Bachelor's thesis

Degree programme in Information Technology

2018

Morad Joseph Khoury

PROGRESSIVE WEB APPLICATIONS

– Fetching and Caching

TURKU AMK

TURKU UNIVERSITY OF APPLIED SCIENCES

BACHELOR'S THESIS | ABSTRACT

TURKU UNIVERSITY OF APPLIED SCIENCES

Degree programme in Information Technology

2018 | 131

Morad Joseph Khoury

PROGRESSIVE WEB APPLICATIONS

- Fetching and Caching

The main objective of this thesis was to create a fully functional progressive web application that behaves like a native application while benefiting from the larger web audience. Progressive web applications technology integrates native application features into web applications.

This thesis covers thoroughly all aspects of the progressive web applications technology such as the manifest, the service worker, background synchronization, web notifications, and media API; moreover, it focuses on fetching and caching as it suggests multiple cache strategies and implements an appropriate caching strategy tailored for the application under development.

The delivered application implements a functional front-end user interface caching data from the back-end database; further, the application allows the user to install the web application on the home screen with a unique application icon, delivers a native application look, provides offline access, maintains low storage demand and grants access to the device camera allowing the user to take pictures and post it immediately even when internet connection is unavailable.

KEYWORDS:

Progressive, Web applications, Fetching, Caching, Service worker, Push-notifications, Media API

Foreward

I would like to deeply thank Professor Jari Pekka Paalassalo and Professor Poppy Skarli for their outstanding support throughout the period of my studies and especially on their assistance, availability, and advice during the writing of this thesis.

Morad Joseph Khoury

CONTENTS

PICTURES

LIST OF ABBREVIATIONS (OR) SYMBOLS

AJAX	Asynchronous JavaScript And XML
API	Application programming interface
CORS	Cross-Origin Resource Sharing
CSS	Cascade Styling Sheet
DB	Data Base
DOM	Document Object Model
GPS	Global Positioning System
HTML	Hyper Text Markup Language
HTTP	Hypertext Transfer Protocol
IDB	Indexed Data Base
JavaScript	Server-side scripting language
JS	JavaScript
JSON	JavaScript Object Notation
Meta tags	Descriptions Attributes
NPM	Node Package Manager
PWA	Progressive Web Application
SW	Service Worker
URL	Uniform Resource Locator
UUID	Universally Unique Identifier
XLR	XML HTTP Request
XML	Extensible Markup Language

1 INTRODUCTION

This thesis dives deep into a relatively new technology, Progressive Web Applications (PWAs). PWA technology introduces a set of features added to a web application to provide a closer experience to a native application. This thesis discusses PWAs features in detail; further, it answers the following main questions.

1. What are PWAs?
2. Why are PWAs used?
3. How is the technology implemented?

In this thesis, a fully functional PWA is created by adding the progressive features gradually; starting from the application manifest that delivers a native application look to the web application, followed by introducing the service worker which is responsible for caching data and providing offline access, moving to implementing background synchronization for the application data, and finally granting application access to device features such as the device camera and push notifications.

The features that make up a PWA are well supported already and reach a huge audience; moreover, PWAs work normally on older browsers. PWAs technology aims to progressively enhance the web applications to feel and look like native mobile applications. PWAs are reliable, fast, responsive, engaging and provide offline functionality.

PWAs are built for the web; hence, only one version is coded using standard web code languages and works on all browsers and devices, unlike native applications that are built for each operating system individually; for example, a native application version is built for Android using an Android code while a separate version is built for IOS using a different code.

Further, the author of this thesis believes that every web application will be rebuilt as a PWA and enhanced with native application features soon since it offers significant advantages and provides an outstanding user experience.

2 OVERVIEW

PWAs is an emerging technology. It introduces a set of features added to standard web applications to import native mobile user features, such as camera, location and push notification; additionally, it enhances the web applications' offline access. PWAs features are powerful tools implemented in the web browser and they significantly enhance the user experience, reliability, speed, and user engagement.

This brief chapter provides an overview of the PWAs technology and compares native applications with mobile web applications; further, it introduces the PWA developed and provides a basic application set up. Finally, this chapter presents the PWAs core blocks and the application featured enhancements.

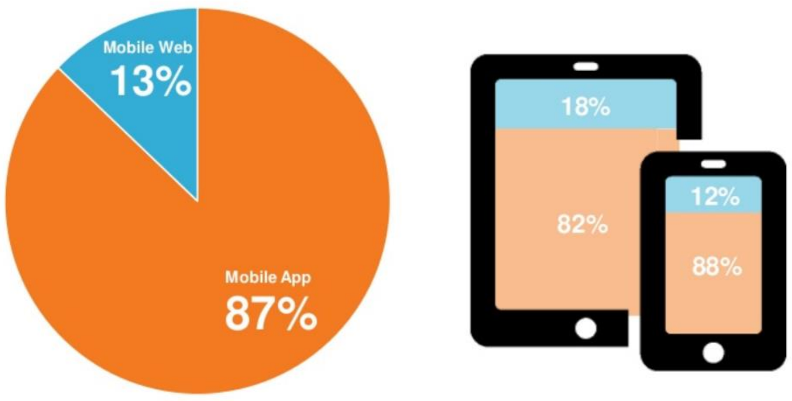
2.1 PWAs concept

The PWA concept refers to new technologies that are added to any existing web application running on the browser and it is intended to improve the capabilities of the web application by adding additional native application features. It progressively improves web applications so that they look and feel like native mobile applications, mainly referring to the native functionalities that web applications fail to offer, such as offline access, adding an icon to the home screen, accessing the device camera or location and synchronizing data in the background. Additionally, the PWAs browser support is growing; however, a PWA lacking browser support behaves like a regular web application. Mainly, PWAs load faster, respond faster to user interactions, provide offline access, grant access to the mobile device features and provide a more engaging user experience such as allowing push notifications to be sent to the user on demand, even when the application is closed.

2.2 Native apps vs mobile web apps

"Mobile applications dominate the time the user spends against the mobile web browsing, with a slightly higher split for smartphones than tablets".

(ComScore 2017)



Picture 1. The share of time spent on mobile.

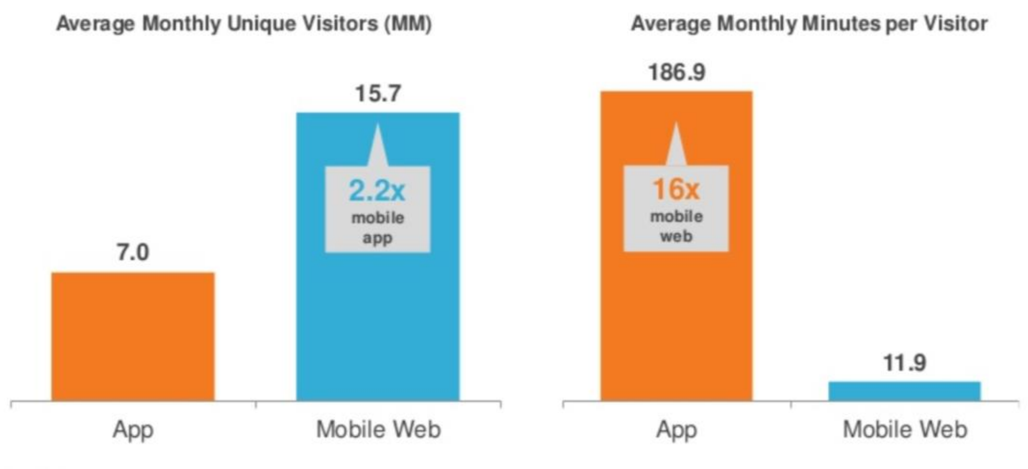
Source: https://www.slideshare.net/comScoremarcom/2017-us-mobile-app-report

Users spend more time using mobile native applications since they are much easier to use, faster and more engaging than web applications. Native mobile applications provide several advanced features causing native applications to overtake web application.

1. Push notifications drive the user back to the application even after the application is closed.
2. Home screen icon provides an easy access and a more professional user experience.
3. A native application grants access to mobile features such as camera and GPS.
4. Some native applications work offline.

The previous native application features mentioned above are powerful reasons to choose native applications over web applications; however, building native applications has its own drawbacks.

1. Building native applications require coding the application in different code languages for different operating systems such as Android and IOS.
2. Eighty percent (80%)of time spent using native applications occurs in the users' top 3 applications on the device.
3. The average user spends 16 times more time using the top native applications than using the top web applications; however, mobile web tends to capture larger audiences (Picture 2).



Picture 2. Top 500 mobile apps vs. top 500 web apps.

Source: https://www.slideshare.net/comScoremarcom/2017-us-mobile-app-report

The statistics above (Picture 2) illustrate the results of a mobile application report conducted in the US for the year 2017. According to the report (ComScore 2017), the average monthly minutes per visitor for native applications is 16 times higher than mobile web applications; however, the average monthly unique visitors for web applications is 2.2 times higher than native applications. Despite the extraordinary greater time spent using native applications, there exists a broader user reach for the mobile web. Users tend to spend more time using native applications; however, they use the same applications; thus, the average application does not receive many visitors on average.

In general, using the web pages on a mobile device for the first time is much easier and faster than using native applications as it only requires opening the browser, whereas using native application requires visiting the play-store / app-store, searching for an application, downloading, then installing it. PWAs offer the best of both sides, in other words, an integration between the capability of native applications and the reach of traditional web applications/pages.

2.3 PWA prototype

This thesis attempts to develop a new PWA prototype. The steps to set up the PWA environment are described as follows:

1. The download of node.js is required to bring up the development server and manage dependencies and libraries **(node v9.11.1)**.

2. This is followed by the installation of NPM (node package manager) inside the project folder. NPM is used to install several dependencies such as the HTTP server used for simulating a web server for the project since a regular file protocol is useless for simulating dynamic web pages **(NPM 6.4.1)**.

3. Finally, the live server is launched using NPM start.

The developed PWA consists of several core building blocks.

1. The service worker which is a JavaScript thread running in a background process even if the application is closed. The service worker provides an offline access by caching files; further, it provides additional PWAs features such as:
    1.1 Background sync which allows sending the synced data once an internet connection is established.

    1.2 Push-notifications which runs independently in the background.

2. The application Manifest which turns the web application into an installable application and adds it to the home screen.

3. Background synchronization which allows the application to sync data in the background and uses the data synced on demand, typically when an internet connection is unavailable.

4. Push notifications which provide real-time interactions and re-engages the user with the application. The developed web application enables web push notifications with the help of the service worker.

5. Native device features enhance the application by providing access to device features such as the device camera.

6. Responsive web design is a main feature of application development; however, it will not be discussed in this project, since it is relatively a main requirement implemented in most applications.

3 CORE CONCEPTS

This chapter introduces the main concepts of PWAs. It discusses technologies and tools used to develop the PWAs; moreover, it explains thoroughly the functionalities, usage and browser support of the application core elements.

3.1 Application Manifest

The web application manifest is an important part of the PWAs technologies. It is a single JSON text file that is added to the project to pass additional information about the application to the browser. The browser uses the corresponding data to advance the application functionalities, specifically it

1. Displays the application differently.
2. Grants application permission for automatic installation on the home screen of the users' device.
3. Provides a native application look to the web application.

An application manifest is a core block, implemented in the root project folder where the index.html file is located. Installing a web application on the device home screen allows the direct launch of the web application, increases user interaction with the app, and delivers a closer native application experience.

Typically, it is recommended to import the manifest file into every HTML page to apply the manifest rules throughout the entire application; moreover, it is deployed using the <link> tag element in the <head> section of the page.

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

Indeed, the application manifest has proved to be a powerful tool, used to fabricate the web applications' appearance and provide richer offline experiences.

3.1.1 Application manifest properties

The application manifest consists of multiple elements.

1. Name: runs a readable name of the application.
   "name": " "

2. Short name: runs a readable short name of the application, this is helpful when there is insufficient area to deliver the complete name of the application.

   ```
   "short_name": " "
   ```

3. Icons: provide an array of image/icon files that can be used by the browser when using an icon is needed; typically, the same icon is defined with different scales in consideration; consequently, the browser selects the appropriate icon size for the correspondent use.

   ```
   "icons": [  {  "src": 〝 ",
   "type": " ",
   "sizes": " " }, ]
   ```

4. Start url: defines the URL that loads first when opening the application.

   ```
   "start_url": " "
   ```

5. The scope: provides a restriction of the manifest file scope of use, whether it is the entire project or selected pages.

   ```
   "scope": " "
   ```

6. Display property: specifies a display style of the application.

   ```
   "display": " "
   ```

7. Display orientation: defines the application orientation of the application.

   ```
   "orientation": " "
   ```

8. Main background color: pre-defines the background that is stated in the CSS file; hence, the background definition loads before the stylesheet. this provides a smooth transition between initiating the application and loading its content.

   ```
   "background_color": " "
   ```

9. Main theme color: specifies the main theme color for the application.

   ```
   "theme_color": " "
   ```

10. Description: provides a description of the application.

    ```
    "description": " "
    ```

11. Direction: emphasizes the text direction of the name, the short_name, and the description object elements.

    ```
    "dir": " "
    ```

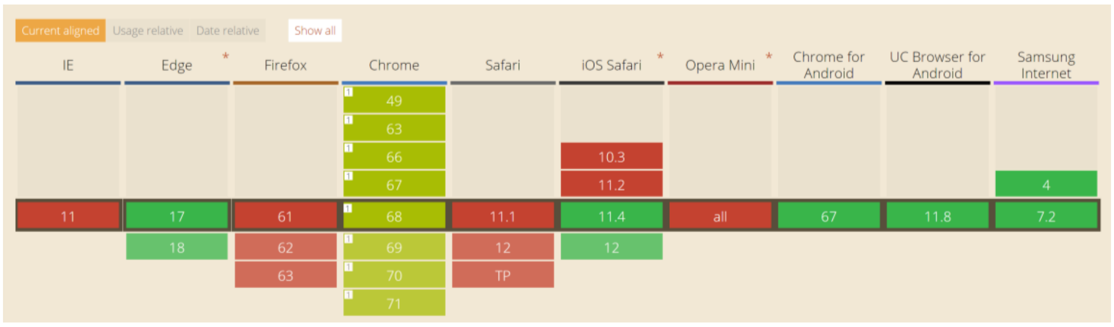
12. Language: emphasizes the text key Language for the name, the short_name, and the description object elements.

    ```
    "lang": " "
    ```

3.1.2 Manifest browser support

The applications’ manifest JSON file is supported by many browsers and the support for the manifest tends to be growing as browsers continue developing to adapt to the new manifest approach. There are many browsers that do not support application manifests yet; however, this does not seem to be an issue, since browsers without application manifest support should behave regularly, and simply ignore the manifest file.

Currently, only Google Chrome offers full support for the application manifest. Picture 3 specifies the manifest browser support up to this date.


Current aligned
Usage relative
Date relative
Show all

IE	Edge	Firefox	Chrome	Safari	iOS Safari	Opera Mini	Chrome for Android	UC Browser for Android	Samsung Internet
			49						
			63						
			66		10.3				
			67		11.2				4
11	17	61	68	11.1	11.4	all	67	11.8	7.2
	18	62	69	12	12				
		63	70	TP					
			71						

Picture 3. Manifest browser support.

Source: https://caniuse.com/#feat=web-app-manifest

The progressive application manifest is not supported by all Safari and Explorer browsers as mentioned above; however, for building a web application, it is possible to define similar properties for the use of unsupportive browsers, meta tags are added to the web application to provide a closer experience of a PWA.

1. safari browser properties:

1.1
```
<meta name="apple-mobile-web-app-capable" content="yes">
```

Is supported by WebKit, it commands the safari browser to handle the application as a web application; thus, grants permission to launch the application from the home screen of the device.

1.2
```
<meta name="apple-mobile-web-app-status-bar-style"
content="black">
```

Specifies the display of the status bar.

1.3
```
<meta name="apple-mobile-web-app-title" content="PWAMorad">
```

Provides an application title.

1.4
```
<link rel="apple-touch-icon" href="/src/images/icons/apple-
icon-57x57.png" sizes="57x57">
```

Defines the application icon, it can specify as many icon sizes as needed.

2. Internet Explorer properties

   2.1 `<meta name="msapplication-TileImage" content="/src/images/icons/app-icon-144x144.png">`

   Defines the application icon.

   2.2 `<meta name="msapplication-TileColor" content="#fff">`

   Defines the background color of the icon.

3.2 Service workers

The service worker is an important building block of the PWAs technology. This project provides an overview of service workers; moreover, it discusses functionalities and implementation of the service worker in the web application under development. The service worker functions behind the scenes. It adds an offline user experience to the web application, and grants permissions to use native applications features, such as accessing the device camera, interacting with the device GPS and permitting push notifications.

3.2.1 Service worker code

The service worker is a JavaScript file and it runs in the background separately and provides extended functionalities with access to an additional set of features. Normal JavaScript code runs on a single thread, attached to an HTML page. The loaded JavaScript code runs on one thread, even if the page has multiple JavaScript files, all files still run on the same thread. The regular JavaScript code accesses the DOM (Document Object Model) of the pages; for example, it accesses alerts, manipulates the DOM, shows or hides elements, and execute actions related to the DOM.

Moreover, JavaScript frame-works act similarly, that is the code is converted to a JavaScript code; consequently, it can manipulate the DOM in the same way.

Although a service worker is a JavaScript file that runs on a single thread, it runs on a separate single thread than the normal JavaScript code runs on, because the service worker runs in the background; consequently, it is decoupled from the application HTML pages.

A service worker registers through the index HTML page initially; however, it becomes available through all pages (e.g., all pages of a domain), unlike a normal JavaScript code.

The service worker lives in the browser even after the pages have been closed. It runs in the background, uses a JavaScript code but works differently than a normal JavaScript file. Further, the service worker is highly reactive to events, it sets in the background, and reacts to different incoming events such as displaying user notifications.

3.2.2 Service worker events

Fetch Event

Every HTTP request is sent using the fetch method which is triggered by the browser or by page-related JavaScript code; for example, an image tag pointing to the image resource. When the browser attempts to display the image, it sends a fetch request.

A fetch request is handled in the service worker file and the service worker behaves as a network proxy. Every HTTP request is sent via fetch; hence, it goes through the service workers.

Furthermore, a fetch request is triggered during the process of loading a JavaScript or a CSS file that has been imported into an HTML file. However, fetch is not triggered with normal AJAX requests; therefore, an HTTP request can be manipulated through the service worker, it can be blocked, cached, and controlled.

Push Notifications

Push notifications are sent from another server; basically, the browser renderer has its own web-push server and it can send push notification from the application server to the browser server, and then it forwards it to the client application.

The service worker listens to such push events, since the service worker lives in the background, even when the application is offline. A push notification aims to direct the user back to the application after the session has ended. The service worker is a powerful tool used to manipulate push notification events. The service worker listens and reacts to the user interactions regarding push notifications.

Background Synchronization

Background synchronization performs a data synchronization for saving the data from getting lost when it fails to send due to poor connection. When failing to send a post due to a poor internet connection, the background synchronization saves the event and resends it when an internet connection is re-established. Google Chrome offers background synchronization.

When closing the application, the service worker still runs; therefore, if an internet connection is re-established, it will react to it and will execute the action stored.

Service worker life-cycle

The service worker file is coded using JavaScript, but it does not execute as a normal JavaScript file; instead, it is registered as a background process file. During the registration of the service worker, two life-cycle phases are committed.

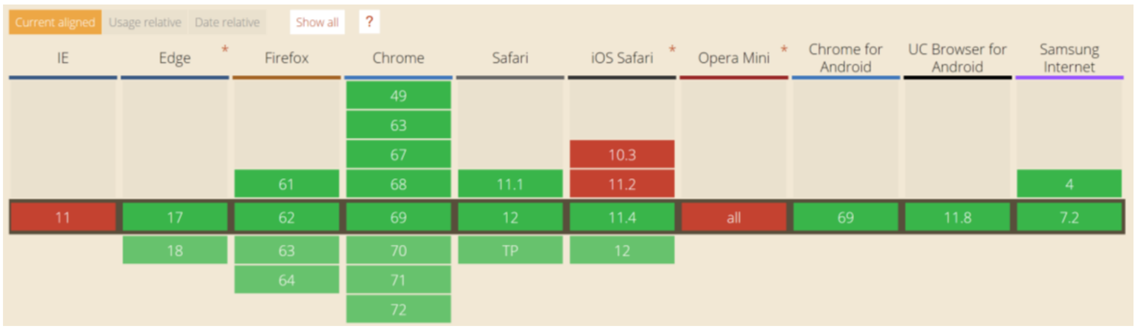
1. Installing the service worker by the browser: the browser emits an install event which is used to execute the code inside the service worker, and it is used to cache initial assets. If the currently installed version of the service worker changes by 1 bite or more, the service worker is updated/reinstalled on the reload of the page.

2. Activation of the service worker: it provides application control of pages of the scope.

Afterwards, the service worker goes into an idle mode in the background process to handle events and if no events are triggered, it does nothing. Further, it enters a terminated mode when unused, and re-engages when an event is triggered.

3.2.3 Service worker browser support

Generally, most browsers acknowledge the service workers and browser support is evolving since many browsers offer the support such as Safari, Chrome, Firefox, and Edge; never the less, some browsers fail to do so. Picture 4 specifies the service worker browser support up to this date.


Current aligned
Usage relative
Date relative
Show all
?

IE	Edge *	Firefox	Chrome	Safari	iOS Safari *	Opera Mini *	Chrome for Android	UC Browser for Android	Samsung Internet
			49						
			63						
			67		10.3				
		61	68	11.1	11.2				4
11	17	62	69	12	11.4	all	69	11.8	7.2
	18	63	70	TP	12				
		64	71						
			72						

Picture 4. Service worker browser support

[Source: https://caniuse.com/#feat=serviceworkers](https://caniuse.com/#feat=serviceworkers)

3.3 Promise and Fetch API

Both fetch API and promise API are native technologies, not supported in older browsers. The following sections cover promise functionality, fetch API, and compare promise API with the traditional AJAX request; moreover, they discuss promise interpretation in the service workers and implementation in PWAs.

As mentioned earlier, JavaScript is single-threaded, that executes code by the starting the Asynchronous tasks while waiting for future responses, it uses callbacks to operate other functions in between the tasks. JavaScript code does not wait for previous processes completion to execute the later code; however, it starts Asynchronous tasks and waits for future responses for those tasks.

"An asynchronous function is a function which operates asynchronously via the event loop, using an implicit Promise to return its result. But the syntax and structure of your code using async functions is much more like using standard synchronous functions." (MDN web docs 2018)

3.3.1 Promises

Promises are used when code turns more complex for callbacks to work with, since callbacks are not capable of handling complex asynchronous code. Promises are relatively new features, supported by some browsers. A promise object denotes the final state of an asynchronous process and its value.

The promise constructor takes one function argument that executes immediately, this function takes two functions arguments, a resolve function, and a reject function.

Promises return one of two results under all circumstances, either **resolve** the value or **reject** the value (return an error).

```
var promise = new Promise(function(resolve, reject){ }
```

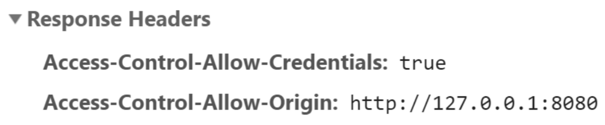
3.3.2 Fetch and CORS

Fetch() is a JavaScript method that produces network HTTP requests and returns promises, hence; it is a clean and an easy method accustomed to get, transfer and read data from a foreign source such as an external web page.

"Cross-origin resource sharing (CORS) is a mechanism that allows restricted resources on a web page to be requested from another domain outside the domain from which

the first resource was served. A web page may freely embed cross-origin images, stylesheets, scripts, iframes, and videos. Certain "cross-domain" requests, notably Ajax requests, are forbidden by default by the same-origin security policy.”

(WikiPedia 2017, Cross-origin resource sharing)



```
▼ Response Headers
   Access-Control-Allow-Credentials: true
   Access-Control-Allow-Origin: http://127.0.0.1:8080
```

Picture 5. Displaying CORS header.

3.3.3 Fetch vs Ajax

AJAX stands for Asynchronous JavaScript And XML. Sending a fetch request using an AJAX method is performed as follows:

1. Sets up the XHR object (an XML http request).
   `var xhr = new XMLHttpRequest();`
2. Creates a connection with a Get request.
   `xhr.open('GET', 'https://httpbin.org/post');`
3. Sets the response type to JSON.
   `xhr.responseType = 'json';`
4. Returns the response on load.
   `xhr.onload = function(){ console.log(xhr.response); };`
5. Handles errors.
   `xhr.onerror = function(){ console.log('Error!'); };`
6. Sends the request.
   `xhr.send();`

The AJAX process works fine; however, it performs differently behind the scenes. AJAX consists of synchronous code; therefore, fails to work alongside the service workers, since service workers implement Asynchronous code. Additionally, the AJAX syntax is more complex than the fetch() API syntax. The fetch API has a clear syntax, implements a simple approach, and it is maintained by the service workers.

3.4 Caching service workers

Caching is the process of storing data in a temporary storage; for instance, requested data of a visited web page is stored in the cache storage for future visits of the same web page. The browser uses the cache storage data rather than requesting the data again from the web page server, it saves loading time and saves additional server traffic. Further, it offers offline access for cached pages, this is useful when the connection is poor or not available. Different cache storages are available for web applications implementation.

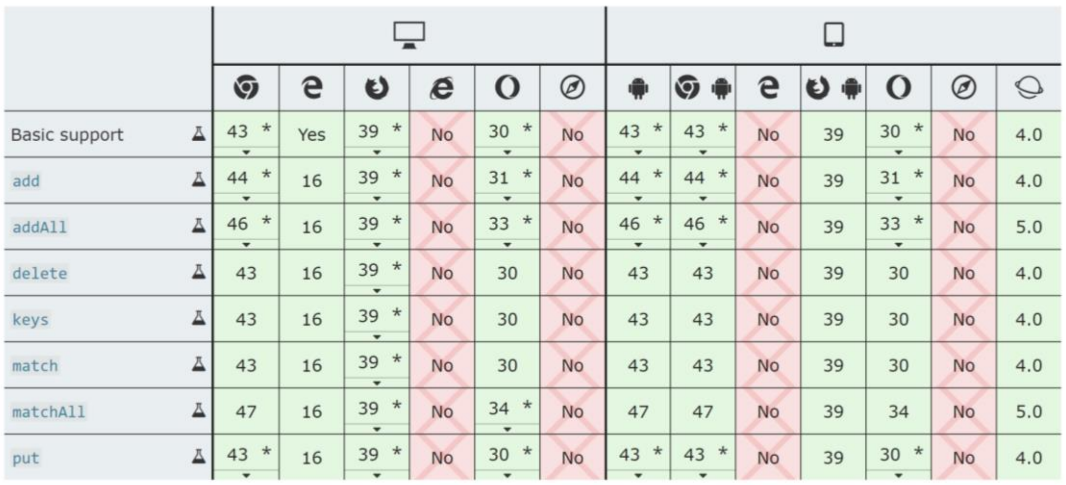
1. The browsers own automatic cache, it has been disabled earlier in the application under development, it is unreliable since it is managed by the browser; therefore, the application has no control over the browser cache nor the data cached in it.

2. Cache API is a separate cache that lives in the browser; however, it is managed by the developer. The cache API holds the URL/HTTP request and sends values as responses. A response requires a reload of the page since caching requires a single time connection that provides an access to the to-be-cached data.

   The Cache API is accessed by the service worker using JavaScript. The cache API provides a method to retrieve data without sending network requests when the connection is unavailable.

3.4.1 Cache browser support

Not all browsers support the caching approach, Picture 6 specifies the cache browser support up to this date.



Basic support	43 *	Yes	39 *	No	30 *	No	43 *	43 *	No	39	30 *	No	4.0
add	44 *	16	39 *	No	31 *	No	44 *	44 *	No	39	31 *	No	4.0
addAll	46 *	16	39 *	No	33 *	No	46 *	46 *	No	39	33 *	No	5.0
delete	43	16	39 *	No	30	No	43	43	No	39	30	No	4.0
keys	43	16	39 *	No	30	No	43	43	No	39	30	No	4.0
match	43	16	39 *	No	30	No	43	43	No	39	30	No	4.0
matchAll	47	16	39 *	No	34 *	No	47	47	No	39	34	No	5.0
put	43 *	16	39 *	No	30 *	No	43 *	43 *	No	39	30 *	No	4.0

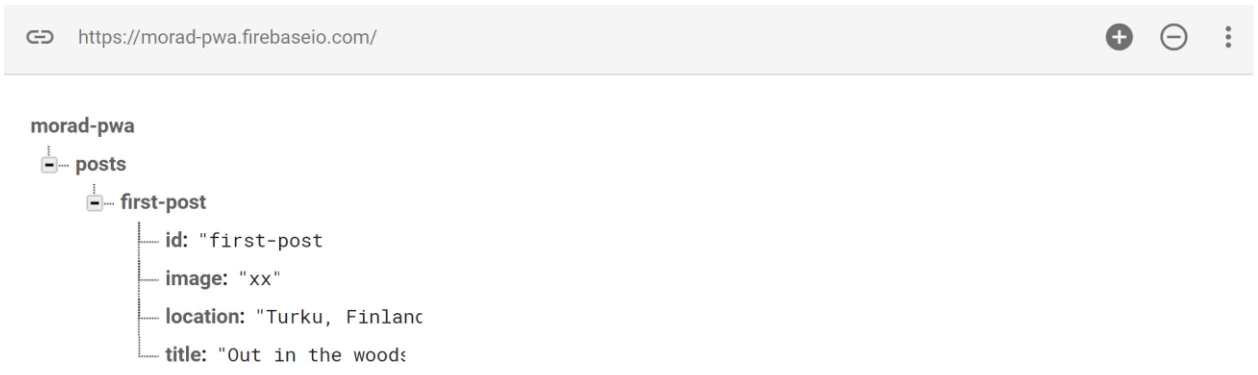
Picture 6. Cache browser support.

Source: https://developer.mozilla.org/en-US/docs/Web/API/Cache

It is good to point out again that the PWA technology is an increment enhancing technology, it works in the supported browsers, whereas nothing will change or be affected concerning the rest of the browsers. Unsupportive browsers continue to behave normally without benefiting from the PWAs technology.

3.5 Dynamic data

In this project a real-time data source is created, using the Google Firebase database, it is a backend server solution managed by Google: https://firebase.google.com/, The back-end Firebase database is set up (Picture 7), it offers file storage while running node.js behind the scenes. The Firebase database implements a unique structure using JSON.


https://morad-pwa.firebaseio.com/

```
morad-pwa
 ⊟ posts
    ⊟ first-post
        id: "first-post
        image: "xx"
        location: "Turku, Finlanc
        title: "Out in the woods
```

Picture 7. Google Firebase database.

After creating a simple database to store the created posts assets (stores the id, image, location, and title), it connects to the front end of the application to fetch data from the database using HTTP requests.

3.5.1 Dynamic caching vs caching dynamic content

Dynamic caching and caching dynamic content are two different concepts. Dynamic caching is implemented as follows.

1. The application sends a network request which gets intercepted by the service worker.
2. The service worker reaches out to the network.
3. The application receives the request from the network, then saves the request in the cache and returns to the web page.

Dynamic caching stores the requested assets in the cache immediately after receiving it. Dynamic caching happens after precaching the static data. The application caches dynamic data after it reaches for the network and receives a response from the network. Dynamic caching refers to dynamically adding data to the cache.

Whereas, Caching dynamic content is different. The cache API is unnecessary; instead, caching is received by the indexedDB (a key-value database). The indexedDB runs in the browser and helps to store both; structured and unstructured data, such as JSON data and XML data, but no files or assets get stored in the cache storage. IndexedDB is used to store frequently modified dynamic data.

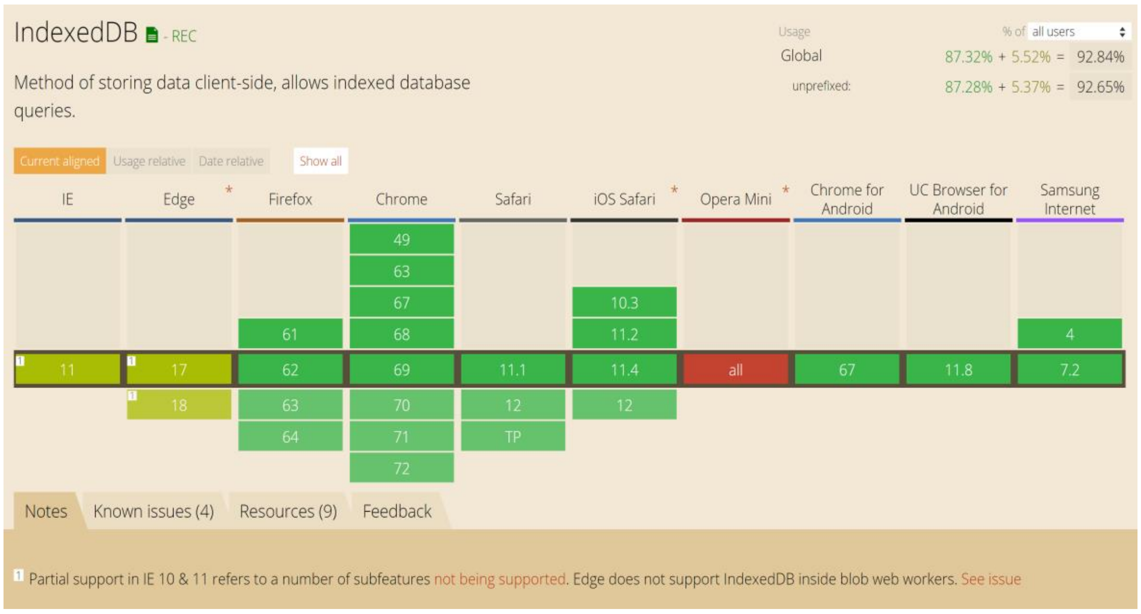
3.5.2 IndexedDB

IndexedDB is a transactional key-value database running in the browser. The indexedDB conduction relies on dependant actions. A single action failure causes the failure of the rest of actions within the indexedDB transaction; hence, the failure of a single entry to reach the database prevents the database from parsing other entries.

IndexedDB stores a significant amount of unstructured data, including files; due to that fact, it is considered as a JavaScript object, it stores different types of data. IndexedDB can also be accessed asynchronously; thus, can be used in the service workers, it is accessed using JavaScript and it is able to contain multiple objects.

3.5.3 IndexedDB browser support

Picture 8 specifies the IndexedDB browser support up to this date. IndexedDB is well supported by browsers, it is safe and secure.



IndexedDB - REC

Method of storing data client-side, allows indexed database queries.

Usage | % of all users
Global 87.32% + 5.52% = 92.84%
unprefixed: 87.28% + 5.37% = 92.65%

Current aligned | Usage relative | Date relative | Show all

IE	Edge *	Firefox	Chrome	Safari	iOS Safari *	Opera Mini *	Chrome for Android	UC Browser for Android	Samsung Internet
			49						
			63						
			67		10.3				
		61	68		11.2				4
11 [1]	17 [1]	62	69	11.1	11.4	all	67	11.8	7.2
	18 [1]	63	70	12	12				
		64	71	TP					
			72						

Notes | Known issues (4) | Resources (9) | Feedback

[1] Partial support in IE 10 & 11 refers to a number of subfeatures not being supported. Edge does not support IndexedDB inside blob web workers. See issue

Picture 8. IndexedDB browser support.

Source: https://caniuse.com/#feat=indexeddb

3.6 Background sync

The PWA implements background synchronization. It is a significant, relatively new feature used to support offline access. Background synchronization synchronizes the sent data even throughout offline access. In theory, the application should work offline and navigates through pages after the cache process; however, it will fail to send data while offline. The application is enhanced to store requests sent during offline access using the background sync.

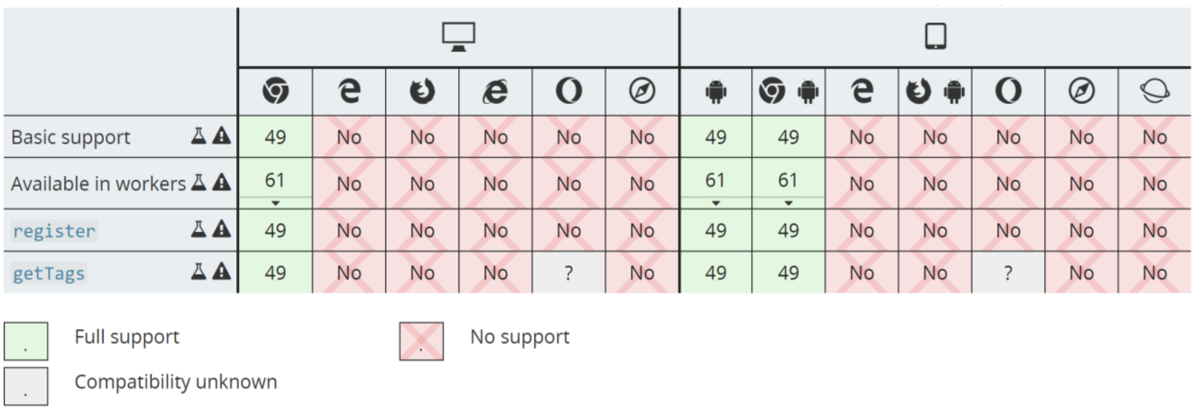
3.6.1 Sending data while offline

During the background synchronization process, the application sends data to the server when the connection is unavailable; after, the application retrieves the data from the server as soon as an internet connection is established after caching the response of the requested data through the service workers.

The service worker registers a sync task to be sent when the connection is available, the sent data is stored in the indexedDB; consequently, the application sends the data when the connection is re-established even if the application is closed, this runs as a background process. Background synchronization performs flawlessly regarding the internet connectivity status.

3.6.2 Sync manager browser support

Picture 9 specifies the sync manager browser support.



Basic support	49	No	No	No	No	No	49	49	No	No	No	No	No
Available in workers	61	No	No	No	No	No	61	61	No	No	No	No	No
`register`	49	No	No	No	No	No	49	49	No	No	No	No	No
`getTags`	49	No	No	No	?	No	49	49	No	No	?	No	No

🟩	Full support	🟥	No support
⬜	Compatibility unknown		

Picture 9. Sync manager browser support.

Source: https://developer.mozilla.org/fi/docs/Web/API/SyncManager

3.7 Web push notifications

Push notifications method is a powerful tool implemented in the native applications and it provides real-time interactions and re-engages the user with the application. The web application enables web push notification with the help of the service worker.

3.7.1 Push notifications overview

The application uses push notifications to redirect the user into the application even when the application is closed, it enhances user real-time engagement and provides a mobile app-like experience. Push notifications push data information into the application and notify the user on demand. The application requests user permission; hence, the application displays notifications independently on demand.

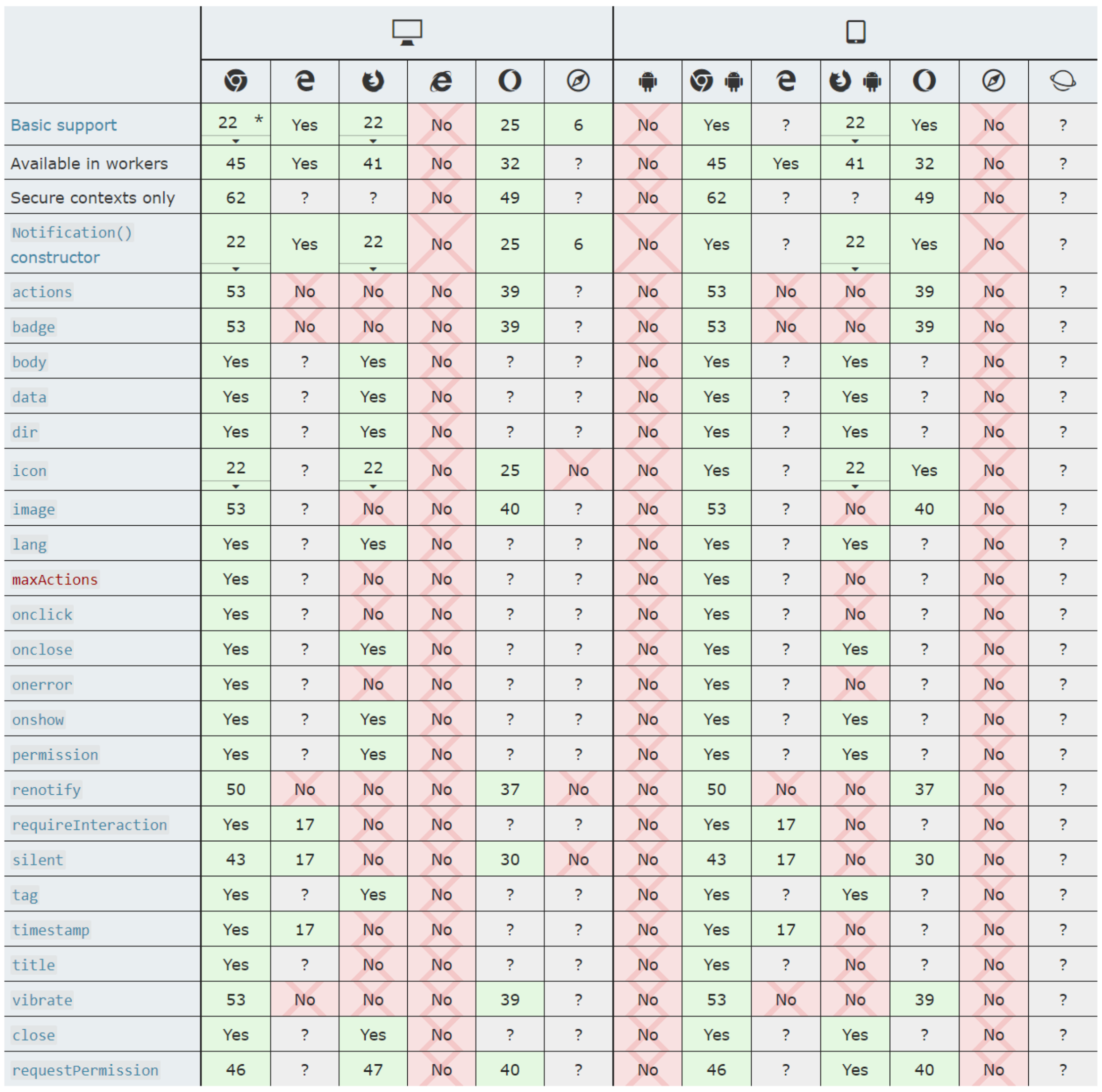
A notification is triggered using JavaScript, it requires push subscriptions granted by the browser to acquire permission. Push subscriptions are handled by the service worker, a subscription refers to a browser-device interaction, a notification is delivered to the web application throughout the browser.

Each browser has its own push server owned by the browser itself; further, the application sets up a new subscription using JavaScript, it automatically reaches out to the push server and fetches an end-point for the application after an authentication process is conducted. The fetched database end-point is a URL used to send new push messages, the browser server forwards the messages to the app.

Further, the web application requires running its own server, the service worker creates a new subscription holding information from the endpoint fetched from the browser servers; moreover, the service worker stores the new subscription data to the application backend server, it later uses the stored data to push messages to the front end of the application. Afterwards, the notification API is used to create and display notifications using plain JavaScript without the use of push events.

3.7.2 Push notifications browser support

Picture 10 specifies the push notifications browser support.



Basic support	22 *	Yes	22	No	25	6	No	Yes	?	22	Yes	No	?
Available in workers	45	Yes	41	No	32	?	No	45	Yes	41	32	No	?
Secure contexts only	62	?	?	No	49	?	No	62	?	?	49	No	?
`Notification()` constructor	22	Yes	22	No	25	6	No	Yes	?	22	Yes	No	?
`actions`	53	No	No	No	39	?	No	53	No	No	39	No	?
`badge`	53	No	No	No	39	?	No	53	No	No	39	No	?
`body`	Yes	?	Yes	No	?	?	No	Yes	?	Yes	?	No	?
`data`	Yes	?	Yes	No	?	?	No	Yes	?	Yes	?	No	?
`dir`	Yes	?	Yes	No	?	?	No	Yes	?	Yes	?	No	?
`icon`	22	?	22	No	25	No	No	Yes	?	22	Yes	No	?
`image`	53	?	No	No	40	?	No	53	?	No	40	No	?
`lang`	Yes	?	Yes	No	?	?	No	Yes	?	Yes	?	No	?
`maxActions`	Yes	?	No	No	?	?	No	Yes	?	No	?	No	?
`onclick`	Yes	?	No	No	?	?	No	Yes	?	No	?	No	?
`onclose`	Yes	?	Yes	No	?	?	No	Yes	?	Yes	?	No	?
`onerror`	Yes	?	No	No	?	?	No	Yes	?	No	?	No	?
`onshow`	Yes	?	Yes	No	?	?	No	Yes	?	Yes	?	No	?
`permission`	Yes	?	Yes	No	?	?	No	Yes	?	Yes	?	No	?
`renotify`	50	No	No	No	37	No	No	50	No	No	37	No	?
`requireInteraction`	Yes	17	No	No	?	?	No	Yes	17	No	?	No	?
`silent`	43	17	No	No	30	No	No	43	17	No	30	No	?
`tag`	Yes	?	Yes	No	?	?	No	Yes	?	Yes	?	No	?
`timestamp`	Yes	17	No	No	?	?	No	Yes	17	No	?	No	?
`title`	Yes	?	No	No	?	?	No	Yes	?	No	?	No	?
`vibrate`	53	No	No	No	39	?	No	53	No	No	39	No	?
`close`	Yes	?	Yes	No	?	?	No	Yes	?	Yes	?	No	?
`requestPermission`	46	?	47	No	40	?	No	46	?	Yes	40	No	?

Picture 10. Push notifications browser support.

[Source: https://developer.mozilla.org/en-US/docs/Web/API/notification](https://developer.mozilla.org/en-US/docs/Web/API/notification)

3.8 Application requirements

The PWA under development requires several different concept implementations, it is expected to behave like a native mobile application at the end of this project.

The application developed in this project is an application that allows the user to take pictures using the device camera if available; otherwise, it activates a file uploader to allow a file upload. The pictures taken by the user are added to the feed page of the application to be displayed alongside the title and the location attributes. Further, the application is expected to offer several key native application functionalities.

1. The application must be installable on the home screen with an application icon.
2. Implements a native application look.
3. Offers a fully functional front end-user interface.
4. Implements a back-end database.
5. Offers real-time user interactions.
6. Supports dynamic data update.
7. Offers offline functionalities by caching and fetching data as required.
8. Supports both; static cache and dynamic cache.
9. Uses push notifications to re-engage the user back into the application.
10. Accesses device features (Camera of the device).
11. Works flawlessly on all browsers.

This chapter has introduced the core concepts implemented to develop the PWA, it explained the different functionalities and approaches of technologies and methods used in the implementation process; further focused on browser support for the several technologies introduced.

After providing a thorough theoretical introduction, this thesis continues to implement the functional parts in the following chapters. At the end of this thesis, the developed application is expected to stand as a fully functional PWA.

4 MANIFEST IMPLEMENTATION

This brief chapter implements the first step towards converting the developed web application into a native application by implementing the application manifest file and providing an application install banner.

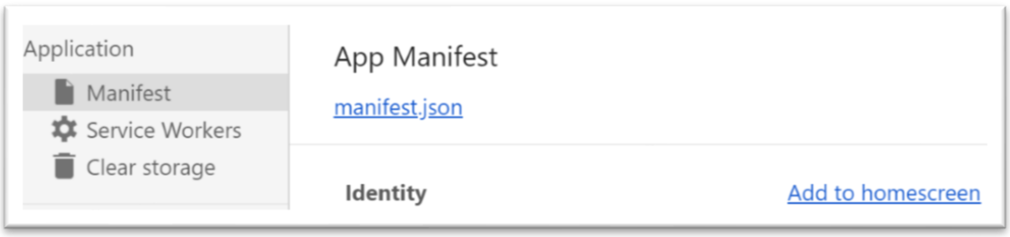
4.1 Implementing the manifest properties

The manifest file contains metadata that enhances the web applications' front-end to look like a native application.

```
"name": "A Progressive Web App".
"short_name": "PWAMorad".
"icons": [  {  "src": "/src/images/icons/app-icon-48x48.png",
"type": "image/png",
"sizes": "48x48" }, ].
"start_url": "/index.html".
"scope": "."
"display": "standalone"
"orientation": "portrait-primary"
"background_color": "#fff"
"theme_color": "#3f51b5"
"description": " implementing PWA technologies."
"dir": "ltr",
"lang": "en-US"
```

4.2 Application install banner

When opening the application developers' tools using an android phone, it can be observed in the manifest tool that an "add to home screen" option is available (Picture 11).



Picture 11. App manifest on an android.

Google Chrome assumes a good point of time to display the install banner is when the application is visited at least twice and with at least five minutes between visits.

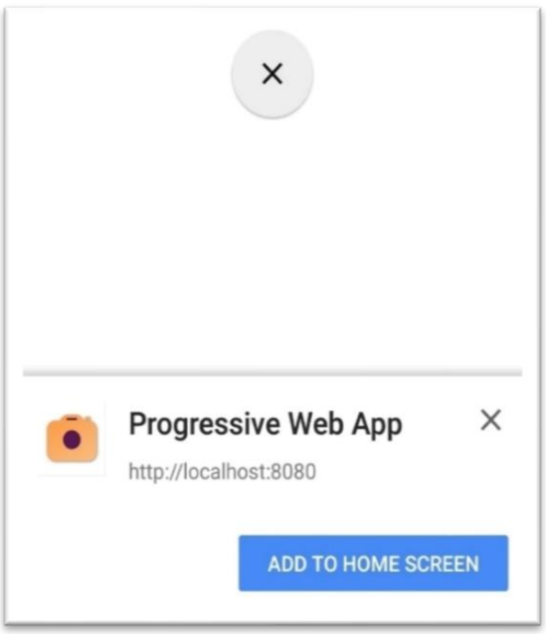
This can be prevented using the preventDefault() method within an event listener.

```
window.addEventListener('beforeinstallprompt', function(event){
 console.log('beforeinstallprompt fired!');
 event.preventDefault();
});
```

For this project, the application install banner will be triggered by pressing the add button in the home page using a custom event called deferredPrompt. This is implemented inside the add button event handler.

```
shareImageButton.addEventListener('click', openCreatePostModal);
function openCreatePostModal() {
 createPostArea.style.display = 'block';
 if(deferredPrompt){
  deferredPrompt.prompt();
  deferredPrompt.userChoice.then(function(choiceResult){
   console.log(choiceResult.outCome);
  });
 }
}
```

The function above checks whether the deferredPrompt is set and calls the deferredPrompt.prompt(), then it shows the banner and remembers the users' choice. The banner is displayed only once, there is no need to display it again if the user installs the application; however, the user still has the option to add the application to the home screen manually if the application was not installed using the install banner.



Picture 12. Add to home screen install banner.

In this brief chapter, the manifest file was implemented, and several properties were added to enhance the applications' display. Finally, the manifest was tested on a real mobile device. After implementing the manifest file, this thesis dives deeper into more technical issues and implements the core technologies that are responsible for merging the web applications technology with the native applications technology.

5 SERVICE WORKERS IMPLEMENTATION

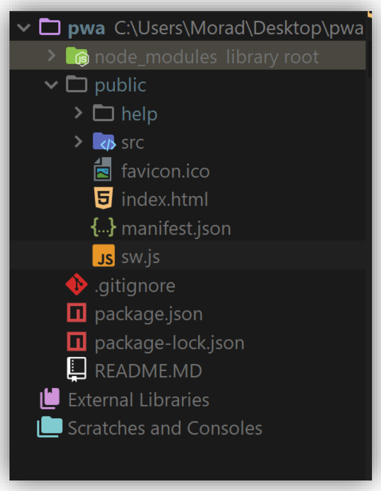
As mentioned earlier, the service worker is an important building block of the PWAs technology. This chapter registers a service worker in the application; further, it provides events interactions and implements the fetch API in the service worker.

5.1 Registering a service worker

There are two types of caches.

1. The default browser cache.
   Package.json **(v6.4.1)** file contains the start server property, it is used for development purposes and not for production ("start": "http-server -c-1").
   The (-c-1) is added to exclude assets-caching performed by standard browser cache. This ensures to reflect changes immediately in the browser; however, it is unnecessary for production, since files are not updated very often.

2. The service worker cache.
   A service worker holds the folders' scope. Typically, the service worker file is placed in the root folder and it is added to the public folder of the project since it is required to control all the pages of the project.



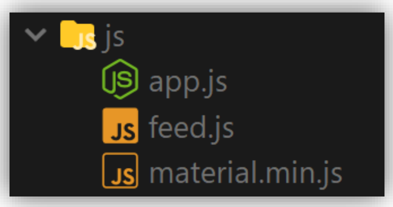
```
pwa C:\Users\Morad\Desktop\pwa
  node_modules  library root
  public
    help
    src
    favicon.ico
    index.html
    manifest.json
    sw.js
  .gitignore
  package.json
  package-lock.json
  README.MD
External Libraries
Scratches and Consoles
```

Picture 13. Service worker file in the project folder.

Before adding code to the service worker file (sw.js), it requires registration. The service worker is registered through the HTML page. It could be registered directly inside the HTML page; however, a more appropriate place is inside the app.js file, since it is imported to both the index.html and help.html files.

```
<script src="/src/js/app.js"></script>
```

The app.js is imported inside index.html and the help.html.



Picture 14. App.js file in the JS folder.

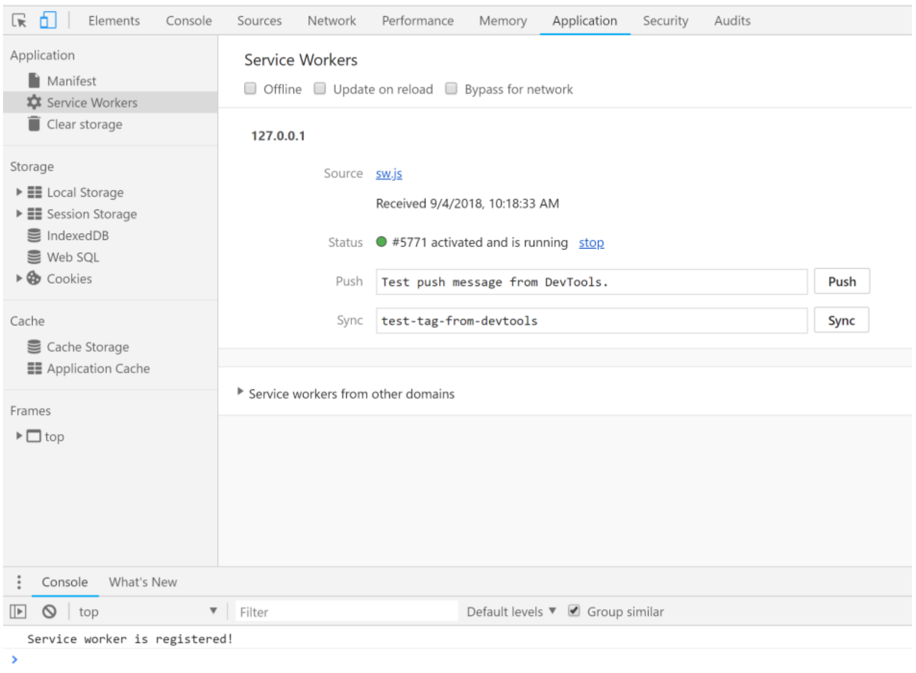
First, the application requires to check if the service worker feature is available in the browser. PWAs technology aims to progressively enhance applications; however, as discussed earlier, some browsers fail to support service workers. Regarding the browser compatibility, a check for the browser support is needed. In case of incompatibility, the service worker is ignored by the browser.

```
if ('serviceWorker' in navigator){
 navigator.serviceWorke .register('/sw.js');
}
```

This registers a service worker upon compatibility and once the registration is complete, it can start executing code using promise functions. This ensures the smoothness of the registration process without blocking another coding process after the registration is complete.

```
if ('serviceWorker' in navigator){
 navigator.serviceWorker.register('/sw.js')
 .then(function(){ });
 console.log('Service worker is registered!');
}
```

Testing the registration in the browser console results in a successful attempt (Picture 15).



Service Workers

127.0.0.1

Source sw.js

Received 9/4/2018, 10:18:33 AM

Status #5771 activated and is running stop

Push Test push message from DevTools.

Sync test-tag-from-devtools

```
Service worker is registered!
```

Picture 15. Service worker registration in the developer tools.

5.2 Service worker reacting to events

The register method used previously can pass a second argument that is used to define a scope property using a string. Afterwards, it gets used to restrict the scope of the service worker in case the application demands the service workers to apply partially within the app. Service workers are defined with full scope in the application usually; however, scope restriction is an important feature to keep in mind.

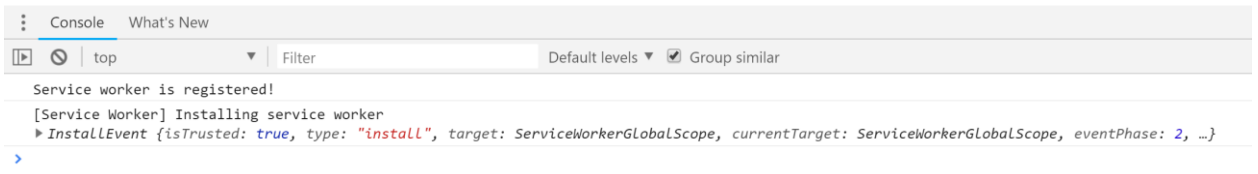
```
.register('/sw.js',{scope: '/some directory') })
```

Additionally, it is important to note that the manifest and the service workers are not related; generally, both files can be used separately. The service worker only performs on HTTPS served web pages. It is a powerful tool and can intercept any requests; thus, it must be encrypted securely. Service workers react to events; hence, event listeners are attached and referred to the service workers using a (.self-keyword). Moreover, The service worker requests access to the background process and adds event listeners on demand.

Service workers consist of normal JavaScript code but deny access to typical DOM events. It provides access to a different set of events as listed below.

1. The install event is fired when the browser attempts to install the service worker, it executes a function with an event object argument. The function gets passed automatically by the browser and its object argument passes information about the installation event as shown in picture 16.

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



```
Service worker is registered!
[Service Worker] Installing service worker
▸InstallEvent {isTrusted: true, type: "install", target: ServiceWorkerGlobalScope, currentTarget: ServiceWorkerGlobalScope, eventPhase: 2, …}
>
```

Picture 16. Service worker install-event success.

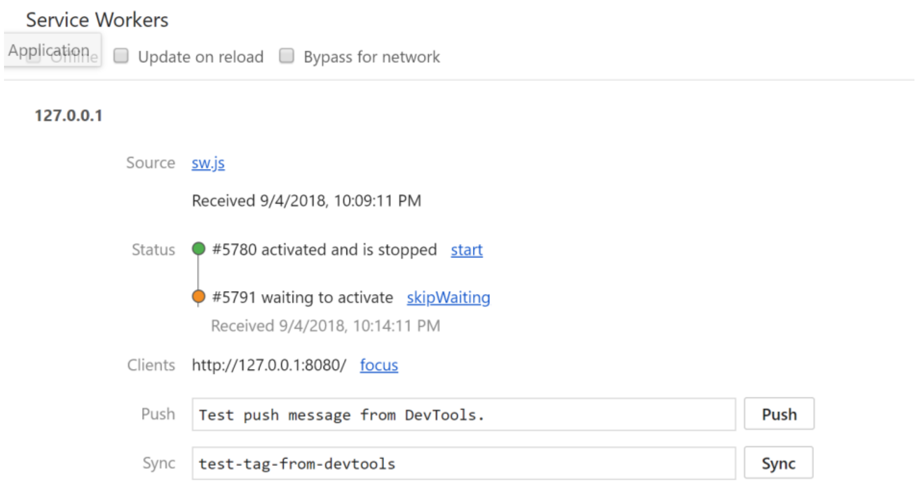
2. An activation event is fired when the service worker is activated after the installation process.

```
self.addEventListener('activate', function(event){
 console.log('[Service Worker] Activating service worker ', event);
```

It returns a statement to ensure that the service workers are activated correctly.

```
return self.clients.claim();
```

The activation process is expected to perform after the registration process is complete; however, the service worker file is a JavaScript file and it is unreliable. In theory, the activation process should execute after the installation of the service workers; however, it fails to do so since it is an asynchronous code. This behavior can be observed in the browsers' developer tools (Picture 17).


Service Workers

Application Offline ☐ Update on reload ☐ Bypass for network

127.0.0.1

Source sw.js

Received 9/4/2018, 10:09:11 PM

Status ● #5780 activated and is stopped start

● #5791 waiting to activate skipWaiting

Received 9/4/2018, 10:14:11 PM

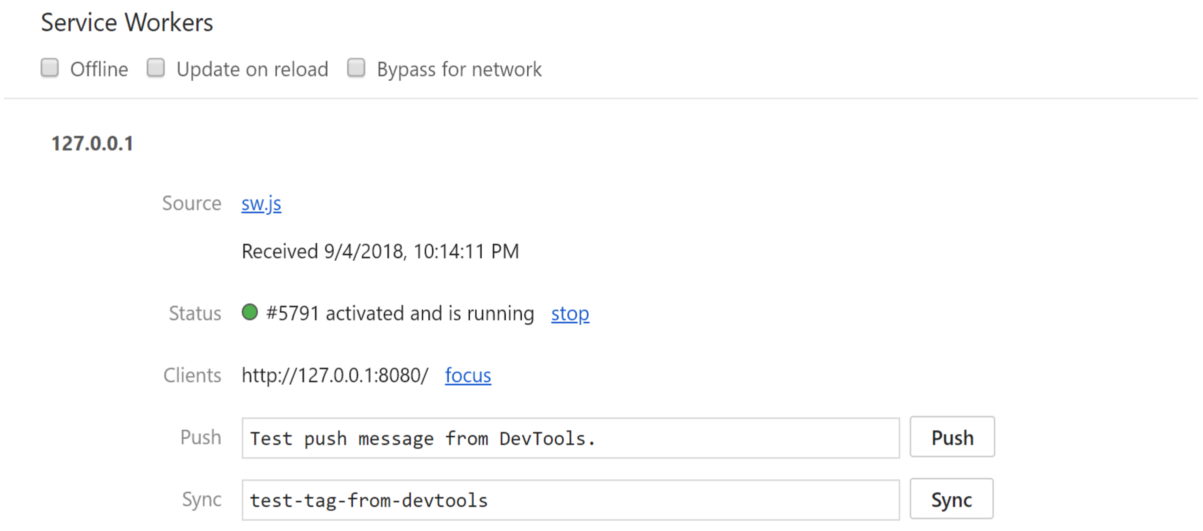
Clients http://127.0.0.1:8080/ focus

Push `Test push message from DevTools.` Push

Sync `test-tag-from-devtools` Sync

Picture 17. Service worker awaits activation.

The service worker has been installed but awaits activation since the page might be still communicating with the old service worker (Picture18). Therefore, activating a new service worker might interrupt the running page. A window re-open is required for the activation of a new service worker.


Service Workers

☐ Offline ☐ Update on reload ☐ Bypass for network

127.0.0.1

Source sw.js

Received 9/4/2018, 10:14:11 PM

Status ● #5791 activated and is running stop

Clients http://127.0.0.1:8080/ focus

Push `Test push message from DevTools.` Push

Sync `test-tag-from-devtools` Sync

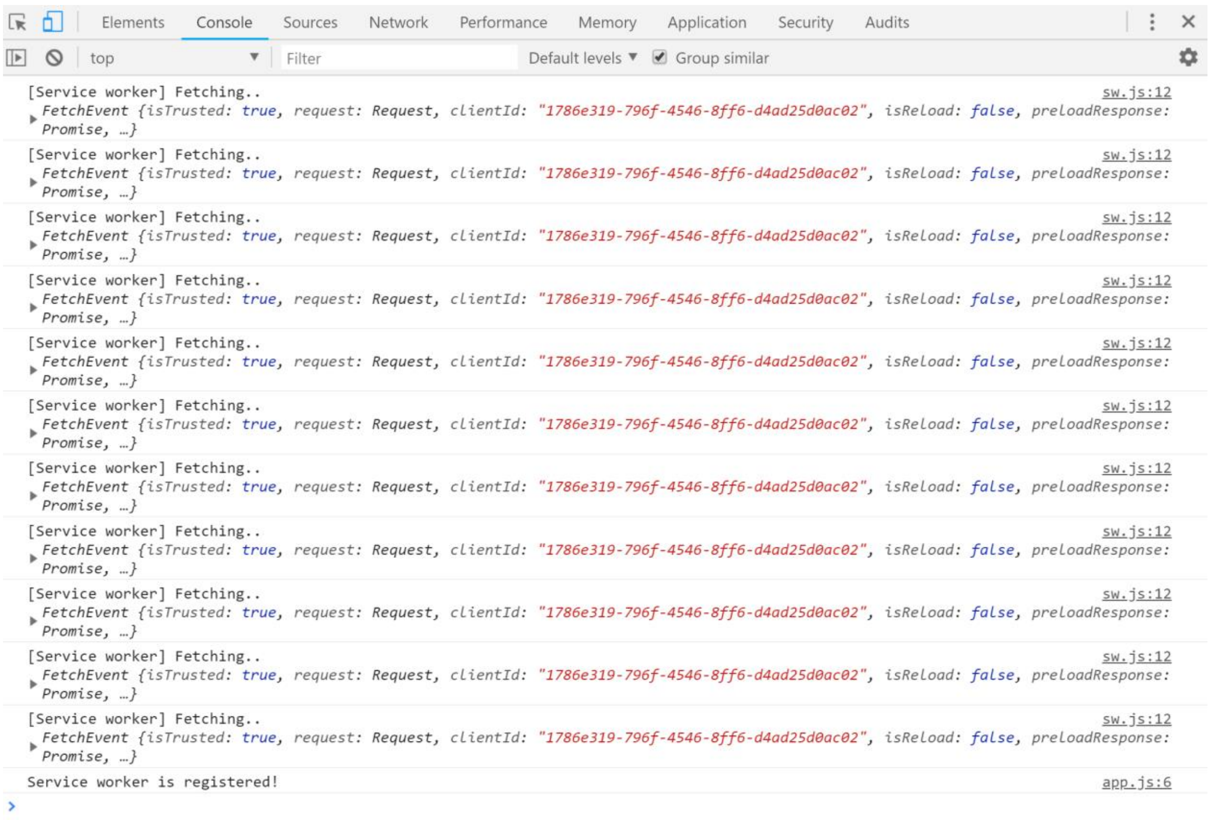
Picture 18. Service worker post activation.

5.3 None life-cycle events

Service workers use fetch events. A fetch event is a none life-cycle event. It is emitted when an HTML page loads a JavaScript file, a CSS file, or an image. Fetch also triggers when manually fetching a request event.

```
self.addEventListener('fetch',function(event){
 console.log('[Service worker] Fetching..',event);
});
```

This code will console.log whenever a fetch event occurs. In the browser, the following fetch events can be observed (Picture 19).



```
[Service worker] Fetching..                                                                              sw.js:12
▸ FetchEvent {isTrusted: true, request: Request, clientId: "1786e319-796f-4546-8ff6-d4ad25d0ac02", isReload: false, preloadResponse:
  Promise, …}
[Service worker] Fetching..                                                                              sw.js:12
▸ FetchEvent {isTrusted: true, request: Request, clientId: "1786e319-796f-4546-8ff6-d4ad25d0ac02", isReload: false, preloadResponse:
  Promise, …}
[Service worker] Fetching..                                                                              sw.js:12
▸ FetchEvent {isTrusted: true, request: Request, clientId: "1786e319-796f-4546-8ff6-d4ad25d0ac02", isReload: false, preloadResponse:
  Promise, …}
[Service worker] Fetching..                                                                              sw.js:12
▸ FetchEvent {isTrusted: true, request: Request, clientId: "1786e319-796f-4546-8ff6-d4ad25d0ac02", isReload: false, preloadResponse:
  Promise, …}
[Service worker] Fetching..                                                                              sw.js:12
▸ FetchEvent {isTrusted: true, request: Request, clientId: "1786e319-796f-4546-8ff6-d4ad25d0ac02", isReload: false, preloadResponse:
  Promise, …}
[Service worker] Fetching..                                                                              sw.js:12
▸ FetchEvent {isTrusted: true, request: Request, clientId: "1786e319-796f-4546-8ff6-d4ad25d0ac02", isReload: false, preloadResponse:
  Promise, …}
[Service worker] Fetching..                                                                              sw.js:12
▸ FetchEvent {isTrusted: true, request: Request, clientId: "1786e319-796f-4546-8ff6-d4ad25d0ac02", isReload: false, preloadResponse:
  Promise, …}
[Service worker] Fetching..                                                                              sw.js:12
▸ FetchEvent {isTrusted: true, request: Request, clientId: "1786e319-796f-4546-8ff6-d4ad25d0ac02", isReload: false, preloadResponse:
  Promise, …}
[Service worker] Fetching..                                                                              sw.js:12
▸ FetchEvent {isTrusted: true, request: Request, clientId: "1786e319-796f-4546-8ff6-d4ad25d0ac02", isReload: false, preloadResponse:
  Promise, …}
[Service worker] Fetching..                                                                              sw.js:12
▸ FetchEvent {isTrusted: true, request: Request, clientId: "1786e319-796f-4546-8ff6-d4ad25d0ac02", isReload: false, preloadResponse:
  Promise, …}
[Service worker] Fetching..                                                                              sw.js:12
▸ FetchEvent {isTrusted: true, request: Request, clientId: "1786e319-796f-4546-8ff6-d4ad25d0ac02", isReload: false, preloadResponse:
  Promise, …}
Service worker is registered!                                                                            app.js:6
>
```

Picture 19. Service workers fetch events.

A fetch event handler can perform more than just logging in the console. It can use the events that get passed into the event handler and overwrite the data that gets sent back.

A Service worker behaves like a network proxy. Triggering the fetch event sends the fetch to the service workers. The fetch response is simply passed on; however, it can be overwritten using the (event.respondWith) method.

In this chapter, a service worker was created and registered for every page in the application. The service worker runs in the background and will be used later to implement the caching technology. Moreover, the service worker implemented fetch events. Fetch events are discussed thoroughly in the next chapter.

6 IMPLEMENTING PROMISE AND FETCH API

As mentioned earlier, Both fetch API and promise API are native applications technologies. This chapter implements both APIs, then adds browser polyfills to further enhance browser support.

6.1 Promises implementation

As mentioned earlier, promises return one of two results under all circumstances, either **resolve** a value or **reject** a value (return an error).

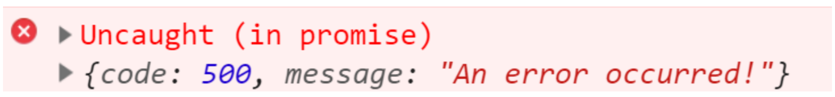
```
var promise = new Promise(function(resolve, reject){ }
```

The `then()` method is used to react to the promise resolve value. A function is inserted inside the `then()` method and it is executed when the promise resolve is called. It returns the resolved value.

```
promise.then(function(text) { });
```

A promise is useful when dealing with multiple asynchronous tasks chained together. The `then()` methods are chained one after another. It is an easy and convenient way to handle values that are related to chained tasks.

Promises might get rejected as well. By default, if a promise is not called to handle reject cases, the following error will be displayed (Picture 20).



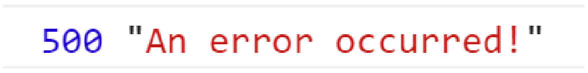
```
▶ Uncaught (in promise)
▶ {code: 500, message: "An error occurred!"}
```

Picture 20. Promise rejection.

A rejection case can be defined by passing another function as a second argument.

```
promise.then(function(test) {
 return test;
},function(error){
 console.log(error.code, error.message)
});
```

Executing this code displays the following (Picture 21).



```
500 "An error occurred!"
```

Picture 21. Promise error function.

This is one way to handle rejections; however, the most common way is to catch the rejection errors. This is implemented by adding a catch() method, then passing the function to be executed on rejected cases.

```
promise.then(function(text) {
 return text;
}).catch(function(error){
 console.log(error.code, error.message)
});
```

The catch() method catches any errors occuring at any step before catch() is called. then() and catch() are crusial methods. The then() method is used to return normal case values, while catch() is used to handle errors. Consequently, it is important to add a catch() method to handle any errors occurring within the registration process.

```
if ('serviceWorker' in navigator){
 navigator.serviceWorker
 .register('/sw.js')
 .then(function(){
 console.log('Service worker is registered!');
}).catch(function(error){
 console.log(error);
}); }
```

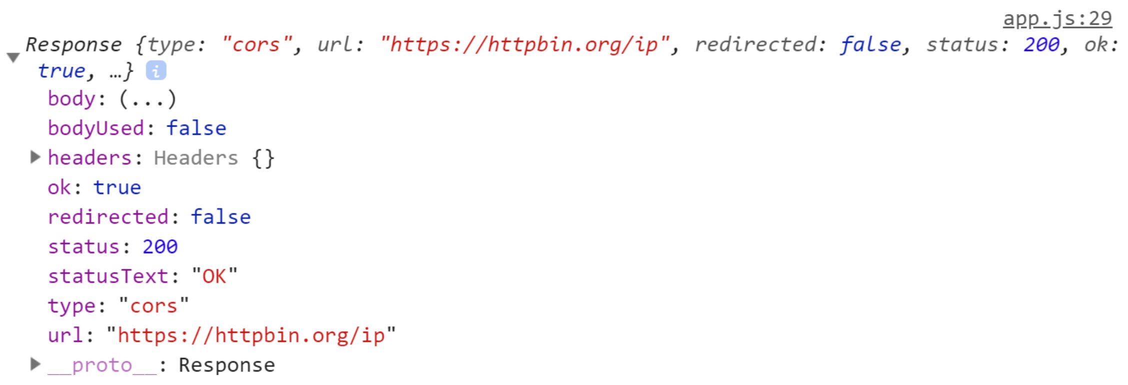
6.2 Fetch Implementation

As mentioned earlier, fetch is a JavaScript method. It produces network HTTP requests and returns promises. Fetch is a clean and easy method accustomed to get, transfer and read data from a foreign source such as an external web page.

Fetching data from a dummy fetch source.

```
fetch('https://httpbin.org/ip')
.then(function(response){
  console.log(response);
});
```

Results in the following response (Picture 22).



```
                                                                        app.js:29
▼Response {type: "cors", url: "https://httpbin.org/ip", redirected: false, status: 200, ok:
  true, …}
   body: (...)
   bodyUsed: false
  ▶headers: Headers {}
   ok: true
   redirected: false
   status: 200
   statusText: "OK"
   type: "cors"
   url: "https://httpbin.org/ip"
  ▶__proto__: Response
```

Picture 22. Fetching data from a dummy source.

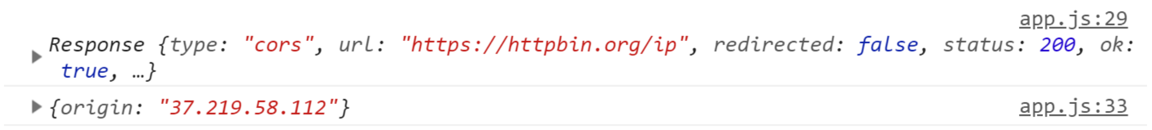
Fetch runs a JSON method. It is a utility method provided by the fetch() API and used to convert the fetch response into a JavaScript object.

```
return response.json();
```

Further, a then() method is used to create a promise of the current fetch().

```
.then(function(data){ }
```

The application console displays the origin and the parsed JSON data as shown in picture 23.

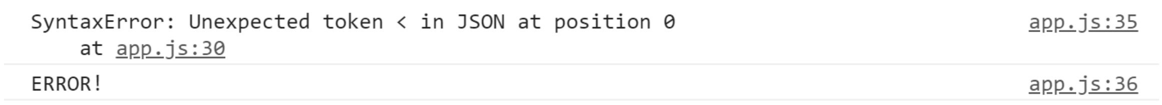


```
                                                                                app.js:29
▶ Response {type: "cors", url: "https://httpbin.org/ip", redirected: false, status: 200, ok:
   true, …}
▶ {origin: "37.219.58.112"}                                                     app.js:33
```

Picture 23. Displaying CORS response.

A fetch is similar to other operations and indeed, it might fail. A catch() method is added to detect potential errors.

```
.catch(function(error){
  console.log(error);
console.log('ERROR!'); });
```



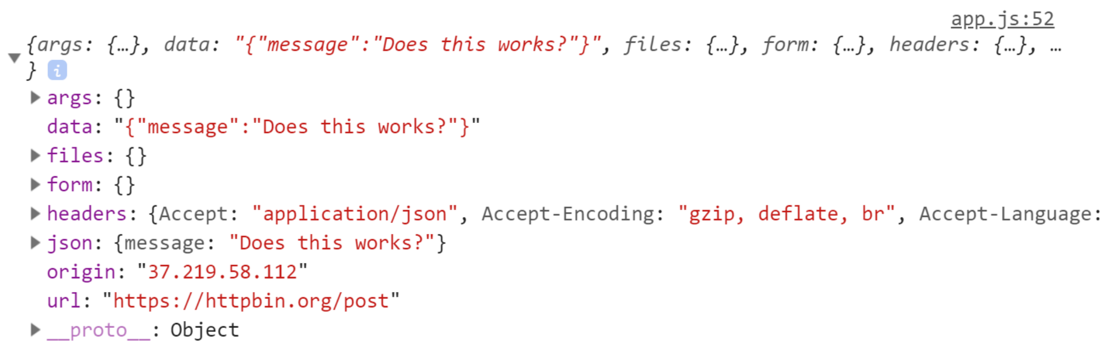
```
SyntaxError: Unexpected token < in JSON at position 0                          app.js:35
    at app.js:30
ERROR!                                                                          app.js:36
```

Picture 24. Fetch event error.

A post request is configured by passing a second argument to the fetch method, then passing a JavaScript object inside the argument. It is used to specify options for the post request.

```
fetch('https://httpbin.org/post', {
 method: 'POST',
 headers: {
  'Content-type': 'application/json',
  'Accept': 'application/json'
}
 body: JSON.stringify({message: 'Does this works?'})
})
```

Finally, JSON.stringify is used to convert the object array into JSON data as shown in picture 25.

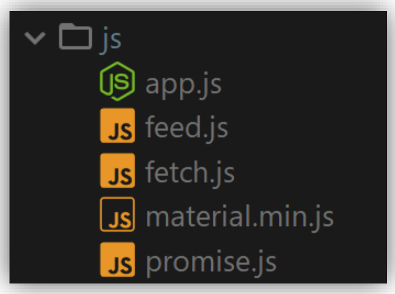


```
                                                                           app.js:52
▼{args: {…}, data: "{"message":"Does this works?"}", files: {…}, form: {…}, headers: {…}, …
 } ℹ
 ▶args: {}
  data: "{"message":"Does this works?"}"
 ▶files: {}
 ▶form: {}
 ▶headers: {Accept: "application/json", Accept-Encoding: "gzip, deflate, br", Accept-Language:
 ▶json: {message: "Does this works?"}
  origin: "37.219.58.112"
  url: "https://httpbin.org/post"
 ▶__proto__: Object
```

Picture 25. POST requests object elements.

6.3 Adding polyfills for legacy browser support

The PWAs technology aims to progressively enhance the application. The fetch() and the promise() methods lack support on older browsers, this reflects in an error in the browsers that fail to offer the support. However, adding polyfills for both the fetch() and the promise() methods manipulates the browsers' settings and forces all browsers to offer support for those methods. Polyfills recreates the methods functionalities for non-supporting browsers. The polyfills are added to the JS folder and are imported to the index.html file before importing any other JavaScript code.



Picture 26. Feed.js and Fetch.js polyfills.

```
<script src="src/js/promise.js"></script>
<script src="src/js/fetch.js"></script>
```

Back in the app.js file, the code is modified to check the browser support for the fetch() and the promise() methods. Polyfills are included in case support is not available. Accordingly, the fetch() and the promise() methods can be used in all browsers. The fetch event listener inside the service worker triggers when an implicit request occurs in the HTML page or when a fetch request occurs inside the JavaScript code.

In this chapter, the fetch API and the Promise API methods were implemented in the applications' service worker. Fetches and promises are used repeatedly in the process of building the application and it is further implemented in the following chapters.

7 CACHING THE SERVICE WORKER

As mentioned earlier, caching is the process of storing data in a temporary storage. The browser uses the cache storage data instead of requesting the same data again from the web page server. This chapter implements different cache storages in the service worker.

7.1 Static Cache

The HTML page consists of static elements, such as menu, buttons, images and static JavaScript code. The static elements are cachable data, occasionally modified and makes up the app shell. On the other hand, dynamic data is an asynchronously changing data when new updates become obtainable. Dynamic data is cached using the JavaScript code in the application; however, it is excluded from the app shell.

The app shell consisting of static elements is cached to provide an active and reachable application view when the application is offline even when dynamic content is missing.

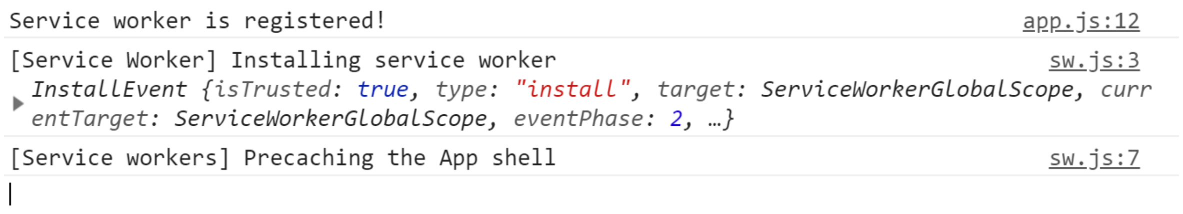
As it has been mentioned, the service worker is re-installed on the update of the service worker file; consequently, it is an appropriate interval to cache the static data such as toolbars and basic styling using the cache API. Afterwards, the static files are fetched to provide an accessible offline web application. Further, static files are stored during the installation of the service workers; thus, static files become available immediately after the first page reload. The application accesses the cache API during the install-event using the caches.open() method.

Since the service worker is an asynchronous code file, other events might try to access the cache even before the install-cache finishes loading. Therefore, the event.waitUntil() method wraps the caches.open() method to prevent such inconveniences. Afterwards, it returns a promise; moreover, the loading of other functions proceeds after the promise is returned.

```
event.waitUntil(
 caches.open('static')
 .then(function(cache){ })
 )
```

Inside the promise, a function with a cache argument is passed. This passes a reference to the functions cache; hence, data can be saved to the cache storage using cach.add(). This allows adding new resources to the cache storage (Picture 27).

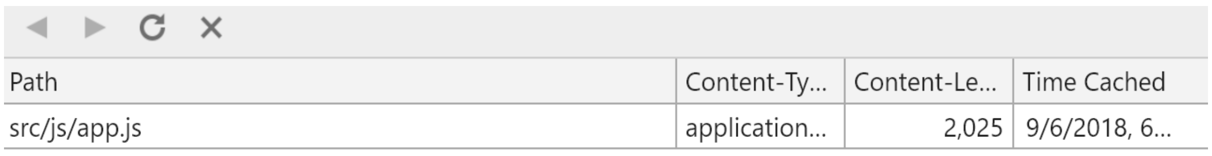
```
self.addEventListener('install', function(event){
 console.log('[Service Worker] Installing service worker ', event);
 event.waitUntil(
  caches.open('static')
  .then(function(cache){
    console.log('[Service workers] Precaching the App shell');
    cache.add('src/js/app.js')
  })
 )
});
```



```
Service worker is registered!                                        app.js:12
[Service Worker] Installing service worker                            sw.js:3
▶ InstallEvent {isTrusted: true, type: "install", target: ServiceWorkerGlobalScope, curr
  entTarget: ServiceWorkerGlobalScope, eventPhase: 2, …}
[Service workers] Precaching the App shell                            sw.js:7
```

Picture 27. Precaching static elements.

After the application reinstalls the service worker, it can be observed in the applications' console that it has successfully pre-cached the app shell. The cached paths can be observed in picture 28.



Path	Content-Ty...	Content-Le...	Time Cached
src/js/app.js	application...	2,025	9/6/2018, 6...

Picture 28. Precached app.js.

The application has cached initial data on the install event of the service worker; however, no data has been fetched yet. A fetch is carried out by modifying the fetch event listeners in the service workers. A respond function is used to respond to caches referring to the overall cache storage.

```
event.respondWith()
```

Afterwards, an event request is matched to search through the sub-caches seeking triggered cache requests.

```
caches.match(event.request);
```

Consequently, a promise response is used to confirm the result of the requests search.

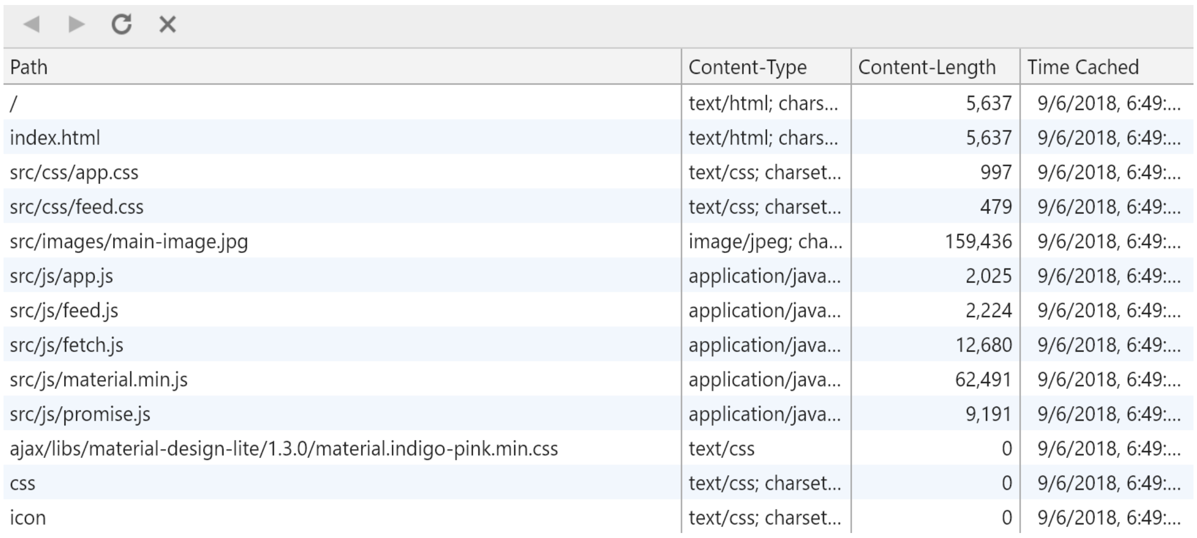
```
.then(function(response){ }
```

The previous function returns the response if it exists; however, it returns null if no response is found.

```
self.addEventListener('fetch',function(event){
 event.respondWith(
  caches.match(event.request)
  .then(function(response){
   if(response) {
    return response;
   } else {
    return fetch(event.request);
    }
  })
 );
});
```

The fetch event applies to all static files. The addAll() method is used to add all the static resources in an array form (Picture 29).

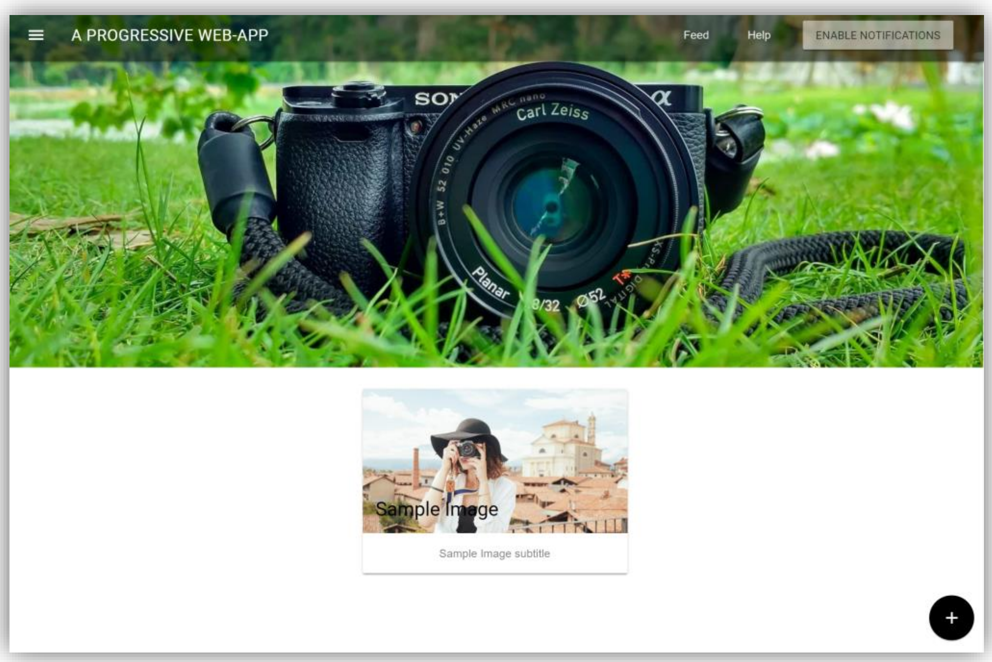
```
cache.addAll([
'/',
'/index.html',
'/src/js/app.js',
'/src/js/feed.js',
'/src/js/promise.js',
'/src/js/fetch.js',
'/src/js/material.min.js',
'/src/css/app.css',
'/src/css/feed.css',
'/src/images/main-image.jpg',
'https://fonts.googleapis.com/css?family=Roboto:400,700',
'https://fonts.googleapis.com/icon?family=Material+Icons',
'https://cdnjs.cloudflare.com/ajax/libs/material-design-
lite/1.3.0/material.indigo-      pink.min.css'
]);
```



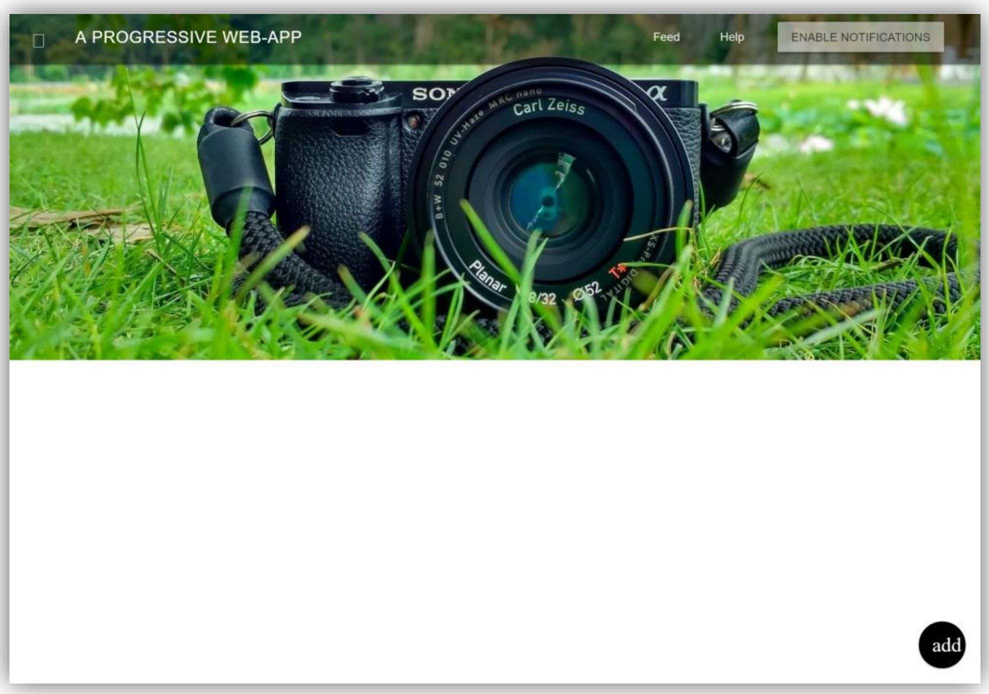
Path	Content-Type	Content-Length	Time Cached
/	text/html; chars...	5,637	9/6/2018, 6:49:...
index.html	text/html; chars...	5,637	9/6/2018, 6:49:...
src/css/app.css	text/css; charset...	997	9/6/2018, 6:49:...
src/css/feed.css	text/css; charset...	479	9/6/2018, 6:49:...
src/images/main-image.jpg	image/jpeg; cha...	159,436	9/6/2018, 6:49:...
src/js/app.js	application/java...	2,025	9/6/2018, 6:49:...
src/js/feed.js	application/java...	2,224	9/6/2018, 6:49:...
src/js/fetch.js	application/java...	12,680	9/6/2018, 6:49:...
src/js/material.min.js	application/java...	62,491	9/6/2018, 6:49:...
src/js/promise.js	application/java...	9,191	9/6/2018, 6:49:...
ajax/libs/material-design-lite/1.3.0/material.indigo-pink.min.css	text/css	0	9/6/2018, 6:49:...
css	text/css; charset...	0	9/6/2018, 6:49:...
icon	text/css; charset...	0	9/6/2018, 6:49:...

Picture 29. Precached static elements.

So far the static cache enhances the web application to perform few functionalities while offline (Picture 31).



Picture 30. The Application online version.



Picture 31. The application offline version 1.

7.2 Dynamic cache

Data can be added dynamically to the cache. Some dynamic caching is handled using the pre-caching method, it is similar to pre-caching static files. A manual fetch event request can be performed inside the index.html file or the app.js file to store data in the cache. Afterwards, the application reaches out to the network to fetch the stored data. Using this type of dynamic caching, items are dynamically added to the cache; however, this does not necessarily offer the caching of dynamic contents such as user inputs or rapidly modified data.

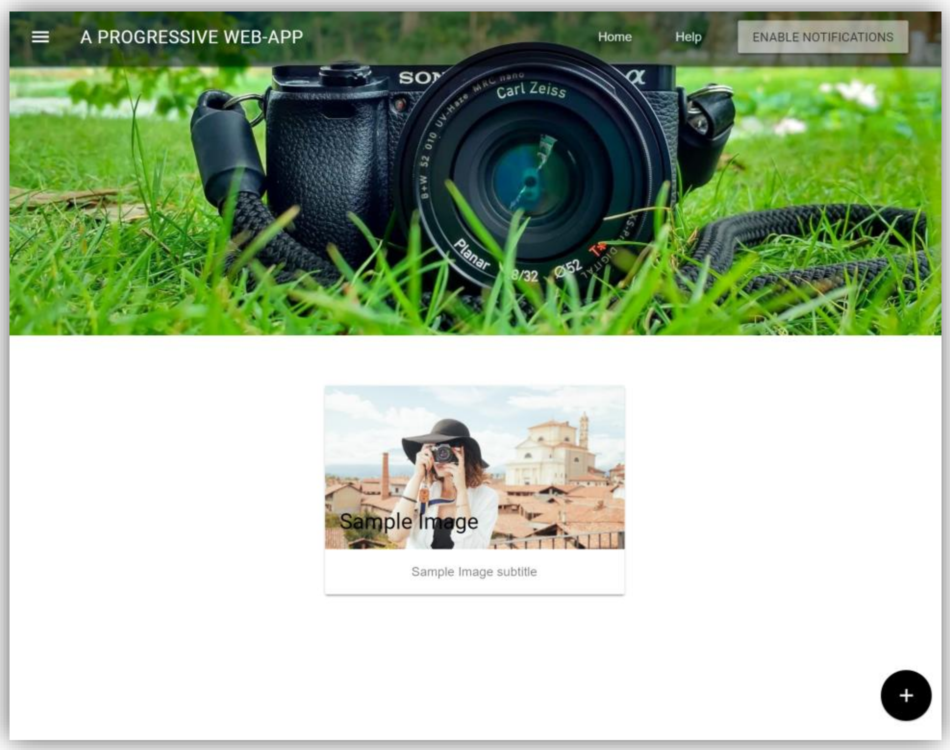
Dynamic cache relies on storing the fetch event response into the cache storage. A fetch event listener is implemented to perform dynamic caching. A promise is chained to store the fetch response. Afterwards, a cache is created inside the promise using the cache.put() method, it puts a new resource and stores data. The cache.put() method takes two arguments:

1. The first argument is the event request URL.
2. The second argument is a response.

Additionally, cloning the response is needed to reuse the original response, since it becomes consumed after storing it.

```
self.addEventListener('fetch', function(event) {
 event.respondWith(
 caches.match(event.request)
 .then(function(response) {
  if (response) {
   return response;
  } else {
   return fetch(event.request)
   .then (function (res) {
    caches.open('dynamic')
    .then(function(cache){
     cache.put(event.request.url, res.clone());
     return res;
    })
  });
 }
})
```

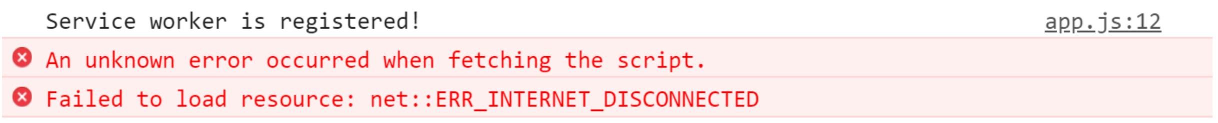
Accordingly, the application reaches out to the network and caches data to the cache storage when data becomes available. This enhances the offline mode of the application under development (Picture 32).



Picture 32. The application offline version 2.

7.3 Handling errors

The application encountered the following errors while fetching (Picture 33).

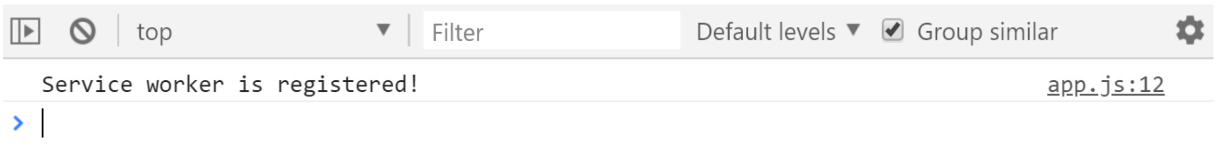


```
  Service worker is registered!                                   app.js:12
⊗ An unknown error occurred when fetching the script.
⊗ Failed to load resource: net::ERR_INTERNET_DISCONNECTED
```

Picture 33. Fetch process errors.

The previously failed fetch request errors are handled by adding a simple empty cache request to handle errors, it simply commands the application to ignore such errors.

```
.cache(function(err){  });
```



```
Service worker is registered!                                   app.js:12
>
```

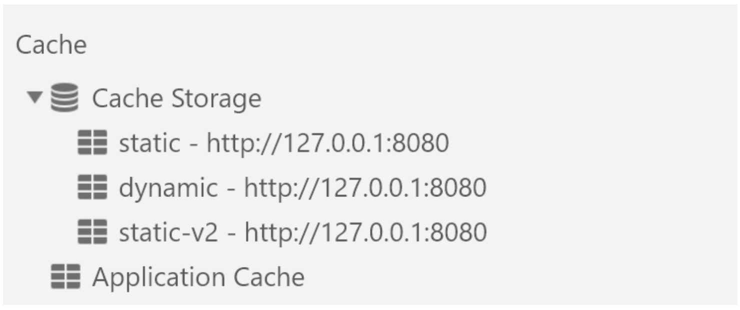
Picture 34. Ignoring fetch errors.

7.4 Updating caches and cleanup

The static cache is filled with pre-cached data, while the dynamic cache is filled with data that got stored after fetching it from the network. Currently the application stores all data inside the dynamic storage, whether it is required or not. This is Inconvenient in several cases, for example when caching a rapidly updating dynamic data from an external server. Storing such data is unnecessary since old data would be displayed in an offline mode. Further, the application aims to store static files dynamically, such as images, CSS files, and HTML files.

On the other hand, updating a static file, such as a CSS property or a JavaScript file takes no effect inside the service worker of the application, since re-installing the service worker in the application requires an update of the service worker file itself. Thus, the application uses code in the service workers to force an update. To update the service worker, a change in the JavaScript code is required. A common way of handling this matter is applying version names to the cache; for example, renaming the static cache to static-v2 will reinstall the service worker on page reload (Picture 35).

```
caches.open('static-v2')
```


Cache

Cache Storage

static - http://127.0.0.1:8080

dynamic - http://127.0.0.1:8080

static-v2 - http://127.0.0.1:8080

Application Cache

Picture 35. Adding static-v2 cache.

Furthermore, renaming the cache avoids reusing the previous cache and guarantees immediate installation of the new cache without breaking the application performance on cache update.

On cache update, the new cache fails to replace the previous cache; instead, the application saves both caches in the cache storage and fetches from both caches. For this reason, the application controls the fetch process and handles old fetches cleanup.

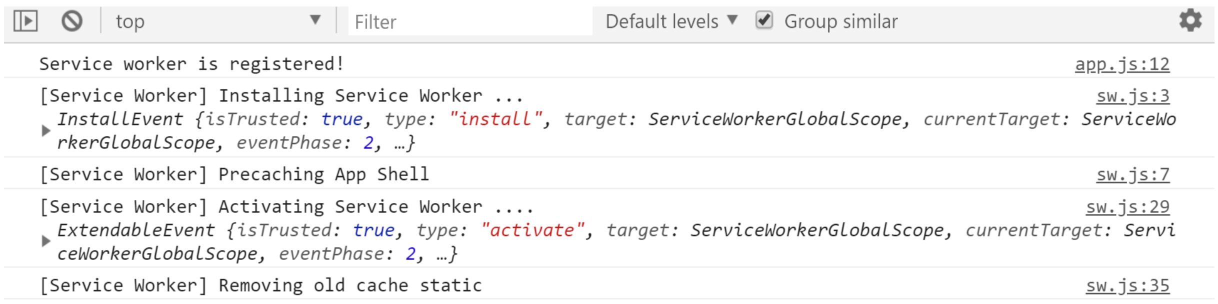
The application handles the cache cleanup within the fetch activation event since the activation happens after reloading the application and installing the new cache version. A waitUntil() event method is implemented for a guaranteed completion before handling more code inside the service worker.

Afterwards, the code proceeds with a cache object referring to the general cache storage followed by keys. Keys return all the sub-caches keys found in the cache storage using a promise. The promise method returns a keyList of all sub-caches.

```
Event.waitUntil()(
Chaches.keys().then(function(keyList){ }) );
```

This is followed by returning the promise.all() method, it is used to return the complete list of promises after transforming the array of strings into an array of promises. Afterwards, a loop takes place to delete old fetches and keep the new ones (Picture 36).

```
if (key !=='static-v2' && key !== 'dynamic'){
  console.log('[Service Worker] Removing old cache', key);
  return caches.delete(key);
}
```



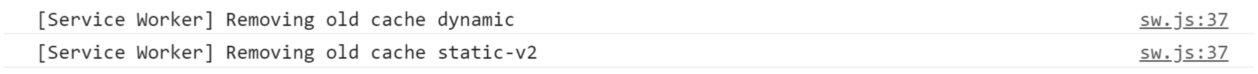
```
Service worker is registered!                                                     app.js:12
[Service Worker] Installing Service Worker ...                                     sw.js:3
▸ InstallEvent {isTrusted: true, type: "install", target: ServiceWorkerGlobalScope, currentTarget: ServiceWo
rkerGlobalScope, eventPhase: 2, …}
[Service Worker] Precaching App Shell                                              sw.js:7
[Service Worker] Activating Service Worker ....                                    sw.js:29
▸ ExtendableEvent {isTrusted: true, type: "activate", target: ServiceWorkerGlobalScope, currentTarget: Servi
ceWorkerGlobalScope, eventPhase: 2, …}
[Service Worker] Removing old cache static                                         sw.js:35
```

Picture 36. Removing static-v1 cache.

A general method is applied for a better use since the cache version names are updated frequently. Using variable holders optimizes the code and enhances it for future use.

```
var CACHE_STATIC_NAME = 'static_v3';
var CACHE_DYNAMIC_NAME = 'dynamic_v2';
```

On code update, the version number is edited only allowing the service worker to reinstall. This ensures loading the latest version of the application file code. Additionally, the cache storage does not get populated with useless previous cache versions (Picture 37).



```
[Service Worker] Removing old cache dynamic                                        sw.js:37
[Service Worker] Removing old cache static-v2                                      sw.js:37
```

Picture 37. Removing outdated cache.

7.5 Advanced caching

7.5.1 Cache on demand

For the following, the dynamic cache is temporarily disabled by commenting out the dynamic cache.put() event to prevent the dynamic caching.

```
// cache.put(event.request.url, res.clone());
```

A new card <div> is added dynamically to the shared moments <div>.

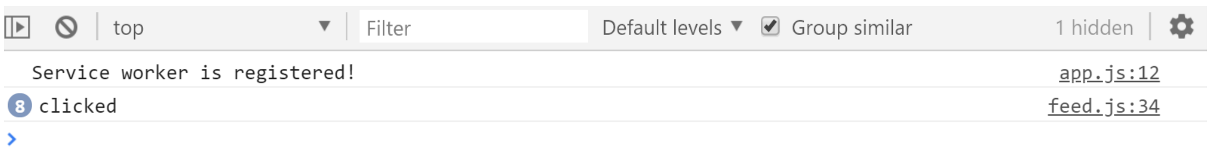
```
<div id="shared-moments"></div>
```

Inside the createCard() function in the feed.js file, a button function is implmented to save a card when required and it is followed by a button click event.

```
var cardSaveButton = document.createElement('button');
cardSupportingText.appendChild(cardSaveButton);
cardSaveButton.textContent = 'save';
cardSaveButton.addEventListener('click', onSaveButtonClicked);
```

The onSaveButtonClicked() function is emplemented to log a string message (Picture 38).

```
function onSaveButtonClicked(event){
  console.log('clicked');
}
```



```
Service worker is registered!                    app.js:12
8 clicked                                        feed.js:34
```

Picture 38. Testing onSaveButtonClicked(event) function.

After verifying the button functionality, data is fetched on the click of the button. Currently, the dummy fetch function fetches fake data from https//httpbin.org/get.

```
fetch('https://httpbin.org/get')
```

```
  .then(function(res) {
  return res.json();
  }) .then(function(data) {
  createCard();
  });
```

Afterwards, data is cached on the button-click event with the help of the onSaveButtonClicked() function. It accesses the cache storage through the front end of the application inside the JavaScript file via the same method used in the service worker. Correspondingly, a cache method with a chained promise method is implemented.

```
caches.open('user-requested') .then();
```

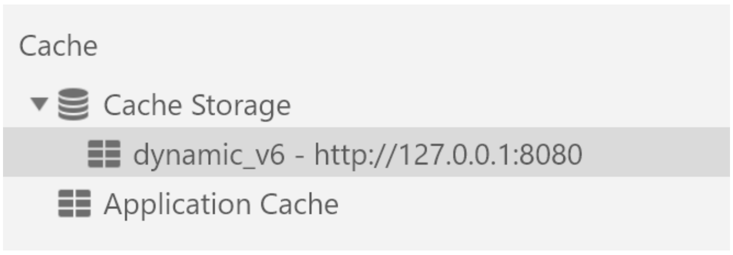
After, it checks for browser support using an if statement.

```
if ('cahces' in window) {
 caches.open('user-requested') .then();
}
```

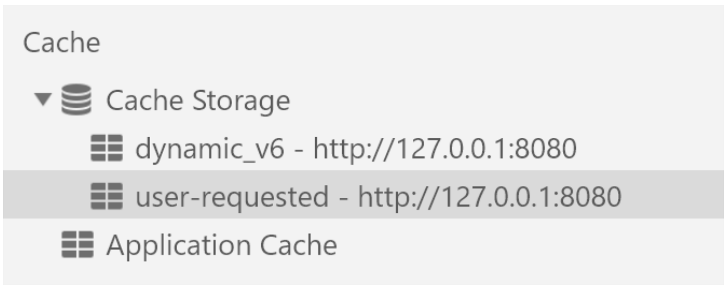
On supportive browsers, it stores the data using the cache.add() method.

```
function onSaveButtonClicked(event) {
 console.log('clicked');
 if ('caches' in window) {
  caches.open('user-requested')
  .then(function(cache){
  cache.add('/src/images/4.jpg');
  cache.add('https://httpbin.org/get');
  });
 }
}
```

After re-installing the service worker, on pressing the save button, the user request gets cached to the cache storage immediately (Picture 40).



Picture 39. Before user-requested caching.


Cache
Cache Storage
dynamic_v6 - http://127.0.0.1:8080
user-requested - http://127.0.0.1:8080
Application Cache

Picture 40. After user-requested caching.

The previous method allows to save data using cache on demand when the user requests the data to be cached; however, this method will not be implemented in the developed application in this project.

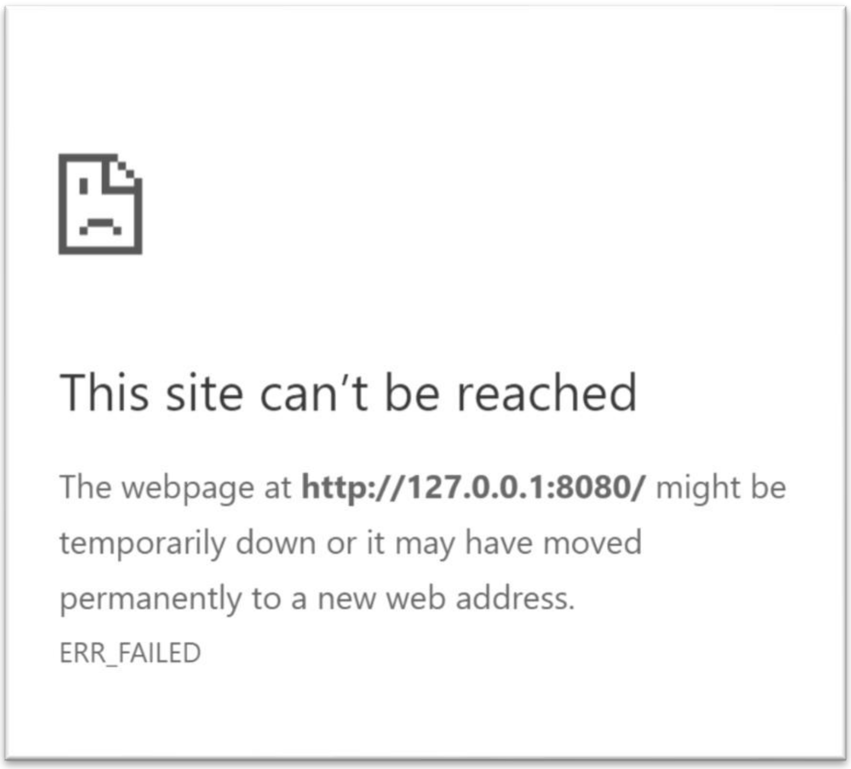
7.5.2 Offline fallback page

The offline fallback page method targets a crucial issue regarding the offline application use. The application includes two pages; the home page and the help page. Both pages get cached to the cache storage after one visit to the page. However, if the help page is not visited in the online mode, then it fails to load in the offline mode since the page skipped the fetch-then-cache process. A default fallback page is a page presented when an unvisited page hops the cache process, it is used to enhance the user experience. The offline fallback page is added to the static cache inside the service worker; consequently, it gets pre-cached as a static element.

The offline.html page has been cached and can be accessed when offline. Further, the fetch response is modified. The offline fallback page is implemented inside an error catching function and it is displayed when a certain page fails to cache a fetch response from the application or from the web.

```
.catch(function(err) {
 return caches.open(CACHE_STATIC_NAME)
 .then(function (cache){
   return cache.math('/offline.html');
  });
 });
}
```

The offline fallback page enhances the user experience, it covers up caching errors received from the application pages and prevents throwing an error even when an error occurs. As a result, it displays a functional page (Picture 42) instead of displaying a "site can't be reached" error page (Picture 41).

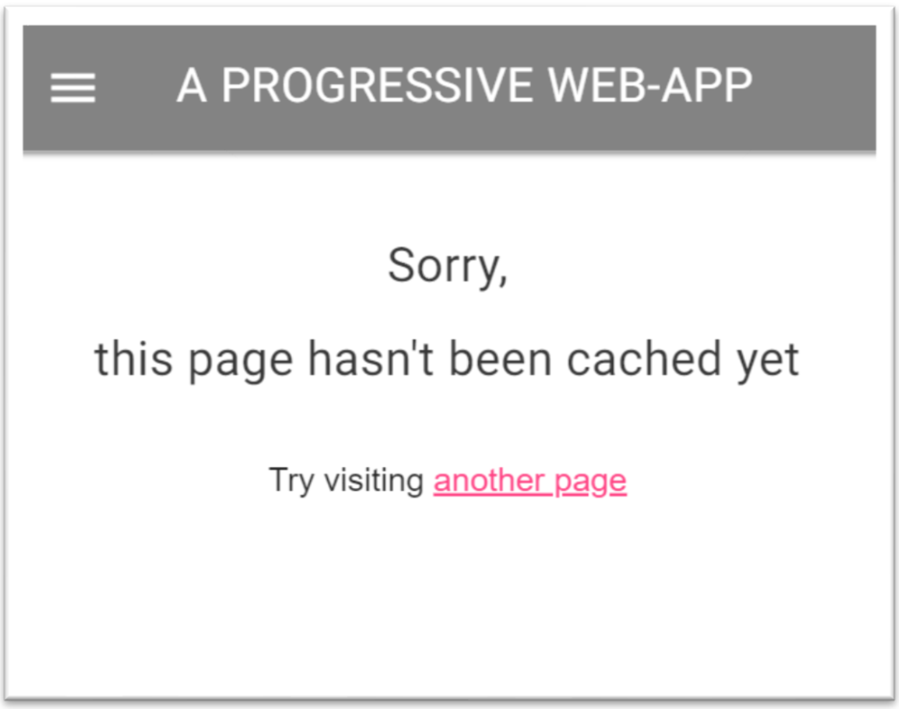


This site can't be reached

The webpage at **http://127.0.0.1:8080/** might be temporarily down or it may have moved permanently to a new web address.

ERR_FAILED

Picture 41. Site can't be reached error.


A PROGRESSIVE WEB-APP

Sorry,

this page hasn't been cached yet

Try visiting another page

Picture 42. Fallback page.

This chapter provided an offline functionality to the PWA by adding the service worker caching. The service worker implemented static caching, dynamic caching, cache cleanup and handled possible errors. Moreover, it provided an offline fallback page when the network is unavailable. More advanced caching strategies are presented and implemented in the next chapter.

8 CACHING STRATEGIES

This chapter focuses on the cache process. It suggests several different approaches to cache data and implements an appropriate cache strategy tailored for the PWA under development.

8.1 Caching-with-network-fallback strategy

The caching-with-network-fallback strategy is a caching strategy that falls back to the network if the attempt to retrieve a cache request fails, it is implemented as follows.

1. The service worker attempts to access cache requests sent by the web page.
2. The service worker handles the cache.

   2.1 If the cache resource is found, it is directly returned.

   2.2 Otherwise, the service worker executes an additional phase. It reaches out to the network seeking a network response, then stores the response in the cache.

This strategy stores a network fallback. It is accessed when needed after the cache process; however, this strategy parses complete data and keeps it in the cache storage even when unnecessary. Thus, the cache storage overpopulates quickly, this is a major drawback when caching dynamic data, since it updates rapidly.

8.2 Cache-only strategy

The cache-only strategy is fairly simple, it is implemented as follows.

1. The service worker attempts to access cache requests sent by the web page.
2. The service worker handles the cache.

   2.1 If the cache resource is found, it is directly returned.

   2.2 Otherwise, the service worker ignores the cache request.

Implementing this strategy is simple, the fetch event-listener overlooks the network. In general, the cache-only strategy tends to be insufficient for real-time applications.

```
self.addEventListener('fetch', function(event) {
  event.respondWith(
   caches.match(event.request)
  );
});
```

8.3 Network-only strategy

The service worker fails to operate in the network-only strategy. This strategy directs a request to the network and consumes the returned response. It works similar to the cache-only strategy; however, it uses a fetch request-event instead of the cache storage API. Network-only strategy is implemented when the application requires the solitary use of network requests.

```
self.addEventListener('fetch', function(event) {
  event.respondWith(
   caches.match(event.request)
  );
});
```

8.4 Network-with-cache-fallback strategy

Network-with-cache-fallback is a compelling strategy. It caches a request in the service worker by reaching out to the network. However, the network-with-cache-fallback strategy implements the stored cache only if the network fetch fails; otherwise, it returns the network response and ignores the stored cache.

The network-with-cache-fallback seems like a decent strategy since it stores assets in the cache storage but only uses it if network access fails. However, this strategy is unreliable when the network fails, because it only reacts after the network fails to fetch the data. In case of poor connection, the network fetch failure can be a long process and might take up to 60 seconds; hence, the fetching from cache storage is delayed up to 60 seconds. This generates a dreadful user experience.

Implementing such a strategy requires sending a network event request as a fetch request, followed by the attempt to access the cache storage.

```
self.addEventListener('fetch', function(event) {
  event.respondWith(
   fetch(event.request)
   .cache(function(err) {
   return  caches.match(event.request);
   })
  );
});
```

8.5 Cache-then-network strategy

Cache-then-network strategy attempts to receive data assets from the cache as rapidly as possible, then tries to fetch a more up to date version via the network. Cache-then-network strategy fetches an initial version cache from the cache storage even if the network fetch fails. Furthermore, it overwrites the cache if the network fetch process completes. Cache-then-network strategy is a dynamic process that keeps the cache data up to date. This strategy is explained in the following steps.

1. The page directly reaches out to the cache and accesses it, while the service worker intercepts a network request sent by the page. Both processes occur simultaneously when reaching out to the cache.

2. It returns the cache data while the service worker reaches out to the network and attempt to receive a network response.

3. Then it returns the network response to the service worker.

4. The data fetched from the network gets stored in the cache storage using the dynamic cache.

5. Fetched data from network gets returned to the page.

Implementing this powerful strategy requires updating the JavaScript code. Inside the feed.js file, a request is issued using JavaScript and it manually accesses the cache. The strategy is implemented as follows.

1. Initiates a browser support check.

   ```
   if ('caches' in window) { }
   ```

2. Searches for the required URL inside the cache storage.

   ```
   var url = 'https://httpbin.org/get';

   if ('caches' in window)

   {caches.match('url') }
   ```

3. Checks if data already exists in the cache.

   ```
   .then(function(response) { })
   ```

4. If the response doesn’t exist in the cache, it gets returned.

   ```
   .then(function(response){
      if (response){
      return response.json(); }
   ```

5. Fetches the URL.

   ```
   fetch(url)

   .then(function(res) {

   return res.json();  })
   ```

6. Chains a response to return new data.

   ```
   .then(function(data){

    console.log(From cache data');

    createCard(); }
   ```

This strategy forces the network caching to wait until the static caching process ends to prevent network caching from initiating the cache data in the presence of a fast network connection. This prevents the static cache from overwriting the network cache in such cases. It is implemented as follows.

1. It creates a variable to hold the state of the network data.
   ```
   var netWorkDataReceived = false;
   ```
2. Sets the value of the state to true after the network data is received.
   ```
   .then(function(data) {
    netWorkDataReceived = true;
    console.log('From web' , data);
    createCard(); });
   ```

Further, the strategy is implemented in the cache function to add a new card holding new data using the createCard () function only if such data does not already exist in the cache storage.

```
.then(function(data){
 console.log('From cache', data);
 if (!netWorkDataReceived){
  createCard();
 }
});
```

At last, the application needs a function to clear cards from the old data before attempting to dynamically add new data. A clearCards() function is created to clear all child nodes in the shared moments area.

```
function clearCards(){
 while (sharedMomentsArea.hasChildNodes()){
  sharedMomentsArea.removeChild(sharedMomentsArea.lastChild)
 }
}
```

Finally, it is implemented in the cache function before calling the createCard() function.

```
.then(function(data){
  console.log('From cache', data);
  if (!netWorkDataReceived) {
```

```
    clearCards();
    createCard();
  }
});
```

8.6 Cache-then-network-dynamic-caching strategy

The cache-then-network-dynamic-caching strategy is similar to the previous cache-then-network strategy. However, The data cached from the network is stored inside the cache storage, unlike the previous strategy that used the network caches without storing it. To save the network caches, the previous cache-then-network strategy is implemented inside the service worker as follows.

1. A fetch event-listener is added to respond to the dynamic cache since it is the cache that needs to get fetched after the installation of the service workers. It responses to the dynamic data and saves it after creating a new cache storage that is reserved for dynamic caching. Initially, the application cache storage is empty before visiting the application pages.

   ```
   self.addEventListener('fetch', function(event) {
    event.respondWith(
     caches.open(CACHE_DYNAMIC_NAME)
    )
   });
   ```

2. A promise function is called, it passes the new dynamic cache as an argument and stores the returned cached response. Then, it returns a fetch that saves the event request.

   ```
   .then(function(cache){
   return fetch(event.request)
   ```

3. A second promise is chained to return the response. This intercepts any requests sent by the front-end of the application, including the fetch requests implemented in the feed.js file.

4. Once the response is returned, a clone is sent to return the original response.

```
      .then(function(res){
        cache.put(event.request, res.clone());
       return res;
      });
```

When using this strategy, the application reaches out to the cache first. If data is available then it is displayed immediately; otherwise, performs a network request seeking a network response. Afterwards, it saves the network response in the dynamic cache storage. This method has proved to be a powerful method; although, offline usage is still restricted.

8.7 Cache-then-network-with-offline-support strategy

The previous cache-then-network-dynamic-caching strategy failed to offer offline access without fetching the dynamic cache. In this strategy, a dynamic data fetch is implemented by creating a dynamic cache fetch event-listener.

```
self.addEventListener('fetch', function(event) { }
```

The event listener handles the request in process. It analyzes the request, checks if requests are found, and handles new requests only since existing requests are already contained in the cache storage. Consequently, it checks the URL provided in the feed.js and stores it in a variable holder.

```
var url = 'https://httpbin.org/get'
```

After, it checks if the event request URL already contains that URL.

```
if(event.request.indexOf(url) > -1){ }
```

In conclusion, this method uses the cache-then-network counterpart for the URL found in feed.js; otherwise, it uses the cache-with-network-fallback strategy. Nevertheless, the service worker requires to load the app shell and the feed.js file first.

```
self.addEventListener('fetch', function(event) {
 var url = 'https://httpbin.org/get';
 if(event.request.indexOf(url) > -1) {
  event.respondWith(
   caches.open(CACHE_DYNAMIC_NAME)
    .then(function (cache) {
    return fetch(event.request)
    .then(function (res) {
     cache.put(event.request, res.clone());
     return res;
   });
  })
 );
} else{
 event.respondWith{
  caches.match(event.request)
  .then(function(response) {
   if (response) {
    return response;
    } else {
    return fetch(event.request)
    .then(function(res) {
     return caches.open(CACHE_DYNAMIC_NAME)
    .then(function(cache) {
    cache.put(event.request.url, res.clone());
    return res;
    })
   })
   .catch(function(err) {
   return caches.open(CACHE_STATIC_NAME)
   .then(function(cache) {
   return cache.match('/offline.html');
      });
     });
    }
   })
  }
 }
});
```

8.8 Cache strategies routing

The last method parses the incoming request URL in the service worker, then selects a strategy depending on the request type. It separates the requests in the code. The application consists of two pages, the home page, and the help page. The application is enhanced to check the page target. The offline fallback page is returned on failing to load the help page; however, caching the offline fallback page is unnecessary when the help page loads successfully. Hence, the code is modified to cache the offline fallback page only when the help page fails to load.

```
.catch(function (err) {
 return caches.open(CACHE_STATIC_NAME)
 .then(function (cache) {
  if(event.request.url.indexOf('/help'))
  return cache.match('/offline.html');
 });
});
```

8.9 Implementing the application cache

8.9.1 Applying Cache only

As mentioned earlier, the cache-only strategy induces several drawbacks; however, it is the most appropriate strategy for this application since a new service worker is pushed to the front end of the application on file update. The updated service worker offers updated files in the cache; hence, the application loads the files directly from the cache and skips attempting to implement the network fetch process as it is unnecessary. The service worker code is modified as follows.

1. A check is conducted to provide the cache state of the pre-cached static files. A variable holder is created, it contains the pre-cached static file in the form of an array.

```
var STATIC_FILES =  [
 '/',
 '/index.html',
 '/offline.html',
 '/src/js/app.js',
 '/src/js/feed.js',
 '/src/js/promise.js',
 '/src/js/fetch.js',
 '/src/js/material.min.js',
 '/src/css/app.css',
 '/src/css/feed.css',
 '/src/images/3.jpg',
 '/src/images/4.jpg',
 'https://fonts.googleapis.com/css?family=Roboto:400,700',
 'https://fonts.googleapis.com/icon?family=Material+Icons',
 'https://cdnjs.cloudflare.com/ajax/libs/material-design-
lite/1.3.0/material.indigo-pink.min.css'
] ;
```

2. Creating a regular expression to check if the URL contains any files found in the STATIC _ FILES array. This is conducted by joining all the static files with the word boundary operator (\\b) and checking if the expression of separate words matches the URL. Finally, it is followed by the test() method to test if the URL fits any of the words.

```
else if (new RegExp('\\b' + STATIC_FILES.join('\\b|\\b') +
'\\b').test(event.request.url))
```

3. The-cache-only strategy is implemented.

```
else if (new RegExp('\\b' + STATIC_FILES.join('\\b|\\b') +
'\\b').test(event.request.url)) {
 event.respondWith(
 caches.match(event.request)
 );
}
```

8.9.2 A better way of parsing static cache URLS

The previous strategy uses regular expression to check if the requested URL is contianed as a static file; however, A better approach is to create a helper function. The helper function isInArray() takes two arguments; a string argument and an array argument. It loops through the entire array with a for-loop to check if any of the strings is a part of the array and returns true when a match is found.

```
function isInArray(string,array){
 for (var i=0; i <array.length; i++){
  if (array[i] == string){
  return true;
  }
 }
return false;
}
```

The isInArray() function replaces the regular expression. The request URL is passed through this function as a string argument and the STATIC_FILES variable is passed as the array argument.

```
else if (isInArray(event.request.url , STATIC_FILES)) {
 event.respondWith(
 caches.match(event.request)
 );
}
```

8.9.3 A better way of serving fallback files

Currently, the application redirects the user to the offline fallback page after failing to access the help.html page.

```
if (event.request.url.indexOf('/help')) {
 return cache.match('/offline.html');
}
```

The previous code accomplishes the requested task efficiently. However, adding more conditions is required when adding new pages to the application. It is an inconvenient method to handle code; hence, instead of checking the URL, a check of the request headers is conducted, it checks whether the request header accepts HTML extensions. This method returns the offline fallback page for all uncached pages without specifying each page distinctly.

```
if (event.request.headers.get('accept').includes('text/html')) {
 return cache.match('/offline.html');
}
```

8.9.4 Cache cleanup

The dynamic cache fills up fast; thus, cleaning the dynamic cache storage is required in several cases. Cache cleanup is implemented in the service worker by adding a new trimCache() function with two arguments passed into it; the cacheName argument and the maxItems argument.

```
function trimCache(cacheName, maxItems){ }
```

The trimCache() function is implemented as follows.

1. The function calls cache.open(cacheName) with the cacheName argument.
   ```
   cache.open(cacheName)
   ```
2. It chains a promise with a function that gives access to that cache.
   ```
   .then(function(cache){ });
   ```
3. The function returns cache.keys() to access the stored requests.

```
return cache.keys()
```

4. It chains a second promise to remove the first and oldest item (keys[0]).

```
if(keys.length > maxItems){
 cache.delete(keys[0])
}
```

5. The trimCache() function is called recursivly, using a third promise.

```
.then(trimCache(cacheName, maxItems));
```

Eventually, the trimCache() stops re-calling itself until the keys length reaches the set maxItems property.

```
function trimCache(cacheName, maxItems){
 cache.open(cacheName)
 .then(function(cache){
  return cache.keys()
  .then(function(keys){
  if(keys.length > maxItems){
   cache.delete(keys[0])
   .then(trimCache(cacheName, maxItems));
   }
  });
 })
}
```

The trimCache() function is called when a cache clean up is needed, it is called in the fetch event listener to clean up the dynamic cache storage.

8.9.5 Getting rid of the service worker

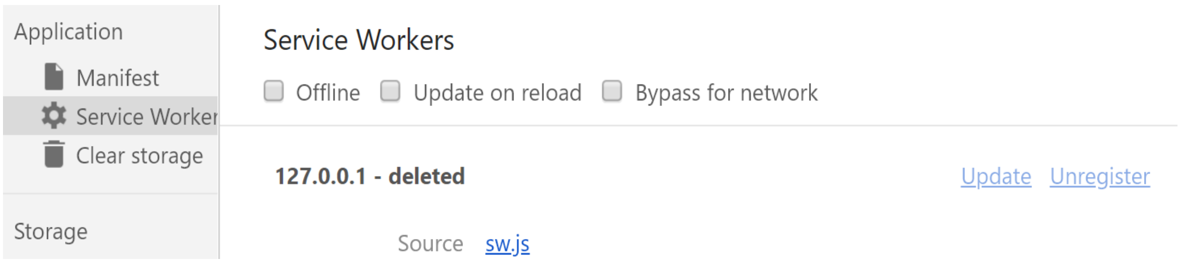
Removing the service worker at a given time interval might be required for advanced application functionalities; however, it is not implemented in this particular application. The process of removing a service worker is fairly simple and it is implemented as described below.

1. The application checks if a service worker exists in the navigator, then returns service workers registrations.

```
If(‘serviceWorker’ in navigator){
 Navigator.serviceWorker.getRegistrations()
}
```

2. It chains a promise that loops through the registered service workers and unregisters them one by one.

```
if('serviceWorker' in navigator){
 navigator.serviceWorker.getRegistrations()
 .then(function(registrations){
  for (var i=0 ; i<registrations.length; i++){
   registrations[i].unregister();
  }
 })
 }
}
```



127.0.0.1 - deleted

Source sw.js

Picture 43. Unregistering service worker.

In this chapter, different caching strategies were introduced, however; only one strategy was implemented as it was custom-made to fulfill the applications' requirements. With the right caching method implemented, this thesis continues to implement the backend database in the next chapter.

9 DYNAMIC DATA INTEGRATION

As mentioned earlier, this project implements Google Firebase real-time data source, it is managed by Google: https://firebase.google.com/. A database is implemented to store the created card assets (it stores the id, image, location, and title). The application fetches data from the database using HTTP requests and displays them in the application (Picture 44).

9.1 Connecting front end with the backend

The application fetches data using HTTP requests inside the feed.js file. It fetches posts and displays the posts one after another. Connecting the application to the database is implemented using the following steps.

1. The application fetches data from the following database URL: https://morad-pwa.firebaseio.com/posts

2. Firebase requires to add a .json extension to the URL since the database has a JSON structure.

   ```
   var url = 'https://morad-pwa.firebaseio.com/posts.json';
   ```

   The URL is updated in the service worker and in the feed.js file.

3. It creates cards based on the fetched data. A new fetch method is required to fetch all the cards available inside the database; thus, a new updateUI(data) function is created, it passes a data argument, loops through the data and calls the createCard(data[i]) function to use the received data.

   ```
   function updateUI(data){
    for (var i=0; i<data.length;i++){
     createCard(data[i]);
    }
   }
   ```

4. The createCard() function is updated to receive the data as an argument.

```
function createCard(data) { }
```

5. The application uses the data passed into the createCard() function to create cards.

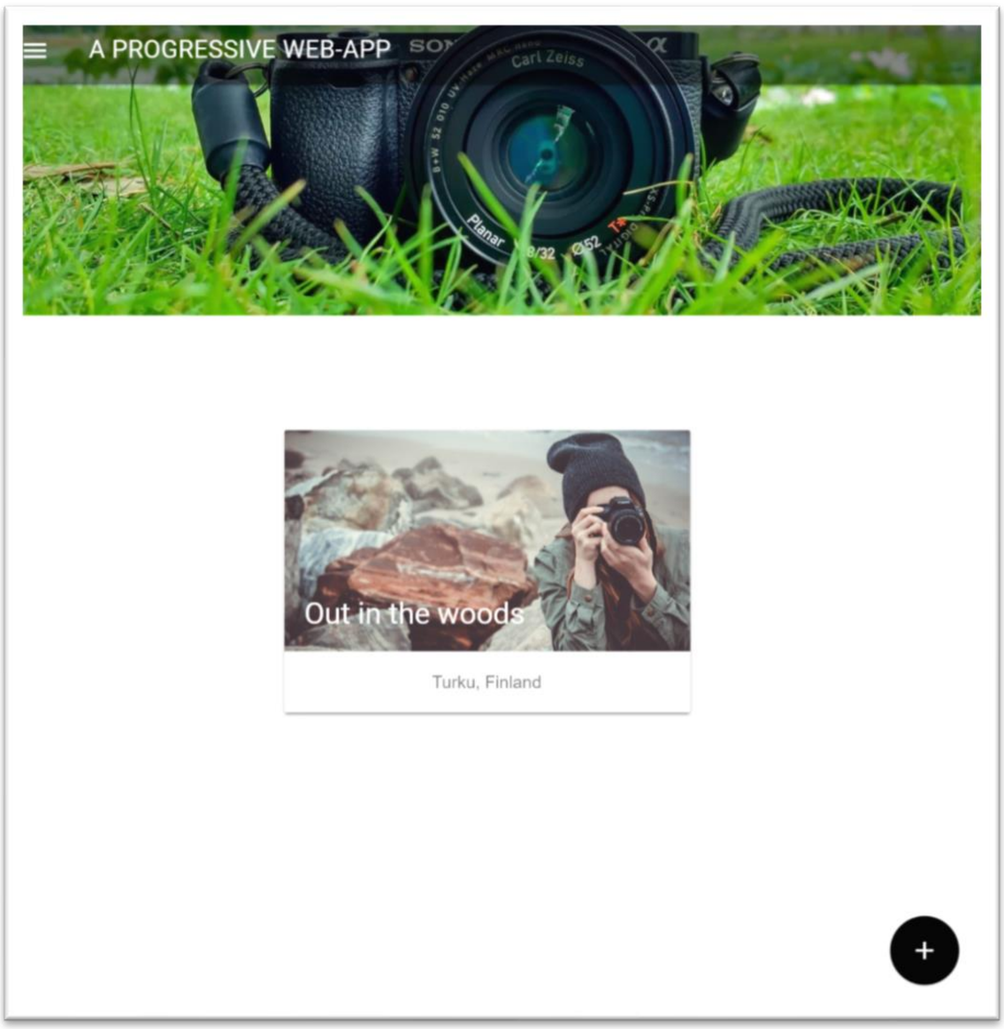
```
cardTitle.style.backgroundImage = 'url('+ data.image +')';
cardTitleTextElement.textContent = data.title;
cardSupportingText.textContent = data.location;
```

6. The updateUI() function is called after converting the data into an array form.

```
var dataArray = [];
for (var key in data){
 dataArray.push(data[key]);
}
updateUI(dataArray)
```

7. The clearCards() function is used inside the updateUI().

```
function updateUI(data){
 clearCards();
 for (var i=0; i<data.length;i++){
  createCard(data[i]);
 }
}
```



Picture 44. Displaying data from the database.

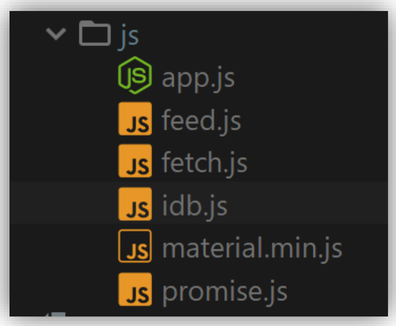
9.2 IndexedDB implementation

9.2.1 Adding The IDB file

The IDB package is downloaded from https://github.com/jakearchibald/idb.

This package wraps the indexedDB and allows the use of promises. The package is imported into the project and it is added to the js folder under the name idb.js. Further, the IDB file is imported to the index.html using a script tag.

```
<script src="src/js/idb.js"></script>
```



Picture 45. Adding idb.js inside the js folder.

9.2.2 Storing fetched posts in the indexedDB

The IDB package gets imported to the service worker using importScripts().

```
importScripts('src/js/idb.js');
```

The IDB package is added to the static file cache for providing offline access. Next, the applications' service worker implements the indexedDB in the process of fetching the posts URL. The code is modified to use indexedDB instead of the dynamic cache. The current code saves the posts URL fetch inside the dynamic cache storage.

```
var url = 'https://morad-pwa.firebaseio.com/posts';
if (event.request.url.indexOf(url) > -1) {
 event.respondWith(
  caches.open(CACHE_DYNAMIC_NAME)
  .then(function (cache) {
   return fetch(event.request)
   .then(function (res) {
// trimCache(CACHE_DYNAMIC_NAME, 3);
    cache.put(event.request, res.clone());
    return res;
    });
   })
 );
}
```

The code is modified to store the posts URL fetch inside indexedDB. It is implemented to open an indexedDB database and create an object storage. The database is stored through a promise using a variable holder.

```
var dbPromise = idb.open('posts-store', 1, function(db){
 db.createObjectStore('posts',{KeyPath: 'id'});
});
```

The idb.open() method opens a new database, it passes three arguments.

1. The name of the database (posts-store).
2. The database version, it is updated on the update of the database.
3. A function that executes after creating the database. It returns an object that accesses the database; inside the function, an object store is created and passes two arguments:

   3.1 The name of the store (posts).

   3.2 A Definition of an object holding a primary-key for each object in the object store, it uses a key property; {KeyPath: 'id'} that defines the name of the key (id). This gives the opportunity to retrieve an object by its id.

The application uses the promise to access the database; however, it creates the object store repeatedly when adding every object. The code is wrapped inside an if statement to check if the object storage alrady exists.

```
if(!db.objectStoreNames.contains('posts')) {
 db.createObjectStore('posts', {KeyPath: 'id'});
}
```

The function that creates the object store is implemented in the code inside the daynamic data fetch-event to use the indexedDB instead of the dynamic cache storage created in the early stages.

```
var url = 'https://morad-pwa.firebaseio.com/posts';
if (event.request.url.indexOf(url) > -1) {
 event.respondWith(fetch(event.request)
 .then(function (res) {
  return res;
  })
 );
}
```

After fetching the response, it gets stored in the indexedDB using a clone copy of the returned response.

```
Var cloneRes = res.clone();
```

Further, the clone response is called using JSON to return a promise.

```
var cloneRes = res.clone();
cloneRes.json()
.then(function(data){
});
```

This returns an untransformed body data, it is transformed using the data keys. This implements the dbPromise() after receiving access to the opened database.

```
for (var key in data) {
 dbPromise
 .then(function(db){ });
}
```

Afterwards, it uses the DB object to write data to the database by using the transaction method. It creates a variable that passes two arguments.

1. The targeted store (posts object store).

2. Type of transaction (Read and write).

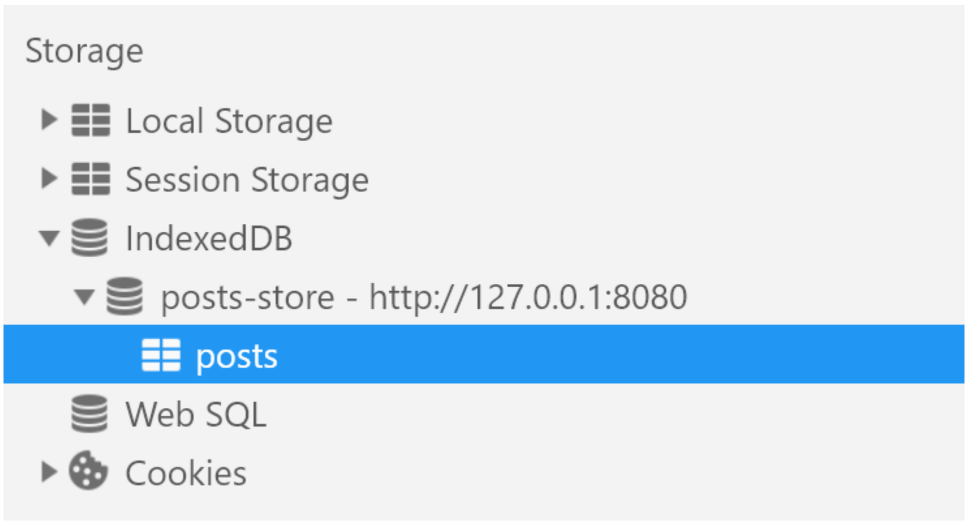
```
var tx = db.transaction('posts','readWrite');
```

Further, it opens a store using the objectStore() method. The indexedDB requires opening a store, then explicitly re-opening the same store.

```
var store = tx.objectStore('posts');
```

Consequently, it provides access to the opened store and uses it to put and store the transformed key-type data keys. Finally, it returns the completed transaction (Picture 46).

```
store.put(data[key]);
return tx.complete;
```



Picture 46. Storing the posts URL fetch inside indexedDB

9.2.3 Using indexedDB in the service worker

So far, the application puts data from the indexedDB into the post-storage. The application is further enhanced to fetch data from the indexedDB. Fetching is processed inside the feed.js file; however, it requires setting up the database inside the feed.js file additionally.

Duplicated code is implemented in several JavaScript files; hence, creating a file to store the shared JavaScript codes is convenient. Utility.js file is created for this purpose and it is imported inside the JavaScript files.

```
importScripts('/src/js/utility.js');
```

Inside utitlity.js, the dbPromis code is added. It opens the database and creates a new object store.

```
var dbPromise = idb.open('posts-store', 1, function (db) {
 if (!db.objectStoreNames.contains('posts')) {
  db.createObjectStore('posts', {keyPath: 'id'});
 }
});
```

After the writeData() method is created, it encapsulates the fetch listener which accesses the promise and writes into it. The writeData() method passes two data arguments.

1. The st argument, it provides information about the store.
2. Data argument, it provides information regarding the data.

```
function writeData(st,data){
 return dbPromise
 .then(function(db) {
  var tx = db.transaction(st, 'readwrite');
  var store = tx.objectStore(st);
  store.put(data);
  return tx.complete;
 });
}
```

Finally, the service worker calls writeData() to access the dbPromise in the fetch listener.

```
writeData('posts', data[key]);
```

9.2.4 Reading data from IDB

The readAllData(st) function is created in the utility.js file to read the data from the requested store and it passes the store argument.

```
function readAllData(st){
 return dbPromise
}
```

After, a promise function is chained to provide access to the database.

```
.then(function(db){}
```

1. It create a read only transaction.

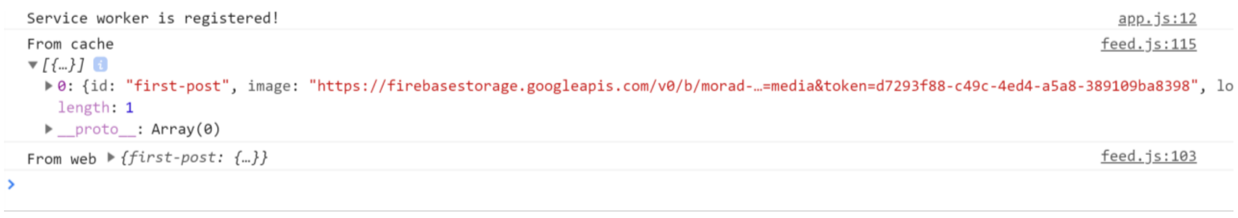
   ```
   var tx = db.transaction(st, 'readOnly');
   ```
2. Then opens an object store.

   ```
   var store = tx.objectStore(st);
   ```
3. Finally, returns the store.

   ```
   return store.getAll();
   ```

```
function readAllData(st){
 return dbPromise
 .then(function(db){
  var tx = db.transaction(st, 'readonly');
  var store = tx.objectStore(st);
  return store.getAll();
 });
}
```

Further, the readAllData() function is implemented in the feed.js file after checking indexedDB browser compatibility to ensure that the function is implemented properly. The readAllData() function returns stored data in an array form; thus, the updateUI() function is called, and it expects data in an array form.

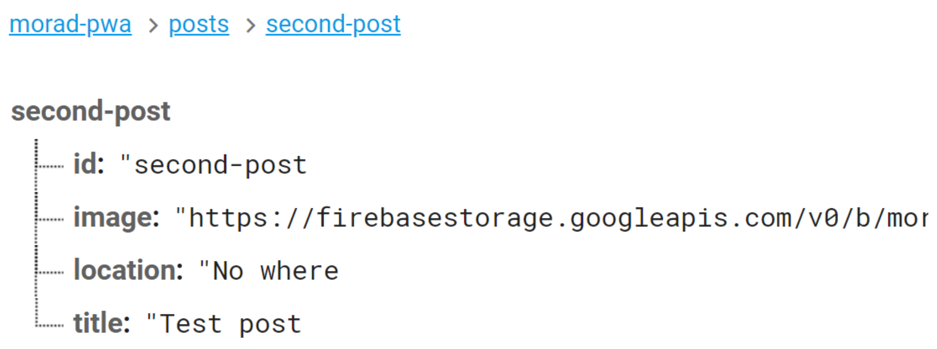


```
Service worker is registered!                                                                                  app.js:12
From cache                                                                                                   feed.js:115
▼[{…}]
  ▶0: {id: "first-post", image: "https://firebasestorage.googleapis.com/v0/b/morad-…=media&token=d7293f88-c49c-4ed4-a5a8-389109ba8398", lo
    length: 1
  ▶__proto__: Array(0)
From web ▶{first-post: {…}}                                                                                  feed.js:103
>
```

Picture 47. Displaying data stored in the indexedDB.

9.2.5 Handling server client mismatch

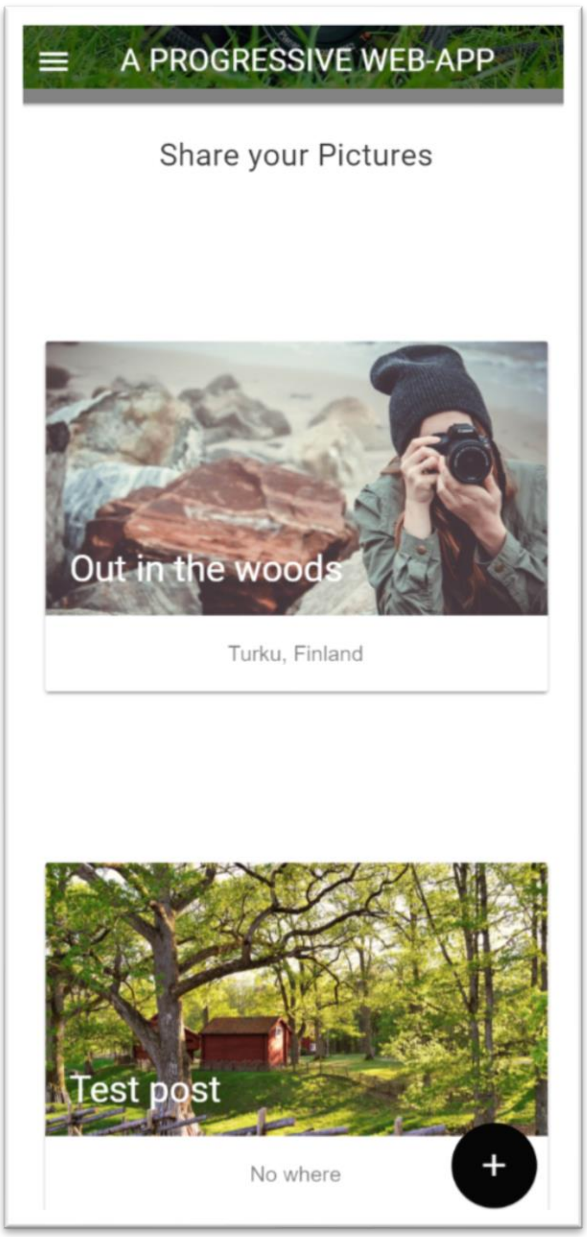
Adding a second post to the database (Picture 48) instantly loads a second card in the front end of the web page holding the second-post data from the database (Picture 49).



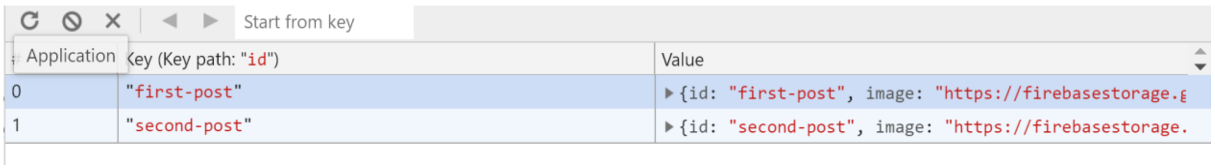
```
morad-pwa > posts > second-post

second-post
  id: "second-post
  image: "https://firebasestorage.googleapis.com/v0/b/mor
  location: "No where
  title: "Test post
```

Picture 48. Adding second-post to the database.



Picture 49. Displaying second-post in the application.



#	Key (Key path: "id")	Value
0	"first-post"	▸{id: "first-post", image: "https://firebasestorage.g
1	"second-post"	▸{id: "second-post", image: "https://firebasestorage.

Picture 50. Displaying second-post in the cache storage.

However, deleting the second post fails to remove it from the cache; therefore, the second post is still active in the offline mode. It is fetched from the indexedDB cache storage, since the put method overwrites the old value, but keeps it in the cache. A simple method to approach this problem is clearing the data using clearAllData() function before writing any data in the service worker.

```
clearAllData('posts')
.then(function () {
 return clonedRes.json();
})
.then(function (data) {
 for (var key in data) {
  writeData('posts', data[key])
 }
});
```

9.2.6 Clearing the database

Previously, the clearAllData() function has been used in the service workers but has not been defined yet. The clearAllData() function is defined in the utility.js file using the folowing steps.

1. The function passes the storage as an argument.

   ```
   function clearAllData(st){  }
   ```

2. Then, it implements a function promise to aquire access to the database returned by the database promise.

   ```
   function clearAllData(st){
    return dbPromise
    .then(function(db){  })
   }
   ```

3. It creates a read and write transaction.

   ```
   var tx = db.transaction(st, 'readWrite');
   ```

4. Then, it opens an object store.

   ```
   var store = tx.objectStore(st);
   ```

5. Afterwards, it removes all elements from the store.

   ```
   store.clear();
   ```

6. Finally, it returns a complete transaction.

   ```
   return tx.complete;
   ```

```
function clearAllData(st){
 return dbPromise
 .then(function(db){
  var tx = db.transaction(st, 'readWrite');
  var store = tx.objectStore(st);
  store.clear();
  return tx.complete;
 })
}
```

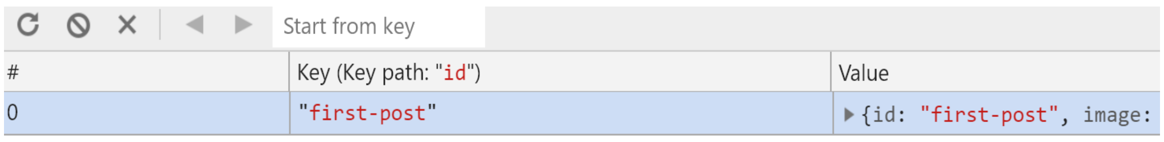
The service worker uses the clearAlldData() function right after cloning the response and before extracting any data.

```
clearAllData('posts');
.then(function(){
});
```

Moreover, it proceeds to extract data. It returns the clone response using a json() method and chains a promise to proceed extracting data.

```
if (event.request.url.indexOf(url) > -1) {
 event.respondWith(fetch(event.request)
 .then(function (res) {
  var clonedRes = res.clone();
  clearAllData('posts');
  .then(function(){
   return clonedRes.json();
```

```
  })
  .then(function(data) {
  for (var key in data) {
   writeData('posts', data[key]);
  }
 });
 return res; }
```



Start from key

#	Key (Key path: "id")	Value
0	"first-post"	▶{id: "first-post", image:

Picture 51. Clearing second-post from the database.

Further, a deleteItemFromData() function is created. It deletes single files from the database, it is similar to the previous clearAllData() function. However, a second id argument is passed to specify the deleted item.

```
function deleteItemFromData(st, id) {
 return dbPromise
 .then(function (db) {
  var tx = db.transaction(st, 'readWrite');
  var store = tx.objectStore(st);
  store.delete(id);
  return tx.complete
 })
 .then(function(){
 console.log('Item deleted');
 });
}
```

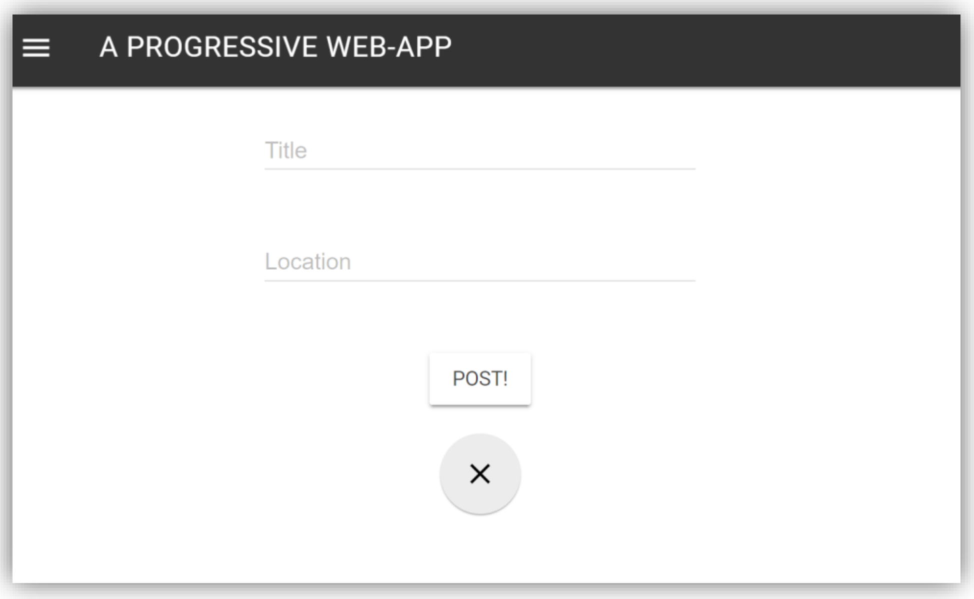
In this chapter, the service worker implemented dynamic content caching using IndexedDB, it accessed the IndexedDB using the IDB wrapper that allows using promises. Further, the service worker worked alongside the IndexedDB to provide offline functionality for dynamic content. However, sending requests while offline is still unavailable and will be implemented in the next chapters. This chapter provided a huge improvement to the application. The application was enhanced to provide fresh updated data to be displayed dynamically in the application.

10 BACKGROUND SYNC IMPLEMENTATION

The PWA implements background synchronization to synchronize the sent data even throughout offline access. In this chapter, the application is enhanced to store requests sent during offline access using background synchronization.

10.1 Basic application setup

The applications post button is not assigned a functionality yet (Picture 52), it reloads the web page on a click event since it is a default behavior.



Picture 52. Application post button.

The application connects the post button through a javascript code in the feed.js file since it holds the feed pages related code. After, it listens to the button-submit request as follows.

1. The code accesses the form using a query selector:

```
var form = document.querySelector('form');
```

2. The code registers a submit listener as shown in the following steps.

2.1 It reaches out to the form and adds an event listener to the submit button.

```
form.addEventListener('submit')
```

2.2 Then, executes a function to access the event.

```
form.addEventListener('submit', function(event){ });
```

2.2 The function prevents the default behavior of reloading the page.

```
event.preventDefault();
```

2.3 Then, it checks if data is submitted (the title and location data). Further, the elements are accessed using query selectors and ignore cases where data is missing.

```
var titleInput = document.querySelector('#title');
var locationInput = document.querySelector('#location');
if (titleInput.value.trim() === '' ||
locationInput.value.trim() === ''){
 alert('Please enter valid data!')
 return;
}
```

3. The application continues by implementing a closeCreatePostModel() function. It hides the form and redirects the application to the home page.

```
function closeCreatePostModal() {
 createPostArea.style.transform = 'translateY(100vh)';
 imagePickerArea.style.display = 'none';
 videoPlayer.style.display = 'none';
 canvasElement.style.display = 'none';
 // createPostArea.style.display = 'none';
}
```

4. Afterwards, the application is ready to register the background synchronization; however, it checks browser compatibility for both the service worker and the sync manager.

```
if('serviceWorker' in navigator && 'SyncManager' in window){ }
```

5. After, the background sync is implemented by reaching out to the service worker and calling the ready property.

   ```
   navigator.serviceWorker.ready
   ```

6. Then, a promise function is chained to access the services worker, this permits interactions with the service worker.

   ```
   .then(function(sw){ });
   ```

Moreover, the code is implemented in the feed.js file, since it provides access to the submit event listener. The function registers a sync event through the service workers; further, it registers an asynchronous task inside the service worker using the register() method. The method passes the id of the task as an argument which will later be used to react to re-established connectivity.

```
sw.sync.register('sync-new-post');
```

7. The data is sent through the submit request; hence, a new object variable is created to hold the title, location and a unique object id.

   ```
   var post = {
    id : new Date().toISOString(),
    title: titleInput.value,
    location: locationInput.value
   };
   ```

8. The code implements the writeData() function to save the posts into an object store. It creates a new object store inside the utility.js to serve syncronization tasks. The object store is passed to the dbPromise variable.

   ```
   var dbPromise = idb.open('posts-store', 1, function (db) {
    if (!db.objectStoreNames.contains('sync-posts')) {
     db.createObjectStore('sync-posts', {keyPath: 'id'});
    }
   }
   ```

   Additionally, the sync-posts storage is passed to the writeData() function.

   ```
   writeData('sync-posts', post);
   ```

9. The writeData() returns a promise with a function to store data and to register the sync task.

```
writeData('sync-posts', post)
.then(function() {
 return sw.sync.register('sync-new-posts');
})
```

10. Further, a second promise is chained to access an HTML class to output a confirmation message.

```
.then(function(){
 var snackbarContainer = document.querySelector('#confirmation-
toast');
 var data = {message: 'Your post was saved for syncing!'};
 snackbarContainer.MaterialSnackbar.showSnackbar(data);
});
```

11. Finally, the function catches potential errors.

```
.cache(function(err){
 console.log(err);
});
```

10.2 Adding a fallback

The previous code targets the browsers that offer sync manager and service worker support; however, some browsers fail to offer such support. The application handles such cases as follows.

1. An else statament is added to handle such cases, it provides a new sendData() function that directly sends data without the sync event.

```
if ('serviceWorker' in navigator && 'SyncManager' in window) { }
else {
 sendData();
}
```

2. The function fetches the data from the backend URL. The fetch event passes two arguments.

2.1 The first argument specifies the URL of the backend database.

2.2 The second argument specifies a POST method, the headers, and the JavaScript body.

```
function sendData(){
 fetch('https://morad-pwa.firebaseio.com/posts.json' , {
  method:'POST',
  headers: {
   'Content-Type': 'application/json',
   'Accept': 'application/json'  },
  body: JSON.stringify ( {
  id: new Date().toISOString(),
  title: titleInput.value,
  location: locationInput.value,
  image: '"https://firebasestorage.googleapis.com/v0/b/morad-
pwa.appspot.com/o/5.jpeg?alt=media&token=d7293f88-c49c-4ed4-
a5a8-389109ba8398"'
  })
 })
}
```

3. Afterwards, it reacts to the response to confirm functionality.

```
.then(function(res){
console.log('Sent data', res);
});
```

4. Finally, the updateUI() is called to update the page.

```
updateUI();
```

This method forces browsers without sufficient support to ignore the background synchronization; hence, it performs a normal fetch to receive data when the connection is available.

10.3 Synchronizing data in the service worker

To implement data synchronization in the service worker, the application registers a new event listener inside the service worker using the following steps.

1. It adds an event listener to listen to the sync event.

   ```
   self.addEventListener('sync', function(event){  });
   ```

   The event listener executes when the service worker detects a network connection.

2. The event sends a request to the server as follows.

   2.1 It checks if the event tags are sync-new-posts event tags.

   ```
   if (event.tag === 'sync-new-posts'){ }
   ```

   2.2 Then, it implements a waitUntil() method to ensure the data is sent properly.

   ```
   event.waitUntil();
   ```

   2.3 The readAllData() function is called inside the waitUntil() method, it uses the sync-posts stored data.

   ```
   readAllData('sync-posts')
   ```

   2.4 A promise function is chained with data passed as an argument.

   ```
   .then(function(data){   })
   ```

   2.5 Inside the function, the data is sent to the server; hence, the sendData() function is modified to loop through all the data, since multiple posts might be available in the store.

```
self.addEventListener('sync', function(){
 console.log('[Service worker] Background syncing', event);
 if (event.tag === 'sync-new-posts'){
  console.log('[Service Worker] Syncing new posts');
  event.waitUntil(
   readAllData('sync-posts')
   .then(function(data){
   for (var dt of data){
```

```
    fetch('https://morad-pwa.firebaseio.com/posts.json' , {
    method:'POST',
    headers: {
    'Content-Type': 'application/json',
    'Accept': 'application/json'
     },
    body: JSON.stringify({
    id: dt.id,
    title: dt.title,
    location: dt.location,
    image: '"https://firebasestorage.googleapis.com/v0/b/morad-
pwa.appspot.com/o/5.jpeg?alt=media&token=d7293f88-c49c-4ed4-a5a8-
389109ba8398"'
      })
     })
    .then(function(res){
    console.log('Sent data', res);
     });
    }
   })
  );
 }
});
```

2.6 However, it fails to use updateUI(), since the DOM restricts service worker access. Hence, the deleteItemFromData() method is created inside the feed.js file to remove processed data using the following steps.

2.6.1 It checks the state of the response using a helper property added to the response object.

```
if (res.ok){ }
```

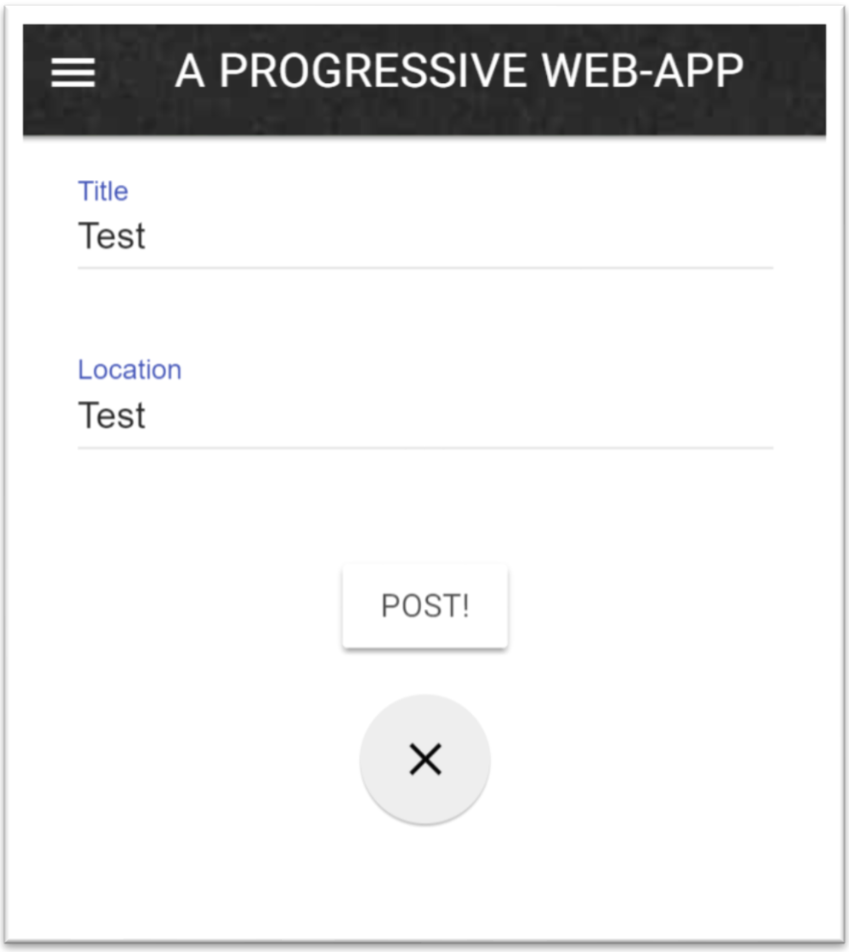
2.6.2 Then, it implements the deleteItemFromData() using the object id.

```
deleteItemFromData('sync-posts', dt.id);
```

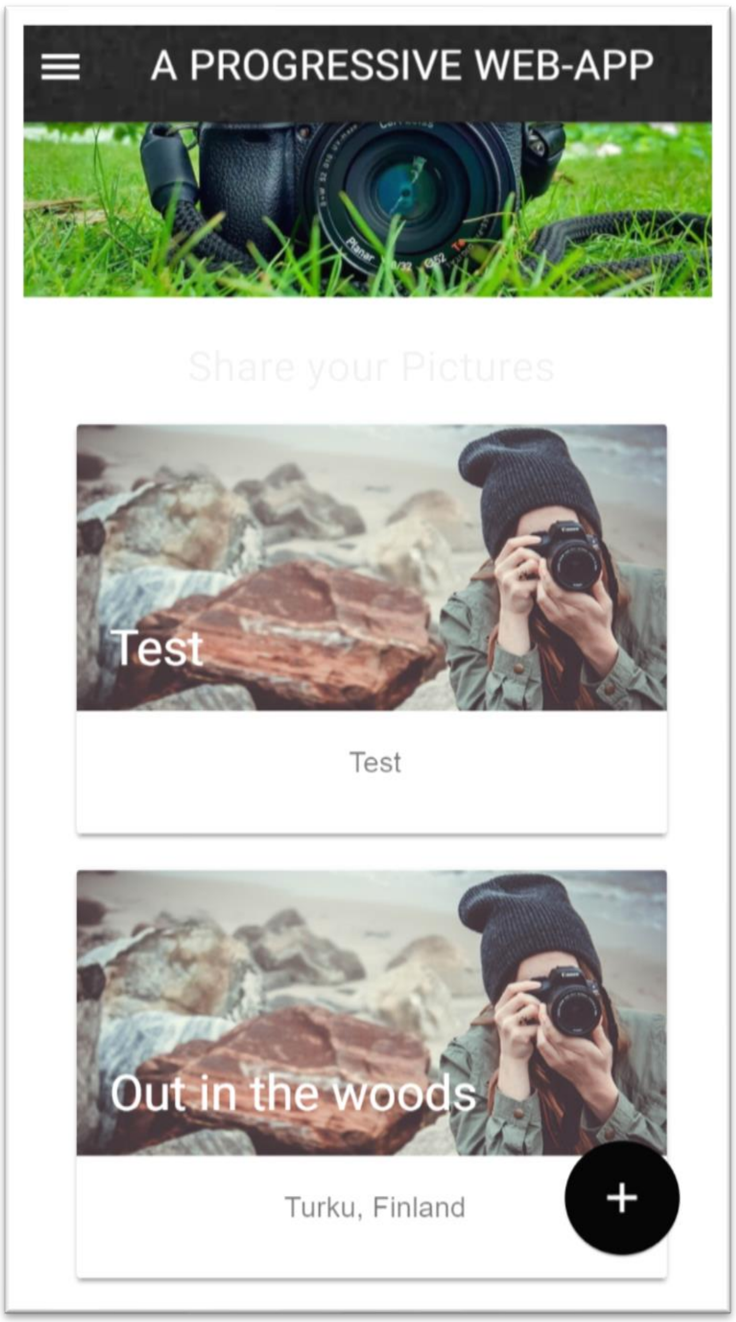
2.6.3 Finally, the function catches errors occurring while sending data.

```
.cache(function(err){
console.log('Error while sending data', err); });
```

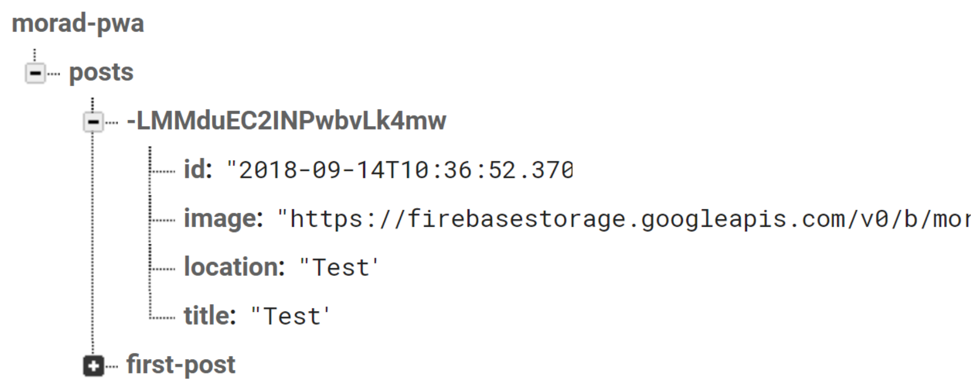
Eventually, the application is able to send multiple posts while offline, it uses background synchronization to save the data and sends it when internet connection re-establishes.



Picture 53. Posting a new post from the application.



Picture 54. Displaying the new post in the application.



```
morad-pwa
  posts
    -LMMduEC2INPwbvLk4mw
      id: "2018-09-14T10:36:52.370
      image: "https://firebasestorage.googleapis.com/v0/b/mor
      location: "Test'
      title: "Test'
    first-post
```

Picture 55. Displaying the new post in the database.

10.4 Server side code

So far, the application used the (http-server -c-1) to serve the Localhost:8080, since the PWA conducts in the browser. A server-side code is required for application production implementation. Firebase offers a cloud functions feature, it is used to synchronize code on demand, it reacts to certain event triggers.

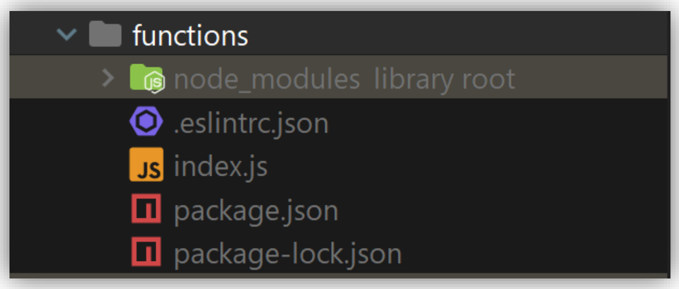
“Cloud Functions for Firebase lets you automatically run backend code in response to events triggered by Firebase features and HTTPS requests.”

(Google development 2017)

Further, Firebase is used as a backend server, the application implements Firebase and installs the Firebase package **(v5.0.1)** through NPM as follows.

- ➔ **NPM install -g firebase-tools**
- ➔ **firebase init**
- ➔ **choose functions and hosting features**
- ➔ **use our public directory**
- ➔ **do not configure as a single page app**
- ➔ **Do not overwrite the index.html page**
- ➔ **Finished**

This sets up files installed by firebase and a functions folder where the application implements new Firebase functions. It allows the application to create a rest API endpoint which runs on the Firebase project server.



Picture 56. Firebase functions folder.

The index.js file handles node.js application code. Afterwards, the application creates a rest API endpoint (storePostData).

```
exports.storePostData = functions.https.onRequest(function(request,
response)  { });
```

Firebase functions are implemented as follows.

1. Functions variable provide access to Firebase functions.

   ```
   var functions = require('firebase-functions');
   ```

2. HTTPS refers to an incoming HTTP request event.

3. The function executes when a request is triggered.

Further, additional Firebase packages are installed to store data.

1. Firebase-admin package **(v6.0.0)**.
2. CORS package **(v2.8.4)**.

Afterwards, the application imports the packages into the index.js file as follows.

1. `var admin = require('firebase-admin');`

   This provides access to the firebase database.

2. `var cors = require('cors')({origin:true});`

   This sends the corresponding headers, this allows access to the HTTP endpoint from an application running on a different server.

The function executes when a request reaches the end-point (storePostData end-point). The function executes as follows.

1. It uses CORS which returns a function with an access to the request and the response and wraps the executed code inside the function. Request and response are passed as arguments, this automatically sends the headers to the origin.

   ```
   cors(request,response,function(){  });
   ```

2. The function requests admin access, this provides access to push a new post object to the posts database.

   ```
   admin.database().ref('posts').push({ })
   ```

   The new object preserves 4 elements.

   a. Id.
   b. Title.
   c. Location.
   d. Image.

   ```
   id: request.body.id,
   title: request.body.title,
   location: request.body.location,
   image: request.body.image
   ```

3. Afterwards, it chains a function promise to execute when the process passes. It sends a new response holding the status code 201, this means that the code is executed successfully. Further, it forwards a message using JSON holding the response id.

   ```
   .then(function(){
    response.status(201).json({message: 'Data stored',
   id:request.body.id});
   })
   ```

4. Finally, the function catches errors and sends a response holding the status code 500 referring to malfunction.

   ```
   .cache(function(err){
   ```

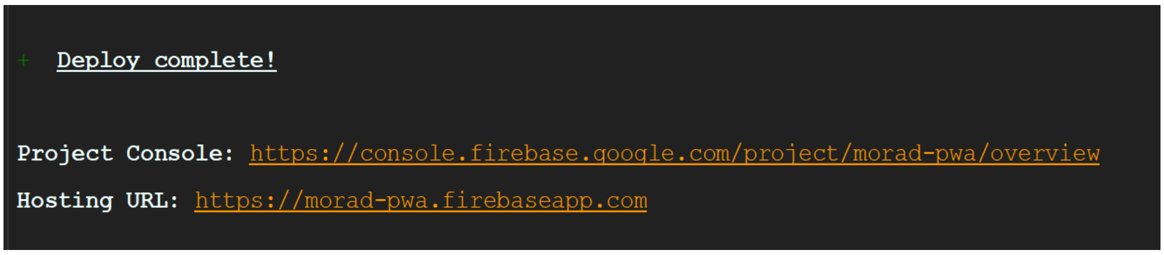
```
     response.status(500).json({error: err});
    });
```

The application is initialized to access the Firebase database with admin privileges; hence, it points to a Firebase service account authentication database file that provides admin privileges.

```
var serviceAccount = require("./morad-pwa-firebase-key.json");

admin.initializeApp({
  credential: admin.credential.cert(serviceAccount),
  databaseURL: 'https://morad-pwa.firebaseio.com/'
});
```

After the application constructs the API, it forwards data requests using the Firebase deploy method. This method deploys the front end application served by the public folder and the Firebase functions folder. The Firebase deployment returns the project URL and the hosting URL (Picture 57).

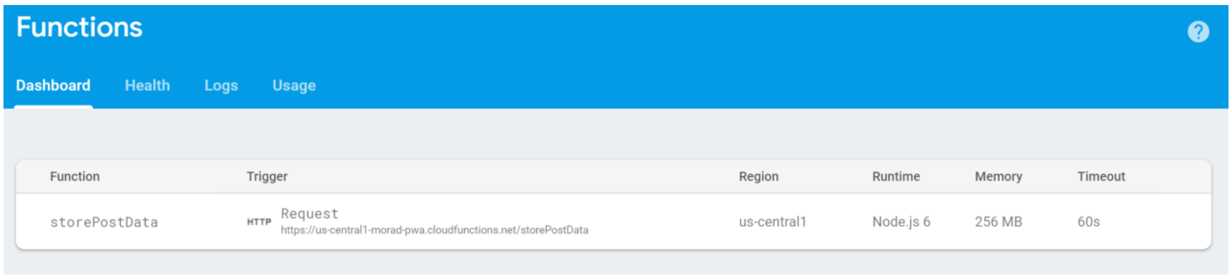


```
+  Deploy complete!

Project Console: https://console.firebase.google.com/project/morad-pwa/overview
Hosting URL: https://morad-pwa.firebaseapp.com
```

Picture 57. Firebase deployment.

Consequently, the storePostData is added to the application database cloud functions (Picture 58).

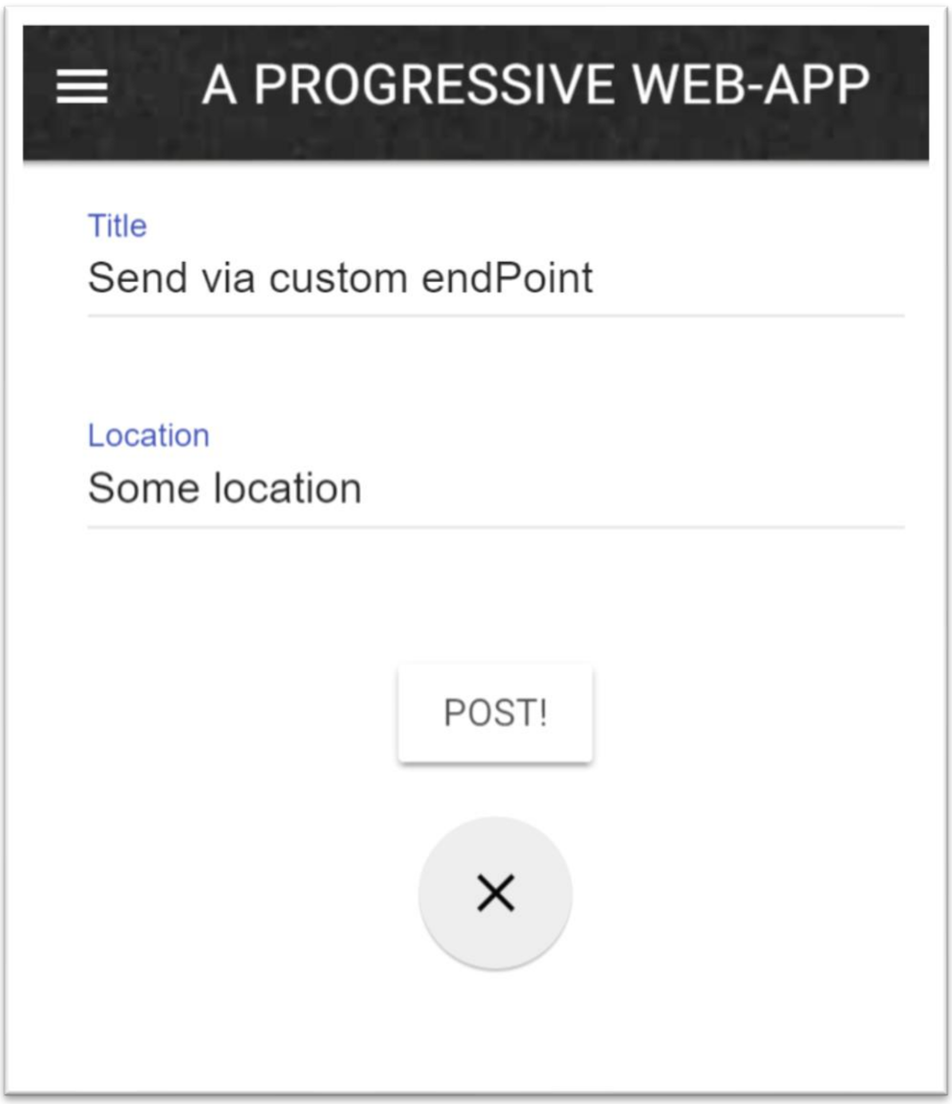


Function	Trigger	Region	Runtime	Memory	Timeout
storePostData	HTTP Request https://us-central1-morad-pwa.cloudfunctions.net/storePostData	us-central1	Node.js 6	256 MB	60s

Picture 58. StorePostData in the database.

The cloud functions storePostData endpoint URL is used to send data requests from this point further; hence, the new URL is implemented in the service worker synchronization event listener and in the fallback sendData() method inside feed.js.

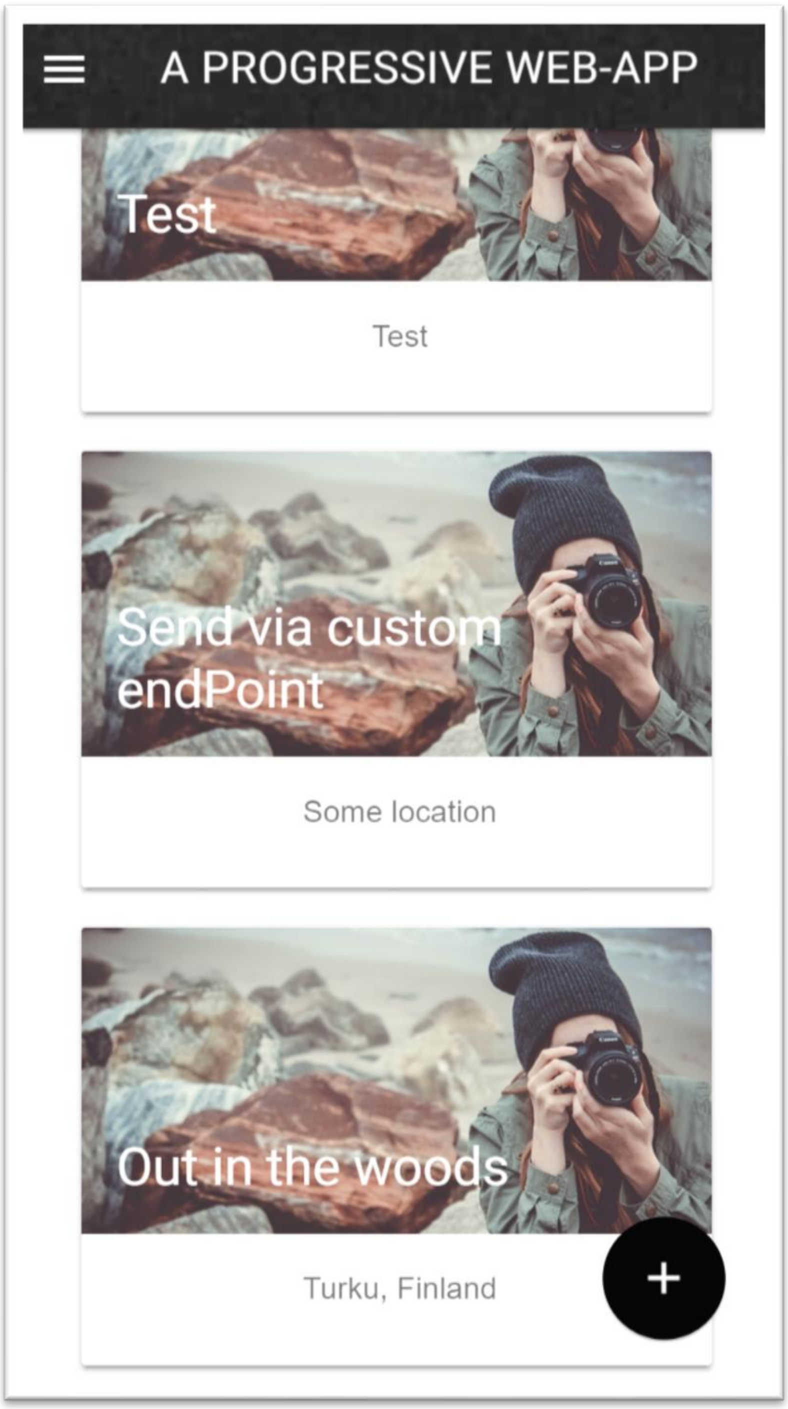
Source: https://us-central1-morad-pwa.cloudfunctions.net/storePostData



Title
Send via custom endPoint

Location
Some location

Picture 59. Sending data via custom end-point.



Picture 60. Displaying data sent via custom end-point.

In this chapter, the background synchronization was implemented in the application, it is a very important feature that enhanced the application to send data requests when offline. Now, the application can sync data when offline and send it when connectivity is re-established.

11 ADDING WEB PUSH NOTIFICATIONS

As previously explained, the application uses push notifications to re-engage the user with the application even when the application is closed. It enhances user real-time interactions and provides a native application experience.

11.1 Requesting permissions

The application requests the user permission to enable push notifications; hence, an “enable notifications” button is created to grant access to display notifications. The enable-notifications function takes an “enable-notifications” class and it is used to react to events in the feed.js file as described in the following steps.

1. It adds a new variable to access the enable notifications button.

   ```
   var enableNotificationsButtons =
   document.querySelectorAll('.enable-notifications');
   ```

2. The application displays the button when the browser supports is available. The code is modified as follows.

   2.1 In the app.css, the button is turned off by default.

   ```
   .enable-notifications{ display: none; }
   ```

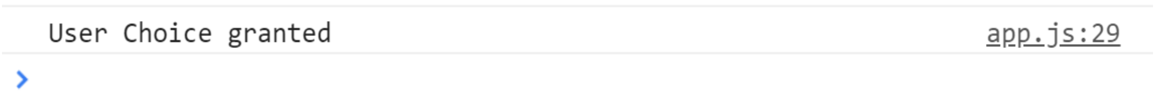
   2.2 In the app.js, the button is enabled when the browser support is available and implements a click event listener to enable notifications when clicked. The application loops through the enable notifications buttons inside the application (two buttons are used in the application).

   ```
   if ('Notification' in window){
    for(var i=0; i<enableNotificationsButtons.length ;i++){
     enableNotificationsButtons[i].style.display = 'inline-
   block';
     enableNotificationsButtons[i].addEventListener('click',
   askForNotificationPermission);
    }
   }
   ```

3. The askForNotfication() function is created.

```
function askForNotificationPermission(){
 Notification.requestPermission(function(result){
  console.log('User Choice', result);
  if(result !== 'granted'){
   console.log('No notification permission granted!');
   } else{   });
}
```

When the user accepts the push-notifications, the application implicitly asks for push permissions correspondingly regardless of standing as a separate technology.



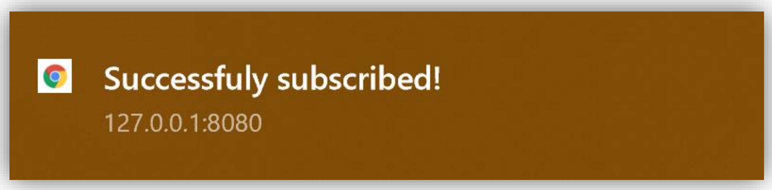
```
User Choice granted                                          app.js:29
```

Picture 61. Accepting Push notifications.

11.2 Displaying notifications.

1. The application creates a new function to confirm receiving the user permission. The function is executed inside the else{ } block in the previous askForNotificationPermission() function.

```
function displayConfirmNotification(){
 new Notification('Successfuly subscribed!');
}
```



Successfuly subscribed!
127.0.0.1:8080

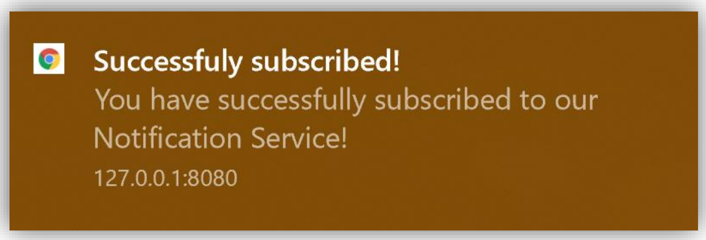
Picture 62. Subscription notification.

2. The function passes options using a JavaScript object. The options argument is passed as a new argument within the new Notification.

```
var options = {};
```

3. The options object gets populated.

```
var options = {
body: 'You have successfully subscribed to our Notification Service!'
};
```



Successfuly subscribed!
You have successfully subscribed to our Notification Service!
127.0.0.1:8080

Picture 63. Adding a message to the push notification.

11.3 Service worker Notifications

Notifications require displaying through the service workers when a connection is not available; hence, the application is modified to display notifications through the service worker when the service worker browser support is available. The application is modified as follows.

1. Inside the displayConfirmNotification() function, the application checks the service worker support within the used navigator.

```
if('serviceWorker' in navigator){   }
```

2. Then, it accesses the navigator and the service worker object. Further, it acquires access to the active service worker registration by calling the ready method.

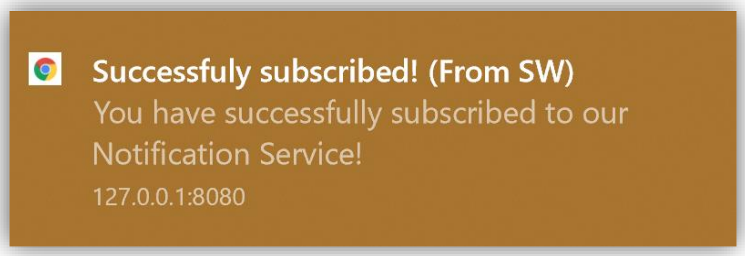
```
navigator.serviceWorker.ready
```

3. The function returns a promise to gain access to the current service worker registration.

```
.then(function(swreg){  });
```

4. After, it calls the showNotification() method and passes the title and the options object.

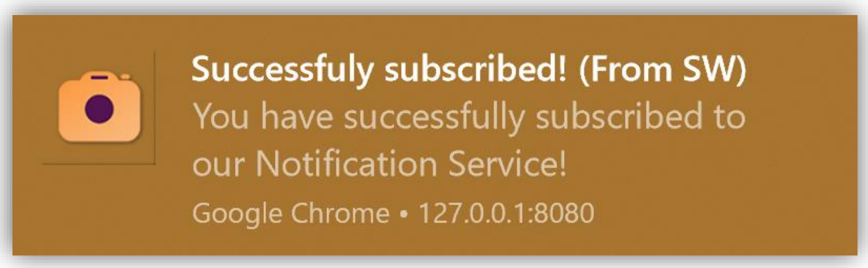
```
function displayConfirmNotification(){
 if('serviceWorker' in navigator){
 var options = {
  body: 'You have successfully subscribed to our Notification
Service!'
 };
 navigator.serviceWorker.ready
 .then(function(swreg){
 swreg.showNotification('Successfuly subscribed! (From SW)',
options);
 });
}}
```


Successfuly subscribed! (From SW)
You have successfully subscribed to our Notification Service!
127.0.0.1:8080

Picture 64. Service worker push notification.

Further options are added to the options object to enhance the push notifications.

1. An icon: `icon: '/src/images/icons/app-icon-96x96.png'`

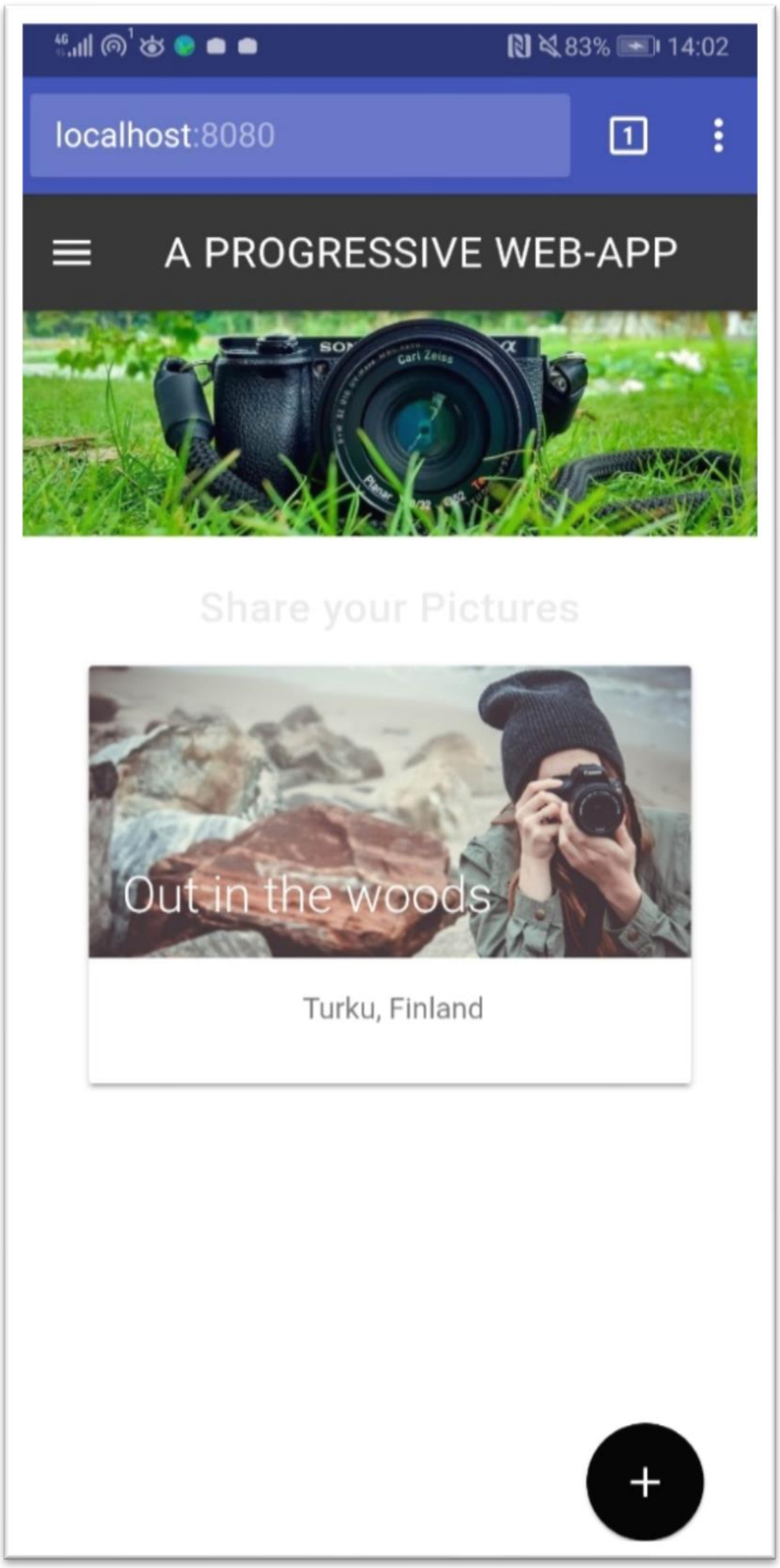


Successfuly subscribed! (From SW)
You have successfully subscribed to our Notification Service!
Google Chrome • 127.0.0.1:8080

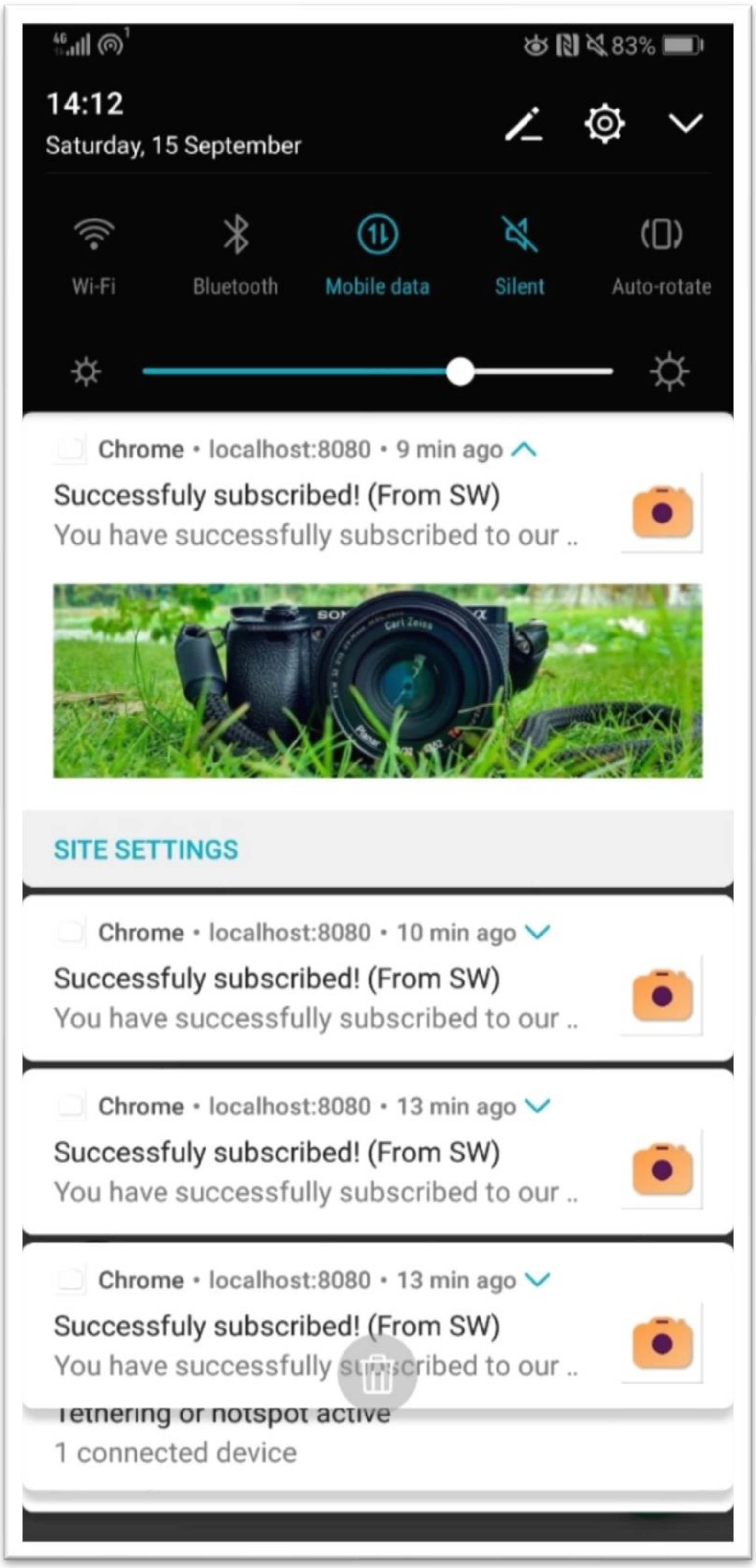
Picture 65. Adding icon to the push notification.

2. An Image: `image: '/src/images/3.jpg'`
3. Text direction: `dir: 'ltr'` (left to right text direction)
4. Language: `lang: 'en-US'`
5. Vibration pattern: `vibrate: [100, 50, 200]`
   Defines a vibration pattern to vibrates for 100 ms, it pauses for 50 ms and then vibrates again for 200 ms.
6. Badge option: adds the application icon to the top bar of the mobile display screen next to battery status, phone hostname, and wifi icon, where icons show in black and white.

   ```
   badge: '/src/images/icons/app-icon-96x96.png'
   ```



Picture 66. A PWA.


14:12
Saturday, 15 September

Chrome • localhost:8080 • 9 min ago
Successfuly subscribed! (From SW)
You have successfully subscribed to our ..

Chrome • localhost:8080 • 10 min ago
Successfuly subscribed! (From SW)
You have successfully subscribed to our ..

Chrome • localhost:8080 • 13 min ago
Successfuly subscribed! (From SW)
You have successfully subscribed to our ..

Chrome • localhost:8080 • 13 min ago
Successfuly subscribed! (From SW)
You have successfully subscribed to our ..

Tethering or hotspot active
1 connected device

Picture 67. Push notifications on an android.

In this chapter, the application was enhanced to use push notifications, it is a powerful tool used to re-engage the users back into the application. Now, the application can display notifications and send push notifications. A fully functional push service was implemented using the applications’ own server hosted by Googles’ Firebase. The application triggers push notification and sends it to the browser server; finally, delivers it to the users’ browser.

12 NATIVE DEVICE FEATURES INTEGRATION

In this chapter, the application is enhanced by accessing the device camera after requiring access permissions. Further, the application permits the user to take photos using the device camera and to upload them to the server.

12.1 Accessing the native device features

To access the native device features, the application provides the following additional tools to the projects post section inside the form.

1. video input.

   ```
   <video id="player" autoplay></video>
   ```

2. canvas.

   ```
   <canvas id="canvas" width="320px" height="240px"></canvas>
   ```

3. button element (used to capture the photo).

   ```
   <button class="mdl-button mdl-js-button mdl-button--raised
   mdl-button--colored" id="capture-btn">Capture </button>
   ```

4. A file picker used to upload a file from storage.

   ```
   <input type="file" accept="image/*" id="image-picker">
   ```

12.2 Acquiring DOM access

The application accesses the camera through the video and the canvas elements. It streams a live picture to the video element. Then uses the canvas to freeze the image on clicking the capture button.

The functionalities are implemented using JavaScript inside the feed.js file as follows.

1. It adds new variables to access the video player, canvas and button.

   ```
   var videoPlayer = document.querySelector('#player');
   var canvasElement = document.querySelector('#canvas');
   ```

```
var captureButton = document.querySelector('#capture-btn');
```

2. Then it provides access to the image picker and the image picker area.

```
var imagePicker = document.querySelector('#image-picker');
var imagePickerArea = document.querySelector('#pick-image');
```

3. Finally, it initializes either the camera or the file picker, depending on the device support. The application implements a new initilazeMedia() function.

```
function initializeMedia() { }
```

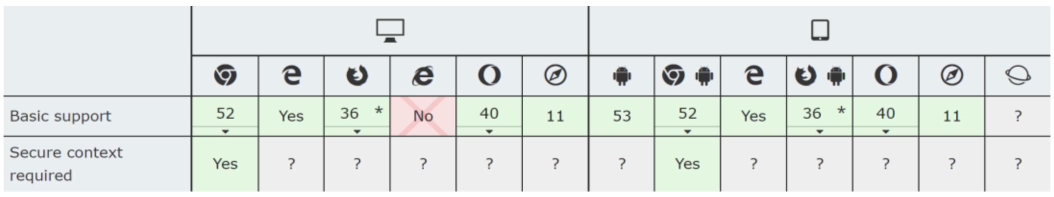
12.3 Polyfills and browser support

For a better progressive user experience, the application provides access to the camera on as many devices as possible. The application is modified as follows.

1. Inside the initializeMedia() function, the application checks for media devices in the navigator .

```
if (!('mediaDevices' in navigator)) { }
```

Picture 68 specifies the media device browser support.



	🖥						📱						
Basic support	52	Yes	36 *	No	40	11	53	52	Yes	36 *	40	11	?
Secure context required	Yes	?	?	?	?	?	?	Yes	?	?	?	?	?

Picture 68. Media device browser support.

Source: https://developer.mozilla.org/enUS/docs/Web/API/MediaDevices/getUserMedia

2. If the media devices are not available, the application creates polyfills to extend the device support.

```
if (!('mediaDevices' in navigator)) {
 navigator.mediaDevices = {};
}
if (!('getUserMedia' in navigator.mediaDevices)) {
 navigator.mediaDevices.getUserMedia = function(constraints) {
  var getUserMedia = navigator.webkitGetUserMedia ||
navigator.mozGetUserMedia;
if (!getUserMedia) {
 return Promise.reject(new Error('getUserMedia is not
implemented!'));
}
return new Promise(function(resolve, reject) {
getUserMedia.call(navigator, constraints, resolve, reject);
});
```

The previous function checks Firefox and Safari support; if no support is available it throws an error.

12.4 Acquiring video image

The application acquires video image as specified in the following steps:

1. After proceeding with a browser that supports media access, the application uses the getUserMedia() method with a video argument passed.

```
navigator.mediaDevices.getUserMedia({video: true})
```

2. Then itchains a promise function to return a stream object.

```
.then(function(stream) {
videoPlayer.srcObject = stream;
videoPlayer.style.display = 'block';
})
```

3. Further, the function catches errors and provides access to a file picker if camera access is denied.

```
.catch(function(err) {
imagePickerArea.style.display = 'block';
});
```

4. Then, it activates the capture button.

```
captureButton.addEventListener('click', function(event) {
canvasElement.style.display = 'block';
```
→ set the canvas.

```
videoPlayer.style.display = 'none'; → disables the video stream .

captureButton.style.display = 'none'; → disables the button.

var context = canvasElement.getContext('2d'); → define image canvas.

context.drawImage(videoPlayer, 0, 0, canvas.width,
videoPlayer.videoHeight / (videoPlayer.videoWidth /
canvas.width));
videoPlayer.srcObject.getVideoTracks().forEach(function(track) {
track.stop(); → stops the camera stream.

 });
});
```

12.5 Storing the image on a server

So far, the captured image lives in the canvas. The canvas returns a URL that is saved in a file to be uploaded to the database. Thus, the dataURLtoBlob () function is used to convert the given base 64 bit URL into a file.

```
function dataURItoBlob(dataURL) {
 var byteString = atob(dataURI.split(',')[1]);
 var mimeString = dataURI.split(',')[0].split(':')[1].split(';')[0]
 var ab = new ArrayBuffer(byteString.length);
 var ia = new Uint8Array(ab);
 for (var i = 0; i < byteString.length; i++) {
  ia[i] = byteString.charCodeAt(i);
}
 var blob = new Blob([ab], {type: mimeString});
 return blob;
}
```

Further, the following code is implemented in the service worker and the feed.js file.

1. It adds a new variable.

   ```
   var picture;
   ```

2. Inside the capture button function after stopping the camera stream, it converts the picture URL into a file using the dataURItoBlob() function.

   ```
   picture=dataURItoBlob(canvasElement.toDataURL());
   ```

3. The image file is uploaded to the database servers’ endpoint as follows.

   3.1 Inside the service worker, the applications’ header is set to accept JSON data.

   ```
   'Content-Type': 'application/json',
   'Accept': 'application/json’
   ```

   3.2 Then, it adds a variable to accept the formData, this allows sending the form data to the backend through an AJAX request.

   ```
   var postData = new FormData();
   ```

   3.3 Then, it appends the data id, title and location to each file.

   ```
   postData.append('id', dt.id);
   postData.append('title', dt.title);
   postData.append('location', dt.location);
   ```

   3.4 After, it passes the file with the picture.

   ```
   postData.append('file', dt.picture, dt.id + '.png');
   ```

   3.5 Finally, it enforces the body to pass postData instead of JSON.

   ```
   data.body: postData
   ```

12.6 Accepting file upload with Firebase

The cloud backend code is enhanced using the following steps to accept the file upload using Firebase and to handle posting the picture file.

1. The application installs the Formidable package **(v1.2.1)**, it is used to extract foreign data and import it into the index.js file inside the Firebase functions folder.

   ```
   var formidable = require('formidable');
   ```

2. The application installs the Google cloud helper package **(v0.58.2)** to access the Firebase cloud storage.

   ```
   var gcconfig = {
    projectId: 'pwagram-99adf',
    keyFilename: 'pwagram-fb-key.json'
   };
   var gcs = require('@google-cloud/storage')(gcconfig);
   ```

3. Next, inside the storePostData() function, the application extracts the incoming form data.

   ```
   var formData = new formidable.IncomingForm();
   ```

4. The request is parsed to access the database.

   ```
   formData.parse(request, function(err, fields, files) { }
   ```

5. The application reaches out to the database using a callback as follows.

   5.1 First, it moves the incoming data from the temporary storage to a second temporary storage using a new node package - fs package **(v1.0.0)**.

   ```
   var fs = require('fs');
   fs.rename(files.file.path, '/tmp/' + files.file.name);
   ```

   5.2 Then saves the data permanently into a cloud storage bucket.

   ```
   var bucket = gcs.bucket('morad-pwa.appspot.com');
   ```

   5.3 Finally, initializes the cloud bucket.

   ```
   bucket.upload('/tmp/' + files.file.name, {
   uploadType: 'media',
   metadata: {
   ```

```
        metadata: {
        contentType: files.file.type,
        firebaseStorageDownloadTokens: uuid → unique id storage
          }
         }
        }
```

During the initialization, the application installs a new UUID (Unique User ID) package **(v3.3.2)** that generates a new unique id string.

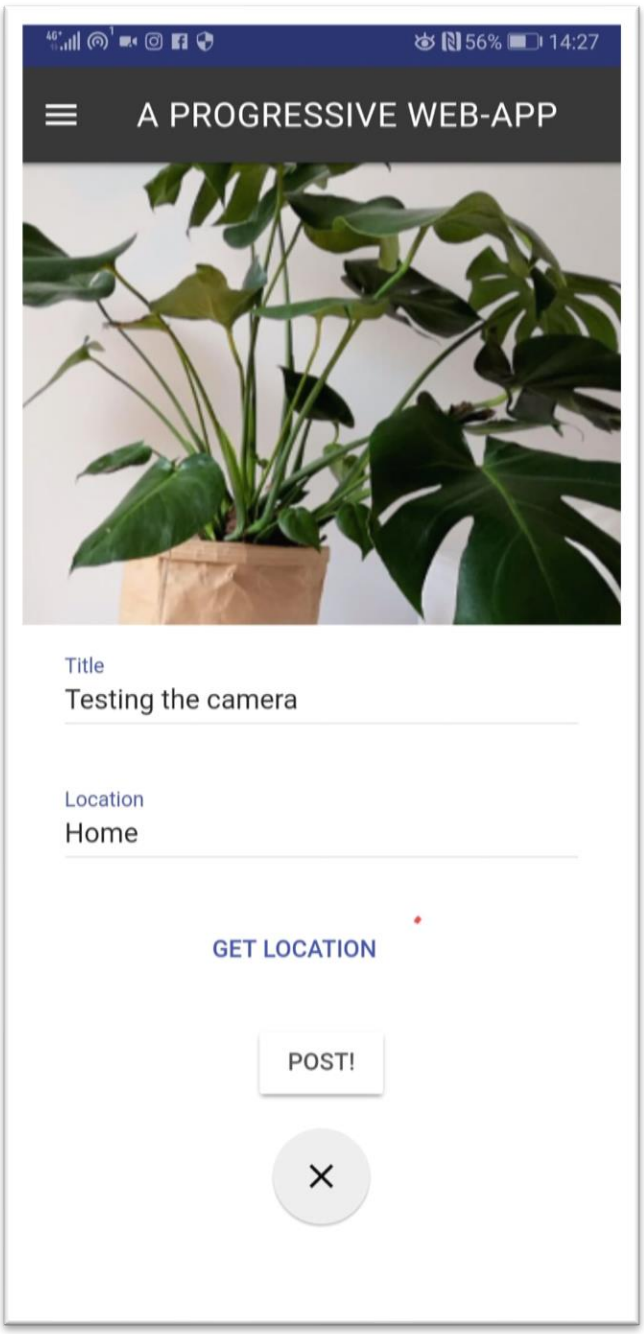
```
npm install --save uuid-v4
```

Afterwards, the application Imports it to the index.js file inside the Firebase functions folder and initializes it in the storePostData; hence, the Firebase generates a download URL automatically.

```
var UUID = require('uuid-v4');
var uuid = UUID();
```

6. Finally, the application creates an error fallbak function.

```
function(err, file) {
 if (!err) {
  admin.database().ref('posts').push({
   id: fields.id,
   title: fields.title,
   location: fields.location,
   image: 'https://firebasestorage.googleapis.com/v0/b/' +
bucket.name + '/o/' + encodeURIComponent(file.name) +
'?alt=media&token=' + uuid
})
```



Picture 69. Application Captures a picture using the device camera.

In this chapter, the camera was accessed by the PWA. The application can now take pictures using the device camera and send it to the database immediately to be displayed on the feed page.

13 CONCLUSION

This thesis discussed thoroughly the PWAs technology and introduced a set of additional features that provided a closer experience to a native application. A fully functional PWA was created by adding the progressive features gradually. The application implemented several different concepts to acquire a closer native mobile application behavior and look.

The developed application allows the user to take pictures using the device camera if available; otherwise, it activates a file uploader to allow an image upload. The pictures taken by the user are added to the feed page of the application to be displayed alongside the title and the location attributes.

The developed application has fulfilled the initial user requirements discussed before starting the implementation.

1. The manifest file was implemented, and several properties were added to enhance the applications display and provide a native application look. Further, the manifest provided installability on the home screen with a real application icon using a custom install banner.

2. The application provides a fully functional front end-user interface and offers the user the option to take pictures using the device camera or upload a picture file if no camera is available.

3. The application provides a fully functional back-end database and uses Googles' Firebase cloud functions to store data posted by the user.

4. The application implements real-time user interactions and posts the images taken by the user to the database and displays it in the front-end of the application immediately.

5. The application supports dynamic data update, as it caches data dynamically and updates the cache storage on the change of dynamic data; moreover, it deletes the old caches and implements the newest cache versions only. The application provides a flawless dynamic user experience.

6. The application offers offline functionalities by caching and fetching data as required. The service worker was created and registered. The service worker runs in the background and implements the cache and fetch technologies. Further, the application implemented background synchronization and enhances the application to send data requests when offline. The application can sync data when offline and send it when connectivity is re-established.

7. The application supports both; static cache and dynamic cache. The service worker implemented static caching, dynamic caching, cache cleanup and handled possible errors; moreover, it provided an offline fallback page.

8. The application uses push notifications to re-engage the user back into the application. The application can display notifications and send push notifications. A fully functional push service was implemented using the applications' own server hosted by Googles' Firebase.

9. The application accesses the camera device feature. The application allows the user to take pictures using the device camera; afterwards, it sends it to the database immediately to be displayed on the feed page.

10. The developed application works flawlessly on all browsers. Polyfills were added to handle unsupportive browsers; further, the application was coded to handle unsupportive browsers and offer further functionalities.

REFERENCES

1. Archibald J. 2013, JavaScript Promises: An Introduction. Google APIs December 16, 2013. Available at https://developers.google.com/web/fundamentals/primers/promises
2. Archibald J. 2014, The offline cookbook. jakearchibald.com December 9, 2014. Available at https://jakearchibald.com/2014/offline-cookbook/
3. Archibald J. 2015, Introducing Background Sync. Google APIs December 11, 2015. Available at https://developers.google.com/web/updates/2015/12/background-sync
4. Archibald J. 2018, IndexedDB Promised. Github.com/jakearchibald/idb June 21, 2018. Available at https://github.com/jakearchibald/idb
5. Basques K. 2018, Get Started with Remote Debugging Android Devices. Google APIs August 22, 2018. Available at https://developers.google.com/web/tools/chrome-devtools/remote-debugging/
6. ComScore 2017, 2017 U.S. Mobile App Report. Slideshare.net August 23, 2017. Available at https://www.slideshare.net/comScoremarcom/2017-us-mobile-app-report
7. Conlin J. 2016, Using VAPID with WebPush. Mozilla services April 4, 2016. Available at https://blog.mozilla.org/services/2016/04/04/using-vapid-with-webpush/
8. Deveria A. 2018, IndexedDB. Caniuse.com August 29, 2018. Available at https://caniuse.com/#feat=indexeddb
9. Deveria A. 2018, Service Workers. Caniuse.com August 29, 2018. Available at https://caniuse.com/#feat=serviceworkers
10. Deveria A. 2018, Web App Manifest. Caniuse.com June 21, 2018. Available at https://caniuse.com/#feat=web-app-manifest/
11. Gaunt M. & Kinlan P. 2018, The Web App Manifest. Google APIs May 14, 2018. Available at https://developers.google.com/web/fundamentals/web-app-manifest/
12. Gaunt M. 2014, Service Workers: An Introduction. Google APIs December 1, 2014. Available at https://developers.google.com/web/fundamentals/primers/service-workers/
13. Gaunt M. 2015, Introduction to fetch(). Google APIs March 12, 2015. Available at https://developers.google.com/web/updates/2015/03/introduction-to-fetch
14. Google development 2017, Cloud Functions for Firebase. Google Firebase March 9, 2017. Available at https://firebase.google.com/docs/functions/
15. Hume D. 2016, ServiceWorker: A Basic Guide to BackgroundSync. Ponyfoo July 19, 2016. Available at https://ponyfoo.com/articles/backgroundsync
16. Kong V. 2015, The 2015 U.S. mobile app report by Comscore. Slideshare.net September 24, 2015. Available at https://www.slideshare.net/victorkongcisneros/the-2015-us-mobile-app-report-by-comscore
17. LePage P. 2018, Add to Home Screen. Google APIs September 14, 2018. Available at https://developers.google.com/web/fundamentals/app-install-banners/
18. MDN web docs 2015, MediaDevices.getUserMedia(). MDN Mozilla April 1, 2015. Available at https://developer.mozilla.org/enUS/docs/Web/API/MediaDevices/getUserMedia

19. MDN web docs 2018, Async function. MDN Mozilla September 10, 2018. Available at https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/async_function

20. MDN web docs 2018, Cache. MDN Mozilla September 14, 2018. Available at https://developer.mozilla.org/en-US/docs/Web/API/Cache

21. MDN web docs 2018, Capabilities, constraints, and settings. MDN Mozilla September 6, 2018. Available at https://developer.mozilla.org/en-US/docs/Web/API/Media_Streams_API/Constraints

22. MDN web docs 2018, Geolocation API. MDN Mozilla September 6, 2018. Available at https://developer.mozilla.org/en-US/docs/Web/API/Geolocation_API

23. MDN web docs 2018, IndexedDB API. MDN Mozilla August 6, 2018. Available at https://developer.mozilla.org/en-US/docs/Web/API/IndexedDB_API

24. MDN web docs 2018, Notification. MDN Mozilla August 26, 2018. Available at https://developer.mozilla.org/en-US/docs/Web/API/notification

25. MDN web docs 2018, Promise. MDN Mozilla August 14, 2018. Available at https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise

26. MDN web docs 2018, Push API. MDN Mozilla September 10, 2018. Available at https://developer.mozilla.org/en-US/docs/Web/API/Push_API

27. MDN web docs 2018, Service Worker API. MDN Mozilla September 21, 2018. Available at https://developer.mozilla.org/en-US/docs/Web/API/Service_Worker_API

28. MDN web docs 2018, ServiceWorkerRegistration.showNotification(). MDN Mozilla September 21, 2018. Available at https://developer.mozilla.org/en-US/docs/Web/API/ServiceWorkerRegistration/showNotification

29. MDN web docs 2018, Sync Manager. MDN Mozilla July 13, 2018. Available at https://developer.mozilla.org/fi/docs/Web/API/SyncManager

30. MDN web docs 2018, Using Fetch. MDN Mozilla September 24, 2018. Available at https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API/Using_Fetch

31. MDN web docs 2018, Web App Manifest. MDN Mozilla August 23, 2018. Available at https://developer.mozilla.org/en-US/docs/Web/Manifest

32. MediaWiki 2018, API:Cross-site requests. Wikimedia project July 10, 2018. Available at https://www.mediawiki.org/wiki/API:Cross-site_requests

33. Medley J. 2016, Web Push Notifications: Timely, Relevant, and Precise. Google APIs October 1, 2016. Available at https://developers.google.com/web/fundamentals/push-notifications/

34. Microsoft docs 2018, Application Manifests, Docs.microsoft.com May 31, 2018. Available at https://docsmicrosoft.com/en-us/windows/desktop/sbscs/application-manifests

35. Schwarzmüller M. 2018, Progressive Web Apps (PWA) - The Complete Guide September 1, 2018. Available at https://www.udemy.com/progressive-web-app-pwa-the-complete-guide/?siteID=eyzsD2QGsYg-7AD.kxR1NjgJdK_B1Vwn2w&LSNPUBID=eyzsD2QGsYg

36. Shilova M. 2017, PROGRESSIVE WEB APPS: KEY BENEFITS, STATISTICS, USE CASES. Apiumhub September 14, 2017. Available at https://apiumhub.com/tech-blog-barcelona/progressive-web-apps/

37. W3schools 2016, AJAX Introduction. W3schools.com October 5, 2016. Available at https://www.w3schools.com/xml/ajax_intro.asp

38. W3schools 2016, JSON - Introduction. W3schools.com December 19, 2016. Available at https://www.w3schools.com/Js/js_json_intro.asp

39. Wikipedia 2017, Cross-origin resource sharing. Wikipedia Encyclopedia June 20, 2017. Available at https://en.wikipedia.org/wiki/Cross-origin_resource_sharing