The Integration of Native Mobile App Features into a Progressive Web App

LAB University of Applied Sciences
Bachelor of Engineering, Information Technology
2023
Aki Virtanen

Abstract

Author(s) Aki Virtanen	Publication type Thesis, UAS	Completion year 2023
	Number of pages 24	
Title of the thesis **The Integration of Native Mobile App Features into a Progressive Web App**		
Degree, Field of Study Engineer (UAS), Information Technology		
Organisation of the client Saima Soft Oy		

Abstract

Progressive web app, as a term, refers to a collection of features that modern web sites utilize. Usually, an app is considered to be a progressive web app if it is both installable and usable offline. As such, they bridge to gap between a web site and a native application.

This thesis investigates if a native web app can be replaced with a progressive web app. This is achieved by implementing the features of a progressive web app into an existing web application. The thesis also considers the business viability of making the swich.

This thesis finds that a progressive web app can implement the most wanted features of a mobile application. These features are push notifications, responsiveness, and offline usage. The business viability is considered good, as many companies that have made the switch report increased user retention.

Keywords

PWA, React, Mobile first, Firebase Cloud Messaging

Contents

1 Introduction

Progressive web app (PWA) is less of a single technology, rather it is a collection of features that modern web sites utilize. The primary features that a website must fulfill to be a Progressive web app are usually considered to be the ability to install the website as an app and offline use of the website. Additionally, for a web site to be installable, it must be served via TLS. There are additional features that a progressive web app should have, such as responsiveness, which is a key factor in making a website look good on mobile devices. (Russell 2015.)

This thesis is made in co-operation with Saima Soft Oy, a software company that is partly owned by the StaffPoint Group. Saima Soft Oy creates and maintains HR management solutions for the Group.

The goal of this thesis is to see how viable a PWA is as a replacement to a traditional iOS and Android application. The primary focus is on the viability of the technology and the primary method is to add PWA features into an existing web application. The most important features wanted in a mobile application, such as push notifications, are also needed.

This thesis first covers the origins and theory of PWAs and their design principles. React hooks are briefly covered. The business viability of PWAs is also investigated. Specifically, the usage statistics are in focus. Then, the implementation of PWA features is covered.

2 Progressive Web Apps

2.1 Origins of Progressive web apps

The term progressive web app was first used and coined by Alex Russell, a Google Chrome engineer at the time, in his article Progressive Web Apps: Escaping Tabs Without Losing Our Soul (2015). Russell explains the reason behind giving this collection of features a unified name by describing how new web technologies are not widely adopted until they are named. He cites XMHTTPRequest as an example, saying it was not broadly known about until it was rolled up into the term AJAX. Russell then recalls developing a list of attributes with web designer Frances Berriman describing a progressive web app. These items from Russell's list make PWAs look and feel like native applications:

- *Responsiveness*, meaning that the web fits on the user's screen regardless of its size.
- *Connectivity independent*, meaning the ability to use the web app without an internet connection.
- *App-like-interactions* refers to what is commonly called an app shell structure. An app shell is a minimalist user interface that is loaded first and cached for offline use, if that feature is enabled, after which the content is loaded, creating native app like navigations.
- *Re-engageable* means the ability to use re-engagement methods native to the operating system, usually push notifications.
- Finally, *installable* means the ability to add a PWA to the operating system as its own app outside of a web browser.

One of the predecessors to PWAs was Apples attempt to make iPhone third-party app development possible without a software development kit (SDK). It ultimately failed due to the limitations that contemporary web apps had at the time when compared to native applications (Ritchie 2018). The ongoing development of these web technologies has closed this gap between the native and web apps.

From a developer point of view, one of the biggest advantages of creating a progressive web app instead of a native application is the ability to use the same code on both mobile and desktop. The PWA functions as a mobile app, so new features are available on both the web and mobile simultaneously. A PWA also requires less maintenance than a native application. (Chand 2020.)

2.2 The Native Capabilities of Progressive Web Apps

2.2.1 Push Notifications

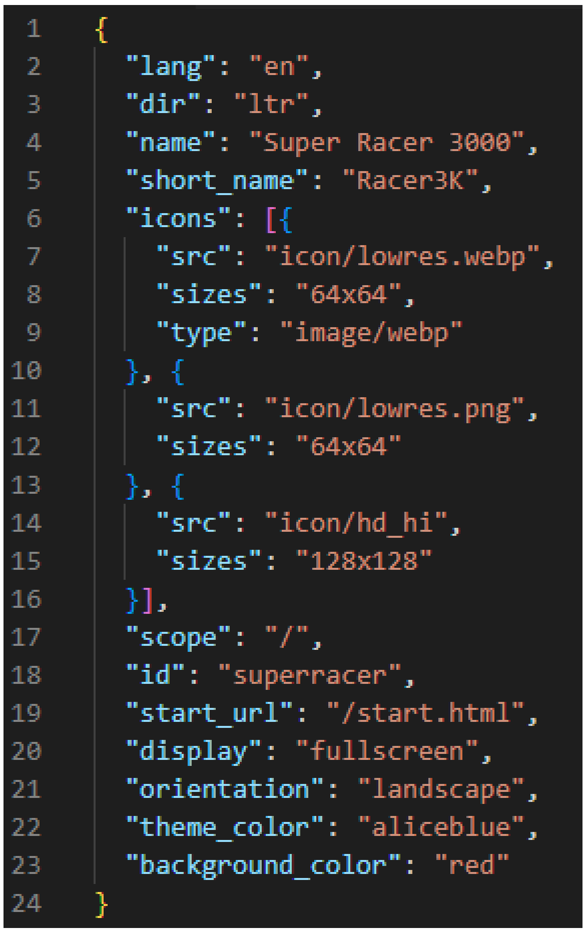
Push notifications as a term comes from server push operation, which is the ability for a server to push information to a client application, for example Outlook, without the client making a request. Therefore, a push notification is a notification that informs the user that they have received information. Using Outlook as an example again, if a user receives an email and they happen to have Outlook open, the new email will be immediately visible to them, as the email server has pushed the email to the client. The client will then send a push notification informing the user of the new email, so they will be aware of the email even if they didn’t have Outlook open.

The push API allows a web app to receive and handle push messages. If used in a service worker, the push API can be used to send push notifications. Service workers are JavaS-cript workers that work between a web app and server on the browser level. Service workers can also keep working after the web app that registered them has been closed. This allows service workers to use the push API when the user is not using the application, creating similar notification functionality as native applications. (MDN Web Docs 2023a.)

Push notifications through the push API are usually authenticated with a Vapid key, a vol-untary application server identification key. This key pair is used to restrict any malicious actors from hijacking messages to another application, as the public key is needed to sub-scribe an app to the push service. For additional security, the key is used to decrypt received push messages. (RFC 8292.)

2.2.2 Installability

To make a web application installable, it must include a web application manifest. Chro-mium-based browsers also require a service worker that caches the web app for offline use. As an example, Figure 1 shows a manifest from the World Wide Web Consortium’s Web Application Manifest (2023) specification.



```json
{
  "lang": "en",
  "dir": "ltr",
  "name": "Super Racer 3000",
  "short_name": "Racer3K",
  "icons": [{
    "src": "icon/lowres.webp",
    "sizes": "64x64",
    "type": "image/webp"
  }, {
    "src": "icon/lowres.png",
    "sizes": "64x64"
  }, {
    "src": "icon/hd_hi",
    "sizes": "128x128"
  }],
  "scope": "/",
  "id": "superracer",
  "start_url": "/start.html",
  "display": "fullscreen",
  "orientation": "landscape",
  "theme_color": "aliceblue",
  "background_color": "red"
}
```

Figure 1. A typical web manifest (World Wide Web Consortium 2023)

The required fields for installability are name and at least one icon with a src parameter, as these are what the application name and icon will be once installed. The non-mandatory fields can still be useful, as they can change the applications look and feel. For example, the background_color field is used on the splash screen that shows up when the app is opened. When a web app is installed, it will appear on the user’s device like a native application. As can be seen in Image 1, the app will still use the browser as a base, but the browser’s user interface can be hidden using the display field of the manifest. (MDN Web Docs 2023b.)

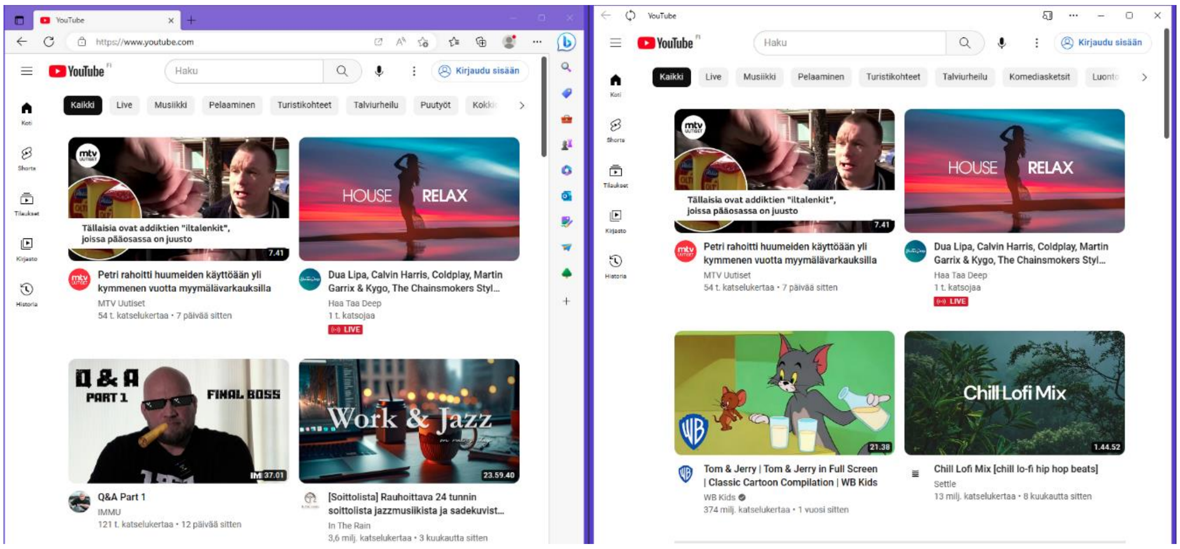

YouTube
https://www.youtube.com
Haku
Kirjaudu sisään
Kaikki
Live
Musiikki
Pelaaminen
Turistikohteet
Talviurheilu
Puutyöt
Komediasketsit
Koti
Shorts
Tilaukset
Kirjasto
Historia
Petri rahoitti huumeiden käyttöään yli kymmenen vuotta myymälävarkauksilla
Dua Lipa, Calvin Harris, Coldplay, Martin Garrix & Kygo, The Chainsmokers Styl...
Q&A Part 1
[Soittolista] Rauhoittava 24 tunnin soittolista jazzmusiikista ja sadekuvist...
Tom & Jerry | Tom & Jerry in Full Screen | Classic Cartoon Compilation | WB Kids
Chill Lofi Mix [chill lo-fi hip hop beats]

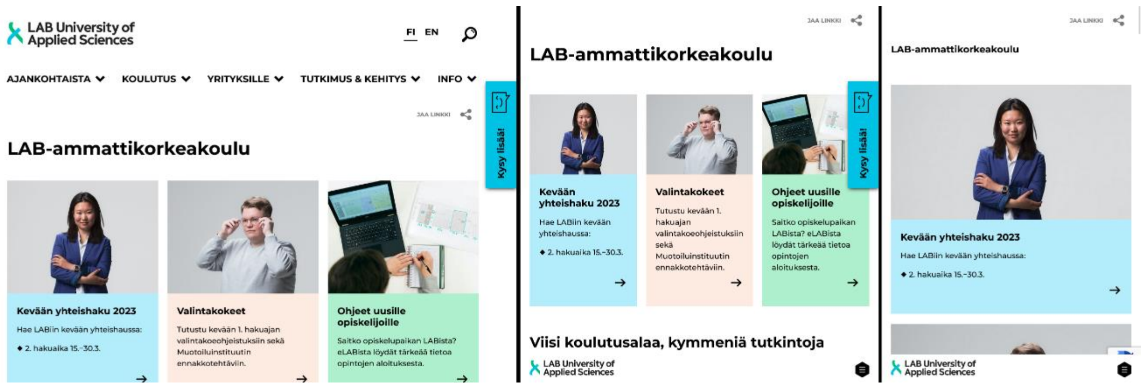
Image 1. YouTube.com on Microsoft Edge next to YouTube installed as an app with Edge

2.3 Design Principles of Progressive Web Apps

2.3.1 Mobile First Design

When creating a web app that will be used on a mobile device, it can be difficult to design it well. One principle that can be used to aid in design is called mobile first. In essence, mobile first design is exactly what it sounds like; designing a web app to primarily be used on the smallest supported device, instead of a large monitor. Then, instead of scaling the site down to fit on a smaller screen, it is scaled up. This approach helps with reaching feature parity between screen sizes, as scaling a web app down and retaining all features is almost impossible. (Boduch 2017, p. 190-192.)

Responsive web design is a strategy that can be used to make a web app look good regardless of the user’s device. Usually this is done by rearranging and resizing the components on the web app. Image 2 shows an example of responsive web design being used on the LAB University of Applied Sciences’ home page. (Gonzalez 2013, p. 5–9.)



LAB-ammattikorkeakoulu

Kevään yhteishaku 2023
Hae LABiin kevään yhteishaussa:
◆ 2. hakuaika 15.–30.3.

Valintakokeet
Tutustu kevään 1. hakuajan valintakoeohjeistuksiin sekä Muotoiluinstituutin ennakkotehtäviin.

Ohjeet uusille opiskelijoille
Saitko opiskelupaikan LABista? eLABista löydät tärkeää tietoa opintojen aloituksesta.

Viisi koulutusalaa, kymmeniä tutkintoja

Image 2. LAB University of Applied Sciences’ home page at different device widths

As Wroblewski (2009) points out, designing a web app with mobile first considerations focuses the design on the most important aspects of the web app. Wroblewski continues by laying out the benefits of mobile first, saying that it requires developers to focus on the most important features of the web app. Further, Wroblewski cites projections and statistics to argue that mobile first design is important as usage of web apps on mobile devices is going to increase. Figure 2 confirms Wroblewski’s assessment, as mobile devices have slowly overtaken desktop. The data comes from Statcounter, which is a web analytics service that collects data from over a million web sites.

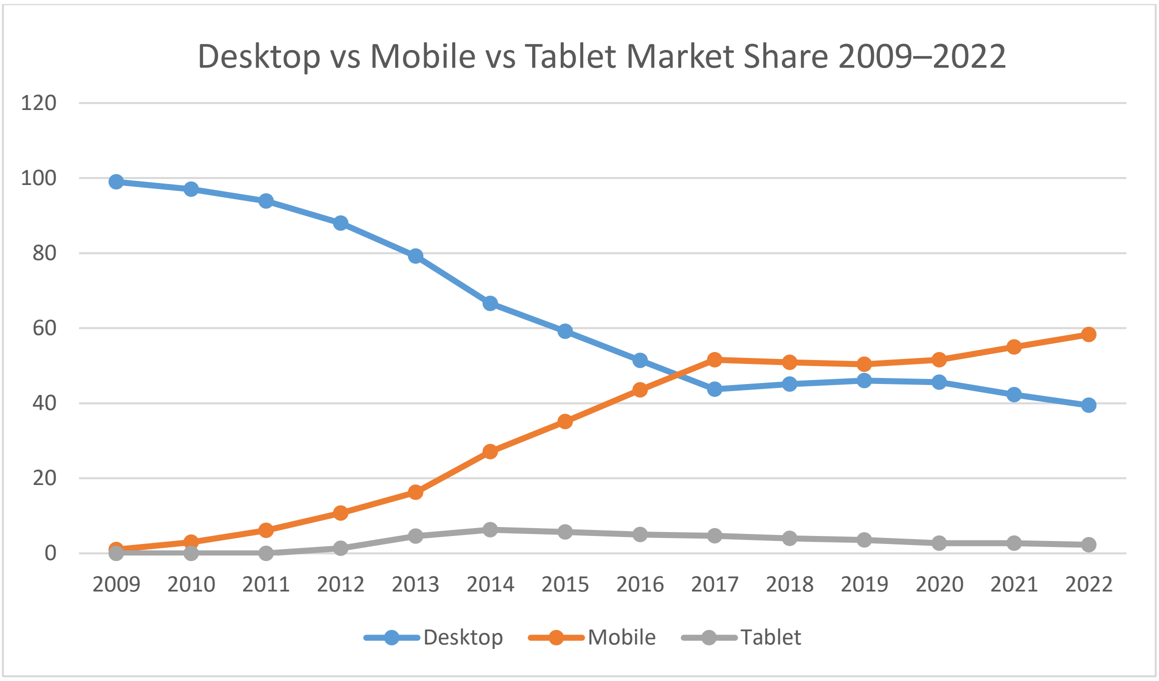


Figure 2. Desktop vs Mobile vs Tablet Market Share Worldwide 2009–2022 (Statcounter)

2.3.2 Offline First Design

Offline first, also known as cache first, is a strategy of showing a web app to a user. The main principle is to save as many resources to the user's browser cache as possible and then use the cache to display the web app instead of fetching the resources from a server. (MDN Web Docs 2023a; Domes 2017, p. 234–235.)

While making a web app offline usable does provide shorter load times for all users, the mobile users receive the most benefits. If a mobile user is moving through an area with a bad connection, they can continue using the site's offline features. For example, the message history of a chat feature could be saved on the user's device and shown even if an internet connection is not available.

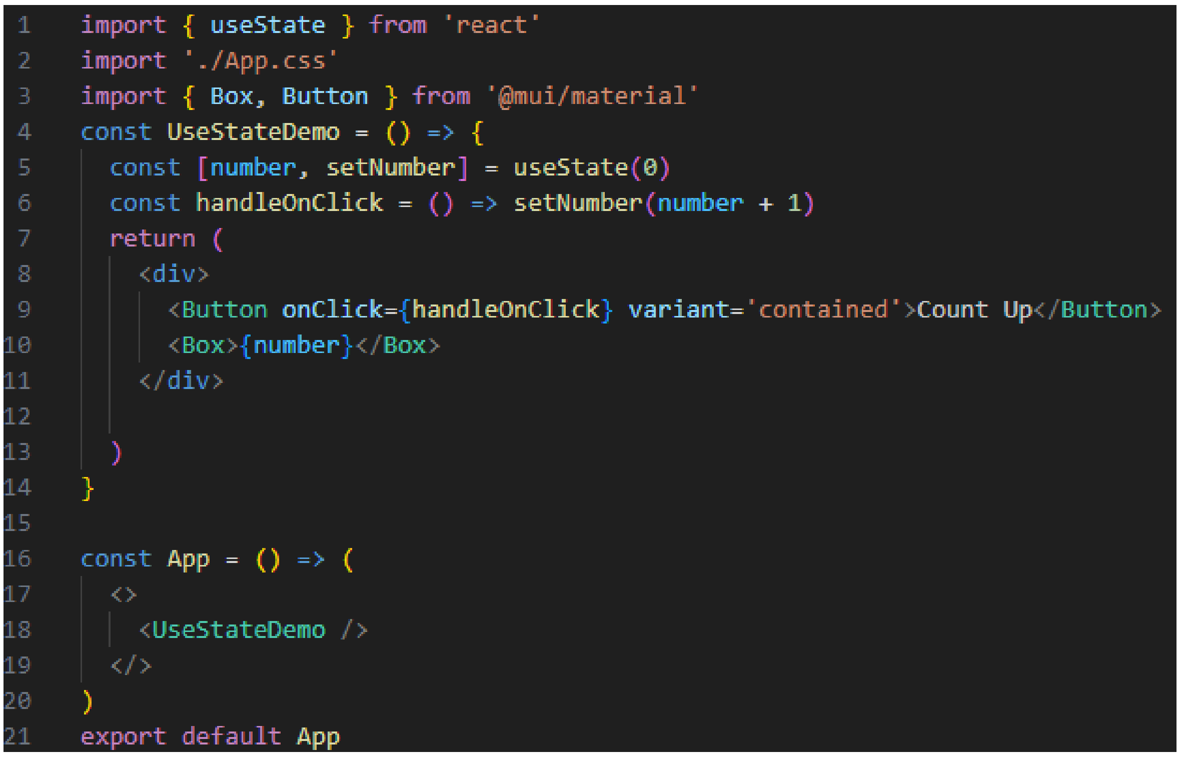
Web app caching is usually done with a service worker. The service worker runs in the background and does the caching of a web app's resources and data. The API calls of the web app are made through the service worker, the response data is cached and then shown to the user. When the user is offline, the service worker fetches the cached data if it exists. (Gambhir & Raj 2018)

2.4 React and Progressive Web App Integration

2.4.1 Hooks

React has hooks that let programmers use the many features of React from their components. Some of the most important hooks are useState and useEffect. (Meta Open Source.)

The use state hook can be used to make a component remember values and re-render if the values change. If a component should show values depending on user input, the hook must be used instead of a normal variable, as functional React components do not remember value changes to local variables. Figure 3 shows an example use case of a useState. (Meta Open Source.)

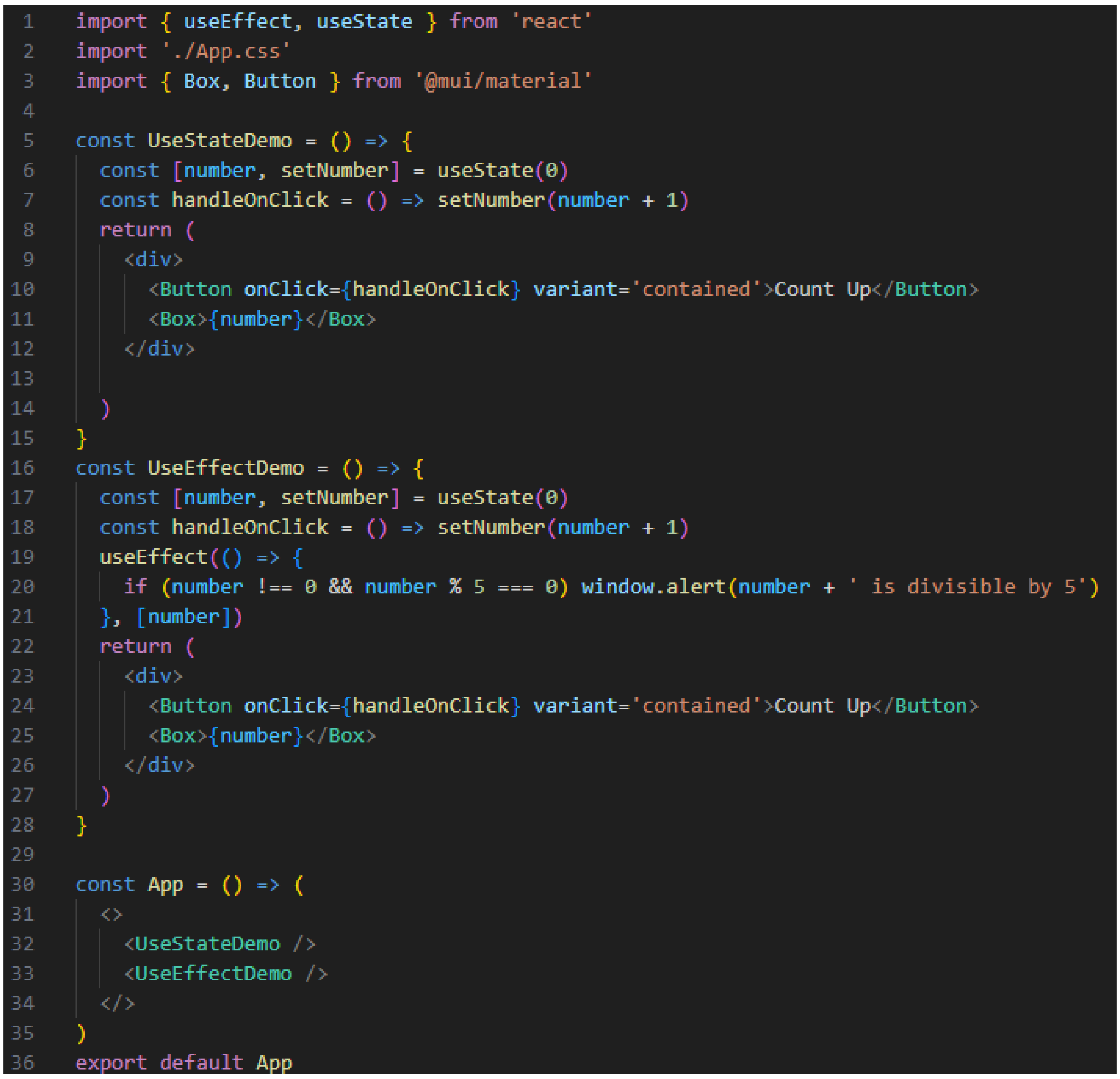


```
1    import { useState } from 'react'
2    import './App.css'
3    import { Box, Button } from '@mui/material'
4    const UseStateDemo = () => {
5      const [number, setNumber] = useState(0)
6      const handleOnClick = () => setNumber(number + 1)
7      return (
8        <div>
9          <Button onClick={handleOnClick} variant='contained'>Count Up</Button>
10         <Box>{number}</Box>
11       </div>
12
13     )
14   }
15
16   const App = () => (
17     <>
18       <UseStateDemo />
19     </>
20   )
21   export default App
```

Figure 3. A demonstration of useState

A counter is shown. It’s value changes by one every time the button is clicked. The value is stored in the number state variable and is changed using the setNumber method, both being returned by the useState hook called on line 5. The hook is given the default value that the variable should have. When the setNumber method is called, the variable is changed, and the component is re-rendered.

The useEffect hook should only be used when the component interacts with a system outside of React. For example, fetching data with user input would be done with a useEffect hook. The hook takes a callback function as a parameter. Optionally, a state can be provided as a dependency. If no state is given, the callback function is run every time the component is rendered. If a state is provided, the call back only happens when that state changes. Figure 4 shows a use case for the hook.



```
1   import { useEffect, useState } from 'react'
2   import './App.css'
3   import { Box, Button } from '@mui/material'
4
5   const UseStateDemo = () => {
6     const [number, setNumber] = useState(0)
7     const handleOnClick = () => setNumber(number + 1)
8     return (
9       <div>
10        <Button onClick={handleOnClick} variant='contained'>Count Up</Button>
11        <Box>{number}</Box>
12      </div>
13
14    )
15  }
16  const UseEffectDemo = () => {
17    const [number, setNumber] = useState(0)
18    const handleOnClick = () => setNumber(number + 1)
19    useEffect(() => {
20      if (number !== 0 && number % 5 === 0) window.alert(number + ' is divisible by 5')
21    }, [number])
22    return (
23      <div>
24        <Button onClick={handleOnClick} variant='contained'>Count Up</Button>
25        <Box>{number}</Box>
26      </div>
27    )
28  }
29
30  const App = () => (
31    <>
32      <UseStateDemo />
33      <UseEffectDemo />
34    </>
35  )
36  export default App
```

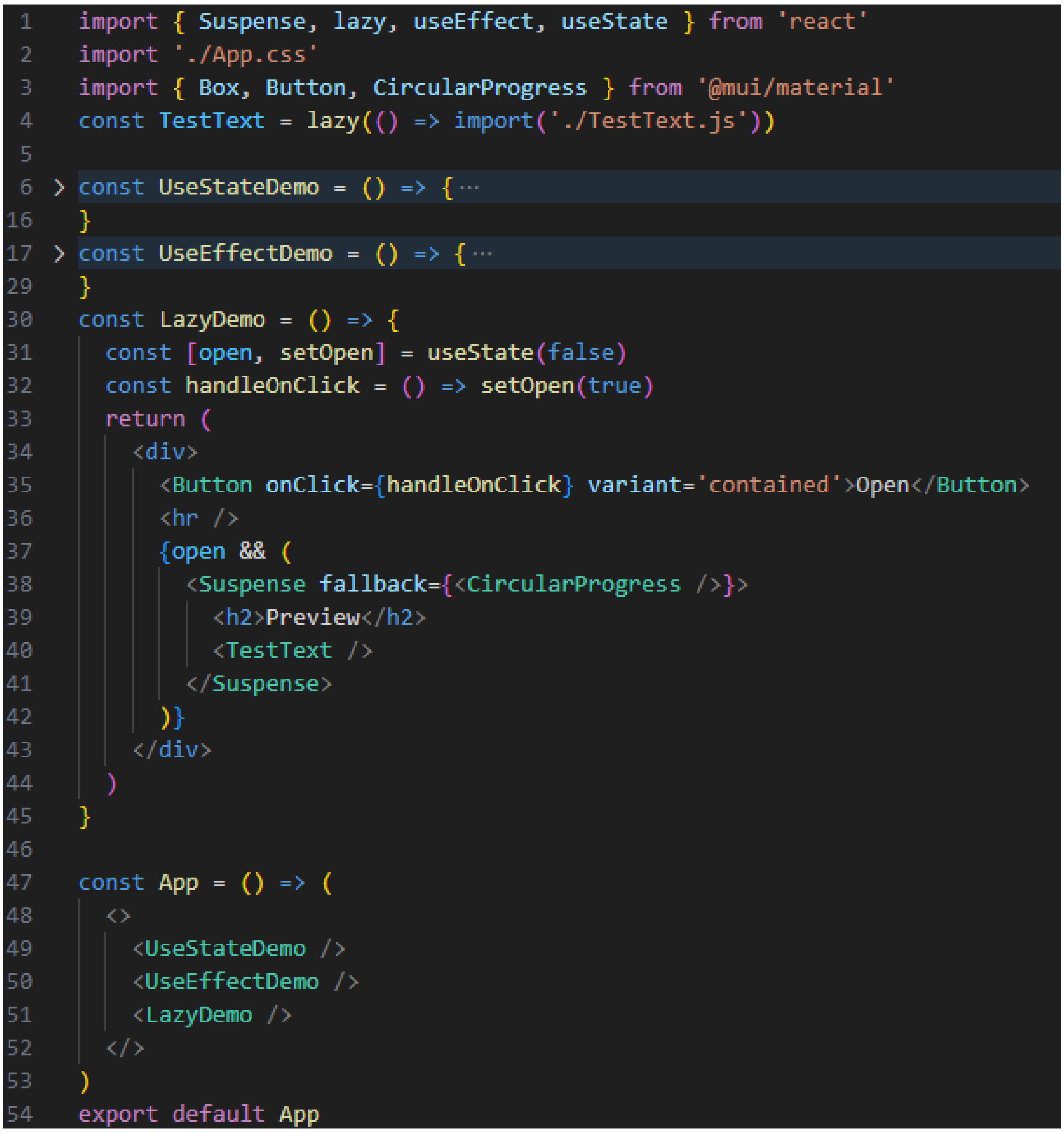
Figure 4. A demonstration of useEffect

Similarly to the previous example, a counter goes up when a button is pressed. The value of the counter is stored in the number state variable. A useEffect hook is created on lines 19 to 21, which has the number as a dependent. When the button is pressed and the counter increases, the callback function will be run. The function shows a browser popup if the number is divisible by 5.

2.4.2 Lazy loading

Lazy loading is used to load a web sites resource only once they are needed. This then shortens the load that a user would have to wait through when they first navigate to the web site. Further resources are loaded once the user navigates to a part of the site that needs them. For example, on a news site, only a couple of articles would be loaded at the start and once the user scrolls down, more would be loaded. (MDN Web Docs 2023c.)

React has a native method for lazy loading simply called lazy. This method can be used to only load components once they are rendered. With a Suspense component, a placeholder can be shown while the component is being loaded. In Figure 5, the Text component is only loaded once the button is clicked. A CircularProgress component from the Material UI component library is provided to the Suspense as a fallback, so it is shown while the TestText component is being loaded. (Meta Open Source.)

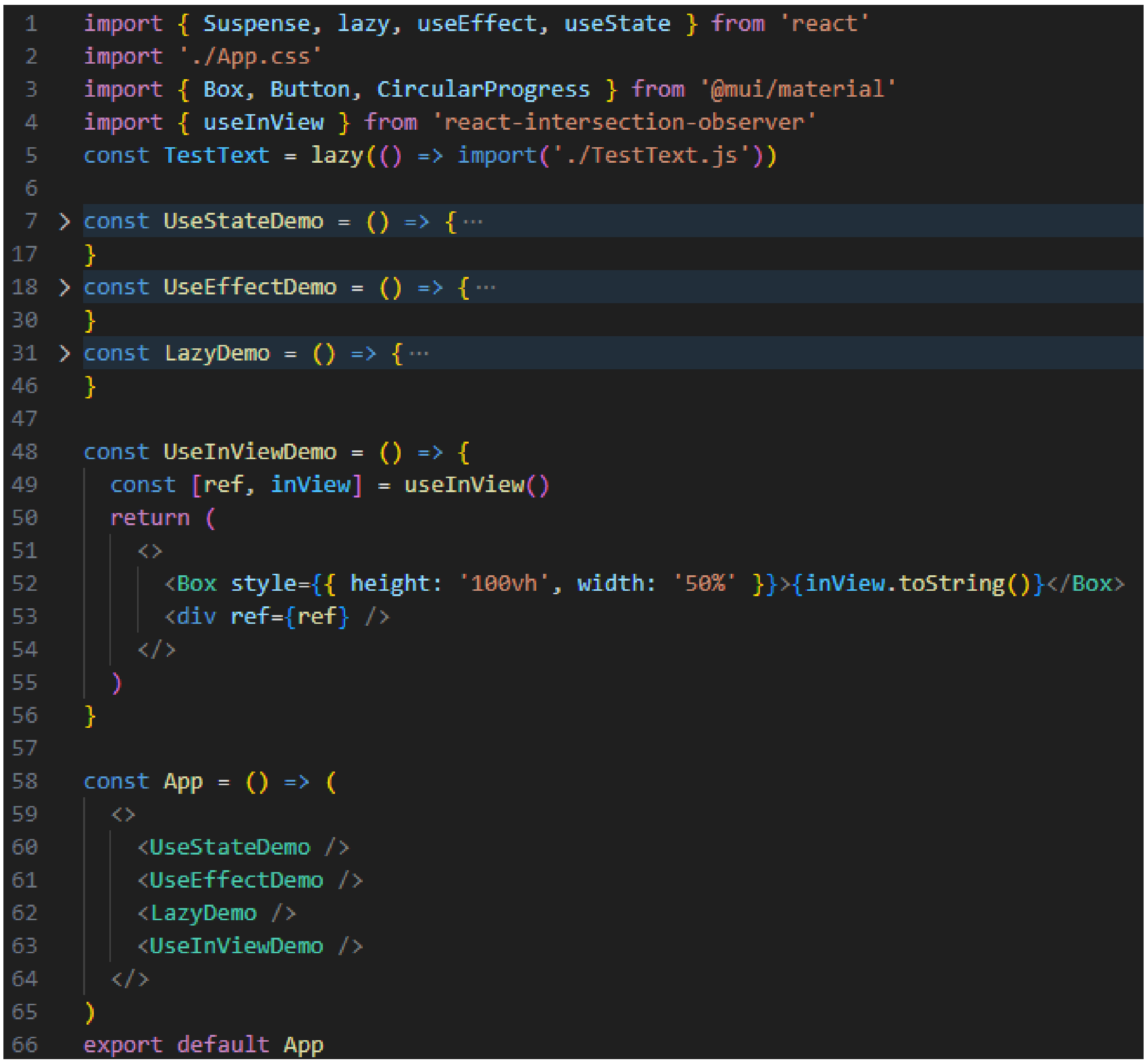


```
import { Suspense, lazy, useEffect, useState } from 'react'
import './App.css'
import { Box, Button, CircularProgress } from '@mui/material'
const TestText = lazy(() => import('./TestText.js'))

const UseStateDemo = () => {⋯
}
const UseEffectDemo = () => {⋯
}
const LazyDemo = () => {
  const [open, setOpen] = useState(false)
  const handleOnClick = () => setOpen(true)
  return (
    <div>
      <Button onClick={handleOnClick} variant='contained'>Open</Button>
      <hr />
      {open && (
        <Suspense fallback={<CircularProgress />}>
          <h2>Preview</h2>
          <TestText />
        </Suspense>
      )}
    </div>
  )
}

const App = () => (
  <>
    <UseStateDemo />
    <UseEffectDemo />
    <LazyDemo />
  </>
)
export default App
```

Figure 5. A demonstration of React lazy loading

The intersection observer API can be used to lazy load data. The API is given a reference to an element that it should watch. Once the element is in view, a callback function is run. The observer can be customized with a threshold value, which will change how much of the element would have to be in view before the callback is run. (MDN Web Docs 2023d.)

A third-party package called React Intersection Observer implements the intersection observer API with React hooks and states. The hook useInView receives the options wanted for the observer and the hook returns a ref and a state. The ref would be given to the component that should be observed and the state would update when the component is in view. Figure 6 shows an example of the hook. (React Intersection Observer authors 2023.)



```
1   import { Suspense, lazy, useEffect, useState } from 'react'
2   import './App.css'
3   import { Box, Button, CircularProgress } from '@mui/material'
4   import { useInView } from 'react-intersection-observer'
5   const TestText = lazy(() => import('./TestText.js'))
6
7   const UseStateDemo = () => {⋯
17  }
18  const UseEffectDemo = () => {⋯
30  }
31  const LazyDemo = () => {⋯
46  }
47
48  const UseInViewDemo = () => {
49    const [ref, inView] = useInView()
50    return (
51      <>
52        <Box style={{ height: '100vh', width: '50%' }}>{inView.toString()}</Box>
53        <div ref={ref} />
54      </>
55    )
56  }
57
58  const App = () => (
59    <>
60      <UseStateDemo />
61      <UseEffectDemo />
62      <LazyDemo />
63      <UseInViewDemo />
64    </>
65  )
66  export default App
```

Figure 6. A demonstration of UseInView

The hook is called on line 49. As no options are given, the default value of zero will be used for the threshold. The Boolean state variable inView is shown as a string. The ref is assigned to an empty div component at the end of the page. Once the page is scrolled down, the inView variable will change to true.

2.5 Analytics of Progressive Web Applications

From a corporate perspective, progressive web apps have been a success. The website PWA stats lists stories of increased user bases and general app usage when different companies switched from native apps to PWAs. These are not small companies either, for example, Twitter, Forbes, and Starbucks are listed among many others. In an article titled Twitter Lite PWA Significantly Increases Engagement and Reduces Data Usage (2017), the PWA released under the name Twitter Lite is discussed in detail. The usage statistics are said to be a 65% increase in pages per session, a 75% increase in Tweets sent and a 20%

decrease in bounce rate. Nicolas Gallagher, a lead engineer on Twitter Lite is quoted saying that Twitter Lite requires only 3% of the storage space that the Twitter native Android application requires.

One concern with PWAs is how to get the app on the user's device. With mobile native applications, the app is submitted to either the Google Play Store or Apple's App Store giving users a unified place from which to download apps. The worry is that how can the users be informed of the web app being installable without dissuading them from installing it. Luckily, there does not seem to be any reason to worry, as Lyft's statistics show a 40% increase in users clicking an install PWA button when compared to a download app button (Dreyer 2022).

3 Upgrading to A Progressive Web Application

3.1 Adding Progressive Web App Features to An Existing Web App

The first step of this project was to analyze what the different PWA features are and what needed to be implemented. Table 1 shows the status of PWA features and whether they were added to the app during this project or existed before hand.

Table 1. The PWA features of the app this project developed

PWA Feature	Preexisting feature or implemented during thesis
Responsiveness	Preexisting
Connectivity independent	Project
App-like-interactions	Preexisting
Re-engageable	Project
Installable	Project

The two PWA features not added during this project, which were responsiveness and app-like-interactions, were still considerations during development. As such, they warrant brief descriptions of how they were implemented. App-like-interactions were implemented with an app shell that was made with React Router and Material UI, the component library that is in use. Responsiveness is done with Material UI's breakpoints defined in the app's theme. The breakpoints are then used in a component to change its attributes depending on the user's screen width. At some point during development, it was decided that the lowest supported device width would be 360 pixels because that is the lowest device width with any significant usage share in Finland (Figure 7).

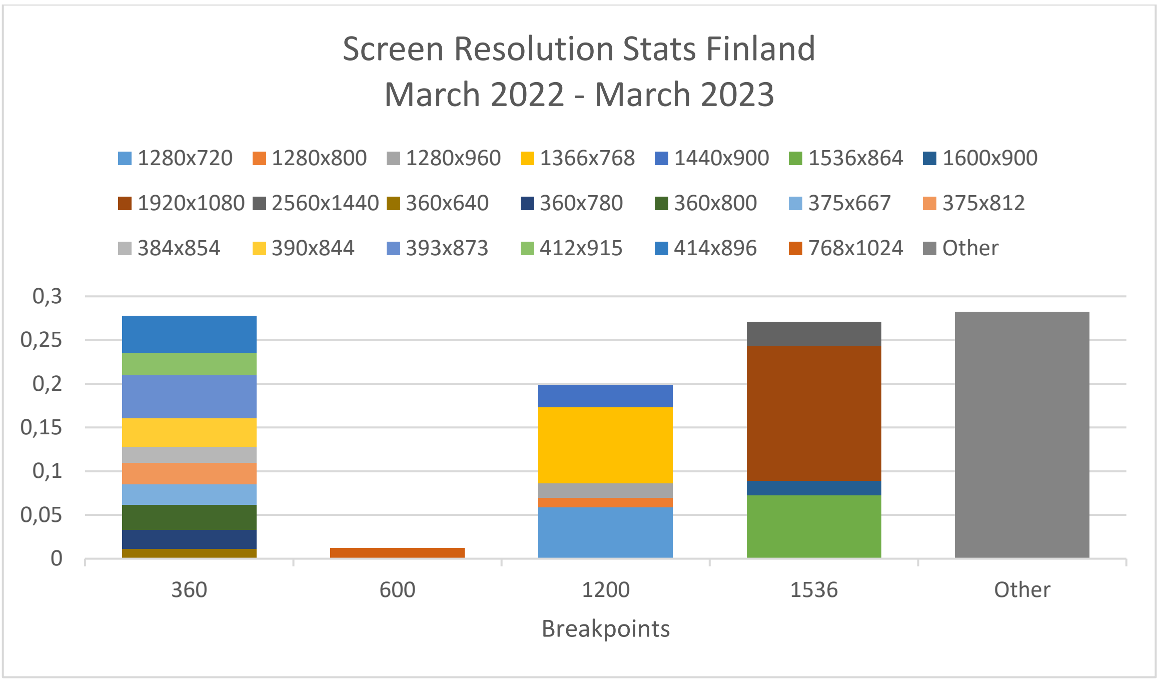
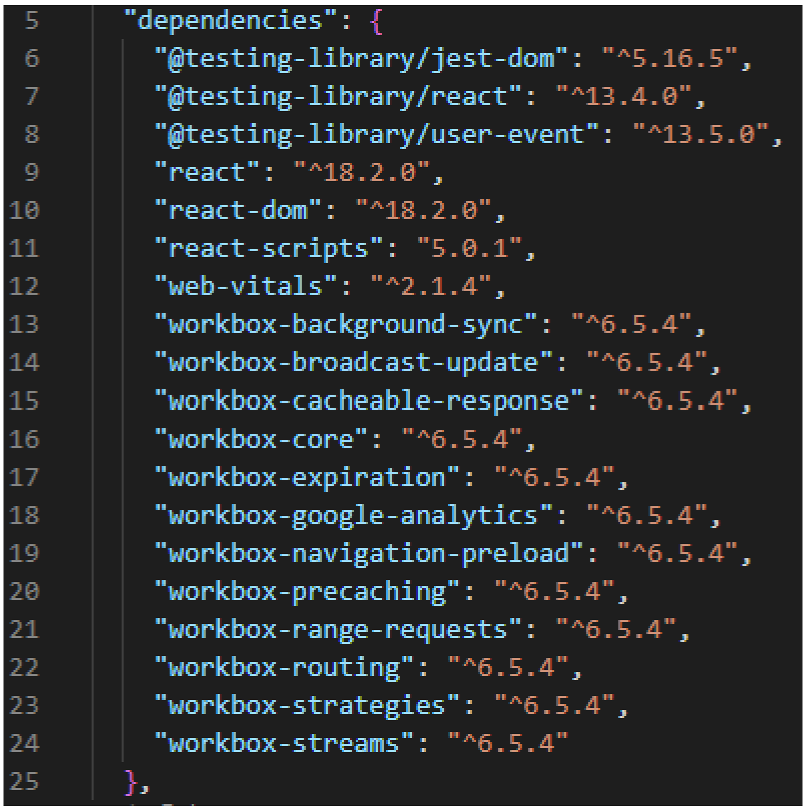


Figure 7. Screen resolution percentage compared to used breakpoints (Statcounter)

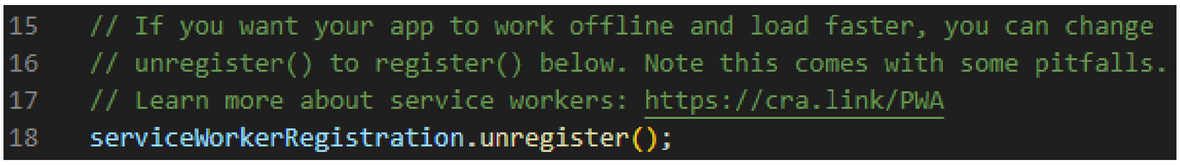
The only requirement of installability that was not met at the beginning of this project was a service worker allowing offline usage. The manifest file was edited to use the correct image as the app's icon, but it had the correct information otherwise. Because the app is made with React and Create React App, the logical place to start was to create a new app with Create React App's PWA template and seeing what the differences were between it and the existing app. The newly created app had twelve packages that were not in the existing app's package.json, those being workbox and its modules (Figure 8).



```
"dependencies": {
  "@testing-library/jest-dom": "^5.16.5",
  "@testing-library/react": "^13.4.0",
  "@testing-library/user-event": "^13.5.0",
  "react": "^18.2.0",
  "react-dom": "^18.2.0",
  "react-scripts": "5.0.1",
  "web-vitals": "^2.1.4",
  "workbox-background-sync": "^6.5.4",
  "workbox-broadcast-update": "^6.5.4",
  "workbox-cacheable-response": "^6.5.4",
  "workbox-core": "^6.5.4",
  "workbox-expiration": "^6.5.4",
  "workbox-google-analytics": "^6.5.4",
  "workbox-navigation-preload": "^6.5.4",
  "workbox-precaching": "^6.5.4",
  "workbox-range-requests": "^6.5.4",
  "workbox-routing": "^6.5.4",
  "workbox-strategies": "^6.5.4",
  "workbox-streams": "^6.5.4"
},
```

Figure 8. Dependencies of an app created by Create React App's PWA template

Two additional files were present in the template app's src folder, service-worker.js and serviceWorkerRegistration.js. Four new lines were present in the index.js (Figure 9).



```
15  // If you want your app to work offline and load faster, you can change
16  // unregister() to register() below. Note this comes with some pitfalls.
17  // Learn more about service workers: https://cra.link/PWA
18  serviceWorkerRegistration.unregister();
```

Figure 9. Lines 15-18 in Create React App's PWA template's index.js file

After installing the new packages and copying the file additions and changes to the existing application, enabling the offline caching was as simple as changing line 18 of index.js to say serviceWorkerRegistration.register().

3.2 Implementing Native App Features in a Progressive Web App

3.2.1 Push Notifications Through Firebase

As the web application that was developed in this project was meant to replace a native mobile application capable of receiving push notifications, they must be a feature in the web app as well. As the mobile app gets push notifications from a backend server through Firebase, the web app should do the same. As Firebase was already used in the web app, no new packages needed to be added. Thus, the first thing that needed to be done was to generate a Vapid Key from the Firebase console to subscribe the app to receive messages.

A new file was created to handle Firebase messaging, which contains the method creating a Firebase push notification subscription (Figure 10).

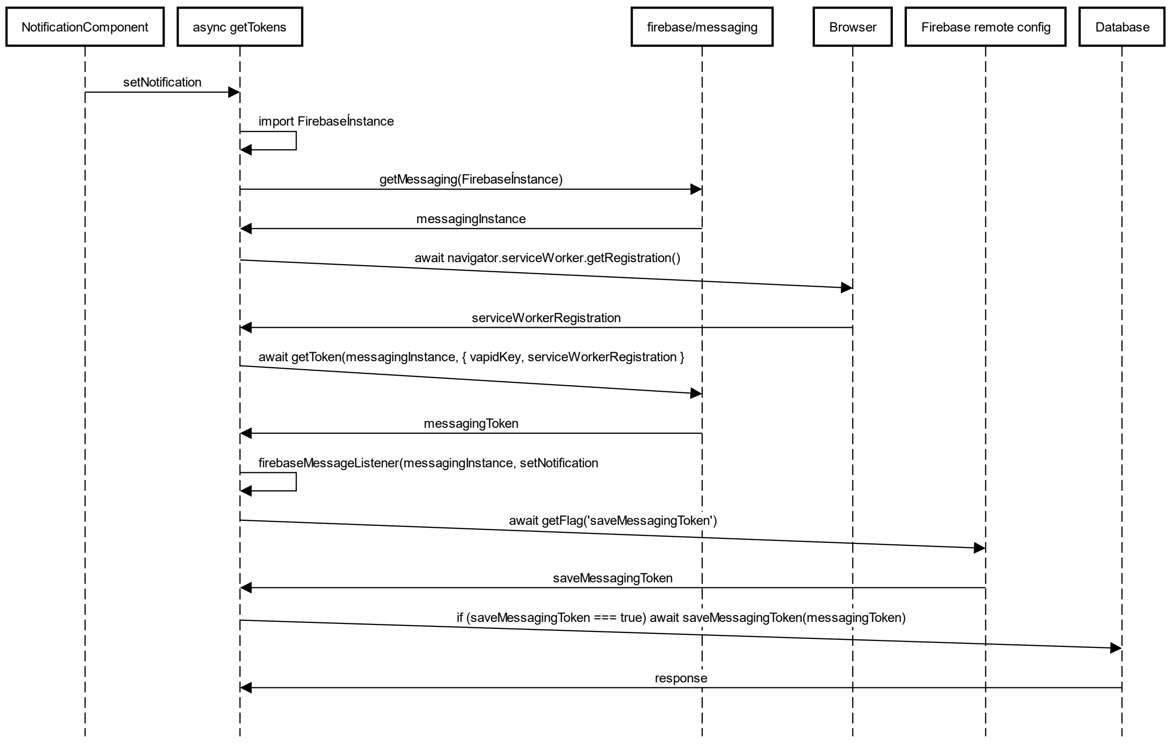

NotificationComponent
async getTokens
firebase/messaging
Browser
Firebase remote config
Database
setNotification
import FirebaseInstance
getMessaging(FirebaseInstance)
messagingInstance
await navigator.serviceWorker.getRegistration()
serviceWorkerRegistration
await getToken(messagingInstance, { vapidKey, serviceWorkerRegistration }
messagingToken
firebaseMessageListener(messagingInstance, setNotification
await getFlag('saveMessagingToken')
saveMessagingToken
if (saveMessagingToken === true) await saveMessagingToken(messagingToken)
response

Figure 10. A sequence diagram of firebase messaging

The getTokens method was created as asynchronous, as it needs to get a messaging token from the Firebase Cloud. A Firebase messaging instance is created with an imported Fire-base instance. The getToken method from Firebase takes a Firebase messaging instance, the previously created Vapid Key and optionally a service worker registration. By default, a service worker file called firebase-sw.js is used, but because a service worker handling the offline cache was registered earlier, the service worker registration is provided. The service worker requires some additions as well, which will be covered later. Additionally, the getTo-ken method also asks the user permission to send push notifications to their device if per-mission has not been granted yet. After receiving the token, a message listener is created with a separate method. The listener sets any received message to a state that the getTo-kens method receives as a parameter. The state change is handled in a component that will be discussed later. An API call is made to save the token to the database. It is stopped by a Firebase remote config flag, as the backend functionality handling message sending to the web application is not yet completed.

The service worker is needed for Firebase messaging to work at all. Figure 11 shows the sequence of action in the service worker.

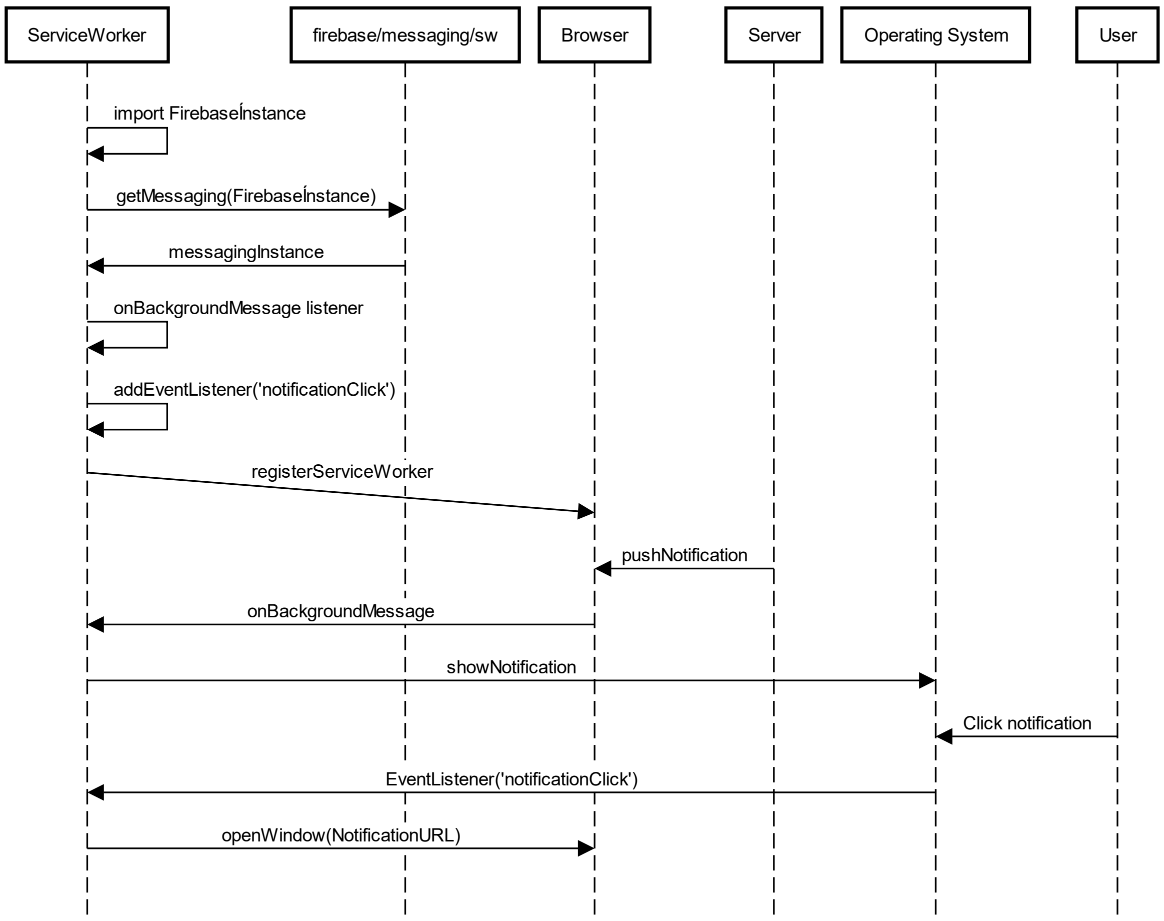


Figure 11. A sequence diagram of the firebase messaging service worker

First, Firebase messaging is initialized in a similar manner as previously, but the getMessaging method is from firebase/messaging/sw instead of firebase/messaging. This service worker specific messaging handles receiving messages by itself, so the next additions are not strictly necessary. Next, the onBackgroundMessage listener is used to customize the native operating system notification the user receives when the web app is not open or in focus. It is given the data found in the notification sent by the server and the web app’s icon as the notification icon. Additionally, if a link is included in the notification, it will be included, otherwise a link to the login page will be used. Then the customized notification is sent to the user with the service worker method of showNotification. Lastly, a listener that listens to when a user clicks on a notification is created. If a user clicks on a notification and has the web app open to the address specified in the notification, the web app is brought to focus. Otherwise, the app is opened to the address.

A new component was created to handle asking the user for permission to receive push notifications and for showing them (Figure 12).

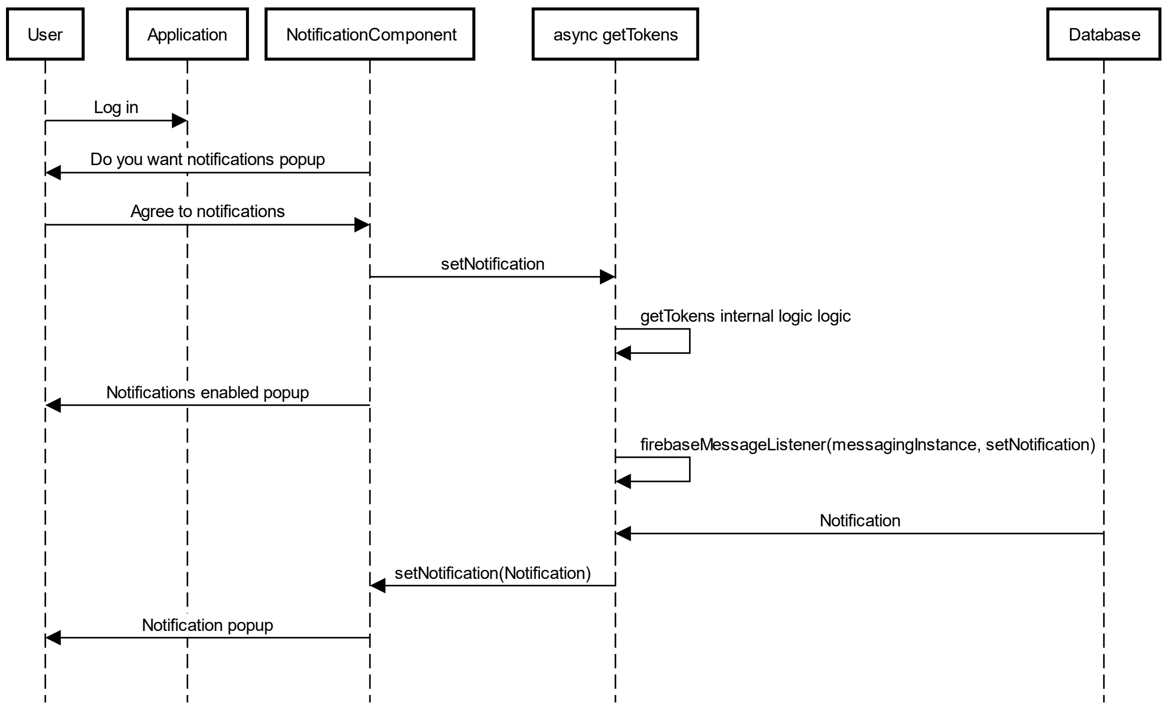

User
Application
NotificationComponent
async getTokens
Database
Log in
Do you want notifications popup
Agree to notifications
setNotification
getTokens internal logic logic
Notifications enabled popup
firebaseMessageListener(messagingInstance, setNotification)
Notification
setNotification(Notification)
Notification popup

Figure 12. A sequence diagram of the notification component

This component is rendered when a user has logged in. A popup asking them if they want to receive notifications is shown if certain conditions are met. The conditions are that the user has not explicitly denied notifications on an app level and has not dismissed the notification. The popup is not shown in development environments as service workers do not work in them. If the user closes the notification, the handleDoNotGetNotifications method is invoked causing the notification to not be show in the future, as well as, showing the user a popup saying that they will not receive notifications. If the receive notifications button in pressed, the method handleGetNotifications is invoked instead. The method calls the getTokens method defined earlier with the set method of the notification. Then a popup saying notifications enabled is show to the user. If the component is loaded and the user has accepted to receive notifications, the handleGetNotifications is called to handle receiving notifications. If a notification is received, another popup with the notification’s information is shown. Both popups are snackbar components with alert components inside them, which are from the Material UI component library.

3.2.2 Message history

The mobile app being replaced has a view showing all messages the current user has received, so naturally the web app must show the message history as well. The backend server saves sent messages to a database so the mobile app can retrieve the history through an API. The same API should be used in the web app as well.

A new React component was created for showing the list of messages. A fetcher hook from React Router is used to retrieve the messages. The fetcher works by using a specified URL's React Router loader. The loader is defined inside the routes file given to React Router. The URL used is provided to the loader and the loader makes the API call fetching the message history. The loaded data ends up in the fetcher hook and is assigned to the notifications state inside a useEffect hook. First method in the hook checks the retrieved data amount to see if the end of data was reached. Then if the fetcher is idle and data has been retrieved, the data is added to the notifications array. The fetcher call mentioned earlier happens inside a useEffect as well, this time happening if the request state is changed. The default values of the request are defined at the start of the file. An intersection observer is defined to load more data when a user has scrolled to the bottom of the loaded list. The ref variable that was gotten from the useInView call is a React ref that is set to an invisible div at the bottom of the message list. Once the div is in view, the inView state will update. The change of the inView state will cause the useEffect to change the request if certain conditions are met. These conditions are that the div at the end is in view, data is not currently being loaded, some data has been fetched and that the end of data has not been reached. The request is changed by the starting amount to get the next messages without reloading messages.

The user can change a couple of the request values with a set of filter components. These components are all from the Material UI component library. The user can choose what messages to see based on the delivery format of the message, the messages content, and a date range of when the message was sent. Changing any of these values calls the handleFilterChange method. This method first clears any loaded messages and then sets the end of data to false to remove messages loaded earlier and to restart the loading if all messages were previously loaded.

The messages are rendered with a Material UI list. If the end of the list is reached, skeleton components are show while the data is loading. The notifications are rendered using a custom notification component, which is a Material UI list item that can be clicked to open the link the notification may have attached to itself. The notifications message and subject are also shown.

3.2.3 Mobile compatibility

All the components created during this project were tested to be usable on Android and iOS devices. The components themselves are loaded into a responsive view that changes depending on the break points mentioned earlier. The notification permission was tested to be working without any specific handling for mobile devices. The app was installed successfully to both devices, as well as a Windows laptop.

4 Summary and conclusion

The primary goal was to implement parts of an existing native application into an existing web app. This was achieved by upgrading web app into a PWA. The web app was deemed to have met certain features of a PWA. The missing PWA features of offline usage, installability and re-engageability were implemented. Offline usage was done with service workers and re-engagement with firebase messaging.

The business viability of PWAs was investigated and found to be good. Many companies report increases in user retention as they switch to PWAs. Users seem to be more likely to install a PWA than a native application. This preference for PWAs most likely stems from their smaller install size and directness of the install when compared to an app store.

The application these PWA features were implemented to will not be released yet, as more features from the native mobile application need to be implemented. The added feature of offline usage will be considered when the app is developed further. The notifications will be developed further, as the backend is not yet capable of sending messages to the app.

References

Boduch, A. 2017. React and React Native: Use React and React Native to Build Applications for Desktop Browsers, Mobile Browsers, and Even as Native Mobile Apps. First edition. Birmingham: Packt.

Chand, M. 2020. What is a PWA (Progressive Web App) and Why Do We Need PWAs? Retrieved 19.04.2023. Available at https://www.c-sharpcorner.com/article/what-is-a-pwa/

Domes, S. 2017. Progressive Web Apps with React. Birmingham: Packt.

Dreyer, C. 2022. 3 Most Incredible Boosts By Lyft PWA. Retrieved 30.04.2023. Available at https://doccly.com/articles/3-most-incredible-boosts-by-lyft-pwa

Gambhir, A. & Raj, G. 2018. 'Analysis of Cache in Service Worker and Performance Scoring of Progressive Web Application'. International Conference on Advances in Computing and Communication Engineering (ICACCE). Paris, 294–299. Available at DOI 10.1109/icacce.2018.8441715

Gonzalez, J. 2013. Mobile First Design with HTML5 and CSS3: Roll Out Rock-Solid, Responsive, Mobile First Designs Quickly and Reliably. First edition. PACKT Publishing.

MDN Web Docs. 2023a. Making PWAs work offline with Service workers. Retrieved 22.3.2023. Available at https://developer.mozilla.org/en-US/docs/Web/Progressive_web_apps/Offline_Service_workers

MDN Web Docs. 2023b. How to make PWAs installable. Retrieved 25.4.2023. Available at https://developer.mozilla.org/en-US/docs/Web/Progressive_web_apps/Tutorials/js13kGames/Installable_PWAs

MDN Web Docs. 2023c. Lazy loading. Retrieved 9.5.2023. Available at https://developer.mozilla.org/en-US/docs/Web/Performance/Lazy_loading

MDN Web Docs. 2023d. Intersection Observer API. Retrieved 9.5.2023. Available at https://developer.mozilla.org/en-US/docs/Web/API/Intersection_Observer_API

Meta Open Source. Built-in React Hooks. Retrieved 9.5.2023. Available at https://react.dev/reference/react

Meta Open Source. lazy. Retrieved 9.5.2023. Available at https://react.dev/reference/react/lazy

Meta Open Source. State: A Component's Memory. Retrieved 9.5.2023. Available at https://react.dev/learn/state-a-components-memory

React Intersection Observer authors. 2023. react-intersection-observer readme. Retrieved 9.5.2023. Available at https://github.com/thebuilder/react-intersection-observer/blob/6c8111c7a04db5665898ea771e0b20c1a9c85467/README.md

Ritchie, R. 2018. App Store Year Zero: Unsweet web apps and unsigned code drove iPhone to an SDK. Retrieved 19.04.2023. Available at https://www.imore.com/history-app-store-year-zero

Russell, A. 2015. Progressive Web Apps: Escaping Tabs Without Losing Our Soul. Retrieved 8.2.2023. Available at https://infrequently.org/2015/06/progressive-apps-escaping-tabs-without-losing-our-soul/

Source, M.O. Synchronizing with Effects. Retrieved 9.5.2023. Available at https://react.dev/learn/synchronizing-with-effects

Statcounter. Desktop vs Mobile vs Tablet Market Share Worldwide. Retrieved 7.5.2023. Available at https://gs.statcounter.com/platform-market-share/desktop-mobile-tablet/worldwide/#yearly-2009-2022

Statcounter. Screen Resolution Stats Finland. Retrieved 26.04.2023. Available at https://gs.statcounter.com/screen-resolution-stats/all/finland/

RFC 8292. 2017. Voluntary Application Server Identification (VAPID) for Web Push. Internet Engineering Task Force. Available at https://datatracker.ietf.org/doc/html/rfc8292 [28 Apr 2023].

web.dev. 2017. Twitter Lite PWA Significantly Increases Engagement and Reduces Data Usage. Retrieved 30.04.2023. Available at https://web.dev/twitter/

World Wide Web Consortium. 2023. Web Application Manifest. Retrieved 25.4.2023. Available at https://www.w3.org/TR/2023/WD-appmanifest-20230329/

Wroblewski, L. 2009. Mobile First. Retrieved 22.2.2023. Available at https://www.lukew.com/ff/entry.asp?933