PWA starter kit

Last year I worked at Google creating ILT curriculum for Progressive Web Applications. It's a great idea and I think the final product worked well, but it's not complete. If you're getting started you need hand holding but what if you've built PWAs before and want a reference or examples that will do what you want so you can either modify them for your project or copy it as is.

I'm making assumpttions that you're already familiar with the technologies that make up a progressive web application so I won't delve too much in the details about what they are. I'm also assuming that you've already created your build system, minimize your scripts and style sheets and other performance optimizations.

Also note that **we're only working on the technical side of a PWA**. When we look at the PWA Checklist we'll see that there are other aspects to a good and exemplary PWA than what we cover in detail below.

If you're interested in learning more about PWAs you can check the manual my team wrote for PWA concepts.

Quick Recap

Frances Berriman and Alex Rusell coined the term **progressive web application** in 2015 to describe a set of technologies that enable web content (sites or applications) to behave more like native mobile apps without loosing the advantages of the web. According to the Alex these applications would work like this:

> 1. The site begins life as a regular tab. It doesn't have super-powers, but it is built using Progressive App features

> including TLS, Service Workers, Manifests, and Responsive Design.
> 2. The second (or third or fourth) time one visits the site — roughly at the point where the browser it sure it's something you use frequently — a prompt is shown by the browser (populated from the Manifest details)
> 3. Users can decide to keep apps to the home screen or app launcher
> 4. When launched from the home screen, these apps blend into their environment; they're top-level, full-screen, and work offline. Of course, they worked offline after step 1, but now the implicit contract of "appyness" makes that clear.
>
> Alex Russell: Progressive Web Apps: Escaping Tabs Without Losing Our Soul

For sites or apps only?

Naming progressive web applications can be a little tricky. Do the technologies only apply to web applications or can we use the technologies when building web sites?

I think that we don't need to make the distinction. These technologies will work just as well with websites when used properly. Granted there are new technologies in the web stack that are more appropriate for applications than sites (thinking about the payment API)

As Aaron Gustafson points out:

> "Web apps" in this context can be any website type—a newspapers, games, books, shopping sites—it really doesn't matter what the content or purpose of the website is, the "web app" moniker is applicable to all of them. The term could just have easily been progressive web site and it may be helpful to think of it as such. It doesn't need to be a single page app. You don't need to be running everything client side. There are no particular requirements for the type of PWA you are developing.
>
> Aaron Gustafson — Progressive Web App And The

> Windows Ecosystem

I like to think that the PWA is a better way to build our web content that combine the best things of the web for an experience that works almost like a native application. We get ease of use and a technology stack that we're familiar and comfortable with (PWAs don't dictate the stack you use, only that they have a manifest and a service worker) and a rapidly growing set of APIs available.

See Aaron's full presentation from Microsoft Build to get a better idea of progressive enhancement in the context of a PWA:

Toolkit or Recipes?

PWAs are a toolkit that will give you the tools and methods to create your applications and sites. You're expected to write more code but you also decide the functionality

To avoid issues like those faced with App Cache service workers make no assumptions about how you will cache your content and what else you can and will do with the tools you have available.

In this post we'll talk about the technologies that make a basic PWA: web application manifest and service worker and discuss basic ways of configuring them.

Rememeber: HTTPS only!

For a PWA to work it must be served through HTTPS with a modern encryption schema. As we'll see in the service worker section they are powerful and any mistakes you make when creating them can have serious security repercusions. so TLS/HTTPS only.

Manifest

The first, and easier, aspect of a PWA is the manifest. This is a JSON file that holds information about the content and gives supporting browsers hints and instructions for installing the content in the user's homescreen.

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

The manifest itself is a JSON file that contanins information about the application. The sample manifest below contains the following information:

- name: Full name of the application
- short name: Short name of the application
- description
- icons: An array of available icons to use for different aspects of the application
- (default) orientation
- start_url: The entry point for the application
- display: how is the application UI presented to the user
- background_color while the application is loading
- text direction: rtl, ltr
- lang: default language

```
{
  "name": "BookReade"BookReader""short_name": "BookReader",
  "description": "An ebook reader application",
  "icons"   "src": "ima": "touch/homescreen48.png",
    "sizes": "": ""sizes""sizes": "image/png"
 "image/png""type""images/touch/homescreen72.png",
    "sizes": "72x72",
    "type": "i"images/touch/homescreen72.png""sizes": "72x72",
    "type"",
    "sizes": "144x144"": ""sizes""src"mage/png"
  }],

  "orientation": "portr": ""sizes"art_url": "index.": ""type": "image/png
  }],
```

```
  "orientation": "portrait",
  "start_url"or": "#fff",
  "dir": "ltr",
  "lang": "": ""display": "standalone",
  "background_color": "#fff",
  "dir": "ltr",
  "lang": "en-us",
}
```

Orientation may be one of the following values:

- any
- natural
- landscape
- portrait

There are additional values for orientation that I'm researching. I think they move the app to primary or secondary display but, as far as I understand them, only landscape and portrait are currently supported. The additional values are:

- landscape-primary
- landscape-secondary
- portrait-primary
- portrait-secondary

The display mode attribute controls how much of the browser's chrome and UI is show for your application. Each of the 4 modes in the table below falls back to the next one on the table, except for browser, which is the default.

Display Mode	Description	Fallback Display Mode
`fullscreen`	All of the available display area is used for content and none of the browser chrome is visible.	`standalone`
`standalone`	The application will look and feel like a standalone application. In this mode, the user agent will exclude UI elements for controlling navigation, but can include other UI elements such as a status bar.	`minimal-ui`
`minimal-ui`	The application will look and feel like a standalone	`browser`

Display Mode	Description	Fallback Display Mode
	application, but will have a minimal set of UI elements for controlling navigation. The included elements vary by browser.	
`browser`	This is the default mode. The application opens in a conventional browser tab or new window, depending on the browser and platform.	(None)

Splash Screen

Chrome 47 and later, display a splash screen for a web application launched from a home screen. This splashscreen is auto-generated using properties in the web app manifest, specifically: `name`, `background_color`, and the icon in the `icons` array that is closest to 128dpi for the device.

Even if it's not a PWA, the ability to save a web site or app to the homescreen greatly improves the user experience for the content.

Service Worker

> Note that this section will make heavy use of arrow functions, let and const, and other ES2015 and newer features.
>
> I've chosen to do this because I treat PWAs as progressive enhancements for evergreen browsers. Older browsers will not support PWAs and their features so it's pointless to try and support them.

The service worker has to main components: the registration and the service worker itself. We place the registration in the index.html file (or whatever we've named our site's entry point) and it'll tell the browser whether the browser supports service workers, where to find it and what to do when it registers and when it fails.

I normally inline this code in my `index.html`. I'm not 100% sure if this will work in a concatenated and minified file; it should but I haven't tried it yet.

```
if ('serviceWorker' in navigator) {
  window.addEventListener('load', () => {
   'load'ator.serviceWorker.register('/sw.js').then(function(registration
     '/sw.js'// Registration was successfulonsole.log('ServiceWorker reg
    }).catch(function(err) {
     // regis'ServiceWorker registration successful with scope: '// regis
     console.log('ServiceWorker registration failed: ', err);
    });
  });
}
```

If the registration is successful we now have a service worker installed for our site/app.

`sw.js` is the actual service worker script. Since the API is e vent based I've broken down the code by events. At the top of the script we do some housekeeping before getting started. We setup to caches (one for precached resources and one for resources we cache during the application's run).

Next we declare a list of resources to cache. In this list include the minimum necessaary for content to render when off line or in low connectivity but not too much so that it'll slow down the initial rendering of your content.

We will use the cache names troughout the service worker to reduce the amoount of typing we have to do. This example is simplified, for a production application you'd have to indicates additional style sheets and scripts to precache on the PRECACHE_URLS array.

```
const PRECACHE = 'precache-v1';
const RUNTIME = 'runtime-v1';

const PRECACHE_'runtime-v1'./', // Alias for index.html
  'styles/main.css',
  'scripts/main.js'
];
'styles/main.css'
```

The first event, `install`, sets up the caches and the list of URLs to cache when the user first access the Service Worker controlled site. 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.

In this event we precache the resources we defined in the PRECACHE_URLS constant. 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 idea is that everytime we run through the `activate` event we check against our caches defined earlier in the service worker. If the existing cache names don't match the current names, the we delete the old caches.

Finally we tell the service wworker to take immediate control of the pages undeer it's scope. The normal behavior is to wait until all the pages visiting the site have either reloaded or closed before the service worker takes over.

```
// 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

It's important to note that response is a readbale stream that we can consume

only once but, because we want to both store in the cache and return the content of the response to the user, we must clone the response (using `response.clone()`) and then return the original request to the user.

```
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;
              });
            });
          });
      })
    );
  }
});
```

The events above represent a basic service worker to cache the shell of an application on first load and then cache resources as the user access them

Other ways to use the service worker's fetch event

Jake Archibald's Offline Cookbook provides additional ideas of when to cache data and different caching strategies. We'll concentrate in the later and talk about

caching strategies.

Cache only

Use this strategy for resources that are static to the site and that were cached during installation.

```
self.addEventListener('fetch', event => {
  // If a match isn't found in the cache, the response
  // will look like a connection error
  event.respondWith(caches.match(event.request));
});
```

Network Only

I use this strategies for resources that I don’t want in the cache like videos, non GET requests and others.

```
self.addEventListener('fetch', event => {
  event.respondWith(fetch(event.request));
  // or simply don't call event.respondWith, which
  // will result in default browser behaviour
});
```

Cache, falling back to network (or Cache First)

Check if the resource is in the cache, if it is respond with it. If it’s not then feetch it from the network, store a copy in the cache and serve it to the client. This is, most likely, the default case. Everything else is special cased

```
self.addEventListener('fetch', event => {
  event.respondWith(
    caches.match(event.request)
    .then(response => {
      return response || fetch(event.request);
    })
```

```
  );
});
```

This is a simplified version of the fetch event example provided earlier.

Cache & network race

There are few situations, particularly with older mobile devices and slow hard drives, where network connectivity will be faster than cache access. In that case we can race the network request and the cache access and return whatever response comes back first.

```
// Promise.race is no good to us because it rejects if
// a promise rejects before fulfilling. Let's make a proper
// race function:
function promiseAny(promises) {
  return new Promise((resolve, reject) => {
    // make sure promises are all promises
    promises = promises.map(p => Promise.resolve(p));
    // resolve this promise as soon as one resolves
    promises.forEach(p => p.then(resolve));
    // reject if all promises reject
    promises.reduce((a, b) => a.catch(() => b))
      .catch(() => reject(Error("All failed")));
  });
};

self.addEventListener('fetch', event => {
  event.respondWith(
    promiseAny([
      caches.match(event.request),
      fetch(event.request)
    ])
  );
});
"All failed"
```

Network falling back to cache

If the fetch request succeeds users get the newest content and if it doesn't then they get the latest version of the content available in the cache. Remember that if the fetch request succeeds you should update the cached content.

There is one thing to consider. If the user has an intermittent or slow connection they'll have to wait for the network to fail before they get any content already on their device. This is a very bad user experience.

```
self.addEventListener('fetch', event => {
  event.respondWith(
    fetch(event.request)
      .catch(function() {
        return caches.match(event.request);
      })
  );
});
```

Cache falling back to network

This is an alternative to network falling back to cache and it requires the page to make two requests, one to the cache, one to the network. The idea is to show the cached data first, then update the page when/if the network data arrives.

Sometimes you can just replace the current data when new data arrives (e.g. game leaderboard), but that can be disruptive with larger pieces of content. Basically, don't "disappear" something the user may be reading or interacting with.

Code in the page:

```
var networkDataReceived = false;

startSpinner();

// fetch fresh data
var networkUpdate = fetch('/data.json')
```

```
.then(function(response) {
  return response.json();
}).'/data.json'n(data) {
  networkDataReceived = true;
  updatePage();
});

// fetch cached data
caches.match('/data.json').then(function(response) {
  if (!response) throw Error("No data");
  return response.json();
}).then(function(data) {
  '/data.json'// don't overwrite newer network data
  if (!networkDataReceived) {
    updatePage(data);
  }
}).catch(function() {
  // we didn't get cached data, the network is our last hope:
  return networkUpdate;
}).catch(showErrorMessage).then(stopSpinner);
```

Code in the ServiceWorker:

We always go to the network & update a cache as we go.

```
self.addEventListener('fetch', function(event) {
  event.respondWith(
    caches.open('mysite-dynamic').then(function(cache) {
      return fetch(event.request).then(function(response) {
        cache.put(event.request, response.clone());
        return response;
      });
    })
  );
});
```

Generic Response

If the content is not in the cache and you're not online to fetch it or the fetch request times our, it would be nice to have a fallback to show the user. It can be as simple as returning a cached offline page like Jake does in the example below; Make sure you cache offline.html when you install the service worker.

```
self.addEventListener('fetch', function(event) {
  event.respondWith(
    // Try the cache
    caches.match(event.request).then(function(response) {
      // Fall back to network
      return response || fetch(event.request);
    }).catch(function() {
      // If both fail, show a generic fallback:
      return caches.match('/offline.html');
    })
  );
});
'/offline.html'
```

Or returning an inline SVG image for images that can't be displayed because they are not cached and not available online.

This example from Jeremy Keith's Adactio runs a cache falling back to network strategy and then, if the resource is an image stores a cloned copy on the cache and returns the image to the user.

If both the cache and fetch fail to return the image the catch portion of the promise is triggered and, if the request was for an image, we return a new Response object containing data to make an inline SVG image rather than provide a broken image and a suboptimal user experience.

```
self.addEventListener('fetch', event => {
let request = event.request;
let url = new URL(request.url);

// For non-HTML requests, look in the cache first, fall back to the networ
```

```
event.respondWith(
  caches.match(request)
    .then(response => {
      // CACHE
      return response || fetch(request)
    .then( response => {
        // NETWORK
        if (request.headers.get('Accept').includes('image')) {
            let copy = response.clone();
            stashInCache(imagesCacheName, request, copy);
        }
        return response;
    })
    .catch( () => {
        'Accept'// OFFLINE
        if (request.headers.get('Accept').includes('image')) {
          return new Response('<svg role="img" aria-labelledby="offline-t
        }
      });
    })
  );
});
```

Jeremy does it for images and Jake does it for HTML, there is no reason why we can’t combine the catch statements from both service workers into one that looks like this:

```
self.addEventListener('fetch', function(event) {
  event.respondWith(
    // Try the cache
    caches.match(event.request).then(function(response) {
      // Fall back to network
      return response || fetch(event.request);
    }).catch(function() {
      if (request.headers.get('Accept').includes('html')) {
        return caches.match('/offline.html');
      }
```

```
      if (request.headers.get('Accept').includes('image')) {
      return new Response('<svg r'Accept'" aria-labelledby="offline-title'
      }

    })
  );
});
```

We could add additional fallbacks matching the content type we want to test and provide different types of fallbacks based on the content.

Handling multiple content types in a fetch request

So far we've used a single strategy for fetching the content of our pages. We can combine these strategies to create a flexible service worker that will cache different types of content differently bassed on the headers it accepts.

For each special case we want to create check that the Accept header includes the type we want to use (for example HTML); we test using `if (request.headers.get('Accept').includes('html'))`. If the headers include the content type then we carry on with caching and providing offline fallbacks.

If the request doesn't match any of our special cases it'll fall through to a default cache first strategy.

```
  if (request.headers.get('Accept').includes('html')) {
    event.respondWith(('html'
      caches.match(event.request)
      .then(response => {
        // Fall back to network
        return response || fetch(event.request);
      })
          .then(response => {
            return caches.open('RUNTIME')
```

```
            .then(cache => {
              cache.put(event.request, response.clone());
              return response;
            });
        })
    .catc'RUNTIME'
      return caches.match('/offline.html');
    });
  });
}
```

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.

The full service worker is available in this Gist.

Headers

The Headers interface of the Fetch API allows you to perform various actions on HTTP request and response headers. These actions include retrieving, setting, adding to, and removing elements and values. A Headers object has an associated header list, which is initially empty and consists of zero or more name and value pairs. You can add to this using methods like `append()`. In all methods of this interface, we match header names by case-insensitive byte sequence.

For security reasons, only the user agent can control some headers. These headers include the forbidden header names and forbidden response header names.

You can retrieve a Headers object via the `Request.headers` and `Response.headers` properties, and create a new Headers object using the `Headers.Headers()` constructor.

We've seen examples of headers when we test to see if a request if of a given mime type. This will test if the request is for an HTML document (mimetype `text/html`) and return a document to notify the application is offline.

```
if (request.headers.get('Accept').includes('html')) {
  return caches.match('/offline.html');
}
```

The Response constructor takes two arguments, the first being the body of the response (the content we get back from feetch). The second argument is an object specifying the status code, status text and headers of the response. We can use these elements to modify the response we return to the client.

In the following example we add a header to include in the returned content. You'll notice a few tricks:

- The response is read only so we have to make the request again in order to process the changes
- We use the technique discussed earlier to add a custom header that we can check if we need to

```
self.addEventListener('fetch', function(event){
  console.log('Caught request for ' + event.request.url);
  event.respondWith(
    fetch(event.request).then(function(response){
      var init = {
        status:     response.status,
        statusText: response.statusText,
        headers:    {
          'X-Foo': 'My Custom Header'
        }
      };

      response.headers.forEach(function(v,k){
        init.headers[k] = v;
      });

      return response.text().then(function(body){
        return new Response(body, init);
      });
  })
  );
```

```
});
```

CORS, NO-CORS and why it matters

One of the reasons why we discuss headers is to introduce the concept of CORS, what it is and how to use it to make requests to an origin different than where the application lives.

Cross-Origin Resource Sharing (CORS) is a W3C spec that allows cross-domain communication from the browser. By building on top of the XMLHttpRequest object, CORS allows developers to work with the same idioms as same-domain requests.

The use-case for CORS is simple. Imagine the site `a.com` has some data that the site `bob.com` wants to access. The web's same origin policy forbids this type of request. However, by supporting CORS requests, the owner of a.com can add a few special response headers that allows b.com to access the data.

CORS is a two step process. The server tell you which, domains other than the origin can access the resource and the client must make explicit that they are asking for a CORS resource.

Setting up CORS request vaaries by server and the type of request you're making. Note that the examples below use a wildcard pattern, meaning we don't care who access the resources. This is not a safe configuration. Make sure you only allow access to hosts you want and not everyone.

To set up blanket CORS permissions in an Apache HTTPS server use:

```
<IfModule mod_headers.c>
  Header set Access-Control-Allow-Origin "*"
</IfModule>
```

To enable CORS in Nginx use the Headers core module which is compiled into the server by default. Then add the following line to your configuration file.

```
add_header Access-Control-Allow-Origin *;
```

When working with Express.js make sure you do the following to enable CORS:

```
app.all('/', function(req, res, next) {
  res.header("Access-Control-Allow-Origin", "*")
  next()
});
```

Information for other servers can be found in the W3C wiki

The client side of the eqution means we have to make the fetch request a CORS request by adding custom headers to the request.

```
var myHeaders = new Headers(); // 1

var myInit = { method: 'GET',
               headers: myHe'GET',
               mode: 'cors', 'cors'// 2
               cache: 'default' };

var myRequest = new Request('racecar.png', myInit);

fetch(myRequest)
  .then(function(response) {
    return response.blob();
  })
  .then(function(myBlob) {
    var objectURL = URL.createObjectURL(myBlob);
    myImage.src = objectURL;
});
```

The example works as follows:

1. Create a new `Headers()`
2. Create an init object that will be added to the request as its second parameter. The important part is the `mode` child that will tell feetch how to process the request. Some of the possible values are:
    1. `same-origin` — If a request is made to another origin with this mode set, the result is simply an error. You could use this to ensure

that a request is always being made to your origin
2. `no-cors` — Prevents the method from being anything other than HEAD, GET or POST. If any ServiceWorkers intercept these requests, they may not add or override any headers except for these. In addition, JavaScript may not access any properties of the resulting `Response`. This ensures that ServiceWorkers do not affect the semantics of the Web and prevents security and privacy issues arising from leaking data across domains
3. `cors` — Allows cross-origin requests, for example to access various APIs offered by 3rd party vendors. These are expected to adhere to the HTTP access control (CORS) protocol. Only a limited set of headers are exposed in the `Response`, but the body is readable

We then make a blob of the response and build a URL to display to the user.

If mode not defined in step 2, the default value of `no-cors` is assumed.

Links and Resources: Headers

- Modifying Service Worker Response
- Fetch API (David Walsh)
- enable CORS
- HTML5 Rocks CORS Tutorial
- `request.mode`

Intercepting Responses

Fetch events allows developers to intercept and replace responses with our own content. We've seen this before when we discussed offline fallbacks and providing alternative content when the user is offline.

But we can intercept requests while we are online. The example below shows how to provide a different response to a request if the URL to fetch include the string `cats.jpg` and replaces it with `dogs.png`

```
self.addEventListener('fetch', function(event){
    console.log('Caught request for ' + event.request.url);
    if (event.request.url.includes('cat.jpg'){
      event.respondWith(new Response('dogs.png'));
```

```
    }
});
```

Just for Fun: caching DASH video segments

Just to prove how flexible and powerful sservice workers are, we'll create a fetch handler that will cache MP4 DASH video.

DASH video creates one or more segments for each video we create and a manifest file that tells a DASH-compatible player the type of video we've created and what audio and video segments are.

To cache the video I've created two functions and an additional case for a fetch event handler.

The `loadFromCacheOrFetch` function is the core of this fetch event. It will open the specified cache and try to match the request.

If the request matches an item in the cache then it will return that item. The fragment will have an additional header that we add when we cache the item.

If the request doesn't match then we take a clone of the request and fetch it. This is where another peculiarity of working with DASH video comes in. A service worker cannot cache partial responses (HTTP 206) so we have to check if the response is ok and **did not** return a status code 206. Only if both conditions are met we fetch the resource and store it in the cache using the `cacheResponse` function.

```
CONST DASH_CACHE = dash_cache_v1;

function loadFromCacheOrFetch(request) {
  return caches.open(DASH_CACHE)
  .then(cache => {
    return cache.match(request)
    .then(response => {
      if (response) {
```

```
        console.log('Handling cached request', request.url);
        return response;
      }

      return fetch(request.clone())
      .then(response => {
        if (response.ok && response.status != 206) {
          console.log('Caching MP4 segment', request.url);
          cacheResponse(cache, request, response);
        }

        return response;
      });
    });
  })
}
```

The cacheResponse function modifiess the response object before putting it in the cache. Because the response is readonly we have to recreate it if we'll make any changes.

We create an array of the data we want to pass to the response object when we actually cache it, that's why the response has uses two parameters, an arrayBuffer and the array that we creted with response data and header.

Response objects are single use. This means we need to call clone() so we can both store the ArrayBuffer and give the response to the page.

```
function cacheResponse(cache, request, response) {
  var init = {
    status: response.status,
    statusText: response.statusText,
    headers: {'X-Shaka-From-Cache': true}
  };

  response.headers.forEach((value, key) => {
    init.headers[key] = value;
  });
```

```
  return response.clone().arrayBuffer()
  .then(ab => {
    cache.put(request, new Response(ab, init));
  });
}
```

For the actual caching I've added one more case to the fetch event to capture the DASH videos. If the request URL ends with an `mp4` or `m4s` extension then we respond by running `loadFromCacheOrFetch` for the request.

```
self.addEventListener('fetch', event => {
  if (event.request.url.endsWith('.mp4') || event.request.url.endsWith('.m
    event.respondWith(loadFromCacheOrFetch(event.request));
  }
});
```

Automating Service Worker Creation with Workbox.js

SW-precache and SW-toolbox made creating Service Workers with dynamic caching much easier but there were two separate libraries and SW-precache required a separate file with all the SW-toolbox libraries routes in it. This made it very error prone to edit and update.

Workbox.js is the evolution of Google's Service Worker Libraries. It consolidates all Service Worker build steps into one task (Gulp, Webpack and NPM Script versions available) and abstracts a lot of the writing and configuration behind the scenes so developers don't need to see the process, only the result.

If you want to create specialized routes manually, Workbox will help you there too.

Workbox is a Node package so I always install it as a development dependency with the command below

```
npm install workbox-build --save-dev
```

At the top of the `gulpfile.js` file place the following constant declaration to bring workbox-build into scope of the file.

```
const wbBuild = require('workbox-build');
```

Copy the task below to your gulpfile. Note that this task uses ES6 arrow functions and promises so it'll work on newer versions of Node (5.x and newer).

```
gulp.task('bundle-sw', () => {
  return wbBuild.generateSW({
   globDirectory: './_site/', './_site/'// 1t: './_site/sw.js', // 2
    s'./_site/sw.js'// 2*\/*.{html,js,css}'], // 3
    globIgnores: ['sw.js'], // 4
    skipWaiting: tru'sw.js'// 4
```

```
    skipWaiting: true, // optional
    clientsClaim: true, // optionalle.log('Service worker generated.');
  })
  .catch((err) => {
    console.log('[ERROR] ', err);
  });
})
'Service worker generated.'
```

The tasks tells `workbox-build`:

1. Where to search for content
2. Where to write the resulting Service Worker
3. What files to add to the Service Worker (all HTML, CSS and Javascript files)
4. What files to ignore. In this example we don't want to cache the service worker itself as caching would defeat the purpose
5. ***Optionally*** set `skipWaiting` and `clientsClaim` to true. This will take over clients imediately after installing the service worker regardless of having tabs/windows open to the site

If it succeeds then we log a success message to console and if we fail we log the error to console as well. It'as important to note that this task runs after all yur other build steps to make sure it will pick up all the changes made during the build process.

The default Workbox task described above produces a basic precaching service worker where we indicate the files that we want to precache. But it does not provide routing or special cases for specific routes. This would be good in most cases but sometimes it's not enough.

Jeff Posnick pointed me to a solution to integrate `workbox-routing` and `workbox-build` on the same Service Worker and still use the a Gulp task to populate the data.

```
importScripts('scripts/workbox-sw.dev.v2.1.0.js');
// const workboxSW = new self.SWLib();
const workboxSW = new self.WorkboxSW();
```

```
// Pass in an empty array for our dev environment service worker.
// As part of the production build process, the `service-worker`
// gulp task will automatically replace the empty array with the
// current precache manifest.
workboxSW.precache([]);

// Use a cache first strategy for files from googleapis.com
workboxSW.router.registerRoute(
  new RegExp('https://ajax.googleapis.com/ajax/libs'),
  workboxSW.strategies.cacheFirst({
    cacheName: 'googleapis',
    cacheExpiration: {
      'https://ajax.googleapis.com/ajax/libs'// Expire after 30 days (expr
      maxAgeSeconds: 30 * 24 * 60 * 60,
    },
  })
);

// Note to self, woff regexp will also match woff2 :PRegExp('.(?:ttf|otf|w
  workboxSW.strategies.cacheFirst({
    cacheName: 'fonts',
    cacheExpiration: {
      // Expire after 24 hours (expressed in '.(?:ttf|otf|woff)$'// Expire
  new RegExp('.(css)$'),
  workboxSW.strategies.networkFirst({
    cacheName: 'css',
    cacheExpiration: {
      maxAgeSeconds: 1 * 24 * 60 * 60,
    },
  })
);

// Use a cache-first strategy for the images
workboxSW.rout'.(css)$'// Use a cache-first strategy for the images'),
  workboxSW.strategies.cacheFirst({
    cacheName: 'images',
    cacheExpiration: {
      // maximum 50 entries
```

```
      maxEntries: 50,
      // Expire after 30'images'// maximum 50 entries
      maxEntries: 50,
      // Expire after 30 days (expressed in seconds)
      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 .htm and .html files use cacheFirstgies.cacheFirst({
    cacheName: 'content',
    cacheExpiration: {
      maxAgeSeconds: 1 * 24 * 60 * 60,
    },
  })
);

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

// Local videos get the same treatment, only pull from the network
workboxSW.router.registerRoute(
  new RegExp('/.(?:mp4|webm|ogg)$/'),
  workboxSW.strategies.networkOnly()
);

// The default route uses a cache first strategy
workboxSW.router.setDefaultHandler({
  handler: workboxSW.strategies.cacheFirst()
```

```
});
```

The solution is a two-step process. We first write our Service Worker as shown below. We pass an empty array as the parameter to `workboxSW.precache` and we populate the empty array from the `bundle-sw` task in Gulp.

We've also created custom routes using `workboxSW.router.registerRoute` to register the route and `workboxSW.strategies` to use one of the following strategies:

- CacheFirst
- CacheOnly
- NetworkFirst
- NetworkOnly
- StaleWhileRevalidate

We can further customize each caching strategy. Let's take the route below as an example. We register a route using a regular expression that will match all png, jpg and gif images and use the cache first strategy.

We refine the caching strategy by giving the cache a name and expiration parameters. In the expiration we add a maximum number of entries (after which the oldest images will be purged from the cache) and a duration in seconds equal to 30 days.

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.

If you want more details about the parameters we can pass to cacheExpiration look at the source code in Github.

```
workboxSW.router.registerRoute(
  new RegExp('/.(?:png|gif|jpg)$/'),
  workboxSW.strategies.cacheFirst({
    cacheName: 'images',
    cacheExpiration: {
      maxEntries: 50,
      maxAgeSeconds: 30 * 24 * 60 * 60
```

```
    },
    cacheableResponse: {statuses: [0, 200]}
  })
);
```

The routes we define in the service worker will not change or will not change to frequently so doing it this way makes sure we get the best of both worlds.

The modified service worker takes a different approach than what we saw before using `workbox-build`. Innstead of building the manifest directly, it injects the list of files in the manifest into the service worker. Remember that we put an empty array on the precache section of the service worker. This is the task that will populate the empty array with the files we need to precache.

```
gulp.task('service-worker', () => {
  return workboxBuild.injectManifest({
    swSrc: 'src/service-worker.js',
 'src/service-worker.js'ce-worker.js',
    globDirectory: '_site',
    globIgnores: ['sw.js'],
    staticFileGlobs: [
      'scripts'_site'/**/*.js',
      'styles/main.css',
      'images/logo.png',
      'index.html'
    ]
  });
});
```

And the best part is that, if we missed anythhing, the files will be cached at run time and we make sure that we still cache the content.

In static or content heavy sites we may want to change the values for `staticFileGlobs` to specific file names that we should cache rather than wildcaard paths. The default value may cache too many files and make the initial caching and subsequent loading take longer than we'd like.

More APIs to make an even better PWA

Before we jump into the PWA checklist we'll talk about APIs we can use in PWAs to enhance performance beyond the basic functionality of PWAs we've discussed so far.

Background Sync

The background sync API provides tools to create one-of and periodic data synchronization after the content was initially fetched for the application. It accomplishes these tasks by adding events for the service worker and additional functions for the service worker.

In the code below we reister a service worker and, when we're reaady, we register a sync event; The name we register here is important; we'll use it as the name of the event when we actually do the sync.

```
if ('serviceWorker' in navigator) {
  window.addEventListener('load', () => {
   'load'ator.serviceWorker.register('/sw.js')
    .then(function(registration) {
      '/sw.js'// Registration was successfulonsole.log('ServiceWorker reg
    })
    .catch(function(err) {
      // regis'ServiceWorker registration successful with scope: '// regis
      console.log('ServiceWorker registration failed: ', err);
    });
  });
}

// Register a one-off sync event {
  return swRegistration.sync.register('image-fetch');
});
'image-fetch'
```

When we do the actual sync event we use the name of the tag we registered at sync registration time. In this case we wait until the function executes. This function is a wrapper for fetching the image. If we're working with more complex content the function can build more elaborated content.

```
self.addEventListener('sync', function (event) {
  if (event.tag === 'image-fetch') {
    event.waitUntil(fetchDogImage());
  }
});

function fetchDogImage () {
  fetch('./doge.png')
    .then(function (response) {
      return response;
    })
    .then(function (text) {
      console.log('Request successful', text);
    })
    .catch(function (error) {
      console.log('Request failed', error);
    });
}
```

Links and Resources: Backkground Sync

- Background Sync Explainer
- Background Sync (Ponyfoo)
- Background Sync (Google Developers)

Push Notifications

Push notifications allow the server hosting your application to push informatin to be displayed even when the tab with your application is in the background or the entire browser is closed.

This is a complex topic and requires several moving pieces. Rather than try and condense the topic here I'll refer you to Matt Gaunt Web Push Book, a

thorough discussion of Web Push and Push Notificaations.

Links and Resources: Push notifications

- Web Push Notifications: Timely, Relevant, and Precise = Web Push Book
- Matt Gaunt's Notification Presentation at SFHTML5
    - Presentation: https://gauntface.github.io/presentations/2017/sfhtml5/#
    - Video: https://www.youtube.com/watch?v=lteJlP3Xbt4

Credential Management API

The Credential Management API lets websites interact with a user agent's password system so that websites can deal in a uniform way with site credentials and user agents can provide better assistance with the management of their credentials. For example, user agents have a particularly hard time dealing with federated identity providers or esoteric sign-in mechanisms that use more than just a username and password. To address these problems, the Credential Management API provides ways for a website to store and retrieve different types of password credentials. This give users capabilities such as seeing the federated account they used to sign on to a site, or resuming a session without the explicit sign-in flow of an expired session.

> There is a working example as part of Google Codelabs

Links and Resources: Credential Management

- Credential Management API (Google)
- Credential Management API (MDN)
- Enabling auto sign-in with the Credential Management API Codelab
- https://developers.google.com/web/fundamentals/security/credential-management/

Payment Request API

Many problems related to online purchase abandonment can be traced to checkout forms, which are user-intensive, difficult to use, slow to load and refresh, and require multiple steps to complete. The Payment Request API is a system that is meant to eliminate checkout forms. It vastly improves user workflow during the

purchase process, providing a more consistent user experience and enabling web merchants to easily leverage disparate payment methods.

Links and Resources: Payment Request API

- Payment Request API (Google)
- Payment Request API (MDN)

Offline Analytics

When working with offline analytics we need to make sure that whatever events happen offline are captured to replay later when the user has regained connectivity. Both Google Offline Analytics libraries discussed below work the same way: They set up a new fetch event handler in your service worker, which responds to requests made only to the Google Analytics domain.

The analytics fetch event uses a network fetch strategy to send analytics events to Google Analytics servers. If the user is online this network request will succeed and the analytics servers will store the data; everything is fine.

If the network request fails, the library will store information about the request to IndexedDB, along with a timestamp indicating when the request was initially made. Each time your service worker starts up, the library will check for queued requests and attempt to resend them, along with some additional Google Analytics parameters:

- A qt parameter, set to the amount of time that has passed since the request was initially attempted, to ensure that the original time is properly attributed
- Any additional parameters and values supplied in the parameterOverrides property of the configuration object passed to goog.offlineGoogleAnalytics.initialize(). For example, you could include a custom dimension to distinguish requests that were resent from the service worker from those that were sent immediately.

If the service worker succeeds, then the request is uploaded to the analytics servers and removed from IndexedDB.

If the retry fails, and the initial request was made less than 24 hours ago, it will be kept in IndexedDB to be retried the next time the service worker starts. Note

that Google Analytics hits older than four hours are not guaranteed to be processed, but resending these older hits “just in case” shouldn’t hurt.

Standalone Offline Analytics

There is a standalone library for offline analytics and you’re writing your service worker by hand. To use it import the plugin using npm

```
npm install --save-dev sw-offline-google-analytics
```

And then use the following code in your service worker, before any fetch event:

```
// Import the library into the service worker global scope:
importScripts('path/to/offline-google-analytics-import.js');

'path/to/offline-google-analytics-import.js'// Then, call goog.offlineGoog
goog.offlineGoogleAnalytics.initialize();

// At this point, implement any other service worker caching strategies
// appropriate for your web app.
```

Workbox Offline Analytics

If you’re using workbox.js you can use the following code:

```
// This code should be placed before 'fetch' event handlers are defined.
// Import the library into the service worker global scope:
importScripts('path/to/offline-google-analytics-import.js');

'path/to/offline-google-analytics-import.js'// Then, call workbox.googleAn
workbox.googleAnalytics.initialize();
```

Links and Resources: Offline Analytics

- Offline Google Analytics
- Workbox Analytics

Online and offline events

The last API I wanted to discuss is online/offline. We need a way to communicate the online status to our users. Online/Offline events provide a solution to this communication need.

Browsers implement this property differently.

In Chrome and Safari, if the browser is not able to connect to a local area network (LAN) or a router, it is offline; all other conditions return true. So while you can assume that the browser is offline when it returns a false value, **you cannot assume that a true value necessarily means that the browser has a working internet connection**. You could be getting false positives, such as in cases where the computer is running a virtualization software that has virtual ethernet adapters that are always “connected.”

In Firefox and Internet Explorer, switching the browser to offline mode sends a false value. Until Firefox 41, all other conditions return a true value; since Firefox 41, in OS X and Windows, the value will follow the actual network connectivity.

An example script, taken from MDN, looks like this:

```
window.addEventListener('load', function() {
  var status = document.getElementById("status");
  var log = document.getElementById("log");

  function updateOnlineStatus(event) {
    var condition = navigator.onLine ? "online" : "offline";

    status.className = condition;
    status.innerHTML = condition.toUpperCase();

    log.insertAdjacentHTML("beforeend", "Event: " + event.type + "; Status
  }

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

The full example is available in Codepen

Links and Resources: Online/Offline Detection

- MDN
- The initial definition in the HTML specification
- navigator.onLine in Chrome Dev channel

The PWA checklist

Google's PWA checklist presents both a basic and an advanced set of requirements for Progressive Web Applications. As I mentioned earlier not all these requirements are technical, some of them have to deal with the performance of your application/site and with best practices in responsive web design.

I've grouped them in two categories: Basic and Advanced requirements. I will comment on individual entries as needed.

Basic Requirements

Serving through HTTPS in a requirement for service workers and most, if not all, modern features available in browsers. Whether you run a PWA or not you should consider moving your site to secure hosting.

If cost is an issue there are way to obtain low cost or free SSL certificates. Tools like letsencrypt.org make it trivial to obtain ceertificates and install them on your server.

Site is served over HTTPS	
To Test	Use Lighthouse to verify Served over HTTPS
To Fix	Implement HTTPS and check out letsencrypt.org to get started

You're already creating responsive content, right? As designers and developers we know that it's not enough to do a desktop-first design and then tweak it for other form factors.

Even if we look at browser popularity we might be surprised at the results. According to Statcounters (gs.statcounter.com) the three most popular browsers, worldwide (08/16 to 08/17), are:

Browser	Market Share
Chrome	54.89%
Safari	14.88%
UC Browser	7.43%
Firefox	5.9%
Opera	4%
IE	3.69%

If you're looking at working in emerging markets, the figures change. According to Statcounter, the Browser market share for mobile devices in Asia breaks like this over the last year (same time period as before).

Browser	Market Share
Chrome	50.46%
UC Browser	22.23%
Safari	10.27%
Opera	6.38%
Samsung Internet	5.97%
Android	3.31%

So any way you look at it, it pays to be responsive.

Pages are responsive on tablets & mobile devices	
To Test	Use Lighthouse to verify Yes to all of Design is mobile-friendly, although manually checking can also be helpful. Check the Mobile Friendly Test
To Fix	Look at implementing a responsive design, or adaptively serving a viewport-friendly site.

Use the service worker to make sure that at least the entry page to your site works while offline. This may be just a matter to cache the assets for the starting URL

when you install the service worker (as we saw earlier).

Runtime caching should take care of ca ching the rest of your PWA.

The start URL (at least) loads while offline	
To Test	Use Lighthouse to verify URL responds with a 200 when offline.
To Fix	Use a Service Worker.

Metadata provided for Add to Home screen	
To Test	Use Lighthouse to verify User can be prompted to Add to Home screen is all Yes.
To Fix	Add a Web App Manifest to your project.

Working with a service worker will smooth out network issues and increase the perceived performance of your site or app. That said we still want the content to load fast and we should take all steps to make sure it happens even for older browsers that don't support service workers or the polyfill.

First load fast even on 3G	
To Test	Use Lighthouse on a Nexus 5 (or similar) to verify time to interactive <10s for first visit on a simulated 3G network.
To Fix	There are many ways to improve performance. You can understand your performance better by using Pagespeed Insights (aim for score >85) and SpeedIndex on WebPageTest (aim for <4000 first view on Mobile 3G Nexus 5 Chrome) A few tips are to focus on loading less script, make sure as much script is loaded asynchronously as possible using <script async> and make sure render blocking CSS is marked as such. You can also look into using the PRPL pattern and tools like PageSpeed Module on the server.

Browsers are getting there in interoperability but they are not there quite yet. Make sure that it works on all your target browsers for mobile and desktop. Fix whatever doesn't work (duh).

Site works cross-browser	
To Test	Test site in Chrome, Edge, Firefox and Safari
To Fix	Fix issues that occur when running the app cross-browser

Page transitions don't feel like they block on the network	
Transitions should feel snappy as you tap around, even on a slow network, a key to perceived performance.	
To Test	Open the app on a simulated very slow network. Every time you tap a link/button in the app the page should respond immediately, either by: ▪ Transitioning immediately to the next screen and showing a placeholder loading screen while waiting for content from the network. ▪ A loading indicator is shown while the app waits for a response from the network
To Fix	If using a single-page-app (client rendered), transition the user to the next page immediately and show a skeleton screen and use any content such as title or thumbnail already available while content loads.

Each page has a URL	
To Test	Ensure individual pages are deep linkable via the URLs and that URLs are unique for the purpose of shareability on social media by testing with individual pages can be opened and directly accessed via new browser windows.
To Fix	If building a single-page app, make sure the client-side router can re-construct app state from a given URL.

Advanced Requirements (Exemplary PWAs)

The basic requirements for a PWA represent an “MVP” of what a PWA can be. These advanced requirements point out to additional resources, concepts and ideas that will make the app even better.

These fall under the “nice to have” or “these are optional but strongly encouraged” parts of a PWA.

Indexability & Social

Whether a PWA or a regular site we want to make sure that search engines can find your app and show it to potential users. Because the concept of PWA is new there hasn’t been much done in the search engine space and discoverability spacces. Microsoft Bing search engine has taken the lead in this area.

> We [Microsoft] are already using the Bing Crawler to identify PWAs on the Web for our PWA research. The Web App Manifest is a proactive signal from developers that a given website should be considered an app; we’re listening to that signal and evaluating those sites as candidates for the Store. Once we identify quality PWAs, we’ll automatically generate the APPX wrapper format used by the Windows Store and assemble a Store entry based on metadata about the app provided in the Web App Manifest.
>
> Aarong Gustafson - Progressive Web Apps and the Windows Ecosystem

For more information, see Google’s guide to social optimization and social discovery.

Site's content is indexed by Google	
To Test	Use the Fetch as Google tool to preview how Google will see your site when it is crawled.
To Fix	Google's indexing system does run JavaScript but some issues may

	need to be fixed to make content accessible. For example, if you are using new browser features like the Fetch API, ensure that they are polyfilled in browsers without support.

Just like regular pages, the pages of a PWA bennefit from metadata. This will help Google and other searc engine crawlerss to better index your content. See the examples under the `to-fix` section for ideas of what you can do with Metadata. Also check James Williams course on HTML5 structured data on Linkedin Learning

Note that this refers to generic metadata, not the one you'd use for Facebook or Twitter. That comes in the next section.

Schema.org metadata is provided where appropriate	
Schema.org metadata can help improve the appearance of your site in search engines.	
To Test	Use the testing tool to ensure title, image, description etc. are available.
To Fix	Markup the content. For example: ▪ A recipe app should have the Recipe type markup for Rich Cards. ▪ A news app should have the NewsArticle type markup for Rich Cards and/or AMP support. ▪ An ecommerce app should have the Product type markup for Rich Cards.

Social metadata is the other part of making you applicaation known and includes Facebook, Twitter, Google+ and others.

Social metadata is provided where appropriate	
To Test	▪ Open a representative page in Facebook's crawler and ensure it looks reasonable ▪ Check that Twitter Cards meta data is present (for example <meta name="twitter:card" content="summary" />) if you feel it would be

	appropriate
To Fix	Mark up content with Open Graph tags and as advised by Twitter.

Some sites still use 2 sites for the same content, one is the standard `www` and the other one is a specialized mobile site, usually called m-dot (`m.site.com`). To help search engine crawlers use the `rel="canonical` attribute of the link tag to tell the crawler which one is the canonical version.

Canonical URLs are provided when necessary	
This is only necessary if your content is available at multiple URLs.	
To Test	Determine whether any piece of content is available at two different URLs. Open both of these pages and ensure they use <link rel=canonical> tags in the head to indicate the canonical version
To Fix	Add a canonical link tag to the <head> of each page, pointing to the canonical source document. See Use canonical URLs for more information.

The history API lets you programmatically control the navigation of your app. Using it means that your users will be able to navigate throughout the application without having to worry if the browser will remember the sites you visited.

Pages use the History API	
To Test	For single page apps, ensure the site doesn't use fragment identifiers. For example everything after the #! in https://example.com/#!user/26601.
To Fix	Use the History API instead of page fragments.

User experience

A PWA is no different than any other app in how users will react to poor UX and UI. We need to present a great user experience as we can.

Ensure all content, especially images and ads, have fixed sizing in CSS or inline on the element. consider ways to present a preview of the content until it’s downloaded.

Content doesn't jump as the page loads	
To Test	Load various pages in the PWA and ensure content or UI doesn't "jump" as the page loads
To Fix	Ensure all content, especially images and ads, have fixed sizing in CSS or inline on the element. Before the image loads you may want to show a grey square or blurred/small version (if available) as a placeholder

This is an interesting case. I seldom see this being enforced but the user experience works consistently and doesn’t make you scroll up or down to get to the section of the content we were looking at.

Pressing back from a detail page retains scroll position on the previous list page	
To Test	Find a list view in the app. Scroll down. Tap an item to enter the detail page. Scroll around on the detail page. Press back and ensure the list view is scrolled to the same place it was at before the detail link/button was tapped.
To Fix	Restore the scroll position in lists when the user presses 'back'. Some routing libraries have a feature to do this for you.

In mobile most applications will use the system’s virtual keyboard. When the keyboard appears, make sure the content is not obscured under the keyboard.

When tapped, inputs aren't obscured by the on screen keyboard	
To Test	Find a page with text inputs. Scroll to put the text input as low on the screen as you can make it. Tap the input and verify it is not covered when the keyboard appears.
To Fix	Explore using features like Element.scrollIntoView() and Element.scrollIntoViewIfNeeded() to ensure the input is visible when tapped.

Content is easily shareable from standalone or full screen mode	
To Test	Ensure from standalone mode (after adding the app to your home screen) that you are able to share content, if appropriate, from within the app's UI.
To Fix	Provide social share buttons, or a generic share button within your UI. If a generic button, you may want to directly copy the URL to the user's clipboard when tapped, offer them social networks to share to, or try out the new Web Share API to integrate with the native sharing system on Android.

Site is responsive across phone, tablet and desktop screen sizes	
To Test	View the PWA on small, medium and large screens and ensure it works reasonably on all.
To Fix	Review our guide on implementing responsive UIs.

Any app install prompts are not used excessively	
To Test	Check the PWA doesn't use an app install interstitial when loaded
To Fix	There should only be one top or bottom app install banner After the PWA is added to the user's home screen, any top/ bottom banners should be removed.

Whenever you use Add to Home Screen you're responsible for not annoying your users by presenting A2HS before the user had the chance to evaluate your site and make a decision or at inoportune times.

The Add to Home Screen prompt is intercepted	
To Test	Check the browser doesn't display the A2HS at an inopportune moment, such as when the user is in the middle of a flow that shouldn't be interrupted, or when another prompt is already displayed on the screen.
To Fix	Intercept the beforeinstallprompt event and prompt later

	Chrome manages when to trigger the prompt but for situations this might not be ideal. You can defer the prompt to a later time in the app's usage.

Performance

Users are pushy and expect more than our apps. They want them fast and they want responsive and they want high quality. This section will look at the performance requirements for great PWAs.

First load very fast even on 3G	
To Test	Use Lighthouse on a Nexus 5 (or similar) to verify time to interactive < 5s for first visit on a simulated 3G network (as opposed to the 10s goal for baseline PWAs)
To Fix	Review the performance section of WebFundamentals and ensure you're following the best practices. You can understand your performance better by using Pagespeed Insights (aim for a score >85) and SpeedIndex on WebPageTest (aim for a score <4000 on the first view on Mobile 3G Nexus 5 Chrome). A few tips are to focus on loading less script, make sure as much script is loaded asynchronously as possible using <script async> and make sure render blocking CSS is marked as such.

Caching

One of the great things about PWAs is that they smooth out network connections making PWAs look like they are faster.

We can accomplish the caching performance with service worker that use cache-first strategies. This will pull the data from the cache and, only if not cached, will fetch it from the network and store it in the cache.

There are exceptions to the rule. You may not want to cache large videos, audio or other large files. If so present the user with cues that the content is not available offline.

Site uses cache-first networking	
To Test	Set the network emulation to the slowest setting and browse around the app. Then, set the network emulation to offline and browse around. The app should not feel faster when offline than on a slow connection.
To Fix	Use cache-first responses wherever possible.

There are times when it’s important for the user to know if they are offline. For example: You should tell the user they are offline if they are trying to complete an ecommerce transation or trying to view videos that are only available online.

Site appropriately informs the user when they're offline	
To Test	Emulate an offline network and verify the PWA provides an indication that you are offline.
To Fix	Use the Network Information API to show the user an indication when they're offline.

Push notifications

This check list only applies if notifications are implemented. Adding push notifications is not a requirement for an exemplary progressive web app.

Always provide context for the user to decide if they should enable notifications. Tell them what types of messages your app will push and be explicit as to what permissions you want. If you’re asking for permission to use Push Notifications then do just that.

If your app is requesting permission for push notification on first visit you better have a good explanation right up front as to why you’re asking before users have a chance to exprience your app.

Provide context to the user about how notifications will be used	
To Test	▪ Visit the site, and find the push notifications opt-in flow ▪ When you are shown the permission request by the browser, ensure that context has been provided explaining what the site wants the permission for ▪ If the site is requesting for the permission on page load, ensure it provides very clear context simultaneously for why the user should enable push notifications
To Fix	See our guide to creating user-friendly Notifications permissions flows.

Please, please, please don't annoy your users by presenting them with Push Notification signups on every page of your site. Remember that if they block your application, it will stay blocked until they decide to unblock it an add notifications from your site.

As the test below indicates, if the user doesn't opt in to your application's notification workflow you shouldn't prompt them againn in the same session.

UI encouraging users to turn on Push Notifications must not be overly aggressive.	
To Test	Visit the site and find the push notifications opt in flow. Ensure that if you dismiss push notification, the site does not re-prompt in the same way within the same session.
To Fix	If users say they don't want a certain kind of notification, do not reprompt for at least a few days (for example, one week).

Site dims the screen when permission request is showing	
To Test	Visit the site and find the push notifications opt-in flow. When Chrome is showing the permission request, ensure that the page is "dimming" (placing a dark overlay over) all content not relevant to explaining why the site needs push notifications.

To Fix	When calling Notification.requestPermission dim the screen. Undim it when the promise resolves.

If you decide to use push notifications in your site make sure they are relevant ant timely. By this I mean that the user only gets notifications when they perform and acction, therre is new/updated information, something happens that requires the user to take action and it's about the site they are visting.

Push notifications must be timely, precise and relevant	
To Test	Enable push notifications from the site and ensure the use cases they're using the push notifications for are: ▪ Timely — A timely notification is one that appears when users want it and when it matters to them ▪ Precise — A precise notification is one that has specific information that can be acted on immediately ▪ Relevant — A relevant message is one about people or subjects that the user cares about
To Fix	See our guide on creating great push notifications for advice. If your content is not timely and relevant to this user, consider using email instead.

Make sure the user has a way to disable and, hopefully, reenable notifications without going to the browser's UI to do so.

Provides controls to enable and disable notifications	
To Test	Enable push notifications from the site. Ensure there is some place on the site that allows you to manage your notifications permissions or disable them.
To Fix	Create a UI that allows users to manage their notification preferences.

Additional features

These are additional features that make PWAs easier to work with. They only apply if you're using the API on your site/app.

User is logged in across devices via Credential Management API	
This only applies if your site has a sign in flow.	
To Test	Create an account for a service and ensure you see the save password/ account dialog show up. Click "Save". Clear cookies for the site (via clicking on the padlock or Chrome settings) and refresh the site. Ensure that you either see an account picker (e.g. if there are multiple accounts saved) or are automatically signed back in. Sign out and refresh the site. Ensure that you see the account picker.
To Fix	Follow our Credential Management API Integration Guide.
User is logged in across devices via Credential Management API	
This check only applies if your site accepts payments.	
To Test	Enter the payment flow. Instead of filling out a conventional form, verify the user is able to pay easily via the native UI triggered by the Payment Request API.
To Fix	Follow our Payment Request API Integration Guide.

Links and References

- https://cloudfour.com/thinks/a-progressive-roadmap-for-your-progressive-web-app/
- https://developers.google.com/web/fundamentals/engage-and-retain/web-app-manifest/
- https://fberriman.com/2017/06/26/naming-progressive-web-apps/
- https://hackernoon.com/a-beginners-guide-to-progressive-web-apps-the-frontend-web-424b6d697e35
- https://infrequently.org/2015/06/progressive-apps-escaping-tabs-without-losing-our-soul/
- https://julian.is/article/progressive-web-apps/
- https://medium.com/@AaronGustafson/your-site-any-site-should-be-a-pwa-97ddcc21c2cb
- https://medium.com/samsung-internet-dev/a-beginners-guide-to-making-progressive-web-apps-beb56224948e
- https://medium.com/samsung-internet-dev/heres-what-you-get-for-free-with-a-progressive-web-app-74b7ac5bdb3a
- https://medium.com/samsung-internet-dev/progressive-web-apps-are-a-toolkit-not-a-recipe-b2fd68613de5
- https://medium.com/web-on-the-edge/offline-posts-with-progressive-web-apps-fc2dc4ad895
- https://www.aaron-gustafson.com/notebook/your-site-should-be-a-pwa/
- https://www.aaron-gustafson.com/notebook/progressive-web-apps-and-the-windows-ecosystem/
- https://www.smashingmagazine.com/2016/08/a-beginners-guide-to-progressive-web-apps/
- Jake Archibald's Offline Cookbook
- Matt Gaunt's Notification Presentation at SFHTML5
    - Presentation: https://gauntface.github.io/presentations/2017/sfhtml5/#
    - Video: https://www.youtube.com/watch?v=lteJlP3Xbt4