THM | CAMPUS GIESSEN | MNI
Mathematik, Naturwissenschaften und Informatik
TECHNISCHE HOCHSCHULE MITTELHESSEN

Manuskript

Progressive Web Apps

Kurs „Hauptseminar – Mobile Technologies“

Niklas Deworetzki

20. November 2019

Inhaltsverzeichnis

1 Einleitung

Diese Arbeit befasst sich mit dem Begriff der Progressive Web App (PWA), einer neuen Technologie, welche die Lücke zwischen herkömmlichen Webanwendungen und nativen Anwendungen füllen soll. Worum es sich dabei handelt, wie die Technologie der PWAs umgesetzt ist und welche Konsequenzen sich daraus ergeben, wird im Folgenden besprochen.

Zunächst wird in Sektion 2 erläutert, wie sich eine PWA von einer herkömmlichen Webanwendung oder einer nativen Anwendung abgrenzt. Als *herkömmliche Webanwendung* wird dabei eine Webseite oder Anwendung bezeichnet, die als Teil einer Website eine Funktionalität bereitstellt und auf HTML sowie CSS und JavaScript basiert. Wichtig ist dabei, dass herkömmliche Webanwendungen nur online durch den Browser erreichbar sind. Im Vergleich dazu sind *native Anwendungen* eine Kategorie von Anwendungen, die direkt vom Betriebssystem eines Gerätes ausgeführt werden und deren Ressourcen lokal vorliegen. Hierbei handelt es sich häufig um Anwendungen, die in systemnahen Programmiersprachen wie C, C++ oder Swift geschrieben sind. Auf Android-Systemen kommen auch Java oder seit neuerem Kotlin zum Einsatz. Im Folgenden werden native Anwendungen auch als Apps bezeichnet, was zum einen dem Trend folgt, dass als Apps nicht mehr nur Applikationen auf Mobilgeräten, sondern generell ausführbare Programme bezeichnet werden und andererseits zur deutlicheren namentlichen Unterscheidung zwischen Webanwendung und nativer Anwendung dient.

Nachdem die Eigenschaften und Abgrenzungen einer PWA besprochen wurden, wird in Sektion 3 erläutert, welche Voraussetzungen im Zusammenhang mit einer PWA auftreten. Hier liegt der Schwerpunkt besonders auf den benötigten Technologien, die für die Funktionalität einer PWA notwendig sind, wodurch sich eine PWA von anderen Anwendungen abgrenzt. Des Weiteren wird auf die Voraussetzungen eingegangen, die von den verschiedenen Browsern an PWAs gestellt werden.

Letztlich werden in den Sektionen 4 und Sektion 5 angeführt, welche positiven und negativen Eigenschaften PWAs mit sich bringen, sodass schließlich in Sektion 6 eine finale Wertung zur Nutzung und Verbreitung von PWAs gegeben werden kann.

2 Was ist eine Progressive Web App?

Der Begriff *Progressive Web App (PWA)* beschreibt eine Kategorie von Webanwendungen, welche eine Reihe an Eigenschaften erfüllen, um das Erlebnis einer nativen Anwendung in modernen Browsern zu bieten. Geprägt wurde dieser Begriff von Alex Russell, welcher in seinem Blog[1] beschreibt, wie durch die Weiterentwicklung und Standardisierung von Browsern und Webtechnologien eine neue Art von Anwendung entstanden ist. Diese Kategorie von Anwendungen entspreche dem nächsten Schritt in einer Reihe von Technologien, die versuchen, Entwicklern die Möglichkeit zu geben, mit nur einer plattformübergreifenden Anwendung auf plattformspezifische Ressourcen zuzugreifen.

So sind PWAs also eine Art Nachfolger von Technologien wie „Universal Windows Platform"-Apps[2] von Microsoft oder die von W3C standardisierten „Packaged Web Apps"[3], welche auch als „Widget" bekannt sind. Der Begriff „Progressive" in Progressive Web App steht für „Progressive Enhancement" (fortschreitende Verbesserung) und beschreibt ebendiese Vorgehensweise, eine Webanwendung auf möglichst vielen Geräten mit verschiedenen Kapazitäten nutzbar zu machen. Die Idee dabei ist es, eine Grundversion der Anwendung zu erstellen, die auf jedem Gerät funktionieren kann. Weitere Funktionalität für potentere Geräte wird dann auf diese Grundversion aufgebaut.

Da der Begriff der PWA zusammen mit den notwendigen Technologien über einen längeren Zeitraum gewachsen ist, gibt es keine feste Definition für das, was eine PWA ausmacht. Die Beschreibung einer PWA basiert vielmehr auf einer Menge an Eigenschaften, die eine Anwendung innehaben kann. Im Folgenden werden daher einige Eigenschaften zusammengetragen und erläutert, die immer wieder für die Beschreibung von PWAs verwendet werden. Ein besonderer Schwerpunkt wird dabei auf die Standpunkte der Google- und Mozilla-Entwickler gelegt, da diese treibende Kraft beim Entwickeln von PWAs sind und gemeinsam die Fähigkeiten dieser Gruppe von Anwendungen ausbauen.

Die gesammelten Eigenschaften einer PWA lassen sich dabei in Kategorien einteilen, welche im Folgenden erläutert werden.

1. **Grundlegende Eigenschaften.** Eigenschaften, die dem Charakter einer PWA entsprechen und ohne die ein Funktionieren der Anwendung nicht möglich wäre.
2. **Verbreitete Eigenschaften.** Eigenschaften einer PWA, welche von verschiedenen Quellen genannt werden, aber nicht unbedingt notwendig für das Funktionieren der Anwendung sind.
3. **Designmerkmale.** Eigenschaften einer PWA, die sich positiv auf das Nutzererlebnis auswirken und optischer Natur sind.

2.1 Grundlegende Eigenschaften

Die wohl wichtigste Eigenschaft einer PWA ist, dass sie lokal installiert werden kann. Wenn ein Nutzer eine Webanwendung nutzt, welche als PWA agiert, so kann dem Nutzer ein Installationsdialog gezeigt werden, um den Installationsvorgang zu starten. Durch den Installationsvorgang werden teile der Anwendung auf dem lokalen Gerät gespeichert und die PWA optisch zu den nativen Anwendung integriert. Ob diese Integration nun über das Hinzufügen eines Desktopicons, eine Verknüpfung auf dem Startbildschirm oder über einen Eintrag in einem Anwendungslauncher erfolgt, ist dabei vom unterliegenden System abhängig. Auf einem Android-Gerät wird es beispielsweise sinnvoll sein, die PWA als Icon auf dem Startbildschirm anzuzeigen. Auf einem Windows-Gerät stattdessen wird dem Nutzer eine Verknüpfung auf dem Desktop lieber sein. Da die PWA nicht eigenständig entscheiden kann, welches Gerät der Nutzer verwendet und wohl unmöglich die Fähigkeit besitzt, für jedes Gerät die entsprechende Konfiguration durchzuführen, muss der Installationsvorgang durch den Browser unterstützt und verwaltet werden.

Beim anschließenden Ausführen einer installierten PWA ist erneut Unterstützung vom Browser notwendig. PWAs als Webanwendung werden mit Webtechnologien wie HTML und CSS aufgebaut und erhalten ihre Funktionalität durch JavaScript. Betriebssysteme bieten jedoch nur eine begrenzte Unterstützung für Programmformate, die nativ ausgeführt werden können. Abgesehen von Skriptdateien wie etwa Shell-Scripts für Linux oder Command-Files für Windows, die in einem betriebssystemabhängigen Format geschrieben sein müssen, wird meist nur die Ausführung von Binärformaten unterstützt[4]. Es wird also Hilfe vom Browser benötigt, welcher die installierte PWA verwaltet und daher alle benötigten Dateien laden und ausführen kann. Zudem ist es dem Browser möglich, für die Ausführung einer PWA bestimmte Regeln festzulegen. In Google Chrome beispielsweise wird jede PWA in einem eigenen Fenster gestartet, wodurch das Nutzererlebnis einer nativen Anwendung entstehen soll[5].

Durch den Installationsprozess werden nur Teile der PWA lokal auf dem Gerät gespeichert. Der Grund dafür ist, dass es meist nicht genügt, eine Website einfach vollständig zu kopieren, um sie offline verfügbar zu machen. Viele Webanwendungen, wie beispielsweise die PWA von Twitter[1], basieren darauf, stetig neue Inhalte von einem Server abzufragen. Die Verwaltung, welche Inhalte offline verfügbar sein sollen und welche nicht, geschieht über sogenannte „Service Worker“. Dadurch, dass jede PWA lokale und entfernte Inhalte verwalten muss, ist das Vorhandensein von Service Workern eine weitere Eigenschaft, die jede PWA besitzt.

Eine Voraussetzung der Service Worker erzwingt eine weitere Eigenschaft aller PWAs. Service Worker können laut Spezifikation nur auf Seiten ausgeführt werden, die über HTTPS ausgeliefert werden. Der Grund dafür ist, dass Service Worker die Fähigkeit besitzen, Anfragen zu filtern, umzuleiten und auszulesen. Die verschlüsselte Verbindung bei HTTPS verhindert dabei die Manipulation durch Dritte beim Ausliefern des Service

[1] https://play.google.com/store/apps/details?id=com.twitter.android.lite

Worker. So soll sichergestellt werden, dass Service Worker nicht für das Ausspionieren von Daten genutzt werden[6].

Letztlich ist das Vorhandensein einer Manifestdatei als Eigenschaft aller PWAs zu nennen. Die Manifestdatei listet die Eigenschaften einer PWA auf und ist somit auch bei jeder aufzufinden.

2.2 Häufige Eigenschaften

Die im Folgenden genannten Eigenschaften werden häufig bei PWAs angetroffen, müssen jedoch nicht notwendigerweise umgesetzt werden. Größtenteils entstehen diese Eigenschaften durch die für PWAs verwendeten Bibliotheken.

PWAs besitzen die Fähigkeit, viele Aufgaben von nativen Apps zu übernehmen. Grund dafür ist, dass der Browser als Laufzeitumgebung den Zugriff auf viele Systemressourcen ermöglicht. Sie können auf die Kamera eines Gerätes zugreifen, Audio- und Videoaufnahmen machen oder Positionsdaten einsehen. Das Vorhandensein dieses Zugriffs ermutigt natürlich, dass die entsprechenden Ressourcen auch von einer PWA genutzt werden.

Da Teile einer PWA lokal gespeichert sind, können schnellere Ladezeiten erreicht werden. Gerade bei einer schlechten Netzwerkverbindung wird dies deutlich. Zudem wird von den Google Entwicklern zusammen mit dem Entwicklungs-Framework für PWAs das Tool „Pagespeed Insights“[2] angeboten, welches eine Analyse der Ladezeiten vereinfacht und Vorschläge zur Optimierung gibt.

Eine weitere Eigenschaft, die bei vielen PWAs auftritt, ist, dass es einfacher ist, Inhalte zu teilen. Da eine PWA auf einer herkömmlichen Webanwendung basiert, ist in der Regel jeder angezeigte Inhalt über eine URL erreichbar. Dies macht ein Teilen der Inhalte sehr einfach, da lediglich die URL kopiert werden muss. Ist nämlich der angezeigte Inhalt allein von der URL abhängig, so genügt die URL, um Inhalte zu verbreiten und zu teilen. Dadurch werden Inhalte komplett unabhängig von Gerät oder Nutzer, was dem progressiven Design entspricht.

2.3 Designmerkmale

Wie sehr PWAs an native Anwendungen angelehnt sind, lässt sich in den Designmerkmalen der Anwendungen wiedererkennen.

Das bereits erwähnte Starten der Anwendung in einem eigenen Fenster beispielsweise soll dem Nutzer das Gefühl verleihen, eine native installierte Anwendung zu starten. Zusätzlich gibt es die Möglichkeit, ein Farbschema für die gesamte Anwendung festzulegen. Der Effekt eines solchen Farbschemas ist, dass die Steuerungskomponenten des Browsers farblich an die PWA angepasst werden können. Dadurch wirkt die Anwendung für

[2] https://developers.google.com/speed/pagespeed/insights/

den Nutzer einheitlich und es wird versteckt, dass die PWA eigentlich nur im Browser angezeigt wird und keine eigenständige Anwendung ist. Das gleiche Prinzip existiert für native Anwendungen, die mit einem Farbschema die Komponenten des Systems einfärben können, um die gesamte angezeigte Oberfläche einheitlich erscheinen zu lassen.

Ebenso erwähnt wurde bereits, dass der Browser als Laufzeitumgebung der PWA einen Zugriff auf Systemressourcen zulässt. Dies beinhaltet auch das Senden und Anzeigen von Benachrichtigungen von einer PWA. Dabei ist nicht nur das Darstellen von Benachrichtigungen auf einer gerade angezeigten Seite möglich, sondern auch Push-Notifications wie von einer nativen App können erstellt werden. Zusätzlich mit den Einträgen einer installierten PWA auf dem Startbildschirm des Nutzers, integrieren sich PWAs so nahezu nahtlos in die Darstellung nativer Apps.

3 Technische Voraussetzungen

Im folgenden Abschnitt werden die technischen Voraussetzungen besprochen, die im Zusammenhang mit PWAs auftreten. Zunächst wird behandelt, welche Voraussetzungen nötig sind, um eine PWA zu installieren und Daten lokal verfügbar zu machen. Anschließend werden die Voraussetzungen aufgelistet, die nötig sind, damit eine PWA mit dem Nutzer und dem Gerät interagieren kann, wie es von den Eigenschaften der PWA erwartet wird. Letztlich werden die Voraussetzungen verglichen, welche von Browsern an eine PWA gestellt werden.

3.1 Installationsformat

Das Installieren einer PWA entspricht größtenteils dem Setzen eines Lesezeichens bei einer herkömmlichen Website. Dem Browser wird mitgeteilt, dass eine Referenz auf diese Website in eine lokale Sammlung aufgenommen werden soll. Moderne Browser, wie Google Chrome oder Firefox, können Verknüpfungen für beliebige Websites auf dem Desktop oder im Startmenü erstellen[7][8]. Dieses Verhalten entspricht genau dem, was bei einer PWA gewünscht ist, welche auch zwischen den nativen Anwendungen auf dem Desktop oder Startbildschirm des Gerätes angezeigt werden sollen.

Bei der Installation einer PWA wird lediglich auf diesen Fähigkeiten des Browsers aufgebaut. Wie bei einer Verknüpfung für eine herkömmliche Website ist es auch bei einer PWA gewünscht, ein Icon zusammen mit einem Namen anzuzeigen. Es fehlen jedoch einige Eigenschaften bei einer solchen Verknüpfung, die eine native App besitzen würde. Für die Konfiguration dieser Informationen existiert ein normiertes Format, welches mit einer PWA ausgeliefert wird. In dem sogenannten *Manifest* einer Website werden die Eigenschaften als JavaScript Object Notation (JSON) festgelegt.

Die Einbindung des Manifests geschieht im Header einer PWA. Dort kann über ein gewöhnliches `link`-Tag eine Manifestdatei eingebunden werden.

```
<link rel="manifest" href="/manifest.json">
```

Abbildung 1: Einbindung einer Manifestdatei

Die Spezifikation der Eigenschaften in der Manifestdatei geschieht über SchlüsselWerte-Paare. Das Definieren der Schlüssel `short_name`, `name`, `icons`, `start_url` und `display` ist dabei verpflichtend.

`short_name` und `icons` beschreiben das Aussehen der Verknüpfung. Der Schlüssel `icons` wird im Plural verwendet, da in der Manifestdatei mehrere Icons in verschiedenen Auflösungen angegeben werden sollten. So kann je nach Auflösung des Gerätes das am besten passende Icon vom Browser gewählt werden. Der lange Name `name` wird für Installationsdialoge und als Beschriftung der Verknüpfung verwendet. Zusätzlich sollte der verkürzte Name unter `short_name` angegeben werden, welcher alternativ angezeigt wird, wenn weniger Platz vorhanden ist[9]. Dies kann bei mobilen Geräten der Fall sein.

Mit dem Schlüssel `display` wird gesteuert, wie der Browser die PWA anzeigt. Hier kann als Wert `fullscreen`, `standalone` oder `browser` angegeben werden. Bei jedem dieser Werte wird die PWA in einem eigenständigen Fenster ausgeführt. Die Werte `fullscreen` oder `standalone` blenden dabei die Anzeigeelemente des Browsers aus. Es wird also keine Navigationsleiste oder URL angezeigt. Ist der Wert `fullscreen` gewählt, so wird die PWA im Vollbildmodus gestartet, sodass zusätzlich die Anzeigeelemente des Betriebssystems ausgeblendet sind. Durch den Wert `browser` werden die Anzeigeelemente des Browsers wie bei einer herkömmlichen Website angezeigt[10].

Der Wert `start_url` ist wichtig für das Verhalten einer PWA. Da die Installation einer PWA wie das Setzen eines Lesezeichens an jeder Unterseite einer Anwendung geschehen kann, genügt es nicht, bloß die aktuelle URL zu speichern. Wenn ein Nutzer beispielsweise gerade sein Profil betrachtet und sich an dieser Stelle entschließt, die PWA zu installieren, so würde er bei erneutem Starten der PWA immer wieder zu seinem Profil zurückkehren, wenn die Eigenschaften eines herkömmlichen Lesezeichens übernommen werden. Es ist aber bei einer Anwendung meistens erwünscht, in ein zentrales Menü herein zu starten, welches zu Beginn angezeigt wird. Die URL für dieses Startmenü wird daher an dieser Stelle angegeben.

Alle weiteren Schlüssel sind optional. Sie dienen der Verbesserung des Nutzererlebnis, können aber auch weggelassen werden, wenn ein Einsatz nicht sinnvoll ist.

Der Schlüssel `description` gibt eine Beschreibung der PWA. Diese kann zusammen mit der Verknüpfung angezeigt werden. Ein anderer optionaler Schlüssel ist `scope`, mit welchem die Reichweite der PWA angegeben werden kann. So kann eine PWA in eine existierende Website integriert werden, indem nur eine bestimmte Gruppe an Routen zugewiesen wird[11].

Weitere optionale Schlüssel dienen der visuellen Integration in den Browser. Über den Schlüssel `background_color` lässt sich beispielsweise eine Farbe festlegen, die vom Brow-

ser beim Laden der Anwendung angezeigt wird, wenn noch keine Inhalte sichtbar sind. So lässt sich ein Farbschema wie bei einer nativen Anwendung imitieren. Der Schlüssel `theme_color` passt zusätzlich die Farbe der Browserkomponenten an, sodass ein einheitliches Farbschema für die gesamte Oberfläche entsteht.

3.2 Datenverwaltung durch Service Worker

Wie bereits erwähnt, werden bei einer PWA Ressourcen sowohl von lokalen Quellen als auch über das Netzwerk geladen. Da eine PWA vor der Installation wie eine herkömmliche Website agiert, ist es nicht möglich, einfach im Quelltext der Anwendung die lokal installierten Ressourcen anzugeben. Diese sind vor der Installation noch nicht lokal verfügbar, wodurch das Abrufen zu einem Fehler führen würde. Auch das Ausliefern von zwei Versionen an den Nutzer, die einmal für den *Online Gebrauch* und einmal für den *Offline Gebrauch* gedacht sind, ist nicht möglich, da die Installation durch den Browser durchgeführt wird, welcher die aktuell angezeigte Version installiert. Demnach würde die Online-Version der Anwendung dauerhaft verwendet werden, was dem Sinn der Installation widerspricht.

Folglich ist es also notwendig, einheitlich die Ressourcen aus dem Netzwerk abzurufen, da sie dort immer erhältlich sind. Werden diese Netzwerkanfragen dann noch lokal abgefangen, kann überprüft werden, ob Ressourcen bereits lokal erhältlich sind. Sollte dies der Fall sein, so ist es möglich, die Anfragen umzuleiten, sodass die lokalen Ressourcen verwendet werden. Für die lokale Verwaltung von Inhalten sind die sogenannten „Service Worker“ zuständig, dessen Funktionsweise im Folgenden näher erläutert wird.

Worker im Allgemeinen sind JavaScript-Programme, die in einem eigenen Kontext ausgeführt werden. Dieser Kontext ermöglicht die Verwendung mehrerer Threads in JavaScript, da jeder Kontext unabhängig und parallel zu anderen ausgeführt werden kann. Ein Service Worker ist ein Worker, der im Hintergrund aktiv ist und dort auf Ereignisse (Events) reagieren kann. Dabei haben Service Worker jedoch keinen direkten Zugriff auf das Document Object Model (DOM) der PWA.

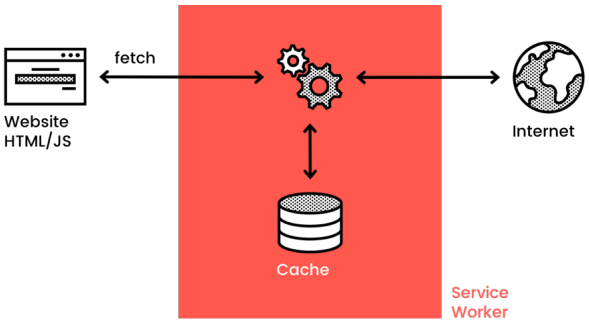

Abbildung 2: Service Worker als Proxy[12]

In dem globalen Kontext der Service Worker befindet sich ein Cache-Objekt, welches den lokalen Speicher der PWA repräsentiert[13]. Dieser Cache muss zunächst populiert werden, was entweder bei der Installation der PWA oder zu späterem Zeitpunkt mit der ersten Anfrage einer Ressource geschehen kann[6]. Anschließend kann bei Netzwerkanfragen auf den Cache verwiesen werden.

Das Reagieren auf Anfragen geschieht über ein „fetch“-Event, auf welches Service Worker reagieren können. Für dieses Event kann eine Funktion registriert werden, welche ein Event-Objekt als Parameter akzeptiert. Das Event-Objekt enthält alle Informationen über die angeforderte Ressource und kann manipuliert werden, um auszuwählen, von wo die Ressource geladen werden soll. Wenn der Cache bereits populiert ist, ist die Auswahl über den Cache trivial.

```
self.addEventListener('fetch', function(event) {
  event.respondWith(caches.match(event.request)
    .then(cachedResponse => cachedResponse || fetch(event.request)))
});
```

Abbildung 3: Auswahl von Ressourcen aus dem Cache.[14]

Wie in Abbildung 3 dargestellt, wird eine anonyme Funktion für das „fetch“-Event registriert. Diese Funktion legt fest, dass auf die Anfrage des „fetch“-Event eine Antwort aus dem Cache gesucht werden soll. Wenn die angeforderte Ressource im Cache vorhanden ist, wird sie als Ergebnis der Anfrage festgelegt. Andernfalls wird eine Anfrage über das Netzwerk gesendet, um die Ressource abzurufen.

Durch die Funktionsweise von Service Workern stellen diese gleichzeitig ein großes Sicherheitsrisiko dar. Wenn ein Dritter einen Service Worker zwischen Server und Nutzer manipulieren würde, hätte dieser Zugriff auf alle Netzwerkanfragen, die der Nutzer der PWA sendet. Um Angriffe eines solchen Mittelmannes zu verhindern, können Service Worker laut Spezifikation nur über HTTPS ausgeliefert werden. Dabei wird der Übertragungskanal verschlüsselt, wodurch eine Manipulation des Service Worker durch Dritte ausgeschlossen wird.

Durch Service-Worker ist es also möglich, einen Cache in jede Website einzubinden. Da diese im Hintergrund arbeiten und auf bereits vorhandene Technologien aufbauen, ist es nicht einmal nötig, eine bereits vorhandene Website anzupassen, um den Cache zu implementieren. Durch das Speichern von Ressourcen im Cache wird das Installieren einer PWA umgesetzt.

3.3 Interaktion mit Nutzern

Eine weiteres Merkmal von PWAs, welches herkömmliche Websites nicht besitzen, ist die Art und Weise, wie mit Nutzern interagiert wird. Native Anwendungen können auf Ressourcen des Betriebssystems zugreifen, um mit dem Nutzer zu interagieren, selbst wenn die Anwendung nicht aktiv vom Nutzer verwendet wird. Weit verbreitet ist beispielsweise die Verwendung von Push-Notifications[15], um Nachrichten oder Informationen an den Nutzer zu leiten.

PWAs sind ebenso in der Lage, Push-Notifications zu senden, was sie von herkömmlichen Websites abgrenzt. Diese können auch Benachrichtigungen senden, jedoch nur, wenn der Nutzer aktiv eine Seite betrachtet. Durch Service Worker, welche unabhängig von einem Seitenaufruf ausgeführt werden, können PWAs auch Benachrichtigungen senden, ohne dass die PWA geöffnet ist[16].

Zunächst muss ein Nutzer zustimmen, dass er Benachrichtigungen von einer Website oder PWA erhalten will. Falls die Zustimmung erteilt wird, wird dies für die entsprechende Website oder PWA gespeichert, sodass ein erneutes Einholen nicht notwendig ist, so lange der Nutzer nicht im Nachhinein die Berechtigung widerruft.

Auf einer geöffneten Website genügt die Zustimmung des Nutzers, um eine Benachrichtigung anzuzeigen. Der beim Seitenaufruf ausgeführte JavaScript-Code kann aktiv eine Benachrichtigung erstellen und anzeigen. Eine PWA, die nicht ausgeführt wird, kann auf diesen Weg keine Benachrichtigung anzeigen. Auch ein Service Worker kann nicht aktiv eine Benachrichtigung erstellen, da dieser nur auf externe Events reagieren kann.

Folglich werden Push-Notifications nicht vom eigenen Gerät erstellt, sondern nur auf diesem angezeigt. Für das Empfangen der Benachrichtigungen wird die Push-API[17] verwendet. Über diese API kann sich ein Service-Worker bei dem Push-Client des Browsers registrieren, welcher mit einem Push-Service zusammenarbeitet.

Für das Senden einer Push-Notification wird nun von einem Backend aus eine Anfrage über das *Web Push Protocol*[18] an den Push-Service gesendet. Dieser verteilt dann die Push-Notification an alle registrierten Clients, die die Benachrichtigung erhalten sollen. Falls ein Client offline ist, wird die Benachrichtigung in eine Warteschlange eingefügt und zugestellt, wenn der Client eine erneute Verbindung aufbaut[19].

Der Client leitet dann die erhaltene Push-Notification als Event an den entsprechenden Service Worker weiter, welcher so auf die Benachrichtigung reagieren kann. So ist es für den Service Worker nun möglich, eine Benachrichtigung auf dem Gerät des Nutzers anzuzeigen, obwohl die PWA des Service Workers nicht aktiv ist, da der Service Worker unabhängig zur PWA ausgeführt wird.

3.4 Interaktion mit Systemressourcen

Um Aufgaben nativer Anwendungen übernehmen zu können, benötigen PWAs auch Zugriff auf die sonst geschützten Systemressourcen. Für native Anwendungen ist der Zugriff auf Systemressourcen wie das Dateisystem oder angeschlossene Geräte meist keine große Hürde. Falls überhaupt, muss vor Zugriff die entsprechende Berechtigung eingeholt werden, wofür native Schnittstellen vorliegen. Damit auch Webanwendungen und somit PWAs auf diese Ressourcen zugreifen können, wird die Unterstützung des Browsers benötigt, welcher als Schnittstelle zwischen Webanwendung und den nativen Ressourcen agiert.

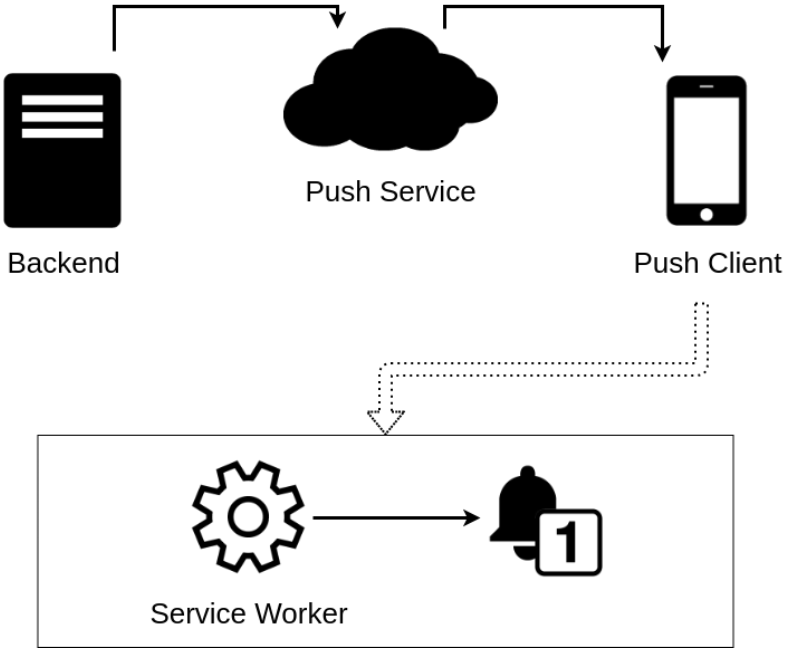

Abbildung 4: Ablauf beim Senden einer Push-Notification

Es ergibt sich dadurch jedoch die Herausforderung, eine einheitliche Schnittstelle zu schaffen, sodass eine Webanwendung unabhängig vom Browser auf Systemressourcen zugreifen kann. Daraus folgt, dass für jede Zugriffskomponente ein eigener Standard entwickelt wird, welcher anschließend in den einzelnen Browsern umgesetzt werden muss.

Zum aktuellen Zeitpunkt existieren für nahezu alle Zugriffskomponenten, die ein Smartphone oder PC bietet, ein Standard oder zumindest ein Entwurf für diesen. In einer Übersicht von Adam Bar[20] werden insgesamt 44 dieser Komponenten gelistet. Als Folge dieser großen Vielfalt ist eine ebenso große Vielfalt an Zugriffsformen entstanden, die in den Standards festgelegt sind. Wie im Folgenden beispielhaft gezeigt, besitzen verschiedene Komponenten deutlich unterschiedliche Zugriffsformen:

Positionsdaten Positionsdaten werden über ein *Geolocation*-Objekt im Navigator des Browsers angefordert. Eine Überprüfung, ob dieses Objekt vorhanden ist, genügt, um Kompatibilität des Browsers festzustellen. Beim Anfragen der Positionsdaten wird automatisch um Erlaubnis des Nutzers gebeten[21].

Benachrichtigungen Benachrichtigungen können über ein *Notification*-Objekt erstellt werden. Dieses befindet sich im globalen Kontext eines Browsertabs. Bevor eine Benachrichtigung angezeigt werden kann, muss aktiv um Erlaubnis gebeten werden[22].

Kamera und Mikrophon Geräte wie Kameras oder Mikrophone liefern einen dauerhaften Strom an Daten. Diese Datenströme können über ein *Media Devices*-Objekt im Navigator des Browsers angefordert werden[23].

Neben der großen Vielfalt an standardisierten Zugriffsformen unterscheidet sich auch, wie verbreitet die Implementierung der Standards ist. Während der Zugriff auf Mikrophon und Kamera beispielsweise noch bei 95% der Nutzer unterstützt ist, so ist das Erstellen und Anzeigen von Notifications nur noch bei 77% der Nutzer möglich[24]. Hinzu kommt, dass die Implementierungen in den verschiedenen Browsern nicht immer komplett fehlerfrei sind, sodass je nach Browser des Benutzers Fehler innerhalb der Webanwendung auftauchen können.

Zum aktuellen Zeitpunkt gibt es zudem einige Komponenten, auf welche PWAs keinen Zugriff haben. Ein Zugriff auf Telefonie-Funktionen existiert nicht. PWAs können also keine Anrufe starten oder SMS schreiben und empfangen. Ebenso bleibt der Zugriff auf spezialisierte Geräte verwehrt, da es keine einheitliche Form für den Zugriff auf diese gibt.

Allgemein betrachtet ist zu sagen, dass bereits auf viele Systemressourcen zugegriffen werden kann, wodurch PWAs viele Funktionen nativer Anwendungen übernehmen können. Jedoch bleibt zu beachten, dass für den Zugriff auf diese Ressourcen jeweils der Browser auch die entsprechenden Berechtigungen benötigt.

3.5 Browserspezifische Voraussetzungen

Nachdem nun besprochen wurde, welche Voraussetzungen nötig sind, um das Funktionieren einer PWA zu gewährleisten, werden nun die Voraussetzungen aufgezählt, welche die einzelnen Browser an eine Webanwendung stellen, um diese als PWA einzuordnen.

Hierbei wird ein besonderer Schwerpunkt auf die Browser *Google Chrome*, *Mozilla Firefox* und *Microsoft Edge* gelegt. In deren Dokumentation[25, 26, 27] existiert ein Abschnitt „Add to Home Screen (A2HS)“, welcher sich mit dem Installationsvorgang einer PWA befasst. A2HS beschreibt die Installation einer PWA, wobei dem Nutzer zunächst ein Installationsdialog gezeigt wird, dessen Bestätigung die PWA lokal verfügbar macht und eine Verknüpfung zur jener auf dem Home Screen des Nutzers erstellt.

Google Chrome Damit der Google Chrome Browser einen Installationsdialog zum Installieren einer PWA anzeigt, ist zunächst einmal die wichtigste Voraussetzung, dass die aktuell betrachtete Webanwendung nicht bereits installiert ist. So soll sichergestellt werden, dass in erster Linie die Notwendigkeit besteht, die PWA zu installieren. Diese Notwendigkeit wird des Weiteren daran festgelegt, dass der Nutzer eine heuristische Erwartung erfüllt, die Anwendung zu installieren. Wird aus der Interaktion eines Nutzers mit einer Webanwendung nicht erkenntlich, dass er bereit wäre, diese zu installieren, so wird kein Dialog angezeigt. Zu dieser Heuristik äußert sich Google nicht weiter, gibt aber

bekannt, dass in einer früheren Version eine „30 sekündige Interaktion mit der Domain“ notwendig gewesen war. Ist also erkennbar, dass ein Nutzer Installationsabsichten besitzt, werden die Kriterien der PWA geprüft. Die PWA muss einen Service Worker registriert haben und über HTTPS an den Nutzer ausgeliefert werden, um für eine Installation in Frage zu kommen. Zudem muss eine Manifestdatei bestehen, welche mindestens den Namen der PWA, eine Start-URL und Icons in der Größe 192×192 sowie 512×512 Pixel definiert.

Mozilla Firefox Die Kriterien des Firefox Browsers sind weniger streng als die von Google Chrome. Hier wird auch verlangt, dass eine PWA über HTTPS an den Nutzer ausgeliefert wird und dass sie eine Manifestdatei enthält. Eine bestimmte Interaktion des Nutzers mit der PWA wird nicht verlangt. Die Manifestdatei muss ähnlich wie bei Google Chrome den Namen der PWA, eine Start-URL und passende Icons definieren. Zudem wird noch das Definieren einer Hintergrundfarbe im Manifest verlangt.

Microsoft Edge Auch der Edge Browser stellt ähnliche Anforderungen an eine PWA. Es wird erwartet, dass die PWA über HTTPS an den Nutzer ausgeliefert wird und Service Worker registriert hat. Ebenso wird eine Manifestdatei verlangt, jedoch nicht angegeben, welche Felder diese mindestens definieren muss. Da der Unterpunkt „Web app manifest“ in der Dokumentation des Edge Browsers auf die von Mozilla verweist, ist anzunehmen, dass die Anforderungen an die Manifestdatei denen der anderen Browser entsprechen.

Weitere Browser Weitere Browser wie Opera[28], Samsung Internet[29] oder der UC Browser[30] definieren ähnliche Kriterien für die Installation einer PWA. Sowohl das Ausliefern einer PWA über HTTPS als auch das Vorhandensein von Service Workern zieht sich als roter Faden durch die Kriterien. Des Weiteren wird von jedem Browser die Manifestdatei der PWA verlangt.

Die Voraussetzungen der verschiedenen Browser sind größtenteils identisch. Es ist festzustellen, dass die Voraussetzungen größtenteils das abdecken, was zum Installieren einer PWA notwendig ist. Service Worker sind notwendig, um die PWA lokal verfügbar zu machen und ohne HTTPS können Service Worker nicht verwendet werden. Dadurch sind die meisten Voraussetzungen der Browser bereits abgedeckt. Des Weiteren wird die Manifestdatei vorausgesetzt, welche definiert, wie die PWA installiert wird. Weitere Voraussetzungen sind nicht nötig, um die Anwendung installieren zu können. Die von Google Chrome definierten Voraussetzung, dass ein Nutzer zunächst mit einer Domain interagieren muss, soll lediglich das Nutzererlebnis verbessern.

4 Vorteile von Progressive Web Apps

PWAs bringen einige Vorteile mit sich. Nutzern wird durch PWAs ein natives Nutzererlebnis für häufig genutzte Webanwendungen geboten. Zudem verbrauchen PWAs zumeist deutlich weniger Speicher als eine native Anwendung, da nur die Manifestdatei, ein Icon und einige lokale Dateien gespeichert werden. Gerade für Nutzer von Mobilgeräten ist dies von Vorteil, da solche Geräte wenig Ressourcen bieten, die durch PWAs weniger belastet werden.

Die größten Vorteile entstehen jedoch aus Sicht der Entwickler und Anbieter einer PWA. Durch Service Worker und Manifestdateien ist es möglich, eine bereits bestehende Webanwendung ohne großen Mehraufwand zu einer PWA zu erweitern. Die Entwicklungskosten sind im Vergleich zu einer nativen Anwendung deutlich geringer, da bereits bestehende Strukturen erweitert werden können. Für die Entwicklung einer nativen Anwendung müssen zunächst alle Strukturen auf der entsprechenden Plattform neu errichtet werden. Zudem bieten PWAs bessere Möglichkeiten zur Interaktion mit dem Nutzer. Es ist für Nutzer sehr einfach, Inhalte zu teilen, da diese über URLs zu erreichen sind. Wenn Nutzer zum Teilen und Weiterverbreiten der Anwendung ermutigt werden, ist zu erwarten, dass eine PWA mehr Nutzer erreichen wird als eine native App. Durch Push-Benachrichtigungen können Nutzer dann zu weiterer Interaktion mit der Anwendung animiert werden. Ein weiterer Vorteil für Entwickler ist, dass Aktualisierungen sofort bei Nutzung der PWA installiert werden können. Service Worker ermöglichen es, immer die aktuellsten Dateien herunterzuladen, ohne dass der Nutzer manuell eine Aktualisierung durchführen muss. So können neue Features leichter durchgesetzt werden und die gezwungene Unterstützung für ältere Versionen entfällt.

Die meisten Vorteile von PWAs entstehen also für Entwickler und Anbieter, welchen schnellere und günstigere Entwicklung ermöglicht wird. Zusammen mit einer höheren Nutzerinteraktion und Motivation kann sich eine PWA so durchaus als lukrativ erweisen. Für Nutzer bieten PWAs ein natives Nutzererlebnis und die Aussicht auf ressourcenschonende Anwendungen.

5 Kritik an Progressive Web Apps

Die größten Kritikpunkte an PWAs entstehen durch die Kerneigenschaften dieser. Als Webanwendungen müssen sie von einem Browser ausgeführt werden, welcher zumeist eine hochkomplexe Anwendung ist. Dies ist gerade auf Mobilgeräten ein Problem, welche durch eine Batterie nur begrenzte Spannungsversorgung haben. Das Ausführen des Browsers zusammen mit der eigentlichen Anwendung erhöht den Batterieverbrauch, wodurch gerade Nutzer auf schwächeren Geräten davon absehen werden, die PWA zu verwenden. Ein weiteres Problem ist der Zugriff auf Systemressourcen und Geräte durch den Browser. PWAs können längst nicht alle Aktivitäten ausführen, zu denen native Anwendungen im Stande sind. Des Weiteren entsteht die Problematik, dass verschiedene Browser verschiedene Schnittstellen für PWAs bieten. Es kann also nicht garantiert werden, dass eine PWA für jeden Nutzer funktioniert. Gerade bei iOS-Geräte wird dies zu einem Problem, da auf dieser Plattform die Unterstützung für PWAs erst spät eingeführt wurde.

Ein weiterer Aspekt, der kritisch betrachtet werden muss, ist die Sicherheit von PWAs. Im Vergleich zu nativen Anwendungen gibt es nämlich keine einheitliche Instanz, die für den Vertrieb und die Kontrolle der Anwendungen zuständig ist. So muss dem Anbieter der PWA vertraut werden, dass diese sowohl sicher als auch frei von eventueller Schadsoftware ist. Die Möglichkeit, mit einer PWA Schadsoftware zu installieren, ist besonders in Hinsicht auf Service Worker bedenklich, welche den Kern einer PWA ausmachen. Die Position von Service Workern ermöglicht es, alle von einer Anwendung gestellten Netzwerkanfragen zu unterbinden und umzuleiten, wodurch sie ein großes Sicherheitsrisiko darstellen. Zwar wird durch die Spezifikation von Service Workern verhindert, dass Dritte diese manipulieren. Eine Absicherung dagegen, dass der Anbieter der PWA die Service Worker zum Ausspionieren der Nutzer verwendet, gibt es nicht.

Die größten Kritikpunkte liegen also im Bereich der Sicherheit, da PWAs ohne weitere Kontrolle verbreitet und installiert werden können. Jedoch ist auch zu bedenken, dass der Funktionsumfang einer PWA immer an den Browser eines Nutzers angepasst werden muss, was zur Folge hat, dass manche Nutzer die Webanwendung nur eingeschränkt oder überhaupt nicht verwenden können.

6 Fazit

Als Fazit ergibt sich, dass PWAs weder herkömmliche Webanwendungen noch native Anwendungen vollständig ersetzen können. Mit dem Anspruch, die Lücke zwischen diesen beiden Kategorien von Anwendungen zu füllen, verbinden PWAs zwar viele gute Eigenschaften beider Welten, können jedoch nicht alle Einsatzgebiete übernehmen.

Für Nutzer bieten PWAs eine nahezu nahtlose Integration in die bereits bestehenden Strukturen nativer Apps und bestechen zudem mit geringerem Speicherverbrauch und schnelleren Ladezeiten. Viel mehr wird aber aus Nutzersicht nicht geboten. Hinzu kommt, dass PWAs keine eigenständigen Anwendungen sind und daher immer die Unterstützung des Browsers benötigen, welcher viele Ressourcen benötigt, was sich besonders im Akkuverbrauch auf Mobilgeräten bemerkbar macht.

Für Entwickler haben PWAs mehr zu bieten. Eine einfache Migration von existierenden Webanwendungen, sofortige Aktualisierungen für die gesamte Nutzerbasis und ein breiter Zugriff auf Systemressourcen eröffnen für Webanwendungen völlig neue Gebiete und ermöglichen eine schnellere Softwareentwicklung, wodurch Entwicklungskosten sinken. Für Entwickler bleibt dabei jedoch zu beachten, dass die Unterstützung für PWAs browserspezifisch ist, was die Testkosten erhöhen kann oder im schlimmsten Fall, die Nutzer einer PWA einschränkt.

Voraussichtlich werden sich PWAs in Bereichen durchsetzen, wo ein Großteil der Inhalte in Echtzeit aus dem Internet stammt. Sozial Media, Online-Zeitschriften und Videoplattformen werden wohl am meisten von PWAs profitieren. Anstelle viele native Anwendungen zusätzlich zu der bereits existierenden Webanwendung zu entwickeln, kann eine bereits bestehende Webanwendung zu einer PWA weiterentwickelt werden und so eine große Anzahl an Nutzern erreichen. Für Seiten mit statischen Inhalten wird es kaum lohnenswert sein, in den zusätzlichen Entwicklungsaufwand für eine PWA zu investieren. Ebenso werden native Anwendungen nicht verdrängt werden, da der direkte Systemzugriff für einige Bereiche notwendig ist, da beispielsweise Telefonie nicht aus einer Webanwendung heraus möglich ist.

Wie stark sich PWAs tatsächlich durchsetzen werden, wird lediglich die Zeit zeigen können, mit welcher PWAs wohl zunehmend an Unterstützung gewinnen werden. Welche Konsequenzen sich daraus ergeben werden, sollte ein Großteil unserer Anwendungen tatsächlich von verteilten Anbietern aus dem Internet stammen statt einer zentralen Stelle, wie es bisher der Trend war, ist auch noch nicht abzusehen.

Literatur

[1] Alex Russell. *Progressive Web Apps: Escaping Tabs Without Losing Our Soul*. URL: https://infrequently.org/2015/06/progressive-apps-escaping-tabs-without-losing-our-soul/ (besucht am 18. 10. 2019).

[2] *Dokumentation für UWP: Entwickler von UWP-Apps - Windows UWP applications | Microsoft Docs*. URL: https://docs.microsoft.com/de-de/windows/uwp/ (besucht am 19. 10. 2019).

[3] Marcos Caceres. *Packaged Web Apps (Widgets) - Packaging and XML Configuration (Second Edition)*. WD not longer in development. https://www.w3.org/TR/2018/OBSL-widgets-20181011/. W3C, Okt. 2018.

[4] Tim Fisher. *List of Executable File Extensions*. 22. Sep. 2019. URL: https://www.lifewire.com/list-of-executable-file-extensions-2626061 (besucht am 19. 10. 2019).

[5] *Progressive Web Apps | Google Developers*. URL: https://developers.google.com/web/progressive-web-apps (besucht am 14. 10. 2019).

[6] Jake Archibald, David Barrett-Kahn und Anne van Kesteren. *Service workers explained*. 3. Nov. 2017. URL: https://github.com/w3c/ServiceWorker/blob/master/explainer.md (besucht am 19. 10. 2019).

[7] Alice Wyman, Michael Verdi, Swarnava Sengupta u. a. *Create a desktop shortcut to a website | Firefox Help*. URL: https://support.mozilla.org/en-US/kb/create-desktop-shortcut-website (besucht am 28. 10. 2019).

[8] Christine Kopaczewski. *How to add a Google Chrome shortcut icon to your desktop - Business Insider Deutschland*. 7. Juni 2019. URL: https://www.businessinsider.de/how-to-add-google-chrome-icon-to-desktop (besucht am 28. 10. 2019).

[9] *Manifest - Name and Short Name - Google Chrome*. URL: https://developer.chrome.com/apps/manifest/name (besucht am 28. 10. 2019).

[10] Matt Gaunt und Paul Kinlan. *The Web App Manifest | Web Fundamentals | Google Developers*. URL: https://developers.google.com/web/fundamentals/web-app-manifest (besucht am 28. 10. 2019).

[11] Pete LePage. *Add a web app manifest | web.dev*. 5. Nov. 2018. URL: https://web.dev/add-manifest (besucht am 28. 10. 2019).

[12] *ServiceWorker-8a0968f1b295f1ff.png*. 16. Juni 2017. URL: https://www.heise.de/developer/imgs/06/2/2/1/9/1/4/9/ServiceWorker-8a0968f1b295f1ff.png (besucht am 04. 11. 2019).

[13] Alex Russell u. a. *Service Workers Nightly*. 29. Okt. 2019. URL: https://w3c.github.io/ServiceWorker/ (besucht am 04. 11. 2019).

[14] Christian Liebel. *Progressive Web Apps, Teil 2: Die Macht des Service Worker | heise Developer*. 16. Juni 2017. URL: https://www.heise.de/developer/artikel/Progressive-Web-Apps-Teil-2-Die-Macht-des-Service-Worker-3740464.html (besucht am 17. 10. 2019).

[15] James Coops. *Push Notifications Statistics (2019) - Business of Apps*. 6. Nov. 2019. URL: https://www.businessofapps.com/marketplace/push-notifications/research/push-notifications-statistics/ (besucht am 06. 11. 2019).

[16] Tarique Ejaz. *Progressive Web App Push Notifications: Making the Web App more Native in Nature*. 5. Nov. 2017. URL: https://medium.com/@tarique_ejaz/progressive-web-app-push-notifications-making-the-web-app-more-native-in-nature-a167af22e004 (besucht am 06. 11. 2019).

[17] Peter Beverloo und Martin Thomson. *Push API*. W3C Working Draft. https://www.w3.org/TR/2019/WD-push-api-20191003/. W3C, Okt. 2019.

[18] M. Thomson, E. Damaggio und Ed. B. Raymor. *Generic Event Delivery Using HTTP Push*. Internet Engineering Task Force (IETF), Dez. 2016. URL: https://tools.ietf.org/html/rfc8030.

[19] Matt Gaunt. *How Push Works | Web Fundamentals | Google Developers*. URL: https://developers.google.com/web/fundamentals/push-notifications/how-push-works#who_and_what_is_the_push_service (besucht am 11. 11. 2019).

[20] Adam Bar. *What Web Can Do Today*. 4. Mai 2019. URL: https://whatwebcando.today/ (besucht am 11. 11. 2019).

[21] Andrei Popescu. *Geolocation API Specification 2nd Edition*. W3C Recommendation. https://www.w3.org/TR/2016/REC-geolocation-API-20161108/. W3C, Nov. 2016.

[22] Anne van Kesteren. *Notifications API*. WHATWG Standard. https://notifications.spec.whatwg.org/. WHATWG, Sep. 2019.

[23] Bernard Aboba u. a. *Media Capture and Streams*. Candidate Recommendation. https://www.w3.org/TR/2019/CR-mediacapture-streams-20190702/. W3C, Juli 2019.

[24] Alexis Deveria und Lennart Schoors. *Can I use... Support tables for HTML5, CSS3, etc*. 4. Nov. 2019. URL: https://caniuse.com/#home (besucht am 11. 11. 2019).

[25] Pete LePage. *Add to Home Screen | Web Fundamentals | Google Developers*. URL: https://developers.google.com/web/fundamentals/app-install-banners (besucht am 11. 11. 2019).

[26] Erika Doyle Navara u. a. *Progressive Web Apps on Windows - Microsoft Edge Development | Microsoft Docs*. 19. Juni 2018. URL: https://docs.microsoft.com/en-us/microsoft-edge/progressive-web-apps#requirements (besucht am 11. 11. 2019).

[27] Masahiro Fujimoto und Holger Jeromin. *Add to Home screen - Progressive web apps | MDN*. 12. Apr. 2019. URL: https://developer.mozilla.org/en-US/docs/Web/Progressive_web_apps/Add_to_home_screen#How_do_you_make_an_app_A2HS-ready (besucht am 11. 11. 2019).

[28] Andreas Bovens und Bruce Lawson. *Dev.Opera — Installable Web Apps and Add to Home screen*. 23. Sep. 2015. URL: https://dev.opera.com/articles/installable-web-apps/ (besucht am 11. 11. 2019).

[29] *Ambient Badging - Samsung Internet Dev Hub - Resources for developers*. URL: https://hub.samsunginter.net/docs/ambient-badging/ (besucht am 11. 11. 2019).

[30] *Add to Homescreen*. URL: https://plus.ucweb.com/docs/pwa/docs-en/zvrh56 (besucht am 11. 11. 2019).