PROGRESSIVE WEB APPS

Offline immer mitgedacht

Mit Service Worker und IndexedDB arbeiten Online-Apps auch ohne Internetverbindung.

Im Auto hört man Spotify, während einer Bahnfahrt schaut man YouTube-Videos und in der Straßenbahn nach Hause werden noch schnell die E-Mails gecheckt und WhatsApp-Nachrichten verschickt. Die heutige Welt ist mobil geworden und mit ihr auch die Geräte und Dienste, welche die Anwender nutzen. Längst schon ist es nicht mehr nötig, sich an den Schreibtisch im Arbeitszimmer zu setzen, um auf das Internet zuzugreifen – das geht heute praktisch von überall. Der Zugriff auf Geschäftsdaten erfordert keine lästige VPN-Einwahl, sondern die Dateien stehen über öffentlich erreichbare und sichere Server-Backends in der Cloud zur Verfügung.

Allerdings gibt es nicht immer und an jedem Ort eine verlässliche Internetverbindung. Wer schon einmal im Zug durch einen Tunnel gefahren ist, weiß das. Zudem gibt es auf dem Land nach wie vor weiße Flecken in der Netzabdeckung, und im Ausland empfiehlt es sich aufgrund hoher Roaming-Gebühren, vielleicht nur auf öffentlich bereitgestellte WiFi-Netze zurückgreifen, deren Qualität allerdings oft schwankt.

Offline-Fähigkeit als Grundpfeiler

Mit dieser Situation müssen natürlich auch Anwendungen Schritt halten. Stellen Sie sich vor, WhatsApp würde sich weigern, eine Nachricht entgegenzunehmen, weil es gerade keine Internetverbindung gibt. Das ist nicht anwenderfreundlich und kann sich eine Anwendung im modernen App-Umfeld nicht mehr leisten. Moderne Apps müssen auch offline gut funktionieren – zumindest im Rahmen der Möglichkeiten, die sich ohne Internetverbindung ergeben.

WhatsApp begegnet diesem Problem zum Beispiel mit einer Warteschlange. Sie nimmt Nachrichten unabhängig von der Verbindungsqualität entgegen und versieht sie zunächst mit einem Uhr-Symbol. Sobald wieder eine stabile Internetverbindung besteht, synchronisiert WhatsApp automatisch die ausstehenden Nachrichten und kennzeichnet sie anschließend mit einem Hakensymbol. Zur Synchronisation muss sich WhatsApp nicht im Vordergrund befinden, der Anwender kann die Nachricht also „absenden“ und dann sein Smartphone wieder in die Hosentasche packen.

In anderen Apps kann der Anwender Inhalte auch proaktiv für die Offline-Nutzung herunterladen: So erlaubt Google Maps das Zwischenspeichern bestimmter Kartenausschnitte und Netflix den Download von Filmen.

Offline von vornherein bedenken

Zum Erstellen offline-fähiger Anwendungen hat sich das Paradigma „Offline First“ etabliert. Es besagt, dass die Offline-Unterstützung von vornherein, schon beim Konzipieren der Anwendung, mitbedacht und auch implementiert werden muss. Eine fehlende oder schwache Internetverbindung soll nicht als Fehlerzustand oder Ausnahme angesehen werden, sondern als Regel. Insbesondere sollen Apps keine Fehlermeldung ausgeben, wenn sie offline sind. Gleichwohl dürfen Apps darauf hinweisen, dass gerade keine Internetverbindung besteht und sie nur ältere Inhalte anzeigen können.

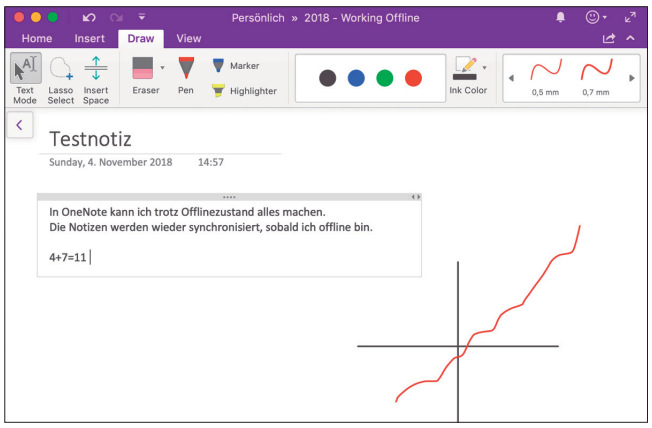
Letztlich muss jede Funktion der Webanwendung mit bedenken, dass der Anwender gerade keine aktive Internetverbindung hat – sei es über eine regelmäßige Update-Anfrage im Hintergrund oder eine Prozedur zur Synchronisierung. Werden diese Punkte beachtet, funktioniert die App immer, online wie offline.

Microsoft OneNote setzt dieses Prinzip vorbildlich um. OneNote ist ein digitales Notizheft, das Einträge mit der Cloud synchronisieren kann, etwa um Daten zu sichern, um über mehrere Geräte hinweg darauf zuzugreifen oder um die Notizen mit Kollegen oder Freunden zu teilen. Die Software erlaubt zu jedem Zeitpunkt das Eingeben von Notizen, unabhängig davon, ob der Anwender gerade über eine aktive Internetverbindung verfügt oder nicht (siehe Bild 1). Der Nutzer wird also nie von dem, was er gerade tut – nämlich Notizen eingeben und bearbeiten –, abgehalten. Dazu legt OneNote die Notizen in einem lokalen Zwischenspeicher ab. Auf diesen kann jederzeit zugegriffen werden, verglichen zum Netz auch besonders schnell. Im Hintergrund gleicht OneNote die Notizen aus dem lokalen Datenspeicher in regelmäßigen Abständen mit dem Zustand in der Cloud ab und hält die Notizen somit aktuell.

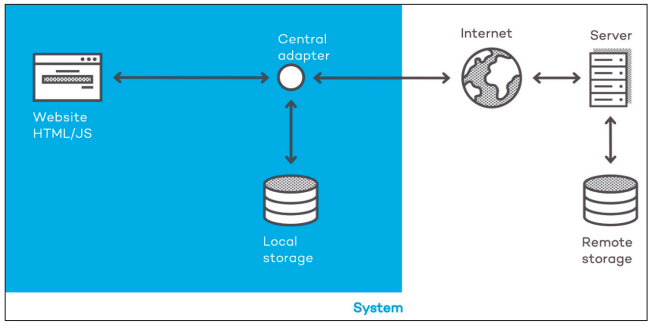
Ein lokaler Datenspeicher ist also Grundvoraussetzung, um Offline-Fähigkeit zu implementieren. Doch wo es verschiedene Datenstände gibt – in der Cloud und auf vielen verschiedenen Endgeräten –, muss eine Synchronisationslogik definiert werden. Bei Twitter ist die Logik recht einfach, da sie neue Tweets einfach entgegennimmt und in den Twitter-Stream einfügt. In anderen Fällen kann es aber auch zu widersprüchlichen Zuständen, also Konflikten kommen; zum Beispiel, wenn zwei Anwender denselben Kundendatensatz bearbeitet haben. Die Auflösung solcher Konflikte ist hochgradig abhängig vom jeweiligen Anwendungsfall.

Das verteilte Versionskontrollsystem Git erlaubt beispielsweise einen komplexen Merge-Prozess auf Zeilenebene. Bei OneNote kommt es nur zu Konflikten, wenn zwei Personen zur selben Zeit denselben Abschnitt auf einer Seite bearbeiten. In diesem Fall legt OneNote eine Kopie der aktuellen Seite als „Konfliktseite“ an und überlässt dem Anwender die Auswahl.

Bei unkritischen Operationen wie einem Nutzer-Profilbild mag es legitim sein, mit dem letzten beim Server einge- ▶



OneNote nimmt zu jedem Zeitpunkt Änderungen entgegen, auch im Offline-Modus (Bild 1)



Die Offline-Fähigkeit wird über einen zentralen Adapter umgesetzt (Bild 2)

henden Stand einfach alle vorherigen Stände zu überschreiben. Das wiederum wäre beim Aktualisieren des Kontostandes in einer Banking-App aber fatal.

Insgesamt muss also auch die Synchronisierungslogik und Strategie zur Konfliktauflösung von vornherein bedacht werden, da sie maximal abhängig ist von der Problemdomäne und dem Anwendungsfall. Die Nutzeroberfläche der Anwendung muss auf Synchronisation und Konfliktzustände sinnvoll hinweisen und beim Auflösen der Konflikte helfen.

Ein zentraler Adapter

Um offline-fähige Anwendungen umzusetzen, ist ein Adapter hilfreich, der an zentraler Stelle prüfen kann, ob der Anwender gerade online ist oder nicht. Sämtliche Anwendungsbestandteile richten ihre Anfragen an diesen zentralen Adapter. Dieser kann je nach Verbindung entweder die Antwort aus dem lokalen Zwischenspeicher zurückgeben oder versuchen, diese über das Netz aufzulösen (siehe Bild 2). Außerdem sind verschiedene Kombinationen aus Zwischenspeicher und Netz möglich, wie das Aktualisieren des Zwischenspeichers bei einer Antwort aus dem Netz oder das regelmäßige Synchronisieren aus dem lokalen Zwischenspeicher über das Netz im Hintergrund.

Der zentrale Adapter darf jedoch von keinem Anwendungsbestandteil umgangen werden, da das Konzept der Offline-Fähigkeit sonst an genau dieser Stelle bricht. So darf es zum Beispiel keine HTTP-Anfragen geben, die direkt gegen das Netz gerichtet werden. Aus diesem Grund erweist sich das nachträgliche Einführen von Offline-Fähigkeit in eine Anwendung als schwierige bis unmögliche Aufgabe, da sämtliche Teile der Anwendung angepasst werden müssen, die Anwenderdaten anfordern oder manipulieren.

Progressive Web Apps und Offline First

Unter der Bezeichnung „Progressive Web Apps“ (PWA) etablierte Google ab 2014 ein neues Anwendungsmodell, das auf modernen Webtechnologien basiert: Hypertext Markup Language (HTML) 5 als Auszeichnungssprache, Cascading Stylesheets (CSS) 3 für das Styling und JavaScript für die Logik. Das Anwendungsmodell wird von den vier großen Plattformbetreibern unterstützt: Google, Microsoft, Mozilla sowie teilweise von Apple. Bei diesem Ansatz implementiert der Entwickler eine Webanwendung, die anschließend überall dort laufen kann, wo ein halbwegs moderner Webbrowser zur Verfügung steht. Insbesondere sollen diese Anwendungen auf einer Ebene mit nativen Anwendungen stehen. Das heißt, sie sollen offline-fähig sein und auch auf mächtige Funktionen wie Push-Benachrichtigungen oder die Synchronisierung von Daten im Hintergrund zurückgreifen können. Die Offline-Fähigkeit ist ein Muss für jede Progressive Web App und wird auch von allen vier Plattformanbietern unterstützt. Extramerkmale wie Push-Benachrichtigungen oder die Synchronisierung von Daten im Hintergrund erlauben derzeit nur Google, Microsoft und Mozilla.

Das Leben einer PWA beginnt im Webbrowser. Der Anwender kann die App auf Wunsch auf Mobilgeräten auf den Home-Bildschirm und auf Desktop-Geräten in die Programmliste installieren. Von dort ausgeführt lässt sich die Anwendung von einer nativen App kaum mehr unterscheiden. Entwickler haben bei PWAs die freie Framework-Wahl: Apps können mit Angular, Vue.js, React oder auch komplett ohne Framework implementiert werden.

Ein zentraler Adapter mit Cache

Technologische Basis für die Progressive Web Apps ist der Service Worker, eine verhältnismäßig neue Webschnittstelle. Diese Schnittstelle ist beim Webkonsortium (dem W3C) als „Redaktionsentwurf (editor's draft) hinterlegt [1]. Damit ist der Service Worker noch keine offizielle Empfehlung des W3C, also noch kein „echter“ Webstandard, doch alle vier großen Browserhersteller haben die Schnittstelle bereits implementiert. Derzeit wird der Service Worker in zwei Dokumenten verwaltet: Das Dokument „Service Workers 1“ strebt einer offiziellen Empfehlung durch das W3C entgegen und wird nur noch behutsam aktualisiert, insbesondere sollen keine neuen Merkmale mehr eingeführt werden. In das Dokument „Service Workers Nightly“ hingegen fließen weiterhin neue Funktionen ein.

Der Service Worker agiert als Proxy, Controller/Interceptor zwischen Webanwendung und Netz und enthält einen eigenen Zwischenspeicher (Cache), in dem er sich Antworten für bestimmte HTTP-Anfragen merken kann. Dieser Cache ist jedoch nicht mit dem altbekannten HTTP-Browsercache zu

verwechseln. Der Service-Worker-Cache folgt insbesondere keinen Caching-Headern und wird auch nicht vom Webbrowser verwaltet, sondern wird allein durch die Anwendung gesteuert.

Auf den Service-Worker-Cache haben der Service Worker, aber auch die Webanwendung selbst Zugriff. Caches werden je Origin isoliert, also nach Protokoll, Hostnamen und Portnummer (zum Beispiel https://example.com). Auf die Caches fremder Origins kann eine Webanwendung nicht zugreifen. Service Worker und Cache werden ab Google Chrome 40 (ab Android 4.1), Mozilla Firefox 44, Microsoft Edge 17 sowie Apple Safari 11.1 (ab iOS 11.3) unterstützt.

Der Service Worker wird zur Laufzeit durch die Webanwendung für einen bestimmten „Scope" registriert. Der Begriff bezeichnet die Kombination aus Origin und Pfad (zum Beispiel https://example.com/myApp/), sodass unter einer Origin auch mehrere voneinander getrennte PWAs betrieben werden können. In dem folgenden Codebeispiel ist die Registrierung eines Service-Worker-Skripts zu sehen. Wird bei der Registrierung kein Scope angegeben, entspricht dieser dem Pfad des Service-Worker-Skripts:

```
try {
  const reg = await navigator.serviceWorker.register(
    './sw.js');
  // Service Worker registration successful
} catch (err) {
  // Service Worker registration failed
}
```

Im Rahmen der Registrierung wird zunächst das referenzierte Service-Worker-Skript vom Webbrowser heruntergeladen, geparst und dann ausgeführt. Service Worker verfügen über verschiedene Lebenszyklus-Ereignisse. Das erste nennt sich *install*: Beim Einrichten des Service Worker lassen sich bestimmte Operationen angeben, die auf jeden Fall ausgeführt werden müssen, damit die Installation des Service Worker als Ganzes erfolgreich abgeschlossen wird. Schlägt eine der Operationen fehl, so wird auch der Service Worker nicht installiert. Das folgende Listing zeigt, dass bestimmte Ressourcen (HTML-, CSS- und JavaScript-Dateien) in den Cache übernommen werden müssen, damit die Anwendung später auch offline funktioniert.

```
self.addEventListener('install', event => {
  event.waitUntil(
    caches.open('pwa-demo-v1')
      .then(cache => cache.addAll([
        '/', '/index.html', '/style.css', '/app.js'
    ]))
  );
});
```

Bei erfolgreicher Installation wird der Service Worker im weiteren Verlauf aktiviert. Er lebt in einem separaten Prozess außerhalb der Webanwendung. Somit kann er auch dann ausgeführt werden, wenn die Webanwendung nicht geöffnet ist, womit sich etwa Push-Benachrichtigungen oder die Hintergrundsynchronisierung implementieren lassen. Außerdem übernimmt der Service Worker die Kontrolle über die Anwendung, indem er als Proxy oder Interceptor agiert: Er kann sämtliche vom Scope ausgehenden HTTP-Anfragen abfangen, inspizieren und manipulieren. Das gilt nicht nur für dynamische Anfragen (zum Beispiel über *XMLHttpRequest* oder das Fetch-API) der Webanwendung, sondern auch für Anfragen des Webbrowsers nach den Anwendungsquelldateien (HTML, CSS und JavaScript) sowie externe Ressourcen (Google Fonts und so weiter). Das passende funktionale Ereignis nennt sich *fetch*. Der Service Worker kann in der Methode zur Ereignisbehandlung entscheiden, ob er die Anfrage über das Netz bedient oder aber über seinen lokalen Zwischenspeicher (siehe Bild 3).

Durch die Kombination von Netz und Cache ergeben sich unterschiedliche Caching-Strategien, die je nach Anfragekategorie eingesetzt werden können:

- Für das Ausliefern der bei Installation im Cache hinterlegten statischen Anwendungsquelldateien bietet sich beispielsweise die Strategie „cache only" an. Bei dieser Strategie werden Anfragen direkt und ausschließlich aus dem Service-Worker-Cache ausgeliefert.
- Wenn es um Daten in Echtzeit geht, etwa Aktienkurse, die bei einem konkreten Anwendungsfall nicht zwischengespeichert werden dürfen, kann hingegen die Strategie „network only" zum Einsatz kommen. Dann fragt der Service Worker diese Ressource ausschließlich über das Netz ab – auf die Gefahr hin, dass der Anwender offline ist und keine Antwort erhalten kann.
- Eine Social-Media-Timeline könnte hingegen der Strategie „network falling back to cache" folgen. Hier wird erst versucht, die Anfrage über das Netz zu beantworten. Sollte die Anfrage in eine Zeitüberschreitung laufen, wird sie stattdessen über den Cache beantwortet. So bekommt der Anwender immerhin irgendeinen Inhalt zu sehen, auch wenn es nicht der aktuelle ist. Allerdings muss der Anwender so lange warten, bis die Anfrage in die Zeitüberschreitung gelaufen ist.
- Wo es nicht unbedingt auf die Aktualität ankommt, beispielsweise bei Zeitungsartikeln, könnte schließlich die Strategie „cache falling back to network" eingesetzt werden. Hier schaut der Service Worker zuerst in seinem Cache nach. Findet er eine passende Antwort, liefert er sie unmittelbar zurück. Findet er keine, leitet er die Anfrage mithilfe der Fetch-Schnittstelle an das Netz weiter. Die Implementierung dieser Caching-Strategie zeigt das folgende Listing. Bei dieser Strategie erhält der Anwender also eine unmittelbare Rückmeldung, allerdings auf die Gefahr hin, dass diese nicht mehr die aktuellste ist. Werden im Offline-Zustand Ressourcen abgerufen, die noch nicht im Cache enthalten sind, können sie aber nicht angezeigt werden.

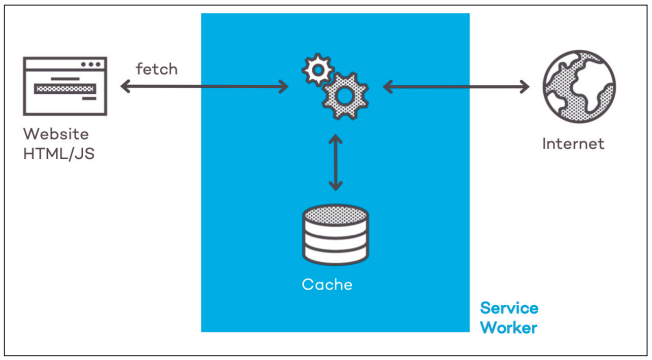
```
self.addEventListener('fetch', event => {
  event.respondWith(
    caches.match(event.request)
      .then(response => response ||
```

▶

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

Daneben gibt es noch weitere Caching-Strategien, die Jake Archibald, einer der Service-Worker-Autoren, in seinem Blog vorstellt [2]. Wer Service-Worker-Skripte nicht selbst implementieren will, kann zu Workbox greifen, einem von Google bereitgestellten Toolkit zur Generierung von Service-Worker-Skripten [3]. Auch das Single-Page-App-Framework Angular unterstützt Service Worker.

Insgesamt setzt der Service Worker also genau den oben beschriebenen zentralen Adapter für die Offline-Fähigkeit um. Am Service Worker führt außerdem kein Weg vorbei, da sämtliche unterhalb des Scope ausgelösten Anfragen ohne Ausnahme durch ihn durchgeleitet werden. Der Service-Worker-Cache eignet sich allerdings nur zum Zwischenspeichern von HTTP-Antworten, zum Beispiel Anwendungsquelldateien.



Die Funktionsweise des Service Worker (Bild 3)

Strukturierte Daten zwischenspeichern

Des Weiteren gibt es auch noch eine Reihe anderer Daten, die offline verfügbar sein sollen: strukturierte Anwenderdaten, zum Beispiel durch die Anwendung verwaltete Artikel- oder Kundendatensätze. Der Service-Worker-Cache ist zum Speichern solcher Daten eher ungeeignet. Stattdessen kommt eine zweite Technologie zum Einsatz: eine Indexed Database (IndexedDB), eine objektorientierte NoSQL-Datenbank für Browser. Auch die IndexedDB wird durch das W3C spezifiziert und hat dort in Version 2 den Rang einer offiziellen Empfehlung [4] [5] [6]. An der dritten Ausgabe wird derzeit gearbeitet.

IndexedDB kann Daten in verschiedenen Datenbanken ablegen, die sich wiederum in Objektspeicher (vergleichbar mit Tabellen) untergliedern. Ein solcher Objektspeicher verwaltet Schlüssel-Wert-Paare. Für Eigenschaften, die besonders schnell durchsucht werden sollen, können Indizes angelegt werden.

Da sich die nativen IndexedDB-Schnittstellen aber in der Verwendung als eher komplex erweisen, gibt es eine Vielzahl von Bibliotheken, welche die Handhabung der Browserdatenbank vereinfachen wollen. Eine solche ist Dexie.js [7].

Folgender Code zeigt, wie die IndexedDB-Datenbank *MyDatabase* mit Dexie.js geöffnet wird:

```
const db = new Dexie('MyDatabase');
db.version(1).stores({
  messages: '++id, content, sent'
});
```

Nach dem Öffnen wird das Schema dieser Datenbank angegeben: Es soll einen Objektspeicher mit dem Bezeichner *messages* geben, der Nachrichtenobjekte halten soll. Die zugewiesene Zeichenkette gibt die Indizes an, die für diesen Objektspeicher angelegt werden sollen.

Der erste Eintrag entspricht dabei immer dem Primärschlüssel, der für die Eigenschaft *id* angelegt und bei jedem neuen Datensatz automatisch hochgezählt werden soll (mithilfe von ++). Weitere Indizes werden für die Eigenschaften *content* und *sent* erzeugt.

Außerdem ist zu sehen, dass auch die Versionierung von Datenbankschemata vorgesehen ist. Auf diesem Weg sind auch Migrationen des Datenbankschemas möglich, wie sie aus Entity Framework bekannt sind. Folgendes Beispiel zeigt das Einfügen eines neuen Datensatzes in den *messages*-Objektspeicher. Das Objekt enthält die Eigenschaften *content* und *sent*. Die Objekteigenschaft *id* wird beim Speichern automatisch gesetzt. Es lassen sich auch weitere Eigenschaften definieren, in diesem Falle *sendDate*, die zwar mitgespeichert werden, aber nicht effizient durchsuchbar sind.

```
await db.messages.add({
  content: 'Wie geht's?',
  sent: false,
  sendDate: new Date()
});
```

Bei Dexie.js muss der Entwickler aber selbst noch die Logik zur Synchronisierung und zur Konfliktauflösung erstellen. Diese Arbeit kann wiederum die Bibliothek PouchDB dem Entwickler abnehmen. Sie speichert ihre Daten ebenfalls in einer IndexedDB und enthält zusätzlich eine Logik zur Synchronisierung der lokalen Datenbankinhalte mit der NoSQL-Datenbank Apache CouchDB. Allerdings muss dieses Datenbanksystem auch zur Architektur der Anwendung passen.

Sowohl die Webanwendung als auch der Service Worker haben Zugriff auf die IndexedDB. Service Worker können somit auch HTTP-Antworten aus der IndexedDB zusammensetzen und für bestimmte Anfragen zurückschicken. Der Zugriff ist wie beim Service-Worker-Cache nach der Origin isoliert. IndexedDB wird von Webbrowsern ab Microsoft Internet Explorer 10, Mozilla Firefox 4, Google Chrome 11 und Apple Safari 7.1 unterstützt.

In den ersten Browser-Ausgaben mit einer Unterstützung für IndexedDB fällt diese jedoch meist unvollständig oder fehlerhaft aus. In späteren Versionen ist sie weitestgehend stabil. Das bedeutet auch, dass IndexedDB von deutlich älteren Webbrowsern unterstützt wird, als es bei den Service-Worker-Schnittstellen der Fall ist.

Background Sync

Mit dem Service Worker und der IndexedDB stehen zwei Schnittstellen zur Verfügung, mit deren Hilfe Progressive Web Apps sowohl Anwendungsquelldateien als auch strukturierte Daten für die Offline-Verwendung zwischenspeichern können.

Jetzt betrachte ich noch die umgekehrte Richtung, die Synchronisierung von Anwenderdaten mit dem Backend. Grundsätzlich kann dies entweder zur Laufzeit der Webanwendung oder durch den Service Worker geschehen. Eingangs hat das Beispiel von WhatsApp gezeigt, wie diese App Nachrichten unabhängig von der Verbindungsqualität entgegennimmt. Dies ließe sich bei einer Progressive Web App über IndexedDB umsetzen. Sollen die Nachrichten nun mit dem Backend synchronisiert werden, sobald der Anwender wieder über eine Internetverbindung verfügt, so kann das Background-Sync-API eingesetzt werden. Diese Schnittstelle definiert das funktionale Service-Worker-Ereignis *sync*, das aufgerufen wird, wenn der Webbrowser der Ansicht ist, dass der Anwender wieder online ist. Besteht gerade eine aktive Internetverbindung, wird das Ereignis unmittelbar ausgelöst.

Das Background-Sync-API wird derzeit allerdings nur von Google Chrome ab Version 49 unterstützt, in Mozilla Firefox und Microsoft Edge wird die Schnittstelle gerade implementiert. Daher muss der Entwickler eine Methode in der Webanwendung anbieten, die zur Laufzeit ausgeführt werden kann. Folgendes Beispiel zeigt, wie der Entwickler aus der Webanwendung heraus das Synchronisieren von Daten anfordern kann:

```
if ('SyncManager' in window) {
  const reg = await navigator.serviceWorker.ready;
  reg.sync.register('send-messages');
}
```

Zuerst wird geprüft, ob die zugehörige Schnittstelle *SyncManager* auf dem globalen *window*-Objekt überhaupt zur Verfügung steht, da die Schnittstelle nicht von jedem Webbrowser unterstützt wird. Falls ja, wird sich auf der Service-Worker-Registrierung, die über die Eigenschaft *ready* bezogen werden kann, die Eigenschaft *sync* finden. Hier registriert sich der Entwickler mit einer bestimmten Kennung, hier *send-messages*. Derzeit wird nur das einmalige Synchronisieren von Daten unterstützt, noch keine periodischen oder geplanten Aufgaben.

Der Service Worker muss sich für das funktionale Ereignis *sync* registrieren und erhält dort Zugriff auf die von der Anwendung übermittelte Kennung. Entspricht diese *send-messages*, so kann er die passende Logik ausführen, zum Beispiel die noch nicht gesendeten Nachrichten aus der Datenbank laden und damit einen HTTP-Aufruf beim Anwendungs-Backend starten. Dieser Code skizziert den Vorgang:

```
self.addEventListener('sync', event => {
  if (event.tag === 'send-messages') {
    event.waitUntil(
      db.messages.where('sent').equals(false).toArray()
        .then(unsentMessages =>
          syncMessages(unsentMessages))
    );
  }
});
```

Schlägt der Vorgang fehl, etwa weil der Anwender zwischenzeitlich schon wieder keine Verbindung mehr hat, kann der Webbrowser das Ereignis noch weitere Male auslösen.

Fazit

Anwender erwarten von modernen Apps, dass diese auch offline zur Verfügung stehen. Das Offline-First-Prinzip hilft bei der Entwicklung offline-fähiger Anwendungen. So insbesondere bei Progressive Web Apps; ihr Anwendungsmodell setzt voraus, dass der Entwickler von vornherein die Offline-Fähigkeit der Anwendung mit bedenkt und mit umsetzt.

Mithilfe der Schnittstellen Service Worker und IndexedDB stehen dazu auch die notwendigen technischen Mittel zur Verfügung. Um den Service Worker führt zudem keine Webanfrage herum, sodass die Implementierung des Offline-First-Prinzips für die Anwendungsquelldateien über diese Technologie zur einfachen Übung wird.

Für strukturierte Anwenderdaten können Entwickler auf eine Vielzahl von IndexedDB-Helfern wie Dexie.js oder PouchDB zurückgreifen. Synchronisierung und Konfliktauflösung müssen dabei aber je nach Anwendungsfall besonders bedacht werden. Durch das Background-Sync-API lassen sich Daten auch dann synchronisieren, wenn der Anwender die App gerade gar nicht geöffnet hat. Somit eignet sich das Anwendungsmodell der Progressive Web Apps hervorragend zum Implementieren von Anwendungen, die das Prädikat „Offline First“ verdienen. ■

[1] W3C, Service Workers 1, www.dotnetpro.de/SL1901PWA1
[2] Jake Archibald, The offline cookbook, www.dotnetpro.de/SL1901PWA2
[3] Workbox, www.dotnetpro.de/SL1901PWA3
[4] Wikipedia, Indexed Database API, www.dotnetpro.de/SL1901PWA4
[5] MDN Web Docs, indexedDB, www.dotnetpro.de/SL1901PWA5
[6] W3C, Indexed Database API 2.0, www.dotnetpro.de/SL1901PWA6
[7] Dexie.js, http://dexie.org

Christian Liebel
ist Softwarearchitekt bei Thinktecture in Karlsruhe, wo er Cross-Plattform-Apps auf Basis von HTML5 und JavaScript entwickelt. Sein Handwerk hat er mit Microsoft-Technologien gelernt und wurde als Microsoft MVP ausgezeichnet.
@christianliebel

dnpCode A1901PWA