

Proprietors Assembly / Hausversammlung

Final Year Project

ALEXANDER GOGL

SUPERVISOR: CHRIS CASEY

COURSE: BSc (HONS) COMPUTING

University of Central Lancashire

27th March 2019

Contents

Abstract	v
1 Literature Review	1
1.1 Introduction	1
1.2 Background of Vue.js	1
1.3 Comparison to other Frameworks	1
1.3.1 Vue 2.6.X	2
1.3.2 React 16.8.X	3
1.3.3 Angular 7.2.X	4
1.3.4 Conclusion	6
1.4 SEO for Single Page Applications	7
1.4.1 Introduction	7
1.4.2 The Problem	8
1.4.3 Possible Solutions	8
2 Project Planning	10
2.1 Introduction	10
2.2 Provided Services	10
2.3 Requirements	10
2.3.1 Digital Noticeboard	11
2.3.2 Polls	11
2.3.3 Forum	11
2.3.4 Issue Management	11
2.3.5 Bilinguality	11
2.4 Considered Technologies	13
2.4.1 SPA Framework	13
2.4.2 SSR Framework	13
2.4.3 CSS Framework	13
2.4.4 Deployment	13
2.4.5 Webserver	14
2.4.6 Summary	14
2.5 Additional Tools and Techniques	14
2.6 Legal and Ethical Issues	15
3 System Design	16
3.1 System Design	16
3.2 Identifying UI Components & Use Case Flows	17
3.2.1 Login Page	17
3.2.2 Home	18

3.2.3	Noticeboard	18
3.2.4	Polls	18
3.2.5	Forum	19
3.2.6	Issue Management	19
3.2.7	Profile Page	20
3.2.8	Add-House Page	20
3.3	Summary	21
3.4	Site Structure	21
4	Implementation	22
4.1	Introduction	22
4.2	Nuxt.js	22
4.2.1	Rendering modes	22
4.2.2	Usage	22
4.3	Vue	23
4.3.1	Separation of Concerns	23
4.3.2	Interpolation	25
4.3.3	v-bind	25
4.3.4	v-model	25
4.3.5	v-on	26
4.3.6	v-for	26
4.3.7	v-if	26
4.3.8	props	27
4.3.9	Component Communication	27
4.3.10	Vuex	27
4.3.11	Computed Properties & Methods	28
4.4	Storage Management	28
4.4.1	State Population	29
4.5	Authentication	30
4.5.1	Authorization	31
4.5.2	Summary	31
4.6	Common UI Items	31
4.6.1	Navigation Bar	31
4.6.2	Search Bar & Create Button	32
4.6.3	Language Switcher	33
4.6.4	House Switcher	33
4.7	Noticeboard	34
4.8	Forum	35
4.9	Polls	36
4.10	Issue Log System	38
4.11	Adding a house	38
4.12	Dashboard	39
4.13	Internationalization	39
4.14	Client-side Validation	40
4.15	Error Handling	40
5	Deployment	42
5.1	Applicability of Containers	42
5.2	Docker	42
5.3	Docker Swarm	42

5.4	Microservices	42
5.5	Reverse Proxy	42
6	Test Strategy	43
6.1	Introduction	43
6.2	Testing Frontends	43
6.3	Test Setup	43
6.4	Continuous Integration	43
6.5	Linting	43
7	Evaluation	44
7.1	Project Objectives	44
7.2	Self Evaluation	44
7.3	Project Evaluation	44
7.4	Commercial Applicability	44
7.5	Conclusions	44
7.6	Future Work	44
	Acronyms	45
	Glossary	46
A	User Stories	47
A.0.1	Noticeboard	47
A.0.2	Polls	47
A.0.3	Forum	47
A.0.4	Issue Management	47
A.0.5	Profile	48
A.0.6	Authentication	48
A.0.7	Bilinguality	48
A.0.8	House	48
B	Use Case Flows	49
B.0.1	Noticeboard	49
B.0.2	Polls	49
B.0.3	Forum	50
B.0.4	Issue Management	51
B.0.5	Authentication	52
B.0.6	Bilinguality	52
B.0.7	Profile	53
B.0.8	House	53
C	Additional Figures	54
D	Additional Listings	59
	References	63
	Literature	63
	Online sources	64
	List of figures	64

List of Code Samples	65
List of tables	66
Index	68

Abstract

Chapter 1

Literature Review

1.1 Introduction

As Vue.js was primarily used to develop this project, this chapter aims to illustrate why Vue was created in the first place and how it compares to other frameworks. Finally, it introduces the reader to [Search Engine Optimisation \(SEO\)](#) for [Single Page Application \(SPA\)](#)'s, why it is such a problem to get a high search engine rank and discusses various solutions to this problem.

1.2 Background of Vue.js

Although existing frameworks were already used in order to create [Single Page Application](#), a concept which was discussed as early as 2003 [24] and used as a term for describing webinterfaces with a smooth almost native-like user experience [11], none of these were designed with the purpose of rapidly prototyping UI interfaces. Angular, already widely used by developers and initially created by Google was too big and bloated of a framework to be sensible for small applications. React, a fairly new framework at the time also proved being too complex just like Backbone.js which was used for large-scale enterprise applications. There was something missing: a lightweight framework flexible enough to quickly build prototypes while not being too hard to master. As existing solutions did not seem adequate for this exact purpose [4, p. 10] new frameworks emerged, with Vue being one of them. Its lightweight approach of reactive data-binding and reusable HTML-based components helped fill that niche. Rising in popularity over the years Vue is now utilized for complex enterprise-grade applications and small prototypes alike and has since been adopted by many developers and companies around the world. The most noteworthy of which are: Facebook, Netflix, Adobe, Xiaomi, Alibaba and GitLab [26]. With more than 130,000 stars on GitHub at the time of this writing, Vue is more popular than both React (122,000 stars) and Angular (45,000 stars).

1.3 Comparison to other Frameworks

As there is an excessive amount of libraries and frameworks available for creating web based applications this section will be limited to comparing Vue to the other most popular frameworks React and Angular. The following aspects of these will be evaluated: basic usage, scaling, performance, build size and ease of learning. As some terms are prone to ambiguity further clarification is necessary: basic usage describes how a framework is typically used when integrated into an environment that allows some kind of build process. For example Webpack is most commonly used as a bundler, providing a build process which main purpose it is to consolidate, transform and transpile code for usage in a browser, allowing developers to use next-gen JavaScript syntax that is not yet widely available in browsers among other things. Scaling discusses the capabilities of adding functionality to a framework. Flexibility describes if a framework enforces specific conventions in other words, how opinionated it is.

1.3.1 Vue 2.6.X

Vue.js is considered a very performant JavaScript framework that "makes it easier for developers to create rich, interactive websites" [9]. Instead of directly updating a sites DOM which is an expensive operation Vue utilizes a virtual DOM [17] - the representation of the actual DOM as a JavaScript object [22]. When updating the UI, changes are made to this object which is a far cheaper operation. To get the real DOM in sync an efficient updating function is called, resulting in reduced inefficiency especially in case many nodes have to be updated [22].

Basic Usage

By default, any valid HTML can be used to define the basic structure of Vue components, which is also referred to as a "template" in Vue terminology. Component-scoped data and logic are defined within script tags and are complemented by functional directives such as "v-if" inside the template. Style sheets are defined within style tags, separating style and logic. By optionally using the scoped keyword, CSS is bound to a specific component, reducing the risk of polluting global style sheets. The composition of template, script and style tags is called a "Single File Component", indicating that only a single file is needed to create a functional and styled component. Additionally, props provide a uni-directional way of passing data from a parent component to a child component, whereas events are commonly used to pass data in the opposite direction. "Methods" can be defined to execute repeatable code, **computed properties** "are calculations that will be cached based on their dependencies and will only update when needed" [4] and **watchers** which "watch" developer-defined properties and run a function everytime the property's value changes.

```
1  <template>
2  <div>
3  <h1 v-if="condition" class="custom-title"> Hello {{ user }} </h1>
4  <h1 v-else> "Hello Stranger" </h1>
5  </div>
6  </template>
7
8  </script>
9  export default {
10    data: {
11      return() {
12        user: "Alex"
13      }
14    },
15    methods: { /*execute code*/ },
16    computed: { /*return data when data changes*/ },
17    watch: { /*execute code when data changes*/ }
18  }
19  </script>
20
21  </style scoped>
22  .custom-title {
23    size: 30px;
24  }
25  </style>
```

Listing 1.1: Vue Single File Component

Scaling

Vue applications can easily be scaled up or down in terms of application functionality depending on the requirements. In order to create sophisticated and large applications, three parts have to be incorporated in general: The core library, [routing](#) and [state management](#), of which all are officially provided by Vue as supporting libraries [17]. This typically goes along with bundling tools such as Webpack or Browserify, which make for a much more powerful development environment. In addition, Vue offers an optional CLI generator interface for scaffolding projects, leaving the choice to the developer which building system and plugins to use. Using this interface, sophisticated applications along with routing and state management can be created. If the goal is to add reactive and interactive elements to an existing webpage, Vue can be used by adding a script tag to any valid site where necessary [15, 17]. In doing so, no bundler is needed for code to function but at the same time developers are deprived from being able to use plugins, preprocessors, and various other tools and are most commonly left with a larger bundle size [17].

Performance & Build Size

The source size of Vue with the additional dependencies of [vuex](#) ([state management](#)) and [vue-router](#) is about 30KB [17]. As for performance, Vue is a very performant and efficient framework in terms of startup time, memory allocation and rendering time when compared to other technologies [23]. Early beta versions of Vue 3.0 even reduce memory allocation and rendering time by about 50%.

Flexibility & Learning Curve

Among other things, Single File Components are easy to use when coming from an HTML background, make transitioning existing applications to Vue smoother, do not have a steep learning curve resulting in faster adoption by beginners and can be further enhanced with various preprocessors [17]. As Vue supports various bundlers and build systems while not enforcing specific ways of usage, you could argue that it is less opinionated than some other technologies.

1.3.2 React 16.8.X

Basic Usage

React describes itself as a "library for building user interfaces" [1, p. 2]. By using the word "library" it is implied that less functionality is shipped as opposed to using a traditional framework [1, p. 2]. Like Vue it also utilizes a virtual DOM [1, p. 81]. In React everything is defined in terms of JavaScript, meaning HTML often coupled with CSS are directly embedded into so called render functions. [JavaScript XML \(JSX\)](#), which is essentially a syntax extension to JavaScript with XML-like features [16] is most commonly used (even though not necessary) to define these render functions and is capable of mixing the full power of a programming language with rendering and UI logic [17]. Styling is most commonly achieved by CSS-in-JS solutions provided by additional libraries, effectively consolidating logic and styling in the same place [6].

```

1  class Welcome extends React.Component {
2      render() {
3          return <h1>Hello, {this.props.name}</h1>;
4      }
5  }

```

Listing 1.2: Usage of JSX Render Function

In addition, JSX render functions provide full leverage of JavaScript, including temporary variables and direct references to these and good tooling support (linting, type checking, autocompletion) [16, 17].

Scaling

Like Vue React can either be used with a bundler or added to a single site by using a script tag. React outsources **routing** and **state management** to the community, fragmenting its ecosystem in the process [17]. More than eleven well-known routing libraries are available for React with similar statistics for **state management** and styling which gives developers a vast variety of available options but also makes choosing the right library for a projects requirements a much more tedious task. Reacts cli "create-react-app" works in a similar fashion like Vues but is much more limited: instead of letting the user choose a variety of options, it assumes that a single page application is to be created with always the same dependencies. However, an existing project can be migrated to a more customized environment with a command provided by React.

Performance & Build Size

The React source itself has about 4.7KB gzipped which makes sense, given that React advertises itself as a library rather than a framework. However, including React DOM (34KB), **routing** with react-router (6.9KB) and **state management** with redux (2.4KB) the total size grows considerably to 48KB. Performance-wise, React is also very efficient when compared to other technologies, in fact, very similar to Vue [23]. The reason for this, is mainly due to the usage of a virtual DOM.

Flexibility & Learning Curve

To take advantage of online resources and documentation **JSX** is almost a requirement. As it is an extension to JavaScript, developers who are familiar with this programming language can easily learn the additional syntax [16]. Without prior knowledge of JavaScript however, the learning curve rises considerably. At the same time, the availability of hundreds, perhaps even thousands of supporting libraries makes React very flexible, given that its core is so slim and additional functionality can easily be added by leveraging these.

1.3.3 Angular 7.2.X

Basic Usage

Even though JavaScript could be used, Angular essentially requires the usage of TypeScript, a typed superset of JavaScript [28]. As opposed to JavaScript, TypeScript comes with static type

checking which naturally makes applications less prone to errors [12] and therefore is often used within very large corporate projects. Like Vue Angular also uses a component-based approach for composing user interfaces, but rather than using a single file, multiple files for HTML, CSS and JavaScript logic are typically created.

```
1  //app.component.html
2  <button (click)="show = !show">{{show ? 'hide' : 'show'}}</button> show = {{
    show}}
3  <br>
4  <div *ngIf="show; else elseBlock">Text to show</div>
5  <ng-template #elseBlock>Alternate text while primary text is hidden</
    ng-template>
6
7  // app.component.ts
8  import { Component } from '@angular/core';
9
10 @Component({
11   selector: 'app-root',
12   templateUrl: './app.component.html',
13   styleUrls: ['./app.component.css']
14 })
15
16 export class SampleComponent {
17   show: boolean = true;
18 }
```

Listing 1.3: Angular Basic Usage Example

Scaling

Designed with the purpose of building large and complex applications Angulars API exposes a lot of functionality which is most commonly only necessary for exactly these types of applications. It includes everything from routing, state management, http calls to complete testing suites, however, Angular can also be added to existing sites using a script tag, only providing core functionality.

Performance & Build Size

Angular applications built with the angular project scaffolding interface angular-cli have around 65KB gzipped, double the space as the other two frameworks. As for performance, Angular is also a performant framework [23] but becomes slow under certain circumstances: For instance if a project utilizes a lot of watchers and data in the scope changes, all watchers are re-evaluated again [17].

Flexibility & Learning Curve

Angulars very big ecosystem and API provide most needed functionality out of the box but at the same time limit its flexibility. What is more, by providing pre-defined ways of interacting with Angular, it inherently becomes opinionated and more difficult to master [17].

1.3.4 Conclusion

All three frameworks solve similar problems but are used in different ways. Vues Single File Components consolidate logic, styles and HTML in the same place, React can and is most often used in a very resemblant way if used with a CSS-in-JS approach, whereas the "Angular way" is to split these parts into separate files.

Applications built with React or Vue can be easily scaled up or down depending on the application-requirements. Large applications typically need the core library, routing and state management. While all of these are officially provided by the Vue core team for Vue applications, React leaves routing and state management to the community [17]. Vue and React both offer an optional CLI generator interface to scaffold projects, however, Vue offers more options, leaving the choice to the developer which building system and plugins to use. React's more limiting approach with create-react-app makes it easier to start a project as it only needs a single dependency but limits the user to a given setup, which can however, be moved to a more customized environment with little effort whereas Angular is completely set up for the development of very large projects and provides most functionality out of the box, scaling up is therefore not a big problem because developers can rely on most of the functionality already existing.

All three frameworks can be scaled down by adding script tags to any site [15, 17] which however removes the powerful building layer.

Vue and React both are similar in terms of runtime performance, are based on a virtual DOM and are used for similar use cases while Angular is a much more heavy-weight framework regarding performance and size.

JSX as well as TypeScript are additional learning steps and can lead to decreased productivity in smaller projects. Being a dynamic language, JavaScript's ability to address quickly changing requirements makes it especially suitable for rapid prototyping [12, p. 72]. By using very HTML-like components Vue is often easier to master than the other two frameworks as most developers have at least basic knowledge of HTML. Nonetheless, Vue makes it possible to use TypeScript if the need arises, paving the way for bigger projects [18]. Angular is not very flexible in the sense that users can pick whatever supporting library they deem best and is said to be an opinionated framework which enforces the "Angular way". Such frameworks are "pragmatic, with a strong sense of direction" [2] often forcing very specific conventions upon its users, effectively restricting what a developer can do with a framework. This also means there is a steeper learning curve but once overcome, can lead to increased productivity [17].

The diagram shown in [Figure 1.1](#) does by no means reflect the properties of the given framework / library 100% accurately as some of them are somewhat subjective (e.g flexibility) but shall rather illustrate the strengths and weaknesses in relation to another when used in a typical manner.

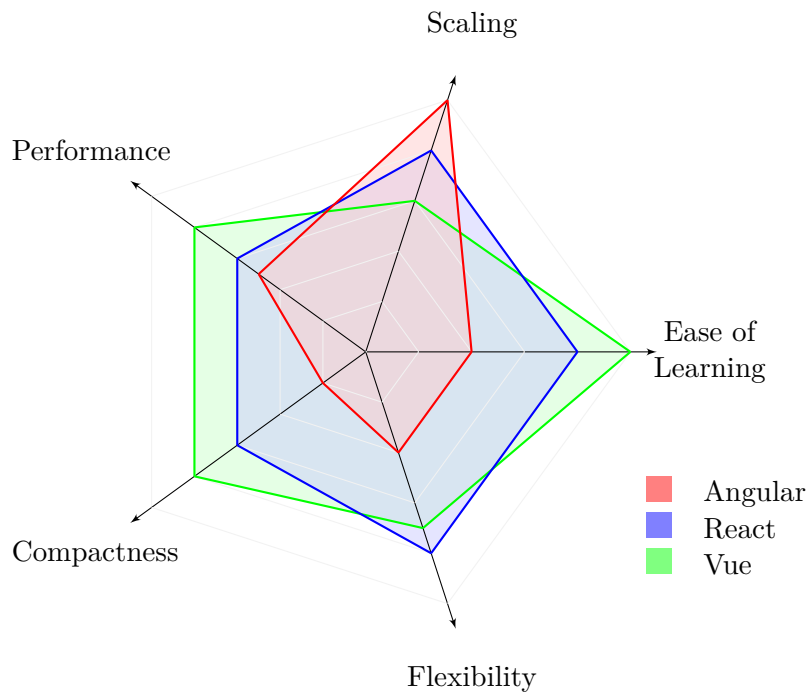


Figure 1.1: Comparison of Vue, React and Angular

1.4 SEO for Single Page Applications

1.4.1 Introduction

There is no doubt that competing companies all over the world want to be found first in the biggest market of all: the internet. As 67.60% of organic clicks are accounted by the first five pages returned by the SERP [7], these same companies are highly incentivized to invest into an improved search engine rank which was found to have a positive impact on an organizations performance [13]. Therefore, the process of Search Engine Optimisation has become ubiquitous in our modern era as it promises to attract more users to a certain website [7]. In other words: more users mean there is a greater probability of conversions, thus generating more revenue.

Unfortunately, successfully implementing SEO is a very delicate matter. In reality it is a set of techniques geared towards software rather than humans. These programs are commonly referred to as crawlers or bots which traverse the Web by exploiting the Web's hyperlinked structure and add their findings to the search engine's index [5, 10]. They start on a websites root page and try to find any available links on that site [5]. The process of choosing which links to follow, itself is dependant on complex algorithms as different crawlers will have to make different decisions [10]. For example a crawler trying to index the web as comprehensively as possible will have different underlying instructions than one that tries to find product reviews [10]. After being added to the index, a site's content is destructured and with additional metrics such as speed, size of images and mobile-friendliness its search engine rank is calculated [5].

Search Engine Optimisation helps crawlers better understand a site's content and "serves as a solution to figure out the nature of each webpage and determine how it can be made worthwhile to the users" [7].

1.4.2 The Problem

In general there are two ways to improve any type of site's rank: on-site optimisation and off-site optimisation [7]. The first pertains to techniques that can be implemented on a site itself, such as efficiency of a webserver, compression of data, deferred loading of images, keywords or the actual content, whereas the latter is a set of techniques which primarily drive more traffic to a certain site, such as social media marketing or the creation of backlinks [7].

Improving a **Single Page Application's** ranking score proves difficult when using on-site optimisation techniques. When Google first started crawling websites in 1998 JavaScript was not a big problem as it was not widely used, so it simply was not executed [20]. The rapidly changing landscape of webdesign however, demanded a more sophisticated approach. Things started to change drastically in 2014 when Google's crawler began executing JavaScript [20]. However, as websites increasingly rely on AJAX [3, p. 97], more specifically the deferred loading of resources, crawlers faced with a problem. The research of Brunelle et al. [3, p. 97] illustrates it as following:

When a crawler fetches a web page (1), it waits for the page and all of the embedded resources to load to consider it fully rendered. At this point, the crawler preserves (2) all of the content on the web page (A). After the page has loaded, the page can request further resources (3), even without user interaction. The resource is then returned to the web page to be rendered (4) producing the final intended result (Ab). Because the crawler preserved the page prior to the page being fully loaded, the secondary content is not preserved and the archived version of the web page is not complete.

This can be seen as one of the biggest disadvantages for **SPA's** as they heavily rely on AJAX and client-side rendering. If crawlers are not aware of their contents they will in turn suffer from a decreased search engine rank.

1.4.3 Possible Solutions

The problem is to be solved by offering crawlers a fully rendered page instead of having them load additional resources. This concept is called **Server Side Rendering (SSR)** in broad terms. Depending on a web application's usage several variations are feasible:

Prerendering

The first is to implement prerendering: a process in which static HTML-files are created for specific routes at build time [19]. This technique is especially useful for sites with content that does not change very often. Typical scenarios would be to create static versions of marketing or about sites [19]. One of the main advantages is a rather simple setup process since creating a static version happens only once during building. At the same time, it becomes obvious that this solution is not very suitable for pages with rapidly changing content - rebuilding a project after every data change is not very efficient after all.

Server Side Rendering

A more sensible approach for sites with often changing content is [Server Side Rendering \(SSR\)](#). As opposed to prerendering, [SSR](#) is a lot more sophisticated and complex to set up as it requires a tight coupling to the used framework. Upon every request by a client, the framework is responsible for creating a static version of the site which reflects the current available data. In technical terms, a static HTML site is rendered on the server by the framework, sent to the browser and hydrated "into a fully interactive app on the client" [19].

Dynamic Rendering

Another possible solution which is still relevant today is to "create a static representation of your web site/application and direct crawlers to use it" and was identified by Fink et al. in 2014 [5, p. 270]. This technique has been given the name "Dynamic Rendering" by Google in 2018 [21]. It differs from prerendering and SSR in one key aspect: real users and crawlers are served alternative content, hence the term "dynamic". When a crawler requests a site, an intermediate service transforms HTML and JavaScript to a static, prerendered version. This intermediate step allows for the creation of prerendered sites whenever needed - it builds upon prerendering but removes its build time constraint.

Evaluation

Prerendering is fairly easy to implement but is not very suitable for web applications with content that changes rapidly. As prerendering happens at build time, a project would have to be built every time content changes - an automated environment could be set up which builds the project after a given time interval, in this case however, it is noteworthy to mention that building is a rather expensive operation which takes longer the bigger the project is and deployment could result in downtime which accumulates over time. A better solution would be to use dynamic rendering, which builds upon prerendering but is run as a separate service, therefore coming with the benefits but not the disadvantages of normal prerendering. The most sophisticated approach and most complex to implement is [SSR](#) which requires intimate knowledge of a framework's internals. However, there are solutions that take away some of that overhead for specific frameworks. Examples would be "Next.js" for React and "Nuxt.js" for Vue but in general, the previous statements apply.

Chapter 2

Project Planning

2.1 Introduction

The project was to develop a front-end to an existing system and had to be planned accordingly. This chapter will give an overview about which parts of the system already exist and identifies the requirements and potential solutions for the frontend. Additionally, it depicts considered technologies and techniques and evaluates the most appropriate development stack.

The main idea is to give tenants, proprietors and facility management a way to communicate over a single point of access. In its first version, they should be able to discuss topics they care about, easily make announcements, submit and track the state of maintenance issues and create polls in order to provide an easy way for decision-making. The requirements are not "hard" requirements in a sense that a customer requirements document or something similar exists, rather they are written as informal user stories. It is the developer's responsibility to develop feasible use cases and flows. It is also noteworthy to mention that this project will initially be released in Austria.

2.2 Provided Services

A Database, Exchange Email Server and API are provided by the project partner. The first is a PostgreSQL instance, the latter an API built on top of Django and Django Rest Framework. These two technologies provide the backend of the system as a whole. The API is documented with Postman, describing each endpoint in terms of what type of data has to be sent and what data will be returned. Apart from making these compatible with a containerized environment (see [Chapter 5](#)), no further action has to be taken in order to use these services.

2.3 Requirements

As mentioned in the introduction, several functional requirements exist. Furthermore, there are some non-functional requirements for the project which have to be taken into account. [Table 2.1](#) depicts these general requirements.

Functional Requirements	Non-Functional Requirements
Digital Noticeboard	High Flexibility
Polls	Easy to maintain
Forum	Easy way of creating apps for mobile phones from frontend
Issue Management	Low learning curve for future developers
Bilinguality	

Table 2.1: Project Requirements

These very high functional requirements can be further broken down into user stories. The most important of which are illustrated along with additional information in the next few sections. For a full list of user stories please see [Appendix A](#). The word "user" is used for referring to every user-group of the system. The specific user-groups are: Tenant, Proprietor, Facility Management.

2.3.1 Digital Noticeboard

Description	A unidirectional way of communication between house parties
User Story	As a user I want to create noticeboard entries so other users can see my messages
Example	Announcing a get-together everyone is invited to

2.3.2 Polls

Description	A method for decision-making
User Story	As a proprietor I want to create polls other proprietors can vote on so that we as a group can make decisions
Example	A facade that has to be repainted with a new color. To get an initial idea which color it should be, Proprietors can create polls and let other proprietors choose between given options
Notes	Polls should not be editable so proprietors cannot change their contents after someone voted

2.3.3 Forum

Description	A way of discussing topics related to a specific house
User Story	As a user I want to ask questions so other users can answer them
Example	A tenant wants to know when garbage cans are emptied

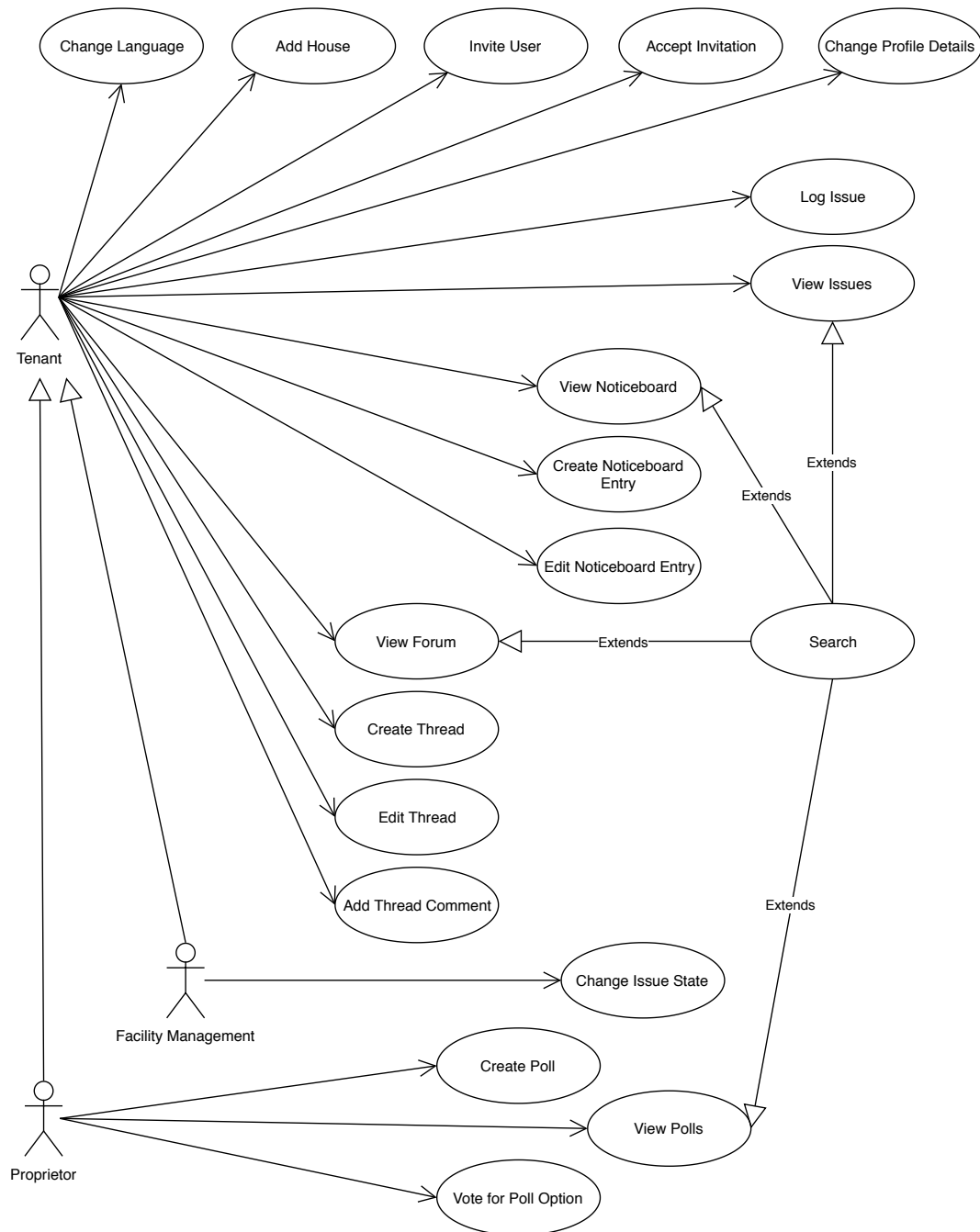
2.3.4 Issue Management

Description	A way of submitting and track the status of maintenance related issues
User Story	As a user I want to submit an issue so that it can be taken care of
Example	A user has a water leak and submits an issue, the facility management can access these issues and take action

2.3.5 Bilinguality

Description	Changing the language
User Story	As a user I want to change the language of the application so I can understand the content
Notes	It is important that users are able to choose between at least two languages. German is the most obvious choice as it is Austria's primarily spoken language. English, the second most well known language in Austria is obvious as well

From these user stories an initial high-level use-case diagram as illustrated in [Figure 2.1](#) can be derived. Since the initial project requirements are quite vague and an iterative approach is followed during development, it makes sense to defer the specification of these use cases to a more appropriate point in development which can be found in [Section 3.2](#). In addition, this is consistent with the principle of "decide as late as possible" of lean development practices.



* All use cases require/include authentication and authorization which are omitted for brevity

Figure 2.1: Proprietors Assembly Use Cases

2.4 Considered Technologies

2.4.1 SPA Framework

To meet both functional and non-functional requirements a [Single Page Application \(SPA\)](#) is a potential solution. They provide an interactive, smooth native-like user interface and can easily be transformed to a [Progressive Web App \(PWA\)](#) if the need arises. PWA's are a fairly new type of application that use traditional web-browser technology but can be run almost as if they are native applications. As illustrated in [Figure 1.1](#), an especially appropriate framework for developing this project can be found in Vue, which perfectly suits the needs of the non-functional requirements: as opposed to React or Angular, its relatively high flexibility in addition to its shallow learning curve allow for a rapid development process.

2.4.2 SSR Framework

Although the project's requirements do not implicate the necessity of implementing any type of [Server Side Rendering \(SSR\)](#), it is possible that rapidly changing content such as publicly available blog posts with comments will be added in a future release. [Section 1.4.3](#) identifies three ways of rendering SPA's on the server side and explains in great detail why this is important. Developing the project from the start while keeping this information in mind could reduce long-term complexity.

Server Side Rendering as defined in [Section 1.4.3](#) would be the most viable option as it comes with the biggest advantages. It can be utilized in one of two ways: By using vue as a framework and the supporting library "vue-server-renderer" or by using an additional framework. The first way requires the developer to configure everything themselves whereas the latter provides pre-configured SSR functionality. One of these frameworks is Nuxt.js which is the most popular SSR framework within the vue community. In addition, it configures automatic routing and state management, which are described in more detail in chapter (Not yet written) .

2.4.3 CSS Framework

CSS frameworks can drastically decrease developing time for frontend projects as they provide predefined tools and components for creating user interfaces. There are various CSS frameworks available which differ greatly in design: in addition to the widely used Bootstrap and Foundation frameworks which have a very basic and familiar design, there are smaller ones which provide different design elements which could help distinguish the project from others. Such frameworks are "Bulma" or "Materialize". Bulma comes with a very distinct design, that does not seem to be used as widely on the web. Furthermore, the vue library "Buefy" builds on top of Bulma's CSS and provides predefined vue components to use. This is very important as it makes it much easier and quicker to build user interfaces.

2.4.4 Deployment

There are various techniques of deploying a technology stack to a production ready environment ranging from installing a webserver, database and additional services directly onto a server (monolith) to using virtual machines for each individual service. A much better and future-proof

approach is the usage of "containers": Each service is encapsulated in its own linux environment with all needed dependencies for it to run. It is important to note that some services are bundled together in the same container if it makes sense: E.g a backend framework which defines REST endpoints and a webserver which exposes these endpoints. A more appropriate word in this case is "microservice". The most popular containerization technology is Docker which gives developers the ability to write "compose" files that define a composition of services. One of the biggest advantages of Docker is that the same services can be run in a development as well as production environment, greatly reducing overhead.

2.4.5 Webserver

The two main webserver technologies are NGINX and Apache httpd. According to various statistics and benchmarks, NGINX can handle a vastly greater number of concurrent connections and can serve static content more efficiently, whereas httpd is better suited for handling dynamic content (e.g. PHP-generated sites). These two technologies however, are often used in tandem: NGINX used as a reverse proxy handling a great number of connections and httpd used to serve dynamic content. With the finished [Single Page Application \(SPA\)](#) consisting only of static content NGINX clearly is more suitable for this project.

2.4.6 Summary

Vue is the perfect framework for this project due to its flexibility and ease of learning. Nuxt will be used to lay the groundwork for [SSR](#). The design of the frontend will be based on Bulma and Buefy giving the project a distinct and distinguishable look. Docker will provide a way of easily deploying the application on any server and development environment. An NGINX-based webserver is used to serve the static frontend.

2.5 Additional Tools and Techniques

Methodology

Kanban, a popular agile methodology is used as a visual management tool during development. The concept revolves around the idea of a "Kanban Board" on which cards are moved around depending on the development step they are in. The backlog consists of user stories which are mapped to cards on the board. Typically, there are four rows on a board, which can vary depending on the project: "To Do" - consisting of user stories in the backlog, "In Progress" - currently worked on user stories, "Testing" - user stories which are currently being tested and "Done" - which holds every user story that was successfully implemented and tested.

VCS

Two services are used: GitHub as a [Version Control System \(VCS\)](#) for the development of the [SPA](#), also providing a Kanban Board and GitLab as a meta project [VCS](#) which consolidates frontend (git submodule), backend (git submodule) and Dockerfiles. This way, the whole system's source code can be easily containerized with docker. Furthermore, GitLab also provides a container registry. The registry exposes an interface for uploading and downloading container

images. This is especially useful for creating images on a local machine, uploading them to the registry and downloading them again onto a remote server where they should be deployed. Doing it this way completely removes the need of storing the source code for the whole project on the remote server and building it from source. Instead, the version number of the used image is increased in a compose file and the container gets updated accordingly when a redeployment is initiated.

Git Flow

Git Flow is a **VCS** methodology which is based around "features". In its most basic structure there is one master and develop branch accompanied by multiple feature branches (usually one feature per branch). The master branch should always be kept in a production-ready state, whereas the develop branch contains the actively developed project-source in which every finished feature branch is merged into. Additionally there are hotfix and release branches which support the others. Git Flow adds an abstraction layer to the development flow and greatly helps building applications by an iterative approach.

Circle CI

Automated testing is important when developing applications. It can reassure the developer that nothing will break and result in unexpected errors. Circle CI is a continuous integration platform which automatically tests code when certain rules are met. It works very well when used in addition to Git Flow and Docker: it can be set up to create a container with the projects source, builds it and runs its tests upon every change made to the develop or master branch or a pull request. Circle CI notifies its users if any step during this process failed preventing faulty code to ever reach a production state.

2.6 Legal and Ethical Issues

As identified in the Technical Report (see Appendix ...) there are two main areas of concern regarding ethical and legal issues: privacy and visual impairment. In order to use the service, users have to provide information about them, including name, email and home address which could lead to GDPR related issues. Providing access to the service to visually impaired users also poses an ethical issue as adaptations have to be carefully thought about.

Chapter 3

System Design

3.1 System Design

In the previous chapter the requirements of the project were illustrated. From these an initial high level overview of the system can be generated as depicted in [Figure 3.1](#).

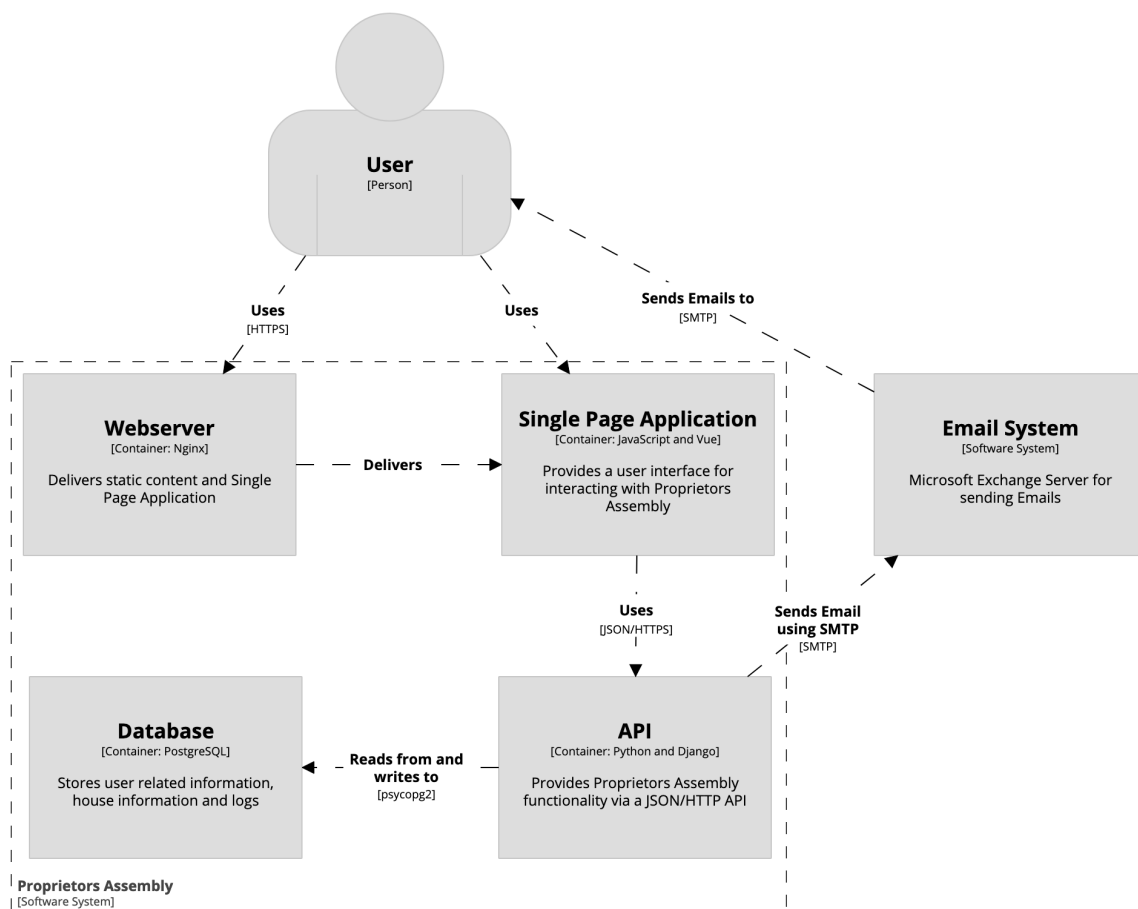


Figure 3.1: System Design

In its basic structure, Proprietors Assembly as a software system consists of an NGINX web-server, a Vue [Single Page Application](#), a Django API and a PostgreSQL database. When a user visits the domain <https://hausversammlung.at>, the webserver will serve the static frontend which communicates with the API. Depending on the use-case, the API will retrieve data from or add to the Database. An additional third-party Exchange server is used to send emails regarding authentication and house details. The diagram intentionally omits certain parts of the system for brevity. One of these is a reverse-proxy service which is the first point of contact from a user's viewpoint. A reverse proxy in this case routes between containers which are

hosted on the same machine, accessible through unique ports. This topic is discussed in depth in [Chapter 5](#).

3.2 Identifying UI Components & Use Case Flows

Interactive components of the frontend are strongly dependant on the required functionality. This section aims to define what is minimally needed on a certain page and to translate functional requirements into identifiable UI elements. This makes it possible to identify elements that are used in multiple places and clearly shows the structure of the frontend. Furthermore, as Vue uses a component-based approach, these elements serve as a starting point for defining which vue-components are required.

Additionally, as mentioned in [Section 2.1](#) it is the developer's responsibility to derive all use cases and flows from the requirements. Since use case flows are very much entangled with how a user interacts with a system and this section is about interactable UI elements, this makes it the most appropriate place to match these to another. The approach taken is as following: 1) the thought process of what a site should provide is laid out 2) use case flows are derived and UI elements are mapped to a flow 3) all UI elements are noted to analyse which occur multiple times. **Note:** as too many use case flows would quickly result in a bloated document, only the most important flows per page/category will be shown, additional flows are depicted in [Appendix B](#).

3.2.1 Login Page

The Login / Register page handles user authentication. A user is required to log into an existing account or create a new one before they can use the system. Traditionally, this is done with a plain HTML-Form. It is sensible to create separate forms for login and register and let the user decide which form to use depending on their needs. Additionally, a forgot-password form should provide a uniform way of requesting a password change.

The user opens the login page or is redirected automatically if not logged in
The user provides their email address and password to a form
The user clicks "Login"
The system checks for appropriate authentication details
The user is redirected to the Home page

Table 3.1: Use Case Flow: Login

The user opens the login page or is redirected automatically if not logged in
The user clicks "Register"
The user provides their email address password, firstname and surname to a form
The user clicks "Register"
The system checks the correctness of the input
A new user is added to the system
The user is redirected to the Home page

Table 3.2: Use Case Flow: Register

Identified Elements: HTML forms for login, register and password-reset

3.2.2 Home

On the home site various data sources should be composed into a consolidated view - a dashboard. It shows the most recent forum items, polls and noticeboard entries in addition to the current number of issues, house parties and squaremeters. A user should be able to switch between a list of houses they are associated with. A navigation bar should contain links to all subpages: Home, Noticeboard, Polls, Forum, Issue Management and the Profile.

The user logs in or clicks the Home link in the navigation bar
The user is redirected to the Home page
The dashboard is shown to the user

Table 3.3: Use Case Flow: View Dashboard

Identified Elements: Noticeboard Item, Poll Item, Forum Item, House Switcher, Navigation Bar

3.2.3 Noticeboard

The Noticeboard page contains a list of all noticeboard entries associated with a house. An input field should let the user search for a specific item. A navigation bar as described for the previous page should allow for page changes. Finally, a button that links to a page where noticeboard entries can be created should exist. If the user is the owner of an entry an edit button should be shown. It is sensible to consolidate creating and editing an item on the same site by using a single form which is either initially empty or populated with the existing data should be edited. Creating a noticeboard requires a title and content.

The user opens the noticeboard page by using the navigation bar
The user clicks a "create" button
The user fills a form with a title and content
The system checks for appropriate authorization
The system adds a new entry to the database
The user is redirected to the main noticeboard page

Table 3.4: Use Case Flow: Create Noticeboard

Identified Elements: Noticeboard Item List, Noticeboard Item, Search Bar, Navigation Bar, Create Button, Create-Edit Noticeboard Entry Form

3.2.4 Polls

On the Polls page every poll associated with a house is displayed. Users should be provided with a way of searching and filtering these items. A navigation bar should allow for quick page

changes. In order to create a poll, users should be shown a button which takes them to a subpage where they can create new polls or edit existing ones. To create a poll, a title and at least two options have to be provided by the user.

The proprietor opens the polls page by using the navigation bar
The proprietor clicks a "create" button
The proprietor fills a form with a title and poll options
The proprietor clicks "Save"
The system checks for appropriate authorization
The system adds a new entry to the database
The proprietor is redirected to the main Polls page

Table 3.5: Use Case Flow: Create Poll

Identified Items: Poll Item List, Poll Item, Search Bar, Navigation Bar, Create Button, Create-Edit Poll Form

3.2.5 Forum

The Forum page shows users a list of all thread items of a house which are searchable. A subpage which is accessible by clicking a button should contain a single form for creating and editing items. A thread needs to have a title, category and content. Additionally, a navigation bar links to the other pages.

The user opens the Forum page by using the navigation bar
The user clicks a "create" button
The user fills a form with a title, question and category
The user clicks "Save"
The system checks for appropriate authorization
The system adds a new entry to the database
The user is redirected to the main Forum page

Table 3.6: Use Case Flow: Create Thread

Identified Items: Thread Item List, Thread Item, Search Bar, Navigation Bar, Create Button, Create-Edit Thread Form

3.2.6 Issue Management

Similar to the other pages, the issue management page should have a navigation bar, a search bar and a button which links to a subpage where items can be created or edited. Its main purpose is to show a list of all house-related issues. An issue has a reference, date, type, location, title and status.

The user opens the Issues page by using the navigation bar
The user clicks a "create" button
The user fills a form with a title, category, location and comment
The user clicks "Save"
The system checks for appropriate authorization
The system adds a new entry to the database
The user is redirected to the main Issues page

Table 3.7: Use Case Flow: Log Issue

Identified Items: Issue Item List, Issue Item, Search Bar, Navigation Bar, Create Button, Create-Edit Issue Form

3.2.7 Profile Page

The profile page provides a way of making changes to a user's account. Initially, they should be able to change their name. Additionally, they should be able to switch languages or house and to invite additional users to their currently active house.

The user opens the Profile page by using the navigation bar
The user changes their names
The user clicks "Save"
The system changes the user's name
The new name is updated throughout the application

Table 3.8: Use Case Flow: Change Profile Details

Identified Elements: Language Switcher, Invite Form, House Switcher

3.2.8 Add-House Page

Users need a way of creating a new house or adding an existing house to their account by using an invitation link. To create a new house a form should be provided which takes the address, total squaremeters and role of the user as input variables. An invitation should be accepted by providing an invitation id and the intended role of the user. After creating or accepting an invitation users should be given the possibility to invite additional users to their newly added house.

The user clicks "Add House" in the navigation bar
The user is redirected to a page where they can add houses
The user is presented with two options: Accept Invitation OR Create new House
The user chooses "Accept Invitation"
The user fills a form with invitation id and user role
The user clicks "Add"
The system checks if the user is eligible to be added to this house
The user is redirected to a page where they can invite other users to their house

Table 3.9: Use Case Flow: Accept Invitation

Identified Elements: Create House Form, Accept Invite Form, Invite Form

3.3 Summary

Some elements are very distinct and are only used once whereas others are used across multiple pages. One element that stands out and is used in almost every single page is the "Navigation Bar". It would be very sensible to not only create a reusable component of this element but to also include it in some kind of base-layout that can be used as a starting point when creating sub-pages. Some other widely used elements are the "Search Bar" and "Create Button". Elements which might not be used very often but reused nevertheless are the "Language Switcher" and "House Switcher". The various forms, lists and items mentioned previously are used only once most of the time and therefore not reused, however, it still makes sense to create reusable Vue components for these to make site composition cleaner and easier.

3.4 Site Structure

Another important aspect of frontend-applications is the question of how sites are related to each other. This concept can best be described as a sitemap. Well designed sitemaps which are often reflected as links in a navigation bar not only give users a clear understanding of the applications structure but can also be turned into machine-readable files which can be accessed by crawlers. From the previous defined pages it is possible to deduce the complete structure of the frontend-application as depicted in [Figure 3.2](#).

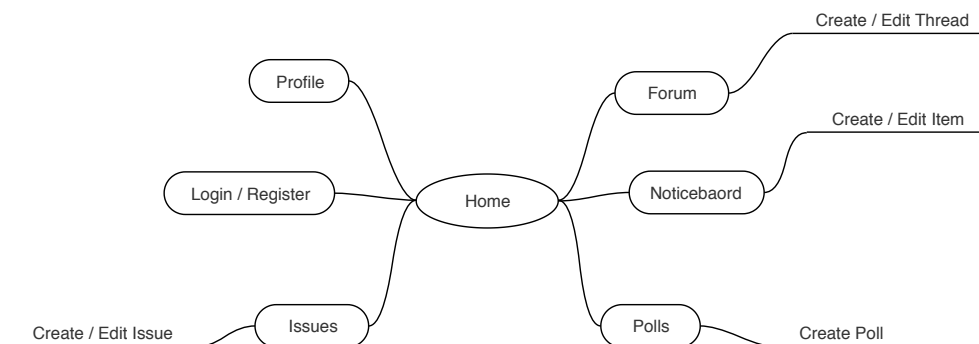


Figure 3.2: Sitemap

Chapter 4

Implementation

4.1 Introduction

Based on the evaluation carried out in [Section 2.4](#) the [Single Page Application](#) was implemented with Vue and Nuxt. To give the reader a better understanding of how the application was built, the first few sections of this chapter will discuss which additional tools and functionalities Nuxt provides, how Vue is used and how [Separation of Concerns \(SoC\)](#) is achieved. Later sections which describe the specific implementation of UI elements do that from a functionality-based perspective and do not go into the specifics of CSS classes used, as these are described in depth in the Bulma documentation. It is enough for the reader to have a basic understanding of HTML and CSS.

4.2 Nuxt.js

Nuxt's primary use case is to provide [Server Side Rendering](#) for Vue-based applications. It also advertises itself as a framework which "presets all the configuration needed to make your development of a Vue.js application enjoyable" [14]. While this blurry description is not very meaningful on its own, it indicates that Nuxt pre-configures certain parts of a Vue application. The most important of which are: [Server Side Rendering](#), automatic setup of routing and store management.

4.2.1 Rendering modes

Nuxt offers three rendering modes: [SSR](#), "Prerendering" and "Single Page Application". For descriptions of the first two please see [Section 1.4.3](#). The "Single Page Application" mode does not offer server-rendering capabilities while still pre-configuring routing and state management. This mode is especially useful for applications that require users to be authenticated before being able to use them. The reason why is simple: [Server Side Rendering](#) can only improve the SEO ranking of a page if crawlers are able to look at the full contents of it. Fully implementing [SSR](#) for an application that only shows a login form would therefore be pointless.

Then why should this project use Nuxt, which is primarily used for [SSR](#) if the application cannot even profit from it because authentication prevents crawlers from seeing anything but a login form? The answer is because 1) state management and routing are auto-configured as well and 2) because in the future, the frontend might expose parts of the system in a publicly accessible way without the need of authentication. By using Nuxt from the beginning, the transition to this is much easier.

4.2.2 Usage

When using Nuxt, different folders contain different parts of the frontend application. Behind the scenes, Webpack - one of the most popular bundler tools consolidates, minifies, transforms

Folder	Description
assets	contains un-compiled assets such as images, global css or fonts
components	contains reusable Vue Single File Components
layouts	contains Vue files that define the applications theme
middleware	defines files which are run before a route changes, e.g check if user is authenticated before redirecting
pages	contains the pages of the application, Nuxt automatically creates routes corresponding to a page's name
plugins	contains plain JavaScript files which are run before instantiating the application, e.g used for inject adding an internationalization plugin
static	files in this directory are directly mapped to server root and are not compiled or renamed in any way, e.g robots.txt
store	contains Vuex store files, every file added in this directory is automatically mapped to the global store
nuxt.config.js	contains the Nuxt configuration for the application, e.g which plugins and middleware to use
package.json	is not Nuxt specific but added by npm and contains application dependencies and scripts

Table 4.1: Nuxt Folder Structure

and transpiles the code of these folders into browser-readable plain JavaScript code.

Typically, functional components are defined inside the components folder. These components might include other components in order to compose more complex architectures. Pages and layouts are also nothing more than Vue components, however, they are used in very specific ways. The file structure of the pages folder is automatically read by Nuxt and routes are generated from it. For example the file `pages/index.vue` is shown when a user visits hausversammlung.at, whereas the file `pages/issues/index.vue` translates to the url hausversammlung.at/issues/. Layouts are used to change the look and feel of the application. For example it is possible to create a layout file which includes a navigation bar on the top of the page. Nuxt then allows developers to set a layout property inside a page which corresponds to a layout file, resulting in easily reusable themes.

4.3 Vue

This section aims to extend the basic description of Vue's usage as described in [Section 1.3.1](#) so that code samples of this chapter are more easily understood. For the sake of brevity, some parts will be omitted if they are not crucially relevant (e.g transitions).

4.3.1 Separation of Concerns

A very important aspect of modern software architectures is [Separation of Concerns \(SoC\)](#) which is often regarded to as principles of modularization of code and object-oriented design [8]. In the case of traditional web design HTML, CSS and JavaScript are often physically separated (separation of file types) leading to increased maintainability of code, a less tightly coupled

system and decreased probability of violating DRY principles. Furthermore, it helps building complex layered systems which can be actively worked on by multiple developers.

Vue however, takes different approaches on SoC depending on the way it is used in a project: either Vue.js is added to existing sites by using a standard HTML script tag or by using a complete bundling tool like Webpack or Browserify to add it as a dependency. Using Vue by adding a script tag has some disadvantages: it only allows developers to target a specific HTML container element on the page Vue is embedded in, and also brings certain disadvantages such as no CSS support inside the Vue component itself and no build steps which requires the developer to fall back to plain HTML and ES5 JavaScript rather than using various preprocessors. [Listing D.7](#) illustrates such an example.

The more preferable approach would be to use a so called “Single File Component”, a file with a “.vue” extension. These files however have to be compiled with build tools like Webpack or Browserify to extract the corresponding concerns. A sample file is depicted in [Listing 4.1](#).

```
1  <template>
2    //HTML
3    <p>{{ text }}</p>
4  </template>
5
6  <script>
7    //JS
8  export default {
9    data() {
10      return {
11        text: 'Hello World!'
12      }
13    }
14  }
15 </script>
16
17 <style scoped>
18  //CSS
19  p {
20    color: red;
21  }
22 </style>
```

Listing 4.1: Vue Single File Component

By defining .vue files developers can utilize syntax highlighting, linting and component-scoped CSS. Furthermore preprocessors like Pug, Babel or SASS can be used to further improve the development process. This however comes with an important tradeoff: as Single File Components rely on the use of build tools, they are not very suitable projects which are not completely vue-based

Clearly HTML, CSS and JavaScript are not separated into their own respective files even when using Single File Components which leaves us with the question of how Separation of Concerns is achieved. As stated in the Vue Documentation:

“... separation of concerns is not equal to separation of file types. In modern UI development, we have found that instead of dividing the codebase into three huge layers that interweave with one another, it makes much more sense to divide them into loosely-coupled components and compose them. Inside a component,

its template, logic and styles are inherently coupled, and collocating them actually makes the component more cohesive and maintainable [27].”

Instead of creating different files in which logic, presentation and design are separated, Vue consolidates these parts into a single file, hence the name “Single File Component”, making the code more maintainable and comprehensible. It is important to note that throughout the project, Single File Components are used.

4.3.2 Interpolation

Interpolation in Vue is the process of substituting a "placeholder" inside HTML with a real value. This special syntax is also referred to as "Mustache" syntax. Listing 4.2 depicts a simple example in which the mustache "msg" tag will be replaced with the corresponding property of the component's data object. Furthermore, when the value changes, Vue automatically takes care of updating the rendered element.

```
1 <span> Noticeboard-Message: {{ msg }} </span>
```

Listing 4.2: Interpolation Example

4.3.3 v-bind

Due to technical restrictions the mustache syntax cannot be used for HTML attributes. Instead a so called "Vue directive" is utilized. Directives are way of telling Vue that a function should be applied to a specific DOM element. In the case of "v-bind" it is to bind data (value) to an attribute (key).

```
1 //long
2 <a v-bind:href="url">dynamic link</a>
3 //shorthand
4 <a :href="url">dynamic link</a>
```

Listing 4.3: v-bind Example

Common examples are href attributes or dynamically bound css classes. Usually, instead of writing "v-bind:" it's shorthand is used which is just a single colon. It is important to note that this type of interpolation only works in a unidirectional way from the data object of a component to its placeholder.

4.3.4 v-model

For elements where user input is crucial, the "v-model" directive is used. It allows for user input to change the data object's properties. In Vue terms this is referred to as "two-way data binding". Listing 4.7 depicts an example use case: a user inputs data into an input field, whatever they type will be displayed below it.

```
1 <input v-model="message">
2 <p>Message of user: {{ message }}</p>
```

Listing 4.4: v-model Example

Behind the scenes, this directive uses a combination of v-bind and input events to provide this type of functionality. In reality it is only syntax sugar. This also means that while v-model works for input, textarea and select elements by default it can be implemented for custom input elements as well by utilizing v-bind and events.

4.3.5 v-on

The directive "v-on" is used in order to catch events that are emitted by a component.

```
1 //long
2 <PollItem v-on:click="alert('component clicked')">
3 //shorthand
4 <PollItem @click="alert('component clicked')">
```

Listing 4.5: v-on Example

In this case, whenever the "click" event is emitted inside the PollItem component it is caught and an alert is displayed. Similarly to "v-bind", "v-on" also offers a shorthand syntax with the "@" symbol.

4.3.6 v-for

To display multiple items at once, the directive "v-for" can be used. It counts the number of elements of a given object and iterates over each to render some defined output.

```
1 <ul>
2   <li v-for="item in items">
3     {{ item.message }}
4   </li>
5 </ul>
```

Listing 4.6: v-for Example

4.3.7 v-if

To conditionally show or hide elements the directive "v-if" can be used. It not only simply hides an element when its "v-if" condition resolves to a falsy value, it completely removes it from the DOM. This means from an end-user perspective it is not possible to access this element, not even by looking at the raw HTML.

```
1 <NoticeboardItem v-if="showItem">
```

Listing 4.7: v-if Example

The very similar directive "v-show" uses additional CSS to conditionally hide elements, without removing them from the DOM - which is a cheaper operation. This can drastically improve performance in case there are very large nested elements which often switch between hidden or shown.

4.3.8 props

Props are a way of passing data from a parent component into a child component. Additionally, rules can be set up to validate a prop's value before it is used. [Listing 4.8](#) shows how a message prop of type String is defined inside a component.

```
1   <template>
2     <div>
3       {{ message }}
4     </div>
5   </template>
6
7   <script>
8     export default {
9       props: {
10        message: {
11          type: String,
12          required: false,
13          default: ""
14        }
15      }
16    };
17  </script>
```

Listing 4.8: Props Example

4.3.9 Component Communication

GUI based applications typically rely on event-driven programming paradigms. Vue is no exception: events are emitted and caught and dictate the course of actions of the application. Typically, props are used to pass data from a parent to a child component, whereas events are emitted with a payload from a child and are caught in the parent with the "v-on" directive. For heavily nested components however, a so called "event bus" or the vuex store is used which makes data available directly where it is needed instead of having to pass it from the deepest child to the highest parent. This project makes use of the props/events method and the vuex store.

4.3.10 Vuex

Vuex is a supporting library for Vue based on the Flux architecture to provide a solution for shared state management. Its concept revolves around a global state which is retrieved with "getters" and altered with "mutations" which in turn are committed by "actions". Mutations change the state's data synchronously so every change is trackable with debugging tools. Actions can be asynchronous and are often used for retrieving data from an API and then updating

the system's state by passing this data to a mutation and committing it. The term used for executing an action is "dispatch". It is important to note that while it is perfectly sensible to create methods in Vue components that reach out to an API and then commit a mutation, it makes far more sense to move these methods to the store as actions, especially if they are reused across the application. Vuex uses a "single state tree" - one JavaScript object that contains the entirety of the application's state. The state is either retrieved directly and unchanged or through getters which provide a way of transforming the returned data first. For example it is possible to directly retrieve the authentication token from the state and manually check its validity or by defining a getter which returns the users authentication state as a boolean value which depends on the token. Additionally, the store can be split up into "modules" and is done so in this project: a store file corresponds to a group of actions, getters and parts of the state. This greatly improves maintainability of the project by making it more comprehensible. [Listing 4.9](#) illustrates some commonly used ways of interacting with the store.

```
1  ...mapGetters("auth", ["isAuthenticated"])
2  this.$store.getters["auth/isAuthenticated"]
3  ...mapState("auth", ["token", "id"])
4  this.$store.state.auth.token
5  ...mapActions("issues", ["getIssues"])
6  this.$store.dispatch("issues/getIssues")
```

Listing 4.9: Vuex Usage

4.3.11 Computed Properties & Methods

A method in Vue is property of an object that contains code which can be rerun. Invoking a method always results in this code being run. Computed properties "are calculations that will be cached based on their dependencies and will only update when needed" [4]. This means they are very efficient as they will not be recalculated when a random property changes, instead Vue intelligently identifies the dependencies of a computed property and only updates it if a dependency's value changes. They are useful for composing data from existing resources. For example there could be an array of Noticeboard entries defined in the data property of a component, a search input field could bbe used to look for a specific item and a computed property would then return an array of found items depending on the search string.

4.4 Storage Management

As previously mentioned, storage management is achieved with Vuex. Furthermore, a module based approach is taken to split the store into multiple files to make it more maintainable. [Table 4.2](#) shows what part of the whole store is added by a certain module.

Module	Description
auth.js	Contains authentication related state, mutations, actions and getters. Examples: login, register, refresh tokens, verify email address, persist tokens to cookies and read them, request password reset
blackboard.js	Contains noticeboard related state, mutations, actions and getters. Examples: add new entry, retrieve all entries, edit an entry, search an entry from state
forum.js	Contains forum related state, mutations, actions and getters. Examples: add new entry, retrieve all entries, edit an entry, search an entry from state
index.js	Root store file which defines actions that get entries from every sub module or reset them. Used for populating the state at once when running the application for the first time
issues.js	Contains issue related state, mutations, actions and getters. Examples: add new entry, retrieve all entries, edit an entry, search an entry from state
polls.js	Contains poll related state, mutations, actions and getters. Examples: add new entry, retrieve all entries, edit an entry, search an entry from state
tenement.js	Contains house related state, mutations, actions and getters. Examples: add new house, retrieve all houses, accept invitations, get details of a house
ui.js	Contains UI related mutations. Used for showing a modal without having to emit multiple events
user.js	Contains user related state such as username or email, mutations, actions and getters. Examples: change user name, get user details (email, id, email state, etc.)

Table 4.2: Storage Structure

To make resetting the state simple every module file defines an action "resetState" which overwrites every property of the module's state with a predefined default value. It is important to overwrite the value and not the property itself to keep it reactive within the Vue instance. This is a known limitation of modern JavaScript and described in more detail on [Vue Change Detection Caveats](#).

4.4.1 State Population

There are two approaches of populating the state which I will refer to as "Just-in-time" and "Just-in-case". With the Just-in-time approach, the state is populated only when it is needed. For example when a user accesses a forum page with a list of threads, these are requested only after the user accessed the forum site. The Just-in-case approach on the other hand, loads these thread items into the store at a specified point in time, before the user accesses the forum site.

The application makes use of both: Just-in-case when the application is run to populate the dashboard with data (the usability of the system would not be very good if a user had to access each sub page first). And Just-in-time to refresh the contents of each sub page upon accessing it.

[Listing D.6](#) shows how state population is requested when the main site of the application is accessed and [Listing D.5](#) shows that a root "populateState" function is run which in turn dispatches the respective "getItems" function of each store submodule. [Listing 4.10](#) illustrates

requesting resources in a Just-in-time manner when accessing a subpage. More specifically, the mounted lifecycle hooks of page components are used in order to automatically dispatch a function which populates a state's module.

```
1  async mounted() {
2    try {
3      await this.$store.dispatch("blackboard/getBoards");
4    } catch (error) {
5      this.$toasted.global.my_error();
6    }
7  }
```

Listing 4.10: Just-in-time state population on subpage access

As some data is needed at any point while using the application it is crucial to make sure that it is actually available. Such data includes the users's first name, and house details. Especially the current house's id is needed before any other data set associated with this house is requested because the API needs this id to return the correct entries. This is achieved by using a middleware which as previously mentioned is run everytime before a user switches to a different page. A code sample can be found in [Listing D.4](#).

4.5 Authentication

In order to use the application, users have to be authenticated. They must provide an email address and a password to a login form and request authentication. [Table B.15](#) depicts the detailed flow for this use case. The API then checks if the user can be authenticated and if successful issues a token, refresh token and their respective expiry times. To improve the system's usability, this data is then persisted by using cookies (see [Listing D.2](#)). Everytime a user wants to move to a different subpage, the validity in terms of expiry time of the tokens is checked. Nuxt lets developers define "middleware" - code which is run before any page change. The middleware which is depicted in [Listing D.1](#) checks for one of three scenarios:

1. User is authenticated & both tokens too old -> logout
2. User is authenticated & only refresh token is valid -> tokens can be refreshed
3. User is not authenticated -> read from cookies

* A user is authenticated if the tokens and expiry times exist in **state**, they do not necessarily have to be valid (this is checked by the middleware)

In scenario 1, there is nothing the application can do to retrieve a new set of tokens, the user manually has to request authentication again. They are therefore logged out of the system, which happens by clearing every cookie file and every property of the state. This is important because if a different user would try to log into the system but the data is not cleared first, they might encounter some information of the previous user such as Noticeboard or Poll entries which are cached in state. In scenario 2 the token has expired but the refresh token is still valid, which means the application can automatically request a refresh. In scenario 3 a user is not authenticated at all. The application therefore tries to retrieve data from cookies if they exist. This is mainly done for usability as the state is nothing more than an in-memory JavaScript object and is cleared whenever a user exits the browser or manually refreshes the page, meaning the user would have to login over and over again. When a user interacts with the application

and therefore the API, they may encounter a "401 unauthorized" error which means that the user does not have valid authentication details. Axios which is used to interact with the API is extended with a custom defined plugin that instructs it to catch these types of errors and to immediately log out the user. [Listing D.3](#) shows how this is implemented.

4.5.1 Authorization

Not only do users have to be authenticated, they also have to be authorized for specific requests. From the API's perspective there are three access levels (the lower the number the higher the authorization level):

- Proprietor - Access Level 1
- Facility Management - Access Level 2
- Tenant - Access Level 3

What a user can do is dependant on their access level and is illustrated in the use case diagram depicted in [Figure 3.1](#). Because a user can be assigned to several houses and can have a different role in each one the roles are not contained inside the user object, but appended as a property for each house.

4.5.2 Summary

Users are authenticated by sending email and password to the API which in turn issues token, refresh token and expiry times. This data is persisted with cookies to improve usability. Before any page change the authentication state of the user is checked with middleware. When a request returns a "401 unauthorized" error this is caught by Axios and the user is logged out. Users can have one of three roles for every house they are assigned to, these roles dictate what a user is and is not allowed to do.

4.6 Common UI Items

In [Section 3.3](#) common UI elements were identified. These include: a Navigation Bar and Search Bar. Additionally, there are various forms for creating different entries. What every form needs is a way of saving the content or cancelling the creation process - a reusable button set which provides functionality for saving and cancelling is sensible. This section discusses how these UI elements are implemented.

4.6.1 Navigation Bar

The navigation bar is used to provide quick-access links to various site-destinations of the application. As the application uses Bulma as the underlying CSS framework, all UI items have to adhere to its guidelines for styling elements. The specific classes needed to build a Navigation Bar are described in the Bulma documentation available at [Bulma Navbar Documentation](#).

The Navbar is a special component: it seems like it is reused across all pages but in reality it is only defined once in the global layout file. In Vue, it is a common convention to such

components with "The" in the file name. For example the navigation bar is defined in a file called "TheNavbar.vue". When a component is used only once, it would theoretically be possible to hardcode links and other objects into that component, without decreasing maintainability. However, it makes for much cleaner code to outsource these into a parent component and pass them as props. For example the navigation bar defines a prop of type array which is used to pass all links that should be displayed. In the template section the "v-for" directive is then used to iterate over each link and to render it. Depending on the user's access levels, some links are hidden. For example the "Polls" link is hidden for user roles that are not of type proprietor. Additionally, props are used in order to set the color and dropdown shadow of the navbar.

Dynamic classes are used to make the hamburger menu (mobile) or dropdown menu (desktop) of the navbar interactive. When a user clicks on it, the click event is caught and the data properties "burgerActive" and "dropdownActive" are set to true. Vue recognizes this change and dynamically adds the "is-active" class to the respective HTML tags which results in the right dropdown menu to expand or the hamburger menu transform into an "X" like shape. [Figure 4.1](#) illustrates the finished component when rendered.



Figure 4.1: Navbar

4.6.2 Search Bar & Create Button

The search bar and the create button can easily be consolidated into the same functional component. This way users can search an item and create a new one while the design is kept minimal. This component is defined in a file called "SearchCreateMenu.vue". It builds upon the Bulma "level" element. The search input field is kept on the left, whereas the create button is kept on the right.

Since this is supposed to be a reusable custom input component it utilizes v-bind to provide a value from the parent component and an event that propagates a value typed by the user up to the parent. This way it is possible to use the "v-model" directive (see [Section 4.3.4](#)) in order to implement two-way data binding for this component.

It has several props to make it adaptable for different use cases:

- value - used for two way data binding
- numberOfItems - shows the number of items currently displayed
- pluralTitle - plural description of type of shown items (e.g Thread)
- singularTitle - singular description of type of shown items (e.g Threads)
- newItem - value of the create button
- link - link that will be redirected to upon create button click

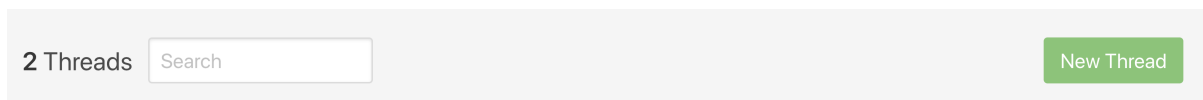


Figure 4.2: SearchCreateMenu Component

Usage: The component is used by including it in a page component along with another component that displays a list of items (e.g noticeboard or poll entries). Whenever a user inputs a search string, the v-model property is updated and automatically triggers that a computed property calls a getter method in the store and returns possible items. These possible items are then used to overwrite the list of displayed items.

4.6.3 Language Switcher

The language switcher component has a very simple purpose: instructing the internationalization plugin (detailed description in [Section 4.13](#)) to switch the language. It builds upon Bulmas "select" input element. The component's data object defines an array of languages which are iterated over and displayed as options of the select component. A global property which is provided by the internationalization plugin is encapsulated with "v-model" meaning that whenever a user selects a language which differs from the currently selected one, it is globally updated. Whenever the language is changed, the Vue directive "v-on" makes sure that the function "storeLocale" is called which sets a Cookie in the browser which saves the selected language.

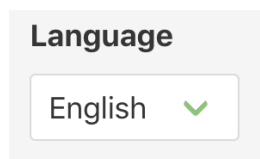


Figure 4.3: LangSwitcher Component

Usage: This component can be used as in a standalone way wherever needed. It is shown on the login page to provide a way of changing languages before a user even uses the application and on the profile page where a user can change preferences.

4.6.4 House Switcher

The House Switcher component works very similar to the Language Switcher component. A select element shows all available houses as options. The user can choose a house and behind the scenes a store mutation is called which changes the active house id in state. This is implemented with a so called "computed setter" - a computed property which in addition to returning cached results, is also capable of "setting" a value. [Listing 4.11](#) depicts how this is implemented: the get function of the computed property returns a cached version of the current house id, the set function updates the id in store and subsequently dispatches an action which repopulates the store with data from the newly selected house's data. The computed property "currentTenement" is then used with v-model which as described earlier is based on the input event and value prop. Vue recognizes when the select element emits an input event and automatically

runs the set function.

```
1   computed: {
2     currentTenement: {
3       get() {
4         return this.$store.state.tenement.id;
5       },
6       set(id) {
7         this.$store.commit("tenement/UPDATE", { id });
8         this.$store.dispatch("tenement/changeTenement");
9       }
10    }
11  }
```

Listing 4.11: Computed Setter

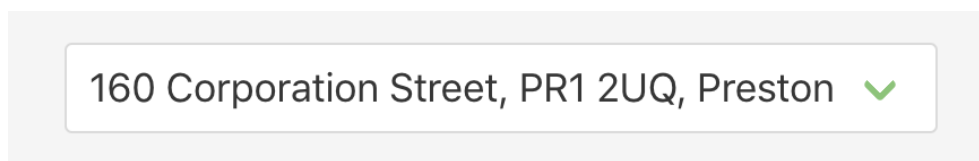


Figure 4.4: ChangeTenement Component

Usage: Similiar to the Language Switcher Component it can be included in any parent component. It is shown on the main page of the application where the dashboard is located to allow for easy traversal through a user's assigned houses.

4.7 Noticeboard

As defined in [Section 3.2.3](#) the noticeboard page shows a list of all noticeboard entries associated with a house, which are searchable by using the input field of the search bar. The identified elements are a list of all entries, a single entry, and a form for both creating and editing entries which is accessible when clicking the create button.

The composition of these elements are as following:

- Noticeboard Page > SearchCreateMenu.vue , BoardList.vue
- BoardList.vue > multiple BoardItem(s).vue
- Noticeboard Create/Edit Page > CreateBoard.vue

* The symbol ">" is read as "is composed of", every page contains the Navbar component

A single noticeboard item is built with the Bulma "tile" component (see [Bulma Tile Documentation](#)). The props of this component are: id, owner, title, content and size. Additionally, a computed property checks if the size prop is passed, if not, the size is calculated depending on the length of the content in terms of characters. The returned string value is a Bulma class which dictates how many rows an element should fill and is added as a dynamic class to the root div element of the component. The "v-if" directive conditionally hides or shows an "edit" anchor tag which links to the edit site of this specific entry. It does that by comparing the user id saved in store to the owner id which is passed as a prop.

The list containing multiple items defines a prop "boards" of type array which is passed from the page to the list component. It then iterates over this list with "v-for" and passes the specific data of the respective entry to a single item.

The page is composed of a Search Bar and a list of noticeboard entries. The entries are populated by making a request to the API when the page component's mounted lifecycle state is reached.

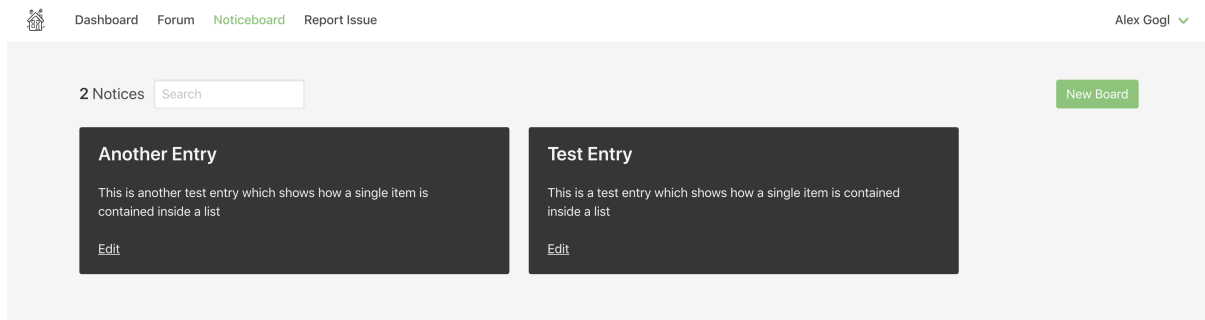


Figure 4.5: Noticeboard Page

4.8 Forum

The Forum page contains a list of all threads associated with a house that are searchable with a Search Bar. When a user clicks on a thread a detail-view is opened which shows the thread in a more detailed manner along with comments and provides a form for creating a new comment.

The composition of components that are related to the Forum are as following:

- Forum Page > SearchCreateMenu.vue, ThreadList.vue
- ThreadList > multiple ThreadItem(s).vue
- Thread Detail Page > ThreadDetail.vue, CommentList.vue, CreateComment.vue
- CommentList.vue > multiple CommentItem(s).vue
- Thread Create/Edit Page > CreateThread.vue

The ThreadList and ThreadItem components are structured like the BoardList and BoardItem components, but use different props and have a different design. An important distinction however, is that ThreadItem components contain a link that redirects to the detail view of the thread. In Nuxt's file structure, subfolders in the pages folder prefixed with underscore are treated differently than normal pages: the router instance is instructed to allow requests to a site with dynamic parameters. E.g. the page folder structure for the thread detail view is as following: "pages/blackboard/_id". When the application is accessed on <https://hausversammlung.at/blackboard/id-of-entry>, the thread detail page does two things in order to show the contents of a specific thread: it checks if thread items exists in the store and if not requests them during the mounted lifecycle state and then retrieves the id parameter of the url to lookup the thread item with that specific id in store (see [Listing 4.12](#)). In the store a getter called "getBoardByID" takes the id as a parameter, finds the associated

thread in store and returns it.

```
1  computed: {
2    board() {
3      return this.$store.getters["blackboard/getBoardByID"](
4        this.$route.params.id
5      );
6    }
7  }
```

Listing 4.12: Retrieving Board Entry from Store

Figure C.1 shows the rendered detail page of a thread item.

4.9 Polls

Polls are a somewhat complicated topic. When a proprietor votes for an option a request needs to be sent to the API and the Poll's UI has to be updated to prevent an additional option vote. The API however, does not return data which could be used on its own, client-side modification has to occur. As the pages Noticeboard, Forum, Polls and Issues share similar structure and functionality (searching items, creating items, displaying a list of items, populating state) these will not be discussed anymore. Instead, this section aims to describe how a Poll's UI is updated and what data has to be transformed in order to do so.

When voting for a poll option, the Poll Item component emits an event with the id of the voted option. The Poll List component propagates this id up to the Poll List component which is responsible for dispatching a store action which makes a request to the API. The API then however only returns information on what option was voted on and who voted on it (see [Listing 4.15](#)). The Poll Item component defines a prop "showResults" which when set to true prevents the user from voting again and shows the results of the poll. The difficulty lies in matching "poll_answer_option" to a specific poll.

```
1  {
2    id: "0aea27f2-4fb6-11e9-9ed5-02420a000613",
3    owner: "f85bd1f6-4a44-11e9-9ed5-02420a000613",
4    poll_answer_option: "69a0e2e0-4f2a-11e9-9ed5-02420a000613"
5  }
```

Listing 4.13: Data returned from API after vote

The solution is to iterate over every poll and to compare every option id "option.id" to a parameter "pollOptionId". If the pollOptionId matches any option.id the Poll's showResults prop can be set to true. Additionally, the total number of votes is incremented. This is achieved by using the JavaScript some() method in a nested manner that tests whether at least one element in the array passes a test implemented by a provided function. [Listing D.8](#) depicts the corresponding code.

Another and more complicated problem is that when the Polls page is initially accessed, a refresh of the data is requested as described in [Section 4.4.1](#) but the API only returns a list of polls, a list of options associated with a poll ("poll_answer_options") and a list of options that were voted on and are associated with a poll ("poll_given_answers"). [Figure C.2](#) shows the

raw data that is returned before any change is made to the data structure. Similarly to the previous problem, the client side needs to make certain changes to the data structure to make polls non-interactable when a user has already voted for that option. Listing 4.14 shows the mutation that is called that makes these changes after all polls are retrieved.

```
1  UPDATE_VOTES(state, userId) {
2    const pollList = [];
3
4    //find all poll_given_answers by userId and add id of answer_option to
    pollList
5    state.polls.filter(o =>
6      o.poll_given_answers.some(answer => {
7        if (answer.owner === userId) {
8          pollList.push(answer.poll_answer_option);
9          o.showResults = true;
10         return true;
11       }
12     })
13  };
14
15  //iterate over poll_answer_options and check if id is in pollList
16  //update if it is
17  state.polls.filter(o =>
18    o.poll_answer_options.some(option => {
19      if (pollList.includes(option.id)) {
20        option.selected = true;
21      }
22    })
23  );
24 }
25 };
```

Listing 4.14: Update Votes

In lines 5 to 13 every poll item and their respective voted items is iterated over and checked if a vote's owner matches the user id of the application's user. If it does, this vote is added to an array "pollList" and the whole poll is instructed to show the results with "o.showResults = true". What is left now is to tell the poll which item the application's user has voted on and to render it accordingly. In lines 17 to 23 all polls are iterated over and it is checked if any vote's id is included in the pollList array and if it is, this specific poll option is set as selected. Figure C.3 shows the data structure of polls after this update.

To visually illustrate what is meant, Figure 4.6 shows the rendered Poll Item Component with the "Yellow" option set as the selected item and the poll being in "showResults" state.

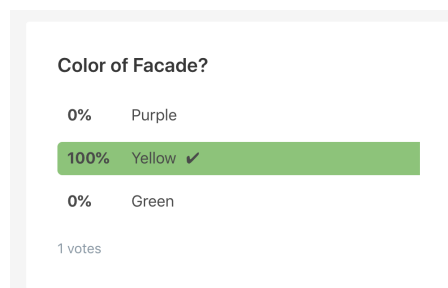


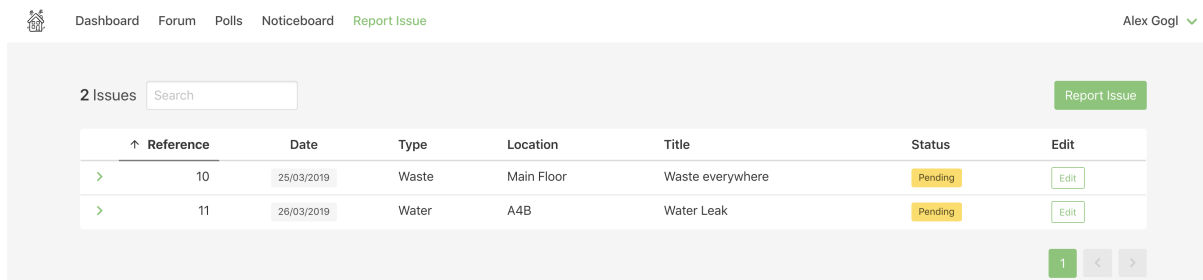
Figure 4.6: Poll Item

4.10 Issue Log System

The Issues page shows maintenance or house related issues. To give users a quick overview of all existing issues, a table was chosen to show them. More information regarding tables in Bulma is available at <https://bulma.io/documentation/elements/table/>. The component structure is similar to the other pages:

- Issue Page > SearchCreateMenu.vue, IssueTable.vue
- Create/Edit Issue Page > CreateIssue.vue

The only distinction is the type of data which is shown and how it is displayed: issues. A table shows Reference, Date, Type, Location, Title and Status which can be sorted in ascending or descending order. Additionally, users can edit issues if they are their owners, a button is shown accordingly. The API currently does not support an issue state being changed by facility management even though it is stated in the requirements. This functionality is therefore not implemented on the client side. Figure 4.7 shows the rendered Issue page.



↑	Reference	Date	Type	Location	Title	Status	Edit
>	10	25/03/2019	Waste	Main Floor	Waste everywhere	Pending	Edit
>	11	26/03/2019	Water	A4B	Water Leak	Pending	Edit

Figure 4.7: Issue Page

4.11 Adding a house

As required by the user stories (see Section A.0.8) and defined by the use case flows (see Section B.0.8), a house should either be creatable by a user or they should be able to accept an invitation. Additionally, they should be able to send new invitations. Implementation is relatively straightforward, for each use case one component exists that provides a form. After the user inputs the required data and submits the form, the API handles the rest. On the client-side the new house is added to an array of houses along with basic information. The user can then switch houses by using the Change House component. Figure C.4 shows the rendered implementation of a form where a new house can be added, Figure C.5 a form to accept an existing invitation and Figure C.6 a form to send invitations.

Sending an invitation requires an email address which will be sent instructions to add an existing house to the user's account. More specifically, the email contains a link with a special id as a query parameter which is retrieved by the application to pre-populate the invitation form.

4.12 Dashboard

The dashboard which is shown on the main page is populated with data that exists in store. The most recent polls, threads and noticeboard entries are shown. Additionally, metrics on number of users, squaremeters of a user, number of houses a user is associated with and number of issues associated with this house are displayed. Figure 4.8 illustrates an example dashboard.

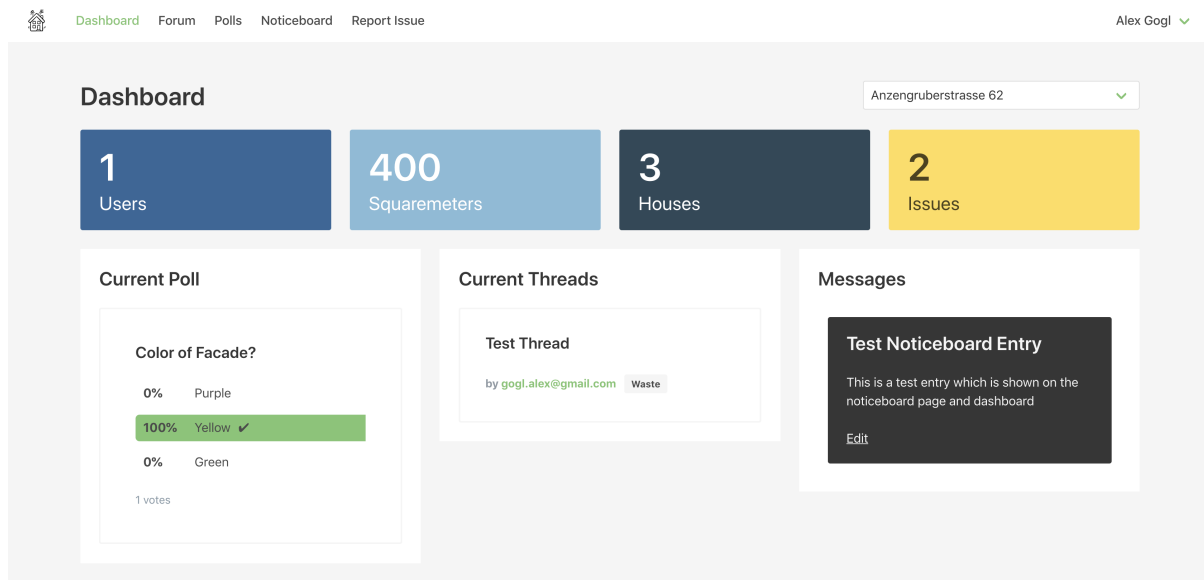


Figure 4.8: Dashboard Page

As already mentioned, the data used to populate the dashboard with, is retrieved from the store. This happens by using vuex store functions, more specifically: getters, as described in Section 4.3.10.

4.13 Internationalization

As the primary target market of the application is Austria, where German is spoken but English is also commonly used, two languages have to be offered. This is achieved by a third-party package called [vue-i18n](#) which is added as a custom Nuxt plugin. Instead of hardcoding the labels, descriptions, titles and other strings into the application, two language files "de-AT.json" and "en-US.json" are created which define key value pairs of a strings and their translations. The plugin injects a method "\$t" into every component which can be used to retrieve the appropriate value.

```
1 //hardcoded
2 <h1 class="title is-3">Create a House</h1>
3 //translation retrieved with injected method
4 <h1 class="title is-3">{{ $t("house.newHouse") }}</h1>
```

Listing 4.15: Data returned from API after vote

The package also injects the global property "\$i18n.locale" into the root Vue instance which represents the currently selected language. For example if German was selected, the property's value would be "de". This property is used by the computed setter described in [Section 4.6.3](#) to globally change the language.

4.14 Client-side Validation

Client-side validation is mainly used as a means of usability. It is possible for a user with malicious intent to bypass any client side validation mechanism and applies to every frontend application. Thorough server-side validation can never be replaced. However, the usability of an application is drastically improved if users get instantaneous feedback on their input. To achieve this, another third party package [VeeValidate](#) is used which utilizes JavaScript to validate input. Another solution would be to rely on the standard HTML validation capabilities of every modern browser. With HTML validation, simple input like emails, strings that should have a min or max character value or even input based on regexes can be validated. However, more sophisticated checks cannot be implemented: an example which also applies to this application is that one inputs's value should not be greater than another's. In this case, such a check is used to make sure that a user does not supply a greater value of their squaremeters than a house has in total during the "add house" process. In practical terms, the plugin injects the additional directive "v-validate" which takes either an object or string containing the rules as a parameter.

```
1 <BInput
2   v-model="share"
3   v-validate="{
4     required: true,
5     max_value: sum_shares
6   }"
7   type="number"
8 />
```

Listing 4.16: v-validate usage

[Listing 4.16](#) depicts a concrete usage example of how it is achieved that one value cannot be greater than another user-supplied value.

4.15 Error Handling

The application has several sources where errors can occur. Naturally, these errors should be dealt with gracefully while providing error explanations to the user. The by far biggest source of error is when a request to the API is made and an error is returned. This can have various reasons such as a timeout exception, failed validation or non existing records. The importance lies in dealing with these errors in a manner that leaves the application still interactive while adhering to Nielsen's "Visibility of System Status" principle. This is achieved with a pattern of catching and rethrowing errors, whereby these are dealt with in the most appropriate place. For example it is not very sensible for the store to initiate showing an error message, rather, the component which initially called the action from store that made the request should deal with it, because at this level there are much more options available. As most API requests are

made from the store, a choice was made to catch and rethrow so components can take care of them. This not only improves maintainability but also makes for a more loosely coupled system. Listing 4.17 depicts a pseudo login store action which rethrows errors when they occur. The code in Listing 4.18 catches these errors and displays an error message accordingly.

```
1  async login() {
2    try {
3      await pseudoApiRequest();
4    } catch (error) {
5      throw error;
6    }
}
```

Listing 4.17: Store Action Error Rethrow

To display error messages, another third-party package [Vue Toasted](#) is used. It is set up in way that also makes error messages bilingual. The associated configuration can be found in the file "plugins/toasted.js".

```
1  async login(authData) {
2    try {
3      await this.$store.dispatch("auth/login");
4    } catch (error) {
5      this.$toasted.global.my_error({
6        message: this.$t("toasts.loginFailed")
7      });
8    }
}
```

Listing 4.18: Catch Error in Component

Another noteworthy point is that throughout the application `async/await` is utilized which according to the JavaScript documentation is used to "... to simplify the behavior of using promises synchronously and to perform some behavior on a group of Promises" [25]. In addition, it makes the code cleaner and more comprehensible.

Chapter 5

Deployment with Docker

5.1 Applicability of Containers

5.2 Docker

5.3 Docker Swarm

5.4 Microservices

5.5 Reverse Proxy

Chapter 6

Test Strategy

6.1 Introduction

6.2 Testing Frontends

Describes which possibilities there are regarding frontend testing: Mainly unit and integration

6.3 Test Setup

Talks about vue-test-utils and jest integration

6.4 Continuous Integration

Gitlab CI / CD

6.5 Linting

Chapter 7

Evaluation

7.1 Project Objectives

7.2 Self Evaluation

7.3 Project Evaluation

7.4 Commercial Applicability

7.5 Conclusions

7.6 Future Work

Acronyms

JSX JavaScript XML. [3](#), [4](#)

MVP Minimum Viable Product. [10](#)

PWA Progressive Web App. [12](#)

SEO Search Engine Optimisation. [1](#)

SPA Single Page Application. [i](#), [1](#), [8](#), [12](#), [13](#)

SSR Server Side Rendering. [i](#), [8](#), [9](#), [13](#)

VCS Version Control System. [14](#)

Glossary

computed properties "Are calculations that will be cached based on their dependencies and will only update when needed" [4]. 2

DOM Defines the logical structure of HTML-based documents and the way a document is accessed and manipulated. 2

routing Is responsible for coordinating page changes on a website. More specifically when used as part of a single page application it allows for switching pages without changing the top level Universal Resource Identifier [3]. 3, 4

state management Provides a centralized store for all the components in a single page application. For example makes it possible to store a username in such a way, that any component can retrieve and use it. 3, 4

watchers A watcher tracks a property of the component state and runs a function when that property value changes. 2

Appendix A

User Stories

A.0.1 Noticeboard

As a user I want to create noticeboard entries so other users can see my messages
As a user I want to edit my noticeboard entries so that I fix mistakes
As a user I want to view all noticeboard items so that I have a complete overview
As a user I want to search for specific items so that I can find an item faster

Table A.1: Noticeboard related User Stories

A.0.2 Polls

As a proprietor I want to create polls other proprietors can vote on so that we as a group can make decisions
As a proprietor I want to view all polls so that I have a complete overview
As a proprietor I want to vote for a poll option so I my decision is accounted for

Table A.2: Poll related User Stories

A.0.3 Forum

As a user I want to create threads so other users can see my question or discussion
As a user I want to edit my threads so that I can fix mistakes
As a user I want to view all noticeboard items so that I have a complete overview
As a user I want to search for specific items so that I can find an item faster

Table A.3: Forum related User Stories

A.0.4 Issue Management

As a user I want to submit an issue so that it can be taken care of
As a user I want to edit my issues so that I can fix mistakes
As a user I want to view all logged issues so that I have a complete overview
As a user I want to search for specific issues so that I can find an item faster
As the facility management I want to change an issues state so I can let users know their issues are taken care of

Table A.4: Issue related User Stories

A.0.5 Profile

As a user I want to change my profile details so they reflect the right data
--

Table A.5: Profile related User Stories

A.0.6 Authentication

As a user I want to create an account so I can use the system
As a user I want to login to an existing account so I can use the system
As a user I want to change my password so that I can access my account
As a user I want to validate my email address so that emails are sent to the right address

Table A.6: Authentication related User Stories

A.0.7 Bilinguality

As a user I want to change the language so I can understand the site's content
--

Table A.7: Bilinguality related User Stories

A.0.8 House

As a user I want to create a new house so I can manage this property
As a user I want to invite other users to my house so that we can manage it together
As a user I want to accept an invitation
As a user I want to switch between houses

Table A.8: Property Management related User Stories

Appendix B

Use Case Flows

B.0.1 Noticeboard

The user opens the Noticeboard page by using the navigation bar
The user sees a list of all Noticeboard entries
The user searches for a specific item by using an input field (optional)

Table B.1: Use Case Flow: View Noticeboard

The user opens the Noticeboard page by using the navigation bar
The user clicks a "create" button
The user fills a form with a title and a message
The user clicks "Save"
The system checks for appropriate authorization
The system adds a new entry to the database
The user is redirected to the main Noticeboard page

Table B.2: Use Case Flow: Create Noticeboard

The user opens the noticeboard page by using the navigation bar
The user clicks on the edit link of an item if they are its owner
The user fills edits title and message of a pre-filled form
The user clicks "Save"
The system checks for appropriate authorization
The system mutates an existing entry of the database
The user is redirected to the main Noticeboard page

Table B.3: Use Case Flow: Edit Noticeboard

B.0.2 Polls

The proprietor opens the polls page by using the navigation bar
The proprietor sees a list of all polls
The proprietor searches for a specific item by using an input field (optional)

Table B.4: Use Case Flow: View Polls

The proprietor opens the polls page by using the navigation bar
The proprietor clicks a "create" button
The proprietor fills a form with a title and poll options
The proprietor clicks "Save"
The system checks for appropriate authorization
The system adds a new entry to the database
The proprietor is redirected to the main Polls page

Table B.5: Use Case Flow: Create Poll

The proprietor opens the polls page by using the navigation bar
The proprietor is presented with a list of polls and their respective options
The proprietor clicks on an option
The system checks for appropriate authorization
The system adds a new entry to the database
The poll is updated showing the results

Table B.6: Use Case Flow: Vote for Poll Option

B.0.3 Forum

The user opens the Forum page by using the navigation bar
The user sees a list of all threads
The user searches for a specific item by using an input field (optional)

Table B.7: Use Case Flow: View Forum

The user opens the Forum page by using the navigation bar
The user clicks a "create" button
The user fills a form with a title, question and category
The user clicks "Save"
The system checks for appropriate authorization
The system adds a new entry to the database
The user is redirected to the main Forum page

Table B.8: Use Case Flow: Create Thread

The user opens the Forum page by using the navigation bar
The user opens a specific Thread
The user clicks on the edit link of the item if they are its owner
The user fills edits title, question and category of a pre-filled form
The user clicks "Save"
The system checks for appropriate authorization
The system mutates an existing entry of the database
The user is redirected to the main Forum page

Table B.9: Use Case Flow: Edit Thread

The user opens the Forum page by using the navigation bar
The user opens a specific Thread
The user fills a form with their comments message
The user clicks "Save"
The system checks for appropriate authorization
The system adds a comment entry to the database
The comment is appended to the list of comments of the thread

Table B.10: Use Case Flow: Add Thread Comment

B.0.4 Issue Management

The user opens the Issues page by using the navigation bar
The user sees a list of all issues
The user searches for a specific item by using an input field (optional)

Table B.11: Use Case Flow: View Issues

The user opens the Issues page by using the navigation bar
The user clicks a "create" button
The user fills a form with a title, category, location and comment
The user clicks "Save"
The system checks for appropriate authorization
The system adds a new entry to the database
The user is redirected to the main Issues page

Table B.12: Use Case Flow: Log Issue

The user opens the Issues page by using the navigation bar
The user clicks on the edit link of an item if they are its owner
The user fills edits a title, category, location and comment of a pre-filled form
The user clicks "Save"
The system checks for appropriate authorization
The system mutates an existing entry of the database
The user is redirected to the main Issues page

Table B.13: Use Case Flow: Edit Issue

The facility management opens the Issue page by using the navigation bar
The facility management chooses a new state for a specific issue from a pre-defined dropdown menu
The system checks for appropriate authorization
The system adds a new entry to the database
The issue new stat is shown

Table B.14: Use Case Flow: Edit Issue

B.0.5 Authentication

The user opens the login page or is redirected automatically if not logged in
The user provides their email address and password to a form
The user clicks "Login"
The system checks for appropriate authentication details
The user is redirected to the Home page

Table B.15: Use Case Flow: Login

The user opens the login page or is redirected automatically if not logged in
The user clicks "Register"
The user provides their email address password, firstname and surname to a form
The user clicks "Register"
The system checks the correctness of the input
A new user is added to the system
The user is redirected to the Home page

Table B.16: Use Case Flow: Register

The user opens the login page or is redirected automatically if not logged in
The user clicks "Forgot Password"
The user provides their email address
The system sends an email to the user's email address
The user clicks on the link sent in the email
The user is redirected to the login page in "reset" mode
The user provides a new password
The system updates the password
The user is redirected to the login page

Table B.17: Use Case Flow: Forgot Password

B.0.6 Bilinguality

The user opens the Profile page by using the navigation bar
The user chooses a language from a dropdown select menu
The system changes the language and saves the preference

Table B.18: Use Case Flow: Change Language

B.0.7 Profile

The user opens the Profile page by using the navigation bar
The user changes their names
The user clicks "Save"
The system changes the user's name
The new name is updated throughout the application

Table B.19: Use Case Flow: Change Profile Details

B.0.8 House

The user clicks "Add House" in the navigation bar
The user is redirected to a page where they can add houses
The user is presented with two options: Accept Invitation OR Create new House
The user chooses "Create new House"
The user fills a form with address, total squaremeters and user role
The user clicks "Save"
The system adds a new entry to the database
The user is redirected to a page where they can invite other users to their house

Table B.20: Use Case Flow: Add House

The user clicks "Add House" in the navigation bar
The user is redirected to a page where they can add houses
The user is presented with two options: Accept Invitation OR Create new House
The user chooses "Accept Invitation"
The user fills a form with invitation id and user role
The user clicks "Add"
The system checks if the user is eligible to be added to this house
The user is redirected to a page where they can invite other users to their house

Table B.21: Use Case Flow: Accept Invitation

The user completes use case "Accept Invitation" or "Add House" or switches to the Profile page
The user fills a form with an email
The user clicks "Add"
The system sends an email to the added user
The form's content is deleted so another user can be added

Table B.22: Use Case Flow: Invite User

The user logs in or clicks the Home link in the navigation bar
The user is redirected to the Home page

Table B.23: Use Case Flow: View Dashboard

Appendix C

Additional Figures

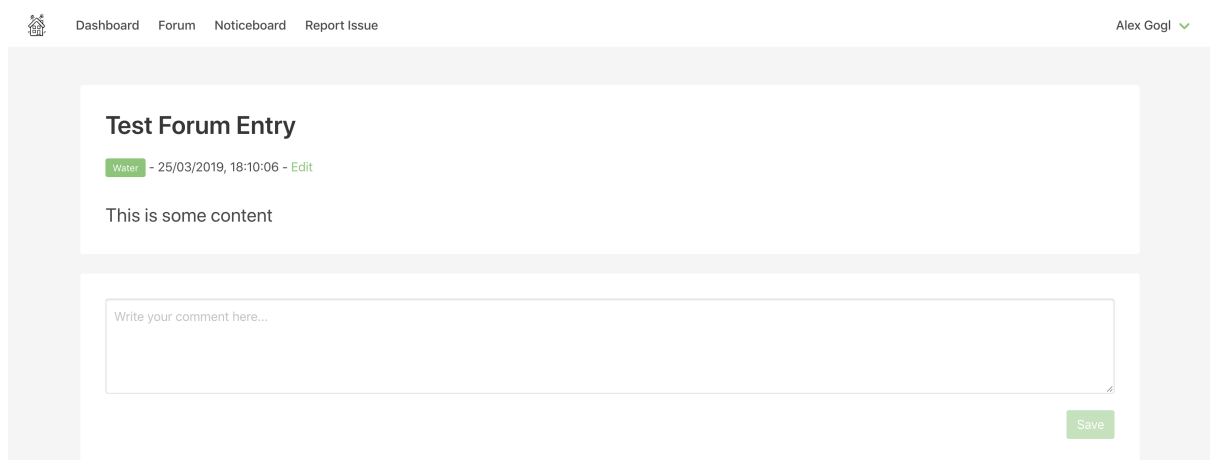


Figure C.1: Thread Detail Page

```

▼ polls: Object
  ▼ polls: Array[1]
    ▼ 0: Object
      created_date: "2019-03-25T18:18:39.088172Z"
      id: "69a0db9c-4f2a-11e9-9ed5-02420a000613"
      owner: "f85bd1f6-4a44-11e9-9ed5-02420a000613"
      participation: 1
      ▼ poll_answer_options: Array[3]
        ► 0: Object
        ▼ 1: Object
          answer_text: "Yellow"
          id: "69a0e2e0-4f2a-11e9-9ed5-02420a000613"
          owner: "f85bd1f6-4a44-11e9-9ed5-02420a000613"
          votes: 0
        ► 2: Object
      ▼ poll_given_answers: Array[1]
        ▼ 0: Object
          id: "0aea27f2-4fb6-11e9-9ed5-02420a000613"
          owner: "f85bd1f6-4a44-11e9-9ed5-02420a000613"
          poll_answer_option: "69a0e2e0-4f2a-11e9-9ed5-02420a000613"
        question: "Color of Facade?"
        tenement: "57ce8306-4f2a-11e9-9ed5-02420a000613"

```

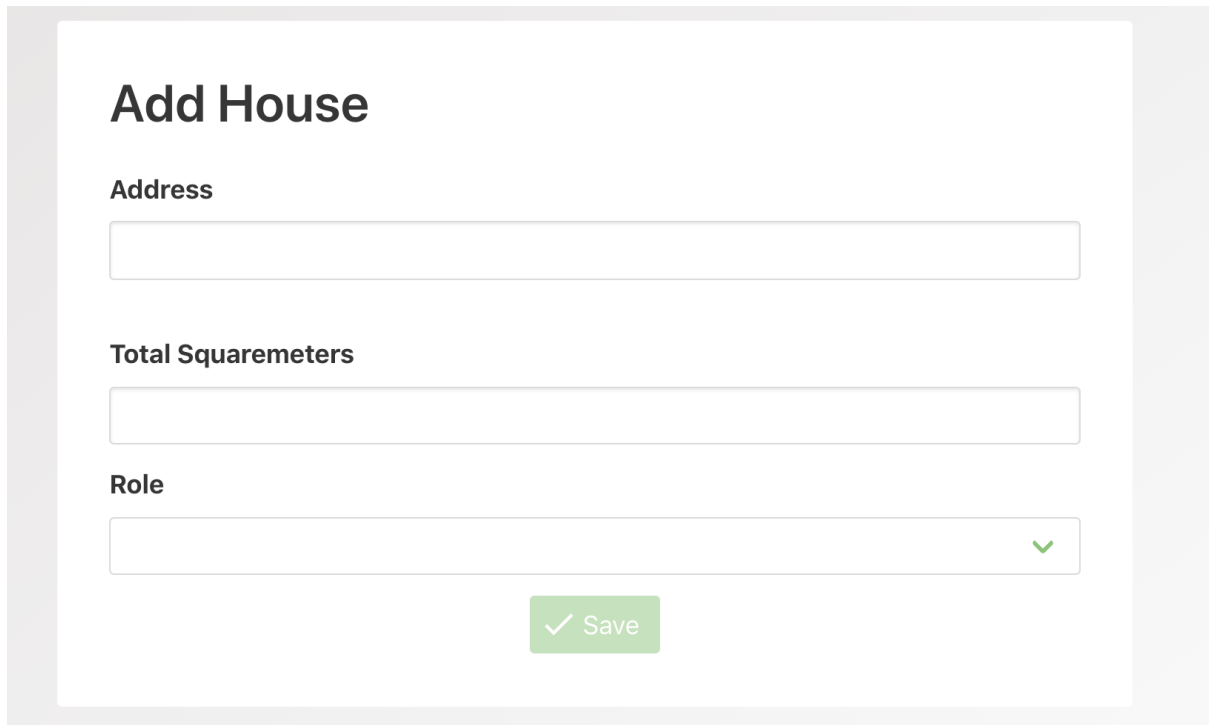
Figure C.2: Poll State Object before updating

```

▼ polls: Object
  ▼ polls: Array[1]
    ▼ 0: Object
      created_date: "2019-03-25T18:18:39.088172Z"
      id: "69a0db9c-4f2a-11e9-9ed5-02420a000613"
      owner: "f85bd1f6-4a44-11e9-9ed5-02420a000613"
      participation: 1
      ▼ poll_answer_options: Array[3]
        ► 0: Object
        ▼ 1: Object
          answer_text: "Yellow"
          id: "69a0e2e0-4f2a-11e9-9ed5-02420a000613"
          owner: "f85bd1f6-4a44-11e9-9ed5-02420a000613"
          selected: true
          votes: 1
        ► 2: Object
      ▼ poll_given_answers: Array[1]
        ▼ 0: Object
          id: "0aea27f2-4fb6-11e9-9ed5-02420a000613"
          owner: "f85bd1f6-4a44-11e9-9ed5-02420a000613"
          poll_answer_option: "69a0e2e0-4f2a-11e9-9ed5-02420a000613"
          question: "Color of Facade?"
          showResults: true
          tenement: "57ce8306-4f2a-11e9-9ed5-02420a000613"

```

Figure C.3: Poll State Object after updating




Add House

Address

Total Squaremeters

Role




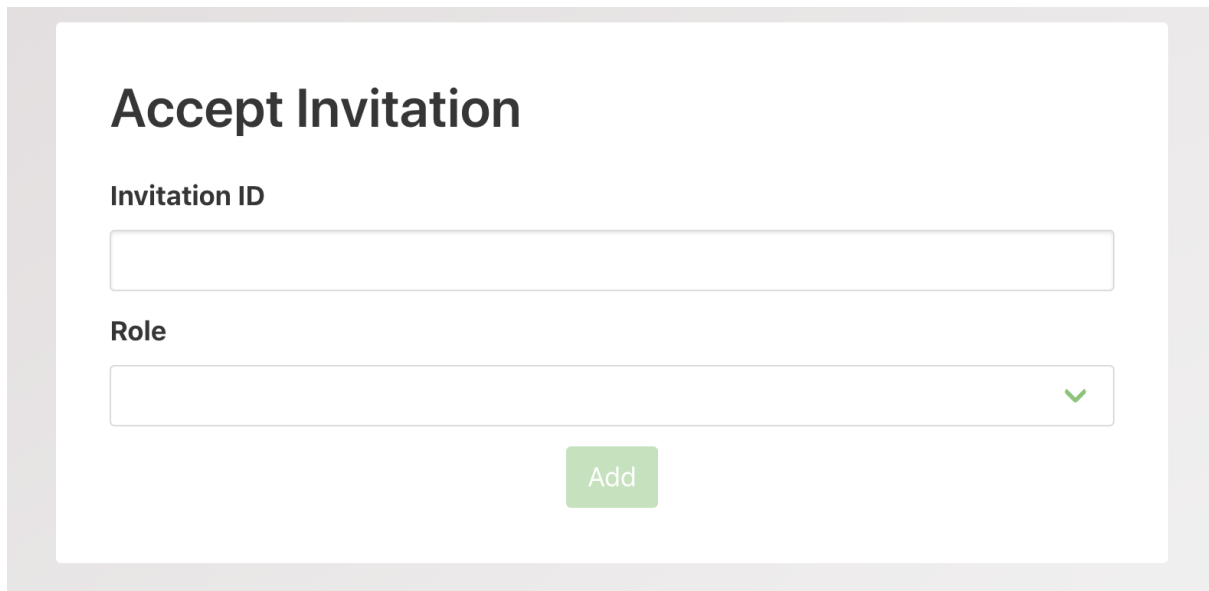
 Save


Figure C.4: Add House Form



Accept Invitation

Invitation ID

Role




 Add

Figure C.5: Accept Invitation Form

Invite User

Email

Add

[Back Home](#)

Figure C.6: Send Invitation Form

Appendix D

Additional Listings

```
1 export default async function(context) {
2
3   if (!context.store.getters["auth/isAuthenticated"]) {
4     console.info("[MIDDLEWARE] Not authenticated, reading data from
      cookie...");
5     try {
6       let result = await context.store.dispatch("auth/initAuth");
7       console.info(result);
8     } catch (error) {
9       console.info(error.message);
10      if (error.code == "nocookies") {
11        return context.redirect("/login");
12      } else if (error.code == "expired") {
13        await context.store.dispatch("auth/logout");
14        return context.redirect("/login");
15      } else if (error.code == "refresh") {
16        try {
17          await context.store.dispatch("auth/refreshAuth");
18        } catch (error) {
19          await context.store.dispatch("auth/logout");
20          return context.redirect("/login");
21        }
22      }
23    }
24  } else if (
25    //both tokens are too old cannot do anything -> logout (clear state)
26    new Date() > new Date(context.store.state.auth.refresh_token_expiry)
27  ) {
28    console.info("[MIDDLEWARE] Tokens are expired, logging out...");
29    await context.store.dispatch("auth/logout");
30    return context.redirect("/login");
31  } else if (
32    //token is too old but refresh token still valid refresh
33    new Date() > new Date(context.store.state.auth.token_expiry) &&
34    new Date() < new Date(context.store.state.auth.refresh_token_expiry)
35  ) {
36    console.info("[MIDDLEWARE] Refresh token still valid, refreshing...");
37    try {
38      await context.store.dispatch("auth/refreshAuth");
39    } catch (error) {
40      await context.store.dispatch("auth/logout");
41      return context.redirect("/login");
42    }
43  }
44 }
```

Listing D.1: Auth Middleware

```

1  saveTokenData({ commit }, result) {
2    //set token data
3    commit("SET", result);
4
5    //set cookie data
6    Cookie.set("token", result.token, { secure: false });
7    Cookie.set("token_expiry", result.token_expiry, { secure: false });
8    Cookie.set("refresh_token", result.refresh_token, { secure: false });
9    Cookie.set("refresh_token_expiry", result.refresh_token_expiry, {
10     secure: false
11   });
12   Cookie.set("user", result.user, { secure: false });
13
14   // Adds header: `Authorization: Token token` to all requests
15   this.$axios.setToken(result.token, "Token");
16 }

```

Listing D.2: Persisting Auth Data

```

1  export default function({ $axios, app, store }) {
2    $axios.onError(error => {
3      if (!error.response) {
4        // network error
5        app.$toasted.global.my_error({
6          message: "You are not connected to the internet"
7        });
8        return;
9      }
10     const code = parseInt(error.response && error.response.status);
11     if (code === 401) {
12       //don't do anything when coming from login page
13       if ($nuxt.$route.name === "login") {
14         return;
15       }
16
17       app.$toasted.global.my_error({
18         message: "Token too old, logging you out!"
19       });
20
21       store.dispatch("auth/logout");
22     }
23   });
24 }

```

Listing D.3: Axios Unauthorized Plugin

```

1  export default async function(context) {
2    if (!context.store.getters["user/isPopulated"]) {
3      try {
4        await context.store.dispatch("user/getUser");
5        await context.store.dispatch("tenement/getTenements");
6        await context.store.dispatch("tenement/getDetails");
7      } catch (error) {
8        console.error(error);
9      }
10   }
11 }

```

Listing D.4: Population Middleware

```

1  export const actions = {
2    resetState({ dispatch }) {
3      dispatch("tenement/resetState", null, { root: true });
4      dispatch("user/resetState", null, { root: true });
5      dispatch("polls/resetState", null, { root: true });
6      dispatch("forum/resetState", null, { root: true });
7      dispatch("blackboard/resetState", null, { root: true });
8      dispatch("documents/resetState", null, { root: true });
9
10     console.log("[STORE] State was reset");
11   },
12
13   resetUserGeneratedData({ dispatch }) {
14     dispatch("polls/resetState", null, { root: true });
15     dispatch("forum/resetState", null, { root: true });
16     dispatch("blackboard/resetState", null, { root: true });
17     dispatch("documents/resetState", null, { root: true });
18
19     console.log("[STORE] User generated data was reset");
20   },
21
22   populateState({ dispatch, getters }) {
23     //dont get other data when we don't have any tenements to get data
24     //from
25     if (!getters["tenement/numberOfTenements"]) {
26       return;
27     }
28     //when populateState is executed we assume getTenements was executed
29     //some time before
30     try {
31       //solution where promise.all() is used might be better for
32       //efficiency
33       dispatch("tenement/getDetails", null, { root: true });
34       dispatch("polls/getPolls", null, { root: true });
35       dispatch("forum/getThreads", null, { root: true });
36       dispatch("blackboard/getBoards", null, { root: true });
37       dispatch("issues/getIssues", null, { root: true });
38
39       console.log("[STORE] State was populated");
40     } catch (error) {
41       this.$toasted.global.my_error();
42     }
43   }
44 };

```

Listing D.5: Just in Case dispatched actions

```

1  async mounted() {
2    try {
3      this.$store.dispatch("populateState");
4    } catch (error) {
5      this.$toasted.global.my_error();
6    }
7  }

```

Listing D.6: Just in Case Store Population on app run

```

1  Vue.component('todo-component', {
2      template: '<h1> {{ newItem }} </h1>',
3      data() {
4          return {
5              items: [
6                  {
7                      id: '1',
8                      title: 'Buy milk',
9                      completed: false
10                 },
11             ],
12             newItem: ''
13         };
14     },
15     methods: {
16         addItem() {
17             // LOGIC OMITTED FOR READABILITY
18         }
19     }
20 }

```

Listing D.7: Vue component without bundler

```

1  ADD_VOTE(state, pollOptionId) {
2      state.polls.some(o =>
3          o.poll_answer_options.some(option => {
4              if (option.id === pollOptionId) {
5                  option.votes++;
6                  //manually set showresults to true so rendering happens
6                      immediately in dashboard
7                  o.showResults = true;
8                  return true;
9              }
10          })
11      );
12  }

```

Listing D.8: Add Vote

References

Literature

- [1] Alex Banks and Eve Porcello. *Learning React: Functional Web Development with React and Redux*. " O'Reilly Media, Inc.", 2017 (cit. on p. 3).
- [2] Kevin Bedell. 'Opinions on Opinionated Software'. In: *Linux J.* 2006.147 (July 2006), pp. 1–. URL: <http://dl.acm.org/citation.cfm?id=1145562.1145563> (cit. on p. 6).
- [3] Justin F. Brunelle et al. 'The impact of JavaScript on archivability'. In: *International Journal on Digital Libraries* 17.2 (June 2016), pp. 95–117. URL: <https://doi.org/10.1007/s00799-015-0140-8> (cit. on pp. 8, 46).
- [4] Olga Filipova. *Learning Vue. js 2*. Packt Publishing Ltd, 2016 (cit. on pp. 1, 2, 28, 46).
- [5] Gil Fink and Ido Flatow. 'Search Engine Optimization for SPAs'. In: *Pro Single Page Application Development: Using Backbone.js and ASP.NET*. Berkeley, CA: Apress, 2014, pp. 267–276. URL: https://doi.org/10.1007/978-1-4302-6674-7_12 (cit. on pp. 7, 9).
- [6] Naimul Islam Naim. 'ReactJS: An Open Source JavaScript Library for Front-end Development'. In: (2017) (cit. on p. 3).
- [7] M. N. A. Khan and A. Mahmood. 'A distinctive approach to obtain higher page rank through search engine optimization'. In: *Sādhana* 43.3 (Mar. 2018), p. 43. URL: <https://doi.org/10.1007/s12046-018-0812-3> (cit. on pp. 7, 8).
- [8] Phillip A Laplante. *What every engineer should know about software engineering*. CRC Press, 2007 (cit. on p. 23).
- [9] Callum Macrae. *Vue. js: Up and Running: Building Accessible and Performant Web Apps*. " O'Reilly Media, Inc.", 2018 (cit. on p. 2).
- [10] Filippo Menczer et al. 'Evaluating topic-driven web crawlers'. In: *Proceedings of the 24th annual international ACM SIGIR conference on Research and development in information retrieval*. ACM. 2001, pp. 241–249 (cit. on p. 7).
- [11] Michael Mikowski and Josh Powell. *Single Page Web Applications: JavaScript end-to-end*. 1st. Generic publisher, 2013 (cit. on p. 1).
- [12] Francisco Ortin and Miguel Garcia. 'A Programming Language That Combines the Benefits of Static and Dynamic Typing'. In: *Software and Data Technologies*. Ed. by José Cordeiro, Maria Virvou and Boris Shishkov. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 72–87 (cit. on pp. 5, 6).
- [13] Zhuofan Yang, Yong Shi and Bo Wang. 'Search engine marketing, financing ability and firm performance in E-commerce'. In: *Procedia Computer Science* 55 (2015), pp. 1106–1112 (cit. on p. 7).

Online sources

- [14] Nuxt.js contributors. *Nuxt.js - The Vue.js Framework*. 2019. URL: <https://nuxtjs.org/> (visited on 16/03/2019) (cit. on p. 22).
- [15] React contributors. *Add React to a Website - React*. URL: <https://reactjs.org/docs/add-react-to-a-website.html> (visited on 18/02/2019) (cit. on pp. 3, 6).
- [16] React contributors. *Rendering Elements - React*. URL: <https://reactjs.org/docs/rendering-elements.html> (visited on 18/02/2019) (cit. on pp. 3, 4).
- [17] Vue contributors. *Comparison with Other Frameworks - Vue.js*. URL: <https://vuejs.org/v2/guide/comparison.html> (visited on 18/02/2019) (cit. on pp. 2-6).
- [18] Vue contributors. *TypeScript Support - Vue.js*. URL: <https://vuejs.org/v2/guide/typescript.html> (visited on 18/02/2019) (cit. on p. 6).
- [19] Vue.js contributors. *Vue.js Server-Side Rendering Guide / Vue SSR Guide*. URL: <https://ssr.vuejs.org/#why-ssr> (visited on 03/03/2019) (cit. on pp. 8, 9).
- [20] Michael Xu Erik Hendriks. *Official Google Webmaster Central Blog: Understanding web pages better*. May 2014. URL: <https://webmasters.googleblog.com/2014/05/understanding-web-pages-better.html> (visited on 01/03/2019) (cit. on p. 8).
- [21] *Get started with dynamic rendering / Search / Google Developers*. Feb. 2019. URL: <https://developers.google.com/search/docs/guides/dynamic-rendering> (visited on 03/03/2019) (cit. on p. 9).
- [22] Anthony Gore. *What's The Deal With Vue's Virtual DOM? - Vue.js Developers - Medium*. Dec. 2017. URL: <https://medium.com/js-dojo/whats-the-deal-with-vue-s-virtual-dom-3ed4fc0dbb20> (visited on 01/03/2019) (cit. on p. 2).
- [23] Stefan Krause. *Interactive Results*. URL: <https://stefankrause.net/js-frameworks-benchmark8/table.html> (visited on 20/02/2019) (cit. on pp. 3-5).
- [24] Ian Oeschger Marcio Galli Roger Soares. *Inner-browsing extending the browser navigation paradigm - Archive of obsolete content / MDN*. URL: https://developer.mozilla.org/en-US/docs/Archive/Inner-browsing_extending_the_browser_navigation_paradigm (visited on 28/02/2019) (cit. on p. 1).
- [25] *mocument.pdf*. URL: <http://127.0.0.1:55414/viewer.html?file=/pdf:%252FUsers%252Fagcty%252Fcode%252FDoubleProject%252Fout%252Fmocument.pdf> (visited on 26/03/2019) (cit. on p. 41).
- [26] Michał Sajnog. *13 Top Companies That Have Trusted Vue.js - Examples of Applications / Netguru Blog on Vue*. Mar. 2018. URL: <https://www.netguru.com/blog/13-top-companies-that-have-trusted-vue.js-examples-of-applications;%20http://gfs.sf.net/gerris.pdf> (visited on 18/02/2019) (cit. on p. 1).
- [27] *Single File Components - Vue.js*. URL: <https://vuejs.org/v2/guide/single-file-components.html#What-About-Separation-of-Concerns> (visited on 17/03/2019) (cit. on p. 25).
- [28] *TypeScript - JavaScript that scales*. URL: <https://www.typescriptlang.org/> (visited on 19/02/2019) (cit. on p. 4).

List of Figures

1.1	Comparison of Vue, React and Angular	7
2.1	Proprietors Assembly Use Cases	12
3.1	System Design	16
3.2	Sitemap	21
4.1	Navbar	32
4.2	SearchCreateMenu Component	33
4.3	LangSwitcher Component	33
4.4	ChangeTenement Component	34
4.5	Noticeboard Page	35
4.6	Poll Item	37
4.7	Issue Page	38
4.8	Dashboard Page	39
C.1	Thread Detail Page	54
C.2	Poll State Object before updating	55
C.3	Poll State Object after updating	56
C.4	Add House Form	57
C.5	Accept Invitation Form	57
C.6	Send Invitation Form	58

Listings

1.1	Vue Single File Component	2
1.2	Usage of JSX Render Function	4
1.3	Angular Basic Usage Example	5
4.1	Vue Single File Component	24
4.2	Interpolation Example	25
4.3	v-bind Example	25
4.4	v-model Example	26
4.5	v-on Example	26
4.6	v-for Example	26
4.7	v-if Example	26
4.8	Props Example	27
4.9	Vuex Usage	28
4.10	Just-in-time state population on subpage access	30
4.11	Computed Setter	34
4.12	Retrieving Board Entry from Store	36
4.13	Data returned from API after vote	36
4.14	Update Votes	37
4.15	Data returned from API after vote	39
4.16	v-validate usage	40
4.17	Store Action Error Rethrow	41
4.18	Catch Error in Component	41
D.1	Auth Middleware	59
D.2	Persisting Auth Data	60
D.3	Axios Unauthorized Plugin	60
D.4	Population Middleware	60
D.5	Just in Case dispatched actions	61
D.6	Just in Case Store Population on app run	61
D.7	Vue component without bundler	62
D.8	Add Vote	62

List of Tables

2.1	Project Requirements	10
3.1	Use Case Flow: Login	17
3.2	Use Case Flow: Register	17
3.3	Use Case Flow: View Dashboard	18
3.4	Use Case Flow: Create Noticeboard	18
3.5	Use Case Flow: Create Poll	19
3.6	Use Case Flow: Create Thread	19
3.7	Use Case Flow: Log Issue	20
3.8	Use Case Flow: Change Profile Details	20
3.9	Use Case Flow: Accept Invitation	21
4.1	Nuxt Folder Structure	23
4.2	Storage Structure	29
A.1	Noticeboard related User Stories	47
A.2	Poll related User Stories	47
A.3	Forum related User Stories	47
A.4	Issue related User Stories	47
A.5	Profile related User Stories	48
A.6	Authentication related User Stories	48
A.7	Bilinguality related User Stories	48
A.8	Property Management related User Stories	48
B.1	Use Case Flow: View Noticeboard	49
B.2	Use Case Flow: Create Noticeboard	49
B.3	Use Case Flow: Edit Noticeboard	49
B.4	Use Case Flow: View Polls	49
B.5	Use Case Flow: Create Poll	50
B.6	Use Case Flow: Vote for Poll Option	50
B.7	Use Case Flow: View Forum	50
B.8	Use Case Flow: Create Thread	50
B.9	Use Case Flow: Edit Thread	50
B.10	Use Case Flow: Add Thread Comment	51
B.11	Use Case Flow: View Issues	51
B.12	Use Case Flow: Log Issue	51
B.13	Use Case Flow: Edit Issue	51
B.14	Use Case Flow: Edit Issue	51
B.15	Use Case Flow: Login	52

B.16 Use Case Flow: Register	52
B.17 Use Case Flow: Forgot Password	52
B.18 Use Case Flow: Change Language	52
B.19 Use Case Flow: Change Profile Details	53
B.20 Use Case Flow: Add House	53
B.21 Use Case Flow: Accept Invitation	53
B.22 Use Case Flow: Invite User	53
B.23 Use Case Flow: View Dashboard	53