Achieving native-like experience on the web with progressive web apps

Master of Science Thesis
University of Turku
Department of Future Technologies
Software Engineering
2020
Oskari Lindén

UNIVERSITY OF TURKU
Department of Future Technologies

Oskari Lindén: Achieving native-like experience on the web with progressive web apps

Master of Science Thesis, 57 p., 8 app. p.
Software Engineering
July 2020

For long developing applications for mobile has required developing a separate app for each platform that the developer wants the app to support. Apple has App store, Google has Play store and Microsoft has Microsoft store. Web apps can be used for creating apps that work on all devices with modern web browsers. Web apps have traditionally been outperformed by native apps and therefore been less popular when comparing to native application development. Progressive web apps (PWAs) are new kind of apps that aim to combine the best of native and web apps. These apps can be installed to the user's device and have access to some of the features that have only been available on native apps in the past.
In this thesis we will compare PWAs with their native counterparts to find out if these apps really deliver what is promised and work as real unifiers between native and the web. We will list the benefits and drawbacks of developing PWAs. We will also turn an existing web app to PWA and see what benefits can be obtained. The results indicate that today PWAs can really be seen as a viable option compared to native apps and that there are barely any features today that could not be implemented with a PWA solution.

Keywords: Progressive Web Apps, Web Apps, Native Apps, Performance

UNIVERSITY OF TURKU
Department of Future Technologies

Oskari Lindén: Achieving native-like experience on the web with progressive web apps

Master of Science Thesis, 57 p., 8 app. p.
Software Engineering
July 2020

Jo pitkään mobiilisovelluskehityksessä on tarvinnut luoda oma erillinen sovelluksensa jokaiselle alustalle, jota halutaan tukea. Applella on App store, Googlella Play kauppa ja Microsoftilla Microsoft store. Luomalla verkkosovellus voidaan taata, että sovellus toimii kaikilla laitteilla, joissa on moderni verkkoselain. Nämä sovellukset ovat kuitenkin perinteisesti toimineet natiiveja sovelluksia huonommin ja olleet täten epäsuosiossa niihin verrattuna. Progressiiviset verkkosovellukset (eng. Progressive web app, PWA) ovat uudenlaisia sovelluksia, joiden luvataan yhdistävän parhaat puolet natiiveista ja verkkosovelluksista. Nämä sovellukset on mahdollista asentaa käyttäjän laitteelle ja niillä on pääsy joihinkin ominaisuuksiin, joihin aiemmin vain natiiveilla sovelluksilla on ollut pääsy.
Tässä tutkielmassa verrataan progressiivisia verkkosovelluksia vastaaviin natiiveihin sovelluksiin, ja selvitetään, vastaako todellisuus sitä, mitä näistä sovelluksista luvataan. Työssä listataan kaikki edut ja haitat, joita PWA-kehitykseen liittyy. Tutkielmassa myös muunnetaan jo olemassa oleva verkkosovellus progressiiviseksi verkkosovellukseksi, ja nähdään, mitä etuja tällä saavutetaan. Tulokset osoittavat, että PWA on tänäpäivänä toimiva vaihtoehto natiivin sovelluksen korvaajaksi. Ei ole olemassa montaakaan sellaista ominaisuutta, jota PWA ei vielä tukisi.

Keywords: Progressiiviset verkkosovellukset, Verkkosovellukset, Natiivit sovellukset, Tehokkuusmittaus

Contents

Listings

List of Figures

List of Tables

List Of Abbreviations

- **A2HS**: Add to home screen
- **AJAX**: Asynchronous JavaScript And XML
- **AMP**: Accelerated Mobile Pages
- **API**: Application Programming Interface
- **CTR**: Click-through rate
- **HTTPS**: Hypertext Transfer Protocol Secure
- **JSON**: JavaScript Object Notation
- **JWT**: JSON Web Token
- **PWA**: Progressive web application
- **TCP**: Transimission Control Protocol
- **TSL/SSL**: Transport Layer Security / Secure Sockets
- **VAPID**: Voluntary Application Server Identification

1 Introduction

According to Holst [1] the number of smartphone users worldwide in 2020 is 3.5 billion and will increase to 3.8 billion in 2021. The development of mobile applications is shifting from native applications to a web based approach as new kind of apps known as Progressive web apps were presented in 2015. [2]

Many companies have already discovered the benefits of the progressive web apps, one example of these is Tinder. The PWA they developed takes only 10% of the storage space compared to their native app and the loading time of the app dropped from earlier 11.91 seconds to 4.69 seconds when switched to PWA. [3] Many are even saying that today the PWA development should be preferred over native application development and that the PWAs will eventually eliminate the need for native apps. [4]–[6][7, p. 9] There are also people who claim the opposite. [8], [9]

Google is currently working on a project where their main goal is to close the ”app gap” between native apps and web apps. The term ”app gap” refers to the feature gap there currently is between these app types as the native apps are able to do things that web apps can not. For example, web apps are not able to access the file system of the device. There are many interesting upcoming features that the web apps will soon support such as access to the user's contacts via Contact Picker API and accessing the file system directly via Native File System API just to name a few. [10] The Contact Picker API was already released for mobile users early 2020 with Chrome version 80 [11].

1.1 Motivation

Progressive web apps can be used as cross-platform solutions when creating mobile applications. Instead of having to create separate applications for all different platforms it is enough to create a single PWA that works basically on any device that supports modern web technologies.

The author has worked in a Finnish company HRSuunti Oy for past 3 years. The company offers workforce management solutions for other businesses. The main product of the company is HRSuunti Net which is a remote desktop application aimed for people at the human resources who plan the work shifts for employees. One of the company's products is a mobile web app called HRMobi. This application is targeted for the employees who work on companies that use HRSuunti products as their workforce management software. With HRMobi employees can view their work schedules and inform foreman the actual time they were working. They can also see contact information of their foreman and other employees. With bulletin board functionality foremen can easily send urgent messages for employees.

HRMobi has been developed as a web application from the start instead of making it a native app. This choice was made because one of the company's main selling point is that no downloads or installations are required to use their software. As HRMobi is a web app it can be accessed simply by typing the URL in the browser. Shifting to progressive web app seems the best choice as this allows to get some benefits of native applications while still maintaining the ease of access.

The application could benefit from the opportunities that progressive web apps represent. For example sending push notifications would allow users to be notified of urgent events quickly. Therefore the author's motivation is to gather information of PWAs and then use that knowledge to make the best PWA possible out of the existing web app, HRMobi. The acquired knowledge can further be applied to other products of the company too.

There are still quite few comprehensive articles available about progressive web apps. Another motivation for this thesis is to gather information that someone else can later use as a base for further studies on the subject.

1.2 Objective

The aim of this thesis is to compare the available technologies and best practices for creating a cross-platform mobile application which performs as well as possible with the tools available today. The main focus is on PWA as we find out whether this can be used to achieve native-like performance on the web. Another question is whether progressive web apps will eventually surpass native applications and become the main way of developing applications – for mobile at least.

Last we will use this knowledge to convert an existing web application, HRMobi, to follow the best practices found. The application already supports geolocation which is one of the features covered in this thesis. After changes our web app should be classified as a progressive web app. It should also be accessible offline and be able to receive and display push notifications.

The deployment structure of HRMobi represents one challenge for the PWA. The landing page for the app contains an organization selection form. After proceeding from this stage the app redirects to different URL depending on the chosen organization. PWAs are restricted to work on a single origin. Because of this it might be impossible to allow the installation to take place on the main page and then move user to a page within different origin.

One part of this thesis is a systematic literature review on progressive web apps. As the subject is quite new, other non-scientific sources have also been used as references. In addition experimental tests have been conducted to find out whether the results match the theory. We also present a case study on HRMobi and how the transition to PWA has affected it. The results of this study can be used as a

reference for further studies on progressive web apps.

1.3 Research questions

In this thesis we will answer the following research questions about Progressive web apps.

- **RQ 1**: What is the actual definition of PWA?
- **RQ 2**: What benefits and drawbacks developing a PWA has compared to developing a native application?
  - **RQ 2.1**: What features are currently not supported by PWA but can be accomplished with native apps?
  - **RQ 2.2**: In what cases native apps perform better than PWAs?

There are some studies which try to answer similar questions but the subject is quite new so there are not any definitive answers out there at the time. In 2019 Kerssens [12] conducted a study where they aimed to find out whether PWAs could be used as alternatives to native applications. They found out that in Android PWAs launch faster compared to native alternatives while on iOS they do not. They also found out that PWAs consume as much or even less energy than their native counter parts. Their final thought was that PWAs can be seen as viable alternatives for native apps. With our thesis we will focus more on finding out the features that are still missing from PWAs but exist on native.

1.4 Outline

After this chapter this thesis continues by presenting previous research and different app development approaches in Chapter 2. In Chapter 3, we discover what

definitions there are for progressive web apps and go into more detail with different aspects related to PWAs such as service worker and web app manifest. In Chapter 4, we compare PWAs with native apps and find out what benefits and drawbacks there are when developing PWAs instead of native applications. Chapter 5 contains performance testing with geolocation. We also discuss in detail what was done with HRMobi to transform it to a progressive web app. Finally in Chapter 6 we present the conclusions of this thesis, discuss about them and give some ideas for further studies.

2 Background

> For many years, companies have tried to enable developers to create app-like experiences using web technology.
>
> *Jason Grigsby*

In this chapter, we will take a look at previous studies related to Progressive web apps. After that we will briefly explain the different methods how applications can be developed, especially mobile apps. We will also take a look at other methods similar to PWA that have existed before and how they compare with PWA.

2.1 Previous research

Progressive web apps have not been around for very long so the subject has not been researched much yet. By conducting a search on Scopus with search string ”progressive web app” we get only 19 hits (on February 2020). By further inspecting the documents we get 14 documents which involve progressive web app as a central part of the research. The studies are pretty new as only one of the studies is from 2017 and others are even more recent. The list of studies found can be seen from Appendix A.

Seven of the 14 studies present primaly just a simple case study for PWA and

do not include much theory related to progressive web apps. Five of the studies include mainly research and contain useful information that can be used within this thesis. The remaining 3 studies contain a case study but also some useful research information.

One of the studies finds out what security issues there are with PWAs. [13] The results of this study have practical use cases as they can be taken into consideration when creating a PWA.

2.2 App development approaches

Traditionally mobile apps have been developed as native apps. The development of web technologies has opened new possibilities which might even replace the traditional way of developing apps.

2.2.1 Native applications

Native apps target only a specific platform, Android, iOS or Windows for instance. When building a native app one must create a separate application for each platform that need to be supported. [14] This is the main drawback of creating native applications as the maintenance costs increase as multiple codebases have to be maintained. This might also mean having to hire more people as different programming language skills are needed for different platforms. [15]

When writing native apps all of the device's native resources can be accessed. Usually native apps are also faster and perform more smoothly compared to hybrid or web applications.

The distribution of native apps happens through centralized platform specific marketplaces. Applications for Android devices can be downloaded from Google Play and apps for iOS from App Store to give a few examples. One advantage of

such centralized marketplaces is that the apps shown there reach a wider audience. The number of total apps available on the marketplaces is currently almost 3 billion for Google Play and approximately 2 billion for App Store [16]. This means that it might be tough to get people to download a certain app through the app store. It is only a very small amount of applications that forms the majority of apps that are actively used by consumers. [17]

2.2.2 Hybrid applications

Hybrid applications try to combine the best of native and web apps. Usually the process includes writing the program as a web application which is then wrapped inside a native wrapper for each of the targeted platforms using a web view control. [14]

Some popular frameworks today for creating hybrid apps includes Apache Cordova[1] and Electron[2]. With Cordova and Electron the app is developed using standard web technologies such as HTML5, CSS3 and JavaScript. Cordova is mainly used for creating cross-platform mobile apps as with Electron the main focus is on creating desktop apps.

The idea of creating hybrid applications is that all native cababilities are basically available which is often not the case with web appps. For instance, accessing file system is not possible with web apps but it is possible with hybrid apps.

2.2.3 Web applications

Web applications are special kind of interactive websites which function like traditional desktop or mobile applications. These apps have the advantage of using the common web technologies such as HTML5, CSS3 and JavaScript.

[1]https://cordova.apache.org/

[2]https://electronjs.org/

There is no exact definition for what makes a web site a web app. Usually a web site which functions similarly to a desktop or mobile application is described as a web application. One distinction is that a web app has a single use purpose as web sites usually contain many sections and serve multiple purposes. Web apps are also dynamic at least to some extent compared to normal web sites which may instead be static. ”Dynamic” in this case means that the content shown is specific to the user that is viewing the app. Web apps also usually rely heavily on JavaScript.

A major advantage compared to native apps is that web apps do not require any installation and can be accessed instantly anywhere from any web browser. This on the other hand means that the user must have a working network connection. This requirement does not apply to progressive web apps. In Chapter 3 we discuss how PWAs can work offline.

The main drawback with web apps used to be the fact that many native functionalities such as taking photos with the device's camera or fetching the GPS data were not available. Today all of the major browsers support most of these device capabilities which is one reason behind the growing popularity of web apps.

2.3 Application shell architecture

An application shell architecture is a technique where a simple app shell for the application is separated from other application logic. This shell contains all the critical HTML, CSS and JavaScript needed for the initial page render to happen. All content is loaded dynamically. This makes the app feel more like a native app as page transitions can be made fluid. The application shell is usually cached so that on repeated visits the shell is loaded instantaneously. [18]

According to Osmani [18] there are three main benefits of using application shell architecture:

- **Fast and reliable**: Most of the static content are cached and will load instantly on repeated visits. Dynamic content is loaded as needed when the user is using the app.
- **Native-like feel**: Navigation feels more native-like as the page loads dynamically instead of doing full re-renders. Caching also enables the app to work offline.
- **Minimal data usage**: By caching assets efficiently and loading dynamic content only when needed the amount of data downloaded from the server is much less than it would be without these techniques.

Goal is to keep the size of the app shell as small as possible so that it will allow fast load times even when visited for the first time. Because of how TCP works the best results are obtained if the shell size is less than 14kb.[3] [19]

2.4 Application cache

Application cache (shortened as AppCache) is an older way of enabling offline access to web sites. The idea is to allow caching assets such as HTML pages, stylesheets, JavaScript files, etc. on the browser side so that the pages could be accessed even when there is no internet connection [20].

One main part of progressive web apps is a script known as the service worker. This script is installed on the browser where it acts as a proxy and is able to cache files and intercept web requests for instance. We will explain this in more detail in the next chapter in Section 3.4.

[3]TCP splits data to multiple packets. The size of a single packet gets larger with more data to be sent. The first packet sent is 16kb. The first 2kb of the packet contains overhead data. So if the data to be transmitted is 14kb or less only one packet is sent instead of multiple packets.

Archibald [21] pointed out the flaws of AppCache already in 2012. Gaunt [22] agrees that there have been many issues with AppCache API and that service workers were designed to avoid these problems. Because of this service worker can be used as a replacement for AppCache.

Browser manufacturers are currently working on disabling the application cache feature completely on the their browsers. This means that after some time the feature will no longer be supported. [23]

2.5 History of progressive web apps

As early as in 2007 Steve Jobs introduced web apps as a main way of developing apps for iPhone. The release of the App Store for native iOS apps came a year later. The web technologies were not powerful enough at the time and the idea of supporting solely web apps was abandoned as the native apps performed much better. [24]

The phrase ”Progressive web app” was coined by Alex Russell and Frances Berriman on June 2015 [25]. On the blog post ”Progressive Web Apps: Escaping Tabs Without Losing Our Soul” Russell [2] explains how they came up with this phrase when they were discussing about the best practices for making a web app. The required technologies had existed for some time already but no one had given them a name.

On the blog post Russell [2] defines the PWA as a web app which has the following characteristics:

- **Responsive**: The application will scale and work with any size or form of a screen such as mobile device, tablet, TV, etc.
- **Connectivity independent**: The application will work even when offline with the help of a service worker. This of course limits the app to access only

the offline content that has been downloaded before.

- **App-like-interactions**: The application feels like a native app with app-like navigation and transitions.
- **Fresh**: The application will always be up to date thanks to the fact that it is basically a web page that can be fetched again when the user reloads it.
- **Safe**: The application will be served over a secure HTTPS connection.
- **Discoverable**: The application can be found by search engines and will be classified as an ”application” instead of a web page.
- **Re-engageable**: The application will be able to send push notifications to the user.
- **Installable**: The application can be added to the user's home screen.
- **Linkable**: The application has its unique URL so it can be shared and accessed by anyone.

This is the first definition that was given to progressive web apps. In the next chapter we will see other definitions and go into more detail on what defines a progressive web app. With the obtained knowledge we will answer RQ 1 about the definition of PWA.

3 Progressive Web App

> These apps aren't packaged and deployed through stores, they're just websites that took all the right vitamins.
>
> *Alex Russell*

Progressive web apps are web sites which behave like native apps and look like them. PWA is not a framework nor a technique but instead a combination of web technologies and a set of best practices one must follow to get all the benefits of a PWA [26].

Progressive web apps offer features that can not be achieved with regular web apps. Any PWA can be installed on the user's device in a way that it shows up as an app on the device's app list. They can also be launched in a full screen mode. By caching assets PWAs are able to offer offline experience unlike regular web apps. With the help of service workers they are also able to send user push notifications just like native applications.

Service workers are scripts that run separate from the scope of the web page. These scripts can work on the background even after the page has been closed. Service worker also works as a proxy by handling all network requests sent from the web page. We discuss more about service workers in Section 3.4. [22]

Grigsby [27, p. 6] claims that progressive web apps are built using progressive

enhancement. It is a strategy where a baseline experience is offered for everyone and then capabilities of the app are progressively enhanced based on the user's device. Ater [7, p. 21] points out that this kind of approach is not a technical requirement for a progressive web app – it is just something one should do. A simple example for a progressive enhancement is that we can serve a dynamic web page with JavaScript for users who have JavaScript enabled and an alternative static version of the web page for users who have not enabled JavaScript. It seems to be unclear what is the actual definition of PWA. Firtman [28] even claims that there is no unique definition available.

Google gives even more vague definition as they define progressive web apps being "user experiences" with the following attributes: [29]

- **Reliable**: The application loads instantly and works even when there is no network connection available. This is accomplished by the help of service workers which can cache pages and requests offline.
- **Fast**: Works smoothly and fast and responds quickly to the user input – just like a native app would.
- **Engaging**: Looks like a native app when launched as a full screen experience. Is cabable of sending push notifications to users and lives on the home screen of the user's device.

The above definition presented by Google does not describe PWAs on a technical level. The focus is more on how they behave or how they feel. These attributes work as selling points for PWAs instead of defining how they actually work.

3.1 Technical requirements for PWAs

Keith [30] presents a clear technical perspective on the case on his blog post "What is a Progressive Web App?". On that post he sums up that "a Progressive Web App is a website served over HTTPS with a service worker and a manifest file." So to open that up, there are 3 core technical requirements for a PWA:

- **1. HTTPS**: The application has to be served over a secure connection.
- **2. Service Worker**: The application must register a service worker that is able to intercept and alter requests and handle asset caching.
- **3. Web app manifest**: The application has to have a web app manifest file which describes the base attributes of the application such as it's display name, screen orientation, etc.

Even more simple approach is presented by Lee, Kim, Park, *et al.* [13] as they define PWA as being any web site that registers a service worker on the user's browser. This definition covers the demand for HTTPS and Service worker as registering a service worker only works in a secure context[1]. As registering a service worker is possible without web app manifest this definition differs from the above definition.

Google has a progressive web app checklist[2] which lists all the core requirements for PWAs as well as some other features which "optimal" progressive web apps should fullfil. One of the core requirements is that all PWAs must follow a responsive design. This means that the app should scale and look good whether the user is using a device with a small screen or a device with a larger screen. It is not clear if this is a technical requirement that would stop a web app of being a PWA as there are no exact definitions for when a page is responsive and when it is not.

[1]Secure context practically means that if the site is not served from localhost it should be served over https:// URL and use a network channel that is not considered deprecated.

[2]https://web.dev/pwa-checklist/

3.2 Web app manifest

A web app manifest is a file which describes the properties of the PWA. For the browser to ask the user to install the app, a web app manifest is needed. The manifest is in JSON format[3] and must contain at least the fields shown in Listing 3.1 which shows an example of a web app manifest file.

```
{
  "name": "My App",
  "icons": [
    {
      "src": "/images/icons-192.png",
      "type": "image/png",
      "sizes": "192x192"
    },
    {
      "src": "/images/icons-512.png",
      "type": "image/png",
      "sizes": "512x512"
    }
  ],
  "start_url": "/myapp/?source=pwa",
  "display": "standalone",
  "theme_color": "#3367D6"
}
```

Listing 3.1: Example web app manifest

The web app manifest has to be linked to the page by referring to it in the `<head>` element of the page as is shown in Listing 3.2

```
<link rel="manifest" href="manifest.json">
```

Listing 3.2: Linking the manifest to the page

[3]JSON is a file format for describing data in human readable text form

3.3 Installation

Progressive web apps can be installed on the user's device. When installed they look very similar compared to native applications installed on the device. The app can only be run on a fullscreen mode when launched from the home screen icon.

The action of installing a PWA was formerly known as "Add To Home Screen" action (shortened as A2HS). The main benefit with installing a PWA is that the user is more likely to actually use the app. [31]

The process of installing a PWA to the home screen can be seen from Figure 3.1. The image on the left shows how the app looks like when being first visited with the browser. At the bottom an install prompt can be seen. The image at the center shows what the app looks like when launched from the home screen. Finally on the right-hand side we can see that the app runs on its own window

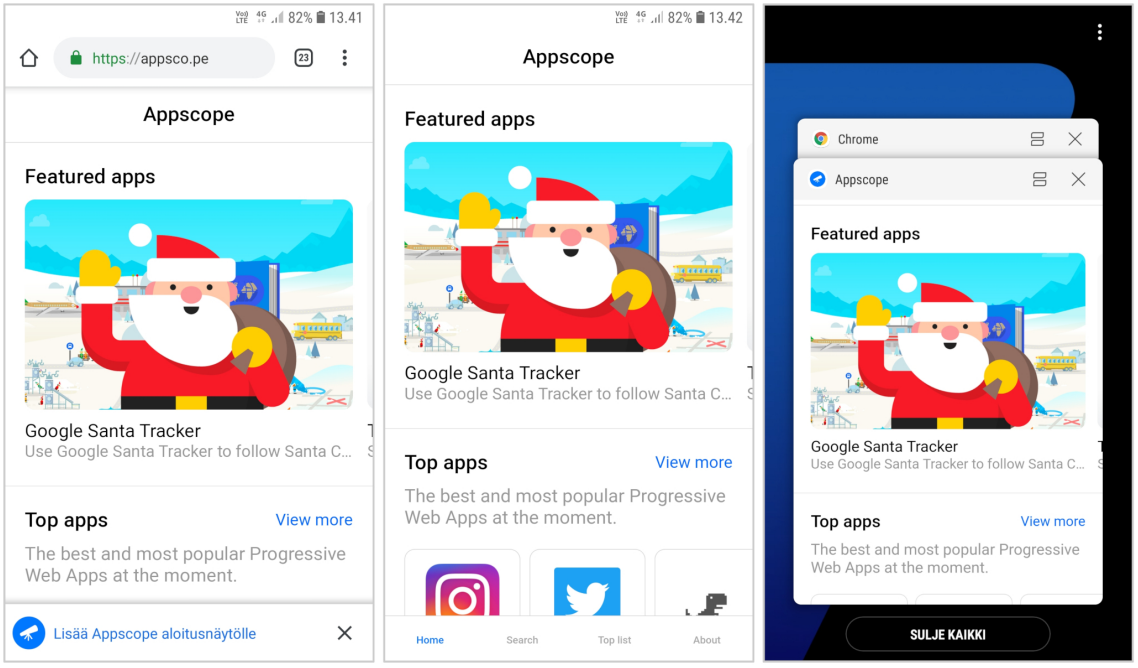

Figure 3.1: The process of installing a PWA and how it looks on the device after launched from the home screen. The screenshots have been taken with Samsung S7 device with Google Chrome version 76.

The requirements that need to be fulfilled for the installation banner to be shown may differ between different browsers. For Google Chrome and Microsoft's Edge [32] the criteria for the install banner to appear are the following:

- The application is not already installed
- The application has to be served over HTTPS and have a service worker registered with a fetch event handler.
- The application has to have a web app manifest that contains at least:
  - short_name or name
  - 2 icons: 192px and 512px
  - start_url
  - display that is set to fullscreen, standalone, or minimal-ui

[33]

Mozilla has slightly different requirements as they do not demand the installation of a service worker while they still require HTTPS. In addition they list background_color as something that is required in the web app manifest. [34]

The installation requirements for Samsung Internet differ from Google Chrome's as they require only 1 icon the size of which has to be at least 144x144. The display attribute has to be standalone or fullscreen. [35]

3.4 Service worker

A service worker is installed when a web page requests to register it. After installation the service worker acts as a proxy that can be used to cache assets and network requests. This enables the app to work and serve the user with something even when there is no network connection available. The service worker is also able to

intercept and alter any requests done within its scope, send push notifications and do background synchronization. [22]

A service worker is a JavaScript worker. Therefore it runs on its own seperated thread meaning that it cannot directly access or manipulate the DOM of the web page. Instead it can receive and send responses to messages sent from the pages via the postMessage interface. [22]

According to Gaunt [22] service worker is a successor of Application Cache which has been previously used to support offline experience. The specification for Service worker was published in 2014 and already by late 2015 many major browsers such as Chrome, Firefox, Opera and Samsung Internet had added support for service workers [7, p. 20]. From a github page by a Google employee Jake Archibald[4] we can see that all of the major browsers currently support most of the features for service workers except for background sync which is currently (June 2020) fully available only for Google Chrome.

3.4.1 Installation

Service worker makes it possible to manipulate and filter web requests. This makes the application vulnerable to so called man-in-the-middle attacks. Gaunt [22] states that because of this it is not possible for an application to register a service worker unless the application is served over HTTPS. The ”https://” prefix in front of a URL means that the web site is being served over a secure connection. The HTTPS protocol is a combination of HTTP and TLS/SSL protocols.

A service worker can be installed by calling `navigator.serviceWorker.register` within the page. A simple example of registering a service worker can be seen in Listing 3.3. A web app will not be classified as a PWA if it was not served over a secure HTTPS connection and did not register a service worker.

[4]”Is service worker ready?” (https://jakearchibald.github.io/isserviceworkerready/)

```
if ('serviceWorker' in navigator) {
  navigator.serviceWorker.register('/sw.js').then(function(registration
    ) {
     console.log('Service worker registration succeeded:', registration)
    ;
  }, function(error) {
     console.log('Service worker registration failed:', error);
  });
} else {
  console.log('Service workers are not supported.');
}
```

Listing 3.3: Installing Service Worker

3.4.2 Lifecycle

A service worker has its own lifecycle that is separated from the web page. To use service workers properly it is important to understand the different states involved in the process. The states involved in the lifecycle of a service worker are illustrated in Figure 3.2

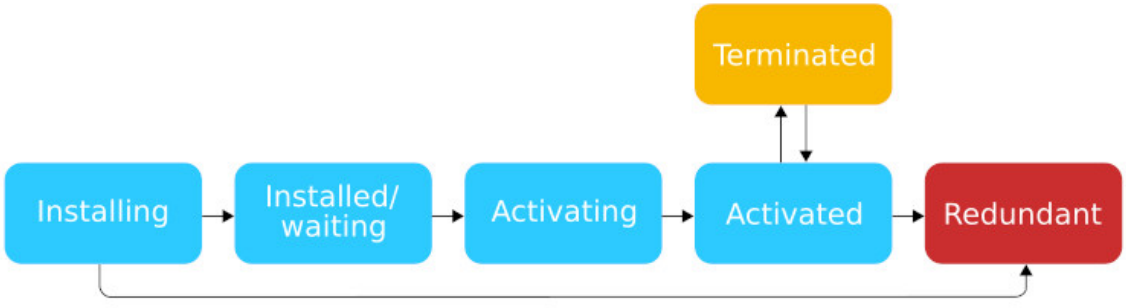

Figure 3.2: The lifecycle and different states of a service worker

Installing

The lifecycle begins when a web page registers a service worker as shown in Section 3.4.1. At this state the service worker is in the installing state.

This is when all of the required assets are downloaded to the cache. If something goes wrong the installation will fail and the service worker will immediately move to the redundant state. For instance a missing asset could cause the installation to fail. This behaviour makes sure that when the service worker is installed all of the cached assets should exist without doubt. [7, p. 44]

Installed/waiting

After being installed the service worker is in a state where it waits to get activated. The service worker cannot usually function unless it has been responsible for the page from the very beginning of the session [7, p. 46]. During this state the page may either have no active service worker at all or it may have an older version of the service worker in control.

The new service worker will become active after all active tabs have been closed and the page is visited again. If there was an older version of the service worker that service worker would now enter the redundant state.

Activating

A service worker is in an ”activating” state when the activate event of the service worker has been fired but it has not yet completed.

The activation state is normally used for removing any old cached files. Older service workers may have been using older versions of the cached assets. During this state we can be sure that these service workers are in the redundant state and therefore the old cached assets are safe to be removed.

Activated

When a service worker has been succesfully activated it enters the activated state where it is able to do all its core functionality such as intercepting network requests and sending push notifications etc.

At any point a service worker may become terminated by the browser to save memory. This behaviour cannot be controlled in any way. When the service worker receives a request it is restarted automatically. This means that global variables should not be used to store data as they can get erased anytime. [22]

Redundant

When a new service worker is activated the old service worker enters the redundant state. In this state the service worker has no impact on the page anymore.

3.4.3 Scope limitation

Because service workers are able to intercept and manipulate network messages it is important that they are registered within a certain scope. If we place the service worker file in a folder ”www.example.com/service1/” then that service worker is able to function within that scope but it is not able to intercept requests for ”www.example.com/other-service/” for example. [7, p. 27]

This behaviour is useful in many cases. One example is a case where there is a website that is hosting multiple different sub sites in different folders ”site1” and ”site2” for instance. If the scope of the servive worker would not be limited to the folder it lives in the owner of the ”site2” might use a service worker to redirect the users of the ”site1” without they knowing it. It would also mean that ”site1” and ”site2” could not both have a service worker active at the same time but because of the scope limitation they can. [7, p. 27]

3.4.4 Working offline

PWAs should function even when there is no internet connection. Of course it is not possible for the app to have all the same functionality as it would if it was online. It is important that the app can tell the user that the app is in offline state. The offline functionality can be achieved with the help of a service worker. [27, p. 83]

Service worker enables the offline functionality because it is registered locally on the users device. It can cache pages and other network request responses. When offline the Service Worker can serve pages and responses from the cache to the user.

3.5 Saving data

With PWAs there are many alternatives when it comes to saving data. In one of the articles on Web fundamentals they suggest using the Cache API for saving all the assets and requests that are needed for loading a page offline. For all other data they suggest using IndexedDB. [36]

3.5.1 CacheStorage

CacheStorage is used for storing requests and their corresponding responses for quicker access and for enabling offline access. This API is usually accessed through service worker. We will go into more detail with different caching strategies later in this chapter in Section 3.6.

3.5.2 IndexedDB

IndexedDB can be used for storing things like application state or data generated by the user. By using IndexedDB the startup of the application can be made faster as the user does not have to wait for all the initial app state related data to be fetched

from the server. IndexedDB is ideal for storing larger structured data. Indexes increase the performance when conducting searches on the stored data.

IndexedDB may not work with all browsers the same way. The user is also able to delete or alter the data in the IndexedDB. This means that the code should always be written that in mind. Have error handlers and never rely that the data actually exists in the IndexedDB. This means that there should always be a fallback for missing data. [37]

3.5.3 Web Storage

Web Storage API can be used for storing data to the user's device in key/value pairs. This is more convinient comparing to cookies and typically browsers allow more storage space to be used with Web Storage API than with cookies. [38]

Web Storage consists of two objects: sessionStorage and localStorage. The first is used for storing data that expires after the session ends in other words when the browser or the tab is closed. The localStorage object can be used for data that should be kept stored between sessions. [38]

3.5.4 Browser differences

Different browsers allow different amount of data to be saved to the user's device. Chrome allows the browser to use up to 60% of the device's total space. Firefox has a limit of 2GB for each origin. Safari allows up to 1GB to be stored. Firefox and Safari allow more space to be used but only after prompting the user to allow it. [39]

Many of the browsers will automatically remove stored data on CacheStorage and on IndexedDB if the browser runs out of allocated disk space. In Safari the data used by a single site will be deleted if the user has not interacted with the site for seven days [40].

3.6 Cache management

Service workers allow front-end developers to create their own patterns on how to handle caching. The service worker has full control over the cache so the developer can decide which items should be cached and how the cached items are updated and deleted. [7, p. 31] Items in the cache will not expire and they have to be removed within the code. Every browser has their own storage limit for cached data and if this limit is exceeded the browser will start to remove cached items one complete cache at a time until there is enough free space again. [41]

A Service Worker can be used in many ways. For instance one could save requests to the cache so that even if the app is online new information would not be fetched from the internet if the user asks for the same information within 5 minutes from the actual request.

Listing 3.4 shows a very simple example of using the Cache API. Three URLs for the application are stored in the cache when the service worker gets installed.

```
var CACHE_NAME = "cache_v1";
var CACHED_URLS = [
  "/index.html",
  "/styles.css",
  "/logo.png"
];

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

Listing 3.4: Saving assets to cache on install

Caching assets can be used for speeding up the application use by keeping some content on the cache that does not change often. Other use case is to offer the user some of the content even if there is no internet connection available at the time.

Next we will show different strategies for caching the assets and when to fetch assets from the cache and when to fetch them from the internet. Typically different strategies are used for different type of assets in an application. All strategies are based on ”Offline Cookbook” from Archibald [42] if not otherwise stated.

3.6.1 Caching strategies

This section presents some of the most common strategies for what to cache and when to cache.

Cache on install

This strategy adds given assets to the cache when the service worker is installed. This strategy should be used for static assets that do not change often. These assets are also ones that are needed for offline access of the application.

Cache on user interaction

This strategy allows the user to choose what assets they want to store for offline access. In this case the assets are being cached dynamically by what the user decides to cache. A good example of this is a news app where the user can choose that they want a certain article to be available for later offline reading.

Cache on network response

In this strategy the assets are being stored to the cache after they have been fetched from the internet. This strategy works well for assets that change often but which should still be available offline.

Stale-while-revalidate strategy

This strategy suits best for cases where it is not so important to show the user the latest version of the asset. The idea is to get the asset from the cache if it exists and then update the asset in the cache after in the background. The new version of the asset will be available for the user the next time this resource is being requested.

Cache on push notification

This strategy opens the cache and stores assets in it in the background when the user receives a push notification. This strategy is mainly used for caching assets that are related to the push notification received. If the notification is about a news article then the cached asset might be the content of the article so the user could open it up instantly if they click the notification. Archibald [42] requests developers to use this strategy with notifications as otherwise if the user is offline, clicking the notification might simply cause the notification to be dismissed without showing any content because as there is no connection, no data can be fetched and shown.

3.6.2 Strategies for serving assets

In this section different strategies are shown for serving assets that the browser requests.

Network only strategy

This is the most basic strategy for serving assets. Assets are being fetched from the internet and an error is thrown if the app is unable to fetch them for any reason.

Cache only -strategy

In this strategy the assets are being fetched from the cache only. Normally instead of using this strategy some other strategy is used such as the following strategy.

Cache, network fallback strategy

In this strategy assets are fetched from the cache but if they are missing then they will be fetched from the network instead. This strategy can also be called ”offline first” strategy.

Cache then network strategy

This strategy consists of two separate requests. First the content is fetched from the cache and displayed to the user if it was found. At the same time another request is sent for fetching the same asset from the network. When the response arrives, the content will get updated and the user sees the fresh version of the content. The cache also gets updated at the same time.

3.7 Background sync

We can imagine a case where someone is writing a comment on a webpage but after hitting the ”send” button the web page tells that there is no network connection available and the action cannot be executed. With native apps this is not a problem because they have the ability to work on the background even if they have been closed by the user.

WhatsApp’s native app is a good example of an app that makes a good use of this feature. While offline and sending a message on WhatsApp the message will appear on the screen as usual but it will have a dim clock symbol on it. The symbol indicates that the message has not yet been sent but will get send after the network connection is established. This functionality can be tested by closing the app and then restoring the network connection. In the background WhatsApp will send the message and it will get delivered to the recipient without the sender having to re-open the app. Progressive web apps can use background sync to achieve the same

functionality.

Background sync is a web API which lets us send updates to the server from the application even when the app is not in active state. This is useful for the situations where the user makes some updates with the application while being offline. The updates are then automatically sent to the server on the background when the connectivity returns. The service worker is required for the background sync to function.

Archibald [42] shows that background sync can also be used for fetching and caching assets on the background even when the application is not open. This means that the application will run faster when the user use it as assets have been fetched on the background before. On the other hand this approach could possibly cause much unnecessary battery usage as network requests and cache updates are done on the background even if the user is not going to use the app for a while.

Ater [7, p. 136] thinks that Background sync is one of the most valuable tools given to the developers lately and that it is one of the key components that make up a modern progressive web app.

Listing 3.5 shows a simple example of registering a background sync task. The execution of this task within a service worker can be seen in Listing 3.6.

```
navigator.serviceWorker.ready.then(function(swRegistration) {
  return swRegistration.sync.register('myFirstSync');
});
```

Listing 3.5: Registering a background sync task

```
self.addEventListener('sync', function(event) {
  if (event.tag == 'myFirstSync') {
    event.waitUntil(doSomeStuff());
  }
});
```

Listing 3.6: Running a scheduled background sync task on service worker

3.8 Push notifications

Push notifications can be used to send urgent messages to the user even when the application is not actively running on their device. This is probably one of the most visible features of PWA and something that only native and hybrid applications were able to do in the past.

Push notifications in web apps are actually implemented using two separate APIs: Push API and Notification API. The first is responsible for receiving messages from the server and the latter is responsible for displaying notifications to the user. [7, p. 184] A service worker is required for push notifications to work on mobile [7, p. 191].

For being able to send push notifications the user must first accept the prompt from the browser that asks for their approval. The browser will only ask the user this once and if the user denies it the permission can no longer be obtained this way. There are many web sites which ask this permission the very moment the user enters the site for the first time. This usually leads for the user to deny the permission as they don't see the value of the notifications they are prompt to receive. A survey conducted in 2015 showed that 52% of mobile users found notifications as an "annoying distraction" [43]. More recent survey conducted in 2017 by the same company shows that users are getting more perceptive towards receiving push notifications [44].

For these reasons it is important to carefully plan when to ask the user for the permission to send push notifications. The best way to do this is to make the user see the value they are getting out from this choice. The permission prompt should only be shown to the user after they have pressed a button that says "Enable push notifications" for instance. A good habit is also to first show a custom dialog that asks for the permission and only if they accept that show the real permission dialog. This way if the user denies the first permission request the app is still able to ask

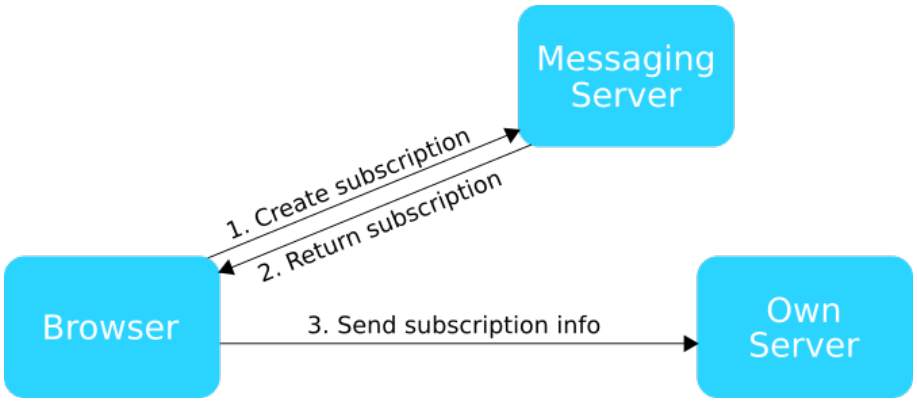

Figure 3.3: Subscribing for receiving push notifications. Adapted from [7]

the user again as they have not denied the actual request.

3.8.1 Subscribing for push notifications

To be able to receive push notifications the browser must subscribe to a special messaging server which will act as a proxy between the application server and the user. This is done to protect the user from unnecessary spam of notifications.

There are three entities related to receiving and sending push notifications: a browser, a server and a messaging server. The browser is responsible for subscribing as a push notification recipient to the messaging server and then sending the information about the subscription to the application server. It is also responsible for receiving and acting on the received push messages. A messaging server is responsible for receiving new subscriptions and forwarding messages from servers to the right recipients. The server is responsible for keeping the record of subscription information related to users and then sending a notification to a specific user at a specific time. The phases of subscribing for push notifications can be seen from Figure 3.3.

3.8.2 Sending push notifications

When the application server needs to send a push notification it uses the subscription object provided by the browser before in the subscription phase. The subscription object contains all the information needed for the application server to send a push notification for the specific client. This information includes for instance the URL of the messaging server.

There are libraries available that makes this process much easier. Without a library, sending the request would require setting various HTTP headers including a JWT (JSON Web Token) authorization header to name one [7, p. 207]. The process of sending a push notification can be seen from figure 3.4.

3.8.3 Security considerations

Push notifications pose a security threat as they can be used for malicious purposes. Lee, Kim, Park, *et al.* [13] shows that push notifications can easily be used for phishing someone's password or to lure someone to the attacker's website. All the attacker has to do is to create a PWA, get the user to visit that PWA, get the user to accept the prompt for the website to send notifications and after that the attacker can freely send any push notification to the user at any time. They can even pretend to be something they are not as the logo of the push notification can

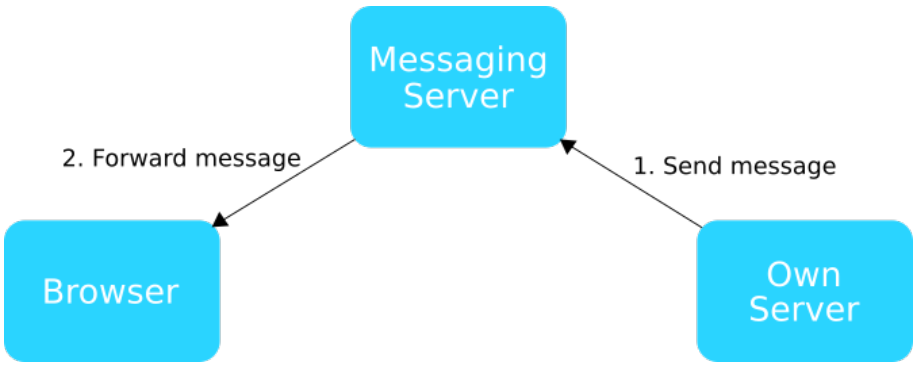

Figure 3.4: Sending push message to a user. Adapted from [7]

be set to anything.

3.8.4 Facebook Messenger push notifications

Push notifications are supported in all other major browsers, but not in Safari. Facebook offers a workaround for that with their Messenger push notifications [5]. They claim that these notifications are more effective compared to Web push notifications by saying that the CTR (click-through rate) for Messenger push notifications is double compared to regular Web push notifications [45]. Downside is that the user has to have Facebook installed on their device to receive Messenger push notifications.

[5]https://www.izooto.com/facebook-messenger-push-notifications-definitive-guide

4 PWA versus native

Results of a study conducted by Fransson [46] on 2017 show that taking a picture with a native app is considerably faster compared to a similar solution using a PWA. On the other hand apps that need geolocation services work actually slightly faster with PWA compared to a native app.

In this chapter, we will cover what features are currently available only on native apps and how PWAs performance compare to the native apps in different features. After that we will take a look at some new APIs which will enable to use features on PWAs that only native applications used to have in the past. Most of these features are available only in experimental versions of browsers so they are not officially supported at the moment (June 2020).

4.1 Benefits of PWA compared to native

A major advantage with PWAs is that the app has to be coded once and it will work with basically any modern browser on any device. With native apps one is forced to create separate apps – one for each platform they want their app to support. Development is also faster and it will cost less money compared to creating a native application. There are much more people out there who are capable of doing web development than those who are able to code native Android or iOS apps. In many cases it would require hiring a separate person to handle the code for a different platform. [8], [47] There are also more code libraries and components available that

can be used to speed up the development process.

PWAs work everywhere and do not depend on any specific market place and they do not require user to download them before use [8]. They are also always up to date if the user is not in offline mode as the app is just an enhanced web page. This also makes the development process faster and little cheaper as the developer does not have to submit the app to any app store. Deploying the app only requires setting up a web server.

PWAs take significantly less storage space on users device compared to similar native apps. One example of this is Twitter. Their progressive web app, Twitter Lite, takes less than 3% of the device storage space compared to their native app [48]. Another advantage is that PWAs do not download everything related to the app when the user first visits the page. Instead they gradually download all the needed assets based on what the user does within the app. PWAs also usually start up faster compared to native apps. As we mentioned in Chapter 1, Tinder saw that the start time of their app was cut to half thanks to the shift from native app to PWA [3].

In top of all this, PWAs are easier to reach compared to native apps. Progressive web apps can be shared as easily as sharing a web page. By clicking the link the new user gets an instant access to the app as no installation is required. As all PWAs are regular web sites they can also be indexed by search engines which makes it possible for anyone to find them easily by conducting a search online.

4.2 App distribution

Native apps are always distributed through app stores while PWAs are accessed using the URL pointing to them. It is currently (June 2020) not possible to deploy a PWA automatically to any other marketplaces than Microsoft Store. If the PWA meets all the required criterion for app installation, the Bing crawlers will automat-

ically import the PWA and make it an installable Windows 10 app to Microsoft Store[1]. One can also wrap their existing PWA in a Trusted web activity on Android Studio to create an app that can be published on Google Play Store. Rybczonek [49] explains the steps required for this on his article "How to Get a Progressive Web App into the Google Play Store".

4.3 Drawbacks compared to native

How well PWAs work depends on the browser the user is using. For example Safari for iOS does not support Push API at the moment of writing this (June 2020). This means that these users are not able to receive push notifications through the PWA. This is one reason why someone would want to make a native app instead of a PWA. It is also not as easy to monetize progressive web apps as it is native apps. Because PWAs are essentially web pages anyone with the URL can access them. With native apps it is possible to set a price that users must pay before they can install the app.

Native apps are also considered to be more safe in terms of data security. A recent study shows some of the reasons behind these claims. They found out that push notifications can easily be used to trick people by pretending to be something else as the sender of the push notification can customize the look of the notification. They also noticed that service workers can be abused by the fact that they can keep running on the background[2]. [13]

[1]https://docs.microsoft.com/en-us/microsoft-edge/progressive-web-apps-edgehtml/microsoft-store

[2]One way to abuse service workers is to keep them running in the background and use them to mine bitcoins or something similar without the user noticing.

Feature	iOS Safari	Chrome for Android
Service Worker	✓	✓
Push API	x	✓
Background Sync API	x	✓
IndexedDB	✓	✓
File API	✓	✓
MediaStream API	✓	✓
Web Bluetooth	x	✓
Geolocation API	✓	✓
Web Share API	✓	✓

Table 4.1: Browser comparison

4.4 Features coming to web

There are still some features not currently available for Progressive web apps. Table 4.1 represents a summary of what features are already supported in iOS Safari and in Chrome for Android. The data shown in the table is based on the data available on the caniuse.com site[3]. The name of the feature is typed to the search field on the web site and then the table is inspected for iOS Safari and Chrome for Android. The results have been updated in June 2020. From the results we can see that iOS Safari is missing support for Web Bluetooth, Push API and Background Sync API. These features already work with Chrome for Android.

Geofencing is a feature that exists in native applications where the device notifies the user if they exit or enter a certain defined georegion. According to the website, whatwebcando.today[4], this feature is not implemented by any of the browser vendors and the proposition is abandoned. The same functionality can be achieved by

[3]https://caniuse.com/

[4]https://whatwebcando.today/geofencing.html

periodically querying the Geolocation API.

4.4.1 Google Chrome origin trials

Origin trials give developers a chance to experiment with new features coming to Google Chrome which have not yet been officially released. By opting in for an origin trial the developer will get an experimental feature to work on their origin for any user accessing it with Chrome. The features available as an origin trial can be tested but can not be used in production yet.[5]

4.4.2 Native File System API

Accessing the file system of the user's device is currently not possible with PWAs but will be possible when the Native File System API is launched. Currently it is available as an origin trial with Chrome version 78 or later.

The main function of the API is `window.chooseFileSystemEntries()` which opens up a file picker dialog. The function only works when used within a secure origin and by reacting to a action performed by a user such as clicking a button. Listing 4.1 shows an example usage where a file picker is opened when a user clicks a button. The contents of the file is then read and outputted to a textarea.

```
let fileHandle;
butOpenFile.addEventListener('click', async (e) => {
  fileHandle = await window.chooseFileSystemEntries();
  const file = await fileHandle.getFile();
  const contents = await file.text();
  textArea.value = contents;
});
```

Listing 4.1: Reading a file with the Native File System API

[5]More information about Google Chrome origin trials can be found here: http://googlechrome.github.io/OriginTrials/developer-guide.html

With the API it is also possible to save files. Another feature is possibility to prompt the user to select a folder which content can then be enumerated in the application. As this API is quite powerful it has been designed to be as transparent for the user as possible. In Google Chrome an icon is shown on the navigation bar to indicate that the user has given permission for the application to access the file system. [50]

4.4.3 Contact Picker API

The Contact Picker API is a new API that enables accessing the contact list of the user's device. This API is not widely supported yet but was launched with Google Chrome version 80 [11]. It is not supported on any other browser yet so it is only available on Android [51].

With Contact Picker API the app is not able to access these contacts directly. The user will be shown a dialog instead where they can choose exactly which contacts they want to share with the app. Typical use-cases for this API are email applications and all other apps related to messaging to name some examples. Also social networking apps may use user's contact list to see which of these contacts are already using the app or to send out invites to them.

We created a simple test app for using the Contact Picker API. The code can be seen from Appendix B. The code was tested on Samsung S7 phone running Android 8.0 with Google Chrome version 80.0.3987.147. Picking the contacts worked as expected.

4.4.4 Badging for App Icons

One feature that is currently (June 2020) not supported in PWAs is badging. In native applications it is possible to add a small badge to the app icon that can for example notify the user of how many unread messages they currently have. This

Figure 4.1: Badging on WhatsApp

behaviour is illustrated in Figure 4.1.

At the time of writing the Badging API is available in Chrome 79 and later in a form of origin trial [52]. With the API it is possible to either clear the badging from an app icon or set it to any given number. How the API works is shown on Listing 4.2.

```
1 // Set the badge
2 const unreadCount = 24;
3 navigator.setExperimentalAppBadge(unreadCount);
4
5 // Clear the badge
6 navigator.clearExperimentalAppBadge();
```

Listing 4.2: Using the Badging API

5 Implementations and testing

In this chapter we introduce a practical experiment where we test how fetching a geolocation differes between a PWA and a native application. We will also discuss what improvements were made to HRMobi when we turned it from a regular web app to a progressive one.

5.1 Performance testing with geolocation

To find out if there are any significant performance differences we decided to experiment how PWA compares with native application. Two separate similar looking applications were developed to test the differences between PWA and native application. The implementations for these applications can be found from github[1]. All of the tests were conducted by using Samsung S7 device.

We chose to do performance testing with geolocation as this feature exists in HRMobi. Our test for geolocation is very simple. We fetch the geolocation of the device using the API available and we measure how long it takes between the request and getting the location data. With PWA we are using the Geolocation API[2] and with native Android app we are using the LocationManager[3].

[1]PWA: https://github.com/Ziigur/pwa-android-comparison

Android: https://github.com/Ziigur/PWACompareApp

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

[3]https://developer.android.com/reference/android/location/LocationManager

At first our tests showed that fetching geolocation with our web app would take only 80 milliseconds in average. Later we found out that this was possibly because of the Geolocation API using the last known location instead of fetching a new location. We arranged a new test where the location services of the device was toggled off and then back on between each test runs. We made 30 test runs with the web app and with the native Android app. By leaving out ten fastest and ten slowest times and calculating the average from rest of the times we got these results: With the web app, fetching geolocation takes 1 130 milliseconds and with the native app it takes 770 milliseconds.

Timings varied a lot especially with the native Android app. With the web app the timings were more predictable as the longest delay was 2 950 millisecons as with the native Android app it was 27 700 milliseconds. All benchmark results can be found from Appendix C.

It is hard to say anything conclusive of the results. The results also contradict with the test conducted by Fransson [46] as their results showed that fetching geolocation with PWA is significantly faster compared to native Android app.

5.2 HRMobi

HRMobi is a web application where employees are able to see the schedule, when their shifts start, where they should be working and so on. Employees are also able to change the working times on completed shifts and write a note for the foreman if their shift was longer or shorter than it was planned to be.

The app also contains a bulletin board where foremen can leave messages for employees. Bulletins can even require an acknowledgement from employees so that the foreman can be sure that all employees have received and seen the message. There are also features targeted for employees whose working hours vary much and who work at many different places which is common in cleaning industry for

instance. The app contains a system where employees click a ”Work begins” button when they start working and then ”Work ends” button when they are done. This way the real time spent on the work is recorded and used as a base when checking who has worked for how many hours.

In this section we will cover what changes were made to HRMobi and see how it compares with how it functioned before the changes. There were some challenges especially with offline functionality which we discuss in more detail later in this section. Adding push notifications on the other hand was quite straightforward as there are some libraries available that simplify the process.

5.2.1 Challenges

The current design of HRMobi presents some challenges for converting the application to a progressive web app. The deployment structure of the app is not very suitable for PWAs because the app is divided between multiple origins. There are also challenges caused by the lack of PWA support on different platforms that should be supported.

Multiple origins

The architectural structure of the application is clustered in a way that there are currently three different production builds at three different servers. Each of these servers has a backend application that serves the client side files from a folder within that server.

When a new user starts to use HRMobi there are two steps. First the user navigates to https://www.hrsuunti.fi/hrmobi/. This page prompts the user to pick the company that he/she works on. When the company is selected the user is redirected to one of the three separate servers depending on the company they chose. This behaviour is illustrated on Figure 5.1. The problem here is that the

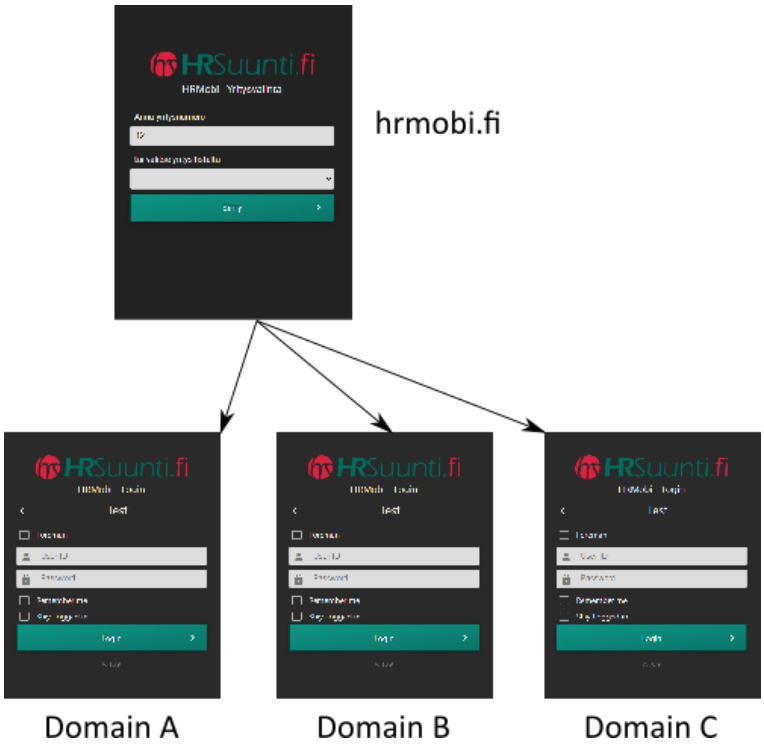

Figure 5.1: Multiple origins in HRMobi

PWA installation and usage must take place within a single domain. This means that the installation / ”add to home screen” has to take place on the second domain and not on the one where the company selection is presented. In practice this means that the user should be well informed about this behaviour. The first page should not actively prompt the user to install the app but instead this should be done within the second domain where the real application lives. This also means that there should be different web app manifests for different production builds as the absolute URL must be supplied within each of them. Renzulli [53] discuss this issue in his article ”Progressive Web Apps in multi-origin sites”.

iOS Safari support

Supporting PWA features on Safari poses some challenges as some of the features do not work as well. One of the main issues is that it is very hard to share session data between installed PWA and the browser version of the app. On Safari when

PWA is installed all of the session, cookie and local storage data is lost that was stored with the browser version of the app. [54] With HRMobi it is important that the cookie data would be shared as the ID of the organization is stored in cookies. Losing the data causes at least the frustration for the user having to type it in again.

Caputa [54] suggests a solution to this problem. The solution is based on the fact that cache content is shared between the browser and the standalone version of the app. By creating a fake endpoint with service worker we can share data between instances. By using POST commmand we can save the body data to the cache and with GET command this same data can be retrieved.

Offline access

Another challenge is related to the offline usage of the application. Currently the application redirects to the login page of the app every time the page is refreshed. When logging in many company related settings along with other information such as employee lists are downloaded from the server. If the application was to work offline then the application should let the user somehow login without checking the credentials against the server. The best way would be to change the whole login process in a way that the application would store all required information locally and this information is then showed to the user if they are offline.

5.2.2 Service worker

The first thing to do when upgrading an existing web app to PWA is to make the app register a service worker. The basic use of a service worker is to make it cache some files during the installation making the app offline capable. This will also make the app load faster by downloading some files from the service worker cache instead of fetching them from the real server. The cache management is a bit tricky as when updating the app it might not be enough to refresh the page to get the updates.

It might require closing all tabs where the app is currently open and then opening it up again. This is caused by the service worker's life cycle which we discussed in Chapter 3.

When we implemented a service worker for HRMobi we ran into problems with the cache management but finally found a solution suggested by Fabulich [55]. The solution we chose was to force the service worker skip waiting when it has registered a new version of itself and after that refresh all open tabs so that the new service worker will get updated on all tabs. The issue with this approach is that if there is anything in progress in other tabs the forceful refresh might cause the user to lose their progress. In our case the situation where the app is open in multiple tabs at the same time is not very common and there are not many places where the user could be in the middle of any progress that long.

We conducted some measurements to test how much the load time was affected by adding a service worker. We used the network tab of Google Chrome's Developer console to see how long it takes for the page to load. First we added the development build of HRMobi to home screen to ensure that the service worker was working. Then we refreshed the app 30 times and wrote down the load time. Then we opened the current production build of the app on browser, refreshed the page 30 times and wrote down the load time between each refresh. We used Chrome's option to simulate "Fast 3G" connection on both cases so that we could see how long it takes for the app to load if the connection speed is little slower.

Our results show a significant difference in load times. PWA version of the app takes only **830 milliseconds** to load in average as the non-PWA version of the app takes **3 500 milliseconds**. In other words the PWA version is four times faster compared to the non-PWA version of the app. All of the timings can be seen from Appendix D.

5.2.3 Offline access

By loading assets from the cache we were able to add offline support to HRMobi. The offline access for HRMobi was succesfully implemented by using both IndexedDB and LocalStorage. Arrays of objects such as the list of shifts and the list of employees were saved to IndexedDB. Other data such as login response were saved to LocalStorage. Normally one would use cache storage for saving HTTP responses but the LocalStorage was chosen in this situation because the login request may be called with varying query parameters. Using to LocalStorage it is easier to define in what format and where the response is saved.

5.2.4 Push notifications

One of the main benefits of upgrading HRMobi to a progressive web app is that it allows us to add push notifications to the application. We came up with a list of situations where push notifications could be used in HRMobi. Here is the list of these scenarios.

- An employee will receive a notification when a foreman creates a new bulletin board message which is marked as ”urgent” and the employee is one of the receivers for that message.
- An employee will receive a notification when a foreman publishes new shifts that can be seen by the employee.
- An employee will receive a notification when a foreman chooses to send a push notification for certain employees to notify them about an extra shift that can be taken.
- A foreman will receive a notification when an employee catches one of the extra shifts available.

We decided to implement a separate Node.js server to handle user subscription and sending push notifications to users. This approach was chosen because of the ease of developing a push server with Node.js. There is an existing npm package[4] which contains functions for generating VAPID keys and sending push notifications. VAPID stands for Voluntary Application Server Identification and is a method used for making sure that only single application server is able to send push notifications [56].

Our Node.js implementation has two API endpoints. One endpoint is for subscribing a new user and the other is for sending push notifications for given users. We chose to use SQLite as our database format for storing the data of subscribed users.

5.2.5 Background sync

We decided that we will not add background sync support for HRMobi at this point. Some events require to take place in real time and therefore by enabling background sync we found that the app might just become needlessly complicated for developers and end users. Background sync might easily give the user a feeling that something works and something they sent went through while in reality the thing might not have been sent to the server because of network issues. Other reason for not implementing this is that it is not yet supported on iOS Safari.

5.2.6 Performance improvements

Progressive web apps should perform as well as possible. For this reason we made other changes to HRMobi to make it function better. Google offers a tool called Lighthouse[5] which can be used to measure different qualitative aspects of a web

[4]https://www.npmjs.com/package/web-push

[5]https://developers.google.com/web/tools/lighthouse

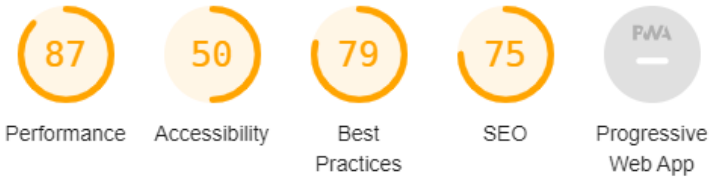

Figure 5.2: Current performance with Lighthouse tool

page. This tool contains five metrics it uses to evaluate the page. These metrics are performance, accessibility, best practices, SEO and progressive web app. We used the tool to analyze the current production version of HRMobi. The results obtained from the Lighthouse tool are shown in Figure 5.2. The tool also tells the developer which things there are that need to be fixed. We then used these quidelines to make the improvements on HRMobi.

Performance score was improved from 87 to 97 by making most of the JavaScript and CSS load asynchronously and therefore not block the initial page render. We moved critical CSS code to the `<head>` element of the page so that some styling is applied to the page when it loads. To make other CSS code load asynchronously we used a trick[6] where we added the attributes shown in Listing 5.1 to the `<link>` elements fetching the external CSS files.

```
media="print" onload="this.media='all'"
```

Listing 5.1: Trick for fetching CSS asynchronously

Accessibility score was improved from 50 to 100 by adding alt attributes to images so that the text will be rendered if the image file is missing or fails to load. We also added aria-label attributes to inputs so that screen readers will work with them.

Best practices score was improved from 79 to 93 by switching from older HTTP/1.1 to newer HTTP/2. The Lighthouse tool suggests that we should remove external

[6]https://www.filamentgroup.com/lab/load-css-simpler/

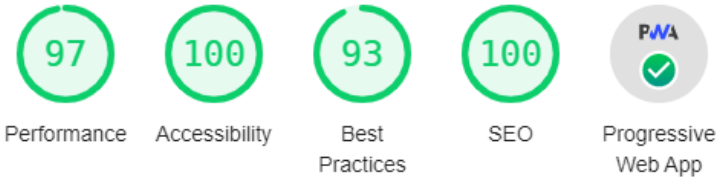

Figure 5.3: Performance with Lighthouse tool after changes

libraries such as jQuery and jQuery Mobile as they have some known vulnerabilities. There is a newer version of jQuery with no known vulnerabilities but the newest version of the jQuery Mobile is from 2017 and it still contains some known vulnerabilities. After experiementing with these we found out that these versions of jQuery do not function properly together. We decided that we will later attempt to gradually replace all jQuery code with native JavaScript.

Finally SEO score was improved from 75 to 100. Adding alt attributes for images also affected this score and on top of that we added a missing meta description for the page.

After our changes the Lighthouse tool also classifies our page as a PWA because the page registers a service worker, loads offline and serves a manifest that meets the installation criteria[7]. The improved results of the Lighthouse audit can be seen in Figure 5.3.

5.2.7 Future work

Progressive web apps can easily be deployed on Google Play store by creating a Trusted Web Activity. This is not achievable before we have solved the problem with multiple origins and versions of the app. One possible way to avoid the multi-origin issue would be to create the shell of the app in one domain and then make it fetch different version of files from different places depending on the organization

[7]https://web.dev/installable-manifest/

chosen. We experimented with this a bit already and were able to create a working setup where all client side files were served from one server and all fetch requests were made to other server.

The startup time on first load is another things that could be improved further. This can be achieved by splitting the JavaScript to multiple files. The HTML document should also be splitted as every page of the app is currently inside a single large HTML file.

6 Discussion and conclusion

In this chapter, we will discuss the implications of our results, give the conclusion of this thesis and then present some topics which could be researched further.

6.1 Discussion

It is not clear what it takes for a web page to be classified as a web app and if a web page that registers a service worker and has a valid web app manifest should be classified as a progressive web app or not. Should there be a term ”progressive web page”? The term ”progressive web page” or ”progressive web site” is used in some articles on the Internet [47]. In one article they even point out that the term ”progressive web app” is a successor of the term ”progressive web page” [57].

As PWAs are replacing native apps we are moving in a direction where browsers begin to act as operating systems themselves hosting other apps [57]. This feels counterintuitive as it can be thought of just adding additional layer of complexity. On the other hand this change brings web pages on the same level with native applications. Browsers also enable cross-platform functionality and other handy features such as possibility to share the link for a certain app.

Progressive web apps are still missing some of the features that exist on native but PWAs are quickly catching up. Contact Picker API was added to the mobile version of Chrome in early 2020 [11]. Access to the file system through the File System API is currently available on Chrome as an origin trial and will probably be

released in the near future.

6.2 Conclusion

As we conducted a systematic literature review we found out that there has not been much research on the subject of progressive web apps yet. For this reason most of our information presented on this thesis is based on online sources.

To answer RQ 1 we have presented different definitions found for Progressive web apps. For some reason the definition of progressive web app is not clear for many. Definitions range from strict technical definitions to ones describing the non-technical aspects like the user experience. Based on what we have learned from literature and through our own experiments we suggest that the technical definition of Progressive web app should be the following.

A progressive web app is a website which registers a service worker and serves a valid web app manifest.

This is basically the same definition as what Keith [30] presented on 2017. The only difference is that our definition does not contain the need for HTTPS. Because registering a service worker always requires the site to be served over HTTPS it is not needed in the definition of PWA.

Progressive web apps seem to compete against native apps pretty well at the time of writing this thesis (June 2020). Only major browser that does not quite yet support all the main features of PWAs is Safari on iOS. Push notifications are not supported by Safari yet which might still lead into having to build a native app for iOS even though other platforms could work with PWAs as complete replacements for native apps.

To answer RQ 2.1 we did some research where we compared the features available for progressive web apps with features available for native applications. These

findings are presented on Chapter 4. Currently some of the most important missing features from Chrome browser are ability to access the file system and adding a badge to the app icon. Ability to view user's contacts is missing from iOS Safari. This feature was just recently added to Chrome [11].

Safari does not support Push API either but there is an interesting workaround for this using Facebook Messenger Push Notifications. This require Facebook messenger application to be installed on the users device to start with.

Some other features which Safari does not support at the time of writing (June 2020) include Background Sync, Bluetooth, Speech Recognition, Touch ID, Face ID and accessing battery information. [28]

There are possibly even more of these missing features as there are certainly some less often used native features that have not been taken into consideration in native – PWA comparisons. Also the feature support for PWA varies with different browsers.

In Chapter 4 we found out answer for RQ 2.2 by studying whether some of the features on native are still more performant compared to the implementations with progressive web apps. According to the study conducted by Fransson [46], taking a picture is still something where native apps perform better than PWAs.

In Chapter 5 we did some performance testing with fetching Geolocation with native Android app and with a PWA implementation. Based on our results there is no significant difference between the two.

In the same chapter, we showed the steps that it took for us to transform an existing web app HRMobi to work as a progressive web app. We did not have time to implement all beneficial aspects of PWA for HRMobi within the scope of this thesis. Neither did any of the implementations yet get to the production build so we can say nothing about how these changes will affect actual users. Our own tests with the development build shows that the changes to the app make the app start up

faster and make it possible to use some of the features even if offline. We were also able to implement push notifications which was one of the most requested feature to be implemented in HRMobi.

6.3 Suggestions for further work

As this topic evolves in a fast pace and the browser support changes it might be interesting after year or two to research this same topic and find out if the support is better and if there are still any reasons to build native applications. A specific topic that could be researched is whether the Facebook Messenger notifications can be used to replace regular push notifications or not. The overall support for PWA in iOS Safari is also one subject that could be researched further as our tests and implementations were mainly run on Chrome for Android.

The effects of upcoming 5G network could be one topic of interest as they say that 5G might make fetching files from network even faster than fetching files from local storage. This might negate some benefits that PWA delivers as cached files becomes less important when network is available. On the other hand faster connections could boost up the use of web based applications even more.

A web component framework called Accelerated mobile pages (shortened AMP) is something that can be used alongside with PWA to enhance its capabilities even further. The framework was announced by Google in 2015. The idea is to create light weight and fast web pages that operate fluently on mobile and on other devices that have less processing power. Combining these techniques can have a positive impact as AMP and PWA have also been referred as ”the mobile’s new power couple”. [58]. This topic should be researched further to find out what results can be achieved.

WebAssembly can be used to translate higher level languages such as C, C++ or Rust to work on the web. WebAssembly enables programs to run faster compared to JavaScript. One could research whether WebAssembly can be used to enable

web apps to perform as well as native apps also with applications that require more computing power.

References

[1] A. Holst, *Number of smartphone users worldwide from 2016 to 2021*, Available at: https://www.statista.com/statistics/330695/number-of-smartphone-users-worldwide/, 2019.

[2] A. Russell, *Progressive web apps: Escaping tabs without losing our soul*, Available at: https://infrequently.org/2015/06/progressive-apps-escaping-tabs-without-losing-our-soul/, 2015.

[3] A. Osmani, *A tinder progressive web app performance case study*, Available at: https://medium.com/@addyosmani/a-tinder-progressive-web-app-performance-case-study-78919d98ece0, 2017.

[4] C. Love, *Why progressive web apps should be your business choice over a native app*, Available at: https://love2dev.com/pwa/progressive-web-app-vs-native/, 2019.

[5] A. Khan, A. Al-Badi, and M. Al-Kindi, *Progressive web application assessment using AHP*, Available at: https://www.sciencedirect.com/science/article/pii/S187705091930955X, 2019.

[6] C. Moll, *Apps are faltering. but progressive web apps seem pretty legit.* Available at: https://neatly.io/an-advanced-progressive-web-apps-and-amp-seo-guide-2019/, 2016.

[7] T. Ater, *Building Progressive Web Apps*. O'Reilly, 2017.

[8] Z. Anastasia, *Should you choose a PWA or native app for your ecommerce business?* Available at: https://rubygarage.org/blog/pwa-vs-native-app, 2019.

[9] smArtapps, *Native application VS progressive web app: Which one should you choose?* Available at: https://medium.com/inside-smartapps/native-application-vs-progressive-web-app-which-one-should-you-choose-5eeaaf6ee92d, 2018.

[10] Google, *Unlocking new capabilities for the web (Google I/O '19)*, Available at: https://www.youtube.com/watch?v=GSiUzuB-PoI, 2019.

[11] P. LePage, *A contact picker for the web*, Available at: https://web.dev/contact-picker/, 2020.

[12] T. Kerssens, *Applicability of progressive web apps in mobile development*, Available at: https://staff.fnwi.uva.nl/a.s.z.belloum/MSctheses/MScthesis_Tjarco.pdf, 2019.

[13] J. Lee, H. Kim, J. Park, I. Shin, and S. Son, "Pride and prejudice in progressive web apps: Abusing native app-like features in web applications," English, in *Proceedings of the ACM Conference on Computer and Communications Security*, Cited By :1, 2018, pp. 1731–1746.

[14] W. S. El-Kassas, B. A. Abdullah, A. H. Yousef, and A. M. Wahba, "Taxonomy of cross-platform mobile applications development approaches," *Ain Shams Engineering Journal*, vol. 8, no. 2, pp. 163–190, 2017, ISSN: 2090-4479. DOI: https://doi.org/10.1016/j.asej.2015.08.004. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S2090447915001276.

[15] L. Corral, A. Janes, and T. Remencius, *Potential advantages and disadvantages of multiplatform development frameworks – a vision on mobile environments*,

Available at: https://www.sciencedirect.com/science/article/pii/S1877050912005303, 2012.

[16] Statista, *Number of apps available in leading app stores as of 2nd quarter 2019*, Available at: https://www.statista.com/statistics/276623/number-of-apps-available-in-leading-app-stores/, 2019.

[17] S. Hyrynsalmi, T. Mäkilä, A. Järvi, A. Suominen, M. Seppänen, and T. Knuutila, "App store, marketplace, play! an analysis of multi-homing in mobile software ecosystems," vol. 879, Jan. 2012, pp. 59–72. DOI: 10.13140/RG.2.1.2540.5524.

[18] A. Osmani, *The app shell model*, Available at: https://developers.google.com/web/fundamentals/architecture/app-shell, 2019.

[19] C. Love, *What is an app shell and do you need one?* Available at: https://love2dev.com/blog/app-shell/, 2019.

[20] E. Bidelman, *A beginner's guide to using the application cache*, Available at: https://www.html5rocks.com/en/tutorials/appcache/beginner/, 2010.

[21] J. Archibald, *Application cache is a douchebag*, Available at: https://alistapart.com/article/application-cache-is-a-douchebag/, 2012.

[22] M. Gaunt, *Service workers: An introduction*, Available at: https://developers.google.com/web/fundamentals/primers/service-workers/, 2019.

[23] Mozilla, *Remove support for appcache*, Available at: https://bugzilla.mozilla.org/show_bug.cgi?id=1237782, 2019.

[24] ScandiPWA, *History of progressive web apps*, Available at: https://medium.com/progressivewebapps/history-of-progressive-web-apps-4c912533a531, 2019.

[25] A. Romo, *Seriously, though. what is a progressive web app?* Available at: https://medium.com/@amberleyjohanna/seriously-though-what-is-a-progressive-web-app-56130600a093, 2018.

[26] S. Kapoor, *Progressive web apps 101: The what, why and how*, Available at: https://www.freecodecamp.org/news/progressive-web-apps-101-the-what-why-and-how-4aa5e9065ac2/, 2018.

[27] J. Grigsby, *Progressive web apps*. A Book Apart, 2018.

[28] M. Firtman, *Progressive web apps on iOS are here*, Available at: https://medium.com/@firt/progressive-web-apps-on-ios-are-here-d00430dee3a7, 2018.

[29] Google, *Progressive web apps*, Available at: https://developers.google.com/web/progressive-web-apps/, 2019.

[30] J. Keith, *What is a progressive web app?* Available at: https://adactio.com/journal/13098, 2017.

[31] P. LePage, *Provide a custom install experience*, Available at: https://web.dev/customize-install/, 2020.

[32] E. Navara, *Progressive web apps on windows*, Available at: https://docs.microsoft.com/en-us/microsoft-edge/progressive-web-apps, 2018.

[33] P. LePage, *Add to home screen*, Available at: https://developers.google.com/web/fundamentals/app-install-banners.

[34] Mozilla, *Add to home screen*, Available at: https://developer.mozilla.org/en-US/docs/Web/Progressive_web_apps/Add_to_home_screen, 2019.

[35] Samsung, *Ambient badging*, Available at: https://hub.samsunginter.net/docs/ambient-badging/.

[36] M. Cohen, *Web storage overview*, Available at: https://developers.google.com/web/fundamentals/instant-and-offline/web-storage, 2019.

[37] P. Walton, *Best practices for using IndexedDB*, Available at: https://developers.google.com/web/fundamentals/instant-and-offline/web-storage/indexeddb-best-practices, 2019.

[38] Mozilla, *Web storage api*, Available at: https://developer.mozilla.org/en-US/docs/Web/API/Web_Storage_API, 2020.

[39] P. LePage, *Storage for the web*, Available at: https://web.dev/storage-for-the-web/, 2020.

[40] J. Wilander, *Full third-party cookie blocking and more*, Available at: https://webkit.org/blog/10218/full-third-party-cookie-blocking-and-more/, 2020.

[41] Google, *Caching files with service worker*, Available at: https://developers.google.com/web/ilt/pwa/caching-files-with-service-worker, 2019.

[42] J. Archibald, *The offline cookbook*, Available at: https://developers.google.com/web/fundamentals/instant-and-offline/offline-cookbook, 2019.

[43] C. O'Connell, *The inside view: How consumers really feel about push notifications*, Available at: http://info.localytics.com/blog/the-inside-view-how-consumers-really-feel-about-push-notifications, 2016.

[44] R. Gibb, *How consumers perceive push notifications in 2018*, Available at: http://info.localytics.com/blog/push-notification-survey-2018, 2018.

[45] iZooto, *Getting started with facebook messenger push notifications*, Available at: https://www.izooto.com/facebook-messenger-push-notifications-definitive-guide, 2020.

[46] R. Fransson, *Comparing progressive web applications with native android applications: An evaluation of performance when it comes to response time*, 2017.

[47] Hipster, *What is a progressive web app? – progressive web app (PWA)*, Available at: `https://hipster-inc.com/what-is-a-progressive-web-app-pwa/`, 2019.

[48] N. Gallagher, *How we built twitter lite*, Available at: `https://blog.twitter.com/engineering/en_us/topics/open-source/2017/how-we-built-twitter-lite.html`, 2017.

[49] M. Rybczonek, *How to get a progressive web app into the google play store*, Available at: `https://css-tricks.com/how-to-get-a-progressive-web-app-into-the-google-play-store/`, 2019.

[50] P. LePage, *The native file system API: Simplifying access to local files*, Available at: `https://web.dev/native-file-system/`, 2019.

[51] A. Bar, *Contacts*, Available at: `https://whatwebcando.today/contacts.html`, accessed: 2020-04-22., 2020.

[52] P. LePage, *Badging for app icons*, Available at: `https://web.dev/badging-api/`, 2018.

[53] D. Renzulli, *Progressive web apps in multi-origin sites*, Available at: `https://web.dev/multi-origin-pwas/`, 2019.

[54] M. Caputa, *How to share cookie or state between progressive web application in standalone mode and safari on iOS*, Available at: `https://www.netguru.com/codestories/how-to-share-session-cookie-or-state-between-pwa-in-standalone-mode-and-safari-on-ios`, 2018.

[55] D. Fabulich, *How to fix the refresh button when using service workers*, Available at: `https://redfin.engineering/how-to-fix-the-refresh-button-when-using-service-workers-a8e27af6df68`, 2017.

[56] M. Thomson and P. Beverloo, *Voluntary application server identification (VAPID) for web push*, Available at: `https://tools.ietf.org/id/draft-ietf-webpush-vapid-03.html`, 2017.

[57] Beyond Java, *Progressive web apps*, Available at: `https://www.beyondjava.net/progressive-web-apps`, 2016.

[58] A. Habeeb, *An advanced! progressive web apps and AMP SEO guide 2019*, Available at: `https://neatly.io/an-advanced-progressive-web-apps-and-amp-seo-guide-2019/`, 2019.

[59] S. Priyadarsini, J. M. Varghese, A. Mahesh, and T. S. George, "Shopping spree: A location based shopping application," English, *International Journal of Engineering and Advanced Technology*, vol. 8, no. 6, pp. 1451–1455, 2019.

[60] G. De Andrade Cardieri and L. A. M. Zaina, "PWA-EU: Extending PWA approach for promoting customization based on user preferences," English, *Proceedings of the ACM on Human-Computer Interaction*, vol. 3, no. EICS, 2019.

[61] F. Fonseca, H. Peixoto, J. Braga, J. Machado, and A. Abelha, "Smart mobile computing in pregnancy care," English, in *Proceedings of 34th International Conference on Computers and Their Applications, CATA 2019*, Cited By :1, 2019, pp. 219–224.

[62] U. Paul, M. Nekrasov, and E. Belding, "EmerGence: A delay tolerant web application for disaster relief," English, in *HotMobile 2019 - Proceedings of the 20th International Workshop on Mobile Computing Systems and Applications*, 2019, p. 167.

[63] C.-T. Yeh, T.-M. Chen, and M.-C. Chen, "A progressive web app for evaluating occupation indicator," English, *Journal of Internet Technology*, vol. 20, no. 7, pp. 2217–2223, 2019.

[64] D. Terán, F. Tapia, J. Rivera, and H. Aules, "Use of e-health as a mobility and accessibility strategy within health centers in ecuador with the aim of reducing absenteeism to medical consultations," English, in *CEUR Workshop Proceedings*, vol. 2486, 2019, pp. 319–333.

[65] G. de Andrade Cardieri and L. M. Zaina, "Analyzing user experience in mobile web, native and progressive web applications: A user and HCI specialist perspectives," English, in *ACM International Conference Proceeding Series*, Cited By :3, 2018.

[66] D. Fortunato and J. Bernardino, "Progressive web apps: An alternative to the native mobile apps," Portuguese, in *Iberian Conference on Information Systems and Technologies, CISTI*, Cited By :1, vol. 2018-June, 2018, pp. 1–6.

[67] H. Wijaya and R. A. Abbas, "Animation effectiveness for e-learning with progressive web APP approach: A narrative review," English, *International Journal of Engineering and Technology(UAE)*, vol. 7, no. 4, pp. 112–120, 2018.

[68] K. Hu and J. Zhu, *A progressive web application on ancient roman empire coins and relevant historical figures with graph database*, English, ser. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics). 2018, vol. 11197 LNCS, pp. 235–241.

[69] P. Loreto, J. Braga, H. Peixoto, J. Machado, and A. Abelha, "Step towards progressive web development in obstetrics," English, in *Procedia Computer Science*, vol. 141, 2018, pp. 525–530.

[70] L. E. Nugroho, A. G. H. Pratama, I. W. Mustika, and R. Ferdiana, "Development of monitoring system for smart farming using progressive web app," English, in *2017 9th International Conference on Information Technology and*

Electrical Engineering, ICITEE 2017, Cited By :2, vol. 2018-January, 2017, pp. 1–5.

[71] A. Biørn-Hansen, T. Majchrzak, and T.-M. Grønli, "Progressive web apps: The possible web-native unifier for mobile development," English, in *WEBIST 2017 - Proceedings of the 13th International Conference on Web Information Systems and Technologies*, Cited By :13, 2017, pp. 344–351.

Appendix A Scopus search results

Paper	Year	Type	Topic of the study
[59]	2019	case study	Shopping Spree: A Location Based Shopping Application
[60]	2019	theoretical	PWA-EU: Extending PWA Approach for Promoting Customization based on User Preferences
[61]	2019	case study / theoretical	Smart Mobile Computing in Pregnancy Care
[62]	2019	case study	EmerGence: A Delay Tolerant Web Application for Disaster Relief
[63]	2019	case study	A progressive web app for evaluating occupation indicator
[64]	2019	case study	Use of e-Health as a Mobility and Accessibility Strategy within Health Centers in Ecuador with the Aim of Reducing Absenteeism to Medical Consultations
[65]	2018	theoretical	Analyzing User Experience in Mobile Web, Native and Progressive Web Applications: A User and HCI Specialist Perspectives
[13]	2018	theoretical	Pride and Prejudice in Progressive Web Apps: Abusing Native App-like Features in Web Applications
[66]	2018	theoretical	Progressive Web Apps: an alternative to the native mobile Apps
[67]	2018	case study	Animation Effectiveness for E-Learning with Progressive Web App Approach: a Narrative Review
[68]	2018	case study / theoretical	A Progressive Web Application on Ancient Roman Empire Coins and Relevant Historical Figures with Graph Database
[69]	2018	case study / theoretical	Step Towards Progressive Web Development in Obstetrics
[70]	2017	case study	Development of Monitoring System for Smart Farming Using Progressive Web App
[71]	2017	theoretical	Progressive Web Apps: The Possible Web-native Unifier for Mobile Development

Table A.1: Search results

Appendix B Contact Picker API

```
<!DOCTYPE html>
<html>
  <head>
    <meta name="viewport" content="width=device-width, initial-scale
   =1.0">
  </head>
  <body>
    <script>
      async function openContacts() {

        if (!navigator.contacts) {
          alert("not supported!")
          return;
        }
        const props = ['name', 'email', 'tel'];
        const opts = {multiple: true};

        try {
          const contacts = await navigator.contacts.select(props, opts)
   ;
          alert(contacts);
        } catch (e) {
          alert(e);
        }
      }
```

```
24      </script>
25      <button onclick="openContacts()">Contacts</button>
26    </body>
27  </html>
```

Listing B.1: Testing Contact Picker API

Appendix C Benchmark timing results

PWA Geolocation Samsung S7		**Android Geolocation Samsung S7**	
Runs	**Milliseconds**	**Runs**	**Milliseconds**
1	2950	1	7239
2	2470	2	10344
3	2531	3	10468
4	2421	4	3127
5	865	5	27719
6	712	6	5544
7	776	7	3170
8	395	8	3605
9	936	9	966
10	229	10	210
11	1044	11	243
12	246	12	70
13	2470	13	95
14	355	14	524
15	840	15	2383
16	2485	16	842
17	1052	17	809
18	2566	18	403
19	831	19	356
20	2601	20	238
21	2578	21	531
22	291	22	1004
23	349	23	242
24	2571	24	517
25	1072	25	848
26	494	26	990
27	592	27	462
28	2466	28	760
29	507	29	578
30	1461	30	881

Table C.1: Geolocation timings with PWA and native Android application

Appendix D Benchmark load time results on HRMobi

non-PWA version of HRMobi		**PWA version of HRMobi**	
Runs	**Milliseconds**	**Runs**	**Milliseconds**
1	3320	1	842
2	3730	2	867
3	3320	3	827
4	3670	4	838
5	3320	5	860
6	3730	6	826
7	3760	7	824
8	3320	8	844
9	3320	9	827
10	3750	10	825
11	3790	11	841
12	3730	12	832
13	3720	13	825
14	3330	14	825
15	3330	15	801
16	3330	16	841
17	3670	17	850
18	3790	18	825
19	3660	19	816
20	3670	20	822
21	3330	21	826
22	3350	22	810
23	3740	23	825
24	3710	24	820
25	3640	25	832
26	3350	26	820
27	3650	27	831
28	3320	28	830
29	3320	29	836
30	3310	30	948

Table D.1: HRMobi app load time