The service workers' impact on web applications performance

Dominik Białecki

30-10-2018

1 Problem statement

1.1 Introduction

Web applications today are becoming more and more similar to the desktop ones. The designing of service workers was one of the breakthrough point to achieve this. A service worker is a background script with is working simultaneously with the main application. It can provide a lot of useful features, such as providing Push Notifications support, resource caching, background synchronization or making the application work offline.

1.2 Research goal, metrics, expected conclusions

This research's goal is to measure if applying a service worker to a web application leads directly to it's performance growth. The first metric which is going to be measured is the page load time. Applying a service worker could slow down the very first application load, but should rapidly increase following ones. Furthermore it will be measured how does service workers' caching impacts HTTP response times.

2 Service workers

Service workers are JavaScript files containing a script which is intended to act as a proxy between network, the browser and web applications. These scripts control their host applications extending its' usability. Service workers run separately from the main browser thread and they are independent of the application. Because of these properties developers mostly use them to intercept network requests, cache application data and requests responses and provide push messages which works even when the main thread of browser is closed (there are no opened tabs). However background characteristics of service workers also has some negative consequences. First of all they can not directly make impact on the DOM. They can only communicate with the main application threat by using events and event listeners. Another limitation comes

from the fact of service workers being implemented to be fully asynchronous. Because of this fact can not use neither synchronous XXMLHttpRequests nor localStorage.

3 Service workers' legacy

Before service workers were invented, there were a similar, background working scripts called web workers. Just like service workers, web workers are working in another thread which can perform tasks without interfering with the user interface, so they can perform expensive scripts without freezing it. Actually, service workers are a type of web worker. But where is the difference and what have cause that nowadays these well-earned ancestors are slowly being forgotten?

First of all the fundamental difference between these two is their lifecycle. Because of web workers being intended to provide parallelism, there can be many of them on a one tab, but their lifespan is strictly connected to their tab and they are terminated when it is closed. A service worker is more like a web application valet. Service worker has access to all tabs with it's host application opened. There can also be several service workers connected to the application, but all of them run separately from the application, and work even when application's tab is not opened, so service workers can provide offline support while their ancestors can't.

Another huge difference is (or was) browser support. Due to service workers being rather a fresh feature they are not well-supported by browsers.

Figure 1: Web worker browsers support

Web workers are supported on older browsers and browser versions, such as Internet Explorer 10 and even on not widely-used mobile browsers such as Blackberry Browser or Opera Mobile. Service workers were in Jan 21 2015 on Chrome 40 and even since then not mainly-used browsers have implemented them. Moreover service workers implementations differs on particular browsers. Some of the browsers offers only partial support. For example Safari's service worker can deliver push messages only when the browsers is running and Background Sync API, which provides periodic synchronization is available only on few browsers.

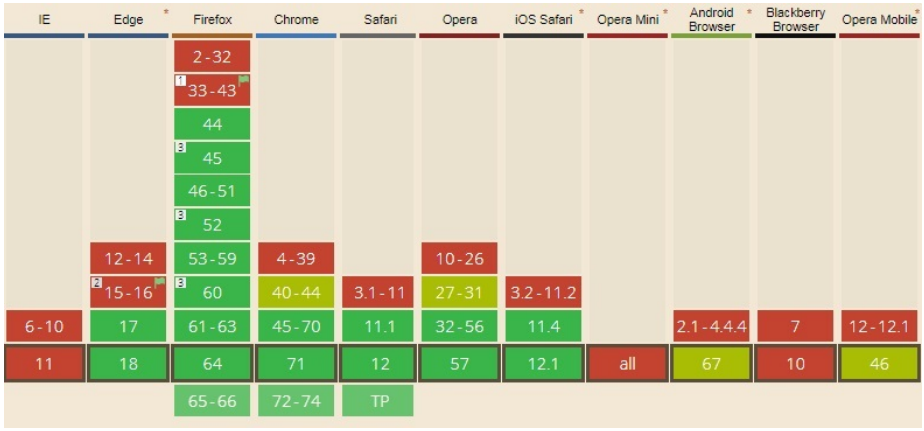

Figure 2: Service worker browsers support

Figure 3: Background Sync API browsers support

4 Test scenarios

To measure service worker's impact on web application there was written a simple web application. The web application have been implemented using Angular framework, and the service worker comes from @angular/service-worker package.

The application has two pages containing tables filled with data which comes from a HTTP GET request. The tables have filter, sort and pagination options. On the first page data transformations are computed in main thread JavaScript code, on the second page computations are made on the service application and retrieved by specifying request's query parameters. Beside those two pages, there is third one, which simply includes 20MB of large images. It's destiny is to have a large load time.

There were also few different versions of the application. One version didn't provide any service worker, and two other were providing service workers with different caching strategies. Measurements have been made on Google Chrome. It provides very useful developer tools which could be used to measure all the metrics and moreover it could artificially slow down network speed to provide even more measurements' scenarios.

5 Implementation

The web application have been implemented using Angular framework, and the service worker comes from @angular/service-worker package. This service worker can be configured by providing specific JSON file. Developer can, inter alia, specify cache size limit, maximal data age and caching strategy. The strategy can be set to *performance* or *freshness*.

Service workers with *performance* strategy try to provide HTTP response as soon as possible. Provided that requested resource exists in the cache, the script uses it as a response even before the API responds. This strategy is suitable for rarely-changing resources, such as user's personal data or avatars.

By contrast the *freshness* strategy is not focused on reducing response times. The application preferentially fetches data from the network and caches responses. This caches responses are used only if response time exceeds specified time. This strategy's main purpose is to make responses come at least fast as developer wants or to work offline.

To emulate conditions similar to production-mode applications, all request-filtering operations had to pass through *debounceTime* function. It ensures that only the last request in some specified time will be emitted. The specified time in the application was 200ms. This operation is necessary to prevent applications from spamming HTTP requests.

6 Measurements

6.1 Page load time

Page load time was measured using the application page with large images. Because service workers are being installed after application loads for the first time, it was important to measure this metric on the very first application visit and after page's resources were cached. Tests were made using Google DevTools' *Audit* tab. While testing page this benchmark simulates Fast 3G network speed and slows down CPU performance 4 times. Then it refreshes the page normally and in few different scenarios, for example while having JavaScript disabled or in offline mode, to examine the application. The metrics which *Audit* provided used to measure page load time were *First Meaningful Paint* and *Time to Interactive*. The first one stands for the time in which the primary content of a page is visible, while the other marks the time in which the page is fully interactive.

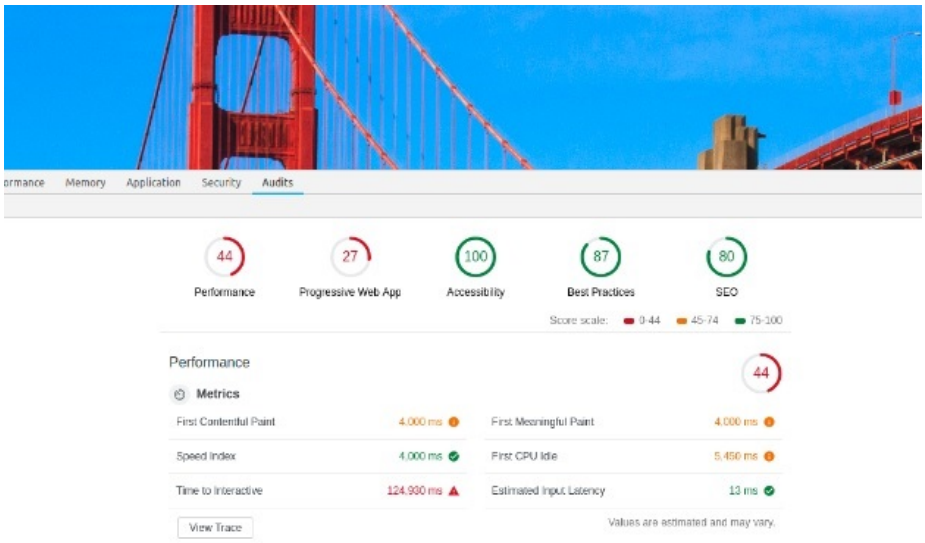

Figure 4: Google DevTools Audit (page loading time without service worker)

There were made three measurements of each scenario, but measurements were exactly the same on every attempt. This was because because *Audit* calculates those metrics in the same way despite of attempts quantity. Furthermore there was no loading time differences on the version without service worker on the very first and next loads.

	No service Worker	Freshness First Load	Freshness	Performance First Load	Performance
1st Meaningful Paint	4	4,02	0,97	4,46	0,49
Time to Interactive	124,93	124,94	1,5	125,54	1,05

Figure 5: Average page loading times [ms]

6.2 Data transformation times

This measurements were made on application pages containing tables. On the application versions with service workers, data was cached before the measurements (those were not first requests). Google DevTools' *Performance* tab was used to get this metric. This tools records all application events, such as scripting, loading or printing, in specified time and then shows its' duration times.

In attached example, to measure filter time, there was selected period between *onkeydown* event and *Update Layer Tree* rendering event.

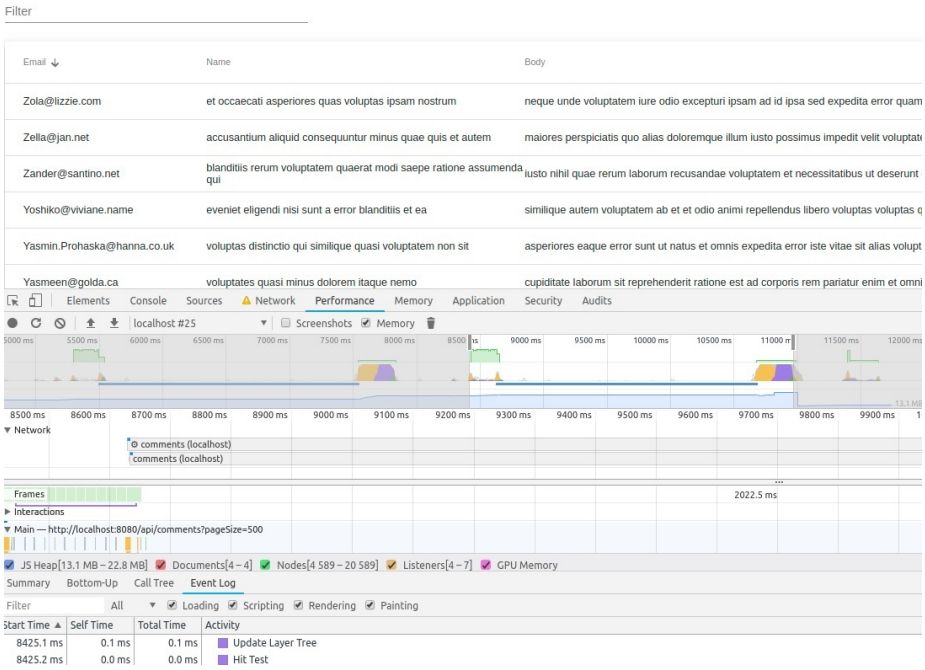

Figure 6: Google DevTools Performance

The API service was deployed on the same computer on which was deployed the web application. The operations which were sorting, paginating and filtering were made on 500 objects. The measurements were made on 3 network speeds (simulated by Google DevTools): normal (unmodified), Fast 3G and Slow 3G.

	Normal	Fast 3G	Slow 3G	Service worker Cache
Avg response time [ms]	108	597	2034	29

Figure 7: HTTP response times

Each scenario was measured 5 times. Network speed doesn't have any impact on measuring computation speeds in case of browser computation and service workers with performance strategy.

	No service worker			Freshness			Performance	Browser computations
Network speed	Normal	Fast 3G	Slow 3G	Normal	Fast 3G	Slow 3G	N/A	N/A
Sort	542	1028	2516	553	1056	2557	354	211
Filter	409	850	2312	441	931	2473	294	48
Paginate	433	876	2349	443	975	2381	314	122

Figure 8: Computation times [ms]

7 Conclusions

7.1 Load times

Caching page resources case cause a very great page load time reduction. The larger is the data provided in web page, the more time can be saved by caching it. The very first page load times don't vary that much between applications with or without service workers. Freshness first load time is negligibly longer then the one without service worker, but arguable could be longer due to fact, that resources have to be passed through the service worker which caches them. Performance first load time is longer than the others, because all of the requests have been first searched in cache, and then redirected to API when they were not found.

If it comes to subsequent page visits, page load times on applications with service workers are drastically shorter than no-cache loads. First meaningful paint have been reduced by over 4 in case of freshness strategy, and by over 8 on performance strategy. That is because this metric doesn't take in count waiting for all the data (large images) to come. Application is saved in the browser, and so it first paints that much faster. Time to interactive metric in measured until the moment, in which all of the images are fully loaded and so it depends on network connection even further. Artificially slowed down network speed and CPU performance caused this metric to be more then 120 times reduced on subsequent page visits on applications with service workers. In this scenario service workers with freshness strategy at first sent requests, and then when responses turned out to be large data, they looked for the resources in cache. This is why the other service workers had better performance.

7.2 Data transformation times

First of all, it is easy to notice, that sorting takes longer than paginating, which takes longer than filtering. It is because those algorithms have different complexity. The very important fact to consider while looking at table in *Figure 8* is that all HTTP requests were delayed by 200ms. Taking this in fact, research have revealed, that some of these operations could take less time when performed by service worker, than when computed by browser's script. It is because in fact browser has to perform computations to get the results, while service worker just loads data from cache.

In every network speed scenario applications with freshness strategy service workers were performing quite worse than applications without service workers.

Once again this is caused by data passing through service worker.

7.3 Summary

Service workers, when correctly used, can have a great impact on web application performance. They are a great example of using multiple threads to improve applications. Service workers with freshness strategy seem to be increasing HTTP response times, but the fact is, that this difference is really low, and in real-world scenarios, when the network isn't always perfect and sometimes works really slow, or event doesn't work at all, freshness strategy service workers can be a life-saver for web applications. Furthermore, there are no contraindications to provide both of the service workers strategies in one service worker. Cached request routes can be separated in different groups, and each of them can be configured differently. When service workers are used correctly, different types of data can be provided from cache, or directly from API, and in case of loosing the connection user is still going to have access to the application. Service workers are a huge step ahead towards web applications performing just like local ones.

References

[1] https://caniuse.com/

[2] https://developer.mozilla.org/en-US/docs/Web/API/Service_Worker_API

[3] https://developers.google.com/web/ilt/pwa/introduction-to-service-worker

[4] https://angular.io