BACHELOR'S THESIS 2024

Development of a Tenant Configuration Application

Per Lundegård

ISSN 1650-2884
LU-CS-EX: 2024-14

DEPARTMENT OF COMPUTER SCIENCE
LTH | LUND UNIVERSITY

KANDIDATARBETE
Datavetenskap

LU-CS-EX: 2024-14

Development of a Tenant Configuration Application

Utveckling av en applikation för konfigurering av tenants

Per Lundegård

Development of a Tenant Configuration Application

Per Lundegård
dat15plu@student.lu.se

April 9, 2024

Bachelor's thesis work carried out at Schneider Electric Buildings.

Supervisors: Felix Nilsson, felix.nilsson@se.com
Niklas Fors, niklas.fors@cs.lth.se

Examiner: Görel Hedin, gorel.hedin@cs.lth.se

Abstract

Software as a Service (SaaS) is a cloud service model that aids the distribution and usage of software applications. In recent years, SaaS models have increased in popularity. Within SaaS architecture, a popular design approach is multi-tenancy, which allows multiple customers to share the same server resources while still providing the appearance and functionality of customized software for each. An essential part of multi-tenancy is the different microservices the software relies on. While the amount of customization required for each customer may differ, some customizations are not optional. As such, one essential microservice for almost every multi-tenant system is the service that handles basic tenant configuration. This offers the opportunity for smooth onboarding and customization for customers. In this thesis, we will create a microservice for tenant customization, along with a management tool. The goal with the management tool is to speed up the process and minimize the technical complexity during customer onboarding. Additionally, we will evaluate the management tool through assessments conducted with UX (User Experience) experts and the intended user group. The findings highlight the significance of evaluation with both of the targeted user group and UX experts to improve the tools usability.

Keywords: BSc, multi-tenant, SaaS, Microservice, Usability

Acknowledgements

I would like to thank my supervisor at Schneider Electric Buildings, Felix Nilsson, for his technical knowledge and assistance throughout the work. I would also like to thank my supervisor at LTH, Niklas Fors, for his guidance and engagement throughout the thesis process.

Contents

1 Introduction **7**
1.1 Goal . . . 8
1.2 Problem Statement . . . 8
1.3 Related Work . . . 8
1.4 Contribution statement . . . 9
1.5 Outline . . . 9

2 Background **11**
2.1 REST API . . . 11
2.2 SaaS Architecture . . . 11
2.2.1 Single-tenant . . . 11
2.2.2 Multi-tenant . . . 12
2.3 Docker . . . 12
2.4 Kubernetes . . . 12

3 User Interfaces **15**
3.1 The Client Application . . . 16
3.2 The Support Application . . . 17

4 Implementation **21**
4.1 Development Process . . . 21
4.2 Operational Logic . . . 21
4.2.1 The Tenant Registry . . . 22
4.2.2 Moving from CSS styling files to microservice . . . 22
4.2.3 The Support Application . . . 23
4.3 Deployment . . . 23
4.4 Alternative Solutions . . . 25

5 Evaluation **27**
5.1 Security . . . 27

5.2 Design Evaluation 28
5.2.1 UX Expert Reviews 28
5.2.2 Think Aloud User Testing 29
5.2.3 Evaluation Conclusion 29

6 Conclusion and Future Work **31**

References **33**

Chapter 1

Introduction

Software as a Service (SaaS) has become a popular way of selling software. In 2015, end-users spent 31.4 billion USD, in 2023, that number had gone up to 197.29 billion USD[9]. When designing a SaaS architecture, an essential decision is what type of cloud architecture to use for your customers to connect to. The options are typically between a single-tenant or a multi-tenant architecture [7]. In a single-tenant SaaS architecture, each customer has their own separate cloud instance, whereas in a multi-tenant SaaS architecture, multiple customers share the same cloud instance. Each solution comes with its own set of pros and cons. Generally speaking, when you have few clients, one would probably choose a single-tenant instance architecture, because of its simplicity. However, single-tenant architecture does not scale well, because it requires individual infrastructure expansions for each customer, meaning multi-tenant architecture becomes more appealing with a larger customer base. A key component of multi-tenancy is the microservices inside the SaaS infrastructure, where each service inside the architecture serves multiple clients. Multi-tenancy setup offers several benefits if done correctly, such as utilizing economy of scale more efficiently, and easier maintenance[3].

Let's dive into the practical implications of these choices on customer onboarding. Imagine there is a new customer who either bought our software or is interested in seeing what it can do. In a single-tenant setup, you would need to set up their own virtual machine, assign them their own server, install the software, and then configure it so the product works as intended, often involving source code modifications.

With multi-tenancy, onboarding a new customer is potentially as simple as adding a couple of rows in a database, depending on the amount of customization required. And instead of having a developer manually inserting rows and data into the databases, the process is further simplified by a web app, which essentially consists of a few forms. This approach significantly reduces both the time and technical requirements of the onboarding process.

This thesis aims to explore the development of a microservice designed for managing tenant configurations. Additionally, we will create a web application that assists support teams in onboarding and configuring tenants using this microservice.

1.1 Goal

This thesis was carried out at Schneider Electric Buildings. One of their many products is a SaaS application where the customer can get an overview of all their buildings and see different type of sensor information, such as electricity usage. We will refer to this SaaS application as the ***client application***. At it's current state in the application layer, the *client application* only supports single-tenancy.

One goal of this thesis was to develop a back-end REST API microservice, designed for managing tenant onboarding, tenant styling and authentication configurations for each customer. We will refer to this API as the ***tenant registry***.

Another goal of this thesis was to develop a stand-alone front-end, the ***support application***, capable of performing Create, Read and Update operations on the back-end *tenant registry*, with the help of forms. Additionally, the *support application* was evaluated by its intended users to assess its usability.

Furthermore, the *client application* first described in this section was modified so it uses the *tenant registry* for loading dynamic styling content, instead of using static content. The modification required backwards comparability, as the strategy for transitioning from single-tenant to multi-tenant is not immediate, as there are security and reliability concerns to change the product in larger ways.

The final step involved the deployment of the *support application* and *tenant registry* into a production environment, which is within a Kubernetes cluster.

1.2 Problem Statement

This thesis aims to address the following questions:

- How should the *support application* be designed so it will be intuitive for the targeted user?
- How should the *support application* and *tenant registry* be deployed in a secure manner while remaining accessible to the intended user group?
- How can the existing *client application* change in order to use *tenant registry* instead of static content, while maintaining backwards compatibility?

1.3 Related Work

In this section, we will present one paper and one blog post within the same subject as our thesis.

The first related work is a Paper, from Song et al. (2019), *Customizing Multi-Tenant SaaS by Microservices: A Reference Architecture*[14]. This paper discusses tenant customization as a whole, inside a proposed multi-tenant system using microservices. It gives a list of principles to use when designing customizable microservices, to enable scalability while keeping isolation between tenants, some of which we follow. A few examples of good principles, which

we follow, are; *Each customization service runs on its own environment*, and *A customization service communicates with the product service and other customization services only via REST API*.

The second piece of related work is a blog post from Kubernetes themselves, which discusses different approaches to achieve multi-tenancy. The approach of using application-level tenancy at Schneider Electric Buildings, which enabled this thesis, involves modifying the entire software to handle multi-tenancy logic. This approach offers the highest level of scalability but requires a large amount of development time and a focus on tenant isolation logic. In a Kubernetes blog post from April 15, 2021, Bezdicek et al. (2021) provide three common tenancy models, which describe different approaches to tackling multi-tenancy using Kubernetes instead of application logic, with their drawbacks and advantages [13]. These are:

- Namespaces as a Service (NaaS): This method uses Kubernetes namespaces in the same cluster to separate tenants. This is a resource efficient approach as all tenants share the same cluster resources, but requires the right security settings.
- Clusters as a Service (CaaS): Here, each tenant gets their own complete cluster, offering good isolation but is not as efficient in resource utilization as NaaS.
- Control Planes as a Service (CPaaS): In this approach, some cluster resources are shared while others are not, putting it in between NaaS and CaaS in terms of scalability and isolation.

1.4 Contribution statement

The work presented in this thesis contributes to a broader architectural transformation underway at Schneider Electric Buildings. While many of the architectural decisions and other features were made by Schneider Electric Buildings, unless mentioned, the work presented in this thesis is our own.

1.5 Outline

This thesis is set up as follows: Chapter 2, Background, explains the technical knowledge relevant to this thesis. Chapter 3, User Interfaces, shows what the end product looks like and how users can interact with it. Chapter 4, Implementation, describes the steps taken to build everything. Chapter 5, Evaluation, briefly discusses the security of the system, and looks at UX user evaluation of the developed *support application*. Chapter 6, Conclusion, concludes the thesis and suggests ways to continue the development of our software.

Chapter 2
Background

This chapter will provide a background of some of the ideas and technology behind web services and Software as a Service platforms.

2.1 REST API

A REST (REpresentational State Transfer) API is a set of rules and best practices for building and interacting with web services[10]. It allows for software applications to communicate in a standardized way. REST APIs are designed around resources which one can do Create, Read, Update, Delete (CRUD) operations on. They are based on standard HTTP methods such as GET, PUT, POST and DELETE. The resource communication often use JSON (JavaScript Object Notation) or XML (eXtensible Markup Language) for transferring data.

REST APIs are stateless, meaning every request is independent and contains all information required to make that request. This ensures that neither client nor server needs to remember any state between messages, making REST APIs ideal for building web services as it allows for horizontal scaling, meaning performance scales with more machines.

2.2 SaaS Architecture

2.2.1 Single-tenant

Single-tenancy refers to a software architecture where a single instance of the SaaS and supporting infrastructure serves a single customer. In this model, each customer has their own dedicated resources, including databases, virtual machines, and application instances, ensuring data isolation and a high level of customization. Single tenancy provides greater control over the environment, allowing for easier custom configurations and specific customer re-

quirements. It comes with the drawback of being more expensive, with more complex maintainability, especially if the amount of users increases.

2.2.2 Multi-tenant

When talking about multi-tenancy, it is important to differentiate it from multi-user. A multi-user system would for example allow multiple individuals, often within the same organization, to access and use the system's resources simultaneously. In a multi-tenant system, multiple independent organizations use the same software platform, while being logically isolated from each other. Multi-tenancy is the capability of a SaaS application to answer to multiple tenants, through a single server and service instance, providing each tenant with a similar experience to having a dedicated server [8].

Multi-tenant Customization

To get the similar experience, multi-tenant infrastructure will have to provide a way to customize each tenant just like in a single-tenant instance. Tenant customization is a wide topic, and is considered as a key factor when designing a SaaS application [1], as different customers may require different services, data fields, interfaces, and so on. However, some type of standard customization is required. For example, they all need to be able to login into the software, so login information is required for all tenants. They will probably also require their own look and feel, i.e. styling. In single-tenant instances, customization could be directly made into the product source code. However, as they share the same instance as with other customers in multi-tenancy, all of the customization has to be managed by a microservice.

2.3 Docker

Docker is a Platform as a Service product that helps developers to package applications into containers. Multiple containers can be built to communicate with each other, but are isolated from each other and bundle their own dependencies. This means that it solves the "works on my machine"-problem by providing the software with all it needs, ensuring consistent behaviour between different machines[1].

2.4 Kubernetes

Kubernetes is an open-source platform designed to automate deploying, scaling, and managing application containers [2]. It has support for different types of container technologies, with Docker being one of the most popular. Kubernetes plays its role in the deployment and management of microservices within SaaS applications, as it provides an environment to manage these microservices. Kubernetes automates the scaling of stateless applications based on current load and ensures service availability, automatically replacing any application instances that go down.

[1] https://docker.com/
[2] https://kubernetes.io/

Kubernetes Namespaces

Kubernetes namespaces is a scope of resources which are managed inside a Kubernetes cluster. One usage of namespaces are the scenarios where multiple tenants share a Kubernetes cluster and need to keep their resources separate.

Kubernetes Pods

Kubernetes Pods are the smallest deployable units that can be created and managed in Kubernetes. A pod can consist of one or multiple containers, sharing IP address, and can find each other via localhost. Pods are used to run instances of applications or services.

Chapter 3

User Interfaces

This chapter will provide an overview of the interfaces developed or modified as a result of this project, showing how they work upon interaction.

First, we have the *client application* in section 3.1, an SaaS product developed by Schneider Electric, facing the customer. The *client application* enables users to monitor various values, such as sensor status and electricity statistics. In section 3.2, we present the *support application*, the application which should be used by support team to onboard customers. A higher-level overview can be seen in Figure 3.1 which showcases how the different applications communicate with each other, including their targeted user group.

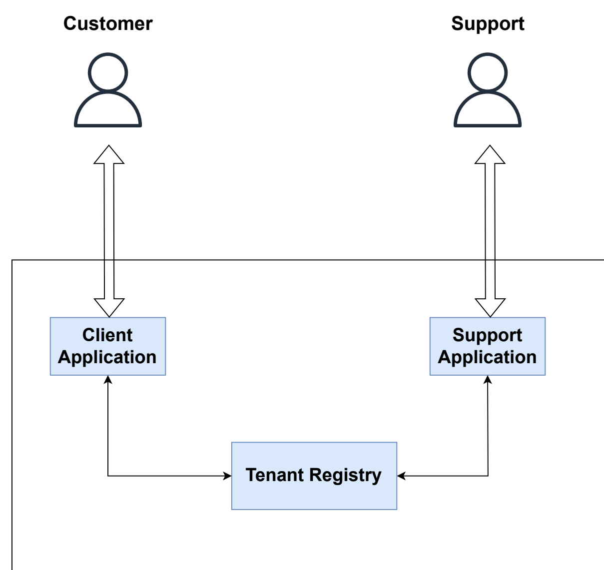

Figure 3.1: System Architecture

3.1 The Client Application

In Figure 3.2, a placeholder of the *client application* is displayed, showing the interface which customers interact with upon accessing their instance. It should be noted that the development of this application was made by Schneider Electric Buildings, and not by us. Our contribution involved modifying this application to integrate with the *tenant registry* for styling purposes, moving away from the use of static content. The visible customization include the icon in the top left corner and the color of the top menu.

In Figure 3.3 we can see the personalized styling applied to the *client application* during its loading phase. The color and images for the loading screen may differ from those we can see in the previous interface.

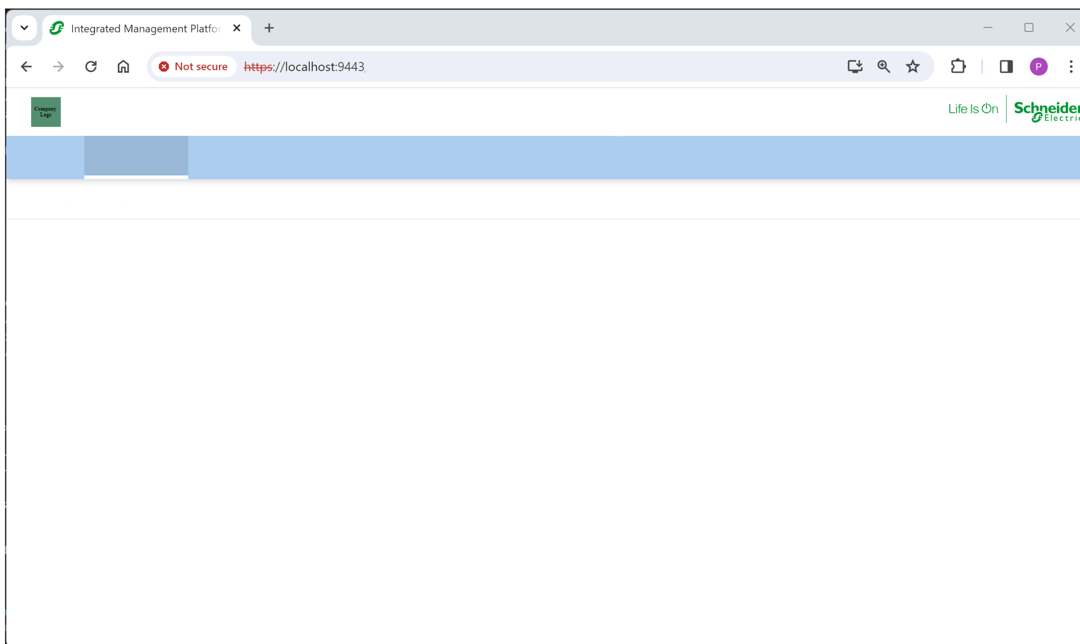

Figure 3.2: Client Application Interface

Figure 3.3: Client Application Loading

3.2 The Support Application

In Figure 3.4, we can see the initial page of the *support application* where we can onboard customers by interacting with the *tenant registry*. When interacting with the *tenant registry*, one would first have to choose, or add, the specific tenant for customization. The drop down menu we can see on the top left in Figure 3.4 displays a list of all current tenants, with the option to create a new tenant.

Customization options for tenant styling are shown in Figure 3.6, which represents the form used when creating or updating a customers styling. Should a customer already have styling, the form will pre-populate the default values for each element, otherwise, the form will remain blank. This applies for all of the tabs (Tenant Information, Styling and Authentication settings). The forms for creating and modifying tenants and their authentication settings can be seen in Figure 3.5 and 3.7. This application is our own work.

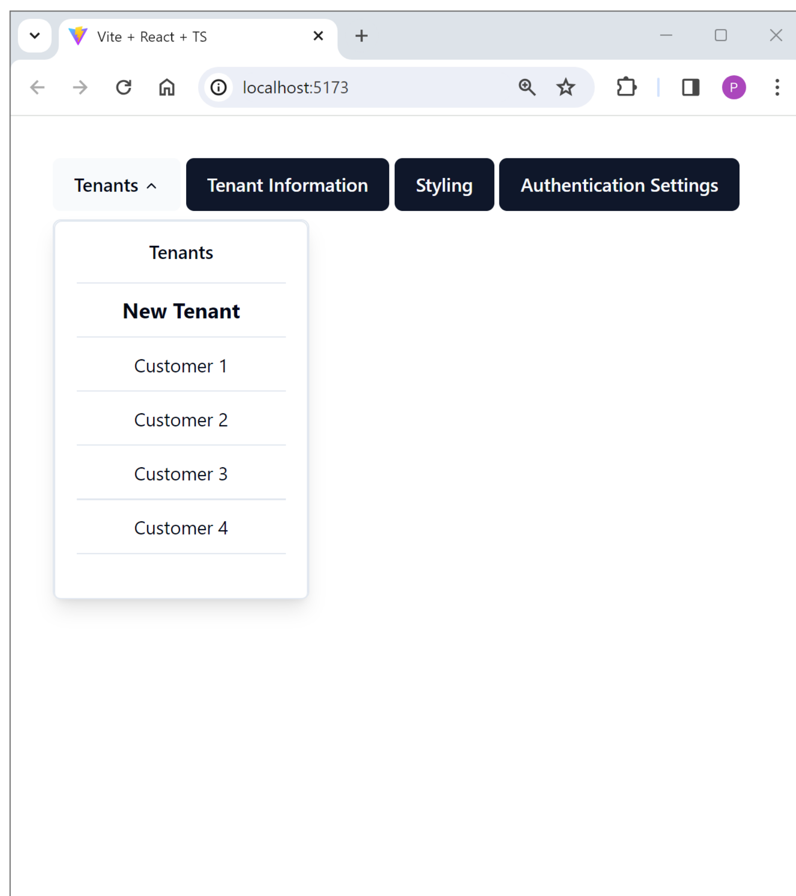

Figure 3.4: Support Application Selection Page

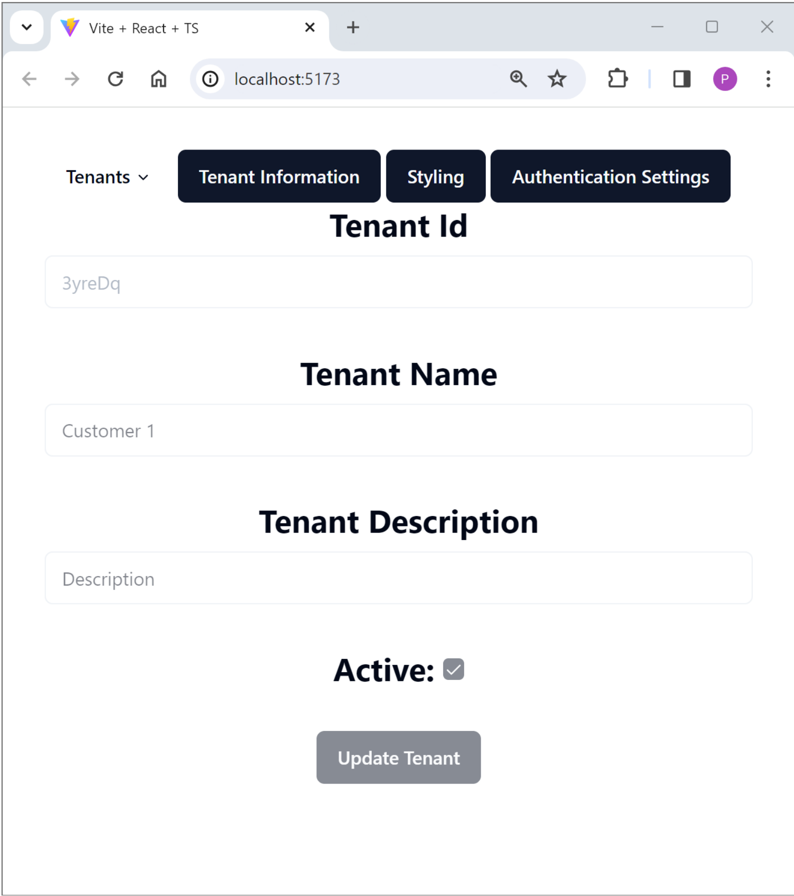

Figure 3.5: Tenant Settings Page

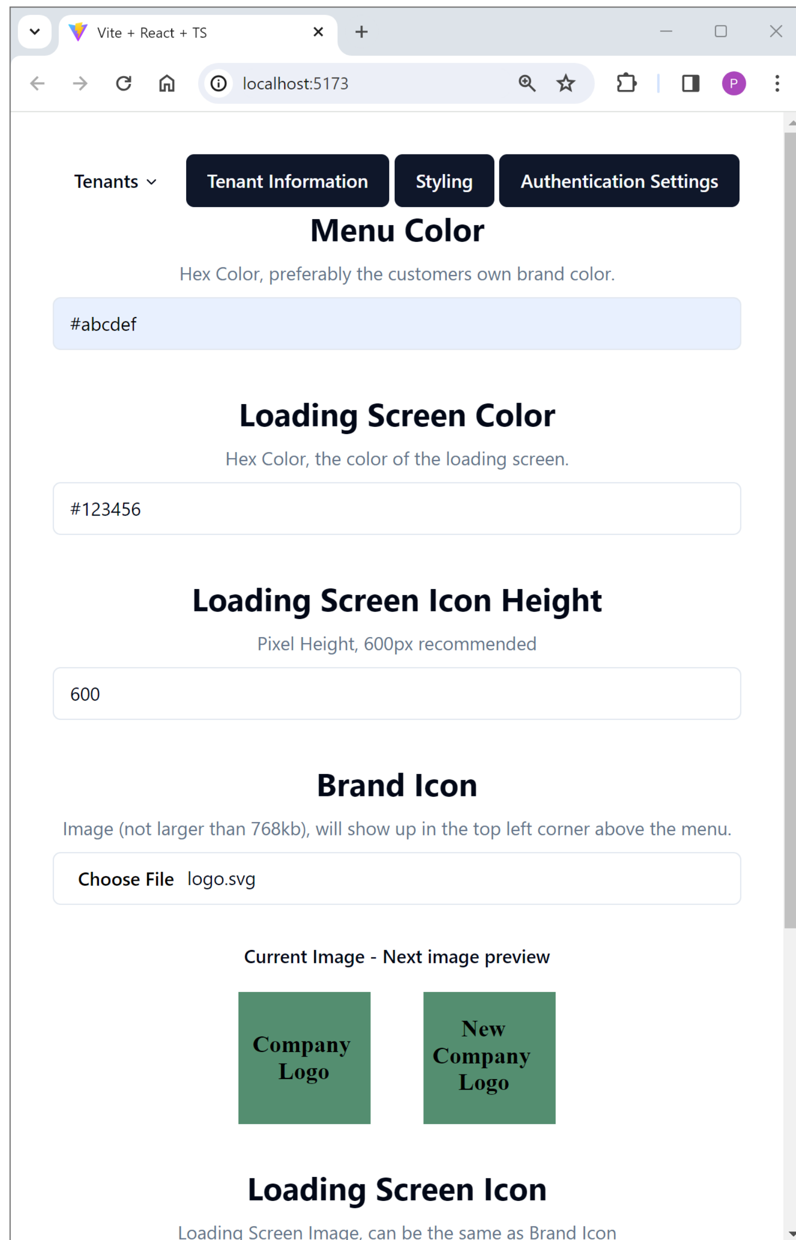

Figure 3.6: Styling Settings Page

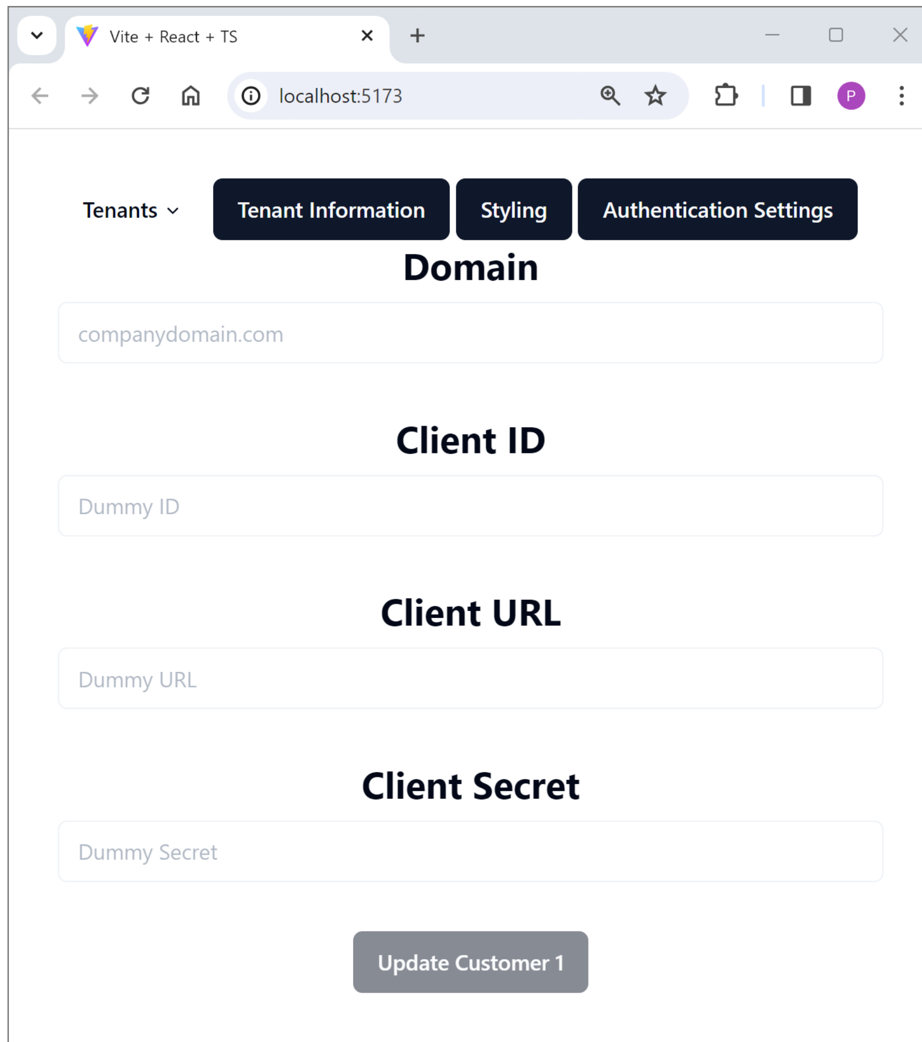

Figure 3.7: Authentication Settings Page

Chapter 4
Implementation

In this chapter, we will go through how the development process was made, and how the different applications work in the background.

4.1 Development Process

This thesis work was carried out within a team at Schneider Electric Buildings, located in Lund. We were on-site four days a week and worked remotely once per week. Microsoft Azure DevOps was utilized for project management. For code changes, the version control system Git was used. The code change process involved creating pull requests for proposed changes, each associated with a specific Git branch. Once completed, the team then had to peer-review the pull requests. This process provided us with insights into writing quality code and maintaining consistency across the codebase. Stand-ups were held every morning, where each team member described the work completed the previous day and their focus for the current day. These meetings offered an opportunity to discuss ideas with the entire team and receive feedback, ensuring that the development was heading in the right direction. Contact with the university was maintained on a weekly basis, offering valuable feedback on both the thesis work and the thesis itself.

4.2 Operational Logic

In Figure 4.1 we can see the architectural overview of the different parts of the system. The *tenant registry* is a service, running in the background, waiting for requests. The *support application* interacts with the registry by both fetching and posting information, whereas the *client application* will only fetch. Communication with the *tenant registry* is made through REST.

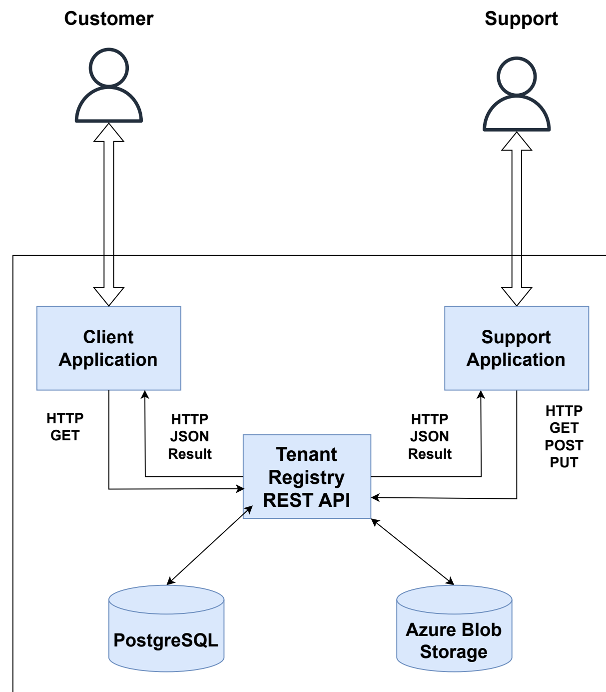

Figure 4.1: Tenant Registry Communication

4.2.1 The Tenant Registry

The *tenant registry* is implemented as a service written in the programming language Go. The service listens to a designated port, using an HTTP listener to handle incoming requests. Upon requests, it performs queries to a PostgreSQL database to retrieve relevant data for the request. The PostgreSQL database currently consists of three tables, one for tenant information, one for styling, and one for authentication settings. Additionally, the service accesses an Azure Blob Storage for image files, currently used with the styling customization, a choice made due to the scalability of blob storage. This also prevents the database from becoming overloaded with large customer image files. The retrieved data is then merged together and returned as a JSON object to the requester, either the *support application* or *client application*. A diagram illustrating this communication is provided in Figure 4.1. The tenant registry was written by us, and required just over 1100 lines of code in Go. The quantity of non-blank and non-comment lines of code was calculated using a Visual Studio Code extension named *VS Code Counter*.

4.2.2 Moving from CSS styling files to microservice

To be able to retrieve and set the custom styling, it is necessary for the *client application* to transition from using static CSS files to dynamically loaded styles. Our approach is with backward compatibility in mind, allowing for a gradual migration of customers to the *tenant registry* as needed, without forcing an immediate switch. The support for both static and dynamic styling leads to two specific scenarios within the *client application*:

The first scenario involves the loading page, the one seen in Figure 3.3. This page incorporates custom styling by calling utilizing the *tenant registry*. When first connecting to the *client*

application using a browser, a request for the servers index.html is made. The index.html will then reference a styling file, *style.css*, to set the loading screen styling. The server will check if the file *style.css* exists. If it does, then it will read it and return it as expected. If it does not, it means we are in a multi-tenant instance. The server will then proceed to ask the tenant registry for styling, generate the *style.css* file dynamically, and return it.

The second scenario addresses the application's menu color and brand icon seen previously in Figure 3.2, which are applied after the loading of the application content. As these components are being loaded after the loading screen response, we will require another request to the *tenant registry*, asking for custom styling again. Applying the custom styling in this scenario is fairly simple in and itself, with the use of document property, creating CSS properties dynamically. To maintain backward compatibility in this case, the application cannot simply check for the existence of a styling file as it does during the loading phase. Instead, it checks if the necessary CSS class is present using JavaScript's *document.querySelector*, specifically looking for an applied background image. If the image is present, it indicates the static file is loaded, and no further action is taken. If not, the *client application* applies the styling as fetched from the *tenant registry*, which is loaded at startup, regardless of the static file's presence.

The modification of the *client application* required around 200 additional lines of code, written in TypeScript and the web framework React.

4.2.3 The Support Application

The *support application*, which allows for smoother onboarding and customization, works by doing CRUD on the *tenant registry* through HTTP calls, either GET, PUT or POST. They are all made with the help of React forms, which are used for creation or modification for their respective setting. The form fields and images are validated using a library for React called Zod[1]. An example of evaluation of faulty input can be seen in Figure 4.2. To prevent multiple submissions, the "Update" or "Create" button is disabled temporarily after being clicked, until the POST request is completed. Once the request is finished, the button is re-enabled, and a notification (a toaster) appears to show the outcome of the HTTP request, indicating whether it was successful or encountered an error.

We have implemented the *support application* with around 1000 lines of code, written in TypeScript with the web framework React. It utilizes the Shadcn library for styling customization, and Vite as a build tool.

4.3 Deployment

Currently, the team at Schneider Electric Buildings is undergoing a step-by-step transition to multi-tenancy. Therefore, the deployment of the *tenant registry* and *support application* will proceed as part of a hybrid approach. The deployment process for *tenant registry* and *support application* remains relatively similar for both the current hybrid and true multi-tenancy scenarios.

In the present hybrid solution, new customers do not require a new virtual machine each. Instead, they connect to the same server. This server contains a deployed Kubernetes cluster

[1] https://zod.dev/

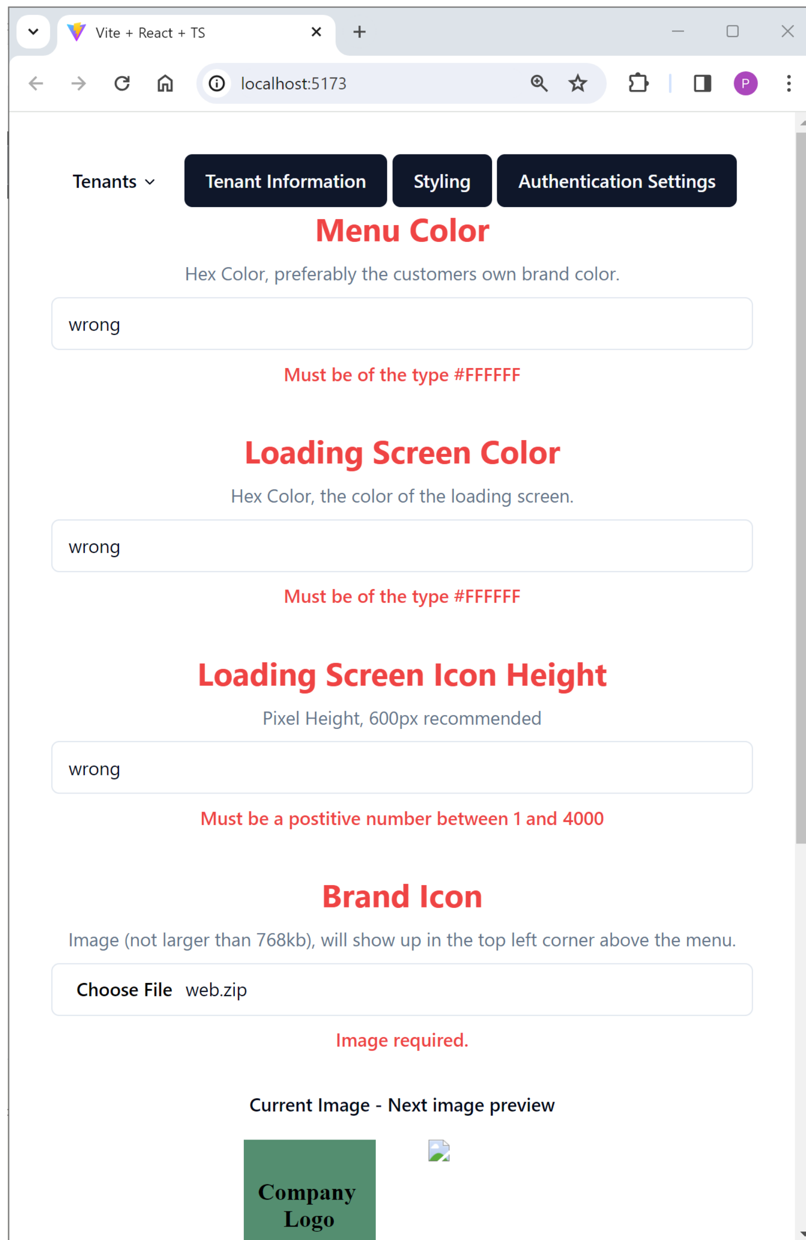

Figure 4.2: Form Validation Using Zod

on Microsoft Azure Kubernetes Service. A gateway redirects customers to their respective Kubernetes namespace. The Kubernetes namespace contains the single-tenant version of their *client application* inside Kubernetes Pods, which isolates their environment from others. The *tenant registry* and *client application* are also deployed into their own pods, outside of the *client applications* namespace. To deploy the *tenant registry* and *support application* in a Kubernetes cluster, they must first be packaged as Docker containers.

Communication wise, when the *client application* wants to speak to the tenant registry, it is forwarded to the tenant registry located outside the namespace, and the communication is handled thereafter. Support queries are directed through the gateway directly to the support application, which accesses the tenant registry directly. This entire process is illustrated in Figure 4.3. As the *tenant registry* is a REST API, it is allowed to scale according to Kubernetes demands, meaning multiple instances of *tenant registry* can be active at the same time.

Unfortunately, due to time constraints, we were not able to continue with the deployment in the sense of implementing this proposed solution.

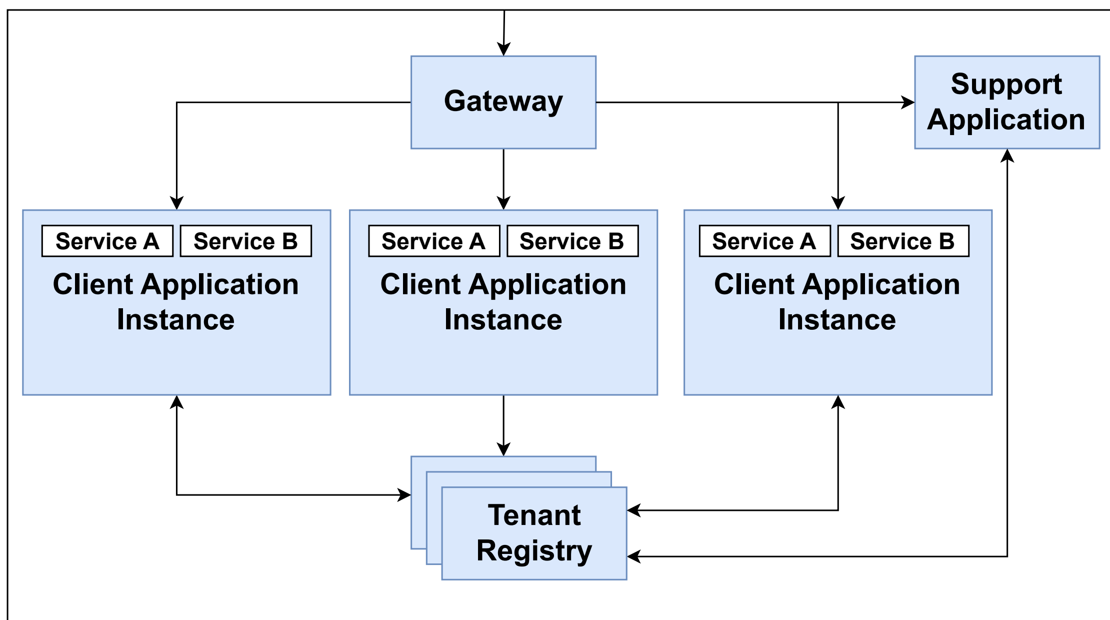

Figure 4.3: Placement of Applications in Kubernetes Cluster

4.4 Alternative Solutions

Some design decisions changed during the development of this product. This section will go through their advantages and drawbacks

Styling Storage in the Tenant Registry Using SQL: In our current solution, when storing the styling inside *tenant registry*, using SQL, there is currently one column per styling feature. I.e. the column "brandcolor" contains the menu color, and "loadingcolor" the loading screen color. One solution that was considered was to store the entire CSS styling inside one column in the database. It is an uglier solution, but comes with the benefit of adding a styling feature not being a breaking change, as adding more columns to a database schema requires

a database migration. This method was rejected because of maintaining code quality and the easier management of database migration in multi-tenant solutions, where a single migration works across multiple tenants. However, within a single-tenant configuration, one must manage migrations for every customer instance separately, making the other choice more appealing.

Image File Storage: In the current implementation, the *tenant registry* stores images in Microsoft Azure Blobs. The reason for this is to keep good practices and avoid storing unnecessary large data in the database. However, as image icons are not supposed to be large, being around 100KB-200KB for larger detailed icons, a discussion occurred if we should store the images in the database instead. Also, for performance sensitive systems, it is recommended to store data smaller than 250KB inside databases rather than as a blob[4].

Handling API Failures: In our current solution, if the *client application* is unsuccessful in its requests to the *tenant registry*, it will continue with a default styling. There was a discussion about whether the *client application* should instead terminate the session if it fails to retrieve styling, considering the more critical nature of the *tenant registry*, apart from the styling feature currently implemented. The decision to continue with default styling was made to maintain basic functionality and user access, with plans to re-evaluate that decision once more critical features are implemented.

Chapter 5

Evaluation

This chapter will go through a brief security discussion and evaluate the usability of the *support application*

5.1 Security

When communicating between services, it is good practice to pay close attention to the data being exchanged. If any of the transmitted data is intended for use in any kind of execution, it is advisable to sanitize the input to prevent security risks. We have two of those scenarios in our thesis work.

One of those are SQL injections, a method by which attackers inject malicious SQL queries into a database query, gaining unauthorized access to data. Since the data in the requests to the *tenant registry* are utilized in SQL queries, it is a potential concern. Luckily SQL injections are easily preventable. To counteract this risk, we avoid dynamic SQL queries and instead implement input validation and sanitation. This was made using Go SQuirreL package, utilizing its query builder to construct safe queries, thus mitigating the vulnerability associated with raw queries.

Another concern is Cross-Site Scripting (XSS), which works similarly as SQL injections, by inserting code into dynamically generated content, but in HTML instead of SQL queries. To prevent XSS attacks in our scenario, we ensure that the dynamic generation of *style.css*, our CSS file for the loading screen, utilize the Go html/template package. This package automatically sanitizes input strings, making them safe for insertion into the HTML/CSS and thus preventing XSS attacks.

5.2 Design Evaluation

In this section, we will explore the Usability of the *support application* through both an expert review and feedback from the targeted user group. We will use a combination of Usability testing using the Think Aloud method, and reviews by UX Experts, to yields better results [2].

5.2.1 UX Expert Reviews

UX Expert review is a review from someone with both experience and knowledge in creating UX applications, combined with experience from monitoring user interaction [2]. Given the broad scope of what an UX expert review should contain, we looked at Jakob Nielsens 10 Usability Heuristics [5, 11] and discussed a selected list of what we though would be relevant to the evaluation of *support application* specifically. The selected Usability Heuristics were:

Visibility of System Status: Keep users informed about what is happening with simple updates.

Match Between the System and the Real World: Make sure what users see on the screen matches what is actually happening in the system.

User Control and Freedom: Let users easily undo and redo their actions to fix mistakes.

Consistency and Standards: Maintain consistency, both internal of the *support application*, and also external consistency (like industry standards).

Help Users Recognize, Diagnose, and Recover from Errors: Use precise error messages, which should suggest a solution and avoid technical language like error codes.

Error Prevention: While clear error messages are beneficial, designing to prevent errors from occurring in the first place is always preferable.

The reviewer was a developer at the team with over 20 years of UI/UX experience. The reviewer provided ratings on the severity of the feedback, allowing us to prioritize improvements. The feedback from the UX Expert can be seen in Table 5.1.

Problem	Severity
Use URL routing instead of dynamic loading of each page	High
Too empty initial page, should be clear on what the first step is	High
It was too vague how to create a new tenant using drop-down list	Medium
The form alignment is unnecessarily large	Medium
Explain to a new user what all the different fields actually means	Medium
Re-route back to the tenant form after new tenant creation	Medium
Add the possibility to update tenant information	Low
The styling form doesn't validate brand icon correctly	Low
Allow for a color picker when choosing colors	Low
Specify what value the image height is (pixels?)	Low
Give the user feedback on which tab they are inside	Low
The need for showing the tenant id field is unnecessary when creating a tenant	Low

Table 5.1: Received Feedback from UX Expert

5.2.2 Think Aloud User Testing

The Think Aloud method is a popular way of evaluating a design or product [6, 12]. This method involves participants verbally expressing their thoughts and actions as they interact with the product. It provides insights into the users thoughts and checks their correlation with our expectations. It is important to keep encouraging the test subject to think aloud, as maintaining a monologue while working is unnatural.

This technique was selected to gather feedback from the intended user group, which, in this case, is the support team. We had a session with a support team member, providing them with the following tasks to perform in the *support application*:

- **Create a tenant**
- **Apply styling to the tenant**
- **Configure authentication for the tenant**
- **Updating tenant styling**
- **Update styling but with faulty values**

We conducted the observation, and took notes of their "thoughts". The user was encouraged to speak their thoughts freely while the test proceeded, without the pressure to filter their thoughts for the sake of appearing knowledgeable.

Key observations made during the testing include the following thoughts and feedback from the user:

- **What does *label of field* mean?**
- **Which is a good height for images?**
- **How can I see the styling results?**
- **Negative values of image height was accepted, which it should not allow.**
- **I like the error messages.**

The test subject successfully completed the task of creating a tenant with styling but struggled with creating the authentication due to a lack of understanding of what the authentication settings actually meant.

5.2.3 Evaluation Conclusion

Both of the evaluations gave us good insight of potential improvements, though they concentrated on different things. The UX Expert focused more on the user experience and intuitiveness, whereas the think aloud method provided us with a better understanding of the depth of explanation required for the program. For example, the UX Expert thought it was very vague how to add a new tenant, the support test subject had no issues with it. That does not mean we should disregard the UX experts feedback, but it could give us a reevaluation on

what changes are more urgent. For instance, the support test subject struggled with configuring authentication settings due to a lack of understanding, a detail the UX Expert review did not catch. It is important to keep in mind that this was evaluated with only one test user for each evaluation. If we had a larger test base, the test results would probably overlap more. Nevertheless, this shows the importance of receiving evaluation from both experts and users.

The Changes

Due to time constraints, we were unable to address the majority of the received feedback. Feedback which was deemed easy to resolve was addressed, such as providing explanations for different forms and names. We noted the remaining feedback for resolution during further development of the tool.

Chapter 6
Conclusion and Future Work

The work presented in this thesis showcases simplifying the onboarding process for multi-tenant SaaS applications. Through the creation of a tenant registry microservice and a suitable management tool, this work has addressed the possibility of tenant-specific configurations with the simplicity of a couple of forms. Throughout the development of this thesis, we wanted to explore the following questions:

Intuitiveness of the *support application*: The feedback we received from both evaluations provided us with valuable insights into on how to continue the improvement of the application. Although we unfortunately did not have enough time to address all the feedback, it gave us valuable insight on evaluation processes and the importance of using multiple reviewing methods.

Deployment to Kubernetes Cluster: While the current deployment is not a final multi-tenancy solution, our deployment proposal shows how one can utilize Kubernetes namespaces to maintain tenancy isolation, while deploying the service outside the namespaces, to allow for gradual migration.

Client application integration: As keeping backwards compatibility for the UI was a requirement, we explored the possibility of keeping backwards compatibility while integrating dynamic styling. We showed that maintaining backwards compatibility was possible in two different ways. Either by generating dynamic CSS files, or using JavaScript's document queries.

Future Work

Unfortunately, due to time constraints, we were not able to further improve the tool's usability. The UI/UX, in response to the feedback received, is an obvious choice for improvement in future work. Additionally, currently the tool supports three main features: tenant configuration, styling preferences, and authentication settings. Future development could aim to expand the tools capabilities within the *tenant registry*, like the addition of feature flags.

References

[1] Saiqa Aleem, Faheem Ahmed, Rabia Batool, and Asad Khattak. Empirical investigation of key factors for saas architecture. *IEEE Transactions on Cloud Computing*, 9(3):1037–1049, 2021.

[2] Aurora Harley. Ux expert reviews. `https://www.nngroup.com/articles/ux-expert-reviews/`. Accessed: 2024-03-08.

[3] Cor-Paul Bezemer and Andy Zaidman. Multi-tenant saas applications: maintenance dream or nightmare? In *Proceedings of the Joint ERCIM Workshop on Software Evolution (EVOL) and International Workshop on Principles of Software Evolution (IWPSE)*, IWPSE-EVOL '10, page 88–92, New York, NY, USA, 2010. Association for Computing Machinery.

[4] Jim Gray. To blob or not to blob: Large object storage in a database or a filesystem. Technical Report MSR-TR-2006-45, April 2006.

[5] Jakob Nielsen. 10 usability heuristics for user interface design. `https://www.nngroup.com/articles/ten-usability-heuristics/`. Accessed 2024-03-08.

[6] Jakob Nielsen. Thinking aloud: The #1 usability tool. `https://www.nngroup.com/articles/thinking-aloud-the-1-usability-tool/`. Accessed 2024-03-08.

[7] Jaap Kabbedijk, Cor-Paul Bezemer, Slinger Jansen, and Andy Zaidman. Defining multi-tenancy: A systematic mapping study on the academic and the industrial perspective. *Journal of Systems and Software*, 100:139–148, 2015.

[8] Rouven Krebs, Christof Momm, and Samuel Kounev. Architectural concerns in multi-tenant saas applications. *Closer*, 12:426–431, 2012.

[9] Lionel Sujay Vailshery. Public cloud application services/software as a service (saas) end-user spending worldwide from 2015 to 2024. `https://www.statista.com/statistics/505243/worldwide-software-as-a-service-revenue/`. Accessed 2024-03-08.

[10] Mark Massé. *REST API Design Rulebook*. O'REILLY, 2012.

[11] Jakob Nielsen. Enhancing the explanatory power of usability heuristics. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '94, page 152–158, New York, NY, USA, 1994. Association for Computing Machinery.

[12] Jakob Nielsen. *Usability Engineering*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1994.

[13] Ryan Bezdicek, Jim Bugwadia, Tasha Drew, Fei Guo, Adrian Ludwin. Three tenancy models for kubernetes. `https://kubernetes.io/blog/2021/04/15/three-tenancy-models-for-kubernetes/`. Accessed 2024-03-08.

[14] Hui Song, Phu H. Nguyen, Franck Chauvel, Jens Glattetre, and Thomas Schjerpen. Customizing multi-tenant saas by microservices: A reference architecture. In *2019 IEEE International Conference on Web Services (ICWS)*, pages 446–448, 2019.