SIMPLE++ MODULAR HEALTHCARE APPLICATION

Mikolaj Grajecki

A Dissertation submitted to
the School of Computing Sciences of The University of East Anglia
in partial fulfilment of the requirements for the degree of
MASTER OF SCIENCE.
SEPTEMBER, 2018

SUPERVISOR(S), MARKERS/CHECKER AND ORGANISER

The undersigned hereby certify that the markers have independently marked the dissertation entitled "**Simple++ Modular Healthcare Application**" by **Mikolaj Grajecki**, and the external examiner has checked the marking, in accordance with the marking criteria and the requirements for the degree of **Master of Science**.

Supervisor: _____
Dr. Dan Smith

Markers: _____
Marker 1: Dr. Dan Smith

_____
Marker 2: Dr. Michal Mackiewicz

External Examiner: _____
Checker/Moderator

Moderator: _____
Dr. Wenjia Wang

DISSERTATION INFORMATION AND STATEMENT

Dissertation Submission Date: **September, 2018**

Student:	**Mikolaj Grajecki**
Title:	**Simple++ Modular Healthcare Application**
School:	**Computing Sciences**
Course:	**Computing Science**
Degree:	**M.Sc.**
Duration:	**2017-2018**
Organiser:	**Dr. Wenjia Wang**

STATEMENT:

Unless otherwise noted or referenced in the text, the work described in this dissertation is, to the best of my knowledge and belief, my own work. It has not been submitted, either in whole or in part for any degree at this or any other academic or professional institution.

Permission is herewith granted to The University of East Anglia to circulate and to have copied for non-commercial purposes, at its discretion, the above title upon the request of individuals or institutions.

_____

Signature of Student

Abstract

The healthcare industry is in need of technological advances that will reduce the strain on the existing system. The shift to digital data storage has already proven useful for healthcare professionals - and it still has a lot of untapped potential. One way in which technology could be used to improve modern healthcare is by developing mobile-first systems that can be used for everyday healthcare needs. Such systems already exist - however, the market for these applications is fragmented and caters primarily to the younger demographic. But it is, in fact, the elderly who are the biggest expenditure for healthcare facilities.

Inspired by an application for the elderly from Buenos Aires, the proposed Simple++ system is a modular healthcare application, designed primarily for the web and tablet devices. Developed in Node.js for its back-end architecture and using Adobe PhoneGap for mobile deployment, it features a selection of functionally independent modules that can be enabled and disabled at will, as well as an accessible and usable user interface, catering to the elderly and those with vision impairments. Experiments with third-party modules were also conducted, in an attempt to prove the system's potential commercial viability.

The final product is fully functional in the way it was originally intended, with some minor issues, primarily to do with front-end design, that could be fixed in later releases. Attempts at implementing third-party modules were successful, albeit with some features disabled. Finally, testing proved that the application is indeed accessible to the elderly and/or disabled user and any existing security risks are minimal.

Acknowledgements

I would like to thank my supervisor, Dr. Dan Smith, for his many suggestions and constant support during this project, as well as his consistently positive attitude.
I would also like to thank Amy Henke for her help in understanding and implementing her application into my system.

Mikolaj Grajecki

at Norwich, UK.

Table of Contents

Chapter 1

Introduction

The need for healthcare is rising, as life expectancy and population both continue to increase. The main concern for healthcare systems is the ability to maintain (or even improve) the quality of care given to patients while controlling the ever-growing costs (Bergmo, 2015). As modern society starts relying more on technology and mobile devices have become a modern necessity, the next logical step is to adapt healthcare to take advantage of this technological shift to minimise reliance on in-person care and the related strain on the funding of public healthcare programs. The use of computer technology in health services has long been beneficial – the move from physical file-based systems to digital databases has improved the speed and access to data, whilst streamlining the process of patients moving between different clinics (Boulos et al., 2011). eHealth, the implementation of various computer technologies to improve both the communication between patient and practitioner and the quality of healthcare, is a hot topic that is being widely researched. There are many reported benefits, such as improved patient satisfaction, a better understanding of their own conditions and improved healthcare delivery (Davis Giardina et al., 2013).

Modern healthcare applications aim to reduce the strain on public healthcare by allowing patients to use a variety of services on their computers and mobile devices. These applications aim to be as simple and easy to use as possible by the widest range of users. But with a mobile library of tens if not hundreds of available healthcare applications, the market is highly fragmented, resulting in low adoption and redundant features. At this time, the NHS (2018) has a growing mobile application library consisting of over 46 applications, each with a singular dedicated function. These applications are not standardized, universal or connected. In an ideal world, I propose

there should be one healthcare application, a hub of sorts, that allows users to pick and choose what services and features they require – and install/enable them accordingly. Rather than requiring users to download multiple applications, in turn creating duplicate data and unnecessary hardware and network strain, this could theoretically be avoided by developing a single modular application which uses plugins that can be enabled and disabled at will by the underlying back-end architecture.

Further to this, existing healthcare applications are often designed with the young modern user in mind and not necessarily for those who use healthcare the most - the elderly. The aging population is an increasing cost to the modern healthcare system and advances are needed to reduce its negative effect (Amalberti et al., 2016). An application that caters primarily to the elderly could be very useful, if only in reducing the number of unnecessary GP visits.

The elderly demographic has special requirements in terms of what the user interface should look like, what features should be present and how the system should function overall. This project is inspired by the +Simple application developed by Buenos Aires Ciudad (2018) - an application that allows the elderly to stay connected to the internet through a simple and clear User Interface (UI). An application designed with accessibility and usability in mind can be used by both those with impairments/disabilities, and those without - the same cannot be said for normal applications which do not consider accessibility. With that in mind, the best strategy for an application to be usable by the widest possible user base is to design it with accessibility as the key feature. As outlined above, the current software landscape for healthcare applications is highly fragmented and not adapted to the needs of impaired users. As such, I believe this application, when developed fully, could fill a gap in the existing software market.

The main aim of this dissertation is to develop a prototype software product – Simple++, a modular healthcare application with accessible design, focusing particularly on visual aid. It should consist of a single screen base, with a plugin system for an array of front-end plugins. These front-end plugins need to have a focus on usability and accessibility, working in web and mobile environments. The target market is elderly healthcare patients, and the software will be implemented with a few demonstration plugins (and attempts will be made to include third-party applications created by other

students). The application should function with both mouse and touchscreen input and should allow at least a small degree of customizability in terms of audiovisual aids.

The particular objectives of this dissertation are:

- Review existing design guidelines and standards for developing accessible software systems
- Research modular programming concepts, requirements, and practices
- Develop an accessible front-end web UI that elderly and/or disabled users can navigate
- Establish a modular backend that allows third-party plugins to be developed and integrated into the application.
- Create a few simple plugins to showcase the ability of plugins to communicate with one another
- Attempt to implement a system developed by another student to test the degree to which third-party plugins are functional

I hypothesise that the front-end design will be completed, perhaps with a few minor design issues; the backend will be functional for bespoke plugins created for the system and the attempts to implement a third-party plugin will be largely unsuccessful due to the specific way in which these plugins were developed without considering how they would work with such a bespoke system.

The literature review will contain an exploration of modular programming concepts as well as usability and designing for the elderly (with a focus on touch devices). The design section will contain research into and a general overview of the particular technologies used, the planned functionality and any potential programming solutions. The implementation section will detail the final system, how it operates and explain any choices made. Finally, the evaluation and discussion section will look over some testing results and discuss any successes and shortcomings.

To avoid confusion in terms of the meaning of the term "module", I first want to outline the terms used within this text. The aim of this project is to develop Simple++, a modular application consisting of several modules, i.e. functionally independent

pieces of software such as First Aid or Nutrition. However, within the context of Node.js, "module" can also refer to Node.js native modules, either obtained via npm or custom-made by developers for use in a particular application. With that in mind, I will use the following terms in this text:

- Module – Node.js native module type (such as Express, ejs or the custom-made for Simple++ userDetails)
- Plugin – one of the application modules (such as First Aid or Nutrition)

The only exception to this is Section 2.2 - the Literature Review section - in which module refers to the general idea of a module as an element of a modular application.

Chapter 2

Literature Review

The aim of this literature review to review existing literature and theory on the main components of this project - accessible/usable design, modular programming and web technologies.

2.1 Usability, accessibility and designing for the elderly

The main aim of this project is to develop a modular application with a design accessible to the elderly. As such, research was conducted to determine exactly what designing such a UI entails.

When developing modern applications, the primary focus is often on functionality for the general population as well as a pleasing visual design. Newell and Gregor (2002) argue that when designing applications, one should consider more than just one demographic. Good design should aim to be inclusive; to provide a usable user interface for the able-bodied users but also the disabled, elderly, and the elderly and disabled. The overall motor and cognitive functions of an elderly person is significantly different from that of the typical young person - and even different from that of a young disabled person who may still possess comparatively more cognitive and/or motor function due to their young age.

There is little evidence to suggest that the elderly are technophobic; rather it is hard for them to understand and properly use modern technology systems because they are not designed with their capabilities in mind. Holzinger et al. (2008) elaborate on this notion, stating that "the elderly are often of the opinion that the benefits associated

with computer use fail to outweigh the necessary effort." It is not necessarily the fact that they are opposed to using these technologies - it is simply the fact that the sheer amount of time and effort required to get accustomed to these systems defeats their original purpose of simplifying certain tasks. Due to the many impairments this age group often experience, using these software systems may be significantly harder for them.

The elderly and disabled can be impaired in sight, hearing, speech, mobility and dexterity/memory. Normal computer screens or non-touch devices are often not the right display for such users. It has been shown that mobile touch devices are easier for the elderly to operate than non-touch devices. Kobayashi et al. (2011) argue that non-touch devices have buttons that are hard to press, displays too small to read and complicated mechanical procedures. Touch devices lack all of the above issues and add features such as finger-based intuitive interactions. Their general findings indicate that basic touchscreen operations are easy for the elderly to perform even without training - furthermore, a week's practice considerably improves performance. Larger screens, such as tablets, are preferred as they can positively impact the performance and usability for the elderly (Chang et al., 2014). For mobile devices, a feature to enlarge crucial elements of the UI should be provided to compensate for the small screen (Chang et al., 2014).

In terms of touchscreen UI design, there are many considerations when designing a usable interface. Hwangbo et al. (2013) argue that button size and spacing are crucial in this regard. As button target size and spacing increase, so does the performance. In specific terms, Burkhard and Koch (2012) found that the touchable elements of an interface should never be smaller than 9mm in diameter to ensure accuracy in pointing.

An important element in any application is data input. For the elderly, this simple task may prove a challenge. Nicolau and Jorge (2012) argue that while larger key sizes on tablets improve accuracy somewhat and tablets can usually compensate about 9% of typing errors, the overall accuracy is still low, especially considering the tendency of this age group to experience shaking and hand tremor. Furthermore, tactile feedback may have an impact on typing and pointing performance - Hwangbo et al. (2013) found that when used alone, tactile feedback was distracting for the majority of tested users.

On the other hand, audio feedback slightly improved performance - however, it was a combination of audio and tactile feedback that provided the best performance.

Demiris et al. (2001) similarly argue that accessible design is key to improving modern software systems and encouraging the elderly and disabled to use them. Aging affects our minds in many ways - information-processing capacity, reduced speed of precise movements, increased time required to recall memories and decreased attention span over long periods of time, amongst many others. As such, tasks as simple as reading and inputting data, something easy for the average modern computer user, require visual and motor coordination that many elderly simply no longer possess. Zimmermann and Vanderheiden (2008) argue that the best approach to ensuring an application is accessible is via appropriate testing, studying potential use cases and outlining guidelines for what is required for such design. For the purposes of this project, testing and use cases will be discussed in Chapters 3 & 5. Based on the findings of Demiris et al. (2001) and the W3C Web Accessibility Initiative (2018), I have compiled the following list of interface considerations to aid my design in the following chapter. An accessible UI should:

1. Be simple and clear; concise.
2. Have alternative ways to access and navigate.
3. Have icons that are simple and any symbols are designed to look like the object/function they represent.
4. Have a large font size, large buttons with labels describing their function.
5. Avoid colour/patterned backgrounds, high contrast to ensure readability.
6. Make sure that colour is not the only source of information.
7. Have a help menu with means of obtaining assistance in several ways.

These guidelines will be used later in this document to design and implement the front-end UI.

2.2 Modular Design Fundamentals

Fleischer et al. (2015) define a module as “a software component with data and functions to process a self-contained task”. Each of such modules should allow easy communication between one another using defined interfaces and ideally continue functioning without disruptions when the underlying framework is updated. The main principle when developing modular software is that of reusability and splitting a large system into ”small, nearly independent, specialized sub-processes” (Lavy and Rami, 2018). As a result, modularity provides a great foundation for continuous development and maintenance. Modular applications, even if initially require more development time, prove useful in the long term as they are more adaptable, easier to comprehend for outside programmers and ultimately more efficient (Walls, 2014).

As a case study example, Trauer et al. (2017) devised a modular system for tuberculosis control, where every module was a separate Python class. Each of these modules produces “consistent set of data structures” to make sure adjusting a single class will not dictate the need to adjust all the other ones. This is a perfect example of the foundation of modular programming - the (relative) independence of each module from one another and, ideally, the underlying architecture.

Of particular interest to this project is the system developed by Hahn and Ryckman (2012) - a modular mobile library application. They argue that the field of computing in library settings is overrun by one-off systems that become obsolete over time. They propose a system that would instead consist of a group of modules, each devoted to a feature that would normally be developed into its own application. The modules communicate with each other - for example, the barcode scanner can share data with the home module to add scanned books to the user’s list. The biggest benefit of this system is that rather than completely discontinue an entire application once it becomes obsolete or superseded, this system would allow to simply remove or replace that single module, rather than abandon an entire system, saving both development and maintenance time and costs.

When developing modules, there are certain rules that should be followed. Trauer et al. (2017) outline a small set of guidelines for developing modular systems. According

to them, each module must be stylistically consistent, include extensive commenting, loops to avoid code repetition and consistent naming conventions. Having such guidelines ensures the code can be understood by external programmers – which is extremely vital in the case of modular software, as each module can, in theory, be written by a different programmer, and the code should be able to be re-used in other modules where appropriate. Further to these, Dennis (2017) devised his own set of principles to support modular software engineering. These include the following:

- The user of a module should not need to know the underlying technical details of how it operates in order to make use of it.
- The functionality of a module must be independent of the site where it was invoked.
- The module interface must be able to pass data between modules.
- Resource management should be performed by the computer system, not by individual modules.

Other than supporting the use of 'plug-in' modules, modular programming also helps with overall code readability, making it easier to test and maintain (Seydnejad, 2016). It is easier to scale and extend a modular application – which is crucial for this project's proposed system, which would, in theory, be continuously upgraded and extended. Wong et al. (2011) argue that modular programming is particularly challenging for inexperienced developers, who may commit early mistakes that undermine the entire modularity of a project. As such, modular systems developed by novice developers are likely to be somewhat less viable.

As the target use for the project's application is healthcare, security of user data is vital. As the chosen Node.js environment (explained more in the following chapter) is relatively new, having only been introduced in 2009, it is expected that it will be more prone to security issues than more mature platforms – however, Ojamaa and Düüna (2012) argue that while (at the time of publishing) there were some outstanding issues or vulnerabilities, most of them "can be avoided with security-conscious programming". In their study of 235,860 third-party Node.js modules, Staicu et al. (2016) have similarly found that there are some common injection vulnerabilities –

however, they propose there is a relatively simple solution and such vulnerabilities can be mitigated easily. Another potential threat is cross-site scripting (XSS) - the injection of rogue code/scripts into dynamically generated pages (Spett, 2005). With that in mind, I will consider the security risks of my software system in Section 4.4. This will include both database injection attacks and potential XSS attacks.

Chapter 3

Design

The Design section outlines the design decisions made for this system, both front-end and back-end. It briefly discusses the inspiration behind the system, as well as outlining the particular technologies chosen to develop the system, along with the reasoning behind those decisions. It also briefly discusses some use cases - more of which will be discussed in the following section, in the context of the finished product.

3.1 Background inspiration

The inspiration behind this system is the +Simple application for the elderly as designed by Buenos Aires Ciudad (2018) (see Figure 3.1). The application aims to provide its elderly users with a clean and simple solution to keep them connected with the digital world, providing easy to use features with accessible design. Things such as email, news, a web browser etc. It is very much a system split into modules, connected to create one cohesive application. It features a clear UI that is easy to navigate and read by those with vision problems, featuring highly contrasted colours and readable fonts. This particular system can be installed on any Android tablet or it can be purchased on a preloaded tablet device. In comparison, with the help of PhoneGap, my system can theoretically run on any desired platform, including Android, as well as any device that has a web browser.

Figure 3.1: Main screen UI of +Simple

3.2 Application Design

3.2.1 Front-end design

The main design elements taken into consideration when mocking up the frontend design were:

- Large icons with a large target area (as large as the background colour around the icon itself)
- Target screen - 16:9, horizontal orientation only; however, still responsive to vertical orientation to a certain degree
- Light-pastel background
- High contrast and saturated colours
- Text accompanying every interactive element (for text-to-speech compatibility)
- Persistent top and bottom bars
- Easy access to help in the navigation bar
- Font at a default large size, increasing up to a desired size using the 'enlarge' function
- Ability to pick plugins

- Dynamically adjust the home screen layout depending on the number of active plugins
- Ability to produce audiovisual cues to indicate unsuccessful taps (togglable in settings)

The UI design of my Simple++ system was very much inspired by the +Simple system as briefly discussed in Section 3.1. The main goal is to design a UI that is first and foremost accessible and usable - rather than designed with modern aesthetics in mind. As such, it features very few elements that exist purely for decorative purposes. Every element should have a functional purpose and it should be kept as minimal as possible, to reduce the amount of potential distraction for the elderly user. The notable exceptions are the planned fade in/out animations and brief flashing/sound notifications when the user fails to click on an interactive element. All of these features can be turned on or off (and are turned off by default), as depending on the particular user, they could either help them navigate the application or confuse them. The Enlarge function will be designed as a popup window; the only other potentially visually-confusing feature to the elderly user. However, not only is this functionality vital to the core of this software system, but it is also designed to be used dynamically - sometimes the font might be too small, other times too big, so the ability to quickly change the font size without leaving the current page is necessary - and that only works with an element that can be shown and hidden on demand. The earliest prototype UI design can be seen in Figure 3.2.

This final design concept was a result of several mockups created. Other mockup prototypes included text and colour-only navigation; the lack of a top bar; differently shaped icons or even a Windows Phone-inspired tile UI. The design visible in Figure 3.2 was by far the most pleasing to the eye as well being the most cohesive - and after reviewing some literature from Section 2, it became obvious that a lot of the other designs were simply not accessible enough, if only for the target area size of the icons being too small.

Figure 3.2: First UI prototype

3.2.2 Languages and technology used

In terms of the coding language required for this task, the aim is to create a hybrid application that will run on the web and mobile devices. The main challenge here is that each platform requires a different coding language to be used – Android uses primarily Java while iOS relies on Objective C/Swift (Chaniotis et al., 2015). This suggests the need to write separate code for each platform. However, all modern mobile platforms also support web apps running on HTML, CSS, and JavaScript (JS) (Shotts, 2014). In order to bridge the gap between a web app and a native mobile app, a framework can be used – such as Apache Cordova. The Cordova platform allows developers to build mobile applications using a combination of HTML5, CSS, and JS to create code that can be used on both Android and iOS (and other platforms) without major alterations. The way Cordova works is by providing "a native wrapper for containing the web-based code" (Malavolta et al., 2015). PhoneGap is a distribution of Apache Cordova, owned by Adobe. It is not the only such framework available on the market – however, compared to some of its competitors, such as Titanium, the development process is less complicated, more streamlined and it is, in general, the preferred option for cross-platform development (Heitkötter et al., 2012).

With PhoneGap as the chosen framework for mobile deployment, the next step is to consider the programming language and platform. As JS is now an integral part of the web landscape, being used in a large number of modern web applications (Tilkov

and Vinoski, 2010), and is natively supported by PhoneGap, it is a logical choice in this case. More specific than just JS, Node.js should be considered as an option. To an extent, Node.js can be treated as an extension of JS - it is a "server-side solution" for JS, tasked with managing HTTP requests (McLaughlin, 2011). Chaniotis et al. (2015) argue that Node.js is a great choice for web applications, as it aids code reusability and is useful for creating "scalable network application" – features vital to modular programming. Compared to general JS, it also provides better performance whilst lowering CPU usage, in turn improving mobile battery life (Charland and Leroux, 2011). It is also relatively simple to learn and use and has limited but robust core functionality – more complex features can be achieved by installing one of many third-party modules (Teixeira, 2012). For data storage, there are two possible options - local storage or hosted database. MongoDB, being an efficient and simple NoSQL database, would work perfectly for this type of application where the data would not necessarily need to be sorted into relational tables (Wei-Ping et al., 2011). An alternative option would be to simply store all the necessary data locally, in the form of JSON (or JSONP for cross-site compatibility) files - both of these will be considered and a decision made at a later development stage (Giang et al., 2014).

Furthermore, several Node modules will be employed to enhance the functionality of the system. There were two main tasks in regards to the back-end architecture:

- dynamically display the contents of a plugin inside the native app UI
- allow these plugins to communicate (by way of writing/reading commonly shared files and using custom Node.js modules)

After much research, Express for Node.JS was chosen as the primary Node.js framework, as it is by far the most expansive web framework available for the platform, allowing for many features to be implemented easily. It has a very simple yet robust set of functionalities that fulfil every task I needed to achieve in this system. Initially, I considered using the full MEAN stack - MongoDB, Express, Angular and Node.js - as it is a very powerful and effective solution to many systems, based purely on JS (Holmes, 2015). However, after additional research and conversations with colleagues about their experience with Angular, I decided it would not be necessary for this particular

project.

Next, for the purpose of displaying the contents of plugins, I needed to use a templating engine. After much deliberation between different engines, such as Pug.js or Handlebars, I have settled with EJS as it proved the most efficient for the type of tasks I required it to do. I have considered and experimented with several approaches in terms of how a plugin should be loaded when a user interacts with its icon:

1. Insert the contents of the plugin as a $< div >$ element into a template.

   This approach involved accessing an HTML file, hosted either internally or externally, scanning its content with an appropriate function, converting the scanned text back into HTML, and inserting it into a template file to display on the screen. The template file consists of the HTML header, top bar, and navigation bar. Any custom CSS or JS would have to be included inside the single file, rather than in external stylesheets or JS files.

   Pros:

   - Fully autonomous from the underlying architecture; any changes to the back-end architecture of the application would not impact the plugin as it is simply an HTML/CSS/JS file.
   - Somewhat more secure as it would limit the ability to load any additional files.

   Cons:

   - Complicated to create plugins that would use more than a single page.
   - No way of accessing Node.JS functionality; cannot specify what should happen when the user opens the plugin (the only option is to load the HTML page).
   - All page routing would have to be maintained within a central file in the application, not easily accessible by third-party developers.

2. Allow each plugin its own folder with its own filesystem.

   This approach involved hosting each plugin within its own folder, with a main.ejs file. This file by default should contain custom-made EJS partial-views provided

by Simple++ to dynamically insert all the necessary HTML header data, top bar and navigation bar. The plugin contents simply need to be inserted as a $< div >$ between these tags. These partial views ensure that the plugin is still largely independent from the site it is invoked on - any changes to the actual application UI will not affect the plugin directly. The folder also must contain its own main Node.JS file which dictates what happens during request-response calls. This file is then imported in the main Node.js server file.

Pros:

- Ability to use separate stylesheets and JS files (for improved code readability).
- Improved functionality and the ability to use several pages with ease.
- Routing customizable by the developer.

Cons:

- Bigger security risk with more autonomy given to external developers.

3. Allow each plugin to use its own static pages and write code that adds a layer on top with the navigation and top bars.

   This approach proved the most challenging and ultimately unsuccessful. There were many issues from the start - aspect ratio problems, overlapping or conflicting elements etc. After spending several days attempting to get this approach to work, I decided it was not worth pursuing any longer and it was abandoned.

The way pages are rendered was the biggest design decision taken in terms of back-end design. The final decision was to use the second approach - allowing each plugin to have its own dedicated folder and filesystem. For all other features, the functions were straightforward and did not require excessive testing before deciding on what approach to take.

3.2.3 Use Cases

As visible in Figure 3.3, the user functionality of the main framework (not including individual plugin functions) is as follows:

Use Case A: Choose a module

The user will choose an available plugin from the icon display, either via touch or mouse input. If enabled in Settings, when their finger/pointer fails to land on an icon target area, the background will briefly flash and a sound will play to indicate the failed action. When a plugin is chosen from the screen, it will redirect the user to the individual plugin page, which has its own set of features. They may always return to the home screen by using the navigation bar at the bottom.

Use Case B: Enlarge text

Upon interacting with the Enlarge button, a popup window will appear with a slider. Moving the slider in the appropriate direction will increase/decrease font size (and the sizes of certain elements to accommodate the larger text). This selection is then remembered for the duration of the local session or until the user changes their selection.

Use Case C: Select active modules

The users may choose which plugins they want to be active via the settings menu. Upon interacting with one of the icons, the background will turn green to indicate active selection. When pressing the save button, the selection will be saved to a local file and the home screen will show only the active icons until a later point when the user changes their selection again.

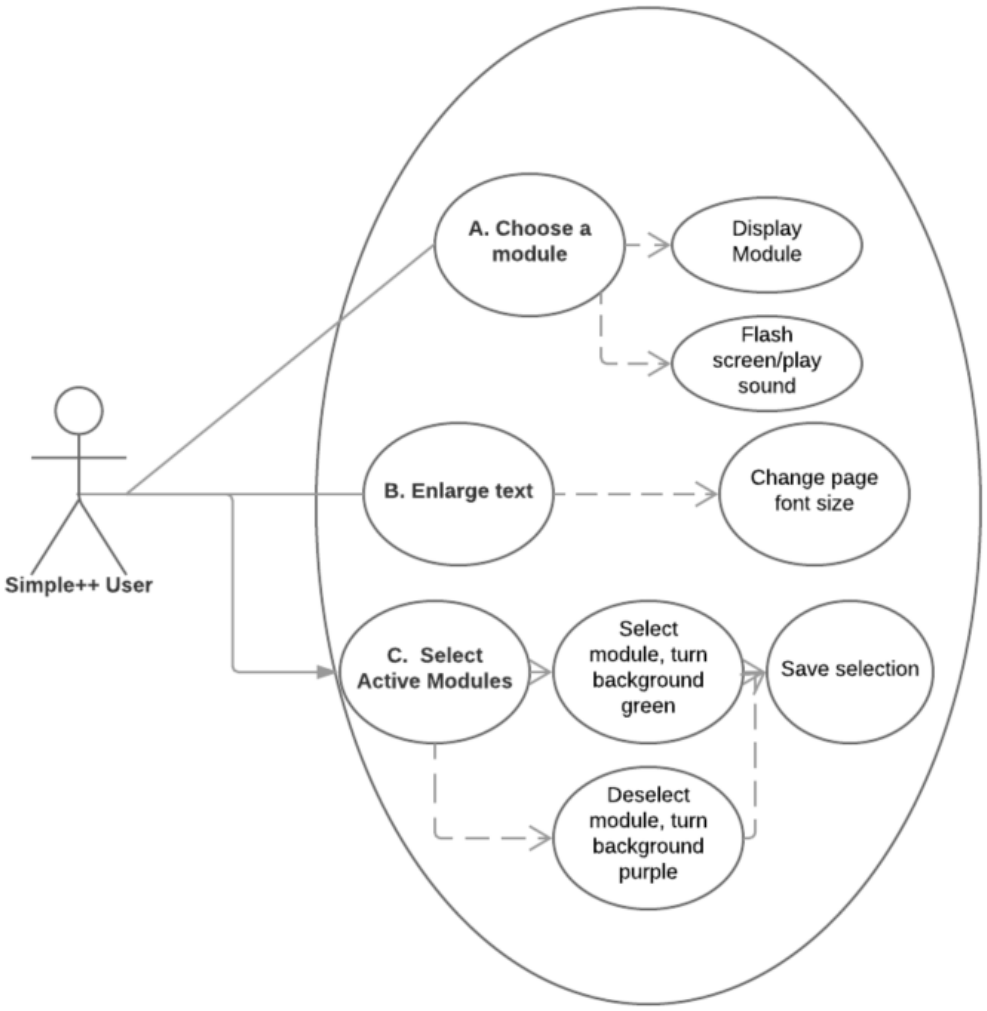

Figure 3.3: Basic function use case diagram

Chapter 4

Implementation and testing

This section discusses the final developed product, explaining the development process, changes made along the way as well as showcasing the architecture of the software and the exact way in which everything on the screen is rendered. In the final subsections, it discusses PhoneGap implementation, as well as counter-measures employed for both current and potential future security risks.

Despite initially seeming quite easy to design and develop - as it is seemingly just a screen with a top bar, navigation bar and a screen of icons - the actual development and implementation proved quite challenging. It was crucial that this application is responsive (at least for tablet-type resolutions), as it should by default function on many devices with different resolutions. I ran into an endless amount of issues during the development - icons not resizing properly, JS functions interfering with CSS attributes, being unable to read JSON files and many issues with PhoneGap deployment. Ultimately, the system performs all that tasks that I wished for it to - with a few shortcomings that I will discuss in the following Section 5.

4.1 Frontend development

At its core, the interactive icons are composed of 2 elements - a white FontAwesome icon and accompanying text - wrapped inside a $< div >$ tag with a high contrast and saturated background colour. The icon and text are set to be a size relative to the current viewport width (vw), with the text additionally changeable via the Enlarge function. As the UI was designed primarily for a tablet-sized device in a horizontal orientation and 16:9 aspect ratio, the UI best scales according to horizontal changes.

The UI was not designed for portrait mode use, much like the inspiration Simple++ system, and therefore such functionality was not seen as necessary to implement - however, it is still, to a certain extent, usable in vertical orientations.

The initial design mockup was made using HTML, CSS, JS, and jQuery, based on a single screen divided into 3 sections - the top bar containing the name of the current page, the main screen for content, and a persistent bottom navigation bar. The top and navigation bars do not change between different pages and as such can be easily added to any custom plugins. The colour palette was selected based on my guidelines for accessible design from Section 2, making sure high contrast is consistently present and text easily legible. Blue was ultimately picked as the main UI colour as it is a clean colour that helps divide the UI into clear sections - much more so than black. Simple fade in/fade out animations were added to the icons to reduce the flashing effect of icons suddenly appearing - however, these animations can be easily switched off in settings and are turned off by default to avoid confusion and negative effects for epileptic users. An audiovisual alert in the form of a brief white background flash and sound effect can also be enabled as a form of audiovisual navigation aid, to inform the user when they have failed to press any icons. Similarly to the animations, this can be toggled in settings and is disabled by default as it may prove too distracting for some. Font family, letter spacing, font sizes were all experimented with to determine the best options that maximize accessibility whilst having the least impact on the overall attractiveness of the design.

The plugin select functionality works on a very simple premise - the selection the user makes in settings is saved in the form of a local JSON file, as an array of $modulename$: $state$ values. Every time the selection is changed, the file is updated with the new selection. Upon loading, the body of the home page loads a script that reads in the contents of the file, processes it into a variable of value $true$ or $false$ for every existing module, and then sets the visibility of each icon accordingly. What follows immediately after is a jQuery function which counts the number of visible icons and adjusts the CSS rulesets accordingly. The margin size, icon size and placement of the icons adapt to accommodate the changing screen real-estate.

The text resize function works by dynamically adjusting various CSS ruleset elements (such as font size, label size, box size) by a given value set by the user. This is primarily a vanilla JS/jQuery function. It uses the JS localstorage functionality to remember the set font value and apply it to every page visited. This setting lasts for as long as the session does, or until the user overrides it by repeating the function.
Once I have reached the point of creating the resize functionality, it became apparent that the overall layout functionality would be a lot harder to achieve without using another framework. With that in mind, I attempted to recreate the UI with the Bootstrap framework - with varying levels of success. While the UI was responsive, scaled well and did not interfere with the code that enables/disables plugins, the resizing of fonts did not cooperate well with the code so I briefly went back to the design that did not involve Bootstrap. After many more attempts at writing my own code to achieve the required layout functionality, I ultimately decided to go back to the Bootstrap prototype and managed to improve the resize functionality enough so that it worked alongside Bootstrap with considerably fewer issues.

Throughout the development process, the UI changed considerably. Initially, it was using default fonts with all white-yellow and blue features, flat icons, larger top bar with unnecessary text (as previously seen in Figure 3.2). That design went through many changes before I decided to change the colour palette of the menu items, reduce the size of the top bar (as it scaled poorly) and exclude it from text resizing, as ultimately it does not contain any crucial information. The second large update changed the menu colours to black, increased the border-radius on the icons and changed the background colour to a pastel grey (see the top of Figure 4.1). The final UI reverted the colour scheme back to blue shades with a slight gradient for added depth, it retained the new pastel grey background, reduced the border-radius on the icons and added a box shadow to create the effect of 'real' buttons in an attempt to make them look more like physical buttons (and by extension make it easier for elderly users to understand). I also applied the same effect to the navigation bar to help visually separate the buttons for those with vision impairments.

Figure 4.1: Earlier version vs current UI

4.1.1 Hardware variety

As a web application first, the Simple++ system can be used on any device with a web browser. It can also be used on supported platforms with a dedicated application as compiled by PhoneGap. While the target mobile device type is a tablet in horizontal orientation, the front-end design was also tested on mobile devices in horizontal orientation, as shown below. The standalone application functionality was not tested on iOS as it required commercial Apple developer certificates and PhoneGap cannot compile an iOS application without a certified key. Instead, iOS devices were simulated and then tested for their responsiveness.

First, an iPhone was tested in horizontal and vertical orientations (see Figures 4.2 and 4.3 respectively). The horizontal orientation responds as expected since the UI

was designed with that orientation in mind - icons scale down appropriately, disabling text to improve readability on a smaller resolution. A JS function runs on page load to determine the orientation of the device - and depending on the orientation, it then overrides certain CSS rules to adapt the layout to the given orientation. As a result, the iPhone vertical orientation main screen can also work, despite this feature not being originally planned for this system. In vertical orientation, the icons are arranged in fewer columns but more rows, with slightly taller icons and all text removed for readability.

Figure 4.2: Simple++ on an iPhone, scaled 736x414, 16:9 horizontal

Figure 4.3: Simple++ on an iPhone, scaled 736x414, 16:9 vertical

Figure 4.4: Simple++ on an iPad (1024x768, 4:3)

The iPad is an interesting example - while it is a tablet device, its aspect ratio is 4:3. Most Android tablets (and indeed most digital consumer displays) have a standard 16:9 display aspect ratio, which means that the iPad has to have a special media query that works with its unique 4:3 aspect ratio. As can be seen in Figure 4.4, the icons are much taller than on a 16:9 display to accommodate the different aspect ratio.

The working layout on iOS devices also means that most comparable Android devices will work in both orientations - as tested on a OnePlus 3T, OnePlus 6, Samsung Galaxy and Pixel 2. It has to be noted, however, that the horizontal orientation is still the target here and that the plugins are not optimized for vertical use. PhoneGap-deployed applications are locked in landscape mode.

4.2 Backend

Figure 4.5 represents a basic block diagram of the system architecture. The application responds to any input in the form of touch/mouse click by connecting to the server.js file, which functions as the main hub for all operations in the system. As this system uses the templating engine EJS to render each page in real time, it cannot simply redirect the user to a static file. Instead, it intercepts the */GET* request and either renders the appropriate system page or if a plugin was chosen, it will query routes.js. Routes.js contains all of the code responsible for handling request/response messages for individual plugins. Every plugin is tasked with handling its own routing requests, provided inside a custom Node.js file. If clicking on an icon sends a */GET* request for

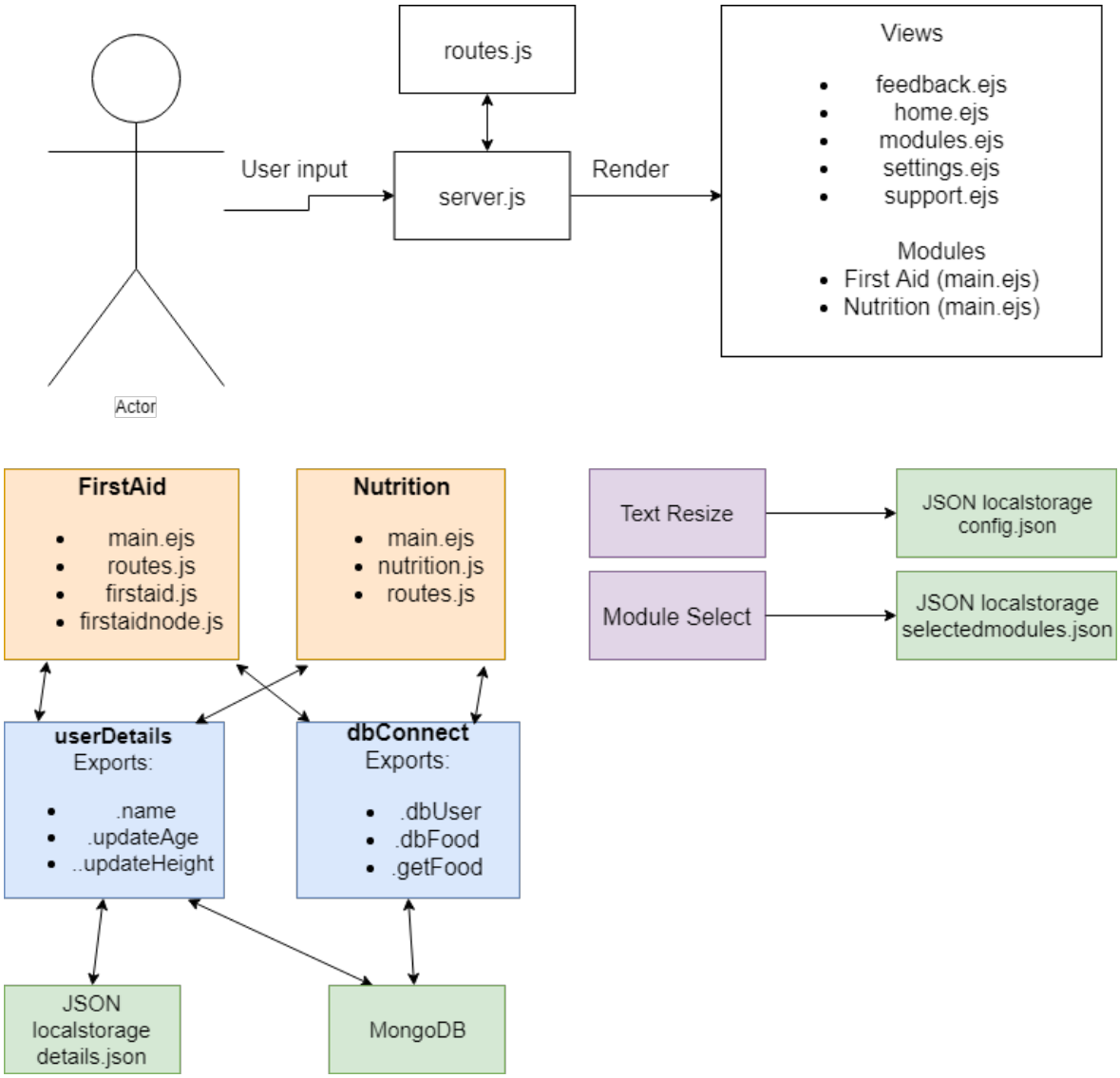

Figure 4.5: Early block diagram showcasing the layout of the Node.js functions

e.g. */firstaid*, then the First Aid Node.js file needs to have code that responds to this request. The diagram also references MongoDB and related functions - these remain implemented into the system but disabled until a time when a database connection is deemed necessary.

The project directory contains several folders that reference modules:

- externalmodules (these are the custom plugins developed for this project such as First Aid)
- nodemodules (Nodejs modules obtained via npm)
- sppmodules (Custom code modules developed for this project for use across the plugins, such as userDetails)

Additionally to the First Aid, Nutrition and Maps plugins created for demonstration

purposes, I have also written several Node-native modules to provide easy access to commonly used functions within plugins – retrieving user data, updating user data, connecting to the database etc. This reduces code duplication significantly and means a potential third-party developer does not need to know how the underlying code functions, they simply need to use Node.js native */require* calls to be able to use the provided functions.

The provided Node.js custom modules are:

- userDetails (provides access to reading user details)
- foodLog (provides access to the food diary log)
- dbConnect (provides access to reading and updating the database, if needed)
- configSelect(allows turning sound/animations on and off)

The application uses primarily local storage in the form of various JSON files (and JS localstorage). There is also a connection established to a MongoDB database, if need be - however, this may create unnecessary security risks and problems with GDPR compliance, so it was disabled in this release. Future use may see the need for a database, and as such, the decision was made to keep the necessary code as it is fully functional in its current form.

Every plugin is self-contained within its own folder, with its own routing and ejs template files – if need be, these could even be hosted on an external server. In the main configuration file for the application, server.js, each plugin's Node code is imported using Node's */require* functionality – meaning any changes in the plugin code will be automatically updated in the main framework with no need for manual changes.

By default, every page is rendered on demand by ejs using the appropriate view file. When clicking/tapping any icon, it sends a */GET* request to the back-end routing system, which in turn renders one of the set-up views on demand, allowing to pass data to dynamically insert into the template. An ejs template may have something like the following:

```
You are <%= age %> 20 years old
```

Age will be replaced by the value of variable *age*, so long as it is passed to the template at the point of rendering via Node.js.

Partial views are provided to render the top bar and bottom bar; the only thing required to write for the individual plugins is the actual content of a page, to be inserted between the top and bottom partial views, as in the following example:

```
<%- include partials/moduletop %>
-- page content --
<%- include partials/modulebottom %>
```

Once rendered, moduletop will be replaced with the HTML header and top navigation bar, while modulebottom will be replaced by the bottom navigation bar and HTML closing tags. The plugin can insert its own content between those two tags as a self-contained $< div >$ object.

4.2.1 Plugins

There are three available plugins for the purpose of demonstration - First Aid (which allows the user to update some of their personal details in case of an emergency), Nutrition (a simple diary allowing the user to note down their meals) and Maps (a third-party plugin, stripped down to basic functionality). These are not designed as plugins meant to exist in a live commercial application - rather, they exist purely for demonstrative purposes, to show how such plugins would look like and show their communication capabilities. As such, the First Aid plugin allows the user to view and amend their personal details, such as name, age, emergency contact, allergies, medication - with the idea that these would be the most important details to know in an emergency. Additionally, it also shows the last 5 days from the food diary module, in case that would be relevant to a doctor/emergency paramedic - whilst simultaneously showcasing plugin communication.

The Nutrition plugin allows the user to note down their meals they had that day - it only shows the last week worth of meals on the side, but the application stores the entire food log history that is accessible via a lookup function within the plugin.

Finally, the Maps plugin is an integration of a third-party plugin developed by another student called "Get Going" (see Figure 4.6).

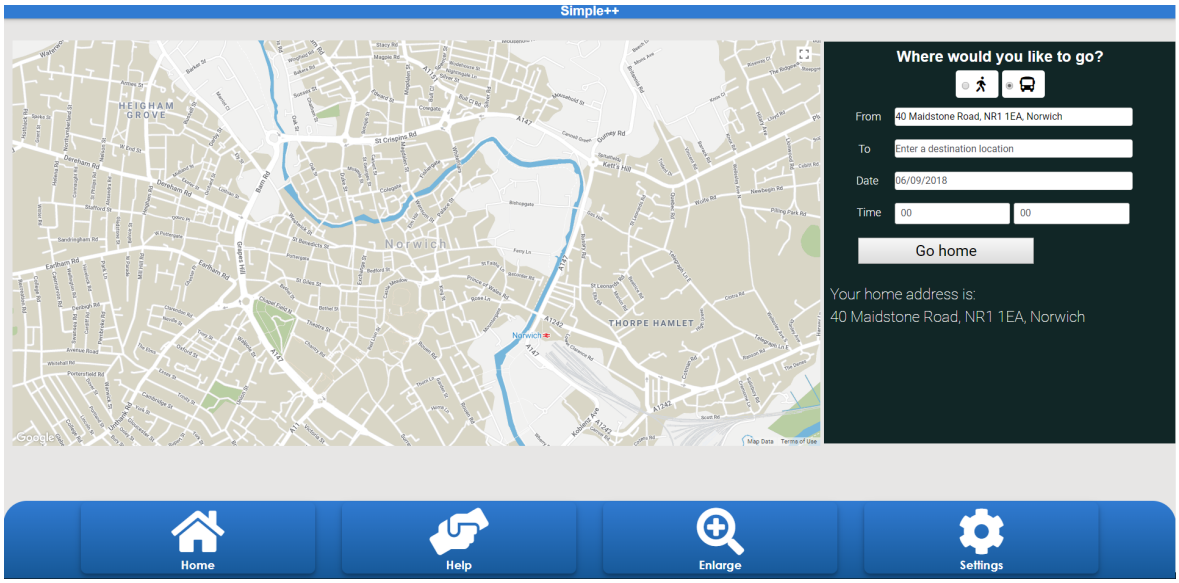

Figure 4.6: Get Going integration

The original web application is in the form of a standard responsive website, with a lot of informational pages dedicated specifically to Norwich residents. The main functional page is the navigation-map system, which shows its users directions to a given location both on the map and in text form underneath. As this application was not written with the Simple++ system in mind, it did not implement easily at first - and in fact, some of the features simply would not work with the way the Simple++ UI is designed and developed. As such, the functionality was limited to just the map navigation system - the user can select their target and current destination and view the best path available on the embedded map. For the purposes of demonstrating plugin communication, I have added a 'Home' button which when pressed, inserts the Simple++ user's home location, as set in the First Aid plugin.

All third party plugins have read-only access to user details by importing the custom-made userDetails module (for example userDetails.medication will return the user's list of medication). They do not, however, have write access - there is no need for these plugins to ever have access to manipulating sensitive data.

4.3 PhoneGap

After considerable development work was already done, I attempted to use PhoneGap in order to create a standalone .apk file to test on my Android device. Unfortunately, server-side environments are not supported by PhoneGap by default; I could not run my system directly through PhoneGap. I will further discuss this issue in Section 5.

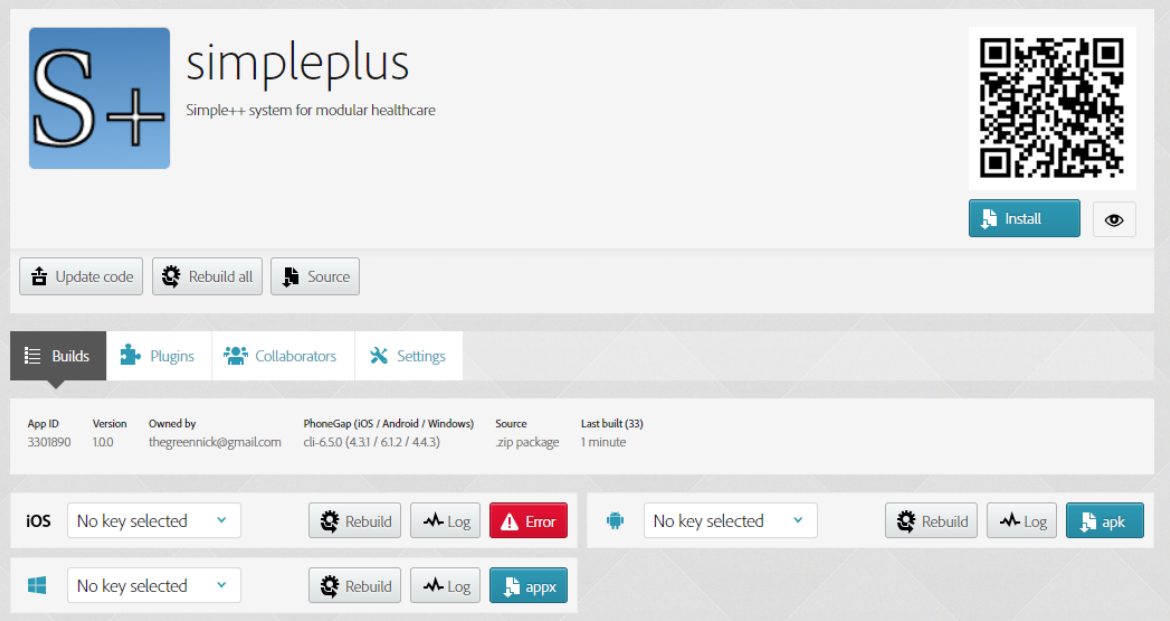

Figure 4.7: Adobe (2018) PhoneGap Build - error indicating no commercial iOS key

The suggested solution is to split the system into front-end and backend portions, build the PhoneGap application with just the frontend and connect the frontend with the live Node.JS server hosted elsewhere online. This, however, would not work with my system either as I use EJS to render every page displayed; it does not have any static HTML pages. After many attempts at achieving a functioning PhoneGap system, I ultimately had to resort to redirecting the PhoneGap application to a web server that hosts the entire live application. This results in a fully functional application that can be installed as a .apk file - the major downside of this is that the application will not work offline, cached or otherwise. However, it functions exactly as I would want it to and the end user sees no difference in experience between a fully local and this web version.

The .apk file was compiled using the cloud-based Adobe PhoneGap Build service as seen in Figure 4.7

4.4 Security

As not only a web application but a potential healthcare application, the security of the system is important - both from external and internal attackers. Initially, I wanted to prevent access by potential rogue developers by limiting their read/write access to system files. After much research, it did not seem that Node.js is capable of restricting such access. Instead, in a hypothetical scenario where a third-party company is developing a module for this system, the developers would simply not have access to the source code of the underlying framework and would develop only their module externally. Using provided custom Node modules (such as retrieving personal details using the userDetails module), they can have read access to relevant data without having a direct way of manipulating the file.

XSS was another potential security risk considered - in this case, I employed a custom Node.js module called 'XSS', installed via npm (2018) which employs a sanitization technique - taking any potentially untrusted data stream and filtering any script tags and any potential rogue code, rendering most such attacks unsuccessful.

Furthermore, despite the disabling of MongoDB, the safety of this feature was also considered and implemented. After some research, it became apparent that the simplest and yet most effective way to protect against this type of vulnerability is to, again, much like XSS, employ a filtering system. For this purpose, I am using the Node.js module content-filter available via npm (2018) and apply the following blacklist:

```
var blackList = ["$", "{", "&&", "||"];
```

These are the most common symbols used in NoSQL attacks (Sullivan, 2011). Upon attempting to use any of these characters on any data input or */GET* requests, the filter stops the request and redirects the user to a page that simply states that they have used a forbidden expression.

To summarize, every user data input is first sanitized for any expressions that could be used for XSS, after which it is filtered for any potential NoSQL injection attacks.

Chapter 5

Evaluation and Discussion

This section evaluates the final product, its successes, and shortcomings. An evaluation of collaborative work with other students is then conducted. A brief outline of version control is then followed by a show of the tests that were conducted on the final system.

5.1 Successes and shortcomings

I believe that the software system that I have developed has met almost all of the aims and objectives I set out in Section 1. The final product is a fully functional modular healthcare application. Plugins are functionally independent but can communicate easily, the UI is accessible, highly contrasted and allows a degree of customization to further improve accessibility. The developed plugins are basic and simple but for demonstration purposes (showcasing data exchange between modules) I believe they work well.

There are perhaps two main shortcomings in this project, the first being PhoneGap integration. During my research into PhoneGap and Node.js, all resources stated that Node.js can be used with PhoneGap without major complications. And that statement would be true, assuming I was not using a templating engine instead of static files. When using both Node.js and a templating engine (EJS in this project), it becomes impossible to create local copies and instead, it has to run on a hosted platform (Heroku in this case) and then linked to the PhoneGap build. What results is a functional albeit not technically local version of the application. The second development task that did not work out as intended was the implementation of a third-party module, which I will discuss in its own section below.

5.1.1 Third-party collaboration

The most notable issue that came up was one of the students deciding to withdraw from the dissertation module; student whose application, a drinks diary, I had been attempting to include as a plugin for my application for a while at that point, as recommended by my supervisor. This particular web application was developed using Angular in a way that required it to be run on a local server, as it does not really use static pages - as is often the case with Angular projects (Jadhav et al., 2015). As the code was abandoned, despite my best efforts at salvaging as much functionality as I could, it would just not work in a satisfactory way. The way my plugin system is implemented requires static code to be inserted into a template to render, which was just not possible with this particular application - as a result, the whole plugin had to be subsequently removed, considerably delaying my progress.

Instead, I collaborated with another student who granted me access to their application - "Get Going", a website/application which aims to help the elderly navigate. As this particular project is completely front-end based and does not require a local server to run, it worked with Simple++ from the very start. However, a few major issues arose. Firstly, this system has a very long loading/startup time, which was even longer when running on a hosted web server. Furthermore, while its responsiveness to viewport changes is great on the web browser, I ran into many issues with how it scaled for a 16:9 UI, especially with top and bottom bars. Finally, its UI is not particularly well suited to a touchscreen device. Rather than struggle to adapt the entire system, I decided to strip it down to the basic functionalities and implemented the main one - the map navigation system.

The overall experience with developing plugins suggests that, as expected, any third-party plugins would have to be created with this platform as the primary target. A bespoke plugin would integrate well and easily into the existing architecture. While it is possible to adapt existing solutions to work with this platform, it is a long process that may involve reworking the entire structure of the plugin.

5.2 Version Control and Timeline

I started development tasks in June, with the first working mockup UI ready at the end of June. The first working prototype with a functional plugin system was ready at the beginning of July. Due to various issues, including personal health and collaboration issues outlined in Section 5.1.1, there was an extended pause in development, with more work beginning in August. Overall, there have been over 60 commits through the length of this project, all pushed to the main GitHub repository (see Figure 5.1).

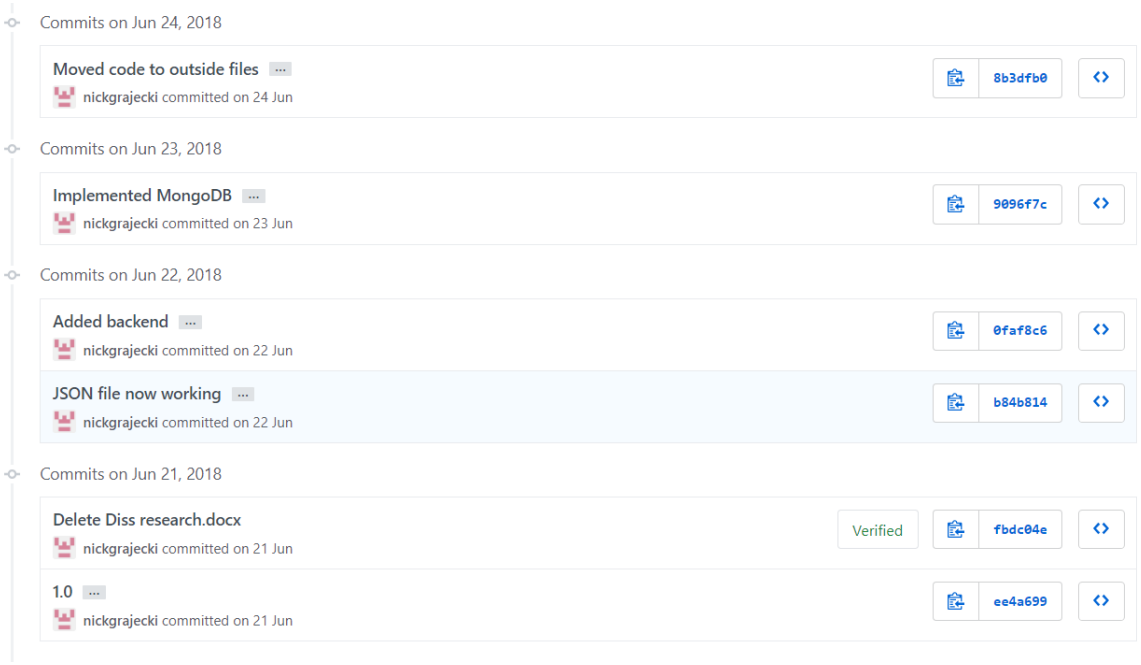

Figure 5.1: GitHub Version Control, nickgrajecki/SimplePlusPlus

The application was deployed to Heroku for easier testing as well as demo access on UEA PCs (which at the time did not have the Node package manager and as such could not run Node.js applications). Heroku automatically pulls any new commits directly from the GitHub repository and deploys a new build of the applications - hence the high amount of commits, as I would push a new commit to test the system externally after any significant change.

5.3 Testing

Once the product was finalised, the next step was to test its functionality and check for any unexpected bugs or problems. First was the accessibility testing - using various

dedicated services to highlight any such issues.

5.3.1 Accessibility Testing

Accessibility testing was conducted using the following services:

- WAVE web accessibility evaluation tool (WAVE, 2018)
- AChecker (2018)
- Web accessibility (2018)

In general, all of these came back with some type of issue. However, after examining the category of issue they were referring to, it became clear that most of these were suggestions rather than outstanding problems, and in some cases not necessarily accurate suggestions. For example, WAVE initially reported some styling errors and an outstanding error in the form of a missing $< h1 >$ tag - which is necessary for screen readers. Additionally, it suggested the contrast, as shown in Figure 5.2, was very low - which is clearly not the case. Some issues with not reading CSS properties correctly may be at fault here.

Figure 5.2: Inaccurate test result, WAVE (2018)

The second tool, ACHecker, categorizes its accessibility review under 3 categories - "Known problems", "Likely Problems" and "Potential problems", denoting a variety of importance. When testing my application, it found the following problems:

- Known problems - 13 instances of "$< i >$ (italic) element used". However, in my system, $< i >$ is used to embed FontAwesome icons, as recommended by their

documentation (FontAwesome, 2018). It is not used as a styling tag and as such it does not impact design or accessibility.

- Likely problems - none detected
- Potential problems - mainly cases of "script may cause flicker" - highlighting almost every script tag in the documents, including references to the jQuery library. This is relevant in only one case - the scripts that animate the screen and enable a white background flash. However, as explained before, these can be turned off and are disabled by default.

Finally, the Web Accessibility tool gave Simple++ a score of 89% (see Figure 5.3). The only error it documented was a single instance of the language not being set in a $< script >$ tag on a single page.

Figure 5.3: Accessibility compliance score, Web accessibility (2018)

5.3.2 Security Testing

Security testing was conducted using an XSS scanner tool as well as manual testing. First, XSS vulnerability was tested using Pentest-Tools (2018) XSS Scanner. As can be seen in Figure 5.4, the overall result indicates that there is no high risk of XSS attacks reported.

Figure 5.4: XSS Scanner results, Pentest-Tools (2018)

Next task was testing Node.js dependencies using the Snyk (2018) service. The results, as visible in Figure 5.5, indicated 4 medium-severity security issues and 1 low severity. However, it turned out that these were a result of an old local version of

Bootstrap that was not in use, and the rest were found within the abandoned code for the drinks diary module. After removing the local Bootstrap files and all the drinks diary code, the subsequent repeat scan found no security risks in the dependencies.

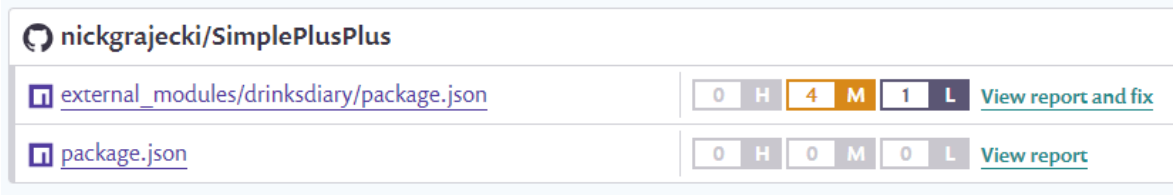

Figure 5.5: Snyk results

The final automated test was conducted using Retire.js (2018) - a penetration test specifically designed for JS and Node.js applications. After scanning my entire application, it found a few security risks - this time, they were all found within the other implemented third-party module, the "Get Going" application.

Figure 5.6: Retire.js results

As can be seen in Figure 5.6, every security risk was caused by importing external jQuery content delivery networks. At the time of writing, 1.12 was the most up to date version and as such there was no existing fix for these.

Outside of external testing, I have conducted manual testing across the app, particularly with the text input fields. When attempting to insert any script code into any text input box, for example, a simple pop-up alert such as:

```
<script>alert('hello')</script>
```

the implemented XSS filter replaces any blacklisted characters to prevent any code from running (see Figure 5.7). I have not been able to successfully replicate an XSS attack within my own testing.

- (2018/09/19) <SCRIPT >alert(String.fromCharCode(72,101,108,108,111,33))</SCRIPT >
- (2018/09/19) <script>alert('hello')</script>

Figure 5.7: XSS filter display

Next, I tested the NoSQL prevention measures using both data input fields as well as */GET* requests. In both cases, when attempting to use blacklisted symbols such as $, the employed content filter redirects the user to an error page that informs them of having used a forbidden character (see Figure 5.8)

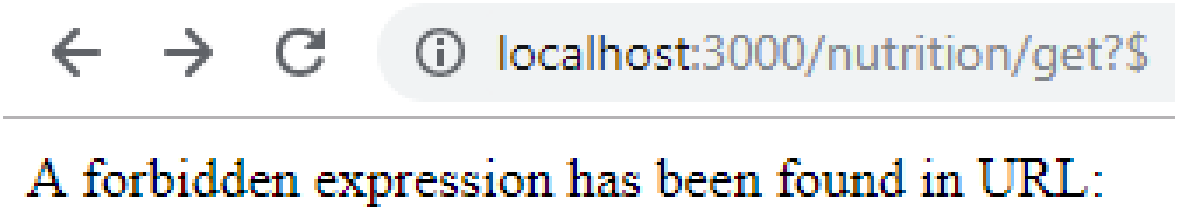

Figure 5.8: Content-filter blocking a URL-based NoSQL injection attempt

As all the tests came back either with very few issues or issues that were easy to fix, irrelevant or outside of my control, I am satisfied with the general security level of Simple++. Overall, I believe that while I ran into many issues throughout the course of developing this system, I am happy with how I have managed to overcome the vast majority of them - and the few that are still outstanding are learning experiences that can be fixed in future versions of the system.

Chapter 6

Conclusion and future work

In this chapter, I discuss the results of my software system development - the achievements, successes, and shortcomings. I briefly overview to what extent I have met the aims and objectives initially set, whether my original hypothesis was accurate, draw some general and personal conclusions and finally make some suggestions for further work.

6.1 Conclusion

I believe that I have largely achieved the vast majority of tasks as outlined in Section 1. Before beginning any work, I have researched modular programming concepts, usable and accessible design for the elderly as well as the best technologies and frameworks to use for this project, as summarised in Chapters 2 & 3. Next, I devised a UI prototype using a combination of HTML, CSS, JS, Bootstrap, and jQuery. I followed the guidelines that I set out for myself, focusing on the accessible experience with a focus on visually impaired users. Next, the modular backend was established using Node.js with Express as the main web application framework and ejs for rendering pages in real time. A few simple plugins were created, primarily to demonstrate their ability to be functionally independent but still capable of communicating. A third-party plugin was implemented with partial success - some design and functionality had to be taken away in order to implement it in a satisfactory fashion. Next, Adobe PhoneGap was used for mobile deployment and testing - again, the results were mixed, as Node.js, when used with a templating engine, cannot be used with PhoneGap natively. However, a solution was found that, while not ideal, does not make a difference to the end user

experience. Finally, testing was done on the finished system to determine any security risks as well as its accessibility and actions taken to remedy any outstanding issues. The majority of the testing did not return any significant risks or risks that could not be immediately removed - which is a success in the final stage of developing this application.

Overall, the system operates as expected. There are some features that could use improvement, such as the aforementioned PhoneGap deployment, the first failed third-party plugin implementation and some issues with JS/Bootstrap not always operating as expected when working in tandem. I initially hypothesized that the frontend UI would be fully functional, perhaps with some minor issues - the final product fits that description. I also predicted that the backend would be functional for custom-made plugins and that it would not work with third-party plugins. This statement is only partially accurate, as out of the two attempted third-party plugins, one was successfully implemented - albeit lacking some content from the original application. If this project was to be started again, I would start doing certain tasks earlier than I originally planned to - PhoneGap implementation especially should have been attempted a lot earlier than it was, as that would have uncovered the underlying issue with using PhoneGap alongside templating engines. These and a few other small errors have definitely become hurdles during this project but ultimately did not cause any issues that affected the final delivery of the system.

Developing such a complex system, from the ground up, whilst having limited previous exposure to JS (and by extension Node.js) proved to be extremely challenging. The languages and tools required a large amount of research and studying to fully understand before even attempting to start developing. Thankfully, with the support of online resources as well as UEA staff and my supervisor, who has been very supportive of any decisions I have taken, I have managed to complete this project to the best of my ability.

6.2 Future considerations and additions

Although the finished product functions as expected, there are a few features that are missing or not functioning quite as planned.

The first feature that could be considered in the future is JSONP support - as all of these plugins were hosted locally, there was no need for ensuring cross-domain request support. However, assuming a third-party plugin was developed that accesses resources from a remote location, all that would be required is to add a function that unwraps the content of the JSONP request (as it is, essentially, a JSON file wrapped inside a function) and then implements its content as it would any other JSON file.

The second task for future development would be better PhoneGap integration - in its existing form, PhoneGap integration does work, but it is not as seamless as it could be. The next step would be to either completely change the way pages are rendered (perhaps base them on static HTML files) or devise a way to render pages on demand when remotely connected to a backend server.

Finally, I would consider either completely removing Bootstrap, or fully committing to its use. The way in which it is employed in this project is scarce - it is mostly used for a few container elements, rather than being the underlying foundation for the entire front-end design. It was used as an ad-hoc patch primarily for the resizing feature that after a long time spent changing and improving, ultimately should have been re-worked from the ground up. Unfortunately, the timeframe of this project did not allow for that. Committing in full to its use or devising a way to achieve all desired functionality without its use would be the preferred way, for cleaner and more maintainable code.

Bibliography

AChecker (2018). Idi web accessibility checker. https://achecker.ca/. Accessed: 2018-09-16.

Adobe (2018). Adobe® phonegap™ build. https://build.phonegap.com/. Accessed: 2018-09-18.

Amalberti, R., Nicklin, W., and Braithwaite, J. (2016). Preparing national health systems to cope with the impending tsunami of ageing and its associated complexities: towards more sustainable health care. *International Journal for Quality in Health Care*, 28(3):412–414.

Bergmo, T. S. (2015). How to measure costs and benefits of ehealth interventions: an overview of methods and frameworks. *Journal of medical Internet research*, 17(11).

Boulos, M. N. K., Wheeler, S., Tavares, C., and Jones, R. (2011). How smartphones are changing the face of mobile and participatory healthcare: an overview, with example from ecaalyx. *Biomedical engineering online*, 10(1):24.

Buenos Aires Ciudad (2018). Simple inclusión digital para los más grandes. http://www.buenosaires.gob.ar/desarrollohumanoyhabitat/personasmayores/massimple. Accessed: 2018-09-18.

Burkhard, M. and Koch, M. (2012). Evaluating touchscreen interfaces of tablet computers for elderly people. In *Mensch & Computer Workshopband*, pages 53–59.

Chang, H.-T., Tsai, T.-H., Chang, Y.-C., and Chang, Y.-M. (2014). Touch panel usability of elderly and children. *Computers in Human Behavior*, 37:258–269.

Chaniotis, I. K., Kyriakou, K.-I. D., and Tselikas, N. D. (2015). Is node. js a viable option for building modern web applications? a performance evaluation study. *Computing*, 97(10):1023–1044.

Charland, A. and Leroux, B. (2011). Mobile application development: web vs. native. *Queue*, 9(4):20.

Davis Giardina, T., Menon, S., Parrish, D. E., Sittig, D. F., and Singh, H. (2013). Patient access to medical records and healthcare outcomes: a systematic review. *Journal of the American Medical Informatics Association*, 21(4):737–741.

Demiris, G., Finkelstein, S. M., and Speedie, S. M. (2001). Considerations for the design of a web-based clinical monitoring and educational system for elderly patients. *Journal of the American Medical Informatics Association*, 8(5):468–472.

Dennis, J. B. (2017). Principles to support modular software construction. *Journal of Computer Science and Technology*, 32(1):3–10.

Fleischer, H., Adam, M., and Thurow, K. (2015). A cross-platform modular software solution for automated data evaluation applied in elemental and structural mass spectrometry. In *Automation Science and Engineering (CASE), 2015 IEEE International Conference on*, pages 758–763. IEEE.

FontAwesome (2018). Basic use. https://fontawesome.com/how-to-use/on-the-web/referencing-icons/basic-use. Accessed: 2018-09-10.

Giang, N., Ha, M., and Kim, D. (2014). Cross domain communication in the web of things: a new context for the old problem. In *Proceedings of the 23rd International Conference on World Wide Web*, pages 135–138. ACM.

Hahn, J. F. and Ryckman, N. (2012). Modular mobile application design. *Code4Lib*.

Heitkötter, H., Hanschke, S., and Majchrzak, T. A. (2012). Evaluating cross-platform development approaches for mobile applications. In *International Conference on Web Information Systems and Technologies*, pages 120–138. Springer.

Holmes, S. (2015). *Getting MEAN with Mongo, Express, Angular, and Node*. Manning Publications Co.

Holzinger, A., Searle, G., Kleinberger, T., Seffah, A., and Javahery, H. (2008). Investigating usability metrics for the design and development of applications for the elderly. In *International Conference on Computers for Handicapped Persons*, pages 98–105. Springer.

Hwangbo, H., Yoon, S. H., Jin, B. S., Han, Y. S., and Ji, Y. G. (2013). A study of pointing performance of elderly users on smartphones. *International Journal of Human-Computer Interaction*, 29(9):604–618.

Jadhav, M. A., Sawant, B. R., and Deshmukh, A. (2015). Single page application using angularjs. *International Journal of Computer Science and Information Technologies*, 6(3):2876–2879.

Kobayashi, M., Hiyama, A., Miura, T., Asakawa, C., Hirose, M., and Ifukube, T. (2011). Elderly user evaluation of mobile touchscreen interactions. In *IFIP Conference on Human-Computer Interaction*, pages 83–99. Springer.

Lavy, I. and Rami, R. (2018). The circumstances in which modular programming becomes the favor choice by novice programmers. *International Journal of Modern Education and Computer Science*, 10(7):1.

Malavolta, I., Ruberto, S., Soru, T., and Terragni, V. (2015). Hybrid mobile apps in the google play store: An exploratory investigation. In *Proceedings of the Second ACM International Conference on Mobile Software Engineering and Systems*, pages 56–59. IEEE Press.

McLaughlin, B. (2011). *What is node?* " O'Reilly Media, Inc.".

Newell, A. F. and Gregor, P. (2002). Design for older and disabled people–where do we go from here? *Universal Access in the Information Society*, 2(1):3–7.

NHS (2018). Nhs apps library. https://apps.beta.nhs.uk/. Accessed: 2018-09-17.

Nicolau, H. and Jorge, J. (2012). Elderly text-entry performance on touchscreens. In *Proceedings of the 14th international ACM SIGACCESS conference on Computers and accessibility*, pages 127–134. ACM.

npm (2018). npm - content-filter. https://www.npmjs.com/package/content-filter. Accessed: 2018-09-19.

Ojamaa, A. and Düüna, K. (2012). Security assessment of node. js platform. In *International Conference on Information Systems Security*, pages 35–43. Springer.

Pentest-Tools (2018). Xss scanner - online scan for cross-site scripting vulnerabilities. https://pentest-tools.com/website-vulnerability-scanning/xss-scanner-online. Accessed: 2018-09-16.

Retire.js (2018). Javascript vulnerability scanner. https://retirejs.github.io/retire.js/. Accessed: 2018-09-17.

Seydnejad, S. (2016). *Modular Programming with JavaScript*. Packt Publishing Ltd.

Shotts, K. (2014). *PhoneGap 3. x Mobile Application Development Hotshot*. Packt Publishing Ltd.

Snyk (2018). Open source security platform. https://snyk.io/. Accessed: 2018-09-17.

Spett, K. (2005). Cross-site scripting. *SPI Labs*, 1:1–20.

Staicu, C.-A., Pradel, M., and Livshits, B. (2016). Understanding and automatically preventing injection attacks on node. js. Technical report, Tech. Rep. TUD-CS-2016-14663, TU Darmstadt, Department of Computer Science.

Sullivan, B. (2011). Server-side javascript injection. *Black Hat USA*.

Teixeira, P. (2012). *Professional Node. js: Building Javascript based scalable software*. John Wiley & Sons.

Tilkov, S. and Vinoski, S. (2010). Node. js: Using javascript to build high-performance network programs. *IEEE Internet Computing*, 14(6):80–83.

Trauer, J. M., Ragonnet, R., Doan, T. N., and McBryde, E. S. (2017). Modular programming for tuberculosis control, the "autumn" platform. *BMC infectious diseases*, 17(1):546.

W3C Web Accessibility Initiative (2018). Introduction to web accessibility. https://www.w3.org/WAI/fundamentals/accessibility-intro/. Accessed: 2018-09-12.

Walls, C. (2014). *Modular java: creating flexible applications with OSGi and Spring*. CreateSpace Independent Publishing Platform.

WAVE (2018). Wave web accessibility tool. http://wave.webaim.org/. Accessed: 2018-09-20.

Web accessibility (2018). Test your site for accessibility. https://www.webaccessibility.com/. Accessed: 2018-09-16.

Wei-Ping, Z., Ming-Xin, L., and Huan, C. (2011). Using mongodb to implement textbook management system instead of mysql. In *Communication Software and Networks (ICCSN), 2011 IEEE 3rd International Conference on*, pages 303–305. IEEE.

Wong, S., Cai, Y., Kim, M., and Dalton, M. (2011). Detecting software modularity violations. In *Software Engineering (ICSE), 2011 33rd International Conference on*, pages 411–420. IEEE.

Zimmermann, G. and Vanderheiden, G. (2008). Accessible design and testing in the application development process: considerations for an integrated approach. *Universal Access in the Information Society*, 7(1-2):117–128.