DEGREE PROJECT IN COMPUTER SCIENCE AND ENGINEERING, SECOND CYCLE, 30 CREDITS

STOCKHOLM, SWEDEN 2018

Native-like Performance and User Experience with Progressive Web Apps

VIKTOR YBERG

KTH ROYAL INSTITUTE OF TECHNOLOGY
SCHOOL OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

Native-like Performance and User Experience with Progressive Web Apps

VIKTOR YBERG

Master in Computer Science
Date: 25 August 2018
Supervisor: Christopher Peters
Examiner: Cristian M Bogdan
School of Electrical Engineering and Computer Science

Abstract

Users spend more time than ever on mobile devices like smartphones and tablets, while native app development continues to become harder due to platform fragmentation. The web is a promising platform for mobile applications because of its easy access and standardised technologies that work unanimously across many different platforms and operating systems. However, native applications have always had an edge over the web because of important features that have not been available anywhere else, such as push notifications, background synchronisation and offline support. Progressive Web Apps aim to bring the web platform closer to native by enabling many of these important features while still running completely in the web browser, with the possibility to install the application, effectively promoting it to a top-level application.

This project will evaluate the capabilities of web-based mobile applications compared to traditional native mobile applications. Three simple proof of concept applications will be built to test the performance and user experience with the help of different JavaScript libraries and techniques for building a Progressive Web App. Then one of the implementations will be further developed and matched against an existing native application with similar features in terms of functionality and performance.

The study finds that for this use case, a Progressive Web App may be used instead of a native app without missing out on any important functionality. This simplifies development and releases, by enabling rich code sharing between the different platforms as well as avoiding the app distribution platforms by distributing the application entirely through the web, automatic and transparent to the users. However, this solution means more responsibility in terms of infrastructure for developers to maintain and optimise as the application needs to be distributed by own servers.

Sammanfattning

Allt mer tid ägnas åt mobila enheter såsom smartphones och surfplattor, medan apputveckling blir allt svårare på grund av spridningen av plattformar. Webben är en lovande plattform för mobila applikationer på grund av dess lättillgänglighet och standardiserade teknologier som fungerar likadant på många olika plattformar och operativsystem. Trots detta har nativa appar alltid haft ett övertag gentemot webben på grund av funktioner som inte varit tillgängliga på andra platformar, såsom pushnotiser, bakgrundssynkronisering och offlinestöd. Progressive Web Apps syftar till föra webbplattformen närmare nativ genom att möjliggöra många av dessa funktioner men fortfarande köras enbart i webbläsaren, med möjlighet att installera applikationen på enheten.

Projektet kommer att utvärdera kapaciteten i webbaserade mobila applikationer jämfört med traditionella mobilapplikationer. Tre stycken proof of concept-applikationer kommer att byggas för att testa prestanda och användarvänlighet med hjälp av olika JavaScript-bibliotek och tekniker för att bygga en Progressive Web App. Därefter kommer en av implementationerna att vidareutvecklas och utvärderas gentemot en existerade app med liknande funktionalitet.

Studien visar att en Progressive Web App i det här användningsfallet kan ersätta en nativ mobilapplikation utan att gå miste om viktig funktionalitet. Det skulle förenkla utveckling och publicering, genom att möjliggöra koddelning mellan olika plattformar och undvika appdistribueringsplattformarna genom att distribuera applikationen enbart genom webben, automatiskt och transparent för användarna. Lösningen innebär dock mer ansvar i form av underhåll och optimering av infrastruktur eftersom applikationen måste distribueras genom egna servrar.

Contents

Acknowledgements

First of all, I would like to thank Marcus Falck, Mats Bäcklund and the others at MeetApp for giving me the opportunity to pursue my thesis at their company.

I would also like to thank my supervisor, Prof. Christopher Peters, and my examiner, Cristian Bogdan, for their feedback and help with improving this thesis.

Chapter 1

Introduction

Mobile platform fragmentation is one of the main technical challenges for developers when building mobile apps. The fragmentation exists both across platforms, with each platform being different in terms of programming languages, APIs and development tools, and within the platforms, with various devices running different versions of the operating system with different hardware setups [12]. Over the last couple of years, the web has evolved from being a simple document-centric platform to a first-class application platform, making it a solid alternative to native environments by enabling cross-platform development. Progressive Web Apps (PWAs) are playing a big part in making it so, by providing web-based user experiences closer to the native feel that users are used to, effectively closing the gap between the two fields.

This chapter will give an introduction to what Progressive Web Apps are and how they are changing the mobile app ecosystem. It will also give an introduction to the study that will be conducted in this thesis.

1.1 Background

Native applications are hard and expensive to develop. They require special software development kits (SDKs) provided by the platform provider in order to build applications and to access native APIs for things like hardware and are therefore limited to the platform they are developed for. Companies looking at targeting a larger audience therefore need to build their application for several different mobile platforms such as Android, iOS and Windows Phone. On top of this, if

they want their users to access all of their functionality on desktop they also need a native desktop application or a website. Most of these areas use different programming languages and require different expertise, and multiple development teams are often needed. Web technologies on the other hand can be used to build applications for mobile devices like smartphones and tablets as well as desktop computers, sharing most or all of the codebase between the platforms. This means that companies only need one development team to build an application that supports multiple platforms.

Previous studies show that the biggest shortcoming of web-based applications is access to native platform APIs [31]. This limitation has held developers back from building mobile applications on the web that can compete with traditional native applications. But recent progress in web capabilities has enabled developers to do more, and we are currently seeing web stacks starting to run at low level, taking over some of the work previously performed by native languages [5]. This includes backend servers but also mobile applications.

PWAs are web applications that combine the best from the web and traditional native mobile apps by providing features that bring the two closer together, such as push notifications, background services and offline capabilities. They are compatible with all devices that have a web browser since they are accessed just like normal websites, which make them easier for users to access and use. According to Google[1], which coined the term "Progressive Web App", they are user experiences that are reliable, fast and engaging. Reliable for working even with uncertain network connections, fast for quickly responding to user interaction and engaging for behaving like a native app on the device. The apps should work for every user, but with progressive enhancements for browsers that support the extra features [16].

PWAs have several advantages over traditional native mobile applications in that they are easier and cheaper to develop and distribute, and easier for users to access since they run inside a regular web browser. The purpose of PWAs is to provide users with well-performing applications packed with features previously only seen in native applications.

This thesis is carried out at MeetApp AB, a company that builds a complete solution for making events and conferences more dynamic and inspiring by engaging event participants through a native mobile

[1]https://developers.google.com/web/progressive-web-apps/

application. Even though all apps are built on mostly the same codebase, they personalise and package their app with custom design and functionality for each company and platform, which means that they need to test and deploy each app separately. As their customer base grows, they need a simpler and more manageable way of deploying application updates and creation of new apps for their customers. The idea behind this project is to provide a web-based cross-platform solution instead of their current native solution to both decrease their maintained codebase as well as simplifying the release and update process for themselves and for their users.

The application that will be built in this project is an informative hub for participants at the events. It is important that the application is easily accessible and that it works similarly to the company's current native mobile application. Furthermore it is important to show that it is possible to achieve the same things on the web platform as you can on the native platforms, should they want to integrate more features in the future.

Some key points that MeetApp wants to achieve with the project are:

- They want the application to run in the browser. Event visitors should have as easy access to the application as possible, without having to install it in order to use it.
- The application should feel like a native application similar to the app that they use now, with support for offline usage.
- Push notifications should be supported. This is an important feature of their current app, providing visitors with live information about events.
- The application should be platform independent for simpler development and testing.

These points make MeetApp a good fit for this project, as PWAs support and are built for the type of features that they need in their application.

The study itself seeks to answer some questions about the viability of web-based mobile applications over traditional native applications. The main question that will be researched in this study is:

How can Progressive Web Apps be built to match functionality, performance and user experience of traditional native mobile applications?

To answer this question, it will be divided into a few smaller questions:

1. Are technologies found in native development available and performant enough to build mobile applications on the web platform?
2. What is missing before web-based solutions can serve as valid replacements for native applications in the mobile field?
3. How well supported are these technologies on different platforms and operating systems?
4. What are the benefits of moving from native to web-based development?

The PWA built in this project will not explore all recent web features that may be used to get the most out of the web platform, but will use the most important ones for the project and bring up a discussion about some other techniques and features.

1.2 Objective

The goal of the project is to see whether MeetApp could benefit from providing their customers with a PWA instead of a native mobile application. A web-based solution would increase their platform support to not only Android, iOS and Windows Phone but also desktop or any other platform with a web browser. It would also likely simplify their development process by only having to maintain one codebase rather than separate ones for each platforms, which would make testing and implementing new features faster and more manageable. Lastly, it would make the release and update processes easier by not having to rely on the mobile app distribution platforms and instead serve the application entirely through the web.

The purpose of the study is to see if the recent technologies on the web platform are available and good enough to compete with the native platforms.

1.3 Related Work

Previous work within the area of writing mobile apps with web code has been largely focused around two different techniques for accessing native features. The first one is to package HTML, JavaScript and CSS code in a native container that gives access to native features that web code cannot access itself, often referred to as *hybrid* development. A popular framework that does this is Ionic[2], which essentially provides a wide range of UI components styled for both Android, iOS and Windows Phone and access to native features through Apache Cordova[3], a bridge between the JavaScript code and the native implementation [4]. All UI components are web-based and displayed inside a webview, a container for displaying web pages inside a native application. This technology enables true cross-platform development as there is no need to use the platforms' native UI components, however, because of the additional layer of using a container for the web-based UI components, the experience is often slowed down with decreased performance [22]. There is also the problem of browser widgets generally not being as well optimised as real web browsers and the experience sometimes becomes noticeably laggy for the user [31].

The other technique is to translate web-based UI building blocks into native components. This is done by mapping JavaScript components that the framework provides to native UI components that render just like if the app would have been written with any of the native languages such as Java or Swift. This way, the entire app can be written in JavaScript and yet natively support multiple platforms. A popular framework that uses this technique is React Native[4], which lets developers build the user interface in JSX — a syntax extension to JavaScript inspired by the HTML markup style. The goal of the framework is not to build a complete cross-platform compatible app from the same codebase, but to reuse the business logic and build custom user interfaces for each platform.

A common thing with these techniques is that they need to be packaged into native applications in order to work. They are both interesting solutions to cross-platform mobile development but neither of them provide easier access to the application than any other native ap-

[2] `https://ionicframework.com/framework`
[3] `https://cordova.apache.org/`
[4] `https://facebook.github.io/react-native/`

plication. Easier access would be achieved by running the application in a browser, so that users can avoid installing the application before using it. Unfortunately, if we want to write mobile apps that run inside the browser, we are limited to the features that the browser provides. However, new technologies and web standards have enabled developers to depart from native containers and let the web browser handle the native APIs and communication with the hardware. This means that instead of using e.g. Cordova for native integration, the web browser could handle that directly.

1.4 Challenges

Users spend more and more time on mobile devices like smartphones and tablets and the time spent on the devices tend to be on a very small set of installed apps. In fact, the average smartphone user spends 50% of their time in one single app and almost 80% in their 3 most used apps [20]. It becomes apparent that it is hard to break into the small group of apps that users use daily or even to get users to try out new apps. The difficulty is to make it easier for users to test new applications and at the same time provide enough functionality and trust for users to re-engage with them. Although MeetApp's solution serves a specific purpose and does not necessarily have the issue of event participants not using their app, it would simplify the process for new users to try out and use it. The problem itself is a more general problem that could be solved by providing users with web-based solutions. That way they do not have to install an app just to try it out.

Analytic data indicates that 53% of visits are abandoned if a mobile site takes longer than 3 seconds to load[5]. However, getting a full-fledged mobile application to load from scratch in under 3 seconds on a mobile connection can be a challenge, compared to just loading a static web page. This makes it important to think not only about the measured performance, but also about the perceived performance — the performance that the user experiences when using the application. Making something appear on the screen within a short amount of time is important to make the users stay on the site.

Lastly, there is the problem of getting users to understand that

[5]Google Data, Global, n=3,700 aggregated, anonymized Google Analytics data from a sample of mWeb sites opted into sharing benchmark data, March 2016.

PWAs are in fact full-fledged applications in the device's system and should be treated as such. It is a relatively new set of features that not all users are aware of yet. Smartphone and tablet users have previously had ways to add bookmarks to the homescreen, which essentially just creates a direct link to the site to be opened in the web browser. PWAs do more than that by "installing" the application, which downloads all resources required to use the application offline, as well as creating a storage model for the application to persistently store data.

1.5 Outline

In chapter 2, a more detailed introduction to PWAs is presented. It will discuss both the foundation of PWAs as well as possible techniques to increase the performance. The third chapter will present the methods that has been used when building the application and performing the study. It will also present and evaluate the proof of concept applications that were built to find the best techniques to use when proceeding to build the final application. Chapter 4 will focus on the implementation of the PWA as well as an evaluation using results found from performance tests. Lastly, chapter 5 will present the conclusions from the study.

Chapter 2

Literature Study

At the very core, a Progressive Web App is just a website running in a web browser. What distinguishes it from other websites is its ability to progressively enhance the experience by adding more features based on what the device supports [14]. The concept has evolved from previously being able to add websites to the homescreen of the device, to now support powerful features such as push notifications and offline usage. This chapter will provide a deeper introduction to Progressive Web Apps.

2.1 Foundation

The foundation of a PWA is its three cornerstones: the service worker, the app shell and the manifest. These three parts together differs a PWA from a regular web application by enabling features that aren't available in regular websites or web apps.

2.1.1 Service Worker

The service worker is the part of a PWA that enables many of the newer features on the web platform, such as push notifications, offline capabilities and advanced network request handling. When the application sends an external network request, the service worker acts as a proxy server between the application and the network, intercepting the request and taking actions based on the state of the network and the application. The request handling is done asynchronously in the background, hidden to the rest of the application [11]. Because of it

being able to intercept requests, they do not have to be rewritten to fit the service worker, making it a powerful drop-in feature for any web application.

The service worker also handles caching of both documents and images for offline use, separate from the web browser's built-in caching mechanism. This is important since a browser's standard cache is a temporary storage which stores files in the same manner as the service worker but cannot be controlled by the developer. In order to achieve a consistent offline experience however, the caching strategy needs to be fully controlled, which makes standard caching too weak. Since every network request goes through the service worker, resources that match some pre-defined condition can be stored in the local cache to be retrieved instantly independent of the network connectivity at a later time.

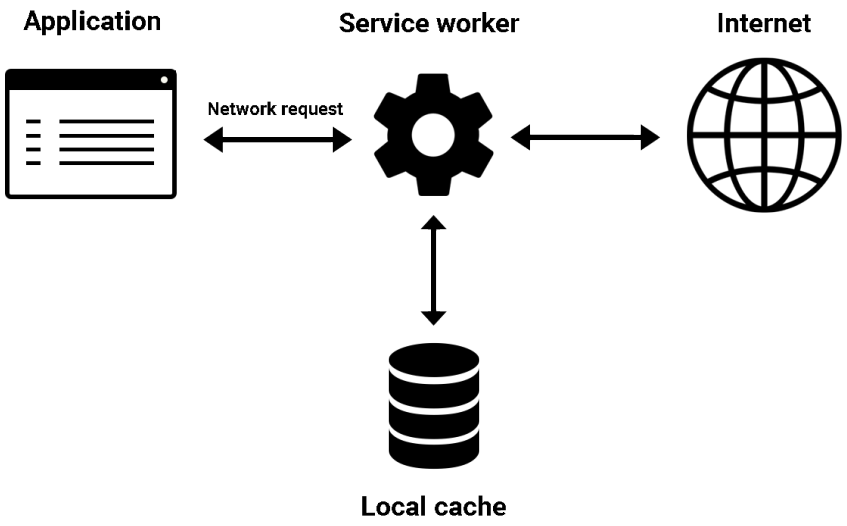

Figure 2.1: The service worker intercepts all network requests and decides whether to fetch from the external resource or from the local cache.

Other than the things mentioned above, service workers can also be used for things such as:

- fetching data from the network in the background,
- pre-fetching resources that the user is likely to request in the near

future,

- calculating geolocation or gyroscope data for app-wide usage.

This technology brings web applications closer toward native app viability by enabling features never before seen on the web platform [26].

2.1.2 App Shell

The app shell is the minimal HTML, CSS and JavaScript required to power the user interface [27]. This approach has become popular in JavaScript-heavy web applications. By loading the app shell before the rest of the content, the time it takes until the first meaningful paint on the user's screen can be reduced drastically. The content can then be loaded dynamically by the shell.

The app shell approach is similar to how native apps work, and by caching the app shell using the service worker, the user interface and navigation can be loaded without making any network requests and functions instantly on subsequent page loads, even in offline mode.

2.1.3 Manifest

The web app manifest is what tells the browser that the website is a PWA. It provides information such as the name of the app, a description, icons, etc. that is used by the device once the app has been "installed", or added to the home screen, to make it look and behave like a native app in the device's user interface.

2.2 User Experience

User experience is an important element in every application. A successful mobile web application needs to have quick responsive navigation that feels like a native application and the user should have an overall pleasing experience using the software. A good user experience can be achieved with familiar user interfaces, smooth gestures and a good looking design, but it also extends outside the app, e.g. with push notifications to notify the user under certain conditions. Push notification support was enabled with the introduction of service workers and has not been possible in web-based applications

before [24]. This feature adds a lot to the overall user experience and puts the web one step closer to the native experience.

User experience can be divided into two primary categories; the *context* — elements that cannot be controlled by the developer, such as platform capabilities, UI conventions and the runtime environment, and the *implementation* — elements that can be controlled, such as performance, design and integration with platform features [5]. This project will focus on the implementation part.

2.3 Functionality

Functionality goes hand in hand with the user experience. For the app to feel convenient to use, it needs to have familiar navigation and gestures. Although the app lives in the web browser, after a PWA has been installed in the device, it runs without the browser interface (the address bar, back button, etc.) and has to provide necessary navigational elements itself, which might be different on different platforms. For example, Android devices often have a physical back button whereas iOS devices instead rely on virtual buttons [5]. This needs to be considered when building a standalone web-based application in order to provide a familiar intuitive user experience.

2.4 Performance & Metrics

Modern web apps have higher standards than the regular web when it comes to performance and responsiveness. Web apps must animate and respond quickly to not give the user the impression that they are getting a degraded experience compared to a native app. To make an app feel fast, there are a couple of things to think about: [11]

1. Respond to user actions within 100 milliseconds, or the user will notice lag.

2. Render the view at 60 frames per second.

3. Make the experience feel smooth even on low-end devices with poor network connection.

4. Make sure the app launches quickly. If the app's content is cached and the browser is still in the device's memory, a web app can launch quicker than a native app.

2.5 Installing a Progressive Web App

On Android, when a PWA gets added to the home screen, the browser downloads the application content necessary to run the app and saves it to storage [19]. The app then gets promoted to a top-level mobile app, meaning that it behaves just like a native app does in the system — it appears in the app drawer, it exists in the tab switcher and it gets its own storage model [21]. This means that the user can see information about the app in the system's settings, such as storage information and network usage and perform actions such as force stopping or uninstalling it [30].

2.6 Web vs. Native

When it comes to performance, functionality and user experience, native has always had an edge over the web. Native code has access to hardware and sensors via SDKs and APIs that are not available from a webpage within a web browser. The native platforms often have great abstractions and user interface controls for performing common tasks, but the problem is that they all differ in approach and implementation, and if one were to rewrite an application for another platform, they would have to write it from scratch [5, 4]. The web can be considered very consistent in this sense, which can be a big advantage when building a mobile application. The World Wide Web Consortium (W3C) develops web standards which the major browser vendors most often follow when developing their web browsers. This makes the web a solid platform to build applications on.

2.6.1 Native

Traditional native mobile apps are built with platform-specific tools and languages separately for each platform, i.e. Android apps are written in Java or Kotlin via the Android SDK and iOS apps are written in Objective-C or Swift via the XCode tool [21]. The native SDKs

give apps unhindered access to the entire hardware platform, providing rich user experiences and an optimised high performance [5].

A few advantages with native apps:

Optimisation

- *Render pixels directly on screen* — One of the biggest advantages native has in terms of rendering performance is that pixels are drawn directly on the screen, with no additional layers in between. This makes for a more fluid experience with no visible view rendering sometimes seen when loading heavy web pages.
- *Optimised animations* — Both Android[1] and iOS support hardware acceleration as a default for their most basic view elements, which helps speeding up the rendering by utilising the GPU.

Engagement

- *Longer lifetime* — Native mobile apps have a longer lifetime than web apps. Once installed, they have a permanent place on the user's home screen with push notification capabilities to remind the user that it exists, as opposed to a web app which cannot directly interact with the user after they have left the page [2].

Recognisability

- *Recognisable UI* — By using the native SDKs and tools that the platform provides, developers can easily build user interfaces that follow the platforms design guidelines and look familiar to users.

2.6.2 Web

Unlike native apps, mobile web apps do not have direct access to the hardware platform. While native apps run at native speed, mobile web apps run on top of additional layers, which increases the amount of computing resources required to perform tasks, which in turn may decrease the app's speed and performance. Browser vendors are actively working at closing the gap even further by supporting hardware

acceleration to speed up animations [34]. New standards are also being developed by W3C, which includes support for technologies such as Web components [7] and Web animations [8].

A few advantages with web over native are:

Development

- *Cross-platform* — Since a web application runs in the web browser, it works on all platforms by nature. Just having to write an application once and have it work for all platforms, including desktop, is very powerful and saves both time and money for the developer.
- Low-level APIs for hardware access are actively being developed, effectively bridging the gap between web and native [5].

Accessibility

- *Access from any device* — Any device that has a web browser can access a web application without installing anything, since they are accessed like normal web pages. This means that developers can completely omit the app stores when releasing or updating their apps and still be sure that all users are using the latest version.

2.6.3 Progressive Web Apps

PWAs solve some major concerns with mobile web apps that developers have by bringing the experience closer to native without removing the benefits of the web. One issue with the web is the lack of ability to get the users to stay engaged with the site. This is solved by introducing native support for push notifications in the browser, effectively extending the experience outside the browser [5]. This allows developers to re-engage with their users through notifications at the operating system level, which greatly improves the chance of users revisiting the site.

PWAs also make it easier to introduce new users to an app, by simply providing a link to a website, and it is easier for users to quickly try out new apps. If the user likes the app, they can add it to their home screen. One concern with this approach is that the apps do not

show up on the app stores, thus not gaining any exposure to users who might not otherwise find the app's website. On the other hand, users tend to find apps by first finding the website, and if the website is a web app, then they are already at their destination [11]. It has also been shown that if an app is not in the top 0.1% of all apps in the app store, there is no significant benefit of being there [11].

PWA user experience

In a study done by W. Jobe, the difference between native apps and mobile web apps is examined from a user's perspective [18]. The study explores the viability of replacing native applications with mobile web applications as they are cheaper and faster to build and can target a larger audience. Two different mobile web apps were developed, one for tracking runs with GPS and one for scheduling and booking runs together with other users. The first app was concluded not to be able to replace a native app, as it required access to native hardware which was not yet fully supported. The other app, which only displayed content and did not require access to any additional device hardware, was more suitable to replace a similar native app and the subjects who tested the app were satisfied with the experience.

Jobe's study was performed before PWA technologies were invented, but it is still interesting as it proves that mobile web applications built completely with web technologies can be compared to native applications in terms of user experience. Since this study, the web platform's API portfolio has grown. The geolocation API, for instance, might have been improved since then as a draft containing improvements to the specification [33] was released after the study was conducted. Other hardware API's have also been added and improved, which could mean that the application may have been built successfully with the technologies available today.

Energy efficiency

Another concern with non-native applications that are coming closer to native functionality is energy efficiency. Energy is one of the most scarce resources in a smartphone, and most users cannot afford to have apps that drain their battery more than usual. This concern is examined in a study by Malavolta, I. et al., which in particular looks at the service worker aspect [23]. The study tests multiple PWAs with

different phones and different network conditions to see if the server worker has any impact on the energy consumption of the device. The hypothesis is that less energy would be consumed since some network requests are stopped by the service worker which is instead fetching from local cache. The issue is that the service worker still has to run in a separate thread the background, which could impact the energy consumption negatively.

The study finds that service workers have no significant impact on energy consumption, regardless of phone quality and network conditions, and that the technology is promising in this sense.

2.6.4 Developer & User Tradeoff

For developers, web-based applications with mobile support means less code to maintain. Historically, companies need multiple development teams in order to support every single platform, including the web and all the major mobile operating systems. By sharing the codebase between platforms, development resources can be decreased considerably.

For users, mobile web applications lead to a more uniform experience. No installs through the app stores are required and users can choose to add the app to the home screen if they want to. Mobile web apps can however be difficult to get as responsive and fluid as a native app, which might make the users feel like they are getting a degraded experience.

2.6.5 Security

There are two security aspects worth considering when comparing native mobile applications to web applications, or PWAs in particular. On one hand, there is the possibility of the application containing malicious software designed to exploit some security flaw of the system or to gain information about the user. The app distribution platforms such as Apple's App Store and Google's Play Store regularly scan apps for malicious code, while the web does not have any authority to do so. This makes the app stores a safer place for users to install apps from.

On the other hand, PWAs require the use of the HTTPS protocol over HTTP for hosting the application itself and for any external re-

quests made from the application. This secures the connection between the client and any remote server — something that native apps do not enforce. Thus, users can be sure that their data is not being stolen if they use the application while connected to a public network.

Both approaches have their advantages and disadvantages security-wise.

2.7 Boosting Web Performance

Increasing the performance and providing a better user experience of a PWA can be done both in the app itself and in the back-end system that powers it. Some techniques to get better performance can be efficient caching, lazy loading, server-side rendering and code splitting to decrease bundle sizes. Optimisations like these are necessary to get a native-like perception in the app.

2.7.1 The PRPL Pattern

The Polymer team at Google has presented a collection of techniques and guidelines to make PWAs perform better; the PRPL pattern [29]. The pattern helps improving things such as the time it takes before the app becomes interactive, maximising caching efficiency and simplifying development and deployment. The PRPL acronym stands for: [32]

- **P: Push** critical resources for the initial route with HTTP/2 Server Push. This means that the server can push resources that it knows the app will need later on the initial page load, thus eliminating the need for multiple requests.
- **R: Render** the initial route fast. The first route should just be the minimal app shell in order to make it as lightweight as possible to decrease rendering time.
- **P: Pre-cache** remaining routes. By doing this, the app can cache other pages before they are requested. This can greatly speed up the application on devices with poor network connection, since a requested route may already be in the cache.
- **L: Lazy load** everything that is not needed at the initial page load. For example, a modal that should appear when the user

clicks a button does not need to be loaded before it is asked for. This also makes smaller components (or chunks of code) more easily reusable, as they can be lazy loaded from the cache on other routes as well.

This pattern is not bound to any specific framework or library, but is more of a set of techniques to get a web application to perform better.

2.7.2 Code splitting

A common practice to reduce the number of network requests is to combine all resources needed by the application into one single bundle that is downloaded and parsed when the page loads. However, this technique can reduce the web browser's cache efficiency since all pages get their own bundle, making shared resources non-shareable between different pages. With HTTP/2, all resources do not have to be bundled into one large file, since required resources can be pushed from the server so that they exist in the app's cache before they are explicitly requested. Instead they can be served separately which means that they can be cached efficiently while still gaining the network benefits of bundling [29].

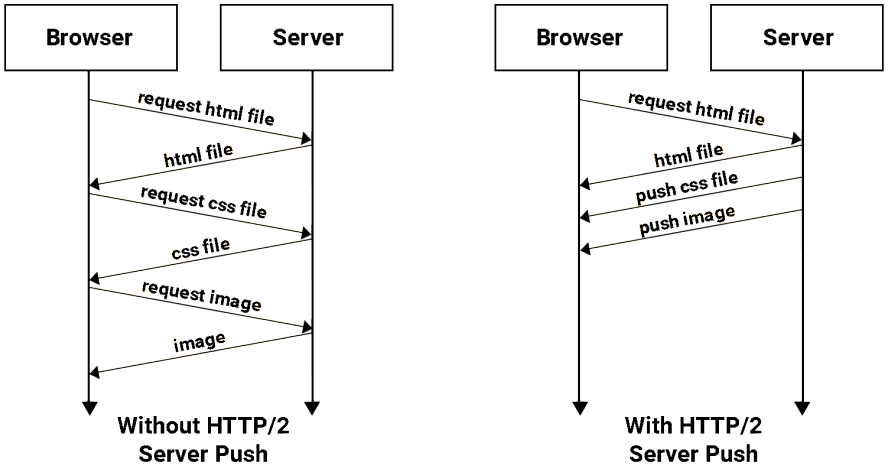

Figure 2.2: With HTTP/2 Server Push, additional resources can be pushed to the client in advance.

2.7.3 Server-side rendering

Modern single-page web applications that rely heavily on JavaScript run on the client's machine as opposed to traditional static web pages that are served directly from a server with minimal client responsibility. The problem with this is that large applications can take long before they display anything on the screen due to the possibly large amount of resources that need to be downloaded, parsed and compiled by the client. Serving a static page is faster than sending a bunch of code and letting the client produce a page out of it.

Server-side rendering is a technique for speeding up the initial page rendering by sending a pre-rendered HTML page along with the JavaScript to the client. The page is then *hydrated*, meaning that the JavaScript is executed to make the page interactive to the user.

Another technique for achieving faster page loads is *Progressive Bootstrapping* [28]. The difference here is that the server does not just pre-render the HTML page — it also adds necessary JavaScript to make it minimally functional directly when it loads. Additional functionality can then be added once more resources have been downloaded and executed. This technique solves the issue of possibly leaving the user in a state where they cannot interact with the page before it has been hydrated.

2.8 Evaluation Method

The performance of a PWA can be measured in multiple different ways. Lighthouse [10] is an automated tool that evaluates the quality of a PWA by looking at best practices and key performance aspects. Best practices includes use of HTTPS and HTTP/2 for own resources, non-deprecated technologies as well as other web-related general best practices to deliver a good user experience. The performance audits look mainly at the load times and interactivity of the application, e.g. time to first meaningful paint, time to first interactive, time to interactive and how well assets are optimised. These aspects have direct impact on the perceived performance of the application as they determine when the user can see the primary content of the application and when they can interact with it, and are considered important metrics in this project.

In addition to the automated tests performed by Lighthouse, man-

ual tests will measure the load times of the application under different circumstances along with rendering performance and how much the PWA feels like a native application. The performance tests should be compared to other similar web approaches and, if possible, similar native technologies used in the existing native application.

Since the performance and user experience are two of the main focus points of this thesis, the results of the tests described above will be the basis of the evaluation.

2.9 Frameworks/Libraries

In order to build a well-performing PWA within the limited timespan of the project, some framework or library is necessary. This section will look at the abilities of some popular JavaScript libraries for building performant web-based application. The libraries that will be examined in this study needs to follow some criteria:

- It needs to be supported by a large company to make sure that it will be managed for some time forward.
- Apps built using the library need to have low startup time. It is preferable if the library's file size is minimal to reduce the size of the dependencies that need to be downloaded on first launch.
- It needs to support an app-like experience with layout and design that follows UI guidelines.

HNPWA's[1] list of PWAs will be used for comparing some different frameworks and technologies for building performant apps. The list consists of unofficial Progressive Web App following a loose specification to build an app to display news from the Hacker News website[2]. It is important to note that the implementations can differ based on server infrastructure and other things not directly related to the front-end library, but they can provide some guidance on which approaches may be most beneficial. Many of the apps are built by people involved in the development of the respective framework or library.

The libraries presented here are the ones that had the shortest time to first interaction among the listed ones.

[1] `https://hnpwa.com/`
[2] `https://news.ycombinator.com/`

2.9.1 React

React[3] is a JavaScript UI library managed and supported by Facebook which has 1.18 billion daily users [13] and is one of the most popular open-source projects on Github[4].

The startup time is the highest out of the four, and the library's file size is the largest. Since it is just a UI library, it does not have all the needed tools to build a complete app with routing and complex data management. Therefore, additional libraries are needed in order to use React in this project.

It does not have any pre-built app-like components like toolbars, drawers and buttons, and instead relies on third party libraries to achieve this.

2.9.2 Preact

Preact[5] is based on React but with a significantly smaller file size. The library is not managed by a large company, but several large corporations such as Uber [9], The New York Times and Pepsi [32] are using it in production. The smaller file size is appealing, but the necessary dependencies for achieving app-like behaviour are still needed.

Compared to React the startup time for the Preact app is reduced but like with React, additional design libraries are required to achieve app-like components.

2.9.3 Polymer

Polymer[6] is a JavaScript library managed and supported by Google, designed for building modern web apps with web components. The library "uses the platform", meaning that it makes the library itself lighter by utilizing the strength of the modern web platform, taking full advantage of web components, service workers and HTTP/2 [15]. By reducing the size of the library, the startup time is decreased.

The library comes with a toolbox filled with app-like components

[3] https://reactjs.org/

[4] https://github.com/search?q=stars:\%3E1\&s=stars\&type=Repositories

[5] https://preactjs.com/

[6] https://www.polymer-project.org/

that follow Google's Material Design guidelines[7], that can be used to build PWAs.

2.9.4 Stencil

Stencil[8] is a compiler for building web components, managed and supported by the Ionic team. The library provides a framework-like experience when developing the web components, but the code compiles to standard web components that run natively in the browser and is interoperable with any other web project, regardless of framework or library.

Since web components do not require any library at runtime in modern web browsers, the additional file size when loading the app is zero. The library can be used alone to build a web application, but it is often used in conjunction with another framework or library. All the previously mentioned libraries are component-oriented and can be used together with Stencil.

[7] https://material.io/guidelines/
[8] https://stenciljs.com/

Chapter 3

Method

The main focus of the study was finding a good approach to building a Progressive Web App. As a way of testing different technologies for building it, a proof of concept application was developed using three different JavaScript libraries that were deemed suitable for the project, picked from the set of libraries loosely examined in the literature study. This chapter will describe the process of implementing the proof of concept applications, as well as picking the right technologies to use in the final implementation of the PWA. The most suited implementation will then be continuously built upon to meet MeetApp's requirements for the application.

The research process for the entire project is as follows:

1. Identifying a proof of concept use case.
2. Determining a few libraries/approaches to build the proof of concept PWA with.
3. Evaluating the PWA based on MeetApp's requirements and pick the best implementation.
4. Continue building on that PWA together with back-end optimisations to provide a seamless user experience.
5. Test and evaluate the performance against an existing native application.

3.1 Proof of Concept

To determine the most suitable technologies to use later in the project and to test the viability of the features needed to fulfil the requirements of the application, three proof of concept applications were built. They all had the same features and were built in similar ways, but with different JavaScript libraries.

The first step was identifying a proof of concept use case. The proof of concept should be a very basic application with only a few especially important features as well as an example of how the navigation and interaction would work in the final app. After identifying some key features from the company's existing native app together with Marcus Falck, CTO of MeetApp, it was agreed upon that the most important features to test were aspects that their native application does well, and that the web has not traditionally been as good at. These features are:

1. Push notifications for user re-engagement.
2. Fast rendering and good scrolling performance.
3. Offline capabilities.
4. Efficient storage and caching for decreased network usage.

While the proof of concept is just a basic implementation of the functionality and design that the final product should have, it should still determine how well these features can be achieved. The application was developed using three different JavaScript libraries; Polymer, React and Preact. These libraries were picked because they are popular among developers for building single page applications and they all have built in support for building PWAs.

Stencil was loosely examined in the literature study but was left out of the project. Since Stencil is just a web component compiler with no extra features beyond the ones defined in the web components specification, another library or framework would likely be needed to build the infrastructure with routing, data binding, etc. Therefore, it was determined that it would not have any positive impact performance-wise for this application.

To test the features mentioned above, the proof of concept included:

- An app shell with a navigation drawer.
- A programme page with a list of programme items.
- A detail page with programme item information and images.
- Offline storage of necessary data for the app to function offline.
- A user interface that follows the Material Design guidelines.

The applications were then uploaded to a web server and tested on both a desktop computer and on a few different smartphones.

3.1.1 Performance

The performance of the different implementations of the application were measured against each other in terms of load times and download size. React, Angular and other popular JavaScript frameworks and libraries are often built on top of the web platform and have their own component abstraction. This has previously been the way to go since we have not been able to extend the DOM with native custom elements until recently. The problem with this approach is that before the page can load, the entire framework or library has to download and compile, making the experience slow for users with slower network connections. This is what distinguishes Polymer from the libraries used in the other proof of concept applications, since Polymer does not have its own component abstraction. Instead, it utilises the standard web components API, which is supported by all modern browsers. This means that there is no need to wait for external libraries before the JavaScript engine knows how to render the components on the screen, since the browser has that functionality built in. This also shows in the performance test, where Polymer had both lower load times and a smaller application size than the other libraries.

All measurements were performed on a desktop machine under the same conditions. The reason for measuring on a desktop machine rather than a real mobile device that the application most of the times would run on, is that the absolute values are not as important as the relative differences. When comparing the different libraries, it is more important to see which one has the fastest load time and the smallest size, rather than seeing the absolute time it takes to load the page, as these values will be greatly affected by the difference in devices

and environment. Desktop web browsers also have better debugging and performance measurement tools than mobile browsers do, which made it easier to get the results from the tests.

3.1.2 Storage & Offline Capabilites

It is important for MeetApp that the app works offline and that users can still see e.g. event and programme information. A strong storage model is therefore needed that does not allow data to be lost when phone storage runs low. Polymer has elements for simple data storage in its app toolbox, but since the storage model is more complex than what can be achieved with those elements, a custom implementation was needed. Therefore, the storage model itself had no significance toward choosing library, but helped with preparing for how to deal with storage in the final application.

3.1.3 Test Bench

Load times and file sizes were tested on a MacBook Pro 2015 with a 2.7 GHz Intel Core i5 processor. The PWA was served from one of MeetApp's Ubuntu 16.04 servers hosted in Microsoft Azure.

Most of the tests have been designed to compare different techniques by measuring the relative difference between them, rather than looking at the absolute results. This is because devices and the environment that the devices run in, such as the platform and network status, will have a significant impact on the absolute values such as load and rendering times and performance. A page that loads in under a second on a high-end device might load in five seconds on a lower-end device with poor network connection, thus making absolute values an inconsistent metric.

3.2 Implementation Details

The implementation of the final application has been focused around the features that MeetApp's current native application has right now and that need to be made sure they work well on the web platform. Some pages have been implemented to replicate the flow of the native application, but most of them include some features proving the

web platform's capability to meet the requirements toward the final application.

The implemented pages include a login page with authentication management, overview and details pages for programme items, events and dialogues, a settings page with native camera access, inbox and chat pages for live chats between participants through Firebase along with a few other pages. These pages and features were picked to cover all of the functionality that the native application has and that would need to be implemented in a rebuild of their native app to a PWA. The most important aspect here is efficient and smart storage, which is used on more or less every page. Storage is essential for the app to use as little network as possible and to work regardless of network connectivity.

The implementation will also focus on caching, rendering performance and user engagement. Test results from comparisons between different techniques will then be presented, along with an evaluation and discussion about the project.

The application is a continuation of the best suited proof of concept implementation, based on the experience of trying out different libraries and technologies along with results from performance tests comparing the different implementations. It was developed using one library and with a more clear set of approaches, compared to the proof of concept applications where various approaches were explored.

3.2.1 Methodology

The PWA was implemented using a feature-driven development methodology [17], where the features were designed to be as small as possible. This made it simple to test and revise the features together with Marcus at MeetApp after they had been implemented to make sure that everything would follow the requirements toward the final application. The general, largest features were identified at the beginning of the project and the smaller more detailed featured were identified along the way. All features were implemented one by one, in an order that suited the project at the time.

The feature-driven development style meant less overhead in terms of project management compared to many other methodologies, which left more room for development. This was beneficial for the project considering the one-man development team and the limited project

time.

The goal of the final application was to have more than a proof of concept application but it should not be ready for production use. It should have all the features that MeetApp requested but would be missing some important things before it could be published. First of all, the PWA should only contain a subset of the pages and features present in the native application, enough to make sure that everything that is needed to build the full application is available. Second, it did not have to be developed with proper error handling for all things that might go wrong during usage. It should contain some error handling for faulty network requests and empty responses but not enough for it to be considered final. Additionally, no unit tests needed to be written as the project's main focus is finding out whether the technologies for building a well-performing web-based mobile application exist, not building a production-ready application.

Chapter 4

Evaluation

This chapter will present the evaluation of the implementation of the final Progressive Web App, mainly based on results from tests described in previous chapters. The evaluation will focus primarily on comparing features well supported in native applications and that the web has traditionally lacked behind in, but also on different techniques and approaches within the web platform. It will also describe the implementation and design choices made to achieve the performance and functionality required in the project. The implementation is a continuation of the best suited proof of concept implementation from the previous chapter.

4.1 The application

The application built in this project is a PWA used as an information hub and interactive platform for participants at conferences or events. It is built as a demonstration of how MeetApp could switch from using their current native mobile application and instead use a web-based solution. The goal of the application is for it to look and feel like a native application, while being significantly easier to develop, publish and update, both for the company itself and for their customers. It uses the company's existing back-end for API requests and image hosting, but is built completely with web technologies in the front-end.

The application can be installed on the device via the browser's web app install banner. Once it is installed, it opens as a standalone application from the device's homescreen with a custom splash screen and full offline capabilities. On Android, it even gets it own storage

model and acts as a first-class application. This together with push notifications increases the user engagement by being able to actively remind the user that it exists.

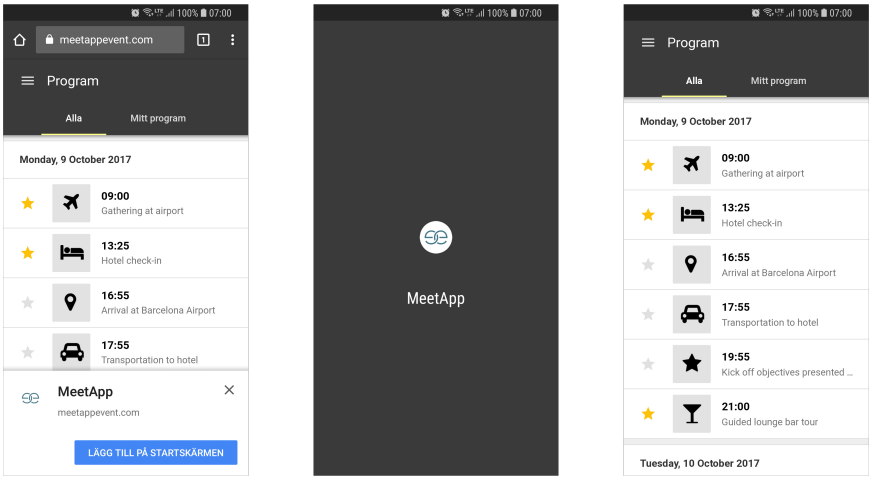

Figure 4.1: After installing the application, it gets a custom splash screen and opens in a standalone instance of the web browser.

4.1.1 Development Library

Polymer was picked as development library for the final implementation of the PWA. It is a library developed by Google with the intent of "using the platform". The library is closer coupled with the web platform than many other libraries and frameworks such as React and Angular, because of its use of web standards such as web components and standard module formats to avoid complex packaging tools like Webpack[1]. Its small size makes it possible to write performant web applications that are light and load fast.

At the very base, Polymer is a library for simplifying the process of writing web components. Since web component support is built into modern browsers, the library itself does not need to provide an additional layer on top of the platform to support modular elements, the way other popular frameworks do. Despite the small size, it still

[1]https://webpack.js.org/

supports important features such as data binding and conditional templates.

Polymer was picked mainly due to its performance advantage over its competitors for this type of application. The load times for all three proof of concept PWAs built in this project were measured using Google Chrome's DevTools which has built-in tools for this purpose. The `DOMContent-Loaded` and `Load` events relates to different steps in the page's lifecycle, the former being the time it takes until the HTML is fully loaded, meaning that the DOM tree is built but stylesheets and images may not have been loaded. The `Load` event happens after `DOMContentLoaded`, when all resources have loaded.

`DOMContentLoaded` can be considered a more appropriate metric as the `Load` event may depend on external resources to load, outside of the web host's control. In these tests however, no external resources are used.

	DOMContent Loaded (ms)	**Load (ms)**	**Size (KB)**
React	420	560	127,8
Preact	220	519	163,5
Polymer	120	406	118,1

Table 4.1: Measured load times for the three different JavaScript libraries used for the proof of concept application.

As these tests were performed on a desktop computer, the different implementations were also tested on a real mobile device to make sure that the results correlated with test results on mobile. Due to the lack of debugging tools in the mobile versions of the popular web browsers, the load times could not be as accurately measured as on desktop, but the results correlated with the desktop results, even though the load times were slightly higher.

4.1.2 Web Components

Web components are a suite of APIs for creating custom HTML tags that are reusable across the entire web. Developers previously have not been able to extend the set of elements that the browsers natively support, such as `divs`, `spans`, `inputs` and `tables`. Since the web

component APIs are standardised, no external dependencies are required in modern web browsers to use custom elements. This could help remove some of the burden from other frameworks and libraries which would otherwise need to provide an additional layer to give developers the notion of components. This is what Polymer does, which makes it a good library for writing a PWA. By using standard web components that the browser already knows how to deal with, the code can use more of the standards agreed upon and thus be loosely coupled with other independent frameworks and libraries, which both allows for reuse across the entire platform as well as speeding up web applications by possibly decreasing the size of the application framework or library.

4.1.3 The Service Worker

The service worker plays an important role in the application, by both managing push notifications in the background, intercepting and handling all outgoing network requests and storing network response resources in the local cache. All network requests originating from the application are intercepted by the service worker which, based on the request, decides whether it should be sent to the external server or if the if the resource should be fetched from the local cache. Polymer uses the sw-precache library[2] which makes it simple to configure what the service worker should store in the cache and from where resources should be retrieved when they are requested. For images specifically, the application uses the *fastest* option, meaning that the service worker will send the external request and then look for the resource in the cache. The response will include the resource from whichever source returned first. This option was picked since it will always return the resource as quickly as possible, while also guaranteeing the latest version of the resource because of the network request, as the cache might not be a reliable source for images that might change.

Other interesting options are *network first* and *cache first*. The former alternative would be a bit slower as the service worker would always wait for the network request to come back before displaying the image. The second option would possibly display old images even if there are updated versions on the image hosting server. There are also the options of only getting resources from network or only from cache.

[2]https://github.com/GoogleChromeLabs/sw-precache

Caching was setup so that the application would work equally well regardless of the network status to make sure that all content on all pages are loaded quickly even if the network connection is slow or absent. The app shell and all pages are stored in its own segment of the cache once they are viewed or the application is downloaded. All images that are loaded from MeetApp's own image hosting servers are cached at runtime to be available offline in another segment of the cache, which speeds up the loading on subsequent page visits. The use of multiple segments is both to keep different types of resources separated for ease of development but also because each segment may have different configurations for the maximum number of stored items in that particular segment before old ones are deleted.

4.2 Performance

Native apps run more efficiently and smooth than web-based apps since they run in tandem with the device platform that they were developed for. On the other hand, smartphones and tablets are so performant these days that native solutions are only necessary for certain types of applications, e.g. graphics heavy apps such as games. However, it is still important that the application has the prerequisites in terms of performant rendering and quick response to interaction to be able to run smoothly on mobile devices.

It is difficult to make a fair performance comparison between the Progressive Web App and MeetApp's native app, as they are written in different languages and run on different platforms. On top of that, the PWA has been implemented with somewhat limited features due to the limited time, which might give it an advantage over the native app. The PWA's startup time also depends on other factors such as whether the web browser it was installed with has recently been used and is in the device's memory. Since it runs in a standalone instance of the web browser, the browser needs to be running before the PWA can launch. Even though there are many factors which might impact the result of a performance test, some tests were performed to get a hint of the capabilities of the PWA. The tests assume that the PWA has been added to the homescreen and has all the dependencies downloaded, similar to what the native app has once it has been installed from the app distribution platform.

Context	Startup time
MeetApp's native app	∼3 s
PWA without web browser in memory	∼3 s
PWA with web browser in memory	∼1 s

Table 4.2: Startup times for the native app and the Progressive Web App in different environments.

The tests measure the time it takes from that the icon on the home-screen is tapped until the splash screen disappears and some content is displayed on the screen. The PWA launches quickly, in fact quicker than the native app if the web browser is in the device memory, but as previously mentioned, the functionality in the two apps are not exactly the same and the test results should be taken with a grain of salt.

The tests also largely depends on the web browser, which in this case was Chrome for Android on a Samsung Galaxy S8 phone. The tests were also performed on a lower-end device with the same results in terms of relative difference between the contexts.

4.2.1 Automated Tests

Lighthouse[3] (version 2.9.3) was used to test the quality of the PWA. The open-source tool has audits for things such as performance, general PWA quality and best practices and ranks them on a scale from 1 to 100. The network is throttled to emulate a 3G connection and the CPU is slowed down four times the speed of the machine to simulate the performance of a smartphone or a tablet. The scoring is based on performance metrics from real website performance data and according to the tool, a score above 90 represents the top 5 percent of top-performing pages.[4]

The application receives a 85 out of 100 score in the performance audit, with a time to first meaningful paint of 3.0 seconds on a 3G connection. The page is first interactive and consistently interactive at the same time, which means that links and interactions are going to work immediately when it has finished loading. The score is good but could be improved by optimising certain parts of the rendering and delivery. However, Lighthouse loads the application completely

[3]https://developers.google.com/web/tools/lighthouse/
[4]https://developers.google.com/web/tools/lighthouse/scoring

without the cache, so on subsequent loads when the cache has been populated with resources, the load time is under a second regardless of the network connection.

One reason that the PWA loads so quickly is because of its small size compared to a native application. The size of a page, without external resources such as images, is around 300KB, and navigating to other pages requires a few extra KBs because of the lazy loading of elements. MeetApp's native Android application is about 15MB. The small size is possible because of the web browser providing much of the functionality required to run the application on the device.

4.2.2 Perceived Performance

Perceived performance is the speed at which the software appears to complete a task. In devices with relatively limited performance, like smartphones, it can be wise to think about how users experience the performance of the application. The perceived performance can be improved by for instance loading small pieces of user interface before the entire application has loaded, such as the app shell with the navigation or just a simple spinner indicating that something is happening. This might give the user a bit more patience compared to just showing a blank screen.

A PWA by default helps with improving the perceived performance on installed applications by adding a splash screen with the application logo and title when the app is launched, similar to what native apps often do. The splash screen is shown when the application otherwise would be loading in the browser, showing a completely white screen. When the application has finished loading, the splash screen is hidden, and no empty screen is ever shown.

4.2.3 Load Times

The first thing that the user will notice when entering the page is the time it takes to load. To decrease the time, there are different techniques that can be used. The very first thing that can be optimised is how the resources are sent over the network, as there are new technologies that greatly improve the way servers can send a web page to the browser. Often in JavaScript-heavy apps, the application code is served as one large bundle that was generated during compile time.

Developers can utilise the new HTTP/2 protocol to send files more efficiently and improve the experience for the user.

Bundled

The Polymer command-line interface (CLI) comes with a few different build configurations; bundled and unbundled. Bundled means that all dependencies required for a page to load are served as one file, to reduce the number of network request needed before the page is fully loaded. The HTTP/1.1 protocol is not optimised for lots of request, which means that bundling can speed up the performance versus serving all dependencies individually. The problem with this approach is that individual dependencies cannot be identified and cached by the web browser, since they are merged together into a larger file. This means that the user either has to download a larger bundle with more dependencies than needed before anything is displayed on the screen, or that dependencies need to be downloaded multiple times in several larger files.

Unbundled

With the unbundled build, all files are served individually, which means that many more network requests are needed before the page is loaded. Despite the many network requests, this approach is more favourable if the app is served using the HTTP/2 protocol with Server Push [3] and the user uses a web browser that supports it. When using the new HTTP/2 protocol, the client requests only the app entrypoint and the server can preemptively send dependencies to the client before they are explicitly requested. Once the client has parsed the entrypoint file and requests its dependencies, they have already been downloaded and live in the browser's cache for future use and no dependencies need to be downloaded more than once. By using this technique, we get the decreased latency by downloading only what is needed for the page load and the performance benefits of fewer network requests as seen with bundling without actually bundling anything, along with efficient browser caching. How the application is served depends on the client browser, but the unbundled build is used for browsers that support it and the bundled build is used as a fallback for older browsers.

4.2.4 Rendering

Rendering large amounts of data is expensive for any system. Native apps often use virtual lists in order to achieve smooth scrolling performance in larger lists. Virtual lists render only the list items that are currently seen on the screen, instead of flat out rendering all items at once. For large lists this is beneficial as only the items inside the screen viewport are actually there. When the user scrolls, the existing items get moved around and the content is replaced, creating the illusion of one large list.

Polymer has built in functionality for virtual lists as part of its app toolbox, which was used in the PWA. Many pages display data in list format and the lists may sometimes be quite large. For instance, an event may have thousands of participants which would take unacceptably long to render using a regular list. Virtual lists were therefore used for all data lists, which yielded a significant improvement in both scrolling performance and rendering times.

This test shows the difference in rendering times for regular lists and virtual lists. The tests were performed on a desktop computer and aims to measure the relative difference between the two list implementations.

List items	Time (ms), normal list	Time (ms), virtual list	Improvement
200	1 434	677	112%
500	3 336	606	450%
1 000	12 448	644	1 833%
2 000	84 502	764	11 060%

Table 4.3: Difference in rendering times for normal lists and virtual lists.

The tests were performed with the virtual list provided by the Polymer app toolbox and normal DOM rendering and the time is measured from when the page has downloaded all dependencies until the list items are displayed on the screen. As seen in 4.3, the time it takes to render the virtual list is constant whereas the rendering time for a DOM rendered list grows exponentially, which makes the former the obvious choice. This particular virtual list renders 27 items on regular sized phone of which 9 are inside the viewport, to leave some time for the engine to switch content when the user scrolls.

Other than just slowing down list rendering, not using virtual lists slows down the entire page by blocking the main thread from performing other UI work. This means that the user is provided a completely blank page with no interactivity until the list has finished rendering, leading to a bad user experience.

4.3 User Experience

Since a PWA can be installed and added to the homescreen to behave like a native app in the device's operating system, users expect the same integrated experience with them as with any of their regular native apps, without noticing any significant difference in functionality and capabilities. To achieve good user experience, the app has support for things that users have come to expect from native apps, such as fast loading and interactions, offline capabilities and push notifications for user re-engagement.

One of the biggest problems when building the custom app-like experience in this application was navigation. In native applications, the developer has full control of the navigation, and can control things such as what happens when the user clicks the physical back button compared to web pages which lives in a controlled browser environment. The navigation in a native app is built like a stack, which together with page transitions to make the pages feel like they are stacked on top of each other, often gives the user a good perception of how the different pages are linked together. Browser navigation however is neither stack-based nor fully controllable by the developer. Visited pages are pushed onto a history list which for security reasons cannot be modified by the developer in any other way than inserting new pages to the end of the list or replacing the current state, which is the last item in the list. The user can however step backward and forward in the list, using functionality that the browser provides.

The navigation functionality that the browser provides by default gives the benefit of not having to worry too much about navigation when building web pages, but it also makes it a bit more difficult to create unique user experiences, on mobile in particular.

A problem that was identified after evaluating the proof of concept application was the unexpected navigation behaviour, especially once the user had installed it as a standalone app. There are a number of

top-level pages that the user can access by clicking its corresponding link in the navigation drawer. Whenever the user is at such a page, they would expect to back out of the app when they click the physical back button of the device. However, since all the pages end up in the navigation history, the user would have to back through the entire history before the app would close.

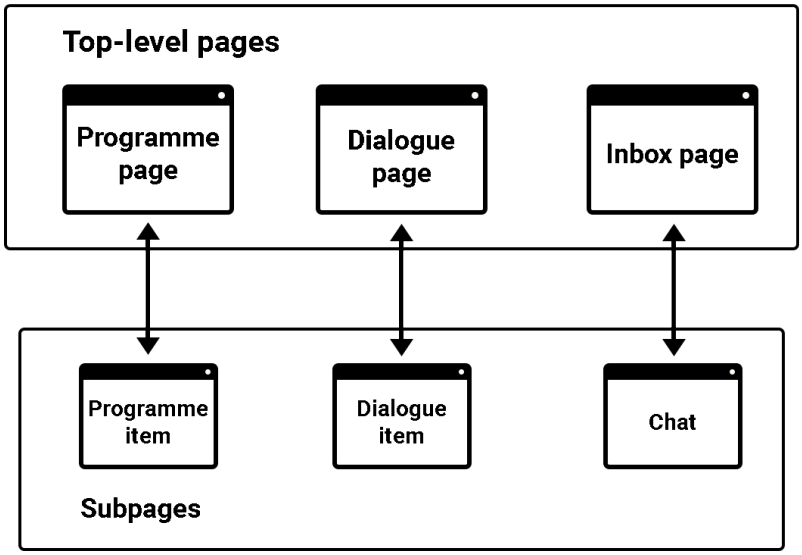

Figure 4.2: The desired navigation hierarchy. Backing out of any top-level page on an installed app should close it.

To solve this issue, the navigation in the application is manually controlled to be able to replace the current state rather than pushing it to the navigation stack when the user clicks on a link in the navigation drawer, similar to how it would work in a native app. This approach works on mobile, but gives a somewhat unexpected experience on desktop where users might expect all pages to end up in the navigation history. Clicking the back button in the browser would take them out of the entire page. Since the primary focus of this application is mobile, it was decided to use the approach where top-level pages replace each other in the navigation history so that pressing back on a top-level page would close the app. In the future, this behaviour could be adapted to the platform that the app currently runs on to match the default navigation that the user expects.

Another problem that surfaces because of the navigation problem

is deep linking. Deep linking is a significant benefit of the web, providing easy direct access to a "deep" page by just entering a URL in the address bar, e.g. a specific event's detail page. After clicking on a push notification that works as a link, the only thing that exists in the browser history is that particular page. This becomes a problem, because when the user clicks the back button in the browser, they end up leaving the page, when they might expect to navigate one level up, to the events summary page. To correct this unexpected behaviour, the browser history had to be "artificially" built. The landing page of a push notification is always the homepage of the application, which figures out where to navigate the user based on some query parameters extracted from the URL. It then builds the navigation stack by pushing all the pages leading to the requested page to the navigation history so that the user can back out to the expected pages after viewing that page. This takes away some of the benefits of deep linking by always having to load the homepage before going to the deep linked page, but it drastically improves the user experience.

4.4 Storage

Reliable storage is a prerequisite for applications to work offline, which has not always been a matter of course on the web. By using new web technologies such as the IndexedDB API [6] and the Cache API [25], developers can fully control storage and caching of resources without risking losing them after the browsing session has ended.

4.4.1 Persistent Storage

IndexedDB has played an important role in making the application feel fast regardless of the network status. IndexedDB stands for Indexed Database and is a transactional key-value store for persistently storing any type of data, such as strings, JSON objects or arrays. The API provides an abstraction for managing data in the database, with support for indexing, transactions, efficient searching and version management.

The database is divided into different object stores, similar to tables in a relational database, but without any predefined data structure. Data that is retrieved from the backend server, such as event information and programme items, are stored in separate object stores

of the database, indexed by their unique identifier. The application also stores the request time and sends it to the server on the subsequent request, resulting in the server only returning information about records that have been added, modified or removed after that time. The database can then efficiently update its stored records.

When the user enters the programme page, all programme items are loaded from the database, making the view display meaningful data to the user instantly even if the network connection is low or non-existent. Thanks to the IndexedDB API being asynchronous, the UI thread is not blocked from performing other tasks, which could be the cause with other synchronous storage solutions, making the interface slow and unresponsive. After loading the data from the database, a request is sent to the server to fetch any new programme changes. When the response comes back from the server, the store is updated and the programme items are loaded once again to update the view.

4.4.2 Caching

While user and application data is stored in persistent storage, other resources such as pages and images are stored in the cache. Using the Cache API, developers can cache resources without relying on the browser to do it for them, further opening up offline capabilities. The Polymer build tool automatically sets up caching for the app shell and the different pages, making navigation available offline. To make the experience even better, caching was enabled for thumbnails and poster images as well. All requested images that match a certain URL pattern are cached during runtime and are available via the service worker from anywhere in the application. This means that if the cache has been populated, the application looks exactly the same regardless of network connectivity.

The cache is divided into different segments designated for specific sets of resources, such as application documents, image thumbnails and poster images. The segments each have a file number limit which prevents the application from taking up large amounts of space on the device. The number limit also helps keeping the cache fresh by deleting old, possibly obsolete resources.

If the app is offline, all resources are fetched from the cache via the service worker. If the app is online, a network request is sent and then the service worker looks for the resource in the cache while waiting for

the network response, which speeds up resource fetching and guarantees the latest resources.

4.5 Back-end

As a way of improving the hosting of the application, the HTTP/2 protocol is used as default for serving the application files to clients. The new protocol improved the load times compared to the older HTTP/1.1 protocol, which is the most common protocol on the web. HTTP/2 Server Push was also tested as a way of further optimising the application delivery and speed up load times for users, by preemptively sending resources to the client after the application landing page is requested. However, the feature did not yield the expected improvement, possibly because the application already had relatively fast load times.

Unfortunately, Server Push could not be tested on the production servers due to problems with configuration. The assumption is however that the result would be the same as on the local system it was tested on.

4.5.1 Updating & Upgrading

The application is served from different subdomains of MeetApp's server for different companies, but the application code is hosted on one single domain. What distinguishes the experience between different subdomains is a custom web app manifest and a configuration file which is served together with the application from the respective subdomain. The web app manifest contains information about the PWA such as name, description and icons and the configuration file contains application information and layout specifications. Since the code does not have to be packaged in order to run on a device, this enables rich code sharing. All that is needed to create a new instance of the application is uploading the manifest and a configuration file to a new subdomain. Compared to using native apps, where the configuration had to be packaged with the application code separately for each company and then individually published to the app stores, this significantly simplifies the process of both publishing and updating the app.

The upgrading process for a PWA is different from a native application. Whenever the application needs to be updated, the new code

is uploaded to the web server where the application is hosted. When a user opens the installed app, the service worker will detect a new version and download it, but the changes will not be applied until the next time the app is launched. This behaviour can be changed to for instance reload the app as soon as the a new version is detected, which might instead lead to a bad user experience with unexpected page reloads. Considering that the updates most often will not be that significant, the best solution for this project was to show a message to the user saying that the application has been updated and that they need to restart it for the new changes to be applied. This also makes the updating process more seamless and transparent to the user than manually having to go and download and install the new version as they would on an app distribution platform.

4.6 Discussion

For the features needed to build this application, the functionality available in PWAs are enough and the application has the prerequisites to function like the existing native application. The set of features implemented in the PWA were the ones deemed most important from MeetApp's existing application with focus on features that native traditionally has done well and the web has not. All of these features were implemented successfully and work as desired. The only thing that felt lacking in terms of what could be achieved with the application was the navigation, which unlike native applications, is not entirely controllable in the browser due to security reasons. In order to create the custom navigation flow that MeetApp's native application has, and that feels most intuitive when using the PWA as a standalone application on a smartphone, some workarounds had to be used. This solved the issues on mobile devices but made the navigation feel somewhat unexpected on desktop browsers, which could be solved by adapting the navigation to the platform that the application is currently used on.

Unlike native applications, the web platform does not have direct access to device hardware, but need additional compatibility layers and therefore cannot reach the same performance for hardware interaction and rendering. However, smartphones and tablets are so performant these days that there are few cases where a native application

is needed to meet the performance requirements. The challenge lies in creating a good user experience with familiar and intuitive design and interaction, something that native has done for a long time and that the platforms have prebuilt components and APIs for doing.

Regarding platform support, the implementations in this project has focused on Chrome on Android, which has the most support for the functionality in PWAs. As of writing this, not all features are available on all operating systems and web browsers. All the major browsers (Chrome, Safari, Firefox and Edge) support the Cache API and the IndexedDB API[5], so cache storage and persistent storage should work everywhere. Service workers have just been released in all the browsers[6], but with some limitations in some of them. For example, only Chrome and Firefox on Android support the Push API, which enables operating system level push notifications[7]. Also, all browsers do not support the standardised web app manifest[8], and instead rely on meta tags to extract information such as name and icons from the application.

The benefit of moving from native development to web development for mobile application is mainly the code sharing aspect. Due to the application running completely in the web browser using standard techniques that modern browsers adhere to, it can run on any platform. PWAs are also built with progressive enhancements in mind, meaning that the browser does not have to support all the features supported by the application. The main focus is that the content can be displayed anywhere and the features depend on what the browser supports. The disadvantage of full code sharing however is the possible limitations in the features that may be implemented due to lack of support in some of the target platforms. In order for the application to work similarly everywhere, all platforms need to support every feature. However, thanks to W3C which together with the browser vendors develops web standards, developers can expect that new standards are implemented in all the major browsers.

These aspects make PWAs a particularly good option for smaller companies with smaller budgets that wants to target a large audience with their mobile application.

[5] https://caniuse.com/#feat=indexeddb
[6] https://caniuse.com/#feat=serviceworkers
[7] https://caniuse.com/#feat=push-api
[8] https://caniuse.com/#feat=web-app-manifest

Chapter 5

Conclusion

It is possible to build native-like applications with web technologies, however, it takes some work to get them to perform well as web-based applications by default are not as performant as native applications. The web platform is moving fast at the moment, while the native platforms are mature and stable and there are generally more defaults and recommended ways on how to do things like storage, layout and and how to achieve good performance. While native apps only require itself to run, Progressive Web Apps also require a well-built backend system that delivers the experience to the user in a good way, which might make web-based solutions a bit more demanding to build in terms of infrastructure. With the web as the distribution platform, the developer has to make sure that the entire stack works good, including the user experience in the app as well as the delivery of the experience.

The goal of this thesis was, as presented in the research question in the introduction, to see how PWAs could be built to match functionality, performance and user experience of traditional native mobile applications. The research question was then divided into four smaller questions. To answer the first question — whether web technologies are performant enough compared to similar native technologies — the performance and load times of the PWA was evaluated against an existing native application. The conclusion is that for this type of application, an information hub with access to a few native APIs without heavy graphics, the available technologies are performant enough to provide a good user experience.

The second question was about what is missing on the web platform before web-based solutions can serve as valid replacements for

native mobile applications. The most obvious flaw is the web's lack of direct hardware access and compatibility with native APIs, which might be a problem when building more sophisticated applications. For this application, all native features that were needed could be accessed through the web browser's API abstractions.

Regarding the third question about platform support, the technology is not completely ready as all the major browsers do not support the entirety of PWAs yet. iOS for instance does not support push notifications which might be considered an issue for certain applications. A good thing though is that users do not need the latest browser with all the modern features in order to be able to use the application. In the end, it is still just a web page with progressive enhancements, built to work in all web browers. Since this is more of a study whether it is possible for MeetApp to switch from their current native solution to a web-based solution, it is not the highest priority that it works on all platforms directly. The technologies used in this project are mostly standards decided upon by W3C and the presumption is that it will work on all platforms in the near future. Most of the features missing from some platforms are likely actively being worked on to close the gap.

The benefits of moving from native to web-based development, as mentioned in the fourth question, are mainly code sharing between all platforms and a simplified release and update process. The fact that the application is served as a web page makes it possible for any device with a web browser to access it, and it also makes it easier to keep the application updated for developers by circumventing the app distribution platforms.

Performance

Since web code do not have direct access to the native platform like native code does, the performance in a PWA will likely never be the same as in an application written with native code such as Java or Swift. However, smartphones are so performant these days that there are few cases where a native application really is necessary for the purpose.

User experience

The user experience by default may not be as good for a web-based application as for a native app, but there is definitely the possibility to make it as good. A PWA's user interface is built entirely with HTML, CSS and JavaScript, which for design purposes are arguably better languages for expressing layouts than e.g. XML and Java which is used in Android. It does however require some more effort to get a good user experience in terms of responsiveness and familiar design when building for the web than when building for native, as the native platforms' SDKs often maintains a wide range of pre-built UI components that the users are familiar with that do not exist by default on the web.

5.1 Future Work

Future work involves mostly exploring more web technologies that may be used to achieve similar performance as native applications for more heavy tasks — one being web workers. Web workers are threads that can run scripts in the background, to relieve the main thread from doing heavy work, and instead focus on UI rendering. However, that kind of performance was not needed in this project as no tasks were heavy enough to noticeably slow down rendering. Web workers could be compared to asynchronous tasks or background threads on native platforms, and is an important feature of the web to achieve good performance in computationally intensive applications.

Bibliography

[1] Android. *Hardware Acceleration*. URL: https://developer.android.com/guide/topics/graphics/hardware-accel.html (visited on 08/12/2017).

[2] T. Ater. *Building Progressive Web Apps: Bringing the Power of Native to the Browser*.

[3] M. Belshe, R. Peon and M. Thomson. *Hypertext Transfer Protocol Version 2 (HTTP/2)*. URL: https://tools.ietf.org/html/rfc7540#section-8.2.

[4] S. Bosnic, P. Ištvan and S. Novak. "The development of hybrid mobile applications with Apache Cordova". In: Telecommunications Forum (TELFOR), 2016 24th.

[5] A. Charland and B. Leroux. "Mobile Application Development: Web vs. Native". In: *Communications of the ACM* 54.5 (2011).

[6] World Wide Web Consortium. *Indexed Database API*. URL: https://www.w3.org/TR/2015/REC-IndexedDB-20150108/ (visited on 05/03/2018).

[7] World Wide Web Consortium. *Introduction to Web Components*. URL: https://www.w3.org/TR/2014/NOTE-components-intro-20140724/ (visited on 12/12/2017).

[8] World Wide Web Consortium. *Web Animations*. URL: https://www.w3.org/TR/web-animations-1/ (visited on 13/12/2017).

[9] A. Croll. *Building m.uber: Engineering a High-Performance Web App for the Global Market*. URL: https://eng.uber.com/m-uber/.

[10] Google Developers. *Lighthouse*. URL: https://developers.google.com/web/tools/lighthouse/ (visited on 12/12/2017).

[11] A. R. Edwards. *The Building Blocks Of Progressive Web Apps*. (Visited on 07/12/2017).

[12] M. Erfani Joorabchi, A. Mesbah and P. Krutchen. "Real Challenges in Mobile App Development". In: 2013 ACM / IEEE International Symposium on Empirical Software Engineering and Measurement. 2013.

[13] Facebook. *Facebook Q3 2016 Results*. Presentation. 2016.

[14] M. Firtman. *High Performance Mobile Web: Best Practices for Optimizing Mobile Web Apps*. 2016.

[15] Google. *About the Polymer Project*. URL: `https://www.polymer-project.org/about` (visited on 11/12/2017).

[16] Google. *Progressive Web Apps*. (Visited on 09/12/2017).

[17] J. Hunt. *Agile Software Construction*. 2006.

[18] W. Jobe. "Native Apps Vs. Mobile Web Apps". In: *International Journal of Interactive Mobile Technologies* 7.4 (2013).

[19] H. Joreteg. *Installing web apps on phones (for real)*. URL: `https://joreteg.com/blog/installing-web-apps-for-real` (visited on 01/12/2017).

[20] A. Lella and A. Lipsman. *comScore, The 2017 U.S. Mobile App Report*. Report.

[21] I. Malavolta. "Beyond Native Apps: Web Technologies to the Rescue! (Keynote)". In: Mobile! 2016 Proceedings of the 1st International Workshop on Mobile Development. 2016.

[22] I. Malavolta. "Web-based Hybrid Mobile Apps: State of the Practice and Research Opportunities". In: 2016 IEEE/ACM International Conference on Mobile Software Engineering and Systems. 2016.

[23] I. Malavolta et al. "Assessing the Impact of Service Workers on the Energy Efficiency of Progressive Web Apps". In: 2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft). 2017.

[24] J. Medley. *Web Push Notifications: Timely, Relevant, and Precise*. URL: `https://developers.google.com/web/fundamentals/push-notifications/#service-worker-involved` (visited on 01/12/2017).

[25] Mozilla Developer Network. *Cache*. URL: https://developer.mozilla.org/en-US/docs/Web/API/Cache (visited on 22/03/2018).

[26] Mozilla Developer Network. *Service Worker API*. (Visited on 05/12/2017).

[27] A. Osmani. *The App Shell Model*. (Visited on 03/12/2017).

[28] A. Osmani. *The Cost Of JavaScript*. URL: https://medium.com/dev-channel/the-cost-of-javascript-84009f51e99e (visited on 12/12/2017).

[29] A. Osmani. *The PRPL Pattern*. URL: https://developers.google.com/web/fundamentals/performance/prpl-pattern/ (visited on 14/12/2017).

[30] Kinlan. P. *What Comes Next for the Web? (Chrome Dev Summit 2016)*. Video.

[31] A. Puder, N. Tillmann and M. Moskal. "Exposing native device APIs to web apps". In: MOBILESoft 2014 Proceedings of the 1st International Conference on Mobile Software Engineering and Systems. 2014.

[32] D. Sheppard. *Beginning Progressive Web App Development. Creating a Native App Experience on the Web*. 2017.

[33] W3C. *Geolocation API Specification. W3C Editors Draft 11 July 2014*. URL: https://dev.w3.org/geo/api/spec-source.html#high-accuracy (visited on 24/11/2017).

[34] T. Wiltzius, V. Kokkevis and the Chrome Graphics team. *GPU Accelerated Compositing in Chrome*. URL: https://www.chromium.org/developers/design-documents/gpu-accelerated-compositing-in-chrome (visited on 06/12/2017).