UNIVERSITY OF MACEDONIA
GRADUATE PROGRAM
DEPARTMENT OF APPLIED INFORMATICS

Progressive Web Apps: Development of cross-platform and cross-device apps using modern web architectures and technologies

MSc Thesis

of

Sapountzi Ibraim

Thessaloniki, October 2020

Progressive Web Apps: Development of cross-platform and cross-device apps using modern web architectures and technologies

Sapountzi Ibraim

B.Sc in Applied Informatics,
University of Macedonia, 2015

Msc Thesis

submitted as a partial fulfillment of the requirements for

THE DEGREE OF MASTER OF SCIENCE IN APPLIED INFORMATICS

Supervisor: Chatzigeorgiou Alexandros

Approved by examining board on 3 November 2020

Abstract

Usage of web and mobile applications nowadays has totally increased from the last ten years. The appearance of social media, has entirely increased the information delivered by the web and its users, while gave the opportunity and fundamentals for non-experienced users to make use of other applications too. Enterprises are now building web applications to manage their internal procedures instead of old school desktop applications. Applications are now a key part in everyday life of humans abroad a variety of handheld devices like laptops, smartphones, smart watches etc. They are also available in desktop, televisions and other devices that exist in a home and make our lives easier. Building native applications for several target platforms and devices is time-consuming and expensive. Progressive Web Applications allow developers to reuse standard web technologies to create applications that may run cross-platform and cross-device, while still narrowing the gap with native applications in capabilities and user experience.

This thesis focuses on what progressive web apps are, the technical aspects behind them, and how they are developed and delivered to final users. These web apps offer features such as installation on the target device, reliability while offline, and engagement with the final users. To achieve this goal and explore all these features, a warehouse management system will be developed, both from the server and the client prospective. Such system's are complex to manage because of their transactional behavior, and the problem that arises is the behavior of the client while offline, which can lead to inconsistent data. Finally, we present the advantages and disadvantages of building such type of applications, compared to other existing cross platform and native solutions for building applications that platform – device depended.

Keywoards: progressive web apps, pwa, cross-platform, web, mobile, warehouse management system, wms

Περίληψη

Η χρήση των διαδικτυακών εφαρμογών και των εφαρμογών κινητών τηλεφώνων έχει σημειώσει μια ραγδαία αύξηση τα τελευταία δέκα χρόνια. Η εμφάνιση των κοινωνικών δικτύων, έχει αυξήσει την χρήση εφαρμογών και την πληροφορία που ανταλλάσσετε μεταξύ των χρηστών, δίνοντας την ευκαιρία ακόμα και σε μη έμπειρους χρήστες να χρησιμοποιούν εφαρμογές. Οι επιχειρήσεις πλέον χρησιμοποιούν διαδικτυακές εφαρμογές – συστήματα για την διαχείριση των εσωτερικών τους λειτουργιών. Οι εφαρμογές και η χρήση τους πλέον καταλαμβάνουν ένα μεγάλο μερίδιο της καθημερινότητας μας μέσω κινητών συσκευών όπως laptops, κινητά τηλέφωνα, ρολόγια κτλ. Επίσης, τις χρησιμοποιούμε στους σταθερούς μας υπολογιστές, στις τηλεοράσεις μας όπως και σε άλλες συσκευές οι οποίες πλέον έχουν την δυνατότητα σύνδεσης στο διαδίκτυο. Η ανάπτυξη εφαρμογών για κάθε ξεχωριστή πλατφόρμα και συσκευή, είναι ακριβή σε πόρους και χρονοβόρα. Οι διαπλατφορμικές 'προοδευτικές' διαδικτυακές εφαρμογές (Progressive web apps), οι οποίες κάνουν χρήση των τεχνολογιών του διαδικτύου και των σύγχρονων προγραμματιστικών διεπάφων των προγραμμάτων περιήγησης, μας δίνουν την δυνατότητα να αναπτύξουμε εφαρμογές οι οποίες δουλεύουν ανεξαρτήτως πλατφόρμας – συσκευής, μόνο με την χρήση των προγραμμάτων περιήγησης, ουσιαστικά μειώνοντας το κενό που υπάρχει μεταξύ των εφαρμογών που έχουν αναπτυχθεί για κάθε ξεχωριστή πλατφόρμα (native), όσον αφορά τις δυνατότητες που παρέχουν καθώς και της εμπειρίας χρήσης τους.

Κατά την ανάπτυξη της εν λόγω διπλωματικής εργασίας, θα αναλύσουμε τι είναι τα Progressive Web Apps, τις κύριες τεχνολογίες που τις διέπουν, και το πώς αναπτύσσονται και τελικά παραδίδονται στους τελικούς χρήστες. Καθώς κάθε διαδικτυακός τόπος – εφαρμογή δεν είναι κατάλληλη για να είναι και εφαρμογή, θα αναπτύξουμε ένα σύστημα διαχείρισης αποθηκών. Τέτοιου είδος συστήματα, επειδή είναι περίπλοκα όσον αφορά την φύση τους καθώς διαχειρίζονται πολλές συναλλαγές, αποτελούν ένα καλό παράδειγμα για να μελετήσουμε και να αξιολογήσουμε την χρήση των progressive web apps, καθώς προκύπτουν προβλήματα στην ενημέρωση του αποθέματος όταν η εφαρμογή πελάτη είναι εκτός σύνδεσης και προσπαθεί να συγχρονίσει τα δεδομένα.

Λέξεις Κλειδιά: διαπλατφορμικές εφαρμογές, προοδευτικές διαδικτυακές εφαρμογές, κινητές συσκευές, σύστημα διαχείρισης αποθήκης

Table of Contents

1 Introduction 1

1.1 Problem - Importance of the subject 1

1.2 Purpose - Objectives 1

1.3 Thesis structure 1

2 Progressive Web Apps 3

2.1 Introduction 3

2.2 Characteristics 4

2.3 Native, Hybrid, Cross compiled, Progressive Web Apps. 8

2.4 Business Success? 9

11

3 Technical components 11

3.1 Secure contexts – HTTPS 11

3.2 Service Workers 12

3.2.1 Lifecycle..........13

3.3 Caching strategies 16

3.3.1 Stale-While-Revalidate..........17

3.3.2 Cache-first..........17

3.3.3 Network-first..........18

3.3.4 Network-only..........19

3.3.5 Cache-only..........20

3.3.6 Cache then network..........20

3.4 Web app manifest 21

3.5 Web storage 23

3.6 Web Push Notifications 25

3.7 Background sync 28

4 Depo - A warehouse management system client 31

4.1 Problem definition 31

4.2 Requirements 31

4.3 Technologies used 33

4.4 Implementation 34

5 Conclusion 40

5.1 Thesis overview and discussion 40

5.2 Future work 43
6 References 44
7 Appendix 49

Table of Figures

Figure 1: Characteristics of progressive web apps [26]....................7
Figure 2: Pinterest after implementing PWA compared to previous mobile version and native apps [25]....................11
Figure 3: Service worker introduction [35]....................13
Figure 4: Service worker lifecycle [32]....................16
Figure 5: Stale-While-Revalidate strategy [35]....................17
Figure 6: Cache first strategy [35]....................18
Figure 7: Network first strategy [35]....................19
Figure 8: Network-only strategy [35]....................19
Figure 9: Cache-only [35]....................20
Figure 10: Cache then network strategy [35]....................21
Figure 11: Web push first step overview [43]....................26
Figure 12: Web push second step overview [43]....................27
Figure 13: Web push step three overview [43]....................28
Figure 14: Menu install button....................34
Figure 15: First visit sign in....................34
Figure 16: Installing in the background....................35
Figure 17: Installation prompt from browser....................35
Figure 18: First boot from device - splash screen....................35
Figure 19: Added to home screen....................35
Figure 20: Stock movements timeline....................36
Figure 21: Stock movements offline....................36
Figure 22: Export items success....................37
Figure 23: Export items form....................37
Figure 24: Export failure out of stock....................38
Figure 25: Export items failure item not assigned to given bin....................38
Figure 26: Offline export....................39
Figure 27: Offline export notify user....................39

Index of Tables

Table 1: Service worker registration..14
Table 2: Service worker installation...14
Table 3: Service worker activation used on update..15
Table 4: Web app manifest file example...22
Table 5: Background sync page context..29
Table 6: Background sync service work context..29
Table 7: Comparison of different cross-platform development approaches.....................41

1 Introduction

1.1 Problem - Importance of the subject

The overall goal of this thesis was to conduct an analysis of the potential problems during the development process of a cross-platform application among the variety of different devices. In particular this research will demonstrate the usage of technologies and modern methodologies in scope of "progressive web applications", and how this type of application development differs from other approaches like "native application development" and "hybrid application development" respectively. Finally, a compare was conducted between the different methods of development from different perspectives in term of cost of development, tooling, security, application delivery, offline experience, background synchronization, user engagement and user experience.

1.2 Purpose - Objectives

The purpose of this thesis, is the analysis, design, and development of a cross-platform and cross-device application using latest web techniques and technologies. In scope of this thesis a warehouse stock management mobile application was developed as a demonstration of the utilization of the technologies researched. The type of application was chosen as an example because of the technical difficulties that arise in terms of management, as of its transactional behavior. Furthermore, we are going to explore different architectures and techniques that are supported from modern browsers, in order to provide features such as offline mode and background data synchronization. As of the complex nature of a warehouse management system and particularly of a stock management sub-system, it will be interesting to see how to overcome and simply issues when an offline experience is presented, while the interaction with these type of systems can ensure data integrity and consistency.

1.3 Thesis structure

This thesis begins with an introduction about what is a progressive web application and what are the unique characteristics it benefits from compared to traditional web, native or other hybrid approaches of application development. Finally the particular benefits will be measured after the implementation or migration to a progressive web applications. In Chapter 3, the components and technology requirements are defined in order to progressively enhance a web application. Additionally a detailed illustration of the core components needed for a PWA (Progressive web application), such as service workers, HTTPS, and web manifest file is presented. Furthermore, new and advanced Web APIs are introduced and explained in detail, such as Web Storage APIs, Web Push Notification and Background Sync API that are essential for an offline first experience. Finally, in Chapter 4, we explore the implementation of the use case of a Warehouse Management System client built as a progressive web application. This included the whole development cycle of requirement analysis, used technologies, features implemented in this application.

2 Progressive Web Apps

2.1 Introduction

Progressive web application is a term firstly introduced by a Google software engineer, Alex Russel along a designer named Frances Berriman, in 2015 [1]. It describe how a web application should behave, in order to be progressive. So, from Russel's perspective, a web application in order to be progressive and not hybrid, should be *responsive*, *connectivity independent*, *app-like interactions*, *fresh*, *safe*, *discover-able*, *re-engage able*, *install able* and *link able*. As we can see, Russel described different aspects of how a web application should behave, how is able to support all features explained, which provides an experience similar to a normal native application. From then till today, 2020, there is no standard term – definition formed for describing what progressive web applications actually are, but as stated by Russel, and maintained till now from Google's prospective[2], *there are just websites that took all the right vitamins*. So, as we can see, it's in the vendor's browser ability to serve web applications that are progressive, thus the term progressive, which means that if the technical components and latest modern Web APIs required to serve a progressive web app are missing and not implemented, the application should behave as a normal web application, without having all these features or having some of them.

Talking about technologies and modern Web APIs, always adopted and implemented by browsers, under the specifications and standards of *W3C*, there is a cold war of browser vendors regarding progressive web apps. As stated, the term Progressive web apps or PWAs, was firstly introduced by an engineer from Google, and since then, this umbrella term is highly promoted by Google. On the other hand while most browser vendors embraced the "Progressive Web App" term, there is one that is still fighting against it: *Apple*. It seems that they don't adapt the term Progressive web apps, but they refer to these kind of apps as 'HTML5 Apps' and/or 'Home screen web apps' instead [3][4]. Apple with Safari browser, running under WebKit engine, seems that till now, has partially implemented APIs that support progressive web apps and lacks a lot of the features needed [5]. But at first glance, the idea of a PWAs wasn't invented either by Google or by Apple, but it was Steve Jobs who first presented the concept in front of the world, during the iPhone introduction in 2007. At that time, it seemed natural that external apps would help to increase the popularity of that device, and Steve Jobs wanted developers to build apps using standard open web technologies [6]. But after some months, Steve announced the iPhone SDK for building mobile apps for iOS, and after a while Apple presented the App store. From then till now, as mobile internet are conquering the world [7], this is the de facto way to build and publish apps for mobiles, and is dominated by Google with Play Store and by Apple with App store. As a result, the global app revenue in 2018 grew 23% to more than $71 billion on iOS and Google Play[8].

Therefore, here comes the mobile web, the result of mobile internet empire. A lot of improvements have been made to websites and web applications in order to fit the mobile internet, like responsiveness, which is handled by CSS media queries. It's a standard that in 2020, every website is responsive, so it fits in a broad variety of devices, from desktop to laptops, tables and mobile phones. But what about the other features – characteristics that power's up an application experience; That's all about progressive web apps, a design pattern to develop and deliver web applications using modern web APIs and architectures, being platform and device independent, giving an app-like experience, with the reach – link ability that the web offers, and the capabilities of native apps.

2.2 Characteristics

In the previous section, we defined what progressive web applications are or rephrase what a web application is intended to be by terms of progressive. It's time to explore which are the characteristics – features that distinguish a PWA from a traditional web application. As explained earlier, a progressive web app is a web application that borrows concepts – techniques from traditional native mobile apps alongside the power and the capabilities of the web. In the time writing, the official definition from Google about what are the characteristics of progressive web apps, refers to the *three app pillars* [9]. That is, they must be *capable*, *reliable*, and finally *install-able*. There are some general defined categories about how applications should be, meaning that they can interact with the device through various API's (*capable*) such as push notifications and geolocation, they are able to boot fast, feel fast, and work offline without internet connection (*reliable*), and of course you can install them in your device like any other application, and not only navigate to it through a browser. These three pillars transform them into an experience that feels like a platform-specific application, such as mobile native applications. Below there is a detailed overview of those characteristics, that these three categories contain. And because a picture is worth a thousand words, in figure 1 we see all these characteristics aggregated together [26].

- **Progressive:** Work for every user, regardless of browser choice because they're built with progressive enhancement as a core tenet. This means that the web application can scale up or down according to where it is installed (i.e device or browser). Features missing by a browser should not break the application, but fallback to another technology or just launch without that feature. Feature detection is generally used to determine whether browsers can handle more modern functionality, while polyffils are often used to add missing features with JavaScript [10].
- **Responsive**: Application should be able to adapt to any screen size that has been launched. This is achieved, as mentioned, through CSS media queries and viewports, and there is a lot of progress to web sites and web applications on this field already. Businesses and developers try to support as many devices their domain could be used for. Fit any form factor: desktop, mobile, tablet, or forms yet to emerge. Mobile first approach is used for progressive web app development, that means that we start designing the application by first fitting the mobile device, in contrast of responsive web development where we first designed for desktop and then scaled down to smaller view ports
- **Network independent:** Offline first, connectivity agnostic, can work offline or on lie-fie (low quality internet) mode. Application can launch offline with the basic screen skeleton known as the app shell model, a concept borrowed from native applications, able to see latest data retrieved from the internet, and when connection is back again, can sync its data to the server. This is achieved by **service workers**, the core technical component of progressive web apps, various cache and data storage APIs, in order to store your static assets and/or your data, background sync API to sync your content to the server [11]. Details for these technical concepts are in the next chapter.
- **Install-able:** Users can install the application from browser or recently from app stores (Play store at the moment), just as any other native application. You can

pin and launch from the home screen, your taskbar, etc. This is achieved by the web app manifest technology [12], and gives a lot of benefits for the distribution of the application, no more specific store troubleshooting.

- **Safe**: Served via HTTPS to prevent snooping and ensure content hasn't been tampered with. HTTPS is required for the core technical components to work, like service workers. If TLS certificate is not available, app cannot respond to the most of the features – characteristics described in this section.
- **Discover-able**: One of the most important features that the web offers, the ability to search and find in search engines. Progressive web apps are identifiable as "applications" thanks to W3C web app manifests and service worker registration scope allowing search engines to find them just like any other website.
- **Link-able:** Alongside the last characteristic described above, another great core feature that the web offers. The ability to send a URL we have just discovered, to someone, without the need of app stores and/or complex installations. Alongside this powerful core feature of web, there are already some new web API's like, web share API and contact API, which are integrated with the underlying OS, and allow you to share your web app to other apps or pick some from your contacts [13] [14].
- **Engage-able:** One of the most major features of native apps, the ability to re-engage your users by sending them notifications even if the application is not running. This is achieved on web by web push and notifications APIs [15] [16].
- **Fresh:** Application always updates to the latest version through the use of service workers. When a new version of your app is available, the app will update-refresh automatically or you can notify your user that a new version is available, and by refreshing the site, the new version will be there. No need of app store reviews and complex deployments – distributions.
- **App-like:** User Interface and User Experience is designed to be more application like with app-style interactions and animations. You can achieve great things with the use of CSS3 features and great CSS frameworks and design systems, e.g material design from Google.

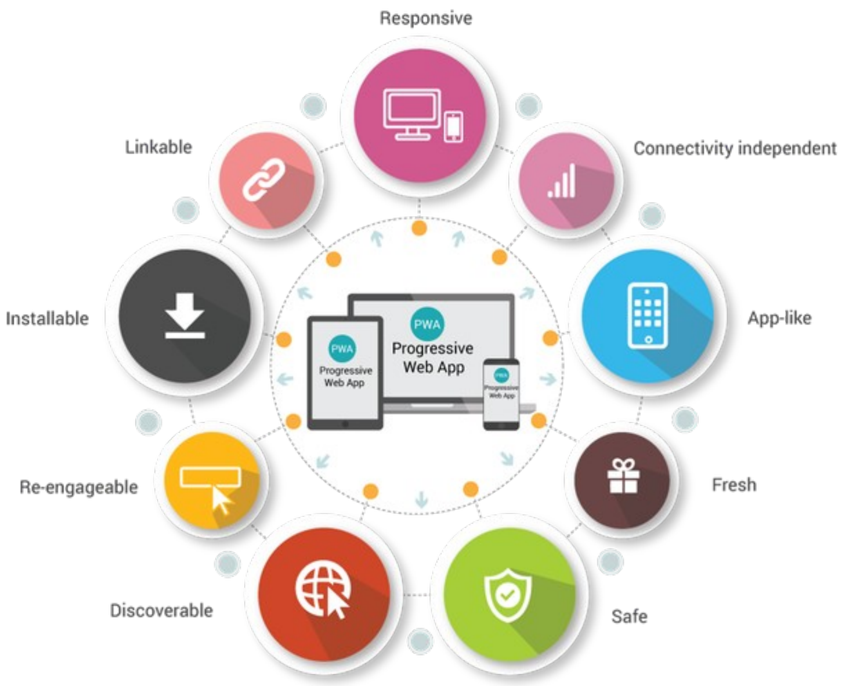


Figure 1: Characteristics of progressive web apps [26]

All of these characteristics describe how a web application can become a progressive web application and fit well in the mobile web. We can see a mix approach with techniques borrowed from native applications and features from web applications. The main question is, should someone implement all of these, in order to be progressive and fit well in the mobile web; the answer is no, it depends in the case and the features you want to provide to your end users. For the time writing, Google offers a PWA checklist, with some core and optimal features in order to get identified as a progressive web app [17]. Furthermore, when Google first introduced Progressive Web Apps, they build a tool called Lighthouse, which scans your website and calculates some metrics in terms of performance, accessibility, best practices etc. One of the reports generated, is how much progressive web app is your site, and how you can make it more progressive [18]. It is offered by default on google chrome dev tools or as a standalone npm library, and it gives greats insights about the performance of your app.

2.3 Native, Hybrid, Cross compiled, Progressive Web Apps.

Where does progressive web apps fit between different app development approaches that already exist; Can we really compare progressive web apps to native apps; From my point of view, there is no such comparison. Native mobile apps that are created for iOS and Android operating systems are very powerful and have the best integration with the device, but as it is known they are expensive in means of cost of development and resources. If you want to target both platforms, you must have two different teams to achieve that, each team building for the specific platform with their available languages and tools. Many businesses do not have the ability to develop and distribute different apps for each platform; so they go with the **hybrid** approach and other mobile cross platform solutions. By hybrid, we mean a single code base, that is developed with web technologies, thus HTML5, CSS3, JavaScript, and with the help of an underlying layer that gives integration's with the OS, such as Apache Cordova and PhoneGap, and finally wrapped in a native container using WebView(an in-app integrated browser component). Another popular choice that has been born the last years, because of the rise of JavaScript and its frameworks, are the **JavaScript - native** mobile apps, that are developed with frameworks like React Native and NativeScript. This type of apps, are built again with the use of JavaScript, but they don't render through a *WebView*, but by using the native user interface libraries on Android and iOS, finally transpiled to native components. However, all the JavaScript code will be executed in a JavaScript virtual machine on-device, so there is no JavaScript to real native code conversion when compiling the app [19]. Another existing solution worth to mention, is cross-complied mobile apps, that are developed through the Xamarin framework, which uses C# and .NET framework, and libraries for each platform, thus when compiled they are transpiled to the platform specific code. Moreover, new cross-platform solutions exist nowadays such as Googles framework Flutter [19] and more.

A lot of mobile apps have been developed the last years by the approaches described, instead of native, but there are many issues in terms of sub-optimal user experience, loss in performance, device fragmentation, hard to integrate with device APIs security issues [20]. So, we can see that a lot of progress has been made in order to have a cross-platform mobile application through a single code base. All these technologies are issuing the same problem with different approaches. All of them are addressing the same problem; the cross-platform mobile application distribution. Can we say that progressive web apps are comparing with all these technologies; the answer is no. All these technologies, or the most of them, are device dependent. All of them finally distribute mobile application through the corresponding application stores. On the other hand, progressive web apps are not intended to be only for mobile. They can run and install on desktop, laptop, tablets, mobiles, and on any device that has a web browser. So, they are device agnostic. The only reason for this comparison, is again, the rise and domination of mobile web. Maybe in the years ahead, one other device will show up, that has the same usability of mobile devices. Progressive web apps, will run anywhere there is a browser engine, with the technologies of the web. And their purpose for now, is to narrow the gap between native devices, with better integration with the operating system and the device they run. Native mobile apps are great in performance, and most suitable for heavy computation apps like games. But, recently the web has introduced WebAssembly (WASM), a binary instruction format for a stack-based virtual machine. WASM is designed as a portable compilation target for programming languages, enabling deployment on the web for client and server applications, and promises to be efficient and fast, due to its binary compiled nature [21]. You can view it as a JVM for the web. After all, maybe with the power that WebAssembly brings and the patterns of progressive web apps, we can see great performant applications that are device independent, and can be distributed by the browser and not only by specific app stores.

2.4 Business Success?

In the last five years, the progressive web apps architecture pattern has been applied by many giant companies such as Twitter, Tinder, Pinterest, Alibaba and more [22]. Most of them, offered a progressive web app as alternative to their mobile web site, not of course as a replacement to their native applications, and saw a huge growth from user's that previously abandoned their mobile site in not more than three seconds, because of the slow load of their mobile versions. In fact, as page load times go from one second to ten seconds, the probability of a user bouncing increases by *123%* [23]. Especially, Twitter was over 80% of users on mobile, they needed to provide more engaging access with lower data consumption – especially for visitors who had a weak internet connection. With the launch of Twitter Lite, a PWA, they increased their user engagement and reduced data usage. Especially, they increased by *65%* pages per session, *75%* increase in Tweets sent, and finally *20%* decrease in bounce rate [24]. The app implements the "Add to Homescreen" prompt, push notifications that work the same as those from native apps, and a special data saver mode, which can greatly reduce the amount of mobile data used. The new PWA is only 600KB over the wire vs. 23.5MB of downloaded data needed to install the native Android app. Finally, Nicolas Gallagher, the Engineering Lead for Twitter Lite, notes: "Twitter Lite is now the fastest, least expensive, and most reliable way to use Twitter. The web app rivals the performance of our native apps but requires less than 3% of the device storage space compared to Twitter for Android. "

Pinterest had its own success story too, by turning its mobile version of the website to a progressive web app. Only 1% of their visitors converted to sign-ups and app installations for iOS and Android. After PWA implementation, visitors spent *40%* more time on Pinterest's PWA compared to the mobile website. Pinterest experienced a *44%* increase in ad revenue rate and a *60%* increase in user engagement. The Pinterest PWA requires only 150 KB of data storage, which is much less than the native Android (9.6MB) and iOS (56MB) apps [25].

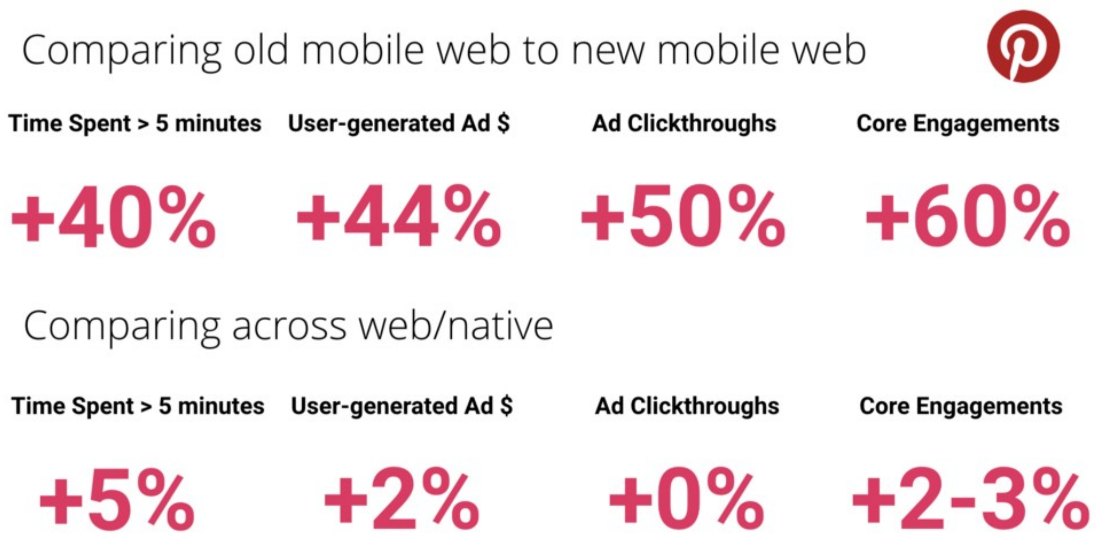


Figure 2: Pinterest after implementing PWA compared to previous mobile version and native apps [25]

3 Technical components

3.1 Secure contexts – HTTPS

One of the most critical core technical components that are required by progressive web apps and other additional core components and modern APIs, such as service workers, is the encryption of data served in the page via TLS under ***HTTPS***. A lot of progress is made in this field nowadays, and it's a de facto way that every site is served over HTTPS. Some years ago browsers started identifying web sites that are not served with HTTPS, and warned the users instead of directing them to the web site – application. Adding a security layer to your site (in the form of SSL certificate) is not only a best practice, but it also helps earn your users' trust and create a positive image of your application in their minds.

A **secure context** is a `Window` or `Worker` where certain minimum standards of authentication and confidentiality are met. Many Web APIs and features are accessible only in a secure context, and even older APIs, such as Geo location API, are updated to require secure contexts, pages that are served through HTTPS [27]. The primary goal of secure contexts is to prevent *MITM (Man-In-The-Middle)* attackers from accessing powerful APIs that could further compromise the victim of an attack [28]. These new modern APIs that allow low-level integration with the device that are used, access to the corresponding file system, web workers – especially service workers that are enabled and are activated even if the web page is closed, can offer a great opportunity to attackers, when not served with HTTPS.

3.2 Service Workers

Service workers are the **most** critical technical components that enable the progressive enchantment of a web application. It is an instance of a web worker, that runs in the background, independent to the main thread of the browser -that one that has access to the *DOM* and renders the page-, thus it is non-blocking and full asynchronous. It acts like a proxy between the network and the client, the application running on the browser, intercepts HTTP requests and serves the responses from the network or from a local cache, according to which caching strategy we have implemented. With a service worker, a lot of progressive web app features are available, like caching the static minimum required asset resources, also refereed as the *app shell [29]*, in order to *boot* fast and *run* fast, working offline by serving cached static content and dynamic content that is persisted in various data stores now available to browsers, enable web push notifications, background syncing and more. They are intended, among other things, to enable the creation of effective offline experiences, intercept network requests and take appropriate action based on whether the network is available, and update assets residing on the server [30]. They can be registered against an origin and a path, and a web application can have more than one service worker registered, each in its scope and context. As scope, we mean the origin that it is registered. So, if a service worker is registered on "*/example/"* page, it can control requests from pages like "/example/", "/example/foo/", "/example/foo/bar", but not from pages that fall under "/example", or "/", which are higher [31].

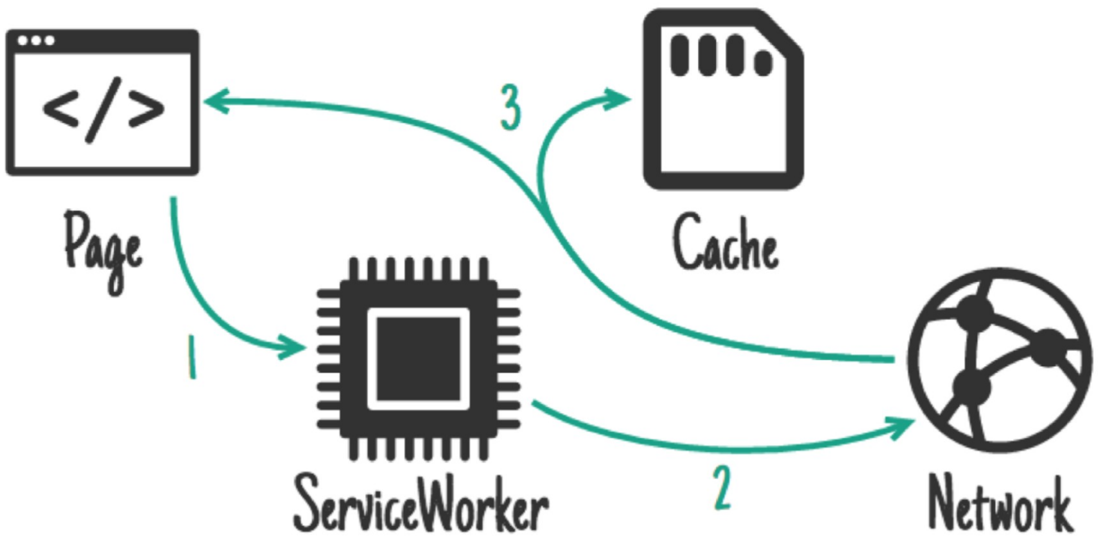


Figure 3: Service worker introduction [35]

So, as we can see in Figure 3 above, a service worker sits between your application and the network, listening to events and network requests, and gives the ability to serve content, either static or dynamic, from the Cache Storage or other persistence storage mechanisms available now in the browser. Being an instance of a Worker, it has no direct communication with the main page thread, but can communicate with other workers and other pages in their scope through *window.postMessage()* or/and messaging channel APIs, and then those pages can manipulate the DOM, allowing an indirect access. By this indirect access, we have the ability to perform heavy calculations tasks on a separate thread, a worker, where the blocking main thread can have huge performance issues and it is used mainly to render the web page.

3.2.1 Lifecycle

Service workers have and maintain their own complex lifecycle on the web page that they are registered to, which is completely separate from the web page's lifecycle. You start by registering the service worker in the page you intent to use it, and then if *registration* is successful, e.g browser supports service workers, it goes through *downloading*, *installing* and *activation* phase. You typically register a service worker, with a code snippet that is shown below [31].

Table 1: Service worker registration

```
if ('serviceWorker' in navigator) {
  window.addEventListener('load', function() {
    navigator.serviceWorker.register('/sw.js').then(function(registration) {
      // Registration was successful
      console.log('ServiceWorker registration successful with scope: ', registration.scope);
    }, function(err) {
      // registration failed
      console.log('ServiceWorker registration failed: ', err);
    });
  });
}
```

First, we start by checking if service worker is available on the current browser it tries to register, thus provide progressive enchantment through feature detection. If service worker is available, it tries to register it and then the other phases of the lifecycle begin a download to the client, and then attempt to install and finally activate it. A typical installation process is defined in the code snippet contained in the table below.

Table 2: Service worker installation

```
var CACHE_NAME = 'my-site-cache-v1';
var urlsToCache = [
  '/',
  '/styles/main.css',
  '/script/main.js'
];

self.addEventListener('install', function(event) {
  // Perform install steps
  event.waitUntil(
    caches.open(CACHE_NAME)
      then(function(cache) {
        console.log('Opened cache');
        return cache.addAll(urlsToCache);
      })
  );
});
```

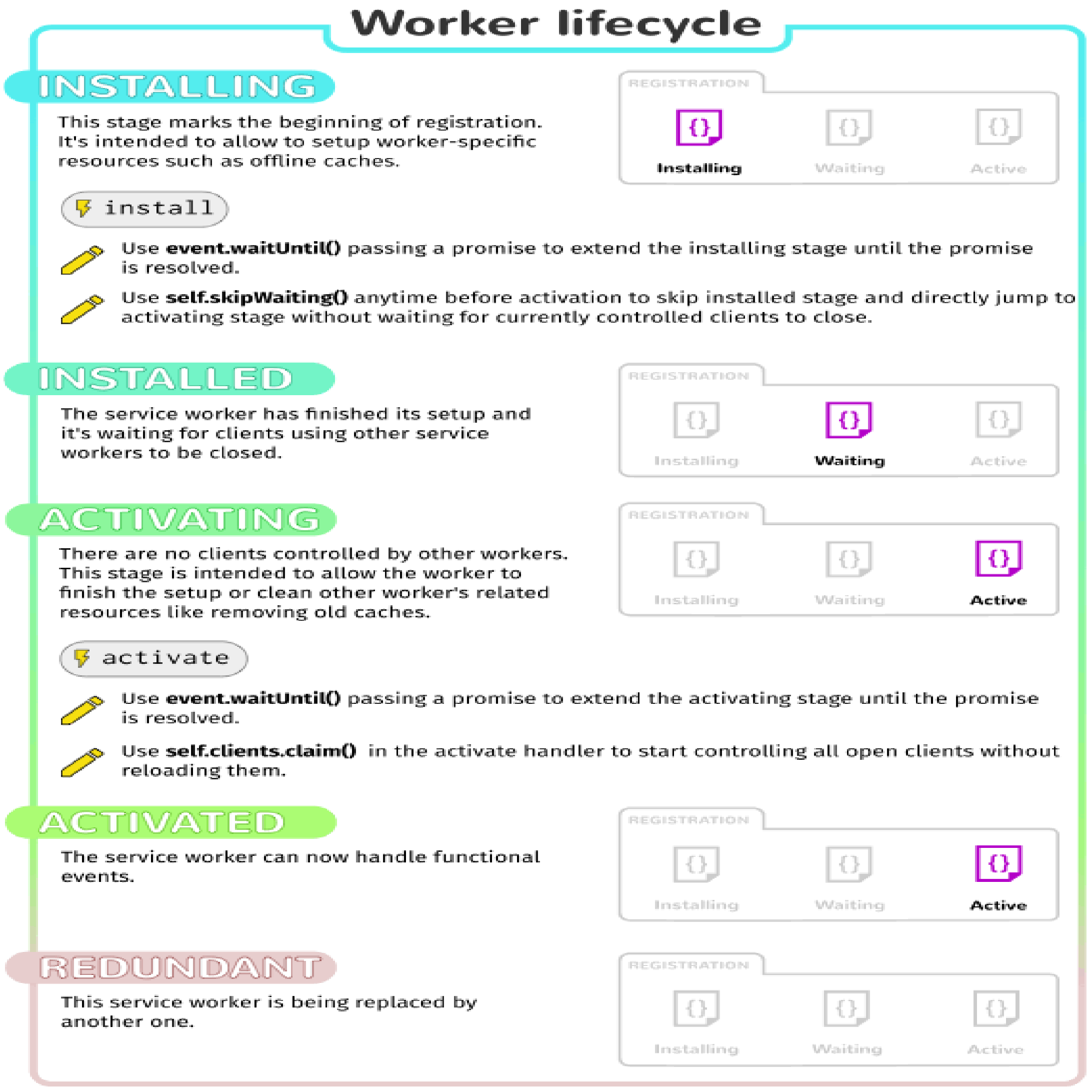
As we can see from the code snippet provided above, the de facto process that occurs in the installation event, is to cache the static assets the app needs to bootstrap. We usually cache CSS files, images, fonts, JavaScript files, all the minimum required parts in order to have a basic app bootstrapped, that can be available even if network is not available. These patterns are also described as the app shell model [34]. Installation is attempted when the downloaded file is found to be new — either different to an existing service worker (byte-wise compared), or the first service worker encountered for this page/site. If this is the first time a service worker has been made available, installation is attempted, then after a successful installation, it is activated. If there is an existing service worker available, the new version is installed in the background, but not yet activated — at this point it is called the *worker in waiting*. It is only activated when there are no longer any pages loaded that are still using the old service worker. As soon as there are no more pages to be loaded, the new service worker is activated (becoming the *active worker*) [30]. Activation event is usually used to perform cache management, after updating a service worker or files to be cached, and it typically clears redundant cached files, and re-updates cache with the desired files. A typical activation phase is described below, after trying to update our service worker.

Table 3: Service worker activation used on update

```
const expectedCaches = ['static-v2'];

self.addEventListener('activate', event => {
  // delete any caches that aren't in expectedCaches
  // which will get rid of static-v1
  event.waitUntil(
    caches.keys().then(keys => Promise.all(
      keys.map(key => {
        if (!expectedCaches.includes(key)) {
          return caches.delete(key);
        }
      })
    )).then(() => {
      console.log('V2 now ready to handle fetches!');
    })
  );
});
```

Finally, there is a last phase on the service worker lifecycle, which is called *redundant*. When a service worker reaches this phase, it means that it has been replaced by another one, so it will leave the scope. A detailed overview of the complex service worker lifecycle is shown in the below figure.



INSTALLING

This stage marks the beginning of registration. It's intended to allow to setup worker-specific resources such as offline caches.

install

Use **event.waitUntil()** passing a promise to extend the installing stage until the promise is resolved.

Use **self.skipWaiting()** anytime before activation to skip installed stage and directly jump to activating stage without waiting for currently controlled clients to close.

INSTALLED

The service worker has finished its setup and it's waiting for clients using other service workers to be closed.

ACTIVATING

There are no clients controlled by other workers. This stage is intended to allow the worker to finish the setup or clean other worker's related resources like removing old caches.

activate

Use **event.waitUntil()** passing a promise to extend the activating stage until the promise is resolved.

Use **self.clients.claim()** in the activate handler to start controlling all open clients without reloading them.

ACTIVATED

The service worker can now handle functional events.

REDUNDANT

This service worker is being replaced by another one.

Figure 4: Service worker lifecycle [32]

3.3 Caching strategies

After examining the service worker lifecycle, we could definitely tell that service workers rely a lot on the Cache Storage API [33]. They use cache to store static assets like required files to bootstrap the app, and then serve them from the cache instead of traversing them sfrom the network. We can also store and serve user data, application data, and whatever content the application produces. A caching strategy is a pattern that determines how a service worker generates a response after receiving a fetch event.

Should it generate the response from the cache or other data persistence options; should it generate from the network; or it can use both, combined together; We will explore some common caching strategies, but every application should decide regarding of its domain context and requirements which strategy to use, or maybe combine many of them. Caching strategies is not a new concept on software engineer field, and whenever cache solutions are implemented, they are always problems and considerations to take in mind of how you will use your cached data.

3.3.1 Stale-While-Revalidate

This pattern allows you to respond to the request as quickly as possible with a cached response if available, falling back to the network request if it's not cached. The network request is then used to update the cache. This pattern is ideal for frequently updating resources where having the very latest version is not vital to the application, e.g avatars.

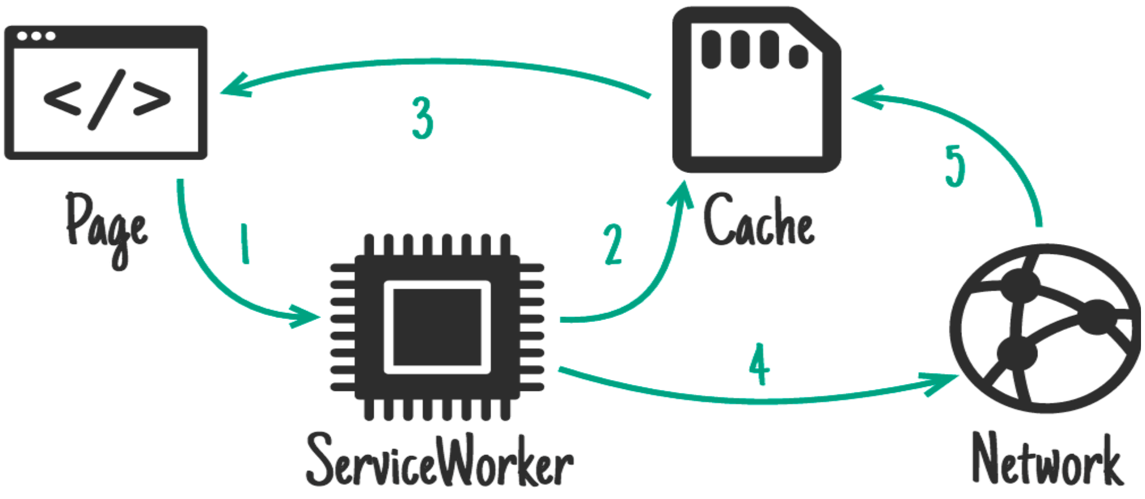


Figure 5: Stale-While-Revalidate strategy [35]

3.3.2 Cache-first

This patterns servers everything from the cache, and if response is not found, fallbacks to network request. So, if the response is already cached, the network will not use it at all. If the request in not already in the cache, the network will be used to serve the request, and the response will be cached in order to be used in the next request.

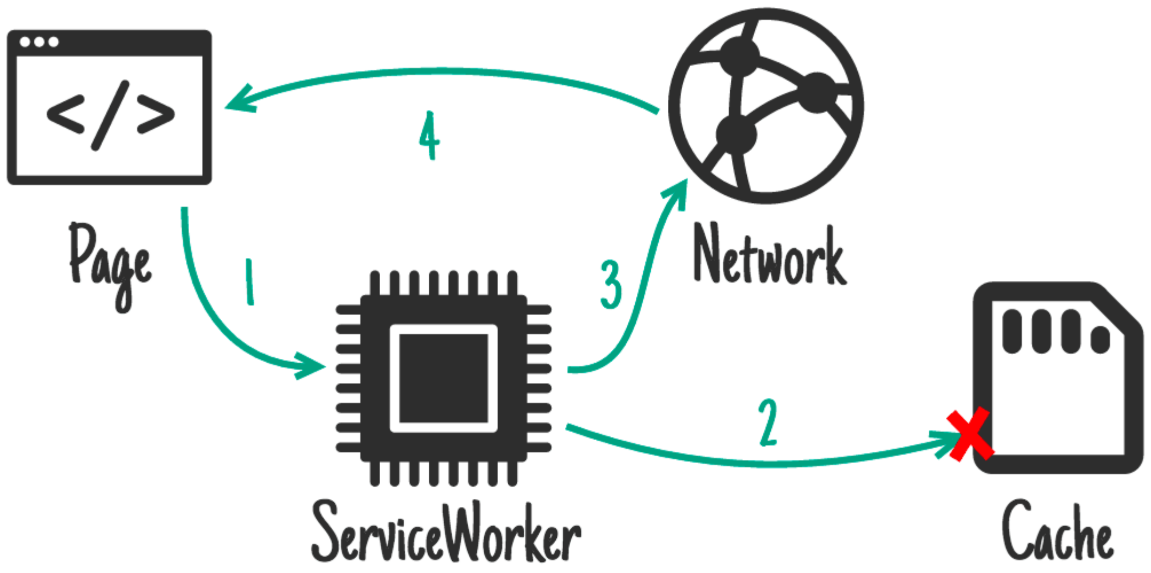


Figure 6: Cache first strategy [35]

3.3.3 Network-first

In this strategy, the network is used to serve the request, then adds it to the cache. When the network is not available it will fallback to the cache, hopefully to serve the latest response requested. This is ideal for requests that are updating frequently, such as content that updates frequently, articles, social medial posts, stock prices etc. With this strategy you give your users the latest up-to date data, but offline users get an older outdated cached version. As this caching mechanism stands great for offline users, for users that are in a lie-fie mode, thus they have a very slow connection, this is not ideal because finally the request will transfer to the wire, having your user's waiting for the new content to arrive. This approach can lead to frustrating user experiences.

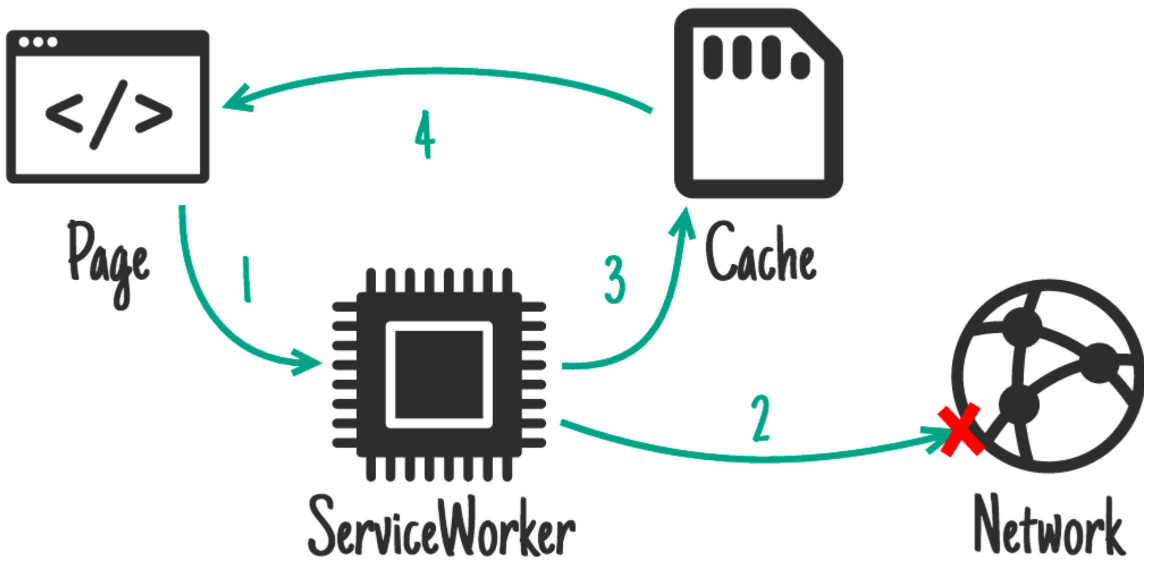


Figure 7: Network first strategy [35]

3.3.4 Network-only

This strategy does not involve cache at all. All the requests are server from the network, like every web application. By using this strategy, you cannot offer an offline first experience because all your data are served from the network and you don't have a fallback to retrieve them from and finally show them.

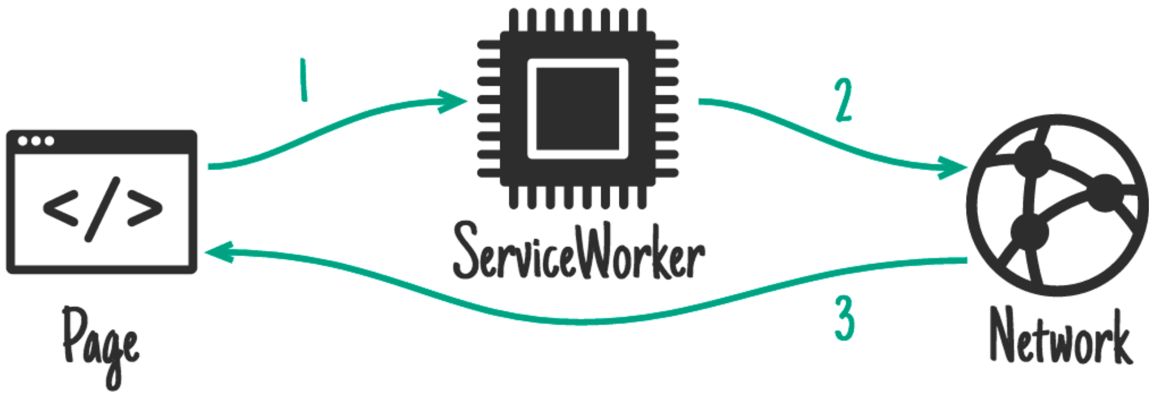


Figure 8: Network-only strategy [35]

3.3.5 Cache-only

With this strategy we serve all the requests only from the cache. It is used only for static content that will never be changed. This pattern is rarely used and cache-first strategy with fallback to network is a better solution.

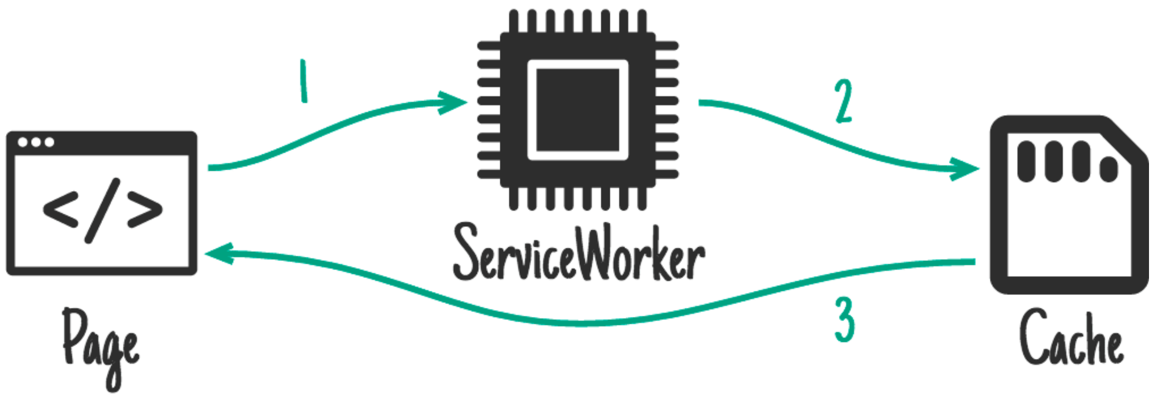


Figure 9: Cache-only [35]

3.3.6 Cache then network

This type of strategy requires two requests. One to the cache and one to the network. In this case, we show the cached content to the user first, and in a second phase we update that data with the response that arrived from the network. Although this pattern seems to work well, just give the user the data you already have and later on update the data that the user sees with the latest arrived, it can also lead to bad user experience, while the user is interacting with the first data and in a second time that data is rendered again, can be confusing for the final user.

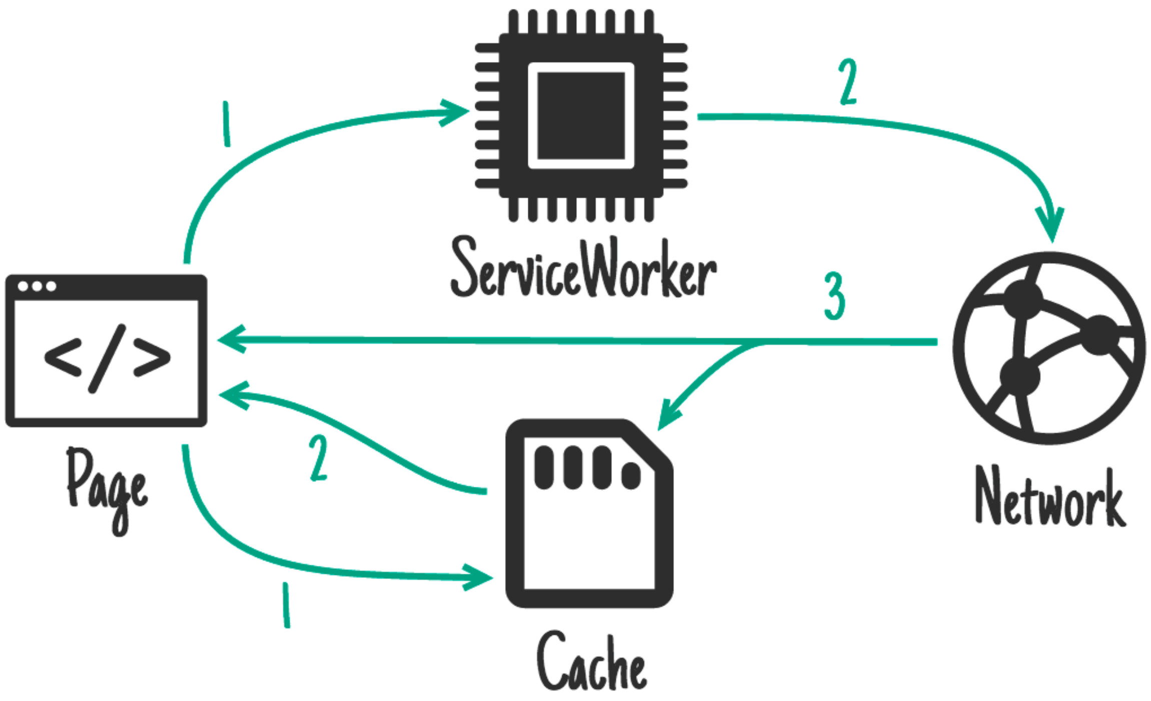


Figure 10: Cache then network strategy [35]

We can see that service workers and the offline first experience have emerged a lot of different strategies and patterns to finally serve data to the user in order to have an offline experience. Above we introduced some of the patterns that already exist in the community, but there are a lot more [35], and its according to the business requirements and the context of the application you are working with, which strategy you will use or a mixed approach of combined strategies. Caching and serving data from a cache is a problem that exists in software development and it is not tight with progressive web apps. Caching can get tricky, so the best approach will be applied according to the problem and the use-case you try to solve and implement.

3.4 Web app manifest

Web app manifest is another core fundamental technology in the progressive web app world. Alongside with HTTPS, an installed and activated service worker, which can do nothing but only exist in the website, web app manifest file offers the install-able characteristic, where finally the user is able to install the app in the device. The manifest is a *JSON* formatted file that defines metadata for the Progressive web app used by a browser or an app store in order to define the installation behavior. When you have the manifest file linked in your HTML document, users will be able to install the app using different techniques depending on the browser, typically called Add to Home Screen, Install or just Add. If your PWA is crawl-able by Bing, Microsoft will automatically add it to the Microsoft Store so Windows 10 users will be able to install it from there, so manifest files help in the discover-ability of the application. Progressive web app manifests include several values such as the application's name, author, icon(s), version, description, a list of all the necessary resources such as app icons, and other things besides. A typical manifest file is shown below [36].

Table 4: Web app manifest file example

```
{
  "short_name": "Depo",
  "name": "Depo Warehouse Management System",
  "icons": [
    {
      "src": "favicon.ico",
      "sizes": "16x16",
      "type": "image/x-icon"
    },
    {
      "src": "depo-192x192.png",
      "type": "image/png",
      "sizes": "192x192"
    },
    {
      "src": "depo-512x512.png",
      "type": "image/png",
      "sizes": "512x512"
    }
  ],
  "start_url": ".",
  "display": "standalone",
  "theme_color": "#000000",
  "background_color": "#ffffff"
}
```

As we can see in the JSON file above, we define some general attributes like app name, description, display – if we want to launch as a standalone experience, full screen or with a minimal UI -, and a list of icons, by which the corresponding OS, will decide which icon is more appropriate for its usage. By including a web app manifest file, browsers will automatically trigger an app installation prompt, asking from the user to add the app to the screen. Otherwise, you can grammatically control this event and give your own installation experience to the users.

3.5 Web storage

The rise of progressive web apps and the offline experience they offer, have emerged in new storage mechanisms and new web APIs in order to address the need of a web moving from simple documents to applications. Nowadays, developers have more options to store and cache data in the client. In the previous section, we took a look in the power of the Cache Storage API and its ability to manage and store data – documents in this new cache mechanism.

Existing solutions such as Cookies, LocalStorage and SessionStorage have their limitations and can cause significant performance issues. They are all blocking the main thread, and their storage capacity is limited to about 5MB [37] and can only store string values. Cookies on the other hand, that exist for a long time, are not appropriate and should never be used for storage, because cookies are transferred on every HTTP request and can result in large requests. Even more, all these storage mechanisms are available only on their scope, and cannot be accessed from other tabs and service workers, where the latest are core requirement in order to have a progressive web app.

Beside the new Cache Storage API, which is mainly used to store documents – files, the whole static content and network resources that needs the app to initial load and render, IndexedDB can be used to store structured data, files/blobs, and persist application state. The API uses indexes to enable high performance searches of the data stored. IndexedDB is a transactional database system, like the well-known and successful old school relational database systems. However, unlike relation databases, which use fixed-column tables, IndexedDB is a JavaScript-based object-oriented database. IndexedDB lets you store and retrieve objects that are indexed with a key; any objects supported by the structured clone algorithm [38] can be stored, that means primitive types, strings, dates, blobs, files, images and more. You need to specify the database schema, open a connection to your database, and then retrieve and update data within a series of transactions. It has a lot of common behavior as relational databases, like the context of transaction, indexes, tables, cursors etc. The concept of transactions is identical, in order to not corrupt your data while modifying them through different windows/tabs [39]. Although it has some common characteristics with relational database systems, like the transactional concept, its data model is closer to NoSQL databases, which are usually key – value storage systems. Although, IndexedDB can cover most cases of client storage needs, it has its limitations too. It does not support internationalization sorting, such as linguistic sorting of relational databases, neither full text searching such as the %LIKE% operator of SQL. Synchronizing concepts are not offered out of the box, the ability to synchronize your client-side storage data with server-side database, although it can be done programmatically. Finally, writing or reading from the database may fail, because of disk limit that has been reached, or the user just cleared all the data stored through browser settings, which can lead finally to data being evicted.

Beside of the Cache Storage and IndexedDB, new APIs have been generated for file storage but are in early stage, thus not supported / implemented from every browser for the time writing. File System API[40] and File System Access API[41] are the new upcoming APIs, where the former provides methods for reading and writing files, such as images, videos, to a sandboxed file system, and the later can save files direct on the local file system, by asking for user permissions, but permissions are not persisted across sessions yet. For now, Cache and IndexedDB APIs are available and supported in every modern browser.

After examining the storage solutions provided from the browsers in order to have and offer a full offline experience, the second question that arises is how much of storage space is available for the application and the origin that is served. The policy determining the maximum amount of data that can be stored in the browser vary per browser vendor. Some browsers may set a limit as a percentage of available disk space (over 50% to 80% for Chrome and Firefox), or a hard limit (around 1GB for Safari) that can be extended through explicit user prompting. For that reason, a new API is upcoming, the StorageManager API [42], which gives the ability to developers to estimate and determine how much storage is available. Beside this functionality, it gives the ability to ask the users in order to persist data via a permission notification, where if user gives permission to persist his data, data eviction that is automatically happening from the browsers is not possible. Besides that, it is up to the application domain and its data to determine an eviction policy of the data. Programmers may delete old data, not critical data in order to free up space when the storage is low, using the API to determine if storage is too low at the given context.

3.6 Web Push Notifications

Mobile applications are great in engaging their users and keep them up to date by using push notifications, which come out of the box. In a world that is dominated with mobile devices, we receive notifications every hour and every day from our installed apps, - email clients, social media apps, chatting apps, e-commerce etc.- Notifications arrive even if we don’t use the app for a while, in order to make us re-engage with the app. Progressive web apps can offer this feature to the web, by using two web technologies, the Push API and Notification API, alongside an installed service worker which is listening to push events in the background, even if the browser tab for the specific app is closed or the browser is not opened at all.

The W3C Push API [15] and Notification API [16] go hand-in-hand to enable push notifications in modern browsers. The Push API is used to set up a push subscription and is invoked when a message is pushed from a server or a push service to the corresponding service worker. The service worker then is responsible for showing a notification to the user using the Notification API and reacting to user interaction with the notification. The process of sending and receiving push notifications can be described in a high level overview by three key steps.

The first step is implemented on the client side, where you subscribe a user to push messaging. Firstly, you kindly ask the user if he/she wants to receive notifications. When you get the permission from the user to do so, you subscribe the user by getting a *PushSubscription* from the browser through the Push API. A PushSubscription contains all the information we need to send a push message to that user. Each PushSubscription works like an identifier to that user's device, thus its unique for every device. In order to subscribe a user and to get a PushSubscription, you have to generate a set of application server keys, also known as VAPID keys, which are unique to your server-side backend implementation. They allow a push service to know which application server subscribed a user and ensure that it's the same server triggering the push messages to that user. Finally, from the backend prospective, you save this push subscription in a storage, in order to retrieve it later when you want to send a push message to that user – device. The first step can be shown in the following figure.

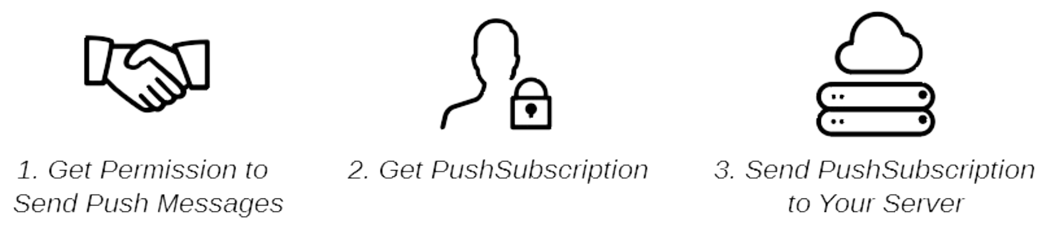

1. Get Permission to Send Push Messages
2. Get PushSubscription
3. Send PushSubscription to Your Server

Figure 11: Web push first step overview [43]

The second step, includes the server-side of the application and the underlying push service that each browser uses to deliver notifications. The server-side of the application, after receiving a push subscription from the client as shown in figure 11, when it is time to send a push notification to the client, makes a HTTP call to the push service, an information retrieved from the user subscription. The structure and the schema of that API call is defined by the Web Push Protocol [44]. Each browser can use any push service it wants, and this is not a problem because every push service expects the same valid API call with the defined schema and format. The API call requires certain headers to be set and the data to be a stream of bytes that are encrypted. So, push service is handled by the underlying browser, which has its own notification delivery service. Chrome uses Google Cloud Messaging, Firefox uses MDN servers, and Safari uses Apple Push Notification Service (APNS) but it does not yet support the Push API. When you trigger a push message, the push service will receive the API call and queue the message. This message will remain queued until the user's device comes online and the push service can deliver the messages. The instructions you can give to the push service define how the push message is queued. The flow of the process can be shown in the following figure.

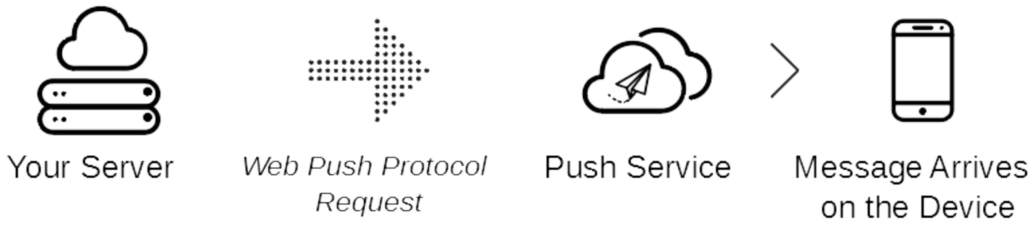


Figure 12: Web push second step overview [43]

Finally, in step three, and when the message has successfully delivered from the push service to the device, and has not expired, the browser receives this push message, decrypts the data and dispatches a push event to the registered service worker. Then the service worker, which is listening to push events that are arrived from the browser, will grab it and finally deliver to the user’s device as a notification. The final step is shown below.

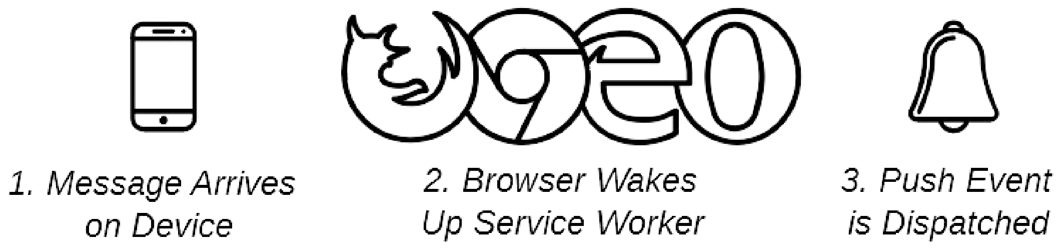

1. Message Arrives on Device
2. Browser Wakes Up Service Worker
3. Push Event is Dispatched

Figure 13: Web push step three overview [43]

3.7 Background sync

Until now, we have seen all the patterns and technologies that enable us to store our apps assets and data, in order to be able to boot and work offline. But beside the data that an application retrieves, usually in a web application HTTP context via GET HTTP requests, there are also data that need to be sent to the server in order to create or edit or patch a resource. An offline application must be able to queue somewhere the requests that get rejected due to network failure, and when it gets online again, send that requests in order to create or modify application state and data.

Background synchronization of data is handled again with a registered service worker in the page and the *onsync* event which can fire in the background so that synchronization attempts can continue despite adverse conditions when initially requested. Background Sync API [45] is intended to reduce the time between content creation and content synchronization with the server. The failed requests due to unreliable network can be queued on a web storage API, such as IndexedDB, and when the connection is back again, re-send those failed requests and clear the data of IndexedDB if the request successfully resolves. Below there is an example of background synchronization.

Table 5: Background sync page context

```
function sendChatMessage(message) {
  return addChatMessageToOutbox(message).then(() => {
    // Wait for the scoped service worker registration to get a
    // service worker with an active state
    return navigator.serviceWorker.ready;
  }).then(reg => {
    return reg.sync.register('send-chats');
  }).then(() => {
    console.log('Sync registered!');
  }).catch(() => {
    console.log('Sync registration failed :(');
  });
}
```

In the above code, we suppose to send chat messages via *addChatMessagesToOutbox* function, where we can store the data – message we want to send in a IndexedDB instance, which have access to service worker, and finally we register to service workers sync event with a string tag, here named ‘send-chats’. The sync event tag must be unique for every sync operation we want to achieve. After that, in our service worker file, we list to sync events, and finally do our operations such as sending HTTP requests for every tag we have defined. Again, with the usage of IndexedDB, we can retrieve the data for that particular request, so we have the payload to be send. The below example shows the operation from the service worker’s scope. One thing to notice here in the code example, is the *event.waitUntil* method, which means that service worker will wait until the request if fulfilled, thus the network is back again.

Table 6: Background sync service work context

```
self.addEventListener('sync', event => {
  if (event.tag == 'send-chats') {
    event.waitUntil(
      getMessagesFromOutbox().then(messages => {
        // Post the messages to the server
        return fetch('/send', {
          method: 'POST',
          body: JSON.stringify(messages),
          headers: { 'Content-Type': 'application/json' }
        }).then(() => {
          // Success! Remove them from the outbox
          return removeMessagesFromOutbox(messages);
        });
      }).then(() => {
        // Tell pages of your success so they can update UI
        return clients.matchAll({ includeUncontrolled: true });
      }).then(clients => {
        clients.forEach(client => client.postMessage('outbox-processed'))
      })
    );
  }
});
```

Background Sync API is not yet standardized, and has been implemented on Chrome and Android since 2016. The implementation supports one-time synchronization and a new API is under the hood called Period Background Sync[46] which allows to periodically sync requests given on a time interval. Although, we represent background sync API for the context of creating or updating a resource offline, the API can be used in order to client polling data from the server every x interval, like email clients do by receiving emails every x minutes and updating our inbox.

4 Depo - A warehouse management system client

A warehouse management system's primary goal is to to manage the movement and storage of items within a warehouse, where handling the connected transactions is a main part of the supply chain. WMS systems also manage the stock based on real-time information about the status of items and storage locations. There is no doubt that a WMS system is significant for most industry businesses that have storage requirements and are importing, exporting, transferring stock items from their warehouse. A WMS can be an independent system or a module of a larger ERP system.

4.1 Problem definition

One of the main core functionalities of a WMS system as described earlier, is the ability to import, export and transfer stock items from/to a location(from now it will defined as Bin). The main reason a business domain like that was chosen in order to present a progressive web app, is that in a warehouse location, that are usually hidden places and underground locations, the connectivity is not reliable. An operator can be in a location in the warehouse where for some reason has no internet, thus offline, or is in a lie-fie mode, which means that some operations may not me fulfilled and finally warehouse can end up with stale stock data.

An example to demonstrate this, is that an operator x1 which is in offline mode, exports stock from b1 location which in the given moment t1 shows to have available stock, but an another operator x2 which is online, has already exported from that location b1 in the time of t2, resulting that the item is out of stock for the operation applied from x1. As a result, operator x1 will end up by not actually exporting the desired amount of stock items, because the stock item has no available quantity to export, so will have to adjust again the stock, by exporting less or not exporting at all.

4.2 Requirements

The requirements for the system are the following and described as use cases:

1. Sign in, user can be able to login and authenticate and getting authorized in the system, and see whatever is granted to see from the menu. One user may only import items, or can export too from the locations.

2. Dashboard that includes all the stock movements recently made in the warehouse, in form of a timeline, with the recent movements first. The dashboard shows each type of movement, import or export, code of item, code of bin, and the quantity moved. While the user is offline, the dashboard have to serve the last stock movements retrieved from the server.
3. Import functionality, given the item code, the bin code, and the desired quantity, an operator can import items to the given bin. If item or bin does not exist, user must be informed. Finally, if the given bin is not assigned to that item, user must be informed that the given bin is not the storage location of that item. In offline or lie fie mode, operations must be queued and finally send when user has gained connectivity in the order they applied.
4. Export functionality, given the item code, the bin code, and the desired quantity, an operator can export items from the given bin. If item or bin does not exist, user must be informed. Also, if the given bin is not assigned to that item, user must be informed that the given bin is not the storage location of that item. Finally, if the quantity supplied to export is not available, thus out of stock, user must be informed that the desired quantity to export is not available, so can't export because it gets out of stock. In offline or lie fie mode, operations must be queued and finally send when user has gained connectivity. If an export operation fails because it gets out of stock, user must be notified by push notification for the given transaction, in order to adjust it later.
5. Settings, user preferences for the application's interface, such as locale, theme properties, dark / light mode. Setting's must be stored locally in order when booted offline to have the desired setting's.
6. The system must be operational on offline – lie-fi mode, so the application must start and boot fast even without internet, application assets are cached in the browser and served from there, application state and data are persisted in the browser, background sync is enabled for the operations of import/export, push notifications are send to the user if subscribed to get informed for failed operations of exporting.

4.3 Technologies used

The web application was developed following the client-server architecture. The communication between the client and the server is achieved using the HTTP protocol, and the client requests for resources through a Restful API exposed by the server .

The back-end is developed using the *ASP.NET CORE* framework and additionally the .net core runtime. It was developed following the Domain-Driven Design concept and patterns, an approach to software development that centers the development on programming a domain model that has a rich understanding of the processes and rules of a domain [47]. Furthermore, it implements the clean architecture principles, which provides a loosely-coupled, dependency-inverted architecture [48]. The structure of the project, consists of a Core project, where application domain entities and business logic lives, an infrastructure project where application's dependencies on external resources are implemented, such as the connection to SQL Server for data persistence and web push functionality. Finally, there is the Web project, where a Restful API is exposed for the communication with other systems.

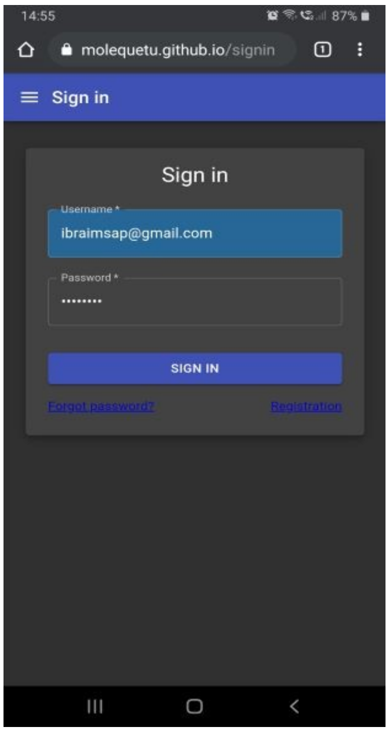
The front-end, was developed using React JavaScript library, which is used to to build user interfaces and SPA applications. React is a component based library, and used the virtual DOM for manipulating the state of the UI. By using the virtual DOM, react computes the difference between the previous state and the current state, and renders only the changes to the DOM, which makes it really fast. It offers a template to quick start a SPA project, named as create-react-app, which is a full environment configured out of the box, using tools such as Webpack module blunder, which delivers our static assets bundled and optimized. For the interface, Material Design was implemented, through material-ui react components [49]. Material Design is a design system offered by Google, which is clean and follows a more application like design approach. Finally, for the scope progressive characteristics, workbox library was used. Workbox [50] is a library provided by Google, which gives out of the box modules to implement progressive web app characteristics. It is recommended for production usage of service workers and other progressive web app characteristics.

Both applications, the back-end and the front-end progressive web app client, are configured for CI/CD. GitHub was used to host the source code, and continues integration with continues deployment are managed through GitHub Actions and Azure cloud instances.

4.4 Implementation

Following the requirements provided, application consists of various components to implement the use cases.

After the first visit to the web application and successful sign in, an install button is offered in order to install the application to the target device. Application assets such as HTML, CSS, JavaScript, images, fonts and other static assets are stored in the Cache using the Stale While Re-validate strategy. Some figures of the client's first visit are given below, presenting the steps of installing the application to the device.



Username * ibraimsap@gmail.com

Password * ••••••••

Figure 15: First visit sign in

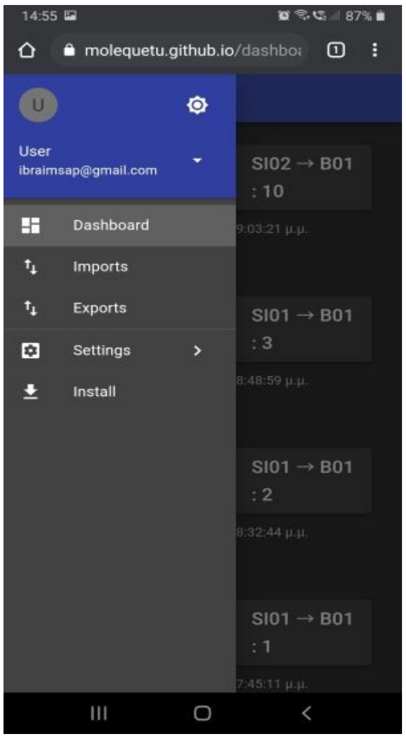


Figure 14: Menu install button

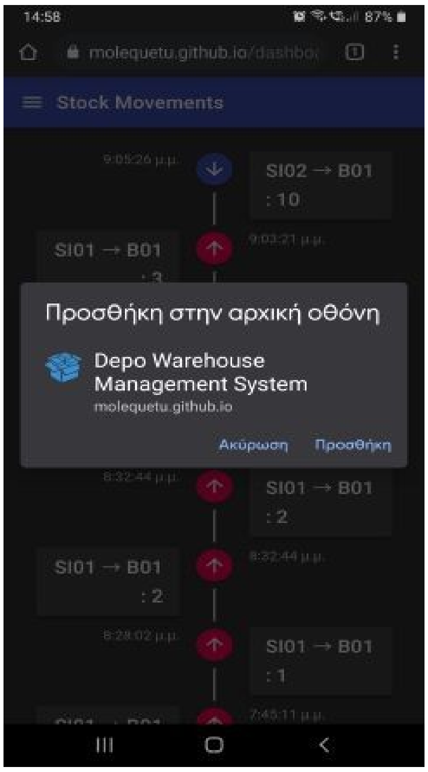


Figure 17: Installation prompt from browser

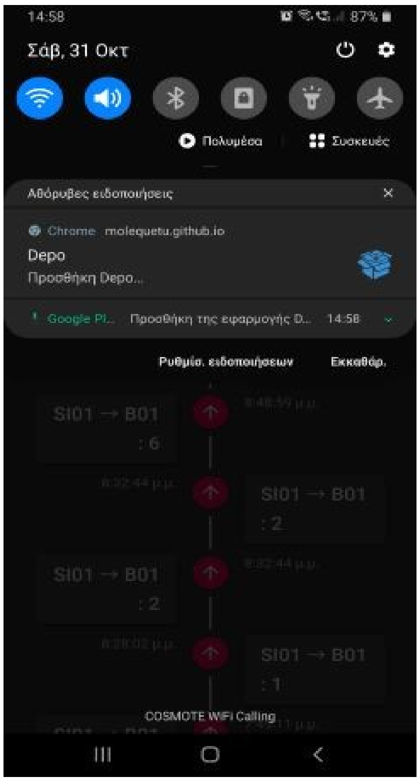


Figure 16: Installing in the background

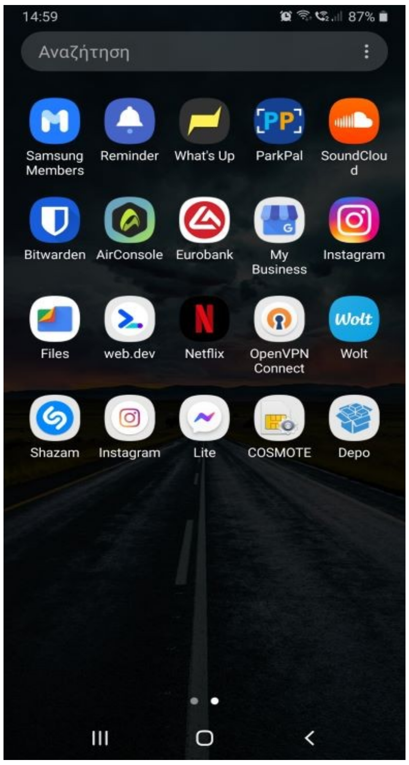


Figure 19: Added to home screen

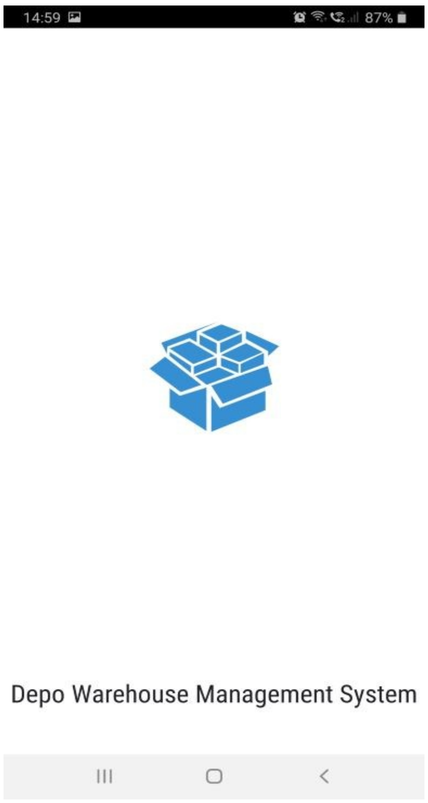
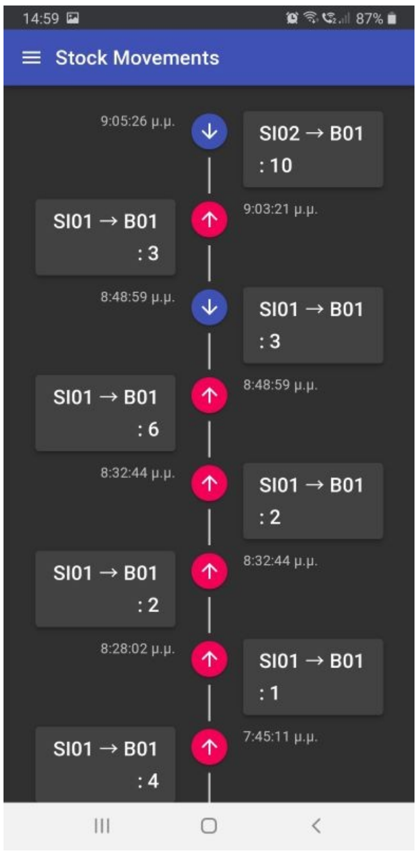


Figure 18: First boot from device - splash screen

The application has been successfully installed to the Android mobile device and it is ready to be used as a standalone application. Web manifest and a registered service worker offered the ability to add to screen. All static required assets are stored in the cache and retrieved from there either online or offline.

The first component in the menu is the Dashboard component, which consists of a StockMovements container that renders the Timeline component. Each timeline row renders information about the stock movement made, order by date descending. Application data is retrieved from the Rest API, and immediately stored to a IndexedDB store, in order to use them when offline. So it follows the network first strategy, falling back to the browser database when there is no connection to retrieve the latest stored data. User is always informed when has no connectivity.


14:59 87%

Stock Movements

9:05:26 μ.μ. SI02 → B01 : 10

9:03:21 μ.μ. SI01 → B01 : 3

8:48:59 μ.μ. SI01 → B01 : 3

8:48:59 μ.μ. SI01 → B01 : 6

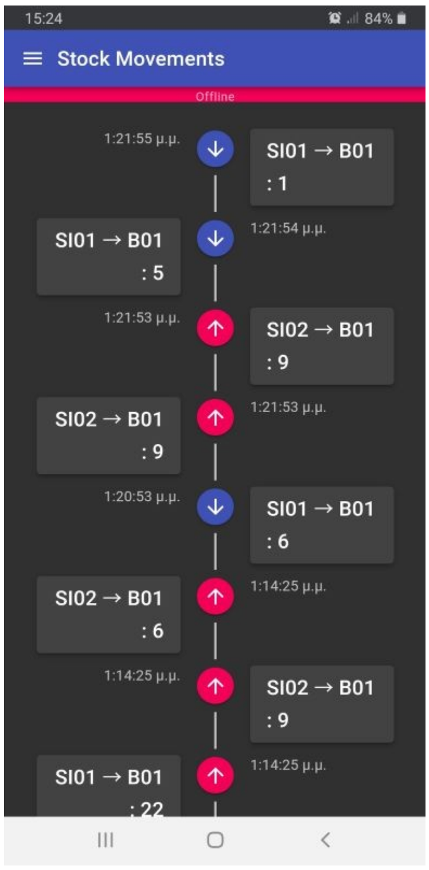
8:32:44 μ.μ. SI01 → B01 : 2

8:32:44 μ.μ. SI01 → B01 : 2

8:28:02 μ.μ. SI01 → B01 : 1

7:45:11 μ.μ. SI01 → B01 : 4

Figure 20: Stock movements timeline


15:24 84%

Stock Movements

Offline

1:21:55 μ.μ. SI01 → B01 : 1

1:21:54 μ.μ. SI01 → B01 : 5

1:21:53 μ.μ. SI02 → B01 : 9

1:21:53 μ.μ. SI02 → B01 : 9

1:20:53 μ.μ. SI01 → B01 : 6

1:14:25 μ.μ. SI02 → B01 : 6

1:14:25 μ.μ. SI02 → B01 : 9

1:14:25 μ.μ. SI01 → B01 : 22

Figure 21: Stock movements offline

The last component we will present is the export component, which is identical to import component. User defines the item to export, the bin location from where the item is stored, and the desired quantity to export. If any error's occur during the export transaction, user is informed. When the supplied quantity is not available, user gets a message that the is no such amount to export. While offline, requests are stored in a queue, in IndexedDB browser database, and when online again, are retrieved from the queue and send to the server, using the Background Sync API. The workflow of exporting items from a bin is shown in the following figures.

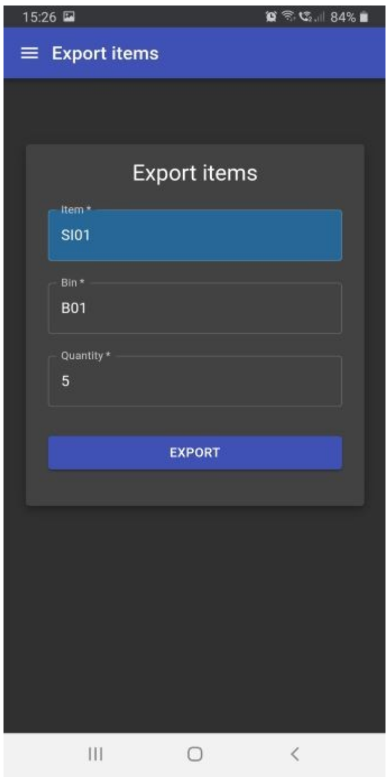


Figure 23: Export items form

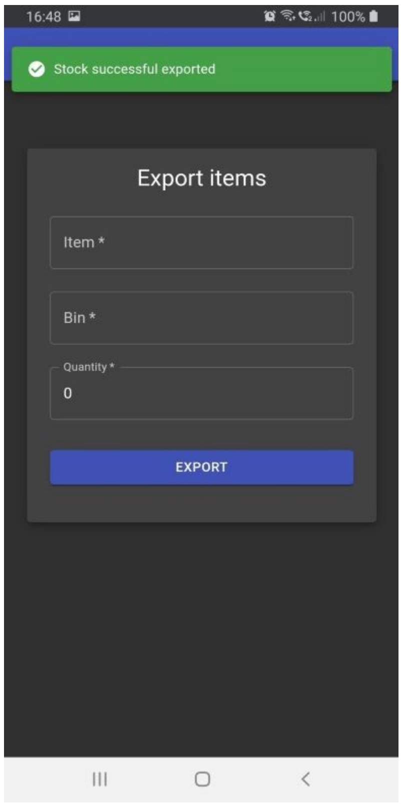
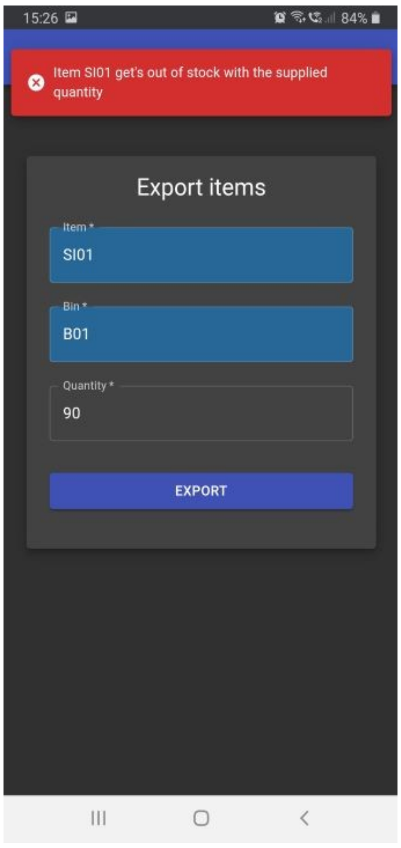
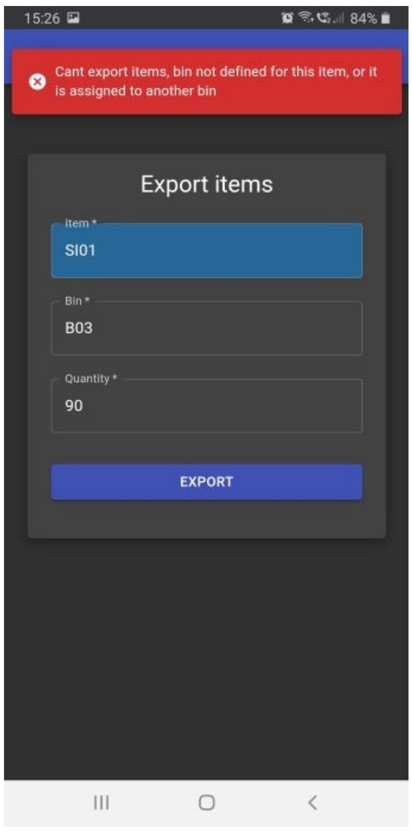


Figure 22: Export items success


15:26 84%

Item SI01 get's out of stock with the supplied quantity

Figure 24: Export failure out of stock


15:26 84%

Cant export items, bin not defined for this item, or it is assigned to another bin

Figure 25: Export items failure item not assigned to given bin

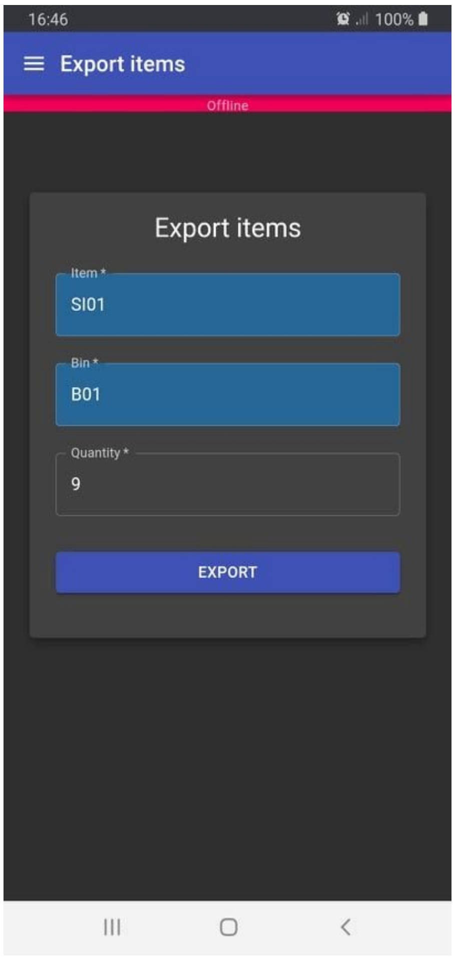


Figure 26: Offline export

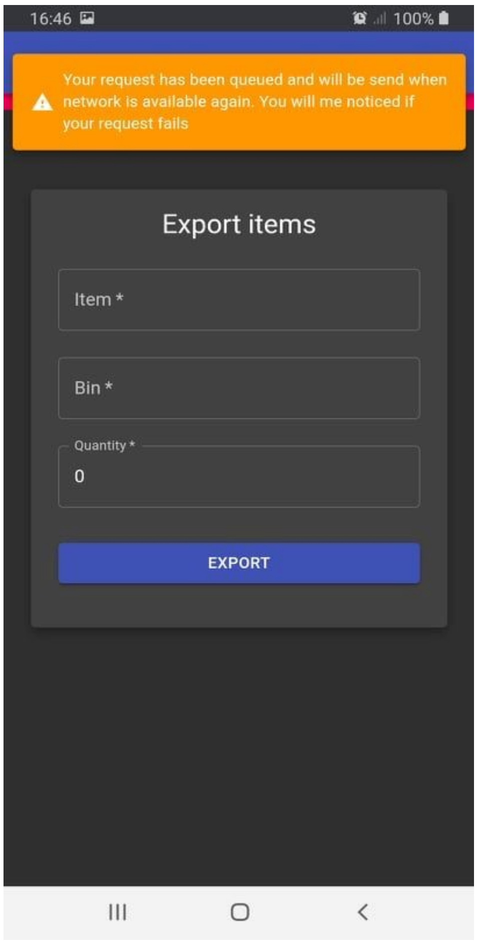

Your request has been queued and will be send when network is available again. You will me noticed if your request fails

Figure 27: Offline export notify user

5 Conclusion

5.1 Thesis overview and discussion

The objective of this thesis was to analyze the characteristics, technologies and architectures used while developing web applications that can target different platform's and devices with the power of the Web.

This type of applications, named as progressive web apps, are able to behave like native applications and have the reach that the web offers. After the implementation of the Depo client, all the requirements have been met and implemented successfully, using the technologies described in previous chapters, such as service workers, web manifest, and various web APIs for data storage, background synchronization, push notifications and more. The cost of development is much lower than of native applications, and with one code base the application can be delivered to different platform's such as desktop and mobile. Furthermore, deployment of the application is much easier and does not require complex, time consuming specific App Store deployments. Another thing worth to mention, is the size of the produced artifact, thus the delivered application, which is much more lower than of native and other cross platform solutions. In particular, Depo client, after installation, consumes only 324KB of storage, where a package produced from other solutions would definitely need a lot of Mb's.

However, all these technologies that enable the integration with the device running the application, are new and not yet mature. On the other side, the problem and the cold war between the browser vendor's that some have partially implemented or not implemented at all some APIs, can bring problems and confuse the users using them. As an example on the application that was developed, is the ability to background sync data, which is only implemented on Chrome and Chromium based browsers such as Edge, Opera, Samsung Internet, where the feature is missing from Apple and Firefox browser.

All these technologies that enable the web applications to get more closer to the mobile world revolution, can deliver applications that are identical to native, but with less cost of development, testing and deployment. However, if an application has high requirements from the device that is running, such as gaming applications, progressive web apps are not ready yet to integrate and perform well. But with the raise of Web Assembly, maybe we can see applications with higher demands on the browser too. The web technologies are emerging fast, new APIs are currently developed, but the lack of browser vendors to agree and implement the same features can result to problems, like some features running on one browser only. Finally, developers must always check and detect by feature each API they use in order to give this capabilities which is time consuming. Below we provide a table comparing all the cross-platform development approaches and native ones.

Table 7: Comparison of different cross-platform development approaches

	Web – PWA	Hybrid	Hybrid-Native	Cross-Complied	Native
Features	Install able, Offline, Background Sync, Push notifications	Install able, Offline, Background Sync, Push notifications	Install able, Offline, Background Sync, Push notifications	Install able, Offline, Background Sync, Push notifications	Install able, Offline, Background Sync, Push notifications
Device Integration	Medium	Medium	Medium	High	High
User Interface – User Experience	Medium	Medium	High	High	High
Performance	Medium	Medium	High	High	High
Cost of development	Low	Medium	Medium	Medium	High
Investment (time and developers)	Low	Medium	Medium	High	High
Cross platform	Yes	Yes	Yes	Yes	No

Cross device	Web, Desktop, Mobile, Laptop, any device that has browser	Mobile, Desktop requires dependencies (Electron)	Mobile, Desktop requires bridges	Mobile only	Mobile only
Discover-ability	High	Medium	Medium	Medium	Medium
Distribution	Web browser, App stores(Google play)	App Stores	App Stores	App Stores	App Stores
Languages	HTML5, CSS3, JavaScript	HTML5,CSS3, JavaScript	JavaScript	C#, Ruby	Java, Objective-C/Swift
Dependencies	None	PhoneGap, Ionic	React Native, NativeScript	Xamarin, Rubymotion	None

As of the table illustrated above we can come up with the result that all solutions have their advantages and disadvantages. Native development is expensive in terms of development time and resources, but great in performance and device integration. Deployment and distribution is harder for native and other solutions compared to progressive web applications. The majority of solutions offer packages for mobiles, while other offer for other devices too, by having a single code base either out of the box, or having the need of external dependencies. Most important factor although for businesses, is the investment in means of people, -developers-, and time. Web technologies obtained the majority of the technological market and many developers are familiar with at least one web technology stack. On the other hand, mobile and platform-specific developers for solutions/frameworks such as Xamarin and React Native, are more rare in the market and be high-salaried as well.

5.2 Future work

While Depo was a proof of concept in order to inspect the cross platform and cross device availability of a web application, and the ability to run offline, more features could be added in order to try and test new web APIs that are currently developed. One interesting feature could be to add the ability for bar code scanning for both the item and bin entities, which in fact are identified by bar code values, by using the camera of the device and the new Barcode Detection API or Shape Detection API. Usually the work flow is that the operator uses a bar code scanner connected with the device to enter the values of item, bins and other associated entities. By adding the feature of native bar code scanning from the application, businesses will not spend money for such devices, resulting to much lower cost.

Another interesting topic worth to study, would be to develop a progressive web app using Web Assembly. These technologies combined together may definitely narrow the gap between web – hybrid applications and native applications, for applications which require high resources from the device they running.

6 References

[1] Progressive Web Apps: Escaping Tabs Without Losing Our Soul. https://infrequently.org/2015/06/progressive-apps-escaping-tabs-without-losing-our-soul/ [Last accessed: 10 September 2020]

[2] Progressive Web Apps. https://web.dev/progressive-web-apps/

[Last accessed: 10 September 2020]

[3] App Updates for HTML5 Apps. https://developer.apple.com/news/?id=01212020a

[Last accessed: 11 September 2020]

[4] Apple shame on you. https://twitter.com/johnwilander/status/1139278479436865536

[Last accessed: 11 September 2020]

[5] Progressive web apps. https://firt.dev/ios-14/#progressive-web-apps [Last accessed: 11 September 2020]

[6] One last thing June 11 2007, 18 days before shipping the iphone. https://youtu.be/ZlE7dzoD6GA [Last accessed: 11 September 2020]

[7] Cisco Annual Internet Report (2018–2023) White Paper. https://www.cisco.com/c/en/us/solutions/collateral/executive-perspectives/annual-internet-report/white-paper-c11-741490.html [Last accessed: 12 September 2020]

[8] Global App Revenue Grew 23% in 2018 to More Than $71 Billion on iOS and Google Play. https://sensortower.com/blog/app-revenue-and-downloads-2018 [Last accessed: 12 September 2020]

[9] What are Progressive Web Apps? https://web.dev/what-are-pwas/ [Last accessed: 14 September 2020]

[10] Progressive Enhancement.

https://developer.mozilla.org/en-US/docs/Glossary/Progressive_Enhancement [Last accessed: 12 September 2020]

[11] Advantages of web applications.

https://developer.mozilla.org/en-US/docs/Web/Progressive_web_appsIntroduction#Advantages_of_web_applications [Last accessed 12 September 2020]

[12] Web app manifests. https://developer.mozilla.org/en-US/docs/Web/Manifest [Last accessed: 12 September 2020]

[13] Integrate with the OS sharing UI with the Web Share API. https://web.dev/web-share/ [Last accessed: 14 September 2020]

[14] A contact picker for the web. https://web.dev/contact-picker/ [Last accessed: 14 September 2020]

[15] Push API. https://developer.mozilla.org/en-US/docs/Web/API/Push_API [Last accessed: 16 September 2020]

[16] Notifications API.

https://developer.mozilla.org/en-US/docs/Web/API/Notifications_API [Last accessed: 16 September 2020]

[17] What makes a good Progressive Web App? https://web.dev/pwa-checklist/ [Last accessed: 16 September 2020]

[18] Lighthouse PWA Analysis Tool.

https://developers.google.com/web/ilt/pwa/lighthouse-pwa-analysis-tool [Last accessed: 16 September 2020]

[19] PWAs vs native web apps. https://firt.dev/pwa-vs-native-web/ [Last accessed: 19 September 2020]

[20] Progressive Web Apps: A Novel Way for Cross – Platform Development, Patrick Mole, Saint Luis University [Last accessed: 19 September 2020]

[21] Web Assembly. https://webassembly.org/ [Last accessed: 21 September 2020]

[22] All Showcases tagged: progressive-web-apps.

https://developers.google.com/web/showcase/tags/progressive-web-apps [Last accessed: 19 September 2020]

[23] Find out how you stack up to new industry benchmarks for mobile page speed.

https://www.thinkwithgoogle.com/marketing-strategies/app-and-mobile/mobile-page-speed-new-industry-benchmarks/ [Last accessed: 19 September 2020]

[24] Twitter Lite PWA Significantly Increases Engagement and Reduces Data Usage.

https://developers.google.com/web/showcase/2017/twitter [Last accessed: 23 September 2020]

[25] A Pinterest Progressive Web App Performance Case Study.

https://medium.com/dev-channel/a-pinterest-progressive-web-app-performance-case-study-3bd6ed2e6154 [Last accessed: 23 September 2020]

[26] 2019 — The Year of Progressive Web Apps https://medium.com/datadriveninvestor/2019-the-year-of-progressive-web-apps-3027aea291f9 [Last accessed: 17 September 2020]

[27] Why HTTPS matters. https://web.dev/why-https-matters/ [Last accessed: 25 September 2020]

[28] Secure contexts.

https://developer.mozilla.org/en-US/docs/Web/Security/Secure_Contexts [Last accessed: 25 September 2020]

[29] The App Shell Model.https://developers.google.com/web/fundamentals/architecture/app-shell [Last accessed: 28 September 2020]

[30] Service worker API.

https://developer.mozilla.org/en-US/docs/Web/API/Service_Worker_API [Last accessed: 28 October 2020]

[31] Introduction to Service Worker.

https://developers.google.com/web/ilt/pwa/introduction-to-service-worker [Last accessed: 23 October 2020]

[32] Using Service Workers.

https://developer.mozilla.org/en-US/docs/Web/API/Service_Worker_API/Using_Service_Workers [Last accessed: 25 October 2020]

[33] Cache. https://developer.mozilla.org/en-US/docs/Web/API/Cache [Last accessed: 18 October 2020]

[34] The App Shell Model.https://developers.google.com/web/fundamentals/architecture/app-shell [Last accessed: 28 September 2020]

[35] The Offline Cookbook. https://web.dev/offline-cookbook/ [Last accessed: 18 October 2020]

[36] Creating a valid manifest.

https://developer.mozilla.org/en-US/docs/Web/Manifest#Creating_a_valid_manifest [Last accessed: 19 October 2020]

[37] Web Storage API.

https://developer.mozilla.org/en-US/docs/Web/API/Web_Storage_API [Last accessed: 12 October 2020]

[38] The structured clone algorithm.

https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API/Structured_clone_algorithm [Last accessed: 12 October 2020]

[39]Basic concepts behind IndexedDB.

https://developer.mozilla.org/en-US/docs/Web/API/IndexedDB_API/Basic_Concepts_Behind_IndexedDB [Last accessed: 13 October 2020]

[40] File System API. https://developer.mozilla.org/en-US/docs/Web/API/FileSystem

[Last accessed: 15 October 2020]

[41] The File System Access API: simplifying access to local files.

https://web.dev/file-system-access/ [Last accessed: 17 October 2020]

[42] StorageManager API.

https://developer.mozilla.org/en-US/docs/Web/API/StorageManager [Last accessed: 17 October 2020]

[43] How Push Works. https://developers.google.com/web/fundamentals/push-notifications/how-push-works [Last accessed: 21 October 2020]

[44] Generic Event Delivery Using HTTP Push draft-ietf-webpush-protocol-12

https://tools.ietf.org/html/draft-ietf-webpush-protocol-12 [Last accessed: 21 October 2020]

[45] Web Background Synchronization. https://wicg.github.io/background-sync/spec/ [Last accessed: 21 October 2020]

[46] Web Periodic Background Synchronization API. https://developer.mozilla.org/en-US/docs/Web/API/Web_Periodic_Background_Synchronization_API [Last accessed: 22 October 2020]

[47] Domain Driven Design, Martin Fowler.

https://martinfowler.com/bliki/DomainDrivenDesign.html [Last accessed: 28 October 2020]

[48] The clean architecture. https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html [Last accessed: 29 October 2020]

[49] Material UI. https://material-ui.com/ [Last accessed: 30 October 2020]

[50] Workbox. https://developers.google.com/web/tools/workbox [Last accessed: 31 October 2020]

7 Appendix

Source code of the back-end implementation:
https://github.com/molequetu/depox

Source code of the front-end implementation:
https://github.com/molequetu/depox-pwa

Depo client progressive web app live instance:
https://blue-water-0241cf303.azurestaticapps.net/