PWA workshop

Part 2

Workbox

Using Workbox

1. Introduction

If you find yourself regularly writing code in your Service Workers that caches resources, you might find Workbox (https://developers.google.com/web/tools/workbox/) helpful. Written by one of the teams at Google, it's a library of helpers to get you started creating your own Service Workers in no time, with built-in handlers to cover the most common network strategies. In a few lines of code, you can decide whether you want to serve specific resources solely from the cache, serve resources from the cache and then fall back, or perhaps only return resources from the network and never cache. This library gives you total control over your caching strategy.

Workbox provides you with a quick and easy way to reuse common network caching strategies instead of rewriting them again and again. For example, say you wanted to ensure that you always retrieve your CSS files from the cache but only fall back to the network if a resource wasn't available. Using Workbox, you register your Service Worker the same way you have in the `Introduction to Service Workers` code lab. Then you import the library into your Service Worker file and start defining routes that you want to cache.

What You'll Learn

- ✓ How to write a service worker using the **workbox-sw.js** library
- ✓ How to add routes to your service worker using **workbox-sw.js**
- ✓ How to use the predefined caching strategies provided in **workbox-sw.js**
- ✓ How to augment the **workbox-sw.js** caching strategies with custom logic
- ✓ How to generate a production-grade service worker with `workbox-cli`

2. Getting set up

Project Set Up

In this code lab, we are building on top of the project started in the `Introduction to Service Workers` code lab.

If you didn't do it already: **Fork** and then **Clone** the following repository: `https://github.com/The-Guide/fe-guild-2019-pwa.git`

```
$ git clone https://github.com/[YOUR GITHUB PROFILE]/fe-guild-2019-pwa.git
$ cd fe-guild-2019-pwa
```

If you want to start directly with `Workbox` checkout the following branch:

```
$ git checkout pwa-workbox-init
```

First install the dependencies

```
$ npm install
```

Then type in the terminal

```
$ npm start
```

and open Chrome at `localhost:8080fe-guild-2019-pwa/`

Install extra dependencies

Navigate to the **project** directory and run from the command line:

```
$ npm install workbox-cli --save-dev
```

For some Windows version, it can hang so install the latest beta instead

```
$ npm install workbox-cli@beta --save-dev
```

`workbox-cli` is a command-line tool that lets us configure, generate, and modify service worker files. You'll learn more about this in a later step.

`workbox-cli` can be run from command line using `npx` that ships with newer versions of `Node` and `NPM`

```
$ npx workbox
```

If `npx` is not present you can install `workbox-cli` globally (with the --global flag)

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

3. Create a basic service worker

Rename the sw.js from the previous code lab and create a new empty one with the same name (sw.js)

ADD in sw.js:

```
importScripts('https://storage.googleapis.com/workbox-cdn/releases/3.5.0/workbox-sw.js');
if (workbox) {
 console.log(`Yay! Workbox is loaded 🎉`);
 workbox.precaching.precacheAndRoute([]);
} else {
 console.log(`Boo! Workbox didn't load 😬`);
}
```

Explanation

The `importScripts` call imports the **workbox-sw.js** library from a Content Delivery Network (CDN). Once the library is loaded, the `workbox` object gives our service worker access to all the Workbox modules.

The `precacheAndRoute` method of the `precaching` module takes a precache "manifest" (a list of file URLs with "revision hashes") to cache on service worker installation. It also sets up a cache-first strategy for the specified resources, serving them from the cache by default.

Currently, the array is empty, so no files will be cached.

Rather than adding files to the list manually like with did in the previous code lab, `workbox-cli` can generate the manifest for us. Using a tool like `workbox-cli` has multiple advantages:

1. The tool can be integrated into our build process. Adding `workbox-cli` to our build process eliminates the need for manual updates to the precache manifest each time that we update the app's files.
2. `workbox-cli` automatically adds "revision hashes" to the files in the manifest entries. The revision hashes enable Workbox to intelligently track when files have been modified or are outdated and automatically keep caches up to date with the latest file versions. Workbox can also remove cached files that are no longer in the manifest, keeping the amount of data stored on a user's device to a minimum. You'll see what `workbox-cli` and the file revision hashes look like in the next section.

> After you're done testing rename the **sw.js** file to **sw-template.js**

4. Configure Workbox

The first step towards injecting a precache manifest into the service worker is configuring which files we want to precache. In this step, we create the `workbox-cli` configuration file.

From the **project** directory, run the following command:

```
$ npx workbox wizard --injectManifest
```

Next, follow the command-line prompts as described below:

1. The first prompt asks for the `root` of the app. The root specifies the path where Workbox can find the files to cache. For this lab, the root is the **fe-guild-2019-pwa/** directory, which should be suggested by the prompt. You can either type "fe-guild-2019-pwa/" or choose "fe-guild-2019-pwa/" from the list.
2. The second prompt asks what types of files to cache. For now, choose to cache CSS files only.
3. The third prompt asks for the path to your source service worker. This is the service worker file, **sw.js**, to which we added code in the previous step. Type "fe-guild-2019-pwa/sw-template.js" and press return.
4. The fourth prompt asks for a path in which to write the production service worker. You can press return.
5. The final prompt asks what we want to name our configuration file. Press return and use the default answer (**workbox-config.js**).

Once you've completed the prompts, you'll see a log with instructions for building the service worker. Ignore that for now (if you already tried it, that's okay). We will explore in the next step.

Let's examine the newly created **workbox-config.js** file.

```
module.exports = {
  "globDirectory": "fe-guild-2019-pwa/",
  "globPatterns": [
      "**/*.{html,ico,json,css,js}",
      "src/images/*.{jpg,png}"
  ],
  "swDest": "fe-guild-2019-pwa/sw.js",
  "swSrc": "fe-guild-2019-pwa/sw-template.js",
    "globIgnores": [
        "../workbox-config.js",
        "sw-template.js",
        "help/**"
    ]
};
```

Workbox creates a configuration file (in this case **workbox-config.js**) that `workbox-cli` uses to generate service workers. The config file specifies where to look for files (`globDirectory`), which files to precache (`globPatterns`), and the file names for our source and production service workers (`swSrc` and `swDest` , respectively). We can also modify this config file directly to change what files are precached. We explore that in the later step.

5. Inject a manifest into the service worker

Now let's use the `workbox-cli` tool to inject the precache manifest into the service worker.

Open **package.json** and create a `build` script to run the Workbox `injectManifest` command. The updated package.json should look like the following:

```
"scripts": {
    "start": "http-server -c0",
    "build": "workbox injectManifest workbox-config.js",
    "deploy": "npm run build && gh-pages -d fe-guild-2019-pwa"
}
```

Save the file and run `npm run build` from the command line.

The `precacheAndRoute` call in **sw.js** has been updated. In your text editor, open **sw.js** and observe that all the **CSS** files are included in the file manifest.

Return to the app in your browser (http://localhost:8080/fe-guild-2019-pwa/). Open your browser's developer tools (in Chrome use `Ctrl+Shift+I` on Windows, `Cmd+Opt+I` on Mac). Unregister the previous service worker and clear all service worker caches for localhost so that we can test our new service worker. In Chrome DevTools, you can do this in one easy operation by going to the Application tab, clicking Clear Storage and then clicking the Clear site data button.

Refresh the page and check that a new service worker was installed. You can see your service workers in Chrome DevTools by clicking on **Service Workers** in the **Application** tab. Check the cache and observe that all **CSS** files are stored. In Chrome DevTools, you can see your caches by clicking on Cache Storage in the Application tab.

Explanation

When `workbox injectManifest` is called, Workbox makes a copy of the source service worker file (**fe-guild-2019-pwa/sw-template.js**) and injects a manifest into it, creating our production service worker file (**fe-guild-2019-pwa/sw.js**). Because we configured **workbox-config.js** to cache `*.css` files, our production service worker has all the CSS files in the manifest. As a result, all the CSS files were pre-cached during the service worker installation.

Now whenever we update our app, we can simply run `npm run build` to update the service worker.

6. Reinject an updated manifest

Let's modify the Workbox config file to precache our entire home page. Replace the contents of **workbox-config.js** with the following code, and save the file:

```
module.exports = {
  "globDirectory": "fe-guild-2019-pwa/",
  "globPatterns": [
      "**/*.{html,ico,json,css,js}",
      "src/images/*.{jpg,png}"
  ],
  "swDest": "fe-guild-2019-pwa/sw.js",
  "swSrc": "fe-guild-2019-pwa/sw-template.js",
    "globIgnores": [
        "../workbox-config.js",
        "sw-template.js",
        "help/**"
    ]
};
```

From the **project** directory, re-run `npm run build` to update **sw.js**. The precache manifest in the production service worker (**sw.js**) has been updated to contain all the static files we had previously.

Refresh the app and activate the updated service worker in the browser. In Chrome DevTools, you can activate the new service worker by going to the **Application** tab, clicking Service Workers and then clicking **skipWaiting**. Observe in developer tools that the `globPatterns` files are now in the cache (you might need to refresh the cache to see the new additions).

If you enable `Offline mode` from the **Network** tab the home page loads but not the fonts, we will fix that in the next step.

Explanation

By editing the `globPatterns` files in **workbox-config.js**, we can easily update the manifest and precached files. Re-running the `workbox injectManifest` command (via `npm run build`) updates our production service worker with the new configuration.

In addition to precaching, the `precacheAndRoute` method sets up an implicit cache-first handler. This is why the home page loaded while we were offline even though we had not written a fetch handler for those files!

7. Add routes to the service worker

workbox-sw.js has a routing module that lets you easily add routes to your service worker.

Let's fix the fonts problem now. Copy the following code into **sw-template.js** beneath the ***precacheAndRoute*** call. Make sure you're **not** editing the production service worker, **sw.js**, as this file will be overwritten when we run ***workbox injectManifest*** again.

ADD this in sw-template.js

```
workbox.routing.registerRoute(
 /.*(?:googleapis|gstatic)\.com.*$/,
 workbox.strategies.staleWhileRevalidate({
     cacheName: 'google-fonts',
     plugins: [
         new workbox.expiration.Plugin({
             maxEntries: 3,
             maxAgeSeconds: 30 * 24 * 60 * 60 // 30 Days
         })
     ]
 }));
```

Restart the server and rebuild the app and service worker with the following commands:

```
npm run build
```

```
npm run start
```

Refresh the app and activate the updated service worker in the browser. Put the app offline and check that the Google Fonts load correctly now. You may need to refresh twice.

Explanation

In this code, we added a route to the service worker using the registerRoute method on the routing class. The first parameter in registerRoute is a regular expression URL pattern to match requests against. The second parameter is the handler that provides a response if the route matches. In this case, the route uses the strategies class to access the staleWhileRevalidate run-time caching strategy. Whenever the app requests the Google Fonts, the service worker checks the cache first for the resource before going to the network.

The handler in this code also configures Workbox to maintain a maximum of 3 entries in the cache. Once 3 entries have been reached, Workbox will remove the oldest one automatically. The fonts are also set to expire after 30 days, signaling to the service worker that the network should be used for those entries.

8. Dynamic Caching and Offline HTML Fallback

In this last step we will bring the service worker to the same level we left it in the `Introduction to Service Workers` code lab. For that we will register a new route with a custom handler

```
workbox.routing.registerRoute(
 routeData => routeData.event.request.headers.get('accept').includes('text/html'),
 args => {
     return caches.match(args.event.request)
         .then(response => {
             if (response) {
                 console.log(response);
                 return response;
             }
             // Clone the request - a request is a stream and can be only consumed once
             const requestToCache = args.event.request.clone();
             // Try to make the original HTTP request as intended
             return fetch(requestToCache)
                 .then(response => {
                     // If request fails or server responds with an error code, return that error immediately
                     if (!response || response.status !== 200) {
                         return response;
                     }
                     // Again clone the response because you need to add it into the cache and because it's used
                     // for the final return response
                     const responseToCache = response.clone();
                     caches.open('dynamic')
                         .then(cache => {
                             cache.put(requestToCache, responseToCache);
                         });
                     return response;
                 });
         })
         .catch(error => {
             return caches.match('/fe-guild-2019-pwa/offline.html');
         });
 }
);
```

Exercises

1. Compare the route above with the `fetch` callback from the **Introduction to Service Workers** code lab. See if you spot any differences
2. Write your own routes that caches the icons and splash screens. The route should match requests to `/images/icons/*` and `/images/splashscreens/*` and handle the request/response using the `staleWhileRevalidate` strategy. Name the caches and think about values for the expiration plugin.
3. Deploy to `GitHub Pages` and test again.

Solution

```
$ git checkout pwa-workbox-final
```

PUSH

Exercise and play

https://serviceworke.rs/push-simple_demo.html