Tampere University

Eetu Ruponen

THE FRONT-END ARCHITECTURAL DESIGN AND IMPLEMENTATION OF A MODULARIZED WEB PORTAL

Faculty of Information Technology and Communication
Master of Science Thesis
May 2019

ABSTRACT

Eetu Ruponen: The front-end architectural design and implementation of a modularized web portal
Master of Science Thesis
Tampere University
Master's Degree Programme in Information Technology
May 2019

This thesis describes the front-end architectural design and implementation process of a modular web portal application, which is going to provide solutions for creating different kinds of tax and transfer pricing reports. All the reports are heavily legislated and currently require a lot of legal knowledge provided by tax lawyers, as well as a lot of information about the corporate finances. This is what the web portal application and the solutions it provides aim to ease.

To evaluate the success of the architectural design to be created, the requirements for the architecture are first defined. These requirements aim to take all of the different aspects of the architecture into account. The front-end architectural design process covers the design of the solution that enables the modularity of the web portal application, the design of the guidelines that can be applied in the development of a single page application, and the planning process of the design system that is utilized in the whole web portal application.

After the architectural design process is completed, the implementation of the web portal application is described. The implementation work done in this thesis included the creation and utilization of the design system planned as part of the architecture, the creation of base application that forms the core of the web portal application and the integration work needed to connect the different module application to the web portal application.

The implemented web portal application and its architecture are then evaluated based on the requirements set in the beginning of the thesis and future improvements are discussed. The resulting web portal application and its architecture, based on the microservice architecture, are still a work in progress, but so far they both have proven to be adequate solutions.

Keywords: front-end, modularized, architecture, web portal, microservice

TIIVISTELMÄ

Eetu Ruponen: Modulaarisen verkkoportaalin front-end:in arkkitehtuurin suunnittelu ja toteutus
Diplomityö
Tampereen yliopisto
Tietotekniikan koulutusohjelma
Toukokuu 2019

Tässä työssä kuvataan modulaarisen verkkoportaalisovelluksen front-end:in arkkitehtuurinen suunnittelu, sekä verkkoportaalisovelluksen toteutus. Luotavan verkkoportaalin tarkoituksena on helpottaa erinäisten verotukseen sekä siirtohinnoitteluun liittyvien raporttien luontia. Kyseiset raportit ovat lainsäädännöllisesti tarkkaan määriteltyjä ja niiden laatiminen vaatii kattavaa tietämystä sekä lakitieteistä, että yritysten taloudellisista tiedoista.

Ennen arkkitehtuurisen suunnitelman laatimista on määritettävä ehdot, jotka kattavat kaikki front-end:in arkkitehtuuriin liittyvät osa-alueet, jotta arkkitehtuurin onnistumista voidaan arvioida. Ensimmäisenä osana front-end:in arkkitehtuuria suunnitellaan ratkaisu, jota hyödyntäen verkkoportaalisovelluksen rakenteesta voidaan tehdä modulaarinen. Tämän jälkeen suunnitellaan ja määritellään ohjesäännöt, joita voidaan hyödyntää SPA-sovelluksen ohjelmistokehityksessä. Seuraavaksi kuvataan verkkoportaalisovelluksessa käytettävän muotoilujärjestelmän (engl. *design system*) suunnitteluprosessi.

Front-end:in arkkitehtuurin suunnittelun jälkeen kuvataan verkkoportaalisovelluksen kehittämisprosessia, jonka osana verkkoportaalisovellukselle suunniteltu muotoilujärjestelmä otettiin käyttöön. Tämän lisäksi luotiin sovellus, joka toimii verkkoportaalisovelluksen ytimenä. Kehittämisprosessin osana kuvataan myös kuinka verkkoportaalisovellukseen osaksi integroidaan uusia moduuleja.

Lopuksi luodun verkkoportaalisovelluksen ja sen mikropalveluarkkitehtuurin perustuvan arkkitehtuurin onnistumista arvioidaan aiemmin määritettyjän vaatimusten perusteella, sekä pohditaan mahdollisia kehityskohteita. Toistaiseksi vielä kehitysvaiheessa olevan verkkoportaalisovelluksen ja sen arkkitehtuurin ratkaisumalleissa ei ole havaittu puutteita.

Avainsanat: front-end, modularisoitu, arkkitehtuuri, verkkoportaali, mikropalvelu

PREFACE

The major part of this thesis was done during the spring of 2019, though parts of the work were done already in the end of 2018. The idea for the work came from the need of my current employer and their future plans with the products they wanted to create. The actual writing of the thesis began in February 2019, when an opening meeting was held with the examiners of this thesis. Though eventually, the major part of the writing was mainly done in the April and May of 2019 due to the sudden increase in motivation.

For providing this sudden increase in motivation, I would then like to thank the Finnish Government and its regulations concerning studying in the university. In addition, I would like to thank my instructors and examiners; Terhi Kilamo, who was one the best lecturers I had during my studies at the Tampere University of Technology, and Kim Jämiä, my former and current colleague, and my superior Tech Fellow. Thank you both for the good guidance throughout this process.

Tampereella, 25th May 2019

Eetu Ruponen

CONTENTS

LIST OF SYMBOLS AND ABBREVIATIONS

API	Application Programming Interface
BEM	Block Element Modifier
BFF	Back-end for front-end
CSS	Cascading Style Sheets
DOM	Domain Object Model
HTML	Hypertext Markup Language
OOCSS	Object Oriented Cascading Style Sheets
SASS	Syntactically Awesome Style Sheets
SMACSS	Scalable and Modular Architecture for CSS
SPA	Single Page Application

1 INTRODUCTION

In recent years the trend in software development has been that ever more complex and larger applications are developed with web technologies instead of building desktop applications. The use of web technologies allows the software to be used in a wide variety of devices with the minimal amount of device specific changes. For this reason the web technologies are chosen as the main form of technology in this thesis and in our company.

The context and motivation of this thesis is in the world of corporate tax reporting, which is traditionally managed by tax lawyers throughout the fiscal year of the corporation. Given that the reporting process requires a lot of working hours from high paid lawyers and management personnel, it makes sense to generalize and automatize the reporting process as far as possible. This will be resolved by the products, currently under development, and the web portal, the design and implementation of which is described in this thesis, and provide additional information about the corporations' structure.

The purpose of this thesis is to provide architectural guidelines for creating a modularized web portal front-end application and to describe an implementation process which follows guidelines determined. The need for modularized web portal comes from the complex and expansive nature of the substance matter the functions of the products are going to handle. The specifications of the applications originate from legal texts, which are hard to transform in to a form that computers and relational databases can understand. Additionally, the vast amount of individual pieces of information related to these specifications and reports needed to be created, it is better to divide them to multiple applications. This makes the information more understandable.

The questions that are answered in this thesis are:

- What kind of architectural design is best suited for developing a modular web portal application?
- What are the different aspects that the front-end architecture should cover?

Addition to answering the above mentioned questions, the reader should have a better understanding why the front-end architecture should not be neglected when creating new web applications.

The rest of the thesis is structured as follows. The first chapter of this thesis contains an introduction to the context and the goals of this work. Chapter 2 provides an insight

to the core concepts related to web applications and the context of this thesis. Chapter 3 describes the architectural design of the web portal's front-end. The requirements for the architecture are defined and based on the requirements the possible solutions are discussed. Chapter 4 describes the planning process of the design system, which will be used in the implementation of the web portal application. The issues related to the design system in existing applications are discussed and solutions presented. The implementation process of the web portal and the tools used during the implementation are described in chapter 5. The chapter also discusses the utilization process of the design system. Chapter 6 covers the evaluation of the requirements set for the architecture in the chapter 3. Additionally, the features that were either left out of the scope of this thesis or discovered during the writing and implementation process, are discussed as future improvements to the web portal. Chapter 7 contains the summary of this thesis work and discusses the writing process.

2 WEB APPLICATIONS

When starting to design and develop a modern web application, there are many different approaches that the developer team can take. Things that can affect the approaches being chosen often include the actual user environment and the possible existing infrastructure. An especially major fact to consider when thinking of the user environment is the use percentage of mobile devices in web browsing, which has increased steadily year after year [17]. The technologies which are used in the development process of a web application can have profound affects on the end result, as some technologies make it easier to create applications that are mobile friendly or focus more on the performance.

2.1 Web portal

A web portal is an application which combines information from multiple sources and provides a single access point to that information for the users [42]. These kinds of applications are often used in larger organizations and communities where sharing of information is the key in making the organizational operations run smoothly [21]. Applications which can be considered as examples of a web portal application are organizational intranet web pages and governmental web services, like online taxation service.

A web portal application can be a lot more than just a collection of static information or forms gathered from multiple sources. Functional web portals can allow the users to modify the content to suit their specific needs, they can allow users to manage their own information and they can allow users to use services provided by the underlying source systems of the portal. Though web portals collect information and functions from multiple sources, the user interface should still be consistent and easy to use, if the web portal design is done well [10].

In our use case the portal and sharing of information helps the organizations and their different divisions to understand each other, because often the case has been that the higher level management personnel and the financial personnel do not know what entities are referred in the other division's reports. There can be entities that are exactly same for both divisions but still the name of the entity is different, or more often there are entities which do not have any similarities with any of the other division's entities. This kind of informational gap in turn causes problems when it is time to release any kind of financial reports, as it requires manual labour to connect the figures to the right reporting unit.

2.2 Single-page application and client-side rendering

Single-page application, or SPA in short, is a web application which only loads a single HTML file to the browser and uses client-side rendering to change the information shown for the user [24]. Because only a single HTML file is loaded to the browser, the page does not need to refresh when the user interacts with the page. This gives the user a smoother user experience.

To be able to change the content of the page and to provide additional information for the user, the DOM (Document Object Model) has to be dynamically updated using JavaScript and HTTP requests to get information from the server [20]. There is a large number of frameworks and libraries – for example React, Vue and Angular – which are designed to create a layer of abstraction for the DOM manipulation process [14][27][40][1]. Although the example frameworks are not compatible with each other, the basic principles behind all of them are the same.

The single HTML file loaded by the browser provides a mounting point for the JavaScript application, which is loaded in addition with the HTML file. The mounting point is basically a single HTML element, most often a -element. When the mounting point is provided for the JavaScript application, it then knows where to start generating HTML content [18].

All frameworks developed for creating single-page applications, have application life-cycle methods with standardized names which give the developers the ability to define what will be changed in the DOM [16][34][22]. At its most basic, there is a method which defines what happens when the application component is going to be displayed for the user and another method determining what happens when the application component is going to be removed from the view.

Another common idea between the frameworks is the component based thinking of the application [5][36][2]. The simplified idea is that every view is a component with its own properties and functions. This idea can be further extended and the end result is an application in which every single visible element is a component of its own.

The aforementioned frameworks and single page applications in general take advantage of the idea of client-side rendering [19][46]. In the past, with multi page applications, it was common that all of the content was generated and rendered into HTML markup in the server side. This content then was again generated and loaded from the server each time the user interacted with the HTML page. The opposite of this is the concept of client side rendering, which means that basically all of the HTML content is created directly in the client. The HTML content is created by the JavaScript initially loaded to the browser. The previously mentioned Domain Object Model allows editing of the existing HTML content or creation of new content, as it represents the whole HTML content of the page as an object tree [20]. Even though single page applications commonly use client-side rendering, it is also possible to use server-side rendering for single page application. Server-side

rendering has its own advantages over client-side rendering as the HTML content can be fully rendered before sending it to the client-side and sending of JavaScript is not required. For instance, Vue provides support for server-side rendering with a separate *vue-server-renderer* -library [41].

2.3 Design system

The purpose of a design system is to provide guidelines and ready-made components for the developers so they can work efficiently and the end result is coherent throughout the application under development or through out all variety of applications [13]. The design system defines how things are implemented and how things are displayed for the end user. This includes implementation guidelines for writing HTML markup and creating styles with either pure CSS (Cascading Style Sheets) or with some extension language, like SASS (Syntactically Awesome Style Sheets) [31].

The guidelines tell the developer how to place the elements inside a view and what kind of styles those elements should have. The styles can include definitions for fonts, colors, sizes, animations, spacing and all other possible styling options. The main thing is that the views follow a pattern and the look of the views is coherent, and – more essentially – the same kind of elements look the same in all views of the application. For example if there is a delete button, it is always positioned similarly related to its container, it always has the same icon and text, and it is always the same colour. This coherency does not just help the developer and make his or her work more efficient, it also helps the end user to be more efficient. This is because the user can learn where elements are positioned in the view and that way can find the desired functions more effectively [13][38].

As the design system defines patterns related to how the elements should be placed in views and what they should look like, it is natural to create general components as a part of the design system [13]. These components provide a layer of abstraction for the developers, so there is no need to constantly rewrite similar HTML element structures over and over again. Instead they can use a general component which can be parameterized to enable customization of the content. Often different kinds of user interactive elements, like buttons and input fields, are a good place to utilize components.

Defining the style sheets of the design system is a crucial part of the whole design system, as those ultimately define the look of the views, components and single elements of the web application. There are multiple different practices of how to define style sheets in a manner that is modular, extendable and coherent. Examples of these kind of practices are OOCSS (Object Oriented CSS), SMACSS (Scalable and Modular Architecture for CSS) and BEM (Block Element Modifier).

In OOCSS the main principles of creating style sheets are separation of structural and visual properties, and separation of content from containers [3]. Structural properties include e.g. height, width and margins, and visual properties include e.g. background-

color, font and borders. Separation of containers and content means that the containers should not be content dependable. For example, it should not matter if a container has an image or text inside it, the container still should behave the same. As the name Object Oriented CSS suggests the DOM elements and their styles are seen as objects. To be more precise, an object here is any repetitive piece of HTML markup. For example, a navigation menu and its items are objects. Because of the idea of dividing different properties to their own classes, the resulting HTML can become bloated with all the different CSS class definitions. With smaller projects the benefits of choosing OOCSS can be also quite minuscule, because the reduced size of the project also reduces the amount of actually shareable CSS properties and classes.

In SMACSS the main principle of creating style sheets is categorizing styles to base, layout, module, state and theme styles [33]. The base styles mean the styles which are created to override the core HTML styles like the styles of a *body* or *html* -tags. Layout styles define how the available space in the view is shared. For example, layout styles define whether all of the content in a single column or in two columns side by side. The module styles define the shared component styles. For example, module styles would include what the items in the navigation menu should look like. The module styles can be compared to the object thinking of styles in OOCSS. The styles categorized under the state part of SMACSS principle include styles that describe how element should be displayed in a certain situation. For example, how an element should look like when it is selected or deselected. The state styles can also be used to define how an element is displayed inside different views. Finally the theme styles contain the styles used to describe how an element is displayed in a view, just like the state styles, but the theme styles can be used to differentiate the styles, for example for client A and B or they can define dark and light themes.

In BEM the main principle of creating style sheets is strict naming convention based on division of styles into blocks, elements and modifiers [11]. The block in BEM is a piece of HTML markup which can be thought to be significant on its own and does not depend on other blocks or elements. For example, a navigation menu can be considered as a block. The element in BEM is an item which does not have a meaning on its own and it is related to a block. For example, items in a navigation menu are elements, because they do not have a meaning without the parenting navigation menu. The modifier in BEM is used to define the state of the element or block, just like the state styles in SMACSS. The naming convention of BEM consists of three parts. The block classes are named descriptively and the class name can contain lower case characters and dashes, for example *.menu*. The element class names consist of the class name of the block, two underscores and the name of the element, for example *.menu__item*. The modifier class name is prefixed with the related block and possible element class names and the actual modifier part is separated with two dashes. For example to represent an active navigation menu item a class *.menu__item--active* can be added for the selected menu item. The modifier can also be added directly to a block, for example *.menu--collapsed* which represents the menu style when the underlying menu items are not visible to the user.

All three principles mentioned can be used on their own or combined with other principles, because all three complement the rules of creating style sheets in a modular and expansible manner. For example, using BEM with either OOCSS or SMACSS and using its naming conventions, the amount of class definitions per HTML element can be reduced.

2.4 Microservices

Microservices are a product of an architectural design ideology where instead of large monolithic applications with a lot of different functionalities, it is more desirable to have multiple smaller independent applications with mainly a single functionality or a single purpose [15][25]. These smaller applications then can form a larger service together, which is actually used by the users. Naturally microservice applications can be easily shared between services, so instead of creating separate user management for every service, a single user management microservice application can be used in all services.

Scaling down the size of a single application brings along many advantages when it comes to developing and maintaining the application.A smaller application means smaller code base which in turn means there are fewer moving parts and places where things can go wrong. The smaller code base can be developed and maintained by a smaller team which is often also a more efficient way to do things. The division of microservices developed by an organization can then reflect directly the organization's structure as one team can own the microservice application through its whole life-cycle from the design and development, until the end of life [25].

The independent nature of microservice applications means they can be deployed to a single or multiple servers in various configurations. This means also that the applications can be scaled easily based on the demand. For example, if a single application handles all large data request from all other applications it can be scaled up, while all other applications run only on a single instance.

Updating and re-deploying microservice applications is a significantly simpler process than deploying a monolithic application. Every independent part can be updated and redeployed separately as long as the interfaces do not change between applications [23]. This means the service can still be live while parts of the service are under maintenance.

When using microservice architecture in creation of larger services, it is clear that there will be a need to make integrations between the microservices. And probably a lot of it. Because of this, the ease of integration should be emphasized when developing microservices [25]. Creating organization-wide guidelines for integration and following them is a key factor in making the integration an easier process. Also when integrating the services, the integration pattern should be considered as it affects how loosely the services are coupled together. For example, if there is a microservice application, which manages creation of users and another microservice, which is responsible of sending emails. And

then a third microservice, which is responsible of managing user groups. Now, if a user is created, a registration email should be sent and the created user needs to be added to a default user group. The communication between the user creation service and the other two can be implemented either by the user creation service sending requests to the other two services, creating a tight coupling between the services. Or the user creation service can provide a possibility for observing events it dispatches and the other services can listen to those events and act accordingly. With this observer pattern the services are loosely coupled and any other service needing the information about a user being created can register themselves to the observer.

Although microservices provide many positive features for developers, it is not a flawless design principle [25][23][28]. When used correctly the idea behind microservices is brilliant and simple. When used incorrectly, simplicity is far from the truth. A common problem which arises when using microservices is that the dependencies between microservice applications become overly complex. Applications emerge which depend on every other application or a single application is required by every other application. At this point the whole idea behind microservices disappears. Developing easily maintainable microservice applications is not an easy task and it requires a lot of careful planning even though the basic idea is simple. The increased complexity of dependencies is not the only issue with microservices. For instance, as naturally one might expect, the API requests between the microservice applications take a longer time to finish than they would if made only internally in a single application [23][25]. Another issue, also related to making requests, is that creation of complex requests increases in difficulty [28]. This is because, the data requested might require composing data from multiple sources. With microservices, this might mean that every data source is in its own database and thus requires making requests to several microservices. With a traditional monolithic application, the data can be fetched with a single request, because the data sources can be combined together inside the application.

During, and after, any software development project, it is preferable to implement and conduct testing for the developed software [25][28]. Conducting testing for a system build with microservices has its pros and cons when compared against testing of a system comprised only from a single application. The pro part what comes with microservices is that the microservices are independent and thus can be tested independently. Also, the size of a single microservice application is relatively small and there are only a limited number of features, which are needed to be tested. The con part then comes from the fact that the microservices are often linked together. So to truly test if the system is working, end-to-end testing is needed. The difficulty of testing then increases as the dependencies between the microservice applications increase.

3 WEB PORTAL'S ARCHITECTURE

Before single-page applications and JavaScript driven client-side rendering of web pages, the architecture of the front-end was most of the time neglected or it was completely dictated by the architecture of the back-end application [13]. But single-page applications have now been here for a while, and still most of the time front-end architecture is neglected, even though the amount of code in a single page application is often significant and the complexity of the code can be greater than in the code of the server application. This results in monolithic applications which are hard to maintain. This is something that needs to be avoided in the design and implementation process of the web portal application described in this thesis. In this chapter the requirements for the web portal's architecture are defined and the chosen architectural solutions are presented. In addit ion to the requirements, one major factor that has an impact on the chosen architectural solutions is the fact that there was already a lot of existing work when the architectural design for the portal began.

3.1 Requirements for the front-end architecture

As briefly stated in the chapter 1, the substantial context of the web portal is in the world of corporate taxation and transfer pricing reporting. Less surprisingly, all of the reporting is strictly legislated and requires a broad substantial knowledge from the personnel responsible for the submission of these reports. In addition to pure knowledge the submission of the reports requires a lot of work hours. Just searching for all of the legal entities, or better known as companies, which are actually part of the corporation can take a substantial amount time. Not to mention, going through all of the financial figures of the corporation. The information required to submit the reports is not just limited to financial figures of the corporations, but also numerous other bits of data are required. This data includes information such as country of the legal entity, currencies used by the legal entities, ownership structure of the legal entities and information about the trading of goods the legal entities have made during the fiscal year. All in all the amount of information required for all of the reports is immense.

Then again all the reports do not require all of the same information and all corporations do not need to report all of the same reports as others, so it is natural to divide the information gathering to multiple instances. This fact dictates the first requirement, which is that the resulting web portal application should be modular in a way that the customers

could only acquire the parts that are necessary for them.

Other requirement for the web portal application, also related to the first requirement, is that the architectural structure should accommodate expanding the application in the future. The fulfillment of this requirement should come hand in hand with the first requirement, but the ease of expansion will require additional consideration when creating the architecture for the web portal. Adding new features to existing systems often can cause conflicts with the existing logic and new features are more likely to still contain bugs that could break the system. Keeping the different parts of the web portal application as independent as possible then can be a major factor, as it can ease the maintaining of the application later on. An additional long-term requirement for the web portal application's expandability is that it should be possible for third parties to create and add content to be a part of the web portal.

As the technological domain of web development and JavaScript frameworks is rapidly changing and evolving, it is preferred that the architecture of the web portal should be as technologically independent as possible. The technological independence would enable the possibility for changing the development stack in the future when creating new features to the web portal. It could also allow the development team to gradually update the existing feature implementations with new technologies, which would reduce technological debt. The technologically independent architecture would also allow possible third party content creators to use their own choice of development technologies.

Another requirement for the web portal's architecture is that the user interface and user experience should stay coherent through out the web portal application. This should be the fundamental idea when designing all of the features for the web portal. As a part of the coherent user experience requirement, the resulting web portal application should feel like it is a desktop application without any unnecessary loading screens. Essentially this means that the web portal should be a single page application to avoid any browser refreshes between views, as mentioned in section 2.2. The coherency requirement of the user interface as such is quite vague and fulfilling it can be a matter of opinion if not defined more precisely. In the scope of the web portal application the coherency requirement is considered to contain the following criterion:

1. Only predefined number of general layouts should be used in the web portal application.
2. User interaction elements used should be identical, per use case.
3. User interaction element placement should be identical, per use case.
4. User interface element sizes should be identical, per use case.
5. The state and feedback of user interactions should be displayed in identical manner, per use case
6. All user elements used should be implemented using only the predefined color scheme.

Fulfilling the aforementioned criteria requires commitment and unified comprehension of the design principles from the developer team. But one important thing to aid in the adaptation process of the design principles, is that there should be documentation of the principles chosen. The importance of documentation is increased when the requirement of possible third party content is considered as well. Enforcing the coherency requirement to apply also in the third party content will be a challenge, as the documentation itself does not enforce this. Instead the content has to be manually inspected.

3.2 Monolithic application

When choosing the architectural principles of a new software development project, one can choose nearly any of the most common principals and the project results to a large monolithic application, with one large and hard to maintain code-base [39]. Even when designing the system to be modular, the end result is still one large code base which is deployed as a single application rather than a truly modularized application which consists of several independent parts.

The monolithic approach in software development has long been the default solution. And that is no wonder, because before cloud computing and cloud based servers, the computational power needed to run the software had to be set up and managed by an on-premise team. That is a lot more labor intensive compared to nowadays, when a new cloud based virtual server instance can be started with a few mouse clicks and the application then can be made available to the public.

Problems with monolithic applications often boil down to the scale and lifespan of the application. When the code base of the application grows, so does the application's complexity [29]. And the more complex the application itself is, the more complex the issues which will come up. Debugging bugs out of a large code base is often a relative nightmare, because of the different kind of dependencies and developer related differences. Even with common practices there are always differences in how each developer implements things. Issues with dependencies can be managed with documentation and clear architectural structure, but there is always that bit of tacit knowledge which only a single developer knows. And that knowledge is often the knowledge you would need to solve the bug.

Because monolithic applications will be deployed as single running instance even a small bug hidden somewhere in the large code base can cause the entire application to crash suddenly [39]. This issue is something that can come up especially when adding a new and less tested feature to an existing system.

Developing a large complex application can take years to complete and in that time inevitably the technologies used in the project become outdated or completely obsolete [29]. Also the developer team's personnel will change or at least become more experienced. This all will create technological debt for the project. New updates to the program-

ming languages or frameworks used can bring new features that can make some tasks much simpler. At the same time, updates can break old solutions or support to features can cease to exist. As the development team gains experience through out the project, they can learn that their old practices have been wrong or sub-optimal. Changing coding practices or updating underlying frameworks when the project is continued for a year or two will cause issues and it will be a difficult decision to make. If the developer team chooses to not to do any changes the project can continue as planned for a while, but completely grind to a halt on the last legs of the project. Or they can choose to update and cause the project to delay, but still to complete. Updating can also drastically facilitate the maintenance of the project afterwards.

Adding or removing features in a monolithic application is not an easy task either, because of the complex dependencies and the possible technological debt . Or it might be even impossible to add some feature, because the technological choices made do not provide support for it.

When it comes to scaling of a monolithic application the problems continue to mount. If the application has multiple services with different amounts of load, you cannot scale only a single service of the application, you have to scale the whole application [39]. For example if the data service which is responsible of getting data from the database is being used by every other service in the application, it automatically becomes a bottleneck of the application. And there is not much you can do about it.

Monolithic applications still have their own place where the cloud based solution is not possible or the scale of the application is relatively stable. For example, an application's physical environment can restrict its access to the Internet and therefore make the cloud based solution impossible. Or if the application is hardware specific, often the only option is to create a monolithic application which is then run locally.

3.3 Microservices in front-end architecture

The architectural idea of microservices was initially introduced as an architectural solution for back-end applications and it stayed that way several years. Starting from 2016 the microservice approach has also been considered to be used in the front-end applications [37]. This is largely due to the development of single-page applications and client-side rendering. The advantages of using microservices remain the same when used for front-end applications as they would be when applied to the back-end applications. As the large monolithic front-end application is divided into multiple independent applications, the granularization of the code base results into more manageable sized projects, which then can be developed by multiple different developer teams. A smaller code base is also less error prone as mentioned in section 2.4.

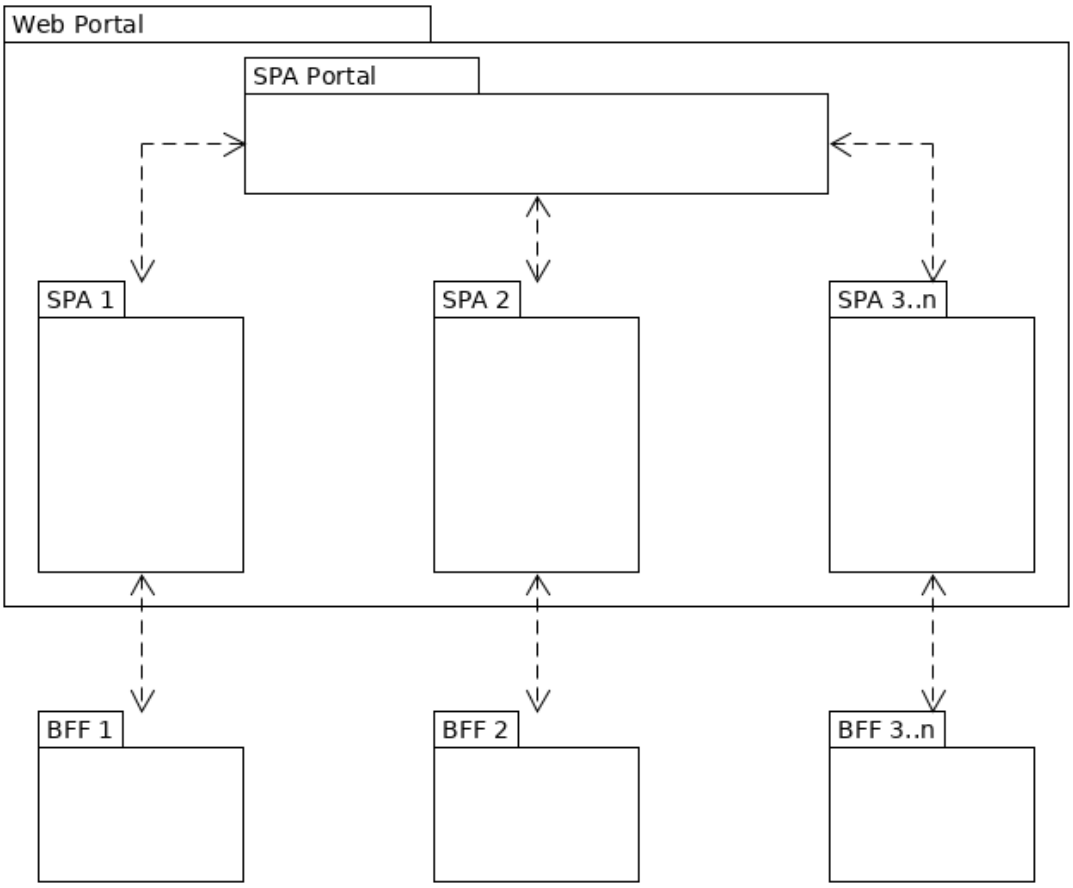

Figure 3.1. *High level architectural structure of the web portal application*

As shown in figure 3.1 the front-end web portal application consists of a single shared single-page application, *SPA Portal*, which is used to handle the portal wide actions, such as navigation between the other single-page applications. All other single-page applications, as shown in the figure, work independent from each other and are only connected to their own back-end API, also known as *back-end for front-end* (BFF). This kind of solution makes it simple to add new single-page applications to the web portal as the single shared application is the only one which needs to be modified when adding a new application. The other applications can continue to run completely unaware of the newly added application.

As one of the requirements for the web portal was that the portal needed to be easily extensible in the future and that all of the different applications which will eventually form the portal itself were not specified beforehand, the utilization of microservice architecture would have clear advantages over any monolithic approach. One significant advantage is the independent nature of a single microservice application – as the final configuration of the portal is not completely specified, it would be easy to divert and alter the configuration of how the portal is displayed to the end user. The architecture would allow combining different applications to a single view or allow applications to be divided into separate applications, if some part of an existing application would be seen to be useful

in multiple parts of the portal. The independence of a single microservice application also makes it easy to extend the portal with new applications in the future without affecting the other functionalities and it allows the portal to be released to the public even before all applications are developed and production ready, as the applications should not be tightly coupled to any of the other applications.

Other major requirement for the web portal's architecture was that it should not be technologically restricted to a single framework and it should be possible to add new features which were developed with a different technology than the rest of the portal, or at least as long as the other technology used would also result into a single-page application. This kind of solution is also possible when utilizing the microservice architecture, as the applications are separate projects and do not have to share technological dependencies unlike in the case of a monolithic application, where the projects are often built with application wide dependencies. Any major technological restrictions with the web portal's architecture would then come from the web browsers, which basically means that HTML is required for presenting content and JavaScript is required to do logical functions in the client-side of the application. These low level technological restrictions then allow new features and applications to be developed with different framework versions or completely different JavaScript frameworks.

Even though the goal of the microservice architecture is to separate different features into independent applications, there is the most probable chance of needing information provided by another microservice application at some point. To implement the interfaces for this kind information sharing there are several different solutions to choose from. Some of the solutions can be implemented purely in the client-side and some of them require the use of back-end solutions. For instance, one possible solution requiring the use of back-end, would be simply to make requests directly to the back-end API of another front-end microservice. This solution would require giving access to the API for another microservice, possibly created by a third party, which can hold some risks considering security. A more preferable variant of this kind of solution would be to create a separate API to serve requests from other front-end microservices. This approach would create a clear separation between the requests, either coming from a third party microservice or the microservice owned by the BFF. If only the client-side is used to share information between microservices, one option would be to use the browser provided storage, like Session Storage or Local Storage. The keys used by the microservice to store data in the storage can be documented and provided for other microservices, which then can use the keys to access the stored data. This solution would also allow the other microservices, and completely unrelated applications as the access is not restricted, to edit the data in the store in unexpected ways. This is often not a desired feature. Another option then would be to use Event or CustomEvent API which allows, as the name could suggest, creating and sending events in the browser [7]. These events can be created, received and used in the JavaScript code, and user defined data can be added to them. When using events, it is possible to share information nearly instantaneously between microservice applications using only the client-side. The how other microservice applications can receive the

event they need or want, is done with a listener. The listeners are defined by giving the name, or the type, of the event it is going to be listening for and by defining a function which it will call in case of the event is detected. Giving one microservice application access to the events created by another application does not require anything from the application emitting the events, but of course it will help immensely if the created events are documented. The documentation could, for example contain the names of the events and describe the content of the event data, and also describe the cases when the event is triggered.

3.4 Architecture of a single page application

Though there are differences considering architectural best practices between different JavaScript frameworks, like between the previously mentioned Vue, React and Angular, some general guidelines can be made. In the scope of the web portal application and the microservice applications that it contains, the general architectural guidelines for developing a single page application will applied to the following items:

- General project structure
- Naming conventions
- Component structure
- Component responsibilities
- Handling of application state
- Handling of data
- Application styles

These guidelines should not be used as the single source of truth, but merely as a starting point. The framework specific best practices and design patterns should be applied on top of these guidelines.

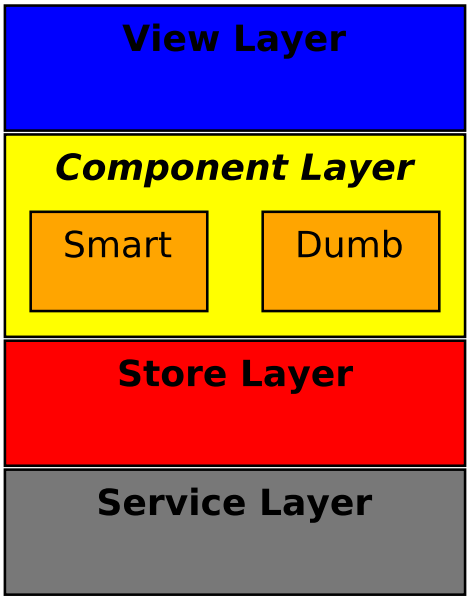

Figure 3.2. *High level single page application architecture*

The general project structure, as in the structure of folders and files of the project, should imply how the single page application's logic is layered. Going through these layers from top to bottom, as illustrated in figure 3.2, on top of everything else is the **view layer**, which is what the user can see in the browser window. Next is the **component layer**

comprised of components which focus on visually presenting the information needed. These components then can used by the views, which will provide the information needed for the components. The **store layer** is next and it holds the data needed by the view layer. To get the data to the store layer there needs to be a way to make requests to an external data source. These requests are handled by the **service layer**.

Figure 3.3. *Generalized example of a single page application project structure*

The figure 3.3 illustrates how the folder and file structure can reflect the architectural structure of the application. The view components are all placed in to the *views* folder and there can also be nesting of views, which can reflect how the application's navigation works. Each view component file is placed in its own folder to allow the view to have its own child components, in addition to the components in the root level *components* folder. With components the practice is the same; every file is in its own folder to allow creation of child components. The names of child components should always have the parent's name as a prefix. This to clarify the relation between the components. Components used in a single page application can be divided into dumb and smart components. The dumb components are called dumb components because they should only interact with their

parent. The parent passes information to the dumb component as properties and the component returns a value or event to the parent if needed. Other than this, the dumb components should not be aware of anything. The smart component on the other hand can access the data in the application's store layer or directly interact with an API via the service layer. The view components in the view layer can be also considered to be smart components. The difference between a smart component and a view component is that the view components have their own route in the navigation of the application and they should not publicly shared. The store layer holds the application wide state and caches the data fetched from external sources. It provides interfaces which enable the view components to get data from store and create or edit the data it holds. These interfaces are generally implemented with an observer pattern to ensure the store layer is not tightly coupled with the component and view layers, and to enable updating the data shown for the user as soon as it changes [8][9]. The purpose of service layer is to provide abstraction to interactions with external data sources and to provide shared helper functions used across the application. The service layer is divided into two separate folders, *services* and *helpers*, as seen in the figure 3.3. This is done to clarify the difference between what is used for outside interactions and what is for application's internal purposes. The content of *services* folder is comprised of utility classes that provide interfaces for making requests to different APIs (Application Programming Interface) and their endpoints. For example, *user-endpoint.js* file will hold utility functions for creating, updating, deleting and fetching users from an external data source.

In addition to all code files that bring functionalities to the application, it is necessary to have files that provide visuals to the application. These images and style sheet files are contained in their own *assets* folder. The contents of this folder define the shared visuals of the application, and the component specific visuals should be placed inside the components themselves. The component specific visuals can be implemented in several different ways. One method is to use inline styles, as in directly applying properties inside the *style* attribute of HTML element. Another option is to use some sort of *CSS-in-JS* library, like the popular Styled Components [12]. This method allows creation of styles with JavaScript syntax directly inside a JavaScript file and it will compile into CSS classes that are named with a hash to avoid any collision with other style sheets. Use of JavaScript also enable the use of variables, which can help with maintaining a unified look in the user interface. Third option is to use scoped or module CSS that are compiled to be component specific. With this option the styles are defined with CSS syntax, or with some CSS extension language, like SASS (Syntactically Awesome Style Sheets) that is used in our development process. As SASS was already widely used in our existing projects, the last option was the best solution to be used in the web portal application. The advantages of SASS are more thoroughly explained in the chapter 4, which explains the ideas behind the design system created for the web portal.

In the component layer, which comprises of smart and dumb components, it is important to know when to create a smart component and when to create a dumb component. The benefit of a smart component is that it is smart, and it does not necessarily need anything

from its parent. This can make using of a smart component very easy, as the developer can just add the component into a new view and it just magically works without any extra work. However, there is also a drawback in the component's smartness. The drawback is that the component has to be exactly aware to where it is connected and what is it doing. As the information for the smart component is not coming from the parent, it needs to be connected either to the application's store or service. Coupling the component in this manner decreases its re-usability in the application and portability to other applications. The benefit of a dumb component is that it does not need to know anything beyond the properties it receives from its parent. With those properties the component can then do what ever it does and possibly raise an event, which its parent then can handle. This of course then requires knowledge of the inner workings of the dumb component and some manual labor from the developer to define the properties needed. Using a dumb component then can be thought to be more difficult, but it also means it is not directly coupled to anything and can be re-used all over.

Considering the mentioned facts, the use and creation of dumb components should be prioritized over the smart components. This way the amount of re-usable code should increase, as the dumb components can be re-used also in the future projects, which in turn can increase the speed of development in the long run. When a need for a new component arises, the first option should always be to start implementing it as a dumb component. If later on it is clear that it cannot be re-used anywhere, then it can be converted to a smart component. One suitable situation, where the smart component can be considered as a default approach, is when a piece of information is only used in a single manner and the component is possibly needed in multiple places. For example, if a single property in the application's state is always defined by a select box input and it is needed in multiple views that are not nested. In this situation it is easier to create a smart component that handles the property changes internally, instead of re-writing the same event handlers in every view.

Through out all of the layers of a single page application, granularity is important. Dividing the code to smaller pieces makes it more understandable, no matter if the code is divided into shareable modules or if a single large file is just split into multiple files. For instance, when planning how to divide the application's store, a good rule of thumb is to create a separate file for every different object type that comes from an external source. When, and if, there are relationships between the objects in the store, then a separate file can be added per relation to hold the code that manages the relation. Dividing the store in this manner keeps the individual file size small, as a single file primarily holds only implementations for creating, updating, fetching and deleting of a single object type. Additionally, the file structure of the store clearly reflects what kind of data is in the store, which helps the developer to understand the application's data structure better.

3.5 Adapting the existing implementations in to the architecture

As some of the features, which are going to be part of the web portal, were already implemented before the actual system level architectural design began, it is necessary to consider how the existing features could be adapted in to the web portal. For an existing feature to be able to be adapted as a part of the new architecture, it is required to be independent to an extent where it does not require other features to be functional. It also needs to have its own API provided by a dedicated *back-end for front-end* -application.

At the time when the web portal architectural design began, two separate applications were complete and ready, and a third application was rapidly being developed. The features contained in these three applications were for the most part independent, but one major feature was shared by all of the three applications – the management of legal entities. For any of the three applications to have any real use cases, they all required to have a possibility to create and edit legal entities. This is because the purpose of nearly every other features was to add new information about the existing legal entities. So, to fully follow the new architectural structure of the web portal, the feature of managing legal entities should be separated from the three applications and made into its own application. This would also ensure that the legal entity management works the same way in all of the applications. Besides this, the existing applications could easily adapted into the architecture as they all had dedicated *back-end for front-end* -applications.

Even though the existing applications fit into the higher level microservice architecture, there were issues considering the architecture at the single page application level. Comparing the first of the three existing applications to the architecture described in section 3.4 and going through the four layers – view, component, store and service – major shortcomings could be found. Looking at the view and component layers, there were not any separation made between the two, as all of the views and components were stuffed into a single folder and the naming conventions were not followed. The principles of how components should be divided into smart and dumb components also were not followed, and major part of the components were smart components even that was not necessary. Another issue with the application's structure was that the store layer did not exist. Because of this the data was being re-fetched every time a view changed, or in some occasions even when a component was changed. Continuously re-fetching data from an external data source decreases the application's performance and usability, as the user needs to always wait for the data to be re-fetched, rather than just be nearly instantaneously loaded from the store. The state of the application's service layer was much better, as it existed and was separated to its own folders. But still, it also had shortcomings considering its granularity. All of the utility functions that were used to interact with an external data source were bundled up into a single file, which made the service hard to understand.

Fortunately, the architectural situation in the other two existing applications was a bit bet-

ter. There was clearly something learned during the development of the first application, because the project structure was improved. The components and views were now divided into their own folders, and naming conventions were better followed. Individual components were also placed into their own folders, which also helped to identify the relations between the components. The issues considering the store and service layers still remained, as the store layer did not exist and the service layer's granularity was still poor. As with the first application, the service layer of the other two applications consisted only from a single file. The issues with the service and store layers in all of the applications should be attended in the future.

4 PLANNING OF THE DESIGN SYSTEM FOR THE PORTAL

The first application, which would be part of the web portal, began its development in the beginning of the year 2018 and it was released to production in the end of the year 2018. The second application started its development in the end of the year 2018 and was completed by the end of March 2019. This meant that one whole application was complete, and the second application was near to completion, before the architectural design for the web portal began. And therefore a lot of user interface design and interface related implementation was already done. The existing work provides a helpful baseline for the design system, as some of the issues are already thought out. In turn the existing work causes issues, because changes need to be made and those will break the user interfaces of the existing applications.

4.1 Issues in existing implementation

Though there was a lot of work already done for the user interface and application styles during the development of the first two applications, the existing work had also a lot of major issues. Probably more issues than the work had actually solved.

One of these issues was the fact that there was a lot of style sheet code, several thousand lines of it, even though the applications themselves were relatively simple and small in size and a third party style sheet library was used to provide a base for the application styles. In addition, all of the code was placed in to a single code file and was unorganized, so it was quite a difficult task to keep track of what was actually there and what was actually used in the existing applications. This was the first issue which needed to be solved, before proceeding with the actual task of creating the design system. After scanning through the existing applications and seeing what parts of the style sheet code was actually in used, more than a third of the existing code was removed due to it was never used in the applications.

While removing any unused styles from the code, it seemed that there were many classes which were completely identical, or tried to complete the same task, or shared a lot of the same attributes compared to another class. This kind of duplication of code adds unnecessary complexity to the styles and makes it more difficult to keep a consistent style

through out the applications. As an example there were three different classes, which all made an element to fill the available vertical space: *.fill-height*, *.full-height* and *.height-full*. Removing these duplicate classes would further improve the understandability of the style sheet code and improve the maintainability.

```
thead tr:not(.is-selected):not(.disable-hover)
th:nth-child(1):hover.triangle-container.nine.second-row.tooltip-trigger {
    background-color: purple
}

.button.is-danger.button.is-small.is-outlined.is-bold:focus-within {
    color: red;
    background-color: white;
}
```

Figure 4.1. *Examples of overly specific selectors in the existing style sheet*

Next clear issue with the style sheets was the way the third party was used in the applications and how its default styles were overridden. First of all, because all of the style sheet code was in a single file and completely unorganized there was no clear way to see if a class was an overridden version of the third party library's class or if it was just a regular class created by our developer team. For instance, the was a class *.table*, completely generally named class with no reference point about its origin. There were also several overly specific selectors, examples of which are seen in the figure 4.1. These kind of selectors were in many cases used to change an elements color in a single view and another similar selector would be used in a another view to do the same thing. Using selectors like the examples shown above are most likely unnecessary and are the kind of code which will cause issues in the future, as they are exceptionally hard to understand and have multiple points of failure.

```
.wider-layout {
    margin-left: 60px;
}


.column.remove-gap-3 {
    padding-bottom: 0px;
    margin-top: 5px;
}


.under-table {
    margin-top: 15px;
    font-size: 12px;
}
```

Figure 4.2. *Example cases of bad naming conventions*

The naming convention used in the existing style sheet code was also an issue, or mainly the lack of naming convention. Every developer had named classes in a different manner

and in many cases the name had nothing to do with the actual content of the style sheet class. For instance, there was a class named *.wider-layout*, as seen in figure 4.2, and what it did was it set the left margin attribute of a element to a value of '60px'. This class definition without any context does not actually tell anything about what it is supposed to do and where it is supposed to be used.

4.2 Improving the existing implementation

To solve the above-mentioned issues, and after removing the all unused style sheet code, several steps was made to clarify the structure of the code and to bring coherence to the application style. The first step was to scan through the used styles and separate them to several files to bring some structure how the styles are formed. The styles were separated by the purpose of the style attributes, for example classes which changes element sizes, paddings or margins were separated to a single file and all classes related to element colors were separated to their own file. Also every class which was used to override or was related to a third party provided classes were put into their own files. The classes which were clearly application and component specific were moved inside the components themselves. Encapsulating the component specific styles into their related component helps to avoid any collision with other styles as the encapsulated styles override by default all other styles inside the component [32].

After the style sheet code was separated to multiple files, it was clearer to see which classes had the same purpose and were identical with each other. For example, the three above mentioned classes which filled the available vertical space, two of them were deleted and their uses were replaced with the single class which was kept. There were also several classes which did nearly the same thing. For example, one class set the padding to a value of '5px' and another class set the padding to a value of '5px', but also set the color as black. These classes could both be necessary, but considering the context, the color attribute in the other class had no effect at all, so it was deemed duplicate and deleted.

There were also several class definitions, where it was clear that the developer had bluntly just added new attributes in an effort to achieve wanted end result, without really understanding how the existing implementation worked. For example, there were multiple classes where positional attributes, like top or bottom, were defined. Even though the position -attribute was not defined first. Without first setting the value definition of the position -attribute, the other positional attributes become redundant as they require the position -attribute to function correctly [6]. In these simple cases, the redundant attributes could be just removed without breaking the actual functionality of the class. The more complex cases, where the defined attributes actually all had a visual and a functional effect, required more insight to what was actually tried to achieve with the implemented class. For instance, there were classes which used relative height definitions to align elements to the bottom of their containers. This kind of implementation may work when

there is not anything else in the container, but when more elements are added to the same container it often causes issues, like overflowing the container. More suitable solution for aligning the element is defining the position -attribute and other positional attributes, or using a flex container and defining flex -attributes for the elements inside the container.

Not mentioned earlier, but the existing projects utilized Syntactically Awesome Style Sheets, better known by its abbreviation SASS, which enables the use of variables and other programming functions to be used in the definition of style sheets [30]. Though the projects used some features, like variables and mixins provided by the extension language, those were not used in scale or to full advantage. The features provided by SASS can be utilized to achieve more modular and unified style sheets. For example, variables can be used to defined all simple shared property values, like colors, padding and margin sizes, and font sizes. These variables then can be imported and used in all style sheets and components. SASS also supports extension of classes, which can be used to implement inheritance like features. For example, a *.button* -class can be defined to act as a base for all buttons. For instance it can define the basic size and shape of the buttons used in the application and other button element classes will be extended from it. The classes which extend the base class, in turn can add definitions for more visual attributes like colors and borders. The mixins provided by SASS can be used to reduce repetition and boilerplate as mixins allow defining multiple attributes at once and parameters can be used to create more generic functionalities. For example, mixins can be used to create all different vendor specific attributes, like *-moz-transition*, *-webkit-transition* and *transition*, with a single line of code. If transitions with different durations are needed, then a parameter can be added to the mixin to define the duration.

The first part of the process in a better utilization of the features provided by SASS, was to create a file for color variables, which would define the whole color scheme for the applications and web portal. The color variables also would be used to override the existing variables of the third party style library, instead of the previously mentioned overly specific selectors. The file would also provide a single source of truth for the developers when they create new applications or features, which ensures that the newly created content is inline with already existing content, at least when it comes to the range of colors used. A similar approach was taken to define the typographical scheme for the applications. The font families and font sizes used in the applications were defined as variables and simple utility classes for different text styles were defined to ease out development of new content. Additional variables were also defined in places where style sheet classes were related to each other in a way where different attribute values should be based on a same underlying value. For instance, when a menu component is collapsed and only visible as narrow element, the main container should fill the freed up horizontal space in the view. In our case the main container's *left margin* -attribute was related to the menu component's *width* -attribute, so a variable was defined to hold the value for these two attributes. This way only a single change is needed to change the horizontal space distribution between the two.

The more programmatic features of SASS in form of mixins and for-loops were used to create a variety of utility mixins and classes. These could be used to provide before mentioned vendor specific prefixes and provide extendable classes. The utility classes also contained classes which were planned to define basic layout structures and could be used as is. For-loop in SASS is particularly helpful in creating classes based of a list of attribute values. For instance, creating a button class for each main color in the application color scheme can be a tedious task by hand, but when utilizing a basic for-loop, same task is done with a few lines. Also creating the classes in this way helps to prevent unnecessary clutter and duplication in the style sheet code. Additionally, it is less prone to any human elements. For example, after adding a new color or changing a color, a developer does not need to remember to create or alter the button class as well.

In addition to better utilization of SASS and its features, a way to create more consistent look and feel to the user interface is to create common shared components between different applications. The shared components are not just elements styled in a certain way, but they also function in a specific manner. In our applications for example, the data is mainly inputted in to a data table which should have some related features to it like sorting, filtering and searching. At first all these different features were always re-implemented in every new table created, and this caused often unnecessary bugs and inconsistencies in how the features worked. To solve this issue, a shared component was created which implemented the needed features in a unified way and reduced the work needed to be done, when adding a new table for inputting data in the applications.

```
// Paper component
$paper-shadow: 4.5px;
$paper-shadow-double: $paper-shadow * 2;
$top-paper-min-height: 125px !default;
$tabs-height: 32px !default;

.paper {
    background: $white;

    &--shadowed {
        @extend .paper;
        min-height: 300px;
        min-width: 300px;
        width: calc(100% - #{$paper-shadow-double});
        height: calc(100% - #{$paper-shadow-double});
        box-shadow: 2px 2px $paper-shadow $grey;
        margin-left: $paper-shadow;
        margin-top: $paper-shadow;
        margin-bottom: $paper-shadow;
        &.top-paper--tabs {
            margin-top: 0;
            min-height: $top-paper-min-height - $paper-shadow;
            height: 100%;
        }
        &.top-paper--no-tabs {
            min-height: $top-paper-min-height + $tabs-height;
        }
    }
}
```

Figure 4.3. *Example use case of a better naming convention and use of variables*

Based on the different methodologies for creating more understandable and modular style sheets, introduced in the section 2.3, a more coherent naming convention was utilized while solving the issues in the existing style sheets and while creating the new shared components. The figure 4.3 illustrates how the naming convention was applied and how variables were utilized. Partly the naming convention chosen was dictated by the third

party library used in the existing applications, which follows its own naming conventions, a mixture of OOCSS and SMACSS. As the third party library was mainly used for defining layouts and simple user input elements, also the classes created by our developer team were named accordingly. But for shared components and for more complex user input elements, the naming conventions used follow mainly the principles of BEM. This kind of naming convention was chosen, because when using BEM's conventions the class attributes in the HTML markup will stay more concise. When writing HTML markup and using the third party library, the class attributes often had at least three or four different class names added. With BEM, mainly a single class name will suffice.

5 IMPLEMENTATION

As the architectural design of the front-end for the portal was finalized it is only a matter of implementation to make the portal a reality. When the implementation of the portal began there were already three applications at least in their end-user testing phase of the development, so there was a time pressure to get the portal implemented and deployed into testing and production environments.

5.1 Technologies and tools used

Some of the choices for the technological solutions and tools were predetermined by the existing and or developed web applications, as well as the existing development team. Because the development of the first partial application for the web portal began long before even the actual designing phase of the portal, it meant that decisions made during the first web applications development would affect the portal as well. This is not an ideal situation by any means, but it also meant that the choices already made would have been proven right or wrong during the first development project.

5.1.1 Vue –framework

Out of many existing JavaScript frameworks which are used in the development of single-page applications, Vue was chosen to be used in our first application's development. And because it was proven to be an intuitive, reasonably mature framework there were no valid reasons why we should change it to another framework like React.

As said, Vue is a quite mature framework, at least in the domain of JavaScript frameworks, original version released in 2014 [44] and the second major version released in 2016 [45]. While not nearly as popular as React or Angular, Vue has constantly increased in popularity and size of community [14]. One of Vue –framework's strengths is the ease of adaptation, when compared against React or Angular. If a developer knows the basics of JavaScript and HTML it is quite easy to create also a Vue application, this is because Vue's template syntax is really close to the basic HTML and the most basic life-cycle methods of the framework are really self-explanatory, like 'created', 'mounted' and 'updated'.

```
<template>
    <div>
        {{result}}
        <button @click="m = m+1">Click
        </button>
    </div>
</template>
<script>
export default {
    props: {n: Number}
    data() { return {m: 1} },
    computed: {
        result() { return (n * m) }
    },
    created() {}
}
</script>
```

Above is a simple example of a Vue component that uses the single file component style to allow hot reloading of the component, if any changes occur while developing. The `data`-function describes the components internal state or model. The properties defined there are reactive and when changes occur in their values a re-render is automatically triggered. The computed properties, in the above example the `result`, can be used to reduce logic inside the template. The difference between a computed property and a basic function is the fact that the computed property only changes when a reactive property it uses changes. Otherwise a cached result is returned instead of re-evaluating the whole function. These are some of the features that make the Vue stand out from other frameworks or libraries.

5.1.2 single-spa –framework

To build the foundation for the web portal we used *single-spa* –framework which is a meta-framework for combining multiple single-page applications into one larger application [4]. *single-spa* –framework offers a simple API for registering and managing connected single-page applications. Registering an application requires implementing hook-methods to the target application. These are used by the *single-spa* –framework for loading, showing and hiding the application. Applications then can be added to the *single-spa* –frameworks own configuration.

User can control the visibility of the single-page applications by defining an activity function which returns a Boolean value based on chosen conditions. For example, applications' visibility can be easily defined by route. This function is needed when registering

the application. There is no limitation set by the *single-spa* –framework when it comes to displaying multiple applications at the same time. For example, it is common to have top level navigation as a separate application which is constantly displayed for the user.

The features that the *single-spa* –framework implements are reasonably simple and could have been implemented by our own developer team. But as the API of the framework was deemed simple and easy to use, there was no point of re-inventing the wheel in this sense. On top of this the framework provides comprehensive documentation and examples on how to use the framework in several different use cases. This all is something which most probably will save time in the development process of the web portal and will need less maintaining in the future.

5.1.3 Webpack

Webpack is a widely used JavaScript module bundler and task runner used to help out developers to build and deploy their JavaScript applications [43]. Webpack is used in the development of modern web applications for handling the bundling of all different JavaScript packages and libraries into a single application. It can be configured to run the application in development mode, which allows developers to debug the application and make changes almost in real time. Or it can be configured to build production version of the application with better performance, smaller size and consisting only of the essential packages. [26]

5.2 Implementation of the web portal application

After drafting the initial architecture for the web portal itself and figuring out the biggest technological issues with the architecture, mainly how to get the different single page applications work together, the actual implementation phase began in March 2019. The implementation process covered the creation of the *SPA Portal* -application – show in figure 3.1, the configuration of the existing applications, the creation of the top level navigation and the utilization of the design system described in the chapter 4.

5.2.1 Top level web portal application

As shown in figure 3.1 in section 3.3 the architectural base of the whole web portal is one single page application which encapsulates or more precisely holds a reference to all other underlying single page applications. The top level web portal application uses or implements the *single-spa* –framework and its required configurations and functions. The most basic implementation could be achieved with only two files, one HTML file and one JavaScript file. The HTML file would provide the mounting point for the *single-*

spa –framework and the JavaScript file would define the configuration and initiate the framework instance by executing the `start` -function of the framework when the file is read. With this kind of setup the framework would run, but nothing would show in the browser as the framework does not provide any user interface.

```
import * as singleSpa from 'single-spa';

function portalActivityFunction(location) {
    // If the application is always visible
    return true;
    // If shown only in certain routes
    // return (
    //     location.href.indexOf(
    //       `${location.origin}/portal`
    //     ) !== -1
    // );
}

// Example use of registerApplication
singleSpa.registerApplication(
    // Name of the application
    'portal',
     // Loading function
    () => import('./portal/portal.js'),
     // Activity function
    (location) => {return portalActivityFunction(location);}
);
```

What the *single-spa* –framework requires in the means of configuration is that the framework needs to know what applications it should be using and where those applications can be found and additionally when those applications should be used or shown in the user interface. Configuring the applications used by the framework is defined with framework's `registerApplication` -function, example of which is shown above. This function takes three required parameters, the name of the application to be registered, a loading function or an application object, and an activity function. The loading function or application object should both provide the *single-spa* –framework the information it needs for loading the application, and for activating and deactivating the application. These three actions are done by implementing three API methods of the framework in the application which is to be registered. The activity function is what the framework uses to determine if the application should be active or not. The activity function gets the current browser URL or location as parameter, which can be used to determine the activity of the application. Alternatively the application can also be always active, which can be useful.

5.2.2 Configuring the underlying applications

For *single-spa* –framework to be able to manage other single page applications, the applications need to implement three interface methods. These methods are the counterparts of the loading function's content mentioned in section 5.2.1. The three methods are `bootstrap`, `mount` and `unmount`. The first is used by the framework to import the application in to the framework application. The second and third methods are used by the framework to show and hide the application from view. Implementing these methods is merely a trivial task because of the existing packages, which allow a developer to implement these with a single statement. The packages are provided for React, Angular, Vue and several other JavaScript frameworks. Example implementation of these methods with Vue shown below.

```
/* import Vue, VueRouter, App, singleSpaVue, router */
const portal = singleSpaVue({
    Vue,
    appOptions: {
        el: '#portal',
        render: h => h(App),
        router
    }
});
export const bootstrap = [
  portal.bootstrap,
];
export const mount = [
  portal.mount,
];
export const unmount = [
  portal.unmount,
];
```

In addition to creating the API methods required by *single-spa* –framework, there are other changes that needed to be made to the applications that are going to be part of the web portal. One alteration that is needed, is including all the required files as part of the application bundle. The application registered to *single-spa* –framework is no longer going to have its own HTML file as a mounting point and with that, there is no longer the possibility of including files and other dependencies through the HTML file. To overcome this, the files need to be either loaded as module to get them to be part of the JavaScript application, or they need to be dynamically added to the DOM after the application is loaded by the *single-spa* –framework.

As part of the requirements set for the web portal implementation, it was required that the applications that would be a part of the web portal should also work as standalone

applications. This requirement creates the need for creating separate configurations for the web portal and the standalone versions of the application. Because of the previously mentioned requirements dictated by the *single-spa* –framework, the web portal version of the application and the standalone version need to have separate entry points in the applications build process. The entry point is simply a single JavaScript file from the build is planned to start. The separate entry point is required because of the *single-spa* –framework methods and the application instance need to be enclosed in a single object. Additionally the build process of the standalone application needs to produce a HTML file which can be used as the mounting point of the standalone application. The requirement for the application working both as part of the web portal and as standalone application also is aligned with the idea of microservices, as previously stated in section 2.4. The requirement also supports the development process of the individual projects, as they can be developed completely separate from each other and the developers can take full advantage of the developer tools provided by many of the JavaScript frameworks, like React and Vue. This advantage is lost if the application development is done through the web portal application.

Because all of the applications which will form the web portal are separate projects and need to be also standalone applications, they also are all hosted in different locations. This raises a challenge as to how the applications can form a single-page application that does not require any extra browser loading screens. This is a challenge which the *single-spa* –framework does not solve on its own, it just trusts that the application is available when it needs it. Fortunately the *single-spa* –framework allows the loading function of the application to be asynchronous. This in mind, it is possible to load the wanted application via HTTP request. To ease the application loading process an external *SystemJs* library is used, which is designed for a such purpose [35].

5.2.3 Managing navigation

To manage navigation between all the single page applications, which are part of the web portal an extra application was created addition to the existing applications as shown in figure 3.1 in section 3.3 named *SPA Portal*. To be able to navigate from one application to another, the *SPA Portal* application needs to know what is the root route of every other application part of the web portal. The more specific routing is left for the underlying applications to manage.

Because all of the applications are served from the same domain, the root routes of the applications need to be unique, or there is a great possibility of conflicting routes. For example, if two different applications share the same root route and both applications have a user information page, it is likely they both have completely identical route of *'https://domain.com/<rootroute>/userinfo'*. In this kind situation the *single-spa* –framework will activate both applications. This feature can be in some cases be useful, but primarily this is not wanted and will cause the user interface to break as two different

applications try to take the whole available view for themselves.

To enable the top level navigation in all of the applications the feature of showing multiple applications at the same time can be utilized. The *single-spa* –framework allows displaying multiple applications simultaneously, if the applications do not share the same DOM element. So using separate DOM elements for the application managing top level navigation and for every other application, and defining the activity function of the navigation application to always return a true value, is all that is required to enable the top level navigation in the whole portal.

5.2.4 Utilizing the design system

The design system, of which planning and implementation process was described in chapter 4, was applied to two of the existing applications, in addition of the web portal's *SPA Portal* -application that was created as a part of this thesis. The utilization of the design system in the third existing application was left outside of the scope of this thesis, as it will also require extensive changes related to the application level architecture.

The utilization process of the design system required an extensive amount of work hours, because it caused changes through out the two applications it was applied to. In the views of the applications, in addition of renaming different style classes, it was required to alter the whole structure of the HTML templates used in the Vue applications. In many cases the better designed classes of the design system allowed removing of multiple unnecessary HTML tags, which reduced the amount of nested tags and made the templates more readable.

While applying the design system to the applications, there were deficiencies and places of improvement discovered. For instance, there was an issue with a layout element that overflowed only in one of the applications, which needed to be sorted out. Another issue was caused by the application specific styles, which in certain state caused the element to be displayed in an unexpected way. Also an additional shared component was created due to realizing the fact that a similar state change in the user interface was differently between the application. All in all the utilization process of the design system was surprisingly labour intensive, but absolutely necessary to do at this point of the web portal's development.

6 EVALUATION

After carefully planning the architecture for the web portal and implementing the core parts of the portal it is time to revisit the requirements set for the portal. After evaluating the requirements, sights can be set to the future and possible improvements discussed.

6.1 Evaluating the requirements set for the web portal

The initial requirements for the web portal's architectural design were previously defined in section 3.1. In the following sections the requirements set are evaluated in the context of architecture, as well as in the context of actual implementation. After all, even good planning does not guarantee a good end result.

6.1.1 Requirement #1 - Modularity

The first requirement set for the web portal was that it should be modular. This on its own is a quite rough requirement, so to specify it a bit more, the architecture should allow the users to acquire only the parts of the web portal they need. Considering this in the context of application architecture, multiple different ways to solve this can be thought out. The requirement can be fulfilled for instance by simply creating multiple classes, which implement the needed features. This however was not the desired solution in our case as the features of single modular piece can be quite comprehensive. Filling the requirement in this way, then would cause the classes to be either really large and complicated, or the amount of classes needed to comprise the functionalities would require extensive user access controlling.

But with the chosen microservice based architecture, the granularity of a single modular part can be more easily adjusted and user access controlled, as user access can be prevented to the whole microservice. Because of this, the scope of a single modular part of the web portal can better reflect the actual business need of the user and better reflect the information requirements of the tax and transfer pricing reports. The microservice based architecture also allows the different parts to be developed separately, as mentioned earlier in section 2.4. This can be seen as a great benefit in our software development process.

Considering the fulfillment of this requirement in the context of the actual implementation, it can be said the requirement is fully filled, at least for now. At the moment there are three applications as a part of the web portal, but in the end there should be around ten separate applications developed by our developer team. This number may further increase in the future, so it is not certain how well the chosen architecture will work. The chosen architecture helps in the development process of a single application, as the architecture forces the project size to be more maintainable.

6.1.2 Requirement #2 - Expandability

As already stated in the section 3.1, the requirement considering the expandability of the web portal should go hand in hand with the first requirement, as often modularity enables also the expansion of existing application. That said, the ease of expanding can differ greatly depending on the way the modularity is implemented. Considering the same example as in the previous section, an application which is expanded by adding new classes to the application can cause issues. These issues often come from the fact there is likely to be dependencies between the classes, or the new class can still contain bugs that are not caught during the development process. When these bugs then come up in the production environment, it might cause the whole system crash.

With the microservice based architecture, the issues that might come with expansions can be more easily avoided. As each microservice application is an independent application, the addition of new applications to the web portal should not affect any other existing application. This of course depends on the way the possible interactions between the microservices are done in the web portal. If there are tightly coupled microservices issues might occur, but with loose coupling these issues should be avoided. The microservice architecture also makes it possible to easily maintain single parts of the web portal. So even if bugs are found in the production environment, only the affected part of the web portal can be taken down and bug fixes applied to it. After fixing the issues the part can be put back online, again without affecting the other parts.

Again considering the requirement in the context of implementation, the requirement can be thought to be fulfilled. By taking advantage of the functionalities provided by the *single-spa* –framework, it is relatively easy to add new applications as part of the web portal. The amount of implementation required by the *single-spa* –framework itself is by any means not a lot, but major part of the work required to get the application to work as a part of the web portal is done directly to the application itself. Each application needs a certain amount of configuration, and for instance the application's entry point should be defined in a certain way. The configurations that are required are explained in detail in the sections 5.2.2 and 5.2.3. Managing all of the required configurations might cause issues in the future, if the amount of applications increases a lot.

6.1.3 Requirement #3 - Expandability with third party content

In addition to the previous requirement, the web portal should be also possible to be expanded by content developed by third party companies. Considering this requirement in the context of architecture, the microservice based architecture should allow this as well as it allows expanding the web portal with the content created internally. This of course is the case only if the third party companies adhere to the architectural requirements set for the applications. The requirement list though is not too long, as it basically consists only of a single item, the application should be a single page application.

Considering this requirement in the context of the implementation, it cannot to be said for certain whether or not it is possible to add third party content to the web portal application. It would be harsh to say it is impossible, but it might not be as easy as one would like it to be. The challenges of adding the third party content to the web portal come mainly from the technological choices made. As stated in previous section, a major part of the work required to add a new application as part of the web portal comes from the configurations mentioned in sections 5.2.2 and 5.2.3. In addition the *single-spa* -framework's support does not cover all possible JavaScript frameworks that are available, so implementing even the required life-cycle methods can provide a challenge. Also if the amount of applications in the web portal increases enough, it might become a challenge to ensure every application has a unique name and root route, which are required by the web portal to able it to function correctly.

6.1.4 Requirement #4 - Technologically independent

As stated in the section 3.1, the technological domain of web development and JavaScript frameworks changes and evolves constantly, so it is preferred that the architecture of the web portal should be technologically independent to allow changing of the development technologies. Again the microservice based architecture with its independent microservice applications enables the development of applications with different technologies. This can be a really beneficial feature considering the requirement of allowing the expansion of the web portal with third party content. This gives the third party content producer the freedom to choose their own technologies, rather than the web portal's restrictions forcing them to use a certain technology. This also aids our own development process, as we can adopt new technologies along the way and possibly update our existing applications to use the new technologies in the future without needing to update every application in one go.

The fulfillment of this requirement is also proved in practice in the current version of the web portal. Even though the existing applications all are developed using the *Vue* framework, the framework versions between the different applications differ. In addition, there are several existing example applications and real life applications which show that

the *single-spa* –framework and the use of microservice based architecture in the front-end allows the heterogeneous use of technologies. Multiple proof of concept projects can be found for instance from the *single-spa* –framework's home page.

6.1.5 Requirement #5 - Coherent user interface and experience

The requirement concerning the coherency of the user interface and the user experience in the web portal was comprised from multiple parts. The parts considered matters like the unity of the individual user interface elements and the unity of general layouts of the user interface.

The first part of the requirement stated that there should be finite number of general layout styles, which all of the views in the whole web portal should follow. This matter at the moment is a bit open, as there is yet no strict definitions on how each and every view is allowed to be laid out. However, the general layouts of the currently existing applications can all be said to be following the same guidelines.

The second part of the requirement stated that the user interface elements used for user interactions should be unified per use case. This means that all of the inputs used for inputting text and the button which is used to add a new item should look the same in every view. Although the design system, which was described in chapter 4, helps in the fulfillment of this requirement, it does not automatically solve any issues. The design system only helps the developers to implement the user interface elements in the right way, but it requires the commitment of every developer in the developer team.

The third part of the requirement stated that the user interface element placement should unified in all of the views. For example, if there is a search bar element on top of a data table it should be there in every view. If the element positions randomly change between the views it makes it more difficult for the user to efficiently use the application. Again, the design system can help the developers to implement the user interface correctly, but it requires commitment and knowledge what the design system contains. The example case mentioned, for instance can be implemented using a shared component, which was created as part of the design system.

The fourth part of the requirement covered the matter of sizing the user interface elements in a unified manner. Again, the design system created helps the developers by providing utility style classes and variables, which can be used to define the size of an element. In addition, a bundle of shared components was created to both help the developers and to enforce the size and style of certain elements. The utility style classes, variables and shared components also help to fulfill the fifth and sixth part of the requirement, which considered the matters of displaying the state and providing feedback for the user, as well as the color scheme used throughout the web portal.

This requirement can be considered to be fulfilled to a degree. Considering the existing applications, in addition of the web portal itself, only a single application at the moment can not be said to implement the design system successfully. This issue should be corrected in the future, as well as expanding the existing design system with new shared components to better match the needs of the web portal's applications.

6.1.6 Requirement #6 - Coherent user interface and experience, third party content

An additional requirement related to the coherency of the user interface and experience was that the coherency should be extended to cover also the possible third party content coming in the future. This requirement was partly neglected in the scope of this thesis due to time limitations and its lower priority level at the moment. However, the design system created can be provided also for the third party content creators to ease their development process. Though to fully fill this requirement, the design system should be complemented with shared components to the extent where it is possible to create complete applications using only them.

6.2 Possible future improvements to the web portal

Even though a lot of issues were discovered and eventually solved while completing this thesis there are still several issues left that will be needed to be attended to in the future. Basically the whole architecture of the web portal is affected by some of the issues left.

Considering the highest level of the architecture – the microservice architecture – and the way it is implemented there are still issues remaining related to the communication between the applications comprising the web portal itself and related to the manner in which the configuration of the applications is done currently. At the moment, the existing applications do not have any form of communication between themselves, if sharing of data at the database level is not taken into account. The issue of how the applications should share information will be needed to be solved in the future. Some of the available options were mentioned in the section 3.3 and from those options the use of shared observer module could be the best choice as it should provide the most control over the information shared. Making the final choice which approach to take is better to be left to time the actual need for this functionality arises, but it is good to be aware of the options beforehand.

The adding of new applications to the web portal is a highly manual process at the moment. To add the new application to the web portal's framework it is required to define the unique name of the application, the actual location of the application (as where it is hosted) and to create the activity and the loading functions. Defining these things

manually for ten applications is laborious, but yet feasible. Defining these for 50 or 100 applications and it becomes a back-breaking task, which can be the case when the third party content creators are taken in to the picture. A less labor intensive and a more automated process should be thought up for the adding of new applications.

Considering the content created by the third parties, there is no existing documentation or interfaces defined which would allow the third party content to be integrated in to the web portal. Additionally the design system of the web portal is not expansive enough to provide a proper support for the needs of the third parties. The documentation needed should at least provide directions on what is required from their application that it can be integrated to the framework our portal uses and what limitations there exists, like the uniqueness requirement of the application name. Also the interface which allows the communication between the web portal's applications would be preferred to be existing at the moment when third party content is allowed to be integrated.

As discussed in the section 3.5, there were issues with the existing applications considering their application level architecture. For instance, one the existing applications had issues throughout the whole application structure. The view and component layers were not defined properly, the rule on how to divide components into smart and dumb was not followed correctly, the store layer did not exist at all and also the service layer's granularity was poor. Additionally the application's user interface does not utilize the design system of the web portal at all. These issues should be corrected before the web portal is released, even though they require a lot of work hours, because the issues mentioned break all of the guidelines defined in this thesis.

With the other existing applications such drastic refactoring projects are not required, as their issues were limited to the store and service layers of the applications. The store layer should implemented to the applications to provide a better and more responsive user experience, as the store enables the caching of data and re-fetching of data is reduced significantly. Also the service layer should be divided into smaller parts, which makes it more understandable and it enhances the developer experience.

One issue what might come up in the future is the size of the applications, which are part of the web portal. If the application sizes grow too much, it can lead to extensive loading times when switching from one part of the portal to another. Also the amount of applications can cause issues, if a single customer has a great number of applications and too many of those are loaded or kept in the memory at the same time. To overcome these kind of issues, there is a need for creating a mechanism to handle the memory use of the web portal.

Addition to the purely technological and technical issues remaining in the web portal, there is still work needed to be done with the design system, as previously mentioned. At the moment, the number of shared components included in the design system is quite low. This should be increased, especially when considering the requirement of coherent user interface and experience throughout the web portal. An extensive set of shared

components is an effective way to enforce the principles of the design system and to ease the development process of our own developer team, as well as the development process of the third parties.

7 SUMMARY

The web portal designed and implemented during the writing of this thesis is still in its early stage and has not yet went into production. The work done for the web portal was fully described in this thesis and as the chapter 6 discussed a lot of work is done, but also a lot of work is left to be done in the future. Hopefully the shortcomings and missing pieces of the web portal are completed during the ongoing year.

Considering the starting point of the architecture design process and reflecting that to the findings described in the chapter 6 it is safe to say that it is not ideal to begin the design process after there is already extensive amount of implementation done. The biggest challenges faced and the most work done during this thesis were related to the fact there was already a lot of existing work done. For instance, the creation of the design system and the process of utilizing it in two of the existing applications required more work than anything else. It could be even said that it took more work than the rest of the design and implementation put together. Thus, the neglect of the front-end architecture design then can truly have a profound cost in the future. This matter should be corrected and better taken into consideration in the forthcoming projects.

Creating the design system as a part of the whole front-end architecture is important, as it benefits both the end user and the developer. This is because the design system helps the developer to create a more coherent user interface with less effort by using the shared components and other utilities provided by the design system. The coherence of the user interface then helps the end user to use the system more efficiently as the user interface elements are familiar through out the system.

By choosing the microservice architecture as the base of the web portal's front-end architecture it is relatively easy to control the level of granularity of the web portal. It also allowed to fully use the existing applications that were already implemented even before the architecture design began. At this time there is no reasonable causes to say anything negative about the web portal's architecture. As stated in section 3.3 one of the advantages of using microservice architecture in the front-end is the technological independence of the microservice applications. This makes it easy to update and change the technologies used in the development process of the applications. So it is possible to choose the best technology for each specific purpose. Also the preferred way of loose coupling the microservice applications together allows the applications to be maintained individually, rather than taking the whole system down when only a single application needs maintenance.

During the writing process of this thesis, which began in February 2019, there has been many ups and downs, as there have been multiple time periods when the work related to the thesis have been put on hold. This was often caused by the maintenance needs of the existing applications, which were already used by actual customers. The microservice architecture of the web portal application was designed and implemented relatively soon after the thesis work began, as the tools used for the implementation were really easy to use. However, creating the design system took a lot of more time than expected and utilizing it was not any easier. Issues with the design system and in the existing applications made also the writing of the actual thesis a bit dull and it was quite hard to find motivation to press on, but as it seems it is now completed.

REFERENCES

[1] *Angular*. URL: https://angular.io/ (visited on 05/14/2019).

[2] *Architecture vverview*. URL: https://angular.io/guide/architecture (visited on 05/14/2019).

[3] C. Arsenault. *OOCSS - The Future of Writing CSS*. June 9, 2017. URL: https://www.keycdn.com/blog/oocss (visited on 04/13/2019).

[4] CanopyTax. *single-spa*. URL: https://single-spa.js.org/ (visited on 05/18/2019).

[5] *Composing with Components*. URL: https://vuejs.org/v2/guide/index.html#Composing-with-Components (visited on 05/14/2019).

[6] *CSS Layout - The position Property*. URL: https://www.w3schools.com/css/css_positioning.asp (visited on 05/04/2019).

[7] M. W. Docs. *CustomEvent*. URL: https://developer.mozilla.org/en-US/docs/Web/API/CustomEvent (visited on 05/16/2019).

[8] *Flux - In Depth Overview*. URL: https://facebook.github.io/flux/docs/in-depth-overview.html#content (visited on 05/16/2019).

[9] E. Gamma, R. Helm, R. Johnson and J. Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995, 293–303. ISBN: 0-201-63361-2.

[10] J. P. Gant and D. B. Gant. Web portal functionality and state government e-service. *Proceedings of the 35th Annual Hawaii International Conference on System Sciences*. Jan. 2002, 1627–1636. DOI: 10.1109/HICSS.2002.994073.

[11] *Get BEM*. URL: http://getbem.com/introduction/ (visited on 04/13/2019).

[12] M. S. Glen Maddern. *styled components*. URL: https://www.styled-components.com/ (visited on 05/16/2019).

[13] M. Godbolt. *Frontend architecture for design systems: a modern blueprint for scalable and sustainable websites*. Sebastopol, CA: O'Reilly, 2016.

[14] A. Goel. *10 Best JavaScript Frameworks to Use in 2019*. Mar. 15, 2019. URL: https://hackr.io/blog/10-best-javascript-frameworks-2019 (visited on 05/14/2019).

[15] H. Harms, C. Rogowski and L. Lo Iacono. Guidelines for Adopting Frontend Architectures and Patterns in Microservices-based Systems. *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ESEC/FSE 2017. Paderborn, Germany: ACM, 2017, 902–907. ISBN: 978-1-4503-5105-8. DOI: 10.1145/3106237.3117775. URL: http://doi.acm.org.libproxy.tuni.fi/10.1145/3106237.3117775.

[16] *Instance Lifecycle Hooks*. URL: https://vuejs.org/v2/guide/instance.html#Instance-Lifecycle-Hooks (visited on 05/14/2019).

[17] *Internet use by individuals*. URL: https://ec.europa.eu/eurostat/documents/2995521/7771139/9-20122016-BP-EN.pdf (visited on 04/13/2019).

[18] M. A. Jadhav, B. R. Sawant and A. Deshmukh. Single page application using angularjs. *International Journal of Computer Science and Information Technologies* 6.3 (2015), 2876–2879.

[19] A. O. Jason Miller. *Rendering on the Web*. May 1, 2019. URL: https://developers.google.com/web/updates/2019/02/rendering-on-the-web (visited on 05/14/2019).

[20] *JavaScript HTML DOM*. URL: https://www.w3schools.com/js/js_htmldom.asp (visited on 05/14/2019).

[21] K. Layne and J. Lee. Developing fully functional E-government: A four stage model. *Government Information Quarterly* 18.2 (2001), 122–136. ISSN: 0740-624X. DOI: https://doi.org/10.1016/S0740-624X(01)00066-1. URL: http://www.sciencedirect.com/science/article/pii/S0740624X01000661.

[22] *Lifecycle Hooks*. URL: https://angular.io/guide/lifecycle-hooks (visited on 05/14/2019).

[23] J. L. Martin Fowler. *Microservices*. Mar. 25, 2014. URL: https://martinfowler.com/articles/microservices.html (visited on 05/15/2019).

[24] M. Mikowski and J. Powell. *Single Page Web Applications: JavaScript End-to-end*. 1st. Greenwich, CT, USA: Manning Publications Co., 2013. ISBN: 1617290750, 9781617290756.

[25] S. Newman. *Building Microservices*. 1st. O'Reilly Media, Inc., 2015. ISBN: 1491950358, 9781491950357.

[26] A. Osmani. *Web Performance Optimization with Webpack*. URL: https://developers.google.com/web/fundamentals/performance/webpack/ (visited on 05/18/2019).

[27] *React*. URL: https://reactjs.org/ (visited on 05/14/2019).

[28] C. Richardson. *What are microservices?* 2014. URL: https://microservices.io/ (visited on 05/15/2019).

[29] C. Richardson. *Pattern: Monolithic Architecture*. 2018. URL: https://microservices.io/patterns/monolithic.html (visited on 05/15/2019).

[30] *SASS Documentation*. URL: https://sass-lang.com/documentation (visited on 04/13/2019).

[31] *SASS Reference*. URL: https://sass-lang.com/documentation/file.SASS_REFERENCE.html (visited on 04/13/2019).

[32] *Scoped CSS*. URL: https://vue-loader.vuejs.org/guide/scoped-css.html (visited on 05/14/2019).

[33] J. Snook. *Scalable and Modular Architecture for CSS*. 2nd. Ottawa, Ontario, Canada: Snook.ca Web Development, Inc., 2012.

[34] *State and Lifecycle*. URL: https://reactjs.org/docs/state-and-lifecycle.html#adding-lifecycle-methods-to-a-class (visited on 05/14/2019).

[35] *SystemJS*. URL: https://github.com/systemjs/systemjs (visited on 05/14/2019).

[36] *Thinking in React*. URL: https://reactjs.org/docs/thinking-in-react.html (visited on 05/14/2019).

[37] ThoughtWorks. *Micro frontends*. URL: https://www.thoughtworks.com/radar/techniques/micro-frontends (visited on 04/20/2019).

[38] *User Interface Design Basics*. URL: https://www.usability.gov/what-and-why/user-interface-design.html (visited on 04/13/2019).

[39] M. Villamizar, O. Garcés, H. Castro, M. Verano, L. Salamanca, R. Casallas and S. Gil. Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud. *2015 10th Computing Colombian Conference (10CCC)*. Sept. 2015, 583–590. DOI: 10.1109/ColumbianCC.2015.7333476.

[40] *Vue.js*. URL: https://vuejs.org/ (visited on 05/14/2019).

[41] *Vue.js Server-Side Rendering Guide*. URL: https://ssr.vuejs.org/ (visited on 05/14/2019).

[42] *Web portal*. URL: https://en.wikipedia.org/wiki/Web_portal (visited on 04/13/2019).

[43] *Webpack*. URL: https://webpack.js.org/ (visited on 05/18/2019).

[44] E. You. *First Week of Launching Vue.js*. Feb. 11, 2014. URL: https://blog.evanyou.me/2014/02/11/first-week-of-launching-an-oss-project/ (visited on 04/06/2019).

[45] E. You. *Announcing Vue.js 2.0*. Apr. 27, 2016. URL: https://vuejs.org/2016/04/27/announcing-2.0/ (visited on 04/06/2019).

[46] A. Zerner. *Client-side rendering vs. server-side rendering*. Apr. 6, 2017. URL: https://medium.com/@adamzerner/client-side-rendering-vs-server-side-rendering-a32d2cf3bfcc (visited on 05/14/2019).