Tampere University

Mikko Koskenranta

# DESIGN AND IMPLEMENTATION OF A WEB-BASED USER INTERFACE FOR A DIFFERENTIAL ION MOBILITY SPECTROMETER

Faculty of Information Technology and Communication Sciences

Master’s thesis

2019-07-01

## ABSTRACT

**MIKKO KOSKENRANTA**: Design and implementation of a web-based user interface for a differential ion mobility spectrometer
Tampere University
Master of Science Thesis, 56 pages, 2 Appendix pages
July 2019
Master's Degree Programme in Information Technology
Major: Software Engineering
Examiners: Professor Kari Systä and M.Sc. Anton Kontunen

Keywords: user interface, usability testing, single page application, JavaScript, differential ion mobility spectrometer

This thesis covers the design and development of a user interface for a differential ion mobility spectrometer (DMS). The user interface is implemented as a single page web application. This way the DMS device can be controlled both from its own screen and by using a web browser on an external device. The user interface must work well in both usage environments, have smooth navigation between views and have good performance even on the limited hardware of the DMS device.

An interactive user interface prototype was first developed to base the final implementation on. The development of the prototype consisted of making mockups of the user interface and then connecting the images together into the interactive prototype. The initial mockups were made based on requirements of the DMS system as a whole and feedback from the development team of the device. The prototype was then used in a usability test that attempted to find usability problems in the initial design. The test revealed multiple minor problems that were fixed in the initial design to make it fit for implementation.

The second part of the thesis attempts to find the best JavaScript library for the user interface of the DMS device. Three libraries that were popular at the time of writing this thesis, Angular, React and Vue, were compared. From them, React was chosen for the user interface, because of its reliance on standard JavaScript features, because it has a more flexible component system than the others and because it seems the most mature as a library.

The user interface was finally implemented based on the user interface mockups and the selected single page application library. The final implementation works well with the rest of the software on the device, is usable from the DMS device and external devices and is fit for managing all functionality of the DMS device.

# TIIVISTELMÄ

**MIKKO KOSKENRANTA**: Web-pohjaisen käyttöliittymän suunnittelu ja toteutus differentiaali-ionimobiliteettispektrometriin
Tampereen yliopisto
Diplomityö, 56 sivua, 2 liitesivua
Heinäkuu 2019
Tietotekniikan diplomi-insinöörin tutkinto-ohjelma
Pääaine: Ohjelmistotuotanto
Tarkastajat: Professori Kari Systä ja DI Anton Kontunen

Avainsanat: käyttöliittymä, käytettävyystestaus, single page application, JavaScript, differentiaali-ionimobiliteettispektrometri

Tämä työ kattaa differentiaali-ionimobiliteettispektrometrin (DMS) käyttöliittymän suunnittelun ja kehityksen. Käyttöliittymä toteutetaan web pohjaisena käyttäen single page application arkkitehtuuria. Näin DMS-laitetta voidaan hallita sekä sen omalta näytöltä, että web selaimella ulkopuolisilta laitteilta. Käytöliittymän täytyy toimia hyvin kummassakin käyttöympäristössä, navigoinnin sen näkymien välillä tulee olla sulavaa ja sen suorituskyvyn tulee olla hyvä jopa DMS-laitteen rajatulla laitteistolla.

Lopullista käyttöliittymän toteutusta varten kehitettiin ensin interaktiivinen prototyyppi, johon toteutus voitaisiin pohjauttaa. Prototyypin kehitys koostui käyttöliittymän mallikuvien tekemisestä ja näiden kuvien liittämisestä yhteen interaktiiviseksi prototyypiksi. Alustavat mallikuvat pohjautuvat koko DMS järjestelmän vaatimuksiin ja laiteen kehitystiimin antamaan palautteeseen. Prototyyppiä käytettiin käytettävyystestissä, joka pyrki löytämään käytettävyysongelmia käyttöliittymän alustavassa suunnitelmassa. Testi paljasti monia pieniä ongelmia, jotka voitiin korjata alustavassa suunnitelmassa. Näin siitä saatiin sopiva toteuttamista varten.

Työn toinen osa pyrkii löytämään parhaan JavaScript kirjaston DMS laitteen käyttöliittymää varten. Kolmea kirjastoa, jotka olivat työn kirjoittamisen aikaan suosittuja, vertailtiin: Angular, React ja Vue. Näistä React valittiin käyttöliittymää varten, koska se tukeutuu JavaScript:n standardiominaisuuksiin, sen komponenttijärjestelmä on joustavampi kuin muiden ja se vaikuttaa kirjastona kypsimmältä.

Käyttöliittymä toteutettiin siitä tehtyjen mallikuvien ja valitun single page application kirjaston pohjalta. Lopullinen toteutus toimii hyvin järjestelmän muun ohjelmiston kanssa, on käytettävissä DMS laitteelta ja ulkopuolisilta laitteilta ja sopii DMS laitteen kaiken toiminnallisuuden hallintaan.

d

# PREFACE

First, I would like to thank everybody at Olfactomics for allowing me to do my thesis from such an interesting topic. It has been great to be allowed to work on something that is interesting to me personally, while also doing something useful for the larger DMS device project.

I would like to thank my thesis supervisor Kari Systä for helping with the every-day problems of working on the thesis and related tasks. I would also like to thank my second supervisor Niku Oksala and my examiner Anton Kontunen for helping with the thesis work. From Olfactomics, I would like to especially thank Markus Karjalainen and Osmo Anttalainen for helping with the user interface work.

This thesis was supported by Competitive State Research Financing of the Expert Responsibility area of the Tampere University Hospital and Pirkanmaa Hospital District grants 9s045, 151B03, 9T044, 9U042, 150618, 9U042, 9V044 and 9X040.

In Tampere, Finland, on 1 July, 2019

Mikko Koskenranta

## CONTENTS

# LIST OF FIGURES

# LIST OF SYMBOLS AND ABBREVIATIONS

| | |
|---|---|
| API | Application Programming Interface |
| DMS | Differential ion Mobility Spectrometer |
| HTTP | Hypertext Transfer Protocol |
| JSON | JavaScript Object Notation |
| JSX | JavaScript XML |
| SPA | Single Page Application |
| URL | Uniform Resource Locator |

# 1. INTRODUCTION

This thesis describes the design and implementation of a user interface for a differential ion mobility spectrometer (DMS). The work will cover designing the user interface and implementing it using modern web technologies. The user interface will be primarily designed to work on the touch screen of the DMS device, but also on the personal computers of users. In that case the user interface will be served by the internal server of the DMS device. The two usage environments have different interaction methods. A personal computer used to control the DMS device most probably has a mouse and a keyboard, while the DMS device itself has a touch screen. This emphasises the need for the user interface to be adaptive for multiple environments.

A differential mobility spectrometer can be described as a sort of an electronic nose [27]. The working principles of a DMS device are similar to those of an actual nose. The details of the technology are not necessary for this thesis, but will be summarised extremely shortly. A gas sample is pumped inside the device, where it interacts with an ionization source, producing sample ions[21]. These ions travel to a measurement chamber, where two perpendicular electric fields are used to separate the sample ions from each other. By manipulating the strength of the electric fields, sample ions with different mobilities will reach a detector plate at different field voltages. By using several different voltage values, a comprehensive representation of the chemical composition of the gas sample can be produced.

Olfactomics Oy is developing the new DMS device to better fit their requirements. The new device will, amongst other things, be faster and more flexible with how it can perform scans than other similar devices. The new DSM device needs a built-in graphical user interface so it is possible to use the device just by itself. There should be no need to use an external computer with some sort of separate application to manage the device. A user interface is also needed so users not familiar with computer technology can operate the device easier, and without having to learn non-graphical interaction methods such as using an HTTP API (Hypertext Transfer Protocol Application Programming Interface). The user interface makes the usage of the device easier to people not familiar with the differential ion mobility spectrometry technology as well. They possibly only have to select a preset of parameters and start a scan.

In addition to working on the touch screen of the DMS device, it is still important to have the user interface accessible from external devices. That way the DMS device can be physically installed anywhere and be used from a more accessible location. The device can then be installed as part of a bigger system and still be easily controlled externally. For more complex use cases, the HTTP API of the DMS device is built to be developer friendly, so it can be used to programmatically access the features of the device. The API design is outside the scope of this thesis however.

The user interface will be built to run in a web browser. While using web technologies potentially results in loss of some performance and memory efficiency of native software, which can be important on the limited hardware of the internal computer of the DMS device, using them makes developing and maintaining the user interface a lot easier. Web pages also offer a good base for maintaining the usability of the user interface on multiple form factors. A web page can work as the internal user interface of the DMS device when displayed in a browser loading the page from inside the device. At the same time the same user interface can be used externally by loading it through local area network. The same server software can be used for both use cases.

The user interface is required to not only operate the measurement hardware, but to also let the user control the software of the DMS device. This includes, for example, viewing saved measurement data, manipulating device settings and managing saving data to online services. The user interface needs to be more interactive than many traditional web pages, which focus more on just displaying data. This means the user interface will be more complex than those pages. On the other hand, the user interface will not have many simultaneous users, which simplifies some aspects of web application developing, such as handling a lot of users accessing the same back-end resources.

The web page will be built as a single-page application (SPA). This means that instead of the traditional model of building server-rendered HTML pages that are linked together with hyperlinks, the whole web application is just one page. All content is loaded and shown using JavaScript and requests to a HTTP API. This way the end product will be close to how a native application would work. It becomes easier to make different parts of the user interface interconnected and features like global errors and notifications are easier to handle.

One main research topic of this thesis is finding a best possible JavaScript library to build the user interface on. The three libraries compared are Angular, React and Vue, which seem to be some of the most popular libraries at the time. The libraries and selection process are described in detail in chapter 5.

In addition to developing the user interface, a small usability experiment was be conducted as part of the thesis. The experiment was used to validate an initial design of the user interface using mockup images connected into an interactive prototype. The testees were asked to perform tasks using the prototype. Their performance was then observed. The results were used to improve the user interface mockups the actual user interface was be based on.

The second chapter of the thesis will discuss previous works that have covered similar applications. It will also discuss some of the technologies used in the user interface. The third chapter describes the surrounding system the user interface operates on. This includes the back-end for the user interface and in less detail the rest of the DMS system. The designing of the user interface is described in the fourth chapter. The design process from initial design to the usability testing of the design to finalising the plans for the user interface is covered. Fifth chapter covers the selection of the user interface library and the implementation of the user interface. The final chapter is the summary, where the design and implementation process is wrapped up.

# 2. BACKGROUND

This chapter goes over literature about some of the topics of this thesis. Previous works of user interfaces of scientific measurement devices, and their differences to the user interface developed in this thesis, are reviewed. The meaning of the term single page web application will also be discussed, together with some theory about them. Finally the usage of modern JavaScript and CSS methods during the development process is described.

## 2.1 Measurement device user interfaces

There are relatively few works about user interfaces built using web technologies for scientific devices. It seems that even though web based user interfaces seem to be popular in products aimed at consumers, they have not become as popular in the scientific professional market.

One of the inspirations for building the user interface of the DMS device using web technologies comes from an older DMS device currently used by Olfactomics. It includes a web based user interface that has many of the same basic features that the one built in this thesis aims to have. The user interface of that device was never fully finished though, as its manufacturer stopped developing it. The user interface includes many non-functional features, making it more of a prototype than a finished product. As most of the people involved in defining requirements for the new DMS device have plenty of experience with this older device, its influence can be seen in parts of the new device.

Many general features from the user interface of the old DMS device are included in the new user interface. This includes basic features such as starting scans, viewing scan results and searching for and inspecting old scan results. Many more features included in the definitions of the new user interface, such as graphically editing scan parameters, projects and properly viewing system state, were not part of the old device. Their invention is most probably influenced by needs that have emerged with using the older device though. The features included in the user interface of the old device were sometimes unfinished and would not always be user friendly. Compared to this, the new user interface aims to offer all features needed in the usage of such a device. It will also be user friendly enough to be usable by as many users as possible.

Damian Wanta et al. describe their work on a web based user interface for a device measuring electrical permittivity of materials in their article "WWW Interface for an Electrical Capacitance Tomography System"[26]. The system described in the article is relatively similar to the new DMS system, and the only example of a similar system in addition to the previously mentioned older DMS device that was found. The system in the article has an FPGA board executing the measurements, connected to a Linux based computer that controls the FPGA and runs a user interface server. The user interface can be accessed from an external computer using a web browser once the computer has been connected to the measurement device through an Ethernet connection.

The user interface the article describes is developed as a single page application[26]. The technologies used are more traditional, using Bootstrap and jQuery as libraries, instead of a fully JavaScript based library like Angular, React or Vue. The web user interface is used to configure and start measurements, load and visualise the measurement results and view and reset the FPGA hardware. Compared to the user interface described in this thesis, the user interface of the article is less complex. It does not include considerations for multiple simultaneous users and there is no way for the server to communicate to the client to allow interactivity. The user interface is also meant to only work on a external computer and the measurement device itself does not include a internal screen or other input methods.

More traditional desktop user interfaces seem to be more common. For example, H. Oji et al. describe a desktop user interface for a X-ray photoemission spectroscopy system in their article "An automated HAXPES measurement system with user-friendly GUI for R4000-10 keV at BL46XU in SPring-8"[18]. While the user interface is built with Visual Basic for only Windows desktops, it still communicates with the back-end via a TCP/IP connection. The user interface is built on top of an older command line tool, which uses a TCP/IP connection as well. The composition of the measurement system is slightly more complicated than in the DMS device of this thesis, as the measurement hardware of article consists of multiple control computers and measurement devices. There are similar features in the user interface though. It allows the user to create and edit measurement configurations. Once a configuration has been saved, it can be loaded by the user and inputted to the device. New scans can then be done using the this configuration.

## 2.2 Single page applications

Single page applications are a relatively modern phenomenon in web application development. Traditionally web pages have been HTML documents linked together using hyperlinks, with CSS added for styling. JavaScript might be used to add some interactivity to the document. Navigating to an another part of a site or application means loading a new HTML document from a server. Dynamic data, for example text from a database, is added to the documents by the server before serving them to the client.

There exists an alternative method to fill dynamic data to a HTML document that works after it has been loaded. JavaScript can be used to load data separately from loading the HTML document. By using JavaScript, data can be fetched from an HTTP API that can serve the data from a database or other source. That data can then be processed and inserted to the HTTP document on client side. This works similarly to how the data would be handled and inserted into a HTML template by a server in a more traditional web page.

More data can be fetched at any point using this alternative method, which means there is no need to reload the whole HTTP document when more data is wanted.

By fetching data using just JavaScript, a whole web page can be built using a single HTTP document that must only be loaded from the server once. That document must pull the JavaScript code needed for the page with it. All of the content of the HTTP document that the user sees and interacts with can then be dynamically rendered using JavaScript code. This method of development makes a web application a single page application [8].

The user interface of the DMS device requires functionality like fluid navigation between different views, global state across the application, such as the state of the DMS hardware and user interface language, and notifications. These are features more commonly seen in native desktop applications. The user interface of the DMS device should function closely like one, especially when used on the built-in screen. These kinds of features are easier to implement when building the web user interface as a single page application. As the HTML page of the application does not need reloading, application state can be loaded to and stored in memory. Notifications are easy to display in any part of the application, as receiving them can be built as part of the whole application. Navigation between different views is fluid as they are all loaded at the beginning and switched out using JavaScript without necessarily making any requests to the server.

While single page applications provide many advantages, there are also possible disadvantages to consider. As the whole application is built as a single web page, by default the Uniform Resource Locator (URL) of the whole application is the same[22]. The URL is formed, among other things, by the address of the web site or application and the path the user is in inside that site. For example, by default the URL for scan management and device settings in user interface of the DMS device may just both be "http://192.168.0.2/".

This is inconsistent with how traditional web pages normally work, where the URL represents the users location on the site. This is because traditionally the server serves the user the HTML file that on the server is located at the path indicated by the URL. Without this there would also be no way to link to or bookmark different parts of the web site.

It is possible to mitigate this in single page applications by dynamically manipulating the URL using JavaScript [22]. This way the single page application can change the URL to match the location of the user on the application. This also works to the other direction, where the user enters the URL of a view in the application and that view is loaded. The server must be set up in a way that it serves the same single page application from all URL paths entered by the user.

This behaviour in single page applications is usually handled by a component called a router. A router in a single page application combines the path in the URL the user has entered with certain views in the application. When the users enters a URL that matches a path configured in the router component, it automatically shows the user the view in the application specified by the configuration. For example, with a router the URL for scan management and device settings in the user interface of the DMS device could be "http://192.168.0.2/scans/" and "http://192.168.0.2/settings/" respectively. These URLs both load the same single page application, but the router of the application makes sure the user is shown the view of the application indicated by the URL.

When the user clicks a link, the click event can be handled by JavaScript code. It manipulates the URL to match the view the user is supposed to be navigated to. The router then switches to that view as it matches the now changed URL. This way the whole navigation of a single page application can be built around the router.

Another possible disadvantage of single page applications is that they are often slower to load than traditional web pages. They must load all of the applications JavaScript code, style sheets and other static content when first loaded. Traditional web pages can load just the needed data on each document loaded. This should not be a big problem in the DMS device, as the application is not expected to be reloaded often. Because of this, a another problem could emerge though. Traditional web pages are reloaded often and memory leaks in the JavaScript code do not often matter. In single page applications, they can add up, and cause performance issues after long use[15]. Memory leaks must be taken into consideration while developing the user interface of the DMS device, as especially on the built-in screen the user interface could run for days at a time.

## 2.3 Using modern JavaScript features during development

When discussing JavaScript based web application development, for example single page applications, one problem is usually quite prominent. The application has to run on many different browsers and different versions of those browsers, and all of them have different levels of support for newer JavaScript features [11]. This usually means that while new features are being developed, and support for them is being added to new browser versions, they can not actually be used in development. Using the new features would mean, that the web application only works on a small subset of browsers used by the user base.

While it is possible to develop web applications using only well supported JavaScript features, that would make the development process a lot harder. Some of the new features are meant to make some operations easier, like checking if an element is in an array with a simple function instead of having to manually loop through the array. Others add completely new concepts to the language, like the ability to use classes (although this could also be classified as just an ease-of-life improvement, but that is outside the scope of this thesis).

There are a few ways to get around this problem. One way is to use polyfills. Polyfills are features of the JavaScript language that are written in JavaScript without using the feature itself. They are meant to replace that feature, if it is not natively supported by the environment [20, chapter 30]. Polyfills are usually written so that they are only used if the actual feature is not present. As native implementations can be optimised by the JavaScript engine of the web browser, they can provide better performance if they are available.

Polyfills use the fact that JavaScript is an object-based, prototype inheritance based language. All objects have a prototype object [3] and inherit all properties of their prototype. These properties can be changed at runtime to add new features to the object, and all objects that use that object as their prototype. Most object prototypes in JavaScript can be edited, including the built-in ones.

Editing a prototype object can be used to inject the polyfill code directly into it. This allows a polyfill to work like an actual native feature would. For example, polyfilling would work with the previously mentioned feature of checking if an array has an element. The "includes" method, that arrays in newer JavaScript versions have, can be polyfilled to the prototype of the Array object. After that, all arrays created have that method available. The polyfilled method can then be used like its native counterpart, meaning there is no need to iterate over the array, or use other workarounds, on any platforms.

Polyfilling will not work in all cases however. Some new features, like the class keyword, can not be added in by adding members to a prototype object. Keywords are not part of the object model, so they must be supported by the JavaScript engine the code is executed in. Supporting them would actually require changes to the engine used.

Another case where polyfilling is not optimal, is when newer features are supported by browsers, but they are behind vendor prefixes. When a new feature is implemented by a browser, a prefix can be added to the methods of the new feature, that tell the feature is not finished yet [25]. The prefix indicates which browser is the one implementing the feature. This way, if the feature might yet not be properly standardised, or the implementation might still be buggy, developers know to work around the possibly unique implementation of that browser.

For developers, this means that when using a prefixed feature, they must use the correct prefix for all different browsers. This could involve checking what browser the code is running in, and then calling prefixed implementation of that browser. If the new feature is something that can be polyfilled, the polyfilled version of the feature could be used. However, this way the performance gains of the native implementation are missed on.

Prefixes are also used in CSS. New CSS features that are not yet finalised are added in with prefixes. This works basically the same way as previously described with JavaScript prefixing. For example, if the ability to set the background colour of an element was a new feature, one would first include the non-prefixed "background-color" property to the CSS file. Then they would add prefixed versions. If Firefox required prefixing for the feature, it would be necessary to add the "moz-background-color" property as well. Firefox would then disregard the unsupported "background-color" tag and use the prefixed version it recognises. CSS prefixing seems to be more common than JavaScript prefixing, and is a more common problem in web development. JavaScript code can not add compatibility for new CSS features, so there is no CSS equivalent of polyfilling.

Using prefixed features is in theory discouraged [25]. The features are experimental and should be given time to mature before usage in production quality projects. Using prefixed features is very common in web development though. This is in part because some features stay prefixed for relatively long. Most of the time browsers remove their prefix as they consider the feature ready, but for some browsers that might take a long time. If the feature is wanted by developers, most disregard the recommendations and use it anyway. This is especially true with prefixed CSS features.

While the usage of prefixed features is discouraged, they are still following a some kind of standard [14], even if the standard is still not finished or the implementation of it is lacking. Using prefixed features is not exploiting any bugs or design flaws that might make using them volatile. The feature might change slightly or add new aspects before being finished properly.

Eventually, when it is ready, all browsers should move to use a feature unprefixed. At that point, the prefixes of prefixed features can be removed from any projects if they were coded manually. As it has been established, using prefixed features is a valid strategy when developing web applications. Using them without worrying about manually adding or removing the prefixes would be useful though.

For these cases, a compiler can help the developer with using new JavaScript and CSS features. In native software development, a compiler usually refers to the application that translates a human processable language, like C, into machine language. In web development terms, a compiler translates JavaScript and CSS code using features not supported by older browsers into code they support. For example, when the class keyword is used in JavaScript code, the compiler compiles it into the more traditional function prototype style. That is supported by basically all JavaScript engines. This way the developer can use newer, better features, and the compiler makes sure the resulting code works on all platforms. For CSS, the compiler can make properly prefixed versions of standard CSS keywords. That way the features work on all browsers that support the feature in some way.

A compiler is also useful for catching less known incompatibilities between browsers. Some JavaScript or CSS features might have bugs or other inconsistencies that can be hard to remember. A compiler can account for them and make sure the compiled code is done so that it works around these potential problems.

There are also compilers that can be used to compile JavaScript from derivate languages. One such language is the Microsoft developed TypeScript [16]. Its main addition to regular JavaScript is strong typing. It also adds other new concepts, such as interfaces, that are popular in other languages. The aim is to make JavaScript code easier to maintain in larger projects. In addition to its own features, TypeScript incorporates new JavaScript features not yet available in most JavaScript engines. The use of TypeScript is required by some large web frameworks, such as Angular. In addition to those, many other frameworks at least support using it, if the developer chooses to.

TypeScript is compiled similarly as regular JavaScript with new features, by using a JavaScript compiler. In addition to compiling the TypeScript code to JavaScript, the compiler does the same work as a JavaScript compiler would and only uses features supported by older browsers in the resulting code. The result is JavaScript code that should run on most browsers.

# 3. THE DMS SYSTEM

This chapter describes the surrounding system the user interface runs on. This includes the hardware of the DMS device and some back-end software indirectly related to the user interface. The requirements of the DMS device are also described briefly.

## 3.1 Hardware

The DMS device will be built in a self-contained chassis. The form factor will be comparable to a desktop tower computer on its side. The only external connection required is a power cord.

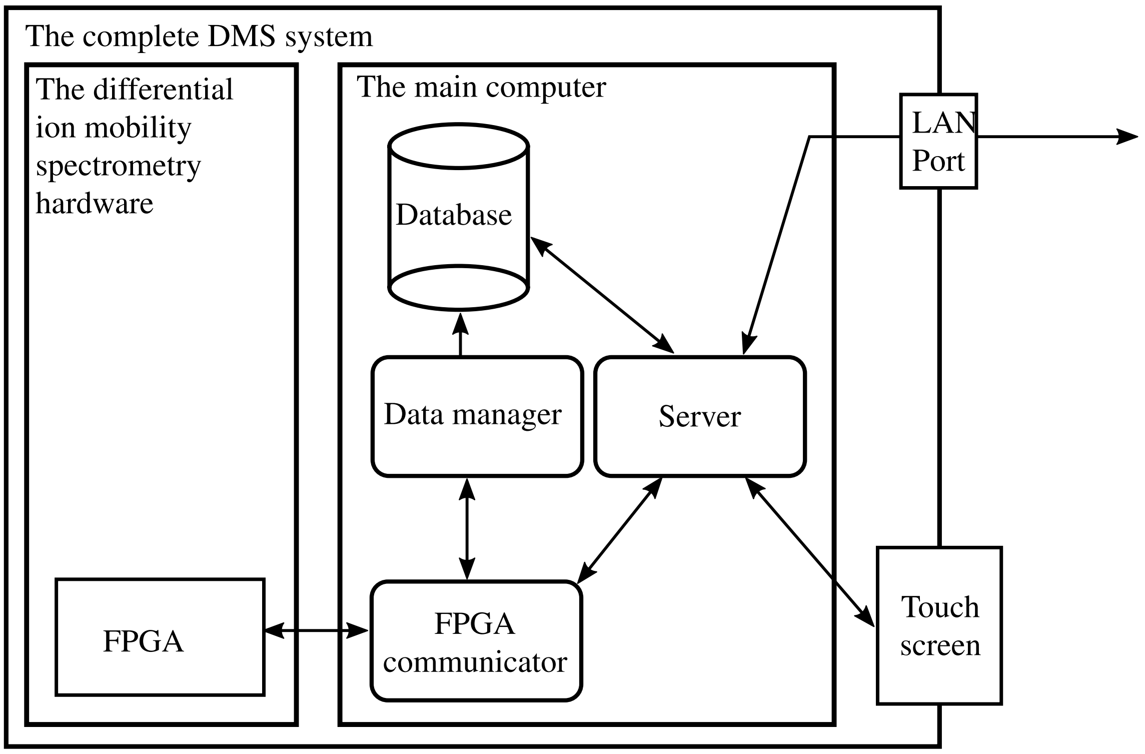


***Figure 3.1**. The DMS device system*

At the center of the hardware of the DMS device is a main computer that controls both the measurement hardware and the user facing functionality. A Raspberry Pi is used in this role at least for first hardware versions. It is widely available and affordable while also being small and powerful enough for the needs of the device. On the Raspberry Pi runs the standard Debian based Raspian Linux distribution. This means the software for the main computer is easy to write as there are no special needs from the operating system.

The actual measurements are controlled by a dedicated FPGA board. This ensures the DMS device is as fast as possible in doing measurements. The FPGA also manages all of the hardware responsible for performing the measurements. The main computer can control the measurements through the FPGA by, for example, stopping and starting them.

The FPGA is capable of doing the measurement process autonomously according to a measurement configuration uploaded to it. The role of the main computer is to upload the measurement configuration to the FPGA and then start a measurement using that configuration when needed. There are also other operations possible between the main computer and the FPGA, such as checking measurement progress and manually adjusting different hardware controllers managed by the FPGA. The communication is two-way, so both devices can send messages to the other when needed.

To interact with the user, the DMS device has several user facing interaction methods. To use the user interface locally, the chassis has a built-in 7 inch touch display. This way there is no necessity for a mouse or other pointing devices. The display supports multiple touch points. The device will have several USB ports that directly connect to the main computer. This way connecting a mouse and a keyboard is possible if wanted. The USB ports can also be used to backup data from the device into an external storage device and perform system updates. Finally, there is a software-controlled power button to power the device on and off.

The DMS device can be connected to a local area network by connecting it to a router using a normal Ethernet connection. After that the user interface and the HTTP API can be used from other devices connected to the same network, assuming that the network is configured correctly. There will not be a way to use these features over internet, even if there is internet connectivity from the local area network. The device itself can connect to the internet for setting system time, cloud functionality and system updates. It will not include support for wireless connections to ease development and certification. The wireless modules on the Raspberry Pi will be disabled.

## 3.2 Software

All software written as part of the development of the DMS device for the main computer is written in JavaScript for Node.js. This includes not only the HTTP and web socket server, but also the software responsible for communicating with the FPGA and managing the internal SQLite database. As all the timing-sensitive operations are performed on the FPGA, there is no need to write the software on more low-level languages.

The Raspberry Pi acting as the main computer runs a regular desktop version of the official Raspbian Linux distribution. It is a custom version of Debian that includes drivers and is optimised for running on different versions of Raspberry Pi [19]. It is officially supported by the Raspberry Pi Foundation. Raspbian includes a standard desktop interface using the PIXEL desktop environment [24].

The official Raspbian desktop is used as a base for the user interface of the DMS device. When the device is turned on, the operating system boots into the PIXEL desktop. The user will not be able to interact with the desktop, but a full screen Chromium browser instance is opened. The browser opens to the web user interface. This way the web user interface can be loaded and used the same way it would on an external computer. The user interface can look like a native, integrated user interface. The native controls of the browser are disabled to not allow the user attempt to visit additional web sites.

The user interface is served by a simple Node.js based server. The same server software is also used for the HTTP API and the WebSocket connection of the user interface. The user interface can be loaded from the local area network the DMS device is connected to by browsing to the IP address of the device.

The user interface loads all of its dynamic data from the HTTP API, which is also meant for end users to programmatically use the DMS device. The API is designed according to REST design style. This means design decisions such as organising all data as resources and statelessness [7, chapter 5]. This fits the use case of the DMS device well, as the API is mostly used to access and manage resources such as parameters, projects, previous measurements and system settings. Actions, like starting a new scan, are mapped as resources that execute the action when requested.

The internal server of the DMS device also incorporates a WebSocket server for signalling changes in system state in real time. This includes events like a measurement starting or stopping or the current active configuration preset being changed. While the WebSocket interface is important for the user interface, it too can be used by the end users to built their own applications.

The server fetches the data required by the HTTP API and WebSocket from one of several data sources. The serial communications application, which handles communication with the FPGA, provides data about its status. For example data about the measurement process. The devices internal SQLite database contains all persistent data about the device, for example configurations, projects and past measurements. The results from measurements are relatively large in size, so they are stored as files on the storage of the main computer. The files are read from the storage based on references to them in the database. Lastly, some system configuration is directly managed by the underlying Linux system. This includes settings such as keyboard configuration and network settings.

The FPGA communicator independently handles communicating with the FPGA. It does operations that do not require outside input, such as performing automatic maintenance, autonomously. In addition to these features, other application in the system, including the server, can request the communicator for data or actions. It then fetches the data from the FPGA and returns it to the requester or communicates the requested action to the FPGA. This way no FPGA logic needs to be included outside the communicator software.

Unlike communicating with the FPGA, the internal SQLite database is directly used by multiple applications. The server accesses data from it directly, as does the data manager that handles saving measurement results to the database and other destinations.

The different applications running on the main computer use Unix sockets to communicate with each other. This communication channel could in theory be used as the basis for the external WebSocket interface as well. That was not done though, as the internal communication contains information that is unnecessary for the end users, some of it is formatted in a unintuitive way and because this would have introduced the possibility to send messages from the outside to the devices internal communication channel. The external WebSocket interface is completely separate from the Unix sockets, but they are still connected indirectly by the server software. For example, when the a Unix socket message telling about a scan starting is received, a similar WebSocket message is sent.

## 3.3 System requirements

This chapter shortly describes some of the higher-level requirements for the DMS system, that affect the user interface. This mostly excludes requirements for the differential ion mobility spectrometer hardware itself.

The arguably most important feature for the DMS device is the ability to perform the differential ion mobility spectrometry measurements. For software, this means the ability to start new measurements and to stop ongoing measurements once started. The starting, interruption or finishing of measurements must be communicated to the user interface. As it is possible to know how many of the defined measurements points have been measured, the progress of the scan should also be communicated.

All measurements are done using measurement parameter presets. These are JSON (JavaScript Object Notation) configuration objects that include ranges of measurement areas or a list of individual points to measure. The user can create, delete and edit the configurations. Before starting a new measurement, the user must have selected a parameter preset to do the measurement with. This means there is always an active parameter preset that the measurements are done with. There is a small transfer delay for moving parameter presets to the FPGA, so having an active preset makes doing multiple measurements with the same preset faster.

Measurements also belong to projects. Projects are named collections of parameter presets and measurement results. They can, for example, be used to conveniently group together all measurements done as part of a single research project. Like parameter presets, projects are also activated. All done measurements are registered to the active project. Parameter presets can belong to multiple projects. The active parameter preset must belong to the active project.

Results of previous measurements must be searchable and viewable. The results must be searchable with a free-form text search that searches various metadata from the results. The results can also be limited to a certain time-of-measurement range. As the measurement result data depends on the parameter preset used to measure it, the correct preset must be easily findable.

The various system settings must be configurable. This includes settings such as internal clock, USB keyboard layout and Internet settings of the DMS device. The usage of the internal storage must be viewable. There should also be a way to back up the internal storage to an external USB storage device and empty the internal storage of the DMS device. The internal storage will be relatively small, so this is important to make the device usable after the storage space fills up. As there are open source libraries used in the software, their licenses must be available through the HTTP API and the user interface when required.

There are some perishable hardware modules inside the DMS device that must be changed from time to time. The state of these must be tracked and the user must be informed when they must be changed. The current status of the modules must be viewable by the user.

The system must have basic cloud support for saving the measurements to a provided cloud service. There must be support for logging in to the cloud service. New measurements must be automatically uploaded to the cloud if the user chooses so. There should be the ability to update the software of the DMS device through either the internet or a USB storage device. The USB storage would contain a update file downloaded using some other device. There should be a notification to the user when system updates are available, if the device has an internet connection.

# 4. DESIGNING THE USER INTERFACE

The first step in the implementation work was designing the user interface. This chapter describes the process from gathering requirements and ideas for the user interface to making different kinds of mockups. The methods and results of a small scale user experience validation test done using an interactive prototype of the user interface are also analysed.

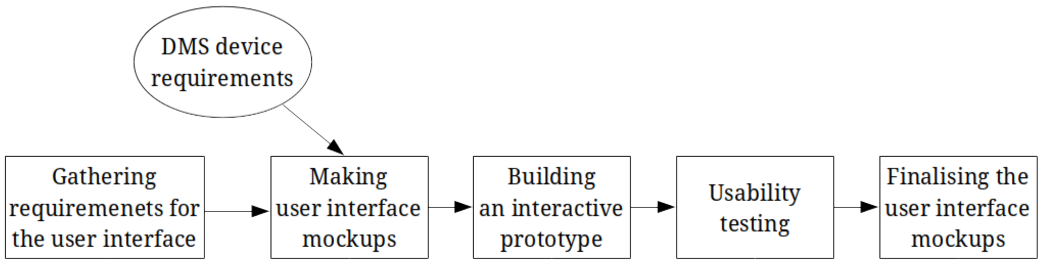

DMS device requirements
Gathering requiremenets for the user interface
Making user interface mockups
Building an interactive prototype
Usability testing
Finalising the user interface mockups

***Figure 4.1**. The process of designing the user interface*

Figure 4.1 shows the the process covered in this chapter. First, requirements for the user interface are gathered. Then, together with requirements of the whole system, user interface mockups are made. These mockups are used to make an interactive prototype which is used in a usability test. Possible usability problems discovered by the test are fixed to form the final design for the user interface.

## 4.1 Planning the user interface

Before the design work started, a few aspects of the user interface design were decided. To make the user interface behave similarly across multiple types of devices, and to make it more touch friendly for the touch screen of the DMS device, a uniform, non-native style for user interface elements was needed across usage environments. Normally different elements on web pages are styled differently on different web browsers and operating systems. This would make the look and layout of the user interface different on different devices. The way Chromium styles the user interface on the Raspbian desktop is not particularly touch friendly, with small user interface elements. Because of this, Google Material design was chosen to be used as a base for the user interface style. They have a ready made, official web element library making the style easy to implement. The elements also have integration libraries for many major web frameworks, which will be discussed more in the next chapter. Material design is specifically designed to work well on multiple screen sizes and input methods [10].

With material design, the general layout and navigation of the user interface was also adapted from how mobile phone applications are commonly built. A mockup of the basic

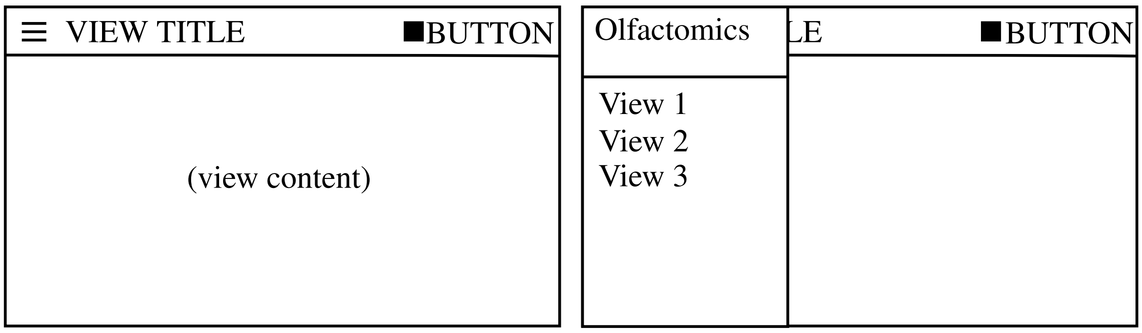


***Figure 4.2**. Coarse user interface layout without and with the navigation drawer open*

user interface layout is shown in figure 4.2. It has a navigation drawer that allows the user to navigate between different views. The drawer is hidden when not used to allow maximum screen space for other content on the relatively small built-in touch screen of the DMS device. In figure 4.2 the navigation drawer is seen open on the left side of the right side image. The top of the screen has a permanent toolbar that contains a button to open the navigation drawer. The button changes to a back button when in sub-views of main views of the application. This is needed especially on the built-in screen, as the user will not have access to the back button of the web browser. The toolbar also shows the title of the current view and can contain action buttons depending on the currently open view.

After some of the base features of the user interface were decided, it was easy to start gathering requirements and designing initial mockups for the user interface. Requirements and design ideas were gathered from two primary sources. The first one was naturally the documentation for the rest of the system. It has the user facing functionality requirements discussed in chapter 3.3 defined, but does not comment on the actual design or layout of the user interface. The other source was a designing meeting with other people working on designing the DMS system.

For gathering design ideas and initial mockups from others, an online meeting was held. Using a web based drawing tool that lets everyone draw real-time on a single canvas, ideas for the user interface were collected. First, a list of wanted features and their details was composed. This was just a text list with wanted views and their features. After that, everyone got a chance to make mockups of views they felt they had expertise or interest in. This way, the people designing other aspects of the DMS device got to express their ideas of what they wanted from the user interface. Results from the meeting were a list of requirements specifically for the user interface and low fidelity mockups about what people would like the user interface to end up looking like.

One consideration during the user interface design phase was the older DMS device mentioned in chapter 2.1. As mentioned in that chapter, that device also has a web based user interface. The user interface of the old DMS device is not well polished, but has some good base concepts. It has also most probably indirectly influenced the thoughts of many people working on the new DMS device about the user interface of that device.

For example, having a big visualisation of the latest scan result on the view used for scan management is something useful the older device did. In the mockups for the new user interface the concept was streamlined by removing a little used drop-down menu allowing to switch between visualisation types and by maximising screen real-estate for the visualisation by having a better designed layout.

## 4.2 Constructing the user interface mockup

Now the system requirements from the rest of the system and requirements, ideas and mockups of what other people involved with the DMS device project had in mind for the user interface were collected. Next step was to combine these into a single proposal for the user interface of the DMS device. The aim was to create high fidelity mockups that look relatively close to how the finished user interface would look like. This is to to convey how the end result of the development work would look like. Those mockups would then be combined into a interactive prototype used in the user experience validation test.

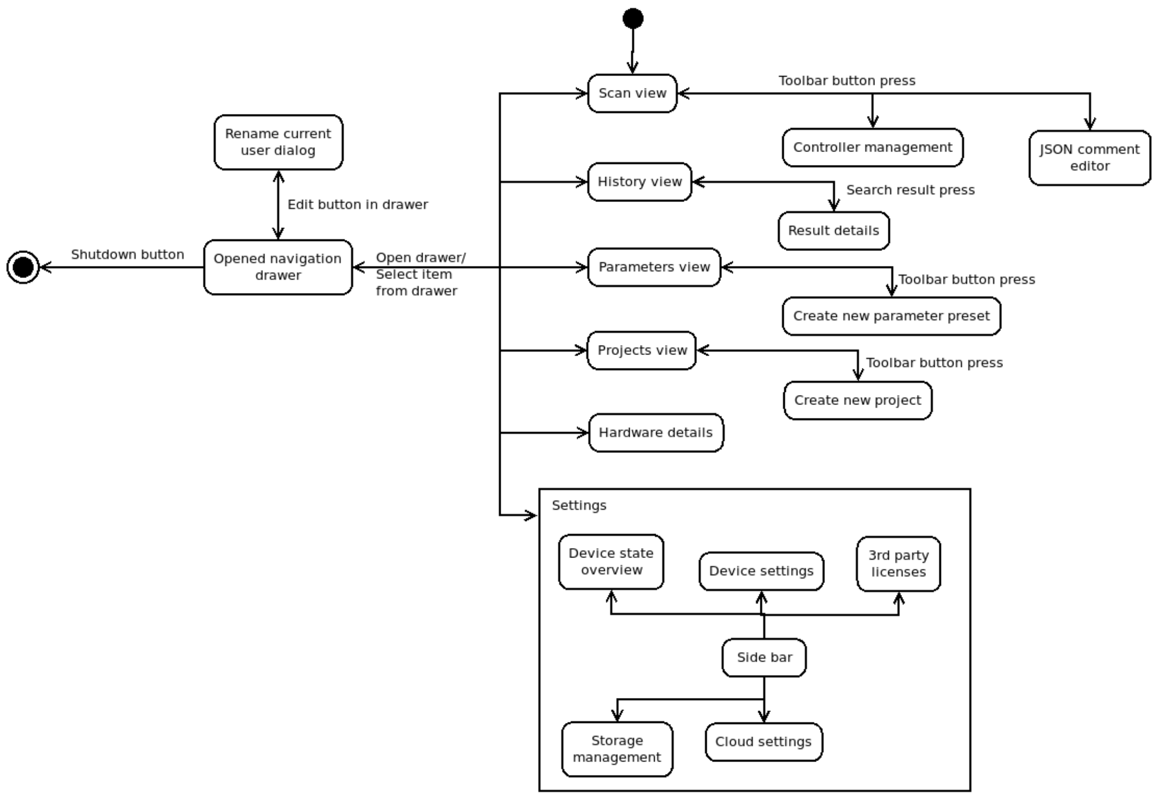
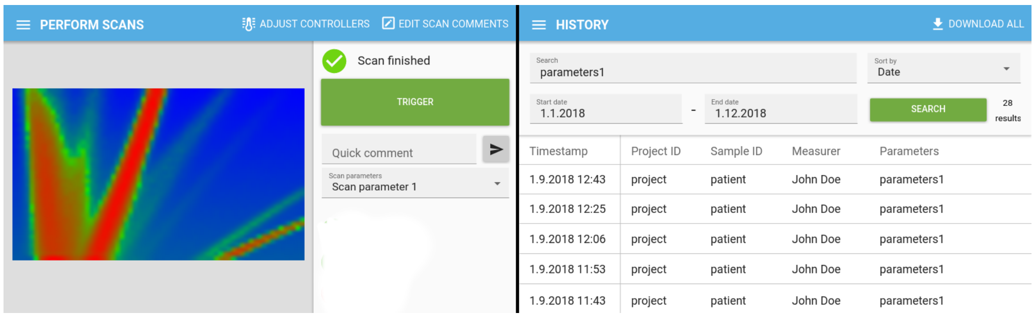


***Figure 4.3***. *UML diagram of navigation in the user interface*

Before starting to draw the high fidelity mockups, navigation inside the application had to be defined. It was only ideated so far, but a concrete plan was still missing. The basic model for navigation was defined earlier, when a navigation drawer was selected for moving between main views of the user interface. Based on the same decision, there is a toolbar at the top of the screen with buttons that depend on the active view. These are a good way to navigate to sub-views. For example, the parameter manager can have a "Create a new parameter preset" button in the toolbar. Based on this, a simple navigation diagram was drawn to document the structure of the user interface. The main features needed from the user interface are made into views accessible from the navigation drawer. Other known needed views are organised as sub-views of them. The navigation diagram is seen in figure 4.3.

With the user interfaces structure documented, designing the high fidelity mockups of each view could be started. The low fidelity mockups made during the group design meeting were used as a base for each view. While they would not be used as is, they were a good reference for what kind of user interface elements were needed. In addition to the low fidelity mockups, the feature requirements that were associated with the view being designed were referenced. Based on these sources, high fidelity mockups of all views of the user interface were formed.

To draw the high fidelity mockups an open source mockup tool called Pencil was used. Compared to using a normal vector image editing program, Pencil has the needed user interface elements as ready made collections [4]. For example, instead of having to draw a checked and not checked checkbox separately, and then copying and pasting them separately when needed, Pencil allows to use an element from a collection. To set the state of a checkbox, each copy of the element has settings that allow to customise its appearance. Pencil has an unofficial collection of Material user interface elements available. In addition to the component collections, Pencil has quality-of-life features, like aligning elements according to other elements on the mockup and easy management of multiple mockups.



Scan parameters: Scan parameter 1

Search: parameters1
Sort by: Date
Start date: 1.1.2018 - End date: 1.12.2018
28 results

| Timestamp | Project ID | Sample ID | Measurer | Parameters |
|---|---|---|---|---|
| 1.9.2018 12:43 | project | patient | John Doe | parameters1 |
| 1.9.2018 12:25 | project | patient | John Doe | parameters1 |
| 1.9.2018 12:06 | project | patient | John Doe | parameters1 |
| 1.9.2018 11:53 | project | patient | John Doe | parameters1 |
| 1.9.2018 11:43 | project | patient | John Doe | parameters1 |

***Figure 4.4.*** *High fidelity mockups made of the DMS user interface*

All of the mockups were designed in the 800x480 pixel resolution of the 7" touch screen for the Raspberry Pi. That way they would look correct on the built in screen. As computer monitors are usually physically larger and have bigger resolutions, the user interface could be expected to scale up well. This could not be easily tested before seeing how the implemented user interface scales on web browsers. All planned views of the user interface were mocked up. Even though not all were going to be used for the interactive prototype, they were helpful later on when implementing the user interface.

After the high fidelity mockups of the user interface were finished, they had to be made into an interactive prototype for the usability validation test. The prototype was made using the InVision online tool. Using it, it is possible to combine mockup images together by hyperlinks. First, all of the mockups exported from Pencil must be imported to InVision. Areas can then be painted on the pictures imported to the service, that when clicked change the visible image to an another one that was imported. This way it is possible to simulate navigating through the user interface. Changes to the user interface, like clicking a button changing some text, are simulated by the button click leading to a another image with the text changed. It is also possible to overlay images over other images. This way it is possible to implement features like the navigation drawer.

Not all features of the user interface were implemented in the prototype. For example, how scan parameters are managed was likely to change during the earlier phases of the development of the DMS device, so there was no sense in implementing it too deeply. Some other features were implemented more thoroughly, if the intention was to use them in the usability test. For example, starting and managing a scan was implemented as fully as possible, as there were multiple planned tasks for the test revolving around those features.

## 4.3 Usability verification test

The user interface has been built from ideas and requirements coming only from the designers of the DMS device. This poses a danger that the usability of the user interface is only designed for their needs. Users with other backgrounds might find it hard to use. The people who helped in defining the requirements of the user interface do not have experience in user interface design, and the final mockups were made by just the author. Because of this, a small usability verification test was performed to see how the user interface prototype fares in actual use.

### 4.3.1 Planning the test

The aim of the usability verification test is to receive general feedback on the user interface prototype and to find out possible usability errors in it. There is no measurable goal set for the test. The test consists of tasks where the test user has to use the user interface prototype, and some questions. The tasks designed for the test users attempt to make them use as many main parts of the user interface prototype as possible to get feedback on most of them. The questions ask for feedback about the tasks and there are a few specific questions aimed at possible usability problems identified during the initial design.

The test was relatively small, with 8 participants tested. This was decided to be enough to find at least the most serious usability errors. General information about the participants is collected in chapter 4.3.3. Participants were selected both from people who had experience with other DMS devices, and people who had no experience with them. As one of the criteria for the user interface is ease of use, it should be usable even if the user does not have previous experience with DMS devices or does not have the technical know-how of how differential ion mobility spectrometry works. On the other hand, the user interface must be usable by actual professionals, who are knowledgeable in the technology and possibly have experience with other DMS devices. Having participants from both user groups should ensure good usability for most users.

No dedicated testing location was used for the tests. Instead, they were done where it was be the most convenient for the test user. The tests were done using a standard laptop running the user interface prototype in a web browser. Because of this it was easy to conduct the tests in workplaces or homes for example.

The user tests were conducted alone by the author. The workload of conducting the usability test was manageable by just one person. There were basically three tasks to do during the tests: advancing the test by giving the test user new tasks, asking the user questions and observing the test and making notes based on the observations. A second person could have done the work of advancing the test, but that was a relatively small portion of work. Observing and note taking must be conducted by the same person in any case. If longer notes have to be written during the tests, it is possible to ask the test user to wait a short while between tasks or questions.

During testing, all observations, answers from the test user and other noteworthy things were written on paper notes by hand. The notes must later be rewritten digitally to produce clear and comparable tests results. The notes do not have to be too legible or complete, as long as the writer of the notes can understand them later when re-writing them. The purpose of the hand written notes is to make sure nothing is forgotten before that.

While usability tests were being conducted, the notes were re-written the same day as the test itself was conducted. This way the amount of information potentially lost due to forgetting something was minimised. The format used for the digital notes is the same as the script in appendix A. The questions are replaced with their answers from the test user.

Video recordings could have been used to make sure details of the tests were preserved. This was deemed to be both unnecessary and too complicated in this case. The test had relatively simple tasks with clear ways to solve them. The system tested was a prototype and not an actual application, and as such had quite simple interaction methods. This means following the tests was possible by just observing it during the test itself. A video recording would probably not have added new information. As the tests were conducted in different spaces depending on who was tested, the video recording hardware would have to have been set up separately for all testing sessions. Getting the positioning of the video recorder right to get usable footage of the test in all situations would have been a big addition to the setup time of each session.

A simple script was made for the usability verification test to make conducting it consistent. The full script is listed in appendix A. The test is structured so, that first the test subject is given instruction for the test. Then they are asked some background information for statistics about the usability test. After that the actual tasks with the user interface prototype start. Finally after the tasks the user is asked a few question about the tasks and the user interface.

As mentioned earlier, the tasks attempt to make the test user explore most of the features of the user interface. All of the tasks are listed in appendix A. The tasks of the usability test will also shortly described here.

Starting a scan and managing scan settings during it was tested thoroughly with 3 different tasks. As this is the main functionality of the device, it was important to focus on it. The first task of the three is simply starting a scan with a correct parameter preset. After that, the user must write a comment for the scan. Finally they must check a value of a hardware controller while the scan is ongoing. These tasks are done back-to-back, as the InVision prototype would be inconvenient to code to remember that the scan has been started if the user navigates to a different view.

In the comment writing task, the user has two options, as there are two ways to comment on scans. Both using the quick comment box and using the JSON based advanced editor was made possible and counted as a successful solution. The task about checking on a controller value involved the user having to press the controller management button in the toolbar of the scan view. There they had to scroll down the view and point out the PID values of the temperature controller. At the start of the next task the test user must also know how to exit this sub-view through the back-button in the toolbar, although they might have encountered a similar situation if they used the JSON comments editor before.

Interacting with other features, like searching scan history and managing settings, were given their own tasks. In the history task, the user had to search for results with the given search term. This is hopefully close to how the feature would often times be used in real situations. In the settings task, the user was tasked with logging in to the cloud service the device supports. This involves finding the settings view, finding the cloud settings section within it and then finally logging in. This task aims to test both navigating the settings view, as well as the usability of the cloud settings.

The first task of checking scan parameters acts almost as an introduction task, as it involves only opening the parameters view and viewing values in it. As mentioned, parameter management features were not completely defined at this point of the project. Making the task more detailed would not have made sense. The last two tasks test the usability of restricted mode. This is a special mode of the user interface that would allow the owner of the device to limit the features of the user interface for less experienced users. This feature will probably be left out of the final user interface. At the time of designing the usability test it was more likely to make it into the user interface.

The tasks deliberately make the user navigate around the user interface. The aim of this is to test navigation and structure. If all of the tasks done in the same view are given to the user in succession, the user might accidentally find the right place to perform the task, as they are already in the right view. Trying to perform the action from some other view might turn out to be unintuitive, but the usability problem was never found because the starting point of tasks was always optimal.

To make the test users have to navigate more, some tasks that are done in the same view, but do not necessarily have to follow each other, were mixed in with other, unrelated tasks. For example, after staring the tasks phase of the usability test at the scan management view, the first tasks take the user to settings and parameter management. Only after these tasks is the user instructed to start a new scan. At that point they must find their way back to the scan management view.

### 4.3.2 The structure of the test

Before the test starts, the test user is told some basic information about the test. This ensures that every test user starts from the same starting point regardless of their experience with usability tests or DMS devices. Otherwise the results from the test would not be comparable. Another aim is to avoid the user getting frustrated or feeling uneasy about the test or even wanting to interrupt the test. It is also a good practise to tell the user about the background of the test and describe what data is collected.

The function and operation of the device is described very briefly as the user might not know what a DMS device is. The user only needs to know some general level information to make sure they understand all the tasks they are given. The user must be told that the DMS device scans samples when manually started. The results from the scans are stored on the device and can be viewed later. Measurements are done using parameters which are editable and also saved on the device itself.

The test user is told that the prototype might contain flaws and that their potential inability to complete tasks is not their fault. This is important to note to avoid frustration in the test user. If there was some big usability problem, they might feel they are just missing something about using the user interface. For the same reason they are explicitly reminded that the user interface is the subject of the test, not the user themself.

Prototypes made with InVision have a few limitations. It is not possible to scroll views, at least if you still want to keep a menu bar at the top of the view. It is also not possible to input text. The user is notified of these limitations and told the workarounds. Scrolling will be simulated by tapping the top or bottom border of a view. Text input likewise is also done just by clicking the text field. If the user would have been clearly struggling with these limitations at some task, the organiser of the test would have reminded them about the workarounds during the test.

The user is also told about the usage of the test data and what data is collected. The purpose of the test is not to create an identifiable list of people, but to get feedback on the user interface prototype. No names or other information that could be used to identify the test users was collected. Age, profession or other information that was collected is not specific enough to recognise a person from. It is important to convey this to the user to not make them think they can be connected to the test or its results later.

An important aspect of user tests like this is understanding why the user does something and what thoughts lead to an action. Because of this, thinking out loud is encouraged during the test. The user is reminded about this at the start of the test and again during the test if they seem to forget. The user must also tell when they have finished a task themselves. This way they can not accidentally finish a task without them knowing. This could happen if the test organiser interrupts the test user immediately when a task is over. The user could also wait for the organisers comment to know when they have succeeded, which would mean that the user could randomly try actions until the organiser tells they have succeeded. The user not knowing what they have done or when they have succeeded in a task would be a usability error.

The task descriptions were given to the user on individual paper pieces. When the user felt they had finished a task, they were given a new one on a new paper piece. This way they could easily reference the exact task description any time they needed to. The users were also told to read the task descriptions out loud when they were given to them. This way the organiser could be sure they read the task right and were attempting to do the correct thing. If the user would have read the task wrong, they could have been corrected immediately.

After the introduction, basic information about the user was collected. Their age, educational background, profession and previous experience with DMS devices. None of the information is specific enough to identify the test user from, but they could be useful as a statistic of all the tests made when analysing the results later. Asking the users about their previous experience with DMS devices was important, because as previously stated, one requirement for the test was to get users with and without experience with DMS devices.

After the background questions, the test user was told that the test was about the begin. They were given the first task paper note. Upon completion, all of the tasks were gone through with new tasks following completed ones. After the final task, the user was asked some questions related to the tasks.

The questions asked included relatively standard questions, like "What was best and worst about the user interface?", "Was using it easy?" and "What would you change about the user interface?". In addition, a question about the test users preference for Material design as the style of the user interface was asked. The purpose of this was to confirm whether the selection of Material design was actually making the user interface easy to approach for new users. A question about the users familiarity with the JSON data format was also included. There are a few places where the user has an options to edit JSON data directly. This question will be used to define what kind of precautions those views might need to not confuse people not familiar with JSON. Finally, the user was asked for possible other comments that they might not have had a chance to tell yet.

### 4.3.3 Results

After conducting the tests and cleaning out the notes from each test, next the results of the usability verification tests were analysed. No individual test results will be handled in this thesis, but findings from all of them will be concluded. There will also be a short look into the demographics of the test users.

#### Test demographics

| | |
|---|---|
| Median age | 25.5 |
| Educational backgrounds | High-school diploma to masters degree |
| Professions | 5 University students<br>2 Researchers<br>1 Medical doctor |
| Experience with DMS devices | 2 people with multiple years of experience<br>2 people with under 1 year of experience<br>4 people with no experience |

***Table 4.1**. The demographics of the usability verification test*

Table 4.1 displays basic information abut the participants of the usability evaluation test. The tests were mostly done around Tampere University. Because of this, the median age is quite young and educational and professional backgrounds are quite uniform. Most people partaking in the test were students. From the point of view of this test, the actually important statistic is the previous experience the test users have with DMS devices. The distribution of this statistic is relatively uniform. As previously mentioned, the user interface is supposed to be easy to use for both people more experienced with the technology and other devices and people with no previous experience. With 4 people with some previous experience and 4 people without any, there should be good amount of data from both groups.

#### Task 1: Checking the parameters of the measurement

In the first task, the test user had to check the parameters of a measurement. There were a few bits of confusion in this task. The users started on the scan management view, which lets them select a parameter preset for scans and then start new scans. Some of the test users tried to use the parameter preset selection drop-down menu to view the information of the preset. There were also comments from the test users about labelling the drop-down menu for selecting the parameter preset just "Parameters" being confusing. Lastly, some users took a small while to find the navigation drawer at first. This was to be expected as this was the very first task. Nobody had this problem after getting acquainted with this form of navigation during the first task.

The ability to jump to parameter configuration was added to the scan management as a result of this task. This makes it easier to view and edit the current parameter preset. This was a feature many tests users were clearly expecting to exist. The label of the parameter preset drop-down menu was also changed to "Parameter presets" to avoid confusion on the purpose of the menu.

### Task 2: Logging in to the cloud service

In the second task, the user needed to log in to the cloud service the DMS device supports. In this task, many test users were confused by the local username that the user interface has in the navigation drawer. They tried to edit it first, but when they saw that it only allowed them to edit a username, they understood that it was not the feature they were looking for. Some test users did not immediately understand that the cloud account settings were located in the settings view accessible from the navigation drawer. They instead spent a while looking for the cloud setting in other views. All but one still found it by themselves after a bit of looking around. Once the test users looked into the settings view, everyone found the cloud settings easily. The cloud settings user interface was easy to use. A few of the test users commented that the switch in the cloud settings telling that new scan results are saved to the cloud is maybe not enough to clearly tell that the feature is turned on. They suggested adding a reminder of that to some other view as well.

The local username visible in the navigation drawer was fixed by making the label read "Local username". This will hopefully make it easier to differentiate from the login information of the cloud account. While some users did not immediately find the cloud settings from the settings view, no changes were done because of that. Lifting the cloud settings up in the navigation tree or otherwise making it more prominent would not make sense for such a small feature. Most users still found it after a bit of searching. A reminder of scan results being saved to the cloud could be added to somewhere else in the user interface. Adding new permanent indicators that are easily visible even on the built-in 7" screen while not making the user interface too crowded can be difficult. A notification message about the results being saved was added instead. It is more space efficient as the notification does not take space in the user interface all of the time.

### Task 3: Starting a scan

Next was the task of starting a new scan. This task has two parts, selecting the correct parameter preset and starting the scan. Some of the test users had problems with selecting the parameter preset. They went to the parameter view instead of scan management view, and tried to select the parameter preset there. After understanding that that was not possible, they found the scan management and understood to select the parameter preset there. Some test users also tried to go back to the parameter view after changing the current parameter preset, and check if the current parameter changed there. Due to the limitations with InVision, that did not work. This was explained to the users as it happened. After that, everybody managed to start the scan. Some of the test users told that they did not understand what the "Trigger" button, which starts the scan, did. They pressed it anyway as it seemed to big and important enough to be the start scan button.

The disconnect between parameter view and scan management seen in the third task was caused by InVision. The current preset is supposed to be changeable from that view and changes to the current preset should update to it. The problems caused by this do not require changes to the design. The label on the button that starts a scan will be changed to "Start scan" to make its purpose more clear. The visual design of the button was proven to be extremely effective though.

### Task 4: Commenting on a scan

The fourth task had the test users write a comment to the ongoing scan. Some test users used the quick commenting feature and some used the JSON comment editor. As the JSON editor in the prototype does not have the test user actually write anything, everybody who tried knew how to use it. Some users admitted to not knowing how to edit JSON though. One test user went back to using the quick comment editor after thinking the JSON editor looked too complex. After posting the comment, many test users told they would have liked some kind of notification that the comment was posted successfully.

The JSON editor clearly needs some kind of a disclaimer, so users who do not know how to use it will not get confused. A pop-up dialog will be implemented to be displayed when the user enter the JSON editor for the first time. It will warn them that the view requires them to know JSON scripting. Quick comments will get a non-obstructing notification message for when one has been posted. This way the user gets proper, clearly noticeable feedback for entering a comment.

### Task 5: Checking on controller values

Next task has the test users check hardware controller values. In this task there was an interesting discrepancy between users who had previous experience with DMS devices and users who had little to none. All users with no experience with DMS devices and one of the users with some experience had no problems with this task. They found everything needed for the task nearly instantly. Users with more previous experience on the other hand had surprising difficulties with finding the button to open the controller management view. The button is located in the toolbar in the scan management view. The more experienced users started looking around other views, not noticing the button in the view they were in after the previous task. They all did eventually find the button by themselves though. There could be other explanations for the discrepancy, but the test users previous experience would be the most obvious one. They could have learned different ways to work from the other devices. The task continued without problems for everyone after they had found the right button.

There are no obvious changes to be made based on the results of the fifth task. While the button for opening the controller management view could be relocated, it might mean it is more difficult to find for the users who had easy time finding it in the current location. The hardware controllers are expected to be adjusted while performing scans, so the buttons placement in the current location is quite logical. The controller management view itself did not cause any problems during the test and thus does not need any adjustment either.

### Task 6: Searching for old scan results

After the hardware controllers, the test users had to search for scan results with a search term. This task went well for all test users. Everybody understood to open the navigation drawer and found the history button from there. Some hesitated about the wording of the tasks description mentioning "scan results" and the navigation drawer item reading "History". This did not lead to anybody being confused though. From the history view everyone searched for results using the search term and completed the task.

No changes are needed because of this task. The confusion between "scan results" and "history" was likely due to task wording rather that unclear labelling in the user interface. Naming the view "history" seemed to be a good option anyway, as all test users understood to look for the scan results there. The user interface of the history view itself caused no confusion and seemed to be well designed. All users immediately found the large search bar the view has and using it seemed intuitive.

### Tasks 7 and 8: Enabling and disabling the usage restriction mode

The final two tasks are about the usage limiting mode. As previously mentioned, this feature will likely not make it to the final version of the user interface, but the results are examined here for the sake of completion. In the tasks, the test user has to find the usage restriction controls in the setting view and enable the usage restriction mode by giving an unlock password. Then they must unlock the restricted mode in the final task. Enabling the mode went well for everyone. Finding the restricted mode settings in the settings view seemed to be intuitive and most test users knew to enable the restricted mode with the right settings. One test user had problems understanding how to enable the restricted mode once they had checked the settings. Unlocking the device from the restricted mode went without troubles with all test users.

Enabling and disabling the planned restricted mode seemed to work quite well as is. Only one of the test users had problems with the process. The view could be improved by adding a small help text that explains the functionality of the usage restriction mode. The prototype has a short text explaining that the password entered while enabling the restricted mode is used to disable it later. The text could be expanded by explaining what the mode does when enabled and that the button next to the password field enables the mode after entering the password.

## Test end questions

A summary of results from the tasks can be seen in table 4.2. After the tasks were completed, the test user was asked some questions. The first question asked was what the test user though was the best aspect of the user interface. Many answers mentioned that the user interface was clearly laid out and intuitive to use. Some of the users mentioned that they liked that the user interface was similar to user interfaces they were already familiar with from elsewhere. One even directly mentioned that it reminded them of their mobile phones user interface. This, as previously mentioned, was the reason Material design was chosen for the user interface.

Next question asked what was the worst aspect of the user interface. Many of the answers mentioned the unclear labelling of elements such as the current parameter presets selection drop down menu. These all came up during the tasks and are mentioned in the analysis of their results. One test user mentioned the placement of the button opening the hardware controllers management view. Another user mentioned that back navigation could have been better in the user interface. They felt that pressing the back button even in the top level views, or the views accessible directly from the navigation drawer, should bring the user back to scan management.

Allowing back navigation between the different views accessible from the navigation drawer would potentially break the idea of navigation drawer based navigation. Each view from which the navigation drawer is accessible is as high on the navigation tree as possible. Other views accessible from the drawer are just alternative top level views in a way. Being able to back away from one of these views into another would break this setting. Many Material design based mobile applications using a navigation drawer or a navigation bar handle back navigation this way as well. If backing into a single top level view was to be implemented in the user interface however, the scan management view could work as the single top level view. If the user accesses the user interface using only its base URL, that view is opened by default already.

The third question was simply asking if the users found using the user interface easy. All test users agreed that they though it was easy to use. The fourth question asked for the users ideas about possible changes to the user interface. This question yielded mostly the same answers as question two. The visibility of scan results being automatically saved to cloud, which was discussed earlier with the results of the tasks, came up again.

The fifth question asked about the test users opinion on the style of the user interface, Material design. All users though it was familiar to use and looked clear. The sixth question asked if the test users knew how to edit JSON. Three of the answers were positive, the rest admitted that they did not know how to edit it. Even people familiar with DMS devices are not likely to posses the programming know-how for editing JSON. This strengthens the notion made earlier that features that require editing JSON directly must somehow be marked with a warning to avoid user frustration. Finally the users were asked if they have any other questions or notes. Nobody answered anything for this question.

## Summary of test results

| Usability problem | Solution |
|---|---|
| Could not access parameter preset details from drop-down menu in scan management | Add a button to access the details from scan management |
| "Parameters" label for parameter preset selection drop-down menu was confusing in scan management | Changed the label to "Parameter presets" |
| Local username was easy to confuse with cloud log-in | Clarified the local username field to be labelled "Local username" |
| It is not clear whether new scans results are being saved to the cloud | A notification was added when a new scan result has been saved to the cloud |
| "Trigger" button for starting a scan was unclear | The buttons label was changed to "Start scan" |
| Many users do not know how to edit JSON script for scan comments | A disclaimer for the advanced comment editing was added. Users can still use the quick commenting feature. |
| Feedback for posting a comment for a scan was wanted | A notification for posting a comment was added |
| Restricted modes functionality and controls can be confusing | A help text could be added to the restricted mode settings |

Table 4.2: Usability problems found during the tasks and fixes made for them

The usability verification test went well. It resulted in plenty of feedback on the design of the user interface that helps polishing it before implementation. The test also showed that there are no major usability issues with the user interface design. Most issues that were discovered were relatively minor. There were a few hard to understand labels, a few cases where system state or state changes were not communicated well enough and so forth. None of these issues are hard to fix or require any big changes to the design.

The usability verification test confirmed that the initial presumptions about the design of the user interface were correct. The choice of Material design and the navigation drawer based, mobile application like layout were easy to use. This was confirmed by both the users managing the tasks well, and by the questions, where all users confirmed they found the style of the user interface familiar and clear. These results were exactly the original intention of these design choices.

The found errors were fixed in the mockups before development started. This way, they represented what the resulting user interface implementation is supposed to look like as closely as possible. Referencing the list of errors during development would have been tiresome and it would have been easy to miss something. With all of the found errors fixed, the implementation could be started.

# 5. IMPLEMENTING THE USER INTERFACE

Now that the user interface has been designed, it is time to start implementing it. The first part of this chapter describes the comparison process of three different single page application libraries, Angular, React and Vue. One of them will be selected for the implementation of the user interface. After the library has been selected, the implementation of the user interface is shortly described. Finally, the selected single page application library is assessed based on the implementation work.

## 5.1 Selecting a single page application library

As it has been decided to develop the user interface as a single page application, a framework is needed to make development easier and to increase the maintainability of the source code. While it would naturally be possible to develop the application with no libraries from the ground up, that would be re-building something already done well by these libraries. The end result would most likely not be as good and a lot of time would be wasted. The basics of single page applications, and why the method was selected for this project, is discussed earlier in the thesis in chapter 2.2.

There are a few criteria that the comparison, and eventually the selection of the library, will be based on. One is that the library should be easy to start using. The author has experience with JavaScript and web development without libraries. If the library introduces a lot of new and complex concepts, it might not be fit for this project. The overall complexity of the library is also a consideration. The user interface is a relatively straightforward project. An extremely complex development environment might make the project more difficult to develop and maintain than necessary.

Another criteria is performance. The library will have to run on the built-in computer of the DMS device. If some of the libraries have considerably worse performance, it might not be suitable for the project. How robust the library is is an another consideration. None of the compared libraries are that old, and it is possible some could still have some reliability issues.

As the user interface is maintained and updated in the future during the life-cycle of the DMS device, the likeliness of the library to be maintained as well, is a factor. As the user interface must work on web browsers used by users of the DMS device, the library must be updated to be compatible with web browser updates.

### 5.1.1 Introduction of compared libraries

There are a lot of web application frameworks aiming to make building single page applications easier. It would be inconvenient to compare all of them, so an initial selection of a few libraries to compare must be made. For this reason the libraries Angular, React and Vue were selected. All three libraries are talked about and used a lot and are in active development at the time of writing this thesis. A short introduction of each will be given later.

A library being popular and talked about means that there are other developers working with it. Choosing a little used library would not only mean that it could be more likely to not have enough features, but there could also be less support for it. Less talked about libraries would also be more difficult to compare. Considering how fast web technologies are advancing and changing, even these three libraries have relatively little credible comparison material available. Choosing actively developed libraries means that they are more likely to be supported for the entire life-cycle of the DMS device. The user interface must work on the browsers of the users of the device, which are being constantly updated. If the library stopped being actively developed, the user interface might have to be rewritten for a different library later on to continue development. The choice of the three libraries is quite subjective, but by doing it, Angular, React and Vue can now be compared more thoroughly.

React is a JavaScript library that focuses only on user interface development. Developed by Facebook, the first release of React was in 2013. In React all code is divided into components, which are self-contained, reusable "building blocks". Components can be defined as JavaScript classes which can store state, or as functions which do not store their own state. Components can render other components and pass them data. This way the page can be built as a tree of components starting from a single root component. The speciality of React is its usage of JavaScript XML (JSX). The content shown to the user must be defined using JSX rather than HTML. The key difference between the two syntaxes is that JavaScript code can be embedded directly into JSX.

Vue is a slightly newer library, being publicly developed since 2014. Like React, Vue focuses purely on being a user interface library. Unlike React, Vue is not mainly being developed by a single company. Instead it is more of a community effort with multiple sponsors. Vue also makes use of components as the "building blocks" of Vue-based web applications. Components can render other components, have state and pass data to the components they render. The point where React and Vue differ is that Vue uses HTML templates for defining what is rendered instead of pure JavaScript or JSX like React. HTML templates are like regular HTML files, except Vue processes them before they are shown to the user to add dynamic content from JavaScript. This is done using a separate templating syntax that extends regular HTML syntax.

Both React and Vue are only user interface libraries. They focus only on features for building the user interface. Other features, such as application wide state management or transferring data between components, must be handled by other libraries. React and Vue also do not have support for routing or manipulating the URL of pages. Routing in single page applications is explained in chapter 2.2. For these features, both of the libraries need supporting libraries. Both libraries have official or well established options for these purposes. Vue has its own ecosystem of official libraries for routing and state management, Vue Router and Vuex. React has well established, while not official, libraries like React Router and React Redux.

Unlike React and Vue, Angular is not only a user interface library. Angular is a fully featured web application development framework, with built-in support for features like state management and routing inside the library. Angular is the newest of the libraries, having its first public release in 2016. The framework is developed by Google.

In Angular, the code is organised into modules. Modules are collections of components and services. It is possible to import functionality of modules into other modules. Components define views that are rendered to the user and inject data into them. Like in Vue, HTML templates are used to define the rendered user interface. Angular services define logic and data not bound to any specific component. Like React and Vue that have root components which act as a starting point for an application, Angular has a root module with a root component that works in a similar way.

One major difference between Angular and React or Vue is that Angular requires the usage of TypeScript. TypeScript as a language extends regular JavaScript and can be compiled back down to it. TypeScript adds static typing among other things to JavaScript at the cost of having to compile your code. TypeScript is discussed more in chapter 2.3.

Compiling might not be considered a big problem though, as a compiler is usually used when developing with React or Vue as well. To support browsers which are less compatible with new JavaScript features, it is common to use a compiler even with normal JavaScript code. Using compilers with regular JavaScript is discussed in more depth in chapter 2.3.

As Angular has most of the features one would need for single page application development built-in, the library is more opinionated than React and Vue [6]. Decisions, like how data should flow inside the application, have already been made for the developer, as the framework is built to support a certain kind of architecture. In React and Vue, how things should be done is more customisable with external libraries and own conventions of the developer.

As mentioned before, both React and Vue have their official, or at least well established, libraries for handling most things Angular does by default. Still, it is also easy to switch to other libraries if desired. This also means that React and Vue do not completely define how things like data flow should be done. Instead the programmer can come up with their own solutions. Most of the time, the easiest way is to pick a library and go with its recommendations, but for people experienced in React or Vue, doing even completely custom solutions would not be hard.

### 5.1.2 Comparing the libraries

To effectively compare the three libraries, two methods were chosen. Firstly, a small test project with all libraries was done to get a feel on how they worked in practise. For Angular and React, a tutorial project is provided as a part of their official documentation. With Vue, its features were tried without a set tutorial project. Secondly, some writings made on the libraries were compared. Relatively few sources on such relatively new technologies exist, but some good materials for helping in making an informed decision on the library can still be found.

There are multiple aspects that will be considered when comparing the libraries. Ease of development with the library is important. The project should not require too advanced features from the library. Unnecessary complex code, even if it would make the structure of large and complex application easier to manage, might not be the right choice for this project. That is not to say it should be hard to write manageable and reusable code. As the author has experience with developing HTML, CSS and JavaScript without using any libraries, the library should not have too much custom syntax to learn. The more it allows to use pure JavaScript, or TypeScript in case of Angular, the better. The maturity of the library and probability of future maintenance is also something to be considered. The user interface will most likely be updated in the future.

#### Comparing the libraries in practice

Reading the documentation of a library is important for understanding it, but to get a feel on how it would work in an actual project, coding something small as a test is helpful. The aim is not to necessarily make anything useful. Coding something gives insight into how application development using the library actually works. Deeper understanding would require committing into a larger project with the library. A small, at the longest few hours long session with each library should already show the basics though.

As mentioned, as part of their official documentation Angular and React provide a small tutorial that works well as a test project. With Angular the tutorial is building a simple hero management application. With React you can make a small tic-tac-toe game. Both of these tutorials seem to try and go over most of the basics of their respective libraries. They have state management, dividing the code into multiple components, passing data between these components and so on.

Vue does not have such a tutorial project to easily test the library with. For this reason, something to test the library with had to be invented. The essentials section of Vue documentation, fully gone through, was a good list of basic features of the library to test. The documentation did not include a tutorial project to develop with them though. The library was tested by making simple buttons, text fields and so on, and then adding functionality to them. This way it was possible to get similar experience with the library as with the Angular and React tutorials for their respective libraries.

To start development with any of the libraries, a development environment must be set up. This requires installing all development libraries needed, including the JavaScript or TypeScript compiler, the used user interface library itself, unit testing utilities etc. All these libraries can be acquired from the npm repository. npm is a repository of JavaScript libraries for use in web and Node.js environments. To download the libraries, Node.js and the npm package manager must be installed on the development operating system.

With React and Vue, one could potentially just install the user interface library itself, and nothing else. This would be enough to develop web applications using only the library. One would have to just create a regular HTML file, which loads the JavaScript file of the library. The library could then be utilised in the developers own JavaScript code. In this case things like browser compatibility would be completely up to the developer themself. Angular requires more setting up even in a minimum configuration, because it is a full web application framework, not just a user interface library. It also uses TypeScript, which requires using a compiler in any case.

Installing the libraries could all be done manually if the developer knows what they are doing. It would be hard for a newcomer such as the author of the thesis though, as there is potentially quite a lot to install. For this reason, all three libraries being tested provide their own development tools. They allow creating the necessary development environment with a single terminal command. Angular has "Angular CLI", React has "Create React App" and Vue has "Vue CLI". The Angular and React tools are command line only, while the Vue CLI provides a basic web based graphical user interface as well.

The limitation of using a development environment set up by a development tool is that it can be harder to customise. Using the defaults for everything can be extremely easy, as basic things like starting a test server or running unit tests are handled by pre-made scripts. Trying to change the defaults can be quite hard, depending on the development tool. The pre-made scripts might not work any more and the custom libraries might not work with how the default environment has been set up. Being slightly restricted might in many cases be more convenient than having to set-up and maintain the environment completely by yourself.

### Testing Angular in practice

Angular includes a tutorial about building simple fantasy hero management application as part of their documentation as an introduction to the library [9]. Before the tutorial could start, the angular development tool must be installed from npm. After that, the test project could be created with a single command. As stated before, Angular defines how it should be used quite strictly. This shows in the creation command, which does not have any customisation options when it comes to libraries or tools used. This is not necessarily a bad thing however, as sensible defaults make it easier for a new developer who might not be able to choose between different libraries.

The "building blocks" of Angular projects are modules, that can contain components and services. A recommended way to divide code is to have a dedicated module for every clearly separate feature. Modules can import and export features to each other. For example, in a web application with user accounts, the user management could be its own module and the rest of the application another. The user management module could export any information of the logged in user that is needed by the rest of the application. All user management related logic and user interface components could be contained within the user management module.

Modules could make dividing the code into sensible parts easy, and together with well planned interfaces between them could make the application into a cleanly connected set of well defined modules. On the other hand, modules can also make building simpler applications with Angular more tiresome, as the code must by default be divided into many different layers and containers.

Angular components contain a TypeScript source file, a HTML template and CSS styling. The TypeScript code defines the component, and names the HTML template and CSS files associated with it. An example of a component definition is seen in code listing 5.1. The code consists of a class that represents the component, and a decorator for the class that defines the metadata of the component. The class is used to store component state and implement logic. The code inside the class can then interact with the HTML template.

```
import { Component, Input } from '@angular/core';
import { ExampleService } from '../example.service';

@Component({
  selector: 'example',
  templateUrl: './example.component.html',
  styleUrls: ['./example.component.css']
})
export class ExampleComponent {
  @Input() inputVariable: number;
  stateVariable: string;

  constructor(private exampleService: ExampleService){}
}
```

***Program 5.1**. Example Angular component code*

At the simplest, data can be injected to the HTML template by naming the wanted variable from the class of the component using double curly brackets. The data of that variable is then loaded from the class. For example, to use the stateVariable string from code listing 5.1, you would have to put {{stateVariable}} inside the HTML template file. When the components is rendered, the curly brackets have been replaced with the contents of stateVariable. More complex operations can be done using Angular specific attributes on HTML tags. These are called the templating syntax. For example, in the Angular tutorial the templating syntax can be used to loop over lists and set listeners for user clicking on a button.

Using a templating syntax is a clear way to inject data into and get data out of a HTML file. It separates logic out of the HTML code by only adding the minimum necessary logic code to the HTML template. On the other hand, the separation also means that things must always be expressed using the Angular templating syntax first. Only then can logic be added to the TypeScript code. This disconnect can make coding harder, especially at first.

Normally, when including a CSS file in a HTML document, the file affects every element on the web page with the correct selectors. In Angular, CSS styling included in metadata of a component is specific to the component. Used selectors, like class names, are prefixed at compile time, so they only apply to the HTML elements of their component. For example, if the CSS class name "name1" is used inside components "component1" and "component2", the compiled application could have class names "component1_name1" and "component2_name1" instead. This way the CSS selectors can only affect HTML elements from one component.

Components can render other components like normal HTML elements by using the names of components as a element names. For example, to render the "example" component from a HTML template, <example/> would be used. To pass data between components, a component can define input variables they need defined to be rendered. A component rendering that component must then provide these variables. For example, to set a input variable called inputVariable in a rendered component to 1, the code to render that component would be <example inputVariable=1 />. This allows for two way data binding, where the components being passed data to, can edit that data and the changes apply to the parent component.

Using components, the user interface can be built by dividing the user interface into small pieces with their own functionalities. The smaller components can then be rendered using parent components that collect the smaller ones into views and layouts. The web application will be a tree of components with a single root component.

Data can be stored in the parent components and be given to their children using the @input decorator as seen in code 5.1. Angular is designed to not rely too much on this method of data sharing though. There can also be a need for unrelated components in the application to share data or events, in which case using input variables would not work anyway.

As Angular is a full web application framework, it has a built-in solution for data flow and management called services. Services are like components, but instead of defining something visible to the user, they can be used to process data. The data from services can be accessed from components by injecting the service to a component as in code listing 5.1. Services can be instanced directly from components or work as singletons depending on the inclusion style.

Services provide a way to divide data storage and processing away from data display. Forcing this can simplify the application code. For example, a component displaying a list of users could have a separate user service that fetches the list of users from a HTTP API, formats the list in a easy to display format and caches it. The component would only have to call a single function to fetch an already formatted list.

Developing data fetching and processing separately into a service can make that logic reusable and easy to test. On the other hand, services can also make simple data processing cumbersome. According to the documentation, most shared data and data processing should be given their own service and then be used in the components. This might be overdoing it in many simpler cases. It could make the application code overly verbose and complex even with simple operations.

Angular has built-in routing available. Routing can be defined in a separate, otherwise empty component that basically defines what URL will render which component. It is possible to link to a certain URL by using an Angular specific routerLink attribute on HTML elements. The routing in Angular seems easy to use and it is useful to have it built in to the library.

### Testing React in practice

Next we will look into React. Like Angular, React has an official tutorial, that aims to go over the basics of the library [5]. Before doing the tutorial, a development environment is needed. For React, the easiest way to set up one is to use the previously mentioned Create React App development tool. It automatically creates a React development environment with testing and development tools pre-installed. With React, the development and build tools can be customised easily, as React does not depend on any single toolchain like Angular. Create React App does not allow customising the development environment by default, but after it has been created, it is easy to swap around tools if wanted. For the purposes of this comparison, the easiest default setup is used.

Angular, as a full web application framework, contains multiple types of "building blocks" to build the application from. This includes modules, components and services. React is just a user interface library, and only has components as its one main "building block". React components are quite similar in scope as the ones in Angular. They define what HTML elements are rendered in a certain part of the user interface, and attach data and CSS to those elements. Unlike Angular, React has two main types of components.

A React component can be a JavaScript class that extends the base Component class, as seen in code 5.2. In that case the component has its own state that can be manipulated to update its rendered HTML. The content rendered by a component is defined in its render function. For example, by manipulating the stateVariable variable in code listing 5.2, the component would re-render as the data of that variable is used in the render function.

```
import React from "react";

export class ExampleComponent extends React.Component{
  constructor(){
    this.state = {stateVariable: 1}:
  }
  render() {
    return (
      <div>
        {this.props.inputVariable}<br/>
        {this.state.stateVariable}
      </div>
    );
  }
}
```

***Program 5.2**. Example React component code*

In addition to state, all React components receive a props object. It contains all data passed to a child component from a parent component that has rendered the child component. The contents of the props object can not be pre-defined like in Angular with its @Input decorator. The parent can pass down any data, all of which is added to the single props object. Unlike in Angular, props are an encouraged way to pass data between components.

Props in React are seemingly more chaotic than using input variables and services in Angular. The input a component needs must be well documented manually. On the other hand, the system is more flexible. It allows for all JavaScript objects to pass down from parent components to their children, including functions. This can make many connections between components simpler than in Angular.

The other major way to define components in React is to make a simple function that accepts props as its only parameter and returns the contents of the page like the render function seen in code listing 5.2. These are called function components, and their usage is encouraged in the React tutorial. The content of function components can only be affected by the props they receive and they do not have their own state. For example, a simple button could be a function component. It could take two props, the text displayed in the button and a function that is called when the button is pressed. There is no need to store any state in the button component, so a function component allows to implement it in only a few lines of code.

Function components seem like a easy way to make small, re-usable components without unnecessary features. Not having to code a whole component class every time could make development faster. Function components also seem easier to debug than normal components, as they always work the same way with the same props as input. Adding state to function components later if needed is not hard.

In Angular, the rendered content was defined by a separate HTML template file. React uses JSX to define the contents of a component directly inside the JavaScript code. JSX allows mixing JavaScript and HTML, meaning React mostly does not have a templating syntax to learn. For example, the text displayed in a button element could be a JavaScript variable instead of the text to display. The contents of the variable are then used directly as the text when rendering the element. In code listing 5.2, a prop variable from a parent component and a state variable are rendered as text as part of the component.

JSX can be useful as there is no need to learn a new syntax to use HTML with React. Including logic to the contents of a components is extremely easy. On the other hand, JSX causes user interface definition code to mix with logic code unlike in Angular. The lack of division could make the source code harder to read and maintain.

As React is just a user interface library, there are no built-in tools to control data like services in Angular. For the most part, a good way to process data in React seems to be to have state and data handling lifted up to a common root component. The child components can then be made function components and the state can flow down through props. This way a separation of data and user interface can be created in a more React-friendly way. For example, a user list components would have a parent component that fetches the user data from a HTTP API, formats it and stores it. The fetched user data is then passed to a child component, that the parent component renders. That child component displays the user list to the user. As the fetched user data is stored in the parent component, the child component does not need its own state and can be a function component.

CSS files in React are not scoped by default, unlike in Angular. While it is possible to include CSS files in components in React, they apply to all of the projects components. This could make application development become difficult fast, as one would have to keep track of all CSS selector names across the whole application. To scope styling, CSS could be included as part of JSX by adding the styling as JavaScript objects. This would not be very reusable and complex styling would make the JSX code harder to read.

There is an easy way to include scoped CSS files with Create React App. By default, projects created with it support CSS modules, with which CSS styling can be separated into files that are scoped when included in React components. When a CSS module is imported into a component, its selector names are prefixed by the compiler to apply only to that component. This works similarly to Angular. CSS modules make using CSS in React applications manageable.

Routing is not available in React by default. It can be achieved by using one of the multiple supporting libraries available for React. The most popular library seems to be React Router, which will be used for this comparison with the other libraries. It works quite similarly to routing in Angular. Component-URL pairs, or routes, can be defined in components as JSX elements. These routing components are rendered when the page URL matches the route defined in the component. React Router has a special Link JSX element to allow manipulating the current route, similarly to the routerLink attribute in Angular. A smart architecture is to have a single root component in the application dedicated to routing, like Angular has a dedicated routing component.

## Testing Vue in practice

Lastly we will look into Vue. It does not have an official tutorial. Instead the "Essentials" section of the documentation was gone through and the concepts in the section were tried in a free-form test project [12]. The first step was to install the development environment from npm. Like Angular and React, Vue has its own development tools for project creation. With Vue CLI, a Vue project can be created with a single command. It has a default recommended configuration for a project, but also supports customising the default set-up with additional or alternative libraries or configurations. For the purpose of this comparison, the defaults were used.

Like React, Vue at its core is just an user interface library. Unlike React, Vue also has official add-on libraries to make it into a full framework. Using them, Vue becomes more like Angular. Vue uses its own version of components as its "building blocks". Vue components define the data, or state, of the component as a function that returns a data object. The rendered content of the component is defined as an HTML template. This template can be just a JavaScript string of the HTML code. The template can also be loaded from an external file if wanted. The HTML template uses a similar system to Angular, where there is a templating syntax used to access and manipulate the data of the component that is defined in JavaScript. With this comes all of the pros and cons discussed with Angular.

As Vue is an extremely flexible library, there are a few ways to define components. The default way discussed as part of the essentials section of the documentation defines components using the component property of the main Vue object. For example, a component named *example* would be defined with code Vue.component("example", {}). Defining components this way has a major downside, as they contain no easy support for scoped CSS and the component names are in a global scope. This way of declaring components would be more suitable for smaller projects, for example embedding Vue managed content into an existing web page. Doing a full single page application using this method would be unnecessarily hard.

To make up for these shortcomings, Vue has a specific single-file component syntax, which allows the developer to define the JavaScript logic, HTML template and CSS for a component in a single file. Using this method, each component has their own .vue file. An example of such component definition can be seen in code listing 5.3.

Single-file components seem to make writing components quite simple. There is no need to handle global CSS or separate template files. Larger components could become cumbersome to develop though, as all the code for a single component exists in a single file. This means even more code than in React, where CSS styling can be stored in separate files. Single-file components also define an additional new syntax to learn instead of using existing CSS, HTML and JavaScript concepts.

```
<template>
<div>
  {{ inputVariable }}
</div>
</template>

<script>
export default {
  name: 'example',
  props: {
    inputVariable: String
  },
  data: function(){
    return{
      stateVariable: 1
    };
  }
}
</script>

<style scoped>
style{
  background: black;
}
</style>
```

***Program 5.3**. Example Vue component code*

Like in React, Vue components have props that can be passed down from parent components. If passed props change, the child component receiving them updates as well. To pass data up from a child component, functions should not be used as props like in React. In Vue there is a dedicated system for this. Vue has an event system, where components can send event that their parents can listen on and react to. Events are named using a free-form string when emitted by a component using the $emit method Vue has. The parent listening to the event must use the same name to receive the event. For example, a button component could send an event named "onPress" when it has been pressed. The parent component could then listen on "onPress" events from the button component to react to the button being pressed.

Both prop functions and events seem to be quite straightforward and do not differ much from each other. Especially compared to Angular, where services are in most cases the recommended way to achieve data flow to both directions. Events add some additional syntax compared to using functions as props though.

Vue does not have anything like Angular services, as it aims to be only a user interface library. This is similar to React. Like in React, one way to handle data could be to use a single parent component to fetch, process and store data for its child components. Vue also has official libraries that could be used to make data management using the library easier.

The core Vue library does not have routing capabilities like React. Unlike React however, it has an official routing library, Vue Router. It can be combined with the base Vue library making the result closer to Angular. Vue Router on a base level works quite similarly to the two previously discussed routing libraries. You define routes that change what component is rendered in a router component based on the page URL. The page URL can be manipulated by using Vue Router specific router-link HTML elements.

## Conclusion of practical comparison

After going through the basic documentation of all three libraries and doing some practical testing, the libraries all have their own strengths and weaknesses. There are also a lot of similarities between the three. In a lot of ways, any of the three libraries could be used. The choice comes down to differences in the design choices of the libraries.

Angular makes writing bad code difficult with its mandatory TypeScript usage and emphasis on dividing user interface definition into components and data manipulation into services. On the other hand, forcing to develop that hierarchic code from the beginning seems to make the projects grow quite large fast. Having to use a very specific, predetermined code structure makes developing with the library harder, even though the end results are more often good.

React is a lot more lightweight. With no predetermined project structure, it is possible to write more lightweight components when needed in the application. However, there is more to go wrong as the developer is responsible for designing a maintainable and readable architecture themselves. Badly designed code will eventually become hard to maintain using any library though.

The structure React applications seem to be designed to take is quite different to Angular. In Angular, the data should flow from Services to Components. In React data should flow from parent components to their children through props. The approach in React seems quite useful for cases where there is less data to transfer, while the Angular approach would scale better. Both libraries still have the ability to use the main data flow method of the other. Angular has some support for props, or input variables, and it is not hard to write service like classes in pure JavaScript or using external libraries with React. The prop system in React did seem a lot more versatile, with the ability to pass down any JavaScript elements, including functions, thanks to JSX. This can not be achieved as easily in Angular.

Vue is a bit closer to React in how it can be used. It is extremely flexible and supports using additional libraries well. It also does not require a specific code structure. Vue has its own official supporting libraries for features like routing, which React does not have. Vue has some Angular-like elements as well, like its templating syntax, making it almost a middle ground between the Angular and React. This shows in structuring Vue projects as well. By default, there are no Angular service analogies, instead data flows down the component tree like in React. But instead of allowing functions to be props, Vue uses events for child to parent communication. Compared with Vue, the more JavaScript based system in React seems more flexible. It does not rely on a more limited templating syntax and uses all JavaScript features in props.

Routing is only handled by Angular by default. In addition, Vue has routing available as an official supporting library. React needs a third party library. Only Angular included routing in their tutorial or introduction, but it still seems like a important topic to discuss. Routing was easy on all three libraries, even if additional supporting libraries were needed. Having to use an external library could affect maintaining the user interface in the future, but even with React, the routing libraries available were quite mature. Changing just the routing library should be quite easy if needed.

### Comparing literary material on the libraries

Now that all three libraries have been briefly tested, some of the few writings made on the three libraries and their advantages or disadvantages will be gone through. There were not a lot of credible direct comparisons of the libraries. There exists a lot of opinion pieces, but the purpose of this comparison is to form an original conclusion on the best library for this project. More unopinionated comparisons are more useful for this purpose.

A. Mlynarski and K. Nurzynska compare the performance of Angular and React, amongst other libraries, in their article "Comparative Analysis of JavaScript and Its Extensions for Web Application Optimization" [17]. Their comparison focuses on displaying large amounts of dynamic data that was constantly updated. Rendering and updating time and frame rate related statistics were measured to determine the performance of libraries in relation to each other. This kind of scenario is not too close to what is required from the user interface of the DMS device. There are some large data sets to display when displaying scan results and search results, but they are not updated often. They are also just a few of the features of the user interface. The comparison in the article should still work as a reference on the performance of Angular and React.

When looking only at Angular and React, Angular did slightly better overall. The article uses a point system developed for comparisons that attempts to take all measured performance characteristics of the libraries into consideration. The highest points any of the compared libraries received was 105. The difference between Angular and React in the point system was 13.5, which is not much. It is noted that React did better with initially displaying the data, but lacked with achieving good frame rates. As constantly updating data is not needed in the user interface of the DMS device, React could perform better in its usage scenarios. In the end the differences in performance will probably not be too noticeable, as the data processing needs should not be nearly as great as in the article and the differences even in the situation the article had were not big.

In the online article "React vs Angular vs Vue.js - What to choose in 2019? (updated)" published by company TechMagic, the three libraries are compared using pros and cons for each [23]. Some of the points of the article and the findings of this thesis will be compared. For Angular, its well defined structure for user interface and data handling and its support for TypeScript is mentioned as good aspects. The library is also mentioned to be quite complex and heavy to learn. This supports the conclusion reached during practical comparison. The separation of components and services seems useful and TypeScript can make the code more robust. This makes the library seem to have a lot to learn though, especially for someone coming from just JavaScript background. Even simple applications can also get complex fast.

According to the article, some of the pros of React are it being easy to learn, it relying on standard JavaScript features and it supporting functional programming concepts. The non-opinionated coding style of React is mentioned to potentially lead to worse code and the multiple ways to use CSS are said to be dividing React developers.

The fact that Reacts relies on standard JavaScript was mentioned earlier in this thesis as a big advantage of the library. The lax code style standards in React could cause less maintainable code, but a good structure is still required in any library. Lack of a standard way to use component scoped CSS files in React by default is not optimal, but once one has been chosen, should not pose great difficulties.

The article mentions Vue to have similarities to both Angular and React [23]. The adaptability of Vue seems to be the main pro the article has for the library. It fits both bigger single page applications and smaller projects. Vue being relatively new to the field and having a smaller market share could be a risk. It is mentioned that there are less resources for help with Vue.

Adaptability does not mean much for the user interface of the DMS device, as it will be a full single page application. Having less resources could be a big down side when learning and developing with Vue. Vue being less used now could also make its future less certain when thinking about the future development of the DMS device.

Another online article comparing the libraries is "Angular vs React vs Vue: Which Framework to Choose in 2019" [2]. The article summarises some aspects of the libraries not talked about in the previous one. The article compares version migrations between the libraries. All three libraries seem committed to relative stability, even between major versions. Angular promises at least one year of migration time. React and Vue both have migration tools available to automate the migration at least partially. The update schedule of the user interface of the DMS device will most likely not be high enough, especially compared to online web pages, to make moderate changes between major library versions a problem.

The article notes that Angular has a steep learning curve. The writer thinks React and Vue are easier to learn, Vue being the easiest. This comes with the usual caveat of it also being easier to write bad code with less forced structure.

In the articles conclusion, Angular is written to be good for larger teams and projects. React is better for smaller projects and teams. It also offers more flexibility. The conclusion states Vue to be the simplest of the three. Its newness to the market is noted however, and it would be justified to be be cautious of the library for now.

Finally we are going to look into the official comparison with the other libraries Vue documentation has [1]. The comparison has comprehensive material on how the Vue developers feel their library compares with its competition. While the text could be biased and it most definitely focuses on trying to sell the user into using Vue, it seems like a good source for comparison keeping these facts in mind.

The comparison writes that React has relatively similar performance to Vue. Vue advertises that it requires less manual optimisation. A third party benchmark linked in the text gives all three compared libraries the relatively same score [13]. The comparison writes about the lack of scoped CSS file support in React compared to Vue. Vue also scales down better, but that is not something that is desired for the DMS user interface. When comparing Vue with Angular, according to the comparison, Vue is much simpler to learn and more flexible. It does not require TypeScript usage or the usage of certain project structure.

### Choosing a library

Now that practical testing on the libraries has been done and articles about them have been reviewed, it is time to choose the library for the user interface of the DMS device. Based on the experience gained with the three libraries and what has been written about the them, React was chosen.

Some aspects of the three libraries were practically the same. They all seemingly have good performance and based on the kind of data needed for the user interface of the DMS device, the written sources support that. The performance needs of the user interface are not expected to be great enough that any of the libraries would cause problems. Even on the internal computer of the DMS device.

The way applications are built using the libraries also share similarities. All three libraries have the concept of components, which allow to build the user interface out of small "building blocks". The way components are connected, populated with data and rendered is quite different though. The way data flows through props in React seems to be the simplest. It lacks the need for using services, events or other library specific concepts. Using regular JavaScript syntax and concepts, such as callbacks, to transfer data and events between components, the library is easy to learn after already knowing JavaScript.

JSX adds some learning to React, but there is less new syntax to learn than with the templating syntax for Angular and Vue. In the end JSX is mostly a mix of JavaScript and HTML, and is fast to learn after already knowing both. Most things with JSX can be figured out based on the two. The separate templating syntax of Angular or Vue, while not extremely complex, would add more learning time to the actual development. Features can not necessarily be figured out using previous knowledge, and require finding relevant information from the library documentation.

Angular services are a good concept for fetching and operating on data. Separating complex logic out of the components is a good practise. While it lacks built-in support for a similar concept, separating logic and the rendered content is still possible in React. It is easy to add a separate library to make managing data easier. Unlike in Angular and Vue, many simple operations between components can be done easier using the versatile props in React. As this is part of the core functionality of the library, React is better for building simple components than trying to use supporting libraries to improve this aspect in the others libraries. As the needs of the user interface of the DMS device are expected to be modest, React should fit it the best.

The Highly structured coding style and usage of TypeScript in Angular are not needed in the DMS user interface project. As the user interface project is mainly developed by just one person, type safety is easier to keep track of. A too heavily structured architecture might make the development process slower and more cumbersome for one person, as there is no simultaneous work going on with the project that would benefit from the structure. React and Vue allow for much lighter structures.

Compared to Vue, Angular and React are backed and used by big corporations. This hopefully means continued development and support for the libraries in the future. React is also older than Angular, and more used than Vue, making React and its community possibly the most mature. In case of problems, finding relevant help online for React can be easier.

## 5.2 Final implementation

At this point the user interface has been well planned and documented, and the library used to build it selected. Only the implementation based on this work is left. The implementation was quite straightforward after good planning, but this section will go over some smaller, more detailed technical solutions.

The basis of the user interface implementation is the routing. For this, the previously discussed React Router library was used. Routing decides what will be shown to the user based on the URL the user has navigated to. A single root component is used to render separately a toolbar and a content area, both which contents are decided by the router. This way each main view of the application, accessible from the navigation drawer as shown in the user interface navigation diagram in figure 4.3, can be implemented in their separate child component. Additionally, the contents of the toolbar are defined in their own, content view independent components. This way, the text and buttons shown there are separated from the main content, allowing for greater flexibility. The contents of the toolbar are still changed by the router together with the main content component.

When in a child view of a view accessible from the navigation drawer, the navigation drawer opening button is changed into a back-button. Such child views are visible in figure 4.3. Alternatively, when using the user interface on their own devices, the user can also use the native back button of their internet browser. Both that and the back button rendered into the user interface do the same thing.

Some data from the DMS device is fetched from the HTTP API when the user interface is first loaded. This data is used in multiple parts of the user interface and as such it would be inconvenient to load it every time it is needed. Fetching it could also slow down the navigation of the user interface, as the user would have to wait for the data to load when switching views. The pre-fetched data has to be kept in a storage that is not affected by the currently rendered React components.

For this use case, the React Redux library is used. It allows injecting data from a globally defined data store into components by rendering the components using a special React Redux injection function. React Redux also automatically triggers React to re-render components if the data used from React Redux is changed. As the stored values can change on the DMS device after they are initially loaded, the WebSocket API of the DMS device sends change notifications on relevant data that tells the user interface to update its stored data. This update process is implemented to a separate, always active component from the rendered components, and possible state updates in the rendered components are triggered from React Redux.

To allow easier localisation of the user interface, all user-visible text is loaded from the React Intl library. It allows the usage of a localisation JSON file, the contents of which can be loaded by using a React Intl specific JSX element or by using the JavaScript API of the library. The used language can be changed any time, allowing the implementation of a language selection menu which is placed in the settings menu. Each language has their own translated JSON file. When the language is changed, the translations are updated in real time, without the need for a page reload.

For material design components, the React Material Web Components (RMWC) library was used. It is a React wrapper around the official Google developed Material Web Components library, which would work on any web page regardless of the used library. Similar libraries to RMWC exists for all major web application libraries, including Angular and Vue.

RMWC allows easy usage of Material design components as normal JSX elements. It also adds a theming support that can be used to support alternative themes for the user interface. This theming support is unique to RMWC. The theming support could be used to implement a high-contrast theme for accessibility or a dark coloured theme for user preference in the future. All colours used in the user interface are saved in a default theme to allow development of new themes in the future.

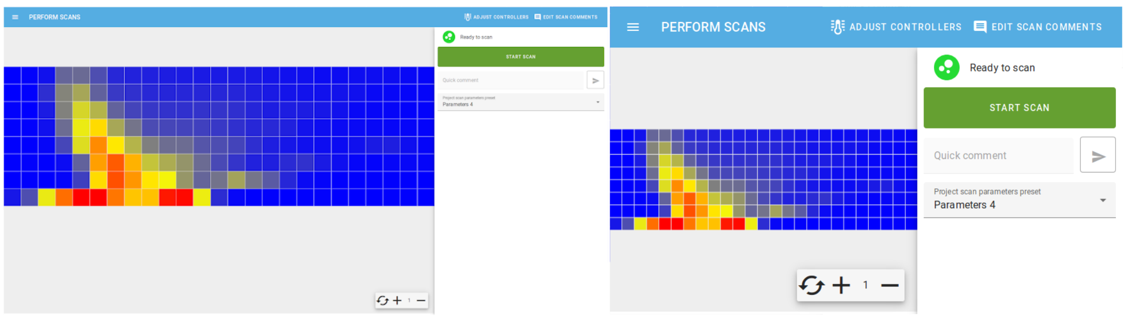


***Figure 5.1***. *The scan management view rendered on a desktop PC (left) and on the own screen of the DMS device (right)*

With these library selections, the user interface was easy to develop according to the user interface mockups made in chapter 4. Easily implementable material user interface components made the look and usability of elements like buttons and text fields easy to get to the same level as the mockups. This way, the user interface is easy to use even on the built-in touchscreen of the DMS device as planned. The implemented user interface ended up looking close to the mockups made earlier.

Pre-loading some of the data makes switching between views fast. This was one of the requirements for the user interface, as it is supposed to feel closer to a native one. Single page applications remove the need to load the page itself when switching views, but fetching dynamic data could slow it down.

The implementation works on both desktop computers and the DMS device as defined at the beginning of the thesis. The React implementation performs well on the limited hardware. As the mockups were planned for the small screen size from the beginning, the user interface fits on it as expected. While some elements can be unnecessary large as they stretch on a larger screen, the user interface is still usable and looks natural on desktop computers. A comparison of the user interface implementation on different screen sizes is seen in figure 5.1

## 5.3 Assessing the selected library

Based on the development work done, React fits the user interface of the DMS device well. It being less opinionated and having less mandatory structures has made developing the user interface as a one person relatively easy. Documenting the source code well has been sufficient to avoid problems with lacking type safety. The structure of the code has been manageable without Angular modules or services, as the structure of the user interface is relatively flat as seen on figure 4.3.

As predicted, connecting components together using props fits the project well. Most top-level views do not share much common data with the rest of the application. The common data that the views do share, is stored in React Redux. View-specific data is fetched and stored in parent components of the views. The parent component fetches all necessary data from the back-end through HTTP requests and then passes the data to a child component though props. This component is the root user interface component. The user interface component can then render any number of smaller user interface components that define smaller parts of the user interface. All necessary data flows down to them through their parent components. Making changes to data from child components is made easy, as one can pass methods of the data component as a prop functions. The child components can call this prop function with the necessary parameters to manipulate data.

For example, the root component of project management is a data component that fetches the list of current projects and the active project. It also stores which project is currently selected for viewing and the data of that project. The data component does not render any user interface elements, but instead it only renders the root user interface component. That component does not have any state, as all data is processed in its parent component. The user interface component can be a function component to make automatic testing easier and reduce application complexity. The root user interface component then renders smaller user interface components, such as the list of all projects and the view allowing the editing of the currently selected project.

This system of using data components is not hard to implement with some planning and documenting even with larger views. With small views it causes very little overhead and makes development fast. Choosing React partly for its flexibility seems like the right choice. Angular services could possibly have made the development of some of the larger views easier, but it would also have caused overhead with the smaller ones.

The performance of React is sufficient in all situations tested. The possibly most performance-heavy task the user interface has to do, visualising the largest possible scan result with over 700 000 data points, is possible even on the internal computer of the DMS device. All other tested tasks perform at least equally well.

React achieves smooth navigation across views as hoped. Together with the Material components library, the resulting user interface functions close to how a native user interface would be expected to run. The usability is relatively close to, for example, how one would expect a native application on Android or similar mobile platform to function.

Developing with React was easy even in situations, where the answer for a problem had to be searched. Maturity of the library was shortly assessed during the library comparison. React seemingly has a sizeable development community online and there is a lot of information available. With all problems encountered during the development of the user interface, an answer was easily found.

With previous knowledge of JavaScript, React was easy to use in the user interface project. There were no cases, where it was necessary to look through React documentation on how the syntax of the library works. JSX was in all cases self explanatory after learning the basics of it during the comparison phase of the thesis. This is not to say it was unnecessary to look though API definitions for JavaScript objects for example, but the JSX syntax used by the library was easy to learn.

# 6. SUMMARY

In conclusion, the DMS device now has a functional, performant and easy-to-use user interface that should allow as many people as possible to use the device without problems. This is important for easy approachability of the user interface, as most users of the device are not going to be experts in differential ion mobility spectrometry. It is also important for the user experience of the device in general, as the user interface is for many people the most interacted with part of the device.

As part of the design work, a usability verification test was performed. It turned out to be a very successful way to find usability issues with the user interface. With the 8 participants, the test discovered 8 usability problems with the user interface. There were problems with recognisability of features and receiving feedback from operations done in the background, to name some. While not necessarily big problems, they could have degraded the overall usability of the DMS device. With the problems in design fixed, the final user interface should not have any major usability problems.

Another major part of the thesis was the selection of a JavaScript library for the implementation of the user interface. From the start, three libraries were chosen for comparison, Angular, React and Vue. Emphasis was put on the libraries ease of development, performance, how much effort learning it would be and maintainability. The selection was done by doing some original testing with the libraries and by reading literary sources comparing them. Finally, React was selected because of its reliance on standard JavaScript features, because it has a more flexible component system than the others and because it seems the most mature as a library.

Lastly, the user interface was implemented based on the usability test and selected library. The resulting user interface, once integrated with the hardware of the DMS device, works well and fulfils its requirements. It can be run on both the touch screen of the device and external computers and it can control the hardware and software functionality of the device. The user interface also performs well even on the limited hardware of the internal computer of the DMS device. In addition to the implementation being successful, the choice of React was analysed based on experiences from the implementation. It was concluded to be as good of a fit as expected while initially choosing the library.

# REFERENCES

[1] Comparison with other frameworks — vue.js. Available (accessed on 31.5.2019): https://vuejs.org/v2/guide/comparison.html

[2] S. Daityari, Angular vs react vs vue: Which framework to choose in 2019, Apr, 2019. Available (accessed on 31.5.2019): https://www.codeinwp.com/blog/angular-vs-vue-vs-react/

[3] Details of the object model, Mar, 2019. Available (accessed on 16.4.2019): https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Details_of_the_Object_Model

[4] Evolus, Home - pencil project. Available (accessed on 12.3.2019): http://pencil.evolus.vn/

[5] Facebook, Tutorial: Intro to react – react. Available (accessed on 9.5.2019): https://reactjs.org/tutorial/tutorial.html

[6] M. Farwell, Yakov fain on angular, IEEE Software, Vol. 34, Iss. 6, 2017, pp. 109–112.

[7] R.T. Fielding, R.N. Taylor, Architectural styles and the design of network-based software architectures, Vol. 7, University of California, Irvine Irvine, USA, 2000.

[8] L. Gloaguen, Single-page application, pros and cons, Oct, 2018. Available (accessed on 19.2.2019): https://www.spiria.com/en/blog/web-applications/single-page-application-pros-and-cons/

[9] Google, Angular - tutorial: Tour of heroes. Available (accessed on 9.5.2019): https://angular.io/tutorial

[10] Google, Introduction - material design. Available (accessed on 21.2.2019): https://material.io/design/introduction/#principles

[11] Handling common javascript problems, Mar, 2019. Available (accessed on 17.6.2019): https://developer.mozilla.org/en-US/docs/Learn/Tools_and_testing/Cross_browser_testing/JavaScript

[12] Introduction — vue.js. Available (accessed on 15.5.2019): https://vuejs.org/v2/guide/

[13] S. Krause, Results for js web frameworks benchmark – round 8. Available (accessed on 31.5.2019): https://stefankrause.net/js-frameworks-benchmark8/table.html

[14] J. Kyrnin, Css vendor prefixes, Jan, 2019. Available (accessed on 24.4.2019): https://www.lifewire.com/css-vendor-prefixes-3466867

[15] C. McKnight, Single page applications – why they make sense, Apr, 2017. Available (accessed on 19.2.2019): http://www.digitalclaritygroup.com/single-page-application-make-sense/

[16] Microsoft, Typescript - javascript that scales. Available (accessed on 24.4.2019): https://www.typescriptlang.org/index.html

[17] A. Mlynarski, K. Nurzynska, Comparative analysis of javascript and its extensions for web application optimization, in: Kozielski, S., Mrozek, D., Kasprowski, P., Małysiak-Mrozek, B., Kostrzewa, D. (eds.), Beyond Databases, Architectures and Structures. Towards Efficient Solutions for Data Analysis and Knowledge Representation, 2017, Springer International Publishing, pp. 539–550.

[18] H. Oji, T. Matsumoto, Y.T. Cui, J.Y. Son, An automated haxpes measurement system with user-friendly gui for r4000-10 kev at bl46xu in spring-8, in: Journal of Physics: Conference Series, 2014, Vol. 502, IOP Publishing, p. 012005.

[19] Raspbian - raspberry pi documentation, Oct, 2017. Available (accessed on 14.1.2019): https://www.raspberrypi.org/documentation/raspbian/

[20] A. Rauschmayer, Speaking JavaScript: An In-Depth Guide for Programmers, "O'Reilly Media, Inc.", 2014.

[21] B.B. Schneider, E.G. Nazarov, F. Londry, P. Vouros, T.R. Covey, Differential mobility spectrometry/mass spectrometry history, theory, design optimization, simulations, and applications, Mass spectrometry reviews, Vol. 35, Iss. 6, 2016, pp. 687–737.

[22] P. Sherman, How single-page applications work, Apr, 2018. Available (accessed on 19.2.2019): https://blog.pshrmn.com/entry/how-single-page-applications-work/

[23] TechMagic, React vs angular vs vue.js - what to choose in 2019? (updated), Mar, 2018. Available (accessed on 27.5.2019): https://medium.com/@TechMagic/reactjs-vs-angular5-vs-vue-js-what-to-choose-in-2018-b91e028fa91d

[24] The MagPi Magazine, Pixel: the brand new desktop for the raspberry pi, 2017. Available (accessed on 14.1.2019): https://www.raspberrypi.org/magpi/introducing-pixel/

[25] Vendor prefix, Mar, 2019. Available (accessed on 23.4.2019): https://developer.mozilla.org/en-US/docs/Glossary/Vendor_Prefix

[26] D. Wanta, J. Kryszyn, J. Buraczyk, W.T. Smolik, Www interface for an electrical capacitance tomography system, in: 2018 International Interdisciplinary PhD Workshop (IIPhDW), 2018, pp. 344–347. ID: 1.

[27] A. Wilson, M. Baietto, Applications and advances in electronic-nose technologies, Sensors, Vol. 9, Iss. 7, 2009, pp. 5099–5148.

# APPENDIX A: USER EXPERIENCE VALIDATION TEST SCRIPT

## A.1 Instructions given to user

- The test tests only the user interface prototype. You are not being tested. If some exercise is hard to do, it is likely caused by design flaws, not you.
- The device is measurement device used to measure gases. The measurements are manually started and last for a limited time. Scans can be started and stopped. The results are saved to the device and can be viewed later. Scans are done using parameters that define their operations.
- There are a few limitations to the prototype. Text can be inputted by clicking the text field. Appropriate text appears in the field. Scroll views that require scrolling by clicking the top/bottom part of the view.
- Remember to think out loud.
- Say yourself when the task you are doing is ready in your opinion. I won't tell when a task is ready.
- The information collected from the tests can and will not be used to identify you.

## A.2 Background information

- Age
- Educational background
- Profession
- Previous experience with similar devices

## A.3 Tasks

1. At first you want to check the parameters of the measurement you are going to make.
    - Note: This should involve the user going to the parameters view
2. Now you want to log in to the Olfactomics cloud, you know the username and password. Make sure new measurements are saved to the cloud.
    - Note: Is finished when the user check that the "Save to cloud" selector is selected
3. Start a scan with parameters "parameters1".

4. Now that the scan is ongoing, you want to write a comment to the measurement that reads it is invalid.
    - Note: Is finished when the user check that the "Save to cloud" selector is selected
5. You also want to check PID values of the temperature controller.
6. Search for all scan results that have been made with the “partameters1” parameters.
7. A student is doing measurements with the device, put it in restricted mode so only scans can be made.
8. Now you want to unlock the device again. Post test questions

## A.4 Questions

1. What was the best aspect of the user interface?
2. What about worst?
3. Would you say using the system was easy?
4. What would you change about the system? Was there anything weird or annoying?
5. How did you think Material design (looking like mobile phones UI for example) worked in the user interface?
6. Would you know how to edit JSON script?
7. Do you have any other comments?