

**Honours Project (CCIS) Final Report**

**2018-2019**

**Submitted for the Degree of: BSc Computing**

**Project Title:** Implementing offline capabilities using the Service Worker API

**Name:**

**Programme: BSc (Hons) Computing**

**Matriculation Number:**

**Project Supervisor:**

**Second Marker:**

**Word Count:** 13,202

**“Except where explicitly stated, all work in this report, including the appendices, is my own original work and has not been submitted elsewhere in fulfilment of the requirement of this or any other award”**

**Signed by Student: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Date: \_\_\_\_\_\_\_\_\_\_\_\_**

**Abstract**

Creating web apps that work offline or with an intermittent network has long been a challenge for developers. Traditionally app developers have had to choose between building a native app or a Single Page Application (SPA), which each have their own limitations and require bespoke skills.

The Service Worker API is a new technology that aims for web pages to benefit from many of the features traditionally found in native apps, such as push notifications and background sync. The aim of this project is to investigate and evaluate the API to determine how it can be used to achieve an improved user experience when working either offline or with an intermittent network.

The results show that for a seamless user experience the app should be able to fall back to using cached data in a timely manner. When the network is fully on or off this is easy, but hard to detect with an intermittent connection. Additionally, while Service Worker allows development of features not normally found in web apps, they often required lots of boilerplate code that slows down development.

The conclusions were that it would be beneficial for browsers to have features that not only detect if the network is online or off, but also provide information about the quality of the network, allowing cached data to be used more intelligently. Additionally, the creation of libraries and frameworks to aid Service Worker development would be welcome.

**Acknowledgements**

Thank you to my project supervisor … for her help.

Table of Contents

[1 Introduction 6](#_Toc6220024)

[1.1 Background 6](#_Toc6220025)

[1.2 Project Outline and Research Question 7](#_Toc6220026)

[1.2.1 Project Outline 7](#_Toc6220027)

[1.2.2 Research Question 8](#_Toc6220028)

[1.2.3 Objectives 8](#_Toc6220029)

[2 Literature and Technology Review 10](#_Toc6220030)

[2.1 What is a Service Worker? 10](#_Toc6220031)

[2.2 Where does Service Worker API fit into the world of PWAs? 10](#_Toc6220032)

[2.3 Features of Service Worker 11](#_Toc6220033)

[2.3.1 Caching resources for offline using Cache API 11](#_Toc6220034)

[2.3.2 Downloading data using Fetch API 11](#_Toc6220035)

[2.3.3 Background storing and syncing data 12](#_Toc6220036)

[2.3.4 Handling long running processes 12](#_Toc6220037)

[2.3.5 Push notifications 12](#_Toc6220038)

[2.4 What other technologies and tools will be needed to develop the project? 12](#_Toc6220039)

[2.4.1 Backend Framework 13](#_Toc6220040)

[2.4.2 Data Storage 13](#_Toc6220041)

[2.4.3 Network Simulation Tools 13](#_Toc6220042)

[2.4.4 Code Editor 13](#_Toc6220043)

[2.4.5 Debugger 14](#_Toc6220044)

[2.5 What criteria could be used to assess the “seamlessness” of an app? 14](#_Toc6220045)

[2.6 What caching strategies can be used in a web app? 14](#_Toc6220046)

[2.7 Conclusion 15](#_Toc6220047)

[3 Methods 16](#_Toc6220048)

[3.1 Primary Research Methods 16](#_Toc6220049)

[3.1.1 Analysis of Service Worker and requirements generation 16](#_Toc6220050)

[3.1.2 Design app that implements requirements 16](#_Toc6220051)

[3.1.3 Implementation of app 16](#_Toc6220052)

[3.1.4 Testing of application in different network conditions 16](#_Toc6220053)

[3.1.5 Evaluation of Service Worker API 17](#_Toc6220054)

[4 Execution 18](#_Toc6220055)

[4.1 Analysis 18](#_Toc6220056)

[4.1.1 Functional Requirements 18](#_Toc6220057)

[4.1.2 Non-functional Requirements 19](#_Toc6220058)

[4.2 Design 20](#_Toc6220059)

[4.2.1 Use Case Diagram 20](#_Toc6220060)

[4.2.2 Class Diagram 20](#_Toc6220061)

[4.2.3 Geofence and Notifications Diagram 21](#_Toc6220062)

[4.2.4 Wireframe Prototype 22](#_Toc6220063)

[4.2.5 Front-end vs back-end 23](#_Toc6220064)

[4.3 Implementation 25](#_Toc6220065)

[4.3.1 Shell and Cache API 25](#_Toc6220066)

[4.3.2 Live data, Fetch API and Indexed DB 27](#_Toc6220067)

[4.3.3 Background sync 29](#_Toc6220068)

[4.3.4 Geofence 32](#_Toc6220069)

[4.3.5 Scheduler and push notifications 33](#_Toc6220070)

[4.3.6 Deployment 37](#_Toc6220071)

[5 Testing & Evaluation 38](#_Toc6220072)

[5.1 Caching shell 38](#_Toc6220073)

[5.2 Fetching and caching data 39](#_Toc6220074)

[5.3 Background sync 40](#_Toc6220075)

[5.4 Geofence 42](#_Toc6220076)

[5.5 Push Notifications 42](#_Toc6220077)

[6 Conclusions 45](#_Toc6220078)

[7 References 46](#_Toc6220079)

[8 Appendix A 50](#_Toc6220080)

[8.1 Appendix – Wireframes 50](#_Toc6220081)

[8.2 Appendix – Test Results 54](#_Toc6220082)

[8.2.1 Appendix – Cache API Tests 54](#_Toc6220083)

[8.2.2 Appendix – Fetch Tests 55](#_Toc6220084)

[8.2.3 Appendix – Push Notification Tests 56](#_Toc6220085)

[8.2.4 Appendix – Background Sync Tests 57](#_Toc6220086)

**Table of Figures**

[Figure 1 - Clumsy Network Simulation Tool 13](file:///C:\Users\alexm\Google%20Drive\Uni\2019\Trimester%20B\honours\final%20draft\Alex%20McBride%20Honours%20Project%20Final%20Report.docx#_Toc6220816)

[Figure 2 - Use case diagram 20](#_Toc6220817)

[Figure 3 - Class diagram for data model 21](#_Toc6220818)

[Figure 4 - How geofence and push notifications work 22](#_Toc6220819)

[Figure 5 - Wireframe of index page 23](#_Toc6220820)

[Figure 6 - Service worker registration code wrapped in a promise 26](#_Toc6220821)

[Figure 7 - Code to cache list of URLs in cache with specified name 26](#_Toc6220822)

[Figure 8 - Fetch event listener which responds with cached data if it exists 27](#_Toc6220823)

[Figure 9 – Gnarly code needed to get a collection from Indexed DB. 28](#_Toc6220824)

[Figure 10 - Code to fetch data from network or fall back to cache in repository.js 29](#_Toc6220825)

[Figure 11 - Background sync process 30](#_Toc6220826)

[Figure 12 – Code from background-sync.js to add an item to the sync queue 30](#_Toc6220827)

[Figure 13 - Example code in sw.js to handle sync event 31](#_Toc6220828)

[Figure 14 - Geofence code showing watching location updates and updating repository 33](#_Toc6220829)

[Figure 15 - Register a push notification with the server 34](#_Toc6220830)

[Figure 16 - Send a push notification to the client 35](#_Toc6220831)

[Figure 17 - Code to send push notification using web-push library 36](#_Toc6220832)

[Figure 18 - Code to send push notification to user's subscriptions 36](#_Toc6220833)

[Figure 19 - Push event handler in the sw.js file 37](#_Toc6220834)

[Figure 20 - The latency and data transferred with and without the cache 38](#_Toc6220835)

[Figure 21 - Average latency for Fetch requests (note logarithmic scale) 39](#_Toc6220836)

[Figure 22 - Number of times the request was fulfilled in different network conditions 40](#_Toc6220837)

[Figure 23 - Sync success counts with increasingly poor network 41](#_Toc6220838)

[Figure 24 - Average latency for a sync operation to complete at increasing drop rates 41](#_Toc6220839)

[Figure 25 - Push notification average latency at increasing drop rates 43](#_Toc6220840)

[Figure 26 - Number of push notifications that succeeded at increasing drop rates 43](#_Toc6220841)

[Figure 29 - The index page wireframe 50](#_Toc6220842)

[Figure 30 - The add reminder page wireframe 51](#_Toc6220843)

[Figure 31 - The edit reminder page wireframe 52](#_Toc6220844)

[Figure 32 - The settings page wireframe 53](#_Toc6220845)

# Introduction

In this section background information about the topic will be introduced, which will look at the history of providing offline capabilities to web pages. Next, the project will be outlined along with the research question the project hopes to answer, and finally the project objectives will be described.

## Background

Since the early days of the web, developers have been concerned with creating good offline experiences for users. As the web is stateless, and little or no information is stored on the browser, when the connection with the server becomes unavailable the user is not able to perform any actions. In the days of desktop computers connected through permanent wired connections this was not a problem, but today with the huge number of mobile devices it has become an issue of significance. Creating a web app that works when offline is a challenge for developers and creating one that works with an intermittent connection harder still (Peischl, et al., 2015).

A popular method to circumvent this is to create the app as a Single Page Application (SPA). This involves putting all the code for a web application inside a single page, which is then loaded when the user first visits the site (Elchuk, 2016). The SPA waits for events from the user, such as interacting with the UI, and then makes requests to the server in the background using AJAX (Asynchronous JavaScript and XML). The web app will programmatically show different views to the user depending on how the user interacts with it. While the initial payload for these sorts of apps is high, the files downloaded are stored in the browser’s cache thereby making future visits more responsive. As the user stays within a single web page the loss of the network connection is not as impactful as it would be using multiple stateless pages. Additionally, SPAs can store data locally within the browser, and retrieve it in a different session, making it able to still function when the user is offline (Wang & He, 2014).

A second method of providing an application with a good offline experience is a native app, typically running on an Android or iPhone device (Charland & LeRoux, 2011). These apps run directly on the hardware as first-class citizens and provide several advantages. Firstly, they have access to features such as the camera and graphics processor. Secondly, the app can run when the device is offline and can store data in the device’s permanent storage. This is powerful and flexible and enables applications to provide a full user experience with high performance and reliability. The downside is that phone apps must be installed through the app store, taking the user away from the task they are trying to complete. What’s more they often require hefty downloads and agreement to confusing permissions. Both Android and iOS apps are built on different runtimes, with different languages and programming APIs, meaning that the app must be built bespoke for each device, vastly complicating development efforts. Smart phone apps also have a privacy hurdle to overcome not found on the web, as apps installed on the user’s device are typically less trusted than web sites, and a survey of US phone owners showed that more than half had uninstalled an app over concerns about the way it collected personal data (Boyles, et al., 2012).

When developers set out to create a new application they need to decide upfront whether providing a good offline experience is critical to the requirements, and if so their development choices are largely limited to either native or single page apps (Gralla, 2015). These are two strikingly different development platforms, each requiring a bespoke set of skills that are not easily transferable. They are also largely insulated platforms, meaning there is little if any code sharing possible between the two. A native app and a SPA can share the same backend service providing it has been implemented in a suitably platform independent way, but this requires even more work by the developer (Puder, et al., 2014). If developers have an existing platform, say a traditional web site that they want to add offline functionality to, this is very difficult to achieve and would likely involve an expensive rewrite of the whole system.

Progressive Web Apps (PWAs) are a new technology developed by Google that aim to improve the offline web experience by making web pages work more like native apps (Krill, 2015). By building on technologies introduced with HTML5, PWAs allow existing web sites to gradually introduce features found in phone apps, without the need for completely rewriting the code. PWAs are comprised of several technologies: a manifest that allows developers to associate meta-data with a web page, allow web pages to respond to requests when offline and handle push notifications, and operating system integration that allows pages to mimic certain aspects of native apps, such as hiding the chrome shell and placing icons on the user’s home screen. Most of these new features are made possible by Service Worker API, which allows developers to store a script on the user’s browser that can intercept requests from the page and choose how to respond to them (W3, 2017).

Progressive Web Apps have the potential to replace native and single-page apps for many development scenarios, by combining the features of a native app with the ease of use and development of the web (Sharma, 2017). For developers with a legacy system it allows them to add features to an existing site gradually, letting them gain the benefit of modern development practices without having to undertake complex and expensive rewrites. However, while a lot of been suggested of this new technology, it is early days and there are still unanswered questions, such as exactly how well PWAs and Service Worker API can offer good offline support, especially when the user is working with an intermittent network connection. The aim of this research project is to investigate this.

## Project Outline and Research Question

This section will outline the goals of the project and describe the research question that the project will aim to answer. It will also list the main objectives for the research, for both the main literature review and the primary research phase.

### Project Outline

The project is of the type ‘develop and test’ and aims to evaluate Service Worker API in the creation of a web application that works in different network conditions. The features of the Service Worker will be analysed, and an application developed that tests each one. The design and implementation of the app will follow the typical software development methodology of analysis, design, implementation, and testing. The application will connect to a remote server and allow the user to view and update data, which will be also cached locally in the browser.

Once completed, the app will be run through multiple testing scenarios which will involve testing each feature while simulating different network conditions. During testing data will be collected, such as response times and failure rates. This data can then be analysed to generate a detailed view of how effectively the technology will be able enhance the end-user experience in situations where a stable network connection is not available.

The results of this research would aid developers in making an informed choice as to which technology they should choose for creating new applications. Previously developers have had to choose between creating a web app or a native phone app, but with PWAs and the Service Worker API they have the option to pick a technology that purports to offer the best of both worlds. This research will investigate how realistic that is, and exactly what this new technology can do.

### Research Question

Can the Service Worker API be used to create a web application that provides a seamless user experience when the user is offline or with an intermittent network connection?

### Objectives

In this section we will look at the objectives of the project, first for the main literature review, and then the primary research phase. We will look at each objective and what it aims to achieve, and how that objective will be undertaken.

#### Literature Review

The main literature and technology review is where information needed to successfully complete the project will be collected. To do this the research question will be broken down into several sub-questions, as follows:

* What is Service Worker API?
* What are the features of Service Worker API?
* What other technologies and tools will be needed to build the application?
* What criteria can be used to assess an app built using Service Worker API?
* What caching strategies can be used when creating an offline web app?

Once these questions have been answered work can move to the next phase of the project.

#### Primary research phase

The primary research phase is where the actual work of project will take place, and where the research instrument will be implemented and evaluated. There are several objectives which will be undertaken during the primary research phase to achieve the project’s aims. These are outlined below.

##### Analyse Service Worker API and design an app that tests it

Once the features of the Service Worker API have been collected an application will be devised to test them. The requirements will be written up as a set of functional and non-functional requirements, these will then be turned into a design that can be implemented.

##### Develop an application that uses Service Worker API

After design has completed the implementation will begin, where the requirements will be turned into working source code. This will involve implementing each requirement of the app and testing it to make sure it is working as intended.

##### Test and Evaluate Service Worker in different network conditions

Once the app has been built it will be evaluated by testing each feature while simulating different network conditions. These will include online, offline, but also with a poor network connection. Intermittent network conditions will be tested using network simulation tools. Once data has been collected the features of Service Worker can be evaluated, their effectiveness measured, and conclusions drawn.

# Literature and Technology Review

The literature and technology review section of the document is where the knowledge needed to develop the project is gathered. The review will first look at the features of Service Worker, so that requirements for an app that explores them can be drawn up, as well as how the technology fits in the world of Progressive Web Apps (PWAs). Next it will look at additional technologies needed to complete the project, such as backend framework and network simulation tools. Finally, the criteria needed to evaluate the app will be considered.

## What is a Service Worker?

A Service Worker is a JavaScript file that is registered with a web page and sits between the browser and the web document and intercepts requests (Google, 2018). This allows the developer to control how the page will respond in certain situations, such as the network connection going offline (Bidelman, 2013). The worker runs like a service in the background and so can also perform other tasks, such as caching resources, handling push notifications, and processing long running operations. A downside to this is that as the script is running like background service it may waste scarce device resources, however as Service Worker follows an event-based model and only stirs to life when needed it has been found to be energy efficient (Malavolta, 2017).

A key aspect of the service workers architecture is that is asynchronous and runs on a separate thread to the main web page, meaning that it cannot access the DOM (Document Object Model) (Mozilla, 2018). Therefore, to update elements of the page requests need to be posted on the main UI thread as messages. Additionally, the SW cannot access synchronous APIs such as XMLHttpRequest (XHR) or localStorage, which are often used in web apps, and instead has specialised APIs for these. For security reasons Service Worker can only be used with HTTPS, which encrypts the connection between the browser and the web server.

## Where does Service Worker API fit into the world of PWAs?

Service Worker is a sub-technology of Progressive Web Apps (PWA), a new concept in web development that aims to provide web pages with functionality that has normally been the preserve of native applications (Google, 2018). PWA is not a framework or library, but rather a checklist of features that make a web pages function more like a native apps, while keeping aspects of what makes the web highly usable (bookmarking, deep linking, no installs, instant updates etc).

The PWA checklist (Google, 2018) lists several attributes to which an app must adhere to conform to what is expected of a progressive app. The checklist is split into two parts, “baseline” for the minimum that must be implemented, to “exemplary” which contains which contains “nice-to-have” but not essential features. The baseline list contains features such as:

* Implement Service Worker API
* Serve pages over HTTPS
* Be responsive on small screen devices such as tablets and mobiles
* Give each page and resource should have its own URL
* Handle requests when the network is offline or only available intermittently

The exemplary checklist is based around general user experience and lists items such as content not “jumping” around as the page loads and ensuring that the back button works as expected (something which can be tricky to pull off in a single page app). Microsoft also has their own smaller checklist where the minimum to qualify as a PWA is to use HTTPS, implement Service Workers, and have an app manifest (MSDN, 2018). The Lighthouse tool can analyse web apps to see how well they conform to the PWA checklist and can also be used to assess the performance and accessibility of pages (Google, 2018).

One issue with using Service Worker API in a web app is browser support. As PWAs are a new technology browser support has been patchy, especially outside of Google Chrome. However, support has been growing and most features are now supported by the four major browsers: Chrome, Firefox, Safari, and Edge (caniuse.com, 2018). A solution to browser incompatibility is progressive enhancement, which allows web apps to use fallbacks and alternatives for when a feature is not available (Fox, 2012). Additionally, polyfills can be used that implement feature needed by a web application that is not available on that specific browser (Smith, 2012). These tools are useful but add dependencies on third party libraries, which can increase download size and code complexity.

For our application we will focus on a pure Service Worker with the use of as little third-party code as possible. When developing our application, we will follow the PWA guidelines to keep our app as compatible as possible. This will make sure that we are following the best practices when creating our application and will mean we have a solid baseline to test against.

## Features of Service Worker

This section of the document will look at the features of Service Worker API so that a comprehensive list of requirements can be created, allowing the API to be tested as thoroughly as possible.

### Caching resources for offline using Cache API

A primary feature of Service Worker is the ability to cache resources for access later, such as when the network connection is unavailable (Google, 2018). This is done using the Cache API, which provides the ability to store key/value pairs in the form of requests and their responses (MDN, 2018). While the cache can be used to store any resource (download size and network bandwidth permitting) it is typically used to store the shell of the application, such as the headers, navigation and footers (PePage, 2018). These normally do not change from screen-to-screen and only need to be downloaded once. When the shell is updated the cache will need to be invalidated, which requires careful management of the service worker’s lifecycle. How effective the cache is at reducing data transferred will need to be tested.

### Downloading data using Fetch API

While the shell of the application is cached, the data that the app is displaying needs be fetched from the application backend. This is typically done by the way of JSON (JavaScript Object Notation) data downloaded from a RESTful API (Cudini, et al., Jun 2017). As Service Workers cannot assess synchronous code and XHR is not available, the Fetch API was introduced as a replacement (MDN, 2018). The Fetch API returns a JavaScript Promise, which can be resolved at a future time to get a response (MDN, 2018). Data will need to be downloaded and cached by our application in Indexed DB, and then served up as needed using different network/caching strategies. The tests for feature this will aim to record that the request was successfully completed, and how long the request took to complete.

### Background storing and syncing data

Many operations can only be completed while the network is active, for instance posting data to the server. If the network becomes unavailable the operation will fail and data lost. Service Worker provides a solution that allows network requests to be stored locally and then synced when the connection becomes active again (Archibald, 2015), a feature previously only found in mobile devices. As Service Worker runs as a background service this can be accomplished even when the user has closed or navigated away from the web page that made the request (W3, 2017). By preventing the loss of information and removing the possibility of annoying error messages, this feature has the potential to improve the user experience greatly. The research will therefore need to gauge how effective the background sync is and how well it can function in different network conditions, especially when the connection is intermittent. This can be tested by storing and syncing data in different network conditions and then recording whether the sync completed successfully and how long it took to complete.

### Handling long running processes

As the service worker is running in the background and is not tied to the context of a single page, it is able to handle operations that extend beyond the lifespan of a single page or request (W3, 2017). This might involve creating a geofence, where certain functionality is only available within defined geographic bounds (Young-Hyun, et al., 2016). Additionally, several pages can share a single service worker script, so data can also be shared among them. Geolocation is just one example of this, another might be collecting sensor data from the device’s compass or gyroscope. To explore this, a long running operation will be created, and tests run both when the browser is offline and when the user has navigated away from the page.

### Push notifications

Push notifications are a good way of driving engagement with an app and have traditionally only been available for smart phone apps (Warren, et al., 2014). Service Worker API provides the ability to send notifications to web pages, regardless as to whether the page is active or not (MDN, 2018). This is managed through the Push API, which was introduced alongside the rest of the Service Worker. As such a key feature of the Service Worker this new feature will need to be assessed, especially in gauging how effectively notifications can get received when in very poor network conditions. This can be tested by sending push notifications to the device while simulating different network errors, and then confirming that the notification was received, and how long it took for the notification to arrive.

## What other technologies and tools will be needed to develop the project?

While the project focuses mainly on Service Worker API, many other technologies and tools will be needed.

### Backend Framework

While most of the app will be client-side JavaScript, a backend service will be needed to communicate with. Node.js is a modern and flexible backend framework that is a good fit for developing web APIs (Chitra & Satapathy, 2017). Any backend framework could be used but using Node.js will allow us to keep the frontend and backend languages the same which will make development easier. Node lacks an extensive standard library, such as those seen in frameworks like .NET and Java, but instead additional modules can be downloaded using the Node Package Manager (NPM). The huge number of packages can cause issues though, through creating lots of hard to manage and often opaque dependencies. In the past package authors have removed their modules without notice causing many applications and libraries to fail (Williams, 2016). For these reasons careful management of package dependencies is needed, and it’s important to be aware of hidden dependencies inside of imported frameworks and libraries.

### Data Storage

Backend storage will use MongoDB, a popular NoSQL document database for storing unstructured and semi-structured data (Butgereit, 2016). Document databases were created in response to the demand for scalable databases that could be searched and queried more easily than key-value data stores (Scott, 2016). MongoDB stores data using a JSON-based API, where data is represented by data structures similar to JavaScript objects, which makes it a good fit when working with Node.js applications. Unlike traditional relational databases like MySQL, data in Mongo is not structured, meaning that structure needs to be enforced in the application’s data layer.

### Network Simulation Tools

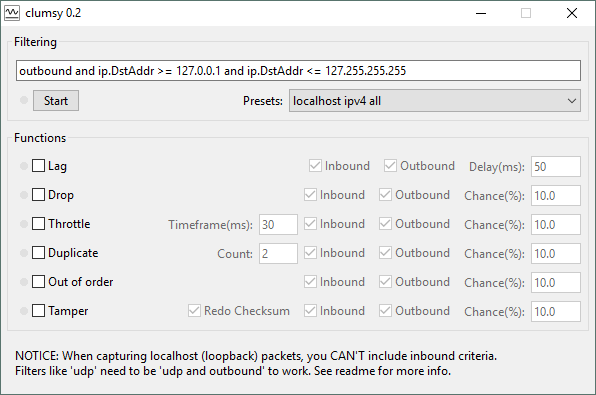
To test the app in different network conditions a network simulation tool will be needed. The most significant problems faced by poor networks are high latency and packet loss (Ullah, et al., 2014). Therefore, we will use a tool called Clumsy to simulate these issues (see Figure 1). Clumsy can mimic many different types of network problems, including lag and dropped packets (Clumsy, n/a). The tool allows the percentage drop rate for packets to be specified, which will allow the network to be tested at varying intervals of failure.

Figure 1 - Clumsy Network Simulation Tool

### Code Editor

An IDE will be needed during development. Any editor can be used, but VS Code is a modern and popular code editor, developed by Microsoft. While the basic editor is quite simple it can be extended with many plugins. VS Code is written in Node itself using the Electron framework, but that’s not relevant for this project. VS Code can connect to the Node runtime for debugging server-side JavaScript, as well as connect to Chrome Developer Tools for debugging client-side code.

### Debugger

Apps built with the Service Worker API can be managed through the Chrome Developer Tools, present in the Google Chrome web browser (Google, 2018). These features include: simulating network being offline, forcing an existing Service Worker to be replaced, as well as viewing the contents of caches and storage. These features will be useful when working with Service Worker and should make developing the app simpler.

## What criteria could be used to assess the “seamlessness” of an app?

The question that the research hopes to answer is whether an app can be made to be “seamless”, in terms of offering an experience to the user which works well offline, or with an intermittent connection. To gauge seamlessness, we need to consider what response times can be used when making measurements. One way to do so is to implement Nielsen’s powers of ten rule where timeframes go up in increments from 0.1 seconds to 100 years, each having its own usability concerns (Nielsen, 2009). A century is outside of the timeframe needed for this application, so we will only consider timescales up to ten minutes.

* 0.1 secs – Actions appear to happen instantly, and users feel that they caused the action to happen. This is within the range of human physiological responses, such as blinking.
* 1 sec – The user feels like the computer caused the action rather than themselves. However, while they may notice the delay their train of thought will stay focused on the task at hand.
* 10 seconds – At this timeframe users get impatient with waiting and may lose their train of thought. The current task they are performing may leave short term memory and it may take a moment for the user to remember what they were doing.
* 1 minute – Tasks that take longer than one minute may be abandoned regardless of their importance, therefore users should be able to complete whole tasks within this timeframe.
* 10 minutes – A whole web site session should be able to be completed within this timeframe.

With this information in mind, to feel ‘seamless’ operation should ideally take 100 milliseconds, but any task that takes under one second is enough to keep the user’s mind on the task in hand. Any task that takes over ten seconds will cause the user to be distracted, but this would be acceptable for something like uploading a file. A whole task should take less than one minute and a typical web session, such as the completion of multiple tasks, should take ten minutes or less. These timeframes will be specified in the non-functional requirements of the app, so they can be used to evaluate the application during the final phase of the project.

## What ***caching*** strategies can be used in a web app?

When dealing with a web app that can work when offline we will need to cache data, so it is important to have knowledge of the different caching strategies which can be used. Different strategies determine how the app will function when the network is offline or being used with a poor network. There are four we are concerned with (Archibald, 2014), which are:

* Cache falling back to network – This strategy prioritises offline first, with online as a backup. We try to get data from the cache and if that fails we request it from the network. This strategy is best if the data from the network does not update very often.
* Cache and network race – This strategy has a race between cache and network and will use the one that returns first, which is normally the cache as network requests tend to have a more latency. However, on devices with poor disk access this can be effective.
* Network falling back to cache – This strategy prioritises online first and uses the cache as a backup. First an attempt to fetch the data from the network is made and if that fails we fall back to using the local cache. This is a good strategy for data that update frequently.
* Cache then network – This hybrid strategy uses the cached data first so that data is displayed immediately, and then performs a request to bring in fresh data from the network. The issue with this strategy is that the data being displayed may not be able to be changed seamlessly and may cause what the user is looking at to vanish. An alternative is to inform the user that new data has been received and ask them if they want to update, but this requires more work to be done in the UI.

Different strategies are useful in different scenarios. For instance, for the shell of our application a cache first strategy can be used, as the shell of our application won’t change much. For downloading data such as JSON and caching it in Indexed DB a network first strategy should be used. Our goal here is to stress the Service Worker as much as possible for testing purposes and that means stressing the parts that interact with the network. We are assuming that fetching data out of the cache will be quicker than fetching it across the network, however we will confirm this during testing. The background sync feature will use a cache first fall back to network strategy.

## Conclusion

To answer the research question, it seems clear that a research instrument should be devised that intersect with each feature of the Service Worker API. The web application will require a backend to communicate with, and that will be implemented using Node.js, with data stored using MongoDB. The Clumsy network simulation tool will be used to test in various network conditions such as lag and packet loss. Using the Nielsen Powers of Ten rules we can evaluate the results to determine if they are seamless or not. Test data will be collected will then be analysed, evaluated, and written up in the final report.

# Methods

This section will look at the primary research methods to be used, including a breakdown of how the application will be developed, before examining how the evaluation of the completed app will be handled.

## Primary Research Methods

The research method chosen for this project is ‘develop and test’. An application that uses Service Worker API will be developed and tested to attempt to answer the research question. Development will follow the traditional software development lifecycle of analysis, design, implementation, and testing. A further phase of evaluation will be added to the end, where the finished app will be evaluated against the test criteria we have drawn up. By developing an app in this way, we will be able to generate first-hand data on how this new technology can be used to improve the user experience in poor network conditions.

### Analysis of Service Worker and requirements generation

The first phase will be to take the analysis of the Service Worker that has been completed and generate a set of requirements for our application. The Service Worker specification lists the main features of the technology as: caching resources, fetching data from the server, background storing and syncing data, handling push notifications, long running operations, and updating the service worker script when it changes (W3, 2017). Requirements will be split into two areas, functional and non-functional, following the typical requirements engineering process (Sommerville, 2016).

### Design app that implements requirements

The next stage will be to invent an application that fulfils the requirements drawn up in the previous section and then design it. This will include the creation of layouts for the application, such as wireframes or mocks, created with a UI prototyping tool. Additionally, a model will also be designed and specified using Unified Modelling Language (UML), which will allow an abstract model of the system to be created that represents a specific viewpoint (Sommerville, 2016). The design could be just a series of features intended purely to test the Service Worker API in a lab-type setting, however to comprehensively test the technology the goal will be to design cohesive real-world app. The intention of this is to make it simpler to evaluate the application at the end of the implementation phase, especially regarding the user experience.

### Implementation of app

The implementation of the app will involve taking the model created during the design phase and converting it into working software. The implementation will involve building both the backend of the application using the Node.js framework and Mongo DB, and the frontend browser application using JavaScript. The coding will be done using VS Code, supported by the Chrome Developer Tools and debugger.

### Testing of application in different network conditions

Testing will involve conducting tests of the Service Worker API’s features in different network conditions. Testing with the network online and offline can be done by toggling the connection with the Chrome Developer tools. However, testing with an intermittent connection is trickier to achieve and will be done with the Clumsy network simulation tool. This tool allows you to simulate a poor or dropping connection by specify the packet drop rates. Each test will collect latency data for different drop rates which will be used to determine seamlessness.

### Evaluation of Service Worker API

The final phase of the project will be evaluation, where the data collected during the testing phase will be evaluated and used to answer the research question. We will compare latency data for various offline, online, and intermittent network tests against the Nielsen powers of ten user experience rules. The research will aid future developers in making informed choices about the technologies and options available to them for creating applications with good offline experiences.

# Execution

This part of the report will look at the execution of the project and provide detailed information about the development of the project’s primary research instrument. It will look at why particular techniques and solutions were adopted and explain how different parts of the project were tackled. This phase was implemented using the traditional software development lifecycle of analysis, design, and implementation, and the report will be broken down into those sections. Normally, the lifecycle would be finished off with a testing phase, however for this project that has been combined with evaluation and will be covered in the next section.

## Analysis

The initial analysis phase examined the features of the Service Worker API and converted them into a set of requirements that could be developed into an application. The end goal was to have an application that could be used to test and evaluate the technology. During the main literature review it was determined that to test the Service Worker we would need to handle the following requirements:

1. Cache app shell for use offline
2. Download live data using Fetch API
3. Cache data locally using Indexed DB
4. Queue updates and sync them when the network is available
5. Run a process outside of the lifecycle of a single page
6. Send push notifications from the server to the user’s device

It would have been possible to simply develop and test these as separate features; however, a cohesive app design was preferred as that makes it easier to test and to gauge the user experience. Therefore, a simple app concept was devised, that will allow each of these requirements to be exercised.

The app is intended to remind students when to attend classes, labs, lectures, and other university activities, to help improve attendance. Students can enter details for a reminder, such as title, room, date, and duration, and then will receive push notifications when these events are due to take place. The shell of the app will be cached, and the reminder data stored in Indexed DB. User input such as creating reminders will be stored and synced when the network is available. The app will monitor the student’s location using a geofence and give more detailed notification updates if the student is on university grounds.

### Functional Requirements

The above app concept was turned into functional requirements, which can be seen in Table 1. The table shows the requirement, some additional info where necessary, and the Service Worker feature that the requirement is designed to cover.

|  |  |  |
| --- | --- | --- |
| **Requirement** | **Info** | **Requirement** |
| Display shell of reminder app | Header, footer, navigation | Cache API (1) |
| List upcoming reminders |  | Fetch API (2), Indexed DB (3) |
| Add, edit, and delete reminders | Details to store:   * title * type (e.g. lecture, lab) * room * date/time * duration (mins) | Background sync (4) |
| Import calendar file containing timetable information | Import reminders from ical file |
| Set university location using Google map interface | University location at latitude/longitude  Distance from this location that will be considered on uni grounds | Long running operation (geofence) (5) |
| Send reminder notifications to subscribed devices | Notify student 1 hour before event  Notify again 5 mins before (if on uni grounds) | Push notifications (6) |

Table 1 - Function requirements and service worker feature they are designed to cover

### Non-functional Requirements

Additional non-function requirements are needed for the app. These are important as they specify timeframes that will be used in the evaluation of the application.

* All functionality must work offline
* Must work with intermittent connection
* Must display reminders list in less than one second
* Importing large calendar should succeed when connection drops
* Importing large calendar should take less than ten seconds
* User must be able to complete all tasks in under one minute
* Notification should be sent within ten seconds of event being detected on the backend
* Must be responsive and scale between desktop and mobile
* App should be compatible with Chrome browser (Service Worker features such as sync not available on Firefox or Edge)

## Design

The design section of the document will describe how the analysis was turned into a design which was able to be implemented. Several artefacts were generated such as a use case diagram, class diagram, system diagram, wireframes, and wireframe prototype. The aim of the design phase was to do as little up-front work as needed to be able to start coding.

### Use Case Diagram

A basic use case diagram as seen in Figure 2 was created for the app, to show the various requirements and how they connect.

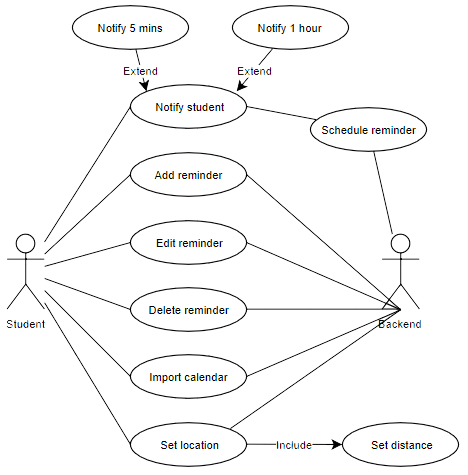


Figure 2 - Use case diagram

This diagram shows how the user can update reminders, while on the backend a scheduler notifies students of upcoming occurrences. The location you are at affects which notifications are sent.

### Class Diagram

A basic class diagram (Figure 3) for the data model was created, that shows the relationship between the user and the reminder classes. A user is composed of reminders, and therefore owns their lifecycle.

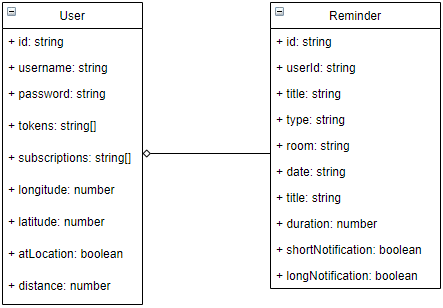


Figure 3 - Class diagram for data model

The tokens field is where the authentication tokens are stored, which are used to authenticate requests from the client to the server.

The subscriptions field stores the clients currently subscribed to push notifications, allowing multiple devices to be subscribed at once, such as both laptop and phone.

The atLocation field is set from the client when the geofence detects that the user is within distance of the longitude and latitude specified, and this is used to determine if the student is on university ground and therefore what notification type to send.

The userId stores the foreign-key relationship between user and reminder, although as we are using the Mongo DB this relationship is enforced programmatically.

The shortNotification and longNotification boolean values are used to store which notification type has been sent previously, to prevent the user being spammed with notifications. The short being five minutes and long being one hour.

### Geofence and Notifications Diagram

The next diagram (shown in Figure 4) was created to show how geofencing and push notifications will work. When the user enters the specified area detected by the geofence, a flag is set on the backend and stored in the data store. The scheduler checks for upcoming reminders in the data store periodically and triggers a push notification when needed. Notifications are then sent to the user’s device where they are displayed in a device dependent way, such as in the notification area.

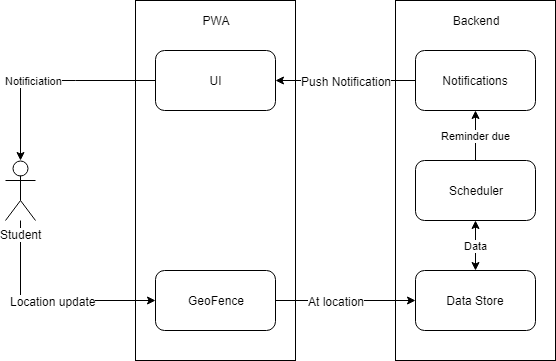


Figure 4 - How geofence and push notifications work

### Wireframe Prototype

A basic wireframe prototype was created using the Moqups online tool and an example is shown in Figure 5. This wireframe shows the main index page of the application. The sidebar and title represent the shell of the application that will be cached, while the reminders list is generated from fetched or cached data. We can add a reminder or edit the settings by following the navigation link. Links on each reminder allow the item to be edited or deleted. Only a single page is displayed here, the full set of wireframes can be found in Appendix – Wireframes.

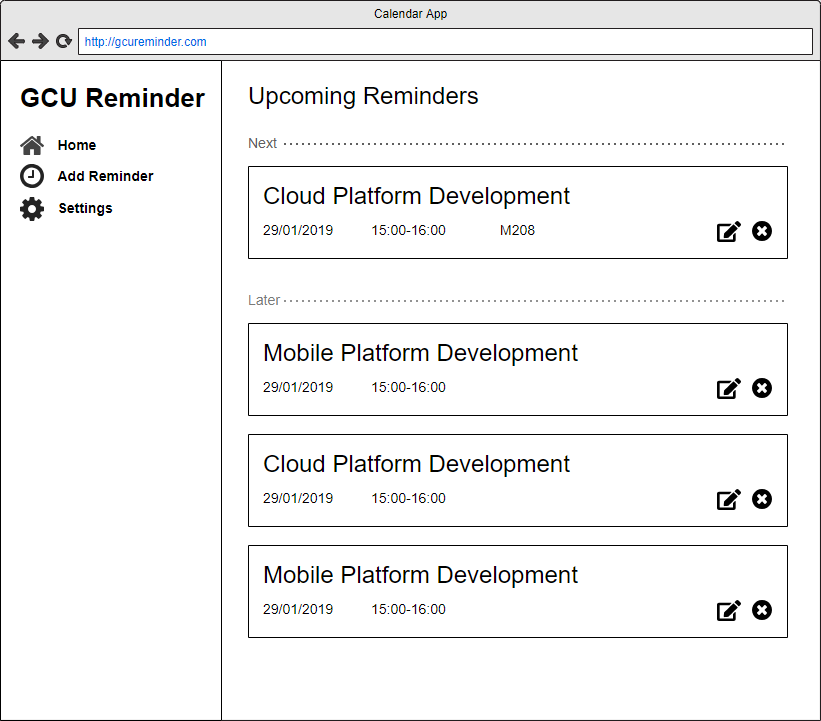


Figure 5 - Wireframe of index page

### Front-end vs back-end

The app requires both a frontend and a backend. The frontend will contain the Service Worker and is the part that will be evaluated. As Service Worker requires a very modern browser to work at all, this allows modern features of JavaScript to be used such as promises, async/await and others. The frontend will not use any libraries or modules except those we create. This means that we will control as much of the code to be tested as possible, meaning that issues arising from dependencies on third party code will not affect the outcome.

The backend will be created as an Express.js application for Node. The backend will serve up the initial resources used by the site, such as the HTML, CSS, and script files used by the shell of the application, as well as the Service Worker itself. After that it will operate as a type of RESTful API and exchange JSON data with the frontend. While the frontend has restrictions about the libraries we can use to aid testing, the backend has no such limitations, so libraries and modules can be used where possible to speed up development.

The following libraries have been evaluated and will be used to implement functionality on the backend.

* <https://mongoosejs.com/> - Mongo DB library for Node apps
* <https://github.com/peterbraden/ical.js> - Node library for reading ical calendar files
* <https://github.com/web-push-libs/web-push> - Node library for sending push notifications

Code will be stored in a GitHub repository. The app will be developed on Windows 10 using the Windows Subsystem for Linux (WSL) feature, which allows the bash shell to be used on Windows and works well when combined with Node.

## Implementation

The next stage of the report will look at the implementation of the research instrument. During this stage the design of the application was turned into working source code. The report will walk through each requirement of the project and describe how each was implemented in turn. The full source can be found in the separate document Appendix B – Code Listings.

### Shell and Cache API

The first requirement implemented was to cache the shell of the application using Cache API. This involved serving up pages from the backend Express.js application, and then using the Service Worker to cache them for access later. Each page is served up separately, rather than using a single page application, as this allows the cache process to be tested more thoroughly. Additionally, it allows the code for different features of the site to kept separate more easily.

The shell comprises four pages:

* index – list of upcoming reminders
* reminder – for adding or editing a reminder
* settings – changing location, distance, calendar import
* login – basic username/password authentication

The pages were implemented using the Pug.js template language which allows the layout for the pages to be expressed using a simple and concise syntax. The site uses unobtrusive JavaScript throughout, so no script code or event handlers are included in the layouts. The pages all share a parent layout template defined with Pug, which defines shared elements such as the header, footer and navigation, as well as style and script imports.

At this point the general structure of the application was created, which comprises both files on the front and back end of the site. The backend Express application consists of the app.js file which initialises the web application, plus several routes which handle the GET and POST requests from the client. Several model classes were created in the ‘models’ folder, that represent users and reminders in the application. All the frontend code was placed in the public folder, which includes the sw.js file that contains the Service Worker, the website manifest containing meta data about the site, and stylesheets and JavaScript files in their respective folders. Each JavaScript module follows the revealing module pattern, which allows inner workings to be hidden using a type of encapsulation like that found in traditional OOP languages.

The cache was implemented in the Service Worker file itself. The scope of the worker defines where it can intercept network requests from, so it must be in the same folder or a parent folder of your app’s JavaScript files. The Service Worker needs to be registered on each page which uses it, in this case all pages of the shell. A utility function called initServiceWorker() was added to the public/javascripts/app.js script, which registers the Service Worker and wraps the result in a promise. The code in Figure 6 shows the function that registers the Service Worker for the app. The code checks to see if the serviceWorker property exists in the navigator object (e.g. if the browser supports it), and if so then registers the sw.js file with the browser.

function initServiceWorker() {

return new Promise((resolve, reject) => {

if ('serviceWorker' in navigator) {

navigator.serviceWorker.register('/sw.js')

.then(resolve)

.catch(reject);

} else {

reject('Service worker not supported');

}

});

}

Figure 6 - Service worker registration code wrapped in a promise

The caching process is then initiated from the ‘install’ event in the Service Worker, which is triggered when the Service Worker is first installed on the browser. A cache with the specified name is opened (or created if it doesn’t exist) and then a list of URLs is added. This operation is wrapped inside of an event.waitUntil() call, which is important as it tells the browser not to terminate the Service Worker script until the operation has finished (MDN, 2018). The code for this is shown in Figure 7.

const CACHE\_NAME = 'gcu-reminder-v7';

const URLS\_TO\_CACHE = [

'/',

'/reminder',

'/settings',

'/stylesheets/style.css',

'/stylesheets/bootstrap.min.css',

'/javascripts/data-store.js',

'/javascripts/index.js',

'/javascripts/repository.js',

// Many others...

];

self.addEventListener('install', event => {

event.waitUntil(caches.open(CACHE\_NAME).then(cache => {

return cache.addAll(URLS\_TO\_CACHE);

}));

});

Figure 7 - Code to cache list of URLs in cache with specified name

Next, we need to handle requests for data, which is done by attaching an event listener to the Service Worker ‘fetch’ event, as shown in Figure 8. This event is raised whenever a script within the scope of the Service Worker makes a request using the Fetch API. We check the cache for a response matching the fetch request, returning the response if it exists. If the response is null, we perform a fetch operation to retrieve a fresh one. We could cache the result of this fetch operation too, but as the shell of our application is entirely cached on installation this is not needed.

self.addEventListener('fetch', event => {

event.respondWith(caches.match(event.request).then(response => {

if (response) {

return response;

} else {

return fetch(event.request);

}

}));

});

Figure 8 - Fetch event listener which responds with cached data if it exists

An issue with the caching process is that every time a new file is added or removed from the site it must be also added or removed from the cache, which is easy to forget to do. Also, the cache name must be changed to force the browser to recreate it and redownload the files. If a single request to cache a file on install fails, then all the files in the list will not be cached. The error message does not notify you as to which file it was that caused the failure, meaning it can be frustrating to track down the problem. Despite these issues the Service Worker cache was easy to implement.

### Live data, Fetch API and Indexed DB

While the shell of the application is always cached, the content of the site is fetched from the server as JSON using AJAX. This is handled using the Fetch API. Data then needs to be cached on the browser using Indexed DB, which is required to access data from inside a Service Worker. Indexed DB is not a particularly nice API to work with, it’s verbose and unintuitive, and makes simple operations such as inserting or retrieving data complicated. Additionally, while it is completely asynchronous, it does not support JavaScript promises. Therefore, a module named data-store.js was added to abstract away these problems, and provide a simple way to add, update, delete, and query data. The module also features a bunch of convenience methods for interacting with reminders and users.

Indexed DB storage was used for three object stores:

* users – for storing the single user object
* reminders – for storing reminder objects
* sync-queue – used for the background sync described later in this document

The code in Figure 9 shows an example of working with Indexed DB, which returns all the documents in the specified collection. All this sort of boilerplate is required for every operation, which made working with the API difficult.

function getCollection(name) {

return new Promise((resolve, reject) => {

if (db == null) {

reject('DB not initialized!');

} else {

const transaction = db.transaction([name], 'readonly');

transaction.onerror = event => {

reject('DB transaction error: ' + transaction.error);

};

const store = transaction.objectStore(name);

const documents = [];

store.openCursor().onsuccess = event => {

const cursor = event.target.result;

if (cursor) {

documents.push(cursor.value);

cursor.continue();

} else {

resolve(documents);

}

};

}

});

}

Figure 9 – Gnarly code needed to get a collection from Indexed DB.

On the server, data is stored in Mongo DB and accessed using the Mongoose.js library, which provides a cuddly interface around the service. The server accepts requests from the client, queries the Mongo DB data store, and then returns it as JSON.

The site uses a simple authentication system. When each page is loaded it checks locally to see if a user exists in the object store, and if not redirects the user to the login page. When the user submits the login form the server authenticates their details against the Mongo database. If the username is found to exist then the password is checked, if no user with that name exists then a new user is created. Once successfully authenticated a JSON object for the user is sent to the browser including an authentication token. This token is newly generated each time the user logs in and must be included with every subsequent request which is made to the server. There can be multiple authentication tokens for each user as they might be logged in on multiple devices. The login system is quite barebones, there’s no way to change your password for example, but it was decided not to bog down the project with creating a complex authentication system that would not contribute towards the research topic.

When data is needed the client uses a network first caching strategy. That means that data is requested across the network using the fetch API, and if it fails then the app falls back to using cached data from Indexed DB. If the network is detected to be offline, for instance using the navigator.onLine built-in property, then the network operation is skipped, and cached data is used. Whenever a request is fulfilled the data in Indexed DB is updated to keep it current. The caching was handled this way so that the data displayed to the user is kept as up-to-date as possible with what’s stored on the server. The code to handle fetching and caching data can be found in the public/javascripts/repository.js module. Example code used to download the data or fall back to cache can be found in Figure 10.

function getReminders(user) {

if (navigator.onLine) {

// try fetch reminders

const promise = fetchJson('/api/reminders/list/' + user.token);

return promise.then(reminders => {

// add reminders to cache

dataStore.setReminders(reminders);

return reminders;

}).catch(() => {

// if fetch fails use cache

return dataStore.getReminders(user.\_id);

});

} else {

// always use cache if offline

return dataStore.getReminders(user.\_id);

}

}

Figure 10 - Code to fetch data from network or fall back to cache in repository.js

The code checks to see if the browser is online, if not it falls back to returning cached data from the data-store immediately. When online a fetch operation is performed, including the authentication token, and if successful the newly found reminders are added to the data-store and then returned. If the fetch operation fails, either through timeout or error, then cached data is returned.

### Background sync

One key feature of the design of the web app, and of Service Worker in general, is the ability to cache data in the background when the browser is not connected to the internet, and then synchronise it when the network becomes available again. This is done through the Service Worker ‘sync’ event. When a sync is needed, for instance when the user adds a reminder, the page requests a sync event by registering a simple string with the SyncManager called a ‘tag’. When the browser is next ready to sync the event is raised including the registered tag. This is pretty much all the functionality that the sync feature gives you and everything else is left up to you. Unlike the Cache API the sync API seems underdeveloped, which may be down to the lack of browser support. As of now the latest versions of Chrome and Opera are the only browsers to implement this functionality (caniuse.com, 2019). Therefore, the background storing as well as the rest of the sync mechanism had to be implemented manually.

To do this we used Indexed DB to store a list of queued items. When the sync request is received we retrieve the items from the queue and then attempt to post each one. If successful, the item is removed from the queue and a confirmation message shown. This process can be seen in the diagram in Figure 11.

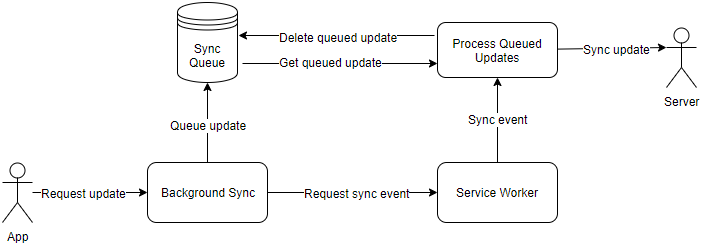


Figure 11 - Background sync process

A module called background-sync.js encapsulates the sync logic. When a request to sync is made a new document is added to the dataStore module including several fields:

* token – the authentication token
* data – the JSON payload to be sent to the server.
* url – the URL to send the request to
* message – the message to display on success

Next a sync request is registered with the Service Worker with the tag ‘background-sync’. If no sync manager is detected on the browser the queue is skipped and a normal fetch attempt is made. Example code for this feature is shown in Figure 12.

async function queue(token, data, url, message) {

const registration = await navigator.serviceWorker.ready;

if ('sync' in registration) {

await dataStore.addSyncItem(token, data, url, message);

await registration.sync.register('background-sync');

} else {

// Sync not supported so just try to send normally.

await postJsonItem(token, data, url);

}

}

function postJsonItem(token, data, url) {

return fetch(url, {

method: 'post',

headers: { "Content-Type": "application/json" },

body: JSON.stringify({

token: token,

data: data

})

});

}

Figure 12 – Code from background-sync.js to add an item to the sync queue

Once a sync event is received by the Service Worker all queued items are retrieved from the data store and an attempt is made to sync each one. If the sync request is successful, then the item is removed from the queue. If not, it remains in the queue to be synced again on the next attempt. A list of successfully synced items is returned to the Service Worker, which then sends a message to the web page to confirm the sync has completed. If multiple syncs are completed then a generic message is sent, otherwise a specific message retrieved from the queued item is used. The message is sent to all connected clients, as a single Service Worker may be handling syncs for multiple browser windows. Example code for handling a sync event can be seen in Figure 13.

// gets sync message to send

function getSyncMessage(messages) {

if (messages.length > 1) {

return 'Updates synced with server';

} else if (messages.length === 1) {

return messages[0];

} else {

return 'No message';

}

}

// sends message to all connection browser clients

function sendMessageToAll(message) {

return self.clients.matchAll().then(clients => {

clients.forEach(client => {

client.postMessage({ message: message });

});

});

}

// handle sync event

self.addEventListener('sync', event => {

// check for our tag

if (event.tag === 'background-sync') {

event.waitUntil(sync().then(syncedItems => {

const messages = syncedItems.map(item => item.message);

const message = getSyncMessage(messages);

return sendMessageToAll(message);

}).catch(console.log));

}

});

async function sync() {

await dataStore.init();

const items = await dataStore.getSyncItems();

const promises = items.map(async item => {

const response = await postJsonItem(item.token, item.data, item.url);

if (response.ok) {

await dataStore.deleteSyncItem(item.id);

return item;

} else {

return null;

}

});

const syncedItems = await Promise.all(promises);

return syncedItems.filter(item => item != null);

}

Figure 13 - Example code in sw.js to handle sync event

While all updates to the server are handled through background sync mechanism, one feature was not able to be included, and that was the file upload. A requirement was to be able to import reminders by uploading an ical calendar file. While it was possible to upload this file to the server as multipart/form data in the normal manner, this could not be achieved using background sync. Background sync requires that data is stored in Indexed DB, which is not possible with file data. For security reasons file data is hidden from JavaScript in the browser. All that could be done is to store the name of the file to be synced, which is not very useful. While the import feature was implemented, the intended background sync part was not.

The calendar file is uploaded with AJAX using the browser FormData object, which is included in the body in a POST request. The usual Fetch API can be used to handle this, however a Content-Type should not be included, as the correct content-type is appended to the request by the browser and any attempt to add your own only interferes. On the server-side the Formidable.js form processing module is used to aid in uploading the form data to a temporary file, and then the node-ical library is used to parse the calendar file contents. The resulting list of reminders is inserted into the database and then sent back down to the client as JSON, to be cached in Indexed DB.

The background sync functionality seems like a headline feature of Service Worker, but in practice it is quite barebones and not well supported. This seems like an area of the Service Worker that needs to be improved to gain the full benefit from it in the future.

### Geofence

To meet the requirement of handling a long running process the idea of using geofencing to affect the timing of push notifications was developed. When an upcoming reminder was detected by the server a notification would be sent an hour before the reminder was due. However, if the student was on university grounds an additional notification would be sent five minutes before. Despite this feature seeming possible during the research phase of the project, due to its inclusion in much of the documentation for the Service Worker API, not to mention the spec, it turned out it was not possible to implement in the fashion intended. Long running processes outside of the lifecycle of a single page are not possible with Service Worker in way the literature suggested. The reasons for this, and implications, will be dealt with during the Testing & Evaluation portion of this report.

That said, the feature was implemented in the app outside of the Service Worker. A request for location updates is made to the geolocation service, which updates the location when the user’s device is moved. Then to determine if the student is on university grounds the user’s distance from the longitude and latitude stored in the settings was checked. If the user is within the specified distance of the latitude/longitude, a flag is set for that user indicating this and synced with the backend server. Then, when sending push notifications, the server can check the flag to determine what type of push notification to send, one hour or five minutes. This process was outlined during the design section way back in Figure 4.

The distance is determined using the Haversine formula, which can determine the distance between two longitudes and latitudes. The result is not hugely accurate over long distances – as it does not account for the curvature of the earth – but is efficient and more than suitable for our purposes. Some example code to handle location requests is shown in Figure 14.

function getDistanceFromLatLonInMetres(lat1, lon1, lat2, lon2) {

function deg2rad(deg) {

return deg \* (Math.PI / 180)

}

const R = 6371; // Radius of the earth in km

const dLat = deg2rad(lat2 - lat1);

const dLon = deg2rad(lon2 - lon1);

const a = Math.sin(dLat / 2) \* Math.sin(dLat / 2) + Math.cos(deg2rad(lat1)) \* Math.cos(deg2rad(lat2)) \* Math.sin(dLon / 2) \* Math.sin(dLon / 2);

const c = 2 \* Math.atan2(Math.sqrt(a), Math.sqrt(1 - a));

const d = R \* c; // Distance in km

return d \* 1000; // Convert to metres

}

async function locationUpdate(pos) {

const lat = pos.coords.latitude;

const lon = pos.coords.longitude;

// Get distance using haversine formula.

const distance = getDistanceFromLatLonInMetres(lat, lon, user.latitude, user.longitude);

// Check if location has changed

const atLocation = user.distance > distance;

if (currentUser.atLocation !== atLocation) {

// update local storage and sync with server

await repository.editAtLocationu.token, atLocation);

}

}

function startLocationUpdates() {

// subscribe to location updates

navigator.geolocation.watchPosition(locationUpdate, locationError, {

enableHighAccuracy: false,

timeout: 5000,

maximumAge: 0

});

}

Figure 14 - Geofence code showing watching location updates and updating repository

This method of implementing the requirement has some downsides, mainly that the location checks will only be made if the user is actively viewing the web app, and not in our Service Worker as desired.

### Scheduler and push notifications

A key feature of the Service Worker is the introduction of push notifications, which allow a notification to be sent from the web server to be displayed on a client. This requires code to be implemented both on the server and client sides of the application.

For the server we used the web-push library, which abstracts away some of the complexity of dealing with push notifications in a Node application. As we are only evaluating the client-side of the notifications, this was a valid way to save time during development. Sending notifications requires that VAPID public and private keys are set in the web-push library, which can be easily generated by calling the webpush.generateVAPIDKeys() method in the Node console. Once generated these keys are stored in environment variables to prevent having to include them in the source code, which would be a security risk. VAPID (Voluntary Application Server Identification for Web Push) is a protocol that allows clients to identify themselves to a push notification server (IETF, 2017).

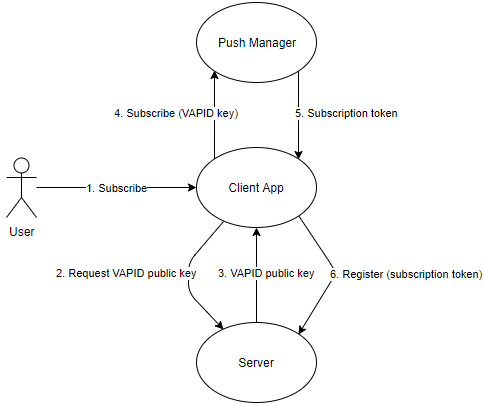


Figure 15 - Register a push notification with the server

There are several steps required when subscribing to push notifications from a browser, which are illustrated in Figure 15. The process is as follows:

1. The user gives permission to subscribe to a push notification
2. A fetch request is made to the server for its VAPID public key
3. The VAPID public key is returned to the client
4. A subscription request is made to the push manager providing the public key
5. The push manager returns a subscription token representing the endpoint of the current device
6. The subscription token is then registered with the server and stored in the data store, where it can be used to send notifications to the device

Next, when we want to send a push notification from the server to the client, we have several different steps, as seen in Figure 16.

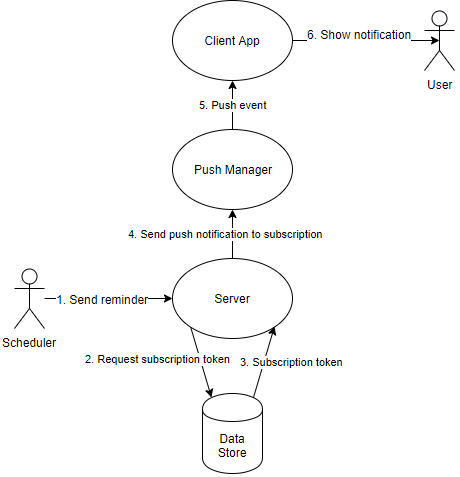


Figure 16 - Send a push notification to the client

A breakdown of the diagram is as follows:

1. The scheduler decides to send a push notification for an upcoming reminder
2. The server app requests the subscription data from the data store
3. The data store returns the subscription data to the app
4. A push notification is sent to the device represented by the subscription endpoint
5. A ‘push’ event is raised by the push manager in the Service Worker script
6. A notification is then shown on the user’s device

An important part of the application was how to detect that a reminder is upcoming so that a notification could be sent. The actual code to do this is simple but the check itself needs to be triggered periodically. The simplest solution was to create a cron job on the server that runs once a minute and calls a script that checks for reminders. The Unix cron command (short from chronology) allows a command to be run at intervals, so we specify that the command should be run every minute between the hours of 8am and 6pm.

The initial plan was to have this script be a small standalone Node app, that would query the data store for reminders, but this idea was quickly quashed. Instead we call a route on the web app api/reminders/check which allows us to reuse the existing mongo DB database connection and have access to the same app context and methods. Additionally, a standalone script might try to access the database concurrently with the web app, which could cause multithreading issues. As Node is single-threaded that is not a problem so long as you are always accessing the database from inside the same application instance.

A simple query is then used to find all reminders that have received neither long and short notifications previously and are due in the next hour. We can then send a notification to the user for that reminder, including our own custom payload, as shown in Figure 17. In this instance we include the type of the notification, as well as the ID of the reminder to show to the user.

function sendNotification(reminder) {

return notifications.send(reminder.userId, {

type: 'reminder',

id: reminder.id

});

}

Figure 17 - Code to send push notification using web-push library

The send function of the notification’s module wraps the web-push send functionality, as shown Figure 18. We retrieve the user for that specific reminder, and then loop through their subscribed devices and send a push notification to each one. If an error occurs sending the notification, then we remove that subscription from the current list.

function send(userId, payload) {

payload = JSON.stringify(payload);

return db.getUserFromId(userId).then(user => {

if (user) {

const promises = user.subscriptions.map(subscriptionJson => {

const sub = JSON.parse(subscriptionJson);

return webPush.sendNotification(sub, payload).catch(err => {

return removeSubscription(user, subscription);

});

});

return Promise.all(promises);

} else {

return Promise.reject('Could not find user for push notification');

}

});

}

Figure 18 - Code to send push notification to user's subscriptions

Next the push notification is received by the device and in the Service Worker script a push event is raised, containing the notification to show as in Figure 19. The JSON payload is unpacked, the reminder retrieved from the data store, then the reminder message shown to the user with the self.registration.showNotification() function. This displays the notification to the user in a device-dependent way.

self.addEventListener('push', event => {

const data = event.data.json();

// check notification type

if (data.type === 'reminder') {

// init data store

event.waitUntil(dataStore.init().then(() => {

// get reminder from ds

return dataStore.getReminder(data.id);

}).then(reminder => {

// get reminder text then show notification in device

const text = getReminderText(reminder);

return self.registration.showNotification(reminder.title, {

body: text

});

}).catch(console.log));

} else {

console.log('Unknown notification type: ' + data.type);

}

});

Figure 19 - Push event handler in the sw.js file

It’s useful to know that the browser can use showNotification() to show a notification at any time, not just in response to a push notification.

### Deployment

The site was deployed to a web host running Ubuntu 18.04 LTS and is running on nginx/1.14.0 web server. The site can be found at the URL: <https://gcureminder.co.uk/>.

# Testing & Evaluation

The final phase of execution is testing, where we will carry out tests for each feature of the Service Worker. These tests will determine how well the application functions both online and offline, and especially with an intermittent network connection. Testing online and offline can easily be done using the Chrome Developer Tools, but to simulate poor network connections we will use a tool called Clumsy.

## Caching shell

Testing the Cache API involved comparing the loading of the shell with differing levels of caching. Firstly, we tested with no cache at all, secondly when installing the Service Worker and creating the cache for the first time, and thirdly with the cache already installed. Each test was conducted ten times and the average latency and amount of data transferred recorded.

The tests were conducted using the ‘Bypass for network’ option in the Chrome DevTools Application tab, which causes all requests to skip the Service Worker, and by checking the option ‘Disable cache (while DevTools open)’ in the Chrome settings preferences, which prevents the browser cache from being used. Before each test the Clear Storage button in the Application tab was pressed to clear any existing Service Worker cache. The results of these tests can be seen in Figure 20.

Figure 20 - The latency and data transferred with and without the cache

The tests show that without a cache on average 192kb was transferred in 294.9ms. When first creating the cache an average of 442kb was transferred in 316.6ms. Finally, once the cache was installed and the shell refreshed, no data was transferred to the browser and loading the page took on average 88.1ms. The full results of these tests can be found in Appendix – Cache API Tests.

These results illustrate that despite an initial trade-off when the cache is first created, the overall savings are quite stark. In our literature review we determined that 100ms was the ideal time for responsiveness, and when using the cache we can achieve a latency of just 88ms, well within that range. This also confirms our earlier assumption that the cache is much faster than using the network.

Considering how easy the Cache API is to implement, this feature results in an easy win for developers and lets them massively reduce the amount of data needed to be transferred each request. Of course, these tests don’t take into consideration the existing browser cache, however an advantage of Cache API is that it lets developers tightly control what gets cached.

## Fetching and caching data

Next, we evaluate loading data using the Fetch API, by downloading the list of reminders in JSON format and caching them using Indexed DB. Our app uses a Network first strategy where it tries to fetch data from the server and when that fails it falls back to using the cache. We can test easily when the network is either on and off by toggling it using the Offline checkbox in the Chrome DevTools. However, to test with an intermittent connection we use the Clumsy tool with the packet drop rate set in intervals between 0% and 100%, going up in steps of 20%. Each test was repeated ten times and the average latency recorded, along with whether the request was eventually fulfilled.

Figure 21 - Average latency for Fetch requests (note logarithmic scale)

In the latency tests in Figure 21 we can see that with the browser offline and using the cache, the time to fetch the reminders is only 7.49ms, which is a very low time as we would expect. When online and with no packet interference we can fetch the data in an average of 145.41ms. Next, we start to simulate poor network conditions, and at 20% drop chance there is not much difference from 0%, however at 40% we start to see an increase to 370.93ms, and then at 60% and 80% we start to see average times of over a minute. Lastly, at 100% we see the time drop down to 8.6 seconds, which likely is because with no packets at all getting through the browser can determine that the request will fail more quickly.

We are not just interested in how long the request took to complete, but also that the request was fulfilled, which is shown in Figure 22. This diagram shows that of our ten tests requests are fulfilled successfully until we hit 60%. At 60% one out of ten requests failed, at 80% six failed, and at 100% drop rate no requests successfully got through.

Figure 22 - Number of times the request was fulfilled in different network conditions

The response latency is critical as it dictates at what point we fall back to the cache, and while we can detect that easily when the browser is either completely online or off, with an intermittent connection it becomes increasingly difficult, with sometimes up to a five-minute wait before determining if the request would fail. This is way outside of our maximum 1-minute rule for the completion of tasks determined during the literature review.

Therefore, there are some other alternatives the developer can use, such as implementing their own timeout using the built-in JavaScript functions, or by applying a different network strategy such as displaying from the cache first, and then updating the UI when network data is finally received. A key thing to note is that even with a high drop rate such as 80% most requests were fulfilled, albeit eventually in some cases, meaning that even though the responsiveness of the app becomes very poor the data is still reliable. The full results of these tests can be found in Appendix – Fetch Tests.

## Background sync

Testing the background sync is like testing the fetch code in reverse, as here we are interested in POST operations rather than GET. This is done by saving a reminder in the browser UI and then recording the time taken for the request to be completed. However, with background sync we are not as interested in the response time as we are in whether request was completed at all. This is because as we are queued and syncing later we know there is going to be some sort of delay so adding a few seconds to it won’t make a huge difference. Additionally, if a request does fail the update is held in the sync queue and another attempt to sync it made later. That said, we did also record the latency for completeness.

Again, we used the Clumsy tool to simulate an increasingly intermittent connection by raising the drop percent chance for each packet in increments of 20%. Each test was completed ten times and the average latency of each request recorded, along with whether the sync operation completed successfully or not. The results can be seen in Figure 23. As we can see between 0% and 60% all the operations complete, however this decreases to 8 at 80%, and then finally 0 at 100%.

Figure 23 - Sync success counts with increasingly poor network

In Figure 24 we can see the average latency, which increases gradually from 45.65ms at 0% (perfect network conditions), to almost a second at 40%, and then three seconds at 60%. From here we start to see a big jump, with the latency going to over a minute at both 80% and 100%.

Figure 24 - Average latency for a sync operation to complete at increasing drop rates

With good network conditions we were able to get a latency below our preferred 100ms, rising to one second at 40%. However, as the network gets worse so do the response times. Again, we are not interested in the latency and even in extremely poor network conditions with 80% of packets being lost, we were still able to get eight of our ten syncs through.

Implementing the background sync took a lot of work, as the Service Worker API does not give you much to start with. Only the basic ‘sync’ event is provided and everything else must be implemented by the developer, including the queue and ever-awkward Indexed DB code. This increased the work needed to develop the feature a lot, but the results are convincing, and will allow web pages to more reliably update the backend server even in extremely poor network conditions. The full tests results are in Appendix – Background Sync Tests.

## Geofence

Next comes the test for the long running operation. Sadly, as explained in the implementation section of the report, this feature could not be implemented as intended. Long running operations like those suggested by research are not possible.

There are several reasons for this, one being that a long running operation is antithesis to some of the core values of the Service Worker, which is event based and only springs into life when needed. This prevents having scripts for many web pages running in the background and sapping device resources. There is also potential for privacy issues if any arbitrary app from the web could, say, start recording your location history and uploading it where it wanted. It’s not hard to see this is something that users would not want.

One thing to note is that mention of this feature is made both in official Google docs, as well as in the W3 spec for the Service Worker. However, specifications such as those from the W3C often lag behind actual implementation in the browser, and typically wait for features and technologies to be implemented before codifying them. It’s also possible that this feature, along with others in the Service Worker API, may change when being implemented in other non-Chrome browsers. As of now it is simply not a feature of Service Worker.

## Push Notifications

Finally, we come to push notifications, which are sent in response to upcoming reminders detected by the scheduler. We are interested that the push notification gets through, but also that it does so in a timely manner, as there’s no point getting a reminder for something after it’s happened.

We can test push notification in general by creating a reminder that’s due within the next hour and waiting for the notification to be sent. However, this proved a little slow for the number of notifications we need to receive to be able to build our data. Therefore, a special endpoint was added to the backend called api/notifications/test which allowed push notifications to be generated as needed. A timestamp was recorded when the notification was requested and again when the notification was received. We used the Clumsy tool to simulate poor network conditions and completed each test ten times, recording the difference between the timestamps (latency) and whether the push notification was completed.

Figure 25 - Push notification average latency at increasing drop rates

In Figure 25 we can see that with perfect network conditions we receive the notification within 41.24ms on average, increasing to 114.63ms at 20%, before falling to 97.76ms at 40%. This is likely just because with only a 40% likelihood to drop a packet it’s still possible for most of them to get through, which suggests maybe we should be running more tests. Next the latency starts to climb to 223.8ms at 60%, then a big jump to 6 seconds at 80%, and then 21 seconds at 100%, which seems to represent some sort of timeout.

Figure 26 shows whether the notification got through at all and we can see that from 0% to 60% all our notifications are received, while at 80% only six are received, and at 100% drop chance none are received. The full results can be found in Appendix – Push Notification Tests

Figure 26 - Number of push notifications that succeeded at increasing drop rates

When a push notification is sent it is passed along to the push service, which will hold onto it and keep trying to deliver it for up to four weeks by default. Therefore, even with a poor network connection the push notification will be received eventually. In our tests we are interested in the timeliness of the notification, as it is a non-functional requirement for our app that notifications are received within ten seconds. For the most part this is the case, even with a very intermittent network with up to 80% packet loss. However above 60% it becomes hit or miss as to whether the notification will be received at all, or if the system will timeout.

It must be noted that there are other issues to consider when receiving a notification. Firstly, it is the browser which receives the notification and whether it is active or not can have a baring. On the desktop the Chrome browser must be running, or its system-tray icon must be showing. If not, then the notification won’t be received until the next time the user starts the browser. On phone or tablet the browser does not need to be active, but if it’s not it can result in a delay of up to several minutes before a push notification is received. Lastly, there is no way to know if a push notification has been delivered, and or if the user has even looked at it. Therefore, they should not be used to send critical pieces of information.

# Conclusions

The aim of the project was to determine how Service Worker API can aid in the development of seamless user experiences when offline, or with an intermittent network connection. Latency is important as being able to fall back to the cache in a timely manner is key in making the user experience feel seamless.

Detecting when the browser is completely on or offline can be done relatively reliably, but as packet drop rates with intermittent connections reach in the region of 80% it can sometimes take several minutes for the failure to be detected. A minute’s delay in our application takes us way out the realms of seamlessness and into frustration. Currently there is no way to specify a timeout value for a fetch request, and the specification for the Fetch API does not stipulate a timeout. In general, timeouts take in the region of two and a half minutes, but in some rare cases up to five minutes. Therefore, a recommendation would be for the Fetch API to allow developers to specify a timeout. Additionally, more clarity from platform holders such as Google would be beneficial, as well as updating the Fetch API spec to specify exact timeout values for browser makers to aim for.

Currently the browser only provides a boolean value navigation.onLine indicating if the network is available, but no way to gauge what state it is in, or how likely a request is to be successfully completed. Therefore, a solution would be to include a feature that indicates not just the overall network state, but also the current quality of the network. Developers would then be able to write code that falls back to cache more quickly and improve the perception of responsiveness by the user. There is much work currently in predicting Quality of Service (QoS) in service-based architectures, for example using Machine Learning to predict QoS issues on networks (Xiong, et al., 2015), so it’s possible this work could be extended to include the detection of network quality within the browser.

It may be that the platform vendors such as Google and Mozilla do not have interest in expanding their work in this area, so a third-party library could be developed to add additional features to Service Worker. Already Google has the Workbox library that removes the need for a lot of the boilerplate code needed by Service Worker, but this could be expanded to add more features to cover some of the less well-developed functionality, such as background sync. Additionally, features to help the detection of network failure conditions could be included. There are proposals for libraries that inject Service Workers automatically into web pages and detect when various resources can be cached, or extra functionality can be added (Pande, et al., 2018), so perhaps in the future it will be possible to add increased functionality to a web page without the need for any extra code to be written at all.

Some network operations, such as uploading a file, could not be implemented as intended. A recommendation then would be that operations such as this could be added to the browser as first-class citizens, as part of the Service Worker API. However, this would require implementation by the vendor and would not be possible in a library.

Based on the findings of the study, further work in this area could involve research into the detection of poor network conditions in the browser, and how to present that information to the developer through the API.

# References

Archibald, J., 2014. *The offline cookbook.* [Online]   
Available at: https://jakearchibald.com/2014/offline-cookbook/#serving-suggestions-responding-to-requests  
[Accessed 03 04 2019].

Archibald, J., 2015. *Introducing Background Sync.* [Online]   
Available at: https://developers.google.com/web/updates/2015/12/background-sync  
[Accessed 11 10 2018].

Bidelman, E., 2013. *A Beginner's Guide to Using the Application Cache - HTML5 Rocks.* [Online]   
Available at: https://www.html5rocks.com/en/tutorials/appcache/beginner/  
[Accessed 01 11 2018].

Boyles, J. L., Smith, A. & Madden, M., 2012. *Privacy and Data Management on Mobile Devices.* Washington, DC: Pew Internet & American Life Project.

Butgereit, L., 2016. *Exploring NoSQLs in a Corporate IT Environment.* New York, NY, USA, ACM.

caniuse.com, 2018. *Service Workers.* [Online]   
Available at: https://caniuse.com/#feat=serviceworkers  
[Accessed 15 11 2018].

caniuse.com, 2019. *Background Sync API.* [Online]   
Available at: https://caniuse.com/#search=syncmanager  
[Accessed 19 3 2019].

Charland, A. & LeRoux, B., 2011. Mobile Application Development: Web vs. Native. *Queue,* 9(4), p. 20.

Chitra, L. P. & Satapathy, R., 2017. *Performance comparison and evaluation of Node.js and traditional web server (IIS).* Chennai, India, IEEE.

Clumsy, n/a. *About Clumsy.* [Online]   
Available at: http://jagt.github.io/clumsy/index.html  
[Accessed 5 11 2018].

Cudini, S., Murzilli, S. & Barbaglia, G., Jun 2017. Definition of REST web services with JSON schema. *Software: Practice & Experience; Bognor Regis,* 47(6), pp. 907-920.

Elchuk, K., 2016. JavaScript: Web page based apps. *Linux Format; Bath*, Dec, pp. 88-91.

Fox, R., 2012. Being responsive. *OCLC Systems & Services; Bradford,* 28(3), pp. 119-125.

Google, 2018. *Debugging Service Workers.* [Online]   
Available at: https://codelabs.developers.google.com/codelabs/debugging-service-workers/#0  
[Accessed 15 10 2018].

Google, 2018. *Lighthouse | Tools for Web Developers.* [Online]   
Available at: https://developers.google.com/web/tools/lighthouse/  
[Accessed 15 10 2018].

Google, 2018. *Progressive Web App Checklist.* [Online]   
Available at: https://developers.google.com/web/progressive-web-apps/checklist  
[Accessed 15 10 2018].

Google, 2018. *Progressive Web Apps.* [Online]   
Available at: https://developers.google.com/web/progressive-web-apps/  
[Accessed 23 10 2018].

Gralla, P., 2015. How to choose the best mobile development environment. *ITworld.com; Framingham*, 28 Sep.

IETF, 2017. *Voluntary Application Server Identification (VAPID) for Web Push.* [Online]   
Available at: https://datatracker.ietf.org/doc/rfc8292/  
[Accessed 20 03 2019].

Krill, P., 2015. Google pushes new scheme for native-like Web apps. *JavaWorld; San Francisco*, 23 Dec.

Malavolta, I., 2017. *Are service workers draining my battery?.* [Online]   
Available at: http://www.ivanomalavolta.com/are-service-workers-draining-my-battery/  
[Accessed 1 11 2018].

MDN, 2018. *Cache.* [Online]   
Available at: https://developer.mozilla.org/en-US/docs/Web/API/Cache  
[Accessed 10 10 2018].

MDN, 2018. *ExtendableEvent.waitUntil().* [Online]   
Available at: https://developer.mozilla.org/en-US/docs/Web/API/ExtendableEvent/waitUntil  
[Accessed 19 03 2019].

MDN, 2018. *Fetch API.* [Online]   
Available at: https://developer.mozilla.org/en-US/docs/Web/API/Fetch\_API  
[Accessed 10 10 2018].

MDN, 2018. *Push API.* [Online]   
Available at: https://developer.mozilla.org/en-US/docs/Web/API/Push\_API  
[Accessed 11 10 2018].

MDN, 2018. *Using Promises.* [Online]   
Available at: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Using\_promises  
[Accessed 23 10 2018].

Mozilla, 2018. *Service worker concepts and usage.* [Online]   
Available at: https://developer.mozilla.org/en-US/docs/Web/API/Service\_Worker\_API  
[Accessed 1 10 2018].

MSDN, 2018. *Progressive Web Apps on Windows.* [Online]   
Available at: https://docs.microsoft.com/en-us/microsoft-edge/progressive-web-apps  
[Accessed 15 10 2018].

Nielsen, J., 2009. *Powers of 10: Time Scales in User Experience.* [Online]   
Available at: https://www.nngroup.com/articles/powers-of-10-time-scales-in-ux/  
[Accessed 03 04 2019].

Pande, N., Somani, A., Samal, S. P. & Kakkirala, V., 2018. *Enhanced Web Application and Browsing Performance through Service-Worker Infusion Framework.* San Francisco, CA, USA, IEEE.

Peischl, B., Ferk, M. & Holzinger, A., 2015. The fine art of user-centered software development. *Software Quality Journal; Dordrecht,* 23(3), pp. 509-536.

PePage, P., 2018. *Your First Progressive Web App - Architect your App Shell.* [Online]   
Available at: https://developers.google.com/web/fundamentals/codelabs/your-first-pwapp/#architect\_your\_app\_shell  
[Accessed 3 11 2018].

Puder, A., Tillmann, N. & Moskal, M., 2014. *Exposing native device APIs to web apps.* Hyderabad, India, ACM.

Scott, J., 2016. Streamlining Business With a NoSQL Document Database. *Big Data Quarterly; Medford,* 2(1), p. 33.

Sharma, A., 2017. Why Progressive Web Apps Have the Potential to Replace Mobile Apps. *PCQuest; Gurgaon*, 5 April.

Smith, M., 2012. HTML5 Development - Polyfills. *InformationWeek; Manhasset,* Issue 1334, pp. 32-33,36-37.

Sommerville, I., 2016. Requirements engineering. In: M. Horton, ed. *Software Engineering (10th Edition).* s.l.:Pearson, pp. 105-130.

Sommerville, I., 2016. System modeling. In: M. Horton, ed. *Software Engineerins (10th Edition).* s.l.:Pearson, pp. 138-166.

Ullah, S. et al., 2014. *Passive packet loss detection in Wi-Fi networks and its effect on HTTP traffic characteristics.* Phuket, Thailand, IEEE.

W3, 2017. *Service Workers 1 - Motivations.* [Online]   
Available at: https://www.w3.org/TR/service-workers-1/#motivations  
[Accessed 11 10 2018].

Wang, J. J. & He, Y., 2014. The Offline Storage Technology Based on HTML5. *Applied Mechanics and Materials,* 687-691(Manufacturing Technology, Electronics, Computer and Information Technology Applications), pp. 2104-2107.

Warren, I. et al., 2014. Push Notification Mechanisms for Pervasive Smartphone Applications. *IEEE Pervasive Computing,* 13(2), pp. 61-71.

Williams, C., 2016. *How one developer just broke Node, Babel and thousands of projects in 11 lines of JavaScript.* [Online]   
Available at: https://www.theregister.co.uk/2016/03/23/npm\_left\_pad\_chaos/  
[Accessed 3 11 2018].

Xiong, L., Yixuan, L., Ruixing, L. & Yi, C., 2015. Web Service QoS Prediction Based on Adaptive Dynamic Programming Using Fuzzy Neural Networks for Cloud Services. *IEEE Access,* Volume 3, pp. 2260-2269.

Young-Hyun, E., Young-Keun, C., Cho, S. & Jeon, B., 2016. Design and Implementation of a Framework of Three-Dimensional Geofence. *International Information Institute (Tokyo). Information; Koganei,* 19(9A), pp. 3895-3900.

# Appendix A

## Appendix – Wireframes

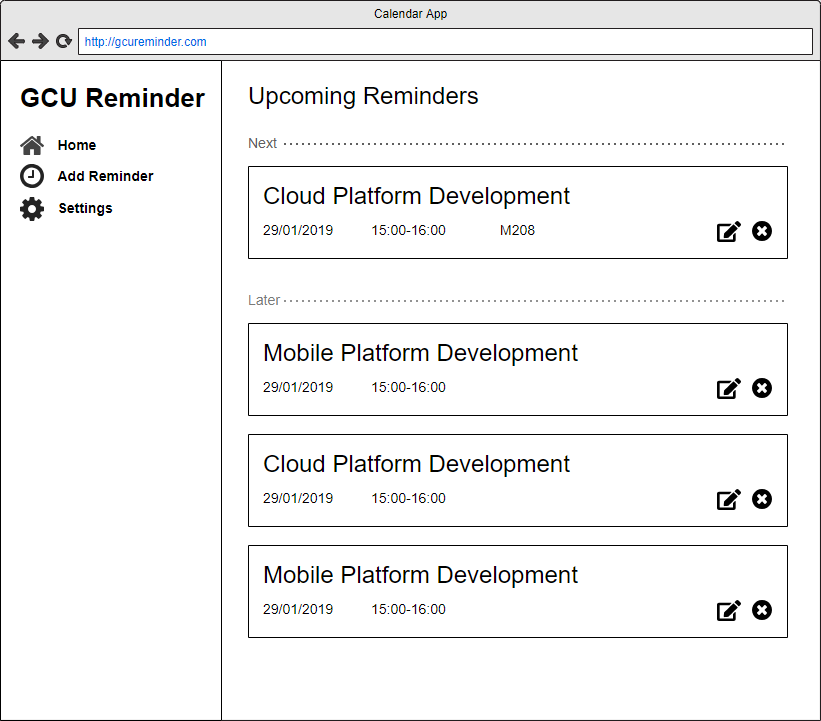
The wireframes created during the design phase.

Figure 29 - The index page wireframe

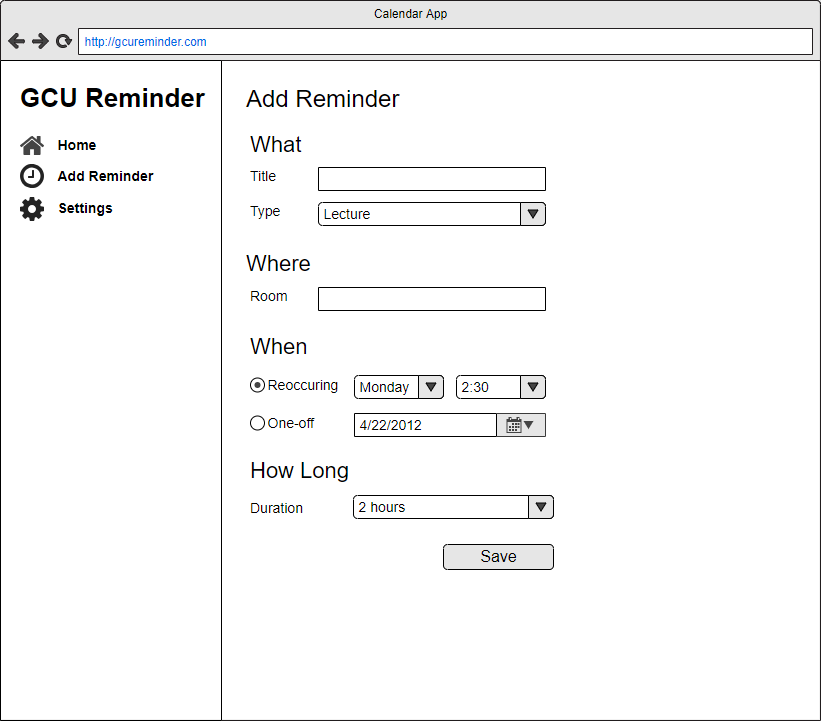


Figure 30 - The add reminder page wireframe

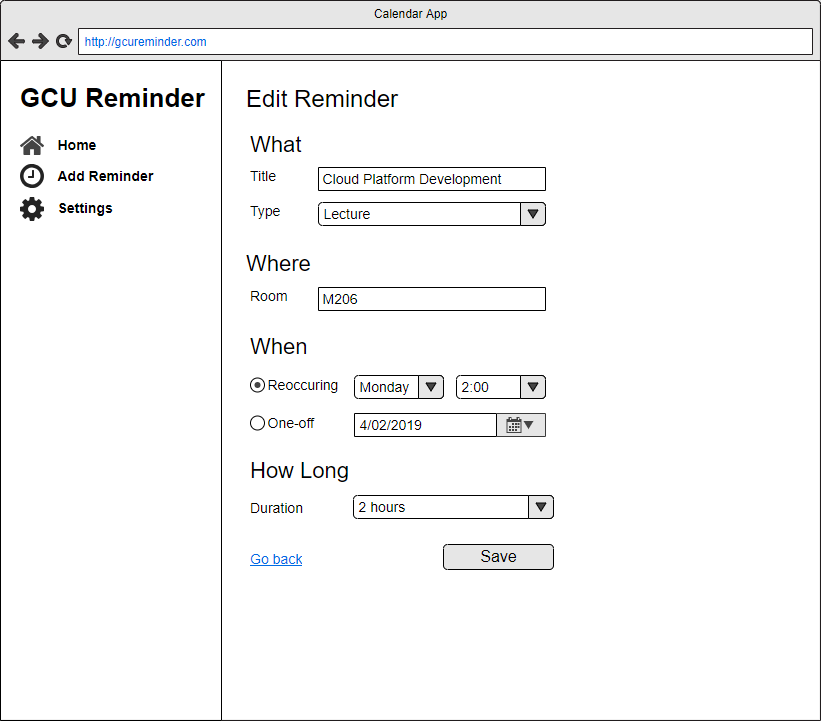


Figure 31 - The edit reminder page wireframe

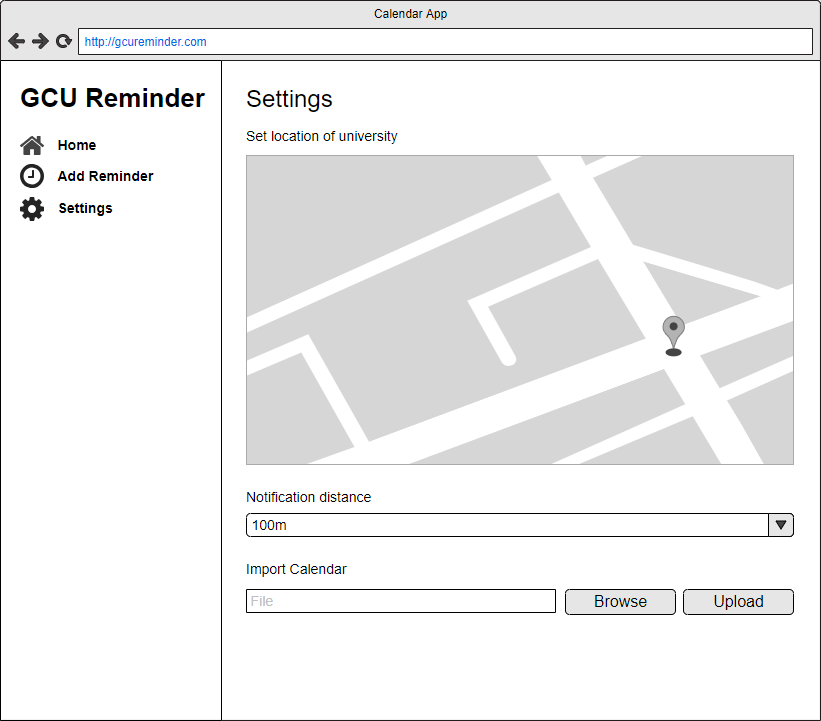


Figure 32 - The settings page wireframe

## Appendix – Test Results

This section of the appendix has full results from each test.

### Appendix – Cache API Tests

Testing loading of shell with no cache, performing test ten times.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Requests Count** | **KB Transferred** | **KB Resources** | **Finish (ms)** | **DOM** | **Load (ms)** |
| 15 | 192 | 189 | 355 | 223 | 227 |
| 15 | 192 | 189 | 341 | 224 | 248 |
| 15 | 192 | 189 | 300 | 203 | 207 |
| 15 | 192 | 189 | 288 | 225 | 229 |
| 15 | 192 | 189 | 316 | 243 | 246 |
| 15 | 192 | 189 | 443 | 362 | 369 |
| 15 | 192 | 189 | 446 | 362 | 368 |
| 15 | 192 | 189 | 425 | 347 | 352 |
| 15 | 192 | 189 | 372 | 292 | 298 |
| 15 | 192 | 189 | 484 | 398 | 405 |

Testing shell when first installing Service Worker (ten times).

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Requests** | **KB Transferred** | **KB Resources** | **Finish (ms)** | **DOM** | **Load (ms)** |
| 49 | 442 | 453 | 841 | 259 | 265 |
| 49 | 442 | 453 | 872 | 381 | 388 |
| 49 | 442 | 453 | 1000 | 237 | 242 |
| 49 | 442 | 453 | 808 | 234 | 241 |
| 49 | 442 | 453 | 787 | 265 | 269 |
| 49 | 442 | 453 | 1040 | 389 | 394 |
| 49 | 442 | 453 | 1010 | 424 | 429 |
| 49 | 442 | 453 | 822 | 316 | 320 |
| 49 | 442 | 453 | 847 | 311 | 315 |
| 49 | 442 | 453 | 927 | 298 | 303 |

Testing shell with Service Worker cache being used (ten times).

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Requests** | **KB Transferred** | **KB Resources** | **Finish (ms)** | **DOM** | **Load (ms)** |
| 17 | 0 | 193 | 2080 | 81 | 86 |
| 17 | 0 | 193 | 1840 | 83 | 89 |
| 17 | 0 | 193 | 1840 | 76 | 80 |
| 17 | 0 | 193 | 1910 | 70 | 81 |
| 17 | 0 | 193 | 1670 | 86 | 92 |
| 17 | 0 | 193 | 2070 | 76 | 81 |
| 17 | 0 | 193 | 1710 | 123 | 128 |
| 17 | 0 | 193 | 2070 | 79 | 84 |
| 17 | 0 | 193 | 2090 | 71 | 78 |
| 17 | 0 | 193 | 2070 | 78 | 82 |

### Appendix – Fetch Tests

Full results from testing Fetch API. Requests were made using cache, and then network with increasing levels of dropped packets. Each test was completed ten times. The latency and whether the request completed successfully were recorded.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Cache** | | | **Network (0%)** | | | **Network (20%)** | | |
| Test # | Latency (ms) | Success | Test # | Latency (ms) | Success | Test # | Latency (ms) | Success |
| 1 | 7.155 | y | 1 | 152.595 | y | 1 | 194.195 | y |
| 2 | 9.075 | y | 2 | 134.840 | y | 2 | 110.850 | y |
| 3 | 6.185 | y | 3 | 108.510 | y | 3 | 116.155 | y |
| 4 | 9.345 | y | 4 | 132.375 | y | 4 | 100.460 | y |
| 5 | 4.710 | y | 5 | 176.725 | y | 5 | 133.650 | y |
| 6 | 8.110 | y | 6 | 108.6499 | y | 6 | 202.635 | y |
| 7 | 8.145 | y | 7 | 184.7351 | y | 7 | 108.605 | y |
| 8 | 8.865 | y | 8 | 159.9602 | y | 8 | 113.655 | y |
| 9 | 5.005 | y | 9 | 164.470 | y | 9 | 100.855 | y |
| 10 | 8.290 | y | 10 | 131.210 | y | 10 | 111.835 | y |

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Network (40%)** | | | **Network (60%)** | | | **Network (80%)** | | |
| Test # | Latency (ms) | Success | Test # | Latency (ms) | Success | Test # | Latency (ms) | Success |
| 1 | 1159.51 | y | 1 | 21026.75 | y | 1 | 21026.75 | y |
| 2 | 121.815 | y | 2 | 308773.8 | y | 2 | 308773.8 | n |
| 3 | 194.425 | y | 3 | 472.5752 | y | 3 | 472.5751953 | y |
| 4 | 401.31 | y | 4 | 21107.5 | y | 4 | 21107.50488 | y |
| 5 | 611.04 | y | 5 | 302008.3 | n | 5 | 302008.2949 | n |
| 6 | 123.1301 | y | 6 | 2021.495 | y | 6 | 83643.51514 | n |
| 7 | 442.3098 | y | 7 | 358.03 | y | 7 | 94.71020508 | y |
| 8 | 417.675 | y | 8 | 107.085 | y | 8 | 19595.15503 | y |
| 9 | 116.9551 | y | 9 | 619.72 | y | 9 | 18188.41504 | y |
| 10 | 121.0901 | y | 10 | 10613.15 | y | 10 | 301506.3913 | n |

|  |  |  |
| --- | --- | --- |
| **Network (100%)** | |  |
| Test # | Latency (ms) | Success |
| 1 | 40165.49 | n |
| 2 | 10693.06 | n |
| 3 | 2675.14 | n |
| 4 | 12212.89 | n |
| 5 | 1023.355 | n |
| 6 | 1620.735 | n |
| 7 | 2139.47 | n |
| 8 | 6903.95 | n |
| 9 | 8438.17 | n |
| 10 | 589.825 | n |

### Appendix – Push Notification Tests

Tests for the push notification. Push notification requests sent from browser to server. Time between request being sent and notification being received recorded, as well as if the notification was finally received. Tests conducted ten times at with increasing levels of packet drop. Note: packet drop only applied to incoming packets from server.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Network (0%)** | | **Network (20%)** | | **Network (40%)** | |
| **Latency (ms)** | **Pushed** | **Latency (ms)** | **Pushed** | **Latency (ms)** | **Pushed** |
| 88.28 | y | 156.14 | y | 105.06 | y |
| 62.72 | y | 104.40 | y | 116.79 | y |
| 28.81 | y | 108.12 | y | 98.92 | y |
| 31.33 | y | 104.28 | y | 98.26 | y |
| 31.97 | y | 131.95 | y | 88.88 | y |
| 42.96 | y | 103.98 | y | 96.86 | y |
| 42.42 | y | 96.76 | y | 91.00 | y |
| 31.72 | y | 96.58 | y | 109.09 | y |
| 23.91 | y | 104.35 | y | 83.71 | y |
| 28.32 | y | 139.73 | y | 89.05 | y |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Network (60%)** | | **Network (80%)** | | **Network (100%)** | |
| **Latency (ms)** | **Pushed** | **Latency (ms)** | **Pushed** | **Latency (ms)** | **Pushed** |
| 157.37 | y | 6676.97 | n | 21012.12 | n |
| 105.06 | y | 1157.82 | y | 21013.07 | n |
| 113.18 | y | 960.33 | y | 21011.10 | n |
| 182.31 | y | 6193.35 | y | 21013.48 | n |
| 112.47 | y | 8134.25 | n | 21012.71 | n |
| 109.30 | y | 112.24 | y | 21009.28 | n |
| 1166.71 | y | 1075.80 | y | 21009.00 | n |
| 97.85 | y | 94.59 | y | 21011.61 | n |
| 96.12 | y | 4088.19 | n | 21010.95 | n |
| 97.62 | y | 31037.04 | n | 21012.92 | n |

### Appendix – Background Sync Tests

Tests of the background sync mechanism, recording the time between a request being queued and it being successfully completed. Each test was then repeated ten times with increasing amounts of packet drop. The latency and whether the request successfully completed were recorded.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Network (0%)** | | **Network (20%)** | | **Network (40%)** | |
| **Success** | **Latency (ms)** | **Success** | **Latency (ms)** | **Success** | **Latency (ms)** |
| y | 61.13 | y | 37.49 | y | 322.29 |
| y | 42.98 | y | 36.66 | y | 41.52 |
| y | 80.82 | y | 32.98 | y | 37.42 |
| y | 43.48 | y | 35.99 | y | 1014.66 |
| y | 36.37 | y | 34.94 | y | 336.82 |
| y | 36.49 | y | 34.57 | y | 2992.08 |
| y | 48.56 | y | 43.33 | y | 38.35 |
| y | 33.68 | y | 34.67 | y | 4008.41 |
| y | 38.20 | y | 33.76 | y | 35.66 |
| y | 34.81 | y | 296.03 | y | 282.90 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Network (60%)** | | **Network (80%)** | | **Network (100%)** | |
| **Success** | **Latency (ms)** | **Success** | **Latency (ms)** | **Success** | **Latency (ms)** |
| y | 3770.33 | y | 302.56 | n | 328574.88 |
| y | 352.16 | y | 270.04 | n | 155431.85 |
| y | 4192.06 | y | 3537.82 | n | 21016.32 |
| y | 333.00 | n | 303106.09 | n | 21018.58 |
| y | 41.79 | y | 50290.22 | n | 21018.01 |
| y | 32.66 | y | 192264.54 | n | 21016.25 |
| y | 2100.34 | y | 275.97 | n | 21010.81 |
| y | 42.67 | y | 288.35 | n | 11656.59 |
| y | 616.49 | y | 1392.89 | n | 21870.29 |
| y | 17062.62 | n | 107322.55 | n | 25281.08 |