BABEL–ING: THE PURSUIT OF A UNIVERSAL DEVELOPMENT LANGUAGE

By

Michael Stumpf

Supervisor(s): Marie Brennan

SUBMITTED IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE OF
HIGHER DIPLOMA IN SCIENCE IN COMPUTING
AT
INSTITUTE OF TECHNOLOGY BLANCHARDSTOWN
DUBLIN, IRELAND
31 AUGUST 2017

Declaration

I hereby certify that this material, which I now submit for assessment on the programme of study leading to the award of **Higher Diploma in Science in Computing** in the Institute of Technology Blanchardstown, is entirely my own work except where otherwise stated, and has not been submitted for assessment for an academic purpose at this or any other academic institution other than in partial fulfillment of the requirements of that stated above.

Dated: 31 August 2017

Author: _____

Michael Stumpf

Abstract

Software development is a major global industry however developers are often limited by and unable to work effectively due to the lack of commonality amongst the myriad of different programming languages, libraries, and frameworks. Using JavaScript as a foundation, the author explores using the same language, libraries, and frameworks across server-side and web and mobile client-side applications to create a unified experience for both developers and end users. Pursuing this architectural concept fostered DRY programming and shared design patterns, simplified automated testing, made bug fixing easier, and reduced overall development times. Implementing such a system in an organisation would improve developer skillset overlap and reduce technological unknowns thereby making project estimates more accurate, all the while allowing for maximal technical flexibility in terms of of a project's business requirements.

Contents

Abstract i

Contents ii

List of Figures iv

List of Tables v

List of Source Code vi

1 Introduction & Background 1
1.1 Finding a Common Language . . . 2

2 Literature Review 4
2.1 Reimagining Babel . . . 5
2.2 Server-side API . . . 6
2.2.1 Node.js . . . 6
2.3 Web Client . . . 8
2.3.1 ReactJS . . . 9
2.4 Mobile Client . . . 11
2.4.1 React Native . . . 12

3 System Analysis 13
3.1 Inspecting the Site . . . 14
3.2 Software Development Life Cycle . . . 14
3.2.1 Workflows . . . 15
3.3 Software as a Service . . . 18
3.4 Writing up the Building Contract . . . 18

4 System Design 19
4.1 Laying the Foundations . . . 20
4.2 Development Environment . . . 20
4.2.1 Version Control . . . 20
4.3 Deployment Environment . . . 21

4.3.1 Web Server . . . 21
4.3.2 Database . . . 21
4.4 Software Architecture . . . 22

5 Implementation **23**
5.1 Server-side API . . . 24
5.1.1 Configuration . . . 25
5.1.2 Authentication . . . 26
5.2 Web Client . . . 26
5.2.1 Styling . . . 28
5.2.2 Data Flow . . . 28
5.3 Mobile Client . . . 29
5.3.1 Ducks . . . 31
5.4 Deployment & Build Process . . . 32
5.4.1 API . . . 32
5.4.2 Web . . . 32
5.4.3 Mobile . . . 33

6 Testing & Evaluation **35**
6.1 Automated Documentation . . . 36
6.1.1 Swagger . . . 36
6.1.2 UI Storybook . . . 39
6.2 Automated Testing . . . 40
6.2.1 Unit Tests . . . 40
6.2.2 End-to-End Testing . . . 42

7 Conclusions & Further Work **43**
7.1 Complications . . . 44
7.2 Next Steps . . . 44
7.3 Summary . . . 45

Appendices **i**
I Project Planning . . . i
II Project Repositories . . . ii

Bibliography **iii**

List of Figures

How Standards Proliferate 1

Tasks 4
Comparison of Server-Side Languages – Requests/Second 7
Comparison of Server-Side Languages – Time/Request 7
Comparison of Client-Side JavaScript Frameworks 9

Compiling 13
Software Development Lifecycle 14
Software Development Explained with Cars – Waterfall 16
Software Development Explained with Cars – Agile 17

XKCD Phone 19
Distributed Repository Model in Git 21
Project Components Arranged Using a Software as a Service Model. . . 22

Code Quality 23
Data Flow with and without Redux 29
Sharing Code between Web and Mobile Clients 31

Automation 35
Visualising the API with Swagger UI 39

Password Strength 43
Project Schedule i

List of Tables

ReactJS Performance – Chrome 39.0.2171.95 10
ReactJS Performance – Firefox 34.0.5 10
ReactJS Performance – Safari 7.0.2 11

List of Source Code

Folder structure – Server-side API 24
API Installation Script 25
Folder structure – Web Client 27
Folder structure – Mobile Client 30
API Deployment Script 32
Web Deployment Script 32
Sample nginx Configuration for a ReactJS Deployment 33
Swagger Route Documentation 37
Swagger Model Documentation 38
Writing Unit Tests with Mocha, Chai, & Sinon 41
Results of Running Unit Tests 42

Chapter 1

Introduction & Background

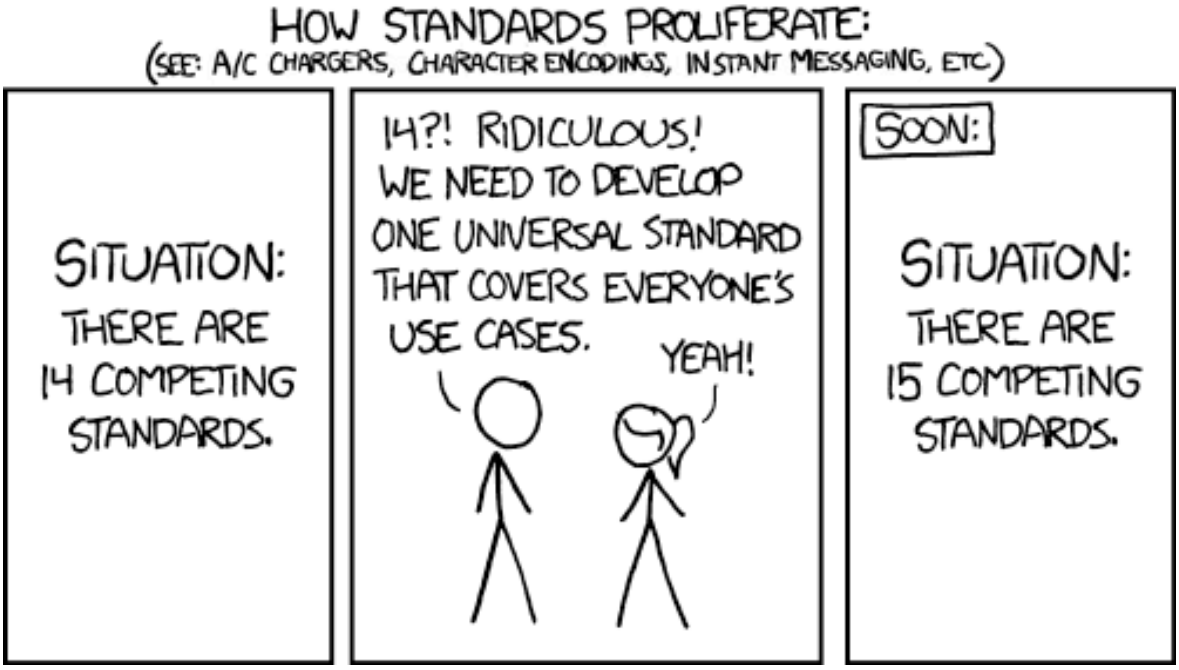

Figure 1.1: (Munroe 2011b)

1.1 Finding a Common Language

We live in a progressively gilded age for software development. There is an ever-growing market of consumers as it is estimated that the percentage of internet users worldwide has risen from 16% to 47% in the last ten years (Union 2014; Union 2016), discussion is ongoing regarding making the internet a basic human right (Wikipedia 2017c), and the number of mobile devices has surpassed the human population in the last few years (Cisco 2017). The surge in demand is also evidenced by the number of active websites skyrocketing from 50,000 in 2004 to nearly 1,000,000,000 in 2014 (Stats 2017) and the roughly 5,000,000 mobile apps in the Apple and Google app stores as of March 2017, a total which continues to grow at a rate of 1,000 to 2,000 apps a day (Portal 2017; AppBrain 2017).

One result of this massive endeavour has been the corresponding expansion of programming languages, libraries, and frameworks (see Wikipedia (2017b) for a non-exhaustive list). While this has afforded programmers a large set of tools from which to pick the best fit for the project at hand, it has also fostered information siloing (Wikipedia 2017a) (e.g. .NET stack vs LAMP stack) and often unnecessary or redundant specialisation which quickly becomes irrelevant in this age of rapid technological advances and growth (e.g. ActionScript). This double-edged sword acting upon programmers greatly influences the iron triangle of project management (speed, cost, and quality) (Bethke 2003, p. 64) as a destabilising agent and, consequently the software development life cycle as a whole (see chapter 3 for a detailed explanation of software workflows and methodologies).

Fred Brooks writes about the difficulty of managing software development in his seminal work, *The Mythical Man-Month: Essays on Software Engineering*. First published in 1975, its lessons continue to be relevant to modern development, particularly in relation to estimating and organising tasks. One the main theses of Brooks's work is what he calls Brooks's Law which, put simply, states that "adding manpower to a late software project makes it later" (Brooks 1975, p. 25). This delay results from the ramp-up time onboarding new developers, an inability to parallelise the workload, and the increased probability of introducing bugs, i.e. diminished coding consistency. In the modern age, additional variables like an abundance of languages and rapid development cycles, like those sponsored by the increasingly popular Agile workflow for instance, have only exacerbated these problems.

The author would argue that delivering software on time is not realised as often as it could be in small to medium-sized organisations due in part to the issues that Brooks notes and also due to the complexities inherent to developing a modern website or mobile app. For instance, developers might use five different languages, CSS, HTML, JavaScript, PHP, and SQL, in order to render a single web page (Frees 2015, p. 84). Mobile development is better in the sense that it relies on fewer languages but there is a less of an ability to reuse code between platforms. For example an iOS app uses Objective-C or Swift, an Android app relies on Java and XML, and a Windows app is

built in C♯ with XAML. Software development is objectively more complicated than it was 40 years ago and the sheer number of languages, libraries, and frameworks (not to mention GUIs and development environments) needed on each of these platforms presents a steep learning curve to any developer. As a result it is in everyone's best interest, programmers and managers alike, to streamline and standardise the development process. And while the points listed above might presuppose finding much common ground, modern developments have presented us with excellent opportunity to regroup, restructure, and reconstruct what is effectively the programmer's Tower of Babel.

Chapter 2

Literature Review

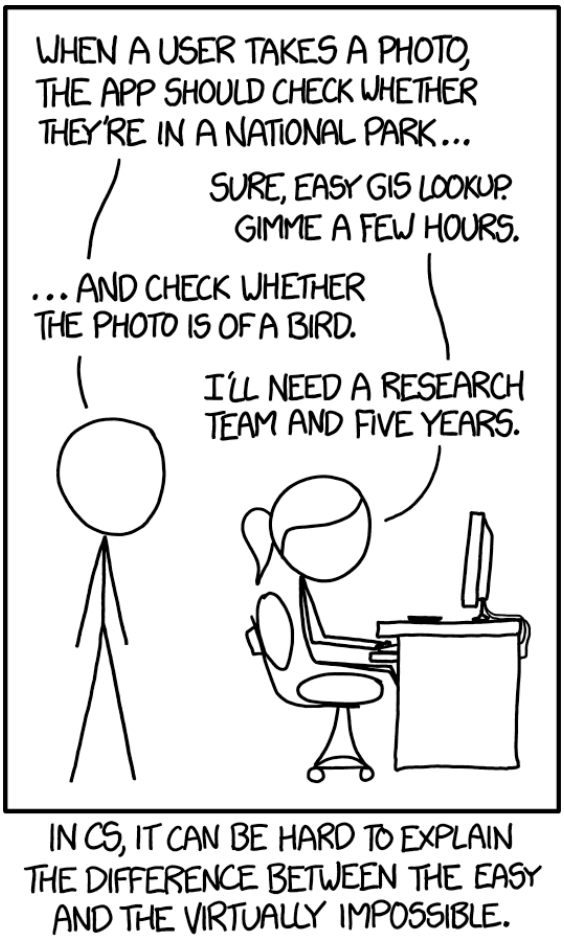

Figure 2.1: "In the 60s, Marvin Minsky assigned a couple of undergrads to spend the summer programming a computer to use a camera to identify objects in a scene. He figured they'd have the problem solved by the end of the summer. Half a century later, we're still working on it." (Munroe 2014b).

2.1 Reimagining Babel

Historically, developer roles have been divided by both the languages they employ and the functionality they focus on, e.g. front-end vs back-end. Suppose we mitigate both of these barriers by employing a common language for the entire development team; how would that affect the software development life cycle? This principal inquiry leads us to the following questions to be answered:

- First, can the same language be used *easily* and *naturally* across server-side, browser-side, and mobile platforms and produce results equivalent to more "traditional" tech stacks?
- Second, how many common frameworks and libraries be reused across different development environments?
- Third, how much bespoke/custom code can be shared across platforms when using a single language?

Although once dismissed as being for "amateurs" (Crockford 2001), the JavaScript language is now considered one of the three core technologies of web development, alongside HTML and CSS,[1] is supported in all web browsers (Flanagan 2011, p. 1) and has reached a critical mass of support from both major industry players and the open source community. Npm for example, the package manager for JavaScript, served its one hundred billionth package a few months ago and is nearing 500,000 registered packages overall, with the library growing at a rate of more than 1,000 packages a day (npm 2017; DeBill 2017). And the release of Google's V8 engine in 2008, which compiles and optimises JavaScript, has increased the language's performance to that of native code (Thompson 2015). Indeed, the use of Google's V8 engine in both server-side applications like Node.js and client-side applications like web browsers has helped make JavaScript into a universal language. It is primarily for this universality that this project has chosen to use JavaScript as it pursues the concept of an "Adamic" language within software development.

This is not the first call for a "JavaScript everywhere" approach (Cuomo 2013) and it will not be the last however three technologies have specifically contributed to this particular watershed moment:

- Node.js
- ReactJS
- React Native

Each of these represents the latest developments in their individual areas of expertise and joining them together presents developers with a whole greater than the sum of its parts.

[1] IBM named JavaScript as one of the best programming languages to learn in 2017 (Felt 2016)

2.2 Server-side API

When thinking about using a common language across development environments, it is easy to conflate a desire to share and reuse resources with a need to consolidate everything within a single application. Technologies like Aptana's Jaxer provided a JavaScript-based server-side experience like PHP where code is stored in tags executed by a "browser's engine running on the server" (Jaxer 2015). However while "tierless" or "tier-splitting" systems like Jaxer purport to promote a more robust relationship between the front-end and the back-end (Guha et al. 2017), any reduction of brittleness from such an approach could arguably be offset by automated testing. Furthermore, the difficulty of separating concerns like presentation, business logic, and data management (Elliott 2014, pp. 99-101) means that horizontally scaling the application would suffer as its user base grows.[2] For these reasons as well as the recognition of multiple client consumers, web and mobile, that the API in this project was built as a stand-alone application.

2.2.1 Node.js

Node.js was first released in 2009[3] and garnered significant attention from both the media and the development community within the first two years of its inception (O'Dell 2011). The main reason for its hype was its use of an asynchronous, low-level I/O API and Google's V8 engine (Teixeira 2012, p. 3) and its resulting high performance compared to other server-side languages like PHP and Python, as shown in Figure 2.2 and Figure 2.3.

[2]The current Jaxer website is built with WordPress, a PHP-based content management system, which is not a vote of confidence in Jaxer's favour in this regard

[3]Aptana scaled back development of Jaxer in November 2009 citing low adoption rates as the reason (Staff 2009)

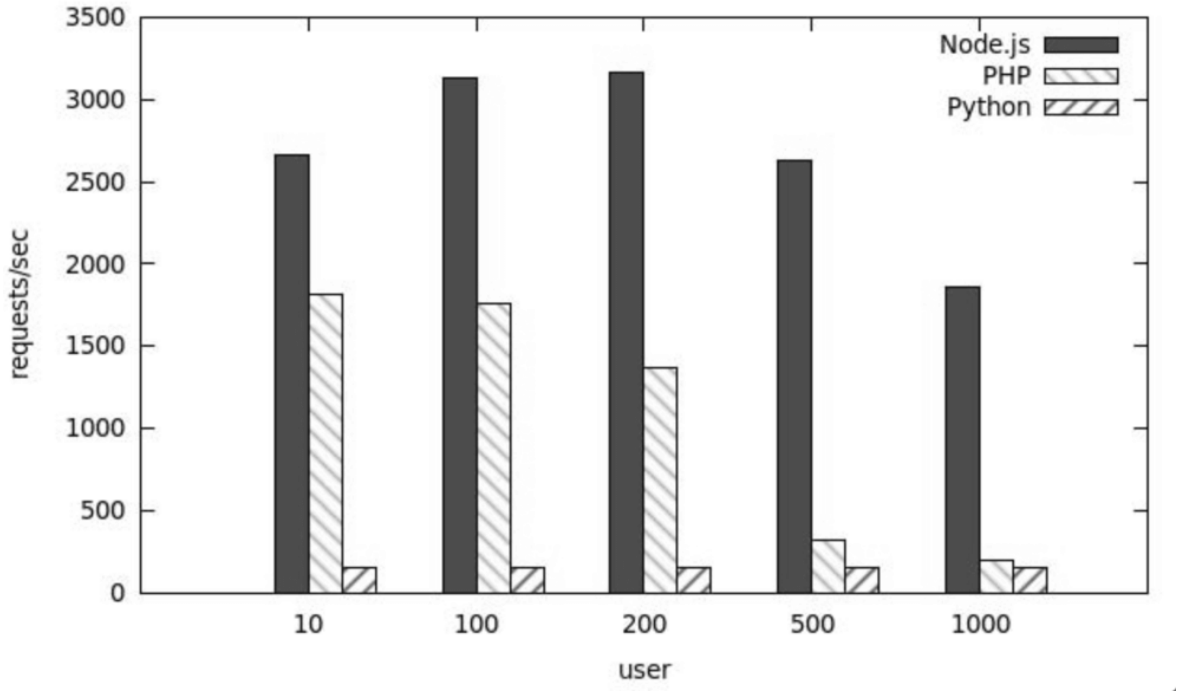

Figure 2.2: "Results for 'Select Operation of DB' mean requests per second" (Lei, Ma, and Tan 2014, p. 665).

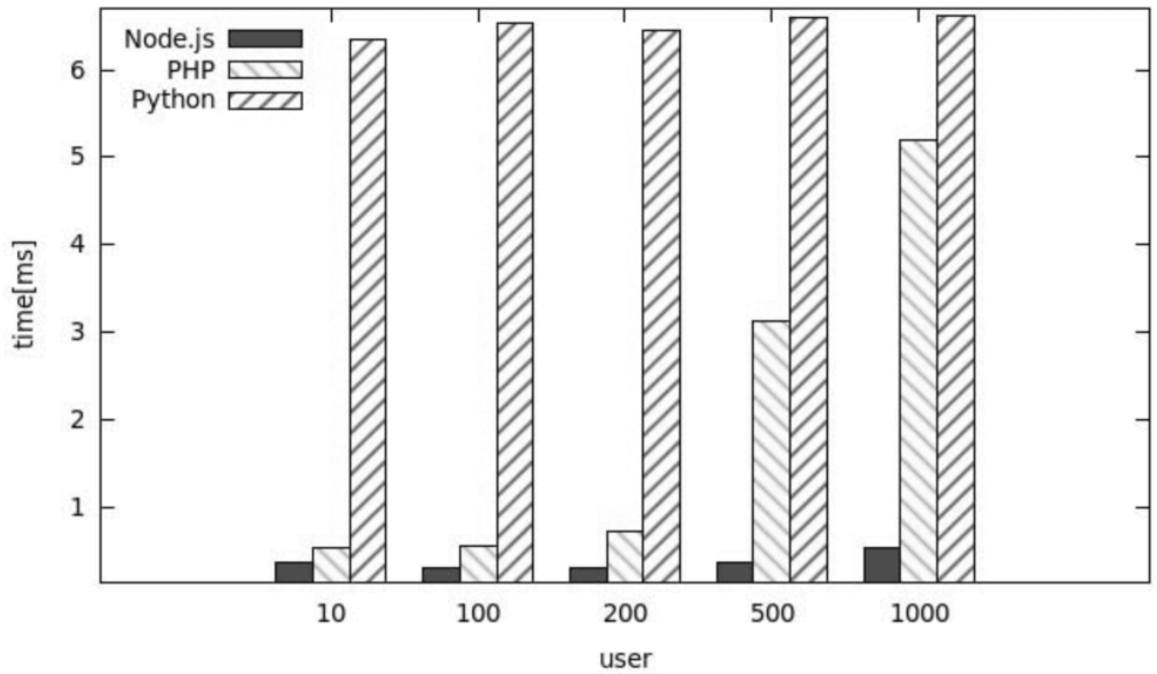

Figure 2.3: "Results for 'Select Operation of DB' mean time per request" (Lei, Ma, and Tan 2014, p. 665).

Due to this efficiency, Node's scalability is not in question, especially for web-based platforms (Ra et al. 2016). In fact, most opponents of Node focus their criticisms on older versions, such as the difference between Node.js and io.js, a long-time fork of Node, before their merger in 2015 (Florescu 2017; Foundation 2015), but these concerns have long since been addressed. And by separating the concerns of the logic and data layers from the presentation layer, we can further capitalise on Node.js's low latency operations and leave rendering web pages to the browser itself instead of the server.

2.3 Web Client

Where there is seemingly a lack of options for server-side JavaScript frameworks there is a paradox of choice for developers using JavaScript client-side. Many of the frameworks that are still around today were first released as early as 2010 such as AngularJs, Backbone.js, and Knockout whereas others are newer like Ember.js (2011), Meteor (2012), ReactJS (2013), and Vue.js (2014). Each framework has its strengths and weaknesses and so, for the purposes of this project, guidelines had to be established in order to sift through the competition. First, the framework had to be lightweight and flexible in order to adapt to as many technical requirements and situations as possible which immediately excluded full platforms like Meteor and downgraded strict MVC frameworks since only the view component was needed to complement Node.js. Second, it needed to have an appropriate level of buy-in from both industry leaders and the open-source community as the technology should be mature but not near expiry. This further eliminated newer and less popular frameworks like Ember.js, Vue.js, and Angular 2, as illustrated in Figure 2.4 below through the number of contributors to the codebase and the number of "stars". And as always, performance was a concern as a sluggish UI spells disaster for any project. In the end ReactJS was left standing as the best choice in the crowded field.

	AngularJS	Angular 2	ReactJS	Vue.js	Ember.js	Meteor.js
Definition	MVW framework	MVC framework	JavaScript library	MVC framework	MVC framework	JavaScript app platform
1st Release	2009	2016	2013	2014	2011	2012
Homepage	angularjs.org	angular.io	reactjs.net	vuejs.org	emberjs.com	www.meteor.com
# Contributors on GitHub	1,562	392	912	62	636	328
GitHub Star Rating	54,402	19,832	57,878	39,933	17,420	36,496

Figure 2.4: Comparison of Client-Side JavaScript Frameworks (Korotya 2017).

2.3.1 ReactJS

Flexibility, community, and performance: ReactJS possesses all three. Since it is a JavaScript library, ReactJS can be grafted onto an existing codebase to support specific features or pages or be used from the beginning as the primary structural concept. It has a widespread support, both in terms of it being created and used by tech giants like Facebook and Instagram and in terms of the open-source contributions to the project on GitHub with over 1,000 contributors to the codebase. And React surpasses all of its competitors with its performance, particularly through its use of a virtual DOM for HTML element manipulation instead of modifying the actual DOM when generating views. Table 2.1, Table 2.2, and Table 2.3 show benchmark comparisons of different frameworks rendering 1,000 <li> elements onto a web page in different web browsers.

#	ReactJS	AngularJs	Knockout	Raw
1	58	236	282	8
2	563	254	367	42
3	34	324	362	40
4	35	271	316	41
5	24	228	340	42
6	23	312	340	41
7	26	261	335	38
8	25	219	327	42
9	22	293	335	39
10	24	254	344	40
Average	83.4	265.2	334.8	37.3

Table 2.1: ReactJS Performance – Chrome 39.0.2171.95 (Harrington 2015).

#	ReactJS	AngularJs	Knockout	Raw
1	68	198	338	8
2	56	239	399	41
3	34	208	452	50
4	32	204	411	46
5	34	280	458	48
6	28	251	417	50
7	28	280	450	51
8	29	244	419	51
9	31	206	449	49
10	28	204	416	49
Average	36.8	231.4	420.9	44.3

Table 2.2: ReactJS Performance – Firefox 34.0.5 (Harrington 2015).

#	ReactJS	AngularJs	Knockout	Raw
1	51	82	138	16
2	37	121	193	65
3	28	114	187	65
4	26	120	171	65
5	26	117	182	66
6	30	118	175	62
7	25	120	173	64
8	28	114	173	66
9	26	113	173	63
10	26	117	178	64
Average	30.3	113.6	174.3	59.6

Table 2.3: ReactJS Performance – Safari 7.0.2 (Harrington 2015).

If we disregard the second test in Chrome as an anomaly, it is clear that ReactJS is superior in terms of rendering performance. But ReactJS is not a one-trick pony either as "React's real power lies in how it makes you write code" (Fedosejev 2015, p. 2). In fact, Facebook has a dedicated page to "thinking in React".[4] The ReactJS library helps structure the code and the JavaScript used in ReactJS is much closer to "vanilla" or plain JavaScript, both of which make it easier for developers read, develop, and maintain. And so while the learning curve and complexity of ReactJS is often listed as a disadvantage for the library, the same can be said for similar frameworks like AngularJs (Da-14 2016). React's satisfaction of all three of the guidelines above would qualify it for this project, however there is one additional feature which clearly differentiates it from competitors; it can also be used to create native mobile apps.

2.4 Mobile Client

Mobile app development has seen explosive growth alongside websites, as noted in chapter 1, however bridging the gap between the two main platforms, iOS and Android, carries serious technical challenges when trying to deliver a consistent experience across different devices. The primary reason for this difficulty is due to the two platform's dramatically different codebases and secondarily because of the different libraries available to support them.

Previous efforts have been made to consolidate the developer experience such as the .NET platform Xamarin. First released in 2011, Xamarin relies on C♯ to provide a

[4] https://facebook.github.io/react/docs/thinking-in-react.html

common language across iOS and Android development. Xamarin compiles the C♯ code into native code for each platform thereby providing a reusable codebase, especially when paired with a framework like MvvmCross, and a native app experience for the end user. And while Xamarin's performance compared to native apps could be debated endlessly (gregko 2013; Cheung 2015; Software 2016), Xamarin's use of the proprietary .NET framework (owned by Microsoft) limits its use outside of a full .NET stack in terms of a common language (Arora 2016). Additionally Xamarin is not free to use beyond personal projects and is not open-source.[5]

JavaScript-based frameworks like Meteor and Ionic have tried to remedy these problems by creating "hybrid" apps which rely upon JavaScript, HTML, and CSS like a standard web page. However, the downside of this approach is that the application is is run in a web view, i.e. an <iframe> for mobile apps (Meteor 2017). This means that native behaviors and styles are not applied automatically and so the end user's experience is inconsistent when compared to using native apps and the app's reception could suffer severely as a result (Anderson 2016; Abed 2016). That's where React Native comes in.

2.4.1 React Native

React Native allows for developers to write code in JavaScript and have it compiled to native apps for both iOS and Android platforms. React Native has additional flexibility in that native code can be injected into React Native to optimise components or React Native can be grafted onto an existing native app to contribute functionality (Facebook 2017c). This means that components can be customised according to the platform that they run on (Facebook 2017d) all while falling under a shared structural umbrella and development environment. React Native also contributes to mobile development through performance increases. It already shown to gain over native code in CPU, GPU, and memory management, such as in Calderaio (2017)'s comparison of Swift and React Native, through its separation of the UI and JavaScript threads, similar to what happens with web pages in ReactJS (Kol 2016). Additional perks of React Native include automatic reloading of JavaScript code so that developers do not have to wait for their code to compile when developing (Facebook 2017a). This kind of instant feedback is an incredible benefit to developers, especially when compared to traditional build time length related to mobile app development. In sum, React Native's approach of "learn once, write everywhere" instead of the concept of "write once, run anywhere" (Occhino 2015) perfectly suits this project's aims of code reusability across a common language while providing a shared experience for both developers and the end user.

[5] https://www.xamarin.com/compare-visual-studio

Chapter 3

System Analysis

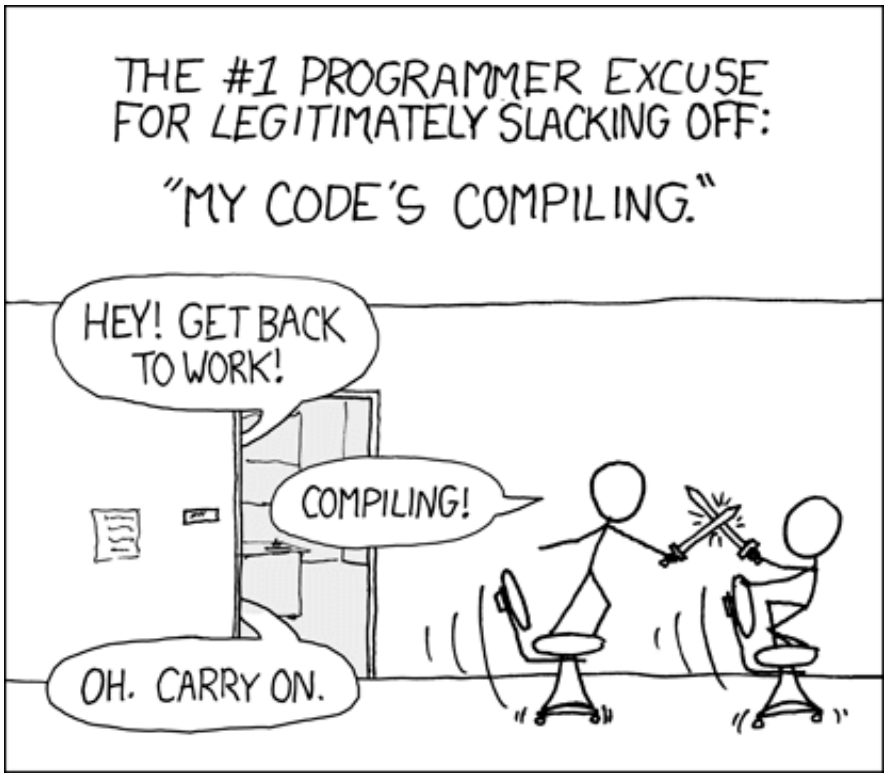

Figure 3.1: (Munroe 2007)

3.1 Inspecting the Site

Referring back to the questions posed earlier in section 2.1, it is now clear that JavaScript can indeed be used naturally across all three development environments and encompass server-side, browser-side, and mobile platforms with results equivalent to or exceeding other languages. The dialogue must now be shifted back to the primary question: how will employing a common language affect the software development life cycle? In answering that, the author believes that secondary questions will also be addressed as, in order to make the concepts discussed "production ready", these projects will need to be built out with code behind it to gain experience with the technologies and architecture and to, iron out the kinks. By taking into account business factors as well as technological concerns, the author hopes that the project will be applicable and responsive to real-world situations and that it will not become merely an ivory tower from which to look down upon.

3.2 Software Development Life Cycle

When designing any modern software architecture, both business and technical requirements need to be factored in and the software development life cycle needs to be examined holistically. For the purposes of this project, we are adhering to the following stages, as illustrated in Figure 3.2: 1) planning, 2) analysis, 3) design, 4) implementation, 5) testing & integration, and 6) maintenance.

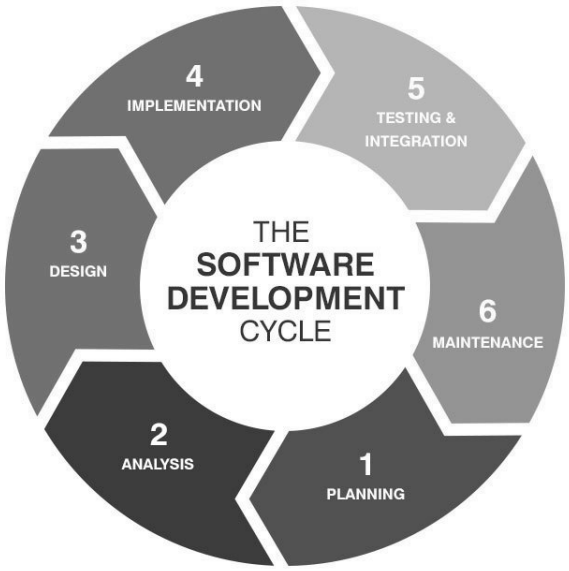

Figure 3.2: (Hussung 2016)

As this diagram demonstrates, the life cycle is cyclical and each step feeds into the other which means that both successes and failures overflow into the following

stages. This mirrors the real-world truism that software development and scheduling is complicated, in much the same way that no two snowflakes are alike: there are too many variables acting upon a single project. As Pressman (2014, p. 722) notes, there are a myriad of possible influences on a project including unforeseen technical difficulties, human difficulties such as a lack of experience, and aggressive/unrealistic deadlines. Because of this, unpredictability must be a key assumption in developing any plan (Pressman 2014, p. 69) and so, in order to overcome as many hurdles as possible yet remain flexible with malleable requirements across diverse situations, a standardised workflow must be incorporated to provide the backbone of structure in the software development life cycle. For the considerations of this project, a brief survey of software workflows will provide insight into possible use cases for the future management and development of this architecture.

3.2.1 Workflows

While the stages of the software development life cycle remain constant, their order and intensity can vary depending on the workflow being employed. In order to best emulate a real-world environment when looking at workflows, Ruparelia (2010, p. 8) finds it helpful to break down the software being developed into three categories:

> Category 1. Software that provides back-end functionality. Typically, this is software that provides a service to other applications.
>
> Category 2. Software that provides a service to an end-user or to an end-user application. Typically, this would be software that encapsulates business logic or formats data to make it more understandable to an end-user.
>
> Category 3. Software that provides a visual interface to an end- user. Typically, this is a front-end application that is a graphical user interface (GUI).

The problem with this project is that its components embody all three of these categories. This could make it difficult to find and adhere to a single workflow therefore consideration for development must be broken down into individual components in order to better manage development and schedules. Separating concerns this way on a management level should provide the same benefits as with coding: better scalability as the development team grows. And while there are a lot of different workflows being advocated, this paper will examine the traditional waterfall methodology as well as the modern Agile approach.

At its core, the waterfall method pursues a linear, stepwise flow where each cog in the machine does not move until the one before it finishes turning. In this way the project progresses from planning to analysis to design to implementation to testing & integration to maintenance. There are incremental steps forward or backward in more complex variations of the model but generally stages with more than one degree of

separation are not in communication with each other. As Ruparelia (2010, p. 9) notes, this workflow is the oldest and helps ensure sufficient documentation is produced at every stage through formalised processes but it is best suited to category 1 projects as it does not respond well to change.

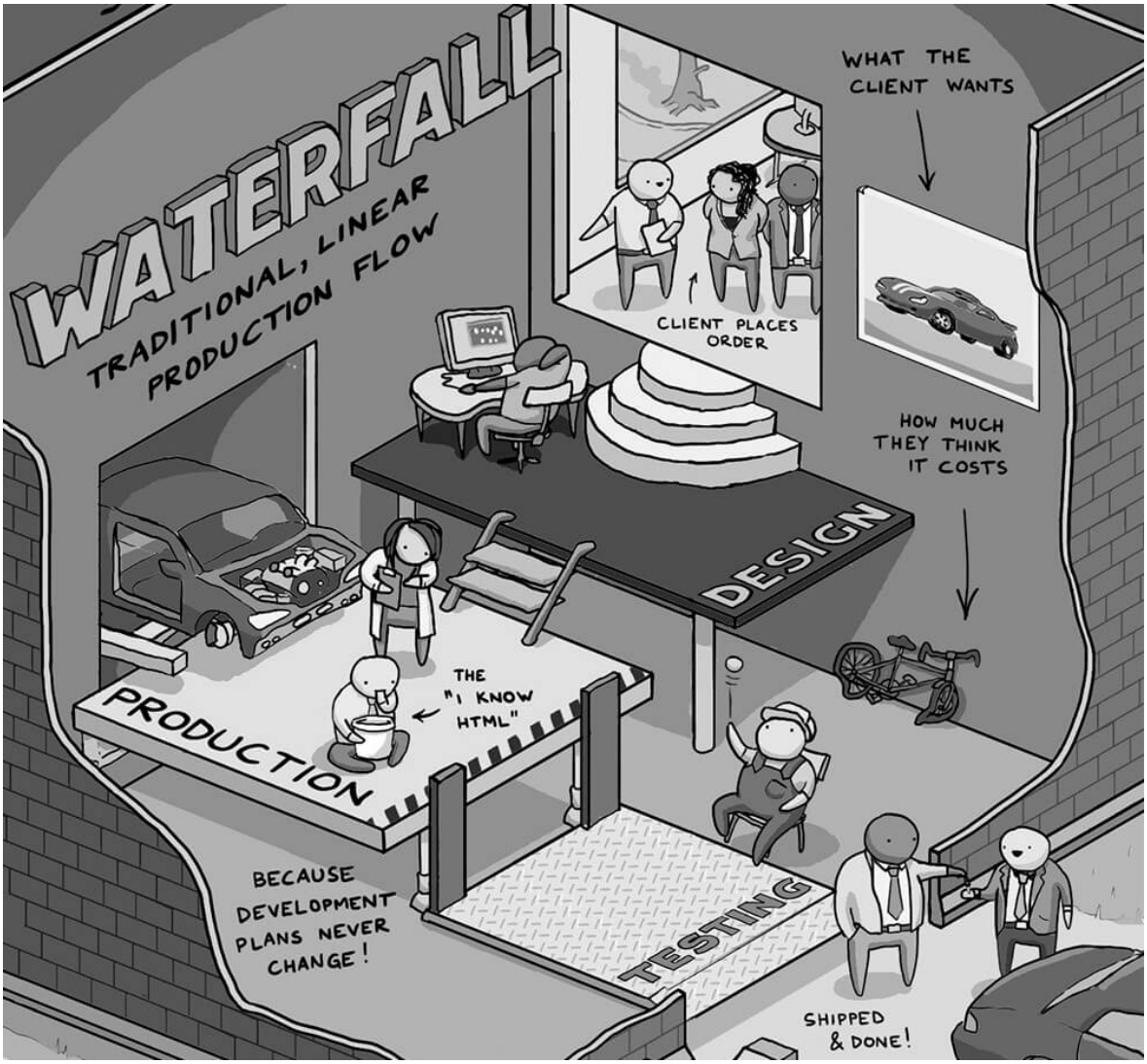

Figure 3.3: See Virkus (2016) for the full illustrated guide to software development workflows.

The Agile workflow is iterative and instead focuses on small, incremental changes and frequent software releases. It works well under the assumption of unpredictability, which (Pressman 2014) mentioned earlier, especially in regard to developing prototypes or proof-of-concept applications or when the client is unsure of what features they want in the software. It also provides greater transparency into the development process for the client and allows the development team to more easily and quickly respond to scope changes and feature requests. The downside is that Agile's small iterations

make it difficult to establish long-term plans and synchronise all of the components in large projects to the same schedule. It also de-emphasises documentation during development which is why Ruparelia (2010, p. 12) considers it more suitable to a category 3 application.

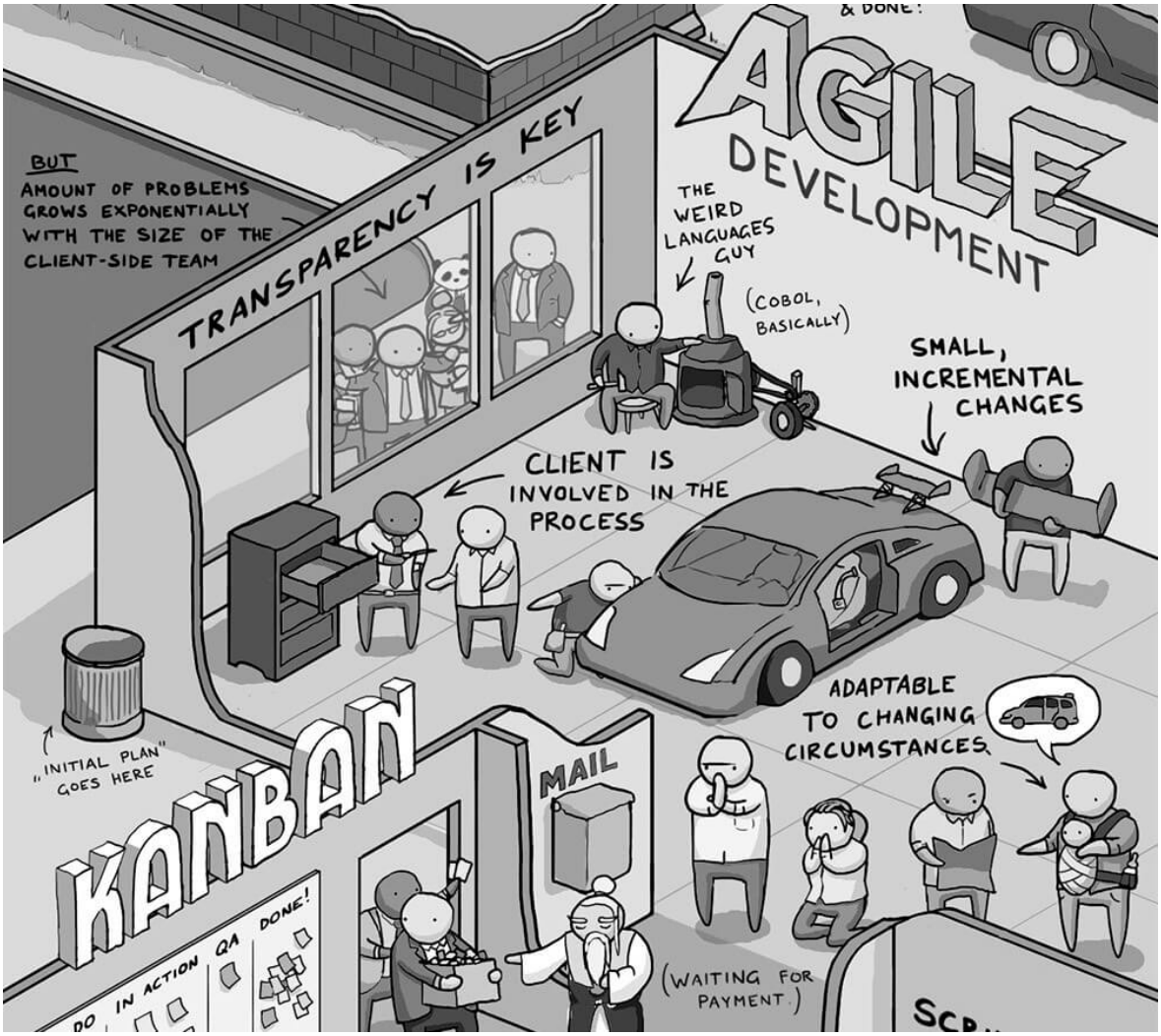

Figure 3.4: (Virkus 2016)

In the author's experience, these workflows tend to be the ones most frequently used across various small to medium-sized organisations. However, regardless of professed adherence to certain methodologies, there are elements of each with overlap with the other. For example, no matter how well planned a waterfall model may be, there is almost certainly scope creep and design changes introduced as the project progresses down the track. Similarly, there are some pieces of functionality in a software project which must occur serially and cannot be parallelised (login functionality for example) no matter how Agile the rest of the current iteration or future iterations may be.

3.3 Software as a Service

Despite the complications that may arise from strict adherence to a workflow, Agile methodologies have been embraced for the purpose of this project. This is due to ease of separating and modularising the components which aids in both long and short-term planning. This also helps push the overall conceptual sphere for this architecture more towards a Software as a Service (SaaS) model. In SaaS, a shared API is accessed by any number of device-agnostic clients, such as a web browser or mobile application (Wikipedia 2017d). This approach suits horizontal scaling, centralised administration, and the low-latency operations of an API technology like Node.js. This would position the architecture well for future business and technological advances such as adding analytics to the data or traffic and commercialising the platform.

3.4 Writing up the Building Contract

By examining possible use cases in future development and management a component-based, modular design, backed by a Software as a Service approach, has been established as the best path forward. The technologies involved have also been identified and selected in chapter 2. So now that the structure of the tower has been analysed, it is time to write up the building contract. When complete, this project aims to achieve all of the following:

- The software is responsive and flexible to both the business and technical requirements of any small to medium-sized company which includes writing bespoke code from scratch or inheriting an existing codebase as well as developing in-house or internal software or creating externally facing software
- The software is able to scale horizontally through a separation of concerns of the server-side API and the web and mobile client components
- The software is written in a common language and reuses common libraries and frameworks to streamline the developer experience
- The software provides a consistent experience to the end user across different client platforms

Chapter 4

System Design

Figure 4.1: (Munroe 2014c)

4.1 Laying the Foundations

As noted in section 3.1, in order to avoid making a purely academic construct, these ideas need to be materialised into tangible software. This exercise will also help refine these concepts further into concrete, measurable objects using industry standard concepts like DRY (don't repeat yourself) programming. Indeed, the conceptual research and implementational planning of the system architecture go hand in hand as higher level concepts will need to feed into technological limitations and vice versa. In this way, both the research and design components of this project have been approached in an iterative fashion.

4.2 Development Environment

In order to start addressing the contract details laid out in section 3.4, we need to start from the bottom up. Establishing a common development environment is crucial to standardizing the developer experience as well as providing entry points into code reuse and debugging/maintenance. One of the things required is an IDE which supports syntax inspections for all of the primary languages envisioned for the project, such as HTML, CSS, JavaScript, JSON, and JSX (ReactJS). For this purpose, the author has chosen WebStorm by JetBrains. Additionally, a single operating system (OS) would also benefit the team and the author would recommend macOS by Apple. It is important to note here that the WebStorm tool runs on many operating systems, as do a bounty of other IDEs, however sharing a common IDE and OS would make onboarding of new team members faster and debugging of local machines easier as all developers would be familiar with the setup. However, in reality these recommendations will probably be disregarded as developers opt for the own configurations which is not ideal but acceptable.

4.2.1 Version Control

One thing that the team will definitely share is a version control system (VCS). There are several different VCSs available like SVN, Mercurial, and Git. For this project, Git was chosen as a VCS due to the fact that Git uses a distributed model involving both a centralised, remote repository and a local repository, see Figure 4.2. These repositories can exist independently and only need to be synchronised when pushing updates from the local machine to the remote server. This feature more easily supports a geographically distributed team or situations where internet access is limited. Additionally there are many free Git repository services like GitHub and Bitbucket.

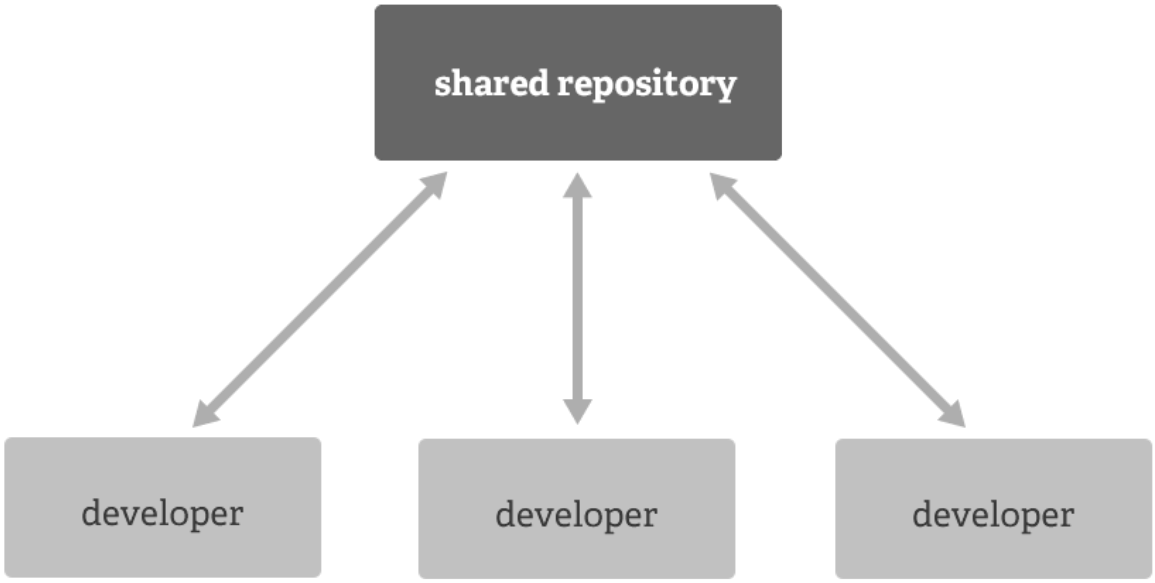

Figure 4.2: Distributed repository model in Git (Git 2017).

4.3 Deployment Environment

In addition to the development environment, guidelines are needed in regards to the deployment environment. Ubuntu 14 was chosen for this project as it is Unix-based and widely available in terms of hosting platforms like DigitalOcean and Amazon Web Services and for tutorials and walkthroughs for setting up server-side services like the web server and database.

4.3.1 Web Server

For web servers, IIS, Apache, nginx are the primary options. For this project, technical stability and comm were the biggest concern and open-source code as a plus so the decision came down to Apache and nginx (IIS is proprietary Microsoft code). In the end nginx was chosen over Apache given its demonstrated performance with Node.js (Chaniotis, Kyriakou, and Tselikas 2015).

4.3.2 Database

In regards to databases, the software architecture is agnostic as it was designed to be paired with any database. MySQL was chosen for the proof-of-concept due to its ubiquity in terms of tutorials, tools, and development resources. Additionally, a relational database was better suited for the prototype being developed as opposed to a

NoSQL solution like MongoDB as the author was more familiar with the technological concepts behind it.

4.4 Software Architecture

Using the Software as a Service model, the overall software architecture will be structured in two primary tiers, the server and the client/s. A secondary tier exists on the server between an API and a database and the clients can be broken down further into a web client, iOS client, and Android client. The duality between these primary and secondary tiers helps visualise the distinct separation of concerns while also recognising their relation to each other within the larger system.

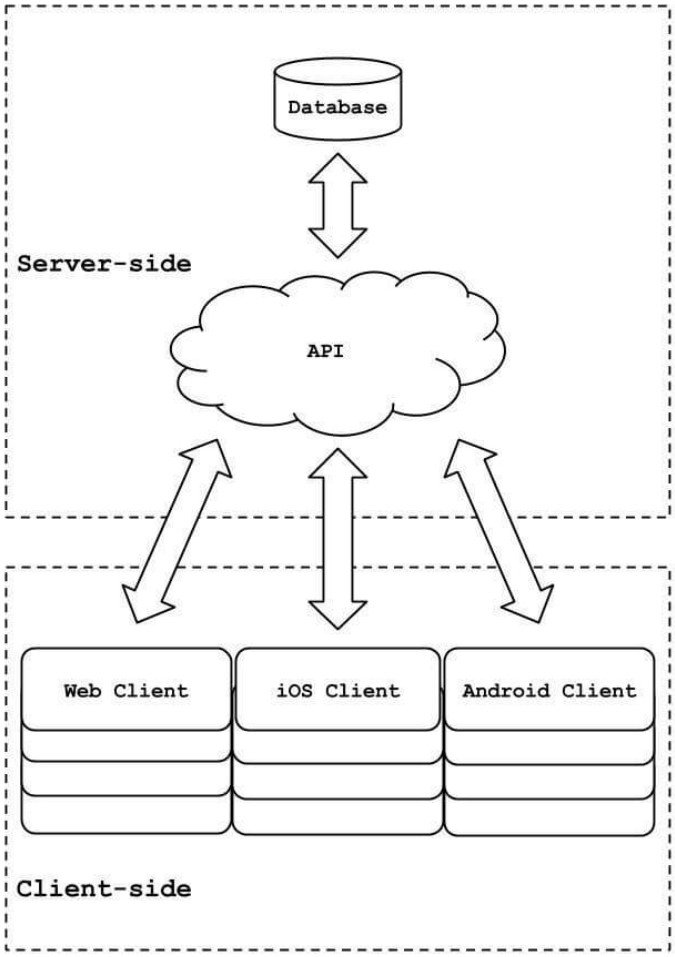

Figure 4.3: Project Components Arranged Using a Software as a Service Model.

Chapter 5

Implementation

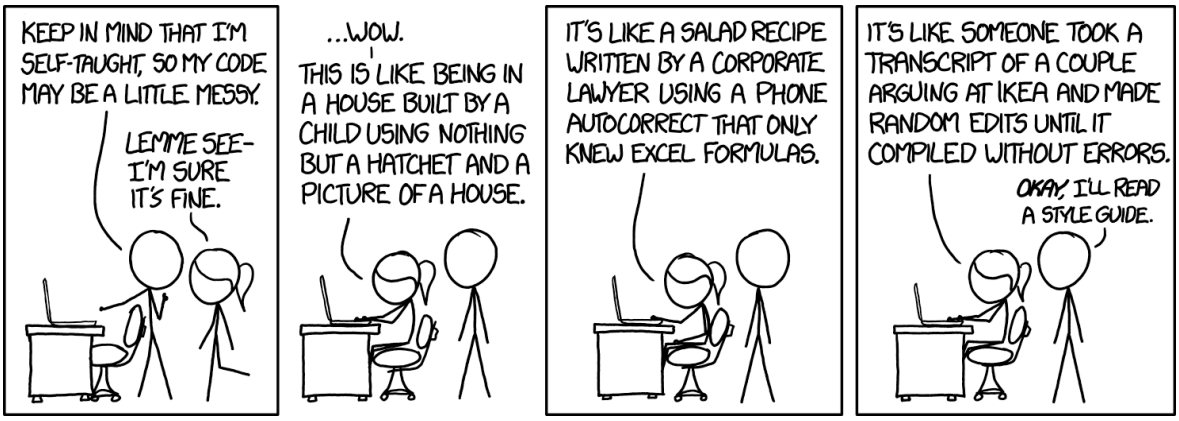

Figure 5.1: "I honestly didn't think you could even USE emoji in variable names. Or that there were so many different crying ones" (Munroe 2015).

In order to implement the design laid out in chapter 4, the primary repository for the project was broken up into three child repositories, called submodules in Git, for the server-side API, the web client, and the mobile client. In addition to the logical difference between them, they can also be generalised as separate repositories for Node.js (API), ReactJS (web), and React Native (mobile).

5.1 Server-side API

The API has six main components: controllers, middleware, models, routes, services, and tests, organised in the following folder structure:

```
|-- controllers
|   `-- ...
|-- middleware
|   `-- ...
|-- models
|   `-- ...
|-- routes
|   `-- ...
|-- services
|   `-- ...
|-- tests
|   `-- ...
|-- .env
|-- Procfile
|-- README.md
|-- package.json
`-- server.js
```

Listing 1: Folder structure – Server-side API

As illustrated in Listing 1, the API is structured according to an MVCS framework which is based on the model-view-controller (MVC) framework with an additional

service layer (Git 2008).[1] In the context of this project, the models relate to the data objects stored in the database like the user object, the controllers are essentially route handlers, and the services reflect the majority of the business logic. Middleware can be placed in between the client's request and the controller logic. Tests are discussed in chapter 6. Server.js, at the root of the project, is the primary entry point for the API.

An example of this logic flow would be as follows: a client makes a request to the API; server.js defines the routes in the routes folder which are then matched against the request; if the request matches a defined route url, the relevant controller is called; middleware like an authentication check may or may not be executed between the route matching and the controller call; the controller may call functions in the services to perform the logic required and then sends the results back to the client in JSON format.

Installing the server requires minimal effort:

```
$ gem install foreman
$ cp .env_example .env
$ npm install
$ npm install foreman nodemon -g
```

Listing 2: API Installation Script

Ruby is used to install foreman, then the environment variables configuration file is copied and ready for inputting the values. Then npm installs the project-specific dependencies and registers foreman and nodemon as global commands in the PATH (the -g option). After installation running the server is as simple as:

```
$ npm run dev
```

5.1.1 Configuration

The API is run on using Foreman. Foreman employs a Procfile to define the commands required to run the Node.js server. This allows for easy exportation when deploying to the remote server using a keep-alive daemon service like Upstart or Monit. Separate Procfiles are used for development and production environments.

Foreman also loads additional configuration specific to the inner workings of the application, stored in a key:value pair format, in the .env file. This serves two purposes: 1) it is easier than manually entering all of the configuration pairs in the command

[1] Note that the API is lacking a 'V' or view component as it has been separated into the stand-alone web and mobile clients.

when starting the server and 2) it allows configuration to be more easily tracked in the VCS and communicated to other team members, i.e. it is more human readable.[2]

5.1.2 Authentication

User login and registration authentication is done through Passport.js using a standard username and password login. In this case, the username is the user's email address which is set to be a primary key in the database so the application does not have to worry about users with matching email addresses. When the email and password are sent to the API, the database is searched for a user with that email address. If found, the password is hashed using the per-user salt saved in the database. If the passwords match, the user is logged in and Passport.js generates an authentication token for that user's session which is sent back upon successful login. That token is then sent on future requests as a bearer token in the header. On each request Passport.js validates that the token is still valid as it has an expiry baked into it. If the token is valid, then the request continues to the controller otherwise a 401 unauthorised response is sent back to the user.

5.2 Web Client

The web client was created with Facebook's Create React App boilerplate tool.[3] This system is designed to create apps in ReactJS without requiring the developer to specify any build configuration. This saves time during both setup and development as there is an existing knowledge base and community which uses this tool and can help with any problems. In addition, it includes high-powered tools like Webpack which bundles all sorts of supplemental logic and helper tools like JSLint for linting the JavaScript code while developing (Politz et al. 2015, p. 2). The web client has two main folders, /public and /src. /public is used for files and assets that lie outside of the ReactJS ecosystem like the index.html. /src is further broken down into the /app folder which contains the core of the ReactJS code and UI-specific assets like fonts, images, and styling. Listing 3 shows the directory structure in the application.

[2]Note that the real .env is not tracked in Git but .env_example is which has the configuration key names but not their values.

[3]https://github.com/facebookincubator/create-react-app

```
|-- public
|   `-- ...
|-- src
|   |-- app
|   |   |-- components
|   |   |   `-- ...
|   |   |-- ducks
|   |   |   `-- ...
|   |   |-- layouts
|   |   |   `-- ...
|   |   `-- services
|   |       `-- ...
|   |-- css
|   |   `-- ...
|   |-- fonts
|   |   `-- ...
|   |-- images
|   |   `-- ...
|   |-- sass
|   |   `-- ...
|   |-- index.js
|   |-- router.js
|   `-- store.js
|-- README.md
|-- bower.json
`-- package.json
```

Listing 3: Folder structure – Web Client

Installing the web client only requires one line, assuming Ruby and Sass are already installed on the machine:

```
$ npm install
```

All that is needed is for npm to install the project-specific dependencies. After installation, running the web client in development mode only needs a single command:

```
$ npm run dev
```

npm scripts are defined in the package.json file at the root of the project. Through a single npm command, several subcommands are actually being run concurrently including a CSS and JS file watcher and the UI storybook runner.

5.2.1 Styling

The CSS preprocessor Sass is used for styling in this project. Using Sass produces much DRY-er code than just using plain CSS through the use of features like mixins which are essentially helper functions for common styling and style inheritance like a standard object-oriented language. For the proof-of-concept application, a flexbox grid framework was implemented to match the mobile client's UI as closely as possible.

5.2.2 Data Flow

Since the "logic" of the web client deals entirely with managing and organising data received from the API, a dedicated state container called Redux was introduced. Redux adds structure to data and the concept of a session by saving the information in an object tree. Actions are dispatched from UI components, like the login form being submitted. A reducer then receives that action, performs the required logic like making an API call and then updates the data tree. In this way the tree is never modified directly but cloned and mutated upon every update. This means that a copy of every state of the tree is available as a reference for debugging or for "hot reloading" a certain state. And since the state is completely based on the client-side, the store is saved to local storage on the user's browser so that their session can continue if the page is refreshed. Figure 5.2 shows how data flows with and without Redux.

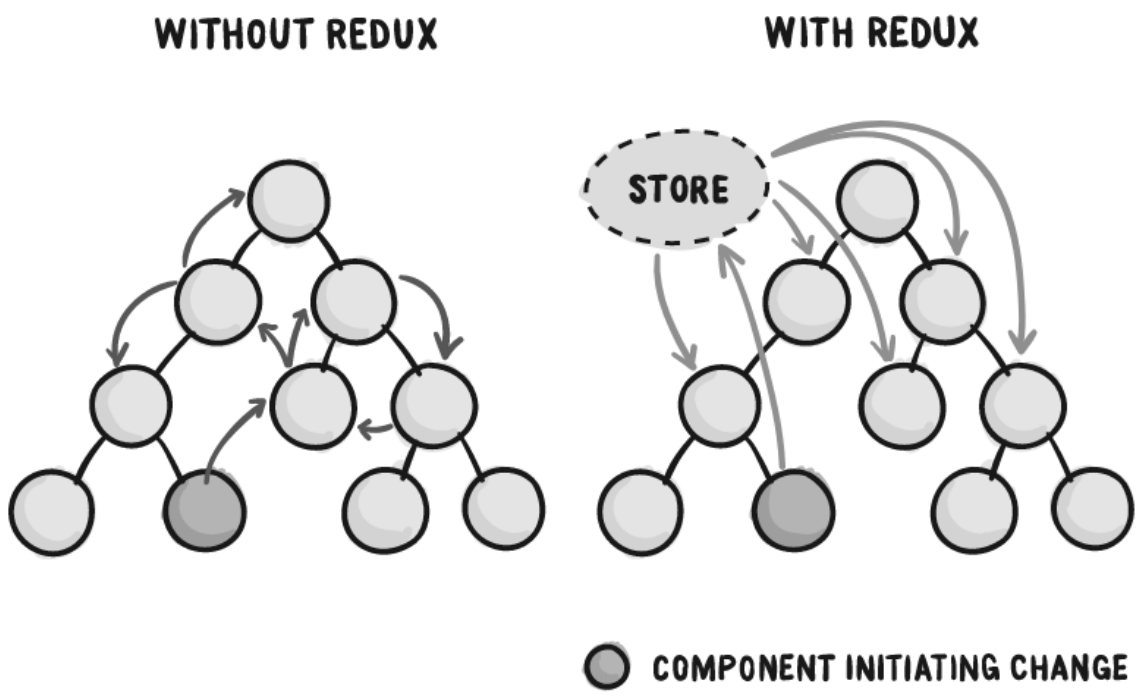

Figure 5.2: Data Flow with and without Redux

"The store can be thought of as a 'middleman' for all state changes in the application. With Redux involved, components don't communicate directly between each other, but rather all state changes must go through the single source of truth, the store" (Westfall 2016).

5.3 Mobile Client

The mobile client was similarly created using Facebook's Create React Native App utility.[4] Like the Create React App tool, Create React Native App is designed to provide developers with a development environment with no build configurations. The project is broken down into three main folders: /android, /ios, and /src which contains common code used by both platforms.

[4] https://github.com/react-community/create-react-native-app

```
|-- android
|   `-- build
|-- ios
|   `-- build
|-- src
|   |-- components
|   |   `-- ...
|   |-- ducks
|   |   `-- ...
|   `-- services
|-- README.md
|-- app.json
|-- index.android.js
|-- index.ios.js
|-- package.json
`-- router.js
```

Listing 4: Folder structure – Mobile Client

Installing the mobile client also only requires one line, after the virtual machines for each environment have been set up that is:[5]

```
$ npm install
```

Running the code requires two commands, one for Android and one for iOS.

```
$ npm run android
```

and

```
$ npm run ios
```

[5] https://facebook.github.io/react-native/docs/getting-started.html

5.3.1 Ducks

Since both the web client and the mobile clients share so much of the same functionality and, thanks to a common language and shared libraries, the author was able to reuse the code for communicating with the API via AJAX, called ducks.[6] Using Git submodules, the ducks repository was installed as a dependency for both the web and mobile clients. They were then able to access the same requests and get the same responses while developers only had to write the code once for it to be propagated to both clients.

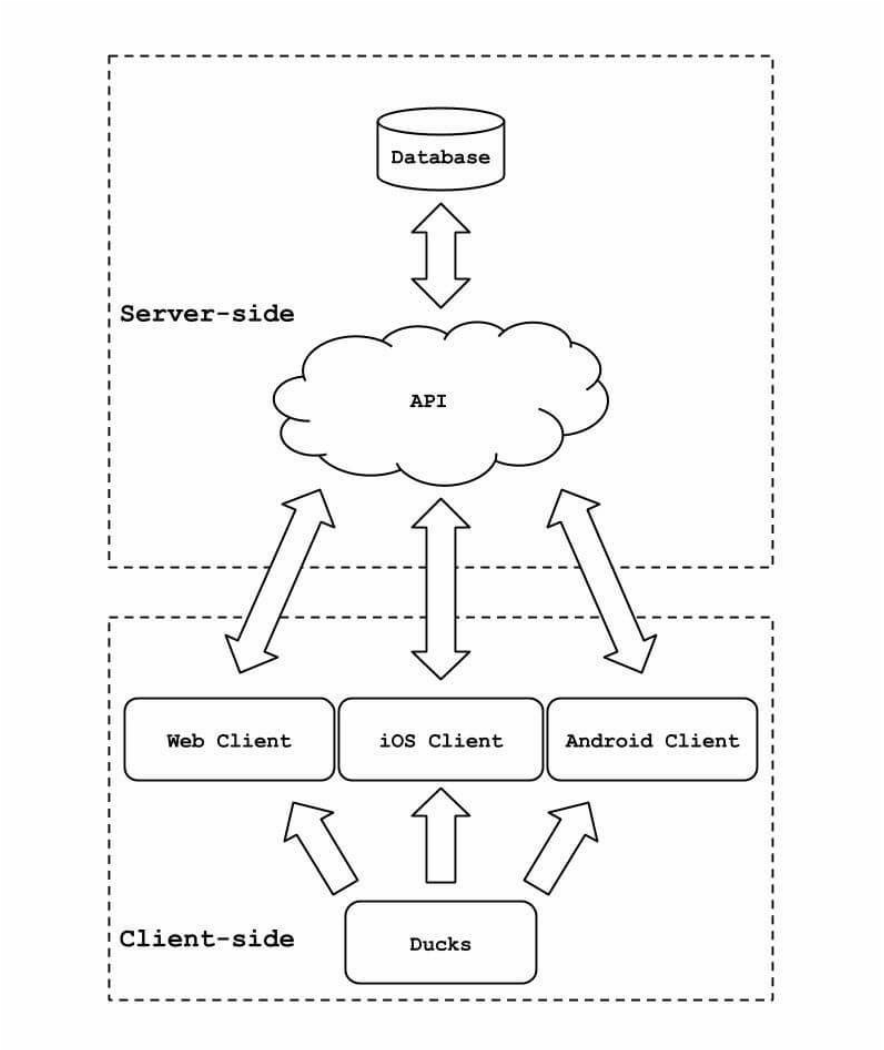

Figure 5.3: Sharing Code between Web and Mobile Clients

[6]"Java has jars and beans. Ruby has gems. I suggest we call these reducer bundles 'ducks', as in the last syllable of 'redux' "(Rasmussen 2017)

5.4 Deployment & Build Process

The deployment and build process is very straightforward for all three components and has been documented in each of the component's ReadMe files.

5.4.1 API

As mentioned in section 4.3, the API will be deployed on an Ubuntu 14 server using Upstart. So after the initial server setup, including installing Node.js, Git, MySQL, and nginx, the deployment script is very simple:

```
$ git pull
$ npm install
$ npm run export
$ sudo stop foreman
$ sudo start foreman
```

Listing 5: API Deployment Script

Git pulls the latest code, npm installs any new or updated dependencies and then exports the config to the Upstart daemon called foreman. The foreman daemon is then stopped and started to refresh its cache of the codebase.

5.4.2 Web

The web client can be deployed anywhere, including being used on a local machine as CORS issued have already been addressed with the API, but the author is assuming a deployment on an Ubuntu box similar to the API. So, after a web server like nginx has been installed, the following commands can be run:

```
$ git pull
$ npm install
$ bower install
$ npm run build
```

Listing 6: Web Deployment Script

Git pulls the latest code, npm and bower install any new or updated dependencies, and then npm compiles the ReactJS project into the /build folder. All CSS and JavaScript files are concatenated and minified into a single CSS and JavaScript file and loaded through the index.html file. Any remaining assets that cannot be concatenated like fonts are also copied to the /build/static folder so that the entire web application is stored within the /build folder. Then all that is needed is to point the web server to the correct path on the server like so:

```
server {
    listen 80;
    root /var/www/thesis/web/build;
    index index.html index.htm;
    server_name www.thesis.com;
    access_log /var/log/nginx/thesis.access.log;
    error_log /var/log/nginx/thesis.error.log;
    location / {
        try_files $uri $uri/ /index.html?q=$uri&$args;
    }
}
```

Listing 7: Sample nginx Configuration for a ReactJS Deployment

5.4.3 Mobile

While React Native does wonders for initial setup and commonising the development platform, building and deploying mobiles are still very dependent on the idiosyncrasies of their respective app stores.[7]

For iOS apps, developers need to follow the usual validation steps to publish on the Apple App Store[8] and also manually edit the Info.plist file in the ios folder to disable HTTP traffic to localhost (enabled by default for React Native (Facebook 2017e)). Then the app can built using the React Native CLI:

```
$ react-native run-ios --configuration Release
```

[7] Note, for both platforms the app is bundled into the /build folder in the respective subdirectory, e.g. /android/build

[8] https://developer.apple.com/library/content/documentation/IDEs/Conceptual/AppDistributionGuide/SubmittingYourApp/SubmittingYourApp.html

Android requires a similar amount of work to satisfy the minimum requirements including generating a signing key (Facebook 2017b).[9] But after that is done, the React Native CLI only requires a single command to build the app:

```
$ react-native run-android --variant=release
```

[9] https://developer.android.com/studio/publish/index.html

Chapter 6

Testing & Evaluation

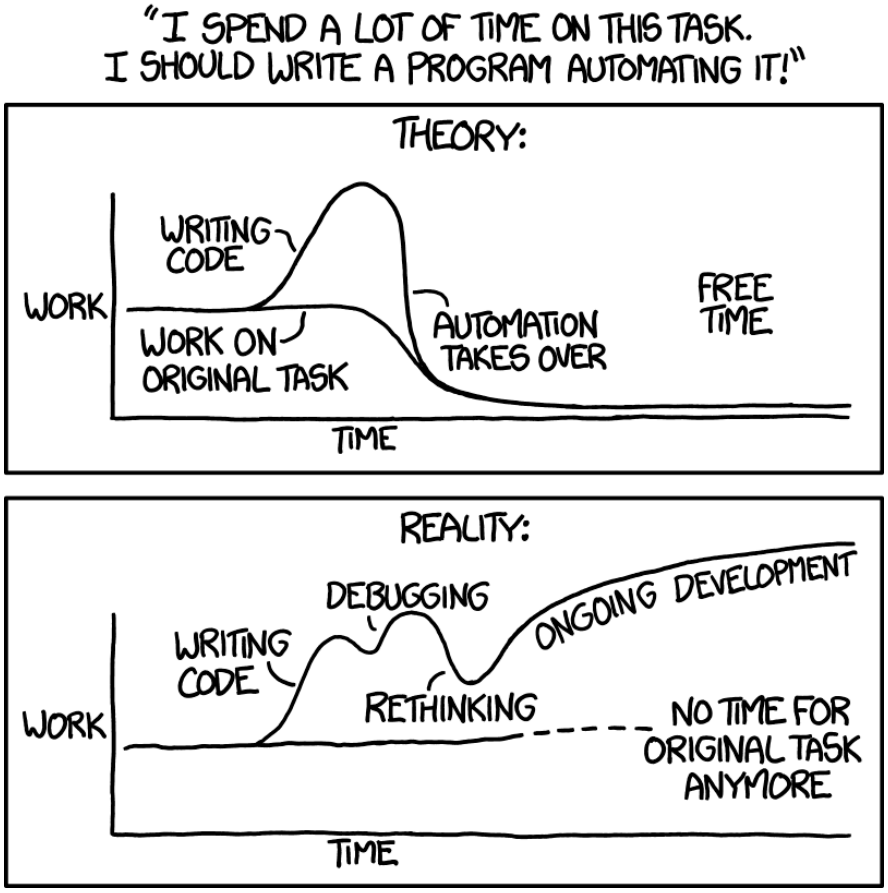

Figure 6.1: " 'Automating' comes from the roots 'auto-' meaning 'self-', and 'mating', meaning 'screwing' " (Munroe 2014a).

6.1 Automated Documentation

As part of satisfying both the technical and business requirements for a project, documentation is needed to keep everyone on the team up to date and working from the same source of truth. In the author's experience, documentation usually has been the first thing eschewed whenever timelines get tight but the lack of this formalised knowledge ultimately causes mistakes thereby paradoxically causing delays. As a result, the author recognises the importance of incorporating time into a project for this task but also understands the realities of finding that time, especially when rapidly prototyping like in an Agile development cycle. One solution to this problem is to include automated documentation in the codebase from the start, especially for I/O-heavy components like the API.

6.1.1 Swagger

Tools like Postman already exist for testing APIs but they lack the ability to respond to changes in the code in real-time, i.e. they're dependent on a manual setup and export in order to be shared with others. Swagger solves this problem by integrating directly into into the codebase. Using JSDoc comment blocks for metadata, Swagger generates a detailed HTML page which documents route definitions and is also able to be used for testing. This helps ease the issue of the documentation being out of date as the JSDoc definitions are in close proximity to the actual code and the UI is rendered anew upon every request for the API documentation. Listing 8 shows a comment block describing the admin-authenticated route to display all of the users in the system and Listing 9 illustrates the user object model. The results of this can be seen and explored in the Swagger UI, as Figure 6.2 shows.

```
/**
 * @swagger
 * /admin/users/{page}/{limit}:
 *   get:
 *     tags:
 *       - Users
 *     description: Returns paginated list of users
 *     parameters:
 *       - in: path
 *         name: page
 *         type: integer
 *         required: true
 *         description: Page of results to get.
 *       - in: path
 *         name: limit
 *         type: integer
 *         required: true
 *         description: Number of results to get per page.
 *     responses:
 *       200:
 *         description: An array of users
 *         schema:
 *           $ref: "#/definitions/User"
 *       401:
 *         description: Unauthorized
 */
router.route('/users/:page/:limit').get(adminController.getUsers);
```

Listing 8: Swagger Route Documentation

```
/**
 * @swagger
 * definition:
 *   User:
 *     properties:
 *       id:
 *         type: integer
 *       role:
 *         type: string
 *       firstName:
 *         type: string
 *       lastName:
 *         type: string
 *       email:
 *         type: string
 *       dateCreated:
 *         type: timestamp
 *       lastModified:
 *         type: timestamp
 *          ...
 */
function mapToSchema(result){
    return {
        id: result.id,
        role: result.role.toUpperCase(),
        ...
    };
}
```

Listing 9: Swagger Model Documentation

Figure 6.2: Visualising the API with Swagger UI

6.1.2 UI Storybook

In much the same way that the API documentation can have life breathed into it, UI components for ReactJS and React Native can be incorporated into a React storybook.[1] Storybooks are a "development environment for UI components" which essentially means a collection of UI elements, classes, colors, etc., much like a traditional style

[1] https://github.com/storybooks/storybook

guide. Additional features are also available like UX components such as navigation to create a narrative, much like the stand-alone UI/UX application InVision. This allows all of the developers, designers, and clients work from the same visual sources. In addition, when code is changed within the application it is reflected automatically in the storybook since it uses the exact same components as the regular app. Like Swagger, storybooks aid in automated documentation and knowledge sharing by providing a common point of reference for everyone involved on the project.

6.2 Automated Testing

Similar to automated documentation, automated testing helps ensure that work that has already been done does not get overwritten or changed and cause issues for other components relying on that code. And similar to documentation, automated testing does not need to wait until the end of the development cycle or the completion of the project itself to be incorporated into the codebase.

6.2.1 Unit Tests

Unit tests are measuring discrete pieces of functionality to ensure that, given a certain input, the expected output has not changed. For this, the author chose to use Mocha, Chai, and Sinon due to the wealth of resources documenting their use with Node.js. Within this setup, Mocha is the testing framework, Chai is an assertion library which helps make the tests more human readable, and Sinon is a library which makes testing AJAX functionality easier. Unit tests can cover a range of functionality from the smallest building blocks like individual functions to larger elements like the login or register functionality. Below are some unit tests set up to test common functions used across the codebase. Listing 10 illustrates the readability of the test code and Listing 11 shows the simplicity of running the tests.

```
const chai = require('chai'),
    assert = chai.assert,
    should = chai.should(),
//services
    utilService = require('../../services/util');

describe('Tests for UtilService', function() {
    describe('the isValueNotNull function', function() {
        it('should detect undefined keys', function() {
            utilService.isValueNotNull({},'key').should.equal(false);
        });
        it('should detect null objects', function() {
            utilService.isValueNotNull(null,'key').should.equal(false);
        });
        it('should detect undefined objects', function() {
            utilService.isValueNotNull(undefined,'key').should.equal(false);
        });
        it('should detect non objects', function() {
            utilService.isValueNotNull('object','key').should.equal(false);
        });
    });

    describe('the getValueByKey function', function() {
        it('should return empty string on failure', function() {
            utilService.getValueByKey({},'key').should.equal('');
        });
        it('should return key on success', function() {
            utilService.getValueByKey({key:'value'},'key')
            .should.equal('value');
        });
    });

    describe('the getFirstValueByKey function', function() {
        it('should return empty string on failure', function() {
            utilService.getFirstValueByKey({},'key').should.equal('');
        });
        it('should return first element of value array', function() {
            utilService.getFirstValueByKey({key:['value1','value2']},'key')
            .should.equal('value1');
        });
    });
});
```

Listing 10: Writing Unit Tests with Mocha, Chai, & Sinon

```
$ npm test
> ThesisAPI@0.0.1 test /thesis/api
> mocha tests/**/*.js

  Tests for UtilService
    the isValueNotNull function
      ✓ should detect undefined keys
      ✓ should detect null objects
      ✓ should detect undefined objects
      ✓ should detect non objects
    the getValueByKey function
      ✓ should return empty string on failure
      ✓ should return key on success
    the getFirstValueByKey function
      ✓ should return empty string on failure
      ✓ should return first element of value array

  8 passing (10ms)
```

Listing 11: Results of Running Unit Tests

6.2.2 End-to-End Testing

Although not incorporated into the proof-of-concept application built using the software architecture proposed in this project, end-to-end testing is also important for maintaining consistency across UI components and user's devices. Selenium is one solution which allows programmers to automate users' behavior across different browsers and devices. PhantomJS is a similar, headless testing tool which means that a browser is not required. This can be good to check common assumptions that are not dependant on the user's browser or device.

Chapter 7

Conclusions & Further Work

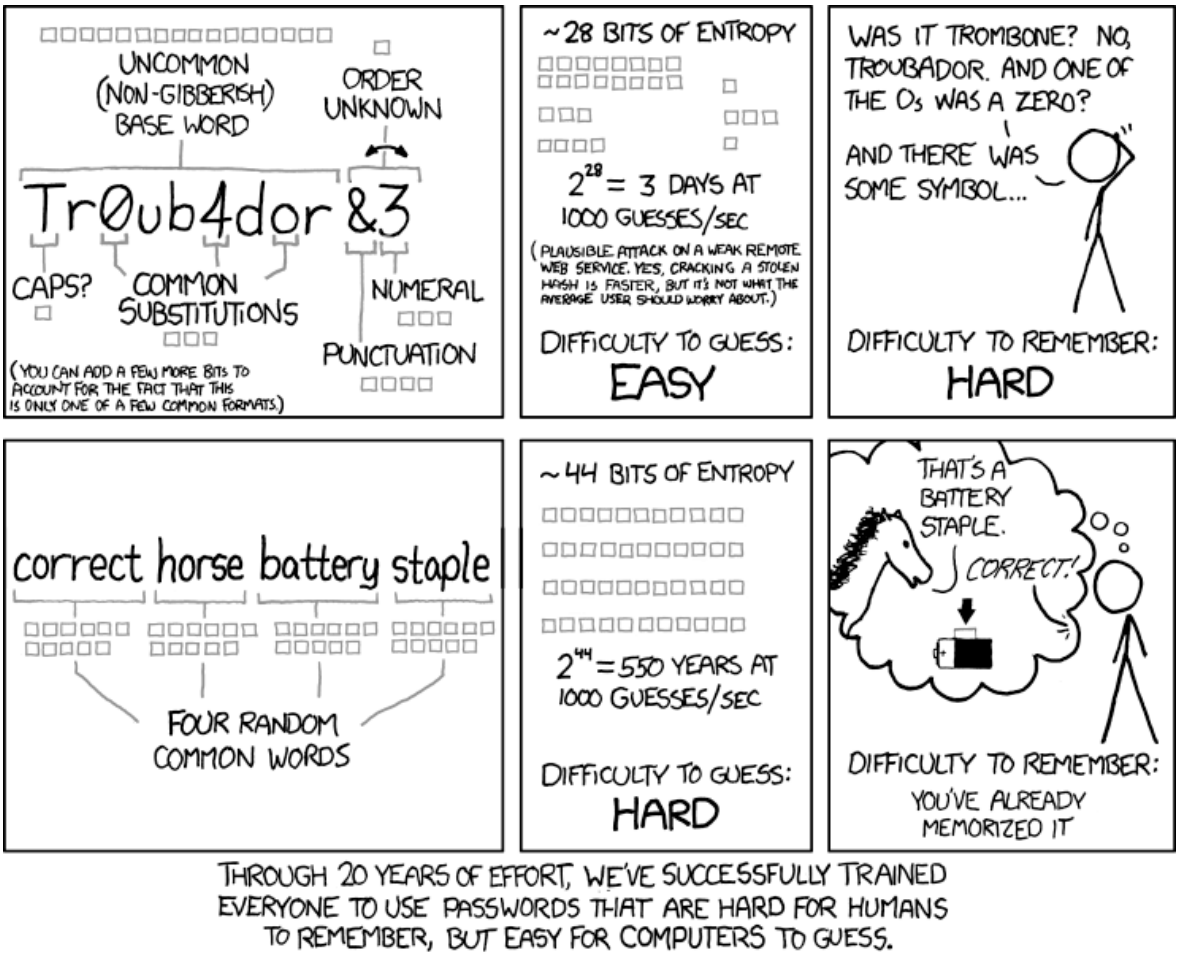

Figure 7.1: (Munroe 2011a)

The best-laid plans of mice and men often go awry, as Figure 7.1 above points out, and this project was no exception. Complications arose and time grew short as the project progressed on its twelve week schedule, as detailed in Appendix I. Below are the author's notes regarding the development of this project overall.

7.1 Complications

Due to the author's previous experience as a web developer, researching and integrating the technologies that comprise the system was not an issue. However, despite possessing experience with JavaScript and several other JavaScript libraries and frameworks like AngularJS, the author found difficulty when first using ReactJS and, subsequently, React Native. ReactJS and Redux's design patterns present a steep learning curve before development time is saved (Da-14 2016) however by the end of the project there was a noticeable positive effect on the effort expended on development and maintenance.

Another complication was the inability to reuse styling across the web and mobile platforms. This meant that common elements like button styles needed to be recreated across both environments and maintenance and debugging suffered as a result. In addition, the language employed was different with Sass being used on the web client and JavaScript on the mobile client.

Adherence to standards also got looser as time wore on. On the server-side, the separation between models and services became muddied as a traditional object-oriented approach gave way to the convenience of having both the object definition and the actions associated with it in the same file. The author believes this overlap might be reduced if JavaScript was more tightly typed or if more time was available.

7.2 Next Steps

In regards to developing the application built out as a proof-of-concept, the author would have liked to work on the following features:

- Add Google & Facebook OAuth 2 authentication for user login and register
- More fully explore front-end development, styling, and user interaction events using ReactJS and React Native
- Find a way to share styling between the ReactJS and React Native codebases like what was done with the ducks services

The author also wanted to further investigate the following software architecture improvements and concepts:

- Pursue the Software as a Service model further and refactor/modularise core elements of the back-end and front-end codebases into separate repositories to allow for additional apps to be built and to push updates to all projects using these modules simultaneously
- Implement holistic benchmark testing against "traditional" tech stacks with similar software architecture and concepts to quantify performance enhancements
- Investigate horizontal scalability of each component through load testing
- Explore the idea of changing JavaScript from a loosely-typed language into a stronger-typed one and the performance related to that switch (Ahn et al. 2014)
- Set up automated deployments through a service like Vagrant or Docker
- Automate code updates through a service like Jenkins

7.3 Summary

In sum, the author believes that this project was successful according to the goals laid out in section 3.4. The software architecture is flexible and applicable to a variety of situations that a small to medium-sized company might encounter including using this system as a stand-alone base on a new project or grafting it onto an existing codebase. The separation of concerns and modularisation of each component allows for both developer and deployment scalability. The software employs a common language and reuses libraries, frameworks, and design patterns across all codebases and the client-side software provides a consistent user experience to the developer as well as the end user. The attributes closely linked the back-end and front-end without being prescriptive or restrictive and will greatly aid in future prototyping, development, and maintenance. While the end results of this project are difficult to discretely quantify, the author has created a template which further iterations can refine and hone to best tailor to the scope of any given project. To this end, the elastic, one-size-fits-most solution outlined in this project will better connect business timelines with development idiosyncrasies by providing common ground between developers and reducing the amount of technical unknowns faced when starting a new project. Finding a common language for development may have started out as a myth but this project has demonstrated that such a construct can very much exist and is not simply babel-ing.

Appendices

I Project Planning

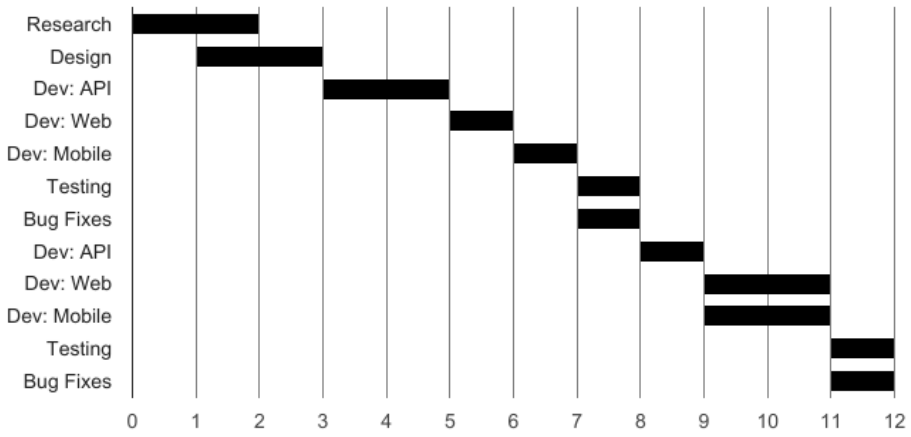

Figure 7.2: Project schedule visualised via a Gantt chart across 12 weeks

Since this project is largely concerned with software architecture and higher level development workflows, the lion's share of the time for this project was invested in research, planning, and API development. Ideally the API provides all of the data that the clients consume through a Software as a Service approach and so it needed to be robust and well designed for scalability reasons. Less effort was expended on front-end technologies however the quality was maintained across components as the end goal remained the same: to provide a platform for further development, regardless of front-end or back-end distinction. Testing and bug fixes were also given a decent allocation of time as they provide opportunities to reflect in preparation for the next iteration/s.

II Project Repositories

Each repository has a README.md file at its root with instructions on how to set up a development environment and how to deploy the component.[1]

- Thesis (Master repository) – https://github.com/mike-stumpf/ITB-Thesis
- API – https://github.com/mike-stumpf/ITB-Thesis.API
- Web – https://github.com/mike-stumpf/ITB-Thesis.Web
- Mobile – https://github.com/mike-stumpf/ITB-Thesis.Mobile
- Ducks – https://github.com/mike-stumpf/ITB-Thesis.Ducks

[1] These instructions have been written and tested only with a macOS environment in mind

Bibliography

Abed, Robbie (2016). *Hybrid vs Native Mobile Apps – The Answer Is Clear*. [Accessed 2017-06-05]. URL: https://ymedialabs.com/hybrid-vs-native-mobile-apps-the-answer-is-clear/.

Ahn, Wonsun et al. (2014). "Improving JavaScript Performance by Deconstructing the Type System". In: *SIGPLAN Not.* 49.6, pp. 496–507. ISSN: 0362-1340. DOI: 10.1145/2666356.2594332. URL: http://doi.acm.org/10.1145/2666356.2594332.

Anderson, Clayton (2016). *Why You Should Consider React Native For Your Mobile App*. [Accessed 2017-06-05]. URL: https://www.smashingmagazine.com/2016/04/consider-react-native-mobile-app/.

AppBrain (2017). *Number of Android applications*. [Accessed 2017-05-27]. URL: http://www.appbrain.com/stats/number-of-android-apps.

Arora, Sunil (2016). *App Development: Cordova vs React Native vs Xamarin vs DIY*. [Accessed 2017-06-05]. URL: http://noeticforce.com/mobile-app-development-cordova-vs-react-native-vs-xamarin.

Bethke, Erik (2003). *Game Development and Production*. Wordware Publishing, Inc.

Brooks, Frederick (1975). *The Mythical Man-Month*. Addison-Wesley Pub. Co.

Calderaio, John A. (2017). *Comparing the Performance between Native iOS (Swift) and React-Native*. [Accessed 2017-06-05]. URL: https://medium.com/the-react-native-log/comparing-the-performance-between-native-ios-swift-and-react-native-7b5490d363e2.

Chaniotis, Ioannis K., Kyriakos-Ioannis D. Kyriakou, and Nikolaos D. Tselikas (2015). "Is Node.js a viable option for building modern web applications? A performance evaluation study". In: *Computing* 97.10, pp. 1023–1044. ISSN: 1436-5057. DOI: 10.

1007/s00607-014-0394-9. URL: http://dx.doi.org/10.1007/s00607-014-0394-9.

Cheung, Harry (2015). *Mobile App Performance: J2ObjC vs Xamarin vs RoboVM vs RubyMotion*. [Accessed 2017-06-05]. URL: https://medium.com/@harrycheung/cross-platform-mobile-performance-testing-d0454f5cd4e9.

Cisco (2017). *Cisco Visual Networking Index: Global Mobile Data Traffic Forecast Update, 2016–2021 White Paper*. [Accessed 2017-05-27]. URL: http://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/mobile-white-paper-c11-520862.html.

Crockford, Douglas (2001). *JavaScript: The World's Most Misunderstood Programming Language*. [Accessed 2017-06-01]. URL: http://www.crockford.com/javascript/javascript.html.

Cuomo, Jerry (2013). *JavaScript Everywhere and the Three Amigos*. [Accessed 2017-06-01]. URL: https://www.ibm.com/developerworks/community/blogs/gcuomo/entry/javascript_everywhere_and_the_three_amigos?lang=en.

Da-14 (2016). *ReactJS vs Angular Comparison: Which is Better?* [Accessed 2017-06-05]. URL: https://da-14.com/blog/reactjs-vs-angular-comparison-which-better.

DeBill, Erik (2017). *Module Counts*. [Accessed 2017-06-01]. URL: http://www.modulecounts.com/.

Elliott, Eric (2014). *Programming JavaScript Applications: Robust Web Architecture with Node, HTML5, and Modern JS Libraries*. O'Reilly Media, Inc.

Facebook (2017a). *Debugging*. [Accessed 2017-06-05]. URL: https://facebook.github.io/react-native/docs/debugging.html.

— (2017b). *Generating Signed APK*. [Accessed 2017-06-11]. URL: http://facebook.github.io/react-native/docs/signed-apk-android.html.

— (2017c). *Integration With Existing Apps*. [Accessed 2017-06-05]. URL: https://facebook.github.io/react-native/docs/integration-with-existing-apps.html.

— (2017d). *Platform Specific Code*. [Accessed 2017-06-05]. URL: https://facebook.github.io/react-native/docs/platform-specific-code.html.

Facebook (2017e). *Running On Device*. [Accessed 2017-06-11]. URL: http://facebook.github.io/react-native/docs/running-on-device.html.

Fedosejev, Artemij (2015). *React.js Essentials*. Packt Publishing Ltd.

Felt, Christine (2016). *Which Are The Best Programming Languages to Learn in 2017?* [Accessed 2017-06-05]. URL: https://www.ibm.com/developerworks/community/blogs/d13a8e32-0870-49cb-9b0d-ba0e34fa6561/entry/Which_Are_The_Best_Programming_Languages_to_Learn_in_2017?lang=en.

Flanagan, David (2011). *JavaScript: The Definitive Guide*. 6th ed. O'Reilly & Associates.

Florescu, Cristian (2017). *Node.js is not suitable for generic web projects*. [Accessed 2017-06-03]. URL: https://coderwall.com/p/diokxg/node-js-is-not-suitable-for-generic-web-projects-i.

Foundation, Node.js (2015). *Node.js Foundation Combines Node.js and io.js Into Single Codebase in New Release*. [Accessed 2017-06-05]. URL: https://nodejs.org/en/blog/announcements/foundation-v4-announce/.

Frees, Scott (2015). "A Place for Node.Js in the Computer Science Curriculum". In: *J. Comput. Sci. Coll.* 30.3, pp. 84–91. ISSN: 1937-4771. URL: http://dl.acm.org/citation.cfm?id=2675327.2675341.

Git (2008). *What Are The Benefits of MVC?* [Accessed 2017-06-10]. URL: http://blog.iandavis.com/2008/12/what-are-the-benefits-of-mvc/.

— (2017). *About: Distributed*. [Accessed 2017-06-09]. URL: https://git-scm.com/about/distributed.

gregko (2013). *Does anyone have benchmarks (code & results) comparing performance of Android apps written in Xamarin C♯ and Java?* [Accessed 2017-06-05]. URL: https://stackoverflow.com/questions/17134522/does-anyone-have-benchmarks-code-results-comparing-performance-of-android-ap.

Guha, Arjun et al. (2017). "Fission: Secure Dynamic Code-Splitting for JavaScript". In: *2nd Summit on Advances in Programming Languages (SNAPL 2017)*. Ed. by Benjamin S. Lerner, Rastislav Bodík, and Shriram Krishnamurthi. Vol. 71. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 5:1–5:13. ISBN: 978-3-95977-032-3. DOI:

10.4230/LIPIcs.SNAPL.2017.5. URL: http://drops.dagstuhl.de/opus/volltexte/2017/7124.

Harrington, Chris (2015). *React vs AngularJS vs KnockoutJS: a Performance Comparison*. [Accessed 2017-06-05]. URL: https://www.codementor.io/reactjs/tutorial/reactjs-vs-angular-js-performance-comparison-knockout.

Hussung, Tricia (2016). *What Is the Software Development Life Cycle?* [Accessed 2017-06-06]. URL: https://online.husson.edu/software-development-cycle/.

Jaxer, Aptana (2015). *Create Apps Using JavaScript and Ajax*. [Accessed 2017-06-05]. URL: http://www.jaxer.org/.

Kol, Tal (2016). *Performance Limitations of React Native and How to Overcome Them*. [Accessed 2017-06-05]. URL: https://medium.com/@talkol/performance-limitations-of-react-native-and-how-to-overcome-them-947630d7f440.

Korotya, Eugeniya (2017). *5 Best JavaScript Frameworks in 2017*. [Accessed 2017-06-05]. URL: https://hackernoon.com/5-best-javascript-frameworks-in-2017-7a63b3870282.

Lei, K., Y. Ma, and Z. Tan (2014). "Performance Comparison and Evaluation of Web Development Technologies in PHP, Python, and Node.js". In: *2014 IEEE 17th International Conference on Computational Science and Engineering*, pp. 661–668. DOI: 10.1109/CSE.2014.142.

Meteor (2017). *Mobile: How to build mobile apps using Meteor's Cordova integration*. [Accessed 2017-06-05]. URL: https://guide.meteor.com/mobile.html.

Munroe, Randall (2007). *Compiling*. [Accessed 2017-05-27]. URL: https://xkcd.com/303/.

— (2011a). *Password Strength*. [Accessed 2017-05-27]. URL: https://xkcd.com/936/.

— (2011b). *Standards*. [Accessed 2017-05-20]. URL: http://xkcd.com/927/.

— (2014a). *Automation*. [Accessed 2017-05-27]. URL: https://xkcd.com/1319/.

— (2014b). *Tasks*. [Accessed 2017-05-27]. URL: https://xkcd.com/1425/.

— (2014c). *XKCD Phone*. [Accessed 2017-05-27]. URL: https://xkcd.com/1363/.

— (2015). *Code Quality*. [Accessed 2017-05-27]. URL: https://xkcd.com/1513/.

npm, Inc. (2017). *npm weekly #84: 100,000,000,000 (one hundred *billion*) package downloads, plus the CLI team has a public RFC out now!* [Accessed 2017-06-01].

URL: https://medium.com/npm-inc/npm-weekly-84-100-000-000-000-one-hundred-billion-packages-plus-the-cli-team-has-a-public-7f7057b53c6f.

Occhino, Tom (2015). *React Native: Bringing modern web techniques to mobile*. [Accessed 2017-06-05]. URL: https://code.facebook.com/posts/1014532261909640/react-native-bringing-modern-web-techniques-to-mobile/.

O'Dell, Jolie (2011). *Why Everyone Is Talking About Node*. [Accessed 2017-06-02]. URL: http://mashable.com/2011/03/10/node-js/#07A97JvP7kqw.

Politz, Joe Gibbs et al. (2015). "ADsafety: Type-Based Verification of JavaScript Sandboxing". In: *CoRR* abs/1506.07813. URL: http://arxiv.org/abs/1506.07813.

Portal, Statista: The Statistics (2017). *Number of apps available in leading app stores as of March 2017*. [Accessed 2017-05-27]. URL: https://www.statista.com/statistics/276623/number-of-apps-available-in-leading-app-stores/.

Pressman, Roger S. (2014). *Software Engineering: A Practitioner's Approach*. 8th ed. McGraw-Hill Education.

Ra, Ho-Kyeong et al. (2016). "Poster: Software Architecture for Efficiently Designing Cloud Applications Using Node.Js". In: *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services Companion*. MobiSys '16 Companion. ACM, pp. 72–72. ISBN: 978-1-4503-4416-6. DOI: 10.1145/2938559.2948790. URL: http://doi.acm.org/10.1145/2938559.2948790.

Rasmussen, Erik (2017). *Ducks: Redux Reducer Bundles*. [Accessed 2017-06-11]. URL: https://github.com/erikras/ducks-modular-redux.

Ruparelia, Nayan B. (2010). "Software Development Lifecycle Models". In: *SIGSOFT Softw. Eng. Notes* 35.3, pp. 8–13. ISSN: 0163-5948. DOI: 10.1145/1764810.1764814. URL: http://doi.acm.org/10.1145/1764810.1764814.

Software, Amell (2016). *Android Performance: Java vs Xamarin vs Xamarin.Forms Part 1*. [Accessed 2017-06-05]. URL: http://amellsoftware.com/2016/07/28/android-performance-java-vs-xamarin-vs-xamarin-forms/.

Staff, Aptana Studio (2009). *Jaxer forums?* [Accessed 2017-06-05]. URL: http://www.webcitation.org/5l1xyh2rY.

Stats, Internet Live (2017). *Total number of Websites*. [Accessed 2017-05-27]. URL: http://www.internetlivestats.com/total-number-of-websites/.

Teixeira, Pedro (2012). *Professional Node.js: Building Javascript Based Scalable Software*. John Wiley & Sons.

Thompson, Seth (2015). *Design Elements*. [Accessed 2017-06-01]. URL: https://github.com/v8/v8/wiki/Design%20Elements.

Union, International Telecommunication (2014). *The World in 2014: ICT Facts and Figures*. [Accessed 2017-05-27]. URL: http://www.itu.int/en/ITU-D/Statistics/Documents/facts/ICTFactsFigures2014-e.pdf/.

— (2016). *ICT Facts and Figures 2016*. [Accessed 2017-05-27]. URL: http://www.itu.int/en/ITU-D/Statistics/Documents/facts/ICTFactsFigures2016.pdf.

Virkus, Mart (2016). *Software Development Explained with Cars*. [Accessed 2017-05-27]. URL: https://toggl.com/developer-methods-infographic.

Westfall, Brad (2016). *Leveling Up with React: Redux*. [Accessed 2017-06-11]. URL: https://css-tricks.com/learning-react-redux/.

Wikipedia (2017a). *Information silo*. [Accessed 2017-05-27]. URL: https://en.wikipedia.org/wiki/Information_silo.

— (2017b). *List of programming languages*. [Accessed 2017-05-27]. URL: https://en.wikipedia.org/wiki/List_of_programming_languages.

— (2017c). *Right to Internet access*. [Accessed 2017-05-27]. URL: https://en.wikipedia.org/wiki/Right_to_Internet_access.

— (2017d). *Software as a service*. [Accessed 2017-06-08]. URL: https://en.wikipedia.org/wiki/Software_as_a_service.