Eine Webseite wird aufgerufen. Der Server liefert sie. Ein Formular wird ausgefüllt und an den Server geschickt. Der Server antwortet mit einer leicht veränderten oder einer neuen Seite. Aus dem Menü wird eine neue Seite ausgewählt – der Server schickt eine neue usw.

Bei einer SPA oder *Single Page* Webseite werden alle Seiten auf einmal geladen. Ein Teil der Verarbeitung findet schon im Browser statt. Wird eine „neue" Seite angefordert, ist die schon im Speicher zu finden und daher blitzschnell verfügbar. Und noch etwas schneller: die erste Seite wird schon angezeigt, während der Rest noch geladen wird. Wartezeiten sind lästig – es gibt schon viele Möglichkeiten, diese zu verringern.

Natürlich müssen da noch ein paar Dinge im Hintergrund laufen. Wer auf die Zurück-Taste drückt, möchte natürlich wirklich die vorherige Seite der App sehen und nicht die Seite, die zuletzt vom Internet aufgerufen worden ist. Diese Verwaltung erledigt zum Beispiel *Preact*.

DOM – das Document Object Model

Was passiert eigentlich mit dem vom Server gelieferten HTML-Code? Nun, der Browser wandelt die Seite in eine hierarchische Struktur, das *Document Object Model* (kurz *DOM,* https://de.wikipedia.org/wiki/Document_Object_Model), um und erzeugt daraus das Bild, das wir sehen. Wenn nun eine kleine Änderung auf der Seite stattfindet, liefert der Server die komplette Seite neu aus. [Na ja – nicht immer. Mit *jQuery* können erstaunliche Effekte erzielt werden. Aber darauf will ich hier nicht eingehen.] Ja, einiges kann schon im Browser zwischen gespeichert werden, aber die Grundidee bleibt: die Seite muss wieder geladen werden.

Programmbibliotheken, wie *React* oder *Preact*, gehen einen anderen Weg: sie merken sich das Modell, registrieren nur mehr die Differenzen zu einer neuen Seite, werten sie aus und können daher die Seite sehr rasch wieder darstellen. Derartige Programme sind nicht unbedingt für PWAs notwendig, beschleunigen aber den Aufbau der Seiten wesentlich und arbeiten mit PWAs sehr gut zusammen.

Und wenn nun der Client selbst die notwendigen Änderungen berechnet, werden gar keine Daten vom Server übertragen. Damit kann die Seite noch schneller aufgebaut werden!

React

React (https://reactjs.org/) ist eine ECMAScript-Programmsammlung zum Erzeugen von Webseiten, vor allem über die Manipulation des DOMs. Aber *React* kann noch mehr: damit PWAs wirklich mit nativen Apps konkurrieren können, muss auch der Zugriff auf die Hardware möglich sein. Über Plugins für *React* können beispielsweise die Kamera, der GPS-Empfänger, der Fingerabdruckleser oder der Vibrationsalarm angesprochen werden. Die Liste der Komponenten, die damit auch für ECMAScript zur Verfügung stehen, wächst rasch.

Preact

Preact (https://preactjs.com/) ist eine ähnliche Bibliothek, die (größtenteils) zu *React* kompatibel ist, aber beim Speicherplatz, bei der Geschwindigkeit und beim Schreiben des Codes zum Teil recht eindrucksvolle Verbesserungen bringt.

Um mit Hardware-Komponenten, wie etwa der Kamera oder dem GPS-Empfänger, arbeiten zu können, muss oft auf die *React*-Programmbibliothek zurück gegriffen werden. Dank des *React*-Kompatibilitätsmodus ist das aber kein großes Problem.

Mein Eindruck: die Beschäftigung mit dieser Bibliothek bringt die größten Vorteile.

Service Worker

Auch mit den bisher genannten Eigenschaften hätten PWAs nur geringe Chancen, mit nativen Apps zu konkurrieren. Was wir benötigen, ist ein Werkzeug, um Nachrichten auf dem Endgerät empfangen zu können. Und dieser Mechanismus muss auch dann funktionieren, wenn die zu Grunde liegende Webseite nicht geöffnet ist. Wir brauchen also ein Stück Software, das auf *Ereignisse* reagieren kann. Da der Zeitpunkt, wann ein Ereignis auftritt, nicht vorhersehbar ist, sprechen wir von einer asynchronen Bearbeitung.

Ein *Service Worker* erledigt diese Aufgaben, lauscht auf Ereignisse und reagiert darauf. Welche Ereignisse können das sein? Zum Beispiel eine ankommende Nachrichten (zum Thema *Push* und *Notification* gleich mehr). Oder eine abgehende Mitteilung konnte (wegen eine fehlenden Internetverbindung) nicht gesendet werden: dann kann der *Service Worker* selbstständig beim Wiederkehren der Verbindung die Übertragung fortsetzen und abschließen. *Service Worker* setzen PWAs nicht voraus, ergänzen diese aber und schaffen so erst vollwertige Apps.

SSL-gesicherte Übertragung

Auf dem lokalen Server (http://127.0.0.1) kann man das alles leicht ausprobieren. Wird aber die App ins weite Internet entlassen, muss eine sichere Verbindung (SSL-gesichert, also über das https-Protokoll) verwendet werden. Vor einiger Zeit war das eine (für private Experimente) relativ teure Angelegenheit. CAcert ist eine Organisation von Freiwilligen, die die Ausgabe von kostenlosen Zertifikaten zum Ziel hat. Aber CAcert-Zertifikate werden von Browsern nicht direkt unterstützt.

Abhilfe schafft *„Let's encrypt"* (https://letsencrypt.org/). Die Zertifikate sind kostenlos, leicht zu installieren und werden von allen Browsern erkannt. Damit gibt es keinen Grund mehr, für eigene Webseiten keine SSL-Verschlüsselung zu verwenden. *Let's encrypt* ist sehr zu empfehlen.

Ich meine, dass diese Initiative mit Spenden unterstützt werden sollte. Auch SCHUL.InfoSMS verwendet Let's encrypt.

Clientseitige Datenbank

Wie werden Daten über die Lebensdauer eines Aufrufs hinaus aufbewahrt? Jeder Browser verfügt über einen *Cache* – einen „Zwischenspeicher". Der *Cache* ist eine Möglichkeit, aber nicht besonders komfortabel. Zusätzlich stehen aber mehrere Datenbanken zur Verfügung, zum Beispiel die *IndexedDB*: jedes gespeicherte Objekt hat einen Schlüssel *(Key)* und ein Datenfeld oder mehrere Datenfelder.

Nur zur Klarstellung: jeder moderne Browser bringt diese Datenbank schon mit, eine zusätzliche Installation ist nicht notwendig.

Push und Notification

Mit *Push* kann ein Server eine Nachricht an einen Client senden. Wichtig: diese Nachrichten werden vom Absender verschlüsselt und dann übermittelt, kein Browser-Hersteller hat darauf Zugriff! *Push*-Nachrichten können unterschiedliche Inhalte haben, zum Beispiel einen Datenbestand im Hintergrund (also ohne Eingriff des Benutzers) aktualisieren. Wenn eine *Push*-Nachricht am Bildschirm angezeigt werden soll, wird daraus eine *Notification*. Aber auch ohne *Notification* wird der Nutzer über die angekommene Push-Nachricht informiert.

Zum Ausprobieren: https://developers.google.com/web/fundamentals/codelabs/push-notifications/ ist ein kurzer Kurs zum Erstellen von Programmen, die einen *Service Worker* einrichten, *Push*-Nachrichten empfangen und als *Notification* anzeigen.

Für *Push*-Nachrichten und *Notifications* gilt: *der Benutzer muss dem Empfang dieser Nachrichten zustimmen!* Spam wird auf diese Weise verhindert. Wenn ein Benutzer den Empfang von Nachrichten aber *einmal* ablehnt, ist nicht einmal mehr eine nochmalige Einladung (durch den Server) möglich. Nur wenn der Nutzer in den Einstellungen seines Browsers die Sperre aufhebt, kann der Server *Push*-Nachrichten senden.

Push-Nachrichten und *Notifications* werden von *(P)react* sehr gut unterstützt.

Die Einführung der *Push*-Nachrichten wird als eine der *wesentlichsten Innovation der Webtechnologie* in den letzten Jahren bewertet.

Noch ein paar Gedanken zum Thema Datenschutz: Aktuelle Browser-Versionen empfangen Push-Nachrichten verschlüsselt.