



TECHNISCHE HOCHSCHULE NÜRNBERG
GEORG SIMON OHM

Fakultät Informatik

**Betrachtung aktueller Chancen von
Progressive Web Apps im Gegensatz zu
Native Apps**

Bachelorarbeit im Studiengang Medieninformatik

vorgelegt von

Mahja Sarschar

Matrikelnummer 320 1818

Erstgutachter: Prof. Dr. Matthias Teßmann

Zweitgutachter: Prof. Dr. Christian Schiedermeier

Betreuer: Dipl.-Ing. Michael Müller

Unternehmen: OPITZ CONSULTING GmbH



Dieses Werk einschließlich seiner Teile ist **urheberrechtlich geschützt**. Jede Verwertung außerhalb der engen Grenzen des Urheberrechtsgesetzes ist ohne Zustimmung des Autors unzulässig und strafbar. Das gilt insbesondere für Vervielfältigungen, Übersetzungen, Mikroverfilmungen sowie die Einspeicherung und Verarbeitung in elektronischen Systemen.

Hinweis: Diese Erklärung ist in alle Exemplare der Abschlussarbeit fest einzubinden. (Keine Spiralbindung)

Prüfungsrechtliche Erklärung der/des Studierenden

Angaben des bzw. der Studierenden:

Name: Sarschar

Vorname: Mahja

Matrikel-Nr.: 3201818

Fakultät: Informatik



Studiengang: Medieninformatik



Semester: Sommersemester



2021

Titel der Abschlussarbeit:

Betrachtung aktueller Chancen von Progressive Web Apps im Gegensatz zu Native Apps

Ich versichere, dass ich die Arbeit selbständig verfasst, nicht anderweitig für Prüfungszwecke vorgelegt, alle benutzten Quellen und Hilfsmittel angegeben sowie wörtliche und sinngemäße Zitate als solche gekennzeichnet habe.

Nürnberg, 31.07.2021, *Mahja Sarschar*

Ort, Datum, Unterschrift Studierende/Studierender

Erklärung zur Veröffentlichung der vorstehend bezeichneten Abschlussarbeit

Die Entscheidung über die vollständige oder auszugsweise Veröffentlichung der Abschlussarbeit liegt grundsätzlich erst einmal allein in der Zuständigkeit der/des studentischen Verfasserin/Verfassers. Nach dem Urheberrechtsgesetz (UrhG) erwirbt die Verfasserin/der Verfasser einer Abschlussarbeit mit Anfertigung ihrer/seiner Arbeit das alleinige Urheberrecht und grundsätzlich auch die hieraus resultierenden Nutzungsrechte wie z.B. Erstveröffentlichung (§ 12 UrhG), Verbreitung (§ 17 UrhG), Vervielfältigung (§ 16 UrhG), Online-Nutzung usw., also alle Rechte, die die nicht-kommerzielle oder kommerzielle Verwertung betreffen.

Die Hochschule und deren Beschäftigte werden Abschlussarbeiten oder Teile davon nicht ohne Zustimmung der/des studentischen Verfasserin/Verfassers veröffentlichen, insbesondere nicht öffentlich zugänglich in die Bibliothek der Hochschule einstellen.

Hiermit genehmige ich, wenn und soweit keine entgegenstehenden Vereinbarungen mit Dritten getroffen worden sind,
 genehmige ich nicht,

dass die oben genannte Abschlussarbeit durch die Technische Hochschule Nürnberg Georg Simon Ohm, ggf. nach Ablauf einer mittels eines auf der Abschlussarbeit aufgebrachten Sperrvermerks kenntlich gemachten Sperrfrist

von Jahren (0 - 5 Jahren ab Datum der Abgabe der Arbeit),

der Öffentlichkeit zugänglich gemacht wird. Im Falle der Genehmigung erfolgt diese unwiderruflich; hierzu wird der Abschlussarbeit ein Exemplar im digitalisierten PDF-Format auf einem Datenträger beigefügt. Bestimmungen der jeweils geltenden Studien- und Prüfungsordnung über Art und Umfang der im Rahmen der Arbeit abzugebenden Exemplare und Materialien werden hierdurch nicht berührt.

Nürnberg, 31.07.2021, *Mahja Sarschar*

Ort, Datum, Unterschrift Studierende/Studierender

[Formular drucken]

Datenschutz: Die Antragstellung ist regelmäßig mit der Speicherung und Verarbeitung der von Ihnen mitgeteilten Daten durch die Technische Hochschule Nürnberg Georg Simon Ohm verbunden. Weitere Informationen zum Umgang der Technischen Hochschule Nürnberg mit Ihren personenbezogenen Daten sind unter nachfolgendem Link abrufbar: <https://www.th-nuernberg.de/datenschutz/>

Kurzdarstellung

In der vorliegenden Bachelorarbeit wird untersucht, ob und welche Vor- und Nachteile das Entwickeln einer Progressive Web App – eine Webanwendung, die Funktionalitäten wie Offlinebetrieb und Installation unterstützt – im Gegensatz zu einer nativen Anwendung bieten. Ferner soll damit die Frage beantwortet werden, ob sie diese ersetzen kann, um die Chancen von Progressive Web Apps zu analysieren. Die Besonderheit dabei ist, dass für die Entwicklung der Native App ein plattformunabhängiger Ansatz gewählt wurde, um die Technologien auf eine vergleichbare Ebene zu bringen.

Für den Vergleich wurde auf Basis eines Kriterienkatalogs, bestehend aus Funktionalität, Kompatibilität mit verschiedenen Betriebssystemen und Entwicklungsaufwand, jeweils eine App entwickelt. Diese stellt die aktuellen COVID-19 Fallzahlen dar und unterstützt neben dem Offlinebetrieb und der Installierbarkeit die Funktionen Standortzugriff, Kontaktzugriff und Benachrichtigungen.

Bei den implementierten Funktionalitäten wird deutlich, dass Progressive Web Apps mittlerweile durch eine Vielzahl von modernen Schnittstellen des Webs eine Alternative zu Native Apps darstellen können. Dabei ist jedoch eine Schwachstelle von Progressive Web Apps die fehlende Unterstützung einiger Funktionalitäten von iOS. Der Entwicklungsaufwand einer Progressive Web App ist geringer als der einer Native App.

Zusammenfassend zeigt das Ergebnis, dass Progressive Web Apps das Potential besitzen, native Anwendungen zu ersetzen. Aktuell ist die Technologie jedoch nicht ausreichend ausgereift, dass sie jegliche native Anwendung ersetzen kann und muss somit abhängig von den Anforderungen des Anwendungsfalls betrachtet werden.

Inhaltsverzeichnis

Abbildungsverzeichnis	vii
Tabellenverzeichnis	viii
Listings	ix
1 Einleitung	1
1.1 Zielsetzung	1
1.2 Umfeld	2
1.3 Aufbau der Arbeit	2
2 Grundlagen	3
2.1 Native Applikationen	3
2.2 Cross-platform Applikationen	4
2.3 Progressive Web Apps	4
2.4 React	7
2.4.1 Komponenten	9
2.4.2 Hooks	10
2.5 React Native	11
3 Kriterienkatalog zum Vergleich der Technologien	14
3.1 Funktionalitäten	14
3.1.1 Installierbarkeit und Aktualisierungen	14
3.1.2 Offlinebetrieb	15
3.1.3 Standortzugriff	16
3.1.4 Kontaktzugriff	16
3.1.5 Benachrichtigung	17
3.2 Kompatibilität mit verschiedenen Betriebssystemen	17
3.3 Entwicklungsaufwand	18
4 Implementierung einer Progressive Web App und Native App	19
4.1 Einrichten der Entwicklungsumgebung	20
4.2 Installierbarkeit und Aktualisierungen	25
4.3 Offlinebetrieb	27
4.4 Standortzugriff	31

4.5 Kontaktzugriff	32
4.6 Benachrichtigungen	35
5 Ergebnisse und Diskussion	39
5.1 Funktionalitäten	39
5.2 Kompatibilität mit verschiedenen Betriebssystemen	45
5.3 Entwicklungsaufwand	48
5.4 Diskussion	51
6 Fazit und Ausblick	53
6.1 Fazit	53
6.2 Ausblick	53
Literaturverzeichnis	55

Abbildungsverzeichnis

2.1 Architektur React Native	12
4.1 Ordnerstruktur Progressive Web App	21
4.2 Ordnerstruktur React Native App	23
4.3 Exemplarische Darstellung einer Liste mit <i>FlatList</i> in iOS	24
4.4 Speichermanagementstrategien	28
4.5 Erfragung des Standortzugriffs im iOS Browser	31
4.6 Ausschnitt eines Sequenzdiagramms des Web Push Prinzips	35
5.1 Darstellung der installieren Apps	40
5.2 Darstellung der bestimmten Koordinaten	43
5.3 Benutzeroberfläche der Contact Picker Application Programming Interface (API) in Chrome	44
5.4 Vergleich Benachrichtigungen PWA (oben) und React Native App (unten) . .	45
5.5 Vergleich Implementierung React PWA und React Native App	51

Tabellenverzeichnis

5.1 Kompatibilität der Progressive Web App mit verschiedenen Browsern auf ei- nem Android Gerät	46
5.2 Kompatibilität der Progressive Web App mit verschiedenen Browsern auf ei- nem iOS Gerät	46
5.3 Entwicklungsaufwand der beiden Anwendungen	48
5.4 Nutzwertanalyse der beiden implementierten Anwendungen	52
5.5 Nutzwertanalyse der beiden Technologien im Allgemeinen	52

Listings

2.1 Schlichtes Beispiel der index.js einer React Applikation	7
2.2 Nutzung von JSX	8
2.3 Beispiel der Nutzung von useState	10
2.4 Beispiel für Nutzung der useEffect-Hook	11
4.1 Fertiges App Manifest der PWA	25
4.2 Zugriff auf Kontakte	33

Abkürzungsverzeichnis

APK Android Package

API Application Programming Interface

APNs Apple Push Notification service

AVD Android Virtual Device

CDN Continous Delivery Network

DOM Document Object Model

FCM Firebase Cloud Messaging

GPS Global Positioning System

HTTPS Hypertext Transfer Protocol Secure

IDE Integrated development environment

ipa iOS App Store Package

JSON JavaScript Object Notation

OS Operating System

PWA Progressive Web App

SPA Single-Page Application

SDK Software Development Kit

UI User Interface

URL Uniform Resource Locator

Kapitel 1

Einleitung

Egal ob Spotify, TikTok oder Twitter - mobilen Anwendungen sind aus unserem Lebensalltag nicht mehr wegzudenken. Dafür sprechen auch die stetig zunehmende Anzahl an App Downloads, welche 2020 bei 218 Billionen liegt [1].

Bei der Entwicklung dieser Anwendungen gibt es jedoch immer wieder auftretende Problematiken. Beispielsweise die fehlende Kompatibilität einer plattformspezifischen App mit verschiedenen Betriebssystemen und der daraus folgende erhöhte Entwicklungsaufwand. Außerdem muss der Nutzer erst überzeugt werden, die Anwendung zu installieren, bevor er sie überhaupt nutzen kann.

Ein aufstrebendes Konzept, dass sich die Universalität des Webs zu nutzen macht, ist das der Progressive Web Apps (PWAs). Diese besonderen Anwendungen sind über das Web erreichbar und sollen die Freiheiten einer Browseranwendung mit den Vorzügen einer nativen Applikation vereinen. 2015 von dem Chrome Entwickler Alex Russel und dem Designer Frances Berriman als solche benannt, bieten PWAs immer mehr Möglichkeiten an [2]. Dabei spielt auch die Weiterentwicklung von Webschnittstellen eine bedeutsame Rolle. Mit Technologien wie den Service Worker und dem App Manifest ermöglichen PWAs Funktionalitäten, die bislang nativen Anwendungen vorenthalten waren. Dazu gehören beispielsweise Push Benachrichtigungen, Standortzugriff, Offlinebetrieb und Installierbarkeit.

Gerade in der aktuellen Zeit, in der laut einer Statistik von *StatCounter* 4,32 Billionen Menschen das Internet von mobilen Endgeräten aus nutzen, bietet sich somit eine enorme Chance für die Zukunft von mobilen Anwendungen und des Webs [3]. Daher ist der Vergleich von PWAs mit einer Native App relevant und wird in der vorliegenden Arbeit untersucht.

1.1 Zielsetzung

Im Rahmen dieser Arbeit soll untersucht werden, welche Vor- und Nachteile das Entwickeln einer PWA gegenüber einer nativen App bezüglich Funktionalität, Kompatibilität

mit verschiedenen Betriebssystemen und Entwicklungsaufwand bietet. Um beide Verfahren vergleichen zu können, wird für die Entwicklung der Native App das Framework React Native verwendet, da dies - ähnlich zu PWAs - plattformunabhängig ist. Diese Besonderheit wurde bisher noch nicht betrachtet. Außerdem sollen dabei auch die aktuellen Grenzen von PWAs aufgezeigt und somit die Frage beantwortet werden, ob PWAs Native Apps zum jetzigen Zeitpunkt ersetzen können.

1.2 Umfeld

Das Thema der Arbeit wird in Zusammenarbeit mit dem IT-Consulting Unternehmen OPITZ CONSULTING bearbeitet. Die Betreuung findet durch Senior Consultant Michael Müller statt. Das Unternehmen entwickelt Lösungen für seine Kunden in den Bereichen Applications, Analytics, Infrastructure und Integration. Da sich die Software- und Webentwicklungsbranche rasant verändert und weiterentwickelt, ist es für die Firma unerlässlich, neue Technologien und Innovationen zu erkennen. Daraufhin kann Wissen in diesen Bereichen aufgebaut werden, um Lösungen mit diese Technologien potenziellen Kunden anzubieten. Besonders interessant sind hierbei diejenigen Innovationen, die mit geringem Aufwand eine Vielzahl an Vorteilen mit sich bringen.

1.3 Aufbau der Arbeit

Zuerst soll auf die Grundlagen von mobilen Anwendungen und PWAs sowie der JavaScript-Bibliothek React und dem Framework React Native eingegangen werden. Im dritten Kapitel wird ein Kriterienkatalog festgelegt, anhand dessen die implementierten Anwendungen später beurteilt werden. Dieser soll die Funktionalität, die Kompatibilität mit verschiedenen Betriebssystemen und den Entwicklungsaufwand in Abhängigkeit der verwendeten Technologie berücksichtigen. Im Anschluss wird die Programmierung der zwei Applikationen vorgestellt und auf spezielle Vorgehensweisen eingegangen. Der Zweck der Apps ist es, die aktuellen COVID-19 Fallzahlen der offiziellen Datenbank des Robert-Koch-Instituts darzustellen und weiterführende Funktionalitäten wie Installation, Offlinebetrieb, Standortzugriff, Kontaktzugriff und Benachrichtigungen zu unterstützen. Zuletzt sollen die zwei Anwendungen anhand des Kriterienkatalogs verglichen und bewertet werden.

Kapitel 2

Grundlagen

In diesem Kapitel werden die für diese Arbeit notwendigen Grundlagen aufgezeigt. Diese sollen für ein einheitliches Verständnis des Themas sorgen und unterschiedliche Vorstellungen von Fachbegriffen angleichen.

Generell lassen sich mobile Anwendungen in drei Kategorien einteilen: Native, Hybrid und Web Applikationen. Im Folgenden wird einerseits genauer auf Native und Hybride Applikationen und anderseits auf eine spezielle Form von Web Apps, genannt PWAs, eingegangen.

2.1 Native Applikationen

Als native Applikationen bezeichnet man Anwendungen, die plattformspezifisch – ergo speziell für ein Operating System (OS) – implementiert werden. Das wird dadurch ermöglicht, dass sie mit dem Software Development Kit (SDK) des Plattformherstellers entwickelt werden und somit kompletten Zugriff auf jegliche Funktionalitäten der Geräte besitzen. Außerdem entsteht durch Verwendung der plattformspezifischen UI-Komponenten¹, sogenannten Views, eine einheitliche Benutzerschnittstelle, die sich in allen Anwendungen des Betriebssystems widerspiegelt. Um solche Apps mit ihren SDKs zu entwickeln, wird die zur SDK zugehörige Programmiersprache verwendet. Die bekanntesten Programmiersprachen sind Java oder Kotlin für Android Geräte, Objective-C und Swift für iOS. [4]. Eine aktuelle Statistik von *Statcounter* macht deutlich, dass die am meisten verbreiteten Betriebssysteme Android und iOS sind, weswegen im Verlauf dieser Arbeit ausschließlich auf ebendiese eingegangen wird [5].

Um Applikationen im jeweiligen App-Store veröffentlichen zu können, müssen diese eindeutig identifizierbar sein. Das erfolgt durch den Prozess der sogenannten *Signierung*, der ebenfalls plattformabhängig durchgeführt wird.

Eine App kann nach der Signierung und Überprüfung nur durch eine Aktualisierung verändert (z. B. Update oder Bugfix) werden. Deshalb folgt auf eine Aktualisierung auch

¹User Interface Komponenten, z. Dt. Benutzeroberflächenkomponenten

eine erneute Überprüfung und Veröffentlichung der App.

Ferner können Android Apps auch in ihrer Rohfassung auf Android Geräten installiert werden. Hierfür muss der Entwicklermodus aktiviert und das Android Package (APK)² auf das Endgerät geladen werden. Das Äquivalent in iOS ist die iOS App Store Package (ipa)-Datei.

2.2 Cross-platform Applikationen

Unter cross-platform (z. Dt. plattformunabhängig) Applikationen versteht man Anwendungen, die auf einer Code-Basis aufbauen und zur Laufzeit zu mehreren Anwendungen für unterschiedliche Endgeräte kompiliert werden. Der Vorteil solcher Applikationen ist, dass sie, obwohl sie auf demselben Code basieren, sich komplett den plattformspezifischen Stil anpassen. Somit verringern sich auch die Entwicklungskosten, die bei Implementierung von jeweils einer Anwendung pro Betriebssystem anfallen würden. Klar abzugrenzen sind plattformunabhängige Anwendungen von hybriden Anwendungen. Denn das Endprodukt bei letzterem ist eine Web Applikation, die sich durch eine WebView in einem nativen Container dem Kontext (z. B. Betriebssystem, Auflösung) des Geräts, in dem sie aufgerufen wird, anpasst [6].

Gängige Frameworks zur plattformunabhängigen Entwicklung sind Electron, Ionic und React Native [7]. Diese stellen meist eine begrenzte Anzahl an vorprogrammierten UI-Komponenten und Funktionalitäten zur Verfügung. Es gibt außerdem eine Vielzahl von Community Lösungen, die für wiederkehrende Anforderungen Lösungen bieten.

2.3 Progressive Web Apps

PWAs sind in erster Linie Web Applikationen, also Anwendungen, die mit den üblichen Web Technologien wie HTML, CSS und ECMAScript (JavaScript) implementiert sind. Laut Mozilla Developer Network zeichnen sie sich dadurch aus, dass zusätzlich folgende technische Voraussetzungen erfüllt sind: Sie müssen auf einer sicheren Verbindung aufbauen, eine oder mehrere Service Worker besitzen und über ein App Manifest verfügen [8]. Ersteres wird gewährleistet durch eine Hypertext Transfer Protocol Secure (HTTPS) Verbindung, während Service Worker JavaScript-Skripte sind, die im Hintergrund auf dem Client und unabhängig von der Anwendung selbst ausgeführt werden [9]. Sie sind die wichtigste Komponente hinter den meisten Funktionalitäten, die eine PWA bietet, weswegen im Laufe dieses Kapitels genauer auf ebendiese eingegangen wird. Die letzte benötigte Komponente ist das App Manifest. Dabei handelt es sich um eine JavaScript

²Plattformspezifische Sammlung von Dateien, die wie Quellcode und Ressourcen für die Installation und Nutzung der App beinhaltet

Object Notation (JSON)³-Datei, in der Einzelheiten zur Anwendung und deren Installation angegeben werden [10]. Da es sich dabei, trotz der zusätzlichen nativen Funktionalitäten, um eine Web Applikation handelt, lassen sich PWA über eine Uniform Resource Locator (URL) im Browser aufrufen und sind somit zunächst unabhängig von dem Gerät, auf dem sie aufgerufen werden. Üblicherweise werden PWA als Single-Page Application (SPA) entwickelt. Das bedeutet, dass die Anwendung aus einem einzigen HTML-Dokument besteht und Inhalte dynamisch geladen werden. Ein Vorteil davon ist, dass sich dadurch auch die Anzahl der Anfragen des Clients an den Server verringern, da nicht nach jeder Anfrage die Seite komplett neu angefordert werden muss. Der Nachteil von SPA ist, dass im Gegensatz zu klassischen Webanwendungen Funktionalitäten wie Deep-Linking und die Navigation durch die Browser-Steuerelemente eigenständig implementiert werden müssen [11]. Bislang gibt es keinen offiziell definierten Web Standard für diese spezielle Art von Webanwendung.

Ein bedeutsames Konzept hinter PWAs ist das „Progressive Enhancement“. Dieses verlangt, dass PWAs auf allen Endgeräten grundlegend funktionieren sollen, und schrittweise – falls der Browser und das Gerät dies unterstützt – in ihrer Funktionalität erweitert werden können [12]. Die Prüfung, ob der Browser die Anforderungen erfüllt, findet durch das Window-Objekt statt, dass das aktuelle Browser Fenster repräsentiert. Dieses enthält unter anderem das *navigator*-Property, dass Informationen über den Browser besitzt und die Unterstützung der Service Worker API⁴ signalisiert. Dadurch kann wiederum geprüft werden, ob andere Schnittstellen z. B. die *Geolocation*⁵ in dem aktuellen Browser unterstützt werden.

Zur Evaluierung von PWA bietet sich *Lighthouse* an, ein vorinstallierte DevTools Erweiterung im Google Chrome Browser. Dort wird per Mausklick ein Testbericht zur aktuellen Anwendung erstellt, in der unter anderem auch Installierbarkeit und PWA-Optimierung geprüft werden. Dabei orientiert es sich an den, von Web.dev⁶ definierten Kernfunktionalitäten von PWA, die lauten: „starts fast, stays fast“, „works in any browser“, „responsive to any screen size“, „provides a custom offline page“, und „is installable“. Ferner werden noch Kriterien angegeben, die die PWA optimal machen, beispielsweise „can be discovered through search“ oder „provides context for permission requests“[12].

Service Worker API

Wie bereits erwähnt, ermöglicht der Service Worker Funktionalitäten, die PWAs zur Konkurrenz von nativen Anwendungen machen. Dennoch kann die Service Worker API

³Programmiersprachen-unabhängiges Datenformat

⁴z.Dt. Programmierschnittstelle.

⁵Eine Schnittstelle zur Bestimmung und Beobachtung des aktuellen Standorts des Nutzers.

⁶Eine von Google Developers unterstützte Webseite mit einer Vielzahl von Artikeln zu Webentwicklungstechnologien.

grundsätzlich in jeder Anwendung implementiert werden und ist nicht auf PWA beschränkt, da es sich bei einem Service Worker um einen Web Worker handelt. Er fungiert dabei als eine Art Proxy Server, der zwischen der Anwendung, dem Browser und dem Netzwerk platziert ist und somit Zugriff auf Netzwerkanfragen besitzt [10]. Er besitzt seinen eigenen Thread und durch seine Position hat der Service Worker keinen Zugriff auf das Document Object Model (DOM). Die Voraussetzung für die Nutzung eines Service Workers ist, dass der verwendete Browser diese unterstützt und die Verbindung über HTTPS läuft [13]. Letzteres wird damit begründet, dass Sicherheitsprobleme wie Man-in-the-middle-Angriffe, die durch die Stellung des Service Workers als Proxy begünstigt werden, vermieden werden können.

Ein Service Worker besitzt die Lifecycle-Events *install*, *activate* und unter anderem das funktionale Event *fetch*. Diese ermöglichen es, auf bestimmten Ereignissen zur Laufzeit einer Anwendung zu reagieren. Der Service Worker wird üblicherweise beim ersten Aufrufen der Webseite registriert. Anschließend ist der Service Worker jederzeit verfügbar und läuft selbstständig im Hintergrund der Anwendung. Während des *install*-Events sollten diejenigen Dateien in den Cachespeicher aufgenommen werden, die sich im Laufe der Anwendung nicht ändern. Das betrifft beispielsweise Styling Sheets, das App Manifest und Bild Dateien [14]. Außerdem kann auch die index.html, die als Eintrittspunkt von SPAs dient, im Cache abgelegt werden.

Das *active*-Event wird dafür genutzt, veraltete Cachespeicherinhalte zu bereinigen und vorherige Service Worker Registrierungen zu entfernen.

Um auf Netzwerkanfragen zu reagieren, gibt es das *fetch*-Event. Hier ist es möglich, angeforderte Ressourcen ebenfalls im Cache abzulegen. Dafür gibt es verschiedene Strategien, zwischen denen je nach Anwendungsfall der Applikation abgewägt werden kann. Gängige Strategien sind der *Cache First*- oder der *Cache then network*-Ansatz [15]. Durch weitere funktionale Events wie *push*, *notificationclick* und *sync* und die Nutzung von alten und neuen Programmierschnittstellen ermöglicht der Service Worker das moderne, native-ähnliche Web. Jede dieser APIs sollten ebenfalls im Sinne von *progressive enhancement* eingebunden werden. Im Laufe dieser Arbeit wird genauer auf die Cache API, die Notification API, die Push API sowie die Geolocation API eingegangen.

Für alle der genannten Programmierschnittstellen ist es nötig, die Erlaubnis des Nutzers zu erfragen. Zugriff und Verwaltung aller erteilten Erlaubnisse bietet die Permissions API. Diese verfügt über ein Permission Registry, das Permissions für Schnittstellen wie *geolocation*, *bluetooth*, *speaker* und *device-info* enthält. Laut der W3C⁷ Spezifikation der Permissions API gibt es drei Status der Erlaubnis: *granted*, *denied* und *prompt*. Außerdem wird aufgrund des hohen Einflusses von Permissions unterschieden zwischen Funktionalitäten, die in unsicheren Kontexten und jenen, die nur in sicheren Kontexten (HTTPS) verwendet werden können [16].

⁷World Wide Web Consortium, eine Organisation, welche Spezifikationen für das Web veröffentlicht.

Mittlerweile bietet Google, unter anderem zur vereinfachten Implementierung und Verwaltung von Service Workern, das Tool *Workbox* an. Es handelt sich dabei um eine Bibliothek, die die gängigsten Funktionalitäten von Service Workern zur Verfügung stellt, wodurch wiederkehrende Prozesse eliminiert werden.

2.4 React

React ist eine von Facebook entwickelte, open-source JavaScript-Bibliothek, die seit 2013 publiziert ist. Sie zeichnet sich dadurch aus, dass sie in erster Linie zum Erstellen von User Interfaces entwickelt wurde. Durch die ReactDOM-Bibliothek, wird die Anwendung um das Rendern dieser Benutzeroberflächen erweitert. Daran lässt sich auch erklären, warum React im Gegensatz zu Vue oder Angular kein Framework ist. Werden Projekte mit diesen Frameworks erstellt, erhält der Entwickler eine Vielzahl von eingebauten Werkzeugen zum entwickeln von skalierbaren, komplexen Webanwendungen. Im Gegensatz dazu ist React als User Interface (UI)-Bibliothek leichtgewichtig und ermöglicht individuelle Erweiterung zur Anpassung an die Anforderungen des Projekts [17]. Dennoch können auch vielschichtige Anwendungen, z. B. Facebook oder Paypal mit React umgesetzt werden.

```

1 import React from 'react';
2 import ReactDOM from 'reactdom';
3
4 var element = React.createElement(
5   'h1',
6   { className: 'greeting' },
7   'Hello world.'
8 );
9 ReactDOM.render(element, document.getElementById('root'));

```

Listing 2.1: Schlichtes Beispiel der index.js einer React Applikation

Die einfachste Möglichkeit React in einem Projekt zu nutzen ist, es über eine Continous Delivery Network (CDN) einzubinden. Dabei ist es ein Anliegen der Entwickler, dass nur so wenig React genutzt werden kann, wie benötigt. Ferner ist es möglich React über Package-Manager wie npm in ein Projekt zu importieren oder durch Toolchains⁸ wie Create-React-App über die Kommandozeile eine SPA zu erstellen [18]. Im Codeausschnitt 2.1 ist ein Beispiel zu sehen, wie eine React Anwendung in ihrer kleinsten Form aussehen kann. Zuerst müssen die Module React und ReactDOM importiert werden.

⁸Eine Sammlung von Werkzeugen, die zum unkomplizierten Aufsetzen eines Produkts dienen.

Dann wird per Aufruf der `React.createElement(...)`-Funktion mit den Übergabeparametern HTML-Element, Attribute und Inhalt ein React Element erstellt. Zuletzt wird in Zeile 9 die `ReactDOM.render(...)`-Funktion genutzt, um das erstellte Element einem anderen Element zuzuordnen und damit eine Hierarchie zu erzeugen. In diesem Fall konkret dem HTML-Element mit der `id root`. Die `render`-Funktionen kann als weiteren Übergabeparameter die *Properties* eines Elements oder einer Komponente enthalten, worauf im Laufe des Kapitels genauer eingegangen wird.

Eines der Argumente zur Nutzung eines Programmiergerüsts wie React ist dessen Implementierung des Virtual Document Object Model. Dieses baut auf dem normalen DOM auf und ermöglicht es, dass nur diejenigen UI-Elemente neu gerendert werden, deren Daten sich verändert haben. Des Weiteren bietet sich durch die Abkapselung in Komponenten ein hohes Maß an Wiederverwendbarkeit. Außerdem ist React wie bereits erwähnt leichtgewichtig, da es sich bei der Hauptbibliothek nur um die Implementierung der wichtigsten Bestandteile handelt. Weitere Funktionalitäten wie der React Router zur Programmierung von der Navigation in einer SPA oder anderen Bibliotheken können nach Bedarf importiert werden. Ebenso gibt es UI-Bibliotheken wie MaterialUI oder PrimeReact für React, die häufig implementierte Komponenten im modernen Design anbieten. Generell lassen sich PWA jedoch mit jedem Framework oder auch mit einer einfachen Vanilla-JavaScript Implementierung verwirklichen.

Kritik erlangt React vor allem wegen des Vorwurfs, dass es gegen das Entwurfsprinzip der Trennung der Verantwortlichkeiten (Separation of Concerns) verstößt. Dies wird in der Webentwicklung so umgesetzt, dass verschiedene Technologien wie HTML, CSS und JavaScript jeweils in eigenen Dateien modelliert oder programmiert werden. Im Gegensatz dazu steht jedoch Reacts Syntax Erweiterung JSX. Diese ermöglicht die drei genannten Technologien innerhalb von JavaScript zu entwickeln. Ein Beispiel dafür ist in 2.2 zu sehen. Wichtig ist hier jedoch, dass die Zeile 8 auch in JavaScript geschrieben werden kann, da es sich hierbei letztendlich um den Aufruf der `React.createElement(component, props, ...children)`-Funktion handelt [19].

```

1 import React from 'react';
2 import ReactDOM from 'reactdom';
3
4 const Example = (props) => {
5     const greeting = 'Hello world'
6
7     return (
8         <h1>{{ greeting }}</h1>
9     )
10 }
```

Listing 2.2: Nutzung von JSX

Außerdem gibt es einige Änderungen, die sich durch diese Art zu programmieren ergeben. Beispielsweise kann in JSX auf das Semikolon am Ende einer Zeile verzichtet werden und die CSS-Klasse `class` nennt sich `className`. Letzteres ist eines von mehreren Syntaxänderungen bei JSX. Dem zugrunde, dass jeder JSX-Code in JavaScript-Code umkompiliert wird. Das Schlüsselwort `class` ist dabei in JavaScript ein reserviertes Wort für Klassen und nicht für eine CSS-Klasse. Ein weiteres Beispiel ist `htmlFor` statt `for`. Durch diese Syntax bietet React eine inklusive Dateistruktur. Die Entwickler begründen das damit, dass es hier im Gegensatz zu anderen JS-Frameworks, in denen es pro Komponente jeweils eine getrennte HTML-, CSS- und JavaScript-Datei gibt, lediglich um eine Trennung der Technologien, nicht aber der Verantwortlichkeiten, handelt. Diese Syntax hingegen verbinden die Render Logik enger mit den Benutzeroberfläche und führt bei einer korrekten Aufteilung in Komponenten zu einer starken Kohäsion. Das bietet dem Entwicklern mehr Übersichtlichkeit und Verständnis für Zusammengehörigkeit. Auf der anderen Seite werden komplexe Komponenten jedoch durch diese inklusive Struktur schnell unübersichtlich, weshalb es sinnvoll ist, eine Aufteilung in Unterkomponenten angelehnt an deren Funktionalität vorzunehmen. Wichtig ist hierbei auch eine organisierte Ordnerstruktur aufrecht zu erhalten, damit die Anwendung wartbar bleibt.

Im Folgenden wird genauer auf einige Grundkonzepte von React eingegangen, wobei React-spezifische Begriffe bewusst nicht übersetzt werden.

2.4.1 Komponenten

Wenn Teile des Codes abgekapselt und wiederverwendet werden sollen, wird eine Komponente erstellt, die meist in einer Datei mit demselben Namen implementiert wird. React unterscheidet dabei zwischen zustandslosen und klassenbasierten Komponenten. Ersteres bezeichnete ehemals Komponenten, die nur zur Darstellung von zusammengehörigen UI-Elementen genutzt wird. Sie sind schlank, wiederverwendbar und leicht zu warten. Klassenbasierte Komponenten hingegen basieren auf herkömmlichen ES6⁹-Klassen und bieten sogenannte States, die lokale Daten einer Komponente verwalten, und einen Lifecycle. Die Hooks API bietet jedoch seit React 16.8 ein neues Konzept an, um States in sogenannten funktionalen Komponenten zu organisieren und somit die Vorteile von zustandslosen und klassenbasierten Komponenten zu vereinen. Im nächsten Kapitel wird genauer auf die Funktionsweise von Hooks eingegangen.

⁹ECMAScript 6, eine 2015 veröffentlichte Version von ECMAScript.

Zur Kommunikation und Datenaustausch zwischen Eltern- und Kind-Komponenten werden *Properties* und Callback-Methoden genutzt. Ein *Property* ist eine Art Übergabeparameter, die von der Eltern- an die Kindkomponente weitergegeben wird. Das Kind kann mit diesen Daten die eigene *ui* und Funktionalität entwickeln oder wiederum der eigene Kindkomponente geben. Wichtig ist dabei, dass übergebene Informationen lediglich gelesen, nicht aber verändert werden sollen. Die Kommunikation nach außen funktioniert über Events. In der Kindkomponente wird dafür eine Callback-Methode aufgerufen und somit signalisiert, dass die Eltern denjenigen Code ausführen sollen, der als Reaktion auf die Veränderung in der Kindkomponente dient.

2.4.2 Hooks

In Version 16.8 erfolgte die Einführung von *Hooks*. Diesen bieten die Möglichkeit, *States* und andere React Funktionalitäten zu implementieren, ohne eine JavaScript Klasse deklarieren zu müssen. Dadurch vereinfachen sich Komponenten, die ehemals von der *Component*-Klasse abgeleitet wurden, um in Konstruktoren Zustände definieren zu können. Der Rückgabewert dieser funktionalen Komponenten ist die UI-Deklaration selbst [20].

Es gibt unterschiedliche Arten von *Hooks*, die gängigsten sind der *useState*- und der *useEffect-Hook*. Sie werden erstmalig direkt nach dem Rendern der Komponente ausgeführt. Außerdem gibt es die Möglichkeit, eigene *Hooks* zu definieren.

Die *useState-Hook* dient zur Deklaration eines lokalen *States*. In Zeile 4 des folgenden Beispiels 2.3 wird der *State counter* in der Komponente Example initialisiert. Dieser erhält den Standardwert 0 und verfügt über einen Getter – hier genannt *counter* – und den Setter, genannt *setCounter()*, über diese Funktion der Zustand geändert werden kann.

```

1 import React, { useState } from "react";
2
3 const Example = (props) => {
4     const [counter, setCounter] = useState(0);
5 }
```

Listing 2.3: Beispiel der Nutzung von *useState*

Der *useEffect-Hook* ersetzt die Lifecycle-Methoden *componentDidUpdate*, *componentDidMount* und *componentWillUnmount* der klassenbasierten Komponenten. Es bietet sich deshalb an, Ressourcenanfragen hier zu behandeln. Das Beispiel in 2.4 zeigt einen einfachen *Effect* welcher die frühere *componentDidMount*-Funktion und *componentWillUnmount()*-Funktion ersetzt. Der erste Übergabeparameter des *useEffect*-Aufrufs ist

eine Funktion, die ausgeführt werden soll und der Zweite ein Array, genannt *dependency array*. Wenn das Array leer ist, bedeutet das, dass der *Effect* nur einmal nach dem Rendern der Komponente ausgeführt werden soll. Durch jedes Element, das diesem Array hinzugefügt wird, startet erneut diejenige Funktion, die als erster Übergabeparameter übergeben wurde.

```

1 import React, { useEffect } from "react";
2
3 const Example = (props) => {
4     const [counter, setCounter] = useState(0);
5
6     useEffect(() => {
7         console.log('Component did render!');
8     }, []);
9
10    useEffect(() => {
11        console.log('Counter was updated!');
12    }, [counter]);
13
14    // setCounter() is called somewhere in the component
15}

```

Listing 2.4: Beispiel für Nutzung der useEffect-Hook

Durch das *dependency array* haben *Effects* den Vorteil, dass sie sehr fallspezifisch auf Änderungen der Daten reagieren können und somit umso mehr den dynamischen Gedanken der SPA realisieren.

2.5 React Native

React Native ist eines der meist genutzten Frameworks zur Entwicklung von Native Apps [5]. Es basiert auf React und ist ebenfalls von Facebook entwickelt und veröffentlicht worden. Bekannte Apps, die auf React Native basieren, sind selbstverständlich die Facebook und Instagram App, aber auch die Unterkunftsbuchungsapp Airbnb oder die Lieferdienstapp UberEats [21].

Wie bereits im Kapitel Grundlagen erklärt, werden native Applikationen üblicherweise mit der dafür vorgesehenen Programmiersprache implementiert. React Native nutzt die Möglichkeit, mit JavaScript die Schnittstellen und *Views* von Native Anwendungen anzusprechen und das Ganze somit als nativen Code zu rendern. Zur Laufzeit werden

dabei durch die *Core Components* von React Native die jeweils korrespondierenden *Native Components* von Android, iOS und anderen Betriebssystemen angesprochen [22]. Das wird ermöglicht durch die Architektur von React Native, die in drei Komponenten unterteilt ist: eine Laufzeitumgebung für JavaScript auf der Zielplattform, eine Bridge und ein Native Module. In der 2.1 ist dieses Aufteilung abgebildet. Ersteres bietet in

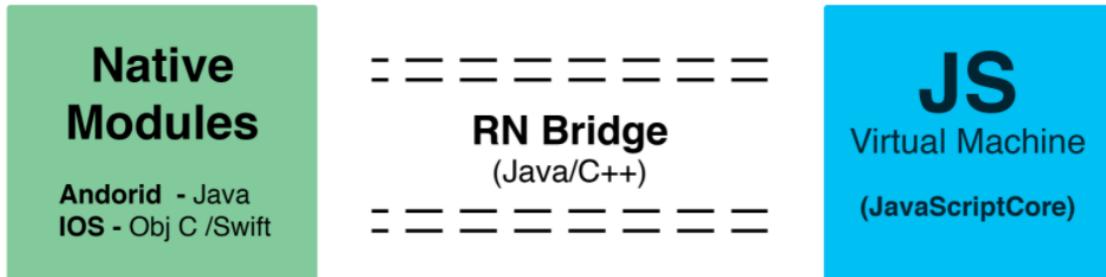


Abbildung 2.1: Architektur React Native

Quelle: [23]

iOS JavaScriptCore und in Android eine Lösung, die von React Native zur Verfügung gestellt wird. Dabei gibt es in React Native plattformübergreifende Abstraktionen wie *View*, *Image* oder *NetInfo*, die korrespondierende Native Komponenten besitzen. Die Bridge ist für die Kommunikation zwischen Native Module und JS-Umgebung verantwortlich. Sie erhält von ersterem Events wie Toucheingaben oder Netzwerkanfragen und vom Native Module Anweisungen für das UI oder APIs. Der Austausch findet dabei asynchron über JSON statt. Der letzte Teil der Architektur ist das Native Module. Dies ist die Anwendung, die der Nutzer verwendet [24]. Es handelt sich somit bei Apps, welche mit React Native programmiert wurden, nicht um hybride Apps, da sie nicht nur in einem Native Container gerendert werden, sondern tatsächlich auf native Schnittstellen zugreifen. Die Entwicklung solcher Apps ist dennoch plattformunabhängig, da die Implementierung in JavaScript erfolgt und die Umwandlung erst zur Laufzeit geschieht. React Native Applikationen lassen sich dadurch schwierig in die bereits erklärten Kategorien von mobilen Anwendungen einordnen.

Zur Nutzung von React Native Apps benötigt das Gerät mindestens iOS 11.0 oder Android 5.0 [20]. Genau wie andere Native Apps können sie in den plattform-spezifischen App Store veröffentlicht und dort vom Nutzer heruntergeladen, installiert und aktualisiert werden. Dabei ist der benötigte Signing-Prozess derselbe wie bei Native Applikationen.

Zur Implementierung von Komponenten und Funktionalitäten in React Native gibt es mehrere Optionen. Die erste Option ist das Nutzen der bereits erwähnten *Core Components*. Weiterhin kann auf Community Lösungen zugegriffen werden, die Brücken zu nativen Schnittstellen implementieren. Sollten die *Core Components* nicht ausreichen, ist das meist die effizienteste Wahl, da viele Funktionalitäten bereits zufriedenstellend und

stabil entwickelt und veröffentlicht wurden. Unter <https://reactnative.directory/> sind diese Bibliotheken auffindbar und durch einen errechneten Directory Score nach mehreren Kriterien wie Beliebtheitskurve, Sterne auf GitHub oder Anzahl der Downloads bewertet. Die letzte Option ist es, Native Modules selbstständig zu programmieren. Das bedeutet, dass eine *Bridge* zwischen React Native und der nativen Komponente implementiert werden soll, wodurch diese zur Laufzeit mit JavaScript angesprochen werden kann. Somit ist es generell möglich alles was eine native Anwendung kann, auch in React Native umzusetzen.

Kapitel 3

Kriterienkatalog zum Vergleich der Technologien

Nach der Erläuterung der Grundlagen, soll nun auf die Kriterien eingegangen werden, nach denen die Technologien im Verlauf der Arbeit verglichen werden. Diese sind Funktionalität, Kompatibilität und Entwicklungsaufwand der Anwendung.

Als Reaktion auf die Coronakrise soll eine App programmiert werden, die zur Darstellung und Durchsuchung der aktuellen COVID-19 Fallzahlen dient. Dabei soll diese installierbar sein und auch bei schlechter oder fehlender Internetverbindung funktionieren. Außerdem soll der Nutzer eine Filterung der Daten auf Basis seines aktuellen Standorts oder auf Basis der Adresse eines seiner Kontakte durchführen können. Zuletzt soll es dem Nutzer möglich sein, Benachrichtigungen zur aktuellen Lage der Fallzahlen zu erhalten.

Ziel der Anwendung ist es, schnell auf die Coronakrise zu reagieren und möglichst vielen Nutzern mit verschiedenen Betriebssystemen ihre Funktionalitäten anbieten zu können. Daher erfolgt keine Gewichtung der Kriterien, da alle Kriterien von gleicher Bedeutung sind.

3.1 Funktionalitäten

Applikationen werden entwickelt, um Nutzern einen Mehrwert in ihrem Alltag zu bieten. Je mehr Funktionalität eine Anwendung unterstützt, desto mehr Nutzen kann sie bieten. Egal ob in der Web- oder Appentwicklung, wenn die Anwendung keinen Mehrwert bietet, wird sie nicht verwendet und wird dadurch vom Markt verdrängt. Im Folgenden werden diejenigen Funktionalitäten von Apps vorgestellt, die für den Vergleich ausgewählt wurden.

3.1.1 Installierbarkeit und Aktualisierungen

Beschreibung

Die Apps, die der Nutzer oft verwendet, sollten schnell erreichbar sein. Eine Installation wird daher bevorzugt und stellt eine grundlegende Funktion von mobilen Applikationen dar. Für das verwendete Gerät bedeutet das Installieren, dass es Kapazität seines Speichers der Applikation zur Verfügung stellen muss. Das ist dahingehend erwähnenswert, dass einige Nutzer auf das Installieren der App verzichten müssen, wenn ihnen unzureichend Speicherplatz zur Verfügung steht.

Falls nun Aktualisierungen des Herstellers verfügbar sind, möchte der Nutzer diese auch erhalten und durchführen können. Dadurch können beispielsweise vorherige Fehler in der App bereinigt oder neue Funktionalität ermöglicht werden. Für die Entwickler ist es ebenfalls wichtig, dass die Anwendung stets in der aktuellsten Version installiert ist. Denn wenn es Änderungen der Schnittstellen gibt, stellt das eine besondere Problematik für die auf dem Endgerät installierte App dar.

Kriterium

Mit diesem Kriterium soll geprüft werden, ob die Anwendung installiert werden kann und inwieweit das den Speicherplatz des Endgeräts belastet.

3.1.2 Offlinebetrieb

Beschreibung

Eine wichtige Funktion von mobilen Anwendungen ist der Offlinebetrieb. Das bedeutet, dass die Anwendung auch ohne oder unter schlechter Internetverbindung verwenden werden kann. Dabei muss natürlich unterschieden werden zwischen Anwendungen, die generell keinen Zugriff auf das Internet benötigen und jenen, die ihre Funktionalität im Offlinebetrieb einschränken müssen. Meistens bieten Applikationen eine Mischung aus beiden Optionen an. Um aus technischer Sicht eine Unabhängigkeit von der Netzwerkverbindung zu schaffen, müssen Daten lokal im Speicher des Geräts abgelegt werden. Das ermöglicht dem Nutzer, seine Daten jederzeit verfügbar zu haben und somit Abhängigkeiten von äußeren Umständen zu minimieren. Vorteilhaft für den Nutzer ist dabei auch, wenn die Prozesse, die im Offlinebetrieb angestoßen wurden, gehalten werden, bis das Gerät wieder eine stabile Internetverbindung hat. Auf diesem Wege wird gewährleistet, dass jegliche Aktivitäten erfolgreich durchgeführt werden. Nachteil des Abspeicherns der Daten ist zum einen erhöhten Speicherverbrauch und bei einer großen Anzahl auch eine Steigerung des Batterieverbrauchs.

Kriterium

Dieses Kriterium soll prüfen, ob die Anwendung auch mit fehlender Netzwerkverbindung lauffähig ist.

3.1.3 Standortzugriff

Beschreibung

Um mobile Anwendungen auf die eigenen Bedürfnisse anzupassen, ist der Standortzugriff eine Option. Dabei greift die Anwendung unter anderem durch das Global Positioning System (GPS) auf den Standort des Nutzers zu und kann diesen weiterverarbeiten, um standortabhängige Informationen darzustellen. Der Zugriff bezieht sie dabei auf den aktuellen Standort sowie auf Bewegungen des Nutzers. Meist muss bereits während des Installationsprozesses der Applikation die Zustimmung des Nutzers für das Nutzen von standortbezogenen Inhalten eingeholt werden. Wird dies genehmigt, ist es der Anwendung auch möglich, im Hintergrund auf GPS-Daten zuzugreifen.

Gängige Anwendungsfälle sind die Abfrage des Standorts für Wetterinformationen, Navigation oder die Anzeige von Dienstleistungen in der Nähe des aktuellen Standorts.

Kriterium

Das Kriterium hier ist, ob die Anwendungen Zugriff auf den Standort des Nutzers besitzen. Außerdem ist von Bedeutung, ob und wie die Informationen mit der App weiterverarbeitet werden können und wie akkurat die Daten sind.

3.1.4 Kontaktzugriff

Beschreibung

Eine weiterer Zugriff auf native Schnittstellen eines mobilen Endgeräts bieten die Kontakte. Diese sind meist auf dem Gerät oder dem verknüpften Google oder Apple Konto hinterlegt. Sie können neben Bild, Name und Telefonnummer auch andere Kontaktdaten wie Anschriften oder E-Mail-Adressen beinhalten.

In vielen Messaging Apps wie WhatsApp, Telegram oder KIK werden diese Daten verarbeitet, um dem Nutzer die Möglichkeit zu geben, seine Kontakte abgesehen von SMS oder Anrufen zu kontaktieren.

Kriterium

Das Kriterium prüft, ob die Anwendung Zugriff auf die Kontaktliste des Nutzer erhalten kann, um Kontaktdaten weiterzuverarbeiten.

3.1.5 Benachrichtigung

Beschreibung

Neben dem Kontaktzugriff spielen auch Benachrichtigungen eine große Rolle bei der Interaktion mit Nutzern. Sie lassen sich aus technischer Sicht in zwei Kategorien unterteilen: nicht-persistente und persistent Benachrichtigungen. Sie unterscheiden sich darin, dass erstere nur erscheinen, wenn die Anwendung in dem Moment in Benutzung ist, wohingegen das Empfangen von persistenten Benachrichtigungen jederzeit erfolgen kann. Letztere können entweder von der Anwendung selbst oder von einem Server ausgelöst werden und werden deshalb auch als Push Benachrichtigungen bezeichnet. Beide Fälle bieten den Vorteil, dass die Nutzer immer wieder auf die Anwendung aufmerksam werden und sie somit motiviert sind, diese öfters zu nutzen. Wichtig ist hierbei die Zahl der Benachrichtigungen pro Tag oder Woche zu planen. Denn wenn der Nutzer zu viele Nachrichten erhält, kann das als störend empfunden werden und zur Deaktivierung der Benachrichtigungen oder im schlimmsten Fall zur Deinstallation der Anwendung führen. Push Benachrichtigung können kurze Informationen durch Texte, Bilder oder Buttons beinhalten. Letzteres ist besonders wichtig, da der Nutzer darüber aus dem Menü heraus mit der Anwendung interagieren und auch auf diese weitergeleitet werden kann. Außerdem könnten sie Benachrichtigungstöne auslösen oder dem Icon der Anwendung ein Badge¹ anheften.

Konkrete Beispiele für Inhalte von Benachrichtigungen sind aktuelle WhatsApp Nachrichten, neue Freundschaftsanfragen auf Facebook oder neue Suchergebnisse für die gespeicherte Ebay-Kleinanzeigen-Suche.

Kriterium

Die Anwendung muss fähig sein, Benachrichtigungen von einem Server zu erhalten. Dies sollte auch funktionieren, ohne dass die Anwendung im Vorder- oder Hintergrund geöffnet ist.

3.2 Kompatibilität mit verschiedenen Betriebssystemen

Beschreibung

Durch die Vielzahl von Betriebssystemen und Geräten ist es aufwendig, Anwendungen zu implementieren, die überall im selben Maße lauffähig sind. Projektleiter und Entwickler müssen deshalb im Voraus genau abwägen, auf welchen Systemen die Applikation verfügbar sein soll. Hierbei kommt es bei Webanwendungen nicht nur auf das

¹Badges sind kleine Anzeigen am rechten oberen Rand des App Icons. An ihnen erkennt man die Anzahl an ungeöffneten Benachrichtigungen.

Betriebssystem und dessen Version, sondern auch auf den verwendeten Browser und dessen Version an. Bei Android Geräten ist das vorinstallierte Browser Chrome und bei iOS Safari. Auf welchen Geräten und Browern die Anwendungen unterstützt werden sollen, wirkt sich außerdem direkt auf den Entwicklungsaufwand aus.

Kriterium Das Kriterium besteht darin, dass die Anwendungen auf verschiedenen Betriebssystemen und Browern funktionieren müssen. Dabei ist ausschlaggebend, wie viele der bereits definierten Funktionalitäten auf dem Browser oder Betriebssystem verfügbar sind. Je mehr Betriebssystem- und Browerversionen unterstützt werden, desto mehr ist dieses Kriterium erfüllt.

3.3 Entwicklungsaufwand

Beschreibung

Bei der Implementierung von Anwendungen, egal ob für das Web oder mobile Endgeräten, lassen sich aus dem Entwicklungsaufwand deren Kosten berechnen. Dabei fließen außerdem der Entwicklungszeitraum, Fähigkeiten des Teams und der Funktionsumfang in die Kalkulation ein. Wichtig ist, wie bereits angemerkt, auch die Kompatibilität mit verschiedenen Endgeräten. Den je mehr Geräte unterstützt werden sollen, desto aufwendiger und somit kostenlastiger ist meist die Implementierung.

Wird jedoch zur Entwicklung ein plattformunabhängiger Ansatz gewählt, kann das bereits den Entwicklungsaufwand reduzieren. Gerade wenn schnell reagiert werden soll – zum Beispiel in der COVID-19-Krise – ist es praktisch, wenn die Entwicklung möglichst effizient und die Anwendung auf vielen Geräten lauffähig ist, um ihre Nutzer effektiv zu unterstützen. PWAs sind generell plattformübergreifend, da sie im Browser geöffnet werden. Bei nativen Apps wird die Plattformunabhängigkeit durch Frameworks wie React Native ermöglicht. Dennoch besteht bei beiden Ansätzen die Möglichkeit, dass zur Unterstützung von bestimmten, geforderten Betriebssystemen(-versionen) oder Browern ein zusätzlicher Entwicklungsaufwand entsteht.

Kriterium

Auf Basis der bereits genannten Kriterien, soll hierbei gemessen werden, wie lange für die Implementierung dieser benötigt wird. Dabei soll einerseits die Zeit zur Einarbeitung in die verschiedenen Technologien betrachtet werden sowie die tatsächliche Zeit, die zum programmieren der Funktionalität benötigt wird.

Kapitel 4

Implementierung einer Progressive Web App und Native App

Die Daten für die zu implementierenden Applikationen werden von der offiziellen REST¹ API des Robert Koch-Instituts bezogen. Sie werden täglich um Mitternacht prozessiert und sind in den frühen Morgenstunden in der API verfügbar. Konkret wird die Datenbank „RKI Corona Landkreise“ genutzt, die die Zahlen gruppiert nach Landkreisen im GeoJSON und JSON-Format zur Verfügung stellt. Sie besitzt insgesamt 2.117.948 Datensätze und als CSV-Datei eine Größe von 308 MB[25].

Für die Darstellung der Fallzahlen wurden hierbei folgende, auf den Landkreis bezogene, Attribute ausgewählt:

- Art des Landkreises (*BEZ*)
- Name (*GEN*)
- Fälle (*cases*)
- Fälle der letzten 7 Tage pro 100.000 Einwohner (*cases7_per_100k*)
- Fallzahlen pro 100.000 Einwohner (*cases_per_100k*)
- ID (*AdmUnitId*)
- Zeitpunkt der letzten Aktualisierung der Daten (*last_update*)

In der Anwendung werden die Fallzahlen zum Zweck der Übersichtlichkeit auf zwei Nachkommastellen gerundet.

Für die Implementierung der zu vergleichenden Anwendungen wurden die open-source Bibliotheken React und React Native verwendet. React dient bei der PWA als Unterstützung zur Entwicklung einer SPA, wohingegen React Native eine komplettes Framework zur plattformunabhängigen Entwicklung von nativen Anwendungen anbietet. Ein Vergleich mit diesen Bibliotheken bietet sich einerseits insofern an, dass deren Grundlage dieselbe Programmierweise darstellt, nämlich das deklarative und komponentenbasierte

¹Ein gängiges Datenformat des Datenaustauschs im Web.

Konzept von React. Anderseits sind durch die Verwendung von React Native beide Technologien plattformunabhängig und bilden somit eine solide Grundlage für den Vergleich. Diese Eigenheit der Implementierung unterscheidet sich insofern grundlegend von der tatsächlichen nativen Entwicklung, als dass diese nicht betriebssystemspezifisch stattfindet. Dafür muss die dafür vorgesehenen Programmiersprachen und ein Integrated development environment (IDE) wie Android Studio mit Kotlin und Java genutzt werden. Trotz dieser Besonderheit wird in der vorliegenden Arbeit eine React Native App als Native App mit einer PWA verglichen, da sich generell mit React Native alle Funktionalitäten programmieren lassen, die Native Apps anbieten.

Das Besondere bei der Entwicklung einer Native App mit React Native ist, dass dies plattformunabhängig geschieht. Wie bereits in Kapitel 2 geschildert, handelt es sich dabei um einen Prozess, der von einem JavaScript Thread über *Bridges* mit einem Native Thread kommuniziert und somit zur Laufzeit native Schnittstellen und UI-Elemente aufruft. Dadurch kann mit JavaScript Code geschrieben werden, der zur Laufzeit sowohl Android als auch iOS Komponenten anspricht. Um eine Funktionalität mit JavaScript implementieren zu können, muss jedoch entweder bereits eine Community Lösung einer *Bridge* für diese gewollte Funktion existieren oder diese selbst entwickelt werden.

Beide Applikationen sind im Quelltext-Editor Visual Studio Code und bis auf Ausnahmen bei der Native App mit JavaScript programmiert.

4.1 Einrichten der Entwicklungsumgebung

Progressive Web App

Zum Aufsetzen der Umgebung wurde die bereits genannte Create-React-App-Toolchain verwendet. Diese wird durch npm² mit dem Befehl `npx create-react-app <Name der Anwendung>` erstellt. Hierfür wird mindestens Node³ 10.16 und npm 5.6 benötigt. Nachteile der Verwendung einer Toolchain sind jedoch die Abhängigkeit von den dadurch integrierten Bibliotheken und das daraus resultierende Sicherheitsrisiko. Auf der anderen Seite ermöglicht die Toolchain einen schnellen Einstieg zur Entwicklung einer SPA. Dies liegt vor allem an den vorgefertigten Konfigurationen zur Vereinfachung der Programmierung wie *Babel*⁴, *ESLint*⁵ und *Webpack*⁶. Create-React-App bietet außerdem eine Reihe an Templates, die das Aufsetzen einer zum Projekt passenden Entwicklungsumgebung eben-

²ehemals Node Package Manager, Anwendung zur Installation von Node.js Packages.

³Node.js ist eine lizenzzfreie und plattformunabhängige JavaScript Entwicklungsumgebung

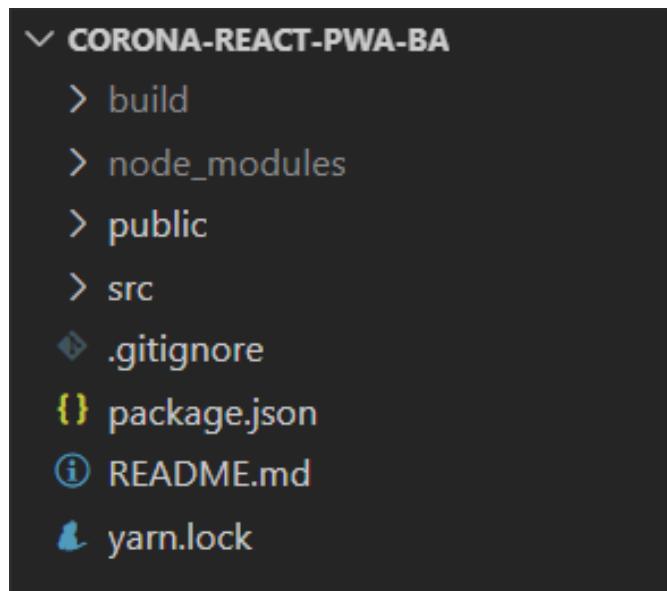
⁴Compiler, der mit ECMAScript 2015 geschriebenen Code rückwärtskompatibel macht durch dafür vorgesehene Polyfills.

⁵Codeanalysetool, welches den Entwickler auf problematische Codestellen aufmerksam macht.

⁶Modulbündler, der aus vielen einzelnen Dateien, ein oder mehrere große Dateien generiert. Dies erhöht die Performance der Anwendung.

falls erleichtern sollen. Beispielsweise das PWA Template, das Werkzeuge wie *Workbox*⁷ und einen vorimplementierten Service Worker mitinstalliert. Darauf wird in dieser Anwendung verzichtet, um die Funktionsweise des Service Workers selbst programmieren zu können und diese dadurch besser zu erfassen.

Die Abbildung 4.1 zeigt die Ordnerstruktur nach dem Aufsetzen der Anwendung per Toolchain.



Quelle: Eigene Darstellung

Abbildung 4.1: Ordnerstruktur Progressive Web App

Im *public*-Ordner befinden sich diejenigen Dateien, die an den Client versendet werden. Konkret sind das ausgewählte Bilddateien, das App Manifest und die *index.html*-Datei. *Webpack* nutzt die *index.js* als Eintrittspunkt und generiert beim Ausführen der *ReactDOM.render(element, document.getElementById('root'))*-Funktion einen Komponentenbaum auf dem *div*-Element mit der *id root*. Die *index.js* befindet sich im *src*-Ordner. Auf ebendieser Ebene befindet sich außerdem die *yarn.lock*, in der alle Abhängigkeiten der Anwendung gelistet sind, sowie die *package.json*, die über Metadaten verfügt.

Mit dem Befehl „*npm run start*“ wird eines der Skripte von Create-React-App zum Starten von *Webpack* ausgeführt. Dadurch ist die Anwendung direkt nach dem Aufsetzen unter <http://localhost:3000/> lokal aufrufbar. Obwohl localhost kein HTTPS besitzt, sind die Funktionalitäten von PWAs möglich, da es sich hierbei lediglich um die Entwicklungsumgebung handelt.

Zum Erhalten der Daten wird mit der *Fetch*-Funktion eine *GET*-Anfrage an die API der Corona-Datenbank des Robert Koch-Instituts versendet und die Antwort in ein

⁷Von Google entwickelte Sammlung von JavaScript Bibliotheken zur vereinfachten Implementierung von Service Workern.

Array gespeichert. Dieses wird in der *render*-Funktion mithilfe der *Array*-Funktion *map* durchlaufen. Die Daten werden somit im UI übersichtlich dargestellt.

Generell müssen Webanwendungen auf einen Webserver veröffentlicht werden, um im World Wide Web aufrufbar zu sein. Speziell bei PWAs ist hierbei bedeutsam, dass dieser Webserver eine verschlüsselte Verbindung mittels HTTPS aufbaut, da dies eine technische Voraussetzung für PWAs ist. Zum Hosting der Webapplikation wird deshalb der Hostingservice Netlify verwendet, welcher eine HTTPS Verbindung bereitstellt. Durch den Befehl „npm run build“ kann eine minimierte Version der Anwendung erzeugt werden, die eine optimierte Endfassung für das Deployment der App bietet. Nach erfolgreichem Hochladen des Build Ergebnisses ist die PWA über den Link <https://corona-react-pwa-ba.netlify.app/> verfügbar.

React Native App

Um eine React Native Anwendung zu programmieren, gibt es zwei Ansätze: *managed workflow* und *bare workflow*. Ersteres wird durch das zusätzliche Expo Framework verwaltet, das einen schnellen Einstieg in die plattformunabhängige Entwicklung ermöglicht. Diese läuft mit der zugehörigen Expo Go iOS oder Android App, indem die programmierte App in Echtzeit über einen Packager gerendert wird. Somit kann die App sofort ohne weiteres auf dem jeweiligen Endgerät unabhängig vom Betriebssystem getestet werden.

Der Nachteil dieses Ansatzes ist, dass mit Expo kein nativer Code für die genannten Betriebssysteme programmiert werden kann und dadurch die Funktionalitäten auf die von Expo angebotenen APIs beschränkt sind. Für die Funktionen, die in dieser Arbeit betrachtet werden, würde Expo ausreichen, jedoch soll an dieser Stelle kein weiteres Framework genutzt werden. Deshalb wird der *bare workflow* verwendet. Doch auch dieser Ansatz besitzt einen Nachteil. Denn trotz der eigentlich plattformunabhängigen Entwicklungsweise besteht keine Möglichkeit, mit Windows oder Linux eine iOS App zu programmieren. Laut der Dokumentation von React Native wird hierfür macOS benötigt.

Im Folgenden wird daher die Implementierung einer Android App mit React Native beschrieben und die Umsetzung derselben Funktionalitäten auf iOS nur skizziert. Wichtig ist dabei, dass aus dem vorliegenden Code trotzdem eine iOS App erzeugt werden könnte. Die gleiche Funktionalität wie bei der Android App ist jedoch meist nicht ohne weitere Anpassungen wie das Zulassen von Berichtigungen über Apples IDE XCode verfügbar.

Zur Erstellung eines React Native Projekts wird der Befehl *npx react-native init <Name der Anwendung>* verwendet, der ebenfalls von npm zur Verfügung gestellt wird. Außerdem müssen diejenigen Einstellungen in der Entwicklungsumgebung vorgenommen werden, die auch zur Programmierung von nativen Android Apps benötigt werden.

Das betrifft beispielsweise das Installieren von Android Studio und der Android APK. Nach dem Ausführen des Befehls liegt die Ordnerstruktur vor, die in der Abbildung 4.2 dargestellt ist.

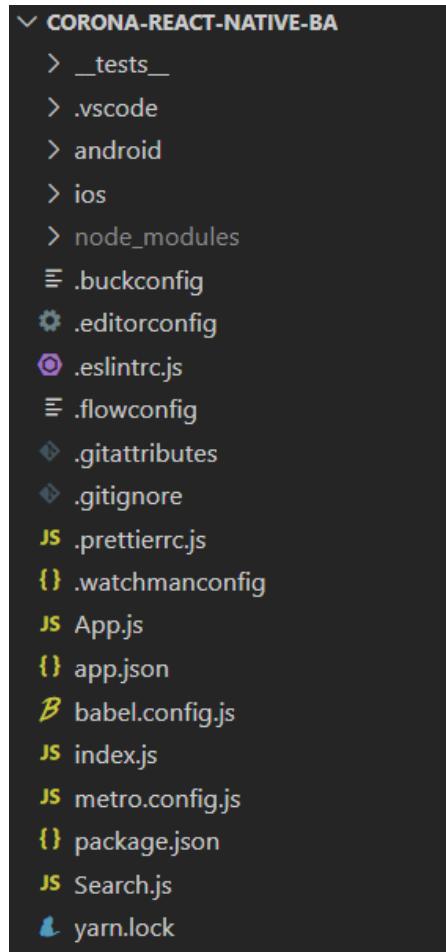
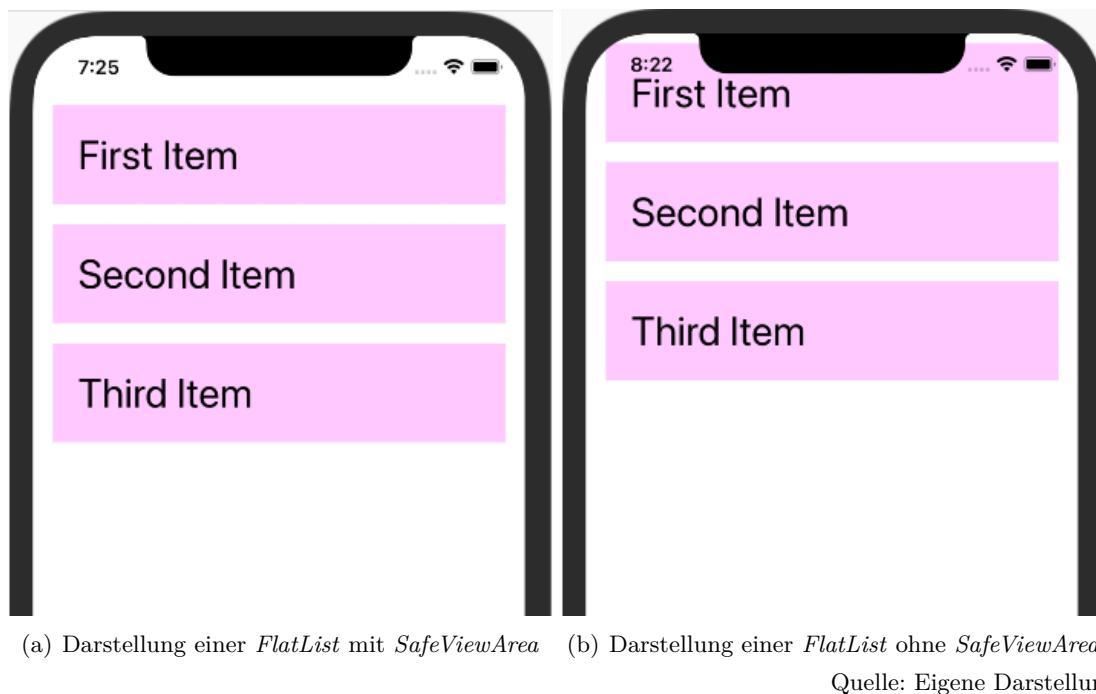


Abbildung 4.2: Ordnerstruktur React Native App

Quelle: Eigene Darstellung

Die Ordner *ios* und *android* beinhalten plattformspezifischen Code wie die *Info.plist*- und die *AndroidManifest.xml*-Datei, die Metainformationen über die Anwendung bereitstellen. Auf derselben Ebene befindet sich die *App.js*-Datei, in der die Anwendung mit JavaScript implementiert wird.

Um die Corona-Daten in der React Native App darzustellen, wird ebenfalls die *Fetch API* genutzt. Diese verwendet die *RCTNetworking*-Klasse als Bridge zu den nativen Modulen in Android und iOS. In Android ist das der *OkHttpClient* und in iOS *XHTTP*. Die Daten werden nach dem Abrufen durch die *fetch*-Funktion in einer *FlatList* angezeigt. Der Vorteil der *FlatList* ist, dass es nur diejenigen Elemente rendert, die im sichtbaren Teil des Bildschirms sind. Dadurch werden Ressourcen gespart und das Scrollen der Liste erscheint für den Nutzer flüssiger. Im Hintergrund spricht React Native mit der

Abbildung 4.3: Exemplarische Darstellung einer Liste mit *FlatList* in iOS

FlatList-Komponente die Komponente *ListView* in Android und in iOS an. Die *FlatList* wird umschlossen von einer *SafeAreaView*-Komponente. Dabei handelt es sich um eine iOS-spezifische Komponente, die empfohlen wird, um die UI in iOS zu verbessern. In der Abbildung 4.3 wird deutlich, welche Auswirkung das Fehlen dieser Komponente für die Benutzeroberfläche hat.

Damit auf gewisse Funktionalitäten des Geräts zuzugreifen werden kann, benötigten Anwendungen teilweise die Erlaubnis des Nutzers. Android unterscheidet dabei zwischen „normal“ und „dangerous“ *Permissions*. Beispiele für letzteres sind die *android.permission.ACCESS_FINE_LOCATION* oder *android.permission.READ_CONTACTS*. Alle anderen *Permissions* werden aus dem *AndroidManifest.xml* generiert.

Bei iOS Apps werden die benötigten Berechtigungen der App in XCode festgelegt und beim Installationsprozess abgefragt. Nur wenn der Nutzer diese Berechtigungen akzeptiert, kann die Anwendung installiert werden. Daher wird in der App für die Nutzung von Funktionalitäten wie Standort- oder Kontaktzugriff keine zusätzliche Erlaubnis des Nutzers benötigt.

4.2 Installierbarkeit und Aktualisierungen

Das Ziel dieser Funktionalität ist es, dass die Anwendung auf das mobile Endgerät des Nutzers installiert werden kann. Dadurch sollen er schnell auf die App zugreifen können.

Progressive Web App

Um eine Webanwendung installierbar zu machen, benötigt sie ein App Manifest und *HTTPS*. Für chromiumbasierte Browser ist außerdem ein Service Worker notwendig. Dort bedeutet das Installieren in neueren Browerversversionen, dass eine sogenannte WebAPK aus dem App Manifest generiert wird. Somit ist diese dann auch unter allen Apps im App Drawer und den Einstellungen aufgelistet. Dennoch ist sie Teil von Chrome und verliert beispielsweise ihre Daten, wenn die Cache-Daten von Chrome geleert werden. Bei iOS Geräten bedeutet das Installieren lediglich ein Lesezeichen für die Anwendung zum Startbildschirm des mobilen Endgeräts hinzuzufügen, wie es auch mit herkömmlichen Webseiten und -anwendungen möglich ist. Deshalb erscheint die App weder im App Drawer, noch unter den Einstellungen.

Wie bereits im Kapitel 2 angemerkt, handelt es sich bei dem App Manifest um eine JSON-Datei, die Informationen über die Anwendung bereitstellt. Um das Installieren zu ermöglichen, muss dieses die in Abbildung 4.1 gezeigten Attribute beinhalten. Ersteres gibt an, wie die Anwendung dargestellt werden soll. Damit sie einer mobilen App gleicht, wird hier der Wert *standalone* oder *fullscreen* empfohlen. Dadurch verschwindet die für Webanwendungen übliche URL-Leiste. Durch die Angabe einer *background_color* erhält die Anwendung eine Hintergrundfarbe beim Starten der App. Diese wird angezeigt, bis Stylingsheets oder Hintergrundbilder der PWA geladen sind. Das *icons*-Attribut besitzt ein Array aus Objekten, welche die Dateipfade der Icons in verschiedenen Größen angeben. Die *start_url* gibt die relativen URL an, die beim Starten der installierten Anwendung geöffnet werden soll [14].

Das Manifest wird per „<link>“-Tag in die index.html eingebunden und dessen Informationen sind danach auch im Application-Tab der Google Developer Tools einsehbar [26].

```

1  {
2      "name" : "COVID-19 Fallzahlen" ,
3      "icons" : [
4          {
5              "src" : "favicon.ico" ,
6              "type" : "image/x-icon" ,
7              "sizes" : "64x64" ,

```

```

8   },
9   {
10    "src": "logo192.png",
11    "type": "image/png",
12    "sizes": "192x192"
13  },
14  {
15    "src": "logo512.png",
16    "type": "image/png",
17    "sizes": "512x512"
18  }
19 ],
20 "start_url": ".",
21 "display": "standalone",
22 "background_color": "#ffffffff"
23 }
```

Listing 4.1: Fertiges App Manifest der PWA

Wichtig ist dabei, dass das Manifest als experimentell markiert ist, da es nicht von allen mobilen Browsern und Betriebssystemen komplett unterstützt wird. Aktuell betrifft das Safari, welches das Installieren aus Browsern, die auf iOS statt mit ihrem eigenen HTML-Renderer mit WebKit⁸ laufen (Chrome, Firefox und Opera), nicht unterstützt [27]. Denn jede iOS App muss laut Punkt 2.5.6 der Apple App Store Review Guidelines als Engine WebKit nutzen [28]. Wenn der Nutzer nun die Anwendung im Browser aufruft, kann er sie auf browserabhängige Weise installieren und somit mit einem Klick vom Startbildschirm oder App-Drawer aus öffnen. Beim Aufrufen der Seite, öffnet sich im Chrome Browser auf Android außerdem eine Installationsaufforderung für die App am unteren Bildschirmrand. Diese zusätzliche Funktionalität muss aktuell für den Safari Browser explizit implementiert werden.

Für das Ermöglichen von Aktualisierungen der PWA muss keine weitere Programmierung vorgenommen werden. Sobald ein neues Deployment der Anwendung vorgenommen wird, ist sie automatisch auf dem neusten Stand, selbst wenn die PWA installiert ist.

React Native App

Zur Ermöglichung der Installation einer React Native App bedarf es keiner konkreten Implementierung. Es muss nur – wie bei nativen Anwendungen – die betriebssystemabhängige Signierung durchgeführt werden. Durch diesen Prozess erhält die Anwendung

⁸HTML-Renderer von Safari.

einen *Release key* und einen *Upload Key*, durch die sie eindeutig identifizierbar ist und auf die alle zukünftigen Aktualisierungen referenziert werden. Die genauen Schritte sind in den offiziellen Dokumentationen von Apple und Android nachzulesen. Zusätzliche Schritte für iOS, die eine Besonderheit von React Native sind, sind das Verbieten von HTTP Anfragen und das Umstellen der App im *Release*-Schema. Danach kann die resultierende APK-Datei in den Google Play Store oder die ipa-Datei in den Apple App Store hochgeladen werden. Dort werden sie getestet und verifiziert. Zuletzt kann der Nutzer sie im jeweiligen Store suchen und herunterladen.

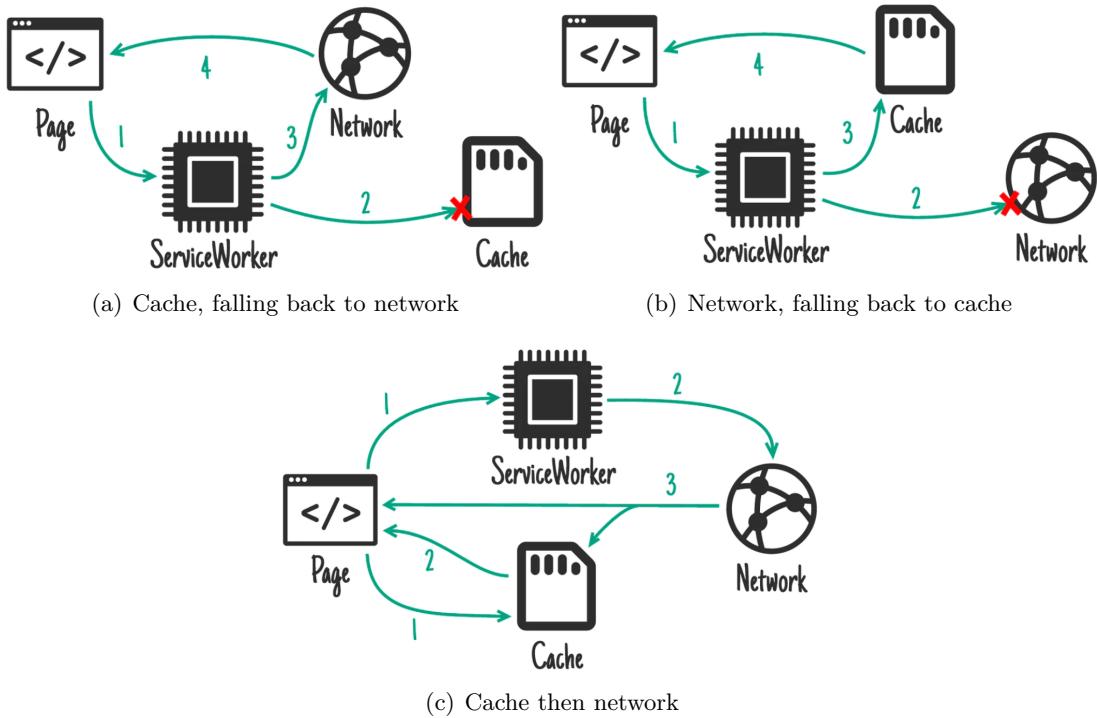
Ähnlich wie bei der PWA gibt es bei Android Apps die *AndroidManifest.xml*-Datei, welche Informationen zur Installation der App bereitstellt, wie das App Icon, Berechtigungen der App oder die mindestens benötigte Android Version. Bereits bei der Installation durch den Play Store wird dabei nach Berechtigungen für die Nutzung der App gefragt, die aus der *AndroidManifest.xml* ausgelesen werden. Bei iOS Apps ist das die *Info.plist*-Datei.

4.3 Offlinebetrieb

Die Anwendung soll auch ohne Internetverbindung aufrufbar, mit Daten gefüllt und die Funktion des Durchsuchens nutzbar sein.

Progressive Web App

Generell benötigten Nutzer eine Internetverbindung, um auf Webanwendungen zuzugreifen. Durch moderne Webschnittstellen ist mittlerweile auch den Offlinebetrieb ermöglichen. Hierfür wird der Service Worker genutzt. Er kann auf eine statische, individuelle Seite weiterleiten, welche die standardmäßig angezeigte Seite bei fehlender Internetverbindung ersetzt. Alternativ können aber durch den Service Worker auch Daten abgespeichert und im Offlinebetrieb angezeigt werden. Hierfür gibt es neben der bereits seit 2015 bestehenden *IndexedDB* auch die Cache API, mit der Daten als Request-Response-Paar abgespeichert werden können. Aus einer Vielzahl von clientseitigen Speichermechanismen im Web (*LocalStorage*, *SessionStorage* oder *Cookies*) eignen sich speziell diese beiden zur Nutzung in einer PWA. Das ist damit zu begründen, dass sie persistent sein können und asynchron ablaufen [29]. Letzteres verhindert, dass der Main Thread der Webanwendung blockiert wird und die Nutzer unter Umständen lange Warte- /Ladezeiten bei der Bedienung der Anwendung erfahren. Gerade die Cache API wird jedoch im Zusammenhang mit PWAs besonders empfohlen, weil diese im Gegensatz zur *IndexedDB* auch statische Ressourcen speichert [30]. Die Speichergröße von beiden Speichern ist abhängig von dem Browser, in dem die Webanwendung aufgerufen wird.



Quelle: [31]

Abbildung 4.4: Speichermanagementstrategien

Durch das Zwischenspeichern der Daten wird ermöglicht, dass die PWA dem Nutzer selbst ohne oder mit schlechter Internetverbindung Rückmeldung in Form einer Benutzeroberfläche zur Verfügung stellt. Das kann entweder eine individuelle Seite oder dieselbe Seite mit veralteten Daten sein. Ferner ist das Caching der Daten zum Offlinebetrieb der Anwendung nicht nur hilfreich, wenn keine Internetverbindung besteht, sondern reduziert auch allgemein die Anfragen an den Server.

Eine Anwendung kann mehrere Caches besitzen, die zur Unterscheidung benannt werden. In der programmierten PWA werden an zwei Stellen Daten im *cache-v1*-Cache gespeichert. Zuerst geschieht dies beim Installieren des Service Workers im *install*-Lifecycle-Event. Hier werden statische Ressourcen wie das Manifest und Bilddateien in den Cache aufgenommen.

Die zweite Stelle ist während des *fetch*-Events zur Speicherung von Netzwerkkabfragen. Dabei ist nicht nur die Speicherung von Bedeutung, sondern auch wie dieses Cache-Daten in der PWA genutzt werden. Hierfür definiert Jake Archibald mehrere Strategien, wovon die „Cache, falling back to network“-, „Network, falling back to cache“- und „Cache then network“-Strategie ausgewählt und für diesen Anwendungsfall untersucht wurden [31].

Ersteres bedeutet, dass der Service Worker beim Ausführen der Anwendungen zuerst prüft, ob die Daten im Cache-Speicher verfügbar sind, und wenn dies nicht zutrifft, eine Netzwerkanfrage stellt. Die Anwendung ist dadurch performant, der Nachteil ist jedoch, dass die Daten beim Aufrufen der App nicht auf dem aktuellsten Stand sind.

Zweiteres verfolgt einen gegenteiligen Ansatz, bei dem nur im Falle einer fehlenden Netzwerkverbindung die Daten aus dem Cache verwendet werden. Auf diese Weise sind die Daten stets aktuell, jedoch erhöht sich auch die Anzahl an Netzwerkanfragen und somit der Verbrauch von Datenvolumen, gerade bei dieser großen Datenmenge, die vom Robert Koch-Institut zur Verfügung gestellt wird. Außerdem muss der Nutzer warten, bis die Netzwerkanfrage fehlschlägt, bis die Cache-Daten genutzt werden. Dies kann für Nutzer mit geringer Internetverbindung länger dauern und dadurch die Nutzererfahrung verschlechtern.

Der „Cache then network“-Ansatz stellt gleichzeitig eine Anfrage an den Cache und eine an das Netzwerk. Sobald die Daten aus dem Cache geladen sind, werden sie in der Anwendung dargestellt. In jedem Fall werden die Daten aber auch nach Antwort der Netzwerkanfrage im Cache aktualisiert [31]. Diese Variante benötigt eine Implementierung im Service Worker und in der Anwendung selbst, um zwei Anfragen auszulösen.

Die Entscheidung fiel zuletzt auf den „Cache then network“-Ansatz, mit einer zusätzlichen Besonderheit. Denn dadurch, dass bei der Anwendung einerseits die Aktualität der Daten eine große Rolle spielt, anderseits die Daten sich nicht über den Tag hinweg ändern, ist eine ständige zweite Anfrage an das Netzwerk redundant. Deshalb ist in der Anwendung eine Abfrage implementiert, welche prüft, ob das Datum der Robert Koch-Institut-Daten mit dem heutigen übereinstimmt. Nur wenn dies nicht zutrifft, wird eine Anfrage an das Netzwerk gestellt und daraufhin die Daten im Cache aktualisiert.

Durch das Zwischenspeichern der Daten wird ermöglicht, dass die PWA dem Nutzer selbst ohne oder mit schlechter Internetverbindung Rückmeldung in Form einer Benutzeroberfläche zur Verfügung stellt. Das kann entweder eine individuelle Seite oder dieselbe Seite mit veralteten Daten sein. In der implementierten PWA ist letzteres der Fall und über eine Zeitstempel erfährt der Nutzer die Aktualität der angezeigten Daten. Hierbei ist wichtig anzumerken, dass der Offlinebetrieb nur dann gewährleistet ist, wenn der Service Worker bereits auf dem Gerät installiert ist. Dies ist der Fall, wenn die Anwendung in einem Browser-Tab geöffnet ist oder auf den Startbildschirm heruntergeladen wurde.

React Native App

Mobile Anwendungen sind im Gegensatz zu PWAs generell offline aufrufbar. Der Grund dafür ist, dass das Rendern der App nicht abhängig von einer Netzwerkverbindung ist, weil sie bereits auf dem Endgerät existiert. Dennoch ist es sinnvoll, auch hier

eine lokale Speicherung der Corona-Daten vorzunehmen, um diese trotz Offlinebetriebs anzeigen zu können.

Seit React Native Version 0.59 sind einige Komponenten als veraltet deklariert. Das betrifft unter anderem *AsyncStorage* und *NetInfo*. Sie wurden jedoch nicht entfernt, sondern lediglich aus der Kernbibliothek react-native exkludiert und in jeweils eigenen Bibliotheken mit eigenen Verantwortlichen verschoben [32]. Diese Bibliotheken sind dem GitHub Repository @react-native-community zu finden und benötigen nach der Installation per npm auch das React Native-spezifische *Linkings*⁹.

Der *AsyncStorage* legt persistent Daten unverschlüsselt in einem lokalen Speicher ab. Der Speicherplatz ist dabei standardmäßig auf 6 MB beschränkt, allerdings kann dieser auch manuell erweitert werden [24]. Die Daten werden bei iOS Geräten im *Application Support*-Verzeichnis abgelegt. In Android wird für die asynchrone Speicherung *SQLite* verwendet.

Die Implementierung der Speicherung erfolgt nach der „Network, falling back to cache“-Strategie. Demnach wird erst abgefragt, ob das Gerät eine aktive Internetverbindung besitzt und wenn dies fehlschlägt, die Daten aus dem *AsyncStorage* verwendet.

Dafür stellt die *NetInfo*-Komponente Informationen zur aktuellen Netzwerkverbindung bereit. Diese wird genutzt um im Falle einer fehlenden Internetverbindung, die Corona-Daten aus dem *AsyncStorage* zu entnehmen.

Zuerst werden jedoch die Daten nach erfolgreicher API-Abfrage im *AsyncStorage* gespeichert. Eine Anforderung dabei ist, dass sie vorher in ein JSON-String konvertiert werden müssen.

Im initialen *useEffect*-Hook wird beim nächsten Aufruf der App geprüft, ob eine Internetverbindung besteht. Wenn dies nicht zutrifft, wird per asynchroner Funktion auf die Daten im *AsyncStorage* zugegriffen. Sobald die Internetverbindung wiederhergestellt ist, werden die Daten per Netzwerkanfrage von der API abgerufen.

Zum Testen des Offlinebetriebs muss nun das Internet des Rechners, auf dem der Emulator geöffnet ist, ausgeschaltet werden. Dadurch wird nachgewiesen, dass die Anwendung trotz fehlender Netzwerkverbindung die im *AsyncStorage*-gespeicherten Corona-Daten anzeigen kann.

⁹Ein Prozess bei dem diejenigen nativen Fähigkeiten, die von einer Bibliothek benötigt werden, der Anwendung hinzugefügt werden. Durch diesen zusätzlichen Schritt verringert sich die Größe der Basis Anwendungen, die eventuell keine nativen Funktionalitäten benötigt.

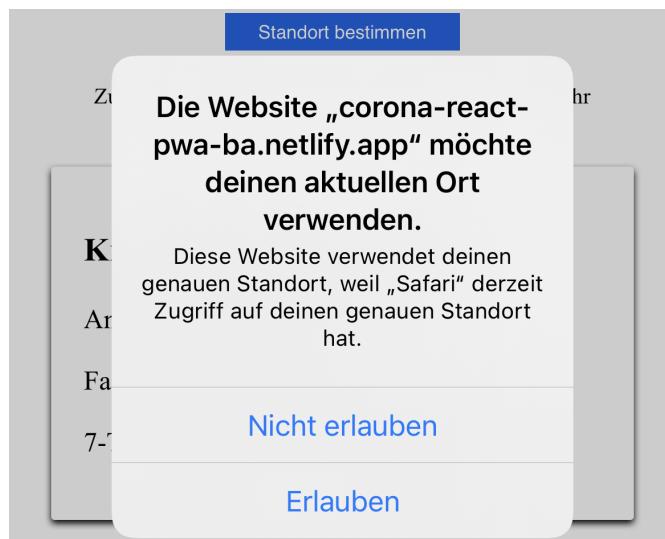
4.4 Standortzugriff

Anhand des Standorts soll der Nutzer die Liste der Landkreise nach dem Landkreis filtern, in dem er sich aktuell befindet.

Progressive Web App

Der Standortzugriff wird im Web über die sogenannte Geolocation API ermöglicht. Die Bestimmung erfolgt dabei unter anderem durch GPS und verschiedenste Netzwerksignale. Sie wird vom *navigator*-Objekt zur Verfügung gestellt und bietet unter anderem Zugriff auf Längen- und Breitengrad, Genauigkeit des bestimmten Standorts und die Höhe über dem Meeresspiegel des Gerätes [33].

Zur Implementierung dieser Funktionalität wird der Nutzer beim Klicken des *Standort bestimmen*-Buttons zuerst, wie in Abbildung 4.5 zu sehen, nach seiner Zustimmung gefragt.



Quelle: Eigene Darstellung

Abbildung 4.5: Erfragung des Standortzugsriffs im iOS Browser

Sobald diese erteilt ist, wird die Methode *Navigator.getCurrentPosition()* aufgerufen und deren Rückgabewert – ein Objekt des Typs *GeolocationPosition* – einer Variable zugewiesen.

Über eine kostenfreie Drittanbieter API wird durch den Prozess des Reverse Geocachings aus den *location.coords.latitude* und *location.coords.longitude* Daten die Aufenthaltsstadt bestimmt. Der Wert des *input*-Feldes wird dann auf den ermittelten Landkreis gesetzt und die Liste der Landkreise automatisch nach dieser gefiltert.

React Native App

Zur Implementierung des Standortzugriffs in React Native wird die Community Lösung *react-native-geolocation-service* verwendet, welche nach Aussage des Erstellers für Android und iOS programmiert ist. Diese greift über eine *Bridge* auf diejenigen Schnittstellen zu, die in Android und in iOS verantwortlich für den Standort sind. In Android wird speziell wegen eines Timeout-Problems nicht auf die übliche *Geocoder*-Klasse zugegriffen, sondern auf die *FusedLocationProviderClient* API des Google Play Services. In iOS spricht die *Bridge* den *CLLocationManager* des *CoreLocation*-Modules an.

Nach dem Installieren der Bibliothek muss für Android das Codefragment `<uses-permission android:name="android.permission.ACCESS_FINE_LOCATION"/>` in der `AndroidManifest.xml` hinzugefügt werden. Das erlaubt der Anwendung auf den Standort zuzugreifen, wenn die App benutzt wird. Da es sich beim Standortzugriff um eine „dangerous“ *Permission* handelt, muss zuerst der Nutzer dennoch erst eine Erlaubnis erteilen. Dies geschieht mit dem Befehl `PermissionAndroid.request()`. Für iOS wird neben dem Aktivieren des Standortzugriffs in XCode keine explizite Erlaubnis benötigt.

Die Abfrage der Erlaubnis erfolgt in der `Search.js` in einem *Effect*, welcher beim Öffnen der Anwendung ausgelöst wird. Nach Zustimmung des Anwenders wird per API-Anfrage an denselben Drittanbieter wie bei der PWA der Name der Stadt aus den Daten über den Längen- und Breitengrad des Standorts des aktuellen Geräts erschlossen. Hierbei ist zu betonen, dass im Normalfall bei nativen Anwendungen nach der Standortbestimmung auf die betriebssystemintegrierte Reverse Geocoding Funktion (beispielsweise bei Android die Google Maps Geocoding API) zurückgegriffen werden könnte. In dieser Arbeit wird jedoch wegen der dadurch anfallenden Kosten darauf verzichtet.

Der logische Aufbau dieser Funktionalität konnte hier von der React PWA übernommen werden, lediglich die Events wurden abgeändert. So nennt sich das *onChange*-Event einer Texteingabe in React nun *onChangeText*-Event in React Native, während das *input*-HTML-Element zum *TextInput*-Tag wird.

4.5 Kontaktzugriff

Für die Covid-19-Fallzahlen App soll es umgesetzt werden, dass der Nutzer die Liste der Landkreise nach der Adresse eines Kontakts filtern kann.

Progressive Web App

Die Kontakte eines mobilen Endgeräts stehen der Webanwendung über das *navigator*-Interface zur Verfügung. Konkret ist es dessen *contacts*-Property, dass in chromiumbasierten Browsern seit Version 80 die Contact Picker API implementiert und im HTTPS

Kontext nutzbar ist. Falls kein HTTPS verhanden ist, existiert das Property nicht im *navigator*.

Mittels der *select*-Funktion kann ein Kontakt auf Basis des Namens ausgewählt werden. Dabei wird bei der Implementierung festgelegt, welche Kontaktdaten bei der Auswahl angezeigt werden. In der PWA soll nun aus der Adresse des Kontakts die Stadt entnommen und dieser Wert in das Suchfeld gesetzt werden. Dadurch wird die Liste der Landkreise nach der Stadt der Adresse des Kontakts gefiltert. Die *address* eines Kontakts besitzt die gleichen Werte wie die des *PaymentAddress*-Interfaces, das aus der Payment Request API bekannt ist. Diese sind beispielsweise *city*, *country* und *region*.

Auch für diesen Eingriff in die Kontaktliste des Nutzers muss explizit eine Erlaubnis erteilt werden. Die Abfrage erfolgt beim Betätigen des Buttons und ist wie beim Standortzugriff eine browserbasierte Abfrage.

Das Implementieren dieser Funktionalität gestaltet sich als problematisch, da die Contact Picker API lediglich auf mobilen Browsern zur Verfügung steht. Deshalb wurde die Funktion vollständig mit dem Android Emulator programmiert. Dafür wird statt durch *http://localhost:3000* mit der IP-Adresse auf die laufende Webanwendung zugegriffen. Doch auch hier gibt es Komplikationen mit der Unterstützung des *contacts*-Attribut im *navigator*. Zuletzt hat lediglich das Deployment der PWA geholfen, auf die Kontakte zuzugreifen. Grund dafür kann sein, dass die Contact Picker API nur mit HTTPS verfügbar ist, jedoch sollte das nicht für *localhost* gelten, wie bei der Implementierung der bisherigen Funktionalitäten. Durch die Bestätigung der Sucheingabe wird in einer Funktion der Code in Abbildung 4.2 ausgeführt.

```

1 const props = [ "name" , "address" ];
2 if ( "contacts" in navigator ) {
3     try {
4         const contact = await navigator.contacts.select( props , {} );
5         onQueryChange( contact [ 0 ].address [ 0 ].addressLine [ 0 ] );
6         // Further processing
7     } catch ( ex ) {
8         // Handle any errors here .
9     }
10 } else {
11     // Handle no support
12 }
```

Listing 4.2: Zugriff auf Kontakte

Mit dem Befehl *navigator.contacts.select(props,)* kann auf die Kontaktliste eines Geräts zugegriffen werden. Der erste Übergabeparameter ist dabei ein *Array* mit denjenigen

Attributen, welche angezeigt werden sollen, während der Zweite weitere Konfigurationen wie beispielsweise eine Mehrfachauswahl von Kontakten zulässt. Der Rückgabewert ist in diesem Falle ein *Array*, dass einen Kontakt enthält. Per *address[0]* wird auf die erste Adresse des Kontakts zugegriffen. Eigentlich kann daraufhin durch *.city* die Stadt dieser Adresse erreicht werden, jedoch erlaubt der verwendete Emulator bei der Erstellung eines Kontakts nur die Eingabe der Adresse in eine Eingabezeile. Deshalb wird hier auf die Adresszeile zugegriffen, in der zur Vereinfachung aktuell nur die Stadt notiert ist. Dieser Wert wird nach Bestätigung der Auswahl direkt in das Suchfeld übernommen und die Liste der Landkreise somit nach dieser Stadt gefiltert.

React Native App

Bei der React Native App wurde dies mit der Bibliothek „react-native-contacts“ umgesetzt. Diese greift in Android auf die *ContactsContract* Klasse und in iOS auf die *CNContact* zu.

Nach der Installation der Bibliothek müssen in der AndroidManifest-Datei erneut eine *uses-permissions* ergänzt werden, in diesem Fall die *android.permission.READ_CONTACTS*-Erlaubnis für das Lesen der Kontakte. Da es sich auch hierbei um ein „dangerous“ Berechtigung handelt, muss erneut zuerst nach der Erlaubnis des Nutzers gefragt werden. Dies wird in einer *asnyc*-Funktion erledigt. Sie wird in einem *Modal* aufgerufen, in dem nach dem Namen des gesuchten Kontakts gefragt wird. Nach der Betätigung des *Bestätigen*-Buttons wird nun mit der Funktion *Contact.getContactsMatchingString(text)*, die als Übergabeparameter einen *string* annimmt, ein Kontakt gesucht, der diesen Über-gabeparameter als Teil seines Namens enthält.

Das Ergebnis ist ein *Array* aus Objekten von Kontakten, auf die das Kriterium zutrifft. Zur Vereinfachung der Implementierung wird nur der erste Kontakt betrachtet und aus dessen *postalAddresses* die erste Adresse ausgewählt. Diese wird dann per Aufruf von *onQueryChange* automatisch in das Suchfeld gesetzt und somit die Liste der Landkreise nach dieser Stadt gefiltert.

Bei der Implementierung wurde einige Mal keine Daten angezeigt und die Console beinhaltete „*_U*: 0, *_V*: 0, *_W*: null, *_X*: null“. Es stellte sich heraus, dass dies damit verbunden ist, dass beim Anfordern der Ressourcen nicht auf das Auflösen des *Promises* gewartet wurde. Das konnte mit dem Schlüsselwörtern *await* in den asynchronen Funktionen gelöst werden.

4.6 Benachrichtigungen

Die entwickelte App soll seinen Abonnenten Benachrichtigungen senden können. In diesen können beispielsweise aktuelle Veränderungen der Inzidenz oder Informationen über neue Fallzahlen angezeigt werden.

Wie in Kapitel 2 beschrieben, lassen sich Benachrichtigungen in nicht-persistente und persistente Benachrichtigungen unterteilen. Im Zuge dieser Arbeit soll nun erklärt werden, wie letzteres umgesetzt werden kann, da diese eine höhere Relevanz zur Interaktion mit dem Nutzern bieten und hierfür auch Schnittstellen genutzt werden, die von nicht-persistenten Benachrichtigungen verwendet werden. Somit wird auf die Nutzung beider Arten eingegangen.

Für beide Anwendungen muss ferner das Management der Nutzererlaubnis implementiert werden. Dies wird zur Begrenzung des Umfangs in der Arbeit exkludiert und im Folgenden lediglich auf die Programmierung der Push Benachrichtigungsfunktion selbst eingegangen.

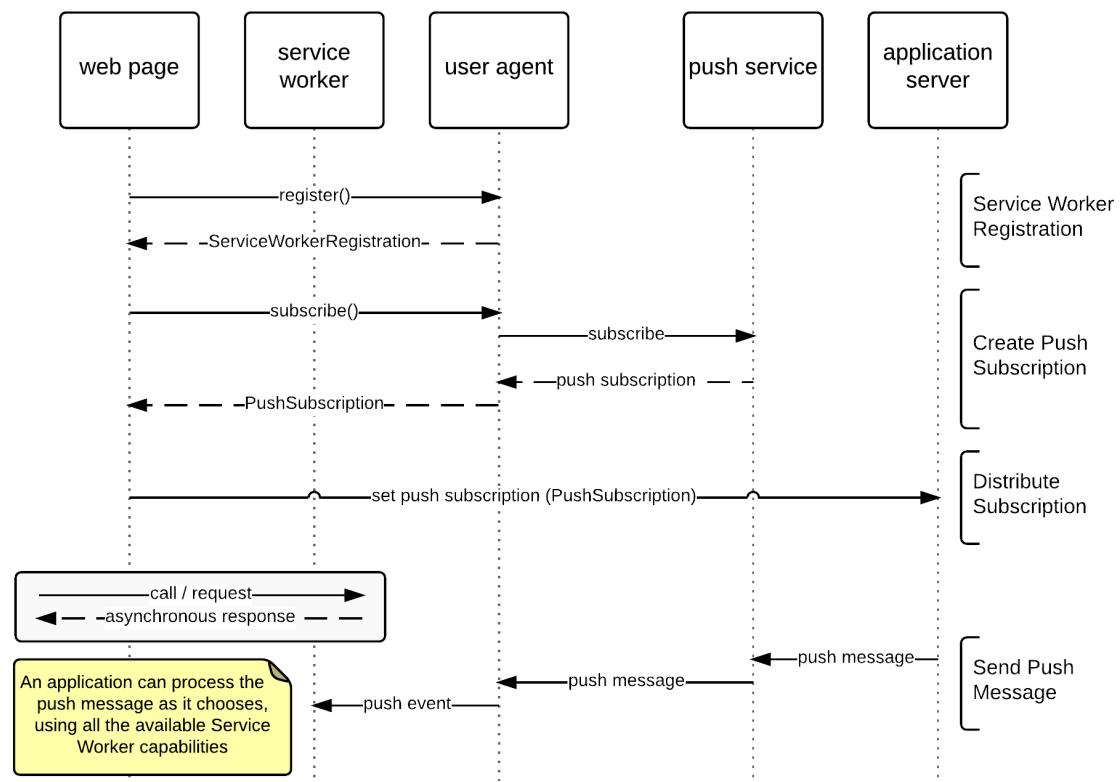


Abbildung 4.6: Ausschnitt eines Sequenzdiagramms des Web Push Prinzips

Progressive Web App

Zum besseren Verständnis der Implementierung soll zuerst das Konzept von Push umrissen werden. Der Name Push bezieht sich darauf, dass keine Aktion vom Client vorgenommen werden muss, um diese Informationen vom Server zu erhalten. Wie auf der rechten Seite in der Abbildung 4.6 dargestellt, besitzt das Konzept vier Abschnitte. Die erste Station betrifft die Installation des Service Workers. Dieser wird benötigt, um Push Benachrichtigungen im Hintergrund zu erhalten, da er unabhängig von der Anwendung selbst aktiv ist und somit immer auf einen Push reagieren kann.

Die zweite Station findet im *UserAgent*, also auf Clientseite statt. Um Push Benachrichtigungen zu ermöglichen, muss hier die Erlaubnis des Nutzers abgefragt werden. Wird dies erlaubt, wird durch eine *PushSubscription* ein Abonnement für das Gerät des aktuellen Nutzers erstellt. Dieses Abonnement wird benötigt, um das Endgerät eindeutig zu identifizieren und somit für spätere Aktionen verfügbar zu machen. Deshalb wird das *PushSubscription*-Objekt an das Backend oder den Server versendet, um dort in einer Datenbank abgespeichert zu werden. All dies ist Bestandteil der Push API.

Die dritte Station umfasst das Auslösen einer Push Benachrichtigung in einem Backend. Hier muss ein API-Aufruf an einen Push Service getätigt werden, in dem alle nötigen Informationen zum Inhalt und Empfänger der Push Benachrichtigung vorhanden sind. Dieser Aufruf wird bezeichnet als *Web Push Protocol*, welches durch einen IETF¹⁰-Standard definiert ist. Der Push Service ist Teil eines jeden Browsers zur Verwaltung von Push Benachrichtigungen und ist deshalb dessen Implementierung überlassen. Jedoch ist festgelegt, dass der Service stets Anfragen in Form des *Web Push Protocols* verarbeiten kann, wodurch für Entwickler die Form des Push Services irrelevant ist.

Die letzte Station behandelt das *Push*-Event auf dem Endgerät, welches die empfangenen Daten verarbeitet und daraus eine Benachrichtigung erzeugt. Dies wird von der Notification API abgedeckt, welche Zugriff auf die betriebssystemsspezifischen Benachrichtigungsfunktionen besitzt. Hierfür wird im Event die Anweisung *self.registration.showNotification(title, options)* ausgeführt. Der Übergabeparameter *title* ist dabei der Titel der Benachrichtigung und *options* ein Objekt mit einer Vielzahl von Optionen wie der Text der Benachrichtigung, Aktion-Buttons, Icons und Vibrationsanweisungen. Im Service Worker einer Webanwendung kann auf dieses Event reagiert werden und aus den Informationen, die vom Push Service gesendet werden, eine lokale Benachrichtigung erzeugen. Dies ist auch möglich, wenn die PWA geschlossen ist, da der Service Worker unabhängig von der Anwendung selbst aktiv ist. Eine aktive Internetverbindung ist jedoch unabdingbar, da der Push Service die Benachrichtigung sonst so lange einbehält, bis wieder eine Verbindung besteht.

Um den Fokus für diese Arbeit darauf zu richten, wie das Erhalten von Benachrichtigungen bei PWAs implementiert wird, wird der Backend-as-a-Service *Firebase* genutzt. Wichtig ist dabei, dass diese Funktionalität auch komplett selbst implementiert werden

¹⁰Internet Engineering Task Force. Eine Organisation, die Web-Standards entwickelt.

könnte, indem ein eigener Webserver mit beispielsweise Node.js aufgesetzt wird. *Firebase* wird von Google zur effizienten Implementierung eines Backendsystems zur Verfügung gestellt. Speziell werden dessen Dienste Firebase Cloud Messaging (FCM) und die Real Time Database genutzt. Über dessen Konsole können einmalige sowie regelmäßige Benachrichtigungen an alle registrierten Nutzer versendet werden. Da im Rahmen dieser Arbeit kein komplettes Management von Benachrichtigungen behandelt werden soll, werden die *Keys* des *PushSubscription*-Objekts lediglich in der Real Time Database abgelegt. Daraufhin können über die *Firebase* Console manuell Push Benachrichtigungen an alle *Keys* versendet werden.

Damit Benachrichtigungen in der Anwendung empfangen werden können, muss erst die Erlaubnis des Nutzers erfragt werden. Dies geschieht durch die Notification API über den Aufruf von *Notification.requestPermission()* beim Klicken des „Fallzahlen abonnieren“-Buttons. In der asynchronen Rückgabe Funktion wird durch die *messaging.getToken()*-Funktion von *Firebase* ein Token für das Gerät generiert. Im Service Worker wird dann im Falle eines Pushes vom Server im *push*-Event durch die *showNotification(title, body)*-Funktion eine Benachrichtigung entsendet. *Firebase* ersetzt die Nutzung des *push*-Events jedoch durch seine eigene *messaging.setBackgroundMessageHandler()*-Funktion, in der dasselbe durchgeführt wird.

React Native App

Der Prozess von Push Benachrichtigungen ist bei Native Apps ähnlich aufgebaut. Der Push Service ist hier jedoch nicht browser- sondern betriebssystemabhängig und bezeichnet sich als *Operating system push notification service (OSPNS)*. Bei Android Geräten werden Push Benachrichtigung von dem Google Cloud Messaging (heute Firebase Cloud Messaging) verwaltet und für iOS gibt es den Apple Push Notification service (APNs). Native Anwendungen benötigen wie PWAs eine Internetverbindung zum Empfangen von Push Benachrichtigungen.

Auch bei der React Native Applikation wird zur Vereinfachung des Prozesses das FCM von *Firebase* verwendet, zumal es APNs integriert. Dafür wird die Bibliothek „react-native-firebase“ genutzt, die Push Benachrichtigungen an Android und iOS ermöglicht. In dieser Arbeit wird jedoch nur exemplarisch auf die Einrichtung von FCM mit iOS eingegangen. Die Native App greift nach erfolgreicher Einrichtung, die wie bei der PWA abläuft, auf dieselbe Real Time Database zu wie die PWA.

Push Benachrichtigungen sind eine der Funktionen, für die keine explizite Berechtigung auf Android Geräten vorhanden sein muss. Auf der anderen Seite wird dies von der verwendeten Bibliothek nur für die iOS Implementierung benötigt. Daher muss das Ergebnis der Funktion *messaging().requestPermission()* abgefragt und dementsprechend Benachrichtigungen erlaubt oder verwehrt werden.

Auch bei nativen Anwendungen wird zwischen persistenten und nicht-persistenten Benachrichtigungen unterschieden. Das unterscheidet sich bei der Implementierung insoweit, dass das Auslösen einer Benachrichtigung nicht in der Komponente selbst mittels des `messaging().onMessage()`-Funktionsaufrufs in einem `useEffects` geschieht, sondern in der `index.js`-Datei durch die `messaging.setBackgroundMessageHandler()`-Funktion allgemein initialisiert wird. Diese Implementierung ermöglicht nur das Empfangen von Push Benachrichtigungen, wenn die App im Hintergrund oder geschlossen ist.

Kapitel 5

Ergebnisse und Diskussion

Die Evaluierung der beiden Entwicklungsansätzen PWA und Native App erfolgt nun auf Basis des in Kapitel 3 erläuterten Kriterienkatalogs. Hierbei soll erläutert werden, ob und welche Vor- und Nachteile das Implementieren einer PWA im Gegensatz zu einer Native App darstellt. Außerdem wird die Frage beantwortet, ob für diesen Anwendungsfall die PWA eine Native App ersetzen kann.

5.1 Funktionalitäten

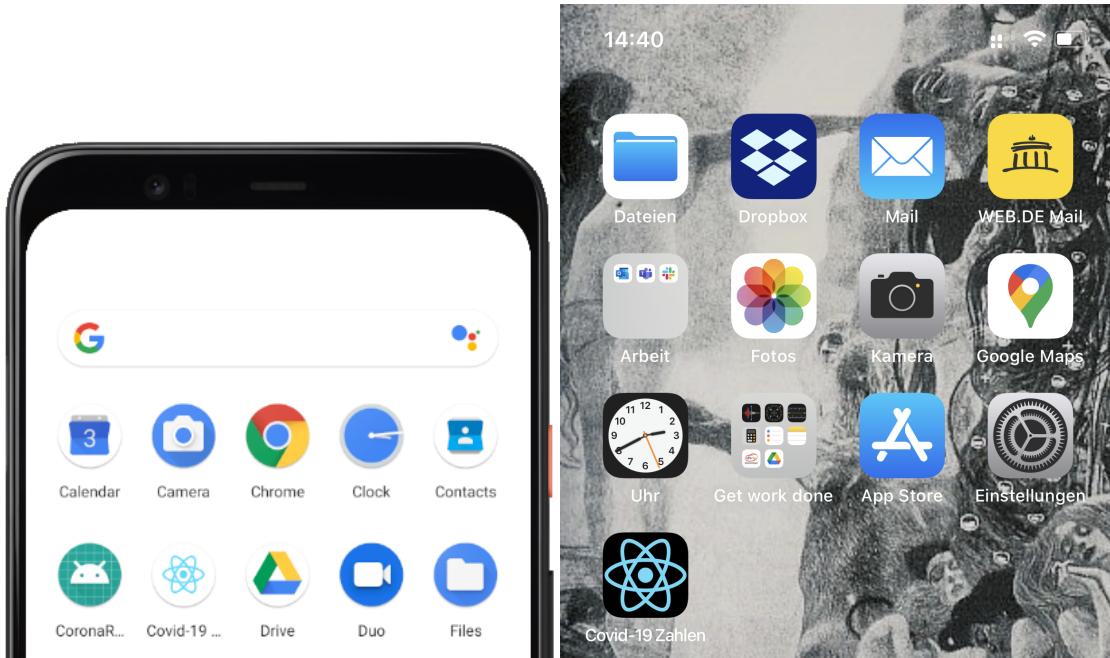
In diesem Kapitel wird lediglich darauf eingegangen, ob Schnittstellen zur Implementierung der geforderten Funktionalitäten vorhanden sind und wie viele Möglichkeiten diese anbieten. Die Kompatibilität der Funktionalitäten mit verschiedenen Betriebssystemen und Browsern ist Teil des nächsten Kapitels.

Installierbarkeit und Aktualisierungen

Lange Zeit war das Kriterium der Installierbarkeit ein signifikanter Unterschied zwischen nativen Apps und Web Applikationen. Durch den Service Worker und dem Manifest ist dies nun möglich. Zur Installation der programmierten PWA wird der Link <https://corona-react-pwa-ba.netlify.app/> aufgerufen und das browserabhängige Vorgehen durchgeführt. In Android ist die „App installieren“-Option unter dem Einstellungen-Symbol zu finden. Danach ist die App auf dem Startbildschirm verfügbar und wird beim Aufrufen automatisch aktualisiert. Hierbei fällt auf, der Nutzer automatisch durch eine Banner am unteren Bildschirmrand darauf hingewiesen wird, dass er die Anwendung herunterladen kann. Bei Android Geräten erscheinen die Apps nach der Installation durch das generierte WebAPK zudem im App-Drawer und als App in den Einstellungen des Geräts. Um die PWA auf einem iOS Gerät zu installieren wird nach dem Aufrufen der App mit Safari das Teilen-Symbol geklickt und die Option „Zum Home-Bildschirm“ gewählt. Auch hier ist die Anwendung dann vom Startbildschirm aus aufrufbar und wird stets auf dem neuesten Stand des Deployments aktualisiert.

Bei der nativen Anwendung öffnet der Nutzer den App Store oder Play Store, navigiert

zur gewünschten App, akzeptiert alle Berechtigungen und kann sie dann installieren. Die Apps können dann, wie in Abbildung 5.1 abgebildet, über den Startbildschirm aufgerufen werden.



(a) React Native App und PWA im App-Drawer auf einem Android Gerät (b) PWA im Startbildschirm auf einem iOS Gerät

Quelle: Eigene Darstellung

Abbildung 5.1: Darstellung der installierten Apps

Auf dem Startbildschirm sind die Anwendungen in iOS optisch nicht voneinander unterscheidbar. Mit dem genutzten Emulator wird demgegenüber in der Chrome Version 83 durch ein kleines Browser-Symbol auf dem App-Icon deutlich, dass es sich hierbei um eine Webanwendung handelt. In aktuelleren Browerversversionen wie Chrome 91 fällt auch dieses weg, wodurch PWAs auf Android Geräten ebenfalls aussehen wie Native Apps. Mittlerweile können auch Begrüßungsbildschirme¹ für PWAs implementiert werden, wenn die Attribute *name*, *background_color* und *icons* im App Manifest vorhanden sind [35].

Ein Vorteil von PWAs ist dennoch, dass sie grundsätzlich nicht installiert werden müssen. Alle Funktionalitäten wie Standortzugriff oder Push Benachrichtigungen können aus dem Web heraus geschehen. Native Apps hingegen können erst genutzt werden, wenn sie installiert sind. Somit können Nutzer, deren Speicherplatz nicht für die Anwendung ausreicht, diese nicht benutzen. Die in dieser Arbeit entwickelten Apps benötigt generell wenig Speicherplatz, da die Daten vor allem vom Server abgerufen werden, aber auch

¹Wenn eine App gestartet wird, zeigt sie einige Sekunden eine Art Ladebildschirm an, bis sie bereit ist. Dabei wird meist das Logo der App und eine Hintergrundfarbe angezeigt.

hier macht sich der Unterschied bemerkbar: die React Native App verbraucht 66.02 MB und die PWA nur circa 0.28 MB. Das ist darauf zurückzuführen, dass die PWA generell weniger Ressourcen verbraucht, da es sich um gängige Webtechnologien handelt. Gerade bei der React Native App gibt es viele Hintergrundprozesse, die Speicherplatz benötigen, um die Kommunikation zwischen Main Thread und JavaScript Thread zu ermöglichen. Aber auch beim Vergleich zwischen klassischen, nativen Anwendungen und PWA ist der Unterschied ähnlich: Android Apps benötigen durchschnittlich 25 MB Speicherplatz [36].

Im Allgemeinen verbraucht die Installation einer PWA außerdem weniger Datenvolumen als die einer Native App. Ein Beispiel dafür ist die Twitter Lite PWA, welche im Gegensatz zu den 23.5 MB der Twitter Android App lediglich 0.6 MB verbraucht [37]. Dies ist aber kritisch zu betrachten, da PWAs Inhalte in neuen Routen erst laden, wenn diese aufgerufen wird und somit auf langer Sicht ebenfalls so viel Datenvolumen verbrauchen können wie Native Apps.

Ferner ist es möglich, PWAs im Google Play Store zu veröffentlichen, wodurch sie einerseits durch das Web, anderseits durch den Store für Nutzer zur Verfügung stehen. Somit können Entwickler zur Verbesserung der Auffindbarkeit der Anwendung sowohl SEO² als auch ASO³ durchführen. Der Apple App Store unterstützt die Veröffentlichung von PWAs aktuell nicht. Dennoch sind PWAs generell unabhängig von App Stores, da sie über das Internet erreichbar sind. Gerade dies kann allerdings eine Schwachstelle von PWAs darstellen, da sie trotz der Veröffentlichung über HTTPS, im Gegensatz zu Native Apps keine manuelle Verifizierung zur Veröffentlichung durchlaufen müssen.

Ein entwicklungstechnischer Vorteil von PWAs gegenüber nativen Apps ist, dass sie einfacher zu verwalten sind. Wenn es eine neue Version der Anwendung gibt, wird das Deployment durchgeführt und jeder Nutzer erhält automatisch die aktuelle Version. Bei nativen Applikationen steckt ein Mehraufwand dahinter, da sie signiert und verifiziert werden müssen, um dann vom Nutzer aus dem jeweiligen Store aktualisiert werden zu können.

Diese Art von Verwaltung bei PWAs hat für die Entwickler außerdem den Vorteil, dass die Nutzer stets die neueste Version der Anwendung besitzen. Somit ist vor allem bei langjährigen Projekten, in denen sich eventuell Schnittstellen über die Zeit ändern, Persistenz bei der Nutzung der App garantiert und es muss keine Abwärtskompatibilität implementiert werden. Dadurch können Entwicklungs- und Wartungskosten der Anwendung reduziert werden.

Offlinebetrieb

²SEO ist kurz für Suchmaschinenoptimierung

³Kurz für App-Store-Optimierung

Beide Anwendungen wurden auf ihre Weise offlinefähig implementiert. Bei der PWA bedeutet dies, dass ein Service Worker programmiert werden muss, der durch die Cache API-Anfragen abspeichert und diese bei Offlinebetrieb nutzt. Hierbei wurden verschiedene Caching-Verfahren betrachtet und diejenige ausgewählt, die am besten zur Anwendung passt.

In der React Native App wurde der *AsyncStorage* implementiert. Die Daten, die durch die Netzwerkanfrage zur Verfügung stehen, werden in dessen persistente Speicher hinterlegt und genutzt, wenn keine Internetverbindung besteht. Sie bestehen auch weiterhin, wenn die Anwendung komplett geschlossen wird. Sobald die Verbindung wieder hergestellt wird, werden die Daten erneut aus dem Internet geladen.

Der Unterschied in der Implementierung des Offlinebetriebs ist, dass bei der PWA auf fehlschlagende Netzwerkanfragen reagiert und bei der Native App der Stand der Internetverbindung abgefragt wird. Somit werden bei der nativen Anwendung trotz der vermeintlichen „Network, falling back to cache“-Strategie nicht mehr Ressourcenanfragen gestellt als bei der PWA.

Die Anwendungen unterscheiden sich in sofern nicht voneinander, dass beide erst nach der Installation offline verfügbar sind. Für die PWA bedeutet dies, dass sie zumindest einmal aufgerufen werden muss. Denn nur wenn der Service Worker installiert und aktiviert ist, hat er die Fähigkeit, im Falle einer fehlenden Netzwerkverbindung mit zwischengespeicherten Daten zu reagieren. Die React Native App muss ebenfalls durch das Herunterladen aus einem App Store installiert werden. In beiden Fällen ist es außerdem möglich Vorgänge, die im Offlinebetrieb durch den Nutzer vorgenommen werden, zurückzustellen und erst auszuführen, wenn der Internetzugriff wiederhergestellt ist. Für die PWA bedeutet dies, dass die Background Sync API implementiert werden muss.

Standortzugriff

Der Standortzugriff konnte in beiden Anwendungen gleichermaßen realisiert werden. In der PWA wird dafür die Geolocation API des Webs verwendet und bei der React Native App durch eine Bibliothek im Hintergrund auf die Google Location Service API in Android und Core Location API in iOS zugegriffen.

Ein Nachteil der PWA bei dieser Funktionalität ist, dass durch die Geolocation API lediglich der Zugriff auf die Koordinaten des Standorts oder das Beobachten der Position möglich ist. Bei nativen Anwendungen gibt es in beiden Betriebssystemen inkludierte Schnittstellen (*Geocoder* und *CLGeocoder*), mit denen das Reverse Geocoding durchgeführt werden kann. Dadurch entfällt bei Native Apps die Nutzung von externen Dienstleistern zur Bestimmung der Aufenthaltsort.

Außerdem ist die Geolocation API, die in der PWA genutzt wird, abhängig von einer aktiven Internetverbindung. Dies ist bei Native Apps nicht der Fall, denn mobile Endgeräte können durch das verbaute GPS auf den aktuellen Standort des Nutzers zugreifen.

Zuletzt ist bei PWAs im Gegensatz zu nativen Apps die Weiterverarbeitung der Standortdaten im Sinne von beispielsweise Geofencing⁴ nicht möglich. Ein Entwurf einer Spezifikation zur Implementierung einer solchen Funktion von 2017 ist von W3C als obsolet markiert [38]. Gerade hierfür ist darüber hinaus die Genauigkeit der Koordinaten relevant. Ein Beispiel dafür ist das Benachrichtigen des Nutzers beim Betreten unterschiedlicher Räume in einer Wohnung, ferner aber auch das Bestimmen des Standorts eines verlorengegangenen Geräts. Für die implementierte App ist die Genauigkeit der Koordinaten nicht von Bedeutung, da lediglich die Stadt ermittelt werden soll, in welcher der Nutzer der App sich befindet. Beim Testen wird jedoch deutlich, dass die Koordinaten, wie in Abbildung 5.2, exakt übereinstimmen.

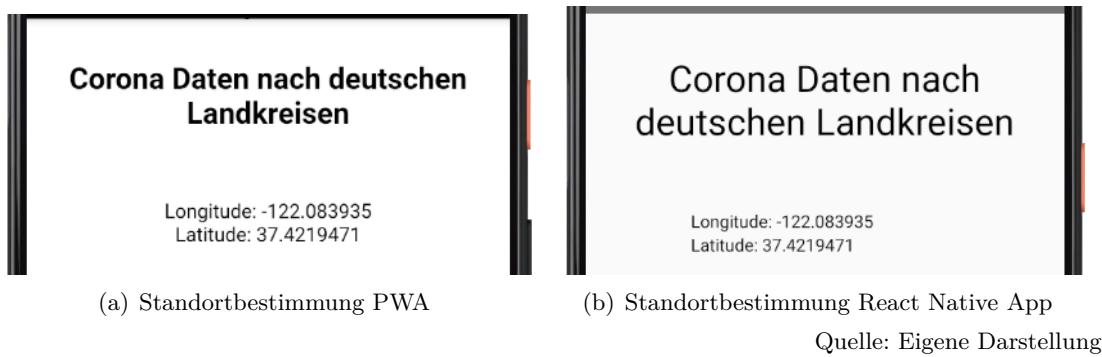


Abbildung 5.2: Darstellung der bestimmten Koordinaten

Kontaktzugriff

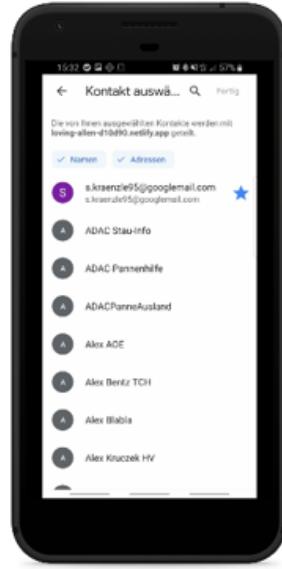
Diese Funktionalität wurde mit der Contact Picker API implementiert. Durch sie ist es möglich aus den vorhandenen Kontakten des Geräts einen Kontakt auszuwählen und auf dessen Daten, darunter auch die Adresse, zuzugreifen.

Mit React Native gibt es keine Probleme mit der Umsetzung. Durch das Einbinden einer *Bridge* mittels einer externen Bibliothek konnte diese Funktionalität programmiert werden.

Der Zugriff auf eine grundlegende Schnittstelle wie der Kontaktliste ist für native Anwendungen selbstverständlich. Dabei ist sowohl das Lesen als auch die Erstellung und Modifikation von Kontakten möglich. Die Contact Picker API bietet hingegen aktuell nur das Lesen der Kontaktliste an. Das geschieht in einer browsereigenen Darstellung-

⁴Geofencing bezeichnet das Auslösen von Benachrichtigungen beim Betreten oder Verlassen von definierten Bereichen und Orten.

form, die in 5.3 für den Chrome Browser abgebildet ist. Entwürfe für das Erweitern der Funktionalität dieser API liegen aktuell nicht vor.



Quelle: Eigene Darstellung

Abbildung 5.3: Benutzeroberfläche der Contact Picker API in Chrome

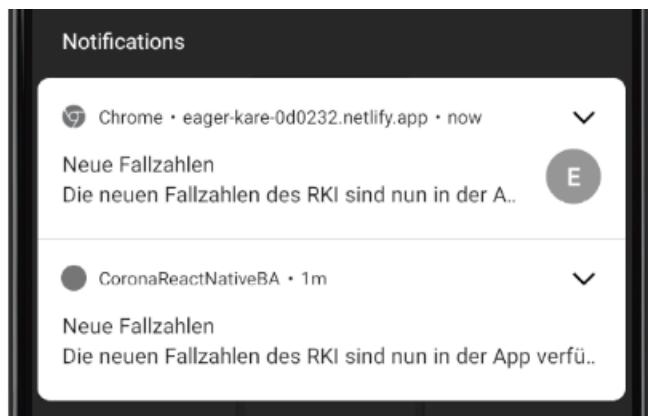
Benachrichtigungen

Optisch lassen sich die Benachrichtigungen von PWAs, wie in Abbildung 5.4 zu erkennen ist, nicht von denen von Native Apps unterscheiden. Dies ist damit zu erklären, dass in beiden Fällen auf die betriebssystemspezifische Benachrichtigungsfunktion zugegriffen wird. Auch im Aspekt Aktionen bieten die zwei Apps dieselben Möglichkeiten. Bei beiden ist es möglich, Aktionen zu definieren, die von der Benachrichtigungszeile aus getätigter werden können. Das betrifft beispielsweise das Beantworten einer Nachricht oder das Löschen einer E-Mail.

Dennoch wird bei den Benachrichtigungen eine weitere Schwachstelle von PWAs deutlich. Der Safari Browser erlaubt auf mobilen Endgeräten aktuell weder persistente noch nicht-persistente Benachrichtigungen von Webanwendungen. Laut *caniuse* implementiert der Safari Browser eine eigene Version der Push API, die jedoch nicht für Safari auf iOS verfügbar ist [39].

Zusammenfassung

Generell besitzt das Web alle Schnittstellen, um die im Kapitel 3 definierten Funktionalität umzusetzen. Bei der Implementierung der Funktionalitäten wird dennoch deut-



Quelle: Eigene Darstellung

Abbildung 5.4: Vergleich Benachrichtigungen PWA (oben) und React Native App (unten)

lich, dass PWAs nicht beliebige native Schnittstellen aufrufen können, sondern lediglich Webschnittstellen nutzen. Diese befinden sich noch im Aufbau und sind deshalb nicht umfassend, als experimentell markiert, nicht von jedem Browser unterstützt oder existieren nicht. Das hat bei der Verwendung jener Schnittstellen zufolge, dass sich im Laufe der Zeit Schnittstellen verändern, ersetzt oder sogar entfernt werden. Deshalb sollten diese mit Vorsicht genutzt werden. Beispielsweise ist das App Manifest aktuell vom *W3C* als experimentell gekennzeichnet und die Contact Picker API noch ein inoffizieller Entwurf [14][40].

5.2 Kompatibilität mit verschiedenen Betriebssystemen

Durch das Testen der Apps auf jeweils einem iOS und einem Android Smartphone soll nun die Kompatibilität der beiden Anwendungen überprüft werden. Hierfür wird ein Apple iPhone 12 mit iOS 14 und ein Huawei P10 Lite mit Android 10 verwendet. Die Native App wird mit einem Google Pixel 4 Emulator mit Android 11 getestet.

Progressive Web App

Die PWA ist unter dem Link <https://corona-react-pwa-ba.netlify.app/> erreichbar. Hierdurch kann sie mit jedem Browser eines internetfähigen Geräts erreicht werden, wodurch sie größtenteils betriebssystemunabhängig sind. Falls ein Browser dennoch gewisse Funktionalitäten der PWA nicht unterstützt, wird durch den Ansatz des Progressive Enhancements sichergestellt, dass die Seite zumindest die aktuellen Corona-Daten darstellt und diese nach dem Namen des Landkreises filtern kann.

In der Tabelle 5.1 sind die Ergebnisse des Testens aufgelistet. In Klammern dahinter steht zusätzlich jeweils die Browerversion, ab dem laut *MDN Web Docs* diese Funktionalität unterstützt wird. Bei der Verwendung eines Android Geräts können die meisten Funktionen genutzt werden, lediglich der Kontaktzugriff entfällt auf dem Firefox Webbrowser. Für die Installierbarkeit mit Opera gibt es von *MDN Web Docs* keine Angaben über die Browerversion. Die Verifizierung konnte jedoch beim Testen mit der Opera Version 64 nachgewiesen werden. Über die Nutzung der PWA auf einem Android Gerät mit dem Safari Browser konnten keine Daten erhoben werden, da die Safari App im Google Play Store nicht zu Verfügung steht.

Tabelle 5.1: Kompatibilität der Progressive Web App mit verschiedenen Browsern auf einem Android Gerät

Funktionalität	Browser auf Android Gerät			
	Chrome	Firefox	Opera	Safari
Installierbarkeit	ja (39)	ja (53)	ja (k. A.)	-
Offlinebetrieb	ja (40)	ja (39)	ja (27)	-
Standortzugriff	ja (18)	ja (4)	ja (11)	-
Kontaktzugriff	ja (80)	nein	ja (57)	-
Benachrichtigung	ja (42)	ja (44)	ja (37)	-

Die Tabelle 5.2 zeigt die Ergebnisse der Kompatibilität bei Nutzung eines iOS Geräts. Hierbei wird deutlich, dass die PWA, wenn sie im Chrome, Opera oder Firefox Webbrowser aufgerufen wird, nicht die Funktionalität der Installierbarkeit unterstützt. Das ist damit zu begründen, dass iOS diese Browser mit WebKit rendert, statt mit den eigenen HTML-Renderern (z. B. Blink für Chrome). Ferner fehlt die Unterstützung des Kontaktzugsriffs und Push Benachrichtigungen. Ersteres ist seit iOS Version 14.15 und Safari 14.1 auch bei iOS Gerät als experimentelle Funktion verfügbar, jedoch ist das nur auf die Kontaktdaten *name*, *email* und *tel* (Telefonnummer) beschränkt [41]. Dennoch zeigt dies, dass sich Apple einer Implementierung moderner Browserschnittstellen für den Safari Browser annähert. Über eine Umsetzung von Push Benachrichtigungen auf mobilen Endgeräten gibt es aktuell keine Auskünfte [41].

Tabelle 5.2: Kompatibilität der Progressive Web App mit verschiedenen Browsern auf einem iOS Gerät

Funktionalität	Browser auf iOS Gerät			
	Chrome	Firefox	Opera	Safari
Installierbarkeit	nein	nein	nein	ja (11.3)
Offlinebetrieb	ja (40)	ja (44)	ja (27)	ja (11.3)
Standortzugriff	ja (18)	ja (4)	ja (11)	ja (3)
Kontaktzugriff	nein	nein	nein	teilweise (14.7)
Benachrichtigung	nein	nein	nein	nein

Neben den teilweise fehlenden Funktionalitäten in bestimmten Browsern und Browserversionen ist weiteres Problem von PWAs, dass einige der verwendeten Schnittstellen als experimentell gekennzeichnet sind. Das hat bei deren Verwendung jener Schnittstellen zufolge, dass sich im Laufe der Zeit Schnittstellen verändern oder sogar ersetzt werden. Diese Unsicherheit kann für einige Projekte ein ausschlaggebender Faktor gegen die Implementierung einer PWA sein. Aktuell betrifft das APIs und Technologien wie das App Manifest, die Push API und die Funktionen der Contact Picker API [14, 42, 43].

React Native

Die React Native Android App kann nach dem Ausführen des Befehls „npx react-native start“ und „npx react-native run-android“ auf einem Emulator oder einem angegeschlossenen Android Gerät genutzt werden. Für die Nutzung auf einem iOS Gerät muss der korrespondierende Befehl „npx react-native run-ios“ auf einem Mac-PC durchgeführt werden. Dadurch, dass bei der Entwicklung der Anwendung kein Mac-PC zur Verfügung stand, konnte nicht zeitgleich geprüft werden, ob die React Native Anwendung alle Funktionalitäten auch in iOS unterstützt. Es wird dennoch davon ausgegangen, dass alle geforderten Funktionen wie bei der Android App funktionieren, weil alle verwendeten Bibliotheken als kompatibel mit Android und iOS gekennzeichnet sind.

Zusammenfassung

Die Unterstützung von PWAs auf Apple Geräten hat sich seit iOS 11.3 verbessert, da mit dieser Version Schnittstellen wie der Service Worker, das App Manifest oder xx verfügbar sind. Allerdings ist es, wie bereits im Kapitel 4 beschrieben, aktuell nicht möglich, Push Benachrichtigungen auf einem iOS Smartphone zu erhalten oder den Zugriff auf die Kontakte zu ermöglichen. Das ist ein großer Nachteil von PWAs, da 26,34 Prozent des Marktanteils somit nicht bedient werden können [44]. Außerdem geht damit ein ausschlaggebender Faktor zur laufenden Interaktion mit den Nutzern verloren. Auch in naher Zukunft arbeitet Apple aktiv nicht an einer Implementation dieser Funktionalität.

Rein von der Auswahl der verschiedenen Betriebssysteme, auf denen eine PWA aufgerufen werden kann, hat eine PWA mehr Reichweite. Da die Kompatibilität jedoch lückenhaft ist, kann dies nicht als Vorteil der PWA gesehen werden. Vor allem die fehlende Unterstützung von iOS Geräten spricht gegen deren Etablierung. Eine React Native App verspricht mehr Kompatibilität als eine Native App, jedoch konnte auch dies nicht bestätigt werden, da die iOS App in diesem Falle bereits beim Aufsetzen der Entwicklungsumgebung gescheitert ist. Bei einer Ausstattung mit einem Mac-Computer wäre dies weniger problematisch, denn dort ist eine zeitgleiche Entwicklung für iOS und Android möglich. Dies trifft ebenso für die Nutzung des Expo Frameworks zu, da dadurch auch auf einem Windows Computer eine auf beiden Betriebssystemen lauffähige App

entwickelt werden kann. Native Apps selbst sind nach wie vor betriebssystemabhängig, wodurch sich die Kompatibilität mit verschiedenen Betriebssystemen erübrigert.

Zusammenfassend bedeutet dies dennoch, dass PWAs im Bezug auf Kompatibilität mit verschiedenen Betriebssystemen Native Apps voraus sind, da sie generell auf beiden betrachteten Betriebssystemen lauffähig sind. Die Unterstützung einzelner Funktionalitäten ist wiederum eine Schwachstelle von PWAs, da manche nicht auf iOS Geräten existieren. Plattformunabhängige Apps hingegen bieten – wenn eine geeignete Entwicklungsumgebung zur Verfügung steht – für die geforderten Funktionalitäten am meisten Kompatibilität mit verschiedenen Betriebssystemen.

5.3 Entwicklungsaufwand

Zur Bestimmung des Entwicklungsaufwands wurde einerseits die Dauer der Recherche gemessen und andererseits die reine Implementierungsdauer. Ersteres umfasst dabei die Einarbeitung in Technologien und das Studieren der Spezifikationen, die für die Funktionalitäten benötigt werden. Die Anwendungen wurden mit einem soliden Grundwissen in HTML, CSS und JavaScript entwickelt. Tiefgehende Kenntnisse der Bibliothek und des Frameworks waren vor der Programmierung der Apps nicht vorhanden. In der Tabelle 5.3 ist die Auflistung der Aufwände dargestellt.

Tabelle 5.3: Entwicklungsaufwand der beiden Anwendungen

Funktionalitäten	Recherche		Implementierung	
	PWA	Native App	PWA	Native App
Einrichten der Entwicklungsumgebung	60 min	240 min	25 min	60 min
Installierbarkeit	30 min	60 min	30 min	15 min
Offlinebetrieb	100 min	40 min	45 min	30 min
Standortzugriff	30 min	45 min	15 min	30 min
Kontaktzugriff	60 min	45 min	45 min	30 min
Benachrichtigung	300 min	250 min	120 min	200 min
Gesamt	580 min	680 min	280 min	395 min

Beim Einrichten der Entwicklungsumgebung konnten in beiden Fällen Toolchains zur Beschleunigung des Prozesses genutzt werden. Dennoch hat dies bei React Native mehr Zeit beansprucht als bei dem Aufsetzen der PWA, weil neben der Installation der CLI weitere Vorehrungen zur Einrichtung der Umgebungsentwicklung getätigter werden mussten. Das betrifft beispielsweise das Konfigurieren von Android Studio oder das Aufsetzen eines Android Virtual Device (AVD). Bei der PWA musste lediglich der Befehl „npx

create-react-app corona-react-pwa-ba⁵ ausgeführt werden und ein vollständiges Setup zum Entwickeln einer Single Page Application ist erstellt. Diese konnte dann sofort mit Webpack auf einem Webserver gestartet und weiterentwickelt werden.

Die Installierbarkeit hat bei PWAs circa doppelt so lange benötigt, weil konkrete Implementierungen vorgenommen werden mussten, um dies hierbei zu ermöglichen. Beispielsweise musste das App Manifest erstellt und Icons zur Verfügung gestellt werden. Native Apps sind generell installierbar, sie müssen lediglich zur Veröffentlichung in einem App Store signiert werden.

Das Ermöglichen des Offlinebetriebs hat bei der PWA mehr Zeit beansprucht, da hier die verschiedenen Arten des Cachings der Daten betrachtet und ausgewählt werden mussten. Für die Umsetzung in der Native App wurde der gängige *AsyncStorage* in Kombination mit der React Native Bibliothek *NetInfo* gewählt.

Das Ermöglichen des Standortabfrage hat in beiden Anwendungen ungefähr gleich viel Zeit in Anspruch genommen.

Die Recherche für die Implementierung des Kontaktzugriffs benötigte in der PWA länger, da diese Funktion erst seit Chrome 80 auf Android zu Verfügung steht und somit wenig Literatur vorhanden ist [40]. Auch die Programmierung selbst ist aufwendiger, da der Kontaktzugriff nur mit dem Emulator über HTTPS und nicht mit *localhost* möglich ist, weshalb die PWA stets deployt werden musste.

Die Benachrichtigungen sind bei React Native mit mehr Aufwand verbunden, da die Kompatibilität auf beiden betrachteten Betriebssystemen gewährleistet werden muss. Jedoch sind die Zeiten auch hier mit Vorsicht zu betrachten, da die PWA auf iOS generell keine Benachrichtigungsfunktion zur Verfügung stellen kann und somit möglicherweise nötige Polyfills beim Entwicklungsaufwand entfallen.

Der Gesamtaufwand für die Recherche und Implementierung der PWA beträgt somit circa 16 Stunden (850 min) und fast 18 Stunden (1075 min) für die React Native App.

Insgesamt wird deutlich, dass für die Einführung und Implementierung der React Native App im Gegensatz zur PWA mehr Zeit beansprucht wurde, obwohl Teile des Codes wiederverwendet werden konnten. Besonders ausschlaggebend war dabei die Einarbeitung in die Technologien zur plattformunabhängigen Implementierung. Denn zur Entwicklung mit React Native muss einerseits React Native verstanden und anderseits Kenntnisse in nativer Programmierung aufgebaut werden. Dies betrifft beispielsweise den Aufbau nativer Anwendungen und die betriebssystemspezifischen Programmiersprachen.

Hierbei muss außerdem beachtet werden, dass in dieser Arbeit aufgrund der Tatsache, dass kein Mac-Computer zur Entwicklung der Native App für iOS vorhanden war. Das wirkt sich insofern auf den Entwicklungsaufwand aus, dass theoretisch für die Implementierung mehr Zeit in Anspruch genommen werden muss, weil eventuell auftretende Schwierigkeiten behoben werden müssen. Zum Beispiel ist die Einrichtung von Push Be-

⁵Letzteres ist der Name der Anwendung und kann frei gewählt werden

nachrichtigungen für iOS mit *Firebase* ein hinzukommender Aufwand, da dies separat von der Android Version gemacht werden muss. Somit steigt der tatsächliche Entwicklungsaufwand der React Native App zusätzlich.

Die Ergebnisse der Entwicklungsaufwände sind nach Meinung des Autors kritisch zu betrachten, da bei der Programmierung der PWA deutlich weniger fremder Code verwendet wurde als für die React Native App. Hier ist es nämlich Teil des Arbeitsablaufs, sich vor-implementierten Bibliotheken zu bedienen, da oftmals *Bridges* für gängige Funktionalitäten bereits existieren. Würden diese nicht verwendet werden, wäre der Entwicklungsaufwand der React Native App deutlich höher. Auch für die PWA könnten Bibliotheken und Werkzeuge genutzt werden, wie etwa das bereits erwähnte Tool „Workbox“ von Google. Dennoch spricht es für die PWA, dass keine Abhängigkeit von anderen Bibliotheken benötigt werden, sondern die Funktionalität lediglich durch moderne Schnittstellen geschieht, die das Web anbietet. Eine Native mit Java oder Swift implementierte App besitzt ebenso weniger Abhängigkeiten als die React Native App, da sie auf die nativen Schnittstellen der Betriebssysteme zugreifen. Diese Abhängigkeit ist also nur eine Besonderheit von React Native Apps.

Außerdem ist an dieser Stelle anzumerken, dass der Entwicklungsaufwand einer mit React Native entwickelten App nicht gleichzusetzen ist mit einer tatsächlich nativen App. Diese benötigt für dasselbe Ergebnis zwei Implementierungen, eine für Android und eine für iOS. Dadurch kann sich der Recherche- und Entwicklungsaufwand verdoppeln, zumal auch eine Einarbeitung in zwei verschiedene Programmiersprachen und Entwicklungsumgebungen stattfinden muss.

Ein großer Nachteil der Entwicklung mit React Native ist, dass aktuell nicht komplett plattformunabhängig programmiert werden kann. So ist ein Mac-Computer die Voraussetzung zur Implementierung von nativen iOS Komponenten in einer React Native Anwendungen. Das betrifft auch die Ansprache von spezifischen Schnittstellen wie Touch oder Face ID, Bluetooth und selbst Batterieverbrauch von iOS Geräten. Theoretisch können auch externe Dienstleister diese Teile der Implementierung übernehmen, jedoch bringt das weitere Entwicklungskosten und eine höheren Managementbedarf. Im Gegensatz dazu sind PWAs in der Entwicklung grundlegend unabhängig, da es sich dabei um normale Webanwendungen handelt, welche keine spezifische Entwicklungsumgebung benötigen.

Ferner muss betont werden, dass es sich bei React Native App trotz der Programmierung großer Teile in JavaScript im Endeffekt um zwei Technologien handelt, die der Entwickler beherrschen muss. Einerseits JavaScript für das Implementieren einer Anwendung mit React, andererseits das Wissen über native Entwicklung, Schnittstellen und betriebssystemabhängige Programmiersprachen, um gegebenenfalls plattformspezifischen Code zu ergänzen. Das wirkt sich insofern auf den Entwicklungsaufwand aus,

<pre> 61 <input 62 id="search" 63 type="text" 64 value={query} 65 onChange={(event) => setQuery(event.target.value)} 66 /> </pre>	<pre> 9 <TextInput 10 id="search" 11 type="text" 12 value={query} 13 onChangeText={onQueryChange} 14 style={style.input} 15 /> </pre>
--	---

(a) React App Texteingabe Implementierung (b) React Native Texteingabe Implementierung

Quelle: Eigene Darstellung

Abbildung 5.5: Vergleich Implementierung React PWA und React Native App

dass JavaScript-Entwickler, die keine noch Erfahrungen mit nativer App-Entwicklung haben, mehr Zeit für die Einarbeitung in die Thematik benötigen.

Ein besonderer Vorteil dieser Konstellation von Technologien ist jedoch, dass Teile der Logik und Aufteilung der Komponenten aus der React Web App in die React Native App übernommen werden können und somit Zeit gespart wird. Ein Beispiel dafür ist in der Abbildung 5.5 verdeutlicht. Hieran wird sichtbar, dass in vielen Fällen die Syntax nur abgeändert werden muss, um die React Version der Implementierung in React Native zu übertragen. Das ist vor allem dann eine Option, wenn im Laufe des Projekts klar wird, dass die Funktionalitäten, die PWAs aktuell anbieten, nicht ausreichen. Diese Art von flexibler Entwicklung bieten native Anwendungen grundsätzlich nicht.

5.4 Diskussion

In der Tabelle 5.4 soll nun auf Grundlage der dargestellten Argumente die Erfüllung der einzelnen Kriterien in Form von Punkten bewertet werden. Auf eine Gewichtung der Kriterien wurde durch den Anwendungsfall bewusst verzichtet. Die Bewertung beruht auf folgenden Definitionen:

- 0 Punkte: Das Kriterium konnte nicht erfüllt werden.
- 1 Punkt: Das Kriterium konnte teilweise erfüllt werden.
- 2 Punkte: Das Kriterium konnte vollkommen erfüllt werden.

Beim Kriterium der Funktionalität erreichen beide Apps für den definierten Anwendungsfall die maximale Punktzahl, da alle geforderten Funktionen mit beiden Technologien umgesetzt werden können. Die Kompatibilität mit verschiedenen Betriebssystemen war bei PWAs nicht vollkommen gegeben, da Safari, wie bereits erläutert, einige Funktionalitäten nicht unterstützt. Das Kriterium des Entwicklungsaufwands erfüllt die PWA wiederum besser, weil sowohl die Recherche- als auch die Implementierungszeit geringer

Tabelle 5.4: Nutzwertanalyse der beiden implementierten Anwendungen

Kriterium	PWA	Native App
Funktionalität	2	2
Kompatibilität mit vers. Betriebssystemen	1	2
Entwicklungsaufwand	2	1
Summe	5	5

war als bei der React Native App. Deshalb erhält die PWA in diesem Aspekt eine höhere Punktzahl.

Das Ergebnis zeigt, dass in diesem Anwendungsfall die Native App durch eine PWA ersetzt werden kann.

Nun sollen in einer weiteren Tabelle 5.5 die Erfüllung der Kriterien durch die zwei Technologien im Allgemeinen betrachtet werden. Hierbei wurde zusätzlich eine tatsächlich Native App für den Vergleich aufgenommen, denn diese unterscheidet sich insofern, dass nicht kompatibel mit verschiedenen Betriebssystemen ist. Auch der Entwicklungsaufwand ist höher, denn um dasselbe Ergebnis zu erreichen wie bei einer PWA oder React Native App muss für jedes Betriebssystem eine eigene App mit der jeweiligen Entwicklungsumgebung entwickelt werden. Im Gegensatz zur vorherigen Tabelle wird deutlich, dass PWAs weniger Funktionalitäten unterstützen als React Native oder Native Apps. Beispielsweise betrifft das in dieser Arbeit nicht behandelte Funktionen wie das Geofencing oder die User Idle Detection⁶ [?]. Die anderen Kriterien sind bei der PWA und React Native App gleichermaßen erfüllt.

Tabelle 5.5: Nutzwertanalyse der beiden Technologien im Allgemeinen

Kriterium	PWA	React Native App	Native App
Funktionalität	1	2	2
Kompatibilität mit vers. Betriebssystemen	1	2	0
Entwicklungsaufwand	2	1	0
Summe	4	5	2

Anhand des Vergleichs der beiden Nutzwertanalysen wird klar, dass PWAs nicht allgemein als Ersatz zu nativen Anwendungen gesehen werden können. Die Ergebnisse legen zudem zusammenfassend nahe, dass eine Entscheidung für eine der beiden Implementationen abhängig von den Anforderungen an die zu entwickelnde getroffen werden sollte.

⁶Das Erkennen, ob der Nutzer den Bildschirm aktiv betrachtet.

Kapitel 6

Fazit und Ausblick

6.1 Fazit

In dieser Arbeit wurden anhand der Kriterien Funktionalität, Kompatibilität mit verschiedenen Betriebssystemen und Entwicklungsaufwand PWAs mit Native Apps verglichen. Hierfür mit den Technologien jeweils eine App entwickelt, welche die COVID-19-Fallzahlen darstellt und filtern kann. Anhand dieses Anwendungsfalles wurde analysiert, welche Vor- und Nachteile das Entwickeln mit diesen Technologien hat und ob eine PWA die Native App ersetzen kann.

Dabei hat sich herausgestellt, dass PWAs mittlerweile viele Funktionalitäten von nativen Anwendungen wie Installierbarkeit, Offlinebetrieb, Standortzugriff, Kontaktzugriff und Benachrichtigungen umsetzen können und deshalb gerade für unkomplizierte Anwendungen eine Alternative bieten. Besonders attraktiv für Unternehmen ist dabei, dass es sich generell um eine für den Entwickler und Nutzer plattformunabhängige Lösung handelt, wodurch Ressourcen gespart werden können. Außerdem ist der Entwicklungsaufwand der *pwa* geringer als der einer nativen Anwendung.

Dennoch ist ein entscheidender Faktor für die Etablierung von PWAs die fehlende Unterstützung durch iOS, da dadurch 26,34 Prozent des weltweiten Marktanteils von mobilen Betriebssystemen weniger Funktionalitäten zur Verfügung stehen [44]. Dies betrifft wie in der Arbeit dargestellt die Push API und Notification API, ferner aber auch den Zugriff auf Bluetooth oder die Badging API. Doch in Anbetracht der Webschnittstellen, die sich seit dem Aufschwung von PWAs 2015 entwickelt haben, wird deutlich, dass die Zukunft von mobilen Anwendungen vielfältig ist.

6.2 Ausblick

Für anknüpfende Arbeiten könnten die Analyse von PWAs und React Native Apps intensiviert werden, indem umfassender auf die verschiedenen Funktionalitäten eingegangen wird, die mit diesen beiden Vorgehensweisen umsetzbar sind. Beispiele hierfür sind die

Einbindung der Background Sync API, Payment Request API oder Web Share Target API.

Interessant wäre außerdem noch ein Performance Vergleich. Da die für diese Arbeit implementierte Anwendung lediglich aus einer Seite mit Daten einer externen Schnittstelle besteht, hat sich der Vergleich nicht angeboten. Anders sieht das jedoch bei größeren Anwendungen aus, welche eine Vielzahl von Bildern, Videos und Einträgen nutzen und verwalten wie Twitter oder Pinterest. Auch die Ansprache der Schnittstellen eines mobilen Endgeräts sind unter dem Aspekt der Performance zu betrachten, denn dies geschieht performanter mit der betriebssystemspezifischen Programmiersprache.

Ein weiterer wichtiger Faktor, der in zukünftigen Arbeiten betrachtet werden sollte, ist die Sicherheit von Progressive Web Apps. Generell besteht eine grundlegende Sicherheit durch den Zugriff mit HTTPS, jedoch sollte hier auch die umfassend betrachtet werden, welche Maßnahmen zur Verbesserung der Sicherheit von Webanwendungen getroffen werden können. Durch die Position des Service Workers als Proxy ist dieser besonders anfällig für bösartige Angriffe [46]. Der Sicherheitsaspekt ist vor allem bedeutsam für mobilen Anwendungen, die sensible Daten verarbeiten.

Literaturverzeichnis

- [1] M. Iqbal, “App download and usage statistics (2020),” 06.05.2021.
- [2] A. Russell, “Progressive web apps: Escaping tabs without losing our soul,” 2015.
- [3] StatCounter, “Percentage of mobile device website traffic worldwide from 1st quarter 2015 to 1st quarter 2021,” 2021.
- [4] W. Jobe, “Native apps vs. mobile web apps,” *International Journal of Interactive Mobile Technologies (iJIM)*, vol. 7, no. 4, p. 27, 2013.
- [5] StatCounter, “Mobile operating system market share worldwide: Apr 2010 - apr 2021,” 2021.
- [6] M. Schickler, M. Reichert, and R. Pryss, *Entwicklung mobiler Apps: Konzepte, Anwendungsbauusteine und Werkzeuge im Business und E-Health*. eXamen.press, Berlin: Springer Vieweg, 2015.
- [7] o. V., “Mobile and desktop,” 2020.
- [8] MDN contributors, “Progressive web apps (pwas).”
- [9] M. Gaunt, “Service workers: an introduction.”
- [10] MDN contributors, “Service worker api.”
- [11] M. Wenzel, G. Warren, Y. Victor, P. Marcano, S. Ghosh, D. Pine, and S. Smith, “Choose between traditional web apps and single page apps (spas),” 2020.
- [12] S. Richard and P. LePage, “What makes a good progressive web app?.”
- [13] A. Russell, J. Song, J. Archibald, and M. Kruisselbring, “Service workers nightly,” 2021.
- [14] MDN contributors, “Web app manifest.”
- [15] C. Liebel, *Progressive Web Apps: Das Praxisbuch*. Rheinwerk Computing, Bonn: Rheinwerk Verlag, 1. auflage ed., 2019.
- [16] M. Lamouri, M. Cáceres, and J. Yasskin, “Permissions,” 2020.
- [17] R. Barger, “Is react a library or a framework? here’s why it matters,” 2021.

- [18] Facebook, “React without jsx.”
- [19] Facebook, “Who’s using react native.”
- [20] Facebook, “Introducing hooks.”
- [21] Facebook, “React native.”
- [22] Facebook, “Core components and native components.”
- [23] o. V., “React native internals,” o. J.
- [24] E. Behrends, *React Native: Native Apps parallel für Android und iOS entwickeln.* Heidelberg: O’Reilly, 1. auflage ed., 2018.
- [25] “Rki corona landkreise,” 2020.
- [26] M. Cáceres, A. Gustafson, M. Giuca, A. Kostiainen, M. Lamouri, and K. Rohde Christiansen, “Web application manifest,” 2021.
- [27] o. V., “Add to home screen (a2hs).”
- [28] “App store review guidelines,” 07.06.2021.
- [29] P. LePage, “Storage for the web: There are many different options for storing data in the browser. which one is best for your needs?,” 2020.
- [30] MDN contributors, “Client-side storage,” o. J.
- [31] J. Archibald, “The offline cookbook,” 2020.
- [32] R. Turner, “Releasing react native 0.59,” 2019.
- [33] M. Cáceres, “Geolocation api,” 2021.
- [34] o. V., “Firefox/ push notifications,” 2016.
- [35] P. LePage, T. Steiner, and F. Beaufort, “Add a web app manifest,” 2018.
- [36] A. Bijlani, U. Ramachandran, and R. Campbell, “Where did my 256 gb go? a measurement analysis of storage consumption on smart mobile devices,” *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, vol. 5, no. 2, pp. 1–28, 2021.
- [37] Google, “Twitter lite pwa significantly increases engagement and reduces data usage,” o. J.
- [38] M. Kruisselbrink, “Geofencing api,” 2017.
- [39] A. Deveria, “Push api,” 2021.

- [40] P. Beverloo and Rayan Kanso, “Contact picker api,” 2021.
- [41] M. Firtman, “ios 14.5 brings the new safari 14.1 to pwas and the web platform: What’s new, what’s missing, new challenges and new capabilities for iphone and ipad,” 2021.
- [42] MDN contributors, “Push api,” 2021.
- [43] MDN contributors, “Contact picker api,” 22.06.2021.
- [44] S. O’Dea, “Market share of mobile operating systems worldwide 2012-2021,” 2021.
- [45] A. Bar, “What web can do today? can i rely on the web platform features to build my app? an overview of the device integration html5 apis,” 2021.
- [46] J. Lee, H. Kim, J. Park, I. Shin, and S. Son, “Pride and prejudice in progressive web apps,” in *CCS’18* (D. Lie and M. Mannan, eds.), (New York, NY), pp. 1731–1746, Association for Computing Machinery, 2018.