Peter Kröner

HTML5

Webseiten innovativ und zukunftssicher

Bibliografische Information Der Deutschen Nationalbibliothek

Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.d-nb.de abrufbar.

Gesamtlektorat: Sacha Storz
Satz: Open Source Press (LaTeX)
Umschlaggestaltung: Open Source Press
Gesamtherstellung: Kösel, Krugzell

ISBN 978-3-937514-97-0 http://www.opensourcepress.de

Kapitel 6

Offline-Webanwendungen

Was machen Webapps, wenn die Internetverbindung gekappt wird? Ganz einfach: Sie funktionieren nicht mehr. Das ist einer von vielen Faktoren, die den Einsatz von Online-Anwendungen in Bereichen, in denen WWW-Anbindung nicht *der* zentrale Punkt der Applikation ist (z. B. bei einem Grafikeditor oder einer Textverarbeitungsapp), noch eher unattraktiv machen. Es ist vielleicht in der Theorie ganz praktisch, Textverarbeitung und To-Do-Listen im Netz zu verwenden, aber dass sie (möglicherweise) hin und wieder einfach nicht erreichbar sind, ist ein ganz gewichtiger Nachteil. Mit HTML5 und diversen erweiternden Technologien hat man aber die Möglichkeit, seine Webapps offline-tauglich zu machen – und zwar *vollständig* offline-tauglich, ohne Kompromisse.

So kann man beim ersten Aufruf einer Seite ganz gezielt Ressourcen durch den Browser vorladen und cachen lassen (also zum Beispiel Skripte, Bilder oder HTML-Seiten), die dann, sollte der Besucher nach dem Aufruf des Index und dem Download der Ressourcen offline gehen, weiterhin zur Verfügung stehen – selbst wenn der Besucher diese Ressourcen bis dahin noch

nie aufgerufen haben sollte. Außerdem ist es möglich, offline durchgeführte Aktionen des Users (wenn er zum Beispiel einen neuen Datenbank-Eintrag anlegt) im Browser zwischenzuspeichern und erst abzusenden, wenn wieder WWW-Zugang vorhanden ist. Bei einer konsequent mit Offline-Features ausgestatteten Webapp ist also für den Nutzer völlig gleichgültig, ob er gerade online ist oder nicht, weil sich die App selbst um dieses Problem kümmert. In diesem Kapitel werden wir genau eine solche Anwendung mit HTML5 und etwas JavaScript entwickeln.

6.1 Vorbereitungen für ein Beispiel

Als Beispielanwendung der Offline-Features von HTML5 werden wir eine kleine To-Do-Listen-Verwaltung programmieren. Diese Webapp wird das Hinzufügen neuer To-Do-Punkte via Ajax, die Anzeige aller eingetragenen Punkte und die Anzeige einer Seite **Über dieses Beispiel** (`about.html`) enthalten. Als Gerüst dient uns die folgende HTML-Datei:

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8">
        <title>To-Do-Listen-App</title>
    </head>
    <body>
        <h1>To-Do-Liste</h1>
        <p>
            <a href="about.html">Über dieses Beispiel</a>
        </p>
        <h2>Einträge</h2>
        <ul id="list">

        </ul>
        <form id="new">
            <h2>Neuer Eintrag</h2>
            <p>
                <label for="todoDate">Datum:</label>
                <input type="date" id="todoDate">
            </p>
            <p>
                <label for="todoTodo">Aufgabe:</label>
                <input type="text" id="todoTodo">
            </p>
            <p>
                <input type="button" value="Speichern!" id="submit">
            </p>
        </form>
    </body>
</html>
```

Die Datei `about.html` könnte wie folgt aussehen:

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8">
        <title>Über das To-Do-Listen-Beispiel</title>
    </head>
    <body>
        <h1>Über das To-Do-Listen-Beispiel</h1>
        <p>
            Das To-Do-Listen-Beispiel ist eine kleine HTML5-Demo.
        </p>
    </body>
</html>
```

Als Browser sollten im Rahmen dieses Beispiels Firefox 3.5+ oder Safari 4+ verwendet werden. Zwar unterstützen andere Browser hier und da auch bereits Teile der Offline-Features, ein sinnvolles Gesamtwerk lässt sich aber im Rahmen von ein paar Seiten nur mit den genannten Browsern entwickeln (Stand Jahresbeginn 2010). Der Teil der Offline-Features, der auch in Chrome und IE8 funktioniert, wird entsprechend besprochen, doch im Allgemeinen werden wir mit Firefox und Safari schon genug zu tun haben. Wichtig sind des Weiteren die Entwicklertools in den jeweiligen Browsern – Firebug, WebInspector und die IE-Tools werden für Logs gebraucht und JavaScript-Konsolen erleichtern die Arbeit mit den diversen Cache-Mechanismen enorm.

Um für den Internet Explorer gerüstet zu sein, brauchen wir abschließend eine Funktion, die uns die Arbeit mit den verschiedenen Eventmodellen der unterschiedlichen Browser abnimmt:

```
// Cross-Browser-addEvent()-Funktion
function addEvent(element, type, func){
    if(document.addEventListener){
        element.addEventListener(type, func, false);
    }
    else{
        element.attachEvent('on' + type, func);
    }
}
```

Sämtliche Funktionen dieser kleinen Demo sollen auch dann zur Verfügung stehen, wenn mitten im Betrieb die Verbindung abbricht. Das erste Ziel unserer Bemühungen ist die Anzeige der informativen Seite **Über dieses Beispiel** – es könnte ja sein, dass jemand `about.html` lesen möchte, wenn er gerade offline ist. Wir brauchen für unsere Webapp also einen Application Cache, in dem wir diese Datei vorrätig halten können.

6.2 Der Application Cache und das Cache Manifest

Ein Cache Manifest ist eine Datei, in der Adressen zu Ressourcen (Bilder, Skripte, HTML) festgehalten werden, die der Browser für diese Website vorrätig halten *muss*. Was in einem Manifest steht, wird vom Browser geladen und gespeichert – in etwa wie im normalen Browsercache, aber präzise durch die Website kontrollierbar, versioniert und automatisch mit dem Netz synchronisiert. Das Ganze ist weniger volatil als der normale Cache und eine bestimmte Ressource muss nicht erst durch den Besucher aufgerufen werden, bevor sie gespeichert wird. Jedes Manifest definiert damit einen eigenen sogenannten *Application Cache* für die jeweilige Webanwendung.

Um die Seite **Über dieses Beispiel** in den Application Cache zu befördern, so dass wir, egal was das Netzwerk nach dem Aufruf der To-Do-Listen-App macht, in jedem Fall Zugriff darauf haben, brauchen wir also ein solches Manifest.

6.2.1 Cache Manifest anlegen

Ein Cache Manifest ist eine normale Textdatei, in der die URLs der zu speichernden Ressourcen aufgelistet werden. Sie muss UTF-8-kodiert sein und ihr MIME-Type muss `text/cache-manifest` lauten.[1] Ein solches Manifest könnte wie folgt aussehen:

```
CACHE MANIFEST
# Die erste Zeile ist Pflicht. Und das hier ist ein Kommentar.

    # Kommentare, Leerzeichen vor Kommentaren und
    # leere Zeilen werden ignoriert.

# Diese Dateien sollen für den Offline-Betrieb gespeichert werden:
img/foo.png
scripts/bar.js

# Jede Datei und jeder Kommentar braucht ihre eigene Zeile
```

Alle Pfade von zu cachenden Ressourcen müssen *relativ zum Manifest* angegeben werden. Wenn wir `about.html` im Application Cache speichern wollen, müssen wir das folgende einfache Manifest anlegen:

```
CACHE MANIFEST
about.html
```

[1] Ausgehend davon, dass Manifeste die Dateierweiterung `.manifest` haben, folgendes in die Datei `.htaccess` eintragen: `AddType text/cache-manifest manifest`

Die Einbindung des Manifests ist denkbar einfach – man gibt einfach die URL der Manifest-Datei im `manifest`-Attribut des `html`-Elements an:

```
<html manifest="cache-manifest.manifest">
```

Wie finden wir nun heraus, ob das Manifest auch wirklich funktioniert? Man könnte natürlich die Index-Seite unserer Webapp aufrufen, die Netzwerkverbindung trennen und dann prüfen, ob man auf `about.html` zugreifen kann. Nur gibt uns das natürlich keine Auskunft über eventuelle Fehler im Manifest, über laufende Synchronisationen mit aktualisierten Ressourcen, 404-Fehler oder den Zeitpunkt, ab dem alle Ressourcen sicher im Application Cache gelandet sind. Was wir brauchen, ist ein umfassendes Logging aller Cache-Aktivitäten und mit den passenden Events ist das auch kein Problem.

Bevor wir uns mit dem Logging befassen, ist aber noch einmal erwähnenswert, dass der Application Cache *nichts* mit dem normalen HTTP-Cache des Browsers zu tun hat. Ein eventuell vorhandenen Cache Manifest hat auf diesen Cache keinerlei Einfluss.

6.2.2 Manifest-Events

Alles in HTML5 hat eine API, das gilt auch für Application Caches. In die Details steigen wir im Abschnitt 6.2.5 ab Seite 162 ein, doch um zu überwachen, ob und wie gut unser Manifest funktioniert, ist es sinnvoll, auf die Manifest-Events vorzugreifen. Wann immer etwas bzgl. Application Cache passiert, werden Events auf dem globalen Objekt `applicationCache` gefeuert. Diese sind im Einzelnen:

Tabelle 6.1: Manifest-Events

Event	Ereignis	Folge-Events
`checking`	Browser versucht das Manifest zum ersten Mal zu laden oder prüft das Manifest auf Updates.	`noupdate`, `downloading`, `obsolete`, `error`
`noupdate`	Das Manifest hat sich nicht verändert.	Keine
`downloading`	Das Manifest wird heruntergeladen oder seine Ressourcen werden erstmals heruntergeladen.	`progress`, `error`, `cached`, `updateready`
`progress`	Ressourcen werden heruntergeladen.	`progress`, `error`, `cached`, `updateready`
`cached`	Ressourcen wurden heruntergeladen.	Keine

Fortsetzung:

Event	Ereignis	Folge-Events
`updateready`	Ressourcen wurden aktualisert.	Keine
`obsolete`	Manifest-Anfrage lieferte Fehler 404 oder 410, Application Cache wird gelöscht.	Keine
`error`	Das Manifest oder die Website konnten nicht korrekt geladen werden, Fehler beim Download der Ressourcen oder das Manifest hat sich während des Downloads geändert.	Keine; der Browser wird, hat sich nur der Manifest-Inhalt geändert, erneut einen Download versuchen.

Für unser einfaches Beispiel brauchen wir diese Events zwar nicht alle (es handelt sich ja nur um eine einfache HTML-Datei, die sich nie ändert), aber genau Buch führen sollten wir zu Übungszwecken dennoch. Deswegen loggen wir alle Events in die JavaScript-Konsole und weil in diesem Fall der Internet Explorer mangels Unterstützung für Caches keine Rolle spielt, realisieren wir das ganz einfach per `addEventListener()`:

```
// Manifest-Events

// Damit sich Chrome und der IE nicht
// verschlucken, fragen wir, ob der Browser
// das applicationCache-Objekt unterstützt

if(typeof applicationCache !== 'undefined'){
    applicationCache.addEventListener('checking', function(){
            console.log('Suche Manifest...');
    }, false);
    applicationCache.addEventListener('noupdate', function(){
            console.log('Nichts neues im Manifest gefunden.');
    }, false);
    applicationCache.addEventListener('downloading', function(){
            console.log('Lade Manifest und Ressourcen...');
    }, false);
    applicationCache.addEventListener('progress', function(){
            console.log('Lade Ressourcen...');
    }, false);
    applicationCache.addEventListener('cached', function(){
            console.log('Application Cache bereit.');
    }, false);
    applicationCache.addEventListener('updateready', function(){
            console.log('Cache aktualisiert.');
    }, false);
    applicationCache.addEventListener('obsolete', function(){
            console.log('Cache veraltet.');
```

```
    }, false);
    applicationCache.addEventListener('error', function(){
            console.log('Fataler Fehler beim Anlegen des Caches.');
    }, false);
}
```

Wenn man nun mit dem Firefox die To-Do-Listen-App aufruft, wird zunächst um Erlaubnis gebeten, Daten der Website lokal zu speichern. Nachdem das bestätigt ist (Safari fragt gar nicht erst), laufen die Events in die Konsole:

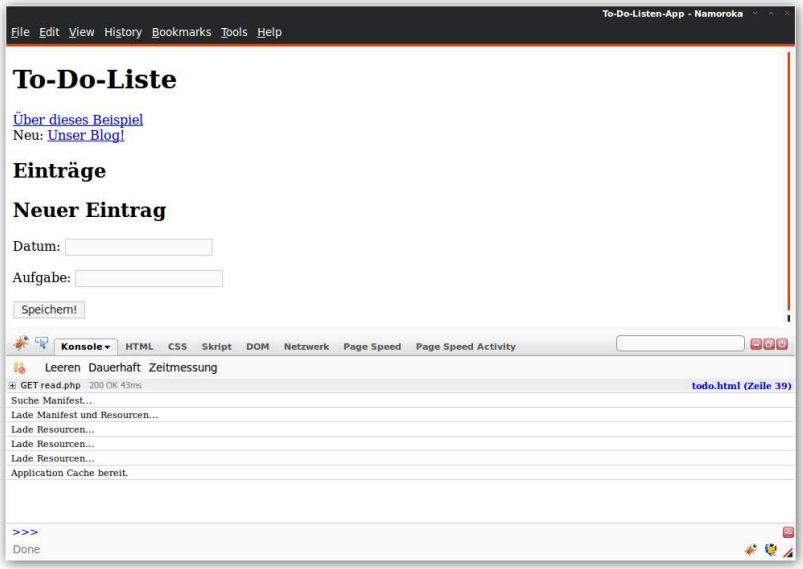

Abbildung 6.1: Ein Application Cache für die To-Do-App wird erstmals erstellt

Ab jetzt kann man den Netzwerkstecker ziehen (bzw. seinen Browser in den Offline-Modus versetzen, den lokalen Webserver abschalten oder die Netzwerkkarte deaktivieren) und auf den Link zur Info-Seite klicken. Obwohl kein Web-Zugriff besteht, wird die Seite angezeigt – dem Application Cache sei Dank.

Dieses Prinzip kann man auf alle Arten von Ressourcen anwenden, Bilder, JavaScript, für `<audio>` bestimmte Soundfiles etc. Wer sehr viele solcher Dateien in den Application Cache seiner User befördern möchte, kann das Manifest natürlich über ein serverseitiges Skript generieren lassen – aber wird man im Regelfall die komplette Webapp cachen wollen? Eher nicht. Etwas Manifest-Feintuning ist angebracht.

6.2.3 Fallbacks und Whitelists

Wenige Webapps sind so kompakt wie unser Beispiel und es ist nicht immer möglich (oder sinnvoll), alle denkbaren Ressourcen im Application Cache zu halten. Daten, die unter keinen Umständen in irgendeiner Form gecacht werden sollen, kann man mit einer Online-Whitelist im Manifest notieren und darüber hinaus Ersatzinhalte festlegen.

Online-Whitelisting

Nehmen wir an, zu unserer To-Do-Listen-App gäbe es auch ein Blog, das unter `blog.html` erreichbar wäre:

```
<h1>To-Do-Liste</h1>
<p>
    <a href="about.html">Über dieses Beispiel</a>
    <br>
    Neu: <a href="blog.html">Unser Blog!</a>
</p>
```

Dieses Blog könnte Hunderte Einträge haben, die wir allesamt nicht im Application Cache halten wollen. Und wir wollen auch nicht, dass jemand, der gerade im Offline-Modus unsere App benutzt, eine zuvor in seinem gewöhnlichem Browser-Cache gespeicherte Version des Blogs sieht – denn es könnte ja inzwischen neue Posts gegeben haben. Das Blog soll also grundsätzlich *nur* über das Netzwerk erreichbar sein und nie über irgendeinen Cache. Hierfür können wir einen Whitelist-Abschnitt im Manifest anlegen:

```
CACHE MANIFEST
about.html

# Alles folgende explizit NICHT cachen
NETWORK:
blog.html
```

Mit `NETWORK:` eröffnet man innerhalb des Manifests einen neuen Abschnitt, in dem alle Ressourcen notiert werden, die nur via Netzwerk erreichbar sein sollen. Der Browser wird nie versuchen, `blog.html` aus einem Cache zu laden, was natürlich bei fehlender Netzwerkverbindung dazu führt, dass der Browser außer einer Meldung **Seite nicht gefunden** nichts anzeigt. Das ist suboptimal, doch zum Glück gibt es die Möglichkeit, Ersatzinhalte festzulegen.

Fallback-Inhalte

Fehlerseiten, die *informativ* sind und über ein *Fehler 404* hinausgehen, gehören im Web zum guten Ton und es gibt keinen Grund, in offline-taug-

lichen Webapps darauf zu verzichten. Also erstellen wir eine informative Nachricht[2]...

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8">
        <title>Seite im Offline-Modus</title>
    </head>
    <body>
        <h1>Offline-Modus</h1>
        <p>
            Dieser Inhalt ist im Offline-Modus nicht verfügbar.
        </p>
    </body>
</html>
```

... und notieren im Manifest, dass diese HTML-Datei als Ersatzinhalt für alle Inhalte fungieren soll, die nicht erreichbar sind:

```
FALLBACK:
/ offline.html
```

Jeder erfolglose Zugriff auf eine Ressource (gekennzeichnet durch den Slash am Zeilenanfang), egal ob mangels Erreichbarkeit oder wegen eines anderen Problems (Serverfehler etc.), führt jetzt dazu, dass als Ersatz die Datei `offline.html` angezeigt wird. Damit ist unser Application Cache Manifest für die To-Do-Listen-App fertig:

```
CACHE MANIFEST
about.html
offline.html

NETWORK:
blog.html

FALLBACK:
/ offline.html
```

Bevor wir uns nun daran machen, in unser Beispiel To-Do-Listen-Funktionalität einzubauen, lohnt sich ein noch genauerer Blick auf den Application Cache.

[2] Die hier entworfene Nachricht ist tatsächlich nicht besonders informativ, für unser Beispiel aber ausreichend. Im echten Einsatz sollte natürlich etwas wirklich Hilfreiches mitgeteilt werden und vielleicht auch, dass man den Rest der Webapp weiterhin benutzen kann. Im Abschnitt 6.3 werden wir dafür sorgen, dass man auch ohne Netzwerkverbindung neue To-Do-Punkte anlegen kann. Otto Normalsurfer dürfte noch die Gleichung „Offline = geht nicht“ im Kopf haben. Ein Hinweis, dass das Unmögliche jetzt möglich ist, wäre also eventuell angebracht.

6.2.4 Manifest-Tipps für Profis

Das Cache Manifest ist an sich zwar nur eine einfache Textdatei, doch man kann damit (und den entsprechenden APIs) mehr als nur URLs auflisten. Mit Abschnitten lässt sich das Manifest gleich mehrfach unterteilen, und mit der API lässt sich auf Updates und andere Events reagieren.

Abschnitte

Ein Manifest kennt drei Typen von Abschnitten: Den „normalen“ Abschnitt mit URLs der für den Application Cache bestimmten Inhalte, den Whitelist-Abschnitt (`NETWORK`) und den Abschnitt für Ersatzinhalte `FALLBACK`. Man kann beliebig viele dieser Abschnitte in einer Manifest-Datei benutzen. Jedes Mal, wenn man einen dieser Abschnitt-Header in die Datei setzt, endet der vorherige Abschnitt und ein neuer beginnt, man schaltet quasi zwischen drei Betriebsmodi hin und her. Der Header für den „normalen“ Abschnitt lautet `CACHE` und man könnte ein Manifest wie folgt aufbauen:

```
CACHE MANIFEST

# Erst mal ein paar Sachen auf die Whitelist setzen
NETWORK:
foo.html
bar.html


# Das Folgende dann bitte cachen
CACHE:
wichtig.html
ersatz.html


# Noch was auf die Whitelist
NETWORK:
bla.html


# Und das ist der Ersatzinhalt
FALLBACK:
/ ersatz.html
```

Unterhalb von `CACHE MANIFEST` (in der ersten Zeile) befindet man sich im normalen Cache-Abschnitt, ohne dass man diesen mit `CACHE` extra deklarieren müsste. Eine Ressource kann in mehreren Abschnitten auftauchen – so wird hier `ersatz.html` in einem Cache-Abschnitt für das Caching markiert und an anderer Stelle als Fallback-Inhalt angegeben.

Automatische Manifeste und ein Fallback für alle Fälle

Es spricht nichts dagegen, Manifeste durch Skripte automatisch zu erzeugen, man muss nur einfach für jede zu cachende Ressource einen Eintrag ausgeben. Zu beachten ist nur, dass *sämtliche* Ressourcen einzeln angegeben werden müssen, dass also zum Beispiel in eine HTML-Datei eingebundene Bilder und Skripte einen eigenen Eintrag brauchen. Der umgekehrte Fall (wenn man das Cachen explizit unterbinden, aber trotzdem einen Fallback-Inhalt anzeigen möchte) ist einfacher zu handhaben:

```
CACHE MANIFEST

FALLBACK:
/ /ersatz.html

NETWORK:
*
```

Dieses Manifest sorgt dafür, dass außer der Ersatz-Datei nichts im Application Cache landet, denn der Stern im Network-Abschnitt erfasst *alle* Seiten und Ressourcen. Gleichzeitig gilt für alle Inhalte der Ersatz `ersatz.html`. Damit würde man sozusagen aus den Offline-Features eine Website bauen, die eigentlich offline gar nicht wirklich etwas tut, sondern nur eine prompte und informative Offline-Meldung anbietet – in etwa wie bei einer angepassten 404-Fehlerseite. Derartiges wäre als zusätzlicher und recht einfach einzurichtender Service für Nutzer empfehlenswert, doch solange Firefox-Browser immer um Erlaubnis fragen, bevor der Cache angelegt wird, würde das beim Durchschnittssurfer wohl mehr Fragen aufwerfen als Mehrwert bieten. Eventuell ändert sich dieses Verhalten ja in zukünftigen Firefox-Versionen – bis dahin ist es wohl ratsam, den Application Cache nur dann zu verwenden, wenn man beabsichtigt, eine tatsächlich offline-taugliche Webapp zu bauen.

Speicherort und Speicherplatz des Application Cache

Wenn man mit dem Application Cache experimentiert, kommt man nicht umhin, ihn hin und wieder zu löschen. Am einfachsten lässt sich das bewerkstelligen, indem man die entsprechenden Dateien von der Festplatte entfernt:

Windows Vista und 7/Firefox

```
C:\Users\<benutzername>\AppData\Local\Mozilla\Firefox\P
rofiles\<profil>\OfflineCache
```

Windows Vista und 7/Safari

```
C:\Users\<benutzername>\AppData\Local\Apple Computer\Sa
fari/ApplicationCache.db
```

Windows XP/Firefox

```
C:\Dokumente und Einstellungen\<benutzername>\Lokale Ei
nstellungen\Anwendungsdaten\Mozilla\Firefox\Profiles\<p
rofil>\OfflineCache
```

Windows XP/Safari

```
C:\Dokumente und Einstellungen\<benutzername>\Lokale Ei
nstellungen\Anwendungsdaten\Apple Computer\Safari\Appli
cationCache.db
```

Mac OS X/Firefox

```
/Users/<benutzername>/Library/Caches/Firefox/Profiles/<
profil>/OfflineCache
```

Mac OS X/Safari

```
/Users/<benutzername>/Library/Caches/com.apple.Safari/A
pplicationCache.db
```

Linux/Firefox

```
/home/<benutzername>/.mozilla/firefox/<profil>/OfflineC
ache
```

Theoretisch (also gemäß den Spezifikationen) sollten sich diese Speicher auch via Browsermenü entfernen lassen, tatsächlich klappt das bisher noch nicht – ein Leeren des Cache schließt weder im Firefox noch in Safari den Application Cache ein. Im Firefox kann der Application Cache aber immerhin seitenweise unter **Extras | Einstellungen | Erweitert | Netzwerk** in Abschnitt **Offline-Speicher** eingesehen und gelöscht werden. Wie viel Speicherplatz belegt werden kann, ist im Firefox unter **Extras | Einstellungen | Erweitert | Netzwerk** konfigurierbar – per Default sind 100 MB eingestellt. Bei Safari scheint es kein (konfigurierbares) Limit zu geben.

6.2.5 Details der Application Cache API

Neben den in Abschnitt 6.2.2 behandelten Events des `applicationCache`-Objekts gibt es zwei weitere Teile der Application Cache API, die der Beachtung wert sind: Die Statusabfrage und der Cache-Swap.

Den Cache-Status abfragen

Die Events des Application Cache haben wir bereits kennen gelernt, als wir in den ersten Beispielen die Manifest-Aktionen geloggt haben. Zusätzlich lässt sich unter `applicationCache.status` auch ereignisunabhängig der aktuelle Status des Cache abfragen. `0` und `5` stehen dabei für die folgenden Zustände: