Implementing a maintainable and secure tenancy model

Master of Science (Tech) Thesis
University of Turku
Department of Computing
Software Engineering
2023
Niklas Niemelä

UNIVERSITY OF TURKU
Department of Computing

NIKLAS NIEMELÄ: Implementing a maintainable and secure tenancy model

Master of Science (Tech) Thesis, 50 p.
Software Engineering
May 2023

Software-as-a-Service is a popular software delivery model that provides subscription-based services for customers. In this thesis, we identify key aspects of implementing a maintainable and secure tenancy model through analyzing research literature and focusing on a case study. We also study whether it is beneficial to change a single-tenant implementation to a multi-tenant implementation in terms of maintainability and security.

We research common tenancy models and security issues in SaaS products. Based on these, we set out to analyze a case study product, identifying potential problems in its single-tenant implementation. We then decide on changing said model, and show the process of implementing a new hybrid model. Finally, we present validation methods on measuring the effectiveness of such implementation.

We identified data security and isolation, efficiency and performance, administrative manageability, scalability and profitability to be the most important quality aspects to consider when choosing a maintainable and secure tenancy model. We also recognize that it is beneficial to change from a single-tenant implementation to a multi-tenant implementation in terms of these aspects.

Keywords: SaaS, Multitenancy, Maintainability, Security

Sisällys

1 Introduction

Software development using Software-as-a-Service (SaaS) model is a popular approach to creating customer products [1], [2]. The SaaS model allows for scaling up the product easily by design. Developing software products for customers introduces concerns for the maintainability and security for said products [3]. As the amount of customers increases, so does the data related to those customers. This can easily lead to high maintenance costs as an increasing amount of customers and data will raise the resource cost on hosting and managing the product. Administrative management can also become laborious if customers require customisation on their version of the product.

The implementation of SaaS products is often either a form of single-tenancy or multi-tenancy. Tenancy refers to the way the product handles the deployment and hosting for customers. Customers most often are referred to as tenants [3]–[5], but importantly tenants are defined individually per SaaS products as the customers can be either businesses or consumers.

Single-tenancy refers to the same model used in traditional software, where the software is deployed individually for customer [6]. In single-tenant models, the deployment and hosting of needed services can be done either by the service provider or the customer depending on customer needs and requirements. However, in the core of cloud computing and SaaS products multi-tenancy should be utilized to take advantage of the economy of scale [7], [8]. Bezemer et al. [3] claim that multi-tenancy as a model has two great benefits. It makes the deployment of a SaaS product much easier, and it improves the

utilization rate of resources.

Managing hosting and customer data can become arduous with a weak design choice for implementation. This thesis aims to research methods to identify weaknesses and strengths in a SaaS product tenancy implementation. We compare alternative methods to be able to assess potential benefits in changing already existing implementations to more fitting solutions. We use a case study to apply this research on an already existing SaaS product. The research questions for this thesis are as follows:

1. Research question 1: What aspects of developing and providing a SaaS product should be considered when deciding on a tenancy model?

2. Research question 2: Is a change from a single-tenant implementation to a multi-tenant implementation beneficial in terms of maintainability and security?

This thesis is composed using a literature review and employing information science research paradigm framework design science presented by Hevner et al. [9].

For literature review, relevant literature was searched for manually in research publication databases relevant to information technology. The search of references was not systematic, and they were picked from searching few of the largest research literature databases on computer science. Mainly works regarding SaaS, cloud computing, multi-tenancy, maintainability and security of such systems were evaluated. Unifying themes were identified and analyzed in this research.

Design science is an iterative process where the goal is to identify, solve and evaluate organizational problems in a continuous cycle. This methodology is presented to apply to existing systems as well, which fits our case study project. The case study product is a SaaS customer product *CT Publisher*, which is already public on the market. CT Publisher is a online portal for sharing product related data and facilitating after-sales. It is developed using C# using .NET framework and hosted in the cloud or locally using Microsoft IIS.

This thesis starts with Chapter 2 discussing different tenancy models in a SaaS product and providing explanation on the benefits of using any of them. The security matter of such products is discussed more in Chapter 3, exploring various measures taken in SaaS software to improve the security of them. Chapter 4 introduces the case study product CT Publisher, and discusses the current architecture and implementation in more depth. The chapter also sets goals and requirements for the product with maintainability and security in the focus. The case product is reviewed against these goals and requirements and a new, better fitting tenancy model solution is selected for it in Chapter 5. The chapter also runs through the implementation process of said new tenancy model. Chapter 6 discusses approaches to validate the maintainability and security of the implementation. In that chapter we also discuss the results of the validation and answer our research questions set earlier in this Chapter. Finally, in Chapter 7 we present the conclusions of the thesis along with discussion on future work possibilities on the subject.

2 Tenancy models

This chapter discusses tenancy models for a SaaS software delivery model. Tenancy model refers to the structure and implementation methods of tenants' data storage [10]. SaaS product owner provides the software to the tenant as a single-tenant or multi-tenant model, each providing differing advantages to the provider and the tenant. We discuss multiple tenancy models of these two types to explore the strengths and weaknesses between them.

Bezemer and Zaidman [3] define multi-tenancy as following: *"A multi-tenant application lets customers (tenants) share the same hardware resources, by offering them one shared application and database instance, while allowing them to configure the application to fit their needs as if it runs on a dedicated environment"*. We use the same definition throughout this publication. Single-tenancy on the other hand refers to having individual instances of the services per customer. However, as discussed further in this Section, it is possible to combine multi-tenant services with single-tenant services to gain various benefits.

As our case study product is built as a .NET Application and hosted through the Microsoft Azure service, the following Sections describing tenancy models use the Microsoft Azure documentation [10] as the primary reference.

2.1 Previous research

There are multiple studies done on multi-tenant applications, which all propose various types of frameworks to handle the multi-tenancy. Multiple studies focus on optimizing the performance in multi-tenant applications. Kalra et al. [11] propose a methodological framework for a microservice based cloud application for their case study. This approach tackles performance issues in a microservice level as opposed to application level. This allows for micromanaging and optimizing the performance cost at high level, moving tenants in and out of shards depending on perdformance. Mace et al. [12] also provide a framework which is implemented at the application level instead. Their framework aims to detect and throttle problematic tenants in a system, further distributing the resources more fairly across all tenants. Weissman et al. [7] conclude that SaaS as a model improves overall competitiveness of a product in the market.

The security is another main concern researched in multi-tenant applications. Bien et al. [5] present a security pattern blueprint for hierarchical data authentication and authorization. Their sample implementation is done in .NET environment similar to the CT Publisher case study environment in this research. Aytac et al. [13] propose a tenant management approach defining roles, access levels, ownerships, and permissions on a Internet-of-Things (IoT) product. This allows for easier management on huge number of objects introduced in IoT products, but the approach could be applied to similar products with huge number of entities as well.

Tang et al. [14] propose an multi-tenant role-based access control model that allows for secure cross-tenant access between selected tenants and resources. This model allows for fast and secure authorization on data access on collaborative cloud environments.

2.2 Single-tenant app and database

Serving an application and database for each tenant provides the best application level isolation [10]. Since every application is a standalone instance, tenants' applications never interact with each other. An application only needs one database for the single tenant it serves. Of all the tenancy models, single-tenant application and database provides the best tenant data isolation.

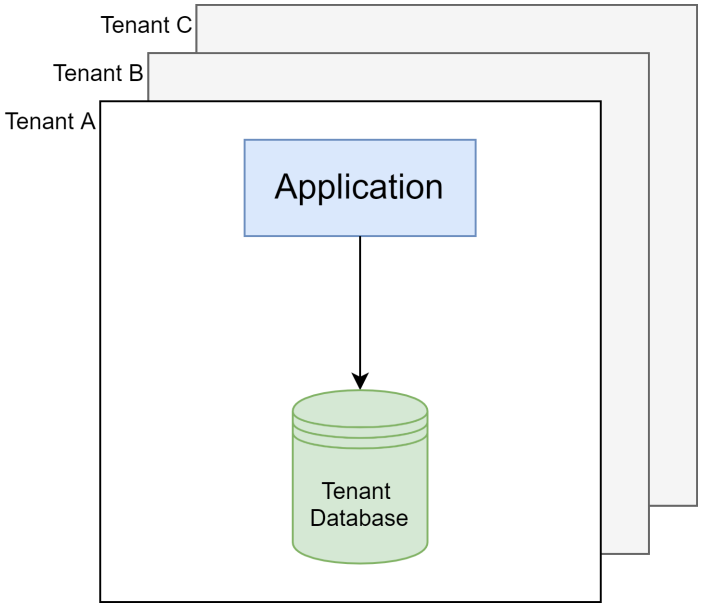

Figure 2.1: *Single-tenancy:* Application and database deployed once per tenant.

Since every tenant needs their own application instance and database, this is the most inefficient model in regards of resource usage. Every tenant needs to be allocated sufficient resources to handle peak loads, and at times resource usage can be negligible. Managing an application and database for every tenant becomes difficult to maintain when scaling up the number of tenants, hence proving the model to be the worst solution in terms of scalability.

Having a database-per-tenant model allows for broad customization per tenant. Providing customization for tenants is discussed in Section 2.6.

2.3 Multi-tenant app with single-tenant database

Instead of installing an application per-tenant like discussed in Section 2.2, the application can be installed once as a multi-tenant application to serve all the tenants. This model has the same level of data isolation, as tenants are still provisioned their own databases. To ensure the tenants are served their corresponding data, a separate database is used as a management tool to improve security and maintainability. Such a database is called the 'Catalog' (as seen on Figure 2.2), which stores and maintains mappings between the tenants and their data. This allows the application logic to route requests accordingly to the correct tenants' databases.

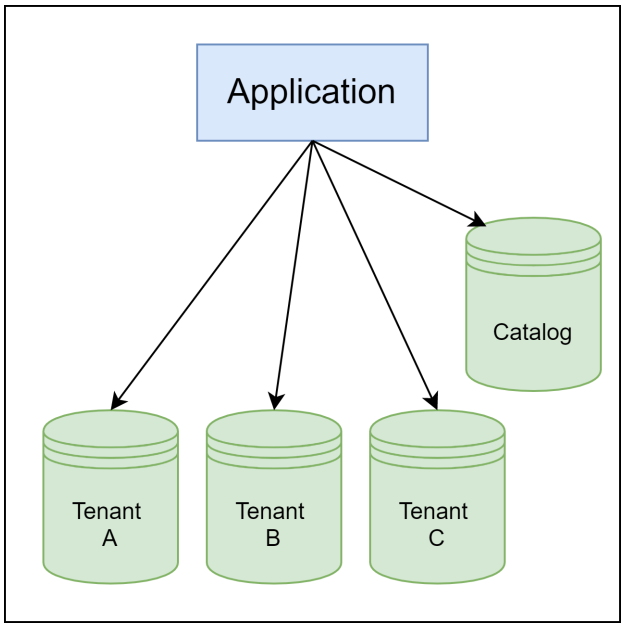

Figure 2.2: *Multi-tenancy* with single-tenant databases.

This model allows for broad tenant customization, as databases can have their schemas optimized for tenant-specific use. Maintaining a multi-tenant application is significantly easier than multiple single-tenant applications, especially when scaling up the amount of tenants. This allows for the multi-tenant application model to be a relatively scalable solution. The separate database instances for each tenant still allows for tenant-specific customization on the database level.

2.4 Multi-tenant app and database

Having a single app serve a single database is the core idea of multi-tenancy. It achieves the lowest cost-per-tenant as the product owner needs to maintain only one instance of the application and the database. To store and retrieve data of a specific tenant, a tenant identifier *tenantID*-column needs to be added to tables in the database. This enables *Row-Level Security* supported in SQL Database which is explained in Section 3.6. For further security, extra logic can be added to ensure that the correct tenant data is always shown.

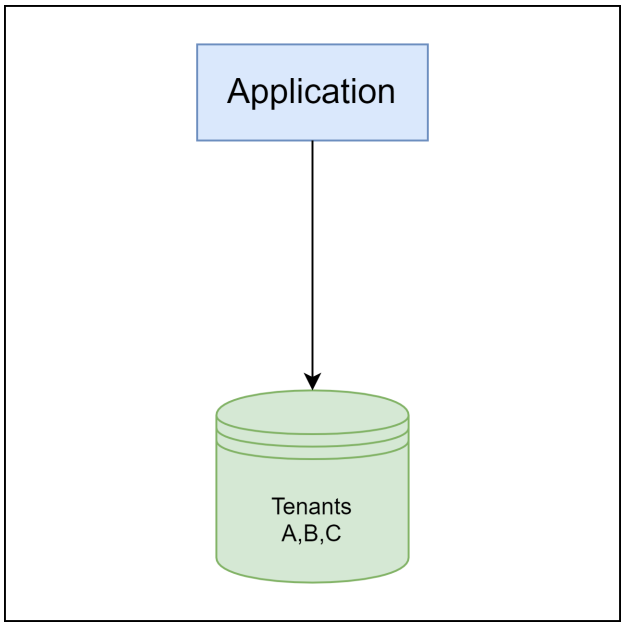

Figure 2.3: A simple *multi-tenant* application and database.

Having only one multi-tenant database serve the multi-tenant application means in addition to data, processing power is also shared between all the tenants. This might mean one overactive tenant hindering the performance of other tenants. As all the tenants use the same database, it is hard to monitor the performance from the database. The performance per-tenant can instead be scoped with application-level monitoring. A multi-tenant database can be scaled up by adding more storage and computing resources.

2.5 Sharded database

When the one multi-tenant database discussed in Section 2.4 becomes unwieldy to manage, multiple sharded databases are the solution. Database sharding provides the most scalable solution for multi-tenancy, as it can be scaled both vertically and horizontally. Sharded pattern merges the key principles of single database and shared database into one system. Since one multi-tenant database will not be able to maintain unlimited amounts of tenants, scaling out horizontally is done by adding more multi-tenant databases called *shards*.

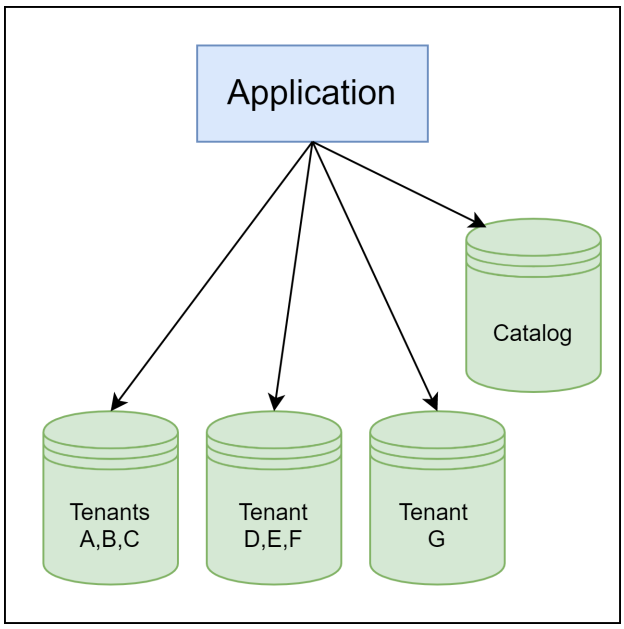

Figure 2.4: *Multi-tenancy* with database shards.

A sharded database consists of multiple multi-tenant or single-tenant database shards, which can each have one or more tenants' data stored in them. Being able to serve both multi-tenant and single-tenant solutions to tenants allows for more flexibility in for product owners providing the SaaS service. Tenants in the same shard share the storage and performance capabilities of the database. Tenants can be moved from shard to shard to balance shard performance, or to provide a single-tenant database for a tenant.

Tenants who require data isolation can opt in for the single-tenant database, where the

tenant is the only tenant in the shard.

2.5.1 Elastic pools

Microsoft offers the option to deploy databases in a same resource group called elastic pools [10]. Elastic pools allows for the resources allocated for the pool as a whole to be distributed differently across the databases in the pool. When one database sees lower usage, the allocated resources in the pool can still be used effectively by another database in the pool. Elastic pools are useful if it is not expected for all the databases in the pool to have their peak usage and resource drain at the same time. Accommodating resources per-database to handle their respective peaks in performance could easily take more resources than using elastic pools in this case, making them a cost-effective solution. The databases still preserve the data isolation benefits from the Multitenant Application, Single-tenant databases model.

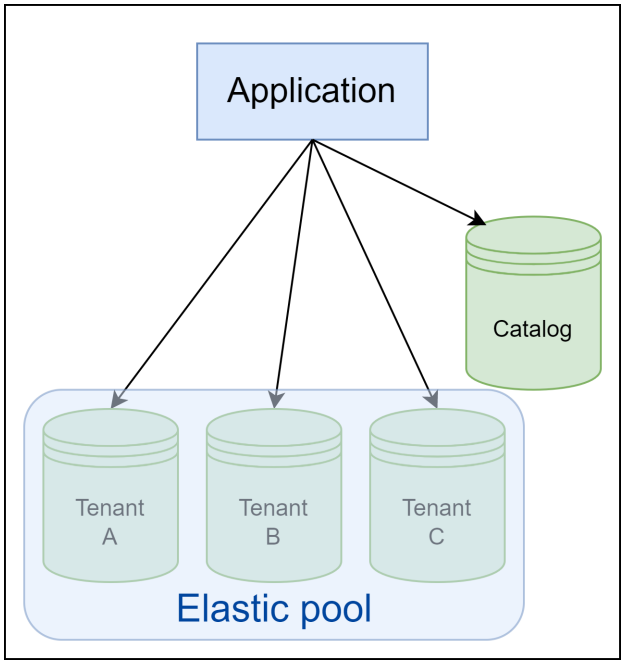

Figure 2.5: *Elastic pool* sharing the performance resources across multiple databases.

2.6 Configuration and customization

Sun et al. [15] conclude that the most ideal solution for a SaaS product owner to offer a standardized product every tenant feels comfortable using. Sun et al. define the difference of configuration and customization as following; configuration scopes small changes in predetermined behavior or looks of the application, where customization covers bigger features that need tenant-specific code changes. Mietzner et al. [4] identify the need for customization with also taking advantage of the economy of scale the multi-tenancy provides, providing tools to balance between the two.

Providing customization options, increased workload and maintaining different versions of code brings forth costs to the product owner that can be minimized by allowing for larger pool of configurable options for the tenants. Better configuration options allows product owners to meet more tenants' requirements allowing for maximizing profitability with no need to provide customized features. Depending on the competition in the product market, it can still be beneficial to offer customization to attract the tenants that look for that kind of product model. Allowing customization strays away from the core idea of a SaaS-product being a subscription-based pricing model. Despite this, the product can be distributed as a SaaS-product for an affordable price, and can be customized for premium price if the tenant so chooses.

Tsai et al. [16] discuss three ways to approach tenant customization; manual, automated and guided. Manual approach is defined by tenants manually making decisions per customization point. On automated approach, the customization choices are based only on tenants requirements, potentially not meeting all of tenants customization needs. The guided approach they propose automatically suggest key customization choices, and the choices will be selected manually.

Song et al. [17] investigate the possibilities of deep customization per tenant in a multitenant environment. Where in the past deep customization was done on the products code base directly, in cloud environments that's not possible as multiple tenants share the

same service instances. To ensure deeper customization possibilities for each tenant, they propose running custom code as intrusive microservices in containers not affecting the main services other tenants use. They acknowledge further research needs to be done to improve on and solve the problems the tight coupling their method introduces.

In practice, User Interface (UI) design is the most visible part of the configurations in the product. Research on designing the UI on the case study product CT Publisher can be found on M. Tuomola's master's thesis [18].

3 Authentication and authorization

In this chapter, we review literature on security, authentication and authorization practices in SaaS software. We discuss the known security issues in SaaS software, websites and APIs in general. We present some known malicious attacks SaaS developers need to take into account while developing their software not to leave them vulnerable. We discuss access control mechanisms in multi-tenant environments and in ASP .NET specifically to prepare for analyzing the case study product.

3.1 Security Issues in SaaS

Sandanayake et al. state that the security layer is very essential in SaaS software, as it provides the facilities of authentication and authorization, logging system and monitoring solutions [19]. They outline current trends in SaaS software and also identify some security issues. Chouhan et al. [20] identify three main categories in SaaS security challenges in the rapidly growing SaaS field: data security, application security and deployment security.

Data security Customers of the product are often highly concerned about data security. In a SaaS scenario, the hosting of the tenant data is no longer the customer's responsibility, but rather the responsibility of the SaaS provider. When a SaaS provider hosts the database in a multi-tenant instance, there are multiple customers confidential data stored on the same instance. As the data is no longer physically isolated, the data isolation aspect has

to be done though authentication and authorization. Data needs to be secured in a way that only authorized people can view the data they have permission to access.

Application security As SaaS application instances are often hosted in the web and viewed through the browser, they are vulnerable to typical website vulnerabilities like SQL injection, Cross Site Scripting (XSS) and Cross-site Request Forgeries [20], [21]. As Application Programming Interfaces (API) are used to power SaaS applications, their security is of utmost importance [22]. Díaz-Rojas et al. [23] mapped systematically the literature on malicious attacks on web APIs. They identified 68 different security threats but the methods most often mentioned were as follows: eavesdropping, leakage of sensitive information, code injection, denial of service attacks, man-in-the-middle attacks, API hijacking, replay attack, brute forcing credentials, and broken authentication.

Deployment security SaaS multi-tenancy is most often made possible using virtualization, which introduces security risks on its own [24], [25]. Wu et al. [26] state that virtualization brings a more complex and risky security environment. They conclude the following main security issues in virtualization: the break of isolation, remote management vulnerabilities, denial of service, virtual machine based rootkits and revert to snapshots problem.

Sharma et al. [27] have found that the information privacy issues in the cloud are increasingly needed. They found that where there are various privacy frameworks, none of them are a standard in the industry. This leads to various levels of protected data in cloud products like SaaS. They raise a few of the existing standards to their comprehensive list of privacy protection in cloud computing: FIPPs, CSA, NIST2020, NIST 800-53, GDPR and CCPA. They also suggest that separate standards need to be built to separate privacy from security.

3.2 Access control

Goyal et al. [28] define access as granting differential access rights to a set of users and allow flexibility in specifying the access right of individual users. Kazmi [29] define access control as the process that determines what activities a user is authorized to perform. However, often authentication and authorization both are included in access control [30]. The goal of access control is to deny access to restricted resources from users that do not have the right to access those resources. On the other hand, if a user has the right to certain resources, we must be able to provide those resources in full scope related to the users right of access. In practice, after proper authentication, the user is only shown resources they are authorized to access.

Solanki et al. [31] suggest that there are multiple of different access control models, but not all of them are fit for the SaaS cloud computing space. They cite the Multi-Tenant Role-Based Access Control (MT-RBAC) model proposed by Tang et al.[14] to be the basis for most access control models in use today. RBAC as it was first defined by Ferraiolo et al. [32] is defining all access through roles. The roles are defined to all users in the system as they would have them in a organization with differing levels of duties, responsibilities and qualifications. This is later standardized multiple times by NIST [33] to enforce three rules: role assignment, role authorization and permission authorization. These are the key principles to intra-tenant authorized communication in a multi-tenant SaaS system.

Authentication is the process of verifying the identity of a role [30]. Authentication is an imperative process in the access control procedure. Users need to authenticate themselves in order for them to gain any authorization over protected resources. Once a user has authenticated themselves and are authorized to certain resources, their actions on accessing those resources may be monitored and logged.

3.3 Protecting against ASP .NET specific attacks

Al-Amro et al. [21] identify security vulnerabilities in ASP .NET websites. They provide concrete examples of how SQL injection and XSS attacks would work. They propose a tool to check for such vulnerabilities. To this day, there is no standard set to protecting against these attacks, but there are unequivocal best practices to follow, which are presented in Microsoft's own documentation as well. They address protection against XSS as follows in five steps, all containing a form of input validation and output encoding [34]. They also stress that output encoding must be always done no matter the kind of validation and input sanitization has been done.

The Open Worldwide Application Security Project (OWASP) has guidance on how to securely implement data access [35]. They state that using Entity Framework is a very effective SQL injection prevention mechanism in itself, but other measures should be taken as well where possible. They also stress the importance of using available encryption algorithms and never trying to write your own encryption. Other important best practices include but are not limited to: always using HTTPS, using .NET Identity and using logging and monitoring.

3.4 Token-based authentication

In place of only using username and password to authenticate a user directly, token-based authentication allows for the user to authenticate with username and password to obtain an access token. This token now can be sent the to verify the users authentication, removing the risk of sending usernames and passwords over the internet, as doing so would be a vulnerability no matter how well they would be encrypted. A popular standard for token-based authentication is JSON web token (JWT), which is a compact format token used in scale on websites all over. JWT allows for sending identities in a standard format JSON object. We will be going over a authorization framework that utilizes token-based

authentication in Section 3.5.1. This framework doesn't specify the format for such tokens. JWT is widely used in web applications over all and in .NET built solutions as well, as it provides a lightweight format to the tokens.

3.5 Identity frameworks

There are multiple open identity frameworks like OAuth2.0 [36], SAML [37] and OpenID [38]. We will be focusing on the OAuth2.0 protocol specification, as it is the framework used in our case study product and widely used in projects using similar technologies.

3.5.1 OAuth2.0 Core (RFC 6749)

OAuth 2.0 is a authorization framework aimed to increase security of user data in client-server communication [36]. Göçer et al. [39] state that OAuth2.0 identity framework is the most used framework for authorization. OAuth proposes an additional authorization layer by implementing a resource owner approved *authorization server*. Following the OAuth2.0 protocol in an abstract example shown in Figure 3.1, client first requests an authorization grant from the resource owner (or indirectly from the authorization server). This can be done by authenticating themselves, for example, with username and password. After authentication, the users rights to access resources can be evaluated. If they do not have the rights to access any resources, an authorization grant will not be given. However, if the user is eligible for accessing some or all of the resources, they are given the authorization grant. With the authorization grant, the user requests an access token from the authorization server. Authorization server then valuates if the authorization grant is indeed valid, and returns a access token only if it is. This access token represents the authorization on the specific application and resources that user has the right to access. The access token can now be used to request data from the resource server. This access token is proof to the resource server that the user is authorized. The resource server then

validates the access token, and if valid, returns the requested resources to that user.

Access tokens have a certain lifetime given by the authorization server, and only works as a proof of access until that timestamp. A token can be stored and reused until it expires. The ability to store the access token securely is a strong benefit to using access tokens in general. As the token itself does not contain any user credentials, they can be stored and reused for some time. Comparing tokens to using only usernames and passwords, storing them for reusing purposes would leave them very vulnerable for malicious attacks. When an access token ultimately expires, a refresh token can be used to obtain a new access token. The refresh tokens are issued from the authorization server in tandem with the access tokens. A request containing the refresh token is sent to the authorization server, and again represents the authorization granted to the user. If the authorization server validates the refresh token, it returns a newly issued access token. Optionally, the authorization server can also issue a new refresh token.

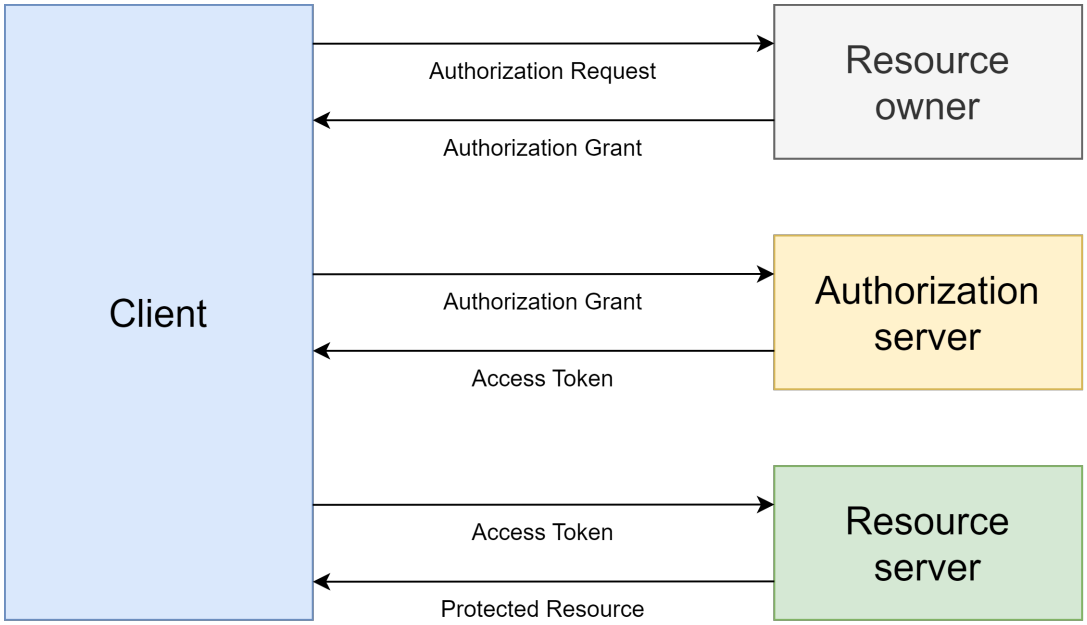

Figure 3.1: Abstract OAuth2.0 protocol in a simplified form.

3.5.2 Entity Framework Identity

Entity Framework provides an API for authentication in .NET applications, *Identity*. Identity is used to store and manage users, passwords, roles, claims, tokens and more. Identity is best used in tandem with a authorization server framework following a access control protocol like OAuth2.0, allowing for token-based authentication. Identity is a great way to set roles for users with various levels of access rights. These roles define the authorization level given to users. In other words, authorization defines the resources a user has access to. As defined in OAuth2.0 and explained in Section 3.5.1, Entity Framework provides all the methods for using a authorization server with access tokens and refresh tokens in .NET web API.

3.6 Row-level security

Row-level security (RLS) enables access control for multitenant databases in fine granularity. In a multi-tenant environment, tenant data located in the same database are filtered by adding a *tenantId* column to all tables. All database queries must use the *tenantId* value as a identifier when fetching data from the database. This allows for multi-tenant applications to create a policy that logically separates each tenant's data rows from every other tenant's rows.

More general usage of RLS may be to restrict access to data based on security policies, such as predetermined roles. RLS used in this way helps to achieve Principle of Least Privilege (PoLP) commonly enforced in information security. PoLP is a concept used to define a user should only be given access to the least amount of data that they specifically need to complete their workflow. Following PoLP improves overall security of a system as it reduces the attack surface susceptible for malicious attacks.

3.7 Data encryption

Data encryption is essential to secure the data sent over the internet. This allows us to protect restricted data and access tokens by first encrypting them, then sending them over the internet and then decrypting them to finally read the transferred data. The intention of encryption is to protect the data in case of a hijack, as a malicious attacker should not be able to decrypt the hijacked message. There are symmetric and asymmetric encryption methods as presented by Simmons [40]. Symmetric encryption uses one key to both encrypt and decrypt data. Asymmetric encryption uses two keys, a public key to encrypt and a private key to decrypt data.

Asymmetric encryption, or public key encryption, provides a more secure approach to encryption than symmetric encryption, albeit with the cost of taking more time. It might be beneficial to combine both types of algorithms to create even more secure, hybrid encryption [41]. However, it is common in public to use them both as is, but for separate functions. Symmetric encryption is better used to encrypt large amounts of data, and asymmetric encryption is used to encrypt smaller chunks of data, like an HTTP request.

.NET provides multiple implementations of many standard cryptographic algorithms, both symmetric and asymmetric, like AES and RSA respectively. These can be used to encrypt data like access tokens in .NET applications.

3.8 Security standards and policies

There are several third party organizations that provide a set of standards, guidelines and frameworks for secure data management and consumers right to access their data. *Compliance* is the act of ensuring your organization and products are using said standards. Compliant software satisfies customer needs and pose advantages with competitors as stated by Moyón et al. [42]. We already have mentioned The National Institute for Standards in Technology (NIST) [43], which publishes and upkeeps various security stan-

dards considered industry best-practices. The International Organization for Standardization (ISO) [44] is also known for providing some of the most-used security standards. OAuth2.0 [36] and JWT discussed in this chapter are very popular authorization frameworks. Complying to these kinds of widely used security standards will increase the perceived security of one's organization and product [42].

There are also policies and laws in place that protect people's data, and as such, are very important to abide by. An example of such law is General Data Protection Regulation (GDPR) [45] for organizations that target or collect data related to people in the European Union (EU). As it is a law, GDPR compliance is a must in security measures across the industry.

4 Case Study: CT Publisher by ATR Soft

This chapter discusses the background of the case study product CT Publisher with its current architecture and implementation. We identify current problems in CT Publisher and outline the goals we want to achieve for the product in the scope of this thesis work. We define requirements for the product in terms of the aforementioned goals and the research questions set in Chapter 1. These requirements will be used to drive our decision making on a new tenancy model and implementation outlined in Chapter 5.

4.1 Background

ATR Soft is a full-service software development company, which produces software to support the business of companies [46]. The case study product development we work on this thesis is on their product *CT Publisher* [47]. CT Publisher is an online portal for sharing product related data. Use cases can vary per-customer, as CT Publisher is tailored to fit each customers' needs separately. One of the main uses for the product is after-sales service in the form of providing maintenance info and sales of spare parts. Currently, customers are able to choose for the product to be fully provided and hosted for them or they can opt to host it by themselves on-premises. The product follows the SaaS model allowing the sales of the product on a license-basis.

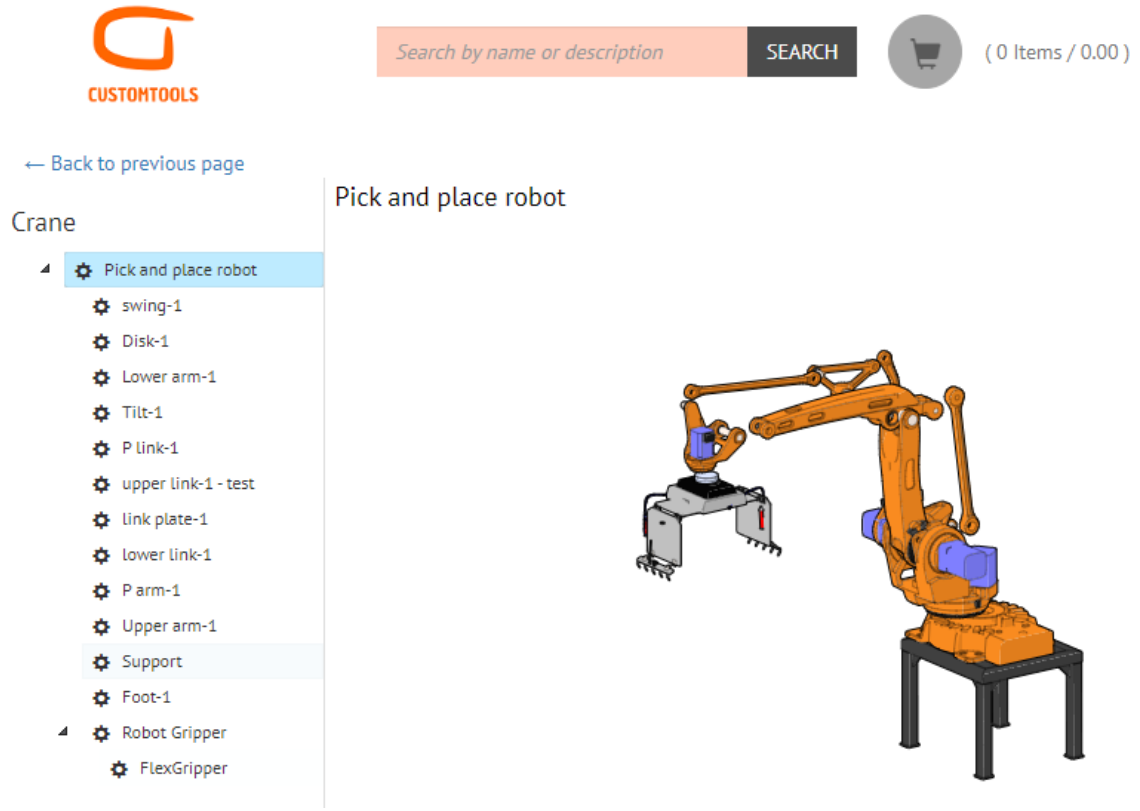

Figure 4.1: An example 'Pick and place robot' product shown on CT Publisher demo.

CT Publisher in theory can be used as any type of online store in general thanks to its high customizability. Customers can generate interactive graphics from CAD files with clickable hotspots as seen with an example on Figure 4.1. This view can be displayed on the website to aid with sharing product related data. Each part is interactive and reveals further details of the part when interacted with as seen on Figure 4.2. This makes specific parts easy to find on the website, allowing for viewing documentation and purchasing replacements effortless. All these features also come in a CT Publisher Editor desktop application, which end-users get access to when purchasing the product. The Editor can be used to manage all customer data, like managing prices and updating manuals and instructions.

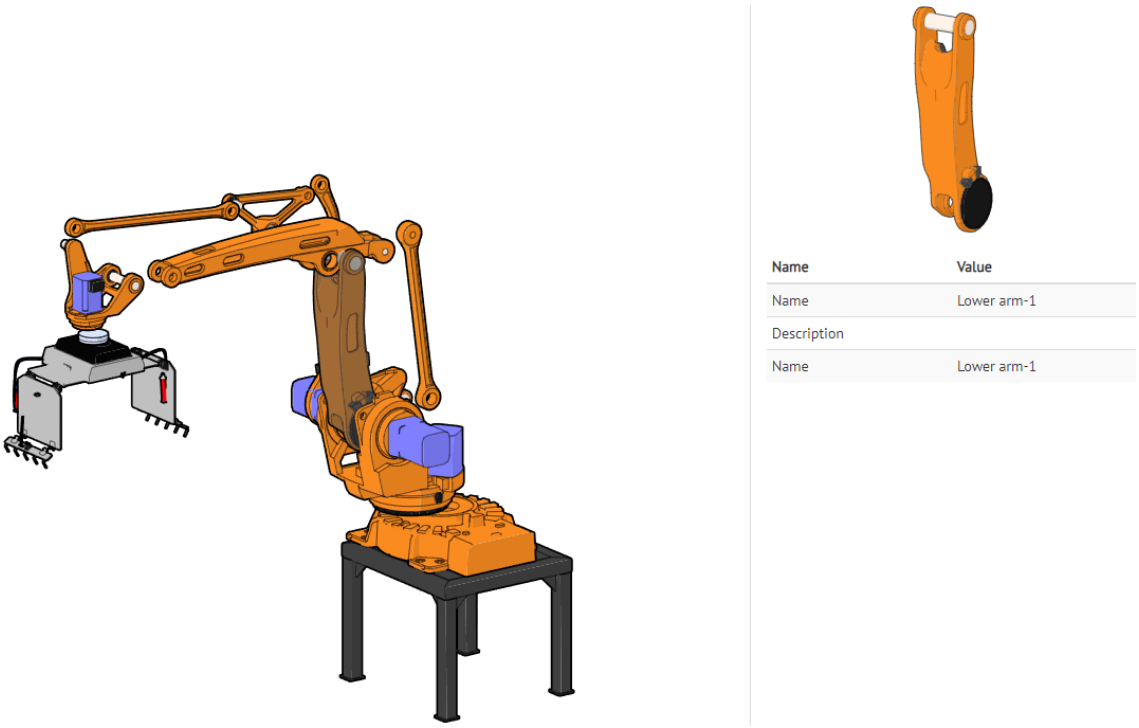

Figure 4.2: Interacting with a part will allow you to see further details on that part.

4.2 Current solution in CT Publisher

This Chapter outlines the technologies used in the CT Publisher product development and hosting.

The CT Publisher product is developed using C# and ASP.NET Framework. The .NET Framework allows for the use of the MVC (Model-View-Controller) design pattern conceived originally by Reenskaug [48]. This follows the design principle separation of concerns, breaking up the code into chunks with independent logic for completing a simple task each. Separating the application logic into models, views and controllers, provides a effective approach to updating, testing and debugging code.

The project also utilises Entity Framework [49] as the data access technology. It is an object-relational mapper for .NET which enables users to interact with data through .NET objects representing the application's domain. EF introduces a "Code First" -workflow to creating models through C# classes and creating databases using these models.

The hosting of the needed project instances is implemented through Microsoft Internet Information Services (IIS) [50]. The use of IIS is integrated in ASP.NET which allows for easy development and deployment of the site using the included pipelines. IIS also has native C++ extensibility to build solutions beyond the application layer. This makes it possible to develop custom application components like authentication schemes and monitoring and logging. IIS allows for servers to be hosted natively on Windows machines or Azure cloud services. This allows for customers to host their services on-premises if they so require. This however makes updating the services much harder, as such services are no longer managed by the product owner. Cloud hosting provides more maintainable hosting for the product owner, and is the preferred way to host all product services.

4.3 Current problems

Current system hosts all the tenants' databases on one server, where every tenant has their own database. In addition, each tenant has two hosted services associated with them; an API and a website. Therefore updating release versions means upgrading three different entities per tenant. This quickly becomes a large time sink as these updates are done manually. Agile development on the CT Publisher product provides increments to the product every four weeks. These updates take administrative work linearly dependent to the amount of customers. Each new customer added to the management increases the manual labor on each update process.

The problem is aggregated in the cases where customers have opted to host the product themselves on-premises. On these cases, we rely on the customer to update to the new versions on their time. The leading reason why customers have wanted to host CT Publisher themselves is their concerns for security in a shared multi-tenant environment. Customers need to be assured that their confidential data is not leaked to any other customers. These customers have indicated that providing successful security tests would

entice them to opt to a multi-tenant system.

Kappes et al. [51] define access control goals with five key points: isolation, sharing, efficiency, interface and manageability. We can adapt to use the same goals in our project as follows.

1. *Isolation:* Tenant is able to choose identities for its users. Tenants need to be certain that their data is shown to only their users.

2. *Sharing:* Provide flexible access control if needed. Allow access control to be defined as role-based within tenants organization.

3. *Efficiency:* Access control needs to support performance and scalability in all cases.

4. *Interface:* Access control must allow for wide customization possibilities for tenants.

5. *Manageability:* Access control must not interfere with effortless administrative management.

4.4 Goals

In this chapter, we will define some general goals we want to achieve in the new implementation. More specific requirements are set on the basis of these goals in Chapter 4.5.

Effortless deployment of CT Publisher demo. ATR Soft wants to provide easy trial access for potential customers with a CT Publisher demo. With current single-tenant application and database tenancy model explained in Section 2.1, it takes excessive work to setup and deploy a demo environment for every potential customer. The overhead deployment time needs to be reduced to minimal to maximize time-efficiency, and by extension, the profit of the product. It is not sensible to put the same resources into

building a trial version as it is to build it for a confirmed, paying customer. The goal is to provide an advantageous solution for both the customer and product owner with the SaaS model, where the customer is seeking for a low-cost product and the vendor is looking to maximize the profit [52]. For that, creating an easier process for creating trial versions for customers satisfies is essential. Further providing the full product from the trial version should be made fast and simple for each party.

Develop existing solution for better maintainability. The current single-tenant solution provides excellent customizability. Like discussed in Section 2.6, customization options can be generalized to be offered as configurations instead. The goal is to offer a simple and straightforward, yet configurable, product for customers, while having the maintenance costs low for the product owner. Current configurations offered through a custom add-in service should be preserved. Manual work for updating and maintaining the product needs to be minimized.

Improve and measure security. The main reason a customer chooses to host the product on-premises is the concerns for security in a multi-tenant system. We need to provide sufficient proof of security to such customers. Such proof could be architectural measures done to allow secure data flow, as well as results from successful test cases.

Scalability and costs to allow for profitability. In essence, the main goal of a SaaS provider is to provide a service profitably. There needs to be an appropriate return of investment (ROI) against the cost of developing and providing the service. Even though multiple tenants share a multi-tenant application, an individual tenant will expect the application to be scalable and be able to meet their level of demand. At the same time, we as the service providers need the product to be scalable in order to properly account for increasing amount of customers and users. We need to be able to bill the tenants appropriately. Whilst a fixed rate is the simplest to implement, a usage-based pricing model

would suite better. Resource usage may need to be monitored for such pricing model. In addition to usage-based pricing, we could provide varying pricing for varying levels of service. These levels could included and not be limited to, different functionality, different usage limitations, have different service level agreements (SLA) or some combination of such factors. Revenue from the application must be sufficient to cover both the capital and running costs of the application.

SaaS security compliance Customers should be able to easily determine our SaaS security compliance of industry regulations and frameworks. This makes it possible for the customers to ascertain if the product security is in their required standard. This increases the credibility of the service providers overall and is a certificate of a certain level of security. This compliance would help prevent us losing any potential or existing customers.

4.5 Requirements

In this chapter, we will set the requirements we will strive to achieve in the new implementation. These requirements will be referred to when deciding on which tenancy model to implement, and will be guiding the implementation process throughout. These requirements have been decided with the CT Publisher project team members and approved by the product manager. Our research questions defined in Chapter 1 also influence defining the requirements.

A total of four requirements were set to guide us on the project. These requirements will be assigned abbreviations R1–R4 for ease of referring later in this document.

4.5.1 R1: Maintainability

The product and its trial version must be easily maintained to accommodate for an increasing amount of customers. In practice, any operation on application instance or database

must be easy and fast for the product owner. The per-tenant cost to maintain the product must stay low even when the product gains more customers and users.

4.5.2 R2: Tenant security

The customer data must be protected with appropriate measures to achieve satisfactory tenant security. As CT Publisher has multiple customers, we as the vendor have a responsibility to ensure that tenant data is sent and shown only to the correct tenant. This must be done through proper tenant authorization handling through all the application instances; website, web API and desktop application. This restricts the data flow correctly so that a data request from one tenant cannot be authorized and hence shown to a application user from another tenant. This authorization must also allow only minimal relevant data shown to the tenants in relation to their user role in the authorization system. These roles should generally reflect the user's position in the customer workflow. This per-product restricted access is an important data security requirement to allow the customers to manage and serve their data securely and accurately within their organization.

4.5.3 R3: Customer integration

A fast and simple deployment process to provide a demo version of the product for trial customers and new customers is required. The trial version must be able to be produced with minimal resource cost as to allow for plentiful deployments of these whenever needed for customer acquisition. The trial version must demonstrate basic product functionality that produces sufficient user experience to get a good understanding of the product workflow. This demo version of the product can be then extended to already acquired customers to test out features they might want in their final version.

4.5.4 R4: Performance

It is decided that the activities of other tenants must not affect the availability, performance or scalability of the service. As a tenant of the product, you must have the required resources allocated to achieve a beforehand decided desired performance levels. This requirement is aimed to satisfy the customers' desire to have their applications running smoothly with minimal performance drops and downtime.

5 Chosen solution

In this chapter, we go through the process of choosing a solution based on the requirements set in Section 4.5. These requirements are used to guide us through the selection process, weighing in pros and cons in every tenancy pattern. Then we discuss the implementation of a new model and assess how the change affected satisfying our requirements.

5.1 Scope

In the course of this section, we refer to the specific requirements we set in Section 4.5. The abbreviations R1–R4 set in that section are now used to refer to the four requirements respectively. These requirements are then used to assess the different tenancy models introduced in Section 2. The research questions are also tightly kept in mind when assessing tenancy models for the new solution.

5.1.1 Current solution: Single tenant application and database

Single-tenant app and database, discussed in Section 2.1, is the current solution deployed for the product. The problems encountered through the use of this model drew attention to looking for another tenancy model and implementation for the product. The greatest pitfall of this model for this product was not satisfying *R1: Maintainability*. It quickly became apparent early in the product's life cycle that maintainability could easily be a bottleneck when scaling up the product. As seen in Figure 5.1, hosting three different

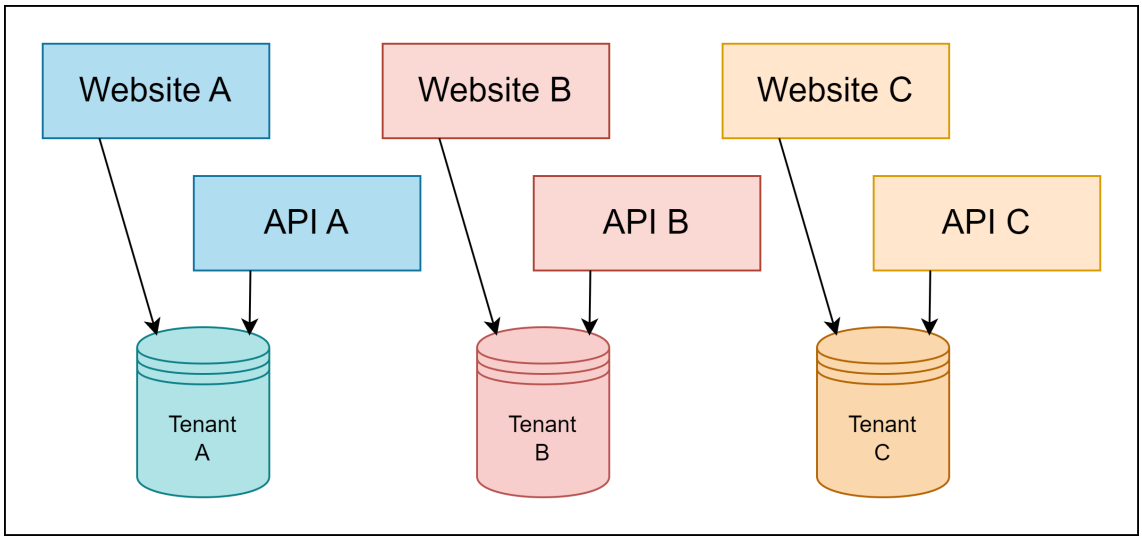

Figure 5.1: Website and API services are instanced and hosted once per tenant in the current solution.

services per customer, a website, a database and a web API quickly became strenuous to update manually. This leads to slow upgrading of customers' systems or more infrequent updates altogether all the while being costly for the product owner due to the time spent upgrading the services one by one.

R2: Tenant security is satisfied with the single-tenant model. Separate databases per customer or tenant is a highly significant architectural security design choice [53]. Like with all architectural choices for the database solution, this assumes a proper implementation of user roles. Satisfying this requirement was a high priority from the very start of the project development cycle.

R3: Customer integration is not very efficiently met in this model. Creating a trial version for customers violates *R1: Maintainability*, as it takes hosting three different services to get the trial up and running. As setting this trial up takes time in itself, customising the trial further than the bare minimum functionalities and themes might not prove worth it for the customer acquisition.

Single-tenant application and database is the most independently managed tenancy model per-customer, affecting *R4: Performance*. When allocated enough resources, the

customers are not hindered by any other customers. This resource allocation can be costly per customer though, making this possibly the most costly option of all the tenancy models [3]. In the current state of the product, customers can even choose to host the product themselves. This transfers the responsibility of managing performance from us, the product owner, to the customer.

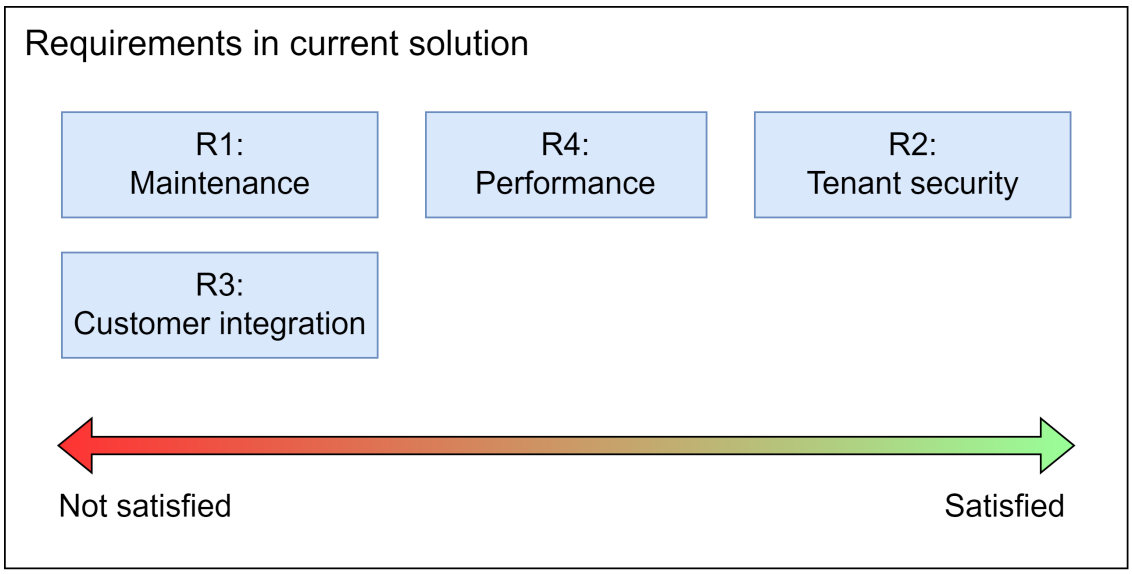

Figure 5.2: *Requirements* on a scale of satisfaction in the current solution.

Assessing the product through the requirements *R1–R4*, it becomes obvious that the current implementation does not meet them all on sufficient levels. This can be roughly visualised in the Figure 5.2. *R2: Tenant security* is the only one of these four requirements being satisfied properly, while the other three clearly struggling to be met in a sufficient manner. This confirms the evident need of restructuring the product architecture in a way that allows all four of the requirements to be satisfied. This will be mainly done by choosing and implementing a more suitable tenancy model. After such a model is implemented, if needed, it should allow for further improvements in the product in relation to the same requirements.

5.1.2 Choosing a new tenancy model

When considering which tenancy models would satisfy *R1: Maintenance* the best, the multitenant app and database would seem the most efficient tenancy model available [3], [13], [53]. To update the services, a total of three updates is sufficient to update all services for all clients. This would cut the time to update the product to a fraction of what it currently is. Extending this model would be to choose the sharded databases model, which allows for even higher amount of tenants while making it more reliable and easier to maintain. On the topic of using multiple databases in the solution, it would be possible to take advantage of elastic pools. Elastic pools could however be detrimental to satisfying *R1: Maintenance* on the cost of saving resources. The added maintenance cost could be of needing to adjust the resources given per certain elastic pools; if the databases in a same pool hit the cap of resources allocated, all the database operations in the pool would slow down. This could be solved relatively easily by monitoring and readjusting the resources per elastic pool as needed.

Multi-tenant app with single-tenant database is another noteworthy option. It allows for updating the product website and web API instances once, but then still leaves the customers databases the need to be updated separately. This at face value would seem to cut the time and resources to update the product by almost a third, but it is likely to be even bigger gain than that. The database instances might not need updating every time the product is updated. In other words, usually when updating, the increments in the product are to the website and web API instances which would need to be updated only once. Already at this point, the multi-tenant application and singe-tenant database model seems like a strong option.

Satisfying *R2: security* on an architectural level is already optimal in the current solution from a data storage standpoint [3]. The benefits of the data isolation on separate databases is a huge advantage compared to the models with a form of multitenant databases [13]. Restricting the data access accordingly between the customers and ten-

ants becomes a question of secure data logic, encryption and transfer [53]. *R3: Customer Integration* is rather easy to satisfy using multi-tenant application and single-tenant database. When a demo database is already prepared, a simple trial version to showcase the functionality is as easy as in a fully multi-tenant system. If satisfied with the demo trial version, a database needs to be created with their respective data. Single-tenant databases model allows for the customer to affect the form of the data stored.

R4: Performance per tenant could be sacrificed using a full multitenant model, as the tenants share all the instances between them. Using single-tenant databases, the amount of requests done to the database is guaranteed not to affect other tenants. Performance of the application and web services can vary as all the tenants are using the same instances. This can be avoided with sufficient resource allocation in hosting them.

To summarize on the set requirements, we can decide on best approaches per requirement as follows:

1. R1: Multi-tenant application is the simplest to maintain. One deploys it once, and once its running, one does not need to worry about it.

2. R2: Data security is a big concern for the current tenants of the CT Publisher product, and the demand for data protection is on the rise. To best satisfy future customers, even the demo environment must have proper data protection. Proper application level security will be applied, but single-tenant databases provide data isolation per tenant.

3. R3: We want to provide all the same features in the demo version as in the version current tenants have. This is why a multitenant app is a good choice, providing all the same flexible configuration options that are already in place in CT Publisher.

4. R3: The tenant data needs to be easy to detach from the demo system into elsewhere, whether it be in house servers, cloud, or the tenant's own on-premises

servers. Based on this, single-tenant databases are easy to move from ATR Soft's demo environment into another environment.

5. The scalability is not a big concern for the demo, as the amount of users in the demo environment at a single time is most likely low.

6. R4: Performance is isolated in a full tenant system, allowing customers full control of resources given to them. On a multitenant system performance can be affected from another tenants use, but can be taken into consideration when allocating resources.

Balancing between maintainability and security was not easy. With these considerations in mind, it was obvious we wanted to implement a multi-tenant application. The lower maintenance costs and the ease of configurability were the main factors that drove forward this decision.

The single-tenant database solution was decided for the data storage. This allows for customers to have full control of their data albeit being a maintenance hindrance.

A multi-tenant application with single-tenant databases approach is chosen for CT Publisher demo. Implementing the multi-tenant application involves performing many of the necessary steps to implement a multi-tenant database as well. This means that a future implementation of multi-tenant databases is possible with relatively minimal effort. Such an implementation was deemed to require more comprehensive security measures implemented to be considered than possible in the scope and time frame of this thesis work. As where the product currently stands, single-tenant databases satisfy the security needs of the customers and do not hinder the maintainability of the service significantly at this scale.

5.2 Implementation

As we decided to use a model with multi-tenant application and single-tenant databases, we can utilize the already existing test- and demo-databases to test our multi-tenant application implementation. The implementation is explained on a rather general level here, as the CT Publisher is a protected product that belongs to ATR Soft.

The work flow from research to implementation was not straightforward in this project. The implementation of the new tenancy model required some preparation. The technologies used were not known to the author beforehand, so I had to first get familiar with the programming languages, frameworks and programs used. I first got familiar with the current single-tenant model and analyzed and tested how the website, web API and database instances were running on test environment on a virtual machine. Being familiar with the current solution then allowed me to start implementing the new multi-tenant model smoothly. There was still some amount of back and forth switching between implementing the new solution and researching more as problems arose in the code and when my understanding further grew on the project. This was important to verify the implementation was satisfying all the requirements accordingly.

As the current solution directly has a single website and API services pointing to a single database, it only needs to know that single specific connection to that tenant's database. The connection is served as a connection string object that points to a specific database. The usage of multiple connection strings needs to be implemented first to allow for the website and API instances to communicate with more than one database. The data flow plan seen in Figure 5.3 shows how the connection string is fetched from Catalog, the creation of which is explained next. Website and API only need to know the subdomain they have been accessed on to be able to acquire the correct connection string.

To manage all the connection strings for the tenant databases, the Catalog database discussed in Section 2.3 needs to be created. Microsoft SLQ Server Management Studio (SSMS) was used to manage our databases. It allowed us to view and analyze our test

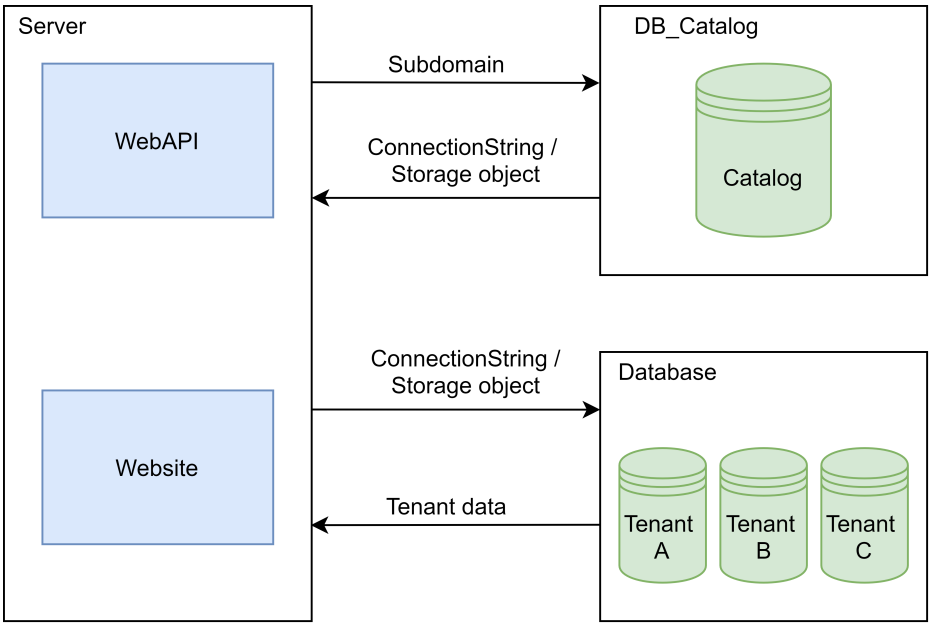

Figure 5.3: CT Publisher data flow chart.

databases and the Catalog-database. The Catalog was created through code-first approach to creating databases. We use Entity Framework Object-relational mapping (ORM) to create models using classes. This allows us to work with the database using .NET objects, intancing these defined classes through a derived database context. Data on these models is read and saved through the context "DbContext" in the Catalog. All the databases in the project also derive from the same database context to connect to databases.

These connections are created through using connection strings. Only the connection string to the Catalog is known, where all the other connection strings are stored. In the single-tenant solution only the customer database connection string was known, as the applications were instanced per customer. The connection string was stored in the configuration files. The catalog connection string is stored in the same way.

With sufficient rights, we can now request a connection string from the Catalog to connect to a customer database. In the multi-tenant application system, we now need to know which tenant is trying to connect to which database. We defined a system to fetch the corresponding customer from URL of the HTTP request itself.

All customers using the product use a unique subdomain for their websites. The customers get to define that subdomain to use their company or brand name. For example, "subdomain1" would access their CT Publisher website through the URL address `subdomain1.ctpublisher.com`.

Using .NET, we utilize a provided `HttpContext`-class through which we can access the URL of the request. From there, we identify the subdomain name and use it to fetch the corresponding connection string from the Catalog. The connection string is used to create a connection to the corresponding database only if the tenantID identifier on the connection string matches with the subdomain on the database request. The requested resources are then fetched from the correct database and finally the tenant data shown to the end user. This satisfies R2: tenant security when authorization and authentication is made for every request to that database to ensure the data is only given to tenants of that database.

The feature of fetching the subdomains from the URL ties into R3: customer integration. We automated the creation of a demo site through using a new subdomain. We added the option to whitelist new subdomains on the fly. After this, trying to access the new website starts the automated process to create the website, web API and database. Naturally, the new connection string to the new database is also added to the existing catalog. The whitelisting of subdomains is done to prevent anyone from trying to create endless new sites and trying to overload the system. This also satisfies R4: performance, as such attacks are prevented to affect any existing tenants' performance.

With the multi-tenant application setup and running, we need to lean on access control methods to further satisfy R2: tenant security. Token-based authentication and the use of authorization server like outlined on OAuth2.0 specification discussed in Section 3.5.1 were implemented to add an additional level of security. The access tokens granted per tenant allow for role-based access control. These roles are assigned to tenants users based on the principle of least privilege, allowing them access to only the resources they need.

5.3 Local setup

Testing the project locally can be done using IIS Express. However, there are a few things that need to be setup for the website and web API to work.

On IIS Administrative tools, we need to generate and install a *wildcard certificate*. This wildcard certificate allows for multiple tenants accessing the same instances of the website and web API. The certificate usually needs to be binded to specific websites. With a wildcard certificate, we can bind it to a website and all of its subdomains by using an asterisk "*" appending our website URL as follows: `"*.ctpublisher.com"`.

The IIS Express has its configuration file applicationhost.config located at

```
<project directory>\.vs\config\
```

The config file allows for adding a wildcard site binding. It can be done manually using a text editor or programmatically:

```
c:\Program Files\IIS Express\appcmd.exe" set site
   /site.name:CTPublisherWebsite
   /+bindings.[protocol='http',bindingInformation='*:<port
   number>:localhost
```

In this example, the binding is set to an address `*:80:localhost`. As we do not want to allow every subdomain binding possible, we will only add bindings when needed using a format `*:<portNumber>:<tenantName>:localhost` so an example `bindingInformation` would look like this: `*:80:tenant:localhost`.

When using Windows servers on our local machine, we can use the netsh-commands provided in HTTP Server API to manage your HTTP-settings. By using the following command line command,

```
netsh http add urlacl url=http://url:port/user=Everyone
```

we can reserve specific URLs on certain ports to test the multitenant websites on. The sites need to be reserved first with the command for the sites to work locally. We created a simple Windows batch file (.bat) to be ran easily by anyone wishing to test their services locally before launching them fully. This file is ran through the command line as:

```
setup.bat project url port protocol
```

If some of the parameters were omitted, default values were added. The program also adds the appropriate bindingInformation to the IIS Express applicationhost.config file. After running this file, everything is setup on our machine to simply run the website and web API as we wish.

Documentation was created in depth to aid in installing the development environment for the multi-tenant application on a clean virtual machine and IIS.

6 Validation

In this chapter, we define methods to validate and analyze the change and results in our new implementation. We go over both qualitative and quantitative methods to validate our work. The implementation is reviewed both by the author and the team, who give thoughts and experience of the advantages over the previous implementation. We then lean on literature to review the effectiveness of our new implementation. We then go over tests to assess the maintainability and security qualitatively.

6.1 Personal and team review

We managed to identify multiple problems with the existing solution, and designed goals and requirements to fix those problems. We structured a new implementation to comply with the requirements. A new tenancy model was implemented successfully and many of the product management processes streamlined to be simpler and faster. Maintaining the product became evidently easier in terms of user experiences, as we implemented systems to allow for better maintainability. The product owner was satisfied with the streamlined processes introduced and testified immediate benefits in maintainability in the new implementation. The product owner also reported all the set requirements were satisfied properly. Other team members were also positive about the effects of the research and implementation finished through the scope of this thesis. Based on these personal reviews on user experience, we can evaluate the requirements R1–R4 are satisfied on a practical level.

6.2 Theory-based review

The implemented work in the scope of this thesis can be assessed against the presented research literature on the field. The server-side implementation of the website and web APIs allow for them to be instanced only once, but used by multiple customers. This allows for multi-tenancy on the application and increases the performance when implemented correctly [3], [5], [7], [8], [10], [11], [13], [16], [19], [31]. The databases are implemented as single-tenant, which proves to be more secure as the data isolation is better for security than in a multi-tenant database [20], [22], [24], [27], [31], [53]–[55]. For access control security, we used established protocols, frameworks and industry best practices [21]–[27], [29], [31]–[33], [36], [39], [53], [55]. These include following OAuth2.0 protocol on authorization, using Entity Framework Identity to further strengthen access control, conforming to GDPR on data storage and using encrypting data best practices.

Based on the literature these sources provide, we can evaluate that all the requirements R1–R4 are satisfied on theoretical level.

6.3 Tests to measure maintainability and security

This section discusses tests designed to measure the maintainability and security of the product to validate the new implementation.

6.3.1 MSTest

MSTest is a library available to test .NET Applications on multiple platforms, imported using `Microsoft.VisualStudio.TestTools`. MSTest allows for designing unit tests for our application. We use MSTest to design tests per function in our project. When implemented, it allows for automated tests during development. Implementing proper unit testing was important for the change in tenancy model, as it allowed us to retest the changed code whenever we made changes. Unit testing has become an accepted practice

in the field [56]. When implementing the new tenancy model, we made sure to write unit tests for functions as we wrote them. We confirmed that the new implementation passed all the existing and new unit tests designed.

6.3.2 Mocking user data

Testing the new implementation using mocked user data is very important on further affirming R2: tenant security and solving data security discussed in Section 3.1. We need concrete evidence of only the correct data shown to the correct tenants. These tests were designed to test the authentication and authorization mechanisms in place. Performing these tests is discussed in the next Section 6.3.3.

6.3.3 Windows Communication Foundation Service Host and Test Client

Windows Communication Foundation (WCF) Service Host allows us to host and test services with ease on our Visual Studio development environment. We can then use the Test Client, which is a GUI tool that enables users to input test parameters, submit that input to the service and view the response that the service sends back [57]. This tool was used to design and use simple tests on our IIS hosted test environment with mocked data. Any problems occurring during these tests were addressed and fixed.

These tests confirmed data security, as the mocked tenants were shown only their data. As wrong or another tenants' data was not shown, this is further proof of satisfying R2: tenant security.

6.4 Future outlook on tests

The design of further tests for attacks covered in Chapter 3, like SQL injection and XSS, was started. However, due to time constraints on the employment of the author and the

organization, these designs were ultimately left unfinished and naturally not yet implemented. These tests will further show that the product is compliant with current security measures and standards, and as such, should be implemented in full in the future. On the scope of this thesis, these designs were rudimentary but not comprehensive.

Further monitoring on the new hybrid tenancy model implementation needs to be done to further assess the scalability and costs to allow for profitability. Like we discussed in Section 4.4, performance monitoring could reveal some tenants using higher proportion of the resources than others, allowing for just pricing models on customers of varying sizes in terms of performance.

7 Conclusions

In this Chapter, we provide a summary of the study work and results said work provides. Then we answer the research questions set in Chapter 1. We also discuss the limitations of the study and consider future prospects for the case study product. We also review the value proposition of this thesis for the case study product and similar products in SaaS.

7.1 Summary of the study work

The goal of this thesis was to analyze a case study SaaS product against research literature on maintainability and security. SaaS tenancy models and security practices on authentication and authorization were reviewed. Analyzing the options on tenancy models, whether single-tenant, multi-tenant or hybrid tenancy, we detected that the case tenancy model of full single-tenant was not a tenancy model widely used in similar products. We identified problems of administrative maintainability, security and its measuring as well as scalability and cost. We discussed goals designed to improve upon these issues. We set four requirements based on achieving these goals. To satisfy these goals, we determined that a new tenancy model should be implemented along with features to improve the maintainability.

The full single-tenant tenancy model the product originally had was changed to a hybrid tenancy model of multi-tenant application and single-tenant databases. This was done using the Entity Framework, adding a Catalog-type database to the system for discerning tenants and allowing for multi-tenant instances for the website and web API. Validation

methods for measuring and confirming the maintainability and security of the new tenancy model were discussed and partly performed. Further measuring and testing was left outside of the scope of this thesis due to time constraints.

The new tenancy model was implemented successfully and validated accordingly. Some new features were added to further aid with the maintainability for the product owner in terms of hosting, updating and developing the product. The research questions were answered based on literature and the results of the implementation. We identified security, maintainability and performance to seem to be the most prevalent concerns when choosing a tenancy model for a SaaS product. As a product enters different phases in its life cycle, it might be beneficial to change to a different tenancy model. We provided many concerns to take into account on top of the aforementioned three most prevalent aspects, which might push to a decision on restructuring the tenancy model in use. We demonstrated that in our case study product, it was beneficial to change from a single-tenant application and database to a hybrid multi-tenant application and single-tenant database. A full multi-tenant application and database solution could further reduce costs on hosting the product, but will require further efforts to satisfy the security requirements on the product. Until then, the hybrid model is considered the best option on the case study product.

7.2 Answering research questions

In this Section, we will provide answers to the research questions set in Chapter 1 based on the research and results on this thesis.

7.2.1 Answering research question 1

What aspects of developing and providing a SaaS product should be considered when deciding on a tenancy model? We identified multiple important aspects to be

considered that are beneficial to maintainability and security on SaaS products. These include security and data isolation, efficiency and performance, administrative manageability, scalability, profitability, explained in more depth below.

Data isolation is crucial for security. Physical isolation proves the best security, but in a multi-tenant environment that might not be possible. Data isolation must be implemented in a logical form conrofming to row level security in these cases. *Efficiency and performance* aspects ensure customers receive the service with minimal downtime and in the full capacity promised. Service providers can monitor performance per customer and scale resources and billing accordingly. *Administrative manageability* is important to support the *scalability* of SaaS products. When designing the service to be scalable to allow for ever increasing amount of customers, processes to maintain the product need to be designed in a support-friendly manner. Maintaining, updating and providing support for the product shouldn't be heavily affected by an increasing amount of customers. Obtaining said customers should be made cost-efficient and adding said customers to the system should take minimal effort and cost to maximize *profitability*.

7.2.2 Answering research question 2

Is a change from a single-tenant implementation to a multi-tenant implementation beneficial in terms of maintainability and security? Yes. We approached the question in terms of making the change in our case study product. The implementation was beneficial for maintainability and security and the supplementary aspects discussed in answering research question 1. As such, we recognize the results of the case study can be generalized to hold true in any similar SaaS product with the same tenancy model or facing the same issues. In a SaaS environment any form of multi-tenancy over single-tenancy will have sizable and measurable gains to maintainability. Appropriate security measures can be taken to improve the security of any tenancy model.

7.3 Study limitations

As the validation done was focused more on qualitative analysis, more quantitative analysis would be effective to further justify the work on the case study. For a more comprehensive analysis of the results of changing the case study tenancy model, the old model would have needed to be monitored and measured with quantitative methods. These same methods could have then been used on the new implementation, allowing us to compare the quantitative results. This would have allowed us to have a more defined and objective quantitative analysis on the thesis work done on the case study.

The case study project took longer to complete than anticipated. This is a common recurring trope in the computer science industry, where projects more often than not take more time to complete than planned. Unfortunately, time limitations posed by the employment relationship with the author and the case study organization, the scope of the thesis was ultimately reduced from the original plans. While maintainability and security were qualitatively and quantitatively improved, further work on analysing the results would have further increased the validity and reliability of this study. Despite the limited validation completed on the scope of this thesis, we laid out the ground work for such tests to be completed in the future based on our security issues analysis.

7.4 Value proposition of the study

We believe the study added significant value to the case study product CT Publisher. We provided timely upgrades to the maintainability and security of the product, and also made the development environment more efficient for further development and testing. Further tests could be designed on the basis of the research in this study. Researching and exploring the tenancy models and security issues prepared the case study product to be analyzed in a critical manner, and allowed us to identify problems in the current implementation efficiently together with the case study organization. This research also

prepared the organization to better plan for the future vision of the product with more knowledge of the possible tenancy models, maintainability features and security issues.

The research in this thesis presents valuable insights on what aspects to consider when building any SaaS product. The case study gives insight to potential problems one might face in SaaS development, and the information presented here can be helpful and applied at any point of a SaaS product development or life cycle. We provide tools and examples on how to evaluate such a software project. We show with the case study how to define goals and requirements for a scalable, maintainable and secure SaaS product. We follow through an approachable implementation on the case study presenting how implementing a new tenancy model might look in a SaaS product in need of a tenancy model change. This information may be used to aid in developing and improving upon a similar product in the SaaS environment.

References

[1] P. M. Mell and T. Grance, "The nist definition of cloud computing", National Institute of Standards and Technology, 2011. DOI: 10.6028/NIST.SP.800-145. [Online]. Available: https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-145.pdf.

[2] S. E. Kafhali, I. E. Mir, K. Salah, and M. Hanini, "Dynamic scalability model for containerized cloud services", *Arabian Journal for Science and Engineering*, vol. 45, pp. 10 693–10 708, 12 Dec. 2020, ISSN: 21914281. DOI: 10.1007/s13369-020-04847-2.

[3] C.-P. Bezemer and A. 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)*, ser. IWPSE-EVOL '10, Antwerp, Belgium: Association for Computing Machinery, 2010, pp. 88–92, ISBN: 9781450301282. DOI: 10.1145/1862372.1862393. [Online]. Available: https://doi.org/10.1145/1862372.1862393.

[4] R. Mietzner, A. Metzger, F. Leymann, and K. Pohl, "Variability modeling to support customization and deployment of multi-tenant-aware software as a service applications", in *2009 ICSE Workshop on Principles of Engineering Service Oriented Systems*, May 2009, pp. 18–25. DOI: 10.1109/PESOS.2009.5068815.

[5] N. H. Bien and T. D. Thu, "Hierarchical multi-tenant pattern", in *2014 International Conference on Computing, Management and Telecommunications (ComManTel)*, Apr. 2014, pp. 157–164. DOI: 10.1109/ComManTel.2014.6825597.

[6] R. Mietzner and F. Leymann, "Generation of bpel customization processes for saas applications from variability descriptors", in *Proceedings of the 2008 IEEE International Conference on Services Computing - Volume 2*, ser. SCC '08, USA: IEEE Computer Society, Jul. 2008, pp. 359–366, ISBN: 9780769532837. DOI: 10.1109/SCC.2008.85. [Online]. Available: https://doi.org/10.1109/SCC.2008.85.

[7] C. D. Weissman and S. Bobrowski, "The design of the force.com multitenant internet application development platform", in *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '09, Providence, Rhode Island, USA: Association for Computing Machinery, 2009, pp. 889–896, ISBN: 9781605585512. DOI: 10.1145/1559845.1559942. [Online]. Available: https://doi.org/10.1145/1559845.1559942.

[8] W.-T. Tsai, Q. Shao, Y. Huang, and X. Bai, "Towards a scalable and robust multi-tenancy saas", in *Proceedings of the Second Asia-Pacific Symposium on Internetware*, ser. Internetware '10, Suzhou, China: Association for Computing Machinery, 2010, ISBN: 9781450306942. DOI: 10.1145/2020723.2020731. [Online]. Available: https://doi.org/10.1145/2020723.2020731.

[9] A. R. Hevner, S. T. March, J. Park, and S. Ram, "Design science in information systems research", *MIS Quarterly*, vol. 28, no. 1, pp. 75–105, 2004, ISSN: 02767783. [Online]. Available: http://www.jstor.org/stable/25148625 (visited on 05/09/2023).

[10] V. contributors, *Multi-tenant saas database tenancy patterns*, Mar. 2023. [Online]. Available: https://docs.microsoft.com/en-us/azure/sql-database/saas-tenancy-app-design-patterns#d-multi-tenant-app-with-database-per-tenant (visited on 06/02/2023).

[11] S. Kalra and T. V. Prabhakar, "Towards dynamic tenant management for microservice based multi-tenant saas applications", in *Proceedings of the 11th Innovations in Software Engineering Conference*, ser. ISEC '18, Hyderabad, India: Association for Computing Machinery, 2018, ISBN: 9781450363983. DOI: 10.1145/3172871.3172882. [Online]. Available: https://doi.org/10.1145/3172871.3172882.

[12] J. Mace, P. Bodík, R. Fonseca, and M. Musuvathi, "Retro: Targeted resource management in multi-tenant distributed systems", in *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, Oakland, CA: USENIX Association, May 2015, pp. 589–603, ISBN: 9781931971218. [Online]. Available: https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/mace.

[13] K. Aytaç and Ö. Korçak, "Multi-tenant management in secured iot based solutions", in *2022 32nd Conference of Open Innovations Association (FRUCT)*, Nov. 2022, pp. 56–64, ISBN: 9789526924489. DOI: 10.23919/FRUCT56874.2022.9953817.

[14] B. Tang, Q. Li, and R. Sandhu, "A multi-tenant rbac model for collaborative cloud services", in *2013 Eleventh Annual Conference on Privacy, Security and Trust*, Jul. 2013, pp. 229–238. DOI: 10.1109/PST.2013.6596058.

[15] W. Sun, X. Zhang, C. J. Guo, P. Sun, and H. Su, "Software as a service: Configuration and customization perspectives", in *2008 IEEE Congress on Services*

Part II (services-2 2008), Sep. 2008, pp. 18–25. DOI: 10.1109/SERVICES-2.2008.29.

[16] W.-T. Tsai and X. Sun, "Saas multi-tenant application customization", in *2013 IEEE Seventh International Symposium on Service-Oriented System Engineering*, Nov. 2013, pp. 1–12. DOI: 10.1109/SOSE.2013.44.

[17] H. Song, F. Chauvel, and A. Solberg, "Deep customization of multi-tenant saas using intrusive microservices", in *Proceedings of the 40th International Conference on Software Engineering: New Ideas and Emerging Results*, ser. ICSE-NIER '18, Gothenburg, Sweden: Association for Computing Machinery, May 2018, pp. 97–100, ISBN: 9781450356626. DOI: 10.1145/3183399.3183407. [Online]. Available: https://doi.org/10.1145/3183399.3183407.

[18] M. Tuomola, "Käyttäjäkeskeinen käyttöliittymäsuunnittelu web-sovellukseen: Case atr soft", Finnish, M.S. thesis, University of Turku, Dec. 2020. [Online]. Available: https://urn.fi/URN:NBN:fi-fe202102053817.

[19] T. C. Sandanayake and P. G. C. Jayangani, "Current trends in software as a service (saas)", in *International Journal for Innovation Education and Research*, vol. 6, Feb. 2018, pp. 221–234. [Online]. Available: https://doi.org/10.31686/ijier.vol6.iss2.969.

[20] P. K. Chouhan, F. Yao, S. Yerima, and S. Sezer, "Software as a service: Analyzing security issues", in *International Conference on Big Data and Analytics for Business (BDAB 2014)*, New Delhi, India, Dec. 2014. [Online]. Available: https://doi.org/10.48550/arXiv.1505.01711.

[21] H. AL-Amro and E. El-Qawasmeh, "Discovering security vulnerabilities and leaks in asp.net websites", in *Proceedings Title: 2012 International Conference on Cyber Security, Cyber Warfare and Digital Forensic (CyberSec)*, Jun. 2012, pp. 329–333. DOI: 10.1109/CyberSec.2012.6246175.

[22] P. K. Tiwari and S. Joshi, "A review of data security and privacy issues over saas", in *2014 IEEE International Conference on Computational Intelligence and Computing Research*, Dec. 2014, pp. 1–6. DOI: 10.1109/ICCIC.2014.7238432.

[23] J. A. Díaz-Rojas, J. O. Ocharán-Hernández, J. C. Pérez-Arriaga, and X. Limón, "Web api security vulnerabilities and mitigation mechanisms: A systematic mapping study", in *2021 9th International Conference in Software Engineering Research and Innovation (CONISOFT)*, Oct. 2021, pp. 207–218. DOI: 10.1109/CONISOFT52520.2021.00036.

[24] M. Shakir, M. Hammood, and A. K. Muttar, "Literature review of security issues in saas for public cloud computing: A meta-analysis", *International Journal of Engineering & Technology*, vol. 7, p. 1161, 3 Jun. 2018. DOI: 10.14419/ijet.v7i3.13075.

[25] M. Pearce, S. Zeadally, and R. Hunt, "Virtualization: Issues, security threats, and solutions", 2, vol. 45, New York, NY, USA: Association for Computing Machinery, Mar. 2013. DOI: 10.1145/2431211.2431216. [Online]. Available: https://doi.org/10.1145/2431211.2431216.

[26] H. Wu, Y. Ding, C. Winer, and L. Yao, "Network security for virtual machine in cloud computing", in *5th International Conference on Computer Sciences and Convergence Information Technology*, Nov. 2010, pp. 18–21. DOI: 10.1109/ICCIT.2010.5711022.

[27] T. Sharma, T. Wang, C. Di Giulio, and M. Bashir, "Towards inclusive privacy protections in the cloud", in *Applied Cryptography and Network Security Workshops: ACNS 2020 Satellite Workshops, AIBlock, AIHWS, AIoTS, Cloud S&P, SCI, SecMT, and SiMLA*, ser. Lecture Notes in Computer Science, Rome, Italy: Springer-Verlag, 2020, pp. 337–359, ISBN: 978-3-030-61637-3. DOI: 10.1007/978-3-030-

61638-0_19. [Online]. Available: https://doi.org/10.1007/978-3-030-61638-0_19.

[28] V. Goyal, O. Pandey, A. Sahai, and B. Waters, "Attribute-based encryption for fine-grained access control of encrypted data", in *Proceedings of the 13th ACM Conference on Computer and Communications Security*, ser. CCS '06, Alexandria, Virginia, USA: Association for Computing Machinery, 2006, pp. 89–98, ISBN: 1595935185. DOI: 10.1145/1180405.1180418. [Online]. Available: https://doi-org.ezproxy.utu.fi/10.1145/1180405.1180418.

[29] S. N. Kazmi, "Access control process for a saas provider", University of Turku, Jun. 2019. [Online]. Available: https://urn.fi/URN:NBN:fi-fe2019092329507.

[30] S. Han, G. Skinner, V. Potdar, and E. Chang, "A framework of authentication and authorization for e-health services", in *Proceedings of the 3rd ACM Workshop on Secure Web Services*, ser. SWS '06, Alexandria, Virginia, USA: Association for Computing Machinery, 2006, pp. 105–106, ISBN: 1595935460. DOI: 10.1145/1180367.1180387. [Online]. Available: https://doi-org.ezproxy.utu.fi/10.1145/1180367.1180387.

[31] N. Solanki, W. Zhu, I.-L. Yen, F. Bastani, and E. Rezvani, "Multi-tenant access and information flow control for saas", in *2016 IEEE International Conference on Web Services (ICWS)*, Jun. 2016, pp. 99–106. DOI: 10.1109/ICWS.2016.21.

[32] D. Ferraiolo and D. R. Kuhn, "Role-based access controls", vol. abs/0903.2171, Mar. 2009. [Online]. Available: https://doi.org/10.48550/arXiv.0903.2171.

[33] E. J. Coyne, T. R. Weil, and R. Kuhn, "Role engineering: Methods and standards", in *IT Professional*, vol. 13, Nov. 2011, pp. 54–57. DOI: 10.1109/MITP.2011.105.

[34] R. Anderson and V. contributors. ”Prevent cross-site scripting (xss) in asp.net core”. (Mar. 2023), [Online]. Available: https://learn.microsoft.com/en-us/aspnet/core/security/cross-site-scripting (visited on 05/26/2023).

[35] V. contributors. ”Dotnet security cheat sheet”. (2021), [Online]. Available: https://cheatsheetseries.owasp.org/cheatsheets/DotNet_Security_Cheat_Sheet.html (visited on 05/26/2023).

[36] D. Hardt, *The OAuth 2.0 Authorization Framework*, RFC 6749, Oct. 2012. DOI: 10.17487/RFC6749. [Online]. Available: https://www.rfc-editor.org/info/rfc6749.

[37] T. Wisniewski, G. Whitehead, and e. a. Hinton Hp, ”Saml v2.0 errata 05”, May 2012. [Online]. Available: http://docs.oasis-open.org/security/saml/v2.0/errata05/os/saml-v2.0-errata05-os.html.

[38] *Openid connect core 1.0 incorporating errata set 1*. [Online]. Available: https://openid.net/specs/openid-connect-core-1_0.html (visited on 06/02/2023).

[39] B. D. Göçer and Ş. Bahtiyar, ”An authorization framework with oauth for fintech servers”, in *2019 4th International Conference on Computer Science and Engineering (UBMK)*, Sep. 2019, pp. 536–541. DOI: 10.1109/UBMK.2019.8907182. [Online]. Available: https://doi.org/10.1109/UBMK.2019.8907182.

[40] G. J. Simmons, ”Symmetric and asymmetric encryption”, *ACM Comput. Surv.*, vol. 11, no. 4, pp. 305–330, Dec. 1979, ISSN: 0360-0300. DOI: 10.1145/356789.356793. [Online]. Available: https://doi.org/10.1145/356789.356793.

[41] Q. Zhang, "An overview and analysis of hybrid encryption: The combination of symmetric encryption and asymmetric encryption", in *2021 2nd International Conference on Computing and Data Science (CDS)*, Jan. 2021, pp. 616–622. DOI: 10.1109/CDS52072.2021.00111.

[42] F. Moyón, P. Almeida, D. Riofrío, D. Mendez, and M. Kalinowski, "Security compliance in agile software development: A systematic mapping study", in *2020 46th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, Aug. 2020, pp. 413–420. DOI: 10.1109/SEAA51224.2020.00073.

[43] *The national institute for standards in technology*. [Online]. Available: https://www.nist.gov/ (visited on 06/02/2023).

[44] *The international organization for standardization*. [Online]. Available: https://www.iso.org/ (visited on 06/02/2023).

[45] *Regulation (eu) 2016/679 of the european parliament and of the council of 27 april 2016 on the protection of natural persons with regard to the processing of personal data and on the free movement of such data, and repealing directive 9546ec (general data protection regulation)*. [Online]. Available: https://eur-lex.europa.eu/eli/reg/2016/679/oj (visited on 06/02/2023).

[46] "Atr soft oy". (2023), [Online]. Available: https://www.atrsoft.com/ (visited on 05/18/2023).

[47] "Ct publisher". (2023), [Online]. Available: https://www.ctpublisher.com/ (visited on 03/16/2023).

[48] T. Reenskaug, "The model-view-controller (mvc) its past and present", Aug. 2003. [Online]. Available: http://home.ifi.uio.no/trygver/2003/javazone-jaoo/MVC_pattern.pdf.

[49] "Entity framework 6". (2020), [Online]. Available: https://learn.microsoft.com/en-us/ef/ef6/ (visited on 03/12/2023).

[50] "Microsoft internet information services (iis)". (2023), [Online]. Available: https://www.iis.net/ (visited on 03/12/2023).

[51] G. Kappes, A. Hatzieleftheriou, and S. V. Anastasiadis, "Multitenant access control for cloud-aware distributed filesystems", English, *IEEE Transactions on Dependable and Secure Computing*, vol. 16, no. 6, pp. 1070–1085, Nov. 2019. DOI: 10.1109/TDSC.2017.2715839. [Online]. Available: https://ieeexplore.ieee.org/document/7949056.

[52] A. Omezzine, N. B. B. Saoud, S. Tazi, and G. Cooperman, "Sla and profit-aware saas provisioning through proactive renegotiation", English, IEEE, Oct. 2016, pp. 351–358. DOI: 10.1109/NCA.2016.7778640. [Online]. Available: https://ieeexplore.ieee.org/document/7778640.

[53] K. Gupta, S. Kumar, and O. Agnihotri, "Data isolation in multi-tenant saas environment", in *2016 International Conference on Computing, Communication and Automation (ICCCA)*, Apr. 2016, pp. 1290–1292. DOI: 10.1109/CCAA.2016.7813917.

[54] N. Pustchi and R. Sandhu, "Mt-abac: A multi-tenant attribute-based access control model with tenant trust", in *Network and System Security*, M. Qiu, S. Xu, M. Yung, and H. Zhang, Eds., ser. Lecture Notes in Computer Science, vol. 9408, Cham: Springer International Publishing, Nov. 2015, pp. 206–220, ISBN: 978-3-319-25644-3. DOI: 10.1007/978-3-319-25645-0_14. [Online]. Available: https://doi.org/10.1007/978-3-319-25645-0_14.

[55] R. Maheshwari, A. Toshniwal, and A. Dubey, "Software as a service architecture and its security issues: A review", in *2020 Fourth International Conference on Inventive Systems and Control (ICISC)*, Jan. 2020, pp. 766–770. DOI: 10.1109/ICISC47916.2020.9171145. [Online]. Available: https://doi.org/10.1109/ICISC47916.2020.9171145.

[56] E. Daka and G. Fraser, "A survey on unit testing practices and problems", in *2014 IEEE 25th International Symposium on Software Reliability Engineering*, 2014, pp. 201–211. DOI: 10.1109/ISSRE.2014.11. [Online]. Available: https://doi.org/10.1109/SEAA51224.2020.00073.

[57] V. contributors, *Wcf test client (wcftestclient.exe)*, 2021. [Online]. Available: https://learn.microsoft.com/en-us/dotnet/framework/wcf/wcf-test-client-wcftestclient-exe.