Service Workers: The easy and the hard way

Service Workers are awesome. They give you a performance boost and the capability of limited offline work without having to create apps for each platfomt that we want to use our app in.

Service workers are the core of PWAs (Progressive Web Applications) where we take advantage of a set of web technologies to make apps that work closer to how native apps would.

We'll concentrate on the service worker and in using Workbox.js and vanilla Javascript to create the service worker. I've chosen to go with Workbox first because the abbstraction makes it easier to create the type of abstractions I need.

Once I have a working service worker I will try to duplicate the functionality without libraries to compare how easy/hard it is.

Registering the service worker

We register the worker on page load to improve performance. This will work the same whether we're using a vanilla Javascript service worker or if we implement it using Workbox.

Place the following script tag in the main page of your site or application.

```
<script>
  if ('serviceWorker' in navigator) {
    window.addEventListener('load', () => {
      navigator.serviceWorker.register('./sw.js');
      console.log('Service Worker registered');
    });
  }
</script>
```

Workbox.js

Workbox is a libbrbary from Google that automates the creation of a service worker and hides a lot of the complexities inherent in the service worker creation.

This particular service worker is optimized for performance. We want to cache all the resource and either serve them from the catch falling back to the network or serving cached content while, at the same time, fetching the asset from the network and placing it in the cache for future use.

Defining Items to Precache

We have an external configuration file to run Workbox precaching. We'll talk about the command at the end of the section.

The config file (`workbox.config.js`) tells us the following things:

- The source file for the service worker (`swSrc`)
- The destinatioon for the service worker (`swDest`)
- The root for the files we want to precache (`globDirectory`)
- The list of files to precache (`globPatterns`)

```
module.exports = {
  'swSrc': 'js/sw.js',
  'swDest': 'sw.js',

  'globDirectory': '.',
  'globPatterns': [
    'index.html',
    'css/index.css',
    'js/zenscroll.js',
    'pages/404.html',
    'pages/offline.html',
  ],
};
```

Workbox Handlers

I've chosen to decouple the routing and the strategies for each type of content that I want to cache.

The first type of content creates the `content-cache` cache where we will later store cached HTML content

```
// Cache strategies definitions
// HTML caching strategy
const contentHandler = workbox.strategies.cacheFirst({
  cacheName: 'content-cache',
});
'content-cache'
```

The CSS handler will hold our stylesheets, both local and third party, to style the content.

The CSS cache does a couple special things via Workbox plugins:

- It accepts opaque responses for third party resources that wouldn't normally be cached
    - It uses the cacheableResponse plugin
    - The status code '0' represents opaque responses
- It sets an expiration date to 14 days
- It sets the maximum number of items the cache will hold. When the cache is full new entries will push the oldest out of the cache (and delete them)

We cache opaque responses at our own risk since we have no way of knowing if the request succeeded or not. I considered the risk and accepted it because the networks where I'm pulling third party resources from are big and stable (Cloudflare and Google Fonts).

In a future iteration I may want to move third party CSS resources to its own cache like I do with fonts, but I haven't been able to figure out how to.

```
// CSS caching strategy
const cssHandler = workbox.strategies.cacheFirst({
```

```
  cacheName: 'css-cache',
  plugins: [
    new workbox.cacheableResponse.Plugin({
      statuses: [0, 200],
    }),
    new workbox.expiration.Plugin({
      maxAgeSeconds: 60 * 60 * 24 * 14,
      maxEntries: 30,
    }),
  ],
});
```

The Javascript handler will hold our local and third party scripts.

The cache does a couple special things via Workbox plugins:

- It accepts opaque responses for third party resources that wouldn't normally be cached
    - It uses the cacheableResponse plugin
    - The status code '0' represents opaque responses
- It sets an expiration date to 14 days
- It sets the maximum number of items the cache will hold. When the cache is full new entries will push the oldest out of the cache (and delete them)

We cache opaque responses at our own risk since we have no way of knowing if the request succeeded or not. I considered the risk and accepted it because the networks where I'm pulling third party resources from are big and stable (Cloudflare and Google Fonts)

In a future iteration I may want to move third party JS resources to its own cache like I do with fonts, but I haven't been able to figure out how to.

```
const jsHandler = workbox.strategies.staleWhileRevalidate({
  cacheName: 'scripts-cache',
  plugins: [
    new workbox.cacheableResponse.Plugin({
      statuses: [0, 200],
    }),
```

```
    new workbox.expiration.Plugin({
      maxAgeSeconds: 60 * 60 * 24 * 14,
      maxEntries: 30,
    }),
  ],
});
```

As a performance improvement one of the best things I can think about doing is caching images so we only take the hit on the large download the first time that we download an image. Not saying we don't need to optimize them but want to make sure that images load as fast as possible and that means that we should load them from cache if possible.

```
const imageHandler = workbox.strategies.cacheFirst({
  cacheName: 'image-cache',
  plugins: [
    new workbox.expiration.Plugin({
      // Cache for a maximum of 30 days
      maxAgeSeconds: 60 * 60 * 24 * 30,
    }),
  ],
});
```

Fonts will be the largest assets we cache with the service worker so we want to keep a few around and keep them for a while so we don't have to download them as often. This is particularly important because the primary font, Roboto variable font, is 1MB in size when compressed as a woff2 font.

We're keeping 5 locally hosted fonts (ones that are not served through Google fonts) and keeping them for 30 days although I may want to cache them longer and keep more of them, particularly if I work with smaller font subsets.

```
const fontHandler = workbox.strategies.cacheFirst({
  cacheName: 'fonts-cache',
  plugins: [
    new workbox.expiration.Plugin({
      maxAgeSeconds: 30 * 24* 60 * 60,
```

```
      maxEntries: 5,
    }),
  ],
});
```

External resources present an interesting problem when working with service workers. All responses from third party resources up vote are ‘opaque’ responses and represent the result of a request made to a remote origin when CORS is not enabled.

Caching opaque request makes it imposible for Workbox to detect and warn you if the request failed (there is no status code to tell it, and you, that it failed).

Workbox will cache opaque resources when using uses `networkFirst` or `staleWhileRevalidate` as the strategy. Both of this strategies will query the network first (`networkFirst`) or will query the network regardless of success or failure (`staleWhileRevalidate`)

To make sure that the resources are cached we use the cacheableResponse plugin to tell Workbox that we want it to cache resources with a status code of 0, regardless of the strategy we use.

```
const extFontHandler = workbox.strategies.cacheFirst({
  cacheName: 'external-fonts',
  plugins: [
    new workbox.expiration.Plugin({
      maxAgeSeconds: 30 * 24 * 60 * 60,
      maxEntries: 5,
    }),
    new workbox.cacheableResponse.Plugin({
      statuses: [0, 200],
    }),
  ],
});
```

Defining routes

The next step is to create routes that will use the strategies we just defined to

cache the content and, where appropriate, provide fallback options.

These routes use Workbox routing and its `registerRoute` method to associate a file or extension, a handler (defined above) and zero or more additional conditions for each route.

I've broken he routes based on content and, where appropriate, on referring source.

The first type of resource that we cache outside of the precache. We make sure that there is a response and then do one of three things:

- If there is no response then provide an offline page
- If the page is not found then return a 404 page
- If there is an error then give the user an error page

```
// Routing Definitions and Fallbacks
workbox.routing.registerRoute(/.*\.html/, (args) => {
  return con/.*\.html/r.handle(args)
    .then((response) => {
      if (!response) {
        return caches.match('pages/offline.html');
      } else if (response.status === 404) {
        return caches.match('pages/404.html');
      }
      return response;
    })
    .catch((response) => {
      return caches.match('pages/error.html');
    });
});
```

Next, we handle CSS requests and match them to the CSS extension. If there is no response or the response status is 404 we return nothing.

If there is an error we return nothing.

With CSS I don't want to store anything in the cache that is not CSS unlike what we did with HTML where we wanted different responses for each event.

```
//  CSS other than index.css
workbox.routing.registerRoute(/.*\.css/, (args) => {
  return cssHandler.handle(args)
    .then((response) => {
      if (!response || response.status === 404) {
        return;
      }
      return response;
    })
    .catch((response) => {
      return;
    });
});
```

The route that handles Javascript uses the handler we defined earlier without any major adjustments. If the route is not OK it'll produce an error which is what we want. Javascript failuers will usually render the site or app unusable.

```
// JS other than index.js and any JSON file
workbox.routing.registerRoute(/.*\.js/, (args) => {
  return jsHandler.handle(args);
});
```

When caching fonts we want to make sure that the browser caches all font formats that we may want to use (even though we only use woff2 fonts locally) to give ourselves the flexibility of using other formats without having to change the route later.

This will only cache local fonts, not those loaded from Google Fonts; we'll handle third party fonts in a different route.

```
// Fonts
workbox.routing.registerRoute(/.*\.(?:ttf|otf|woff|woff2)/, (args) => {
  return fontHandler.handle(args);
});
```

This route will handle fonts loaded from Google Fonts. The URL may bbe

`fonts.googleapis` or `fonts.gstatic` so we create a new regular expression that matches both sites and stores the fonts in a different cache than our local fonts.

```
// Third party fonts
workbox.routing.registerRoute(/^https:\/\/fonts\.(googleapis|gstatic)\.com
  return extFontHandler.handle(args);
});
```

When we look at images we need to make sure that we're caching all the kinds of images that we plan on using.

The route we're creating includes all image formats exceptWebP since it's not widely supported but we include both GIF and SVG.

```
// Images.
workbox.routing.registerRoute(/.*\.(?:png|jpg|jpeg|svg|gif)/, (args) => {
  return imageHandler.handle(args);
});
```

Injecting the pre-cache items

The final step is to incorporate the precache files that we defined in `workbox-config.js` into the service worker file.

The command requires you to install the workbox node module globally

```
npm install workbox-cli --global
```

The module makes the `workbox` command available on the terminal. Once installed, run the following command to inject the files we want to precache into the service worker.

```
workbox injectManifest workbox-config.js
```

Additional Enhancements

There are a few things that I'm working on beyond the service worker that we described in the previous sections.

Local Videos

We are still using animated GIFs to create animations and demos. We may be able to create smaller animations using MP4 video instead of GIF but until we do the GIF images stay in this cache. We'll use the video cache to store any other videos that we download locally, not from Youtube or Vimeo.

```
const videoHandler = workbox.strategies.cacheFirst({
  cacheName: 'videos-cache',
  plugins: [
    new workbox.expiration.Plugin({
      maxAgeSeconds: 180 * 24 * 60 * 60,
    }),
  ],
});
```

The route will match 4 types of video: WebM, MP4, OGG and AV1. These videos will download locally, not from Youtube or Vimeo. We'll handle that in a different route

```
// videos.
workbox.routing.registerRoute(/.*\.(?:webm|mp4|ogg|mkv)/, (args) => {
  return videoHandler.handle(args);
});
```

Third Party Videos

The previous route handles video loaded using the `video` tag hosted locally and using one of the formats that are common for HTML video.

The problem is that this may be too quick to fill out the storage quota for the domain. I'm still torn between using this strategy that will load from cache and then fall back to the network and using a netowrk only strategy to save bandwidth and disk space.

```
const extVideoHandler = workbox.strategies.cacheFirst({
  cacheName: 'external-video',
    plugins: [
      new workbox.expiration.Plugin({
        maxAgeSeconds: 180 * 24 * 60 * 60,
      }),
      new workbox.cacheableResponse.Plugin({
        statuses: [0, 200],
      }),
    ],
});
```

The regular expression used in this route matches the pages and the embeds for both Vimeo and Youtube.

```
workbox.routing.registerRoute(/^(https:\/\/)
(player\.vimeo.com|www\.youtube\.com)
\/([\w\/]+)([\?].*)?$/, (args) => {
  return extVideoHandler(args);
});
```

Because the regular expression is more complex than I'm used to I've chosen to document it to make sure that I remember what it's supposed to do.

- ^ asserts position at start of a line
- 1st Capturing Group (https://)
    - `https` : matches the characters https: literally (case sensitive)
    - `\/` matches the character / literally (case sensitive)
    - `\/` matches the character / literally (case sensitive)
- 2nd Capturing Group `(player\.vimeo.com|vimeo\.com|www\.youtube\.com)`
- 1st Alternative player.vimeo.com
    - `player` matches the characters player literally (case sensitive)
    - `\.` matches the character . literally (case sensitive)
    - `vimeo` matches the characters vimeo literally (case sensitive)
    - `.` matches any character (except for line terminators)
    - `com` matches the characters com literally (case sensitive)
- 2nd Alternative `vimeo\.com`

        - `vimeo` matches the characters vimeo literally (case sensitive)
        - `\.` matches the character . literally (case sensitive)
        - `com` matches the characters com literally (case sensitive)
    - 3rd Alternative `www\.youtube\.com`
        - `www` matches the characters www literally (case sensitive)
        - `\.` matches the character . literally (case sensitive)
        - `youtube` matches the characters youtube literally (case sensitive)
        - `\.` matches the character . literally (case sensitive)
        - `com` matches the characters com literally (case sensitive)
        - `\/` matches the character / literally (case sensitive)
    - 3rd Capturing Group `([\w\/]+)`
        - Match a single character present in the list below [\w/]+
        - + Quantifier — Matches between one and unlimited times, as many times as possible, giving back as needed (greedy)
        - `\w` matches any word character (equal to [a-zA-Z0-9_])
        - `\/` matches the character / literally (case sensitive)
    - 4th Capturing Group ([?].*)?
        - ? Quantifier — Matches between zero and one times, as many times as possible, giving back as needed (greedy)
        - Match a single character present in the list below [?]
        - `\?` matches the character ? literally (case sensitive)
        - `.*` matches any character (except for line terminators)
        - $ asserts position at the end of a line

“Vanilla” Javascript, no libraries

The idea is to create the service worker using only native APIs and seeing how much of the Workbox functionality I can duplicate without having to use a library.

Constants and common items

Individual constants for each of the caches. We store them separately so we can change the names to make cleanup easier without having to change the array we define later.

```
const PRECACHE = 'precache-v1';
const CONTENT = 'content-cache-v1';
const CSS = 'css-cache-v1';
```

```
const JS = 'scripts-cache-v1';
const IMAGES = 'image-cache-v1';
const FONTS = 'fonts-cache-v1';
```

expectedCaches is the array of cache names we expect to exists in the client.

If they don't we're fine, we fetch from network; If they do, we're fine, we fetch from cache and if they exist with a different version then we delete the old one

```
const expectedCaches = [
  PRECACHE,
  CONTENT,
  CSS,
  JS,
  IMAGES,
  FONTS,
];
```

`urlsToPrecache` is the list of URLs that we want to cache on install. The `/` URL indicates the root file, in this case, index.html

```
const urlsToPrecache = [
  '/index.html',
  '/css/index.css',
  '/js/zenscroll.js',
  '/404.html',
  '/offline.html',
];
```

Install Event

The install event will take all the URLs in the `urlsToPrecache` array

```
self.addEventListener('install', (event) => {
  self.skipWaiting();
```

```
  console.log('install event fired');
  event'install event fired'n(PRECACHE).then((precache) => {
    return precache.addAll(urlsToPrecache);
  })); // ends wait until
}); // ends install event
```

Activate Event

Activate will perform cleanup on the caches.

If there is a cache that doesn't exist in the Array that we pass to the event then it's deleted This is the reason why we went through such a convoluted way to define the caches. We can change the names individually and then they'll get deleted the next time the user visits the site.

```
self.addEventListener('activate', (event) => {
  clients.claim();
  event.waitUntil(
  caches.keys().then((keys) => {
    Promise.all(keys.map((key) => {
      // if the cache is not one in the list
      if (!expectedCaches.includes(key)) {
        // delete it
        return caches.delete(key);
      }
    }))
    .then(() => {
      console.log('Everything cleaned up');
    });
  }));
});
'Everything cleaned up'
```

Fetch Event

The fetch event is where most of the work will happen. Unlike the work in Workbox service worker we don't break the routes into different blocks in the vanilla service

worker.

There are some aspects where the vanilla service worker still doesn't match the Workbox implementation.

All the following sections are inside the fetch event listener. I've broken them down for ease of reading.

This is an important item to remember for all fetch handlers that put resources in caches using the cache API:

Put a copy of the response in the cache, otherwise the code will throw an exception because response is a stream that can only be consumed once

When we define the event we do two things right away:

We define a constant to hold the vaslues of the event's request object.

If the request doesn't match the GET HTTP method we return without doing anything. The service worker will only work with GET requests.

```
self.addEventListener('fetch', (event) => {
  const request = event.request;

  if (request.method !== 'GET') {
    return;
  }
```

The first handler is for fonts. Since the fonts are requested from the `typography-*.css` stylesheets I have to make sure that the font is loaded from the stylesheet or that the file ends in one of the four font formats I work with.

```
  if (request.url.match(/\.(ttf|otf|woff|woff2)$/) ||
    (request.referrer.includes('typography'))) {
    event.waitUntil(
      // Open the fonts cache
      caches.open(FONTS)
        .then((cache) => {
          // return the font to the user
```

```
        return cache.match(request);
      })
      .then((response) => {
        // Open the cache
        caches.open(FONTS)
        .then((cache) => {
          // Fetch the resource from the network
          return fetch(request)
          .then((response) => {
            // Put a copy of the resource in the cache
            return cache.put(request, response.clone())
            .then(() => {
              return response;
            });
          });
        });
      })
  );
}
```

Caching Javascript resources has a different objective: To cache all the Javascript files that are not in the install precache. The match query means include all files with a `.js` extension except one that includes `zenscroller` in the URL (Zenscroller is cached at install).

```
if (request.url.match(/\.(js)$/) && (!request.url.includes('zenscroll'))
  event.waitUntil(
    caches.open(JS)
    .then((cache) => {
      return cache.match(request);
    })
    .then((response) => {
      caches.open(JS)
      .then((cache) => {
        fetch(request)
        .then((response) => {
          return cache.put(request, response.clone())
```

```
            .then(() => {
              // Return the response
              return response;
            });
        });
      })
  );
  return;
}
```

For CSS we want to make sure we cache all files with a `.css` extension and that are not fonts (denoted by a `.woff2` extension).

```
// two places. Working on figuring out a solution
if (request.url.match(/\.(css)$/) &&
  !request.url.includes('woff2')) /\.(css)$/t.waitUntil(
    'woff2'// Open the content cache
    caches.open(CSS)
      .then((cache) => {
        // return the CSS to the user
        return cache.match(request);
      })
      .then((response) => {
        // Open the cache
        return caches.open(CSS)
        .then((cache) => {
          // Fetch the resource from the network
          fetch(request).then((response) => {
            // Put a copy of the response in the cache
            return cache.put(request, response.clone())
            .then(() => {
              // Return the response
              return response;
            });
          });
        });
```

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

In the cache for images we want to make sure that we add only images only (those that match `jpeg`, `jpg`, `png`, `gif` and `svg`) and not the assets that reference the images (HTML and CSS).

If the image is not in the cache and we can't retrieve it from the network we provide a local fallback as a new response using an SVG image.

```
if (request.url.match(/\.(jpe?g|png|gif|svg)$/) &&
  (!request.url.match('/\.(html|css)$/'))) {
  event.waitUntil(
    caches.open(IMAGES)
    .then((cache) => {
      // return the IMAGES to the user
      return caches.match(request);
    })
    .then((response) => {
      // Open the cache
      return caches.open(IMAGES)
      .then((cache) => {
        // Fetch the resource from the network
        fetch(request)
        .then((response) => {
          // Put a copy of the response in the cache
          return cache.put(request, response.clone())
          .then(() => {
            // Return the response
            return response;
          });
        })
        .catch((error) => {
          return new Response(OFFLINESVG, {
            headers: {
```

```
              'Content-Type': 'image/svg+xml',
            },
          });
        });
      });
    })
  );
  return; 'Content-Type'// If we get to here, bail out
}
```

The final handler for assets is for HTML content by using a stale while revalidate strategy, we return the resource in the cache, if it's not in the cache then we fetch it from the network, store a copy of the response in the cache and return the resource to the user.

In this cache we could be more sophisticated and return different responses based on whether we're offline, the resource was not found or any other network failure. But for an MVP, this is enough.

```
if (event.request.headers.get('Accept')
  .includes('text/html')) {
  'text/html'// Open the content cache
  caches.open(CONTENT)
  .then((cache) => {
    // return the content to the user
    return caches.match(request);
  })
  .then((response) => {
    // Open the cache
    caches.open(CONTENT)
    .then((cache) => {
      // Fetch the resource from the network
      fetch(request)
      .then((response) => {
        // Put a copy of the response in the cache
        cache.put(request, response.clone())
        .then(() => {
          // Return the response
```

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