Long form publishing with progressive subcompact applications

For the past few months I've been working at Google building a set of instructor-led courses on how to build progressive web applications. This has made me think of how to push some of these concepts into what I call "Progressive subcompact publications". These concepts are different than ePub Next and any number of formats vining for use, each of which have issues that are hard to overcome:

- They seek to replace the installed EPUB (and Kindle) user base. Since most users of iBooks and Kindles are locked in to their devices and readers this is not a good idea
- There will never be uniform buy in to new specs or ways to publish content and, unless you can get a majority of publishers to implement your specification, schema or idea you will be competing with a behemoth that is very slow to evolve (not questioning the reasons, just making a statement)
- Some people are trying to establish their format as a defacto standard (use this instead of what you already have) and that's dangerous
    - It's dangerous if you fail to get full buy in because it segments the market even further
    - it's dangerous if you succeed because the defacto standard becomes a de jure standard and you have to support it and work all the warts that were ok when you were developing it (check the Javascript specifications for the amount of baggage carried over to keep old code from breaking)

Instead I'm looking at progressive subcompact publications as a starting point for an exploration of how far we can push the web as a publishing medium.

What are progressive web applications

Alex Rusell coined the term "Progressive Web Applications" in Progressive Web Apps: Escaping Tabs Without Losing Our Soul. It is an umbrella term for a series of technologies and best practices to make our users experience feel more like native

applications without loosing what makes the web awesome. The characteristics of these apps (as defined in the post) are:

- Responsive: to fit any form factor
- Connectivity independent: Progressively-enhanced with Service Workers to let them work offline
- App-like-interactions: Adopt a Shell + Content application model to create appy navigations & interactions
- Fresh: Transparently always up-to-date thanks to the Service Worker update process
- Safe: Served via TLS (a Service Worker requirement) to prevent snooping
- Discoverable: Are identifiable as “applications” thanks to W3C Manifests and Service Worker registration scope allowing search engines to find them
- Re-engageable: Can access the re-engagement UIs of the OS; e.g. Push Notifications
- Installable in mobile: to the home screen through browser-provided prompts, allowing users to “keep” apps they find most useful without the hassle of an app store
- Linkable: meaning they’re zero-friction, zero-install, and easy to share. The social power of URLs matters.

Note that none of these ideas involve implementing new technologies. They are all in the specification pipeline at W3C or WHATWG and have multiple browser implementations already in the market.

These technologies also don’t stop you from using the new, shinny and awesome stuff coming down in CSS, Javascript and related APIs and technologies. Nothing stops you from using WebGL 2.0, CSS Grids and other awesomeness now available or coming soon to your browsers.

We will also briefly explore what it would take to make PSPs into full desktop and mobile applications using Electron and Apache Cordoba / Adobe PhoneGap. Again this is not meant to be a perfect solution but an exploration of possibilities.

What is subcompact publishing

> It seems that perfection is attained, not when there is nothing more to add, but when there is nothing more to take away.
> Antoine de Saint Exupéry

The term Subcompact Publishing was coined by Craig Mod to describe a new and different publishing methodology rooted in the digital world rather than an extension of traditional publishing methods and systems.

> According to Mod:
>
> - Subcompact Publishing tools are first and foremost straightforward and require few to no instructions. Compare this to the instructions on how to navigate the current crop of digital magazines
>
> - The editorial and design decisions around them react to digital as a distribution and consumption space. We no longer buy print magazines but read them online. How can we leverage the online publishing and reading experiences?
>
> - They are the result of dumping our publishing related technology on a table and asking ourselves — what are the core tools we can build with all this stuff? Don't think of online as just an extension of print but explore what things you can do only online and how that enhances the reader's experience

Furthermore Craig describes subcompact publications as having the following characteristics:

- **Small issue sizes (3-7 articles / issue)**
- **Small file sizes**
- **Digital-aware subscription prices**
- **Fluid publishing schedule**
- **Scroll (don't paginate)**
- **Clear navigation**
- **HTML(ish) based**
- **Touching the open web**

Reading the essay it shows that it's geared towards magazines but, with a few modifications, it applies equally to books and other long form content. For this project, geared towards books and other collection types of publications, I've changed some of the definitions of Subcompact Publishing as listed below:

- **Small issue sizes (3-7 articles / issue) / Small file sizes** Because we are using technologies that allow us to load content on demand and to cache the content on the user's browser the need to keep the content small, both issue size and file size becomes less relevant. We can load the shell of our book independently of the content and load the content in smaller bites. For example we can load the first 10 chapters of a book right away and then load the rest of the content on demand. This does not mean we should forget about best practices in compressing and delivering the content but with Service Workers and caching available we can worry more about the content itself rather than how it's delivered. If we add http2 and server push to the mix the speed gain becomes significant if implemented correctly

- **Fluid publishing schedule** Because we can update the content of our web publications whenever it's necessary we can push new or updated content at any point, without having to worry about releasing the entire package again or having to go through a vendor's store approval process

- **Scroll (don't paginate)** Unless we have a compelling reason

- **Clear navigation** We have trained our users to accept certain metaphors for navigating our web applications. There is no compelling reason to change that now and, if there is, it better be a very good reason

- **HTML based** Here is the main divergent point from Craig's conception of subcompact publications. PSPs are meant for the web and, if the developer chooses, for HTML-based publishing formats. iBooks and, especially, Kindle are closed ecosystems where it's very difficult to get in to the ecosystem beyond using the tools they provide to adapt your format to their specifications... It is already hard enough to work with different browsers and the uneven CSS support... There is one browser among the major vendors that supports epub natively (thank you, Edge)

- **Using the open web** One of the biggest draws of the web is that it requires no installation proccess or approval for content delivery to the end users. Leveraging this makes the idea of Progressive Subcompact publications easier to work with, even if DRM and other rights management issues are not tackled from the start

This is how a progressive web application looks like. It may also be how our web reading experiences look like in the not too distant future

Progressive
web
application
- Taken
from
Google's
Using
App
Install
Banners

How does this all work?

At the core of our progressive subcompact publications is a service worker. This woker is a type of shared worker and it also works as a network proxy for your requests, you can cache responses, provide new responses based on the request you make, provide the basic mechanism to do push notifications and content synchronization in the background.

We'll break down the service worker in two sections: the script and the installation script you add to your entry point (usually `index.html`)

Service worker: The script

This is a fairly common pattern to build a service worker that will perform the following tasks:

- Caches the content of our application shell
- Automatically cleans up old cached content when the service worker is updated
- Fetches app resources using a 'cache first strategy'. If the content requested is in the cache then serve it from there. If it's not on the cache then make a network request for the resource, serve it to the user and put it in the cache for later requests

```
var CACHE_NAME = 'my-site-cache-v1';
var cacheWhitelist = [CACHE_NAME];
var urlsToCache = [
    '/',
    '/styles/main.css',
    '/script'/'in.js',
```

```
    'images/banner.png'
];

self.addEventListener('install', function(event) {
    event.waitUntil(
    caches.open(CACHE_NAME)
      .the'images/banner.png'
        console.log('Opened cache');
        return cache.addAll(urlsToCache);
      })
  );
});

self.addEventListener('activate', function(event) {
  event.waitUntil(
    caches.keys().then(function(cacheNames) {
      return Promise.all(
        cacheNames.map(function(cacheName) {
          if (cacheWhit'Opened cache'cacheName) === -1) {
            return caches.delete(cacheName);
          }
          })
           );
      })
    );
});

self.addEventListener('fetch', function(event) {
  event.respondWith(
  caches.match(event.request)
  .then(function(response) {
    if (response) {
      return response;
    }

    return fetch(event.request)
      .then(function(response) {
        'fetch'// Check if we received a valid response
```

```
        if (!(response)) {
          throw Error('unable to retrieve file');
        }

      var responseToCache = response.clone();
      caches.open(CACHE_NAME)
        .then(function (cache) {
            cache.put(event.request, responseToCache);
          });
        return response;
      })

      .catch(function(error) {
        console.log('[Service Worker] unable to complete the request: ', e
      });
    })
  );
});
```

There are things that we are not covering on purpose for the sake of keeping the code short. Some of these things include:

- For this example we've assumed a minimal set of elements to cache for the application shell. We can be more detailed and add fonts and other static resources. We may also assign the array of items to cache on install to a variable to make it easier to work with
- Providing a solution for when the content is not in the cache and the network is not available. We can cache default feedback for text-based content or programmatically generate an svg image for fallback
- Putting content in different caches so deleting one group of resources doesn't delete all of them
- We make no effort to add hashes to the resources we cache so we can do proper HTTP cache busting when needed

But at 60 lines of Javascript that will work in 3 of the 5 major browsers (and soon in all of them) I think it does a pretty good job.

Service worker: The registration

Assuming that we saved the service worker as `sw.js` we can write the code below inside a script tag on our entry page (`index.html`).

```
if ('serviceWorker' in navigator) {
  console.log('Service Worker is supported');
  navigator.serviceWorker.register('sw.js').then(function(reg) {
    console.log('Yay!', reg);
  }).catch(function(err) {
    console.log('boo!', err);
  });
}
```

This script checks for service worker support by testing if the string `serviceWorker` exists in the navigator object. If it does then service workers are supported, we log a message to the console and then register the serviceworker.

If the serviceWorker string doesn't exist in the navigator object then service workers are not supported. The catch statement will trigger and we log something to the console.

That's it. The combination of those two scripts gives us consistent performance across devices and the possibility of work offline after accessing the content once while online.

Service Worker: Push Notifications

Using Push notifications we can communicate events and new information to the user through the Operating System's push notification system and UI.

A good introduction to Push notiications is the Google Developers article: Push Notifications: Timely, Relevant, and Precise. It provids a context for how and when to use push notifications.

Google provides Firebase Cloud Messaging as the server component of push notifications. It makes sense since it's a Google product but, in my opinion, it

makes it harder to streamline the development process (particularly in trying to figure out where they placed the ID that you need to configure push messaging and notifications).

There are other alternatives I'm currently researching.

Background Sync

If you write an email, instant message, or simply favourite a tweet, the application needs to communicate that data to the server. If that fails, either due to user connectivity, service availability or anything in-between, the app can store that action in some kind of 'outbox' for retry later.

Unfortunately, on the web, that outbox can only be processed while the site is displayed in a browsing context. This is particularly problematic on mobile, where browsing contexts are frequently shut down to free memory.

This API provides a web equivalent to native application platforms' job scheduling APIs that enable developers to collaborate with the system to ensure low power usage and background-driven processing. The web platform needs capabilities like this too.

We need to check if the user is online or not. For this we use `navigator.online` to test the device's connectivity status.

This is a basic test but it's not foolproof. A flase result will always be false but a positive result will not always be true as the user may be caught in 'lie-fi', a status where the devices reports connectivity but there is no real network to access.

```
// Connection Status
  function isOnline() {
    var connectionStatus = document.getElementById('connectionStatus');
    if (navigator.onLine){
      connectionStatus.innerHTML = 'You are currently online!';
    }
    else {
      connectionStatus.innerHTML = 'You are currently offline. ' +
      'Any requests made will be queued and synced as soon as you '+
      'are connected again.';
```

```
  }
}
window.addEventListener('online', isOnline);
window.addEventListener('offline', isOnline);
isOnline();
```

When we register the Service Worker we add a click event handled to take care of sync registration. This event will activate the sync registrration and return a promise that will resolve when the sync has reistered.

```
if ('serviceWorker' in navigator) {
  navigator.serviceWorker.register('./sw.js')
.then((r'./sw.js'on) => {
  return navigator.serviceWorker.ready;
}).then((registration) => {
  // register sync
  document.getElementById('requestButton')
    .addEventListener('click', () => {
    registration.sync.register('image-fetch').then(() => {
      console.log('sync registered');
    }).catch(function(error){
        conso'requestButton'to fetch image.');
      });
    });
  }).catch(function(error){
  console.log('Unable to register Service Worker.');
  });
}
else{
  console.log('Service Worker functionality not supported.');
}
```

In most situations we can get away with using a build tool to generate the Service Worker. Workbox provides tutorials and code examples for Webpack, Gulp and using NPM scripts to generate the Service Worker. An example using gulp is shown below:

```
gulp.task('bundle-sw', () => {
  return wbBuild.generateSW({
   globDirectory: './_site/',
    swDes'./_site/'e/sw.js',
    staticFileGlobs: ['**\/*.{html,js,css}'],
    globIgnores: ['sw.js'],
    skipWaiting: true, '**\/*.{html,js,css}'// optional
    clientsClaim: true, // optional {
    console.log('Service worker generated.');
  })
  .catch((err) => {
    console.log('[ERROR] ', err);
  });
})
'Service worker generated.'
```

In the future we will be able to create periodic sync calls like this:

```
navigator.serviceWorker.ready.then(function(registration) {
  registration.periodicSync.register({
    tag: 'get-latest-news',         // default: ''
    minPeriod: 12 * 60 * 60 * 1000, // default: 0
    powerState: 'avoid-draining',   // default: 'auto'
    networkState: 'avoid-cellular'  // default: 'online'
  }).then(function(periodicSyncReg) {
    // success
  }, function() {
    // failure
  })
});
```

This will initiate a background sync every 12 hours but only if the

Service Worker Tooling (the old way)

Doing it by hand is fun and teaches you a lot about the inner workings of service workers but having to update the files you want to cache and how to define the

routes you want to use to cache your content.

sw-precache is a Google tool developed to atuomate creation of service workers with application shell caching on installation. The tool can be used from command line or as part of a build system (Grunt, Gulp and others).

It will also take care of importing additional scripts to use sw-toolbox (described in the next section).

A gulpfile.js using sw-precache looks like this:

```
// Assigning modules to local constants
var gulp = require('gulp');
// Required for sw-precache
va'gulp' = // Required for sw-precacheache = require('sw-precache');
// Array of paths. Currently only uses the src to represent the path to so
var paths = {
    src: './'
};

gulp.task('sw-precache'er'// Array of paths. Currently only uses the src t
  staticFileGlobs: [
    paths.src + 'index.html',
    paths.src + 'js/main.js',
    paths.src + 'css/main.css',
    paths.src + 'images'index.html'/**/*'

    ],
    importScripts: [
      'node_modules/sw-toolbox/sw-toolbox.js',
      paths.src + 'js/toolbox-scripts.js'
    ],
    stripPrefix: paths.src
  }, callback);
});
```

sw-toolbox automates dynamic caching for your service worker. It creates customizable routes for your caching and provides for express-like or regular-expression-based routes to match routes and resources.

In the gulpfile.js above the `importScripts` section imports two files:

- sw-toolbox.js is the library that will run the custom routes
- toolbox-scripts contains our custom toolbox routing

The script itself is wrapped in an immediately-invoked function expression (IIFE) to keep our code from polluting the global namespace. Inside the IIFE we work with different routes.

All these routes use the get HTTP verb to represent the action the router will take.

The toolkbox then takes a pattern to match the route against and a cache strategy.

There is an optional cache object that contains additional parameters for the cache like (cache) name, maximum number of entries (maxEntries) and maximum duration of the cache in seconds.

The `toolbox-scripts.js` looks like this:

```
(function(global) {
'use strict';

// The route for any requests from the googleapis origin
global.toolbox.router.get('/(.*)', global.toolbox.cacheFirst, {
  cache: {
    name: 'googleapis',
    maxEntries: 20'/(.*)'  origin: /\.googleapis\.com$/
});

// We want no more than 50 images in the cache.
// We use a cache first strategy
global.to/\.googleapis\.com$/// We want no more than 50 images in the cach
  cache: {
    name: 'images-cache',
    maxEntries: 50
  }
});
```

```
// pull html content using network first
global.addEventListner('fetch', function(event) {
  if (event.request.headers.get('accept').includes('text/html')) {
    event.respondWith(toolbox.networkFirst(event.request));
  }

  // you can add additional synchronous checks
  // based on event.request.
});

// pull video using network only.
global.t'images-cache'get('(.+)', global.toolbox.net'(.+)'// pull html co
});

// the default route is global and uses cacheFirst
global.toolbox.router.get('/*', g'/*'// the default route is global and u
})(self);
```

Registering the automatically generated service worker is no different than registering the manually generated script. Assuming that we saved the service worker as `sw.js` the registration code in our entry page (`index.html`) looks like this:

```
if ('serviceWorker' in navigator) {
 console.log('Service Worker is supported');
 navigator.serviceWorker.register('sw.js')
 .then(function(reg) {
   console.log('Yay!', reg);
 }).catch(function(err) {
   console.log('boo!', err);
 });
}
```

Service Worker Tooling (the new way)

Since I first wrote this essay, Google has updated the Service Worker librariess and consolidated them into one library called Workbox and it has made all the separate libraries into modules of the larger workbox tool.

To achieve functionality equivalent to the old precache + sw-toolbox libraries we have to take a two step process:

First we write the Service Worker with all the routes that we need and a placeholder for the array of items we want to precache.

In the build file (gulpfile in this case) we add (or modify) the service worker build task will determine what files to precache and insert those in to the `workboxSW.precache` array that we left empty when creating the Service Worker.

The Service Worker

The service worker does a few things.

1. It imports the `workbox-sw` script using `importScripts`
2. It sets up a constant to use when we reference the library
3. It creates a place holder for the array of items to precache
4. it creates and register routes for different items.

```
// Alternatively, use your own local copy of workbox-sw
importScripts('https://unpkg.com/workbox-sw@0.0.1');// 1

const workb'https://unpkg.com/workbox-sw@0.0.1's // 1n empty array for ou
// Gulp will populate the array when we build the project
workboxSW.precache([]); // 3

// All navigation requests should be routed to the App Shell.
// since we're not using app shell it's commented out
// workboxSW.router.'re not using app shell it');

// Use a cache first strategy for files from googleapis.com
```

```
workboxSW.router.registerRoute( // 4
  new RegExp('.googleapis.com$'),
  workboxSW.strategies.cacheFirst({
    cacheName: 'googleapis',
    cacheExpiration: {
      // Expire after 3 days (expressed in seconds)
      maxAgeSeconds: 3 * 24 * 60 * 60
    }
  })
);

// Use a cache-first strategy for the images
workboxSW.router.registerRoute(
  new RegExp('/.(?:png|gif|jpg)$/'),
  workboxSW.strategies.cacheFirst('.googleapis.com$'// Use a cache first s
      maxAgeSeconds: 30 * 24 * 60 * 60
    },
    // The images are returned as opaque responses,
    // with a status of 0.
    // Normally these wouldn't be cached;
    // here we opt-in to caching them.
    // If the image returns a satus 200 we cache it too
    cacheableResponse: {statuses: [0, 200]}
  })
);

// Match all .html and .html files use cacheFirst
// cache the resources for 24 hours
workboxSW.router.registerRoute(
  new RegExp('/.html$|.htm$/'),
  workboxSW.strategies.cacheFirst({
    cacheName: 'content',
    cacheExpiration: {
      maxAgeSeconds: 1 * 24 * 60 * 60
    }
  })
);
```

```
// For video we use a network only strategy.
// We don't want to clog the cache with large video files
workboxSW.router.registerRoute(
  new RegExp('/.(?:youtube|vimeo).com$/'),
  workboxSW.strategies.networkOnly()
);

// The default route uses a cache first strategy
workboxSW.router.registerRoute('/*',
  workboxSW.strategies.cacheFirst()
);
'/.html$|.htm$/'// maximum 50 entriese('/*',
  workboxSW.strategies.cacheFirst()
);
```

The build file

The new `service-worker` task changes from generating the full service worker to only injecting the manifest into the actual service worker file.

```
gulp.task('service-worker', () => {
  return workboxBuild.injectManifest({
    swSrc: 'src/service-worker.js',
 'src/service-worker.js'ce-worker.js',
    globDirectory: '_site',
    staticFileGlobs: [
      'rev/js'_site'/**/      'rev/styles/*.css''rev/styles/*.css'/*.css'
      'images/**/*'
    ]
  });
});
```

What can we do beyond offline?

The service worker script we discussed in the prior section is the core of a PSP but there's way more we can do to make our reading experiences behave more like native applications. There are more things we can do that may or may not be fully

related to Service Workers but they do gives us additional tools in the arsenal to work in crafting our reading experiences.

While not directly related to service workers this feature may help get better re-engagement from your users:

- Installation in mobile home screens

Also not directly related to progressive web applications, we can also preserve data, not just content on our web applications using

- IndexedDB

We'll discuss them in the sections below.

Installation in mobile home screens

Using the W3C App Manifest specification and the existing metatags for adding an app to the homescreen in mobile devices we enable our users to add our web content to the homescreen of mobile devices to foster a higher level of interaction and reengagement with the content.

It's next to impossible to remember all the items you can include in your manifest. Rather than go through tutorials for the reduced set required for Android's add to homescreen (documented in Google's web fundamentals we can use tools like Manifestation (either as a Node Package or web based) to generate a complete manifest for our applicaiton. The Node version can also be used as part of a Gulp/Grunt build system.

HTML5 Doctor has a good an up to date reference on App Manifest. Another source of information is the Mozilla Developer Network article on Web App Manifest.

Expect a deeper dive on Web Application Manifest some time in December.

IndexedDB

We've had client-side storage solutions for a while now. Sessions Storage, WebSQL and IndexedDB. Until recently they had no uniform support among brosers and one (WebSQL) is no longer being developed because all the implementations relied on SQLite as the backend and this was considered to violate the "two

interoperable implementations" requirements for W3C specs.

I've chosen IndexedDB as the engine to store data for my offline applications because, as complicated as the API is work with, there are wrapper libraries that make the work easier and will work across browsers, even Safari (which has a deserved reputation for shitty IndexDB implementations).

Knowing how much of a pain it can be to write bare IndexedDB code, I've picked Dexie as my wrapper library. It is easy to use and, for browsers who have issues with indexedDB like Safari, provide a transparent fallback to WebSQL. It also uses promises rather than callbacks and, once you start working with promises, you will never go back to callbacks :-)

The example below shows how to create a database and a store for the a theoretical friends datastore.

```
var db = new Dexie("friends");

// Define a schema
db.version(1).stores({
  friends: 'name, age'
});
'name, age'
```

We then open the database

```
// Open the database
db.open()
  .then(() => {
    console.log('database open');
  })
  .catch(error => {
    alert('Uh oh : ', error);
  });
```

We can then insert records into the datastore one at a time or using a transaction.

Transactions group one or more actions into an atomic unit. If any of the actions composing a transaction fails then the entire transaction fails and the

datasore is rolled back to the state before the transaction began.

```
// Insert data into the database
db.friends.add({
  name: 'Camilla', age: 25
});

'Camilla'// Insert data into database using transactions populateSomeData
  return db.transaction("rw", db.friends, () => {
    db.friends.clear();
    db.friends.add({ name: "David", age: 48 });
    db.friends.add({ name: "Ylva", age: "rw");
    db.friends.add({ name: "Jon", age: 76 });
    db.friends.add({ name: "Måns", age: 56 });
    db.friends.add({ name: "Daniel", age: 55 });
    db.friends.add({ name: "Nils", age: 42 });
    db.friends.add({ name: "Zlatan", age: 21 });

    // Log da"Jon"// Log data from DB:
    db.friends.orderBy('name').each(friend => {
        log(JSON.stringify(friend));
    });
  })
  .catch(e => {
    log(e, "error");
  });
}
```

We can then retrieve data from the store usning queries similar to the SQL syntax. An example of this query retrieves all the names from the data store where the age is over (above) 35 and then display the names.

```
// Query friends datastore
db.friends
  .where('age')
  .above(35)
  .each (function (friend) {
```

```
    console.log (friend.name);
  });
```

There may be occasions when we need to delete the database, maybe because we don't need it again or maybe because we screwed up and want to start over.

```
db.delete().then(function() {
    console.log("Database successfully deleted");
}).catch(function (err) {
    console.error("Could not delete database");
}).finally(function() {
    "Could not delete database"// Do what should be done next...
});
```

This is a very broad and quick overview of Dexie. If you want more information check the Dexie.js Tutorial to get started.

There are other wrapper libraries for IndexedDB but Dexie is the most flexible and forgiving one for me.

Other fun things we can do

Annotations

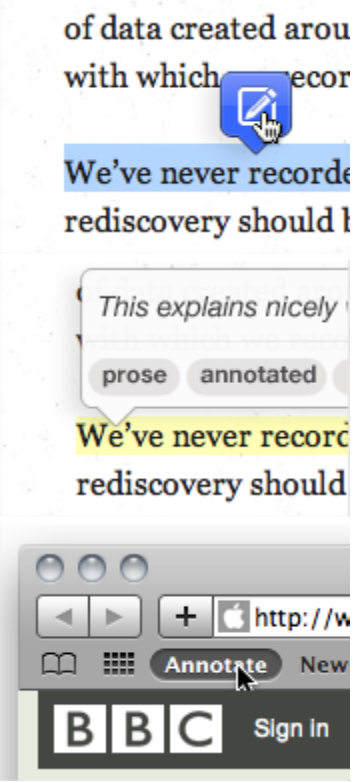
I still remember the first time I made an annotation from a Kindle book available in their public site (kindle.amazon.com). I saw the possibilities as limitless until I realized that there were limitless as long as you bought the book from Amazon and read it on a Kindle device or application.

Every time I've turned around and searched for some way to annotate the web I've come with these two solutions but I've never had a project they work well with. I think PSPs are the perfect place to put this in practice. There are two libraries I think are particularly appropriate: Emphasis and annotator.js which provide different ways to make and share annotations from your PSPs.

Emphasis provides dynamic paragraph-specific anchor links and the ability to highlight text in a document, it makes the information about the highlighted elements of the page available in the URL hash so it can be emailed, bookmarked,

or shared.

Annotator provides a more traditional annotation interface that is closer in spirit to the Kindle annotation UI that attracted me to the concept when I first saw it.



Another tool that sounds interesting is MIT's Annotation Studio but it seems to be geared towards MIT Hyperstudio's larger project and not necessarily ready as a standlone solution, that said, your milleage may vary.

The thing to consider if how these annotation tools store the annotations. Do they use server-side databases? If so how do we cache new annotations when the reader is offline? Google Analytics provides a possible example where we store the annotations in indexedDB and then play them back when the user goes online.

Structuring and styling our content

We want our content to look awesome regardless of the device. How do we accomplish this?

A good starting point is Ethan Marcotte's Responsive Web Design (or whatever Responsive Web Design book is your favorite). We want the experience to scale to whatever device or platform we're targeting and, whatever design we choose, the

first thing we need to make sure of is that it'll work well in phones, tablets and desktops (in other words, everywhere there is a browser).

Once we have a layout we can start thinking about, what to me is, the most important part of any long form project: typography. I've written extensively on what typography is and how it works on the web now we need to take the next steps.

This is where the first set of choices happen: What layout do we choose? How do we create something engaging without becoming repetitive? How do we craft a reading experience that matches the content?

I first saw Jen's presentation at SFHTML5. I see it as a challenge and an opportunity to think differently about the way we create and layout our content on the web. For longer form content this also speaks to letting the content dictate the layout and not the other way around. What is it that makes magazine layouts so interesting?

I collect electronic versions of GQ, Wired, Vanity Fair, Fast Company and Harvard Business Review and the biggest question when I read them is how can we make this reading experience in the open web? The ads from magazines are what intrigue me the most... and where a lot of my most radical ideas come from.

After watching this presentation from Beyond Tellerand I couldn't help reading the new edition of Hardboiled Web Design. Clarke advocates that creativity should be at the center of our online design work... It speaks to the need of art directed web design and bespoke designs rather than using the same design over and over.

If we drop the book metahphor from our online reading experiences, There is no limit to what we can do with our online publications. We need to go back to our content and see how we can enrich it and what technologies we can use to do so... we have a lot of layout tools that, a few years ago, were only possible in InDesign and other Desktop Publishing Tools or took a lot of extra workarounds to do in CSS/HTML/JavaScript.

Now we need to get out our collective comfort zone and challenge both ourselves and our future readers with layouts that go beyond what we see on the web today.

One last example of what we can do with our new css tools and how much we can be true to our creative selves without having to lie to our web developer selves: Justin McDowell uses new CSS technologies to recreate works from the Bauhaus school.

How have reader expectations changed? (UI for the performance obsessed)

As consumers of information we've all become more demanding. We want the content faster and we want to be engaged with the content we access online. PSPs should be no different in their performance than traditional web applications.

Rather than provide a one-size-fits-all solution I present some of the data Nielsen first articulated in 1993's *Usability Engineering* We'll use these values to draw some basic conclusions that will start the thinking about performance and how PSPs can work towards achieving those performance goals.

- **0.1 seconds** — Operations that are completed in 100ms or fewer will feel instantaneous to the user. This is the gold standard that you should aim for when optimising your websites.
- **1 second** — Operations that take 1 second to finish are generally OK, but the user will feel the pause. If all of your operations take 1 second to complete, your website may feel a little sluggish.
- **10 seconds** — If an operation takes 10 seconds or more to complete, you'll struggle to maintain the user's attention. They may switch over to a new tab, or give up on your website completely. Of course this depends on what operation is being completed. For example, users are more likely to stick around if they've just submitted their card details in the checkout than if

they're waiting to load a product page.
- **16 milliseconds** — Given a screen that is updating 60 times per second, this window represents the time to get a single frame to the screen (1000 ÷ 60 = ~16). People are exceptionally good at tracking motion, and they dislike it when their expectation of motion isn't met, either through variable frame rates or periodic halting.

We want to get to the site's first meaningful paint on initial load in as close to 1000 milliseconds as possible.

Animations should take no more than 16 milliseconds in order to reach 60 frames a second.

Performance Optimizations

We can optimize our resources so that time to first meaningful interaction is as short as possible. First load will also cache the resources needed for our application shell and then dynamically . We want to optimize this to last as little as possible.

This is not just speech for the sake of speach. Take the following graphic (from Soasta's Page bloat update: The average web page is more than 2 MB in size). How can we minimize the number of resources to load and cache the first time we access a page? How many of these resources can be reused on pages accross the site? How can we optimize images to reduce their size?

We have gotten lazy or, possibly, made the wrong assumptions. The graphic below, also from Soasta's blog post, show how the size of our web content has changed over the years... and it shows no signs of decreasing.

I've added an image compression step to my build file based gulp-imagemin to make sure that I reduce the size of the images on my apps and sites. The Gulp plugin comes bundled with the following **losless** compression tools

- gifsicle — Compress GIF images
- jpegtran — Compress JPEG images
- optipng — Compress PNG images
- svgo — Compress SVG images

The actual Gulp task looks is this:

```
gulp.task('imagemin', function() {
  return gulp.src('src/images/original'src/images/originals/**'({
      progressive: true,
      svgoPlugins: [
        {removeViewBox: false},
        {cleanupIDs: false}
      ],
      use: [mozjpeg()]
    }))
    .pipe(gulp.dest('src/images'))
    .pipe($.size({
      pretty: true,
      title: 'imagemin'
    }));
});
```

As usual, your mileage may vary.

How are you serving your content?

PSPs do not avoid performance requirements... however the questions are slightly different. Are you serving your content with HTTP 1.x or HTTP2? When I first heard the question asked I laughed... why should this matter?

HTTP2 has changed the way we work with web content. What we used to consider as patterns and best practices are no longer necessay and may be considered anti patterns.

Because of the way HTTP2 serves the content it is not necessary to concatenate our CSS and Javascript, it actually work better if you work with individual minimized files; HTTP2:

- is binary, instead of textual: Binary protocols are more efficient to parse, more compact “on the wire”, and most importantly, they are much less error-prone, compared to textual protocols like HTTP/1.x, because they often have a number of affordances to “help” with things like whitespace handling, capitalization, line endings, blank lines and so on

- is fully multiplexed: Multiplexing addresses head-of-line-blocking by allowing multiple request and response messages to be in flight at the same time; it's even possible to intermingle parts of one message with another on the wire
- can use one connection for parallelism: One application opening many connections simultaneously breaks a lot of the assumptions that TCP was built upon; since each connection will start a flood of data in the response, there's a real risk that buffers in the intervening network will overflow, causing a congestion event and retransmits
- uses header compression to reduce overhead: If you assume that a page has about 80 assets and each request has 1400 bytes of headers, it takes at least 7-8 round trips to get the headers out "on the wire". In comparison, even mild compression on headers allows those requests to get onto the wire within one round trip – perhaps even one packet.
- allows servers to "push" responses proactively into client caches: When the server pushes the content that the client will need before it needs it we can save networks requests by caching the content in the browser (HTTP) cache

This doesn't mean we shouldn't minimize and compress resources but bundling them together is less important now that we don't have to be concerned with saturating the connection to the server and can leverage HTTP2 push to let the server prefetch content to the client.

Improving performance is a never ending game. Ilya Grigorik illustrates this point in his presentation from I/O 2016.

What can we build

So far we've discussed how to build a progressive subcompact publications. We have a good technology stack of APIs and technologies that make our publications responsive so we only need one codebase for multiple devices, have diverse capabilities such as data visualizations, 2D and 3D virtual reality and many others.

At the same time we can control more and more of the intangibles and the tangibles outlined by Massimo Vignelly in the Vignelli Cannon.

The Intangibles	The Tangibles
▪ Semantics ▪ Syntactics ▪ Pragmatics ▪ Discipline ▪ Appropriateness ▪ Ambiguity ▪ Design is One ▪ Visual Power ▪ Intellectual Elegance ▪ Timelessness ▪ Responsibility ▪ Equity	▪ Paper Sizes ▪ Grids, Margins, Columns and Modules ▪ A Company Letterhead ▪ Grids for Books ▪ Typefaces, The Basic Ones ▪ Flush left, centered, justified ▪ Type Size Relationships ▪ Rulers ▪ Contrasting Type Sizes ▪ Scale ▪ Texture ▪ Color ▪ Layouts ▪ Sequence ▪ Binding ▪ Indentity and Diversity ▪ White Space

Those that we can't control we can work around, for example rather than worrying about paper size we can worry about different layouts depending on the screen size available to you.

This doesn't mean we want the same layout for the web that we want for print. We're closer to it with the introduction of CSS Grid in early 2017 and creative uses of existing content like those in Jen Simmons' Lab but we're not completely there yet.

Now that we've gotten an idea of where we're going we'll discuss what we can do and provide ideas and examples of what we can use PSPs for.

Resilient Web Design

Jeremy Keith's Resilient Web Design provides a basic idea of what we can do with PSP and PSP-like applications can do. We go to the website and, if it support service workers, will pre-cache the content of the site (the ecomplete HTML content and the associated CSS And Javascript) and store it in the browser so we can view the content regardless of network connectivity.

The book is responsive so it'll work regardless of what form factor you use to view it and, in mobile devices, you can save it to the home screen in a mobile device or a shelf in Chrome for desktop.

Links, resources, patterns and ideas

- Responsive Web Design
  - Responsive Web Design — Ethan Marcotte, A Book Apart
  - Responsible Responsive Design — Scott Jehl, A Book Apart
  - Going Responsive — Karen McGrane, A Book Apart
  - Responsive Design: Patterns and Principles — Ethan Marcotte, A Book Apart
  - Design For Real Life — Eric Mayer & Sara Wachter-Boettcher, A Book Apart
  - Inclusive Design Patterns — Haydon Pickering, Smashing Books
- Service Workers
  - caniuse.com support matrix
  - Specification
  - Is ServiceWorker ready? — Jake Archibald
  - Service Workers: an Introduction — Web Fundamentals
  - Service Worker API — MDN
  - Support by browsers
    - MS Edge
    - WebKit - Safari
    - Firefox
    - Chrome / Opera
  - Background Sync
    - Background synchronization explained
- Accessibility
  - A11y Project
  - Chrome Accessibility Developer Tools
  - A11y accessibility checker
  - Introduction to web accessibility course — Google
  - The WAI Forward — Heydon Pickering, Smashing Magazine
  - Colour Contrast Analyser— The Paciello Group
  - Accessibility Testing Tools Updated — The Paciello Group
- Service Worker related
  - Web Push Notifications
    - Web Push Notifications: Timely, Relevant, and Precise — Web Fundamentals
    - What Makes a Good Notification?

        - Video: Web Push Notifications — Google I/O 2016
        - Bringing Push Notifications to the Mobile Web — @scale
        - Deep Engagment with Push Notifications — Progressive Web App Summit 2016
    - Installation on mobile home screen
        - Add to homescreen — Android
        - Web App Install Banners — Chrome for Android
        - Increasing Engagement with Web App Install Banners — Chrome for Android
        - Configuring Web Applications — iOS
        - Web manifest specification — HTML5 Doctor
- Peformance
    - General Readings
        - High Performance Browser Networking — Ilya Grigorik, O'Reilly
        - Web Performance Warrior: The business of speed - Andy Still, O'Reilly
    - Performance Patterns
        - Introducing RAIL: A User-Centric Model For Performance
        - Serve your content (introduces the PRPL pattern)
    - HTTP2
        - High Performance Browser Networking, chapter 12 — Ilya Grigorik, O'Reilly
        - Are you ready for HTTP/2 Server Push?! — Akhil Jayaprakash
        - Innovating with HTTP2 server push — Ilya Grigorik
- Readings
    - Performance: Showing Versus Telling — Lara Hogan
    - An Introduction to perceived performance — Matt West
    - Response Times: The 3 Important Limits — Jakob Nielsen
    - Powers of 10: Time Scales in User Experience — Jakob Nielsen
    - Response time in man-computer conversational transactions — Robert Miller
    - The information visualizer: An information workspace — Card, Robertson and Mackinlay
    - A Beginner's Guide to Perceived Performance: 4 Ways to Make Your Mobile Site Feel Like a Native App — Kyle Peatt
    - A Study on Tolerable Waiting Time: How long Are Web Users Willing to Wait?— Fiona Fui-Hoon Nah
    - Interaction in 4-Second Bursts: The Fragmented Nature of Attentional Resources in Mobile HCI— Antti Oulasvirta, Sakari Tamminen, Virpi Roto, and Jaana Kuorelaht

- Quantifying Interactive User Experience on Thin Clients— Niraj Tolia, David G. Andersen, and M. Satyanarayanan
- Characterizing Web Use on Smartphones— Chad C. Tossell, Philip Kortum, Ahmad Rahmati, Clayton Shepard, Lin Zhong
- Playing With Tactile Feedback Latency in Touchscreen Interaction: Two Approaches— Topi Kaaresoja, Eve Hoggan, Emilia Anttila