Progressive Web Apps barrierefrei entwickeln

entwickler.de – entwickler.de Deine Wissensplattform

[…]Weiterlesen…
von Markus Lemcke

Öffentliche Stellen des Bundes sind seit September 2016 gesetzlich dazu verpflichtet, barrierefreie Apps einzusetzen [1]. Progressive Web Apps haben dabei eine großen Vorteil gegenüber nativen Apps: Bei einer nativen App gibt es für drei unterschiedliche Betriebssysteme drei unterschiedliche Richtlinien. Wenn PWAs barrierefrei nach der EN 301 549 [2] entwickelt werden, kann eine App auf drei unterschiedlichen Betriebssystemen eingesetzt werden und die Barrierefreiheit ist auf allen dreien gleich gut.

Seit dem 1. Mai 2002 gibt es das Behindertengleichstellungsgesetz (BGG). Dieses Gesetz wurde im Jahr 2016 überarbeitet und wurde im September 2016 neu verabschiedet. Seit September 2016 sind öffentliche Stellen des Bundes nach § 12a Barrierefreie Informationstechnik Absatz 1 zur Barrierefreiheit bei Apps verpflichtet. Entsprechend haben auch die drei großen Unternehmen Google [3], Apple [4] und Microsoft [5] Richtlinien zur barrierefreien Appentwicklung veröffentlicht.

Bei der Überlegung, wie viele Menschen von barrierefreier Appentwicklung profitieren, stellt man fest, dass es nicht nur 7,8 Millionen schwerbehinderte Menschen [6] betrifft, denn auch einigen der 18,3 Millionen Senioren [7] können barrierefreie Apps das Leben erleichtern.

Vorteile von PWAs

Es gibt native Apps und Progressive Web Apps (PWAs). Als native Apps werden Anwendungen auf mobilen Endgeräten bezeichnet, die speziell für das Betriebssystem des jeweiligen Endgeräts konzipiert und entwickelt wurden. Sie werden meist über die App Stores, die an das Betriebssystem gekoppelt sind, als kostenfreie oder auch kostenpflichtige Anwendungen vertrieben. PWAs hingegen werden mit HTML, CSS und JavaScript entwickelt, sind somit plattformunabhängig und können auf Desktop- und Mobile-Betriebssystemen eingesetzt werden.

Richtlinien, die dafür sorgen, dass Apps von allen Menschen mit unterschiedlichen körperlichen Einschränkungen bedient werden können, haben so viele Prüfungskriterien, wie notwendig sind, um alle körperlichen Beeinträchtigungen zu berücksichtigen. Die Anzahl der Prüfungsschritte bei den Richtlinien von Google, Apple und Microsoft sind unterschiedlich. Das bedeutet, wenn bei einer Richtlinie Prüfungsschritte fehlen, werden Menschen mit bestimmten körperlichen Beeinträchtigungen ausgrenzt.

PWAs sind plattformunabhängig, sie können also auf allen Betriebssystemen eingesetzt werden. Wenn eine PWA nach der EN 301 549 barrierefrei entwickelt wird, ist sie auf allen Betriebssystemen für alle Menschen mit körperlichen Beeinträchtigungen barrierefrei [8].

https://phpconference.com/session-qualification/ipc-webentwicklung/?layout=contentareafeed&widgetversion=1&utmtrackerversion=1&seriesId=oLGXGfBxxeHKh6rMj
Um es auf den Punkt zu bringen: Die Entwicklung von barrierefreien PWAs sorgt dafür, dass alle Menschen mit unterschiedlichen körperlichen Beeinträchtigungen eine App auf jedem Betriebssystem bedienen können.

Richtlinie EN 301 549

Der Standard für Barrierefreiheit von Webinhalten, den das W3C 1999 eingeführt hat und der seitdem global angewendet wird, sind die Web Content Accessibility Guidelines (WCAG). Diese internationale Richtlinie wird kontinuierlich weiterentwickelt und erfuhr 2018 mit den WCAG 2.1 [9] ihre vorerst letzte Aktualisierung.

Die WCAG in ihrer aktuellen Fassung ist die Grundlage für die europäische Norm EN 301 549. Allerdings geht es in der EN 301 549 nicht nur um Barrierefreiheit im Web, sie beinhaltet ebenso Barrierefreiheit bei

- Hardware
- Nicht-Web-Dokumente
- Software
- Dokumentation und unterstützende Dienste

Die EN 301 549 ist also die erste Richtlinie, die der Komplexität des Themas digitale Barrierefreiheit Rechnung trägt und ist deswegen umfassender.

Um den Unterschied zwischen EN 301 549 und WCAG 2.1 deutlich zu machen, hier ein Beispiel: Wenn eine Webseite in Frankreich barrierefrei gemacht werden soll, dann kommt die EN 301 549 zur Anwendung. Bei einer Webseite in Amerika kommt hingegen die WCAG 2.1 zur Anwendung.

Die EN ist eine Richtlinie für Webprogrammierung. Deswegen ist es keine gute Idee, native Apps, die mit Java oder Swift entwickelt werden, nach der EN 301 549 barrierefrei zu entwickeln. PWAs nach der EN 301 549 barrierefrei zu machen, passt perfekt, weil diese mit HTML, CSS und JavaScript entwickelt werden.

Einzelne Kriterien der EN 301 549 umgesetzt

In den folgenden Abschnitten schauen wir uns anhand von praktischen Anwendungsbeispielen an, wie bestimmte Prüfungsschritte umgesetzt werden können.

Screenreadertauglichkeit

Ein Screenreader ist eine Vorlesefunktion für blinde Menschen. Da PWAs plattformunabhängig sind, gibt **Tabelle 1** einen Überblick über Screenreader, die für unterschiedliche Betriebssysteme zur Verfügung stehen, und wie diese aktiviert werden können.

Screenreader	Betriebssystem	Menü
Sprachausgabe	Windows 11	Einstellungen | Barrierefrei-heit | Sprachausgabe [10]
NVDA	Windows 11	Downloadlink https://nvda.bhvd.de/
Talkback	Android 12	Einstellungen | Bedienungshilfen | Talkback [11]
Voice Over	IOS 15.5	Einstellungen | Bedienungshilfen | Voice Over
Voice Over	macOS 10.15	Systemeinstellungen | Bedienungshilfen | Voice Over [12]
Orca	Ubuntu 22.04	Einstellungen | Barrierefreiheit | Bildschirmleser [13]

Tabelle 1: Überblick über unterschiedliche Screenreader

Screenreadertauglichkeit bedeutet, dass eine App-Oberfläche so entwickelt ist, dass sie von Screenreadern vorgelesen werden kann. Um das responsive Webdesign besser verwirklichen zu

können, können Div-Container als Schaltflächen verwendet werden. Warum dies wichtig ist, wird im Abschnitt „Motorische Einschränkungen“ erklärt. Damit der Screenreader weiß, dass der Div-Container eine Schaltfläche ist, muss das *role*-Attribut [14] den Wert *button* bekommen. Um einen Text festzulegen, der von Screenreadern vorgelesen wird, muss das Attribut *aria-label* verwendet und ihm ein Wert zugewiesen werden. Hier ein HTML-Beispiel:

```
<div role="button" aria-label="Quiz starten">START</div>
```

Mit den Attributen *role* und *aria-label* wird der Div-Container zur screenreadertauglichen, responsiven Schalfläche.

PWAs können nicht nur in mobilen Betriebssystemen, Android und IOS, sondern auch auf Desktopbetriebssystemen (Windows, Ubuntu und macOS) ausgeführt werden. Diese Möglichkeit ist für Menschen interessant, bei denen aufgrund einer Behinderung eine motorische Einschränkung in den Händen vorhanden ist. Blinde Menschen können ebenso den Wunsch haben, eine PWA auf einem Computer oder Laptop bedienen zu können. Sobald eine PWA auf einem Computer ausgeführt wird, ist es wichtig, dass sie komplett ohne Maus, also nur per Tastatur bedienbar ist [15]. Deswegen sollten alle Bedienelemente per Tabulatortaste erreichbar sein. Dem Div-Container, der als Schaltfläche verwendet wird, wird deswegen das Attribut *tabindex* hinzugefügt. Hier ein HTML-Beispiel:

```
<div tabindex="0">START</div>
```

Nach dem Hinzufügen des Attributs sind die Div-Schalter per Tabulatortaste erreichbar und somit ist die Grundvoraussetzung des Tastaturbedienbarkeit erfüllt. Dazu gehört auch, dass wichtige Funktionen per Tastenkürzel ausgeführt werden können. Das hilft blinden und sehbehinderten Menschen. Tastenkürzel können mit JavaScript wie folgt realisiert werden:

```
document.addEventListener("keydown", (varevent) => {
```

```
  switch (varevent.key) {
    case "s":
      document.getElementById("btnStart").click();
      break;
    case "w":
      document.getElementById("btnWeiter").click();
      break;
  }
});
```

Der JavaScript-Code sorgt dafür, dass der Schalter Start durch Drücken des Buchstaben *s* und der Schalter Weiter mit dem Buchstaben *w* ausgeführt werden kann.

Sichtbarkeit des Tastaturfokus

Dieses Thema bekommt dann große Bedeutung, wenn eine PWA auf einem Computer oder Laptop ausgeführt wird. Menschen mit einer Sehbehinderung haben Probleme, zu erkennen, welches Bedienelement den Tastaturfokus hat. In Eingabefeldern wird der Textcursor oft als schmaler senkrechter Strich dargestellt. Das ist für Menschen mit einer Sehbehinderung sehr schwer zu erkennen. Microsoft hat im Betriebssystem Windows 11 hierfür eine Lösung. In Einstellungen | Barrierefreiheit | Textcursor kann bei Textcursor-Indikator [16] die Darstellung des Textcursors angepasst werden.

In PWAs kann der App-Entwickler dafür sorgen, dass aktive Bedienelemente die Farbe Gelb als Hintergrundfarbe zugewiesen bekommen. Wenn ein Bedienelement bei Aktivierung eine gelbe Hintergrundfarbe bekommt, ist das für Menschen mit Sehbehinderung sofort sichtbar. Das kann mit CSS sehr einfach gelöst werden:

```
#btnStart:focus{background-color: yellow; color: black;}
```

Barrierefreier Farbkontrast

Menschen mit einer Farbfehlsichtigkeit können nicht immer einer Farbe den richtigen Namen zuordnen. Ihnen fehlt das Gefühl, welche Farben zusammenpassen. Ein Text mit einer dunklen Schriftfarbe auf einer dunklen Hintergrundfarbe ist für sie ebenso wenig erkennbar wie Text mit einer hellen Schriftfarbe auf einer hellen Hintergrundfarbe. Für diese Menschen ist es wichtig, dass eine App-Oberfläche einen barrierefreien Farbkontrast zwischen Schriftfarbe und Hintergrundfarbe [17] hat. Die Überprüfung des Farbkontrasts auf Barrierefreiheit einer App-Oberfläche kann mit der kostenlosen Software Colour Contrast Analyser [18] vorgenommen werden.

Zur generellen Frage der Farbgestaltung von PWA-Oberflächen ist es auch möglich, sich Anregungen von Material Design [19] von Google zu holen.

Motorische Einschränkungen

Motorische Einschränkungen betreffen „kleine Bewegungen“ (Feinmotorik) und „große Bewegungen“ (Grobmotorik). Menschen mit motorischen Einschränkungen in den Händen können Probleme haben, zu kleine Schaltflächen anzutippen. Google führt deswegen in seinen Richtlinien zur barrierefreien Appentwicklung das Kriterium „Use large, simple controls“ auf [20]. Hier empfiehlt Google eine Mindestgröße von Schaltflächen von 48dpx48dp. Bei der Entwicklung von PWAs wird empfohlen, diese Mindestgröße umzusetzen, weil sie auch von dem Accessibility Scanner (so heißt das Überprüfungstool von Google) kontrolliert wird.

Menschen mit motorischen Einschränkungen können auf die Idee kommen, eine PWA auf einem Tablet, iPad oder Computer zu bedienen, mit der Hoffnung, dass dort die Schaltflächen größer dargestellt werden. Wie im Abschnitt Screenreadertauglichkeit

erklärt, ist das der Grund, warum die Schaltflächen nicht als JavaScript-Buttons

```
<button onclick="myFunction()">Click me</button>
```

sondern als Divs

```
<div role="button" aria-label="Quiz starten" tabindex="0">START</div>
```

definiert werden.

Mit Media Queries (gehört zu Cascading Style Sheets) kann dafür gesorgt werden, dass bei einer Displaygröße von 768 x 1024 die Schaltflächen größer dargestellt werden. Folgender CSS-Code zeigt, wie es geht:

```
@media only screen and (min-width: 768px){
  .schalter{
    height: 6.0rem;
  }
}
```

Es wird die Displaygröße 1024 × 768 abgefragt, die bei Tablets und iPads oft anzutreffen ist. Die Abfrage funktioniert ebenfalls bei Bildschirmauflösungen von Computern und Laptops. Wenn die Displaygröße zutrifft, werden die Div-Container, welche eine CSS-Klasse *Schalter* besitzen, in der Höhe auf 6.0 rem angepasst. Somit werden die Schaltflächen größer und Menschen mit motorischen Einschränkungen in den Händen haben es leichter, eine Schaltfläche mit dem Finger oder der Computermaus anzutippen.

Übernahme von Einstellungen des Betriebssystems

Bestimmte Personengruppen mit körperlichen Einschränkungen nehmen grundsätzliche Anpassungen im Betriebssystem vor mit der Erwartung, dass die Apps diese Einstellungen übernehmen. Sehbehinderte Menschen passen im Betriebssystem die

Schriftgröße an. Menschen mit einer Farbfehlsichtigkeit können im Betriebssystem den hohen Kontrast aktivieren.

PWAs sollen, wenn installiert, so plattformnah wie möglich aussehen und sich auch so verhalten. Deswegen ist dieses Thema etwas komplex. Für Menschen mit einer Sehbehinderung ist es wichtig, dass die App-Oberfläche vergrößert bzw. gezoomt werden kann. Wie dies auf unterschiedliche Betriebssysteme umgesetzt wird, zeigt Tabelle 2.

Betriebssystem	**App-Oberfläche zoomen**
Windows	Im App-Menü (3 senkrechte Punkte) gibt es einen Menüpunkt Zoomen
Ubuntu	Im App-Menü (3 senkrechte Punkte) gibt es einen Menüpunkt Zoomen
macOS	Im App-Menü (3 senkrechte Punkte) gibt es einen Menüpunkt Zoomen
Android	Einstellungen | Bedienungshilfen | Text und Anzeige | Anzeigegrösse
IOS	Bedienungshilfen | Zoom

Tabelle 2: Vergrößern der App-Oberfläche auf unterschiedlichen Betriebssystemen

An diesem Beispiel wird deutlich, dass es keine einheitliche Regel gibt. Bei drei Betriebssystemen wird das Zoomen in der App-Oberfläche umgesetzt und bei zwei Betriebssystemen wird die Einstellung im Betriebssystem übernommen. Dieses Wissen ist wichtig, wenn App-Entwickler bestimmte Barrierefreiheitsfunktionen auf unterschiedlichen Betriebssystemen testen möchten.

Progressive Web Apps auf Barrierefreiheit validieren

Nach der App-Entwicklung mit Barrierefreiheit im Blick muss

zum Schluss herausgefunden werden, ob alles tatsächlich wie gewünscht funktioniert. Es gibt zwei grundsätzliche Methoden, PWAs auf Barrierefreiheit zu testen: ein automatisierter Test oder ein Test von Hand aufgrund der Richtlinien EN 301 549.

Zunächst wird der automatisierte Test mit dem Accessibility Scanner [21] betrachtet. Dieser ist für Smart-phones und Tablets ab Android 6.0 geeignet [22]. Auf einem Tablet mit Android 12 ist er erfreulicherweise schon vorinstalliert. Mit dem Accessibility Scanner können native Apps und PWAs auf Barrierefreiheit überprüft werden. Aktiviert wird er in Einstellungen | Bedienungshilfen | Accessibility Scanner. Zunächst wird eine Erlaubnis benötigt, dass der Scanner über anderen Apps eingeblendet werden darf. In den Einstellungen können das Textkontrastverhältnis, das Bildkontrastverhältnis und die Größe des Berührungsbereichs angepasst werden.

Es gibt zwei Möglichkeiten, mit dem Accessibility Scanner eine PWA auf Barrierefreiheit zu überprüfen: „Aufnehmen“ und „Snapshot“. „Aufnehmen“ macht jedes Mal ein Screenshot, wenn sich die App-Oberfläche ändert. Mit dieser Methode kann man sehr schnell eine komplette PWA auf Barrierefreiheit überprüfen. Bei „Snapshot“ kann der App-Entwickler festlegen, wann er die App-Oberfläche überprüfen möchte. Wenn der Accessibility Scanner Fehler findet, dann sind die Fehlermeldungen leicht verständlich formuliert, sodass der App-Entwickler weiß, was er zur Behebung tun muss.

Eine weitere Möglichkeit, die Barrierefreiheit einer PWA zu überprüfen, ist das kostenlose Tool Google Lighthouse [23]. Google Lighthouse ist im Browser Google Chrome in den Entwickler-Tools zu finden. Folgende Schritte müssen umgesetzt werden, um eine PWA mit Google Lighthouse auf Barrierefreiheit zu überprüfen: Die PWA in Google Chrome öffnen. Die Entwickler-Tools öffnen über das Menü Weitere Tools | Entwickler-Tools oder mit der Tastenkombination STRG + UMSCHALT + I. Jetzt das Menü „>>“ aktivieren und dann das Menü Lighthouse auswählen. Hier den Modus auf Navigation (Default)

lassen. Bei Gerät die Option Mobil auswählen. Bei Categories die Option Bedienungshilfen auswählen. Jetzt mit Aktivieren des Schalters Analyze page load die Analyse auf Barrierefreiheit starten.

Wenn die Analyse beendet ist, wird ein Kreis angezeigt, in dem eine Zahl steht. Die Zahl 100 ist die beste Bewertung, die erreicht werden kann. Als Erstes werden Prüfungskriterien angezeigt, die nicht bestanden wurden. Weiter unten werden Elemente angezeigt, die manuell geprüft werden müssen. Noch weiter unten werden *Bestandene Prüfungen* und ganz unten *Nicht zutreffend* angezeigt. In der Kategorie *Zusätzliche Elemente zur manuellen Überprüfung* gibt es noch einen Link zur Seite „How To Do an Accessibility Review“ [24]. Hier gibt Google-Mitarbeiter Rob Dodson Tipps in Form eines YouTube-Videos und Text, wie eine Überprüfung auf Barrierefreiheit durchgeführt werden kann.

Die dritte Möglichkeit ist, die PWA nach den 98 Prüfungsschritten der EN 301 549 von Hand zu prüfen. Zusätzlich ist es hilfreich, die App von Menschen mit Behinderungen testen zu lassen. Ob beispielsweise eine PWA für blinde Menschen bedienbar ist, weiß ein blinder Mensch am besten.

Fazit

Das Entwickeln von barrierefreien PWAs ermöglicht es, barrierefreie Apps zu entwickeln, die auf allen Betriebssystemen den gleichen Standard in Sachen Barrierefreiheit zu Verfügung stellen. Das bedeutet, dass es keine Personengruppe gibt, die von der Nutzung einer App auf einem bestimmten Betriebssystem ausgeschlossen wird.

Markus Lemcke ist seit elf Jahren selbstständig im Bereich Barrierefreiheit von Webseiten, Software und Betriebssystemen. Er ist Dozent an Hochschulen und schreibt als Autor für Fachmagazine. Sein Schwerpunkt ist die barrierefreie Softwareentwicklung mit Java und C#.

Links & Literatur

[1] https://www.gesetze-im-internet.de/bgg/__12a.html

[2] https://www.etsi.org/deliver/etsi_en/301500_301599/301549/03.02.01_60/en_301549v030201p.pdf

[3] https://developer.android.com/guide/topics/ui/accessibility/apps

[4] https://developer.apple.com/library/archive/documentation/UserExperience/Conceptual/iPhoneAccessibility/Making_Application_Accessible/Making_Application_Accessible.html

[5] https://docs.microsoft.com/de-de/windows/apps/design/accessibility/developing-inclusive-windows-apps

[6] https://www.destatis.de/DE/Themen/Gesellschaft-Umwelt/Gesundheit/Behinderte-Menschen/_inhalt.html

[7]

https://www.destatis.de/DE/Themen/Querschnitt/Demografischer-Wandel/Aeltere-Menschen/bevoelkerung-ab-65-j.html

[8] https://www.bitvtest.de/bitv_test/das_testverfahren_im_detail/pruefschritte.html

[9] https://www.w3.org/TR/WCAG21/

[10] https://www.youtube.com/watch?v=vAwf4WFq1jg

[11] https://www.youtube.com/watch?v=ABdOTZUGZR0

[12] https://www.youtube.com/watch?v=6MjR498CyMM

[13] https://www.youtube.com/watch?v=HIKGTi8O9KQ

[14] https://webtest.bitv-test.de/index.php

[15] https://webtest.bitv-test.de/index.php?a=di&iid=267&s=n

[16] https://www.youtube.com/watch?v=cjl0gTkAVc8

[17] https://webtest.bitv-test.de/index.php?a=di&iid=260&s=n

[18] https://www.tpgi.com/color-contrast-checker/

[19] https://material.io/design/color/the-color-system.html

[20] https://developer.android.com/guide/topics/ui/accessibility/apps

[21] https://support.google.com/accessibility/android/answer/6376570?hl=de

[22] https://www.youtube.com/watch?v=GRV1kucMqIo

[23] https://developers.google.com/web/tools/lighthouse

[24] https://web.dev/how-to-review/

Progressive Web Apps: Service Worker und mehr

Progressive Web Apps: Service Worker und mehr

[expand title="mehr lesen…"]

Dienst am Kunden

- Progressive Web Apps
- Addy Osmani: Getting started with Progressive Web Apps
- Understanding Progressive Enhancement (Aaron Gustafson)
- Graceful degradation versus progressive enhancement
- What is a Polyfill?
- Nolan Lawsons Progressive web apps reading list
- Is ServiceWorker Ready?
- Can I use Service Workers
- Web App Manifest. Living Document
- Configuring Web Applications
- Compare stats
- About httparchive

Progressive Web Apps: Service Worker und mehr

Dienst am Kunden

Sebastian Springer

Designer kennen Progressive Enhancement seit gut zehn Jahren: Webseiten zunächst so zu planen, dass alle etwas damit anfangen können. Entwickler sogenannter Web-Apps hauchen mit diesem Prinzip ihren Webdokumenten die Eigenschaften von Apps ein.

iX-TRACT

Das Konzept der Progressive Web Apps bündelt eine Sammlung von Techniken und Best Practices, nach denen man eine Webapplikation aufbaut, die mehr wie eine App denn wie ein klassisches Webdokument wirkt.

Service Worker sind an der Schnittstelle zum Server angesiedelte Hintergrundprozesse, die Netzanfragen abfangen sowie beantworten können. Sie helfen, eine Web-App offlinefähig zu gestalten.

Weitere in Progressive Web Apps verwendete Techniken sind ein Manifest, das das Verhalten der Anwendung festlegt, sowie Push Notifications, über die Nachrichten an den Nutzer verschickt werden können.

Bei Progressive Web Apps (PWAs) verschwimmen durch den Einsatz fortgeschrittener Browsertechniken die Grenzen zwischen App und Webseite. Eine solche Anwendung kann auf einem Gerät installiert und selbst ohne bestehende Internetverbindung verwendet werden. Mit den den PWAs zugrunde liegenden Techniken wollen Browserhersteller, allen voran Google, den Betrieb von Applikationen auch über schwache und unzuverlässige Internetverbindungen ermöglichen, ohne auf die aktive Kommunikation mit dem Benutzer zu verzichten.

Progressive Web Apps sind weniger ein konkretes Muster, nach

dem jemand eine Applikation erstellt, sondern mehr eine Sammlung von Techniken und Best Practices, nach denen man eine Webapplikation aufbauen sollte. Der Schlüssel zum Erfolg von PWAs ist die Tatsache, dass diese Anwendungen überall verfügbar sind und mit nahezu jedem Browser und auf einer Vielzahl von Systemen zum Einsatz kommen können. Mit sogenannten Service Workern kann man eine PWA vollständig offline betreiben oder nur Teile der Applikation nachladen, was den entstehenden Netz-Traffic erheblich verringert.

Einige Komponenten einer PWA erfordern es, dass die Kommunikation ausschließlich über gesicherte Verbindungen stattfindet, was letztlich den Benutzern zugutekommt. Wie erwähnt verbinden PWAs die Vorteile einer Webseite mit denen einer App. Ein Benutzer ist in der Lage, mit einem Klick die Anwendung auf seinem System zu installieren, verliert darüber aber nicht die Option, zur Navigation innerhalb der Applikation Links zu verwenden oder Bookmarks für bestimmte Status zu erstellen und diese mit anderen zu teilen. Durch den Einsatz von Standardtechniken wie HTML, CSS und JavaScript können auch Suchmaschinen PWAs auffinden. Gerade Google unterstützt diese Art von Anwendungen und stellt selbst zahlreiche Ressourcen wie eine umfangreiche Dokumentation und etliche Beispielapplikationen zur Verfügung. Ihre Stärke spielen PWAs vor allem bei mehrmaliger und regelmäßiger Verwendung aus, da hier die Caching-Strategien greifen und die Performance der Applikation spürbar verbessern. Ziel einer PWA ist es, eine solche Art der Verwendung zu unterstützen und aktiv mit dem Benutzer zu kommunizieren.

Progressive Enhancement: Kern einer PWA

Wie der Name andeutet, basieren PWAs auf dem Konzept des Progressive Enhancement. Das Gegenstück dazu ist Graceful Degradation, was in der Vergangenheit vielen Webapplikationen als Grundlage diente. Hierbei entwickeln und testen die Webprogrammierer die Applikationen nur für die neuen Versionen

der Mainstream-Browser. Falls noch Zeit und Budget übrig bleibt, finden noch kleinere Anpassungen statt, sodass auch ein älterer Browser die Applikation verwenden kann. Das führt dazu, dass sie sich nur noch eingeschränkt nutzen lässt, weil einige Features nur umständlich oder überhaupt nicht nutzbar sind.

Progressive Enhancement stellt dagegen den Inhalt einer Applikation in den Vordergrund. Es bedeutet, keinerlei Annahmen über die verwendeten Browser zu treffen. Die beste Beschreibung von Progressive Enhancement liefert Aaron Gustafson in einem Blogartikel bei A List Apart (Links wie dieser sind über den blauen Balken „Alle Links“ am Ende des Artikels zu finden), in dem er dieses Konzept mit einem Erdnuss-M&M vergleicht. Der Kern ist der Inhalt der Anwendung, der aus dem Markup und den Daten besteht. Alle Browser können ihn konsumieren, da es sich lediglich um Struktur und Information handelt.

Cascading Style Sheets (CSS) bilden den Schokoladenmantel von Progressive Enhancement und bereiten den Inhalt optisch ansprechend und gut nutzbar auf. Diese Schicht sorgt außerdem dafür, dass der Inhalt für das jeweilige Gerät und die verwendete Auflösung optimiert dargestellt wird. In dieser Schicht kommen zudem neuere CSS-Features zum Einsatz, die nicht mehr unbedingt jeder Browser unterstützt. Die Schwierigkeiten der Unterstützung lassen sich durch Polyfills lösen. Selbst wenn der Browser einen Style nicht versteht, kann er den Inhalt anzeigen.

Schließlich bildet JavaScript die Zuckerschicht des Progressive Enhancement. Hier kommen die Service Worker, Push Notifications und viele andere Features zum Einsatz. Hinsichtlich der eingesetzten Frameworks und Bibliotheken treffen weder Progressive Enhancement noch PWAs eine Aussage. Eine Applikation kann in reinem JavaScript ohne ein Framework oder mit AngularJS beziehungsweise React erstellt werden. Das Konzept lässt sich auf nahezu jeder Plattform umsetzen. Im

Kern geht es um eine alte Strategie: Separation of Concerns. Bei der Erstellung einer Anwendung soll der Entwickler die Struktur von der Darstellung und der Logik trennen.

Herzstück einer PWA: Service Worker

Schwankungen in der Qualität der Netzverbindung sind für Webentwickler allgegenwärtig. Deshalb ist es logisch, dass eine PWA solche Schwankungen in der Verbindung zum Server ausgleichen können muss. Die Spanne reicht von leicht verzögerten Antworten eines Servers bis hin zum vollständigen Nichtvorhandensein einer Verbindung.

Offlinefähige Applikationen sind in der Webentwicklung nichts Neues. Schon seit Langem unterstützen Browser den Application Cache, bei dem eine Manifest-Datei angibt, welche Dateien ein Browser herunterladen und welche er vom Server beziehen soll. Letztere lädt er einmal vom Server herunter, sie verbleiben danach im Cache des Browsers. Hat der Client eine Verbindung zum Server, prüft Ersterer, ob das Manifest verändert wurde. Wenn das der Fall ist, untersucht der Browser, ob die Dateien im Cache noch aktuell sind. Diese Methode für offlinefähige Webapplikationen lösen in Zukunft sogenannte Service Worker ab. Aus diesem Grund haben sich die Entwickler von Firefox entschlossen, seit Version 44 des Browsers eine Warnmeldung auf der Konsole anzuzeigen, wenn das AppCache-Feature verwendet wird.

Im Gegensatz zum AppCache, der auf Basis einer statischen Datei arbeitet, ist ein Service Worker ein Hintergrundprozess, der an der Schnittstelle zum Server angesiedelt ist und Netzanfragen abfangen sowie beantworten kann. Neben dieser Proxy-Funktion unterstützen Service Worker Features wie Push Notifications und Background Sync. Dazu später mehr.

Da ein Service Worker ein Hintergrundprozess ist, kann die Web-App nicht direkt mit dem DOM interagieren. Service Worker und die Webapplikation tauschen lediglich Informationen aus.

Zu diesem Zweck kommt *postMessage* zum Einsatz. Einen ähnlichen Ansatz verfolgen Web Worker, die man als Hintergrundprozess nutzen kann, um rechenintensive Operationen aus dem Browser-Prozess auszulagern. Sie können ebenfalls mit dem Server kommunizieren. Ihr Schwerpunkt liegt jedoch auf der Ausführung von Applikationslogik.

Service Worker kümmern sich dagegen mehr darum, Infrastruktur für eine Applikation zur Verfügung zu stellen. Um die Abhängigkeit von einer permanent vorhandenen Netzverbindung zu lösen, müssen Webentwickler zunächst einen Service Worker erstellen. Für die lokale Entwicklung gelten keine weiteren Anforderungen. Soll die Applikation jedoch auf einem Server arbeiten, muss man sicherstellen, dass HTTPS verwendet wird. Der Grund dafür ist, dass ein Service Worker an einer zentralen Stelle in der Applikation installiert sein muss. Eine Man-in-the-Middle-Attacke hätte hier fatale Folgen. Eine verschlüsselte Verbindung erschwert das zumindest.

Dienst installieren, aktivieren und beenden

Der Lebenszyklus eines Service Workers besteht aus drei Phasen: Installieren, Aktivieren und nach der Arbeit Terminieren, um die Ressourcen des Systems zu schonen. Der Worker wird aktiviert, sobald er entweder ein *fetch*– oder ein *message*-Event erhält. Dieses ressourcenschonende Beenden des Worker-Prozesses hat zur Folge, dass der Worker selbst nicht langfristig einen Status speichern kann. Sollte das erforderlich sein, kann man auf die IndexedDB des Browsers zurückgreifen.

Um den Worker-Prozess zu registrieren, greift man auf das *serviceWorker*-Objekt der globalen *navigator*-Eigenschaft des Browsers zurück. Die *register*-Methode des *serviceWorker*-Objekts akzeptiert einen Dateinamen als Argument. Die angegebene Datei enthält den tatsächlichen Quellcode des

Service Workers. Die Registrierung sowie zahlreiche andere Funktionen eines Service Workers laufen entkoppelt vom Programmfluss ab (asynchron). Zur besseren Steuerung solcher asynchronen Operationen kommen Promises zum Einsatz. Der Rückgabewert der *register*-Methode ist ein solches Promise-Objekt, über dessen *then*-Methode man Callback-Funktionen für den Erfolgs- und Fehlerfall der Registrierung des Service Workers einbinden kann. Sie werden ausgeführt, wenn die Registrierung beendet ist. Die wiederum ist der einzige Schritt, der direkt in der Applikation stattfindet, alles Weitere setzt die Datei des Service Workers um.

Listing 1: Service Worker registrieren

```
if ('serviceWorker' in navigator) {
    navigator.serviceWorker.register('serviceWorker.js')
        .then(function success(reg) {
            console.log('Registration successful: ', reg);
        }, function failure(err) {
            console.log('Registration failed: ', err);
        });
}
```

Prüft man vor der Registrierung, ob der Browser dieses Feature unterstützt, und führt den Code nur aus, falls dies der Fall ist, erfüllt dies eine wichtige Anforderung: Die PWA funktioniert auch auf Systemen, die ein bestimmtes Feature noch nicht implementiert haben. Verzichtet man auf eine solche Überprüfung, wird eine Exception geworfen, was im ungünstigsten Fall zur Beendigung der Applikation führt. Listing 1 enthält den Quellcode für die Registrierung eines Service-Worker-Prozesses. Er kann beliebig oft ausgeführt werden, da der Browser prüft, ob der Worker schon registriert ist, und den Vorgang in diesem Fall nicht wiederholt.

Der Worker-Prozess ist nur für das Verzeichnis verantwortlich, in dem die Datei liegt. Das bedeutet, dass ein Service Worker, dessen Datei im Document-Root-Verzeichnis des Webservers liegt, für sämtliche Anfragen auf diesen Webserver zuständig

ist. Befindet sich die Datei in einem Unterverzeichnis, fängt er nur Anfragen in dieses Verzeichnis und darunterliegende ab.

Callback für statische Inhalte

Listing 2: Installieren eines Service Workers

```
self.addEventListener('install', function (e) {
    e.waitUntil(
        caches.open('app')
            .then(function (cache) {
                return cache.addAll(['/node/style/style.css'])
            })
    )
});
```

Auf das Registrieren folgt das Installieren. In dieser Phase geht es normalerweise darum, Caches für statische Inhalte wie Stylesheets, HTML- und JavaScript-Dateien sowie Bilder und andere Mediendateien anzulegen und zu befüllen. Das *install*-Event löst diesen Vorgang aus und setzt ihn in Form einer Callback-Funktion um. Die Behandlung des Cache ist ein zweistufiger asynchroner Prozess, der auf Promises basiert, die der JavaScript-Standard ECMAScript seit der Version 6 aus dem Jahre 2015 vorsieht. Zunächst erzeugt der Worker einen Cache mit einem frei wählbaren Namen und füllt ihn im zweiten Schritt mit einem Array von Dateinamen. Wie dieser Prozess genau funktioniert, zeigt Listing 2.

Listing 3: Service Worker aktualisieren

```
self.addEventListener('activate', function (e) {
    caches.keys().then(function (names) {
        return Promise.all(
            names.map(function (name) {
                if (name !== 'v1.0.2') {
                    return caches.delete(name);
                }
            })
        )
```

```
    })
});
```

Nachdem der Service Worker installiert ist, kann man im Aktivierungsschritt weitere Verwaltungstätigkeiten an den Caches durchführen. Typischerweise findet die Aktualisierung alter Cache-Inhalte in diesem Schritt statt. Der Quellcode aus Listing 3 löscht sämtliche Caches, die nicht den Namen v1.0.2 tragen. Das soll sicherstellen, dass nur die neueste Version der Dateien ausgeliefert wird.

Listing 4: Cache Handling

```
self.addEventListener('fetch', function (e) {
    e.respondWith(
        caches.match(e.request)
            .then(function (response) {
                if (response) {
                    return response;
                }

                return fetch(event.request);
            })
    )
});
```

Aktualisiert der Benutzer eine Seite, löst das *fetch*-Events aus, die ihrerseits Service Worker abfangen und beantworten können. Dazu registriert man eine Callback-Funktion für das *fetch*-Event. Die Repräsentation dieses Events verfügt über eine Methode *respondWith*, mit der man steuern kann, wie die Antwort an den Browser aussehen soll. Die Methode akzeptiert eine Promise als Argument, die man beispielsweise durch ein *caches.match* erzeugen kann, wenn man im Cache nach einem passenden Treffer sucht. Ist kein Eintrag vorhanden, kann die Anfrage mit einem Aufruf der *fetch*-Funktion an den Server weitergeleitet werden. Listing 4 enthält den Quellcode für eine einfache Bedienung von Anfragen aus dem Cache.

Erzeugt man mit *caches.open* eine Referenz auf einen benannten

Cache, kann ein Worker über die *put*-Methode diesem Cache neue Einträge hinzuzufügen, um zukünftige Anfragen aus dem Cache beantworten zu können.

Hat man sich an den Umgang mit den Promises gewöhnt, stellt der Einsatz von Service Workern in einer Applikation keine große Hürde mehr dar. Man muss sich nur stets vor Augen halten, dass Service Worker Teil von PWAs sind, die auf jedem System funktionieren sollen. Service Worker sollten daher lediglich die Funktionsvielfalt erweitern und die Benutzung verbessern und keinesfalls ein erforderlicher Bestandteil einer Anwendung sein.

Eine PWA sollte installierbar sein. Diese Anforderung gilt besonders für mobile Geräte. Da vor allem Google und Mozilla zu den Unterstützern von PWAs zählen, wundert es nicht, dass die Installierbarkeit einer Webapplikation auf Android-Geräten sich von selbst versteht. Apple geht auf seinen iOS-Geräten einen anderen Weg, der ein wenig Mehraufwand erfordert. Die Konfiguration der Applikation erfolgt für iPhones über verschiedene *link*– und *meta*-Tags. Genauere Informationen hierzu findet man auf Apples Developer-Seiten (siehe *iX*-Link unter „Configuring Web Applications“).

Zurück zur Installation auf einem Android-Gerät. Seit Chrome 39 lässt sich eine Webanwendung über das Browser-Menü auf dem Homescreen installieren.

Apples Safari bietet ein ähnliches Feature. In beiden Fällen handelt es sich nicht wirklich ums Installieren, sondern vielmehr um einen speziellen Link zum Browser, der allerdings dafür sorgt, dass sich eine Webanwendung wie eine native App anfühlt. Dieses Feature erreicht jedoch schnell seine Grenzen, wenn die Internetverbindung nicht zuverlässig ist, da der Browser standardmäßig keinerlei Dateien zwischenspeichert. Das bedeutet, dass man dieses Feature idealerweise mit dem Einsatz von Service Workern kombinieren sollte.

Mit Version 42 hat Chrome ein neues Feature mit dem Namen App Install Banner eingeführt. Diese Browserfunktion präsentiert dem Benutzer ein Banner, das ihm dieselben Optionen wie das bisherige „Add to Homescreen“ bietet, allerdings wesentlich prominenter sichtbar ist.

Wie die „App“ sich verhalten soll

Damit jetzt nicht jede Webseite ein solches Banner einführt und dem Benutzer bei jedem Besuch einer beliebigen Webseite zuerst ein Installationsbanner entgegenkommt, müssen einige Voraussetzungen erfüllt sein: Ein Service Worker muss für die Webapplikation registriert sein, sie muss per HTTPS ausgeliefert werden und über ein Web-App-Manifest verfügen (siehe unten). Damit das App Install Banner zur Anzeige kommt, muss der Benutzer eine Seite mindestens zweimal im Abstand von mindestens fünf Minuten besucht haben. Und für dieses Feature gilt wieder, dass es für Browser, die es unterstützen, einen Mehrwert für den Benutzer bietet, das Nutzungserlebnis für alle anderen allerdings nicht einschränkt.

Listing 5: Web-App-Manifest

```
{
  "short_name": "MyPWA",
  "name": "My first progressive web app",
  "icons": [
    {
      "src": "hello.png",
      "sizes": "96x96",
      "type": "image/png"
    }
  ],
  "start_url": "/index.html",
  "display": "standalone",
  "orientation": "landscape"
}
```

Eine der Anforderungen für das Funktionieren von App Install Banners ist das Vorhandensein eines Web-App-Manifests.

Dahinter steckt nichts anderes als eine Beschreibungsdatei, die festlegt, wie sich die Applikation auf einem Gerät verhalten soll, wenn der Anwender sie als App auf dem Homescreen installiert hat. Diese Manifest-Datei ist nicht mit dem AppCache-Manifest zu verwechseln, das früher beim AppCache zum Einsatz kam. Beim Web-App-Manifest handelt es sich um eine JSON-Datei, die ein *link*-Element einbindet. Browser, die dieses Feature nicht unterstützen, ignorieren das Element schlicht. Von daher erfüllt das Web-App-Manifest die wichtigste Anforderung einer PWA: keine Einschränkung für ältere Browser (siehe Listing 5).

Die verschiedenen Eigenschaften bestimmen, wie sich die App verhält, wenn sie auf einem Gerät installiert ist. So kann man beispielsweise den Namen angeben, den das App Install Banner anzeigen soll. Unter der Eigenschaft *icons* lassen sich Icons in verschiedenen Auflösungen und Dateitypen aufführen – beispielsweise, welches auf dem Homescreen des Benutzers angezeigt wird. Die *start_url* gibt den Startpunkt der Applikation an. Die *display*-Eigenschaft definiert, wie der Browser in Erscheinung treten soll. Der Wert *browser* sorgt dafür, dass alle Browser-Controls wie der Zurück-Button verfügbar sind. *minimal-ui* schränkt die sichtbaren Steuerungselemente weiter ein, und *fullscreen* lädt die Applikation bildschirmfüllend und ohne Steuerungselemente. Mit dem Wert *standalone* übernimmt die Webapplikation das Look-and-Feel nativer Apps. Dieser Wert wird am häufigsten für installierte PWAs verwendet. Die Eigenschaft *orientation* gibt an, ob die Applikation im Hochformat (*portrait*) oder Querformat (*landscape*) zu laden ist. Es gibt weitere Eigenschaften – Genaueres ist dem Working Draft des W3C (siehe den genannten blauen Balken „Alle Links“) zu entnehmen.

Progressive Web Apps sollen die Bindung zum Benutzer stärken und ihn über Änderungen auf dem Laufenden halten. Push Notifications (Listing 7) basieren auf den bisher vorgestellten Service Workern und dem Web-App-Manifest. Der

Vorteil dieser Benachrichtigungen ist, dass der zugrunde liegende Service Worker zunächst terminiert und keine Ressourcen benötigt. Geht eine Nachricht ein, wird der Service Worker aktiv und kann mit der Nachricht umgehen. Das funktioniert sogar, wenn der Browser nicht aktiv ist.

Benachrichtigungen per *push* abonnieren

Listing 6: Push Notifications abonnieren

```
navigator.serviceWorker.register('sw.js').then(function(reg) {
    reg.pushManager.subscribe({
        userVisibleOnly: true
    }).then(function(sub) {
        console.log('endpoint:', sub.endpoint);
    });
});
```

Einer der Sicherheitsmechanismen von Push Notifications ist, dass Anwendungen nicht ohne Weiteres einem Browser eine Benachrichtigung schicken können. Für deren Versand kommen spezielle Dienste wie Googles Cloud Messaging (kurz GCM) zum Einsatz. Nachdem der Entwickler ein Projekt für den Notification Service der Applikation im GCM erstellt hat, muss er die Sender-ID in der Manifest-Datei der PWA eintragen. Danach wird die Applikation für Push Notifications registriert und ein Handler für das *push*-Event erstellt. Beide Schritte enthält der JavaScript-Code der Applikation. Das Abonnement der Push Notifications geschieht während der Registrierung des Service Workers (siehe Listing 6).

Listing 7: Anzeige einer Push Notification

```
self.addEventListener('push', function(event) {
    event.waitUntil(
           self.registration.showNotification('New content
available', {
            body: '5 New datasets available',
            icon: 'images/icon.png'
        }));
```

```
});
```

Event-Handling wiederum findet im Service Worker selbst statt: über das Auslösen des *push*-Events, wenn eine Nachricht eingeht. Die *showNotification*-Methode stellt sicher, dass die Nachricht dem Benutzer angezeigt wird.

Außer *push*– gibt es mit *notificationclick* ein weiteres Event, das beim Umgang mit Push Notifications hilfreich ist. Ein Benutzer löst es durch einen Klick auf eine Benachrichtigung aus; es kann dazu dienen, Browserfenster in den Fokus zu bringen oder neu zu öffnen. Push Notifications sind ein mächtiges Werkzeug, da sie die Aufmerksamkeit des Benutzers auf sich ziehen – und das, obwohl die Applikation zu diesem Zeitpunkt geschlossen sein kann. Aus diesem Grund sollte man mit solchen Benachrichtigungen eher sparsam umgehen, da man damit schnell über das Ziel hinausschießen kann und den Benutzer eher verärgert, was im harmlosen Fall zu einer Deaktivierung der Push Notifications und im schlimmsten Fall zur Abkehr des Benutzers von der Applikation führt. Eine Push Notification sollte man nur im Falle eines für den Benutzer wichtigen Ereignisses versenden – keinesfalls zu Werbezwecken.

Ziel einer PWA ist eine optimale Performance. Um dies zu erreichen, muss die Applikation eine kurze Ladezeit haben. Und genau an diesem Punkt greift die App-Shell-Architektur ein. Die App Shell bezeichnet den Rahmen der Applikation, also nur das fürs Ausführen der App wirklich erforderliche HTML, CSS und JavaScript. Alle weiteren Inhalte kann die Software zu einem späteren Zeitpunkt nachladen. Dieser Rahmen besteht in der Regel aus statischen Inhalten, die der Service Worker gut zwischenspeichern kann. Dynamische Daten lädt er erst, wenn die „Basis“ vorhanden ist.

So zeigt die App dem Benutzer zunächst das Grundgerüst der Applikation, wie die Navigation oder ähnliche sinnvolle Inhalte, die sich nicht allzu häufig ändern. Sobald weitere Inhalte verfügbar sind, werden sie ebenfalls angezeigt. Der

Benutzer erhält auf diese Weise schnelles Feedback und kann die Applikation schon zu einem frühen Zeitpunkt nutzen. Eine solche Architektur eignet sich allerdings nicht für jede Applikationsart. Ihre Stärke kann die App Shell nur in einer dynamischen Applikation ausspielen. Besteht eine Anwendung hauptsächlich aus statischen Inhalten, ist diese Architektur eher hinderlich und wirkt sich durch die zusätzlichen Anfragen für das Nachladen der Daten negativ auf die Performance aus. Eine solche Architektur lässt sich durchaus auf älteren Browsern und auf verschiedenen Plattformen umsetzen. Ihre Vorzüge kommen jedoch vor allem im Zusammenspiel mit den Caching-Fähigkeiten der Service Worker zur Geltung.

Fazit

Obwohl längst nicht alle Browser Service Worker unterstützen (wie man bei caniuse.com nachschlagen kann, siehe „Alle Links“), sollte jeder Webentwickler die Ideen hinter Progressive Web Apps bei der Arbeit beherzigen. Das gilt für die Erweiterung bestehender Applikationen genauso wie für die Erstellung neuer.

Viele entwickeln Webapplikationen nur noch für die neuen Mainstream-Browser. Genau dagegen richten sich die Verfechter von Progressive Web Apps. Dabei verschwimmen die Grenzen zwischen Webdokument und App auf mobilen Geräten, indem eine PWA installierbar und ihr Verhalten über Manifest-Dateien beeinflussbar ist. Service Worker sorgen für einen Performanceschub, indem Entwickler kontrollieren können, wann die PWA welche Anfragen aus dem Cache oder vom Server beantworten soll. Diese Caching-Möglichkeit geht bis zur vollständig offlinefähigen Anwendung, die sich bei einer bestehenden Verbindung mit dem Server synchronisieren kann. PWAs bieten mit diesen Funktionen erheblichen Mehrwert, außerdem stärken sie die Bindung zum Nutzer, indem sie ihn durch Push Notifications über aktuelle Änderungen auf dem Laufenden halten.

Google ist einer der größten Treiber hinter PWAs, aber Mozilla stellt ebenfalls immer mehr Ressourcen für diese Art von Applikationen zur Verfügung. Den Stand der Entwicklung sieht man vor allem bei den Service Workern. Firefox, Chrome und Opera kennen sie schon. Bei Microsoft ist die Unterstützung noch nicht so weit fortgeschritten, mit der Integration wurde bereits begonnen. Einzig Apple mit dem Safari-Browser hinkt im Rennen um die neuen Techniken hinterher. Aber selbst hier gibt es erste positive Anzeichen, dass dieses Feature früher oder später Einzug in den Browser halten wird. (hb) Sebastian Springer arbeitet als JavaScript-Entwickler bei der MaibornWolff GmbH in München. Als Dozent und Autor für JavaScript will er die Begeisterung für professionelle Entwicklung mit JavaScript wecken.

[/expand]