Kapitel 3

# Zehn Eigenschaften, die PWA einzigartig machen



*Progressive Web Apps sind keine klar definierte Technologie, sondern eine Checkliste von Eigenschaften, die eine solche Anwendung umsetzen soll. Es liegt in den Händen der Entwickler, die Einträge abzuhaken.*

Wenn Entwickler eine Windows-Anwendung schreiben möchten, haben sie die Auswahl aus einer Vielzahl von Technologien: .NET, Java, Delphi, Webtechnologien und viele weitere Entwicklungsplattformen kommen infrage. Letztlich haben diese nur eines gemeinsam: die Ausführungsplattform Windows. Bei Progressive Web Apps verhält sich das ähnlich. Auch hier können Entwickler mit einer Vielzahl von Frameworks wie Angular, React, Vue.js oder sogar komplett ohne Bibliothek arbeiten. PWA ist lediglich ein Gattungsbegriff für einen Typ Webanwendung, der um Funktionsmerkmale angereichert wird, die man früher nur nativen Anwendungen zugeschrieben hätte. Aus diesem Grund kann es auch keine Blaupause zur Entwicklung von Progressive Web Apps geben – diese sind so vielfältig wie die Anwendungsentwicklung selbst. PWAs stellen vielmehr eine Sammlung von Eigenschaften dar, die eine solche Webanwendung umsetzen soll – eine Checkliste von Charakteristiken, an der Entwickler sich entlanghangeln können.

Um welche Eigenschaften es geht und wie diese zu interpretieren sind, unterscheidet sich je nach Quelle. Die erste Definition von Alex Russel, beschrieben in Abschnitt 1.2.4, »Progressive Web Apps als Über-Pattern«, umfasste beispielsweise neun Eigenschaften, Mozilla und Microsoft nennen in ihren Entwicklerdokumentationen acht, bei Google sind es zehn.[1] An diesen orientiere ich mich in diesem Kapitel. Es geht um folgende Eigenschaften:

1. *Progressive*: Anwender älterer Browser sollen nicht ausgeschlossen werden.
2. *Responsive*: Verfügbare Bildschirmabmessungen sollen bestmöglich genutzt werden.
3. *Connectivity Independent*: Die App läuft auch offline oder bei schwacher Verbindung.

1 *https://developers.google.com/web/fundamentals/codelabs/your-first-pwapp/*

4. *App-like*: PWAs erscheinen und verhalten sich wie native Anwendungen.
5. *Fresh*: Die Anwendung ist trotz Offlinekopie immer aktuell.
6. *Safe*: Die Ressource wird über HTTPS ausgeliefert.
7. *Discoverable*: Die PWA ist von einer klassischen Website unterscheidbar.
8. *Re-engageable*: Die Anwendung kann den Benutzer dank Pushbenachrichtigungen zur Wiederverwendung der App bewegen.
9. *Installable*: Die PWA lässt sich installieren.
10. *Linkable*: Die Anwendung ist über eine URL verlinkbar, sogar bis hin zu einem bestimmten Zielzustand (Deep Linking).

Dabei lasse ich die Interpretationen aus den unterschiedlichen Quellen einfließen. Progressive Web Apps sollten im Sinne der Nutzer sämtliche Eigenschaften erfüllen. Es gibt darüber hinaus Charakteristiken, die als absolute Muss-Kriterien angesehen werden. Diese stelle ich in den jeweiligen Unterkapiteln heraus.

## 3.1 Voranschreiten mit Progressive Enhancement

**Kurzdefinition**

Die Eigenschaft *Progressive* bedeutet, dass ältere Webbrowser nicht von der Nutzung einer Progressive Web App ausgeschlossen werden sollen. Setzt der Anwender einen starken Webbrowser ein, erhält er umgekehrt einen erweiterten Funktionsumfang.

Die erste Eigenschaft, Progressive, steckt schon im Namen des gesamten Anwendungsmodells. Schlägt man das Wort in einem Wörterbuch nach, so liest man dort: »Sich in einem bestimmten Verhältnis steigernd, entwickelnd«. Diese Eigenschaft ist auf zwei Arten zu verstehen: Einerseits beschreibt sie das Wesen von Progressive Web Apps: Die von Alex Russell ursprünglich formulierte Idee ist, dass PWAs ihr Leben zunächst in einer Registerkarte des Browsers beginnen. Stellt der Browser fest, dass der Anwender eine PWA häufig besucht, kann er dem Anwender ein Banner anzeigen, das ihn zur Installation der Anwendung auf dem Homebildschirm auffordert. Von dort aus gestartet, wird die PWA in einem eigenen Fenster beziehungsweise im Vollbildmodus ausgeführt. Auf diese Art entwickelt sich die im Browser-Tab ausgeführte Webanwendung also schrittweise, progressiv, zur nativen App. Andererseits beschreibt diese Eigenschaft die abwärtskompatible Natur von Progressive Web Apps. Möglicherweise kennen Sie das Prinzip der *Graceful Degradation*: Dieses beschreibt, dass sich ein System Fehlern gegenüber tolerant verhält, indem es im Fehlerfall Teilfunktionalitäten abschaltet, um den restlichen Betrieb aufrechtzuerhalten. Bei PWAs kommt hingegen die positive Umkehr dieses Prinzips zum Einsatz: *Progressive Enhancement*. Dabei wird bei gegebener Unterstützung seitens des Systems







zusätzliche Funktionalität angeboten, während eine plattformübergreifende Grundfunktionalität gewährleistet wird.

So stehen die PWA-Schnittstellen etwa nicht in allen Browsern und Browserversionen, die von Internetnutzern weltweit verwendet werden, zur Verfügung: In vielen Unternehmen wird nach wie vor der Microsoft Internet Explorer eingesetzt. Dieser Browser wird nicht mehr aktiv weiterentwickelt, aus Gründen der Abwärtskompatibilität zu Altanwendungen jedoch noch am Leben gehalten. Der Browser wird vermutlich nie eine Unterstützung für Service Worker erhalten. In Kapitel 2, »Mächtiges modernes Web«, haben Sie außerdem gesehen, dass auch nicht alle aktuellen Web-APIs auf jeder Plattform unterstützt werden. Progressive Enhancement sagt aber: Das macht nichts! Jeder Benutzer soll die bestmögliche Erfahrung erhalten, die auf seiner Plattform realisiert werden kann. So soll eine Progressive Web App zur Verwaltung von Kundenstammdaten grundsätzlich auch im Internet Explorer lauffähig sein. Denn der funktionale Kern der Anwendung, die Darstellung von Listen und Detailmasken mit Formularen, ist natürlich auch in älteren Browsern problemlos umsetzbar. Auf erweiterte Funktionalität wie Pushbenachrichtigungen muss der Anwender in diesem Fall zwar verzichten, aber die Anwendung verweigert deswegen nicht komplett ihren Dienst.

In vielen Fällen ist es auch möglich, Fallbacks bereitzustellen, sollte eine Plattform eine bestimmte Funktionalität nicht unterstützen: So implementiert der Internet Explorer zwar keine Service Worker und somit auch nicht deren Pushschnittstellen, doch unterstützt er (je nach Version) andere Pushtechnologien, um über extern eingetretene Ereignisse informiert zu werden. Benachrichtigungen auf Betriebssystemebene kann der Internet Explorer ebenfalls nicht anzeigen (übrigens wie auch der mobile Safari), doch gibt es verschiedene Bibliotheken, um Toast-Benachrichtigungen innerhalb des Darstellungsbereichs der Website anzuzeigen. Somit hat der Anwender grundsätzlich Zugriff auf denselben Informationsumfang, zumindest solange er die Website im Vordergrund geöffnet lässt. Umgekehrt erhalten Anwender mit stärkeren Webbrowsern einen erweiterten Funktionsumfang und somit eine bessere Anwendererfahrung. Entwickler können davon ausgehen, dass langsam mehr und mehr Anwender auf modernere Browser umsteigen und die Unterstützung moderner Webschnittstellen durch die Browser stetig zunimmt. Somit kommen nach und nach automatisch mehr Anwender in den Genuss des erweiterten Funktionsumfangs.

Die technische Umsetzung dieses Prinzips nennt sich *Feature Detection*. Eine Anwendung greift nicht blauäugig auf eine Schnittstelle zu, sondern prüft erst, ob sie auf der jeweiligen Plattform überhaupt zur Verfügung steht. Das klingt zunächst banal, doch kann ein Zugriff auf eine nicht existente Schnittstelle dazu führen, dass die komplette Anwendung bricht. Abbildung 3.1 zeigt das am Beispiel einer Demoanwendung aus Kapitel 2, »Mächtiges modernes Web«, die im Internet Explorer 11 ausgeführt wird.

Die Feature Detection ersetzt im Zusammenhang mit modernen Webschnittstellen frühere Methoden zur Auswertung der Browserkennung (*User Agent String*) oder die Verwendung von Browserherstellerpräfixes (z. B. `mozGetUserMedia`, `msGetUserMedia`). Im ersten Fall wurden nachträglich für einen Browser bereitgestellte APIs nie genutzt, im zweiten steht die API nur in diesen Browsern zur Verfügung.

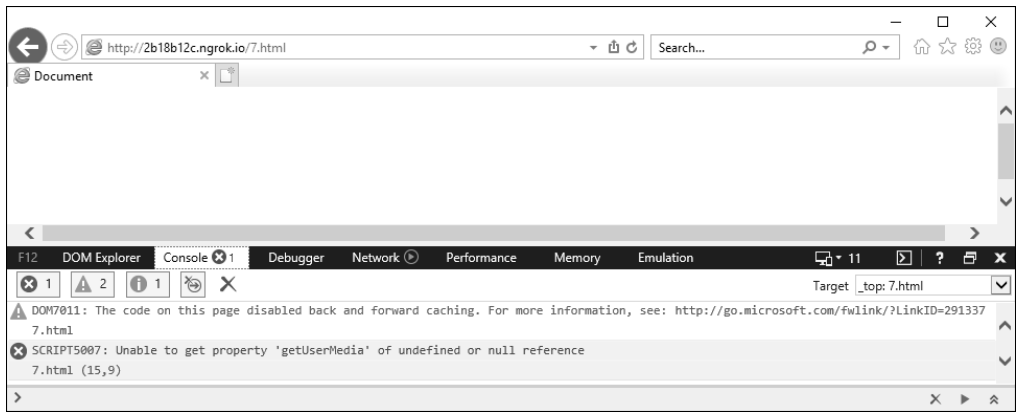


**Abbildung 3.1** Greift man auf nicht vorhandene Schnittstellen zu, können Anwendungen in älteren Browsern brechen.

Glücklicherweise lässt sich Feature Detection mit JavaScript sehr einfach umsetzen, denn letztlich handelt es sich nur um eine einfache Verzweigung. Steht die Schnittstelle oder das Funktionsmerkmal zur Verfügung, wird darauf zugegriffen, andernfalls kann (sofern anwendbar) eine Fallback-Lösung angeboten werden, oder die Funktion wird auf diesem Gerät nicht angeboten – zum Beispiel indem der zugehörige Menüeintrag nicht dargestellt wird.

```
if ('serviceWorker' in navigator) {
  navigator.serviceWorker.register('sw.js')
    .then(/* … */);
}
```

**Listing 3.1** Erst prüfen, dann nutzen: Feature Detection in JavaScript

Darüber hinaus sind die Beispiele aus Kapitel 2 im Internet Explorer nicht lauffähig, weil sie moderne JavaScript-Sprachfeatures einsetzen, die in diesem Browser noch nicht unterstützt werden. Doch auch hier gibt es Abhilfe: *Source-to-Source-Transpiler* wie *Babel* oder TypeScript erlauben Entwicklern, schon heute modernes JavaScript zu schreiben und dieses in für ältere Browser verständliche Fassungen zu übersetzen, sofern das Feature auf diesem Browser auf irgendeinen anderen Weg umgesetzt werden kann. Das gewünschte Zielsprachlevel kann dabei vom Entwickler festgelegt werden. Wenn der Internet Explorer eines Tages vernachlässigt werden kann, wird das





Sprachlevel einfach angeboten, und der Source-to-Source-Transpiler übersetzt schlichtweg weniger Code. Die ursprüngliche Codebasis der Anwendung muss dafür jedoch überhaupt nicht angepasst werden.

Durch die Umsetzung der Eigenschaft Progressive wird insgesamt also sichergestellt, dass die Anwendung mit ihrer Kernfunktionalität auf einer möglichst breiten Masse an Geräten ausgeführt werden kann, wohingegen Anwender mit stärkeren Webbrowsern in den Genuss zusätzlicher Funktionalität kommen. Diese Anwenderbasis dürfte aufgrund der stetigen Verbreitung moderner Browser kontinuierlich wachsen, ohne dass der Entwickler die Codebasis dann noch einmal anpassen muss.



## 3.2 App-ähnlich: Sieht aus wie eine App, fühlt sich an wie eine App

**Kurzdefinition**

Die Eigenschaft *App-like* besagt, dass eine Progressive Web App Aussehen und Verhalten nativer Anwendungen übernehmen soll. Das schließt gestalterische Aspekte wie Animationen, Navigationsstrukturen, aber auch die Nutzung nativer Schnittstellen ein.

Über den Namensbestandteil »Progressive« habe ich bereits gesprochen. Kommen wir zum nächsten wesentlichen Bestandteil: »App«. Progressive Web Apps sollen app-like sein, also so aussehen und sich so anfühlen, wie es Anwender von anderen, nativen Apps gewohnt sind. Diese Eigenschaft berührt verschiedene Ebenen der Softwareentwicklung – von der Anwendungsarchitektur über die Funktionalität bis hin zur Benutzeroberfläche.

Betrachten wir einmal klassische Anwendungen. Sie funktionieren wie folgt: Zunächst müssen die zur Ausführung erforderlichen Anwendungsdateien auf das Gerät übertragen werden, zum Beispiel mithilfe eines Installationspakets oder über einen App Store. Diese Anwendungsdateien enthalten alles, was die Anwendung benötigt, um zu starten: Sichten, Logik und Assets. Nur wenn entfernt gespeicherte Daten bezogen oder manipuliert werden müssen, nimmt die Anwendung Kontakt zu einem Webserver auf, ansonsten funktioniert sie autonom. In Abschnitt 1.3.2, »Single-Page Web Applications«, haben Sie gesehen, mit welchem Konzept diese native Anwendungsarchitektur ins Web kommt: eben in Form von Single-Page Web Applications. Auch hier werden alle Anwendungsdateien zum Start geladen und im Speicher gehalten. Eine Navigation innerhalb der Anwendung löst keine tatsächliche Websitenavigation im Sinne einer serverseitigen Anfrage aus, die mehrere Sekunden dauern kann und bei fehlender Internetverbindung komplett fehlschlagen würde. Stattdessen erfolgt der Sichtwechsel unmittelbar. Zur Implementierung solcher Single-Page Web Applications stehen diverse Frameworks wie Angular oder React zur Verfügung,

die oftmals auch bestimmte Architekturmittel mitbringen, um selbst Anwendungen größeren Umfangs im Web implementieren zu können. Ob Entwickler ein Framework einsetzen und für welches sie sich entscheiden, ist aus Sicht der Progressive Web Apps unerheblich.

Auf Funktionsebene können und sollen Progressive Web Apps von nativen Funktionen Gebrauch machen. Einige davon haben Sie bereits in Kapitel 2, »Mächtiges modernes Web«, gesehen. Moderne Web-Apps erhalten durch mächtige Webschnittstellen unter anderem Zugriff auf die Kamera oder das Mikrofon des Geräts, können hardwarebeschleunigte 2D- und 3D-Grafikinhalte darstellen, auf diverse Eingaben von der Maus über den Finger bis zum Gamepad reagieren, Medieninhalte wiedergeben, den Standort des Benutzers abrufen, Daten lokal speichern, sie können auch offline verwendet werden und den Benutzer mit Pushbenachrichtigungen über externe Ereignisse informieren.

Und schließlich sollen sich Progressive Web Apps auch auf Ebene der Benutzeroberfläche ihren nativen Vorbildern angleichen. So sollen PWAs Navigationsstrukturen, Steuerelemente, Effekte und Animationen einsetzen, die auch bei nativen Anwendungen zum Einsatz kommen. Einige Beispiele zeigt Abbildung 3.2. Eine Progressive Web App ist schlecht umgesetzt, wenn sie einen Fremdkörper auf dem System darstellt. Gut umgesetzt ist sie hingegen dann, wenn der Benutzer vergisst, dass er eigentlich gerade mit einer Website interagiert.

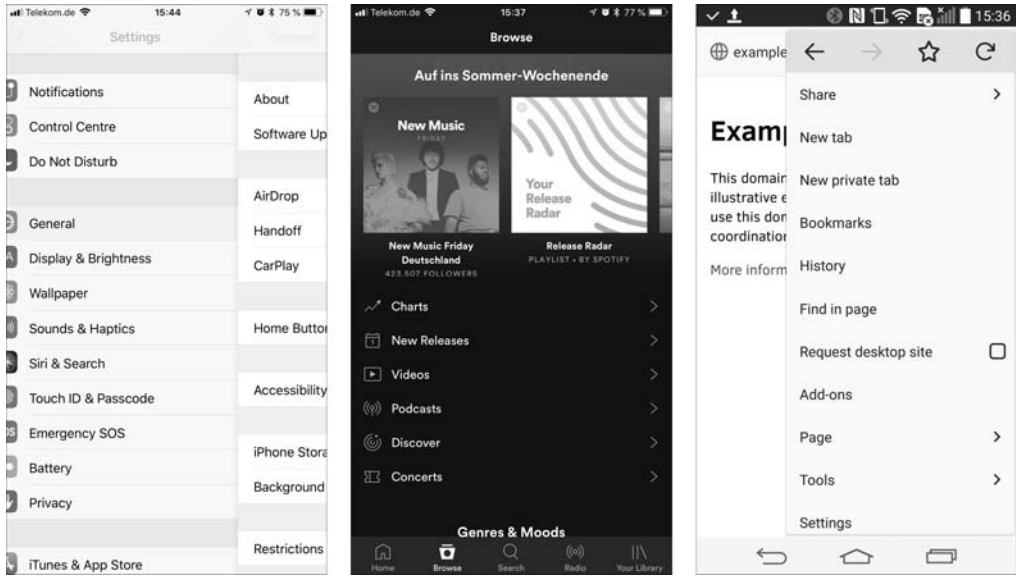


**Abbildung 3.2** Navigationskonzepte mobiler Anwendungen: scrollbare, mehrstufige Menüs in den Einstellungen von iOS (links), die Tab-Bar-Navigation in Spotify sowie aufklappbare Menüs unter Android (rechts)







Aus dem Umfeld der Progressive Web Apps stammt auch das Konzept der *App Shell*: Darunter versteht man den Anwendungsrahmen, also die Teile der Benutzeroberfläche, die häufig oder immer dargestellt werden: Menüs, Titel- oder Statuszeilen. Die Idee ist, dass alle zum Aufbau der App Shell erforderlichen Quelldateien stets offline vorgehalten werden. Das Grundgerüst kann somit beim Start der Anwendung besonders schnell geladen werden, unabhängig von der Qualität und dem Vorhandensein der Internetverbindung. Da die Quelldateien nicht jedes Mal übertragen werden müssen, wird zudem das Datenvolumen des Anwenders geschont. Die App Shell implementiert darüber hinaus die Navigation zwischen verschiedenen Sichten, im Optimalfall unter Verwendung von Animationen, und ist für die Anzeige von Fehlermeldungen zuständig, wenn eine Operation aufgrund zu schwacher oder fehlender Verbindung nicht durchgeführt werden kann. Dies zeigt Abbildung 3.3. Damit grenzt sich die App Shell vom dynamischen Inhalt der Anwendung ab, die diese vom entfernten Webserver in Form von HTML-Fragmenten oder JSON-Datenstrukturen (*JavaScript Object Notation*) bezieht. Es ist dabei selbstverständlich möglich, dass auch dynamischer Inhalt zur Offlineverwendung zwischengespeichert wird oder dass innerhalb des dynamischen Inhalts Komponenten verwendet werden, die von der App Shell bereitgestellt werden. Mit diesem Konzept schließt sich dann wieder der Kreis zu den Single-Page Web Applications, denn die App Shell lässt sich mithilfe gängiger SPA-Frameworks sehr einfach implementieren. In Kapitel 9, »PWA und Angular: Single-Page-Application-Framework einsetzen«, wird dies umfassend beschrieben.

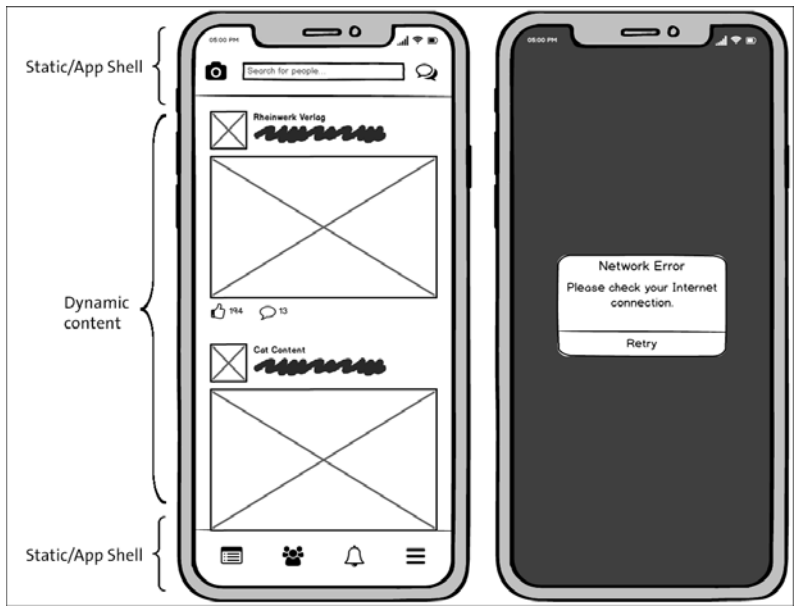


**Abbildung 3.3** Die App Shell stellt Menüleisten zur Verfügung (links) oder zeigt Fehlermeldungen an (rechts).

Die korrekte Umsetzung der Eigenschaft App-like ist ausschlaggebend für den Erfolg einer Progressive Web App. Wer eine Website schnell und mit minimalem Aufwand als PWA verpackt, wird damit vermutlich keinen größeren Erfolg haben. Für eine erfolgreiche Progressive Web App gelten die gleichen Gütekriterien hinsichtlich Bedienung, Funktion, Aussehen und Performance wie für eine in App Stores platzierte Anwendung.

## 3.3 Verbindungsunabhängigkeit: Kein Funkloch hält Sie auf

**Kurzdefinition**

Die Eigenschaft *Connectivity Independent* meint, dass eine Progressive Web App auch bei fehlender oder schwacher Internetverbindung funktionieren muss. Der Anwender soll bei der Durchführung der Kernaufgabe nicht beeinträchtigt werden, auch wenn die Ausführung (zum Beispiel Versand einer E-Mail, Speichern eines Datensatzes) übergangsweise nicht möglich ist.

Die nächste Eigenschaft lautet Connectivity Independent. Jeder, der oft unterwegs ist, kennt das Problem: Die Qualität und die Verfügbarkeit mobiler Internetverbindungen schwanken stark. Fährt die Bahn gerade durch den Tunnel, steht vielleicht gar kein Internet mehr zur Verfügung, an anderen Orten reicht der Empfang allenfalls für langsames *EDGE*. Aus Sicht des Anwenders ist es wünschenswert, dass eine Anwendung solche Situationen meistert, sie also auch offline oder bei schwacher Verbindung »einfach funktioniert«. Das Prinzip der Verbindungsunabhängigkeit kann auf verschiedenen Ebenen umgesetzt werden. In der allereinfachsten Ausprägung würden nur die Quelldateien, die zur Ausführung der Anwendung benötigt werden, offline gehalten. In dieser Form wird die Eigenschaft Connectivity Independent auch als Muss-Kriterium für jede Progressive Web App angesehen.

Das Mittel zur Umsetzung stellt im PWA-Umfeld der Service Worker dar, in dessen Cache Antworten für bestimmte Abfragen zwischengespeichert werden können. Auf die Quelldateien der Anwendung kann dann auch unabhängig von der Internetverbindung zugegriffen werden, somit startet sie ebenfalls offline. Ein Service Worker wird durch eine JavaScript-Datei implementiert, die typischerweise neben der Clientanwendung auf dem Webserver liegt und durch sie im Browser registriert wird. Die Registrierung ist für einen bestimmten *Scope* gültig (Kombination aus Protokoll, Hostname, Port und Pfad). Der Service Worker läuft außerhalb der Clientanwendung, aber noch innerhalb des Webbrowsers. Er ist somit der zentrale Adapter, der entscheiden kann, ob er eine Anfrage über das Netz oder seinen Cache bedienen möchte (siehe Abbildung 3.4). Da er auf diese Art stellvertretend für den Webserver antworten kann, ist der Service Worker besonders mächtig. Das gilt im Übrigen auch für







fremde Quellen, etwa wenn Sie JavaScript-Bibliotheken von Content Delivery Networks nutzen oder externe Schriftarten einbinden. Auf den Service Worker und seine Schnittstellen gehe ich im Detail in Kapitel 5, »Service Worker: Einer muss ja arbeiten«, ein.

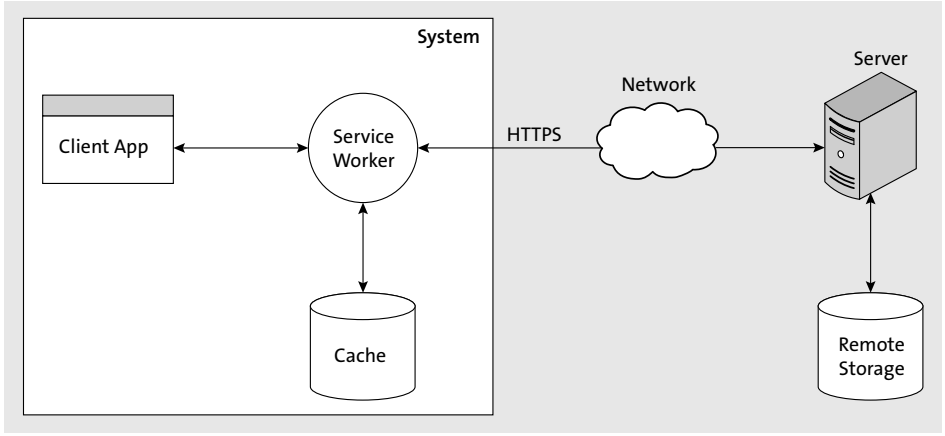


**Abbildung 3.4** Umsetzung der Offlinefähigkeit mithilfe des Service Workers

Im Spektrum zwischen lokalem Zwischenspeicher und Netz ergeben sich verschiedene *Caching-Strategien*: Das Szenario *Cache First* beschreibt, dass zuerst die zwischengespeicherte Version ausgeliefert wird. Damit wird die Website deutlich schneller geladen, der Benutzer sieht jedoch möglicherweise zunächst eine ältere Fassung der Anwendung oder Website. Umgekehrt gibt es das Szenario *Network First*, bei dem zunächst versucht würde, die Anfrage über das Netz zu bedienen. Dann wäre die Anwendung oder Website natürlich aktuell. Läuft die Anfrage in eine Zeitüberschreitung, würde stattdessen die Fassung aus dem Zwischenspeicher geladen werden. Somit ergibt sich aber eine Verzögerung, bis das Timeout greift. Neben diesen Szenarien gibt es noch viele weitere Optionen, Netz und Zwischenspeicher zu kombinieren. Schließlich könnte der Service Worker eine Antwort auch komplett selbst zusammensetzen oder anstelle der eigentlichen Anwendung eine statische Seite ausliefern, etwa wenn der Anwender offline ist, aber für die sinnvolle Verwendung der App auf jeden Fall eine Internetverbindung benötigt wird.

Im Sinne Ihrer Anwender sollten Sie jedoch bei der Offlinefähigkeit Ihrer Quelldateien nicht stehen bleiben, sondern auch Anwenderdaten offline halten. Ein gutes Beispiel für eine Anwendung, die dieses erweiterte Prinzip der Verbindungsunabhängigkeit umsetzt, ist die Notizanwendung *Microsoft OneNote*. Dort können Sie jederzeit Notizen erfassen, unabhängig davon, ob Sie gerade eine stabile Verbindung zum Internet haben oder nicht. Die Notizen werden hier zunächst in einem lokalen Datenspeicher hinterlegt, auf den jederzeit zugegriffen werden kann. Wenn eine Internetverbindung besteht und sie ausreichend stark ist, werden die Notizen im Hin-

tergrund an den entfernten Datenspeicher übertragen, wie Abbildung 3.5 zeigt. Der Benutzer wird zu keinem Zeitpunkt von weiteren Eingaben abgehalten.

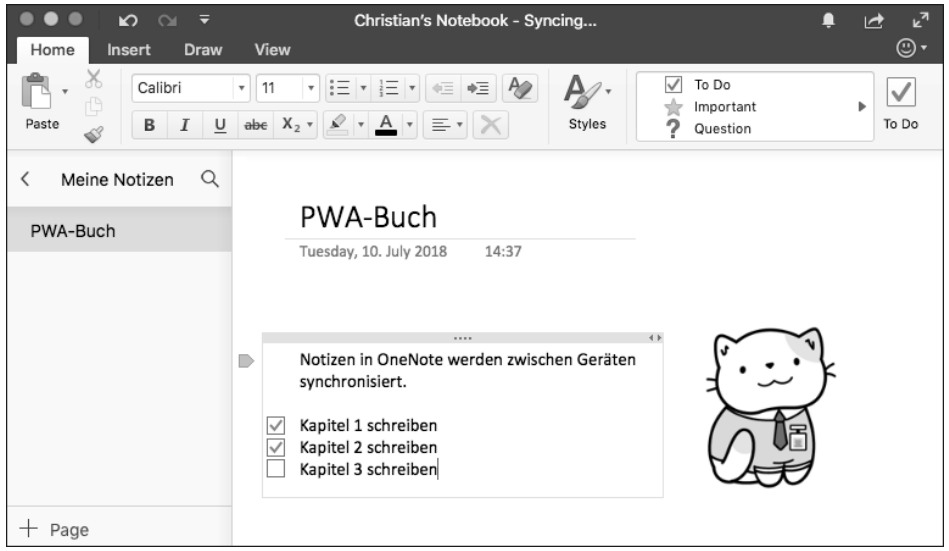


**Abbildung 3.5** Notizen in Microsoft OneNote werden zuerst lokal erfasst und dann synchronisiert, wie in der Titelzeile zu sehen ist.

Dieses Vorgehen ist zwar sehr benutzerfreundlich, doch zieht es nicht zu unterschätzende Aufwände für den Entwickler nach sich: Denn durch die Einführung des lokalen Datenspeichers muss der Entwickler nun eine Synchronisierungslogik implementieren, um Änderungen an den entfernten Server zu übermitteln und Änderungen von dort zu empfangen. Mehr noch: Verwendet der Benutzer die Anwendung auf mehr als einem Gerät oder greifen mehrere Anwender auf dieselben Ressourcen zu, kann es zu Konflikten, also widersprüchlichen Datenbeständen, kommen. So braucht das Gesamtsystem also auch eine Strategie zur Konfliktauflösung, die je nach fachlicher Domäne und Anwendungsfall ausgewählt werden muss: Während etwa bei einer Adressänderung in einer Kundendatenbank die Strategie *Last writer wins* ausreichend sein mag, wäre sie bei den Kontobuchungen einer Bank problematisch. Im Web können Anwenderdaten mithilfe verschiedener Technologien offline persistiert werden, etwa mithilfe der clientseitigen Datenbanktechnologie *IndexedDB*.

Als Entwickler sollten Sie sich unbedingt während der Entwurfsphase Gedanken über den Grad der Offlinefähigkeit machen, den Ihre Anwendung später implementieren soll. Der nachträgliche Einbau von Offlinefähigkeit ist aufgrund der grundlegend abweichenden Architektur mit einem immensen Aufwand verbunden. Aus diesem





Grund hat sich auch das Paradigma Offline First etabliert: Offlinefähigkeit muss von vornherein bedacht, konzipiert und implementiert werden.



## 3.4 Immer schön frisch bleiben: der Service-Worker-Updateprozess

**Kurzdefinition**

Die Eigenschaft *Fresh* bedeutet, dass eine neue Websiteversion trotz lokaler Offlinekopie schnellstmöglich bereitgestellt werden soll. Gleichzeitig wird sichergestellt, dass es nicht zu Inkonsistenzen zwischen zwei Websiteversionen kommen kann.

Die nächste Eigenschaft nennt sich Fresh und bezieht sich unmittelbar auf die zuvor genannte Eigenschaft Connectivity Independent. Wie oben beschrieben, wird der Service Worker im Webbrowser registriert und steht dort auch ohne Internetverbindung zur Verfügung. Der Service Worker speichert dabei immer eine bestimmte Websiteversion zwischen. Somit stellt sich die Frage, was passiert, wenn eine neue Fassung des Service-Worker-Skripts auf dem Webserver bereitgestellt wird. Der neue Service Worker könnte etwa Änderungen in der Art und Weise vornehmen, wie Quelldateien zwischengespeichert werden. Im Optimalfall soll der Anwender natürlich direkt auf die neue Fassung des Service Workers zugreifen. Es könnten jedoch mehrere Registerkarten der Progressive Web App geöffnet sein. Es wäre problematisch, wenn verschiedene Registerkarten auf unterschiedliche Versionen des Service Workers zurückgreifen würden. Daher wird ein Updateprozess benötigt, um dem Anwender schnellstmöglich Zugriff auf den neuen Service Worker zu gewähren, ohne Probleme und Inkonsistenzen hervorzurufen. Den Updateprozess beleuchte ich in Kapitel 5, »Service Worker: Einer muss ja arbeiten« im Detail, daher verzichte ich an dieser Stelle auf eine weitere Beschreibung.

Weil diese Eigenschaft eher im Aufgabenbereich des Webbrowsers liegt, wird sie bei Mozilla oder Microsoft nicht erwähnt. Ich deute die Eigenschaft *Fresh* gern auch in einer alternativen Form: Eine Progressive Web App sollte nicht nur proaktiv, zum Beispiel in definierten Zeitabständen, Änderungen abrufen (Polling), sondern sich von außen per *Push* über extern aufgetretene Ereignisse informieren lassen. Nach diesem Prinzip arbeiten unter anderem Apps sozialer Netze wie Facebook, Messenger wie WhatsApp, Nachrichten-Apps und zahlreiche E-Mail-Clients. Dies ist ein weiteres wichtiges Merkmal moderner Anwendungen, das die Benutzererfahrung maßgeblich verbessern kann. Mit dieser Deutung ist die Eigenschaft *Re-engageable* verwandt, die ich weiter unten beleuchte.

## 3.5 Sicher: Mit großer Macht kommt große Verantwortung

**Kurzdefinition**

Die Eigenschaft *Safe* besagt, dass die Anwendung über HTTPS übertragen werden muss.

Die nächste Eigenschaft nennt sich Safe. Progressive Web Apps sollen die Sicherheit von Browsern, Mobilgeräten und Desktopcomputern natürlich nicht untergraben. Diese Eigenschaft bezieht sich primär auf eine gesicherte Verbindung zwischen Webbrowser und Client, es lohnt sich jedoch auch ein Blick auf die übrigen Sicherheitsmaßnahmen im Browser. Denn ein bestimmtes Maß an Sicherheit ist bei Websites wie Progressive Web Apps grundsätzlich schon gegeben.

### 3.5.1 Sicherheitsmaßnahmen im Webbrowser

Da eine PWA letztlich »nur« eine Website mit bestimmten Zusatzfeatures darstellt, unterliegt sie den gleichen Sicherheitsbestimmungen wie jede andere Website auch. Wie jeder Browser-Tab läuft auch eine Progressive Web App in einer Sandbox. Auf andere geöffnete Registerkarten kann eine PWA nicht wahlfrei zugreifen, ebenso ist kein Aufruf beliebiger nativer Schnittstellen möglich, da diese nicht in JavaScript bereitgestellt werden. Umgekehrt eröffnen sich in Progressive Web Apps die gleichen Angriffsvektoren wie in übrigen Webanwendungen, allen voran Attacken nach dem Schema des *Cross-Site Scripting* (XSS) oder der *Cross-Site Request Forgery* (CSRF).

XSS wird auch als »Buffer Overflow des Webs« bezeichnet, da es zu den Hauptangriffsarten zählt. In einer Webanwendung wird eine Sicherheitslücke ausgenutzt, um fremde Inhalte in einem vertrauenswürdigen Kontext zu laden. Nimmt eine Webanwendung in einem Formular den Namen des Anwenders entgegen und stellt die Angabe auf der Bestätigungsseite ohne Eingabebehandlung (*Sanitizing*) dar, kann ein Angreifer HTML-Code injizieren: Gibt der Anwender `<script>alert('hacked')</script>` als Name im Formular ein, würde sich auf der Bestätigungsseite eine Hinweisbox öffnen. Problematisch ist, dass dieses JavaScript im JavaScript-Kontext der Website läuft und somit vollständigen Zugriff auf Daten und Objekte der Website erhält. Somit können die Benutzersession oder lokal gespeicherte Daten ausgelesen und an einen entfernten Server übertragen werden. Auf Anwendungsebene helfen Frameworks wie Angular oder React dem Entwickler dabei, diesen Angriffsvektoren entgegenzuwirken.

Im Webbrowser findet darüber hinaus das Konzept der *Same-Origin Policy* Anwendung: Skripte, Stylesheets und weitere Elemente aus fremden Quellen (*Origin*, Kombination aus Protokoll, Hostname und Port), die ebenfalls auf Objekte und Daten der eigenen Website zugreifen können, dürfen ohne zusätzliche Maßnahmen wie *Cross-*







*Origin Resource Sharing* (CORS) zunächst einmal nicht geladen werden. Wenn es dennoch erforderlich ist, dass externe Skriptdateien der Stylesheets geladen werden, kann das Sicherheitsfeature *Subresource Integrity* verwendet werden. Dazu wird ein `integrity`-Attribut auf dem zugehörigen `script`- oder `link`-Element der Ressource platziert. Dieses enthält einen Hash-Wert, gegen den die übertragene Ressource geprüft wird. Dies zeigt Listing 3.2. Versucht eine Quelle, zu einem späteren Zeitpunkt eine andere Ressource zu liefern (zum Beispiel weil die Domain eines ehemaligen *Content Delivery Network* von einem Hacker übernommen wurde), stellt der Webbrowser die geladene Ressource nicht bereit. Der Angriff wird somit abgewehrt. Subresource Integrity wird ab Microsoft Edge 17, Mozilla Firefox 43, Google Chrome 45 und Apple Safari 11 (nur auf dem Desktop) unterstützt.

```
<script src="https://cdnjs.cloudflare.com/ajax/libs/jquery/3.3.1/
jquery.min.js" integrity="sha384-tsQFqpEReu7ZLhBV2VZlAu7zcOV+rXbYlF2cqB8txI/
8aZajjp4Bqd+V6D5IgvKT"></script>
```

**Listing 3.2** Laden eines externen Skripts mit Subresource Integrity

Als zusätzliche Sicherheitsschicht gibt es die *Content Security Policy* (CSP), die die erlaubten Origins für definierte Ressourcentypen über die Same-Origin Policy hinaus einschränken kann. Damit kann die Angriffsfläche für XSS-Attacken weiter reduziert werden. Es ist ebenfalls möglich, bestimmte Origins auf eine Whitelist zu setzen. CSP ist als HTTP-Kopfzeile realisiert, die vom Webserver mitgeschickt wird. Sollte der Entwickler keine Kontrolle über den Webserver haben, besteht die Möglichkeit, innerhalb des HTML-Dokuments ein Metatag mit dem Attribut `http-equiv` zu setzen, um den HTTP-Header zu simulieren. Listing 3.3 zeigt eine beispielhafte Content Security Policy: Hierbei sind Skripte (`script-src`) nur zulässig, wenn sie von der Quelle `https://cdnjs.cloudflare.com` stammen, Bilder (`img-src`) dürfen aus jeder beliebigen Quelle geladen werden (`*`), alle übrigen Ressourcen (`default-src`) müssen von derselben Origin geladen werden (`self`). CSP steht in den vier großen Browsern Microsoft Edge, Mozilla Firefox, Google Chrome und Apple Safari in aktuellen Versionen zur Verfügung.

```
Content-Security-Policy: default-src 'self'; img-src *; script-src https://
cdnjs.cloudflare.com
```

**Listing 3.3** Content-Security-Policy-Kopfzeile

Bei CSRF macht sich ein Angreifer hingegen den Umstand zunutze, dass ein Anwender bei einem Onlinedienst angemeldet ist und der Webbrowser eine gültige Sitzung (zum Beispiel als Cookie) speichert. Der Angreifer versucht beispielsweise, dem Anwender eine bestimmte URL unterzuschieben (etwa als verborgene Kurz-URL), an die der Anwender eine HTTP-Anfrage stellt. Da Cookies vom Browser ohne zusätzliche

Konfiguration mit jeder Anfrage an die Website mitgeschickt werden, wird die fremde Anfrage im Kontext des Anwenders ausgeführt. Schutzmechanismen sind vom Webserver vorgegebene Einmalpasswörter (Anti-Forgery-Tokens), die bei einer Anfrage mitgeschickt werden müssen, oder die Verwendung alternativer Autorisierungsmethoden (zum Beispiel über die HTTP-Kopfzeile `Authorization`).

Darüber hinaus steht eine Vielzahl weiterer HTTP-Kopfzeilen zur Verfügung, um den erlaubten Funktionsumfang der Website weiter einzuschränken. So gibt es zum Beispiel den Header `Feature-Policy`, der die auf der Website erlaubten Webschnittstellen (wie den Zugriff auf die Kamera oder die Geolocation API) einschränkt oder nur für Skripte aus bestimmten Origins zulässt. Der Header `Referrer-Policy` kann genutzt werden, um zu bestimmen, mit welchen Origins die Herkunftsangabe (Header `Referer`, sic!) geteilt wird. Websites erfahren standardmäßig, von welcher fremden Website sie aufgerufen wurden, es sei denn, es findet ein Aufruf von einem gesicherten Kontext (HTTPS) auf einen ungesicherten Kontext (HTTP) statt. Diese Kopfzeile hilft, die Übertragung der Herkunftsangabe genauer zu spezifizieren. Weiterhin existiert der Header `Frame-Options`, der das Einbetten der Website in einen *Inline-Frame* (IFrame) verbietet. Somit können Clickjacking-Attacken vermieden werden, bei denen ein Angreifer die Website in einem IFrame darstellt, bestimmte Steuerelemente aber mit durchsichtigen Klickbereichen überlagert, die dann eine ganz andere Aktion auslösen können als die vom Benutzer vorgesehene (zum Beispiel Abgreifen der Login-Daten über ein gefälschtes Formular). Die korrekte Umsetzung all dieser Header kann der Entwickler unter *https://securityheaders.com/* prüfen.

### 3.5.2 Sichere Verbindungen mit HTTPS

Kern der PWA-Eigenschaft Safe ist jedoch die sichere Auslieferung der PWA-Quelldateien sowie die Übertragung der Anwenderdaten: Bei Progressive Web Apps kommen viele moderne Webschnittstellen zum Einsatz. Am Beispiel des Service Workers haben Sie gesehen, dass dieser eine besonders große Macht besitzt, da er stellvertretend für Webserver antworten kann. Außerdem kann er losgelöst vom Lebenszyklus der Website ausgeführt werden, etwa um Push-Notifications entgegenzunehmen, wenn die PWA geschlossen ist. Aufgrund dieser mächtigen Features ist die Installation eines Service Workers nur dann erlaubt, wenn die Website über eine gesicherte Verbindung übertragen wurde. Im Web setzt man dies mithilfe des *Hypertext Transfer Protocol Secure* (HTTPS) um. HTTPS basiert seinerseits auf der *Transport Layer Security* (TLS), frühere Ausgaben dieses Protokolls sind auch unter dem Namen *Secure Sockets Layer* (SSL) bekannt. Den Einsatz des Protokolls erkennt der Anwender am *https*-Bestandteil der URL und am Schlosssymbol in der Adresszeile.

Mithilfe von HTTPS werden verschiedene Schutzziele erreicht: Zum einen muss sich der Server mit einem gültigen digitalen Zertifikat für den Domainnamen, unter dem







er Inhalte ausliefert, authentifizieren. Der Webbrowser erkennt nur Zertifikate an, die von einer vertrauenswürdigen Zertifizierungsstelle ausgestellt wurden. Welche das sind, legt entweder das Betriebssystem (zum Beispiel bei Google Chrome und Microsoft Edge) oder der Browser selbst fest (Mozilla Firefox). Je nach Zertifikatsklasse prüft die Zertifizierungsstelle nicht nur die Zuordnung zwischen dem Antragsteller und seiner Domain (*Domänenvalidierung*), sondern auch die Existenz und Identität des Betreibers und nimmt dessen Kennzeichnung in das Zertifikat auf. Dies ist etwa bei der besonders strengen *Extended Validation* (EV) der Fall. Bei EV-Zertifikaten wird in der Adresszeile zusätzlich der Name des Betreibers angezeigt. Diese Zertifikate kommen oft bei Onlineshops oder Banken zum Einsatz, wie in Abbildung 3.6 zu sehen. Davon abgesehen werden sämtliche Daten, sowohl Quelldateien als auch Anwenderdaten, bei HTTPS *verschlüsselt* übertragen. Somit sind sie vor Abhören und Manipulation geschützt; Vertraulichkeit und Integrität werden gewahrt.

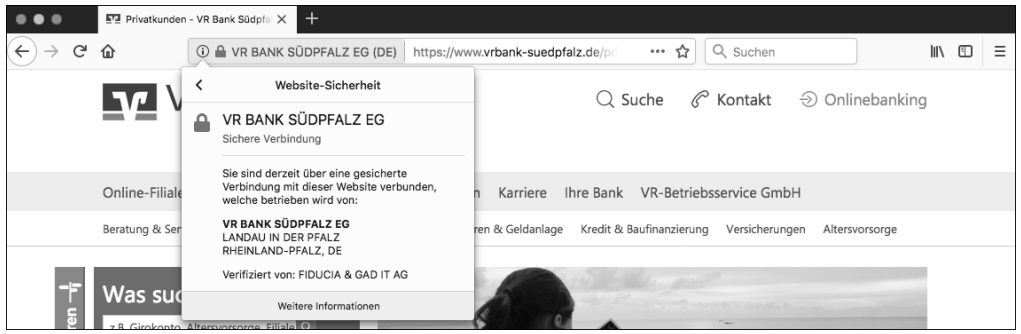


**Abbildung 3.6** Bei Extended-Validation-Zertifikaten wird auch die Identität des Anbieters geprüft.

Da Service Worker nur bei HTTPS-Verbindungen registriert werden können, ist der Einsatz dieses Protokolls für alle Progressive Web Apps ein absolutes Muss-Kriterium. Die einzige Ausnahme besteht zur Entwicklungszeit, denn auf der eigenen Maschine (unter *localhost*) bereitgestellte Websites dürfen Service Worker auch ohne gesicherte Verbindung registrieren. Doch nicht nur für PWAs ist der Einsatz von HTTPS interessant: Manche Webschnittstellen liefern Zugriff auf sensible Daten, wie die in Kapitel 2, »Mächtiges modernes Web«, vorgestellten APIs Geolocation sowie Media Capture and Streams. Einige Webbrowser gewähren Websites nur noch dann Zugriff auf diese Schnittstellen, wenn sie über HTTPS ausgeliefert wurden. Bei manchen neueren Webschnittstellen wie der Web Share API oder Übertragungstechniken wie HTTP/2 wird HTTPS sogar ganz allgemein zur Voraussetzung erklärt. Aus diesem Grund, und weil der Anwender durch diese Maßnahmen geschützt wird, ist der Einsatz von HTTPS mittlerweile für sämtliche Websites empfehlenswert. Um HTTPS einsetzen zu können, benötigt der Entwickler aber ein *TLS-Zertifikat*. Spätestens bei diesem Begriff zucken Systemadministratoren regelmäßig zusammen, denn TLS-

Zertifikate haben eine Gültigkeitsdauer und laufen zu einem bestimmten Zeitpunkt aus. Jemand muss sich also darum kümmern, das Zertifikat rechtzeitig zu erneuern. Je nach Zertifikatstyp und Gültigkeitsdauer können TLS-Zertifikate auch sehr teuer werden.

Doch auch diese Problematik ist mittlerweile gelöst, zumindest wenn Sie keinen Wert auf eine bestimmte Zertifikatsklasse legen. So gibt es beispielsweise die Initiative *Let's Encrypt* der Linux Foundation. Die Domain der zu schützenden Website respektive Web-API muss bei diesem Ansatz öffentlich erreichbar sein. Mithilfe eines Plug-ins kann der Server über ein automatisiertes Verfahren ein kostenfreies, domänenvalidiertes Zertifikat für die eigene Domain bei der Zertifizierungsstelle von Let's Encrypt anfordern. Dessen Gültigkeitsdauer fällt mit nur 90 Tagen zwar vergleichsweise kurz aus, es wird jedoch vor Ablauf der Gültigkeit durch das Plug-in vollautomatisch erneuert. Weder Entwickler noch Administrator müssen künftig mehr eingreifen, da auch die Verwaltung des privaten Schlüssels sowie die Verschlüsselung der Inhalte vom Plug-in arrangiert werden. Für viele Webserver wie *Apache* oder die *Microsoft Internet Information Services*, aber auch für Webserverdistributionen wie *Plesk* stehen kostenfreie Plug-ins zur Verfügung. Der Dienst *Cloudflare* bietet ebenfalls kostenfrei aufschaltbare Zertifikate an, viele Hoster und Clouddienste stellen oftmals eine Domain zur Verfügung, auf der ein Wildcard-Zertifikat geschaltet ist. Die eigene Anwendung wird dann unter einer Subdomain ausgeführt, die durch das Wildcard-Zertifikat des Anbieters mit abgesichert wird. Somit gibt es auch für private Websites oder andere Projekte ohne größeres Budget oder Personalausstattung keine Ausrede mehr, kein HTTPS einzusetzen – zumal Google seit dem im Juli 2018 veröffentlichen Chrome 68 alle via HTTP übertragenen Seiten in der Adresszeile als »nicht sicher« markiert.

**Umleiten nicht vergessen**

Vergessen Sie in diesem Zuge bitte nicht, die Anwender, die Ihre Website oder Ihre Progressive Web App zunächst über HTTP besuchen, auf die HTTPS-Variante weiterzuleiten. Auch dafür gibt es Plug-ins oder Middleware, damit Sie das ohne größeren Aufwand erledigen können. Um zu erzwingen, dass der Webbrowser mit einer Website nur noch über HTTPS spricht, gibt es die *HTTP Strict Transport Security* (HSTS). Dabei leitet der Webserver zunächst von HTTP auf HTTPS um. Auf HTTPS-Antworten platziert er den Header `Strict-Transport-Security`:

```
Strict-Transport-Security: max-age=31536000; includeSubDomains
```

Dadurch wird der Webbrowser angewiesen, zu einer bestimmten Domain für eine definierte Zeit (im obigen Beispiel für die Dauer eines Jahres) nur noch verschlüsselte Verbindungen aufzunehmen. Optional kann mithilfe der Direktive `includeSubDomains` angegeben werden, dass der Schutz auch auf alle Subdomains ausgeweitet werden soll. Der Browser nimmt dann das Umschreiben des Protokolls von HTTP auf







HTTPS clientseitig vor, noch bevor die Anfrage den Browser verlässt. Das hilft, Man-in-the-Middle-Angriffe einzudämmen. Die Gefährdung bleibt allerdings beim ersten Request über HTTP bestehen. Daher wird HSTS auch als TOFU-Technologie bezeichnet (*Trust On First Use*). Um den ersten Request ebenfalls abzusichern, steht Websitebetreibern die HSTS-Preload-Liste zur Verfügung. Diese Liste befindet sich im Lieferumfang aller großen Browser (Google Chrome, Mozilla Firefox, Apple Safari, Microsoft Edge und Internet Explorer). Websitebetreiber können sich unter *https://hstspreload.org* für die HSTS-Preload-Liste eintragen. Voraussetzungen sind ein gültiges Zertifikat, das Umleiten von HTTP nach HTTPS, eine Mindestgültigkeit des HSTS-Headers von einem Jahr, der Einsatz der oben genannten Direktive `includeSubDomains` und somit das Absichern aller Subdomains. Werden HTTPS-Requests ebenfalls umgeleitet, müssen auch auf diesen Weiterleitungen HSTS-Header gesetzt werden. Abschließend muss der HSTS-Header um die Direktive `preload` ergänzt werden. Die korrekte Umsetzung kann etwa mit dem *SSL Server Test* von Qualys geprüft werden: *www.ssllabs.com/ssltest*. So besteht der Schutz ab dem ersten Aufruf.

Insgesamt wird durch den Einsatz von HTTPS gewährleistet, dass Ihre PWA sicher ausgeliefert wird und Anwenderdaten geschützt bis zur Gegenseite übertragen werden können. Dazu stehen viele kostenfreie und komfortable Lösungen zur Verfügung, sodass dieser Schritt ohne größeren Aufwand erfolgen kann. Bei Progressive Web Apps wird darüber hinaus die URL zum wichtigen Sicherheitsmerkmal: In manchen App Stores tummeln sich Apps von Drittanbietern, die Aussehen und Namen bekannter Anwendungen imitieren. Der Anwender kann diese Täuschung oft nur schwer erkennen. Bei einer PWA genügt die Prüfung der Adresszeile.

## 3.6 Einer für alle: Responsive Webdesign

**Kurzdefinition**

Die Eigenschaft *Responsive* gibt an, dass sich eine Progressive Web App an die verfügbaren Bildschirmabmessungen anpassen soll, sodass sie vom Smartphone bis zum Desktopcomputer sinnvoll verwendet werden kann.

Das erste iPhone wurde bei seiner Veröffentlichung als Kombination aus iPod, Telefon und »Internetgerät« vorgestellt. Es wird oft darüber gestritten, wie viel Innovation wirklich in Apples erstem Smartphone steckte. Eines ist aber unbestreitbar: Der Safari war der erste mobile Webbrowser, mit dem sinnvoll auf die Desktopversionen von Websites zugegriffen werden konnte. Zuvor implementierten Websitebetreiber oftmals ein zweites, mobiles Internetangebot mit stark reduziertem Funktionsumfang. Doch dieses redundante Angebot muss unterhalten und weiterentwickelt werden, außerdem stellt sich spätestens ab dem breiten Aufkommen von Tablets die Fra-

ge, an welcher Grenze man zwischen Vollversion und mobiler Ausgabe schneidet. Heute verfügt fast jede ernst zu nehmende Plattform über einen halbwegs potenten Webbrowser, vom kleinen Smartphone über Tablets, Notebooks und Desktopcomputer bis hin zum 60-Zoll-Fernseher. Die Grenzen zwischen den Plattformen sind also fließend, gleichzeitig möchten Entwickler wenn möglich mit einer einzigen Codebasis die komplette Bandbreite an Plattformen und Geräten abdecken. Damit das gelingt, braucht die Anwendung ein Gewand, das sich an die Abmessungen all dieser Bildschirme anpassen kann. Die Lösung dieses Problems nennt sich *Responsive Webdesign*.

Das Prinzip ist einer der Wegbereiter für Progressive Web Apps. So lautet die nächste PWA-Eigenschaft Responsive. Mit einer einzigen Codebasis soll eine PWA alle Geräte vom Smartphone bis zum Fernseher abdecken können. Das Websitelayout wird dazu meist an der Breite der verfügbaren Bildschirmfläche ausgerichtet. Es werden bestimmte Triggerpunkte definiert, an denen das Layout umbricht. So wird bis zu einer bestimmten Bildschirmbreite die Fassung für Smartphones angezeigt. Menüs werden hier standardmäßig eingeklappt, um mehr Platz für den Inhalt zu lassen. Mit zunehmender Größe können dann vollständige Menüs und Auswahllisten sowie zusätzliche Inhalte dargestellt werden – schrittweise über das Tablet bis hin zur Darstellung für Desktopgeräte, auf denen der meiste Platz zur Verfügung steht (siehe Abbildung 3.7). Responsive Webdesign setzt somit das Prinzip des Progressive Enhancement auf der Ebene des Layouts um.

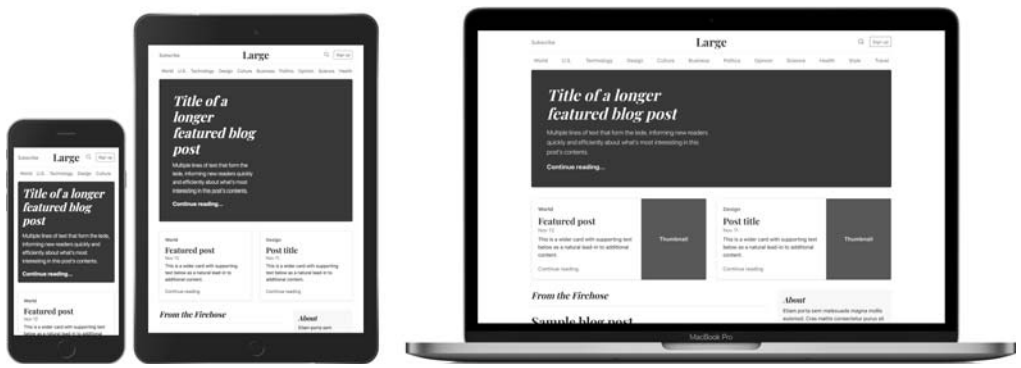


**Abbildung 3.7** Beispiel für ein Responsive Webdesign, das sich verschiedenen Bildschirmabmessungen anpasst

Allerdings unterscheiden sich nicht nur die Bildschirmabmessungen, sondern auch die Eingabemethoden auf den unterschiedlichen Geräten: Desktopcomputer und Notebooks haben etwa eine Tastatur, sodass dem Anwender Tastenkürzel angeboten werden können, die auf mobilen Geräten aber nicht funktionieren. Ebenso steht bei Desktopcomputern praktisch immer eine Maus oder ein Trackpad zur Verfügung, mit denen pixelgenaue Eingaben vorgenommen werden können. Steuerelemente







können hier deutlich kleiner gestaltet und mit geringeren Abständen voneinander platziert werden, als dies bei Mobilgeräten mit Stift- oder der deutlich ungenaueren Fingereingabe der Fall ist. Das alles heißt aber nicht, dass zwangsläufig jeder Use Case einer Anwendung auf allen Geräten bereitgestellt werden muss. Anwendungsfälle, die die Präzision einer Maus voraussetzen, können so lediglich auf Geräten mit Mauseingabe aktiviert werden. Zu guter Letzt gibt es noch *Convertibles*, bei denen die Eingabemethoden zur Laufzeit sogar wechseln können. Das Ziel der Eigenschaft Responsive ist, dass jedem Anwender die beste Erfahrung auf seinem Gerät geboten wird.

```
<meta name="viewport" content="width=device-width, initial-scale=1">
```

**Listing 3.4** Definition des Viewport-Metatags

Im Kopf des HTML-Dokuments muss zur Unterstützung responsiver Websites das Metatag `viewport` integriert werden. Andernfalls rendern mobile Webbrowser Seiteninhalte in einem virtuellen Fenster mit für Desktopcomputer üblichen Abmessungen, das dann in den Bildschirm des mobilen Geräts eingepasst wird. Um stattdessen die Breite des Gerätebildschirms anzunehmen, wird im Inhalt dieses Metatags die Eigenschaft `width=device-width` angegeben. Außerdem wird die Eigenschaft `initial-scale=1` gesetzt. Diese gibt an, dass die initiale Zoomstufe bei 100 % liegen soll. Diese Angabe behebt zudem Anzeigeprobleme in iOS, wenn das Gerät gedreht wird und sich die Bildschirmausrichtung ändert. Über das `viewport`-Metatag können noch weitere Eigenschaften (`user-scalable`, `min-scale` und `max-scale`) definiert werden, die dazu genutzt werden konnten, die Pinch-to-Zoom-Funktion von Safari zu deaktivieren. Somit erscheinen Websites und Webanwendungen zwar nativer, doch können Anwender keine Websites mit kleinen Schriftarten bei immer höher auflösenden Bildschirmen mehr vergrößern. Seit Safari auf iOS 10 werden diese drei Eigenschaften daher ignoriert.

Da es Responsive Webdesign schon einige Tage gibt, existiert eine Vielzahl fortgeschrittener, quelloffener und kostenfreier Frameworks wie *Bootstrap*, *Foundation* oder *Material Design Lite*. Diese implementieren das Responsive Webdesign und bringen Stile für häufig verwendete Steuerelemente mit. Die technische Basis stellen *CSS3 Media Queries*, das *CSS Flexible Box Layout Module* (*Flexbox*) sowie das *CSS Grid Layout Module* dar. Welche Techniken genau eingesetzt werden, hängt vom jeweiligen Framework ab. Außerdem ist es mithilfe der zuvor genannten Techniken mit vertretbarem Aufwand und gestalterischem Geschick auch möglich, eigene *UI-Kits* zu implementieren, die komplett eigene Gestaltungsrichtlinien umsetzen und auf die Anwendungsfälle der Anwendung besser zugeschnitten werden können.

Gegen die Eigenschaft Responsive gibt es jedoch auch Vorbehalte. Ein häufig anzutreffender Einwand gegen dieses Konzept ist, dass Cross-Platform-Apps ja gar nicht wie native Anwendungen aussehen würden. Das stimmt jedoch nur teilweise, da

durch die Verwendung von Systemschriften und -farben durchaus der Eindruck erzeugt werden kann, dass der Benutzer mit nativen Steuerelementen interagiert. So arbeitet beispielsweise das Cross-Platform-Framework *Ionic*, und auch der Webbrowser macht nichts anderes: Unter Windows ist keine einzige auf einer Website dargestellte Schaltfläche ein echter, nativer *Win32*-Button, sondern nur entsprechend gestaltetes HTML. Anwendungen wie Spotify, Google Chrome oder Telegram geben sich hingegen nicht einmal die Mühe, sich an die Gestaltungsrichtlinien des Betriebssystems anzupassen. Stattdessen kommt ein plattformübergreifend wiedererkennbares Design zum Einsatz. So sieht Spotify auf jedem Gerät aus wie Spotify – mit dunkler Hintergrund- und grüner Akzentfarbe. Die Wiedergabesteuerung ist überall prominent platziert, und die Anwendung verhält sich ähnlich, egal ob im Browser, als Desktop-App oder auf dem Smartphone. Wer Spotify also plattformübergreifend nutzt, muss sich nicht auf dauerhaft wechselnde visuelle Welten einlassen, sondern nimmt den Dienst von Plattform zu Plattform mit. So überrascht es nicht, dass auch der Spotify-Client auf Webtechnologien basiert.

Um sicherzustellen, dass die Eigenschaft *Responsive* korrekt umgesetzt wird, hilft das Paradigma *Mobile First*. Hier werden Anwendungsfälle und Screens erst für das Gerät mit dem kleinsten Bildschirm und den ungenauesten Eingabemethoden konzipiert, das Smartphone. Mit zunehmender Bildschirmgröße und genaueren Eingabegeräten werden diese Anwendungsfälle dann für Tablets und schließlich Desktopcomputer erweitert. Somit kann die Anwendung auf einem Mobilgerät sinnvoll bedient werden, ohne dass Desktopbenutzer eingeschränkt werden. Denn Anwendungen, die wie zu groß geratene Handy-Apps aussehen, möchte auf dem Desktop natürlich auch niemand bedienen.

Im Beispiel in Kapitel 9, »PWA und Angular: Single-Page-Application-Framework einsetzen«, zeige ich Ihnen mit *Angular Material* eine Vorlage, die ein Responsive Webdesign umsetzt. Weitere Vorlagen stelle ich in Kapitel 10, »App-like aussehen«, vor.

## 3.7 Auffindbarkeit: Websites und Apps unterscheiden

**Kurzdefinition**

Die Eigenschaft *Discoverable* besagt, dass eine Progressive Web App mithilfe des Web App Manifests von regulären Websites unterschieden werden muss.

Die nächste Eigenschaft hört auf den Namen Discoverable. Da Progressive Web Apps im Kern nur Websites mit gewissen Extras sind, muss es eine Möglichkeit geben, eine PWA von einer ganz gewöhnlichen Website zu unterscheiden. Dazu wurde das Web App Manifest aus der Taufe gehoben, eine separate Spezifikation. Es handelt sich dabei um eine Manifestdatei im JSON-Format. Diese Datei definiert an zentraler Stel-







le Metadaten zur vorliegenden Webanwendung, unter anderem Titel, Anzeigeart, Beschreibungstexte und Symbole:

```
{
   "name": "Meine PWA-Demo",
   "short_name": "PWA-Demo",
   "start_url": ".",
   "display": "standalone",
   "theme_color": "#1976d2",
   "description": "Ich zeige, was Progressive Web Apps können.",
   "icons": [{
       "src": "assets/icons/icon-192x192.png",
       "sizes": "192x192",
       "type": "image/png"
   }]
}
```

**Listing 3.5** Ein beispielhaftes Web App Manifest

Damit ersetzt das Web App Manifest zugleich frühere Ansätze, bei denen diese Eigenschaften mithilfe von Metatags direkt in der ausführenden HTML-Datei gesetzt wurden. Das kam etwa bei den in Kapitel 1, »Im Web, als App: Geschichte und Einstieg«, erwähnten Web Clips zum Einsatz, und vielleicht erinnert sich der eine oder andere Leser auch noch an die »Pinned Websites« des Internet Explorer 9 oder die Windows 8 Live Tiles für Webseiten, die ebenfalls über solche Metatags gesteuert wurden. Auf die Manifestdatei wird aus der HTML-Datei der PWA heraus verwiesen, wie Listing 3.6 zeigt.

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

**Listing 3.6** Verweis auf die Manifestdatei

Die darin enthaltenen Angaben könnten Suchmaschinen nutzen, um eine eigene Suchkategorie für Apps anzubieten, oder eben Webbrowser, um die Website mit besonderen Funktionen auszustatten, wie es im PWA-Umfeld durch die Möglichkeit zur Installation auf dem Homebildschirm oder der Programmliste der Fall ist. Von der Möglichkeit, Progressive Web Apps mithilfe des Web App Manifests zu erkennen, macht schon heute Microsoft Gebrauch: Der Crawler der hauseigenen Suchmaschine Bing durchsucht das Web nach Progressive Web Apps. Findet der Bot eine PWA, wendet er eine automatisierte Prüfung darauf an. Sollte diese bestimmte Gütekriterien erfüllen, übernimmt Microsoft die im Web App Manifest angegebenen Daten, erzeugt ein Windows-Anwendungsbündel und stellt die Anwendung automatisch in den Microsoft Store zum Download ein.

Weil ich mich mit dem Web App Manifest sowie der automatischen Bereitstellung von PWA im Microsoft Store im nächsten Kapitel ausführlich befasse, kürze ich die weitere Beschreibung der Eigenschaft Discoverable an dieser Stelle ab. Da der Webbrowser zwingend wissen muss, dass es sich bei der vorliegenden Website um eine Progressive Web App handelt, ist der Einsatz des Web App Manifests für jede PWA ein absolutes Muss-Kriterium.

## 3.8 Installierbarkeit: So kommt Ihre PWA auf den Homebildschirm

**Kurzdefinition**

Die Eigenschaft *Installable* gibt an, dass Progressive Web Apps auf den Homebildschirm beziehungsweise in die Programmliste des Betriebssystems installiert werden können.

Kommen wir nun zur Eigenschaft Installable, die sehr eng mit dem eben beschriebenen Web App Manifest zusammenhängt. Denn damit eine Progressive Web App auf dem Gerät installiert werden kann, benötigt sie zwingend ein Web App Manifest. Von jeher sind wir es gewöhnt, dass auf unseren Geräten installierte Anwendungen auf dem Desktop in der Programmliste beziehungsweise auf Mobilgeräten auf dem Homebildschirm zu finden sind. Wenn Progressive Web Apps es mit ihren nativen Gegenstücken auf Augenhöhe aufnehmen sollen, müssen sie auch an diesen Orten zu finden sein. Über die Verknüpfung hinaus soll sich die Anwendung ebenso verhalten wie eine native Anwendung – auf dem Desktop also in einem eigenen Fenster mit nativer Fensterdekoration laufen und auf Mobilgeräten vollflächig. In der folgenden Beschreibung spreche ich nun nur noch vom Homebildschirm. Desktopcomputer sind mit der Programmliste aber in gleicher Weise gemeint, »vollflächig« bedeutet dort dann das eigene Fenster mit nativer Fensterdekoration, und App-Switcher bezeichnet die Taskleiste.

»Installieren« meint bei Progressive Web Apps allerdings lediglich, eine Verknüpfung der Anwendung auf dem Homebildschirm anzulegen. Funktionen wie die Offlinefähigkeit oder der Empfang von Pushnachrichten sind auch ohne Icon auf dem Homebildschirm schon lauffähig – sie werden allein durch den Service Worker bereitgestellt, der im Stillen und ohne Nutzerabfrage registriert wird.

**Aufgabe: Installierte Service Worker zählen**

Wie viele Service Worker sind in Ihrem Browser schon registriert? Rufen Sie unter Google Chrome *chrome://serviceworker-internals* oder unter Mozilla Firefox *about:serviceworkers* auf. Bei Chrome wird die Anzahl in der Zeile REGISTRATIONS IN in Klammern angezeigt. Bei mir sind es 131.







Bei Progressive Web Apps ist auch kein Installationspaket vorgesehen, da die im Browser ausgeführte Anwendung exakt deckungsgleich mit der späteren auf dem Homebildschirm hinterlegten Anwendung ist. Deswegen wurde ein anderes Konzept gesucht, um Anwendungen aus dem Browser heraus installieren zu können. Es besteht immer die Möglichkeit, Progressive Web Apps manuell auf dem Gerät zu installieren. Dies entspricht dem ursprünglichen Gedanken von Apple, den Sie in Abschnitt 1.1.2, »Der Browser als Anwendungsplattform«, schon gesehen haben. Dazu öffnet der Benutzer das passende Browsermenü und wählt den Eintrag ADD TO HOMESCREEN.

Der ursprüngliche Gedanke von Alex Russel war, dass der Browser die Nutzung der Website überwacht und bei häufiger Verwendung einer Website den Anwender per Banner zur Installation vorschlägt – analog zu den Smart-App-Bannern für zugehörige App-Store-Apps aus Abschnitt 1.1.4, »Das Verhältnis von Web und Apps«. Genau so war es in Google Chrome noch bis Version 67 implementiert.

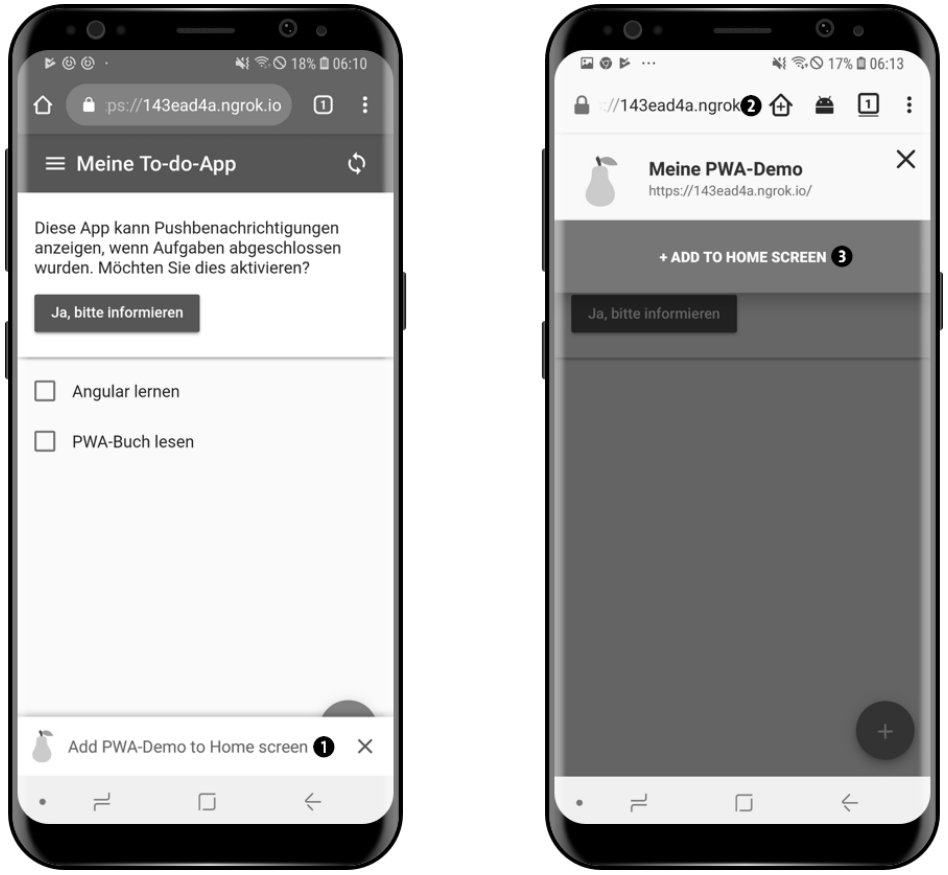


**Abbildung 3.8** App-Install-Banner und Add-to-Home-Screen-Button

Dieses Banner wird jedoch schrittweise verschwinden: Seit Google Chrome 68 ist stattdessen eine »Mini-Infobar« zu sehen, eine deutlich kleinere Leiste am unteren Bildschirmrand. Diese zeigt Abbildung 3.8 auf der linken Seite ❶. Diese soll schließlich durch eine Schaltfläche in der Adresszeile ersetzt werden, wie es sie schon jetzt bei Mozilla Firefox gibt (❷ in Abbildung 3.8 auf der rechten Seite). Klickt man darauf, öffnet sich ein Dialog, über den der Anwender das Installieren der Anwendung bestätigen kann ❸. Die Informationen, die hier angezeigt werden, stammen alle aus dem Web App Manifest: Dargestellt werden der ausführliche Titel und das Symbol. Außerdem können PWAs unter bestimmten Bedingungen auch selbst zur Installation auffordern, mehr dazu finden Sie im nächsten Kapitel.

Auf dem Homebildschirm ist die Kurzbezeichnung der Anwendung zu sehen, wie sie im Web App Manifest definiert wurde, darüber hinaus ihr Symbol (siehe Abbildung 3.9 auf der linken Seite).

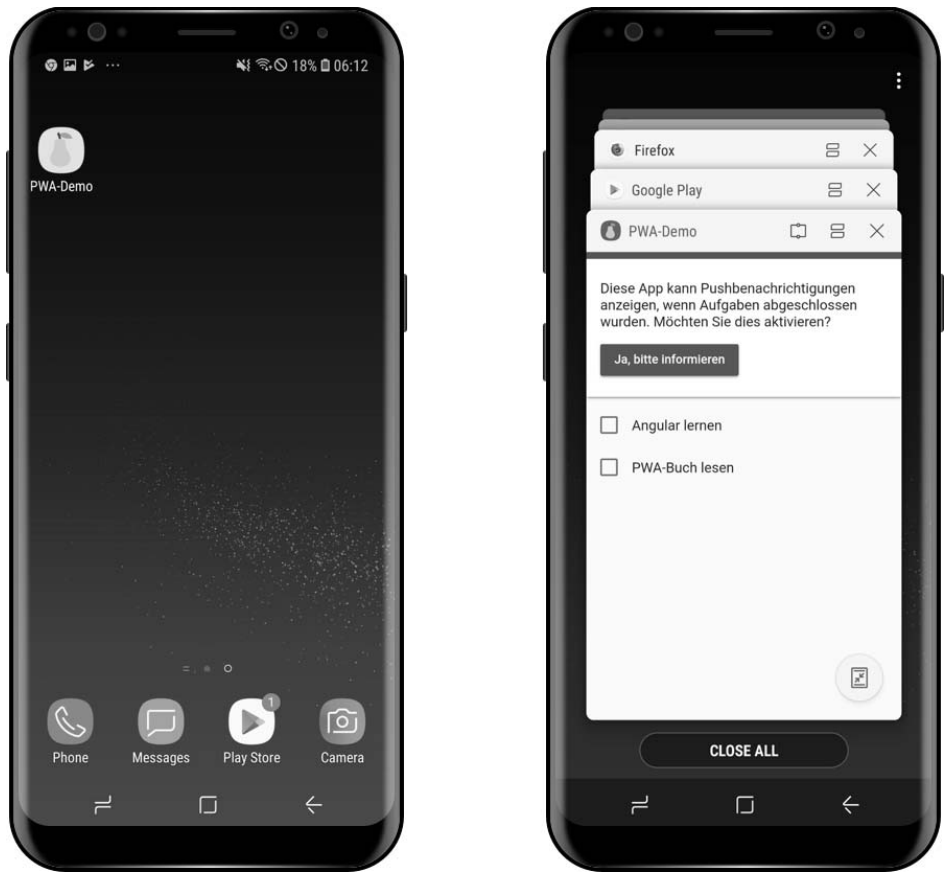


**Abbildung 3.9** Progressive Web App auf dem Homebildschirm und im App-Switcher







Öffnet man die App über diese Verknüpfung auf dem Homebildschirm, startet sie nun vollflächig und nicht mehr im Browser. Dieser Anzeigemodus (`standalone`) wurde ebenfalls im Web App Manifest festgelegt. Somit erhält die Anwendung nun auch einen Auftritt im App-Switcher des Betriebssystems, wie in Abbildung 3.9 auf der rechten Seite zu sehen. Auch hier stammen die Informationen aus der Manifestdatei.

Zum Zeitpunkt des Verfassens dieses Buchs besteht die Möglichkeit zum Hinzufügen der Progressive Web App auf den Homebildschirm auf den Mobilausgaben von Google Chrome, Mozilla Firefox und Apple Safari. Ab Google Chrome 70 funktioniert das auch unter Microsoft Windows, die Unterstützung für macOS und Linux muss voraussichtlich noch bis Version 72 per Flag aktiviert werden. Mehr hierzu und zum Web App Manifest folgt im nächsten Kapitel.

## 3.9 Nutzer binden: Anwender mit Pushbenachrichtigungen zurückholen

**Kurzdefinition**

Die Eigenschaft *Re-engageable* besagt, dass eine Progressive Web App ihre Nutzer mithilfe von Pushbenachrichtigungen binden kann.

Eine weitere zentrale Eigenschaft ist Re-engageable: Hier stellt sich die Frage, wie Anwender zur Wiederverwendung einer App bewogen werden können. Das geschieht klassischerweise mit Push-Notifications, die den Anwender über extern eingetretene Ereignisse (zum Beispiel Aktualisierungen, neue Inhalte, Breaking News, Nachrichten eines Kontakts) sowie abgeschlossene Aktivitäten informieren oder an bekannte Ereignisse erinnern (zum Beispiel Ablauf eines Timers, Auslauf einer Onlineauktion, Kalendererinnerung). Das Konzept kennen Sie sicherlich von Free-to-play-Spielen (»Nur heute! 300 grüne Diamanten für 5,99 Euro«) oder sozialen Netzen (»Peter hat Ihren Beitrag kommentiert«). Zum Empfang und zur Darstellung einer solchen Benachrichtigung muss die Anwendung nicht unbedingt geöffnet sein. Beispiele dafür zeigt Abbildung 3.10 auf der linken Seite. Die *Notifications API* des Service Workers erlaubt Websites, sich auf die exakt gleiche Art in das native Benachrichtigungszentrum zu integrieren. Abbildung 3.10 zeigt auf der rechten Seite Benachrichtigungen der PWA Twitter Lite. Den einzigen Rückschluss darauf, dass es sich um eine PWA handelt, lässt die dort dargestellte URL *mobile.twitter.com* zu. Mithilfe der *Push API* kann der Service Worker externe Pushereignisse entgegennehmen – auch dann, wenn die Anwendung nicht geöffnet ist – und mithilfe der Notifications API als native Push-Notification anzeigen.

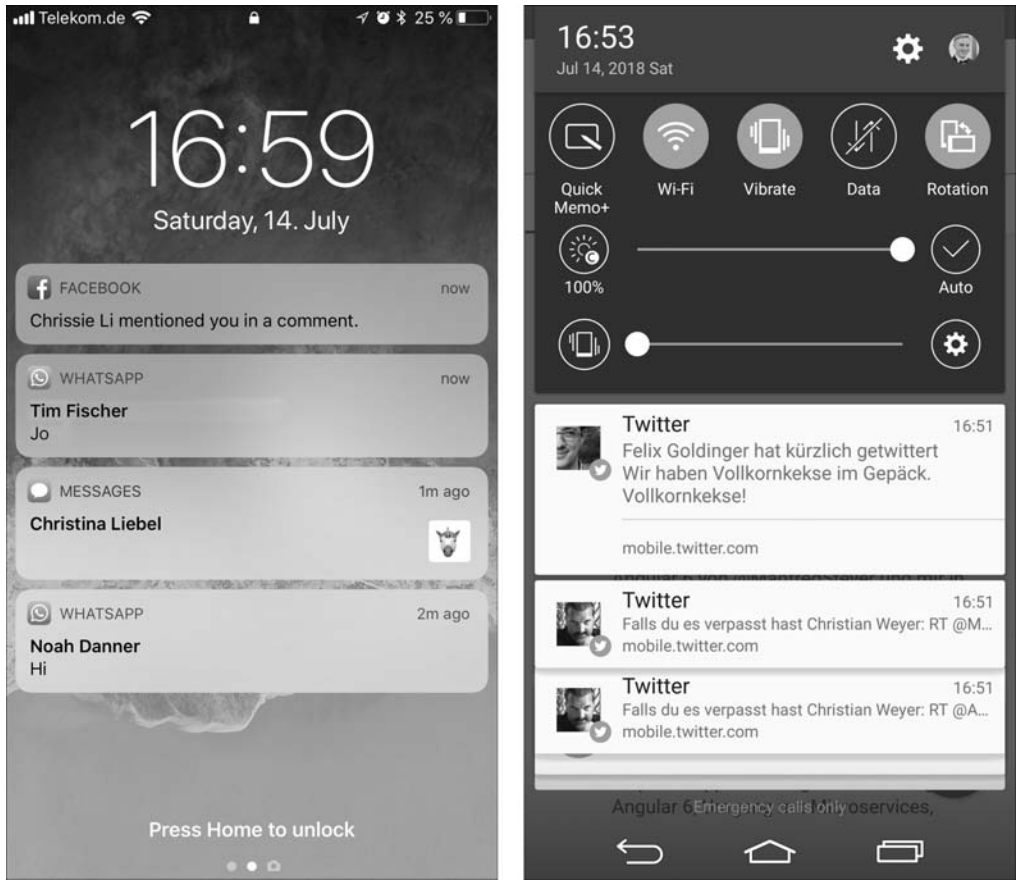


**Abbildung 3.10** Pushbenachrichtigungen sind ein zentraler Bestandteil der mobilen Anwendungserfahrung (links), auch Progressive Web Apps können sich nahtlos einfügen (rechts).

Reguläre Websites nutzen diese Funktionalität ebenfalls zunehmend, darunter Facebook und verschiedenste Nachrichtenportale. Um eine missbräuchliche Verwendung der Schnittstelle zu vermeiden, fragt der Webbrowser den Anwender zunächst, ob dieser den Empfang von Pushbenachrichtigungen zulassen möchte. Bei vielen Websites erscheint diese Abfrage jedoch direkt, nachdem die Website geladen wurde, und verdeckt somit den Websiteinhalt. Facebook blockiert während der Abfrage sogar den Zugriff auf den Inhalt, wie Abbildung 3.11 zeigt. Stattdessen sollte die Erlaubnis für Pushbenachrichtigungen erst eingeholt werden, wenn der Anwender bereits mit der Website interagiert hat und somit Interesse an den Inhalten zeigt. Auf diese Weise steigen die Chancen, dass der Anwender dem Empfang von Pushbenachrichtigungen auch zustimmt.







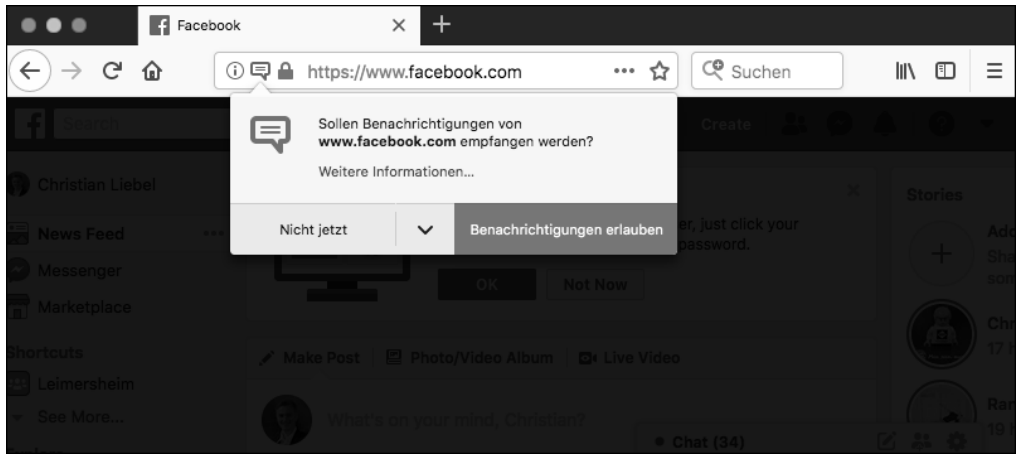


**Abbildung 3.11** Facebook fragt die Berechtigung zum Empfang von Pushbenachrichtigungen ab.

Technische Basis ist in allen Fällen der Service Worker in Kombination mit Push API und Notifications API. Damit der Webbrowser auch dann noch Pushbenachrichtigungen empfangen und darstellen kann, wenn die Anwendung nicht geöffnet ist und alle Browserfenster geschlossen sind, führt dieser einen dauerhaft aktiven Hintergrunddienst aus. Bei Pushbenachrichtigungen sind drei Akteure involviert:

1. Zum einen ist das die Anwendung, die sich beim Webbrowser für Push-Notifications registriert. Zuvor muss der Anwender zustimmen, dann erhält die Anwendung Zugriff auf die erforderlichen Informationen, um Pushbenachrichtigungen auf *dieses eine Gerät* schicken zu können. Diese Informationen schickt die Anwendung typischerweise an ihr Backend. Die Anwendung erhält Kenntnis darüber, ob der Anwender die Anforderung angenommen oder abgelehnt hat.
2. Der Ansprechpartner für den Webbrowser beziehungsweise das Gerät ist bei der Push API jedoch nicht der eigene Server, sondern der *Push-Dienst* der jeweiligen Plattform: Bei Google Chrome kommt im Hintergrund *Firebase Cloud Messaging* zum Einsatz, bei Microsoft Edge die *Windows Push Notification Services* und bei Mozilla Firefox (auf dem Desktop) der Dienst *Mozilla Web Push*. Nur über diese Dienste können Pushbenachrichtigungen auf das Gerät gesendet werden.
3. Möchte das Backend der Anwendung Push-Notifications an die Anwender verschicken, muss es also mit den unterschiedlichen Push-Diensten der Anbieter sprechen können. Um die Kommunikation zu vereinfachen, wurde das Protokoll *Web Push* eingeführt, das eine einheitliche Schnittstelle zum Versand von Pushbenachrichtigungen darstellt. Die Nachrichten werden zwischen Server und Client verschlüsselt, sodass die Inhalte auf dem Übertragungsweg weder mitgelesen noch

manipuliert werden können. Der Push-Dienst versucht dann, die Nachricht an die Endgeräte zuzustellen. Vom Push-Dienst erhält das Backend die Information darüber, ob die Nachricht erfolgreich an die ausgewählten Endgeräte geschickt werden konnte.

Gegenwärtig funktioniert dieses Konzept auf den Desktopplattformen mit Google Chrome, Microsoft Edge und Mozilla Firefox sowie auf Android mit Chrome und Firefox. Unter Safari werden bei Drucklegung noch keine Pushbenachrichtigungen aus dem Service Worker heraus unterstützt. Zumindest auf dem Desktop bietet Safari mit *Safari Push Notifications* allerdings eine alternative Schnittstelle an. Auch damit ist es möglich, Benachrichtigungen anzuzeigen, wenn Safari nicht geöffnet ist. Zur Verwendung ist jedoch eine Mitgliedschaft beim Apple-Developer-Programm erforderlich. Außerdem kann die Schnittstelle nicht vom Web-Push-Protokoll angesprochen werden. Der Server müsste daher eine separate Implementierung für diese Anwender anbieten. Es gibt jedoch auch Drittanbieter wie *OneSignal*, *Pushpad* oder *PushAlert*, die wiederum eine abstrahierte Schnittstelle über verschiedene Pushschnittstellen wie Safari Push Notifications und Web Push bereitstellen. Diese Dienstleister bieten darüber hinaus die Möglichkeit, Empfänger in Segmente einzuteilen, bieten Berichte zu Reichweite und Konversionsrate der Benachrichtigungen an und erlauben A/B-Tests. Allerdings empfiehlt es sich, zuvor die Geschäftsmodelle der Anbieter genauer zu studieren. So ist OneSignal für den Entwickler zwar absolut kostenfrei, doch sammelt dieses Unternehmen Daten und verkauft sie an Werbetreibende und Forschungsunternehmen.

Neben Push-Notifications stellen Anwendungen auch oft ein *Badge* auf dem Homebildschirm dar. Dieses kann zum Beispiel anzeigen, dass es ungelesene Statusmeldungen gibt oder wie viele ungelesene Nachrichten vorliegen. Hierzu gibt es im Service-Worker-Umfeld derzeit noch keine Lösung, mit der *Badging API* (siehe Abschnitt 4.5, »Zukunftsmusik: Badging API«) ist es jedoch auf der Agenda des Google-Chrome-Teams.

## 3.10 Verlinkbar: auf Anwendungen und Zustände verweisen

**Kurzdefinition**

Die Eigenschaft *Linkable* gibt an, dass auf die Progressive Web App mithilfe einer URL verwiesen werden kann, im Idealfall bis zu einem bestimmten Zielzustand oder einer Zielsicht.







Die zehnte und letzte Eigenschaft nennt sich Linkable. Diese Eigenschaft gibt es praktisch geschenkt dazu. Sie besagt, dass auf eine Anwendung per URL verwiesen werden kann. Es braucht weder App Store noch Setup-Paket, um einem Freund oder Kollegen eine Anwendung zu zeigen. Stattdessen schickt man diesem einfach »einen Link« zur Anwendung, und schon befindet er sich mittendrin.

Darüber hinaus kann mithilfe der URL sogar auf einen bestimmten Zielzustand innerhalb der Anwendung verwiesen werden (Deep Linking), sofern der Entwickler dies in der Anwendung so umgesetzt hat. Auch hier kommen uns wieder die Single-Page Web Applications zu Hilfe, denn diese enthalten oftmals eine Unterstützung für *clientseitiges Routing* innerhalb der Anwendung. Mithilfe dieses Ansatzes werden Anwendungssichten bestimmte URLs zugewiesen.

Die Navigation innerhalb der SPA führt automatisch zum Wechsel der angezeigten URL in der Adresszeile, umgekehrt wird bei Eingabe einer bestimmten URL die Anwendung direkt auf der zugeordneten Sicht geladen. Diese URLs müssen auch nicht statisch sein, sondern können Parameter enthalten. Unterstützt ein Framework clientseitiges Routing nicht direkt, gibt es oftmals Plug-ins dafür. Zur Umsetzung des Ansatzes stehen zwei Strategien zur Verfügung:

### 3.10.1 Hash-basiertes Routing

Zum einen gibt es das sogenannte *Hash-basierte Routing*, bei dem die URLs nach diesem Muster aufgebaut werden:

*https://my-social-network.invalid/#/profile/christian.liebel*

Die anwendungsspezifische Route ist dabei durch das Rautezeichen abgetrennt. Dieses sogenannte Hash-Fragment der URL kann eine clientseitige Anwendung frei manipulieren und hat nur innerhalb des Webbrowsers eine Bedeutung; zum Server wird es nie geschickt. In diesem Beispiel wird auf eine Profilansicht verwiesen, als Parameter wird der Benutzername »christian.liebel« mit übergeben. Diese URL kann nun mit anderen Personen geteilt werden. Öffnen Sie die Adresse in Ihrem Webbrowser, lädt die Anwendung unmittelbar das gewünschte Profil.

Nach diesem Schema arbeitet beispielsweise Gmail, wie Abbildung 3.12 zeigt: Die Route »inbox« verweist auf den Posteingang. So bekommt jeder Ordner eine eigene URL, die etwa als Lesezeichen abgespeichert werden kann. Klickt man darauf, öffnet sich Gmail direkt im gewünschten Ordner.

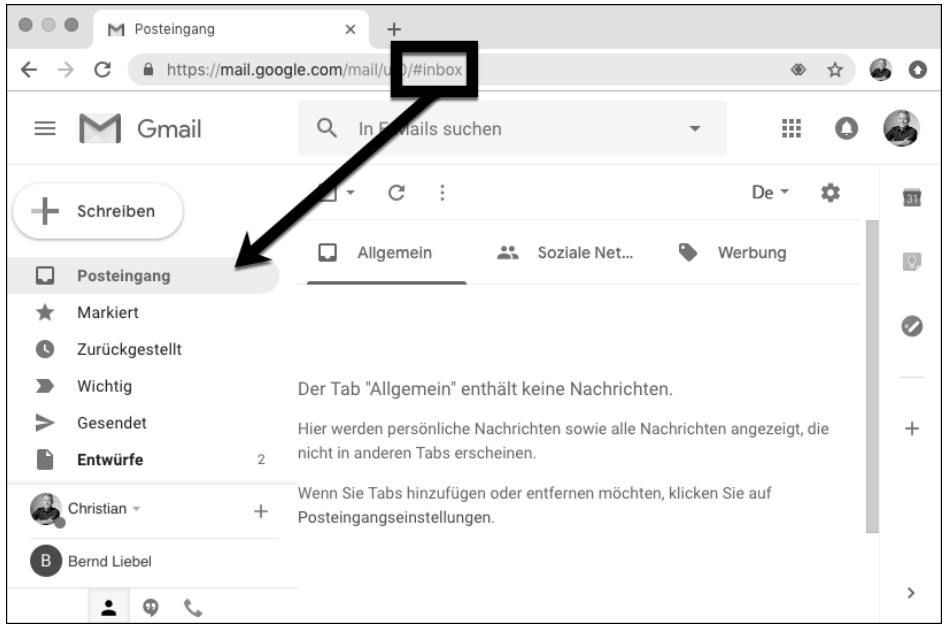


**Abbildung 3.12** Google Mail setzt clientseitiges Routing um.

### 3.10.2 Pfad-basiertes Routing

Dann gibt es das sogenannte *Pfad-basierte Routing*, das auf der *History API* basiert. Über diese Schnittstelle kann eine Webanwendung die in der Adresszeile dargestellte URL und den Verlaufsstapel der Registerkarte (Vorwärts-/Rückwärtsschaltflächen) manipulieren. Die dabei entstehenden URLs lassen keinen wirklichen Rückschluss mehr darauf zu, dass es sich eigentlich um eine clientseitig ausgeführte Webanwendung handelt:

*https://my-social-network.invalid/profile/christian.liebel*

Diese Form des Routings setzt jedoch voraus, dass der Server unter jeder möglichen Route wieder die Single-Page Web Application laden kann. Denn wenn Sie diese URL abrufen, sucht der Server zunächst nach dem Pfad »profile/christian.liebel« und findet dort vermutlich nichts. Dank Methoden zum Umschreiben von URLs (*URL Rewriting*) lässt sich das zwar umsetzen, doch erfordert es etwas mehr Arbeit auf dem Server. Diese URLs wirken nicht nur deutlich schöner, sondern lassen sich in Kombination mit *Server-Side Rendering* auch gut verwenden. Wenn der Entwickler Einfluss auf die Konfiguration der späteren Serverumgebung nehmen kann, empfehle ich den Einsatz des Pfad-basierten Routings.







### 3.10.3 Deep Linking und Link Capturing

Was zunächst banal klingt, erweist sich als sehr mächtig und ist ein einzigartiges Funktionsmerkmal des Webs. Versuchen Sie doch mal, einem Kollegen einen Verweis zu schicken, der sein E-Mail-Programm öffnet und zur dritten Eigenschaftenseite des Einstellungsdialogs navigiert. Das dürfte schwer möglich sein, doch im Web handelt es sich um eine von jeher gegebene Eigenschaft. Auf mobilen und Desktopbetriebssystemen hat sich eine Annäherung an dieses Konzept etabliert, die sogenannten URI Schemes, die Sie bereits in Abschnitt 1.1.4, »Das Verhältnis von Web und Apps«, kennengelernt haben. Hier können sich Anwendungen für bestimmte Pseudoprotokolle oder URLs registrieren und können aufgerufen werden, wenn der Anwender diese abrufen möchte. Außerdem sind auch Deep-Links möglich, also Verweise auf gezielte Zustände oder Sichten innerhalb der Anwendung. So ruft zum Beispiel der URI *ms-settings:privacy-microphone* unter Windows die Einstellungen-App auf, die dann die Datenschutzeinstellungsseite für das Mikrofon darstellt. Unter iOS öffnet hingegen die URL *www.google.com/maps/@49.1231618,8.2960259,13z* die native App von Google Maps auf der angegebenen Koordinate, sofern die App auf dem Gerät installiert ist.

Dieses Konzept könnte wiederum zurück in die Welt der Progressive Web Apps schwappen. Denn hier ist seitens Google für die Zukunft angedacht, dass das Anklicken der URL einer auf dem Homebildschirm oder der Programmliste hinterlegten Progressive Web App nicht mehr den Webbrowser auf dieser Seite öffnet, sondern stattdessen die PWA in ihrem nativen Fenster und auf dem gewünschten Zielzustand (*Link Capturing*).

## 3.11 Zusammenfassung

- Progressive Web Apps sind keine greifbare Technologie, sondern ein Typ Webanwendung, der eine Sammlung bestimmter Eigenschaften umsetzt. Die Eigenschaften Discoverable, Connectivity Independent und Safe sind Muss-Kriterien.
- Dank des Web App Manifests lassen sich Anwendungen auf den Homebildschirm beziehungsweise in die Programmliste des Geräts installieren sowie von normalen Websites unterscheiden.
- Mithilfe des Service Workers läuft die Anwendung auch bei fehlender oder schwacher Internetverbindung.
- Die Anwendung der Paradigmen Mobile First und Offline First stellt sicher, dass Anwendungen auch offline und auf Mobilgeräten durchgängig sinnvoll bedient werden können.