Saving individual post for offline access

Most service workers will cache content automatically when you vistit a page based on how the worker is configured, regardless of whether the user wants to access the content offline or not.

Different people, including Una Kravets and Sara Souedain have described their experiences implementing service workers that will only cache content on request.

Overview

The idea of this project is to change the service worker in my layout experiments site to do the following:

1. Precache the resources as it’s currently done
    1. In this case, cache the index page and associated assets
2. For each article on the site:
    1. Provide a link or button for the user to cache the page
    2. When the user clicks the button, cache the page and associated assets

Vanilla Javascript

We can work with service workers in vanilla Javascript. There are many resources available that provide examples to follow. Some of them are:

- Jake Archibald’s Offline Cookbook
- Mozilla’s Service Workers Cookbook
- Una Kravets Implementing “Save For Offline” with Service Workers
- Making a Simple Site Work Offline with ServiceWorker
- Offline Content with Service Worker

Using Una’s post as guidance we’ll build the code to cache posts on demand.

We’ll do the work backwards and begin with the service worker. Because we’re letting users decide what they want to save for offline view, we don’t need as much work as we would in a regular service worker.

The `install` event will precache all our assets, listed in `assetsToCache` the first time we visit the page.

Using `skipWaiting()` and `clients.claim()` makes sure that the service worker takes over the page and site as soon as possible, without waiting for the page to reload.

```
// Service Worker (sw.js)
const precacheName = 'precached-content';
cons'precached-content' 'v1';
const precacheFullName = precacheName + '-' + precacheVersion;

'v1'// Assets to cachest assetsToCache = [
  // '/',
  /'/'
  // 'index.html',/index.css',
  'js/zenscroll.min.js',
  'pages/404.html',
  'pages/offline.html',
];

self.addEventListener('install', function(event) {
  event.waitUntil(
    caches.open(precacheFullName)
    .then(function(cache) {
        return cache.addAll(assetsToC'js/zenscroll.min.js'en(function() {
      return self.skipWaiting();
    })
  );
});

// Activate event
self.a// Activate event
self.addEventListener('activate', function(event) {
  return self.clients.claim();
});
```

The fetch event also helps me start learning how to use async/await in a service worker instead of raw promises. The idea is that in the fetch event we'll only capture the root document (`index.html` or `index`.php). Everything else we've either precached or we will the the user decide if she want to cache it for offline view.

```
// Get current path
  const requestUrl = new URL(event.request.url);

  // Save the index file on origin path only
  if (requestUrl.origin === location.origin && requestUrl.pathname === '/
    event.respondWith(async function() {
      const cache = await caches.open(precacheFullName);
      con'/'cachedResponse = await cache.match(event.request);
      const networkResponsePromise = fetch(event.request);

      event.waitUntil(async function() {
        const networkResponse = await networkResponsePromise;
        await cache.put(event.request, networkResponse.clone());
      }())
      return cachedResponse || networkResponsePromise;
  }());
}); // closes fetch event listener
```

In most service worker registration scripts I see only the first part of the script below. We use a feature query to detect support for service workers, we register one if it's supported and log it to console or log the failure to console and bail.

But in this script we need to do more.

We assume that the page has a button to initiate the caching process (unlike normal service workers where visiting the page will trigger caching). We create three variables to hold information we will use later:

1. The location of the page we want to cache (**window.location.pathname**)
2. The button that will triger the event (**querySelector('.offline')**)
3. A list of all the images in the page (**querySelectorAll('img')**)

If there is an offline button we register a click event handler for it and we build an array of all the elements that we want to chache.

The last step is to put the resouces in the cache. We take advantage that the Cache API is accessible from regular pages **and** from service workers, so we can add content to a cache from both locations.

```
if ('serviceWorker' in navigator) {
  // Attempt to register it
  navigator.serviceWorker.register('/sw.js').then() => {
    '/sw.js'// Successsole.log('ServiceWorker registration successful');
  }).catch((err) => {
    // Fail
'ServiceWorker registration successful'// Fail
    console.log('ServiceWorker registration failed: ', err);
  });

  // Set variables for use in the event listener;
  const cacheButton = document.querySelector('.offline');
  const imageArray = document.querySelectorAll('img');

  // Event listener
  if(cacheButton) {
    cache'.offline'// Event listener
  if(cacheButton) {
    cacheButton.addEventListener('click', (event) => {
     event.preventDefault();
     const pageResources = [currentPath, ...imageArray];

      caches.open('offline-' + currentPath)
      .then((cache) => {
        cache.addAll(pageResources)
        .then(() => {
          console.log('Article available offline.');
        })
        .catch((error) =>{
          console.log('Offline saving failed');
        });
      });
    });
  }
}
```

Letting the user decide if she wants to cache the content can take away the performance advantage of a service worker.

At worst she doesn’t get speedier subsequent visits to the site because she chose not to cache any pages, and at best, she gets partial speeds ups of the content she chose to cache.