Haaga-Helia
University of Applied Sciences

Analysis of deploying a React PWA on Google Play store using Trusted Web Activity

Yoseph Alemu

Bachelor's Thesis

Degree Programme in BITE
2020

Haaga-Helia
University of Applied Sciences

Abstract

Date 22.11.2020

Author(s)
Yoseph Alemu

Degree program
Business Information Technology

Report/thesis title
Analysis of Deploying a React PWA on Google Play store using Trusted web Activity

Number of pages and appendix pages
44 + 6

With the increased use of mobile devices across the world, it has become essential that companies utilize mobile application trends to reach more users and expand their platform availability. However, due to the high price of developing native applications. Moreover, from a web developers perspective, the additional skill required to develop native applications makes it hard to consider developing these applications.

This project aims to show an alternative approach to create mobile applications from an already available web app. The project will explore this idea by developing a React web app that meets PWA's criteria and wrapping it with TWA to publish the application in Google Play Store. Nevertheless, due to the time and resource constraints, it will not include the detailed development of the React application, Native app development, and publishing to the Apple App Store. However, it will provide necessary information about React, PWA, TWA, and web performance optimization.

Progressive Web Applications are faster, available offline, responsive, installable, and secure. These features make PWA a good candidate for mobile use, but PWA needs a browser to run. Therefore, by utilizing the power of the browser and web APIs, web applications can now present web content with the feel and features of Native applications.

This thesis presents the analysis of progressively enhancing a React Application and the process of web performance optimization that was needed to meet the requirements of using Trusted Web Activities

The thesis concludes that PWA and TWA combination has good potential for future development and makes it easier for developers with web application backgrounds to transition to mobile easily. Nonetheless, optimizing a web application to reach a Lighthouse performance score of at least 80 takes much work. However, considering the high performance that can be achieved, the result will be worth the work.

Keywords
PWA, TWA, React, Mobile Development

Table of contents

Abbreviations and Acronyms

AAB	Android Application Bundle
API	Application Programming Interface
APK	Android application package
AJAX	Asynchronous JavaScript and XML
API	Application Programming Interface
CCT	Chrome Cutom Tabs
CLI	Command Line Interface
CRA	Create React App
CRUD	Create, Read, Update and Delete
CSS	Cascading Style Sheets
CSSOM	Cascading Style Sheets Object Model
DOM	Document Object Model
JSX	JavaScript Syntax Extension
HTML	HyperText Markup Language
HTTP	HyperText Transport Protocol
HTTPS	HyperText Transfer Protocol Secure
JSON	JavaScript Object Notation
MDL	Material Design Lite
NPM	Node Package Manager
SDK	Software Development Kit
TWA	Trusted Web Activity
PWAs	Progressive Web Applications
SPA	Single-Page Application
UI	User Interface
URL	Uniform Resource Locator
WPO	Web Performance Optimization

Table of Figures

Table of Listings

1 Introduction

1.1 Topic

The number of smartphone users is increasing rapidly and provides businesses more opportunities to reach this broad market; therefore, this trend calls for more mobile application development investment. Most companies have a website; unfortunately, they do not have the resources or current necessity to develop mobile applications due to the additional investment needed to create Native applications. According to Laaksonen, mobile developers can take up to 5000 euros just for the prototype version, which is just the cheapest stage. By the end of the project, he estimates that the price tag could reach 15 000 to 30 000 € for a fully functional application. (Laaksonen, 2018)
Besides the price tag, for developers who work with the web application, creating Native apps requires additional skills, which halts the development time. Web developers who create websites are required to learn HTML, CSS, and JavaScript; on the other side, mobile developers need a good understanding of Java, objective-c, Kotlin, swift or other languages, and additional frameworks. (Snigdha, 2020) Therefore, it makes it hard for either of these sides to transition between these technologies.

1.2 Objective

If a company already has a web application but needed to reach more customers by developing a native mobile application available on app stores, the only option was to hire developers to create it from scratch. Still, there are better alternatives before going down that path, by redesigning the already available web app, turning it into a Progressive web app that can be installed on mobile devices and can also take advantage of the hardware already available.
The objective of this work is to show this alternative path for interested developers and other stakeholders before they invest time or money into developing native mobile applications from scratch. It is structured to show the overall development process from creating a React PWA to deployment on Google play. With Trusted Web Activity, it has become easier for a web developer to deploy a web app in Google play like Native apps, but without all the skills required to build Native apps.
TWA was first introduced in 2019 at the Google I/O conference, according to Peter Mclachlan and Andre Bandari from Google (Google I/O, 2019).TWA is the best way to show full-screen web content on the android app, it is a chrome browser with no browser UI, which gives it a Native look, and it is secured with Digital asset links that verify if the app owner is also the content owner. With this technology, developers can now easily turn

their Web applications with Progressive design guidelines, wrap them in trusted web activity, and publish to the store. (Google I/O, 2019)

1.3 Working Method

This thesis will have two parts, theoretical background and a practical section to achieve the above objective. The theoretical section discusses concepts related to the topic. A basic React web application with Finkinno API will be developed in the practical chapter, and the performance will be evaluated. Besides being one of the popular JavaScript libraries, React is maintained by Facebook and a vast community of developers, which gives it consistency, reliability, and hope for further growth; therefore, developers can use it without the fear of it being outdated.
After the development of the React application, it will be enhanced using Service Worker to make it work offline and for increased performance. Finally, it will be wrapped with TWA and deployed on Google Store.

1.4 Limitations

One of the project limitations was the lack of information regarding Trusted web Activities. It was introduced recently; therefore, it does not have enough documents other than the official Google developer website's general documentation. Also, working part-time has put too much strain during the finalization of this thesis.

1.5 Scope

This work will cover the basics related to the topic of this thesis, but due to the lack of time and resources, the following topics will not be included

- Full details of the React web app development
- Android and ios applications development
- Detailed publishing instructions for Google Play Store

2 Theoretical framework

The purpose of this chapter is to discuss the technological options that a company delivers its application software. Commonly application software is developed as a native application or a web application. This theoretical section will have seven subsections To provide a general overview of this project.

2.1 SPA frameworks

SPA framework or JavaScript frameworks are a collection of HTML templates and JavaScript code that comes pre-written to avoid repetitive programming tasks. The term “framework” is usually used interchangeably with “library”; they are both a collection of code that can be reused to make development easier. However, the difference is the control level they give. Libraries provide specific functions that can be called to solve particular problems, giving more control for the coder but do not have full structure. On the other hand, frameworks provide the full structure for development. They can include a set of libraries, but frameworks set how these resources are used also called “inversion of control.”
They give developers the ability to customize and build on top of them instead of starting from scratch to create the desired applications. The use of these frameworks saves development time and makes scaling applications easier. Few JavaScript frameworks are available on the web right now; they are mostly open-source, backed by big technology companies and developers, which means anyone with the skill can contribute, and developers can use these technologies for free. (Scott, 2015)
According to Stack Overflow 2019 statistics, JavaScript is the most popular programming language, and this has been unchanged for the last decade. Also, jQuery, React, and Angular has been named the most popular JavaScript frameworks with 48.7, 31.3, and 30.7 percent, respectively. (Stackoverflow, 2019).
However, for this thesis, due to its popularity and the writer’s experience with this library, React Js will be used as the primary development library and will be discussed in this section further.

2.2 React

“React.js is an open-source front end JavaScript library(not a full-fledged framework) to develop SPA that was created by a team of Facebook developers led by Jordan Walke back in 2011 and became open source in June 201” (Nikhil, 2020). Although Facebook develops it, it is also maintained by a large community of developers and companies (Wikipedia, 2020)

React is considered by most to be one of the most disruptive web app technologies throughout the last decade. It is faster and more versatile than Angular, which is another popular JavaScript framework that is maintained by Google. Additionally, React starter template supports the use of PWA. React is also widely used by large companies such as Facebook, Netflix, Uber, Airbnb, and more. (webapp007, 2019)

According to the React official page in technical terms, “React is declarative, Learn Once, Write Anywhere component-based Library to build single-page application” (ReactJsOrg, 2020).

React is declarative because the programmer does not have to interact with the DOM directly, but the virtual DOM instead, so declare what the component UI should look like in a particular state, and React handles the rest with the DOM. It is also “Learn Once, Write Anywhere” React can also be used to create mobile apps using React-native (which will be discussed further in the coming section), desktop apps with Electron, and back-end development with Node. Therefore, it is cross-platform and minimizes the learning curve for creating applications for different platforms.

As a component-based library, React supports the creation of “self-state managing” UI components, which can be reused. (ReactJsOrg, 2020).

2.2.1 DOM and Virtual DOM

DOM (Document Object Model) represents the actual application structure created by the browser when the page is loaded. It is an interface for the program to change the style, content, and structure of the page. The Object model allows JavaScript to change, add and remove elements, attributes, and CSS, also reacts to HTML events. (Codecademy, 2020)

The manipulation of the DOM is slow and made worse because most JavaScript frameworks update the DOM more than they have to; therefore, React brought a solution by introducing Virtual DOM. (Codecademy, 2020). “The virtual Dom is a fast, in-memory representation of the real DOM, and it’s an abstraction that allows us to treat JavaScript and DOM as if they were reactive” (Fedosejev, 2015)

It works by creating a virtual representation of the real DOM by rerendering the UI whenever there is a change in the data model. Then by comparing it with the previous version of the virtual DOM, it updates the difference to the real DOM.

This creates a fast experience because only the difference in the two virtual representations is updated, and React lets developers write code as if the entire DOM is rerendered every time the application state changes. (Fedosejev, 2015)

2.2.2 JSX

“JSX is an XML/HTML-like syntax used by React that extends ECMAScript so that XML/HTML-like text can co-exist with JavaScript/React code. The syntax is intended to be used by preprocessors (i.e., transpilers like Babel) to transform HTML-like text found in JavaScript files into standard JavaScript objects that a JavaScript engine will parse.” (reactenlightenment, 2020)

In simple terms with JSX helps developers write JavaScript in HTML code, and by using Babel, it is possible to convert JSX into JavaScript code.

As shown in Listing 1, it is possible to write HTML-like text inside a function, but it has to be enclosed in a or single element, multiple closing tags are not allowed, and in Listing 2, the converted version of JSX is shown.

```
const Test = () => {
    return (
            <h1>This is JSX</h1>
    )
}
```

Listing 1 JSX in React functional component

```
function Test () {
    return React.createElement('h1', {}, 'This is JSX');
}
```

Listing 2 JSX converted to JavaScript code with Babel

2.2.3 Components

As discussed in the previous section, React is a component-based library, and these components can be reused when needed. It is possible to breakdown components to render a single word or render many elements; therefore, it makes it easier to handle the rendered UI independently. (Kagga, 2018)

“A component is a JavaScript class or function that optionally accepts inputs, i.e., properties(props) and returns a React element that describes how a section of the UI (User Interface) should appear.” (Kagga, 2018)

As shown in Listing 1, Test is a component that returns a JSX, and it can be exported using “export default,” so other components can reuse it.

The example shown in Listing 1 is a functional component, but there is also another kind of component called a class-based component. They are also called stateless and stateful components, respectively. That is because functional components, unlike their class-

based counterpart, could not store component states and were only used to receive props and render React elements. After the introduction of React Hooks in React version 16.8, it is possible to use state in functional components using the “useState” function(hook). (ReactJs, 2020).

2.3 Native Applications

Native apps are developed specifically for a particular mobile device and are installed directly onto the device itself. (Perera, 2020)

Native apps work only on devices or platforms they are designed for because, unlike web apps that run in browsers and are built with the language of the web(HTML, CSS, JS), Native apps are developed with a specific programming language and with certain devices in mind.

There is a variety of hardware available in the market, such as desktop, mobile, tablet, smartphone; there are also few major Operating Systems that run these devices, such as IOS, Android, and Linux. All these hardware and Operating Systems require machine-specific programming language to develop native Apps. For example, IOS running devices require Apps that are created with languages such as Objective-C or Swift, while Android devices need Java, Kotlin, and C++ programming languages. Due to these specific requirements, developing multi-platform apps were a problem both for clients and developer. (Perera, 2020)

As previously discussed, web applications use the power of HTML, CSS, and JavaScript to present content to users’ screens, and they can only run on devices with a browser. Another solution is using Hybrid applications; Hybrid is, as the name indicates, a combination of Native and Web.

Hybrid apps take advantage of the simplicity of presenting web elements inside Native apps using WebView; this provides the benefit of both technologies, as shown in Figure 2; hybrid apps utilize the high performance, reactivity, and use of hardware features from Native and multidevice support, responsiveness and simplicity from the web. (Saccomani, 2020)

2.3.1 Native vs Web

According to (Saccomani, 2019), smartphone users are likely to spend more time on their native apps that they installed from their respective stores such as Google Play and Apple’s app store, Leaving less time for web browsers to be utilized as they are only used to look up information. (Saccomani, 2019)

There are many reasons why smartphone users prefer to spend time on their native apps.

Engagement

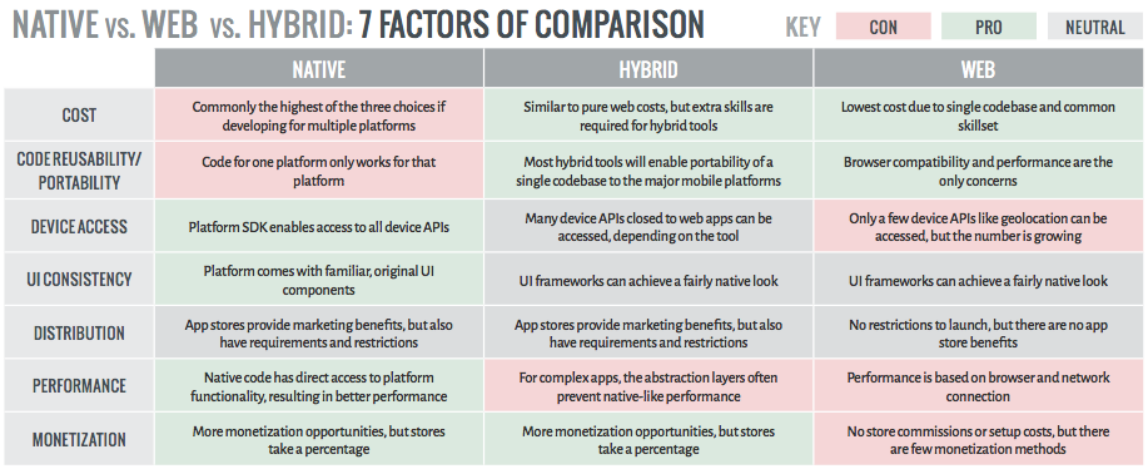
Native apps are engaging; when we wake up in the morning, we can see all the information we need in the form of notifications; these updates are sent by our apps and run in the background even if the application is closed, therefore it will be easier for users to return to the application without meaning to use it. (Saccomani, 2019)

Access

All our native apps come with icons that can be placed on our phone screens, either on the home screen or application list screen, and they can be accessed with a tap of a button. Unlike the mobile web, which makes it harder for users to come back because of the hassle with writing URL in the browser. (Saccomani, 2019)

Features

Native apps have access to mobile hardware's full capabilities; we use the camera in our apps to take a picture and videos, geolocation to use maps and track our location, microphone for sending voice information, and Bluetooth to connect other devices and so much more. Therefore, access to these hardware features has made Native apps powerful and preferable over web apps for users. (Saccomani, 2019)



NATIVE vs. WEB vs. HYBRID: 7 FACTORS OF COMPARISON

KEY: CON | PRO | NEUTRAL

	NATIVE	HYBRID	WEB
COST	Commonly the highest of the three choices if developing for multiple platforms	Similar to pure web costs, but extra skills are required for hybrid tools	Lowest cost due to single codebase and common skillset
CODE REUSABILITY/ PORTABILITY	Code for one platform only works for that platform	Most hybrid tools will enable portability of a single codebase to the major mobile platforms	Browser compatibility and performance are the only concerns
DEVICE ACCESS	Platform SDK enables access to all device APIs	Many device APIs closed to web apps can be accessed, depending on the tool	Only a few device APIs like geolocation can be accessed, but the number is growing
UI CONSISTENCY	Platform comes with familiar, original UI components	UI frameworks can achieve a fairly native look	UI frameworks can achieve a fairly native look
DISTRIBUTION	App stores provide marketing benefits, but also have requirements and restrictions	App stores provide marketing benefits, but also have requirements and restrictions	No restrictions to launch, but there are no app store benefits
PERFORMANCE	Native code has direct access to platform functionality, resulting in better performance	For complex apps, the abstraction layers often prevent native-like performance	Performance is based on browser and network connection
MONETIZATION	More monetization opportunities, but stores take a percentage	More monetization opportunities, but stores take a percentage	No store commissions or setup costs, but there are few monetization methods

Figure 1 Comparison for Native, Web and Hybrid Apps (web2appinfotech, 2016)

2.4 React Native

React Native takes a different approach from both Native and Hybrid Apps; it is neither hybrid nor native. Like React, it is entirely built with JavaScript, which makes code-sharing so much easier. React Native is also built by Facebook; it uses a similar codebase with react, the only difference being "In Reactjs, virtual DOM is used to render browser code in Reactjs while in React Native, native APIs are used to render components in mobile. The

apps developed with Reactjs renders HTML in UI while React Native uses JSX for rendering UI, which is nothing but JavaScript." (Shah, 2020)
There are few ways developers can use while starting to develop React Native applications, one way is to use Expo client, and the other is using the React Native client.
Expo is a set of tools and services that helps developers build, deploy, and quickly iterate on native Android, iOS, and web apps from the same JavaScript codebase while taking advantage of the capabilities of the smartphones' competencies, such as camera, location, notifications, sensors, haptics, and so much more. *Expo* (2020).

2.5 Progressive Web Applications

2.5.1 History of PWA

The word "Progressive web App" was first introduced in 2015 by Alex Russel, who was a chrome developer. However, the interest in this technology goes back to the first introduction of the iPhone in 2007. During the time, Apple's CEO Steve Job was excited about the development of web apps that could harness the capabilities of the Safari browser to run on all iOS devices seamlessly. However, due to a shift in the company's direction, the interest quickly died down. (ScandiPWA, 2019)
Despite that, other technology companies such as Google and Microsoft have taken the initiative in advancing this development approach, but it was Google who first adopted this approach in early 2015. With the increase in mobile internet usage, it became commercially profitable to develop content with mobile users in mind. However, the responsive design was not enough to attract these users without compromising on developing expensive Native apps. (ScandiPWA, 2019)
Microsoft has also been pushing to make PWA a standard for developing web applications. By launching PWABuilder, they have made it easier to turn a typical web application into PWA that can be wrapped and deployed on the biggest App Stores. With the contribution of Google, PWABuilder uses Bubblewrap CLI under the hood to generate a TWA packaged version of the provided application, which can be deployed on Google play store also, APK version of the app that can be tested in android studio or on a mobile device. This service, however, is not limited to Google Play Store; this tool can also wrap the PWA to be published in Apple App Store, Microsoft Store, and Samsung Store.
The early adoption of this approach has given companies business success, and that has sparked interest in other businesses; for example, Alibaba has increased conversion rate by 76%, the Washington Post has seen an 88% increase in their site performance and these are a few of the success stories, more can be seen on Figure 2.

Figure 2 PWA Success stories (ScandiPWA, 2019)

2.5.2 Progressive enhancement

One of the core concepts of building a progressive web application, as the name indicates, is Progressive enhancement; it is a web app design strategy to create and deliver the critical web content and functionalities that can be available to all users no matter the device or browser. This is done by incrementally adding enhancements to devices and browsers that can support newer features without compromising the contents provided to lower versions. (Todd Parker, 2010)

It was hard to estimate and design an application that is available for all users, as the browser technologies and devices(screen size, device version, brand) used are different across users and get updated quickly; besides, it is vital to keep in mind the network issue in other parts of the world. That is why creating the basic user experience that can deliver the basic functionalities is essential. When building a website, the primary goal is to ensure that everyone gets the best possible experience within the capabilities and constraints of their browser. (Todd Parker, 2010)

2.5.3 PWA Features

There are many reasons why PWAs are becoming popular, the most prominent being performance, but that is not the only benefit it brings to the table; it has a lot of features that stand out from previous development approaches such as native feel, installability, offline use, update, responsiveness, and security. (ScandiPWA, 2019)

Native feel: with increased performance and smooth navigation that Native apps are known for, PWA also uses many WebAPIs to achieve features that were traditionally reserved for Native apps, such as geolocation, push notifications, camera access, and much more. Full WebAPI features available for the latest version of Chrome can be seen in Figure 3.

Installable: PWA can be installed on the home screen of mobile devices. With the use of App Manifest, PWA can tell the browser how to install the app. Usually, this feature was available by using the install button that becomes available in the browser when the PWA loads, but with TWA, it can be downloaded and installed from the Google store. With the help of additional tools, however, PWAs can be packaged and deployed in other app stores.

Offline: with the use of Service Workers, PWA can make content available offline after being accessed one time; this is achieved by caching and storing static pages during initial load.

Update: If the device has a connection, the content can be updated on the fly.

Responsive: PWA is made with all user devices in mind; therefore, responsive design is an integral part of the development process. They can be available on any device with a browser; although some features might not be supported by some browsers, it can still provide the basic functionalities.

Security: PWA only runs over secure HTTPS protocol keeping information safe.

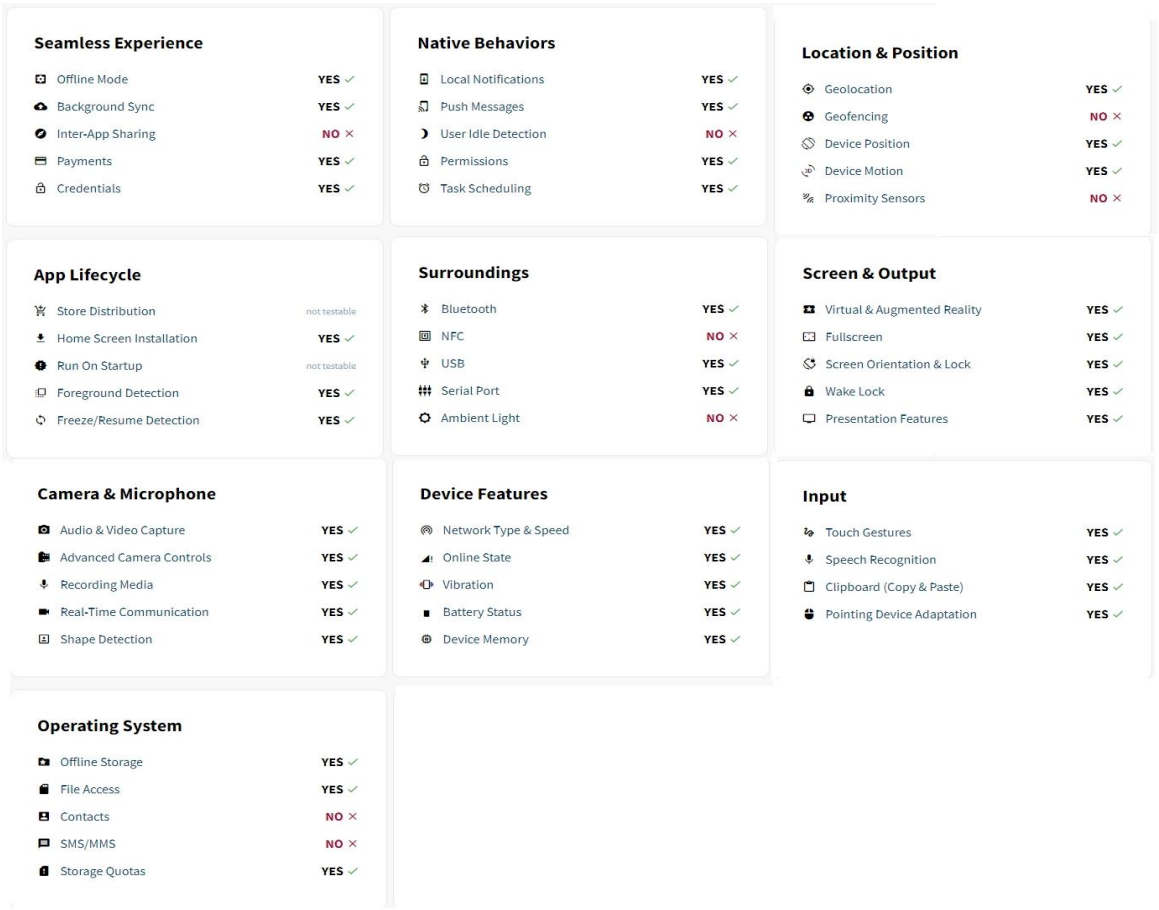


Figure 3 WebAPI Features available in latest Chrome version 86.0.4240.75 (Bar, 2020)

2.5.4 The web app Manifest

It is a single JSON formatted object that is stored in the root folder containing information about the application. It provides the browser with additional information about the application it is running, and if needed, information on installing the application on the home screen of a mobile device. (LePage, 2020)

From a business point of view, this helps the application to be accessible along with other native applications giving it extra accessibility and attention from the users. From the users' point of view, it provides ease of access as they don't have to open the browser just to reach the content needed. Bookmarks are another option, but using them can be time-consuming as users bookmark many websites but end up getting confused about where they are stored.

Web Manifest properties

Name: the web app always has a name provided, but the browser will use this as the title of the splash screen when the application is installed on the mobile home screen.
Short_name: this will be the title used under the icon of the application
Start_url: the content or page that will be loaded after the tap of the icon.
Scope: scope defines the pages that will be included in the PWA; by default, the value of this property comes with "." this means all the pages in the application are included; it can also be made to include a few of the pages by specifying the names.
Display: this property defines what the app should look like once it loads on mobile devices' home screen; the default value is standalone, which hides the browser input and controls which are typically seen on the browser page, thereby giving it a more native app look. There are also additional settings such as fullscreen,minimal-UI, and browser. The fullscreen setting shows the app without any browser UI integrated while the browser displays it with full browser UI.
background_color: defines the color of the background on the splash screen.
Icons: an array of icons that will be used when the app is installed on mobile.

2.5.5 Service Worker

It is a simple JavaScript file that is stored in the browser and stays with it even if there is no internet connection, so users can see the browser partially, even if it is offline. In technical terms, a Service Worker is a JavaScript file that runs in the background on a separate thread from the normal JavaScript. HTML inside the browser loads regular JavaScript files; it runs on a single thread, which means that one command is run at a time, and even if there are many JavaScript files, they are executed Synchronously. The loaded script can then manipulate the DOM to make the static HTML page more dynamic. In applications without Service Workers, the interaction between the server (which provides the content) and the browser (that sends the request) is direct; therefore, browsers need an internet connection to get content back. (MDNcontributors, 2020)
Service Worker is initially loaded by the root HTML page; therefore, after getting registered, it can be used by all pages of the web application. Unlike regular JavaScript, it is not linked with one particular page; therefore, it can run in the background even if the pages are closed and will respond to specific events.
The Service Worker is restricted from using local storage, the DOM, and the window directly because it cannot work asynchronously with the regular JavaScript running inside. However, pages can interact with the Service Worker by direct postMessage, one-to-one

Message Channels, and one-to-many Broadcast Channels; using this, it can indirectly communicate with the DOM. (Geddes, 2019)

Also, for security reasons, Service Workers only run over HTTPS protocol. As the middle ground between the application and the server, it is crucial that the connection is secure and not open for an attack. (MDNcontributors, 2020). However, it can be used over the localhost for development purposes, which will be demonstrated in the implementation chapter.

Before the use of Service Worker, however, it is crucial to know the browser's version that is compatible and support the registration of Service Workers. There are few services online where this can be done. Some even support live browser compatibility testing while coding. For this project, I have used caniuse.com, which shows the full list of features supported by the major browsers.

Figure 4 Service Workers browsers support (Deveria, 2020)

The most important role of Service Workers is to listen to events on both sides of the application, from both the server-side and the client-side. It can listen to events and decide whether to respond with a predefined message, a cached static page, or redirect it to the main page.

By caching pages, the Service Worker ensures that even when there is no connection, it can serve the page without failure; therefore, the user will not get the annoying offline response page from the browser.

There are few events Service Workers listen to make changes, such as Fetch, sync, and Push; these events fall under functional events while register, install, and activate are part of lifecycle events. Functional events are either received from the server or the client, while Lifecycle events belong to the Service Worker itself. (MDNcontributors, 2020)

Figure 5 Summary of Service Worker events (MDNcontributors, 2020)

Fetch events: By making use of fetch API, which is a cleaner way of making AJAX calls, Service Workers could be able to intercept requests that are triggered when a new file is requested from the server. The Fetch API also uses promise for asynchronous operations. A promise is a great way to make a request which might not be available during the time but will be resolved once the promise is ready to return a value.

That is why it is essential to check if the browser supports both fetch API and Promise API; as shown in Figures 6 & 7, they are supported by the major browsers such as Chrome, safari, edge, and opera. Although some sub-features are not available for all browsers, this is a good start. (Love, 2019)

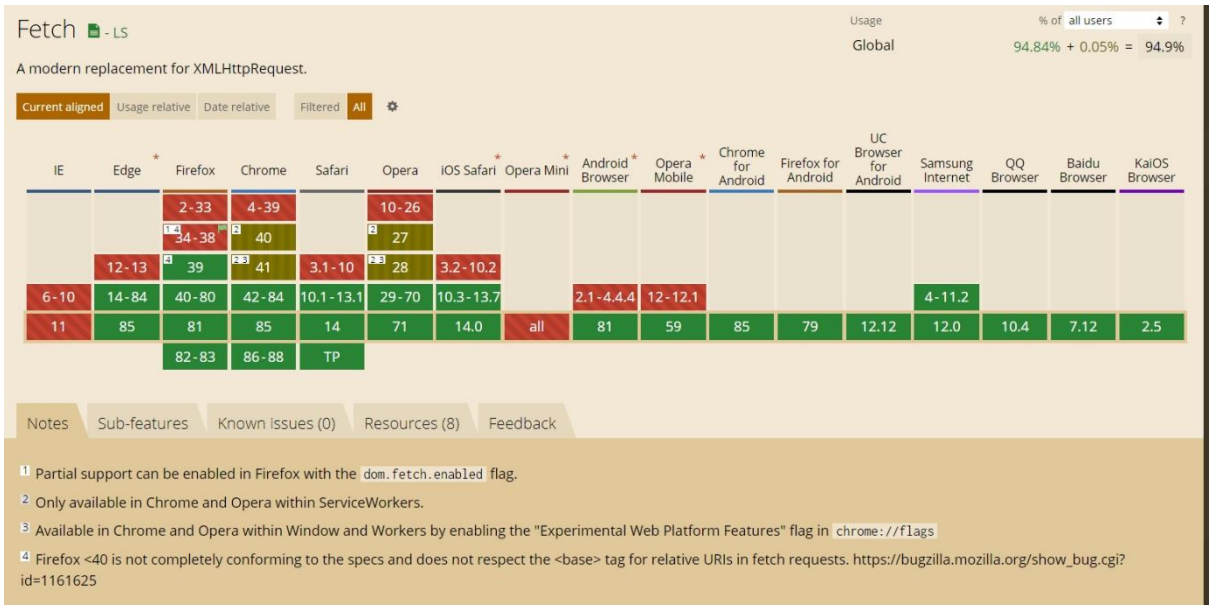


Figure 6 FETCH browser compatibility (Deveria, 2020)

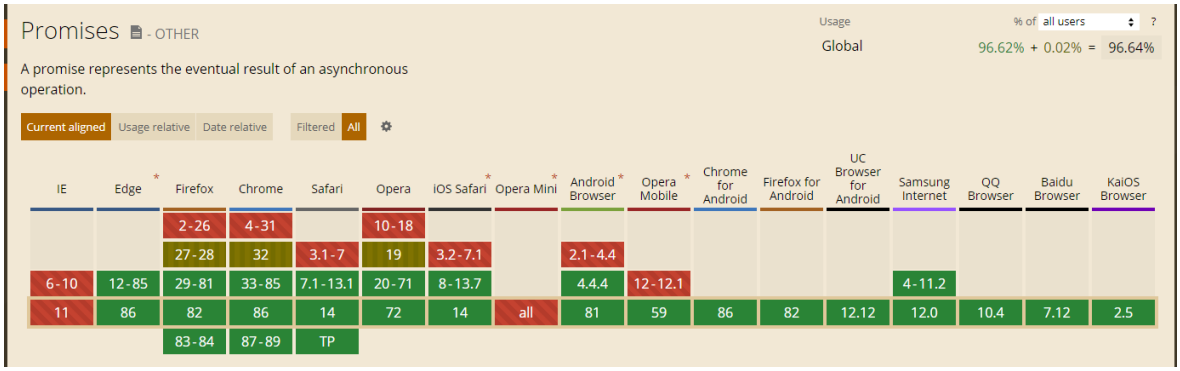


Figure 7 Promise support for browsers

Push events: Service Workers take advantage of the Push API to listen to messages that are sent by web push servers and display them to the user using notification API; this brings native features for our web app, making interaction with the user more efficient.
Interaction: After the notification is loaded on the page, the user will be able to interact with the message, which will trigger an event; Service Worker also listens to these interactions and respond accordingly.
Background synchronization: Service Worker listens to events that are created by the browser when there is no internet connection, that is because it runs in the background even without a connection; therefore, it will be able to respond with a promise which it will execute once the connection is restored until then the actions will be stored in the cache. But it is important to note that Sync API is currently not fully supported by all browsers.

Service Worker Life cycle

This life cycle refers to the stages the Service Worker takes from the loading of the URL up to the activation and final control of the scoped pages. As shown in Figure 8, the root index.html loads the main JavaScript file that is app.js, which runs the Service Worker JavaScript code to run in the background and registers it as a background process after downloading; as stated above, it is executed separately from the regular JavaScript code. But before that happens, the Service Worker checks whether the browser it is running on supports its use; if so, it will register, thereby triggering the installation event; however, it is only activated if the browser is registering the Service Worker for the first time or if there are new updates in the page that are under its scope. (Service Worker API, 2020)

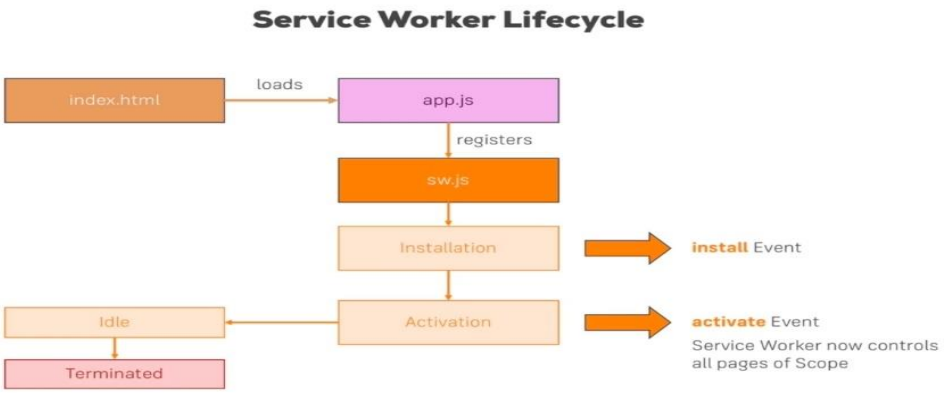


Figure 8 Service Worker life cycle (Rabouw, n.d.)

1st phase (Registration): To make use of Service Worker, it must be registered as a background process; this is achieved by executing a JavaScript code from the root of the application; this code contains the serviceWorker.register() method to download and register the Service Worker when it succeeds, it will be downloaded to the client and try to install itself. (Service Worker API, 2020)

Also, a scope can be defined to tell the Service Worker the limits of control over the pages of the application; this can be seen by using the *registration.scope* method. As shown in Listing 1, the code will check if *serviceWorker* property is available in the navigator(which is the browser); if so, will point to the location of the Service Worker that will register sw.js when the page loads and log a message of success also its scope which is the home page, but normally its scope is the entire page by default, else logs a fail message. If there is an already installed Service Worker, It will return the registration object of the active version. (DevelopersGoogle, 2019)

As shown in Listing 3, the registration is complete after the successful loading of the SW.

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

    });
}
```

Listing 3 Service Worker registration

2nd Phase (Installation): The registration of the Service Worker also triggers the installation phase; this is done by using the **addEventListener**, which is a special event, unlike the normal “addEventListener” that has DOM access. After completed installation, the Service Worker will be able to precache a portion of the pages to serve them on the next reload. As shown in Listings 2, the install event will trigger the installation of the Service Worker. (DevelopersGoogle, 2019)

```
self.addEventListener('install', (event) => {
    console.log('[Service Worker] Installing Service Worker ...', event);
});
```

Listing 4 Installation phase of a Service Worker

3rd phase(activation): Upon successful installation, the phase of activation is triggered but will consider the usage of other Service Workers that are in use before activating; this is to create consistency in the app, so only one should run at a time. After activation, the Service Worker can start monitoring the pages in its scope but ensure that the pages must be reloaded, and the new version should claim those pages.
As shown in Listing 6, it needs to listen to the activate event before going forward.

```
    self.addEventListener('activate', function(event) {
        // Perform some task
      });
```

Listing 5 Activation phase of a Service Worker

2.6 Trusted Web Activities

TWAs helps in integrating the web experience with the Native experience; PWAs already provide this experience with “add to home screen” and other Web API features, but TWA takes this experience further.
TWA provide a full-screen view for web apps on mobile devices, making it look like a Native app and supports the full features of PWAs, also making the PWA deployable in Google store; what makes TWA more powerful is its capacity to utilize Digital Asset Links to confirm the content owner is also the one deploying the application.
TWA is full screen, meaning it shows web content in full display like a regular native app by covering the whole screen of the mobile device except the system UI; this feature is not

achieved by CCT mainly because it has to display the URL bar, TWA solved this problem by using Digital Asset Links to confirm the content owner is also the one deploying the application thereby eliminating the need to show top URL bar and other browsers UI.
TWA builds on the benefits of WebView and Chrome Custom Tab, which were mainly responsible for integrating the web experience with the Native; these technologies can be embedded in native applications to show web content inside the native app screen.
This section provides a clear understanding of this technology and its priors (WebView and Chrome Custom Tab).

2.6.1 Benefits of using TWA

TWA is powered by Web APIs in the browser of mobile devices; therefore, its features are dependent on the available Web APIs. Unlike Native applications, which have full access to the hardware capabilities of mobile hardware, Web APIs lack access to certain features such as Inter-app sharing, User Idle detections, geofencing, proximity sensor, contacts SMS, ambient light, and NFC. These features are based on the current version of Chrome (86.0.4240.75); more features are available in Figure 3.
It is important to note which features are supported by all target users to provide the best experience; if one of the above features is a necessary requirement, it is best to use a Native application, but there are important benefits TWA provides over Native. (RAJ, 2020)
Lightweight: All the content displayed in TWA is fetched from the web; therefore, it is not as bloated as Native apps; the size of the apps will barely cross 2MB; this saves a lot of space in mobile devices with small memory size.
Content-driven: If the desired application is content-driven and doesn't require complicated mobile hardware use, TWAs could be the best option.
Update: Native applications need to be updated every time there is a change, and users without recent update will not get full functionalities, but TWA automatically update the content.

2.6.2 WebView

A WebView "is an embeddable browser that a native application can use to display web content" (Kirupa, 2019).
In typical Native applications, when the user clicks on an external link, the request will try to fetch a web content that gets redirected to the default mobile browser; this is observable when the browser opens on a new screen with all the navigations and the URL bar.
So, the solution is to use WebView to show the web content without redirecting it to the browser; WebView loads contents both from HTTP and HTTPS in the view box without

leaving the native app. Therefore, WebView integrates well with Native apps; it takes requests from the clicked link and renders it on a full-screen tab or small section of the app screen, making the redirection process feel as if the user never left the app. In addition, it gives the developer the freedom to make few tweaks on the view screen to make it feel more in theme with the app, and this is a significant user experience change compared to redirecting to the browser.

Besides the user experience, it makes updating the contents inside very easy; since all the information inside the WebView is coming from the server, the web app is connected to, updates are seamless and automatic.

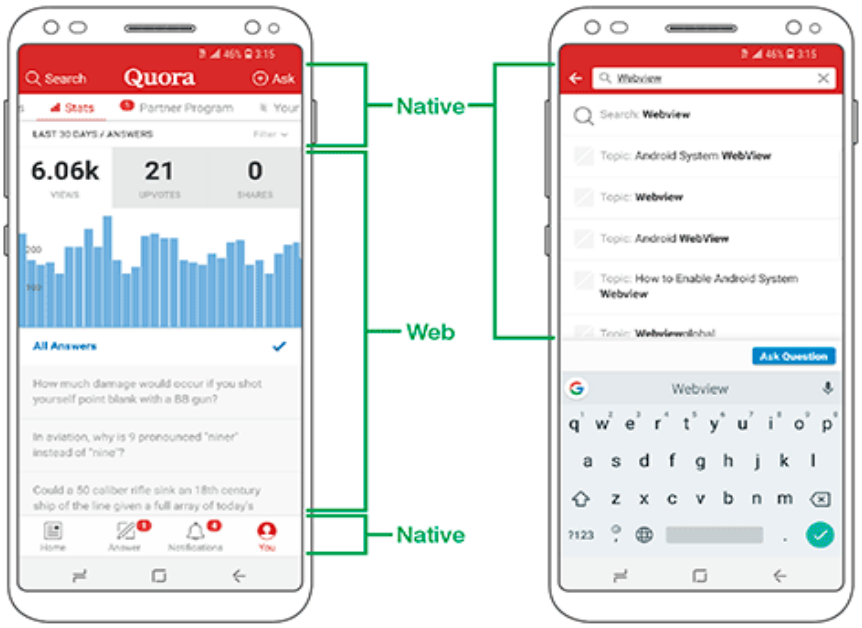


Figure 9 WebView use example (Hoober, 2018)

It is vital to note that even though WebView renders web content, the content should not be data-heavy, which clashes with the native concept of the app. It is also good practice to plan for the contents that should be served inside the WebView and what should stay in the Native; the best way to design is to avoid rendering functionalities that require interaction with the hardware and other complex features such as input boxes and navigations inside WebView. (Hoober, 2018)

As shown in Figure 9, Quora Native mobile app uses WebView to render the necessary text content while handling the complex features with the native app.

WebView has many use-cases, one of the popular being advertising; this makes it easy for apps to serve timely ads to users straight from the ad-server.

2.6.3 Chrome Custom Tabs

Chrome custom tabs are also another option to show web content in native applications, it was introduced on chrome version 45, but now it is supported by almost all android browsers. (Googledevelopers, 2020)
like WebView, it displays web content, but it will not be full screen (browsers' address bar will be visible). Nevertheless, unlike WebView, it gives developers the freedom to customize the UI of the display screen by changing the toolbar color, adding enter-exit animations, and add custom actions to the Chrome toolbar, overflow menu, and bottom toolbar. (Kinlan, 2016)
In addition to making design changes, Custom tabs support the prefetching of content, resulting in faster loading and seamless transition to the content screen. According to Kinlan, by prefetching content, the loading speed can be minimized by 700 ms; this is achieved by connecting to the URL beforehand. (Kinlan, 2016)

2.6.4 Why use CCT over WebView?

CCT supports the full features of the browser, which WebView does not have.
Cookie Jar - WebView cannot access cookie information outside the app, while CCT uses a shared cookie jar allowing sign in and permission features to be automatic.
UI customizations - provides developers with the tools to customize Toolbar color, Action button, Custom menu items, Custom in/out animations, and the bottom toolbar
Update - updating CCT is quite simple; users do not need a separate way to update the features of the customer tab; all it takes is updating the browser.
Security - because CCT has full features of the browser, it takes advantage of the browsers' safety features, protecting the user from unsafe sites.
Performance – CCT enables pre-rendering to provide contents to be fetched in advance, thereby decreasing the load time.

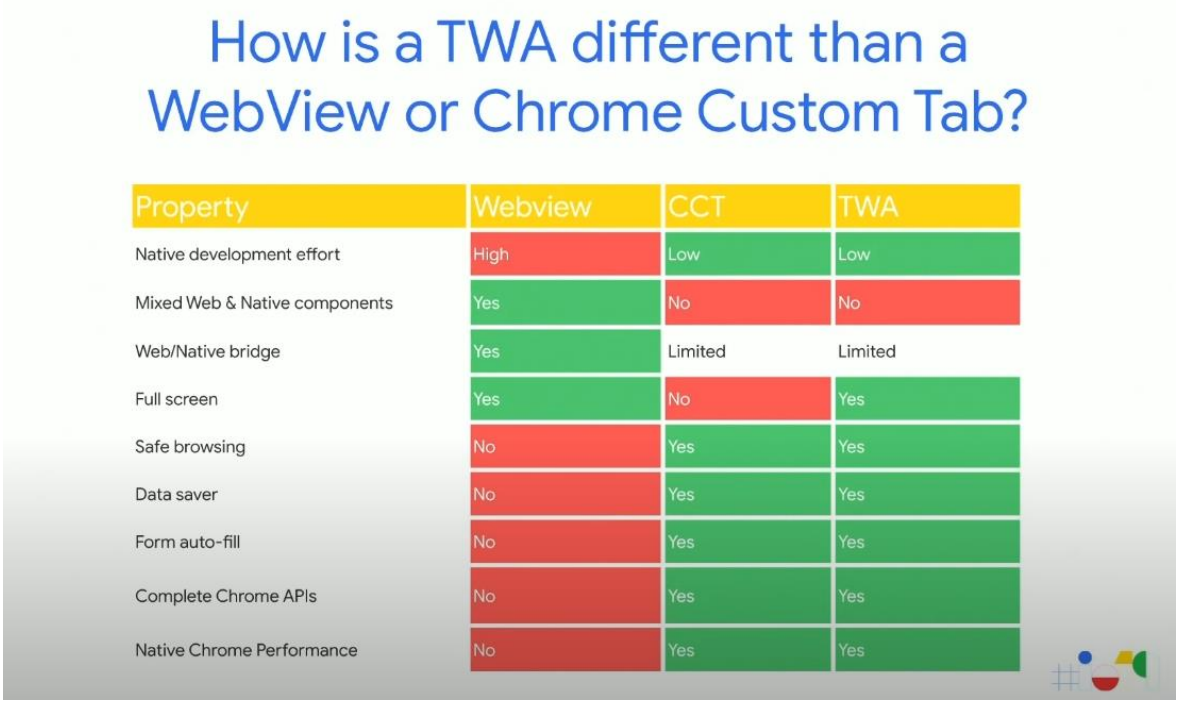


Figure 10 Comparison for Webview, CCT, and TWA

As shown in Figure 10, using the power of the browser, TWA provides more features as compared to Webvies and CCT. Although TWA is similar to CCT, it's the ability to show full-screen content that makes it a good option for a standalone mobile application.

2.7 Web Performance Optimization

Web performance refers to how quickly site content loads, render in web browsers, and how well it responds to user interaction. (contributorsMDN, 2020)

Web performance is a key aspect of user experience; fast-loading sites generate more users, while slow loading sites frustrate users and takes away already existing ones. Especially at this time, where most web activity is done through mobile devices, makes performance very important.

As apparent it is, mobile devices are susceptible to slow connection, but despite this fact, it is the developers' responsibility to make sites that are accessible no matter the connection speed.

According to research done by SOSTA, an American software testing company, web performance was found to be at the center of user experience, business value, and technical metrics; also, the research has made the following key findings. (Everts, 2016)

- 46% of online shoppers responded that shopping on mobile devices to be their least favorite experience.
- Over 3G networks, on average, the load time for mobile devices is 19 seconds.
- 53% of mobile site users leave if the page does not load in under 3 seconds.

- Mobile pages that load under 5 seconds have a 70% more session rate than sites loading above 19 seconds and double the revenue.
- Almost half of the request by servers are related to ads

In addition, case studies done by (wpostats, 2020), have shown that many companies who optimized their site performance have increased their conversion rates, decreased bounce rates, and increased their revenue.

- snipesUSA.com has doubled its conversion rate by 1% after a 30% increase in page loading.
- Rossignol.com have increased their conversion rate by 94% after improving load time by1.9 seconds and speed index by a factor of 10.
- Radins.com also saw a 12% conversion increase after improving the speed index by 51%

There are many success stories related to improved performance that can be found on wpostats, which prove that WPO(web performance optimization) affect business and customer retention; one of the reasons is that performance is one of the key indicators in SEO(search engine optimization), slower loading results in lower site ranking by search engines which lead to fewer users visiting it. (Everts, 2016)

2.7.1 WPO testing test tools

Performance testing is one of the fundamental processes during the development of a web application; the results provide information on how fast the browser loads resources for initial page load over certain connections. Few metrics determine how this is calculated, such as First Contentful Paint, Speed Index, Largest Contentful Paint, Time to Interactive, Total Blocking Time, and Cumulative Layout Shift.

There are many testing tools available online that provide the above-listed results, but Google Lighthouse was found to be the best option; one of the reasons is that it supports tests over local development environment, also lists opportunities to optimize lower performance scores.

First Contentful Paint: First Contentful Paint marks the time at which the first text or image is painted.

Speed Index: Speed Index shows how quickly the contents of a page are visibly populated

Largest Contentful Paint: Largest Contentful Paint marks the time at which the largest text or image is painted.

Time to Interactive: Time to interactive is the amount of time it takes for the page to become fully interactive.

Total Blocking Time: Sum of all time periods between FCP and Time to Interactive, when task length exceeded 50ms, expressed in milliseconds.
Cumulative Layout Shift: Cumulative Layout Shift measures the movement of visible elements within the viewport.

Performance Optimization Solutions will be better explained in the implementation section with the practical results as an example

3 Implementation

In this section, the technical overview of creating a React PWA starting from the initial development of the application, Progressive enhancements, packaging in TWA, and deployment on Google store will be discussed.
The PWA developed in this work was named Cinkino; it is a React App that uses Finnkino API to fetch data from the server and shows available movies in certain cinema locations. The basic setup of the application is created with CRA(create-react-app), a template that is developed by Facebook; it provides the basic features for creating a single page application. Redux JavaScript library will be used to handle the application state; this will help to avoid prop drilling and provide an organized place for storing the applicationlevel states. "Prop drilling (also called "threading") refers to the process you have to go through to get data to parts of the React Component tree." (Dodds, 2018)
In addition, few additional 3rd party dependencies will be used to implement the planned functionalities; these dependencies will be downloaded using NPM (node package manager).

3.1 Features requirement

Most of the features that are planned for this application are dependent on the Finnkino API. It is a public API available on the Finnkino website, available at https://www.finnkino.fi/xml/; requests sent to this API returns an XML formatted response.
However, the search and bookmarking functionalities are not part of it; therefore will need additional logic coded into the app.
The application will also use Google Map API to locate the closest Finnkino cinema. Although the map is not provided by Finnkino API, it will use the area property to fetch and place location data on Google Map.
To summarize, the following features will be available in the Cinkino App

- Searching movies available in the Finnkino Cinemas
- Filtering movies based on location and/or date
- Bookmarking movies for later viewing: After browsing through the list of movies,
- Delete movies in the bookmarks
- Check the location of theatres nearby
- Check out news from Finnkino

3.2 Development environment

Software	Version	Hardware	
OS	Windows x64 10.0.18363	Processor	AMD_Phe-nom(TM)_II_X3_B75
VScode	1.50.1	RAM	4GB
Chrome Browser	83.0.4103.122		
Node.js	12.14.1		
NPM	6.14.8		
Lighthouse	6.0		
Firebase CLI	8.16.0		

3.3 Create-React-App

Create-react-app is a React application starter template; it provides the basic front end environment with the latest JavaScript features. It comes with NPM for downloading third party packages, Webpack for bundling, and Babel to compile ECMAScript 2015.

Babel compiles a new version of JavaScript to older versions of JavaScript so applications can work on older browsers.

CRA also provides start, build, test, and eject scripts, which will come in handy for development and production, but during the development of this application, start and build scripts will be used for running the application and bundling the production version.

```
{
  "name": "cinkinoweb",
  "version": "0.1.0",
  "private": true,
  "dependencies": {
    "@testing-library/jest-dom": "^5.11.5",
    "@testing-library/react": "^11.1.0",
    "@testing-library/user-event": "^12.1.10",
    "react": "^17.0.1",
    "react-dom": "^17.0.1",
    "react-scripts": "4.0.0",
    "web-vitals": "^0.2.4"
  },
  "scripts": {
    "start": "react-scripts start",
    "build": "react-scripts build",
    "test": "react-scripts test",
    "eject": "react-scripts eject"
  },
```

Listing 6 Create-react-app dependencies and scripts

But before going forward on the development, it is a good practice to check the performance level we are starting with, especially for the development of this application, mainly because of Googles' TWA requirement that a web applicationshould be 100% PWA with a performance of 80 minimum. For this purpose, Lighthouse testing tool was found to be a perfect tool; it can be installed as a chrome extension to analyze pages; as discussed in the above section, Lighthouse calculates five basic benchmarks, Performance, Accessibility, Best Practices, SEO, and Progressive Web App.

During initial testing, the scores were fluctuating when the test was done on normal browser mode and incognito mode; this is mainly due to the negative effect of extensions that were installed in the browser.

As seen in figures 11 and 12, there is a 28-point difference in the performance index between these two modes.

Figure 11 CRA development version Lighthouse score on Incognito mode

Figure 12 CRA development version Lighthouse score on Normal mode

The production build, however, is much faster than the development version of the application; CRA has a built-in Webpack bundler which builds a minified bundle of the application, thereby increasing the performance; as seen in Figure 13, the performance increased to 99 when tested in Incognito mode.

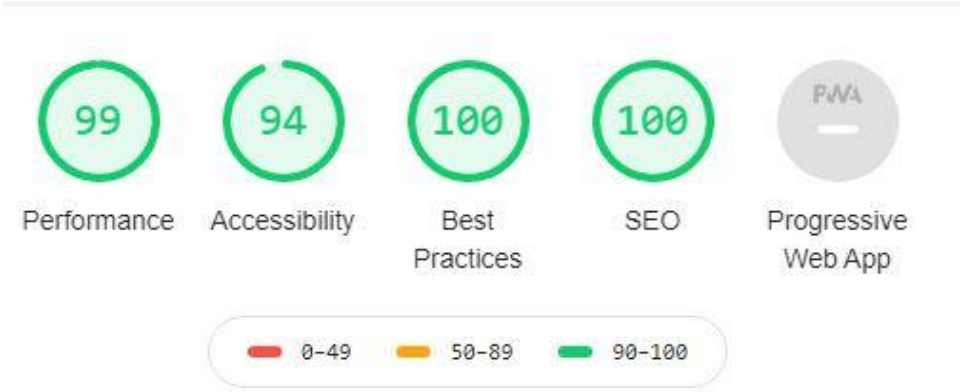


Figure 13 CRA production version Lighthouse mobile score

Chrome dev tools can be used to check the size and see how much bundling saves resources of the resources sent to the browser; by going into the network tab of inspect mode, it is possible to see the resources that the page loaded in the case of CRA, the browser was loading 1.8 Mb of resources before bundling, but after, that size decreased to 139Kb.

3.4 Cinkino React App

Although the details for the development of this application are out of scope for this work, the dependencies used will significantly affect the performance of the app; therefore, it will be discussed briefly in this section.

Additional dependencies and libraries were needed to implement some of the features proposed in section 3.1 to increase the development speed and make the application more responsive for mobile devices. Besides the dependencies shown in Listing 7, Material Design Lite(MDL) was used for styling the application; by using this library, it was possible to develop responsive pages that interact well on mobile devices, although this approach had created a strain on the page load.

```
"dependencies": {
      "@react-google-maps/api": "^1.13.0",
      "axios": "^0.19.2",
      "fast-xml-parser": "^3.16.0",
      "react-autosuggest": "^10.0.2",
      "react-datepicker": "^3.3.0",
      "react-redux": "^7.1.3",
      "react-router-dom": "^5.1.2",
      "redux": "^4.0.5",
      "redux-promise": "^0.6.0",
  },
```

Listing 7 3rd party packages and libraries used

@react-google-maps/api: used for loading Google Maps API
Axios: is a promise-based HTTP client library
fast-XML-parser: used for converting XML data formats to JSON
react-autosuggest: provides automatic search suggestions based on the available data
react-datepicker: a full date and time picking solution
react-router-dom: a routing solution for the components inside the app
redux: is an open-source library that is used for managing application-level state
redux-promise: is an Asynchronous middleware that helps in passing promises inside an action object.

3.5 Cinkino Progressive Web App

To meet the requirements of TWA deployment, the react application build has to be enhanced to become Progressive. According to Lighthouse, for an application to get 100 on PWA metrics, it has to be fast, reliable, installable, and PWA optimized. Sub-Metrics are available under these results and can be found in the Lighthouse tab. As discussed in section 2.5, the first task would be creating Web Manifest that tells the browser how to install the application by providing a banner icon, name, and other properties.
According to Lighthouse metrics, Web Manifest should at least include name, icons with 192x192 px and a 512x512 px, start_url, display, and prefer_related_applications properties.

```
{
  "short_name": "CinkinoWeb",
  “name”: “Cinkion Cinema APP”,
  "icons": [
    { "src": "logo192.png",
      "type": "image/png",
      "sizes": "192x192",
      "purpose": "any maskable"},
    { "src": "logo512.png",
      "type": "image/png",
      "sizes": "512x512"} ],
  "start_url": "/",
  "display": "standalone",
  "theme_color": "#000000",
  "background_color": "#ffffff",
  "lang": "English",
  “description”: “React App to check Finnkino cinema schedule”,
  "scope": ".",
  "orientation": "any"
}
```

Listing 8 Web App Manifest for Cinkino

As per the Manifest shown in Listing 8, the browser will check start_url to display the front of the application with the scope of the entire page; after the Manifest has been created, it can be referenced in the index.html file as a link in such a way as shown in Listing 9.

```
<link rel="manifest" href="%PUBLIC_URL%/manifest.json" />
```

Listing 9 Adding Web manifest

The successful integration of the web manifest can be seen in the browsers inspect mode under the Application tab.

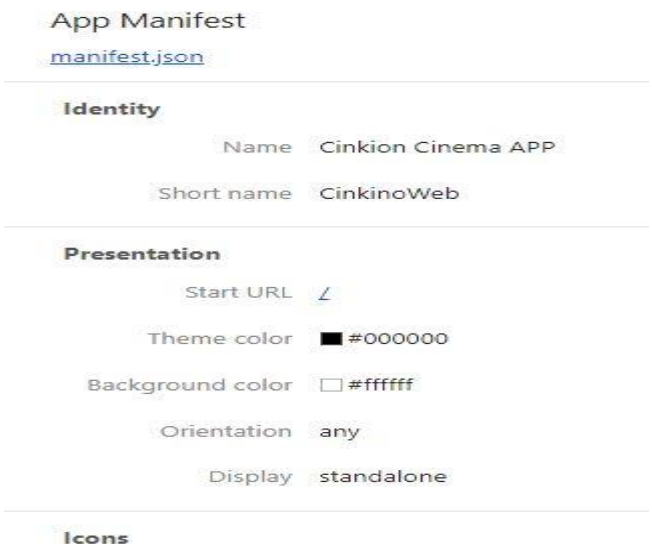


Figure 14 Web Manifest installed successfully

3.5.1 Implementing Service Worker

Service worker enhance web applications to work offline, as mentioned in section 2.5.5, It runs in the background separate from the regular JavaScript; it is like a proxy server which listens to events going in and out of the client.

A service worker can be manually created by writing the JavaScript code in the root folder, enhancing it to listen to specific events and cache required files and pages.

Another option to writing service worker from scratch is by using Workbox, “It is a collection of libraries and tools used for generating a service worker, precaching, routing, and runtime-caching.” (developersGoogle, 2020)

To make development faster, PWABuilder auto-generated Workbox will be used to cache the files of the page. PWABuilder generates the required Workbox with seven options, Offline copy of pages, Offline copy with Backup offline page, Cache-first network, Advanced caching, Background Sync, and Serving Cached media.

Each caching solution can be customized to generate Workbox with the required functionality; for this project, however, Advance Caching was chosen; this solution provides good customization options to cache HTML, JS, Stylesheets, Images, and Fonts, also, the maximum amount of files and how long files should be cached can be customized.
Although it is preferable to cache all the files in the app, this will create an additional problem by taking too much cache storage, therefore for Cinkino App, Images caching will not be available.
There are two types of caching strategies used in this solution, NetworkFirst and StaleWhileRevalidate, the former allows the Service Worker to fetch a response directly from the network; if the request is successful, it will cache the files and make them available when there is a failed request. The latter, however, responds with the cached files by default and requests from the network if they are not in the cache.
StaleWhileRevalidate is preferable if the Apps are not updated quickly, while NetworkFirst is the ideal solution for Apps that require fast updates. (developersGoogle, 2020)
The code shown on Listing 10 will be added in the index.html file; this code will register the Service Worker, to integrate the service worker update when there is new content available and notify users when the pages are ready to be used offline.

```
<script type="module">
  import 'https://cdn.jsdelivr.net/npm/@PWABuilder/pwaupdate';
    const el = document.createElement('pwa-update');
    document.body.appendChild(el);
</script>
```

Listing 10 Adding Service Worker

Once the service worker is available in the root of the app, files that will be precached can be assigned using the code in Listing 10; this code is stored in its JavaScript file by default, it is named PWABuilder-sw.js, this is the file that the code in Listing 10 will check to register the ServiceWorker therefore, renaming it will not work.
The code in Listing 11 uses NetworkFirst strategy to pre-cache all the HTML files in the App, the files will be stored for a day and the maximum limit is 10 HTML pages, this can be adjusted as needed.

```
workbox.routing.registerRoute(
 ({event}) => event.request.destination === 'document',
 new workbox.strategies.NetworkFirst({
  cacheName: HTML_CACHE,
  plugins: [
   new workbox.expiration.ExpirationPlugin({
    maxAgeSeconds: 1 * 24 * 60 * 60,
    maxEntries: 10,}),
  ],
```

```
  })
);
```

Listing 11 NetworkFirst caching strategy using Workbox

3.6 Web Performance Optimizations

After developing the Cinkino react application with the functionality requirements, the Lighthouse performance results were not enough to deploy with TWA, the first problem is that it is not a PWA, and secondly, the performance result was not enough to meet the requirements set for deployment on Google store.

Figure 15 Cinkino Application Lighthouse score before optimization

According to Figure 15, the application needs to fulfill the PWA requirement and increase the performance level by 20%, but this is not the only result delivered by Lighthouse; it also provides suggestions that can help improve all the test metrics.

3.6.1 Compression

Compression is a way to build a bundled version of the application in a way to optimize the code in the served files. CRA uses Gzip in its webpack configuration to compress the build files; while this is a useful compression tool, I have found Brotli performs better with compression. Figure 17 shows that Brotli provides faster and more optimal compression than other compression tools such as Gzip and deflate.
Although it is possible to build a webpack configuration which utilizes Brotli aside from the one provided by CRA, that requires additional expertise with webpack.
The easiest way to implement this compression tool was to deploy using Firebase; as of Aug 2020, Firebase uses Brotli to compress files hosted on its server. (Deng, 2020). The tool used to compress files can be checked in the Network tab under header response in

the content-encoding property either as gzip or br.

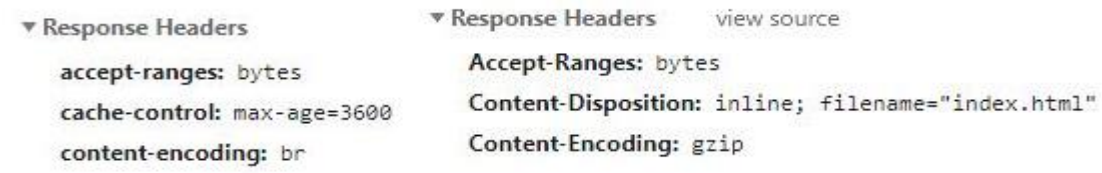


Figure 16 Content-encoding compression properties

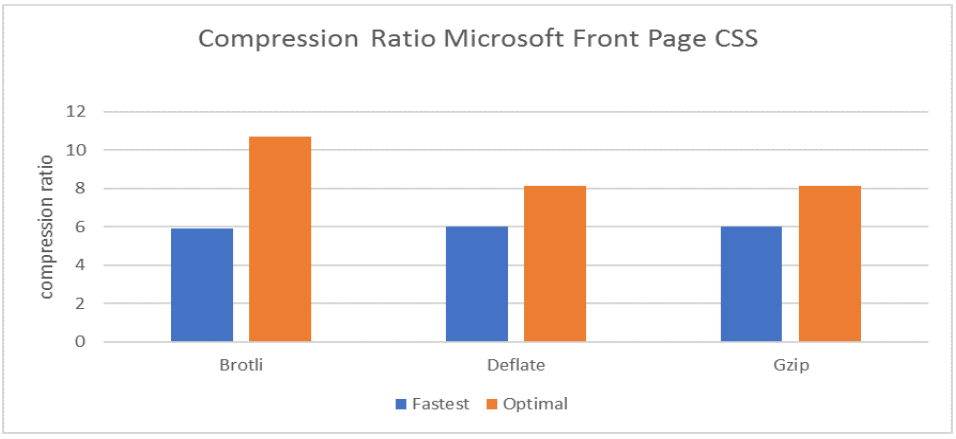


Figure 17 Compression tools comparison (magenx, 2020)

After creating an account on Firebase, the project build was hosted on its server; with additional compression by Brotli, the Lighthouse performance score has increased by 12% and reached 73.

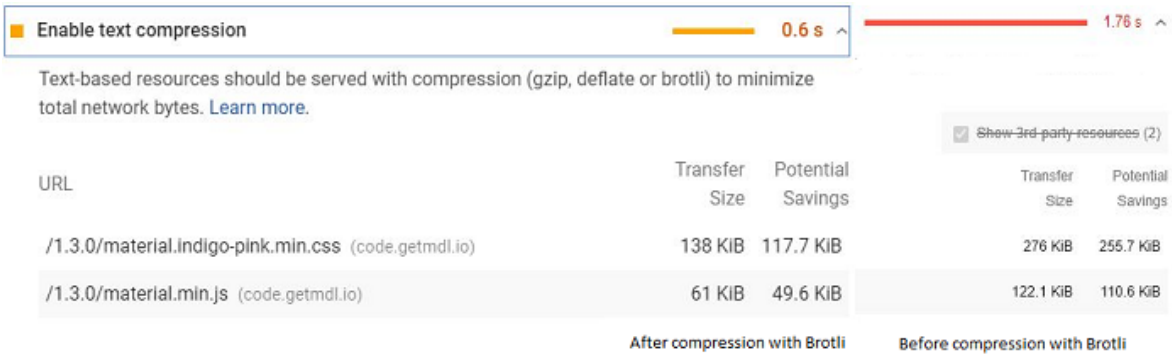


Figure 18 Before and After Firebase Hosting

3.6.2 Code Splitting

Code splitting is a way to send JavaScript files that are essential to the required page load. Normally, the client loads all the JavaScript whether it is needed for the specific page or not; for example, when a user navigates to a web application with a login page,

the browser doesn't know if the user will log in successfully, but will still load the entire JavaScript for the whole application. This approach takes unnecessary loading and parsing time. (Aakash, 2020)

By default, CRA Webpack supports code-splitting using lazy loading and Suspense. Using lazy loading, it is possible to load and run JavaScript code that is needed by a specific component without loading the whole JavaScript code for the entire app.

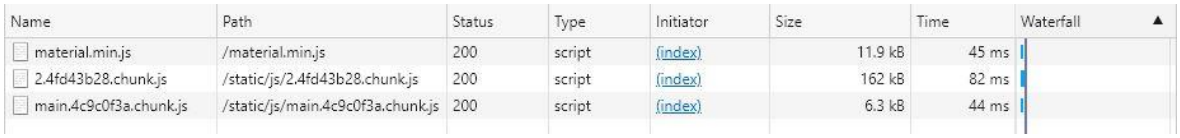
Lazy loading uses a promise that when the specified component is available, it will be imported; if many components need to be loaded, they will have to wait until the promise is resolved, therefore when webpack bundles this app, it creates smaller chunks of JavaScript for each component instead of large size bundle and Suspense provides a fallback during the asynchronous call until it has been resolved and the promise has been returned then it wraps the components that will be returned when resolved. (Aakash, 2020)

Figure 19 shows the bundled JavaScript file, which is fetched when the Application loads, and no matter the user navigates to different pages, the JavaScript loaded will stay the same because the whole application is bundled into these three chunks of large size JavaScript code.

In the case of Figure 20, however, the JavaScript code loaded decreases in size, and each component loads its JavaScript code when it is navigated; the splitting minimized the size from 162kb to 145kb; also, the performance score of the application increased from 73 to 76.

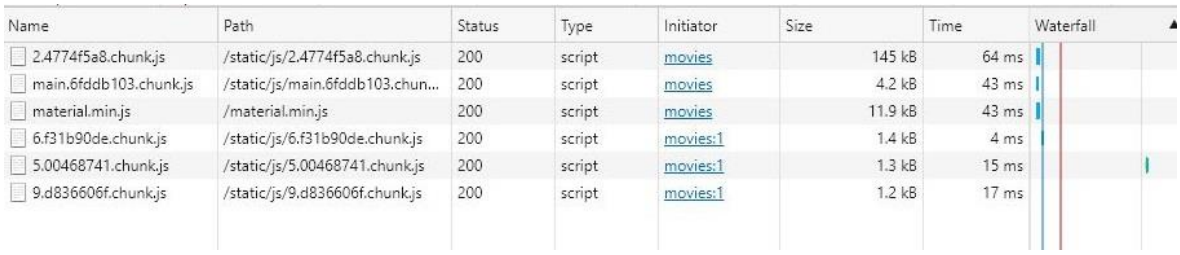
This difference, however, would be much more visible on large scale applications where there are many components.

This can be seen in dev tools Network tab with JS filter on.



Name	Path	Status	Type	Initiator	Size	Time	Waterfall
material.min.js	/material.min.js	200	script	(index)	11.9 kB	45 ms	
2.4fd43b28.chunk.js	/static/js/2.4fd43b28.chunk.js	200	script	(index)	162 kB	82 ms	
main.4c9c0f3a.chunk.js	/static/js/main.4c9c0f3a.chunk.js	200	script	(index)	6.3 kB	44 ms	

Figure 19 Non-Lazy loaded App



Name	Path	Status	Type	Initiator	Size	Time	Waterfall
2.4774f5a8.chunk.js	/static/js/2.4774f5a8.chunk.js	200	script	movies	145 kB	64 ms	
main.6fddb103.chunk.js	/static/js/main.6fddb103.chun...	200	script	movies	4.2 kB	43 ms	
material.min.js	/material.min.js	200	script	movies	11.9 kB	43 ms	
6.f31b90de.chunk.js	/static/js/6.f31b90de.chunk.js	200	script	movies:1	1.4 kB	4 ms	
5.00468741.chunk.js	/static/js/5.00468741.chunk.js	200	script	movies:1	1.3 kB	15 ms	
9.d836606f.chunk.js	/static/js/9.d836606f.chunk.js	200	script	movies:1	1.2 kB	17 ms	

Figure 20 Code splitting with React lazy

3.6.3 Upgrading non-secure HTTP requests

One of the problems faced during testing on other browsers was that the initial page load was done over a secure HTTPS protocol but navigating to other pages request content over non-secure HTTP protocol; this problem was only visible on other browsers such as edge and opera while working well on Chrome. After some research, it was found that the images requested from the Finnkino API were using HTTP instead of the secure option, this can be seen in Figure 21, but by default, Chrome browser was upgrading this request to HTTPS.

By adding the Content-Security-Policy meta tag shown in Listing 18, other browsers can also be told to upgrade the requests to HTTPS, thereby avoiding non-secure requests.

```
<meta http-equiv="Content-Security-Policy" content="upgrade-insecure-requests">
```

Listing 12 Content security Policy upgrade

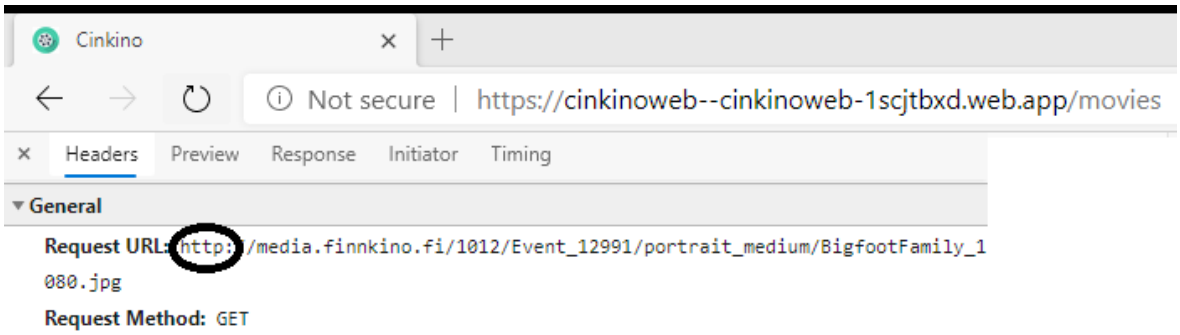


Figure 21 Non-secure HTTP requests

3.6.4 Responsive Images

Images are non-blocking resources, unlike CSS and JavaScript, but they are bigger in size, and fetching them takes more time for the browser. For faster networks, it might not be a problem to load large-size images, but for small screen devices, it is pointless to load high pixel images that take a long time to load, which affects the page load time with full content.

Typically, images are rendered when the browser finds the <img> tag with <src> property, and it loads the image from the source, using srcset; however, the browser can load images based on the screen size parameters defined.

As shown in Listing 13, when the width of the screen is 1200w, the large image with a bigger size will be loaded, while it is 600w and lower medium-size versions of the image will be loaded; these breakpoints can be defined on any screen width but will fall back to the src if the breakpoint is not defined.

With this strategy, it is possible to avoid strains on small screen devices with unnecessarily large size images, saving larger network requests, thereby increasing performance.

```
<img
srcset={`${largeImg.jpg} 1200w,
        ${mediumImg.jpg} 600w,
        ${smallImg.jpg} 400w,`}
width="100%" height="100%"
src={large}
alt="responsive image"/>
```

Listing 13 Responsive Images using srcset

3.6.5 Efficient HTTP cache policy

“All HTTP requests that the browser makes are first routed to the browser cache to check whether a valid cached response can be used to fulfill the request. If there is a match, the response is read from the cache, eliminating both the network latency and the data costs that the transfer incurs.” (Posnick & Grigorik, 2020)

On this project application, the cached resources were being stored for a maximum of 24 hours; therefore, it was flagged by Lighthouse as “Serve static assets with an efficient cache policy.” By changing the response header in the server, in this case, Firebase, the amount of time the cache is stored has been increased to a week.

The result can be seen in the Network tab under the “Headers” property. For example, as shown in Figure 22, material.js is being served from the memory cache of the browser, and it will be stored for a week. Although this has not shown an increase in the performance score, on repeat visits, it will be more visible to the users.

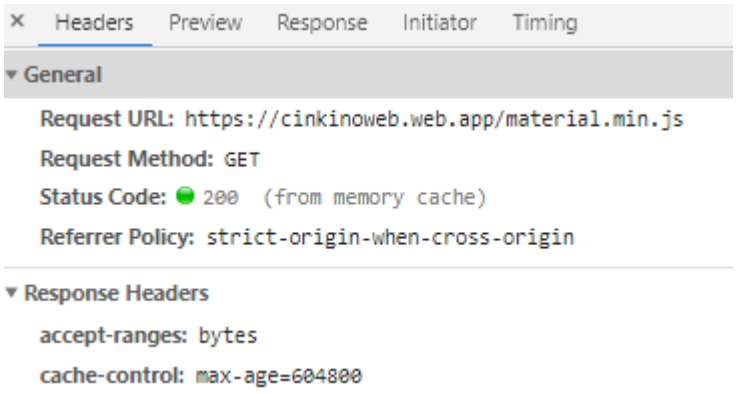


Figure 22 HTTP cache policy for material.js

3.6.6 Publishing Cinkino on Google Play

Now that the React PWA has been optimized to achieve the 80 scores on Lighthouse, it is ready to be wrapped with TWA to be deployed on Google play; one easy way I found during my research is to use Microsoft PWABuilder.

After providing this service with the link of the hosted web App, it generates functional versions that can be deployed on major application stores such as Google Play, Samsung Galaxy, Microsoft, and macOS stores. After submitting the hosted web application link, it checks the Application for Manifest and Service worker to make sure it conforms to PWA requirements.

If the necessary requirements are met, PWABuilder will provide an option to package the application for the above-mentioned stores.

As shown in Figure 23, the tool calculates the score based on the availability of Manifest, Service Worker, and security.

Figure 23 PWABuilder results for https://cinkinoweb.web.app/

As shown in PWABuilder, packaging options cover the major application stores to make applications discoverable by all platform users. In this case, however, the scope of this thesis is to publish PWA on Google Play Store; therefore, the Android option will be used by default; this option uses Bubblewrap to package applications with TWA.

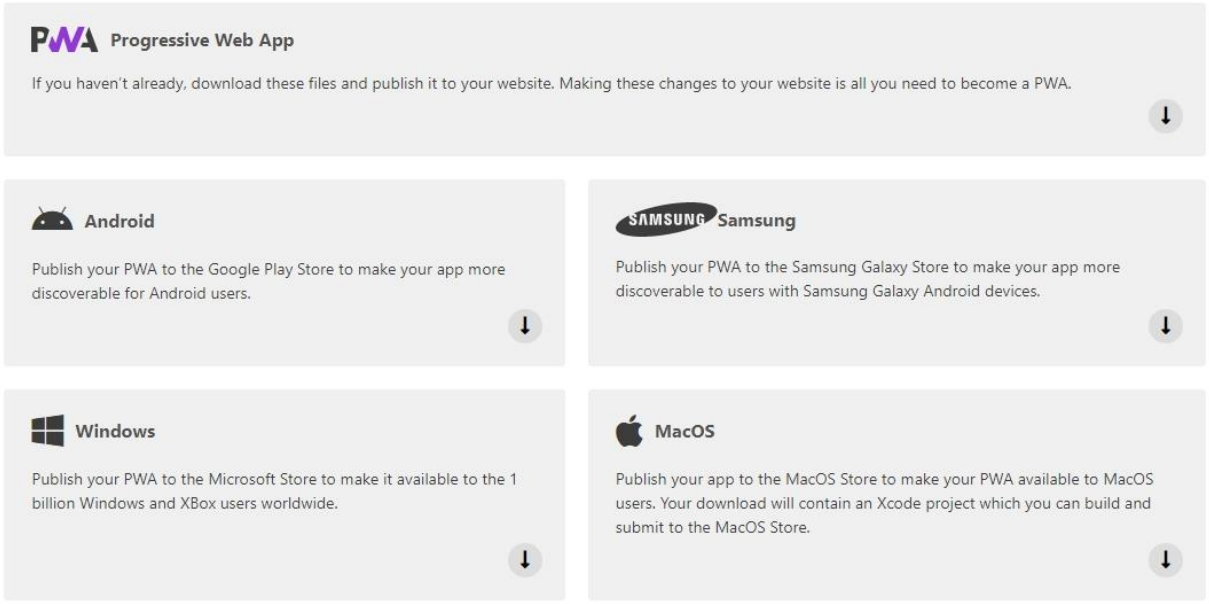


Figure 23 PWABuilder packaging options for major stores

The Android packaging option also provides many properties to customize the features that will be available once it is generated. A few of the important properties include the following.

Fallback behavior: a property to assign a fallback when TWA experience is not working; therefore, it will use CCT and WebView as a fallback option, with CCT being the default.

ChromeOS only: if enabled**,** this property restricts the use of other devices except for ChromeOS

Signing key: this property provides an option to use the already available signing-key, generate a new key, or not use the key at all. Using the “Use mine” option, a new version of the application can be created for continuous updates.

Include source code: if enabled, the final generated zip file will include a Java android source code for the application that can be customized with android studio.

Most of the information in the options form is already prefilled because it uses information from the Manifest by default.

As shown in Figure 24, Once the TWA wrapped version of the application is generated, it contains six files. assetlinks.json contains “sha256_cert_fingerprints” information that proves the PWA owner is also the one responsible for the Android package.

Signing keys provided in the generated file will be useful when a new version of the application is published; therefore, it needs to be stored securely.

CinkinoWeb.aab is a format that will be published to the application store, although the APK version can also be used.

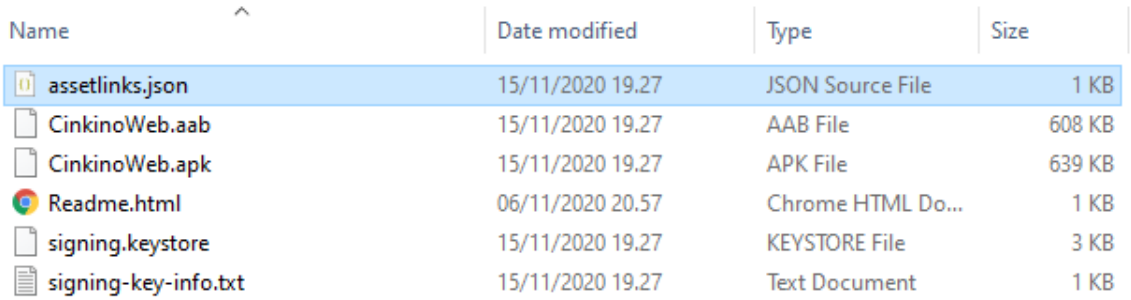


Figure 24 PWABuilder generated android package

As seen in Figure 24, the packaged version of the Cinkino app is less than 700kb, which is impossible to achieve if the app is native.

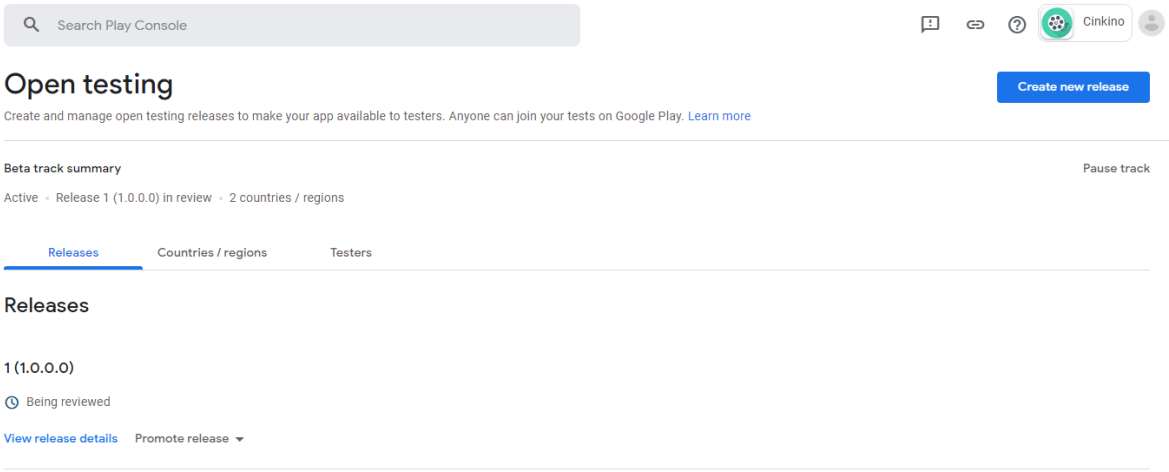


Figure 25 Cinkino release overview on Google play console

The signed version of the application was uploaded on Google Play Store; after filling the required forms confirming Cinkino abide by the store rules, It was submitted for publishing. But because of the store rules, it is currently in the process of review by the Store team. As seen in Figure 25, Cinkino is will be available in Google Play Store once the review process is approved by the store team.

Discussion

In conclusion, the combination of PWA with TWA has a lot of potential for growth.
With increased browser support and active interest from the developing community, the writer believes these technologies can help small companies publish mobile applications easily, and for web developers, this approach highlights the knowledge they already possess and would be a good addition to their skill set.
However, due to the TWA requirements for PWAs to achieve a Lighthouse mobile performance score of 80 minimum, the performance optimization process is a lot of work.
A final performance score of 84 has been reached after extensive optimizations and research. The writer believes it is possible to increase this score by avoiding third-party libraries that request resources over the network.
The use of PWABuilder is highly recommended; it generates full PWA features such as Manifest and service worker as needed; in addition, it generates TWA wrapped android package that is production-ready, therefore saves a lot of time and coding.
React code-splitting features using React lazy were very useful during the performance optimization process.
The publishing of the Cinkino application was not finalized, it was released for open testing on Google Play, but due to the review process that is required, the application is not yet available in the app store. However, the test is done with the APK installed version ran full screen with all the features on a personal mobile device.
The final version of the web application can be found on https://cinkinoweb.web.app/

4 References

Aakash, 2020. *Effective Code Splitting in React: A Practical Guide.* [Online]
Available at: https://hackernoon.com/effective-code-splitting-in-react-a-practical-guide-2195359d5d49
[Accessed 25 10 2020].

Bar, A., 2020. *WHAT WEB CAN DO TODAY?.* [Online]
Available at: https://whatwebcando.today/
[Accessed 15 10 2020].

Codecademy, 2020. *React: The Virtual DOM.* [Online]
Available at: https://www.codecademy.com/articles/react-virtual-dom
[Accessed 12 10 2020].

contributorsMDN, 2020. *The "why" of web performance.* [Online]
Available at: https://developer.mozilla.org/en-US/docs/Learn/Performance/why_web_performance
[Accessed 15 10 2020].

Deng, K., 2020. *New for Firebase Hosting: request logging, Brotli compression, and internationalization.* [Online]
Available at: https://firebase.googleblog.com/2020/08/firebase-hosting-new-features.html
[Accessed 20 10 2020].

Developers, G. C., 2019. *Android WebView 101.* [Online]
Available at: https://www.youtube.com/watch?v=qMvbtcbEkDU&t=560s&ab_channel=GoogleChromeDevelopers
[Accessed 04 10 2020].

DevelopersGoogle, 2019. *Introduction to Service Worker.* [Online]
Available at: https://developers.google.com/web/ilt/pwa/introduction-to-service-worker
[Accessed 04 10 2020].

developersGoogle, 2020. *Lab: Workbox.* [Online]
Available at: https://developers.google.com/web/ilt/pwa/lab-workbox
[Accessed 23 10 2020].

Deveria, A., 2020. *Service Workers.* [Online]
Available at: https://caniuse.com/?search=service%20workers
[Accessed 29 09 2020].

Dodds, K., 2018. *Prop Drilling.* [Online]
Available at: https://kentcdodds.com/blog/prop-drilling
[Accessed 20 10 2020].

Everts, T., 2016. *Mobile Load Time and User Abandonment.* [Online]
Available at: https://developer.akamai.com/blog/2016/09/14/mobile-load-time-user-abandonment
[Accessed 20 10 2020].
Fedosejev, A., 2015. *React.js Essentials.* s.l.:Packt Publishing.
Geddes, D., 2019. *Service worker mindset.* [Online]
Available at: https://web.dev/service-worker-mindset/
[Accessed 30 09 2020].
Google I/O. 2019. [Film] Directed by Andre Bandarra Peter McLachlan. USA: Google.
Googledevelopers, 2020. *Custom Tabs.* [Online]
Available at: https://developers.google.com/web/android/custom-tabs
[Accessed 06 10 2020].
Hoober, S., 2018. *Mobile Apps: Native, Hybrid, and WebViews.* [Online]
Available at: https://www.uxmatters.com/mt/archives/2018/08/mobile-apps-native-hybrid-and-webviews.php
[Accessed 06 10 2020].
Kagga, J., 2018. *Understanding React Components.* [Online]
Available at: https://medium.com/the-andela-way/understanding-react-components-37f841c1f3bb
[Accessed 11 10 2020].
Kinlan, P., 2016. *Chrome Custom Tabs.* [Online]
Available at: https://developer.chrome.com/multidevice/android/customtabs#whentouse
[Accessed 06 10 2020].
Kirupa, 2019. *Understanding WebViews.* [Online]
Available at: https://www.kirupa.com/apps/webview.htm
[Accessed 05 11 2020].
Laaksonen, 2018. *vivecho.* [Online]
Available at: https://vivecho.com/cost-of-mobile-development/
[Accessed 30 09 2020].
LePage, P., 2020. *Add a web app manifest.* [Online]
Available at: https://web.dev/add-manifest/
[Accessed 24 10 2020].
Love, C., 2019. *What Browsers Support Service Workers?.* [Online]
Available at: https://love2dev.com/blog/what-browsers-support-service-workers/
[Accessed 02 10 2020].
magenx, 2020. *Brotli Compression - build nginx dynamic module - magento static optimization.* [Online]
Available at: https://www.magenx.com/blog/post/brotli-compression-build-nginx-dynamic-

module-magento-static-optimization.html
[Accessed 20 10 2020].
MDNcontributors, 2019. *Populating the page.* [Online]
Available at: https://developer.mozilla.org/en-US/docs/Web/Performance/How_browsers_work
[Accessed 24 10 2020].
MDNcontributors, 2020. *Service Worker API.* [Online]
Available at: https://developer.mozilla.org/en-US/docs/Web/API/Service_Worker_API
[Accessed 02 10 2020].
MDNcontributors, 2020. *Using Service Workers.* [Online]
Available at: https://developer.mozilla.org/en-US/docs/Web/API/Service_Worker_API/Using_Service_Workers
[Accessed 03 10 2020].
Nikhil, 2020. *13 Best JavaScript Framework For 2020.* [Online]
Available at: https://www.lambdatest.com/blog/best-javascript-framework-2020/
[Accessed 10 10 2020].
Perera, P., 2020. *A Comparison: Native Apps vs Web Apps.* [Online]
Available at: https://www.technocrat.com.au/blog/comparison-native-apps-vs-web-apps
Posnick, J. & Grigorik, I., 2020. *Prevent unnecessary network requests with the HTTP Cache.* [Online]
Available at: https://web.dev/http-cache/
[Accessed 15 11 2020].
Rabouw, R., n.d. *pwa-fundamentals.* [Online]
Available at: https://pwa-fundamentals.nl/chapters/service-workers.html#service-worker-lifecycle
[Accessed 01 10 2020].
RAJ, V., 2020. *Trusted Web Activities — II.* [Online]
Available at: https://medium.com/groww-engineering/trusted-web-activities-ii-2dbd6d5d6e90
[Accessed 18 10 2020].
reactenlightenment, 2020. *What Is JSX?.* [Online]
Available at: https://www.reactenlightenment.com/react-jsx/5.1.html
[Accessed 11 10 2020].
ReactJs, 2020. *Introducing Hooks.* [Online]
Available at: https://reactjs.org/docs/hooks-intro.html
[Accessed 11 10 2020].

ReactJsOrg, 2020. *React – A JavaScript library for building user interfaces.* [Online]
Available at: https://reactjs.org/
[Accessed 10 10 2020].
Saccomani, P., 2019. *People Spent 90% of Their Mobile Time Using Apps in 2019.* [Online]
Available at: https://www.mobiloud.com/blog/mobile-apps-vs-the-mobile-web
[Accessed 15 10 2020].
Saccomani, P., 2020. *Native Apps, Web Apps or Hybrid Apps? What's the Difference?.* [Online]
Available at: https://www.mobiloud.com/blog/native-web-or-hybrid-apps
ScandiPWA, 2019. *History of Progressive Web Apps.* [Online]
Available at: https://medium.com/progressivewebapps/history-of-progressive-web-apps-4c912533a531
[Accessed 10 10 2020].
ScandiPWA, 2019. *History of Progressive Web Apps.* [Online]
Available at: https://medium.com/progressivewebapps/history-of-progressive-web-apps-4c912533a531
[Accessed 10 14 2020].
Scott, E., 2015. *SPA Design and Architecture: Understanding Single Page Web Applications.* s.l.:Manning Publications.
Shah, H., 2020. *Reactjs vs React Native.* [Online]
Available at: https://www.simform.com/reactjs-vs-reactnative
[Accessed 10 10 2020].
Snigdha, 2020. *25 Best Programming Languages for Mobile Apps & Top Mobile App Development Tools & Frameworks.* [Online]
Available at: https://www.appypie.com/top-programming-languages-for-mobile-app-development
[Accessed 30 09 2020].
Stackoverflow, 2019. *Developer Survey Results.* [Online]
Available at: https://insights.stackoverflow.com/survey/2019#technology
[Accessed 10 10 2020].
Todd Parker, . J. . C. W. P. T., 2010. *Designing with Progressive Enhancement: Building the Web that Works for Everyone.* s.l.:New Riders.
web2appinfotech, 2016. *Mobile App Comparison.* [Online]
Available at: http://www.web2appinfotech.com/mobile-app-comparison-native-vs-hybrid-vs-web/
[Accessed 15 10 2020].

webapp007, 2019. *JavaScript Frameworks 2020.* [Online]
Available at: https://dev.to/webapp007/javascript-frameworks-2020-699
[Accessed 11 10 2020].
Wikipedia, 2020. *React (web framework).* [Online]
Available at: https://en.wikipedia.org/wiki/React_(web_framework)
[Accessed 10 10 2020].
wpostats, 2020. *WPO stats.* [Online]
Available at: https://wpostats.com/
[Accessed 15 10 2020].

5 Appendices

5.1 Appendix 1. Cinkino final Desktop and Mobile performance score

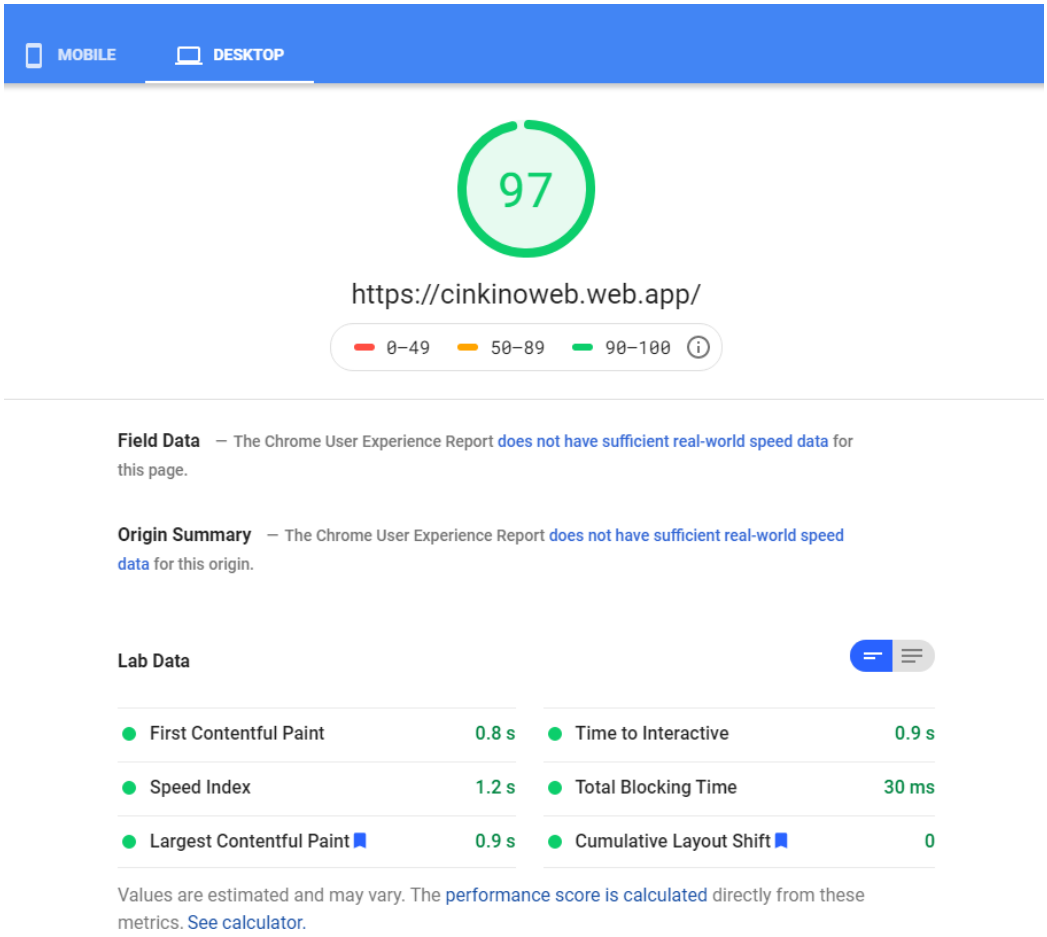


Figure 26 Cinkino final Performance score on Desktop

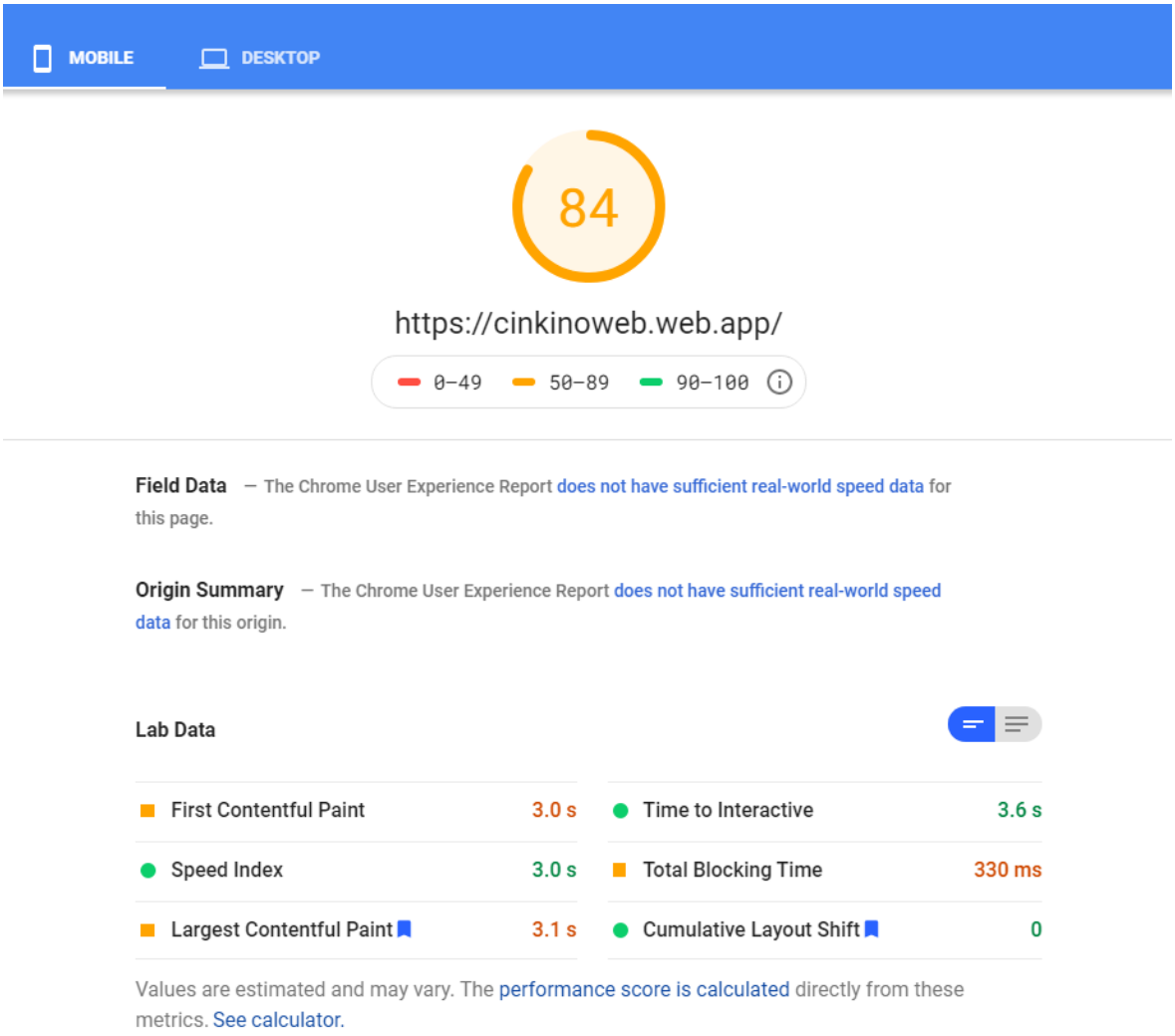


Figure 27 Cinkino final performance score on mobile

Appendix 2. Cinkino final view

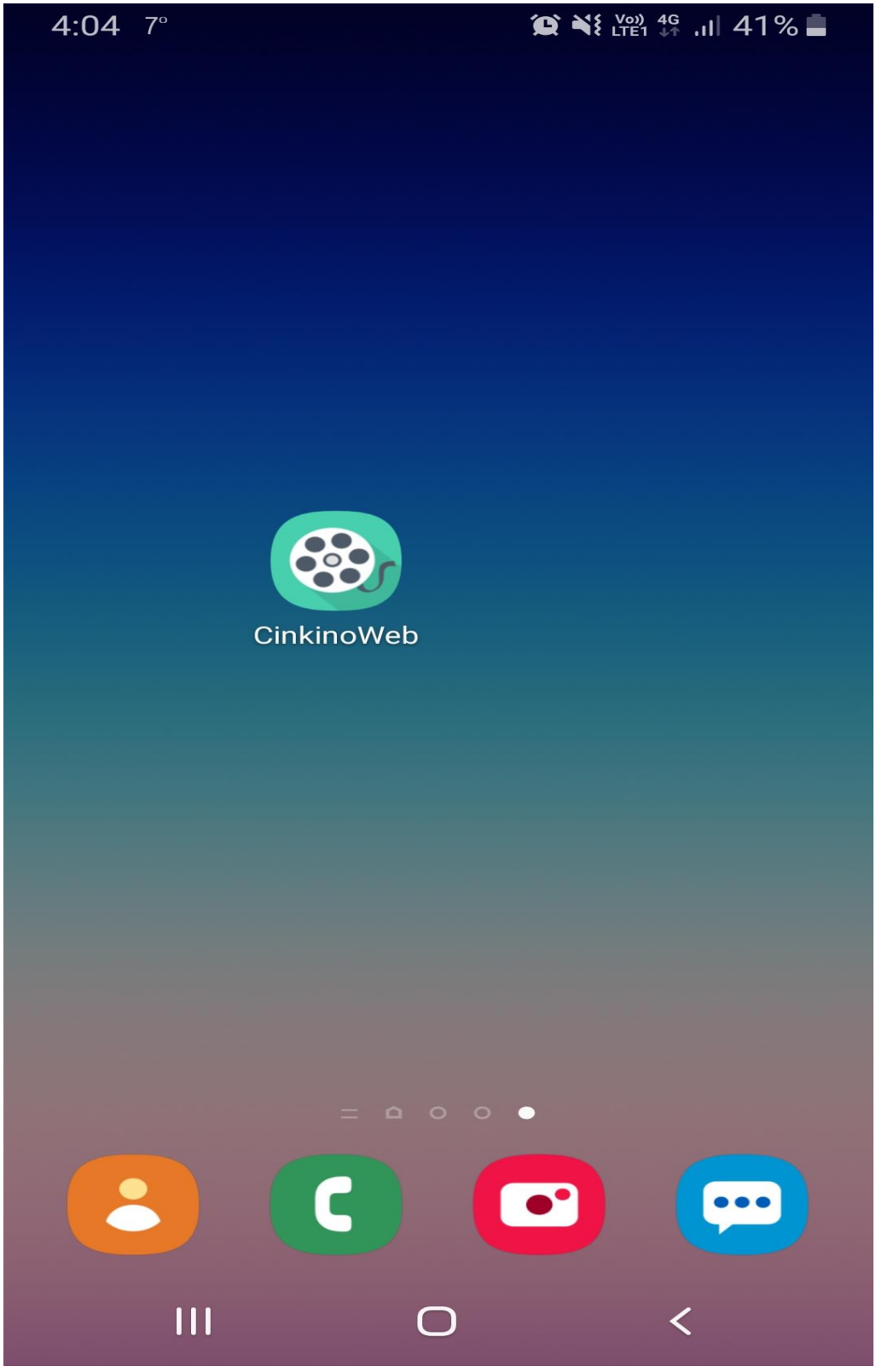


Figure 28 Installed APK version of Cinkino on home screen

Figure 29 Cinkino splash screen on mobile screen

i

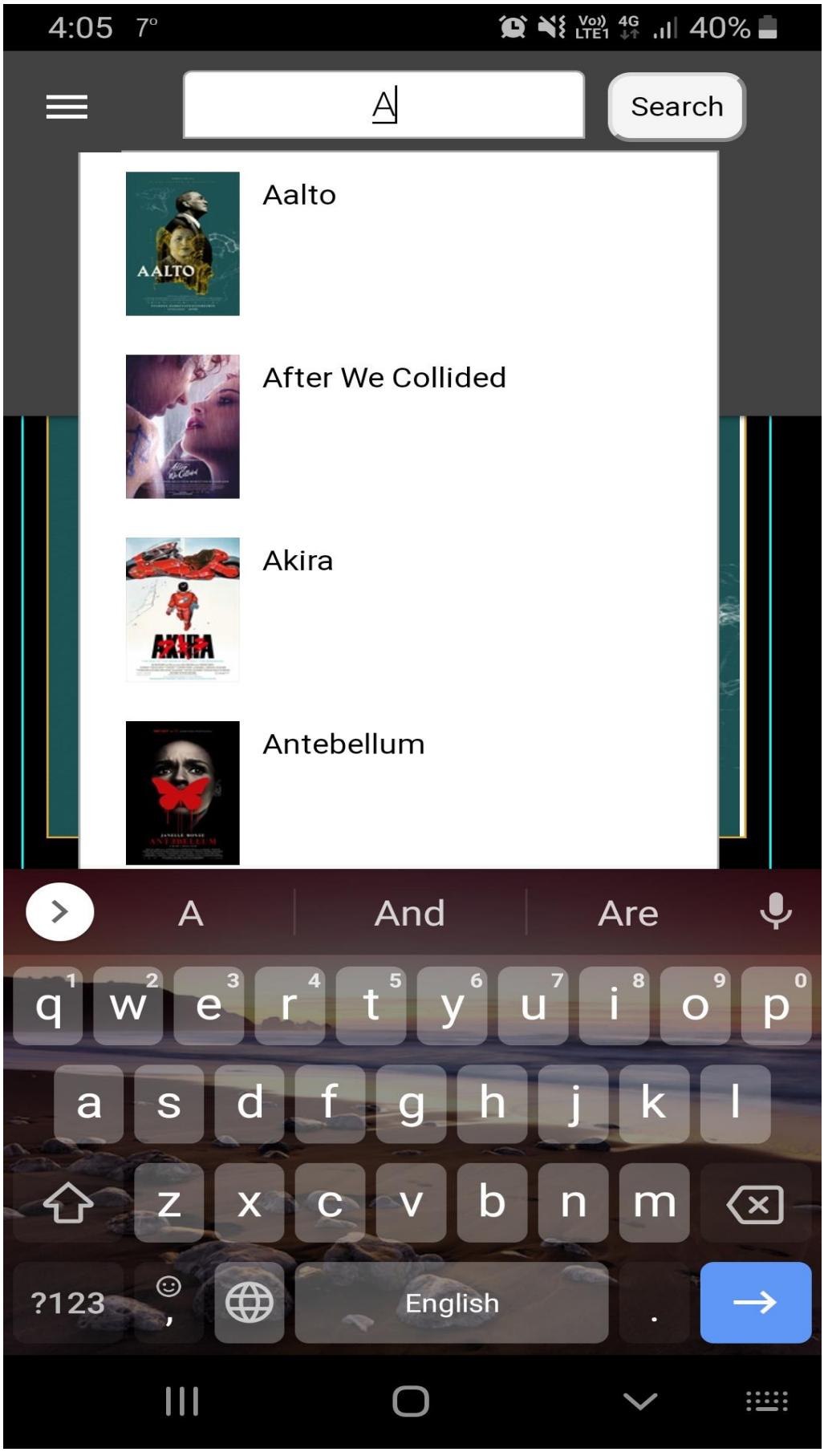


Figure 30 Cinkino Auto suggest Search

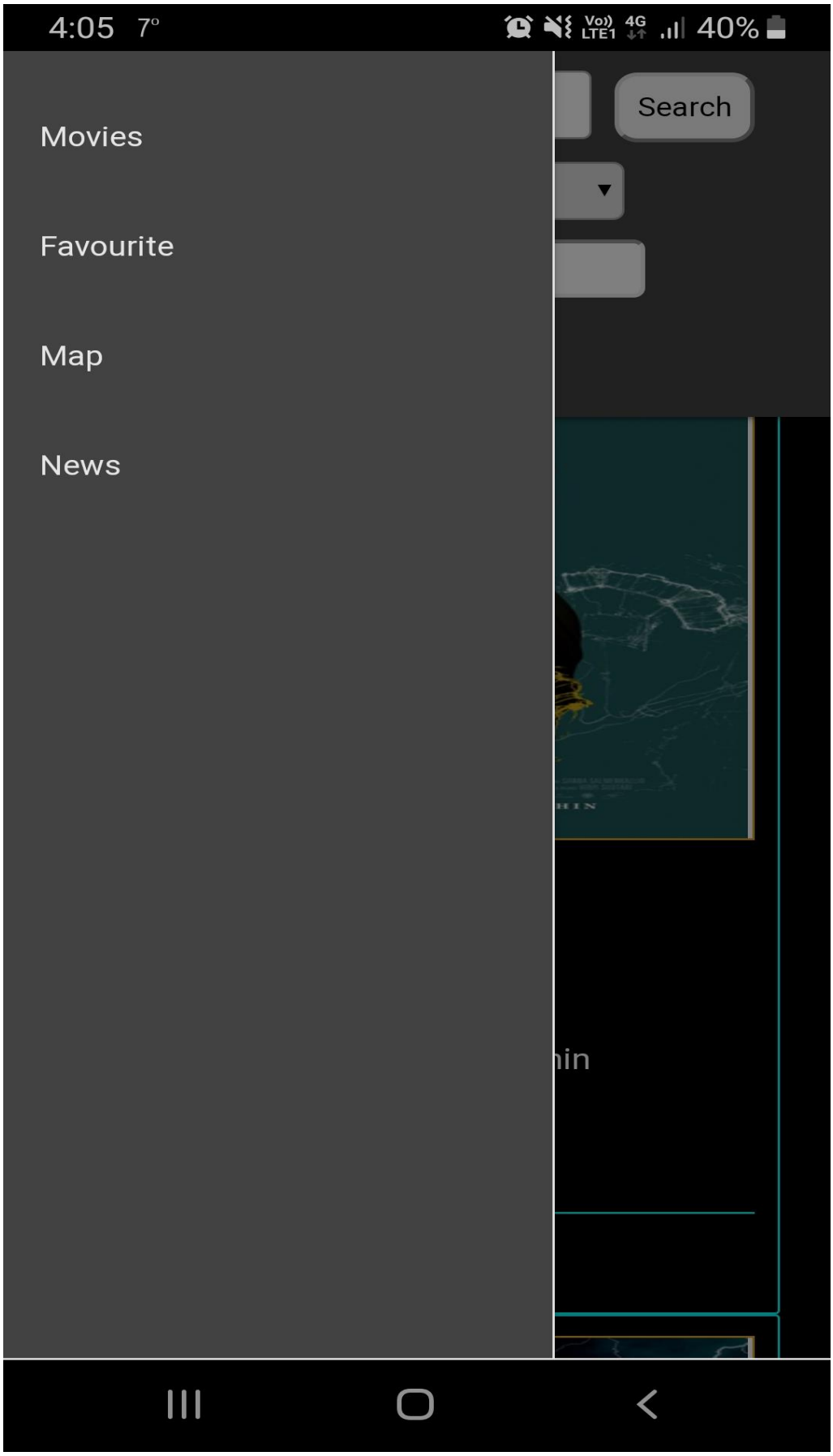


Figure 31 Cinkino sidebar

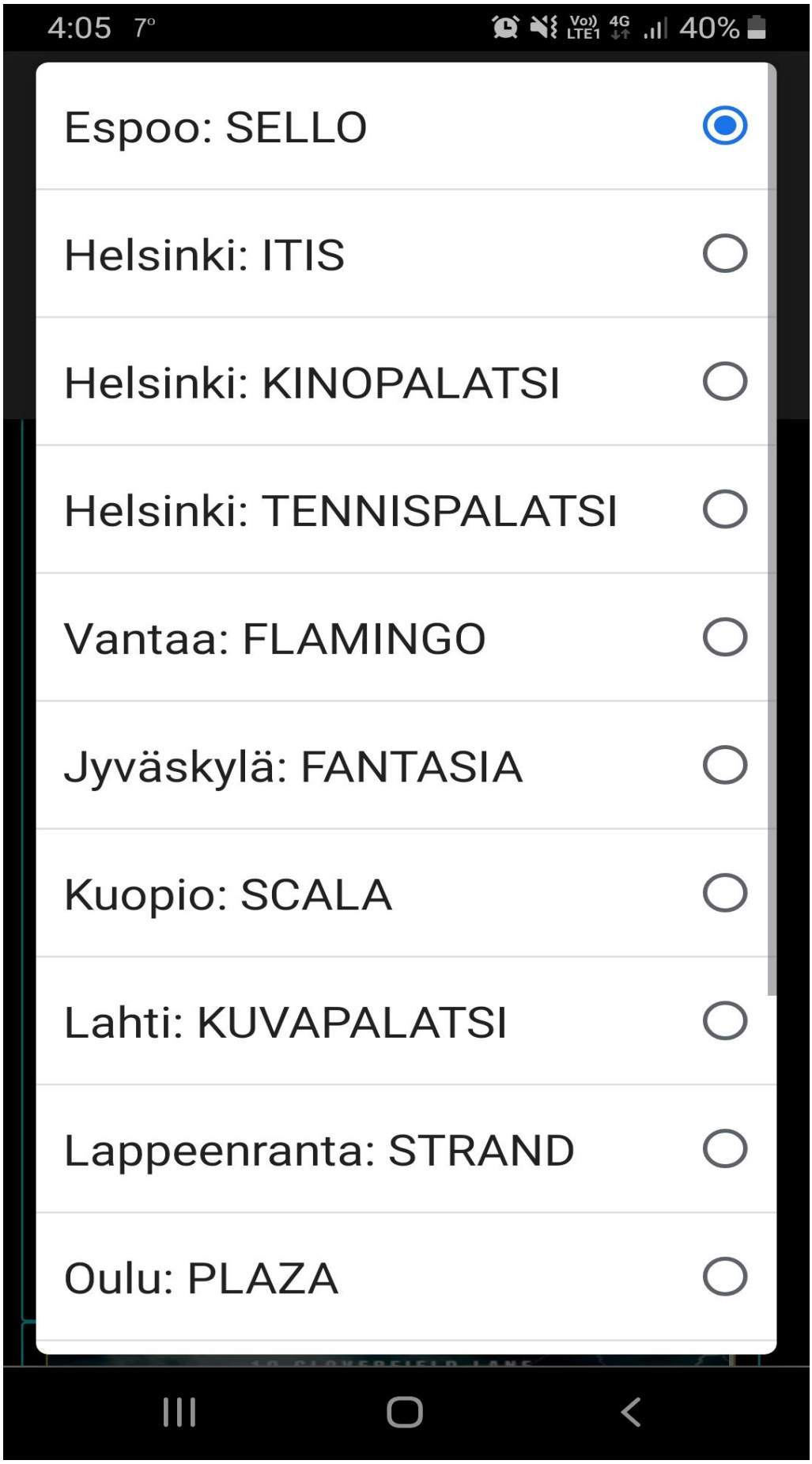


Figure 32 Cinkino Filter by location

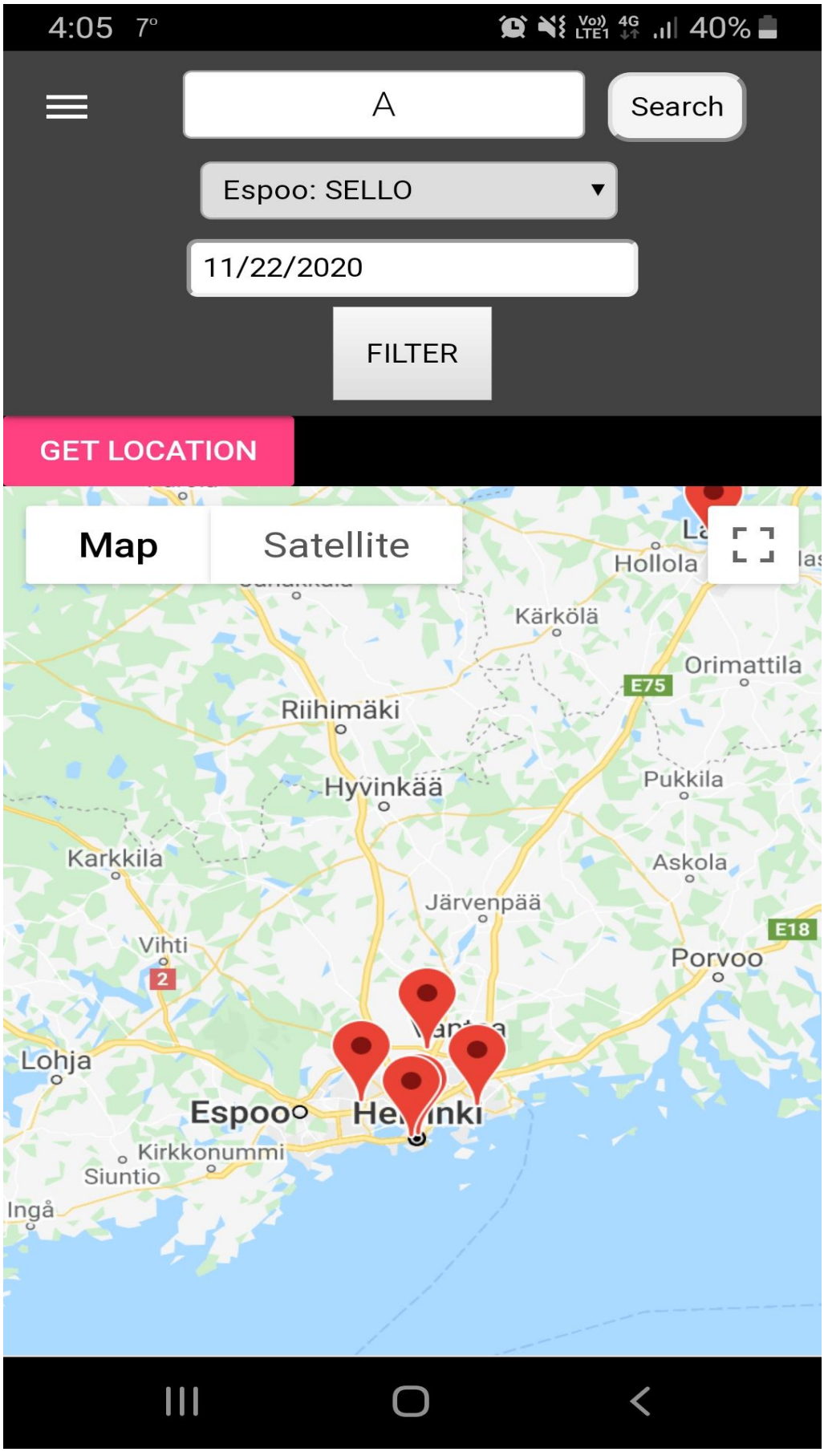


Figure 33 Cinkino Location search

Figure 34 Cinkino News page

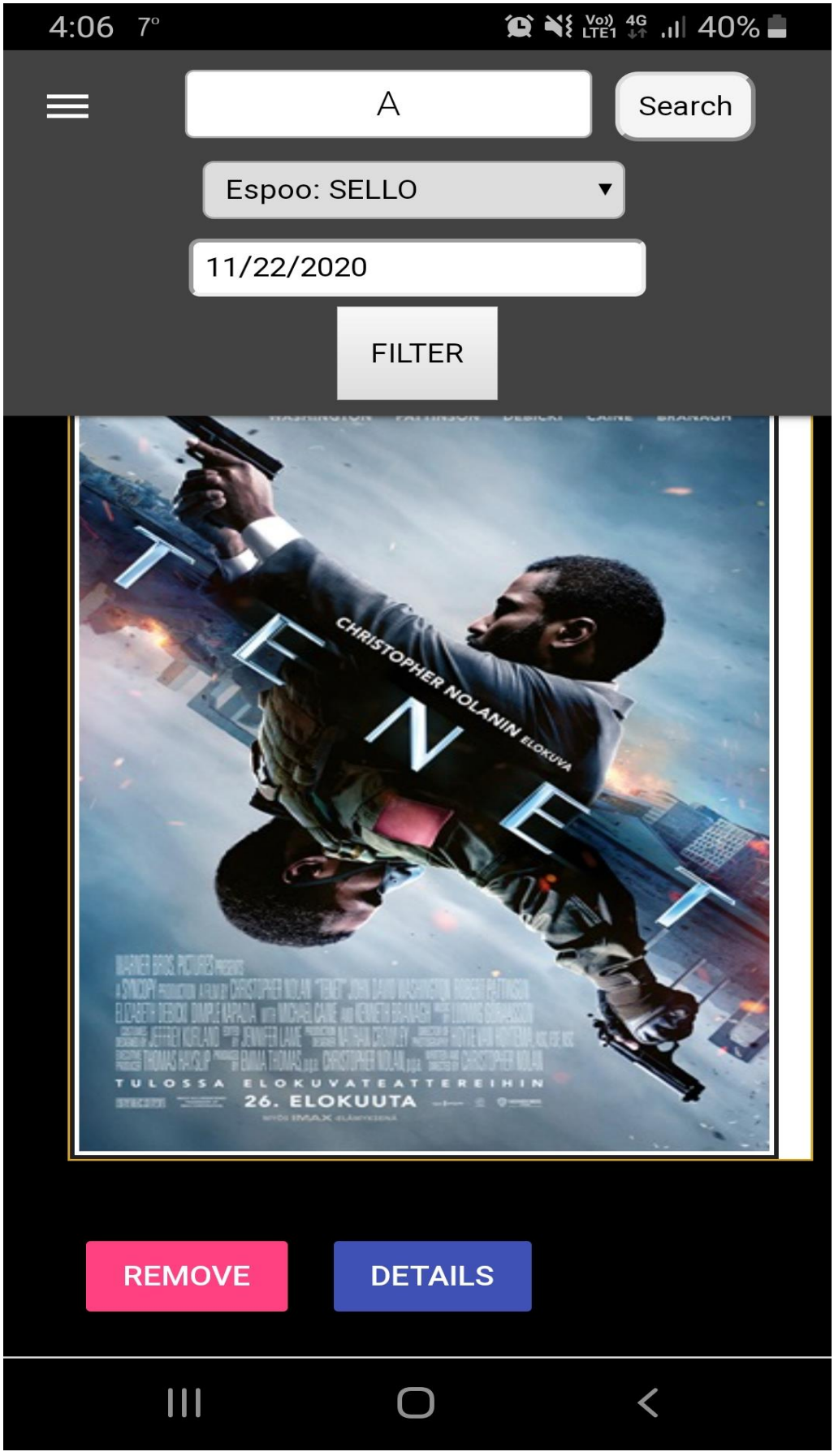


Figure 35 Cinkino Bookmark page