An analysis of the technologies available to create an offline-first progressive web application, with features which compare to a native experience.

George Cook

August 25, 2016

Word count 7684

Abstract

We are in a changing world - the Internet is being used heavily, not always in optimal network conditions, and the web community needs to adapt to these changes. Native applications already have full offline support, so web applications need to address this. This dissertation aims to analyse the available technologies at the forefront of modern web development, to create a web application which functions offline and optimises online experiences.

It has been found that through the use of Service Workers, modern frameworks and local browser storage, an offline web application is possible. However, many advances are necessary to compete with native applications, but with the work of companies like Google, advances could be possible in the near future.

Contents

Chapter 1

Introduction

This research project aims to analyse the components and technologies at the forefront of modern web development. A modern, progressive and offline-first web application will be created from the research obtained, with the goal of creating an experience which competes with a native application.

In his conference talk "Instant Loading: Building Offline-First Progressive Web Apps" (2016), Archibald mentioned that current web applications struggle when there is little or no internet connection available, which is not acceptable. Building offline web applications was a particularly big focus of the Google I/O conference this year, with many talks discussing the need to improve how web applications handle poor internet connections.

These types of applications will become especially important due to the rise in the numbers of people using the internet - usage has increased by 832.5% from the year 2000 to 2015 (Internet World Stats Authors 2016).

During this research, components will be explored to attempt to create an application which functions offline or on a poor connection, drastically improving user experience. This will be worthwhile research as it has an immediate use in the industry.

There are many components which make up a web application; each aspect will be researched starting with potentially the most important: how to achieve the offline functionality. Front-end JavaScript frameworks will be researched, along with database options. The backend language will also be decided upon, and finally the local browser storage options will be researched.

Chapter two will include the findings of the research. Chapter three will be a discussion about the findings and how they fit in with the offline-first model - including code examples of how this fits into the practical part of this dissertation. Finally, Chapter four will be the conclusion.

The practical portion of this project is live and can be accessed at https://stir-recipe.herokuapp.com.

1.1 Methodology

The research involved for this dissertation will be in an exploratory manner. Exploratory research is necessary when there are few resources available on the chosen topic (Lynn University Authors 2016). The aim is to gain a greater understanding of the topic and potentially to guide other researchers on areas of interest to be researched further (Lynn University Authors 2016). As previous research is limited, this project does not aim to provide a complete answer to the

question, but a more general rounded answer on the subject of offline web functionality as a whole.

To research the components thoroughly, a mixture of quantitative and qualitative research will be carried out where suitable. This will help the research, as these two types of research overlap and can complement each other (Brannen 2005, p176). Qualitative research will be used to gain an insight into the subject matter and its importance. Developers will be contacted for their opinions and this will be logged in the research. Quantitative research will be performance-based tests, where a data set will be collected and then used to help make decisions on which components are best suited.

As there is no comprehensive previous research to be read on this subject matter, a literature review section is not necessary in this piece of work.

Chapter 2

Results From Research

2.1 Network Functionalities

2.1.1 Why is controlling the network important?

Web applications have advanced tremendously recently, with the invention of many JavaScript frameworks, and with more interest being put in the web through companies like Google and Facebook. However, the current experience with poor connectivity or no connectivity, is simply not good enough (Archibald 2016*a*).

As society becomes more dependant on technology and the Internet (Internet World Stats Authors 2016), applications need to be able to extend their functionality to allow offline use. The following section will discuss the options available: these are the Application Cache and Service Worker API.

2.1.2 Application Cache API

What is Application Cache?

Application Cache was introduced in the HTML5 specification; it allows developers to cache resources for offline use (Mozilla Contributors 2016*b*). It works through the use of a manifest file and uses local storage to cache resources (Mozilla Contributors 2016*b*).

However, the Application Cache API has been deprecated in favour of Service Worker. Due to this, no further research or implementation has been carried out, as it will be removed from the Web standards (Mozilla Contributors 2016*b*).

2.1.3 Service Worker API

What is a Service Worker?

A Service Worker is a JavaScript worker script, which sits in between the browser and the network (Archibald 2016*c*). Service worker was first shipped in Chrome version 40 in January 2015 (Chromium Contributors 2016). It is being worked on by a number of teams at Google, Samsung, Mozilla and others (Archibald 2016*c*).

Main Features

During Archibald's Google I/O talk, it was mentioned that network requests can be intercepted by the Service Worker. This can then be harnessed in a way to create an application which functions offline, as well as online, to greatly improve user experience (Archibald 2016*a*). To get this offline functionality, content can be cached using the new Cache API in browsers, which can be served when the user does not have connectivity (Osmani 2016).

Performance to a user is a key measure of a website's quality; users simply do not like to wait for content (Osmani 2016). Service Workers can help this by using caching to their advantage; an application shell can be cached and served in response, whilst the dynamic content of the page is fetched through the network (Osmani 2016). It is not just stale content which can be cached, dynamic content can also be cached using a mixture of the Cache API and IndexedDB (Osmani 2016).

A Service Worker is made up of events, including install, activate and fetch (Mozilla Contributors 2016*a*). On install, it is possible to cache assets needed for the page. Activation is where the Service Worker starts functioning on the page - this is mainly to clean up older caches and Service Workers (Mozilla Contributors 2016*a*). The fetch event is potentially one of the most useful events - it allows network requests to be intercepted, allowing developers to create their own responses (Mozilla Contributors 2016*a*).

Service Workers can help user experience when offline even further through the use of the Background Sync API (Archibald 2016*a*). If the user is offline, requests can be sent to the background and once a connection is available, sent to the server (Archibald 2016*b*).

Other than controlling the network, Service Workers can also provide other enhancements such as push notifications (Bidelman 2016). Push notifications are common place in native applications and help with user re-engagement - an example of these being used on the web was on the official Google I/O website in 2016 (Bidelman 2016). Development for push notifications can be made simpler by using OneSignal - full details are available at https://onesignal.com/.

Service Worker And Security

For security, Service Workers can only be served on a secure HTTPS origin (Chromium Contributors 2016). Service Workers can be used to modify network requests, so it is important to protect against "man in the middle" attacks (Chromium Contributors 2016).

Service Worker And Performance

A small test has been done on an application to test the application shell theory (Osmani 2016); a Service Worker has been used to cache the entire interface of a specific page. Google Chrome was chosen for this test, as its developer tools are very helpful, loading times can be checked and screenshots are available to show the first paint time. To get an accurate result, when doing the test without a Service Worker, caching had to be turned off inside the browser.

As can be seen in figure 2.1 on page 7, adding a Service Worker and caching provides performance benefits. Without a Service Worker, the average first paint time is 1475 milliseconds compared to 860 milliseconds for a cached Service Worker response. This is a decrease of just over 41%, so a real benefit to the user.

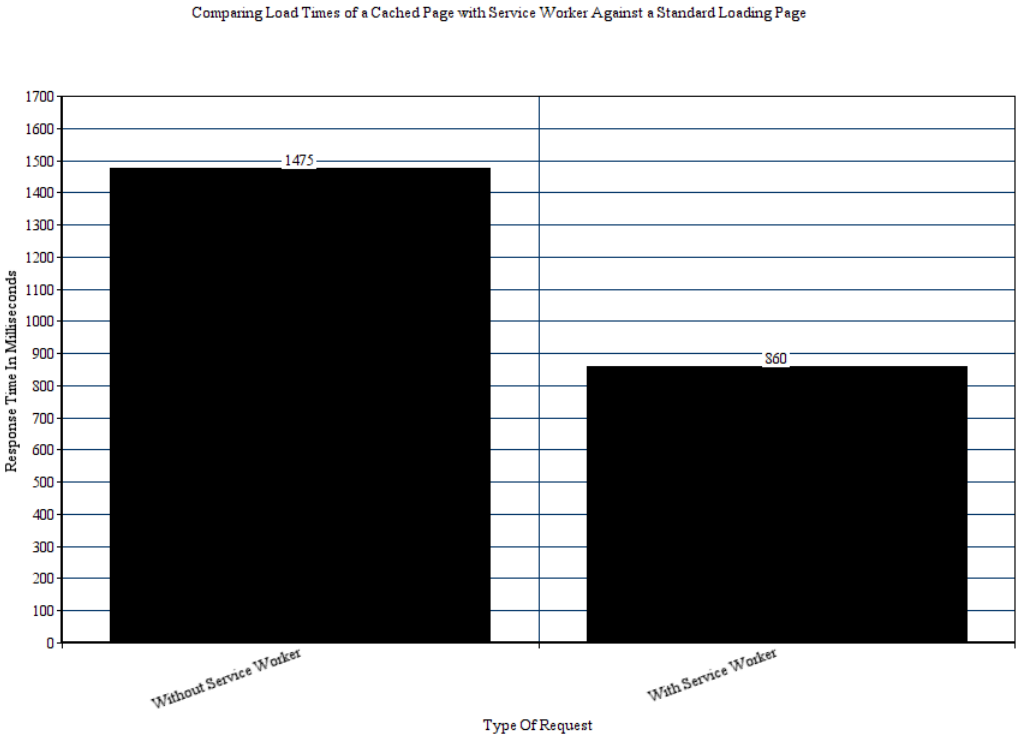

Figure 2.1: Comparing First Paint Render Times - Cached Service Worker Content Against a Standard Request

Add To Home Screen

Web applications now have the ability to be installed directly onto a user's homescreen, which is enabled through the inclusion of a web app manifest (Chrome Contributors 2016). This places a shortcut to the application, and when loaded, it is viewable in fullscreen mode to emulate an native application experience (Chrome Contributors 2016).

App install banners were added to Chrome 42 (Kinlan 2016) - this is a way of asking if the user would like to install your web application to their homescreen. This does have a few requirements: a manifest file must exist, a Service Worker must be registered, the site must be served over HTTPS, and finally the user must have visited at least twice with at least a five-minute gap between visits (Kinlan 2016).

Considerations

Browser support for Service Workers is not currently available across the board. It has been shipped in Chrome and Firefox, development is underway for Edge, and Safari have the Service Worker API under consideration (Archibald 2016*c*). In saying that, as Application Cache is being deprecated, and Service Workers are being recommended for their replacement (Mozilla Contributors 2016*b*), it is hoped that browser support will increase.

However, Service Workers can be set up to allow for Progressive Enhancement, meaning that users are not penalised for not having the latest technology in browsers (Gov Contributors 2016).

2.2 Frameworks

2.2.1 History Of Frameworks

Traditionally web applications were built using a round trip model (Freeman 2014, p.45). This involves requests to the server which returns HTML. Interactions can then make the browser request more HTML from the server (Freeman 2014, p.45). A major drawback of this model is that on each request, the user has to wait for each set of new content (Freeman 2014, p.46).

A newer approach is the Single Page Application architecture, in which only one HTML page is rendered, and AJAX calls are made to render new data or HTML to the page (Freeman 2014, p.46).

Two front-end frameworks will be looked at in detail – Facebook's React and Google's Angular – version one. These have been chosen as they are popular among developers and both have large teams behind them. Please note, both frameworks run on the Single Page Application architecture.

2.2.2 React

What is React?

React is a front-end library, which is built and maintained by Facebook – for documentation please see - https://facebook.github.io/react/.

Main Features

React has a component-based view system (Vepsäläinen 2016, p.i). React is not bound to the document object model, which is an interesting design as it is not just restricted to the web (Vepsäläinen 2016, p.i). This allowed the creation of React Native, which is also developed by Facebook – with documentation being available at https://facebook.github.io/react-native/.

React components are simple, however, they do have a lifecycle of their own which allows complex handling of state and the creation of complex applications (Vepsäläinen 2016, p.30-31).

JSX is included with React, this is an XML-like syntax extension to ECMAScript (Facebook Authors 2016*a*). This allows you to program in a syntax resembling HTML inside of JavaScript (Vepsäläinen 2016, p.5-6). As JSX is not a browser standard, it needs to be compiled into standard JavaScript which the browser can understand (Vepsäläinen 2016, p.10). Webpack is often used for this compiling requirement, Webpack also acts as a bundler for your files (Vepsäläinen et al. 2016, p.i).

When a web application is developed involving dynamic data, the applications state can become problematic, especially if this data needs to be rendered in multiple places simultaneously (Vepsäläinen 2016, p.3). To handle state, React introduced the virtual-DOM, which is a copy of the physical DOM but sits a layer above and can detect changes (Vepsäläinen 2016, p.3). When a change of state takes place, only the necessary pieces of the DOM are re-rendered (Vepsäläinen 2016, p.3).

Like other front-end frameworks, React is rendered on the client mainly, but recent advances can allow rendering on the server (Vepsäläinen 2016, p.3). React has a renderToString method included, which produces HTML on the server rather than the JavaScript in the browser creating the HTML (Creamer 2015). Search engines are more comfortable dealing with HTML than

JavaScript, so this has SEO benefits for the application (Creamer 2015). As HTML is delivered from the server, there is also less time for content to be displayed, meaning a better user experience (Creamer 2015).

React Native

Facebook have created a platform which uses React to build native applications (Facebook Authors 2016*b*). Whilst using React Native, developers have access to the standard platform components such as the navigation drawer on Android (Facebook Authors 2016*b*). Meaning the application will have a familiarity across the whole platform ecosystem, these components are included as standard React components (Facebook Authors 2016*b*).

Considerations

In respect of performance, using the Virtual DOM can lead to increased efficiency. However, implementation of Virtual DOM in React's code base makes the library fairly large for a view-only library – up to 200kB minified (Vepsäläinen 2016, p.3). This can counteract the increased performance provided by the virtual DOM.

Performance Results

A simple test was carried out to measure the basic performance of React. A small breakdown of code for this can be seen in Appendix A. The results of these tests can be seen in figure 2.2 on page 10.

Starting with Chrome, React had an average render time of 310 milliseconds. With Firefox the render time was 289 milliseconds, and finally with Safari the render time was 264 milliseconds. These results will be analysed in more detail in the discussion.

2.2.3 Angular

What is Angular?

Angular is a front-end framework, which was built and is maintained by Google. The full documentation is available at https://angularjs.org/. For reference, Angular is being analysed, not the more recent Angular2. At the time of writing, Angular2 has had a stable version for approximately two months, so the decision has been made, for depth of information, to choose the more documented legacy Angular framework.

Main Features

Angular applications are based around the Model View Controller design pattern (Freeman 2014, p.3). As an observation during the creation of the test for Angular, the framework can be quick to set up. However, there are plenty of other building blocks needed to create a functional application, including modules, directives and many more (Freeman 2014, p.207).

The top level components in an Angular application are modules (Freeman 2014, p.209). These modules are mainly used to organise code, and help create a structured codebase (Freeman 2014, p.209).

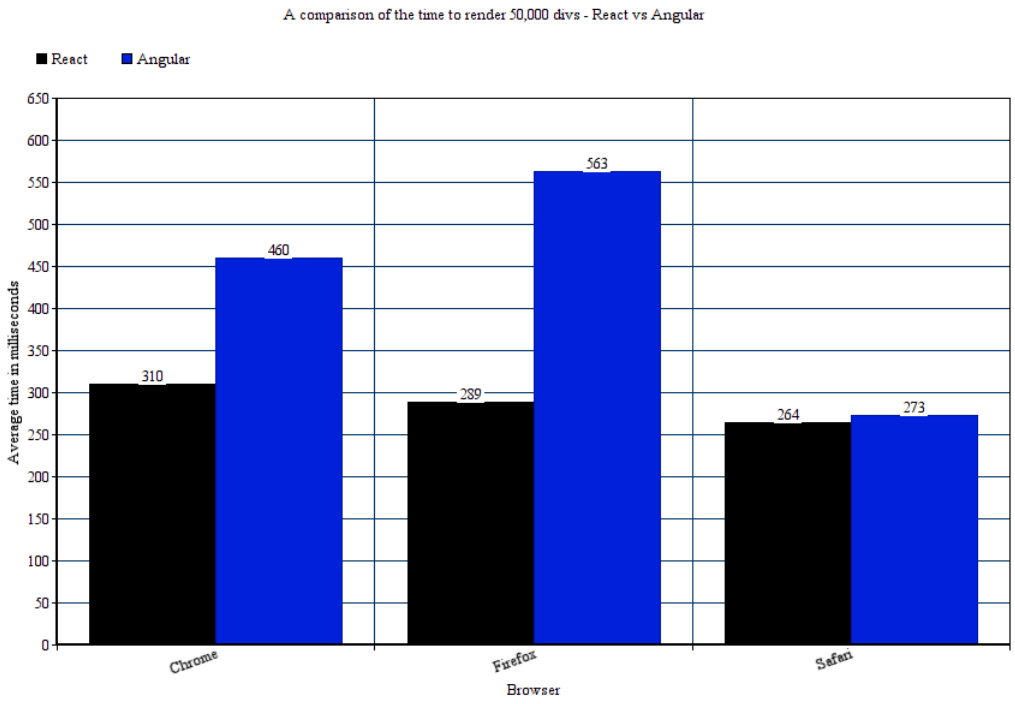

Figure 2.2: Comparing React & Angular - Render times to produce 10,000 random divs. Each test was run 10 times with a mean average being calculated. This test was run across a range of browsers including Chrome Version 51.0.2704.106, Safari Version 9.1.1 and finally Firefox Version 47.0.1.

Directives allow the extension of HTML to create rich applications (Freeman 2014, p.233). Many directives are included in the framework, such as "ng-repeat", which allows simple generation of repeating elements (Freeman 2014, p.233). To extend this further, developers can make their own custom directives to provide further functionalities (Freeman 2014, p.233).

Angular handles data/state through the use of its data bindings (Freeman 2014, p.237). There are two main types of data binding, one-way and two-way (Freeman 2014, p.237-240). One-way data binding involves taking a value from a data model and passing it into a HTML element (Freeman 2014, p.237). Two-way data binding tracks changes both ways – this allows the user to change the data and state of the application (Freeman 2014, p.240). In Angular, bindings are live so whenever the value changes in the model, the HTML changes along with it (Freeman 2014, p.237).

Ionic

Ionic is a platform in which native applications can be made on top of the Angular framework using just HTML, CSS and JavaScript (Ionic Contributors 2016). Developing interfaces with Ionic is greatly helped by the CSS components that are included in the platform (Ionic Contributors 2016). The JavaScript implementations are also simple; they extend Angular's Directive model. (Ionic Contributors 2016).

Considerations

Each time an HTML document is loaded, Angular has a lot of processing to do. HTML elements need to compile, data bindings need to be evaluated and more (Freeman 2014, p.45). This can take time to load depending on the device being used and the internet connection available – if an older device is used on a poor network there can be an obvious loading time for the application, which is not ideal (Freeman 2014, p.45).

Performance Results

As per the previous section on the framework React, please see the results of the rendering tests in figure 2.2 on page 10. For review, the full code for these tests can be viewed in Appendix A.

Starting with Chrome, Angular had an average render time of 460 milliseconds. With Firefox the render time was 563 milliseconds, and finally with Safari the render time was 273 milliseconds. These results will be analysed in more detail in the discussion.

2.3 Database Systems

A database system is required if an application is going to store data remotely. Two types of systems will be researched in detail - MySQL and MongoDB.

2.3.1 MySQL

Background Information

MySQL is an open-source relational database management system. (Vicknair et al. 2012, p.1-2). MySQL is flexible and can be used for a number of applications including the web and embedded

systems (Schwartz et al. 2012, p.1).

As MySQL is open-source, and can be used free of charge, it has a significant market share (Vicknair et al. 2012, p.2). Ahmed et al. has stated that "MySQL is the most successful database system being used all over the world" (2010).

Structure

MySQL is made up of tables and rows (Vicknair et al. 2012, p.1-2; Forta 2004, p.7-10). Tables hold structural data, each record will have its own row in the table when saved (Forta 2004, p.9).

Features Of Database

Indexing allows storage engines to find rows in a database quickly, which is key for high performance (Schwartz et al. 2012, p.147). Within an SQL database, it is thought that Index optimisation is the greatest way to improve query performance (Schwartz et al. 2012, p.147).

MySQL incorporates several security features one being transactions: which are a way of grouping together your queries and providing security that if one query fails, none of the queries will execute and in so doing potentially corrupting data (Schwartz et al. 2012, p.6).

Considerations

In recent years, it has become apparent that the migration of applications onto the web has caused more data than ever to be handled by systems such as MySQL which can cause unpredictable behaviour (Tomic et al. 2013, p.455). This can result in over-provisioning of databases to ensure that there is adequate performance at all times, which can be costly and intrusive (Tomic et al. 2013, p.455).

Database design in MySQL is based around normalisation (Copeland 2013, p.4). Databases are often normalised to avoid unnecessary redundancy within tables (Copeland 2013, p.6). However, when trying to retrieve the data, joins must be used, which are one of the most costly database operations (Copeland 2013, p.6-7).

Performance Results

A number of tests were performed on a MySQL database through the terminal utility provided with MySQl. Each test was run 10 times, the results of which can be seen in the figures are the mean averages of the tests.

As can be seen in figure 2.3 on page 14, the average time for a record to be found from 1,000,000 records by id is 2 milliseconds. In comparison, searching by name has an average time of 224milliseconds which can be seen in figure 2.4 on page 15.

Previously, it was noted that database joins are a costly operation (Copeland 2013, p.6-7) - the next test will address this. Figure 2.5 on page 16 shows that the average time for 1,000 posts to be returned with 10,000 comments per post is 451 milliseconds.

2.3.2 MongoDB

Background Information

In contrast to MySQL, MongoDB is a NoSQL database (Parker et al. 2013, p.1). MongoDB is a highly flexible, document-orientated database and is not relational (Chodorow 2013, p.3)

NoSQL databases are becoming more popular due to the increasing amounts of unstructured data which are being collected by applications (Parker et al. 2013, p.1).

Structure

MongoDB databases are made up of collections containing documents (Chodorow 2013, p.3). Documents are flexible - they can be embedded within another document to create complex hierarchical relationships (Chodorow 2013, p.3).

An interesting design decision of MongoDB is to have non-defined schemas (Chodorow 2013, p.3). Adding and removing fields is easy and common place, which is why MongoDB is great for unstructured data. Due to this, different approaches to schema design will be taken. MongoDB schemas focus on denormalisation, to gain the performance benefits which come with this practice (Copeland 2013, p7-8).

Features Of Database

Scaling in modern applications is now a critical issue - more data is being collected than ever before (Chodorow 2013, p.3). MongoDB has scaling built into the platform, in the form of auto-sharding (Chodorow 2013, p.231). Sharding is the process in which data is divided across several machines (Chodorow 2013, p.231). Manual-sharding can be a tough task so MongoDB includes this as a feature (Chodorow 2013, p.231).

Indexing is included on the platform for faster queries. If an index is not used, then a full table scan has to be performed, which is not ideal (Chodorow 2013, p.81).

On installation, MongoDB comes with a JavaScript shell, which allows incredibly simple interaction with the database (Chodorow 2013, p.13). In the database, data is stored in BSON which is similar to JSON (MongoDB Authors 2016), meaning MongoDB is a great fit for JavaScript applications which are using JSON as their data interchange format.

Considerations

MongoDB does not include a couple of common relational database features which must be noted. As there are no tables, joins are not supported (Chodorow 2013, p.4), meaning that different schema design decisions need to be made. Another feature which is not included is transactions (Chodorow 2013, p.4). However, according to the MongoDB documentation, transaction-like semantics can be used to create a two-phase commit which can emulate a transaction rollback-type functionality.

Performance Results

Similar tests to the MySQL performance tests were performed on a MongoDB database through the terminal utility. Each test was run 10 times, the results which can be seen in the figures are

the mean averages of the tests. As a disclaimer, the query times have been provided through the use of the explain function which comes with all MongoDB queries on the terminal - this is an estimated query time.

As can be seen in figure 2.3 on page 14, the average time for a record to be found from 1,000,000 records by id is 2 milliseconds. In comparison, searching by name has an average time of 295milliseconds which can be seen in figure 2.4 on page 15.

The next test is to check the performance of querying a dataset which contains embedded documents to rival a SQL join. Figure 2.5 on page 16 shows that the query time to get 1,000 posts back with 10,000 comments per post was 0ms. As stated, the tests are not 100% accurate, but this time was noted on each test.

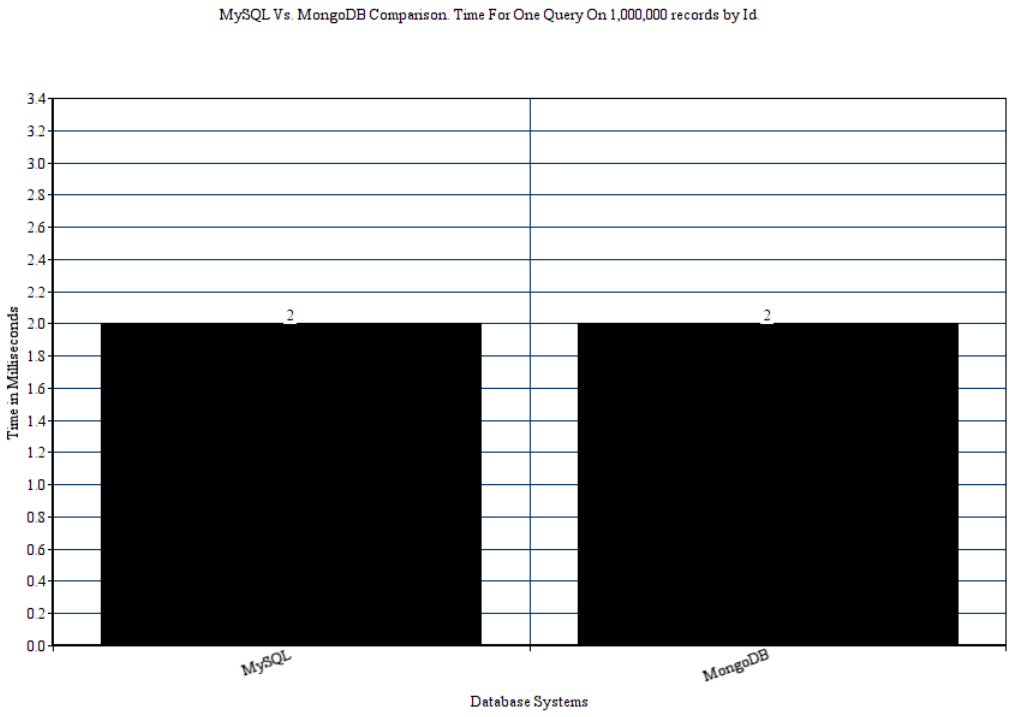

Figure 2.3: Comparing MySQL & MongoDB Query Times Searching by Id

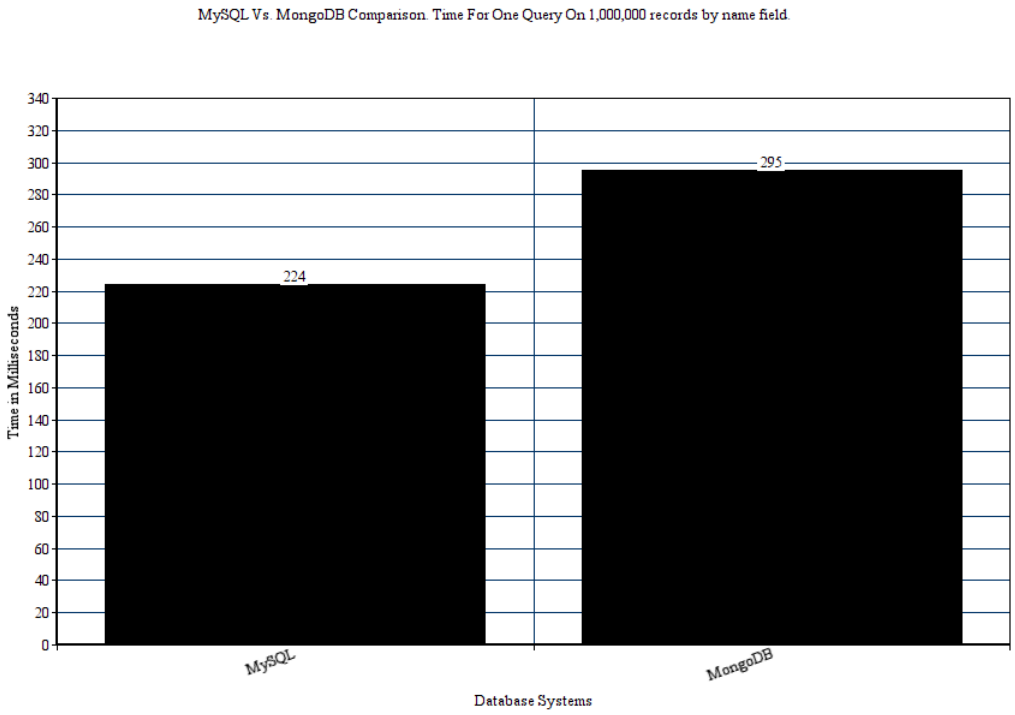

Figure 2.4: Comparing MySQL & MongoDB Query Times Searching by Name

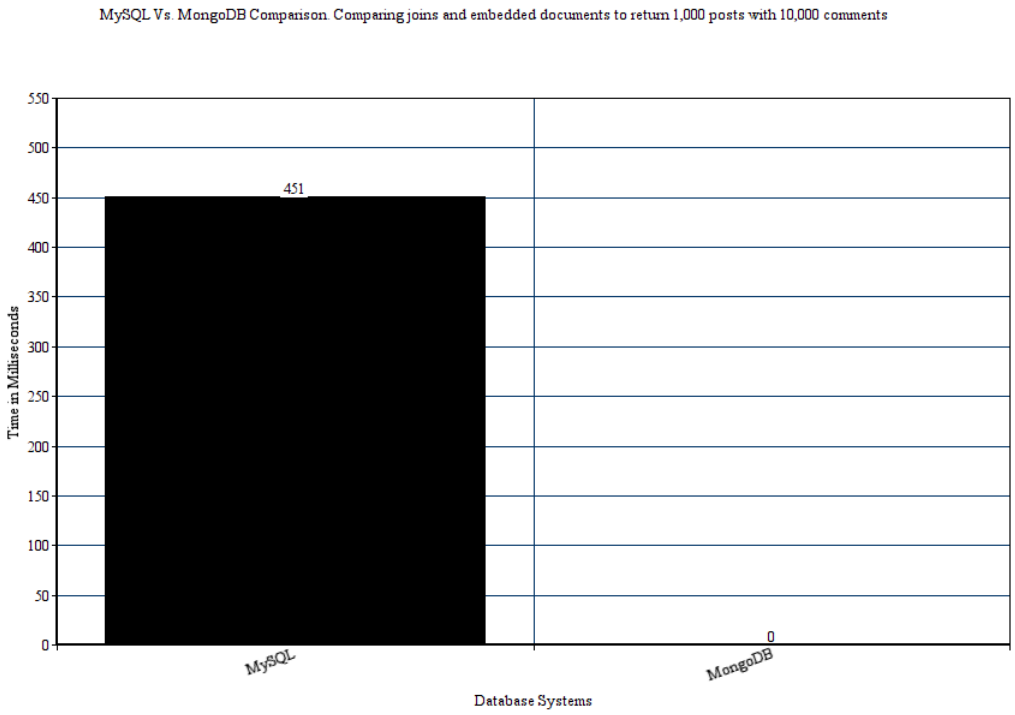

Figure 2.5: Comparing Joins in MySQL with Embedded Documents In MongoDB

2.4 Back-End Language

Two languages will be analysed for the server of the application, one will be a more traditional approach and one will be a more modern approach. The more traditional language will be PHP and the newer approach will be using JavaScript on the server through the Node.js platform.

2.4.1 PHP

What is PHP?

PHP is an open-source, server-side scripting language, created by Rasmus Lerdorf in 1995 (Neves et al. 2013, p.130). PHP is often used on low demand websites, which means it has a large market share (Neves et al. 2013, p.130). The latest version of PHP (PHP7) was released in 2016 - for full information on releases, please visit http://php.net/.

Main Features

Object-orientation lies at the heart of PHP; there is support for classes, objects, constructors and inheritance (Zandstra 2010, p.15). More advanced features are included - static methods, abstract classes, and interfaces for polymorphism (Zandstra 2010, p.41-47).

For a developer, PHP is very easy to get started as you can mix PHP and HTML directly in web pages (Zandstra 2010, p.3). PHP scripts run quickly and can be added to almost any web server. Due to this the PHP community is large (Cholakov 2008, p.1). However, this ease of use can come with some drawbacks. Initially, programmers may not have to structure their code, leading to messy and unmaintainable code (Zandstra 2010, p.3).

Initially, error handling in PHP was problematic, but with the release of PHP5 came Exception handling (Zandstra 2010, p.51). This was a positive move, and is a common approach across the C languages and others such as Java (Zandstra 2010, p.52).

PHP, like many other popular languages, has support for dependency management through the Composer tool (Composer Authors 2016) which was influenced by the popular Node.js package manager NPM (Composer Authors 2016).

Considerations

Using Classes in PHP is not optimised; it takes longer for a PHP class to be instantiated than for the same task in Java (Cholakov 2008, p.2). Experienced developers recognise this and are not motivated to use the less than optimal object-orientated paradigm (Cholakov 2008, p.2).

Naming conventions in PHP seem to have no convention and many inconsistencies, underscores and camel-casing are interchanged which makes the language confusing at times (Cholakov 2008, p.3). Interestingly, variables names are case-sensitive, but function names are not, which can cause mistakes (Cholakov 2008, p.3).

Another irregularity is with PHP's Exception handling. Exceptions are not thrown automatically whenever there is an error in the code, only when issues are within the constructor of an object or problems with extensions like MySQL (Cholakov 2008, p.4).

2.4.2 JavaScript

What is JavaScript?

Introduced in 1995, JavaScript allowed developers to add programs to their web pages (Haverbeke 2015, p.6). JavaScript is now the standard scripting language in all major browsers - an official standard has been decided upon which is known as ECMAScript (Haverbeke 2015, p.6).

Main Features

Techniques can be applied in JavaScript which are near impossible in other more rigid programming languages (Haverbeke 2015, p.6). For example, other languages such as PHP and Java have introduced some features which JavaScript has had for many years, including closures and anonymous functions (Stefanov 2010, p.1).

To its core JavaScript is an object-orientated language (Stefanov 2010, p.3). One point of interest is JavaScript's lack of classes, which can seem a strange concept for traditional object-orientated programmers (Stefanov 2010, p.4).

JavaScript has recently had an upgrade to ECMAScript 6, which is a leap forward for the language (Simpson 2016, p.2). New syntax forms are included for more advanced programming techniques (Simpson 2016, p.2), which are incredibly useful for developers.

Ryan Dahl created Node.js in 2009, to avoid the traditional blocking issues with a threaded PHP and Apache process (Wandschneider 2013, p.1). Whilst these threads are blocked, they are sitting waiting to be completed in a synchronous manor (Wandschneider 2013, p.1). Node.js programs are built around a non-blocking event-driven model, meaning that a request to a database is dealt with asynchronously and other tasks can be carried out at the same time (Wandschneider 2013, p.2).

A common architecture to deal with asynchronous code is with the callback pattern (Wandschneider 2013, p.54). However, this callback model can become confusing; it is easy to accidentally mix up asynchronous and synchronous callbacks (Parker 2015, p.3-6). Promises solve this issue, which have been included in the latest ECMAScript standard (Parker 2015, p.11). Promises allow developers to organise callbacks into sequential steps, which allow for simpler code (Parker 2015, p.11).

Node.js has a lot of functionality built in. For example, the easy creation of web servers through the http module (Wandschneider 2013, p.17-18). Developers often create their own modules and these can be shared with others through the Node Package Manager (Wandschneider 2013, p.92). According to the npmjs.com, currently there are over 250,000 packages for download.

Error handling in JavaScript is based around Exceptions (Wandschneider 2013, p.47). However, as Node.js operates on an Asynchronous model, this can cause a few issues in handling these Exceptions (Wandschneider 2013, p.47). To get around this, there are many design patterns that can be used to standardise error handling in Node.js (Wandschneider 2013, p.54).

Considerations

As Node.js is a relatively new platform, the documentation and online support available is not as established as a language like PHP. This makes finding detailed documentation which discusses Node's weaknesses difficult to find.

2.5 Client-Side Data Storage

2.5.1 What is Client-Side Data Storage?

Over the past decade, web browsers have evolved into powerful applications (Camden 2016, p.1). Client-side data storage is one of the more exciting and useful features; it gives developers an alternative to the traditional request to server process, which is needed to get data (Camden 2016, p.1).

This brings a host of benefits to web applications. Data fetching for these pieces of information is immediate, rather than having to wait for a request (Camden 2016, p.1). Fewer network requests leads to less strain being put on the server (Camden 2016, p.2). Finally, with data being accessible locally, offline web applications become a reality (Camden 2016, p.2).

Three client-side storage techniques will be discussed: these are Local Storage, IndexedDB and WebSQL. Cookies have been excluded due to the fact that they are sent to and from the server, which negates their benefits (Camden 2016, p.3).

2.5.2 Local Storage

Local Storage is part of the Web Storage API, which is focussed on setting and retrieving simple data through the use of a key and value pattern (Camden 2016, p.2). Complex data structures like arrays and objects are not supported, however these values can be encoded in JSON to be stored and then decoded for use later (Camden 2016, p.13).

In use, the API is incredibly simple (Camden 2016, p.13). Local Storage is persistent, unless the user decides to clear their storage (Camden 2016, p.13). Storage limits are variable depending on browsers: the range is from five to ten megabytes (Camden 2016, p.13). Interestingly, Local Storage is very similar to the new Cache API, but it is synchronous so cannot be used in Service Workers (Mozilla Contributors 2016*a*).

As the storage limits are lower for Local Storage, this is not suitable if you plan to store large data. Luckily, there is another API for this which is specialised to large data - IndexedDB.

2.5.3 IndexedDB

IndexedDB is a very flexible storage system, and it has much higher storage limits than Local Storage (Camden 2016, p.27). Support for IndexedDB is good across modern desktop browsers, but is lacking in mobile browsers (Camden 2016, p.27).

However, as complex data can be stored, the API itself is not simple (Camden 2016, p.27). All processes in IndexedDB are handled through events (Camden 2016, p.30), which need to be handled in the code. Libraries have been created to simplify this, including the popular Dexie.js (Camden 2016, p.80). Dexie wraps IndexedDB with promises, but also keeps the core transactional behaviour which is necessary (Camden 2016, p.83).

The highest concept of IndexedDB is the database, which is where all of the data is stored (Camden 2016, p.28). Within these databases, developers can create object stores - which are individual buckets to store data (Camden 2016, p.28).

As IndexedDB can be used in an asynchronous fashion, it is available for use in Service Workers for persistent data storage (Mozilla Contributors 2016*a*).

2.5.4 WebSQL

Before WebSQL is discussed, it is worth noting that it has been deprecated and is in the process of being removed from the specification (Camden 2016, p.63).

WebSQL gives access to miniature databases within the browser, with SQL support for queries (Camden 2016, p.63). Databases are made up of tables and rows as within a relational database (Camden 2016, p.64). This was very useful for developers who were already familiar with server side technologies (Camden 2016, p.63), as it made the transition to browser storage easy.

Chapter 3

Discussion

This section is where decisions are made regarding which technology stack to use. Once the decisions have been made and justified, examples of their use will be included, showing how and why they are useful. These examples will be from the practical aspect of this dissertation, an offline-first recipe storage system.

3.1 Service Worker

Due to the lack of an alternative, Service Worker is the obvious choice to create the offline and network controlling aspects of the web application. The rest of this discussion will be based around which items are the best fit for the application and how they work with Service Workers.

3.1.1 Push Notifications Through OneSignal

Push notifications are an interesting addition to the web and helps rival native platforms, but implementation can be difficult as a lot of steps have to be fulfilled to get them up and running. OneSignal provide an incredibly simple interface for producing push notifications on the web. This service has been used in the practical side of this research.

```
1  importScripts('https://stir-recipe.herokuapp.com/sw.js');
2  importScripts('https://cdn.onesignal.com/sdks/OneSignalSDK.js');
```

Listing 3.1: One Signal Combination of Service Workers. OneSignal use their own Service Worker to create the functionality, which can cause an issue as this overwrites other Service Workers (Pang 2016). During development of the application, this caused major issues as the personal Service Worker to the site was not functioning at all. The solution, as given by the support team, is to import your Service Worker into the OneSignal Service Worker. However, one slight issue does come from this practise. When updating a Service Worker, only the script is checked and not the imported scripts (Archibald 2016*e*). Meaning that if the personal Service Worker is updated, this does not cause an update and the old Service Worker will be used. After the enquiry, the One Signal support team have opened this as an issue on their Github repository - https://github.com/OneSignal/OneSignal-Website-SDK/issues/113. The team behind Service Worker have also stated that they would like to include update checking in imported scripts (Archibald 2016*d*).

3.2 Client-Side Data Storage

As WebSQL is in the process of being deprecated (Camden 2016, p.63), this will not be discussed further as it is not an option for a new project. IndexedDB and Local Storage both have clear value to a web application, both will be used in different scenarios which work to their individual strengths.

3.2.1 Local Storage

As Local Storage is mainly for handling simple data (Camden 2016, p.2), this could be used for a number of different things. One key point to note however, is that Local Storage cannot be used inside a Service Worker. Use must be confined to the front-end of the application only, Service Workers cannot depend on any Local Storage data.

For the examples case, when a user logs into the application they require a token to be stored to enable authorised requests to the server. So once a successful login occurs, the user's token is saved into Local Storage and used later on when making requests. An example is shown in listing 3.2 below.

```
1
2   axios.post(`${ROOT_URL}/sendPost`,
3     {
4       headers: {
5         authorisation: localStorage.getItem('token')
6       }
7     } )
```

Listing 3.2: Local Storage Example. This uses an NPM package called Axios to make AJAX requests. This example shows a request to the send post endpoint on the server which requires a token for authenticiation. The authorisation header is included through the use of the simple Local Storage API. Please note that many steps of the Axios Post request have been simplified for this example.

3.2.2 IndexedDB

IndexedDB is suited to larger amounts of data (Camden 2016, p.27) - in the application this makes storing the offline posts an option. To avoid IndexedDB's difficult API (Camden 2016, p.27), the library Dexie has been used to simplify interaction. An example of this is shown below in listing 3.3.

```
1
2  //delete post from IDB if offline status is false
3      if (!offlineStatus) {
4        db.open().then( function() {
5        return db.posts
6          .where("_id")
7          .equals(post_id)
8          .delete();
9        } ).then( function(doc) {
10         console.log('doc',doc);
11       } );
12
13       //else add post in to IDB as it is now available offline
14     } else {
```

```
15        response.data.forEach( (post) => {
16          if (post._id === post_id) {
17            db.posts.add({
18              _id: post._id,
19              title: post.title,
20              user_id: post.user_id,
21              text: post.text,
22              offline: post.offline
23            });
24          }
25        } );
26      }
```

Listing 3.3: IndexedDB Example. This is a snippet from some client-side logic where the offline button is toggled on and off. Dexie makes the API easier to handle through its use of promises. As can be seen, if the Offline Status changes to false, the post must be deleted from IndexedDB. But if it is changed to true, it can be added into the posts table of IndexedDB.

As IndexedDB can be accessed in a Service Worker (Mozilla Contributors 2016*a*), this gives developers great control over data storage. An example of this can be seen in Appendix B, with the Background Sync event and its related functions to handle the event. It has to be noted that there is a slight issue with the Background Sync event and corresponding API; it is not functioning as expected currently. Developers working on the specification at Google have been contacted and spoken to, however as the feature has just become available in a stable version of Chrome (Archibald 2016*c*), there are still bugs to be fixed and more detailed examples to be made and published.

Dexie is not available inside a Service Worker, so the standard API has been used but has been wrapped in promises for ease of use. Support for this feature is currently only in Chrome (Archibald 2016*c*), so it is important that the feature is not relied upon. It can be an added bonus for users using the specific browser which is required, known as Progressive Enhancement (Gov Contributors 2016).

Upon testing and development for the practical project, it was noted that there is a lack of support for IndexedDB in private browsing in Safari. This causes the application to not function as expected, this bug needs to be looked into more thoroughly.

3.3 Frameworks - Angular Or React

3.3.1 What is the best fit?

As stated in the findings from the initial research, Angular and React are two very different libraries. Angular bases itself upon the Model View Controller design pattern (Freeman 2014, p.3), so it is a full framework where a web application can be made. Alternatively, React is just a view library (Vepsäläinen 2016, p.i), meaning developers have full control over the rest of the application and how it is built. Angular forces developers to work one way and gives no flexibility, which can be seen as restrictive.

Regarding performance, it can be seen in figure 2.2 on page 10, that React has a clear advantage in its rendering speed. React also has advantages through the use of its Virtual DOM and the potential performance benefits this provides (Vepsäläinen 2016, p.3).

If Search Engine Optimisation is a particularly important aspect of a project, as React is able to be rendered initially on the server, this brings a great benefit (Creamer 2015).

In this case, React will be chosen due to its performance and SEO benefits, and will be combined with other libraries to form the proper application structure. It is worth noting that this decision is often down to developer preference, and it can easily go either way.

3.3.2 React Combined With Redux

Many of the examples below will include Redux within the React code. When development started on the React front-end, many examples combined React and Redux together to handle state efficiently. Redux is a state container for JavaScript applications, meaning it can be used with other frameworks such as Angular (Redux Authors 2016). Redux is based upon the principles of the Flux architecture, but makes a few changes (Redux Authors 2016). Redux has specific bindings for React which makes development simple (Redux Authors 2016). Full documentation is available at http://redux.js.org/.

3.3.3 React Combined With Service Worker

Caching And Retrieving Content

Service Worker brings many benefits, one being the application shell technique (Osmani 2016). The application shell architecure is important for applications which make lots of requests and refreshes of the page - it is critical for user experience to get something loaded to the screen as soon as possible (Osmani 2016). As the React application is a Single Page Application, the shell is not so important, as page refreshes are kept to a minimum. However, the whole application can be cached through caching the bundle.js file which is created with Webpack (Vepsäläinen et al. 2016, p.i). This is a typical behaviour for a native application also, an example of this is shown in listing 3.4 below.

```
1  // cached items
2  var CACHE_ARRAY = [
3    "/",
4    "posts/create",
5    "/posts/view",
6    "/bundle.js",
7    "/signin",
8    "/signup",
9    "/signout",
10   "/manifest.json"
11 ];
12 //cache name
13 var CACHE_NAME = "v2";
14 //install event
15 self.addEventListener("install", (event) => {
16   event.waitUntil(
17     caches.open(CACHE_NAME).then( (cache) => {
18       cache.addAll(CACHE_ARRAY)
19     })
20   );
21 });
```

Listing 3.4: Service Worker Install Example. Through the Service Workers install event, a specific cache version is opened and then all of the items in the cache array are stored. As this bundle file is available in the cache offline, it can be served offline which allows the application to stay functional.

An example of how to get this cached content through a Service Worker is shown in listing 3.5 below.

```
var getStaleWhileRevalidate = function (request, cache) {
  return cache.match(request).then(function (response) {
      var fetchPromise = fetch(request).then(function(networkResponse) {
          if (networkResponse.status !== 200) {
              throw new Error('Bad response');
          }
          cache.put(request, networkResponse.clone());
          return networkResponse;
      });
      return response || fetchPromise;
  });
};
event.respondWith(
  caches.open(CACHE_NAME).then( function(cache) {
    return getStaleWhileRevalidate(event.request, cache);
  })
);
```

Listing 3.5: Service Worker Cache Matching Example. The match function allows easy matching from a request to corresponding items in the cache (Mozilla Contributors 2016*a*). A network request is then carried out, and any items which are not in the cache from the request are fetched from the network and then saved into the cache for next time (Mozilla Contributors 2016*a*).

However, issues can arise if a developer needs to change what is cached, which is common place in an agile type of working - as development is rapid (Dinakar 2009, p.579). Service Workers have an event which can handle older caches and Service Workers, the activate event - an example of which is shown in listing 3.6 below.

```
//activate - cleans up old caches if they are still around
self.addEventListener( 'activate' , (event) => {
  var cacheWhitelist = CACHE_NAME;
  event.waitUntil(
    caches.keys().then( (keyList) => {
      return Promise.all( keyList.map( (key) => {
        if (cacheWhitelist.indexOf(key) === -1) {
          return caches.delete(key);
        }
      }));
    })
  );
});
```

Listing 3.6: Service Worker Activate Example. When a new Service Worker is activated, all redundant caches are deleted so that there are no conflicts. When creating an updated Service Worker it is important to increment the cache number so that a new version can be created and used.

Controlling Fetch Events

However, with the application available and functional offline, this can lead to an issue where a user is offline but wants to send a post. This is where the fetch event is used to handle requests and their responses (Ash 2015). An example of this fetch event is shown in listing 3.7 below.

```
1
2   //fetch event
3 self.addEventListener( 'fetch', (event) => {
4
5   if (event.request.url === 'https://stirapi.herokuapp.com/sendPost' ) {
6
7     event.respondWith(
8       caches.match(event.request)
9       .then((response) => {
10        //if response is truthy return it - else fetch the network
11        //if error in network call the fallback
12
13        response ? console.log('returned from cache') : console.log('attempt to
              network');
14
15        if (response) {
16          console.log('response from cache: sw', response);
17          return response;
18        } else {
19          return fetch(event.request)
20          .then( (response) => {
21            console.log('response from SW network: ', response);
22            return response;
23          } )
24          .catch( () => {
25            console.log('network failed');
26            return new Response( "<h1> You cannot login offline </h1>" , {
27              ok: false,
28              status: 503,
29              headers: {
30                'Content-Type' : 'text/html'
31              }
32            })
33          } );
34        }
35      })
36    );
37  }
38
39 });
```

Listing 3.7: Service Worker Fetch Example

This network-first architecture is controlled from the beginning of line 19. If the fetch to the network fails, this is handled in the catch method. You can return a fallback response which the Service Worker will respond with (Mozilla Contributors 2016*a*). In this example, a response status-code of 503 is returned - service unavailable (W3 Authors 2016). This status code is then picked up in the front-end React application, and a specific no connection error is displayed to the user. This drastically improves user experience, mimicking a native-like experience.

Going to the network first for all requests is not always a good idea (Archibald 2016*a*). It can be great for controlling these types of requests, but can cause issues elsewhere.

Please see below in listing 3.8 for a network-first approach for getting the user posts. For those not familiar with Redux - any mention of Dispatch is where the applications state is updated.

```
1   export function getUserPosts(user_id, token){
2     return function(dispatch) {
3       var db = new Dexie('Posts');
4       db.version(1).stores({
5         posts: '_id, title, user_id, text, offline'
6       });
7       var offlinePosts = db.posts.toArray().then( (posts) =>  {
```

```
        console.log('posts:', posts);
      });
      axios.get(`${ROOT_URL}/getPosts?user_id=${user_id}`, {
        headers: {authorisation: token}
      }).then( (response) => {
        dispatch( {type: GET_POSTS, payload: response.data} );
      })
        .catch( (err) => {
          //cached offline posts stored in Indexed DB are dispatched if network
              fails
          dispatch( {type: GET_POSTS, payload: offlinePosts} );
        });
    }
  }
```

Listing 3.8: Network First Fetch Example. This is an Action Creator which helps update the state of the application.

On the surface this appears as though it functions well and it does until the application had poor connectivity. The offline posts, which are stored with IndexedDB, are found and if the network fails these posts are dispatched to update the user interface. However, under poor network conditions, the user has to wait for the network response to come back before any information is painted to the page which can take some time.

This is where an offline-first approach is best, which can be seen in listing 3.9 below.

```
  export function getUserPosts(user_id, token){
    return function(dispatch) {
      var db = new Dexie('Posts');
      db.version(1).stores({
        posts: '_id, title, user_id, text, offline'
      });
      db.posts.toArray().then( (posts) =>  {
        console.log('posts:', posts);
        if (posts.length > 0) {
        //cached IndexedDB posts are sent to user immediately
          dispatch( {type: GET_POSTS, payload: posts} );
        }
      });
      axios.get(`${ROOT_URL}/getPosts?user_id=${user_id}`, {
        headers: {authorisation: token}
      }).then( (response) => {
        //if the network is successful, all posts are dispatched to user
        dispatch( {type: GET_POSTS, payload: response.data} );
      })
        .catch( (err) => {
          dispatch(authError(err.response.data.error));
        });
    }
  }
```

Listing 3.9: Offline First Fetch Example. As soon as the posts are found in IndexedDB, they are dispatched to the user. If a network is available, the retrieved posts are then dispatched to the user providing the updated content. However if no connection is available, the previously rendered content is shown with a message to the user telling them that they are offline. All possibilities are covered, which provides a better user experience.

Informing The User Once Offline Functionality Is Available

As previously explained, all of the above functionality improves user experience. However, this functionality may not be immediately obvious to the user, so it is important to let them know that offline use is available. An example of this is shown below in listing 3.10.

```
1  if (navigator.serviceWorker) {
2        if (!navigator.serviceWorker.controller) {
3          // No service worker is controlling this page
4          // At some point in time in the future, the service worker will be
               installed.
5          // Wait for that point
6          navigator.serviceWorker.ready.then( (registration) => {
7              if (registration.active) {
8                   // Display the message
9                    this.handleShowSnackbar();
10             }
11         });
12       }
13     }
```

Listing 3.10: Inform User Of Functionality. This can normally be done through the registration phase. However due to the use of the One Signal API, the One Signal Service Worker handles the registration. So this solution had to be made, with the help of the One Signal support team. Once the Service Worker has registered, a snackbar message is displayed to the user informing of the offline functionality

3.4 Back-End Languages - JavaScript Or PHP

3.4.1 What is the best fit?

PHP and JavaScript's Node.js platform handle server-side development differently. It is the way in which the languages handle connections which makes this interesting difference.

PHP is based around the traditional apache blocking process (Wandschneider 2013, p.1), whereas Node.js is built upon a non-blocking event-driven model (Wandschneider 2013, p.2). It is this asynchronous model which is one of Node's strongest points. Multiple tasks can be run at the same time, without taking up the server's processes (Wandschneider 2013, p.2).

PHP on the other hand, has been used on the server for a much longer time than JavaScript, meaning that more documentation will be available for PHP. However, JavaScript has the added benefit of NPM (Wandschneider 2013, p.92), which has a large community behind it and over 250,000 packages available to download. PHP's Composer, which is based upon NPM (Composer Authors 2016), does not have the same level of support.

PHP's ease of use and lack of structure can sometimes cause issues for bigger projects (Zandstra 2010, p.3). Node on the other hand requires a structured approach. There are some very popular libraries within NPM which promote structure, one being Express which is a full framework which can be used in a Model View Controller pattern (Wandschneider 2013, p.137). Routing in an express application is very simple: each of the HTTP verbs has their own function where a URL can be specified meaning a RESTful API is simple to create (Wandschneider 2013, p.140-144). It also has to be noted that PHP itself does have a number of frameworks to create applications such as Laravel.

3.4.2 What to choose?

Deciding between PHP and JavaScript is predominantly down to developer preference. For this application, a RESTfull API is needed, and as Node.js and Express offer great support for this, Node.js will be the chosen platfrom. Service Workers do not intereact with this server portion, they only require back a response so this has no impact on the decision.

3.5 Databases - MySQL Or MongoDB

3.5.1 What is the best fit?

MongoDB and MySQL tackle the issue of database handling in a very different manor, MySQL being the rigid relational system and MongoDB being the flexible document based system (Vicknair et al. 2012, p.1-2; Chodorow 2013, p.3).

These two databases have very different schema design practises, MySQL is based around normalisation and non-repeating data (Copeland 2013, p.4). MongoDB has flexible, non-defined schemas, which allow developers to change their schema depending on the record (Chodorow 2013, p.3).

MySQL does have a very important feature which MongoDB lacks: full transactional support (Chodorow 2013, p.4). This means that MongoDB may not be suitable for operations in which transactions are crucial, such as a banking system. However, when looking at the documentation, MongoDB does have a syntax which allows a transactional type behaviour, just without the guarantees.

An interesting feature which MongoDB comes with is Sharding, which is a built-in scaling mechanism to spread data across multiple machines when a venture expands (Chodorow 2013, p.231). Manual-sharding of a database can be a tough task (Chodorow 2013, p.231), so this would be much more difficult in MySQL.

3.5.2 Which Performs Better?

When looking at figures 2.3, 2.4 and 2.5, query times are very similar. MongoDB clearly wins when a more complicated query has to be done, and MySQL needs to use a join. This could be due to its flexible schemas holding the information (Chodorow 2013, p.3).

3.5.3 What to choose?

As with the server-side language choice, this decision can be based upon a specific developer's preference. MongoDB will be chosen as it performs better with more complex queries and scales much better than MySQL, which in today's market is very important.

Chapter 4

Conclusion

The aim of this research project was to look into the technologies available to create an offline-first web application. A lot of successful research has been carried out, mainly combining technologies through the use of the new Service Worker API to provide the offline functionality.

There are many parts which make up a modern web application, whether it is the most useful front-end framework or the most suitable back-end language or database. And finally, client-side data storage offers a great way of storing useful information in the browser.

The React library was chosen for the front-end, mainly due to its performance and Search Engine Optimisation benefits. MongoDB was chosen for the database, due to its performance and scaling benefits. JavaScript's Node.js was chosen for the server-side development - this is mainly due to the resourceful Node Package Manager and its easy creation of a RESTful API. Service Worker was tied into the different components mainly through modifying network requests, to create a better offline experience for the user.

Upon reflection, not everything was achieved from the research, mainly due to the Background Sync event and API not functioning as expected. Another section which could have been improved if more time was available, would have been investigating rendering React on the server for SEO benefits.

There is still much more to be researched regarding offline web applications and how they can be adopted by the masses. If further research is done in this field, Service Workers should be a main focus. The Background Synchronisation event should be researched heavily, as this will provide further enhancements to user experience. A specific point of research could be based upon how having an effective offline web experience could persuade users to adopt more web applications rather than native applications due to install times and functionality, for example.

Chapter 5

Bibliography

Ahmed, M., Uddin, M., Azad, S. & Haseeb, S. (2010), 'Mysql performance analysis on a limited resource server: Fedora vs. ubuntu linux', *SpringSim '10 Proceedings of the 2010 Spring Simulation Multiconference* pp. 1–7.

Archibald, J. (2016*a*), 'Google I/O 2016. instant loading: Building offline-first progressive web apps', `https://www.youtube.com/watch?v=cmGr0RszHc8`.

Archibald, J. (2016*b*), 'Introducing background sync', `https://developers.google.com/web/updates/2015/12/background-sync?hl=en`. [Online; accessed 18-July-2016].

Archibald, J. (2016*c*), 'Is service worker ready?', `https://jakearchibald.github.io/isserviceworkerready`. [Online; accessed 18-July-2016].

Archibald, J. (2016*d*), 'Service worker meeting notes', `https://jakearchibald.com/2016/service-worker-meeting-notes/`. [Online; accessed 23-August-2016].

Archibald, J. (2016*e*), 'Updating a service worker', `https://github.com/w3c/ServiceWorker/blob/master/explainer.md#updating-a-service-worker`. [Online; accessed 18-July-2016].

Ash, A. (2015), 'Building an offline page for the guardian.com'. Remarks by Oliver Ash at Front-End London on the 27th November 2015.
URL: *https://speakerdeck.com/oliverjash/building-an-offline-page-for-theguardian-dot-com-front-end-london-november-2015*

Bidelman, E. (2016), 'Google I/O 2016. building the google i/o web app: Launching a progressive web app on google.com', `https://www.youtube.com/watch?v=__KvYxcIIm8`.

Brannen, J. (2005), 'Mixing methods: The entry of qualitative and quantitative approaches into the research process', *International Journal of Social Research Methodology* **8**, 173–184.

Camden, R. (2016), *Client-Side Data Storage*, O'Reilly Media, Sebastopol, US.

Chodorow, K. (2013), *MongoDB, The Definitive Guide*, O'Reilly Media, Sebastopol, US.

Cholakov, N. (2008), 'On some drawbacks of the php platform', *CompSysTech '08 Proceedings of the 9th International Conference on Computer Systems and Technologies and Workshop for PhD Students in Computing* .

Chrome Contributors (2016), 'Add to homescreen'. [Online; accessed 18-July-2016].
URL: *https://developer.chrome.com/multidevice/android/installtohomescreen*

Chromium Contributors (2016), 'Service workers', https://www.chromium.org/blink/serviceworker. [Online; accessed 18-July-2016].

Composer Authors (2016), 'Dependency manager for php'. [Online; accessed 21-July-2016].
URL: *https://getcomposer.org/*

Copeland, R. (2013), *MongoDB Applied Design Patterns*, O'Reilly Media, Sebastopol, US.

Creamer, J. (2015), 'React to the future with isomorphic apps'. [Online; accessed 21-July-2016].
URL: *https://www.smashingmagazine.com/2015/04/react-to-the-future-with-isomorphic-apps/*

Dinakar, k. (2009), 'Agile development: Overcoming a short-term focus in implementing best practices', *Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications* pp. 579–588.

Facebook Authors (2016*a*), 'Jsx specification', https://facebook.github.io/jsx/. [Online; accessed 21-July-2016].

Facebook Authors (2016*b*), 'React native. a framework for building native apps using react', https://facebook.github.io/react-native/. [Online; accessed 21-July-2016].

Forta, B. (2004), *Sams Teach Yourself SQL in 10 Minutes*, QUE, Indiana, US.

Freeman, A. (2014), *Pro AngularJS*, Springer Science + Business Media, New York, US.

Gov Contributors (2016), 'Using progressive enhancement', https://www.gov.uk/service-manual/technology/using-progressive-enhancement. [Online; accessed 18-July-2016].

Haverbeke, M. (2015), *Eloquent JavaScript. A modern introduction to programming*, No Starch Press, San Francisco, US.

Internet World Stats Authors (2016), 'Internet usage statistics - the internet big picture'. [Online; accessed 18-July-2016].
URL: *http://www.internetworldstats.com/stats.htm*

Ionic Contributors (2016), 'Ionic. create mobile apps with the web technologies you love', http://ionicframework.com/. [Online; accessed 21-July-2016].

Kinlan, P. (2016), 'Increasing engagement with web app install banners', https://developers.google.com/web/updates/2015/03/increasing-engagement-with-app-install-banners-in-chrome-for-android?hl=en. [Online; accessed 18-July-2016].

Lynn University Authors (2016), 'Soc-200 research methods in the social sciences: Exploratory design'. [Online; accessed 18-July-2016].
URL: *http://lynn-library.libguides.com/researchmethods/researchmethodsd8*

MongoDB Authors (2016), 'Json and bson', https://www.mongodb.com/json-and-bson. [Online; accessed 21-July-2016].

Mozilla Contributors (2016*a*), 'Using service workers', https://developer.mozilla.org/en-US/docs/Web/API/Service_Worker_API/Using_Service_Workers. [Online; accessed 18-July-2016].

Mozilla Contributors (2016*b*), 'Using the application cache', https://developer.mozilla.org/en-US/docs/Web/HTML/Using_the_application_cache\#Storage_location_and_clearing_the_offline_cache. [Online; accessed 18-July-2016].

Neves, P., Paiva, N. & Duraes, J. (2013), 'A comparison between java and php', *Proceedings of the International C* Conference on Computer Science and Software Engineering* pp. 130–131.

Osmani, A. (2016), 'Google I/O 2016. progressive web apps across all frameworks.', `https://www.youtube.com/watch?v=srdKq0DckXQ`.

Pang, J. (2016), 'Sometimes i receive the following notification - the site has been updated in the background, out of nowhere?', `https://github.com/OneSignal/OneSignal-Website-SDK/issues/60#issuecomment-226617878`. [Online; accessed 18-July-2016].

Parker, D. (2015), *JavaScript with Promises*, O'Reilly Media, Sebastopol, US.

Parker, Z., Poe, S. & Vrbsky, S. (2013), 'Comparing nosql mongodb to an sql db', *Proceedings of the 51st ACM Southeast Conference* pp. 1–6.

Redux Authors (2016), 'Redux'. [Online; accessed 21-July-2016].
URL: *http://redux.js.org/*

Schwartz, B., Zaitsev, P. & Tkachenko, V. (2012), *High Performance MySQL*, Vol. 4, 3 edn, O'Reilly Media, Sebastopol, US.

Simpson, K. (2016), *You Don't Know JS: ES6 and Beyond*, O'Reilly Media, Sebastopol, US. [Online; accessed 18-July-2016].
URL: *http://shop.oreilly.com/product/0636920033769.do*

Stefanov, S. (2010), *JavaScript Patterns*, O'Reilly Media, Sebastopol, US.

Tomic, A., Sciascia, D. & Pedone, F. (2013), 'Mosql: An elastic storage engine for mysql', *Proceedings of the 28th Annual ACM Symposium on Applied Computing* pp. 455–462.

Vepsäläinen, J. (2016), *Survive JS. React – From Apprentice To Master.*, Leanpub, British Columbia, Canada. [Online; accessed 18-July-2016].
URL: *https://leanpub.com/survivejs-react*

Vepsäläinen, J., Koppers, T. & Rodríguez, R. (2016), *Survive JS. Webpack – From Apprentice To Master.*, Leanpub, British Columbia, Canada. [Online; accessed 18-July-2016].
URL: *https://leanpub.com/survivejs-webpack*

Vicknair, C., Wilkins, D. & Chen, Y. (2012), 'Mysql and the trouble with temporal data', *ACM-SE '12 Proceedings of the 50th Annual Southeast Regional Conference* pp. 176–181.

W3 Authors (2016), '10 status code definitions'. [Online; accessed 21-July-2016].
URL: *https://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html*

Wandschneider, M. (2013), *Learning Node.js*, Pearson Education, New Jersey, US.

Zandstra, M. (2010), *PHP Objects, Patterns, and Practise*, Springer Science + Business Media, New York, US.

Appendix A

React and Angular Performance Test

```
1
2
3  <!DOCTYPE html>
4
5  <html ng-app="test">
6      <head>
7          <title>Performance Comparison for Knockout, Angular and React</title>
8          <link href="http://cdnjs.cloudflare.com/ajax/libs/twitter-bootstrap/3.3.1/
               css/bootstrap.css" rel="stylesheet" />
9          <style type="text/css">
10             * { box-sizing:border-box; }
11             body { padding:30px 0; }
12             h2 { margin:0; margin-bottom:25px; }
13             h3 { margin:0; padding:0; margin-bottom:12px; }
14             .test-data { margin-bottom:3px; }
15             .test-data span { padding:3px 10px; background:#EEE; width:100%; float
                   :left; cursor:pointer; }
16             .test-data span:hover { background:#DDD; }
17             .test-data span.selected { background:#3F7AD9; color:white; }
18
19             .time { font-weight:bold; height:26px; line-height:26px; vertical-
                   align:middle; display:inline-block; cursor:pointer; text-
                   decoration:underline; }
20         </style>
21
22         <script type="text/javascript" src="http://cdnjs.cloudflare.com/ajax/libs/
               angular.js/1.3.3/angular.min.js"></script>
23         <script type="text/javascript" src="http://cdnjs.cloudflare.com/ajax/libs/
               react/0.12.1/react.js"></script>
24         <script type="text/javascript">
25             console.timeEnd("build");
26
27             document.addEventListener("DOMContentLoaded", function() {
28                 _react();
29             });
30
31             _angular();
32
33             function _buildData(count) {
34                 count = count || 1000;
35
```

```
36                var adjectives = ["pretty", "large", "big", "small", "tall", "
                      short", "long", "handsome", "plain", "quaint", "clean", "
                      elegant", "easy", "angry", "crazy", "helpful", "mushy", "odd",
                       "unsightly", "adorable", "important", "inexpensive", "cheap",
                       "expensive", "fancy"];
37                var colours = ["red", "yellow", "blue", "green", "pink", "brown",
                      "purple", "brown", "white", "black", "orange"];
38                var nouns = ["table", "chair", "house", "bbq", "desk", "car", "
                      pony", "cookie", "sandwich", "burger", "pizza", "mouse", "
                      keyboard"];
39                var data = [];
40                for (var i = 0; i < count; i++)
41                    data.push({id: i+1, label: adjectives[_random(adjectives.
                          length)] + " " + colours[_random(colours.length)] + " " +
                          nouns[_random(nouns.length)] });
42                return data;
43            }
44
45            function _random(max) {
46                return Math.round(Math.random()*1000)%max;
47            }
48
49        // React code starts here
50            function _react() {
51                var Class = React.createClass({
52                    select: function(data) {
53                        this.props.selected = data.id;
54                        this.forceUpdate();
55                    },
56                    render: function() {
57                        var items = [];
58                        for (var i = 0; i < this.props.data.length; i++) {
59                            items.push(React.createElement("div", { className: "
                                  row" },
60                                React.createElement("div", { className: "col-md-12
                                      test-data" },
61                                    React.createElement("span", { className: this.
                                        props.selected === this.props.data[i].id ?
                                         "selected" : "", onClick: this.select.
                                        bind(null, this.props.data[i]) }, this.
                                        props.data[i].label)
62                                )
63                            ));
64                        }
65                        return React.createElement("div", null, items);
66                    }
67                });
68
69        var runReact = document.getElementById("run-react");
70        runReact.addEventListener("click", function() {
71                    var data = _buildData(10000),
72                        date = new Date();
73
74                    React.render(new Class({ data: data, selected: null }),
                        document.getElementById("react"));
75                    runReact.innerHTML = (new Date() - date) + " ms";
76                });
77            }
78    // Angular code starts here
79            function _angular(data) {
80                angular.module("test", []).controller("controller", function(
                      $scope) {
81                    $scope.run = function() {
82                        var data = _buildData(10000),
83                            date = new Date();
84
```

```
85                          $scope.selected = null;
86                          $scope.$$postDigest(function() {
87                  document.getElementById("run-angular").innerHTML = (new Date() -
                        date) + " ms";
88                          });
89
90                          $scope.data = data;
91                      };
92
93                      $scope.select = function(item) {
94                          $scope.selected = item.id;
95                      };
96                  });
97              }
98          </script>
99      </head>
100     <body ng-controller="controller">
101         <div class="container">
102             <div class="row" >
103                 <div class="col-md-12" style="background-color:lightblue; padding
                        :5px; width:700px;text-align:center;">
104                     <h3>Performance Comparison between React and Angular.</h3>
105                 </div>
106             </div>
107
108             <div class="col-md-3" style="background-color:lightgray; padding:2px;
                    margin:10px 25px;">
109                 <div class="row">
110                     <div class="col-md-7">
111                         <h3>React</h3>
112                     </div>
113                     <div class="col-md-5 text-right time" id="run-react">Run</div>
114                 </div>
115                 <div id="react"></div>
116             </div>
117
118 <!-- Angular HTML bit -->
119             <div class="col-md-3" style="background-color:lightgreen; padding:2px;
                    margin:10px 10px;">
120                 <div class="row">
121                     <div class="col-md-7">
122                         <h3>Angular</h3>
123                     </div>
124                     <div class="col-md-5 text-right time" id="run-angular" ng-
                            click="run()">Run</div>
125                 </div>
126                 <div>
127                     <div class="row" ng-repeat="item in data">
128                         <div class="col-md-12 test-data">
129                             <span ng-class="{ selected: item.id === $parent.
                                    selected }" ng-click="select(item)">{{item.label
                                    }}</span>
130                         </div>
131                     </div>
132                 </div>
133             </div>
134
135         </div>
136     </body>
137 </html>
```

Listing A.1: React And Angular Performance Test - 10000 divs. This was taken and adapted from https://www.codementor.io/reactjs/tutorial/react-vs-angularjs.

Appendix B

Background Sync Example

```
self.addEventListener('sync', function(event) {
  if (event.tag == 'send_post') {
    console.log('sync from SW - send post');
    event.waitUntil(
      openDatabase('Outbox').then( (db) => {
        return databaseGet('posts', db).then( (posts) => {
          console.log('result from get:',posts);
          return sendAllFromOutbox(posts)
        } )
      } )
    );
  }
});
function openDatabase(name) {
  return new Promise(function(resolve, reject) {
    var version = 10;
    var request = indexedDB.open(name, version);
    var db;
    request.onupgradeneeded = function(e) {
      db = e.target.result;
      e.target.transaction.onerror = reject;
    };
    request.onsuccess = function(e) {
      db = e.target.result;
      console.log('OPENED DATABASE');
      resolve(db);
    };
    request.onerror = reject;
  });
}
function databaseGet(type,db) {
  return new Promise(function(resolve, reject) {
    var transaction = db.transaction([type], 'readonly');
    var store = transaction.objectStore(type);
    var request = store.getAll();
    request.onsuccess = function(e) {
      var result = e.target.result;
      resolve(result);
    };
    request.onerror = reject;
  });
}
```

Listing B.1: Background Sync Example. The idea of this sync event, is that if a user tries to

send a post when offline, the post is sent to an Outbox from the front-end (Archibald 2016*b*). Once the user is back online, the post will be sent from the Service Worker in the background (Archibald 2016*b*).