digital books on the web

Rather than trying to come up with new strategies for taking ebooks forward (strategies we know are not going to work in the long run) I've been exploring processes and technologies to turn web content into publishable content. This will definitely rehash some older posts but, I hope, will provide a fresher perspective to the technologies since it's been a few months since I last looked at them.

The web is getting close to native and that might not be a bad thing

In terms of performance and functionality the web has come closer and closer to native applications without plugins or having to install plugins or full blown applications. If you're in Android the integration is full and very tight. Some of the things your web applications can do when properly configured:

1. Add an icon to the homescreen after the user has interacted with the web app for a while
2. Cache specific data for reliable and offline use
3. Once the user accepts, send her push notifications
4. Update content on the background

In the context of a book-like experience we'll concentrate in #1, #2 and #4 and illustrate how they may be used to create a reading experience.

Enter the PWA technologies

Progressive Web Applications (progressive web apps or PWAs) are a set of Web APIS and conventions to make web applications work more like native applications in Android and iOS.

Add to homescreen

We have been able to add applications to the device's homepage since early versions of the iPhone. What has changed is the automation of the process and what it takes for the browser to decide you've engaged with the application. Apps on the homescreen provide a good user experience and to do that they:

- Should load instantly, regardless of network state. They must put their own UI on screen without requiring a network round trip.
- The brand or site behind the app shouldn't be a mystery.
- Can run without extra browser chrome (e.g., the URL bar). This is a potentially dangerous permission. To prevent hijacking by captive portals (and worse), apps must be loaded over TLS connections.

These concerns give rise to today's Baseline Criteria. To be a Progressive Web App, a site must:

- Originate from a Secure Origin. Served over TLS and green padlock displays (no active mixed content).
- Load while offline (even if only a custom offline page). By implication, this means that Progressive Web Apps require Service Workers.
- Reference a Web App Manifest with at least the following properties:
    - name
    - short_name
    - start_url
    - display with a value of standalone or fullscreen
    - An icon at least 144×144 large in png format. E.g.: `"icons": [ { "src": "/images/icon-144.png", "sizes": "144x144", "type": "image/png" } ]`

Criteria taken from Alex Russell's What, Exactly, Makes Something A Progressive Web App?

So if the user interacts with your site it will eventually prompt him to add the site to the homescreen and, depending on the OS you're working on, be able to interact with it as a full fledged application.

Web Application Manifest

The Web Application Manifest is a JSON file that provides additional information for your application including names, different resolutions for the homescreen icon, a splash screen and a lot of the elements that make for an application. This will also work with Microsoft and iOS devices, not just Android.

The Web Application manifest uses a `link` like the one below to link to the manifest file.

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

An example manifest from Paul Kinlan's Air Horner looks like this.

```
{
  "name": "The Air H"The Air Horner""short_name": "Air Horner",
  "icons"       "src": "ima": "touch/Airhorner_128.png",
        "type": "i": ""type""type"   "sizes": "128x128"
 ": ""sizes"       "src": "images/touch/Air": ""src"2.png",
        "type": "image/png",
       ": ""type""152x152"
      }, {
  "152x152""sizes": "152x152"
      }, {
        "src"   "type": "image/png",
        "sizes": "14": ""type"   }, {
        "src": "images": ": "144x144"
      }, {
        "src"": "image/png",
        "sizes": "192x192"
 ": ""sizes""type"       "src": "images/t": ""sizes"ner_256.png",
        "type": "image/": ""src"     "sizes": "256x256"
      },
      {
   ": ""type": "images/touch/Airhorner_512.png",
        "type": "image/png",
  "images/touch/Airhorner_512.png""src""start_url": "/?homescreen=1",
  "scope": "/": ""start_url""type": "image/png",
        "sizes": "512x512"
      }],
  "start_url"6F3",
  "theme_color": "#2196": ": "/",
  "display": "standalone",
  "orientation": "portrait",
  "background_color": "#2196F3",
  "theme_color": "#2196F3"
}
```

Service Worker

Before we jump into theory and code it's a good idea to remember that service workers are not the first tool created to make content available offline. Google Gears attempted to do this first and failed. Some of the people who worked in the Gears project at Google attempted to create a web version of Gears' offline storage and standardize it as Application Cache.

Application Cache failed to get traction because of its reliance in implicit behavior. Writing the App Cache manifest itself is straight forward. Using it becomes harder when ou have to wade over the many implicit rules that make App CAche work... as a result many people gave up because the pages would never display or would never update the content, regardless if you were online or not.

Jake Archibald wrote Application Cache is a Douchebag to document the problems he experience in a successful App Cache Deployment at Lynrd. It is illustrative of the problems developers experienced when deploying the API and the workarounds they had to make so that users had a moderately successful experience.

With the lessons of Application Cache fresh in mind the brains at the W3C started working on the next iteration of offline caching and performance improvement APIs. It is the Service Worker.

Where App. Cache works with a lot of implicit behavior and assumptions Service Worker forces you to be explicit in what you want to accomplish, whether it's is to designate the resources to cache or intercepting a network request and providing an alternative resource if the user is offline and the resource not cached.

Having to explicitly code the behavior you want gives you a lot of flexibility when choosing what resources you want to cache and how you want to handle individual requests. In the index page of your website put the code below and replace the `/sw.js` with the name of your service worker that must be located in the root directory of your site or application.

```
if ('serviceWorker' in navigator) {
  window.addEventListener('load', function() 'load'navigator.serviceWorke
      '/sw.js'// Registration was successfulonsole.log('ServiceWorker reg
    }).catch(function(err) {
```

```
      // regis'ServiceWorker registration successful with scope: '// regis
      console.log('ServiceWorker registration failed: ', err);
    });
  });
}
```

`sw.js` is the actual service worker script. The first part sets up the caches and the list of URLs to cache when the user first access the Service Worker controlled site. These resources are ones used in the index page and may include images, style sheets, scripts, fonts and others. Just make sure you don't go overboard or you'll defeat the purpose of precaching resources.

This is the place where you update the names of your caches to trigger the automatic update process. We'll discuss this in more detail later.

```
// Names of the two caches used in this version of the service worker.
// Change to v2, etc. when you update any of the local resources, which w
// in turn trigger the install event again.
const PRECACHE = 'precache-v1';
const RUNTIME = 'runtime-v1';

'precache-v1'// A list of local resources we always want to be cached.HE_U
  'index.html',
  './', // Alias for 'index.html'// Alias for index.html
  'styles.css',
  '../../styles/main.css',
  'demo.js'
];
```

The first event we want to set up is the `install` event. In this event we precache the resources we defined at the top of the script. We then make the Service Worker take over the page and site immediately and not use the default behavior of waiting until the browser reloads the content.

```
// The install handler takes care of precaching the resources we always ne
self.addEventListener('install', event => {
  event.waitUntil(
```

```
    caches.open(PRECACHE)
      .then(cache => cache.addAll(PRECACHE_URLS))
      .then(self.skipWaiting())
  );
});
```

The activate handler is the maintenance and cleanup handler. Whenever we update the name of the cache constants at the top of the script, the activation process will delete those caches that are no longer need because the material has been updated.

```
// The activate handler takes care of cleaning up old caches.
self.addEventListener('activate', event => {
  const currentCaches = [PRECACHE, RUNTIME];
  event.waitUntil(
    caches.keys().then(cacheNames => {
      return cacheNames.filter(cacheName => !currentCaches.includes(cacheN
    }).then(cachesToDelete => {
      return Promise.all(cachesToDelete.map(cacheToDelete => {
        return caches.delete(cacheToDelete);
      }));
    }).then(() => self.clients.claim())
  );
});
```

The fetch event is the heart of the Service Worker. This is where we fetch resources for the application and interact with the user. In essence the fetch event does the following

- If the request comes from a different domain skip it
- If the item is in the cache, then return it
- If the item is not in the cache, fetch it from the network and:
    - Store a copy of the object in the cache
    - Return the item to the use

```
self.addEventListener('fetch', event => {
  // Skip cross-origin requests, like those for Google Analytics.
```

```
  if (event.request.url.startsWith(self.location.origin)) {
    event.respondWith(
      caches.match(event.request).then(cachedResponse => {
        if (cachedResponse) {
          return cachedResponse;
        }

        return caches.open(RUNTIME).then(cache => {
          return fetch(event.request)
          .then(response => {
            // Put a copy of the response in the runtime cache.
            return cache.put(event.request, response.clone())
            .then(() => {
              return response;
            });
          });
        });
      })
    );
  }
});
```

This service worker is simple and provides a core set of functionality to work with Service Worker. We can do other things like provide an offline page if the content is not in the cache and the network is down and many other things that we explicitly code.

Push Messaging and Notification

These two APIs are used on top of a service worker to provide push notifications and one-to-many communication services for your app. Say, for example, that you want to notify users when new content is added to your publication or content is updated; you can use push messages to notify the user of these changes.

The code for Push messaging and notifications is very dependent on the backedn you choose for your project. A couple examples:

- Firefox Cloud Services needs serious updates because of changes in the API but it's still a good starting point

- Web Push Notifications: Timely, Relevant, and Precise and associated code sample

Why bother?

I know what you're thinking: "***This sounds like a lot of work, why should I bother?***"

The quickest answer is: because it helps users. It provides easier access to web content and it gives them a way to view the content while they have spotty or no internet connectivity.

Creatively it gives you, the developer, a way to create better and original content. Serialized content, check. Content that requires specific web APIs like WebGL, check. The choices are limited by your imagination.

Links and resources

- Introducing Resilient Web Design
- Taking an online book offline
- Application Cache is a douchebag
- Resilient Web Design