



**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:  Vorname:  Matrikel-Nr.:

Fakultät:  Studiengang:

Sommersemester  Wintersemester

### 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.

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.

Ort, Datum, Unterschrift Studierende/Studierender

## Kurzdarstellung

In der vorliegenden Bachelorarbeit wird untersucht, ob und welche Vor- und Nachteile das Entwickeln einer Progressive Web App (PWA) – eine Webanwendung, die Funktionalitäten wie Offlinebetrieb und Installation unterstützt – im Gegensatz zu einer Native App bieten. Ferner soll damit die Frage beantwortet werden, ob sie diese ersetzen kann, um die Chancen von PWAs zu beurteilen. Um eine Grundlage für den Vergleich zu schaffen, findet hierbei eine plattformunabhängige Entwicklung der Native App statt.

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 der Installation und dem Offlinebetrieb die Funktionen Standortzugriff, Kontaktzugriff und Benachrichtigungen.

Die implementierten Funktionalitäten verdeutlichen, dass PWAs durch moderne Webschnittstellen an Potential gewinnen. Vorteile sind dabei die Unabhängigkeit von der Installation der Anwendung und der geringere Entwicklungsaufwand.

Jedoch ist eine entscheidende Schwachstelle von PWAs die fehlende Unterstützung einiger Funktionalitäten auf iOS Geräten.

Zusammenfassend zeigt das Ergebnis, dass PWAs nur bis zu einem gewissen Grad in der Lage sind, nativen Anwendungen zu ersetzen. Deshalb muss die Entscheidung für die Entwicklung einer PWA zudem abhängig von den Anforderungen des Anwendungsfalls betrachtet werden.

# Inhaltsverzeichnis

<b>Abbildungsverzeichnis . . . . .</b>	<b>vii</b>
<b>Tabellenverzeichnis . . . . .</b>	<b>ix</b>
<b>Listings . . . . .</b>	<b>xi</b>
<b>1 Einleitung . . . . .</b>	<b>1</b>
1.1 Zielsetzung . . . . .	1
1.2 Umfeld . . . . .	2
1.3 Aufbau der Arbeit . . . . .	2
<b>2 Grundlagen . . . . .</b>	<b>3</b>
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
<b>3 Kriterienkatalog zum Vergleich der Technologien . . . . .</b>	<b>13</b>
3.1 Funktionalitäten . . . . .	13
3.1.1 Installation . . . . .	13
3.1.2 Offlinebetrieb . . . . .	14
3.1.3 Standortzugriff . . . . .	15
3.1.4 Kontaktzugriff . . . . .	15
3.1.5 Benachrichtigung . . . . .	16
3.2 Kompatibilität mit verschiedenen Betriebssystemen . . . . .	17
3.3 Entwicklungsaufwand . . . . .	18
<b>4 Implementierung einer Progressive Web App und Native App . . . . .</b>	<b>19</b>
4.1 Einrichten der Entwicklungsumgebung . . . . .	20
4.2 Installation . . . . .	24
4.3 Offlinebetrieb . . . . .	27
4.4 Standortzugriff . . . . .	30

4.5 Kontaktzugriff . . . . .	32
4.6 Benachrichtigungen . . . . .	34
<b>5 Ergebnisse und Diskussion . . . . .</b>	<b>39</b>
5.1 Funktionalitäten . . . . .	39
5.2 Kompatibilität mit verschiedenen Betriebssystemen . . . . .	45
5.3 Entwicklungsaufwand . . . . .	48
5.4 Bewertung der Technologien anhand des Kriterienkatalogs . . . . .	51
<b>6 Fazit und Ausblick . . . . .</b>	<b>55</b>
6.1 Fazit . . . . .	55
6.2 Ausblick . . . . .	56
<b>Literaturverzeichnis . . . . .</b>	<b>57</b>

# 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 Ansicht 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 einem Android Gerät . . . . .	46
5.2 Kompatibilität der Progressive Web App mit verschiedenen Browsern auf einem iOS Gerät . . . . .	47
5.3 Entwicklungsaufwand der Progressive Web App (PWA) und der Native App . . .	49
5.4 Nutzwertanalyse der beiden implementierten Anwendungen . . . . .	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

**PWA** Progressive Web App

**SPA** Single-Page Application

**SDK** Software Development Kit

**UI** User Interface

**URL** Uniform Resource Locator



# 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 218 Billionen betrug [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, das sich die Universalität des Webs zu nutzen macht, ist das der 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 Installation.

Laut dem Online-Statistikportal *Statcounter* nutzen 4,32 Billionen Menschen das Internet über mobile Endgeräte [3]. Aufgrund der Vielzahl der Nutzer und der damit verbundenen enormen Technologiepotentials für die Zukunft lohnt sich der Vergleich von PWA und Native App – beides Technologien, die in mobile Endgeräten Anwendungen finden.

## 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 - eine plattformunabhängige Programmierung ermöglicht. Diese Besonderheit wurde in bisherigen Arbeiten noch nicht betrachtet.

Außerdem sollen dabei auch die aktuellen Chancen 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 diesen 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.

## 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 andererseits auf eine spezielle Form von Web Apps, genannt PWAs, eingegangen.

### 2.1 Native Applikationen

Als native Applikationen werden Anwendungen bezeichnet, die plattformspezifisch – ergo speziell für ein Betriebssystem – implementiert sind. 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-Komponenten<sup>1</sup>, 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 eine erneute Überprüfung und Veröffentlichung der App.

---

<sup>1</sup>User Interface Komponenten, z. Dt. Benutzeroberflächenkomponenten

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

## 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 Object Notation (JSON)<sup>3</sup>-Datei, in der Einzelheiten zur Anwendung und deren Installation angegeben werden [10]. Da es sich dabei,

---

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

<sup>3</sup>Programmiersprachen-unabhängiges Datenformat

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 API<sup>4</sup> signalisiert. Dadurch kann wiederum geprüft werden, ob andere Schnittstellen z. B. die *Geolocation*<sup>5</sup> 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.dev<sup>6</sup> 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 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

---

<sup>4</sup>z.Dt. Programmierschnittstelle.

<sup>5</sup>Eine Schnittstelle zur Bestimmung und Beobachtung des aktuellen Standorts des Nutzers.

<sup>6</sup>Eine von Google Developers unterstützte Webseite mit einer Vielzahl von Artikeln zu Webentwicklungs-technologien.

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 W3C<sup>7</sup> 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].

Mittlerweile bietet Google, unter anderem zur vereinfachten Implementierung und Verwaltung von Service Workern, das Tool *Workbox* an. Es handelt sich dabei um eine Bibliothek,

---

<sup>7</sup>World Wide Web Consortium, eine Organisation, welche Spezifikationen für das Web veröffentlicht.

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 Toolchains<sup>8</sup> 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. 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

---

<sup>8</sup>Eine Sammlung von Werkzeugen, die zum unkomplizierten Aufsetzen eines Produkts dienen.

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.

```

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

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 und verschiedenen Frameworks oftmals 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].

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 JSX 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 Entwickler 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 ES6<sup>9</sup>-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.

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

---

<sup>9</sup>ECMAScript 6, eine 2015 veröffentlichte Version von ECMAScript.

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 meistgenutzten 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 diese Aufteilung abgebildet.



Quelle: basierend auf [23]

Abbildung 2.1: Architektur React Native

Ersteres bietet in 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 JS-Umgebung und Native Module verantwortlich. Sie erhält von letzterem Events wie Toucheingaben oder Netzwerkanfragen und vom JS Module Anweisungen für das UI oder APIs. Der Austausch findet dabei asynchron über JSON statt. Der letzte Teil der Architektur ist das Native Module, welches dem Nutzer als UI zur Verfügung [23].

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 Kommunikation mit nativen Schnittstellen erst zur Laufzeit geschieht.

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 plattformspezifischen 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.

## 3 Kriterienkatalog zum Vergleich der Technologien

Nach der Erläuterung der Grundlagen wird nun auf die Kriterien eingegangen, mithilfe denen die Technologien im Verlauf der Arbeit verglichen werden: Funktionalität, Kompatibilität mit verschiedenen Betriebssystemen 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.

Anhand von Punkten wird die Erfüllung der Kriterien bemessen und zuletzt ausgewertet. Dabei erfolgt bewusst ein Verzicht auf eine Gewichtung der Kriterien. Der Grund hierfür ist, dass es sich in der vorliegenden Arbeit um einen Vergleich der Technologien handelt und deshalb alle Kriterien gleichbedeutend 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 mobilen Anwendungen vorgestellt, die für den Vergleich ausgewählt wurden.

#### 3.1.1 Installation

##### Beschreibung

Die Apps, die der Nutzer oft verwendet, sollten schnell erreichbar sein. Eine Installation wird daher in vielen Fällen bevorzugt und stellt eine grundlegende Funktion von mobilen Anwendungen dar.

Dennoch bedeutet das Installieren für das verwendete Gerät, 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.

### Kriterium

Mit diesem Kriterium soll geprüft werden, ob die Anwendung zur Nutzung installiert werden kann oder muss.

- 0 Punkte: Die Nutzung der Anwendung ist abhängig von deren Installation.
- 1 Punkt: Die Nutzung der Anwendung ist unabhängig von deren Installation.

### 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 unterschieden werden zwischen Anwendungen, die generell keinen Zugriff auf das Internet benötigen und jenen, die ihre Funktionalität im Offlinebetrieb einschränken. Vorteil von letzterem ist auch, wenn die Prozesse, die im Offlinebetrieb angestoßen wurden, gehalten werden können, bis das Gerät wieder eine stabile Internetverbindung besitzt. Auf diesem Wege wird gewährleistet, dass jegliche Aktivitäten erfolgreich durchgeführt werden. Meistens bieten Applikationen eine Mischung aus beiden Optionen an.

Um aus technischer Sicht eine Unabhängigkeit von der Netzwerkverbindung zu schaffen, müssen Daten lokal in einem Speicher des Geräts abgelegt werden. Das ermöglicht dem Nutzer, seine Daten jederzeit verfügbar zu haben und somit Abhängigkeiten von äußeren Umständen zu minimieren.

### Kriterium

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

- 0 Punkte: Die Anwendung ist nach der Installation ohne Netzwerkverbindung nicht lauffähig.

- 1 Punkt: Die Anwendung ist nach der Installation ohne Netzwerkverbindung lauffähig.

### 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 die Verwendung 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

Dieses Kriterium betrachtet, ob die Anwendung Zugriff auf den Standort des Nutzers besitzt und diesen weiterverarbeiten kann.

- 0 Punkte: Die Anwendung besitzt keinen Zugriff auf den Standort des Nutzers.
- 1 Punkt: Die Anwendung besitzt Zugriff auf den Standort des Nutzers.
- 2 Punkte: Die Anwendung besitzt Zugriff auf den Standort des Nutzers und kann diesen selbstständig weiterverarbeiten, beispielsweise in Form von Geofencing<sup>1</sup> oder Reverse Geocoding<sup>2</sup>.

### 3.1.4 Kontaktzugriff

#### Beschreibung

Ein 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.

---

<sup>1</sup>Geofencing bezeichnet das Auslösen von Benachrichtigungen beim Betreten oder Verlassen von definierten Bereichen und Orten.

<sup>2</sup>Der Prozess, bei dem aus Informationen über Längen- und Breitengrad eine lesbare (Teil-)Adresse.

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

### **Kriterium**

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

- 0 Punkte: Die Anwendung besitzt keinen Zugriff auf die Kontakte des Endgeräts.
- 1 Punkt: Die Anwendung besitzt Zugriff auf die Kontakte des Endgeräts.
- 2 Punkte: Die Anwendung besitzt Zugriff auf die Kontakte des Endgeräts und kann Kontakte hinzufügen, verändern und löschen.

### **3.1.5 Benachrichtigung**

#### **Beschreibung**

Benachrichtigungen sind heutzutage Bestandteil jeder App und spielen eine große Rolle bei der Interaktion mit den 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.

Benachrichtigung können kurze Informationen durch Texte, Bilder oder Buttons beinhalten. Letzteres ist besonders wichtig, da der Nutzer bei Push Benachrichtigungen über Buttons aus dem Menü heraus mit der Anwendung interagieren und auch auf diese weitergeleitet werden kann.

Außerdem könnten Benachrichtigungen Töne auslösen, um auf sich aufmerksam zu machen oder dem Icon der Anwendung ein Badge<sup>3</sup> anheften.

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

---

<sup>3</sup>Badges sind kleine Anzeigen am rechten oberen Rand des App Icons. An ihnen erkennt man die Anzahl an ungeöffneten Benachrichtigungen.

**Kriterium**

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

- 0 Punkte: Die Anwendung kann keine Benachrichtigungen erhalten.
- 1 Punkt: Die Anwendung kann nicht-persistente Benachrichtigungen erhalten.
- 2 Punkte: Die Anwendung kann persistente Benachrichtigungen erhalten.

## 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 Betriebssystem, 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 Browsern 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 Browsern funktionieren müssen. Dabei ist ausschlaggebend, wie viele der bereits definierten Funktionalitäten auf dem Browser oder Betriebssystem verfügbar sind.

- 0 Punkte: Die Funktionalitäten der Anwendung sind mit einem Betriebssystem kompatibel und abhängig vom Browser.
- 1 Punkt: Die Funktionalitäten der Anwendung sind mit einem Betriebssystem kompatibel und unabhängig vom Browser.
- 2 Punkte: Die Funktionalitäten der Anwendung sind mit mehreren Betriebssystemen kompatibel und abhängig vom Browser.
- 3 Punkte: Die Funktionalitäten der Anwendung sind mit mehreren Betriebssystemen kompatibel und unabhängig vom Browser.

### 3.3 Entwicklungsaufwand

#### Beschreibung

Bei der Implementierung von Anwendungen, egal ob für das Web oder für mobile Endgeräte, ist der Aufwand der Entwicklung ein maßgebender Faktor für resultierende Kosten. Dabei fließen außerdem der Entwicklungszeitraum, die Fähigkeiten des Teams und der Funktionsumfang in die Kalkulation ein. Wichtig ist, wie im vorherigen Kriterium angemerkt, auch die Kompatibilität mit verschiedenen Endgeräten. Denn je mehr Geräte unterstützt werden sollen, desto aufwendiger und somit kostspieliger 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 Browsern ein zusätzlicher Entwicklungsaufwand entsteht.

#### Kriterium

Auf Basis der bereits genannten Kriterien soll in diesem Kriterium gemessen werden, wie viel Zeit für deren Implementierung 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.

- 0 Punkte: Der Einarbeitungs- und Programmieraufwand benötigt bei beiden Technologien gleich viel Zeit.
- 1 Punkt: Der Einarbeitungs- oder Programmieraufwand benötigt weniger Zeit als bei der anderen Technologie.
- 2 Punkte: Der Einarbeitungs- und Programmieraufwand benötigt weniger Zeit als bei der anderen Technologie.

## 4 Implementierung einer Progressive Web App und Native App

Die Daten für die zu implementierenden Applikationen werden von der offiziellen REST<sup>1</sup> 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[24].

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 ein 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 Konzept von React. Andererseits sind durch die Verwendung von React Native beide Technologien plattformunabhängig und bilden somit eine solide Grundlage für den Vergleich. Diese Eigenheit

---

<sup>1</sup>Ein gängiges Datenformat des Datenaustauschs im Web.

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 npm<sup>2</sup> mit dem Befehl `npx create-react-app <Name der Anwendung>` erstellt. Hierfür wird mindestens Node<sup>3</sup> 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*<sup>4</sup>, *ESLint*<sup>5</sup> und *Webpack*<sup>6</sup>. Create-React-App bietet außerdem eine Reihe an Templates, die das Aufsetzen einer zum Projekt passenden Entwicklungsumgebung ebenfalls erleichtern sollen. Beispielsweise das PWA Template, das Werkzeuge wie *Workbox*<sup>7</sup> und einen vorimplementierten Service Worker mitinstalliert. Darauf wird in dieser Anwendung verzichtet, um die

---

<sup>2</sup>ehemals Node Package Manager, Anwendung zur Installation von Node.js Packages.

<sup>3</sup>Node.js ist eine lizenzfreie und plattformunabhängige JavaScript Entwicklungsumgebung

<sup>4</sup>Compiler, der mit ECMAScript 2015 geschriebenen Code rückwärtskompatibel macht durch dafür vorsehene Polyfills.

<sup>5</sup>Codeanalysetool, welches den Entwickler auf problematische Codestellen aufmerksam macht.

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

<sup>7</sup>Von Google entwickelte Sammlung von JavaScript Bibliotheken zur vereinfachten Implementierung von Service Workern.

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.

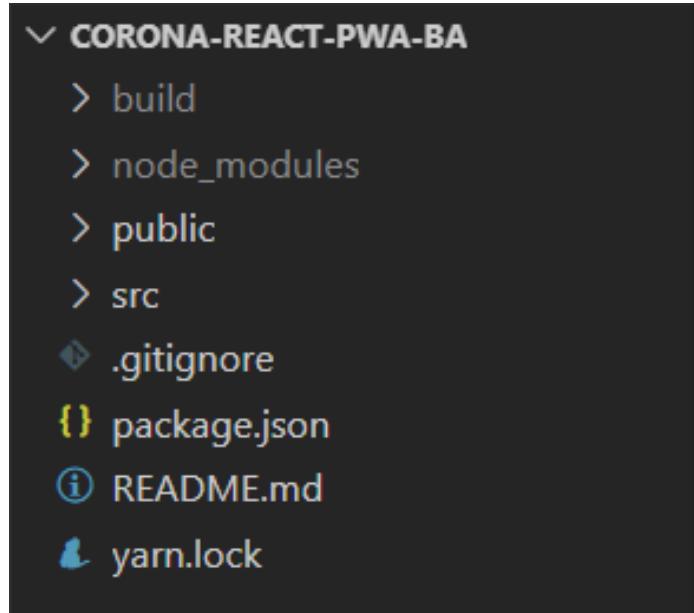


Abbildung 4.1: Ordnerstruktur Progressive Web App

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 *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 Webser-

ver 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.

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.

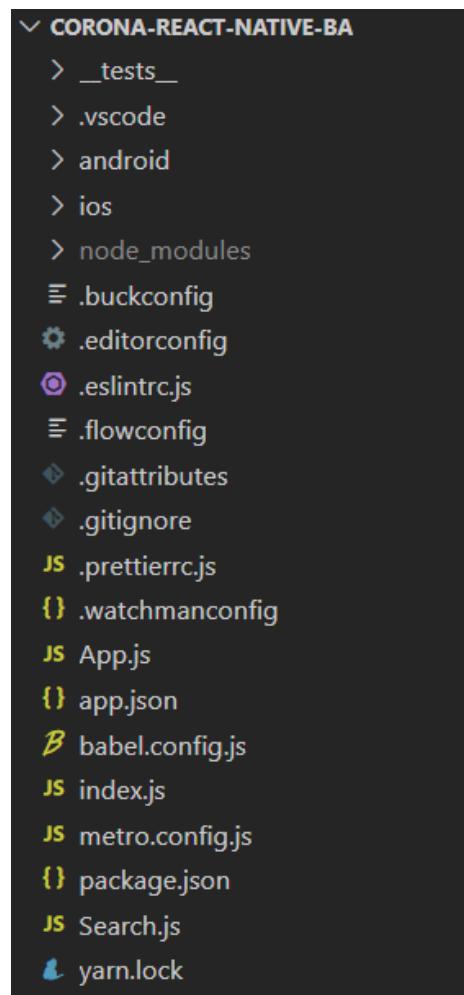
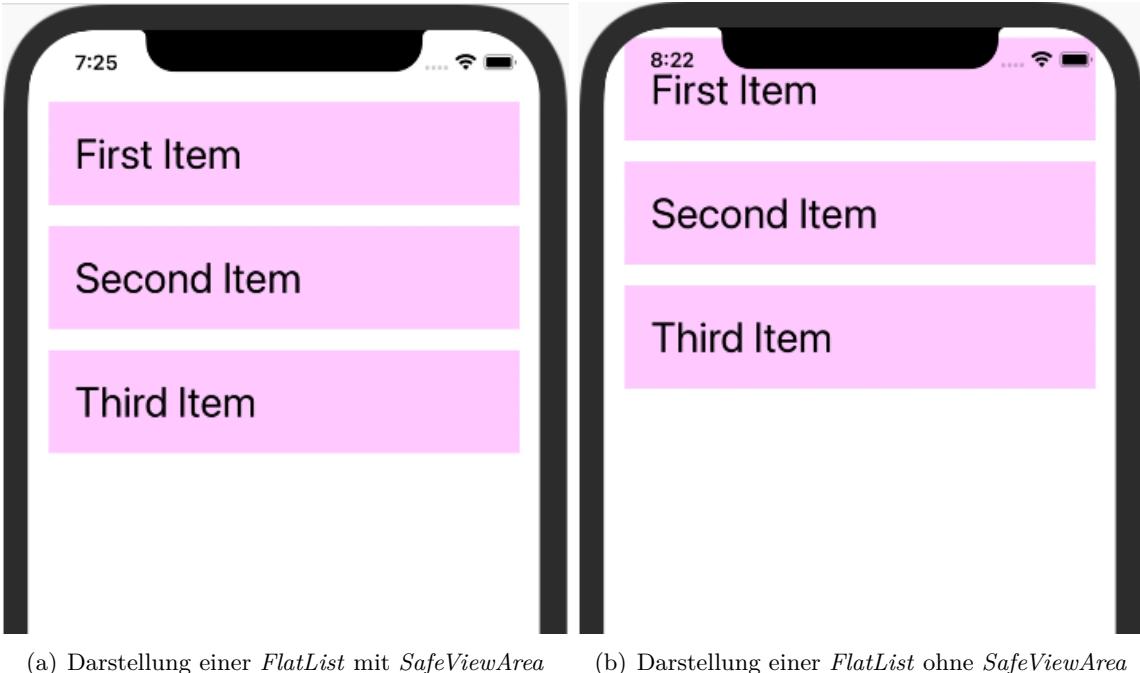


Abbildung 4.2: Ordnerstruktur React Native App

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 rendernt, 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 *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\_*



(a) Darstellung einer *FlatList* mit *SafeAreaView*      (b) Darstellung einer *FlatList* ohne *SafeAreaView*

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

*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<sup>8</sup> 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 Installation

Das Ziel dieser Funktionalität ist es, dass die Anwendung auf das mobile Endgerät des Nutzers installiert werden kann.

### 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 Browerversionen, 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

<sup>8</sup>Die IDE zur Entwicklung von iOS Anwendungen.

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 [25].

```

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": "#fffff"
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 Browern und Betriebssystemen komplett unterstützt wird. Aktuell betrifft das Safari, welches das Installieren aus Browern, die auf iOS statt mit ihrem eigenen HTML-Renderer mit WebKit<sup>9</sup> laufen (Chrome, Firefox und Opera), nicht unterstützt [26]. Denn jede iOS App muss laut Punkt 2.5.6 der Apple App Store Review Guidelines als Engine WebKit nutzen [27].

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 Brower auf Android außerdem eine Installationsaufforderung für die App am unteren Bildschirmrand. Diese zusätzliche Funktionalität muss aktuell für den Safari Brower explizit implementiert werden.

Für das Ermöglichen von Aktualisierungen der PWA muss keine weitere Programmierung vorgenommen werden. Sobald ein neues Deployment<sup>10</sup> 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 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.

---

<sup>9</sup>HTML-Renderer von Safari.

<sup>10</sup>Veröffentlichung der Webseite.

## 4.3 Offlinebetrieb

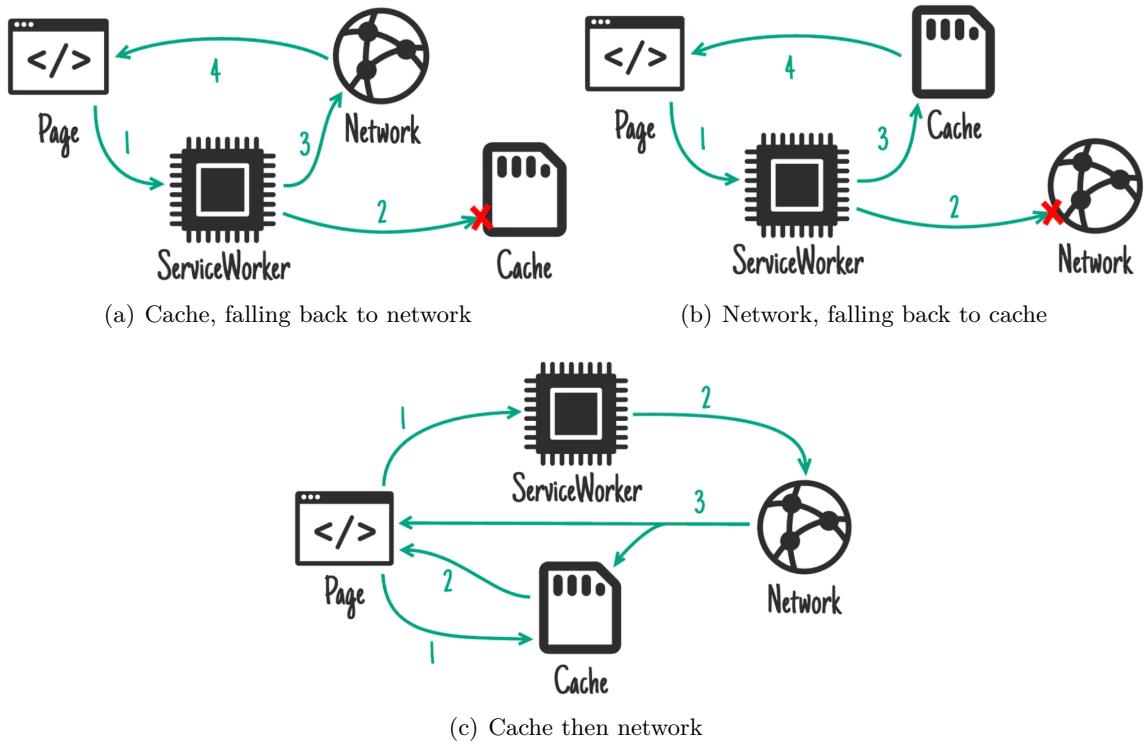
Die Anwendung soll auch ohne Internetverbindung aufrufbar, mit Daten gefüllt und die Funktion des Durchsuchens nutzbar sein.

### 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 [28]. 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 [29]. Die Speichergröße von beiden Speichern ist abhängig von dem Browser, in dem die Webanwendung aufgerufen wird.

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 [30].



Quelle: [30]

Abbildung 4.4: Speichermanagementstrategien

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 [30]. 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*<sup>11</sup> und *NetInfo*<sup>12</sup>. Sie wurden jedoch nicht entfernt, sondern lediglich aus der Kernbibliothek react-native exkludiert und in jeweils eigene Bibliotheken mit eigenen Verantwortlichen verschoben [31]. Diese Bibliotheken sind dem GitHub Repository @react-native-community zu finden und benötigen nach der Installation per npm auch das React Native-spezifische *Linkings*<sup>13</sup>.

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 [23]. 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

<sup>11</sup><https://github.com/react-native-async-storage/async-storage>

<sup>12</sup><https://github.com/react-native-netinfo/react-native-netinfo>

<sup>13</sup>Ein 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.

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.

## 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 [32].

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.

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

Über eine kostenfreie Drittanbieter API wird durch den Prozess des Reverse Geocodings 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.

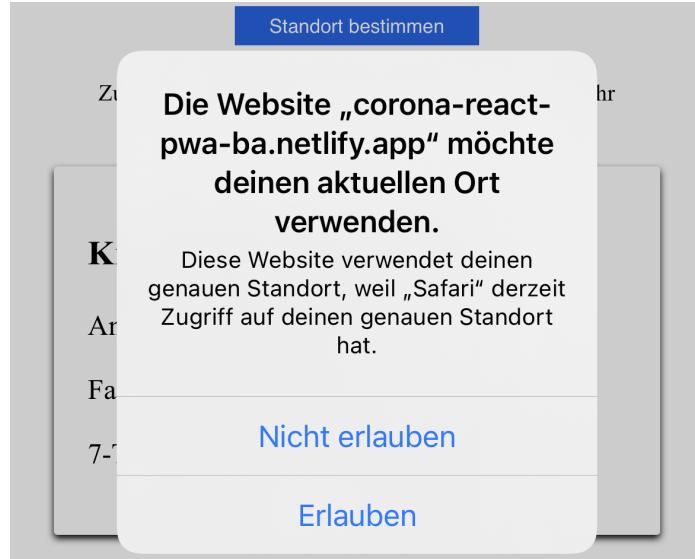


Abbildung 4.5: Erfragung des Standortzugsriffs im iOS Browser

### React Native App

Zur Implementierung des Standortzugsriffs in React Native wird die Community Lösung *react-native-geolocation-service*<sup>14</sup> 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 [33].

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 Standortzugsriffs 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

<sup>14</sup><https://github.com/Agontuk/react-native-geolocation-service>

bei Android die Geocoder 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

Dem Nutzer soll es möglich sein, die Liste der Landkreise nach der Adresse eines Kontakts zu filtern.

### 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 vorhanden 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 zugreifen. 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 Attributnamen, 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`<sup>15</sup> 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 `async`-Funktion erledigt. Sie wird in einem *Modal* aufgerufen, in dem nach dem Namen des gesuchten Kontakts gefragt wird. Nach der Betätigung des *Bestätigen*-Buttons

<sup>15</sup><https://github.com/morenoh149/react-native-contacts>

wird nun mit der Funktion `Contact.getContactsMatchingString(text)`, die als Übergabeparameter einen *string* annimmt, ein Kontakt gesucht, der diesen Übergabeparameter 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.

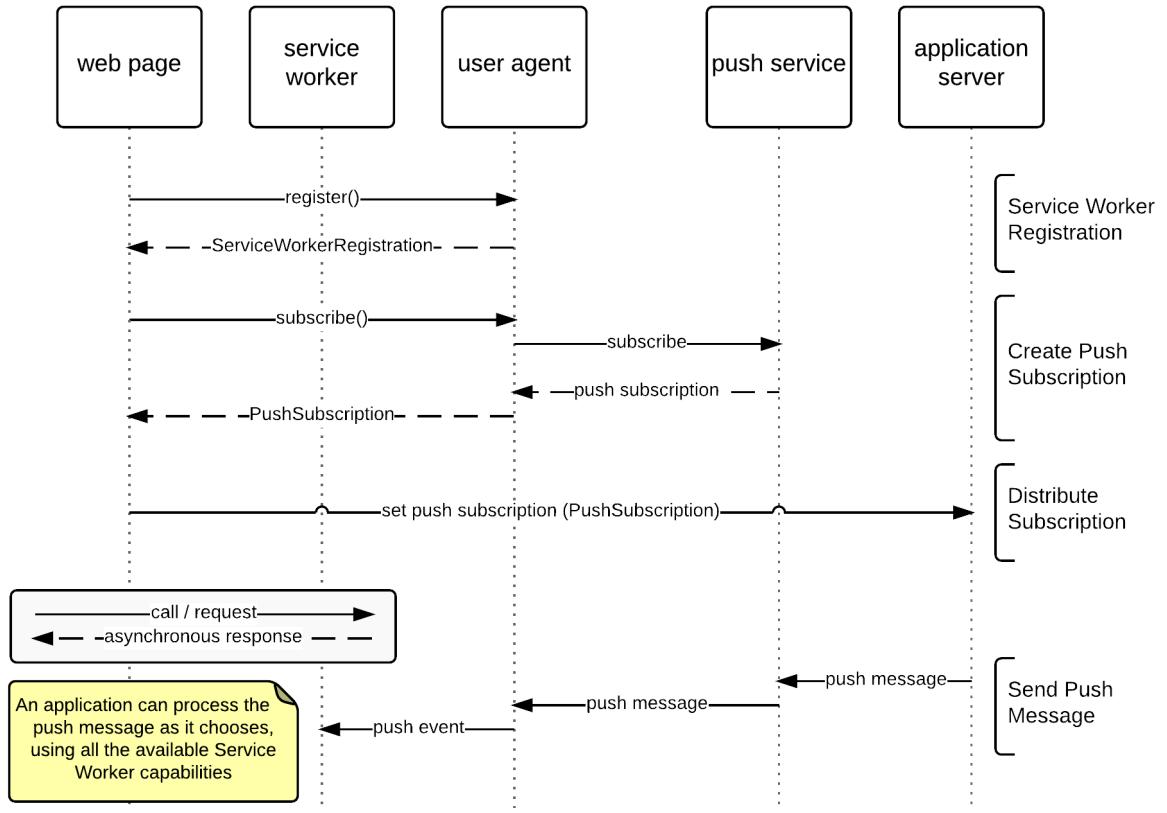
Wie in Kapitel 2 beschrieben, lassen sich Benachrichtigungen in nicht-persistente und persistente Benachrichtigungen unterteilen. Im Zuge dieser Arbeit soll nun erklärt werden, wie letzteres umgesetzt werden kann, da diese eine höhere Relevanz zur Interaktion mit dem Nutzern bieten und hierfür auch Schnittstellen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

### Progressive Web App

Zum besseren Verständnis der Implementierung soll zuerst das Web Push Prinzip 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.



Quelle: [34]

Abbildung 4.6: Ausschnitt eines Sequenzdiagramms des Web Push Prinzips

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 getätigert 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 IETF<sup>16</sup>-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

<sup>16</sup>Internet Engineering Task Force. Eine Organisation, die Web-Standards entwickelt.

API abgedeckt, welcher 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 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.

Eine Besonderheit von der Nutzung des FCM ist außerdem, dass dem App Manifest das Attribut `gcm_sender_id` mit dem Wert der Sender ID hinzugefügt werden muss. Dieser steht unter den Einstellungen des *Firebase*-Projekts.

## 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*<sup>17</sup> 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 insofern, 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.

---

<sup>17</sup><https://github.com/invertase/react-native-firebase>



## 5 Ergebnisse und Diskussion

Die Evaluierung der beiden Entwicklungsansätze PWA und Native App erfolgt nun auf Basis des in Kapitel 3 dargestellten Kriterien. Hierbei soll erläutert werden, ob und welche Vor- und Nachteile das Implementieren einer PWA im Gegensatz zu einer Native App darstellt. Im Anschluss findet eine Bewertung der Technologien auf Basis der Erfüllung der Kriterien statt. Dadurch soll die Frage beantwortet werden, ob die PWA die Native App zum jetzigen Zeitpunkt ersetzen kann.

### 5.1 Funktionalitäten

In diesem Kapitel wird darauf eingegangen, ob Schnittstellen zur Implementierung der geforderten Funktionalitäten vorhanden sind und wie viele Möglichkeiten diese anbieten. Dabei sollen auch die Vor- und Nachteile der Implementierung beschrieben werden.  
Die Kompatibilität dieser Funktionalitäten mit verschiedenen Betriebssystemen ist Teil des nächsten Kapitels.

#### Installation

Bislang war die Installierbarkeit ein signifikanter Unterschied zwischen nativen Apps und Web Applikationen. Durch den Service Worker und dem Manifest ist dies nun auch bei letzterem 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, dass der Nutzer automatisch durch einen Banner am unteren Bildschirmrand darauf hingewiesen wird, dass er die Anwendung herunterladen kann. Bei Android Geräten erscheinen die Apps außerdem nach der Installation durch das generierte WebAPK zusätzlich 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 dem Safari Browser das Teilen-Symbol geklickt und die Option „Zum Home-Bildschirm“ gewählt. Hierbei handelt es sich jedoch im Gegensatz zu Android lediglich um einen App-Shortcut, welcher bei allen Webseiten, die mit Safari aufgerufen werden, möglich ist. Die

Anwendung ist 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 App ist dann über den App-Drawer aufrufbar und wird in den Einstellungen als installierte App angezeigt. Um den Weg zum Aufrufen der Anwendung zu verkürzen, kann sie auch auf den Startbildschirm verknüpft werden.

Die Apps können dann, wie in Abbildung 5.1 abgebildet, über den Startbildschirm aufgerufen werden.



(a) React Native App und PWA im App-Drawer eines Android Geräts (b) PWA auf dem Startbildschirm eines iOS Geräts

Abbildung 5.1: Ansicht der installierten Apps

Hier ist die PWA in iOS optisch nicht von anderen Apps 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 Browserversionen wie Chrome 91 entfällt auch dieses Symbol, wodurch PWAs auf Android Geräten ebenfalls wie Native Apps aussehen. Mittlerweile können auch Begrüßungsbildschirme<sup>1</sup> 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 instal-

<sup>1</sup>Wenn 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.

liert 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 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 lediglich jene heruntergeladen werden, die für den ersten Aufruf der Anwendung benötigt werden. Im Gegensatz dazu gibt es gerade bei der React Native App 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 SEO<sup>2</sup> als auch ASO<sup>3</sup> durchführen.

Der Apple App Store unterstützt die Veröffentlichung von PWAs aktuell nicht.

Die Unabhängigkeit vom App Store kann auch 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.

---

<sup>2</sup>SEO ist kurz für Suchmaschinenoptimierung.

<sup>3</sup>Kurz für App-Store-Optimierung.

## Offlinebetrieb

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 deren Antwort 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 vermeintlich höheren Anzahl an Netzwerkanfragen durch die „Network, falling back to cache“-Strategie nicht mehr Ressourcenanfragen gestellt als bei der PWA.

Die Anwendungen vereint die Tatsache, 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.

Ferner ist es in beiden Fällen 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. Dieselbe Funktionalität kann auch in React Native mit einer Bibliothek umgesetzt werden, die in iOS auf eine ios-spezifische *Background Fetch*-Technik und in Android auf React Natives *Headless JS* zurückgreift.

## 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 Geofencing nicht möglich. Ein Entwurf einer Spezifikation zur Implementierung einer solchen Funktion von 2017 ist von W3C als obsolet markiert [38].

Gerade für das Geofencing 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.

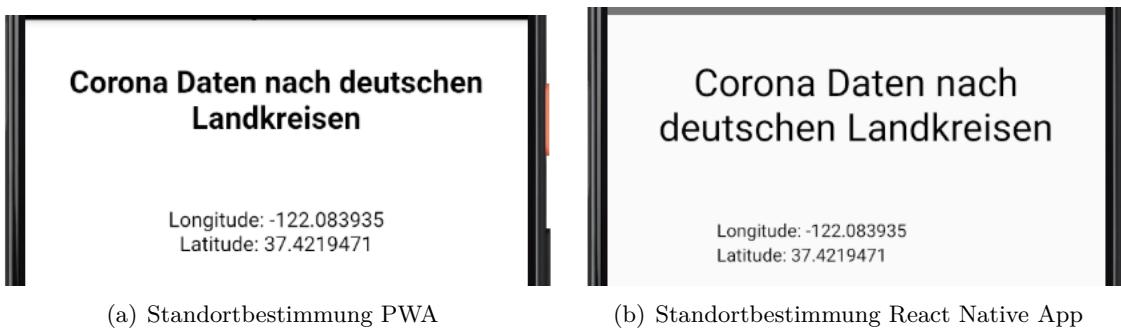


Abbildung 5.2: Darstellung der bestimmten Koordinaten

## 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.

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

Ferner ist die Contact Picker API in iOS Geräte beschränkter, da sie dort lediglich auf die Kontaktdaten *name*, *email* und *tel* (Telefonnummer) zugreifen kann [39].

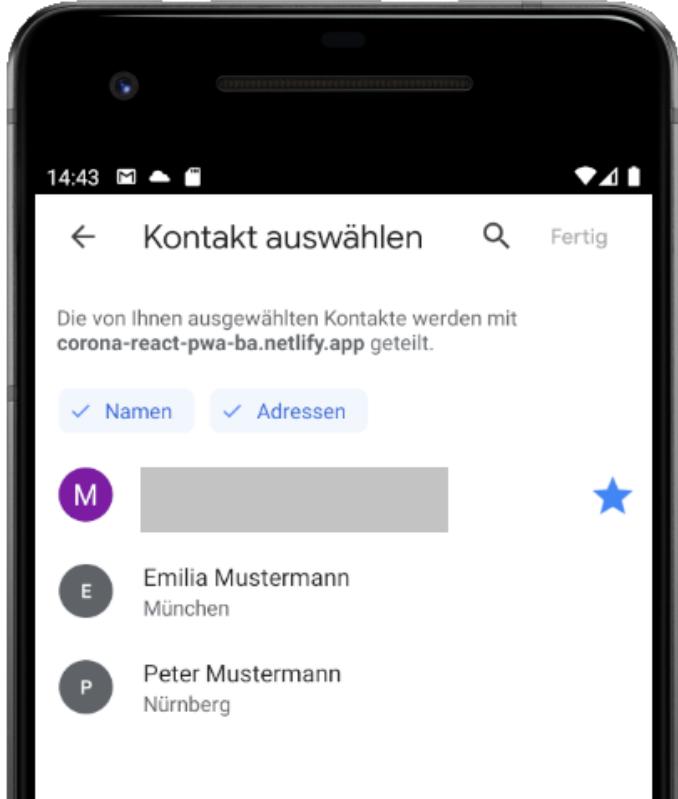


Abbildung 5.3: Benutzeroberfläche der Contact Picker API in Chrome

### 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 getätigten werden können. Das betrifft beispielsweise das Beantworten einer Nachricht oder das Löschen einer E-Mail.

Dennoch wird bei den Benachrichtigungen eine Schwachstelle von PWAs deutlich. IOS ermöglicht 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 iOS verfügbar ist [40]. Unter dem “WebKit

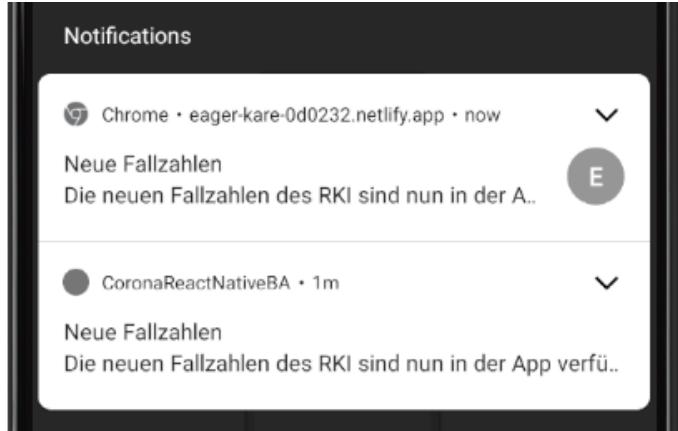


Abbildung 5.4: Vergleich Benachrichtigungen PWA (oben) und React Native App (unten)

Feature Status der offiziellen WebKit Webseite wird durch das Fehlen der Benachrichtigungsfunktion deutlich, dass auch für die Zukunft dahingehend keine Bemühungen erfolgen. Somit können PWAs etwa 26 Prozent des Marktanteils von mobilen Betriebssystemen nicht mit dieser Funktionalität bedienen werden [5].

### Zusammenfassung

Generell besitzt das Web viele Schnittstellen, um die im Kapitel 3 definierten Funktionalitäten umzusetzen. Lediglich die Push API und die Notification API zur Umsetzung der Push Benachrichtigungsfunktion existiert nicht für iOS Geräte.

Bei der Implementierung der Funktionalitäten wird zudem deutlich, dass die beiden Technologien sich grundlegend unterscheiden. Demnach können PWAs nicht beliebige Schnittstellen von mobilen Geräten aufrufen, sondern nutzen lediglich Webschnittstellen. Diese befinden sich noch im Aufbau und sind deshalb nicht umfassend, als experimentell markiert oder existieren nicht.

Wichtig ist darüber hinaus anzumerken, dass mittlerweile immer mehr Funktionalitäten im Web zur Verfügung stehen. Ein Beispiel hierfür ist der Kontaktzugriff, welcher im Chrome Browser seit der Version 80 implementiert ist [41]. Die Kompatibilität mit verschiedenen Betriebssystemen und Browern spielt jedoch die entscheidende Rolle bei der Nutzbarkeit dieser verfügbaren Funktionalitäten. Auf diese Thematik wird deshalb im folgenden Kapitel eingegangen.

## 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 14.7 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 ist. 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 Browser.

Für die Installation durch den Opera Browser gibt es von *MDN Web Docs* keine Angaben über die Browerversion. Diese Funktionalität 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
Installation	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)	-
Benachrichtigungen	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 Funktionalitäten Installation und Benachrichtigungen unterstützt. Das ist damit zu begründen, dass iOS diese Browser mit WebKit rendert, statt mit den eigenen HTML-Renderern (z.B. Blink für Chrome). Denn jede iOS App muss laut Punkt 2.5.6 der Apple App Store Review Guidelines als Engine WebKit nutzen [27]. Selbiges gilt auch für den Kontaktzugriff im Chrome und Opera Browser, während die Funktion beim Firefox Browser wie in der Tabelle 5.1 allgemein nicht unterstützt wird.

Der Safari Browser bietet den Kontaktzugriff seit iOS Version 14.15 und Safari 14.1 als experimentelle Funktion an. In der implementierten Anwendung konnte die Schnittstelle

aufgrund unzureichender Funktionalität nicht erreicht werden, weswegen in der Tabelle an dieser Stelle „teilweise“ vermerkt ist.

Für die Funktionalität der Push Benachrichtigungen besteht aktuell keine Schnittstelle in Safari auf iOS Geräten, weswegen die Kompatibilität hier nicht beurteilt werden kann.

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

<b>Funktionalität</b>	<b>Browser auf iOS Gerät</b>			
	Chrome	Firefox	Opera	Safari
Installation	nein	nein	nein	ja (11.3)
Offlinebetrieb	ja (40)	ja (39)	ja (27)	ja (11.3)
Standortzugriff	ja (18)	ja (4)	ja (11)	ja (3)
Kontaktzugriff	nein	nein	nein	teilweise (14.7)
Benachrichtigungen	nein	nein	nein	-

Insgesamt wird deutlich, dass die Unterstützung von PWAs auf Apple Geräten sich, seit iOS 11.3 verbessert hat, weil dadurch Schnittstellen wie der Service Worker, das App Manifest oder Cache API verfügbar sind.

Wenn nun die Standardbrowser des jeweiligen Betriebssystems betrachtet wird, kann außerdem festgestellt werden, dass die Kompatibilität mit verschiedenen Betriebssystemen bei jenen Funktionen, die verfügbar sind, gegeben ist.

## 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 angeschlossenen 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.

Generell gilt jedoch, dass native Anwendungen nur auf dem Betriebssystem installiert werden können, für das sie programmiert wurden. Die Möglichkeit der Kompatibilität mit verschiedenen Betriebssystemen ist eine Besonderheit der Implementierung mit dem plattformunabhängigen Framework React Native.

## Zusammenfassung

Obwohl bei der Entwicklung einer PWA die Universalität des Webs ein vermeintlicher Vorteil ist, sind sie dennoch nicht komplett unabhängig von dem Betriebssystem, mit dem sie aufgerufen werden. Dies wird deutlich bei Funktionalitäten wie dem Kontaktzugriff oder der Installation, welche in Chrome und Firefox auf einem iOS Gerät nicht funktionieren.

Eine React Native App verspricht mehr Kompatibilität als eine Native App und ist generell unabhängig vom Browser. Bei der Implementierung der App konnte dies jedoch nicht nachgewiesen werden, da die iOS App 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.

Native Apps selbst sind nach wie vor betriebssystemabhängig, wodurch sich die Kompatibilität mit verschiedenen Betriebssystemen erübrigkt.

Zusammenfassend bedeutet dies, dass PWAs in Bezug auf Kompatibilität mit verschiedenen Betriebssystemen Native Apps voraus sind, da sie generell auf beiden betrachteten Betriebssystemen lauffähig sind. Durch Ihre Abhängigkeit vom Browser ist die Kompatibilität dennoch nicht umfassend. 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 Entwicklungsaufwand 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.

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 Vorkehrungen zur Einrichtung der Umgebungsentwicklung getätigkt 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-ba`“<sup>4</sup> ausgeführt werden und ein vollständiges Setup zum Entwickeln einer Single Page Application ist erstellt. Diese konnte dann sofort mit Webpack auf einem

---

<sup>4</sup>Letzteres ist der Name der Anwendung und kann frei gewählt werden.

Tabelle 5.3: Entwicklungsaufwand der PWA und der Native App

<b>Funktionalität</b>	<b>Rechereche</b>		<b>Implementierung</b>	
	PWA	Native App	PWA	Native App
Einrichtung der Entwicklungs-umgebung	60 min	240 min	25 min	60 min
Installation	30 min	60 min	30 min	15 min
Offlinebetrieb	100 min	40 min	55 min	30 min
Standortzugriff	30 min	45 min	15 min	30 min
Kontaktzugriff	60 min	45 min	45 min	30 min
Benachrichtigungen	300 min	250 min	120 min	200 min
<b>Summe</b>	<b>580 min</b>	<b>680 min</b>	<b>290 min</b>	<b>365 min</b>

Webserver gestartet und weiterentwickelt werden.

Die Ermöglichung der Installation hat bei PWAs circa doppelt so lange benötigt, weil konkrete Implementierungen vorgenommen werden mussten, um dies bei der PWA 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 [42]. 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 14 Stunden (870 min) und circa 17 Stunden (1045 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 andererseits 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 Benachrichtigungen 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.

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

(a) React App Texteingabe Implementierung

(b) React Native Texteingabe Implementierung

Abbildung 5.5: Vergleich Implementierung React PWA und React Native App

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, dass JavaScript-Entwickler, die keine 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 Bewertung der Technologien anhand des Kriterienkatalogs

In der Tabelle 5.4 soll nun auf Grundlage der in Kapitel 3 festgelegten Definitionen die Erfüllung der einzelnen Kriterien in Form von Punkten bewertet werden.

Auf eine Gewichtung der Kriterien wurde bewusst verzichtet, da die vorliegende Arbeit den Vergleich der Technologien behandelt und somit alle Kriterien von gleicher Bedeutung sind.

Tabelle 5.4: Nutzwertanalyse der beiden implementierten Anwendungen

Kriterium		PWA	React Native App (Native App)
Funktionalität	Installation	1	0
	Offlinebetrieb	1	1
	Standortzugriff	1	2
	Kontaktzugriff	1	2
	Benachrichtigungen	2	2
Kompatibilität mit vers. Betriebssystemen		2	3 (1)
Entwicklungsaufwand		2	0
<b>Summe</b>		<b>10</b>	<b>10 (8)</b>

Bei Kriterium der Funktionalität werden die Installation, der Offlinebetrieb, der Standort- und Kontaktzugriff sowie die Benachrichtigungen betrachtet. Hierbei erlangt die PWA im Aspekt der Installation gemäß der Definitionen im Kriterienkatalog einen Punkt, da sie für die Nutzung im Gegensatz zur Native App nicht installiert werden muss. Den Offlinebetrieb kann die PWA ebenso zufriedenstellend implementieren wie die Native App, weswegen sie hier volle Punktzahl erreicht. Bei der Implementierung des Standort- und Kontaktzugriffs wird jedoch deutlich, dass die Native App mehr Funktionen anbietet als die PWA. Der Standortzugriff erlaubt zusätzlich das weiterverarbeiten des Standorts, während der Kontaktzugriff über das Lesen der Daten hinaus geht und diese auch erstellen, bearbeiten und löschen kann. Deshalb werden entsprechend des Kriterienkatalogs jeweils nur ein Punkt vergeben. Zuletzt erhalten die PWA und die Native App für die Benachrichtigungen jeweils zwei Punkte, da beide das Empfangen von persistenten Benachrichtigungen implementieren.

Ein weiteres Kriterium ist die Kompatibilität mit verschiedenen Betriebssystemen. Hierbei erreicht die PWA zwei Punkte, da sie einerseits kompatibel mit mehreren Betriebssystemen, andererseits aber abhängig von dem verwendeten Browser ist. Gemäß dem Kriterienkatalog sind hier deshalb zwei Punkte zu vergeben. Eine React Native App hingegen ist generell in mehreren Betriebssystemen verfügbar und zudem unabhängig vom Browser. Demnach erhält die React Native App drei Punkte.

Diese Verfügbarkeit ist eine Besonderheit der mit React Native entwickelten App, weshalb in Klammern hinter der Punktzahl, die bei einer üblichen Native App resultierende Punkte notiert sind. Übliche Native Apps sind, wie im Kapitel 2 erklärt, nur auf einem Betriebssystem nutzbar und erhalten deshalb bei dem Kriterium Kompatibilität mit verschiedenen Betriebssystemen nur einen Punkt.

Zuletzt erfüllt die PWA das Kriterium des Entwicklungsaufwands wiederum besser, weil sowohl die Recherche- als auch die Implementierungszeit geringer war als bei der React Native App. Deshalb erhält die PWA in diesem Aspekt zwei Punkte, wohingegen die Native App null Punkte erreicht.

Zusammenfassend zeigt das Ergebnis, dass die PWA gleich viele Punkte erhält wie die Native App. Dabei wird allerdings auch deutlich, dass das Ergebnis bei Betrachtung einer üblichen Native App anders ausfällt: In diesem Fall erfüllt diese die Kriterien unzureichender als eine React Native App.

### **Interpretation der Ergebnisse**

Die Betrachtung der Punktzahl der beiden Technologien lässt darauf schließen, dass PWAs aktuell in der Lage sind, native Anwendungen zu ersetzen. Dies muss jedoch kritisch interpretiert werden, denn verlangt eine PWA auf iOS das Empfangen von Benachrichtigungen, so kann diese Funktionalität aufgrund fehlender Schnittstellen hier nicht realisiert werden. PWAs können in diesem Anwendungsfall Native Apps also trotzdem nicht ersetzen.

Die Ergebnisse legen zudem zusammenfassend nahe, dass eine Entscheidung für eine der beiden Implementierungen abhängig von den Anforderungen an die zu entwickelnde App getroffen werden sollte.



# 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. Letzteres wurde mit React Native, einem plattformunabhängigen Framework zur Entwicklung von nativen Anwendungen, entwickelt, um eine faire Grundlage für den Vergleich zu schaffen. Für den Vergleich wurde mit diesen Technologien jeweils eine App entwickelt, welche die COVID-19-Fallzahlen darstellt und filtern kann. Dabei wurden außerdem die Funktionalitäten Installation, Offlinebetrieb, Standortzugriff, Kontaktzugriff und Benachrichtigungen implementiert. Anhand der Programmierung dieser App wurde analysiert, welche Vor- und Nachteile das Entwickeln mit diesen Technologien hat. Auf diesen Beobachtungen aufbauend soll geschlussfolgert werden, ob die PWA die Native App ersetzen kann.

Dabei hat sich herausgestellt, dass PWAs mittlerweile viele Funktionalitäten von nativen Anwendungen umsetzen können und deshalb gerade für simple Anwendungen eine Alternative bieten. Gerade bei der Installation weist die PWA im Vergleich zur Native App Stärken auf. Außerdem ist der Entwicklungsaufwand der PWA geringer als der einer nativen Anwendung.

Dennoch ist ein entscheidender Faktor gegen die Etablierung von PWAs die fehlende Unterstützung durch iOS, da deshalb etwa 26 Prozent des weltweiten Marktanteils von mobilen Betriebssystemen weniger Funktionalitäten zur Verfügung stehen [43]. Dies betrifft wie in der vorliegenden Arbeit dargestellt die Push API und Notification API, wodurch die das Empfangen von Benachrichtigungen mit iOS Geräten nicht möglich ist.

Doch in Anbetracht der Vielzahl von Webschnittstellen, die sich seit dem Aufschwung von PWAs 2015 entwickelt haben, wird deutlich, dass die Zukunft von mobilen Anwendungen vielfältig ist.

Zusammenfassend legen die Ergebnisse nahe, dass eine Entscheidung für eine PWA oder eine Native App abhängig vom Anwendungsfall der App getroffen werden sollte.

## 6.2 Ausblick

Für anknüpfende Arbeiten könnten die Analyse von PWAs und Native Apps intensiviert werden, indem umfassender auf die verschiedenen Funktionalitäten eingegangen wird, die mit diesen beiden Technologien umsetzbar sind. Beispiele hierfür sind die Einbindung der Background Sync API, Payment Request API oder Web Share Target API.

Zusätzlich dazu kann aufgrund der stetigen Verbesserung der letzten Jahre angenommen werden, dass eine Erweiterung des Angebots von Webschnittstellen in Zukunft stattfindet. Deshalb ist eine erneute Betrachtung der Thematik für die Beurteilung, ob PWAs Native Apps ersetzen können, unabdingbar.

Interessant wäre außerdem ein Performance Vergleich der zwei Technologien. 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 [44]. Der Sicherheitsaspekt ist vor allem bedeutsam für mobilen Anwendungen, die sensible Daten verarbeiten.

## Literaturverzeichnis

- [1] “Number of mobile app downloads worldwide from 2016 to 2020: (in billions),” 2021. [Online]. Available: <https://www.statista.com/statistics/271644/worldwide-free-and-paid-mobile-app-store-downloads/>
- [2] A. Russell, “Progressive web apps: Escaping tabs without losing our soul,” 2015. [Online]. Available: <https://medium.com/@slightlylate/progressive-apps-escaping-tabs-without-losing-our-soul-3b93a8561955>
- [3] J. Johnson, “Global digital population as of january 2021: (in billions),” 2021. [Online]. Available: <https://www.statista.com/statistics/617136/digital-population-worldwide/>
- [4] W. Jobe, “Native apps vs. mobile web apps,” *International Journal of Interactive Mobile Technologies (iJIM)*, vol. 7, no. 4, p. 27, 2013.
- [5] “Mobile operating system market share worldwide: Apr 2010 - july 2021,” o. J. [Online]. Available: <https://gs.statcounter.com/os-market-share/mobile/worldwide/#monthly-201004-202107>
- [6] M. Schickler, M. Reichert, and R. Pryss, *Entwicklung mobiler Apps: Konzepte, Anwendungsbausteine und Werkzeuge im Business und E-Health*, ser. eXamen.press. Berlin: Springer Vieweg, 2015.
- [7] S. Greif and R. Benitte, “Mobile and desktop,” 2020. [Online]. Available: <https://2020.stateofjs.com/en-US/technologies/mobile-desktop/>
- [8] MDN contributors, “Progressive web apps (pwas),” o. J. [Online]. Available: [https://developer.mozilla.org/en-US/docs/Web/Progressive\\_web\\_apps](https://developer.mozilla.org/en-US/docs/Web/Progressive_web_apps)
- [9] M. Gaunt, “Service workers: an introduction,” o. J. [Online]. Available: <https://developers.google.com/web/fundamentals/primers/service-workers>
- [10] MDN contributors, “Service worker api,” o. J. [Online]. Available: [https://developer.mozilla.org/en-US/docs/Web/API/Service\\_Worker\\_API](https://developer.mozilla.org/en-US/docs/Web/API/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),” Microsoft Docs, 2020. [Online]. Available: <https://docs.microsoft.com/en-us/dotnet/architecture/modern-web-apps-azure/choose-between-traditional-web-and-single-page-apps>

- [12] S. Richard and P. LePage, “What makes a good progressive web app?” 2020. [Online]. Available: <https://web.dev/pwa-checklist/>
- [13] A. Russell, J. Song, J. Archibald, and M. Kruisselbring, “Service workers nightly,” 2021. [Online]. Available: <https://w3c.github.io/ServiceWorker/>
- [14] MDN contributors, “Web app manifest,” o. J. [Online]. Available: <https://developer.mozilla.org/de/docs/Web/Manifest>
- [15] C. Liebel, *Progressive Web Apps: Das Praxisbuch*, 1st ed., ser. Rheinwerk Computing. Bonn: Rheinwerk Verlag, 2019.
- [16] M. Lamouri, M. Cáceres, and J. Yasskin, “Permissions,” 2020. [Online]. Available: <https://www.w3.org/TR/permissions/>
- [17] R. Barger, “Is react a library or a framework? here’s why it matters,” 2021. [Online]. Available: <https://www.freecodecamp.org/news/is-react-a-library-or-a-framework/>
- [18] “React without jsx,” o. J. [Online]. Available: <https://reactjs.org/docs/react-without-jsx.html>
- [19] “Who’s using react native,” o. J. [Online]. Available: <https://reactnative.dev/showcase>
- [20] “Introducing hooks: v17.0.2,” 2021. [Online]. Available: <https://reactjs.org/docs/hooks-intro.html>
- [21] Facebook, “React native.” [Online]. Available: <https://github.com/facebook/react-native>
- [22] “Core components and native components: 0.64,” o. J. [Online]. Available: <https://reactnative.dev/docs/intro-react-native-components>
- [23] E. Behrends, *React Native: Native Apps parallel für Android und iOS entwickeln*, 1st ed. Heidelberg: O’Reilly, 2018.
- [24] “Rki corona landkreise,” 2020. [Online]. Available: [https://npgeo-corona-npgeo-de.hub.arcgis.com/datasets/917fc37a709542548cc3be077a786c17\\_0/about](https://npgeo-corona-npgeo-de.hub.arcgis.com/datasets/917fc37a709542548cc3be077a786c17_0/about)
- [25] M. Cáceres, A. Gustafson, M. Giuca, A. Kostiainen, M. Lamouri, and K. Rohde Christiansen, “Web application manifest,” 2021. [Online]. Available: <https://www.w3.org/TR/appmanifest/>
- [26] A. Deveria, “Add to home screen (a2hs),” caniuse.com, o.J. [Online]. Available: <https://caniuse.com/web-app-manifest>
- [27] “App store review guidelines,” 07.06.2021. [Online]. Available: <https://developer.apple.com/app-store/review/guidelines/>

- [28] 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. [Online]. Available: <https://web.dev/storage-for-the-web/>
- [29] MDN contributors, “Client-side storage,” o. J. [Online]. Available: [https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Client-side\\_web\\_APIs/Client-side\\_storage](https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Client-side_web_APIs/Client-side_storage)
- [30] J. Archibald, “The offline cookbook,” 2014. [Online]. Available: <https://web.dev/offline-cookbook/>
- [31] R. Turner, “Releasing react native 0.59,” 2019. [Online]. Available: <https://reactnative.dev/blog/2019/03/12/releasing-react-native-059>
- [32] M. Cáceres, “Geolocation api,” 2021. [Online]. Available: <https://w3c.github.io/geolocation-api/>
- [33] Agontuk, “react-native-geolocation-service,” 2018. [Online]. Available: <https://github.com/Agontuk/react-native-geolocation-service>
- [34] “Firefox/ push notifications,” 12.01.2016. [Online]. Available: [https://wiki.mozilla.org/Firefox/Push\\_Notifications](https://wiki.mozilla.org/Firefox/Push_Notifications)
- [35] P. LePage, T. Steiner, and F. Beaufort, “Add a web app manifest,” 2018. [Online]. Available: <https://web.dev/add-manifest/>
- [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] “Twitter lite pwa significantly increases engagement and reduces data usage,” 2017. [Online]. Available: <https://developers.google.com/web/showcase/2017/twitter>
- [38] M. Kruisselbrink, “Geofencing api,” 2017. [Online]. Available: <https://www.w3.org/TR/geofencing/>
- [39] 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. [Online]. Available: <https://firt.dev/ios-14.5/>
- [40] A. Deveria, “Push api,” caniuse.com, o.J. [Online]. Available: <https://caniuse.com/push-api>
- [41] MDN contributors, “Contact picker api,” o. J. [Online]. Available: [https://developer.mozilla.org/en-US/docs/Web/API/Contact\\_Picker\\_API](https://developer.mozilla.org/en-US/docs/Web/API/Contact_Picker_API)

- [42] P. Beverloo and Rayan Kanso, “Contact picker api,” 2021. [Online]. Available: <https://wicg.github.io/contact-api/spec/>
- [43] “Mobile operating system market share worldwide,” 2021. [Online]. Available: <https://gs.statcounter.com/os-market-share/mobile/worldwide>
- [44] 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: Association for Computing Machinery, 2018, pp. 1731–1746.