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 hinzufügen, 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]