Service Workers: Third-party libraries

Libraries like Workbox and Upup seek to abstract the service worker code to make it easier for people to use and to create production-ready service workers.

These examples will use the `Workbox` libraries to illustrate how it works and how to make it easier to configure and run service workers.

Installing the necessary Node packages

Workbox is a Node-based application so we need to install the corresponding packages.

We install the CLI first and then we install the other Workbox packages.

We first install the Workbox CLI as a development dependency.

```
npm i -D workbox-cli
```

We install the remaining Workbox packages as dependencies.

```
npm i precacheAndRoute \
registerRoute \
setDefaultHandler \
setCatchHandler \
StaleWhileRevalidate \
CacheFirst \
CacheableResponsePlugin \
ExpirationPlugin
```

The script

The first block consist of cosntants hollding information about the worker.

```
importScripts('https://storage.googleapis.com/workbox-cdn/releases/6.0.2/w

const {
  precacheAndRoute
} = workbox.precaching;
const {
  registerRoute,
  setDefaultHandler,
  setCatchHandler
} = workbox.routing;
const {
  StaleWhileRevalidate,
  CacheFirst
} = workbox.strategies;
const {
  CacheableResponsePlugin
} = workbox.cacheableResponse;
const {
  ExpirationPlugin
} = workbox.expiration;
```

I've broken the Workbox service worker into different sections to better explain them as we go

The first block deals with importing all the packages we will use in the service worker.

We first use importScripts to load the Workbox script synchronously to make sure that it is loaded before we move forward.

We then require the specific Workbox modules that we will use in the document.

```
importScripts('https://storage.googleapis.com/workbox-cdn/releases/6.0.2/w

const {
  precacheAndRoute
} = workbox.precaching;
const {
  registerRoute,
  setDefaultHandler,
  setCatchHandler
} = workbox.routing;
const {
  StaleWhileRevalidate,
  CacheFirst
} = workbox.strategies;
const {
  CacheableResponsePlugin
} = workbox.cacheableResponse;
const {
  ExpirationPlugin
} = workbox.expiration;
```

The first item is to use precacheAndRoute that will list the items that we want to precache.

The `self.__WB_MANIFEST` will be replaced with a list of files to precache. We'll build this list in the command line.

```
precacheAndRoute(self.__WB_MANIFEST);
```

For each of the caches that we want to create there are three things that we need to do:

List the strategy that we want to use.

Provide a name for the cache and list all the plugins that we want to use.

The first route will handle documents with `htm`, `html` and php extensions.

We will use the expiration plugin to restrict the maximum age in seconds for the content in this cache, to restrict the maximum number of items in the cache and to signal that it's ok to purge the cache if we get a quota error.

```
registerRoute(({url}) => url.endsWith(['html', 'htm', 'php']),
  new CacheFirst({
    cacheName: 'Content',
    plugins: [
      new ExpirationPlugin({
        maxAgeSeconds: 30 * 24 * 60 * 60,
        maxEntries: 30,
        purgeOnQuotaError: true,
      }),
    ],
  })
);
```

The cache for CSS stylesheets is very similar to the content cache and uses the same plugin to control the data in the cache.

```
registerRoute(({url}) => url.endsWith('css'),
  new StaleWhileRevalidate({
    cacheName: 'CSS Styles',
    plugins: [
      new ExpirationPlugin({
        magAgeSeconds: 30 * 24 * 60 * 60,
        maxEntries: 30,
        purgeOnQuotaError: true,
      }),
    ],
  })
);
```

The cache for Javascript files introduces a few changes.

We can't control whether an external resource will be served with CORS headers. If it's not served with CORS headers, the response code will be 0 and you won't be able to ssee into the response object at all.

By default, Workbox will not cache these opaque responses unless you use the cacheableResponse plugin to force caching.

We use the `expiration` plugin to restrict the age of the documents in the cache, the number of entries we'll cache and whether the cache will be purged if we go over quota.

```
registerRoute(({url}) => url.endsWith('js') ||
url.endsWith('mjs'),
  new CacheFirst({
    cacheName: 'scripts',
    plugins: [
      new CacheableResponsePlugin({
        statuses: [0, 200],
      }),
      new ExpirationPlugin({
        maxAgeSeconds: 30 * 24 * 60 * 60,
        maxEntries: 30,
        purgeOnQuotaError: true,
      }),
    ],
  })
);
```

The Google Fonts and images caches use the same techniques as the previous route:

- Both use the CacheFirst strategy
- They cache opaque responses
- They have an expiration time
- They have a maximum number of entries
- They purges the cache on quota error.

```
registerRoute(({url}) => {
    url.origin === 'https://fonts.googleapis.com/' ||
    url.origin === 'https://fonts.gstatic.com';
  },
  new CacheFirst({
```

```
    cacheName: 'Google Fonts',
    plugins: [
      new CacheableResponsePlugin({
        statuses: [0, 200],
      }),
      new ExpirationPlugin({
        maxAgeSeconds: 120 * 24 * 60 * 60,
        maxEntries: 50,
        purgeOnQuotaError: true,
      }),
    ],
  })
);

registerRoute(({url}) => url.endsWith([
  'png',
  'jpg',
  'webp',
  'avif',
  'heic',
  'svg']),
  new CacheFirst({
    cacheName: 'images',
    plugins: [
      new CacheableResponsePlugin({
        statuses: [0, 200],
      }),
      new ExpirationPlugin({
        maxAgeSeconds: 30 * 24 * 60 * 60,
        maxEntries: 30,
        purgeOnQuotaError: true,
      }),
    ],
  })
);
```

The default handler will use the StaleWhileRevalidate strategy. Meaning that the content will be pulled from the cache while available and, at the same time, will be

fetched from the network and put in a cache.

```
// Set default caching strategy for everything else.
setDefaultHandler(new StaleWhileRevalidate());
```

`setCatchHandler` will handle any elements that are not in the cache and the network is not available.

We test the type of document returned by the request using request.destination and provide different responses based on the type of document.

`document` : Use matchPrecache to find an load the default offline page.

`image` : Return an new response containing a 400x300 SVG image

For any other content we just generate an error as the response

```
setCatchHandler(({event}) => {
  switch (event.request.destination) {
    case 'document':
      return matchPrecache('pages/offline.html');
    break;

    case 'image':
      return new Response('pages/offline.html'`<svg role="img"
        aria-labelledby="offline-title"
        viewBox="0 0 400 300"
        xmlns="http://www.w3.org/2000/svg">
        <title id="offline-title">Offline</title>
        <g fill="none" fill-rule="evenodd">
        <path fill="#D8D8D8" d="M0 0h400v300H0z"></path>
        <text fill="#9B9B9B"
            font-family="Helvetica Neue,Arial,Helvetica,sans-serif"
            font-size="72" font-weight="bold">
        <tspan x="93" y="172">offline</tspan></text></g>
      </svg>`,
    {
```

```
      headers: {
        'Content-Type': 'image/svg+xml',
        'Cache-Control': 'no-store',
      },
    });
    break;

    default:
      return Response.error();
  }
});
```

SO far we've written the service worker but we're not done. When we set up the worker we used a place holder (`self.__WB_MANIFEST`) where we want to insert the list of files we want to precache but, before we can use it, we have to tell Workbox what files we want to insert.

To do this we'll use the injectManifest command.

To do this, we create a precache manifest file (`workbox-config.js`) where we'll store the settings for compiling our service worker, including the list of files we want to precache.

Because we're inserting the precache patterns and not building the service worker, we need a source (`swSrc`) for the service worker and a destination (`swDest`) for the completed service worker.

```
module.exports = {
  'globDirectory': 'docs',
  'globPatterns': [
    '/',
    'index.html',
    'css/index.css',
    'js/zenscroll.min.js',
    'pages/404.html',
    'pages/offline.html',
  ],
  'swSrc': 'js/sw2.js',
```

```
  'swDest': 'sw.js',
};
```

Once we write the configuration file, we need to make sure the you installed Workbox CLI from NPM

> Once the package is installed, we can run the following

Additional Resources

Jeremy Keith — Going Offline

> Jason Grigsby — Progressive Web Apps