-a-cross-platform-layout-engine/

[8] "Prerequisites for Applications · React Native." [Online]. Available: https://reactnative.dev/docs/new-architecture-app-intro

[9] "Fabric · React Native." [Online]. Available: https://reactnative.dev/architecture/fabric-renderer

[10] "View Flattening · React Native." [Online]. Available: https://reactnative.dev/architecture/view-flattening

[11] R. Pancholi, "Fabric Architecture-React Native," Mar. 2022. [Online]. Available: https://medium.com/mindful-engineering/fabric-architecture-react-native-a4f5fd96b6d2

[12] C. Liebel, *Progressive Web Apps : das Praxisbuch*, 1st ed. Bonn: Rheinwerk Computing, 2019.

[13] "SMS | Ionic Documentation." [Online]. Available: https://ionicframework.com/docs/native/sms

[14] "Web View | Ionic Documentation." [Online]. Available: https://ionicframework.com/docs/v5/core-concepts/webview

[15] "UI jank detection." [Online]. Available: https://developer.android.com/studio/profile/jank-detection

[16] Android Developers, "Drawn out: How Android renders (Google I/O '18)," Sep. 2018. [Online]. Available: https://www.youtube.com/watch?v=zdQRIYOST64

[17] "Performance Overview · React Native." [Online]. Available: https://reactnative.dev/docs/performance

[18] "Write automated tests with UI Automator." [Online]. Available: https://developer.android.com/training/testing/other-components/ui-automator

[19] "Espresso setup instructions | Android Developers." [Online]. Available: https://developer.android.com/training/testing/espresso/setup

[20] "FlatList · React Native." [Online]. Available: https://reactnative.dev/docs/flatlist

[21] "Using List Views · React Native." [Online]. Available: https://reactnative.dev/docs/using-a-listview

[22] "OnePlus 8T - Full phone specifications." [Online]. Available: https://www.gsmarena.com/oneplus_8t-10420.php

[23] "Samsung Galaxy S8+ - Full phone specifications." [Online]. Available: https://www.gsmarena.com/samsung_galaxy_s8+-8523.php

[24] "React – A JavaScript library for building user interfaces." [Online]. Available: https://reactjs.org/

[25] "120Hz animations? · Discussion #2472 · software-mansion/react-native-reanimated." [Online]. Available: https://github.com/software-mansion/react-native-reanimated/discussions/2472

[26] "Apple iPhone 6s - Full phone specifications." [Online]. Available: https://www.gsmarena.com/apple_iphone_6s-7242.php

[27] "Apple iPhone 7 Plus - Full phone specifications." [Online]. Available: https://www.gsmarena.com/apple_iphone_7_plus-8065.php

[28] P. Walton, "Largest Contentful Paint (LCP)." [Online]. Available: https://web.dev/lcp/

[29] Eggplant, "COOK Increases Conversions By Seven Percent Thanks to Faster Load Time." [Online]. Available: https://blog.eggplantsoftware.com/case-studies/cook-increases-conversions-by-seven-percent-thanks-to-faster-load-time

[30] "App startup time." [Online]. Available: https://developer.android.com/topic/performance/vitals/launch-time

[31] "View · React Native." [Online]. Available: https://reactnative.dev/docs/view

[32] "Integration with Existing Apps · React Native." [Online]. Available: https://reactnative.dev/docs/integration-with-existing-apps

[33] K.-I. D. Kyriakou and N. D. Tselikas, "Complementing JavaScript in High-Performance Node.js and Web Applications with Rust and WebAssembly," *Electronics*, vol. 11, no. 19, p. 3217, Jan. 2022, number: 19 Publisher: Multidisciplinary Digital Publishing Institute. [Online]. Available: https://www.mdpi.com/2079-9292/11/19/3217

[34] "Threading Model · React Native." [Online]. Available: https://reactnative.dev/architecture/threading-model

[35] "Analysing Stutter – Mining More from Percentiles," Sep. 2014. [Online]. Available: https://developer.nvidia.com/content/analysing-stutter-%E2%80%93-mining-more-percentiles-0