Pimp my Browser

Bessere Web-Apps durch Web-APIs

Phillip Ghadir, Simon Kölsch

Der Begriff HTML5 fasst eine bunte Sammlung aus verfügbaren und geplanten APIs zusammen, auf die im Browser von JavaScript aus zugegriffen werden kann. Einige dieser APIs ermöglichen heute, Anwendungen für den Browser zu realisieren, die vor wenigen Jahren noch Plug-ins erfordert hätten. Die prominentesten Vertreter sind sicherlich Video oder WebRTC. Zum Beispiel ist heute die Public-Key-Verschlüsselung bereits im Web-Client integrierbar und das Reagieren auf Sensordaten der Betriebs-Hardware des Browsers möglich. Es folgt eine persönliche Auswahl an etablierten und ein Ausblick auf einige noch recht frische APIs.

Die vernetzte Welt funktioniert heute dank einer breiten Akzeptanz von im World Wide Web üblichen Standards. Und obwohl deren Verbreitung quasi universell sein könnte, werden proprietäre Frameworks von Einzelnen – dank der Marktmacht und strahlenden Kraft der Marke ihres Arbeitgebers – „gehyped".

Solange die Frameworks aus der Anwendungsentwicklung entstehen und für ihren Kontext Probleme erfolgreich adressieren, gibt es daran nichts auszusetzen. Wenn sie aber zur Abhängigkeit von proprietären Lösungen verleiten, für die es eigentlich eine allgemein verbreitete Lösung gibt oder geben sollte, ist die Verbreitung solcher Frameworks nicht nur gut.

In dieser Kolumne stellen wir gesammelt APIs im Webumfeld vor, die heute – oder hoffentlich in Kürze – für alle gängigen Browser-Plattformen verfügbar sind, auch wenn je nach API, Browser und Versionen eigene Namenspräfixe verwendet werden. Wir meinen, dass die Kenntnis dieser APIs hilft, den Einsatz irgendwelcher JavaScript-Frameworks besser zu bewerten.

Was geht?

Bevor wir uns also mit den APIs selbst beschäftigen, wären Instrumente zum Auswählen von APIs hilfreich. Einen guten Startpunkt bilden da:

- die API-Übersicht von Eric Wilde (s. Kasten),
- CanIuse.com,
- quirksmode.com und
- microjs.com.

Sind die ersten Schritte getan, helfen dann auch Testsuiten, die Anwendungen in verschiedenen Client-Umgebungen (von IE, Firefox, Chrome, iOS, Android) unterschiedlicher Versionen und mehreren Bildschirmauflösungen testen können. Darauf gehen wir in dieser Kolumne aber nicht weiter ein.

Eingaben und Sensoren

Tastatur und Maus sind nicht alles. Wer eine Web-Site oder Webapplikation für diverse Mobile-Geräte fit machen will, kann mit Hilfe einer Reihe von Device-spezifischen APIs auf Funktionen des Geräts zum Beispiel die Akku-Laufzeit, der Beschleunigungssensoren oder die geografische Ortsangabe auslesen.

Battery Status API

Mit dem Battery Status API lässt sich beispielsweise auf den aktuellen Batterie-Status eines Geräts zugreifen, zumindest unter Firefox, Chrome und auf neusten Android-Geräten. Damit lässt sich erfragen, ob ein Akku installiert ist, wie viele Leistung der Akku noch hat und ob er gerade aufgeladen wird.

Man kann Handler für die Events „wird geladen/oder auch nicht" oder „Energiestand aktualisiert" registrieren.

Page Visibility API

Möchte man im Client darauf reagieren, dass der Benutzer den Tab wechselt oder aber das Fenster minimiert, bietet sich dafür das Page Visibility API an. Diese Programmierschnittstelle fügt im Wesentlichen zwei nur lesbare Attribute zum `document`-Objekt hinzu:

- *hidden:* Wenn die Seite für den Benutzer nicht sichtbar ist, liefert `document.hidden` true, sonst false.
- *visibilityState:* Das Attribut enthält als String einen der vier möglichen Werte visible, hidden, prerender und unloaded, wovon nur der Wert „visible" `document.hidden == false` entspricht.

Wildes HTML5-API-Übersicht

Eric Wilde arbeitet seit Langem an seiner HTML5-API-Übersicht [Wilde] und führt dort verschiedene APIs jeweils mit dem aktuellen Zustand und einer kurzen Erläuterung auf. Die APIs sind aus den unterschiedlichsten Bereichen:

- (Verschiedenes)
- HTML5 Core
- Benutzerinteraktion
- DOM
- Geräte-spezifisch/Laufzeitumgebung
- Dateisystem
- Grafik
- Kommunikation/Netzwerk
- Medien
- Monitoring
- Persistenz
- Sicherheit

Auf die Änderung der Sichtbarkeit kann man über einen Event-Handler für das visibilityChanged-Ereignis reagieren.

Auch wenn das API von sehr vielen Browsern schon seit Langem unterstützt wird, verwenden einige für die genannten Attribute spezifische Präfixe, sodass das Attribut hidden dann „mozHidden“, „msHidden“ oder „webkitHidden“ und das visibilityChange-Ereignis dann „mozvisibilitychange“, „msvisibilitychange“ oder „webkitvisibilitychange“ heißen kann. Man beachte die konsistente Groß-/Kleinschreibung .

Geolocation API

Über das seit vielen Jahren verbreitete Geolocation API lassen sich bei entsprechender Zustimmung durch den Nutzer die Geokoordinaten von dem Gerät erfragen.

Wird das API unterstützt, ist `navigator.geolocation` definiert. Dann kann – sofern der Nutzer die Berechtigung dafür erteilt – die Position des Geräts entweder einmalig über

```
navigator.geolocation.getCurrentPosition( eineCallBackFunktion );
```

oder kontinuierlich über

```
navigator.geolocation.watchPosition( eineCallBackFunktion );
```

ermittelt werden. Mit `navigator.geolocation.clearWatch()` lässt sich so ein Handler dann auch wieder deregistrieren.

Die Callback-Funktion erhält als Argument die Position, die neben dem Längen- und Breitengrad auch noch die Höhenangabe sowie die Information über die Genauigkeit enthält. Eine Genauigkeit bezieht sich auf die Längen- und Breitengrade und eine auf die Höhenangabe. Weitere Informationen sind die Richtung, Geschwindigkeit und der Zeitpunkt der Antwort.

Damit lässt sich Einiges anfangen. Leider zeigen die Beispiele der Üblichen Verdächtigen einfach nur, wie die abgefragten Daten ohne Umschweife direkt zu Google geschickt werden können.

Nebenläufigkeit im Browser

Je komplexer wird, was wir im Browser ausführen möchten, desto schmerzlicher wird die ausschließlich sequenzielle Verarbeitung von JavaScript. Hier schafft das breit unterstützte *Web Worker API* Abhilfe: Diese Programmierschnittstelle ermöglicht die Ausführung von Skripten im Hintergrund parallel zu dem, was auf der aktuellen Seite ausgeführt wird.

In einer separaten JavaScript-Datei verpackt, können Web Worker per

```
var meinWorker = new Worker( "meineSpezielleWebWorker.js" );
```

erzeugt werden. Dabei muss das Worker-JavaScript vom selben Schema und URI-Pfad geladen werden, wie die HTML-Seite, die den Worker per Script-Tag einbindet. Hat das Instanziieren fehlerfrei geklappt, wird der Web Worker in einem eigenen Thread/Prozess – in dem Kontext der aufrufenden Seite – gestartet und ausgeführt, ohne die Ausführung auf der Hauptseite zu beeinträchtigen.

An den Web Worker kann man nun über dessen Methode `postMessage()` Nachrichten schicken:

```
meinWorker.postMessage( irgendwelcheJSONDaten );
```

Web Worker dürfen andere Skripte (via `importScripts()`-Methode) importieren sowie auf die WindowsTimers, den Navigator und XMLHttpRequest zugreifen. Zugriffe auf den DOM (und damit auf die Objekte window, document bzw. parent) sind nicht gestattet.

Die umgebende Seite kann bei dem Web Worker einen On-Message-Handler registrieren, über den sie der Worker benachrichtigen kann. Die umgebende Seite kann einen Worker mit `terminate()` beenden. Innerhalb eines Workers kann er sich selbst per `close()` beenden.

Online/Offline Mode

Es gibt ein API zum Erkennen, ob der Browser im Offline-Modus ist. Leider verspricht der Name mehr, als durch das API tatsächlich geliefert wird. Ist man beispielsweise im Firefox im Online-Modus, obwohl man selbst keinerlei Internet-Verbindung hat, liefert die Programmierschnittstelle immer noch ein „Alles gut, Du bist online.“

In anderen Browsern ist das Verhalten etwas anders, aber auch da erkennt das API nicht den Zustand „Kein Internet“, sofern irgendeine Netzverbindung besteht. Wer hier programmatisch auf ein fehlendes Netz reagieren möchte, könnte sich mit einem geeigneten Monitoring-API behelfen. Darauf gehen wir hier nicht weiter ein.

Es gibt noch deutlich mehr APIs, mit denen auf die Umgebung reagiert werden kann. Als Beispiele werfen wir mal die noch sehr jungen Ambient Light API und Proximity API in die Runde. Belassen wir es dabei und greifen einen etablierten Vertreter für das Client-seitige Persistieren von Daten heraus.

Daten Client-seitig persistieren

Über das quasi universell verfügbare *Web Storage API* stehen jeder Webanwendung im Browser zwei Key-/Value-Maps zur Verfügung, in der Paare von Strings abgelegt werden können. Der sogenannte LocalStorage ist als Attribut des `window`-Objekts zugreifbar und speichert Daten permanent – über die Lebensdauer der Prozessinstanz des Browsers hinweg.

Dahingegen ermöglicht der sogenannte SessionStorage – ebenfalls zugreifbar als Attribut des `window`-Objekts – das Persistieren von Schlüssel-/Wert-Paaren für die Dauer einer Browser-Session. Die hier gespeicherten Daten werden beim Schließen des Browser-Tabs beziehungsweise der Browser-Tabs entfernt. Selbst, wenn dieselbe Seite mehrfach parallel in verschiedenen Tabs geöffnet wird, steht jedem dieser Tabs ein eigener SessionStorage zur Verfügung.

Alles, was von einer Domain geladen wird, kann allerdings auf die gleichen Daten in dem LocalStorage zugreifen. Ist LocalStorage definiert, kann man über

```
localStorage.setItem( "schlÜssel", "wert" );
```

einen Wert setzen und per

```
localStorage.getItem( "schlÜssel" );
```

auslesen. Es ist sichergestellt, dass der Zugriff nur auf Werte ermöglicht wird, die von Skripten der gleichen Domain gespeichert wurden. Analog funktioniert `sessionStorage`.

Das Web Storage API sieht sowohl für die im Session- als auch für die im LocalStorage gespeicherten Daten keine Weitergabe an den Server vor. Das heißt, die über das API gespeicherten Daten verbleiben auf dem Client, sofern die Anwendung diese nicht explizit ausliest und an das Backend überträgt.

Sicher?

Je mehr Anwendungslogik im Browser ausgeführt wird, desto höher sind die Anforderungen an die Sicherheit der Browser als Plattform. Dieser Situation wird mit unterschiedlichen APIs Rechnung getragen.

CORS

Am weitesten fortgeschritten, verbreitet und damit schon fast ein alter Hut ist ohne Frage „Cross-Origin Resource Sharing“, besser bekannt als CORS.

Aktuelle Sicherheitseinstellungen in den Browsern erzwingen normalerweise, dass per JavaScript nur Daten von der Domain geladen werden können, von der auch die Seite geladen wurde, die das JavaScript einbindet. Dies erschwert allerdings die Integration von Services, die sich auf anderen Domains befinden. Mit den Sicherheitseinstellungen wäre der Browser als Applikationsplattform nur für Systeme zu gebrauchen, die im Backend integriert werden.

CORS schafft hier Abhilfe durch die Definition allgemeingültiger, spezieller HTTP-Response-Header, die dem Browser mitteilen, zu welchen anderen Domains zusätzlich welche HTTP-Requests abgesetzt werden können. CORS wird von allen Browsern unterstützt und ist älteren Alternativen, wie beispielsweise JSONP, auf jeden Fall vorzuziehen.

Web Crypto API

Das Web Crypto API erlaubt ohne großen Aufwand, die Funktionalität zu nutzen, kryptografische Schlüssel zu generieren, zu exportieren, zu importieren und anzuwenden. Diese Funktionalität ist ohnehin schon in der TLS-Implementierung des Browsers vorhanden. Das Problem der Authentizität beim Übermitteln von JavaScript-Dateien löst das Web Crypto API nicht.

Länger laufende Requests werden asynchron über das Promises-Pattern behandelt. Dabei erhält man von der Programmierschnittstelle ein „Promises“-Objekt und übergibt eine eigene Funktion, die im Erfolgsfall ausgeführt werden soll. Ein SHA-512-Hash kann zum Beispiel so gebildet werden:

```
var encoder = new TextEncoder("utf-8");
window.crypto.subtle.digest(
        {name:'SHA-512'},
        encoder.encode("Text"))
    .then(function(resultHash) {
        // Code im Erfolgsfall });
```

Dazu kommt ein kryptografisch sicherer Zufallszahlengenerator, der ein übergebenes Array befüllt:

```
window.crypto.getRandomValues(new Uint32Array(5));
```

In aktuellen Browsern ist das API bereits vollständig implementiert und für gängige Algorithmen stabil und benutzbar.

Übrigens: An einem Modell zum sicheren Übertragen von JavaScript wird unter dem W3C-Draft-Titel „Privileged Contexts“ gearbeitet, konkrete Umsetzungen sind aber im Moment noch nicht abzusehen.

Content Security Policy 2 - Level Up

Der Server kann durch Setzen des „Content-Security-Policy“-Headers im HTTP-Response einen zusätzlichen Sicherheitsmodus im Browser aktivieren. Zum Beispiel kann das Auslagern von JavaScript in externe Dateien erzwungen und dadurch das Einschleusen von Code über script-Tags verhindert werden.

Damit wird eine der am weitesten verbreiteten Angriffstechniken auf Webapplikationen eingeschränkt: das Cross-Site-Scripting oder „XSS“. Diese Methode schleust ein bösartiges Skript in den Kontext einer sonst vertrauenswürdigen Seite ein und kann dann Aktionen mit den Berechtigungen des Users ausführen.

Level 2 der Content Security Policy (CSP) ist seit Februar eine W3C Canidate Recommendation, wird also von den Browserherstellern implementiert. Dabei gab es einige Änderungen, die hier den Rahmen sprengen würden.

Level 2 der Content Security Policy führt neue Sicherheitsregeln ein:

- Zum Beispiel sind Redirects nicht mehr pauschal möglich, sondern müssen über den Ausdruck 'unsafe-redirect' erlaubt werden.
- Zudem bezieht die Spezifikation jetzt auch explizit Web Worker ein und
- ermöglicht, das Abschicken von Formularen zu kontrollieren.
- Der Aufruf beliebiger Plug-ins ist nun nicht mehr ohne Weiteres möglich.
- Inline-Elemente wie JavaScript und Stylesheets sind nun zwar möglich, müssen aber über Hashwerte und Nonces abgesichert werden.

Schleust ein Angreifer hier Code in einem Inline-Skript ein, verändert er zwangsläufig den Hash und der Browser wird diesen Code nicht ausführen. Verletzungen dieser Regeln konnten schon bei CSP Level 1 über eine 'report-uri' automatisiert zurückgemeldet werden. Dieser Mechanismus wurde nun durch DOM-Events mit mehr Informationen erweitert.

Inhalte nachträglich per HTTPS

Alle Anforderungen für eine Anwendung zu kennen, ist leider eher die Ausnahme. Bis die Entscheidung getroffen wird, eine Domain über HTTPS erreichbar zu machen, können mehrere Content-Management-Systeme im Einsatz gewesen sein und das eigentliche Markup wird von einem ganzen Dutzend an Services aggregiert und produziert. Teile davon sind vielleicht sogar eine Sammlung von älterem Content mit hartcodierten Links. Ist das Linkziel nicht relativ, sondern das Protokoll mit angegeben, kann es fast unmöglich sein, alte Links auf HTTPS „umzubiegen“. Auch Redirects sind nicht immer die passende Lösung.

Abhilfe könnte hier der Draft „Upgrade Insecure Requests“ schaffen. Damit wird eine zusätzliche Content-Security-Policy geschaffen, welche den Browser einfach anweist, alle Ressourcen über HTTPS und nicht über HTTP zu laden. Ressourcen von anderen Domains sind davon natürlich nicht betroffen. Durch den

```
Response-Header: Content-Security-Policy: upgrade-insecure-requests
```

wird ein Link `<a href="http://example.org/toc">` danach als `<a href="https://example.org/toc">` behandelt. `<a href="http://w3.org">` wird weiterhin gleich behandelt. Google Chrome unterstützt dieses Feature seit Version 43 und Firefox arbeitet an der Implementierung.

Credential Management

Dieser Entwurf nimmt sich dem Problem der Zugangsdaten im Browser an. Passwörter sollten nicht nur häufig geändert werden, sondern können schon fast früher oder später als offen angesehen werden. Daher sollte man am besten für jeden Dienst ein anderes Passwort verwenden.

Nicht jeder greift dabei auf die Hilfe eines extra Passwort-Managers zurück, sondern nutzt zum Beispiel für den Zugang zu Webforen die Passwort-Speichern-Funktion des Browsers selbst. Dabei kann der Browser eine gespeicherte Kombina-

tion von Benutzername und Passwort nur dadurch an die Webapplikation weitergeben, indem er die Input-Felder im Anmelde-Formular ausfüllt. Der Browser erkennt ein mit autocomplete markiertes User/Passwort-Feld und füllt dieses mit den entsprechenden Daten. Eher unübliche Mechanismen, wie Login über einen von JavaScript aus verschickten XMLHttpRequest oder OpenID/OAuth, fallen hier unter den Tisch.

„Credential Management Level 1" hat zum Ziel, ein möglichst einfaches API zur Verfügung zu stellen, bei dem Applikationen nach Zugängen unterschiedlicher Art im Browser fragen können. Einverständnis des Benutzers vorausgesetzt, wäre es so möglich, nicht nur Passwörter einfacher zu übermitteln, sondern auch komplexere Daten wie zum Beispiel „Access-Tokens".

Im Moment befindet sich das API zwar noch in der Entwurfsphase, wird aber als Hauptautor von Google getragen und damit hoffentlich bald in den Browsern zu finden sein.

Wie geht es weiter?

Der Schwerpunkt dieser Ausgabe von JavaSPEKTRUM ist ja die Frage nach der Zukunft von Java und JavaScript. Wir haben in dieser Kolumne ein grobes Bild dessen gezeichnet, was heute problemlos mit aktuellen Browsern möglich ist. Die Entwicklung mit JavaScript wird auch weiterhin erfordern, manche Plattformunterschiede in der Namensgebung beziehungsweise etwas anderen Schnittstellenspezifikationen explizit zu kompensieren.

Viele Unterschiede der Browser-Plattformen können durch mehr oder minder umfangreiche Frameworks ausgeglichen werden. Einige Frameworks können bereits heute manche fehlende Unterstützung von APIs kompensieren. Für Vieles gibt es gleich mehrere Framework-Alternativen. Auf Frameworks selbst konnten wir im Rahmen dieses Artikels leider gar nicht eingehen.

In diesem Artikel hätten eigentlich noch viel mehr APIs vorgestellt werden müssen. Und dabei entwickelt sich die Liste der unterstützten Features nicht nur in Bezug auf Sicherheit, Ausführungsgeschwindigkeit und Kommunikation immer weiter, sondern auch in Richtung der Laufzeitumgebung der Plattform, näher an die Hardware und damit näher an die Peripherie. Dank der wachsenden Sensorik in den Geräten und der Möglichkeit, auch für einen Browser eher exotische Peripherie ansprechen zu können (von Vibrations-Features über Abstandsmesser und Lichtsensoren bis hin zu Midi-Geräten) – die Palette des W3C bietet noch eine Menge an Standards an, auf deren breite Unterstützung wir uns noch freuen können.

Referenzen

[Canluse] `http://caniuse.com`
[Microjs] `http://microjs.com`
[Quirksmode] `http://quirksmode.com`
[Wilde] `https://github.com/dret/HTML5-overview`

Phillip Ghadir baut am liebsten tragfähige, langlebige Softwaresysteme. Er ist Mitglied der Geschäftsleitung bei innoQ und hat sich früh auf Architekturen für verteilte, unternehmenskritische Systeme spezialisiert. Darüber hinaus ist er Mitbegründer und aktives Mitglied des iSAQB, des International Software Architecture Qualification Board.
E-Mail: phillip.ghadir@innoq.com

Simon Kölsch ist Consultant bei innoQ mit den Schwerpunkten Webarchitektur und Security. In seiner Freizeit druckt er Dinge aus Plastik.
E-Mail: Simon.Koelsch@innoq.com