Service Workers: Working offline and updating your content

One of the things that has caught my attention are service workers and the capabilities that they bring to the web.

Installing the service worker

In your site's index page we register the service worker. We only need to do it once in the index page to register the worker for the entire site.

The script will check if the browser supports service workers and only runs the installation code if they are.

We register the service worker using `navigator.serviceWorker.register` with the full path to the service worker as a parameter. It returns a promise that will resolve if the registration is sucessful and reject if it's not.

```
if ('serviceWorker' in navigator) {
  navigator.serviceWorker.register('/sw.js').then(function() {
    console.log('ServiceWorker succesfully registered');
  }).catch(function(err) {
    console.log('ServiceWorker registration failed: ', err);
  });
}
```

In the service worker (`sw.js`) we write the service worker itself.

We split the full cache name in two parts: The name and the version. When creating a new version of the service we only update the version; this will force the browser to install the new version.

The `assetsToCache` array contains all the items that we want to cache before we worry about the pages and their content; things like main JS and CSS files, possibly fonts, and things that we want to have available when the page loads.

```
const cacheName = 'assets';
const cacheVersion = 'v1'
const fullCacheName ='v1'`${cacheName}-${cacheVersion}`;

const assetsToCache = [
  '/css/main.css',
  '/js/main.js',
  '/fonts/raleway.woff2'
];
```

We then cycle through the service worker's lifecycle events and take action where appropriate.

The first event is install. This is a good place to load essential assets for the site.

The old version of the service worker is still handling things so the install event shouldn't affect the existing site.

```
self.addEventListener('install', function(event) {
  event.waitUntil(
    caches.open(fullCacheName).then(function(cache) {
        return cache.addAll(assetsToCache);
    }).then(function() {
      return self.skipWaiting();
    })
  );
});
```

The next event is the installation. Our new service worker has taken over but we haven't started fetching the site's resources so this is the best place to do housekeeping like applying migrations for any IndexedDB databases or deleting old caches.

waitUntil will keep the task from terminating until it is completed.

This example will do delete all the caches that are not used by the current service worker. This way we prevent bloat by deleting unused content.

```
self.addEventListener('activate', function (event) {
  event.waitUntil(
    cache.keys().then(function (cacheNames) {
      return Promise.all(
        cacheNames
          .filter(function (cacheName) {
          })
          .map(function (cacheName) {
            return caches.delete(cacheName);
          }),
      );
    }),
  );
});
```

```
self.addEventListener('fetch', function(event) {
  if (event.request.method !== 'GET') { return; }
    if (/http:/.test(event.request.url)) { return; }

  event.respondWith(
    caches.open(ca/http:/).then(function(cache) {
      return fetch(event.request).then(function(networkResponse) {
        cache.put(event.request, networkResponse.clone());
        return networkResponse;
      }).catch(function() {
        return cache.match(event.request);
      })
    })
  );
});
```

Updating data and notifying the user

One of the most interesting things, to me, about service workers is how to show users that the content has been updated and give them the option to reload the page to update the service worker.

> Most of the work will be done in `index.html`.

I've broken the code for index.html into two sections to make it easier to walk through it.

The `showUpdateBar` function will display the notification notice.

The click event listener for the `#reload` element will reload the page when activated.

```
let newWorker;

function showUpdateBar() {
  let snackbar = document.getElementById('snackbar');
  snackbar.className = 'show';
}

document.getElementById('reload').addEventListener('click', function(){
  newWorker.postMessage({ action: 'skipWaiting' });
});
```

The service worker registration code will:

1. Check if the browser supports service workers
2. If it does then it register the service worker
3. Add an updatefound event listener
4. When the `updatefound` event listener is triggered load the new service worker
5. Add a statechange
6. Test if there has been a change in the state
7. Test the status of the new service worker. If it's installed then run `showUpdateBar`
8. Add a controllerchange event listener
    1. When the window is reloading return and do nothing
    2. When the window is **not** refreshing then reload it

```
// 1
if ('servi'serviceWorker'vigator) {
  // 2
  navigator.serviceWorker.register('/service-worker.js').then(reg => {
      '/service-worker.js'// 3 istener('updatefound', () => {
        // 4
        newWork'updatefound'// 4
        newWorker = reg.installing;

        // 5er('statechange', () => {
          // 6
          switch (newWorker.'statechange'// 6    // 7
            case 'installed':
          'installed'// 7 r.serviceWorker.controller) {
                showUpdateBar();
              }
              break;
          }
        });
      });
    });

    let refreshing;
    // 8
    navigator.serviceWorker.addEventListener('controllerchange', function
      if (refreshing) return;
      window.location.reload();
      refreshing = true;
    });
  }
'controllerchange'
```

Changes to the service worker are more straightforward. We only need to add a message event to process the data that we get from the page.

If the value of the action message is `skip` then we run skipWaiting to force the waiting service worker to become active.

```
self.addEventListener('message', function (event) {
  if (event.data.action === 'skipWaiting') {
    self.skipWaiting();
  }
});
```

This is an example of what you can do with a combination of your service working loading script and the service worker itself.

There’s more we can do; we’ll look at third party workbox libraries to make the process easier.