List-sharing Application

Focusing on Offline-Usage with the Concept of Progressive Web Application

- ListApp -

YH Nackademin

Front-End Development

FEND16, Examensarbete (Graduation project)

Author: Yoko Andrae

Mentor: Jesper Orb

March 17th, 2018

Table of Contents

Application's URL: https://list-app-f6838.firebaseapp.com

GitHub Repository: https://github.com/YAOrangeTime16/list-app

Abstract

The purpose of this project is to create a web application with the concept of Progressive Web Application (PWA) which is led by Google Developers. There are several factors defining PWA, and a few of those elements are related to Service Worker. Since the focus of this report is offline usage, the methods of application development with the implementations for offline usage are described, and finally, the suitability of offline usage is discussed.

1. Introduction

Communications via mobile applications, e.g. using social media services - such as Facebook, Twitter - occupy many parts of our daily life. As of January 2018 over half of all mobile users use messaging apps on a monthly basis[1] while the global mobile population amounted to 3.7 billion unique users[2]. A huge variety of mobile applications are available in the market[3], and mobile users choose and use applications depending on their needs.

With such a societal and mobile trends, my personal experience inspired the idea proposed in the present project. The story is this; when a group of people came to discuss organizing a party event, the communication started on *WhatsApp* (whatsapp.com). For those who did not have the application downloaded, messages were sent via another application, *Messanger* (messanger.com), as well. When it came to scheduling for a place, date and time, *Doodle* (doodle.com) appeared. To share a to-do list or to decide who would take what roles, *Google Doc* (google.com/drive/) was used. Seeing that several applications were used, I came up with the idea that all such functionalities could be gathered in one application. Moreover, the application should be one which anyone can use without the bother of downloading, and which is available anywhere and anytime.

[1] Statista. https://www.statista.com/topics/1523/mobile-messenger-apps/
[2] Statista. https://www.statista.com/topics/779/mobile-internet/
[3] Statista. https://www.statista.com/topics/1002/mobile-app-usage/

1-1. Background

Corresponding to the penetration of mobile devices and the increasing trend of mobile applications, web applications focusing on mobile devices are required to behave as native applications. There are several differences between web- and native applications, but the most obvious is that web applications are available on web browsers (such as Chrome, FireFox, Safari, etc.), and native applications are available at App Store or Google Store. Accordingly, web applications do not require users to download into their devices, whereas native applications do. That can be a disadvantage of native applications, and additionally, the availability of applications can differ between platforms (iOS, Android, etc.)[4] because different platforms require different programming languages for application development (Viswanathan, 2017). Thereby, it can happen that different device users fail to share the same application if it is not supported by either of the platforms. Web applications, to the contrary, are available for anyone or any device as long as they have an access to a web browser.

Doodle is actually a good application provided as a native-, and even as a web application: *It's free and a breeze to set up — people don't even need the app or an account to participate!* (Doodle on Google Play). Inspired by Doodle, my list-sharing application - ListApp - is also to be available and easy to use for anyone. There is one thing, though, that I realize when testing Doodle - it does not allow offline usage (as of the moment when this report is written). Unavailability of offline usage is a disadvantage of web applications[5] compared to native ones, as they are affected by the condition of the network connectivity. However, a concept of today's Progressive Web Application (PWA) solves this problem.

Progressive Web Application

Progressive Web Apps (PWA) are user experiences that have the reach of the web, and are: *Reliable*, *Fast*, and *Engaging*[6]. More detailed, LePage (Web Fundamentals) gives ten keywords regarding the traits of PWA[7]. Here are a couple of them, which are focused on in this project: *Connectivity Independence* (working offline or via low-quality networks) and *Freshness* (up-to-date data).

[4] Kunesh, A. (2017). *The 40 Best To-Do List Apps in 2017*. (The 19 Best Simple To-Do List Apps).
[5] https://en.yeeply.com/blog/advantages-and-disadvantages-of-web-app-development/
[6] Google. Progressive Web Application. https://developers.google.com/web/progressive-web-apps/
[7] https://developers.google.com/web/fundamentals/codelabs/your-first-pwapp

1-2. Purpose

The main purpose of this project is to create a web application - ListApp - with which a group of people can communicate specifically in order to coordinate an event / a meeting - without the need of gathering different applications. More importantly, the application should be available for anybody (regardless of platforms) and anywhere (even offline). Accordingly, ListApp needs to overcome one of the weaknesses of web applications, namely the network connectivity and subsequently the slowness of page loading. In this project, therefore, the concept of PWA is examined, and essential elements for offline usage are focused in a procedure of the application development.

2. Methods

2-1. Database structure

The first phase of the production procedure is to consider the database and application structure. This step helps to figure out how users should be led to the next page, or what data should be called by which action. Mockup[8] (used "Marvelapp.com") is an essential process to make a clear vision of the data structure as well as to visualize the application interface. The overall elements, such as the flow of data or pages, the actions of buttons and links, and the necessity of interactions with database, are determined in this stage. More detailed database structure is established, consequently.

A NoSQL[9] or JSON[10]-type database are deemed appropriate for the project, as JSON-type data are compatible with the client-side language, JavaScript. *Firebase*[11], backed by Google, provided a bunch of solutions for application development, and its Realtime Database structured the database of JSON-type. In this type of database, the data is saved as key-value style.

[8] *Mockups are a way of designing user interfaces (...) in computer images* (Wikipedia)
[9] *Database provides a mechanism for storage and retrieval of data that is modeled in means other than the tabular relations used in relational databases*. (Wikipedia: NoSQL)
[10] *JavaScript Object Notation (JSON) is an open-standard file format that uses human-readable text to transmit data objects consisting of attribute–value pairs and array data types* (Wikipedia: JSON)
[11] https://firebase.google.com/

Accordingly, the database structure for ListApp is outlined along with the style of Firebase Realtime Database (FRD).

2-2. Selection of Design Pattern and JavaScript Library

2-2-1. Single Page Application (SPA)

Before moving onto the description of the application development, there is a thing that might need to be considered in terms of design patterns, Single Page or Multiple Page Applications (Skólski 2016). For the most applications, quick performance is an essential factor to grab users. It gives a good user experience if the application behaves smoothly. Single Page Application (SPA) should be a good model to handle this factor. Although it slows because of the initial download of page assets, it moves fast and smoothly afterwards (Veeravalli, 2017). SPA is defined in Wikipedia[12] as follows: *A single-page application (SPA) is a web application or web site that interacts with the user by dynamically rewriting the current page rather than loading entire new pages from a server.*

Deferring from classic websites, which consist of several static pages sent from a server every time they are requested by the client-side[13]. SPA renders a website dynamically depending on requests from the client-side. In other word, it replaces any parts of the website with new resources without frequent interactions with the server or entire page reloading. To reduce the number of communications with the server-side - and to minimize the amount of content - loading could be key factors to increase the speed of the application behavior.

Single-page is a suitable design pattern for applications that are expected to work offline, as working offline implies no interactions via network nor with the server. An application only requests once and then all data will be stored. Any data can be effectively cached with SPA. Consequently, those data enable the application to work even offline (Skólski, 2016). Additionally, Skólski described that *[SPA] is an app that works inside a browser and does not require page reloading during use. (...) no page reloads, no extra wait time. It is just one web page that you visit which then loads all other content using JavaScript.* Once the contents are

[12] https://en.wikipedia.org/wiki/Single-page_application
[13] The word, “client” or “client-side”, is meant to be the contrast of “server” or “server-side”. Whereas a server receives and sends network resources in response to requests from a client-side, a client is usually a computer or device that sends requests to, and uses the resources sent from a server (ref. Computer Hope, Client)

loaded, the application is able to work using resources that it already holds inside itself, in other words, the loaded contents are saved in the client-side storage and ready to be retrieved locally. As a result, the application will work smoothly regardless the condition of the internet connection. This functionality corresponds to an element of PWA, *Connectivity Independence*.

2-2-2. React.js

Several JavaScript frameworks and libraries, - which handle SPA - are available, and React.js (hereafter referred to React) is one of them. React makes it possible to reduce the amount of page loading or server interactions by means of its trait of using *virtual-DOM*. The system of virtual-DOM is ideal and gives an effective solution in terms of the speed of page loading, and subsequently the speed of the application behavior.

DOM (Document Object Model) and Virtual DOM

A web page is a document. When a web page is opened in a browser, the browser builds up a model of the document's structure and uses this model to show the page on the screen[14], and DOM (Document Object Model) represents this document model - DOM is an object-oriented representation of the web page - so that DOM can manipulate the web page in a way of changing such as the document structure, style and content (DOM manipulation). These changes can be carried out by means of a scripting language, such as JavaScript[15]. However this DOM manipulation can make websites behave slower, and the slowness has become even worse as most JavaScript frameworks update the DOM much more than necessary[16]. This is why the virtual-DOM was adopted.

Codeacademy described that:

"Once the virtual DOM has updated, then React compares the virtual DOM with a virtual DOM snapshot that was taken right before the update. By comparing the new virtual DOM with a pre-update version, React figures out exactly which virtual DOM objects have changed. This process is called "diffing." Once React knows which virtual DOM objects have changed, then React updates those objects, and only those objects, on the real DOM." (Codecademy).

[14] http://eloquentjavascript.net/14_dom.html
[15] https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model/Introduction
[16] Virtual-DOM. https://www.codecademy.com/articles/react-virtual-dom

It is smart and effective to replace only diffs, and then to re-render the page partially rather than to replace the entire page. Since released by Facebook in 2013, React has been adopted by a number of applications and software companies, and its ecosystem has grown to be affluent. *Create-react-app (CRA)* is, among others, a useful tool for React application development. React, basically, requires pre-settings or configurations for developers, such as installing plugins and setting a file so that the React application could recognize and integrate them (e.g. npm packages, compiler etc). Moreover, this procedure is complicating and time-consuming. CRA, which is a starter kit, helps those processes on behalf of developers. It additionally integrates a useful plugin which automatically creates a *Service Worker* file for offline-usage (Service Workers are explained in Section 2-4 later on this report). All those aspects taken into account, using React is an appropriate choice for this project to achieve the speed of application's behavior as a PWA.

2-2-3. React Router v.4

ListApp has two main pages: one for "Group" and the other for "Admin". Fig.1 shows a blueprint of ListApp. The person marked as "Admin" has an account to control its groups and list data. It is eligible to give an ID and a password to each group's members. Each member is allowed to login to its allocated group with the given ID and password. Admin is also a part of members in each group and can always login to any of its groups from the admin's control page.

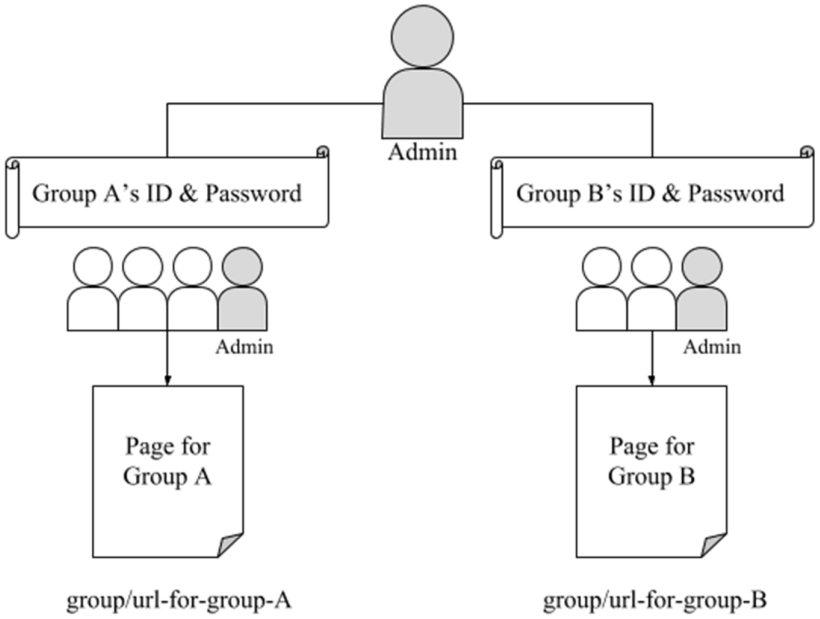

Fig.1. A plan of the application page structure.

Ideally, each group should have its own path, so the application has to handle the different paths, so to speak URLs, depending on which group a user login to.

A pure SPA has a disadvantage in regards to URL and navigation. The issue is that, *with the dynamic loading of modules, it is possible to change states without the browser knowing, and as a result essentially breaking [navigation feature of] the forward, back, and reload buttons of the browser* (Ingram-Westover 2015). When users browse websites, they often use the "back" or "forward" button on the browser to show the previous page, but a pure SPA does not handle such page history on its own. It needs HTML5 API, `history.pushState()` and `history.replaceState()` method, or hash-based routing (Ingram-Westover 2015). To overcome this issue, a routing JS library for React, called React Router, is implemented (version 4.2.2.) on ListApp. React Router should take care of the different paths such as group URLs.

Code snippet 1 shows the code at the part of routing, which is used for ListApp. (Some detailed code are omitted here for the sake of clarity)

```
--- index.js ------------------
 ReactDOM.render(
   <BrowserRouter>
     <App />
   </BrowserRouter>
 , document.getElementById('root'))

--- App.js --------------------
 <Switch>
     <Route exact path='/' render={()=> <ManageGroup />} />
     <Route path='/groups/:id' render={()=> <GroupPage />} />
     <Route path='/admin' render={()=> <ManageUser  /> } />
 </Switch>
```

Code snippet 1: The application's routing

As shown by Code snippet 1, three different paths are set in ListApp; default path [/] is used when the "GroupLogin" (where a user logins as a group member) in the "ManageGroup"

component is rendered, and it will be replaced by [*/groups/:id*] after a user logged in to a group page and subsequently the "GroupPage" component is rendered. For the whole "Admin" page (the component named "ManageUser"), the path of [*/admin*] is given.

Fig.2 shows the structure of the components in the present application. On the entrance page, two different login systems are implemented to be available for application users.

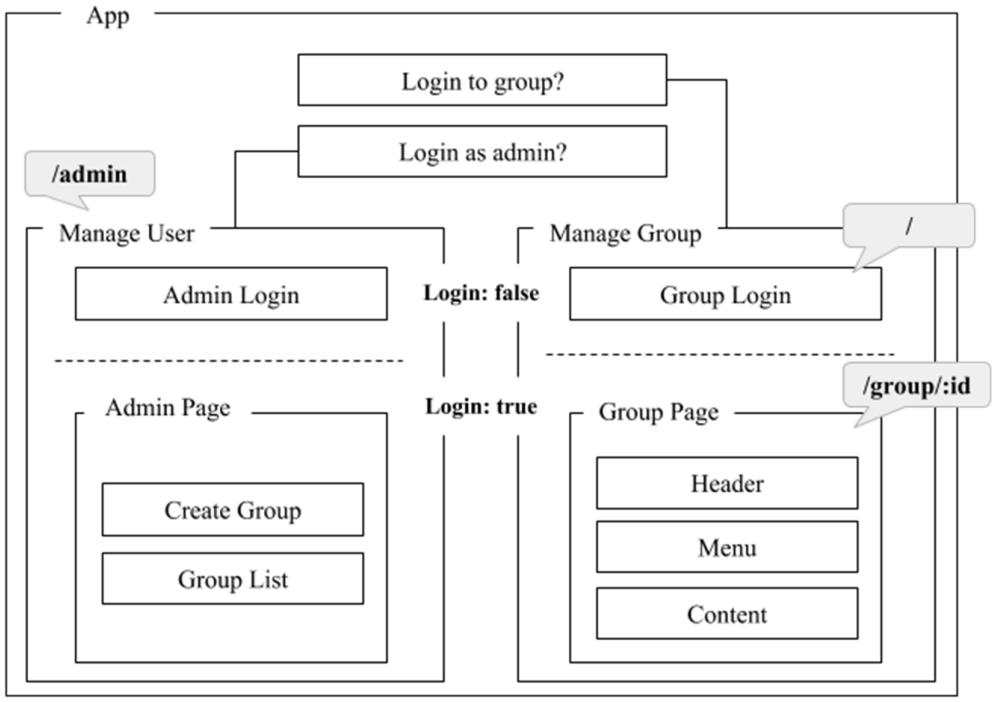

Fig.2. The component structure and routing paths in ListApp

2-3. Database and Authentication

2-3-1. Firebase

Firebase is a supporting tool for building application, and the product is provided under the name of Google. There are two main reasons why Firebase is chosen for the present project.

The primary reason is that FRD facilitates the offline work with its sync functionalities, and consequently, the data freshness would be retained, which is one of an essential factor for PWA. Data is synced regardless of platforms or devices, and remains available even offline[17]. It also

[17] https://firebase.google.com/docs/database/

supports offline detecting tool so that it is possible to branch off the way of data saving - or data fetching - depending on the network connection. Code snippet 2 shows the FRD detecting connection state. The referred data location `info/connected` is a special location to detect the device connectivity status, according to FRD:

```
var connectedRef = firebase.database().ref(".info/connected");
  connectedRef.on("value", function(snap) {
    if (snap.val() === true) {
      alert("connected");
    } else {
      alert("not connected");
    }
  });
```

Code snippet 2. Firebase Realtime Database (FRD) - detecting connection state

The second reason why Firebase is chosen is its beneficial authentication system. Authentication should be necessary in order to protect data inside the application, in this case, each group's data should be available only within the group itself. Group A, for example, is not supposed to see the data of Group B, and vice versa. The application also needs to identify whether a user is logged in as an Admin or a group member, so that it provides different types of data. The person - with an Admin account - should be able to create groups and lists of each group, and maintain those groups and lists, whereas group members should not. There are several ways for authentication available, and two of them are used for ListApp; *email and password based auth* is applied for admin login, and *anonymous auth* for group member login.

Firebase Auth has a function to inform the application of the user's login status. For example if a user re-opens the application to come back to the page, the application would recognize the user and retrieve the user information as long as the user has not logged out previously. Therefore, the page for auth users will be shown directly without requiring user login. This is investigated later in Section 2-5.

2-4. Service Worker

Where PWA is explained, *Service Worker* also comes along. According to Gaunt (Web Fundamentals), Service Workers provide the technical foundation that enables web applications to work like native applications, such as rich offline experiences, periodic background syncs,

push notifications. W3C (The World Wide Web Consortium) also describes that Service Workers are used *to make sites work faster and/or offline using network intercepting, [and] as a basis for other 'background' features such as push messaging and background synchronization*.

As it is mentioned earlier on this report, a benefit of using CRA (create-react-app) is the auto creation of a Service Worker. CRA integrates a plugin called *sw-precache-webpack-plugin*. This is basically a plugin of the JS bundler, *webpack*[18], and includes another plugin called *pw-precache*. What *pw-precache* does is to *generate service worker code that will pre-cache specific resources so they work offline*[19]. With the help of the plugin, CRA generates a Service Worker file which caches the bundled files, so to say css, js and media (image) files of its project.

2-4-1. Testing the Service Worker

Deployment

In order to test how the Service Worker works, the application is deployed, in other words, it is hosted on a server to be available on the web. Service Worker will be successfully registered only when the server has HTTPS (HTTP for secure communication[20]) setup. Firstly, *gh-pages*[21] was used to deploy the application, however, there was an issue that gh-pages did not support *browserHistory* (as of March 2018) which React Router had integrated from the ver.4. To overcome this issue there was an alternative way, such as to switch *BrowserRouter* to *HashRouter*, however, *HashRouter* made the application behave unexpectedly, such as `history.replace` in the code lost its functionality. Eventually, the application is deployed to Firebase Hosting (another Firebase service).

Cached Data

Fig.3 shows, with Chrome DevTools, that the Service Worker is registered. Moreover, Fig.4 shows that index.html, bundled-css, -js and -media files are cached. By means of *sw-precache-webpack-plugin*, all css-, js- and media- files are bundled by the JS bundler, *webpack*, and all

[18] https://webpack.js.org/
[19] https://github.com/GoogleChromeLabs/sw-precache
[20] https://en.wikipedia.org/wiki/HTTPS
[21] https://www.npmjs.com/package/gh-pages

static files - including those bundled data - are cached to the cache storage on the client-side by Service Worker that is also created by the plugin.

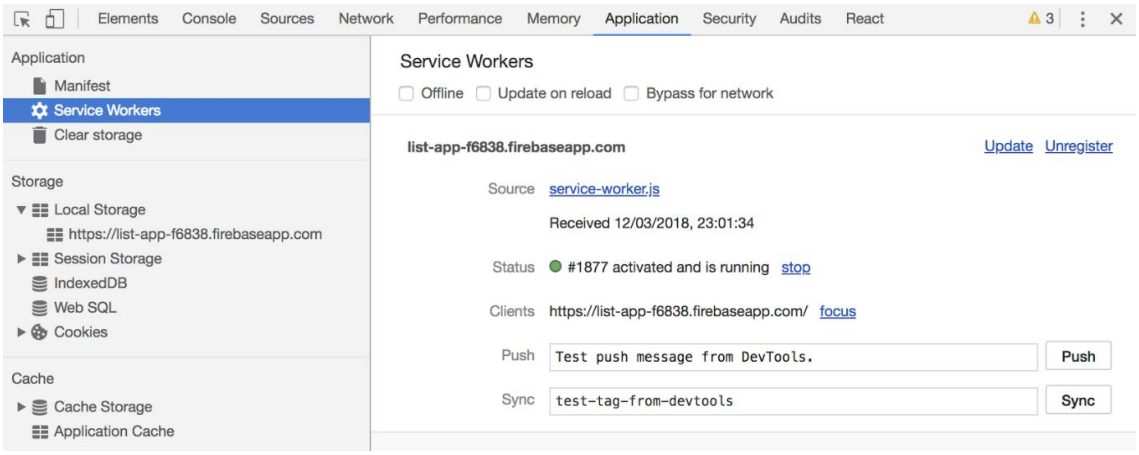

Fig.3. Registered Service Worker for the application

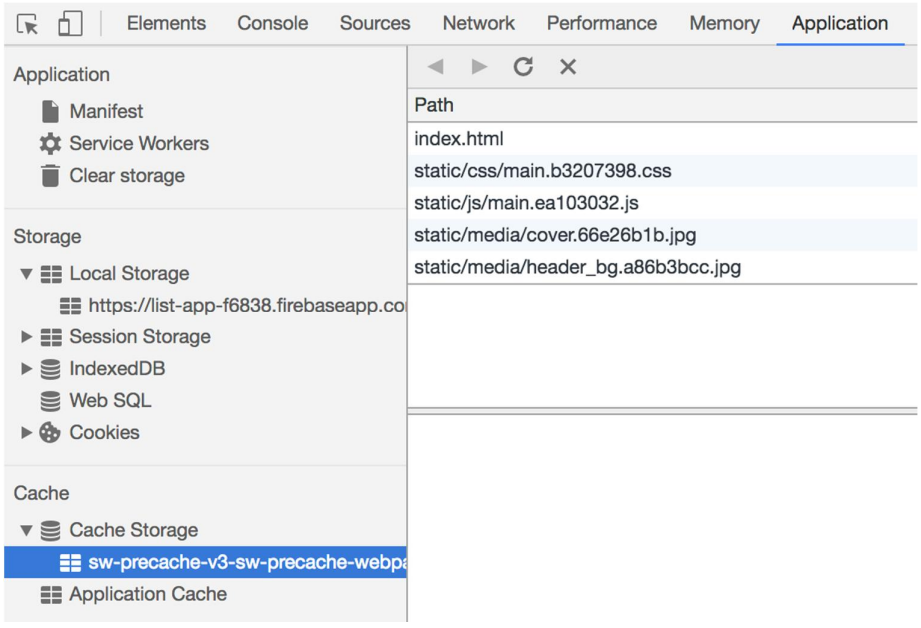

Fig.4. Cached files by CRA's service worker

2-5. Client-side Storage

2-5-1. Offline Test

In going to offline and reloading the browser, all cached files are rendered thanks to the Service Worker caching the static assets but not the dynamic data. The dynamic data, in this project,

are meant to be the list information (list title, description and items), and Fig.5 shows that the list and some information fail to be rendered.

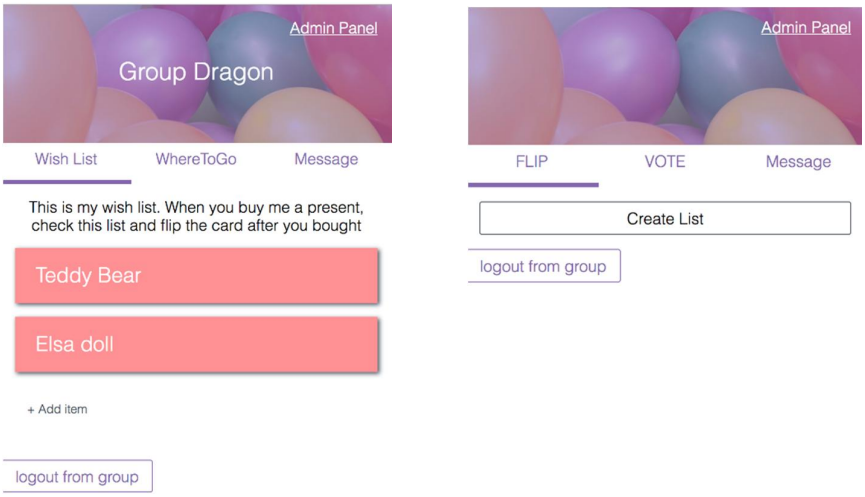

Fig.5. The snapshot on the left shows BEFORE page reload, and on the right shows AFTER page reload

This is caused as a result of the states and props of React. React's states and props hold data to be rendered, and those data are dynamically replaced with the response from the database in ListApp. As the database response is cut off by going offline, those states and props are left to be empty with the act of page reloading, and therefore, there is nothing to render. Only empty information is rendered, that means, no data are shown on the page.

It causes the same problem for the group login page. As the ListApp's Manage Group component is set to show the Group Login if the user is not logged in - the React's states of `loggedinAsMember` holds the value of *false* - otherwise Group Page should be rendered when `loggedinAsMember` holds *true* (Code snippet 3, see also Fig.2).

```
--- ManageGroup.js ---

render(){
 return(
  …
  {
   (loggedinAsMember) //true or false
     ? <Redirect to={`/groups/${groupId}`} />
     : <GroupLogin error={error} loginGroup={loginGroup} />
  }
 )
}
```

Code snippet 3. React props `loggedinAsMember` decides whether GroupPage or GroupLogin is rendered.

User's login information is saved in Local Storage by Firebase Auth, however, the application shows Group Login (Fig.6), instead of Group Page (Fig.7), at this stage of testing. This is because the prop of `loggedinAsMember` has been set to the default value of *false* by page reloading, and no actual data is retrieved by Firebase Auth because the device is offline.

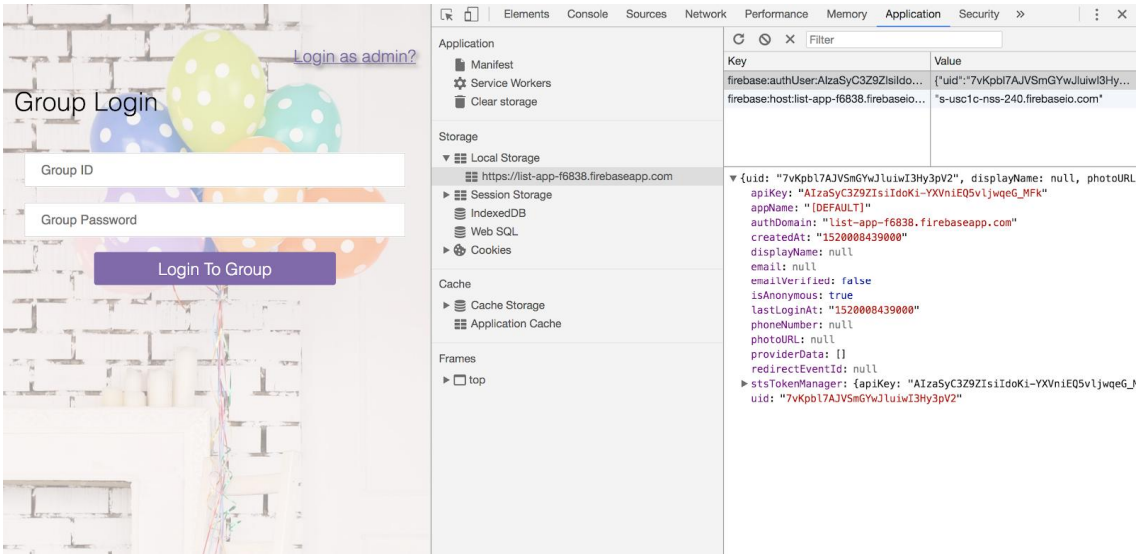

Fig. 6. The Group Login is rendered even though the user is still logged in.

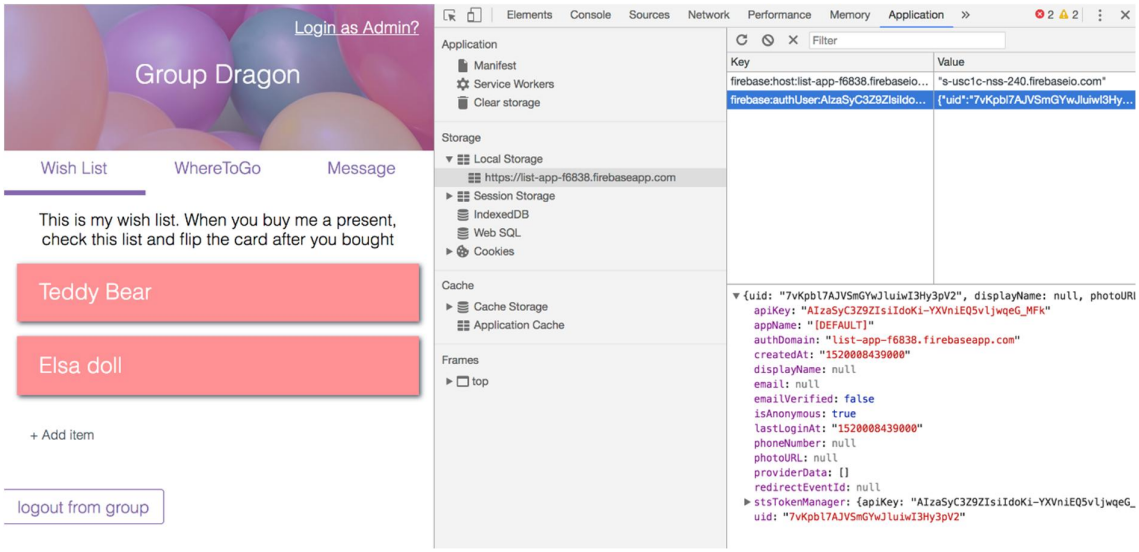

Fig. 7. This Group Page is supposed to be rendered instead, while user is logged in.

2-5-2. Local Storage

Firebase explains about its Auth feature that:

When a user signs up or signs in, that user becomes the current user of the Auth instance. The Firebase Auth instance persists the user's state, so that refreshing the page (in a browser) or restarting the application doesn't lose the user's information.[22]

The Firebase Auth saves the user's auth information into the client-side storage (also called *local storage*[23]). As local storage - the client-side storage - is supposed to hold its data even after the browser is closed, the user status is to be retrieved when the application is reopened. This user data are kept until `firebase.auth().signOut()` is called.

Thanks to this Firebase Auth functionality - saving data locally - ListApp holds the user information as long as the user remains logged in (also unless the user clears the storage on the browser). Therefore, the pages such as the Group Page and the Admin Page appear always even after the page is reloaded or the application is re-opened, so that the user can directly resume using it. However, this is only effective when the device is online. Under the offline status, the React's states will not be set dynamically so that the application fails to render the pages as it is supposed to, meaning it renders the Login, instead of the Group Page.

To show the Group Page while the user is logged in, the React's state, `loggedinAsMember`, needs to be set to *true*, and it can accordingly be done by referring the local storage. In the same way, to solve the issue of rendering of the list information (Fig.5), list data need to be saved locally as well, so that information - which is used to render the React components - is set simply by the data in the local storage.

2-5-3. IndexedDB & LocalForage

A local storage is available and utilized by Web Storage API, and it is supported by most modern browsers - except FireFox Mobile (as of March 2018)[24]. It is a good place to save data

[22] https://firebase.google.com/docs/auth/users
[23] https://developer.mozilla.org/en-US/docs/Web/API/Storage/LocalStorage
[24] https://developer.mozilla.org/en-US/docs/Web/API/Web_Storage_API/Using_the_Web_Storage_API

locally - on the client-side. However, any data need to be saved as “strings”, meaning that objects and arrays need to be “stringified” with the use of `JSON.stringify` before they are saved to the storage[25].

IndexedDB, on the other hand, has a capacity to save the significant amounts of different types of data even blobs (such as media, audio and images)[26] and files. Unlike a synchronous localStorage - which blocks applications when data are saved and retrieved - IndexedDB is asynchronous. However, its usages are very complicated.[27][28] Therefore, there are libraries available for simplifying the usage of IndexedDB. Among others, *localForage* has a simple syntax as localStorage, and also capabilities of IndexedDB.

Accordingly - some retrieved data that are to be used for rendering - such as group and list information, are saved to IndexedDB (Fig.8) with the help of localForage. Those data should be retrieved when the device is detected as offline.

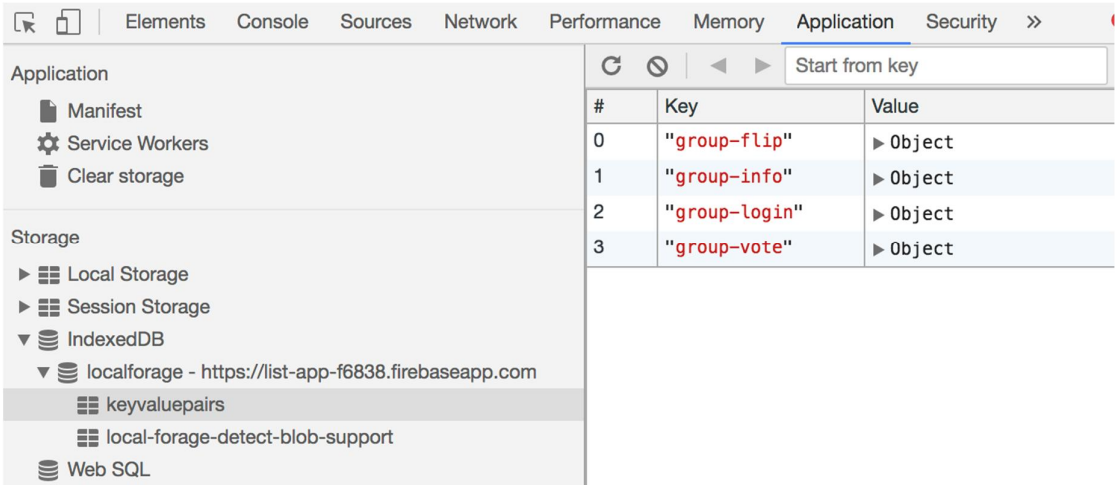

Fig.8. Client-side storage, IndexedDB, holds data saved by localForage

The list rendering issue with page reloading at offline status is solved in this way, however, the data update in the offline status has not been dealt with yet. Data - which are added, removed or updated during offline - need to be sent to FRD as soon as the device comes back to online. This can be handled in the background of the application, by means of *background sync*.

[25] https://pusher.com/tutorials/pwa-react/
[26] https://en.wikipedia.org/wiki/Binary_large_object
[27] https://developer.mozilla.org/en-US/docs/Web/API/IndexedDB_API
[28] http://blog.teamtreehouse.com/using-localforage-offline-data-storage

2-6. Background Sync

For the case of this application - ListApp - the list item can be added, or users can pick up items (= alteration to the list item status) even when the device is offline. If the application does not support background sync, users would be frustrated by low- or no-connection, or they need to wait until the device comes online to change item status. Such offline data update can be achieved by *background sync*.

Background sync uses a Service Worker listening for an event, 'sync', to be fired when the device regains connectivity (Archibald, 2015). To use this functionality, the event of 'sync' needs to be added to the Service Worker. However, background sync is not integrated in CRA's Service Worker (generated automatically by *sw-precache-webpack-plugin*), so that it has to be customized.

To customise any of CRA's configurations, it is usually recommended to eject the project, that means, any further configurations and scripts have to be maintained manually (facebook/create-react-app on GitHub). After some investigations, one utility tool was found available, *cra-append-sw*. It appends customized Service Workers to CRA's Service Worker, and therefore ejecting the project can be avoided (cra-append-sw on GitHub).

Firstly, the 'sync' event tags needs to be added to the CRA's registerServiceWorker.js file so that tags would be registered for use, in this case (code snippet 4), `swRegistration.sync.register('tag name')` is the part. Then, in the separate custom service worker file (Code snippet 5), the code contains functions (`sendAddedItemToFlie`, `sendAddedItemToVote` and `sendNewStatusToVote`) that deal with five tasks;

1. Listening 'sync' event fire,
2. matching the event tag,
3. retrieving local data,
4. sending them to firebase database,
5. removing those local data.

Importantly, in advance, the local data, created during offline to be sent to the database, needs to be saved independently from other local data that is used for rendering for offline.

```
function registerValidSW(swUrl) {
  navigator.serviceWorker
    .register(swUrl)
    .then(registration => {
      ...
    })
    .catch(error => {
      ....
    });
    navigator.serviceWorker.ready.then((swRegistration) => {
      swRegistration.sync.register('updateNewItemFlip');
      swRegistration.sync.register('updateNewItemVote');
    })
}
```

Code snippet 4: the registerValidSW function in the registerServiceWorker.js file of CRA

```
self.addEventListener('sync', function(event) {
  if (event.tag === 'updateNewItemFlip') {
    event.waitUntil(sendAddedItemToFlip());
  }
  if (event.tag === 'updateNewItemVote') {
    event.waitUntil(sendAddedItemToVote());
    event.waitUntil(sendNewStatusToVote());
  }
});
```

Code snippet 5: ListApp's custom Service Worker file

2-7. Limitations

Firstly, owing to the short time frame of the present project, not all elements of PWAs - such as push notifications - are investigated. Moreover, the present application only works partly offline. However, the most important aspects and elements - that are necessary for offline usage - are examined in this project and will be implemented within the application.

Secondly, ListApp is to include the feature of instant messaging with the offline functionality to facilitate group communications as *WhatsApp* or *Messenger* provides. Owing to the same reason as above, it is short of time to implement the feature.

Thirdly, the project assumes that a browser supports Service Worker API although the API is not fully supported by all browsers[29] as of March 2018. The project is only developed, run and tested on Chrome (v.64). However, there is a way to help non-supporting browsers by using progressive enhancement (Archibald, 2015).

3. Result

ListApp is developed with the JS library, React.js. As its design pattern adopts SPA, React Router is integrated to make browsers behave as users usually expect. The application has mainly two login sections, one for Group members and the other for Admin (Fig.9). The application has been deployed at https://list-app-f6838.firebaseapp.com. The Service Worker is registered when the website is visited. The offline functionalities perform after a user logs in.

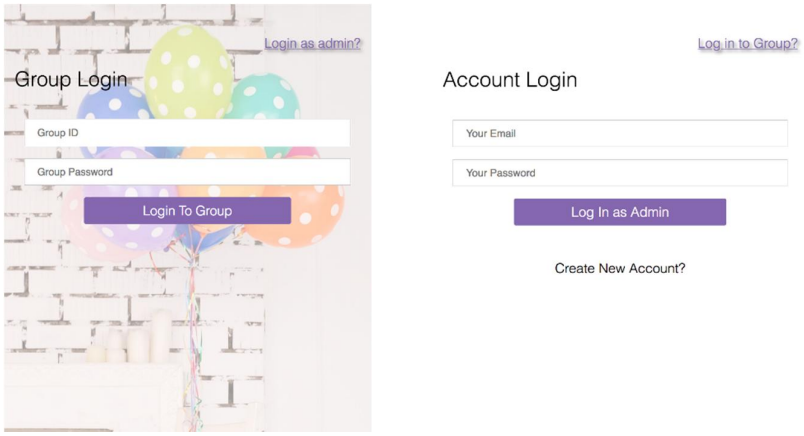

Fig.9. Group login on the left, and Admin login on the right.

With the outcome of the offline testing (see Fig.6 in Section 2-5-1), the code is written in a way that the user's login information is retrieved from the local storage - `localforage.getItem('key')` - and set to the React states and props - `this.setState()`- (Code snippet 6) and, as a result, the page renders successfully the Group Page and its list information with the act of page reload even when the device is offline (Fig.10).

[29] https://developer.mozilla.org/en-US/docs/Web/API/Service_Worker_API

```
_getGroupInfo = ()=>{
    //check network connection
    const connectedRef =
firebase.database().ref('.info/connected');
    connectedRef.on('value', connection => {
      if (connection.val() === false) { //offline
        localforage.getItem('group-login').then(status => {
          if(status.loggeinasMember){
            //check the local storage
            localforage.getItem('group-flip').then(flipInfo =>
{
              flipInfo && this.setState({flipList: flipInfo})
            })
            localforage.getItem('group-vote').then(voteInfo =>
{
              voteInfo && this.setState({voteList: voteInfo})
            })
            localforage.getItem('group-info').then(groupInfo =>
{
              groupInfo && this.setState({groupInfo})
            })
          }
        })
      } ...
```

Code snippet 6. React states are set by the data retrieved from the local storage.

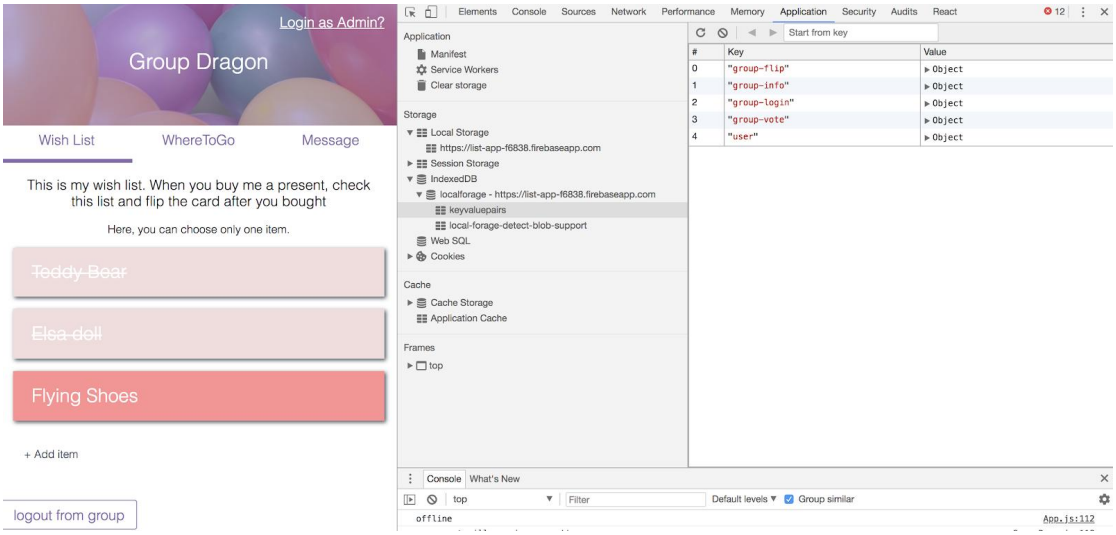

Fig.10. Page reloaded offline - React's states are set by local data.

For updating / changing item status (Fig.11) offline, FRD successfully syncs the changes when the device goes back to online with the background sync feature implemented in the custom Service Worker.

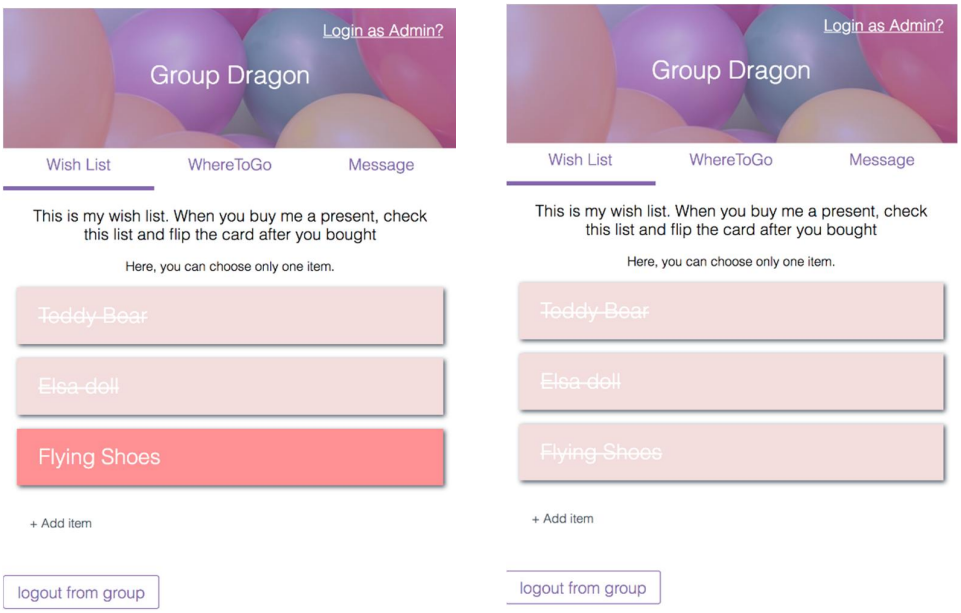

Fig.11. The status of the third item is changed, and it is synced with FRD in the background.

All those local data are removed when the user logs out.

4. Discussion

Offline technology development - which came along with the trend of web applications - is an eye-opener. It will be more common to use this technology so that the discrepancies of the user experience between web- and native applications will be smaller and smaller. Moreover, compared to desktop computers, mobile networks simply do not yet provide the same level of quality and consistency. Accordingly, temporary disconnection or slow service should be regarded when using mobile networks (OfflineFirst org.). As PWAs are built with the core tenet of progressive enhancement (LePage, Web Fundamentals) - the basic idea of progressive enhancement is to start with an assumption of base-level capabilities, and then take advantage of more device capabilities as they become available (Holt, 2017). This project application is developed based on mobile-first - features and design enhanced from the mobile-device-base. It is thereby handling the flaky network connectivity and contain features which enable offline usage. This is important to overcome the instability of network connectivity.

However, offline feature might not be suitable for all cases for mobile applications. Examples include if there is a necessity that several users share the same information or data should be

updated on a real time basis. How far an application works offline - or allows users to use offline - can much depend on the purpose of the application or the information to be delivered. With the present application, some functionalities are suitable for offline use, but some are not. It shares two different types of lists within a group. One is a list of flip-type and the other is of vote-type. Flip-list allows only one of the group members to take each item, whereas items of vote-list can be taken by several members. Accordingly, the flip-list is probably not appropriate for total offline usage as it is based on the first-come-first-served system. The list item status needs to be shared among group members on real time, and consequently any time lag of the data syncing can cause a bad user experience. In this case, the key point here is not "how much the feature allows users to use offline as possible" but it might be "how the feature lets users know limitations of the offline usage" without damaging user experience. In this way, it might be why the mobile application *Doodle,* limits its offline-usage. As Feyerke (2013) argues: *Stop treating a lack of connectivity like an error, (...) make sure error messages hit the right tone*, to give notifications or messages to users at a proper timing with a proper tone is also a factor to be considered when developing PWAs.

It is also a lesson learned that there are many areas and concepts that need to be taken into account when developing an application. All aspects of where the web application runs - such as different browsers, devices, and the condition of network connectivity - affect the application's behavior and availability. There are, accordingly, many possible concepts and patterns to be considered – desktop-first / mobile-first, progressive-enhancement / graceful-degradation, offline-first, usability, availability, etc. – and those need to be selected carefully depending on the target user for the application.

Additionally, Google Developers states that the concept of Progressive Web Applications is based on user experience. It is, indeed, convinced by the present investigation. Targeting mobile users - especially those with a weak or no network connectivity - the offline functionality of PWA's provides better user experience for those targets.

5. Conclusion

With the help of investigated technologies and tools, *Connectivity Independence* and *Freshness* of PWA are achieved for ListApp. However, there is one disadvantage to be acknowledged: the

degree of data freshness is limited because time lag of database syncing is unavoidable under the offline usage, and it depends on the network conditions where devices exist.

It is technically possible to give the web application - ListApp - a full offline usage, however, when it comes to user experience, the suitability for offline usage will vary from application to application. Accordingly, with the acknowledgment for the aspect of offline usage along with user experience, and subsequently providing well planned features and appropriate messages, ListApp would be helpful and useful for event coordination for anybody, any device and anywhere.

6. Next steps

It would be interesting to develop ListApp further and minimize its limitations. Regarding background sync, Firebase is going to launch a new feature called *Cloud Firestore*, and its beta version is already available as of March 2018. This new feature takes care of offline sync, exactly doing the task that this project examined. It would be worth looking at.

References

1. Agrawal, T. (2017). *Why Progressive Web Apps Are The Future Of Web Development*. freeCodeCamp. [Forum]. December 15. https://medium.freecodecamp.org/why-progressive-web-apps-are-the-future-of-web-development-13db7dd5f640 [Accessed 2018-02-06]

2. Archibald, J. (2014). *The offline cookbook*. [Blog]. December 9. https://jakearchibald.com/2014/offline-cookbook/ [Accessed 2018-03-03]

3. Archibald, J. (2015). *Background sync demo in Chrome.* [Online Video] 8 December. https://www.youtube.com/watch?v=l4e_LFozK2k&feature=youtu.be [Accessed 2018-02-06]

4. Archibald, J. (2015). *Introducing Background Sync*. Web Updates. [Guide]. December. https://developers.google.com/web/updates/2015/12/background-sync [Accessed 2018-03-02]

5. Archibald, J. (2016). *Instant Loading: Building offline-first Progressive Web Apps - Google I/O 2016*. Google Chrome Developers. [Online Video]. May 20. https://www.youtube.com/watch?v=cmGr0RszHc8 [Accessed 2018-02-19]

6. Awad, B. (2017). *React Router v4 Tutorial*. [Online Video]. August 3. https://www.youtube.com/watch?v=l9eyot_IXLY [Accessed 2018-02-19]

7. Bidelman, E. (2011). *navigator.onLine in Chrome Dev channel*. Web Fundamentals. [Guide]. July. Updated: January 3, 2018. https://developers.google.com/web/updates/2011/06/navigator-onLine-in-Chrome-Dev-channel [Accessed: 2018-02-25]

8. Codecademy. *React: Virtual-DOM - Fighting Wasteful DOM Manipulation*. [Guide]. https://www.codecademy.com/articles/react-virtual-dom [Accessed 2018-03-08]

9. CoderJourney. (2017). *Get Started Using React Router*. [Online Video]. July 4. https://www.youtube.com/watch?v=BImhGRTunpY [Accessed: 2018-02-21]

10. CodeWithTim. (2017). *React Router Version 4: Link Components*. [Online Video]. November 20. https://www.youtube.com/watch?v=2drsTBFZTQE [Accessed: 2018-02-19]

11. CodeWithTim. (2017). *React Router Version 4*: An Introduction. [Online Video]. June 24. https://www.youtube.com/watch?v=VdyORTskPGA [Accessed: 2018-02-15]

12. Computer Hope. Client. Updated: 2017-12-29. https://www.computerhope.com/jargon/c/client.htm [Accessed 2018-03-08]

13. Dascalescu, D. (2016). *Why "Progressive Web Apps vs. native" is the wrong question to ask.* Medium. [Forum]. August 23: Updaterad februari 2018. https://medium.com/dev-

channel/why-progressive-web-apps-vs-native-is-the-wrong-question-to-ask-fb8555addcbb [Accessed 2018-02-06]

14. Eluwande, Y. (2017). *Build a realtime PWA with React*. Pusher. [Blog]. September 14. https://blog.pusher.com/build-a-realtime-pwa-with-react [Accessed: 2018-02-26]

15. Facebook. *Making a Progressive Web App*. GitHub. [Guide]. Updated: February 4, 2018. https://github.com/facebook/create-react-app/blob/master/packages/react-scripts/template/README.md#making-a-progressive-web-app [Accessed: 2018-02-19]

16. Feyerke, A. (2013). *Designing Offline-First Web Apps*. A list Apart. [Forum]. December 4. http://alistapart.com/article/offline-first [Accessed: 2018-02-25]

17. Feyerke, A.(2013). *Say Hello To Offline First*. Hoodie. [Blog]. November 5. http://hood.ie/blog/say-hello-to-offline-first.html

18. Firebase Docs. https://firebase.google.com/docs/ [Accessed: 2018-02-17]

19. Gaunt, M. *Service Workers: an Introduction*. Google Developers, Web Fundamentals. Updated: February 22, 2018. https://developers.google.com/web/fundamentals/primers/service-workers/ [Accessed 2018-03-03]

20. GeekyAnts. *Progressive Web Apps—The Next Step in Web App Development: A Companion Post to Neeraj Singh's talk at ReactFoo Pune 2018*. Hacker Noon. January 30. https://hackernoon.com/progressive-web-apps-the-next-step-in-web-app-development-372235bf9a99 [Accessed 2018-03-07]

21. Google Chrome Labs. *sw-precache*. https://github.com/GoogleChromeLabs/sw-precache [Accessed 2018-03-03]

22. Haverbeke, M. Eloquent JavaScript. *The Document Object Model*. Chapter 14. http://eloquentjavascript.net/14_dom.html [Accessed 2018-03-08]

23. Holt, B. (2017). *Offline-first web and mobile apps: Top frameworks and components*. TechBeacon. January 24. https://techbeacon.com/offline-first-web-mobile-apps-top-frameworks-components [Accessed 2018-03-09]

24. Horo, T. (2016). *ServiceWorker の Background Sync でオンライン復帰時にデータ送信 [Sending data with Service Worker for Background Sync when regained the network connection]*. Qiita. [Forum]. April 18. https://qiita.com/horo/items/28bc624b8a26ffa09621 [Accessed 2018-03-04]

25. Hume, D. (2016). *ServiceWorker: A Basic Guide to BackgroundSync*. PonyFoo.com July 19. https://ponyfoo.com/articles/backgroundsync [Accessed: 2018-02-25]

26. Ingram-Westover, A. (2015). *Technically Speaking: The Pros and Cons of Single Page Applications (SPAs)*. DialogTech. [Blog]. June 10.

https://www.dialogtech.com/blog/technically-speaking/technically-speaking-the-pros-and-cons-of-single-page-applications-spas [Accessed: 2018-03-07]

27. JSStore. http://jsstore.net [Accessed: 2018-02-26]

28. Karlin, J., Kruisselbrink, M. (2016). *Web Background Synchronization*: *Draft Community Group Report.* Web Platform Incubator Community Group under the W3C Community Contributor License Agreement (CLA). August 2. https://wicg.github.io/BackgroundSync/spec/ [Accessed 2018-02-07]

29. Kirupa. (2017). *Creating a Single-Page App in React using React Router*. November 5. https://www.kirupa.com/react/creating_single_page_app_react_using_react_router.htm [Accessed 2018-02-21]

30. Kunesh, A. (2017). *The 40 Best To-Do List Apps in 2017 - From Simple Task Lists to GTD*. Zapier. May 18. https://zapier.com/blog/best-todo-list-apps/ [Accessed 2018-03-08]

31. LePage, P. *Your First Progressive Web App*. Web Fundamentals. [Guide]. Updated: 2018-02-28 https://developers.google.com/web/fundamentals/codelabs/your-first-pwapp/ [Accessed 2018-03-07]

32. Linnyk, Y. (2017). *All you need is React & Firebase*. CodeMentor. [Community]. February 13. Updated: 2017-08-22. https://www.codementor.io/yurio/all-you-need-is-react-firebase-4v7g9p4kf [Accessed: 2018-03-01]

33. LocalForage. https://localforage.github.io/localForage/ [Accessed 2018-02-26]

34. Lowson, N. (2016). *How to think about databases*. Read the Tea Leaves. [Blog]. February 8. https://nolanlawson.com/2016/02/08/how-to-think-about-databases/ [Accessed: 2018-03-01]

35. McGinnis, T. (2018). *URL Parameters with React Router*. TylerMcGinnis.com. January 16. https://tylermcginnis.com/react-router-url-parameters/ [Accessed 2018-02-15]

36. Mozilla. *Client-side storage*. MDN web docs. [Guide]. https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Client-side_web_APIs/Client-side_storage [Accessed: 2018-03-07]

37. Mozilla. *IndexedDB API*. MDN web docs. [Guide]. https://developer.mozilla.org/en-US/docs/Web/API/IndexedDB_API [Accessed: 2018-02-26]

38. Mozilla. *IndexedDB を使用する*. MDN Web docs. [Guide]. https://developer.mozilla.org/ja/docs/Web/API/IndexedDB_API/Using_IndexedDB [Accessed: 2018-02-26]

39. Mozilla. *Introduction to the DOM*. MDN web docs. [Guide]. https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model/Introduction [Accessed: 2018-03-08]

40. Mozulla. *LocalStorage*. MDN web docs. [Guide]. https://developer.mozilla.org/en-US/docs/Web/API/Storage/LocalStorage [Accessed: 2018-03-08]

41. Mozilla. *Service worker API*. MDN web docs. [Guide]. https://developer.mozilla.org/ja/docs/Web/API/ServiceWorker_API [Accessed: 2018-02-24]

42. Mozilla. *Using the Web Storage API*. MDN web docs. [Guide]. https://developer.mozilla.org/en-US/docs/Web/API/Web_Storage_API/Using_the_Web_Storage_API [Accessed: 2018-03-08]

43. Naylar, I. (2017). *Progressive Web Apps vs Native Apps – Who Wins?* AppInstitute. November 3.https://appinstitute.com/pwa-vs-native-apps/ [Accessed: 2018-03-07]

44. Neagoie, A. (2016). *How to Store User Passwords and Overcome Security Threats in 2017*. Rangle.io. [Blog]. December 23. http://blog.rangle.io/how-to-store-user-passwords-and-overcome-security-threats-in-2017 [Accessed: 2018-02-18]

45. Npm. *byscriptjs*. https://www.npmjs.com/package/bcryptjs (v.2.4.3)[Accessed: 2018-02-18]

46. Npm. *gh-pages*. https://www.npmjs.com/package/gh-pages [Accessed 2018-02-28]

47. Offline First Org. *We live in a disconnected & battery powered world, but our technology and best practices are a leftover from the always connected & steadily powered past*. http://offlinefirst.org/ [Accessed 2018-03-09]

48. O'Shaughnessy, P. (2017). *6 myths of Progressive Web Apps*. Samsung Internet Developers. October 11. https://medium.com/samsung-internet-dev/6-myths-of-progressive-web-apps-81e28ca9d2b1 [Accessed 2018-02-06]

49. Osmani, A. (2016). *Progressive Web Apps with React.js: Part 3—Offline support and network resilience*. Medium. [Forum]. October 15. https://medium.com/@addyosmani/progressive-web-apps-with-react-js-part-3-offline-support-and-network-resilience-c84db889162c [Accessed 2018-03-03]

50. Osmani, A.,Cohen, M., *Offline Storage for Progressive Web Apps*. Web Fundamentals. [Guide]. Updated: 2018-02-28. https://developers.google.com/web/fundamentals/instant-and-offline/web-storage/offline-for-pwa [Accessed 2018-03-01]

51. Patel, C. (2015). *What Compelled Single Page Applications to Jump into Trends*. Bacancy Technology. January 27. https://www.bacancytechnology.com/blog/what-compelled-single-page-applications-to-jump-into-trends/ [Accessed: 2018-03-07]

52. Refsnes Data. *JavaScript Window Navigator*. W3Schools. [Guide]. https://www.w3schools.com/js/js_window_navigator.asp [Accessed: 2018-02-24]

53. Reddit. (2017). *Service workers and create react app?*. /r/reactjs. December. https://www.reddit.com/r/reactjs/comments/7hrl5r/service_workers_and_create_react_app/ [Accessed 2018-03-03]

54. Riley MacPherson, M., Nyman, R., Fabbro, A. (2014). *localForage: Offline Storage, Improved*. February 12. https://hacks.mozilla.org/2014/02/localforage-offline-storage-improved [Accessed: 2018-02-26]

55. Sherman, P. (2017). *A Simple React Router v4 Tutorial*. Medium. [Guide]. March 11. https://medium.com/@pshrmn/a-simple-react-router-v4-tutorial-7f23ff27adf [Accessed: 2018-02-18]

56. Sherman, P. (2017). *A little bit of history*. Medium. [Forum]. March 15. https://medium.com/@pshrmn/a-little-bit-of-history-f245306f48dd [Accessed: 2018-02-18]

57. Silver, A. (2014). *The disadvantages of single page applications*. August 11. https://adamsilver.io/articles/the-disadvantages-of-single-page-applications/ [Accessed 2018-02-06]

58. Skólski, P. (2016). *Single-page application vs. multiple-page application*. Neoteric. [Blog] December 1. https://neoteric.eu/single-page-application-vs-multiple-page-application [Accessed 2018-02-06]

59. Smart Insights. (2018). *5 features of progressive web apps*. January 8. https://www.smartinsights.com/mobile-marketing/app-marketing/5-features-progressive-web-apps/ [Accessed 2018-02-06]

60. Statista. Mobile Messenger Apps - Statistics & Facts. [Accessed 2018-03-09]

61. Statista. Mobile Internet Usage Worldwide -Statistics & Facts.[Accessed 2018-03-09]

62. Statista. Mobile App Usage - Statistics & Facts.[Accessed 2018-03-09]

63. Veeravalli, S. (2017). *Measuring and Optimizing Performance of Single-Page Applications (SPA) Using RUM*. LinkedIn Engineering. [Forum]. February 2. https://engineering.linkedin.com/content/engineering/en-us/blog/2017/02/measuring-and-optimizing-performance-of-single-page-applications [Accessed 2018-03-09]

64. Viswanathan, P. (2017). *Native Apps vs. Web Apps: What is the Better Choice?* Lifewire. [Forum]. December 19. https://www.lifewire.com/native-apps-vs-web-apps-2373133 [Accessed 2018-03-08]

65. Vrinsoft Technology. (2017). *What are the difference between web app and mobile app?* Quora. [Forum]. September 2. https://www.quora.com/What-are-the-difference-between-web-app-and-mobile-app [Accessed 2018-02-07]

66. West, M. *Using localForage for Offline Data Storage.* Treehouse. [Forum]. http://blog.teamtreehouse.com/using-localforage-offline-data-storage [Accessed: 2018-02-26]

67. Wikipedia. *Binary Large Object*. https://en.wikipedia.org/wiki/Binary_large_object [Accessed 2018-03-08]

68. Wikipedia. HTTPS. https://en.wikipedia.org/wiki/HTTPS [Accessed 2018-03-08]

69. Wikipedia. *JSON*. https://en.wikipedia.org/wiki/JSON [Accessed 2018-03-08]

70. Wikipedia. *Mockup*. https://en.wikipedia.org/wiki/Mockup [Accessed 2018-03-08]

71. Wikipedia. *NoSQL*. https://en.wikipedia.org/wiki/NoSQL [Accessed 2018-03-08]

72. Wikipedia. *Single Page Application*. https://en.wikipedia.org/wiki/Single-page_application [Accessed 2018-03-08]

73. WICG. *Background synchronization explained*. GitHub. [Guide]. Updated: March 28, 2017. https://github.com/WICG/BackgroundSync/blob/master/explainer.md [Accessed: 2018-02-25]

74. World Wide Web Consortium. *Service workers explained*. GitHub. [Guide]. Updated: November 2, 2017. https://github.com/w3c/ServiceWorker/blob/master/explainer.md [Accessed: 2018-02-24]

75. Yamada, Y. (2016). *Service Worker、はじめの一歩 [Service Worker, The first step]*. Code Grid. https://app.codegrid.net/entry/2016-service-worker-1 [Accessed 2018-02-26]

76. Yeeply. (2017). *Advantages and disadvantages of web app development.* May 9. https://en.yeeply.com/blog/advantages-and-disadvantages-of-web-app-development/ [Accessed 2018-03-09]

77. Zaza, S. (2017). *How to Make Your Existing React App Progressive in 10 Minutes*. Scotch. [Forum]. September 25. https://scotch.io/tutorials/how-to-make-your-existing-react-app-progressive-in-10-minutes [Accessed: 2018-02-25]

78. @Neos2. (2017). *LocalForage を使ってアプリ内 DB を簡単構築 [Easily develop an app's DB by means of LocalForage]*. Corredor. [Blog]. November 1. http://neos21.hatenablog.com/entry/2017/11/01/080000 [Accessed 2018-03-02]

79. @yamayamasan. (2016). *Dexie.js で IndexedDB を使ってみる*. Qiita. [Forum]. November 28. https://qiita.com/yamayamasan/items/a4297e724b86f4a00fd2 [Accessed: 2018-02-26]

80. @yukika. (2015). *React Component のライフサイクルのまとめと利用用途 [Summary of Lifecycle of React Component and their Usage]*. Qiita. [Forum]. November 25. https://qiita.com/yukika/items/1859743921a10d7e3e6b [Accessed: 2018-02-24]