Umgang nativer Apps mit auf Web-Apps ausgerichteten Optimierungsstrategien

Paul Schreiber
TU Dresden
Dresden, Deutschland

***Zusammenfassung*—Smartphones und Tablets sind die am häufigsten genutzten Endgeräte für mehrere Milliarden Menschen. Mobile Anwendungen unterstützen uns mittlerweile in jeder Situation und sind sowohl im privaten als auch im kommerziellen Bereich nahezu unverzichtbar geworden. Die Entwicklung von nativen Anwendungen für mehrere Plattformen ist allerdings eine kostenintensive Investition. Die Idee der Cross-Plattform-Entwicklung könnte diese Situation erleichtern. Verschiedene Cross-Plattform-Ansätze wurden bisher entwickelt, welche im ersten Abschnitt kurz gegenübergestellt und mit dem nativen Ansatz verglichen werden. Web-Apps werden als bekanntester Ansatz der Cross-Plattform-Entwicklung schon vermehrt eingesetzt, weshalb sich der zweite Teil vertieft mit diesen auseinandersetzen wird. Abschließend werden Optimierungsstrategien für Web-Apps und ihre Auswirkung auf native Apps untersucht.**

***Index Terms*—Cross-Plattform-Entwicklung, Web-Apps, Optimierungsstrategien**

I. Einleitung

Nachdem die Entwicklung von mobilen Geräten und Applikationen rasant voranschreitet, steigt die Anzahl der Smartphone- & Tabletbesitzer weiterhin an[1]. Bereits jetzt stehen dem Nutzer über 3,3 Millionen Apps im Google Playstore und 2,2 Millionen Apps im iOS-App Store zur Verfügung[2]. Die Bandbreite der angebotenen Anwendungen reicht dabei vom einfachen Alltagshelfer bis hin zur komplexen prozessunterstützenden Applikationen im Businessbereich. Immer mehr Unternehmen setzen auf mobile Apps um ihre Kunden auch über das Internet mit Angeboten zu versorgen. Die Möglichkeiten sind dabei sehr vielfältig, von Online-Shops über Spiele, bis hin zum Navigator-Tool für den Öffentlichen Personennahverkehr gibt es nahezu für jeden Anwendungsbereich eine App. Aufgrund der Präsenz verschiedener führender Betriebssysteme (Android ca. 76%, iOS ca. 23% Marktanteil)[3] reicht es aber nicht, sich auf eine Plattform zu spezialisieren, wenn man möglichst viele Nutzer erreichen möchte. Vielmehr ist es sinnvoll, möglichst alle aber auf jeden Fall die beiden größten Plattformen, iOS und Android, anzusprechen. Doch mobile Plattformen haben sich jeweils zu eigenen Ökosystemen entwickelt, da sie unterschiedliche Programmiersprachen, APIs, UI-Kits und IDEs voraussetzen. Das bedeutet, dass jede Plattform mittlerweile so spezifisch ist, dass man ein eigenes Entwicklerteam benötigt um nativ für ein bestimmtes Betriebssystem zu entwickeln. Dies führt dazu, dass sich die Entwicklungskosten rasant erhöhen, je nachdem wie viele Plattformen man mit seiner Anwendung erreichen möchte. Vor allem für kleinere Unternehmen sind diese Kosten eine große Herausforderung und decken sich meist nicht mit dem Nutzen. Um diesem Effekt entgegenzuwirken und die Kosten zu senken, gibt es ein starkes Bestreben das „Develop once, run many“-Prinzip auf mobile Umgebungen, zu übertragen [1]. Ziel ist es, mit einmaligem Entwicklungsaufwand möglichst viele Mobilplattformen bedienen zu können. Um diesen Ansatz zu ermöglichen, gibt es bereits eine Reihe von Konzepten.

[1]https://de.statista.com/statistik/daten/studie/198959/umfrage/anzahl-der-smartphonenutzer-in-deutschland-seit-2010/

[2]https://de.statista.com/statistik/daten/studie/208599/umfrage/anzahl-der-apps-in-den-top-app-stores/

[3]https://de.statista.com/statistik/daten/studie/256790/umfrage/marktanteile-von-android-und-ios-am-smartphone-absatz-in-deutschland/

II. Native App Entwicklung

Bevor der Vergleich zwischen nativen und Cross-Plattform-Apps behandelt wird, werden zuerst einige Grundbegriffe zum Verständnis des Themas erläutert. Eine „Native App“ ist eine auf die jeweilige Plattform angepasste Anwendung, die im Rahmen der verfügbaren APIs und Bibliotheken vollen Zugriff auf die Hardware des jeweiligen Gerätes hat. Sie ist in der jeweiligen Programmierumgebung mit der dafür vorgesehenen Sprache entwickelt worden und wird über die für die Plattform spezifische zentrale Stelle (Apple AppStore[4], Android Playstore[5], BlackBerry World[6],...) vertrieben [2]. Vorteile einer nativen Entwicklung sind der bereits genannte Vollzugriff auf Hard- und Software des Gerätes sowie eine hohe Performanz. Allerdings ist eine solche native Entwicklung immer plattformspezifisch und bringt hohe Entwicklungskosten bei universellen Apps mit sich. Teilweise werden sogar Entwicklungsumgebungen, wie Apples XCode[7] vorausgesetzt, um überhaupt Apps entwickeln zu können, was die Entwicklungskosten weiterhin in die Höhe treibt. Der zeitliche und finanzielle Aufwand, eine App für mehrere mobile Plattformen zur Verfügung zu stellen, ist somit nicht für jedes Unternehmen realisierbar. In den letzten 5 Jahren ist jedoch eine enorme Entwicklung in diesem Bereich zu vernehmen [2]. Im nachfolgenden werden einige Ansätze vorgestellt.

[4]https://www.apple.com/de/ios/app-store/

[5]https://play.google.com/store

[6]https://appworld.blackberry.com

[7]https://developer.apple.com/xcode/

III. Cross-Plattform-Entwicklung

Die Cross-Plattform-Entwicklung befasst sich mit dem Problem der heterogenen Entwicklung für mobile Plattformen. Dabei ist der zentrale Grundgedanke, denselben Code mehrfach nutzen und so möglichst viele Plattformen mit geringem Entwicklungsaufwand bedienen zu können. Je nach Anforderung und Budget gibt es viele verschiedene Ansätze der Cross-Plattform-Entwicklung, wobei einige hier kurz vorgestellt und mit Beispielen hinterlegt werden.

A. Interpretationsansatz

Der Interpretationsansatz wird von einigen bekannten Frameworks, wie Titanium[8] oder React Native[9] genutzt. Diese nutzen Javascript als Programmiersprache und bieten so Web-Entwicklern die Chance auf einen schnellen Einstieg in die App-Entwicklung. Mit jeder Applikation wird eine Javascript-Laufzeitumgebung mitgeliefert, welche den Quellcode zur Laufzeit interpretiert und mittels einer „Javascript-auf-Nativ"-Brücke die native API ansprechen kann. Dadurch ist es möglich auf Hardware- (Sensoren, Kamera,...) und Software-Komponenten (Kalender, Adressbuch,...) zuzugreifen, solange diese vom Framework implementiert sind. Ein weiterer Vorteil ist die hohe Performanz und die gute Usability. Cross-Plattform-Anbieter wie React Native geben dem Entwickler über ein eigenes Document-Object-Model die Möglichkeit das gesamte UI selbst zu gestalten. Somit können Apps sehr nah an ihre nativen Vorbilder angepasst werden und fühlen sich beim Bedienen gewohnt an. Es ist aber genauso möglich, eine fast komplett identische iOS- und Android-App zu entwickeln, falls mehr Wert auf die Corporate Identity gelegt wird. Das Titanium-Framework nutzt ähnlich dem Android-UI-Editor XML-Dokumente um das Interface zu definieren. Die Entwicklung der Frameworks ist jedoch sehr komplex, sodass es teilweise länger dauern kann, bis alle nativen Features unterstützt werden. Wenn man die zuletzt genannten Nachteile vernachlässigt, sind Frameworks mit Interpretationsansatz aber eine ziemlich günstige und sinnvolle Alternative zu nativen Apps, da man nahezu den vollen Funktionsumfang abbilden kann und wenige Abstriche machen muss.

B. Web-Apps

Der wohl bekannteste Ansatz, der stetig an Popularität gewinnt, ist der der Web-Apps. Der Begriff „Web-App" ist nicht einheitlich definiert und es gibt viele Synonyme wie „full screen Web-App" oder „Pure Web-App". In diesem Paper bezeichnet eine „Web-App" Web-Inhalte, die im Startbildschirm eines Gerätes installiert sind und ein fensterloses User-Interface besitzen. Obwohl sich die verschiedenen mobilen Plattformen mit ihren IDEs, Hard-und Software, Programmiersprachen und APIs stark unterscheiden, weisen sie jedoch alle eine Gemeinsamkeit auf – sie besitzen eine Browser-Engine. Durch die zunehmende Digitalisierung ist die Anzahl der Webseiten rasant gestiegen[10], wodurch der Browser als

FEATURES	Progressive Web App	Native App	Renponsive Web
Multi Platform Capability	✓	✗	✓
Low Cost to build	✓	✗	✗
Installation Required	✗	✓	✗
Updates Required	✗	✓	✗
Push Notifications	✓	✓	✗
Easy Sharing	✓	✗	✓
Low Data Consumption	✓	✗	✓
Offline Usability	✓	✗	✗
Faster UI	✓	✗	✗

Abbildung 1. Progressive Web-Apps als App-Modell der Zukunft.[12]

Anzeigemedium nicht mehr wegzudenken ist und sich zu einer ernstzunehmenden Zielplattform entwickelt hat. Durch das Vorhandensein des Webbrowsers oder mindestens einer Browser-Engine auf jedem Mobilgerät, können mobile Webseiten entwickelt werden, welche auf allen Plattformen nahezu gleich aussehen. Mittlerweile können auch mehr als nur Textinhalte dargestellt werden. Der mobile Browser ähnelt zunehmend einer Unterhaltungsplattform mit Spielen, Fotos, Videos, Karten und Musik [3]. Diese weitgefächerte Unterstützung ist die Voraussetzung, um mit nativen Applikationen mithalten zu können. Doch eine native App bietet weitere Features, wie Offline-Support, Zugriff auf die Hardware und Push-Notifications. Vor einigen Jahren waren das noch unschlagbare Argumente gegen Web-Apps, doch mittlerweile stehen diese nativen Apps auf Augenhöhe gegenüber. Web-Apps, die diese Eigenschaften aufweisen, werden als „Progressive Web-Apps" bezeichnet und Google selbst ist davon überzeugt, dass diese das App-Modell der Zukunft sind[11]. Diese Aussage ist nachvollziehbar, da Web-Apps mittlerweile viele Features unterstützen und günstig in der Entwicklung sind. Gleichzeitig decken sie viele Plattformen ab und bieten einige Vorteile gegenüber nativen Apps und normalen Webseiten [s. Abbildung 1].

Die meisten Hersteller der Smartphone-Betriebssysteme stellen Toolkits bereit, die Zugriff auf die Gerätesensorik zulassen und somit den Funktionsumfang von Webapplikationen deutlich erweitern. Dafür kommt proprietäre Software zum Einsatz, wie Googles Web Toolkit [1]. Bisher wurden noch native Anwendungsrahmen wie Cordova[13] oder Electron[14] benötigt, die den Web-Inhalt in einen sogenannten „Wrapper" packen und als Schnittstelle zur nativen Umgebung dienen. Diese werden nun durch native Webschnittstellen ersetzt. Dabei nimmt der sogenannte Service Worker eine zentrale Position ein. Dieser bietet die Möglichkeit, JavaScript unabhängig

[8] www.appcelerator.com
[9] https://facebook.github.io/react-native/
[10] https://de.statista.com
[11] https://developers.google.com/web/progressive-web-apps/
[12] http://halwits.com/wp-content/uploads/2017/08/table.jpg
[13] https://cordova.apache.org/
[14] https://electronjs.org/

Abbildung 2. Google Maps Go – Kein Unterschied zum nativen Original[20]

vom Hauptthread auszuführen. Features wie Offline-Nutzung und Push-Benachrichtigung sind somit einfach realisierbar[15]. Mithilfe des „Web App Manifest“ kann ein App-Icon für den Homescreen festgelegt werden. Somit ist die Web-App vom ersten visuellen Eindruck nicht mehr von einer nativen zu unterscheiden. Lediglich im laufenden Betrieb besteht die Möglichkeit, dass die native App im direkten Vergleich flüssiger läuft und somit unterscheidbar ist.

Neben der Plattformunabhängigkeit haben Web-Apps weitere Vorteile gegenüber nativen Apps. Durch die konsequente Trennung von Inhalt, Logik und Design ergibt sich eine starke Flexibilität der Anwendung. Das HTML-Markup beschreibt den Inhalt, JavaScript kapselt die Logik und CSS ist für die Darstellung der Inhalte verantwortlich. Somit können große Teile der App auf anderen Geräten wiederverwendet werden. Außerdem wird der HTML-Standard durch W3C[16]und WHATWG[17] stetig weiterentwickelt [3]. Diese Eigenschaften verleihen der Web-App großes Potenzial, um ein ernstzunehmender Konkurrent für native Apps zu werden. Allerdings muss man hinzufügen, dass nicht alle Hersteller den gleichen Entwicklungsaufwand in diese Technik investieren. Google ist der Hauptakteur, der Progressive Web-Apps weiter vorantreiben will, doch auch Mozilla, Microsoft und Opera Mini sichern Unterstützung in ihren Browsern zu. Nur Apple reagierte zunächst verhalten auf die Ankündigungen und kündigte erst verspätet an, dass mit der Implementierung des Service Worker begonnen wurde und diese ab iOS 11.3 Beta unterstützt werden[18]. Google hingegen veröffentlicht seit November 2017 in kürzester Zeit eine ganze Reihe an Web-Apps unter dem Namen „Go-Apps“[19]. Diese sind eine Art Light-Variante der bekannten Google-Apps, wie zum Beispiel Youtube, Google Maps oder Google Assistant. Man spürt beim Benutzen aber keinen Unterschied zwischen der nativen App oder der neuen Go-App. Auch dem User Interface ist nicht anzumerken, dass es sich hierbei um eine Web-App handelt, wie man in Abbildung 2 gut erkennen kann.

[15]https://www.w3.org/TR/service-workers-1/

[16]hrefhttps://www.w3.org/ https://www.w3.org/

[17]hrefhttps://whatwg.org/https://whatwg.org/

[18]https://www.heise.de/developer/artikel/Progressive-Web-Apps-Teil-1-Das-Web-wird-nativ-er-3733624.html

[19]https://www.googlewatchblog.de/2018/02/google-maps-go-die/https://www.googlewatchblog.de/2018/02/google-maps-go-die/

IV. Optimierung von Web-Apps

Dieser Abschnitt beschäftigt sich mit Optimierungsstrategien von Web-Apps. Außerdem wird untersucht, inwiefern native Apps dieselben Strategien verwenden können. Zunächst werden einige Grundlagen des Web-Aufbaus erläutert, um später die Optimierungen besser nachvollziehen zu können. Die Basis des heutigen „World Wide Web“ ist die 1999 veröffentlichte Hypertext Markup Language (HTML). Diese war von Beginn an als plattformunabhängige offene Dokumentenbeschreibungssprache konzipiert [1]. Das 1996 publizierte Protokoll HTTP/1 und sein Nachfolger HTTP/2 waren weitere wichtige Grundsteine um die Datenübertragung zwischen Client und Server sicher zu stellen. Diese Technologien bilden die Grundlagen des Internets und sämtlicher darauf aufbauender Plattformen. Zu diesen beiden Basistechnologien gesellen sich JavaScript, eine dynamisch typisierte Skriptsprache für Manipulationen des dem HTML-Standard zu Grunde liegenden Document Object Models (DOM). Um die Darstellung des Dokumentes anzupassen, werden sogenannte Cascading Style Sheets (CSS), eine Sprache zur Definition von Dokumentvorlagen auf Basis von HTML Tags, verwendet [1].

Doch diese unoptimierten Grundtechnologien allein, genügen mittlerweile nicht mehr um die heutigen heterogenen Anforderungen an das Internet zu bedienen. Die Vielfalt der internetfähigen Endgeräte reicht von der Smartwatch über das Tablet bis hin zum Desktop-PC und jedes Medium muss individuell angepasst werden. Angefangen bei unterschiedlichen Display-Größen bis hin zur Verbindungsgeschwindigkeit gibt es viele variable Größen, die berücksichtigt werden müssen. Während das „Desktop-Internet“ in erster Linie für Anwendung auf stationären Computern mit verlässlicher Internetverbindung konzipiert war, gibt es in der Welt der mobilen Endgeräte neue Herausforderungen zu bewältigen [1]. Die Mobilität, die Smartphones ermöglichen, bringt die Problematik mit sich, dass Verbindungen unterwegs aufgebaut und genutzt werden, die langsam oder aufgrund schlechtem Empfang instabil sind. Eine Web-App besteht häufig aus einer Reihe von HTML-Dokumenten, die aufgrund des verwendeten „Lazy-Evaluation-Prinzips“ Stück für Stück geladen werden müssen. Bei Verbindungsabbrüchen bedeutet dies, dass der Benutzer seine Arbeit mit der Applikation beenden oder unterbrechen muss, bis wieder eine Datenfunkverbindung besteht [1]. Die Gründe für Optimierungen sind häufig dieselben. Meistens geht es darum, die Ladezeit oder das Ladevolumen zu verringern, um dem Nutzer die optimale User Experience zu gewährleisten. Eine Studie von Google zeigt, dass Webseiten, die länger als drei Sekunden zum Laden benötigen, von 53% der Besucher wieder verlassen werden [4]. Dies hat natürlich katastrophale Auswirkungen, wenn man es aus der Sicht eines Onlineshops oder anderen Dienstleistern betrachtet, die mit jedem Abbruch einen potenziellen Kunden verlieren.

[20]http://geoawesomeness.com/wp-content/uploads/2017/12/Google-Maps-Go.jpg

Hinzu kommt, dass Googles Suchalgorithmus Webseiten mit einer geringeren TTFB („Time to first Byte“) bevorzugt und höher im Google-Ranking einstuft. Dies führt dazu, dass schnelle Seiten bei Suchanfragen bevorzugt dem Suchenden als Ergebnis geliefert werden. Als Betreiber einer langsamen Webseite erleidet man hier weiteren Verlust an Seitenbesuchern, da man in den Einträgen erst an späterer Stelle, wenn nicht sogar erst auf der nächsten Seite erscheint[21]. Die Studie zeigt weiterhin, dass jeder Zweite eine Ladezeit von circa zwei Sekunden für eine Webseite erwartet aber knapp 80% aller mobilen Webseiten länger als 10 Sekunden zum Laden im 3G-Netz benötigen [4].

Ähnlich sieht es bei der Optimierung des Datenvolumens aus. Zwar führt die Reduzierung des Datenvolumens automatisch meist zu einer besseren Ladezeit aber es gibt auch noch andere Vorteile. Zum einen gibt es in vielen Mobilfunkverträgen begrenzte Datenvolumen, wodurch jeder Nutzer nur ein bestimmtes Kontingent zur Verfügung hat. Zum anderen existiert noch keine globale Netzabdeckung von 3G oder einer schnelleren Mobilfunktechnik, sodass einige Standorte nur mit EDGE[22] oder ähnlichen Technologien versorgt sind. Das resultiert in niedrigeren Übertragungsgeschwindigkeiten und somit teilweise zu Ladezeiten, die über 30 Sekunden liegen. Aus verschiedenen Studien geht hervor, dass eine kurze Ladezeit einer Webseite Grundvoraussetzung für die Erfüllung der User Experience ist [5], [6]. Somit ist es schnell ersichtlich, dass eine Optimierung der Ladeperformanz von Nöten ist. In den folgenden Abschnitten werden einige Optimierungsstrategien vorgestellt, welche dieses Problem von verschiedenen Seiten beleuchten. Dabei wird zwischen der Optimierung der Daten und der Kommunikation unterschieden [7]. Es gibt noch einige andere Strategien, welche hier allerdings den Rahmen sprengen würden.

A. Optimierung der Daten

Die Optimierung der Daten selbst versucht das Problem der Ladezeiten an der Wurzel zu beheben. Das bedeutet, es wird aktiv versucht, die übertragenen Daten zu reduzieren, indem diese vor der Übertragung komprimiert werden. Der Sinn der Reduktion von Daten ist dabei, dass nahezu kein Inhalt verloren geht. Bei einem Bild werden zum Beispiel verschiedene Kompressionsarten angewendet, die dieses um ein Vielfaches verkleinern. Die Qualität des Bildes nimmt allerdings in so einem geringen Maße ab, dass man dies mit bloßem Auge kaum wahrnimmt. Hier werden nun zwei Beispiele für Reduktion von Daten als Optimierungsstrategie erläutert.

Kompression

Die Kompression ist eine der wichtigsten Optimierungsstrategien im Web-Kontext [8]. Die Bewertung einer mobilen Webseite oder Web-App hängt, wie bereits erwähnt, stark von der PLT[23] ab und diese verringert sich, wenn große Dateien vorher komprimiert wurden. Durch die Reduzierung der zu übertragenden Dateigröße, können also Prozesse beschleunigt und enorm viel Zeit gespart werden. Man unterscheidet generell in verlustfreie und in verlustbehaftete Kompression. Wie der Name schon sagt, beschreibt die verlustbehaftete Kompression die Überführung einer Datei in eine Form die weniger Speicherplatz benötigt, wobei Daten entfernt werden, deren Verlust kaum wahrnehmbar ist. Ein Beispiel dafür ist die JPEG Kompression, bei der eine Kombination aus verlustbehafteter und verlustfreier Kompression genutzt wird um Bilder um ein Vielfaches zu verkleinern [8], [9]. Durch diese Technik kann die Dateigröße je nach Komprimierungsstufe und Ursprungsgröße um mehrere Megabyte reduziert werden. In der Praxis bedeutet das, dass die komprimierten Bilder schneller vom Server zum Client gesendet werden können und sich die PLT verringert. Bei heutigen Webseiten und somit auch Web-Apps ist dies ein Standard-Verfahren, welches weitgehend eingesetzt wird. Auch native Apps profitieren von dieser Optimierung da diese den gleichen Server anfragen können und somit ebenfalls die komprimierten Dateien laden. Anwendungen der verlustbehafteten Kompression finden sich auch bei anderen Formaten im Video- und Audio-bereich wieder [10].

Die verlustfreie Kompression kann ebenfalls zur Komprimierung von Dateien im Web genutzt werden. Vorallem CSS- und JS-Dateien lassen sich über Tools wie Minify[24] problemlos verkleinern[25], indem semantisch nicht benötigte Zeichen, wie Tabulatoren, Leerzeichen und Kommentare entfernt werden. Je nach Aufbau des Dokumentes, können hier 30%-50% der Ursprungsgröße reduziert werden [8]. Oft bleibt diese spezielle Optimierung den Web-Apps vorbehalten. Da die meisten nativen Apps keine CSS-Skripte benötigen und über den Store der jeweiligen Plattform geladen werden, kann auch keine Kompression genutzt werden. Es gibt allerdings Ausnahmen, die dann natürlich einen Nutzen aus diesem Verfahren ziehen können.

B. Optimierung der Kommunikation

Auch die Optimierung der Kommunikation hat erheblichen Einfluss auf die PLT einer Web-Anwendung. Durch geschickte Mechanismen wie Caching müssen oft angefragte Ressourcen nicht mehrfach geladen werden und durch die Planung des Datenzugriffs können Engpässe instabiler Datenverbindungen kompensiert werden. Dabei bleiben die eigentlichen Daten unberührt und nur die Kommunikation zwischen Client und Server wird optimiert.

Priorisierung

Eine Art der Optimierung der Kommunikation ist die Priorisierung [7]. Ein Kennzeichen des Webseiten-Ladeprozesses ist, dass nicht alle Komponenten die gleiche Nützlichkeit und Wichtigkeit haben. Einige Ressourcen

[21] https://neilpatel.com/de/blog/wir-haben-143-827-urls-analysiert-und-die-haufig-ubersehenen-geschwindigkeitsfaktoren-entdeckt-die-google-rankings-beeinflussen/

[22] Enhanced Data Rates for GSM Evolution

[23] Page Load Time

[24] https://www.minifier.org/

[25] https://developers.google.com/speed/docs/insights/MinifyResources

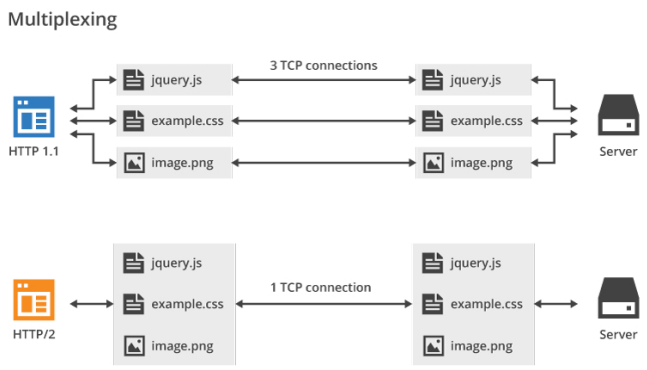

Abbildung 3. Vergleich TCP-Verbindungen HTTP/1 & HTTP/2[26]

müssen auf der Client-Seite noch verarbeitet werden (z. B. HTML, CSS, JavaScript), andere dienen nur der visuellen Vervollständigung einer Webseite oder Web-App (z.B. Bilder und Schriftarten). Da nicht alle Komponenten zur gleichen Zeit geladen werden können und die Durchschnittsgröße der Webseiten steigt, ist es besonders wichtig einen geeigneten Priorisierungsalgorithmus zu finden [11]. Genau diesen wichtigen Aspekt erkannten die Entwickler des Internetprotokolls HTTP/2. Beim Vorgänger HTTP/1.1 musste pro TCP-Verbindung ein Seitenelement (HTML-, CSS-, JS-Datei) komplett übertragen werden, bevor das nächste Element angefangen wurde. Das Problem war, dass beim Übertragen besonders großer Dateien, das sogenannte „head-of-Line-blocking“ eintrat. Daraus folgte, dass keine Datei der nachfolgenden Übertragungen beginnen konnte, da die aktuelle Übertragung noch nicht beendet war. Bei einer ungünstig aufgebauten Webseite konnte es vorkommen, dass eine unwichtigere große Datei, mehrere kleine schnell benötigte Dateien blockierte. Daraufhin gab es einige Workarounds, die versuchten dieses Problem zu beheben [12]. Mit HTTP/2 wurde dieses Problem behoben, indem eine einzige TCP-Verbindung aufgebaut wird, die parallel benutzt werden kann [s. Abbildung 3]. Das ermöglicht einen enormen Performanzgewinn, da die Inhalte zur benötigten Zeit priorisiert werden können. Außerdem ist es möglich, jeder Komponente eine Reihenfolge je nach Wichtigkeit zuzuordnen. Somit können wichtige Strukturdaten, wie HTML, Darstellungsbefehle, wie CSS-Anweisungen oder JS-Dateien für Anwendungslogik priorisiert werden und der Nutzer kann schneller mit der Seite interagieren. Weniger wichtige Ressourcen, wie Bilder oder Schriftarten, werden nachgeladen, sobald alle wichtigen Elemente geladen wurden. Dies führt zu einer erheblich verkürzten Ladezeit, bis der Nutzer die Seite sinnvoll nutzen kann. Eine Verbesserung für native Apps bringt diese Optimierungsstrategie aber nicht. Native Apps besitzen bereits eine Struktur, wie der Inhalt angeordnet ist und laden sich oft nur aktuelle Inhalte vom Server. Somit benötigt eine native App weder HTML- noch CSS-Dateien und eine Priorisierung dieser bringt keinen Vorteil. Im Gegenteil, wenn eine native App die gleiche Anfrage stellt wie eine Web-App, benötigt sie eigentlich nur Inhalte, wie zum Beispiel neue Tweets oder Nachrichten. Stattdessen bekommt sie in den ersten Sekunden nur Strukturdaten, die sie nicht verarbeiten kann. Erst nachdem die Übertragung dieser abgeschlossen ist, folgen relevante Daten wie Bilder. Um diese Blockierung der nativen App zu vermeiden, sollte immer eine separate API verfügbar sein, die von der nativen App angefragt werden kann und nur relevante Daten liefert.

Nachrichten-Zustellung

Auch die Nachrichten-Zustellung kann bei der Client-Server-Kommunikation optimiert werden. Einer der wichtigsten Optimierungen, welche unter anderem von Google empfohlen werden[27], ist das sogenannte Caching [13]. Ziel ist es, die Anzahl der HTTP-Requests zu verringern [8] und somit wertvolle Zeit zu sparen. Ein Cache speichert Antworten auf häufige Anfragen an den Server zwischen, sodass dieser nicht jedes Mal angefragt werden muss. Dabei gibt es verschiedene Arten von Caching, die sich durch den Ort der Anwendung des Cachings unterscheiden. Der lokale Cache, der sich auf dem Gerät befindet, speichert Daten, die häufig angefragt werden und kann auch als Speicher für Vorlade-Strategien wie „Prefetching“ verwendet werden. Proxy- und Server-Cache werden angefragt, wenn der lokale Cache die Anfrage nicht bedienen kann. Es kann zum Beispiel sein, dass ein anderer Nutzer, der im selben Proxy angemeldet ist, die gleiche Anfrage vor kurzer Zeit gestellt hat. Somit muss nicht extra der Server angefragt werden, da die Antwort auf die Anfrage noch im Cache des Proxys liegt. Im Falle der Web-App muss der Entwickler keinen großen Aufwand betreiben, da der Browser bereits einen Cache integriert hat. Da die Web-App einer mobilen Webseite stark ähnelt, hat man teilweise nur minimalen Entwicklungsaufwand in diesem Gebiet. Somit können häufig benötigte Daten mit wenig Aufwand zwischengespeichert werden. Bei der nativen App sieht das etwas anders aus. Diese kann leider keinen Gebrauch des Browsercaches machen und die Entwickler müssen sich selbst darum, kümmern einen Cache-Algorithmus zu implementieren.

Planen des Datenzugriffs

Eine weitere wichtige Optimierungsstrategie ist das vorausschauende Planen des Datenzugriffs. Auch hier gibt es verschiedene Ansätze, wovon einer nachfolgend erläutert wird.

Eine weit verbreitete Methode um Ladezeiten zu optimieren und Latenzen zu vermeiden ist „Prefetching“ [14]. Anhand von verschiedenen Strategien wird versucht zu erraten, welche Daten vom Nutzer als nächstes benötigt werden. Durch Ansätze wie PREPP[28] kann auf Basis des Nutzerverhaltens

[26] https://blog.cloudflare.com/content/images/2015/12/http-2-multiplexing.png

[27] https://developers.google.com/speed/docs/insights/mobile

[28] Predictive Practical Prefetch

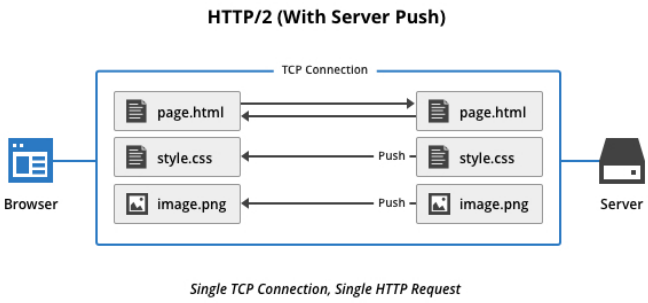

Abbildung 4. Optimierung durch Server Push[30]

eine statistisch wahrscheinliche Handlung abgeleitet werden [15], welche Informationen der Nutzer als nächstes benötigt. Somit können diese Daten vorsorglich in den Cache geladen werden, sodass beim tatsächlichen Anfordern der Daten keine Ladezeit entsteht. Um ein möglichst gutes Nutzererlebnis zu garantieren, hängt alles von der Qualität des Algorithmus zur Bestimmung der zunächst benötigten Daten ab [16]. Sowohl Web-Apps als auch native Apps können Nutzen aus diesem Verfahren ziehen, allerdings müssen entsprechende Algorithmen implementiert werden.

Server Push

Die letzte hier vorgestellte Optimierung der Kommunikation ist der Server Push. Ähnlich dem Prefetching werden hier Daten vorgeladen, die der Nutzer als nächstes benötigen könnte. Der Server Push ist ein Bestandteil des neuen HTTP/2-Protokolls [11]. Folgendes Beispiel soll die Vorteile dieser Technik näher beleuchten. Normalerweise fragt ein Browser nach einer index.html und erhält diese dann vom Server. Beim Verarbeiten der Datei erkennt er Verlinkungen zu CSS-Dateien und Javascript-Code, die zur Darstellung der angeforderten Seite nötig sind. Nun müsste der Browser alle verlinkten Ressourcen beim Server anfragen. Die Kommunikation, die bei den „Anfragen“ der verlinkten Dateien entstehen kosten natürlich Zeit. Mithilfe des Server Push löst HTTP/2 das Problem, indem im HTTP-Header alle Dateien definiert sind, die der Browser benötigen wird um die Seite korrekt anzuzeigen[29]. Während die angefragte Seite also verarbeitet wird, erhält der Browser mittels Server Push bereits die benötigten Ressourcen und weitere Nachfragen entfallen [s. Abbildung 4]. Da diese Technik durch das HTTP/2-Protokoll ermöglicht wird, profitieren nur Web-Apps und Webseiten von dieser Optimierung. Eine Unterstützung seitens der nativen App ist aber in Zukunft vorstellbar.

V. Diskussion

Der Vergleich der vorgestellten Strategien zur App-Entwicklung zeigt, dass Progressive Web-Apps einen der vielversprechendsten Ansätze darstellen, da sie mittlerweile mit nativen Apps auf Augenhöhe stehen und von Google als App-Modell der Zukunft bezeichnet werden. Die visuelle Ähnlichkeit und die fortschreitende Unterstützung von Hard- und Software machen die Unterscheidung zwischen PWA und nativer App immer schwieriger. Nichtsdestotrotz gibt es auch bei PWAs noch Schwächen, die sich vor allem in der vollständigen Unterstützung durch alle Plattformen zeigen. Jüngste Artikel zeigen, dass Apple es den Entwicklern von Progressive Web-Apps schwer macht, da nicht alle Features ohne Probleme unterstützt werden[31]. Die vorgestellten Optimierungsstrategien sprechen hingegen für die Web-Apps, da diese Strategien oft einfacher zu implementieren sind als bei nativen Apps und einen erheblichen Einfluss auf Ladezeiten vorweisen. Zusammenfassend kann man sagen, dass Progressive Web-Apps zum ernstzunehmenden Konkurrenten nativer Apps geworden sind. Aufgrund Googles umfangreicher Unterstützung bei der Weiterentwicklung von PWAs, lohnt es sich auf jeden Fall die Wahl der App-Entwicklungsstrategie zu überdenken.

VI. Zusammenfassung

In diesem Paper wurden aktuelle Möglichkeiten der App-Entwicklung vorgestellt, wobei der Fokus auf der Idee der Cross-Plattform-Entwicklung lag. Dabei wurden der Interpretationsansatz von Frameworks wie Titanium oder React Native und der Ansatz der web-gestützten Entwicklung mit dem der nativen Entwicklung verglichen. Im Anschluss wurden Optimierungsstrategien für Web-Apps und ihre Auswirkungen auf native Applikationen behandelt.

Literatur

[1] Felix Willnecker, Damir Ismailović, and Wolfgang Maison. *Architekturen mobiler Multiplattform-Apps*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[2] Prof. Dr. Gunnar Teege, Oscar Bertin, J¨org Czanderle, Volker Eiseler, and Christian Rolly. Anwendungsentwicklung fur Smartphones. 2012.

[3] Kristin Albert and Michael Stiller. *Der Browser als mobile Plattform der Zukunft – Die Möglichkeiten von HTML5-Apps*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[4] DoubleClick by Google. The need for mobile speed.

[5] Matteo Varvello, Jeremy Blackburn, David Naylor, and Kostantina Papagiannaki. EYEORG : A Platform For Crowdsourcing Web Quality Of Experience Measurements. 2016.

[6] Ron Kohavi, Alex Deng, and Roger Longbotham. Seven Rules of Thumb for Web Site Experimenters. pages 1–11, 2014.

[7] Thomas Springer. Application Development for Mobile and Ubiquitous Computing Seminar Introduction Basic Idea :.

[8] Alex Nicolaou. Best Practices on the Move: Building Web Apps for Mobile Devices. *Queue*, 11(6):30, 2013.

[9] Torsten Zichner. JPEG - Kompression. 2002.

[10] Stephen J Solari. *Digital Video and Audio Compression*. McGraw-Hill Professional, 1st edition, 1997.

[11] Maarten Wijnants, Robin Marx, Peter Quax, and Wim Lamotte. HTTP/2 Prioritization and its Impact on Web Performance. In *Proceedings of the 2018 World Wide Web Conference on World Wide Web - WWW '18*, pages 1755–1764, New York, New York, USA, 2018. ACM Press.

[12] Anne-Sophie Brylinski and Aniruddha Bhattacharjya. Overview of HTTP/2. In *Proceedings of the Second International Conference on Internet of things, Data and Cloud Computing - ICC '17*, pages 1–6, New York, New York, USA, 2017. ACM Press.

[29] https://www.cyon.ch/blog/HTTP2-Server-Push

[30] https://support.cloudflare.com/hc/en-us/articles/115002816808-How-do-I-enable-HTTP-2-Server-Push-in-WordPress

[31] https://t3n.de/news/progressive-web-apps-ios-safari-942769/

[13] San Mateo, Ravindra Prakash, and San Jose. (12) United States Patent. 2(12), 2005.
[14] Ryen W White and Fernando Diaz. Search Result Prefetching on Desktop and Mobile. 35(3), 2017.
[15] Abhinav Parate, B Matthias, David Chu, and Benjamin M Marlin. Practical Prediction and Prefetch for Faster Access to Applications on Mobile phones. pages 275–284, 2013.
[16] Zhimei Jiang and L Kleinrock. Web prefetching in a mobile environment. *IEEE Personal Communications*, 5(5):25–34, 1998.

Auf alle verwendeten URL's wurde am 07.06.2018 erfolgreich zugegriffen.