INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO
MESTRADO EM ENGENHARIA INFORMÁTICA




Integrating a Software Asset Management Solution in a Software Development Company

JOÃO MIGUEL ALMEIDA FERREIRA DA SILVA
Setembro de 2024

POLITÉCNICO
DO PORTO

ISEP INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO

Integrating a Software Asset Management Solution in a Software Development Company

João Silva

A dissertation submitted in partial fulfillment of the requirements for the degree of Master of Science, Specialisation Area of Software Engineering

Supervisor: Nuno Silva, Professor, DEI/ISEP

Evaluation Committee:
President:
Luís Lino Ferreira, Professor, DEI/ISEP

Members:
António Rocha, Professor, DEI/ISEP

Porto, September 15, 2024

Statement of Integrity

I hereby declare having conducted this academic work with integrity.

I have not plagiarised or applied any form of undue use of information or falsification of results along the process leading to its elaboration.

Therefore the work presented in this document is original and authored by me, having not previously been used for any other end.

I further declare that I have fully acknowledged the Code of Ethical Conduct of P.PORTO.

ISEP, Porto, September 15, 2024

Dedicatory

I want to thank Mindera for always allowing me to explore my interests and for providing this project idea for me. To ISEP, for having me and easily being the best decision I have made during my studies. To all my friends, without whom I wouldn't be able to achieve any of this. Thank you for always having my back and giving me so many memories that I'm fond of. Without all of you, nothing would be the same. To Tiago, my partner in crime during my studies at ISEP. My partner from day one, for every project and, now, for life. To my family, who always supported me and never stopped believing in my capabilities. And, lastly, to my beautiful girlfriend, Irene. Words cannot describe how happy I am every day that I am by your side. Every day I find out it truly is possible to love a person to its fullest capacity. I love you and always will. For now, and forever.

Abstract

In recent years, the complexity of software development, or even other types of work, has been increasing because of the need for various licensed products and services, with each new tool introducing distinct licensing requirements. As organizations increasingly rely on a broad array of software, managing these licenses has become a significant challenge. The details of tracking multiple software licenses, ensuring compliance, and avoiding legal or financial penalties can strain an organization's resources. This is particularly true for medium-sized companies like Mindera, which operates under a decentralized, self-organizing model. In such organizations, individual employees are often tasked with managing software licenses in addition to their core responsibilities, creating inefficiencies and potential risks.

In Mindera, over twenty different software licenses are provided to over a thousand employees, ranging from integrated development environments to design tools. These licenses are managed by different groups of people within the organization, and this decentralized system leads to a variety of challenges. It is difficult to ensure that licenses are revoked when employees leave, track the aggregate cost of licenses, or determine usage patterns for cost optimization. Furthermore, employees often experience delays when requesting new licenses, impacting their productivity and, in some cases, the overall performance of the organization.

To address these challenges, this project aims to develop a centralized, automated platform for managing software licenses within Mindera. The platform will streamline license tracking, simplify revocation, and automate communication with software providers, offering a comprehensive solution to the company's software asset management (SAM) needs. The system will enable dynamic policy definitions, allowing certain actions such as the automatic issuance or revocation of licenses based on predefined rules. Moreover, it will integrate with user support tools to make it easier for employees to request licenses, and maintain historical records to allow for better insights, cost control, and reporting.

The solution designed implements two services, *Core Service*, which will handle all of the information regarding the licenses, and the *Frontend Web*, which will be a user-facing service to allow the data to be properly managed.

Keywords: Software Asset Management, Mindera, Software Licenses, React, Bun

Resumo

Nos últimos anos, a complexidade do desenvolvimento de software, ou até mesmo de outros tipos de trabalho, tem vindo a aumentar devido à necessidade de vários produtos e serviços licenciados, sendo que cada nova ferramenta introduz requisitos de licenciamento distintos. À medida que as organizações passam a depender cada vez mais de uma ampla gama de softwares, a gestão destas licenças torna-se um desafio significativo. Os detalhes quanto ao rastreamento de múltiplas licenças de software, garantir conformidade e evitar penalidades legais ou financeiras podem sobrecarregar os recursos de uma organização. Isto acontece, particularmente, em empresas de tamanho médio como a Mindera, que opera sob um modelo descentralizado e auto-organizado. Em organizações deste tipo, os colaboradores são frequentemente encarregados de gerir licenças de software para além das suas responsabilidades principais, criando ineficiências e potenciais riscos.

Na Mindera, são fornecidas mais de vinte licenças de software a mais de mil colaboradores, abrangendo desde ambientes de desenvolvimento integrados a ferramentas de design. Estas licenças são geridas por diferentes grupos de pessoas dentro da organização, e este sistema descentralizado gera uma variedade de desafios. É difícil garantir que as licenças são revogadas quando os colaboradores deixam a empresa, rastrear o custo agregado das licenças ou determinar padrões de utilização para otimizar custos. Além disso, os colaboradores frequentemente enfrentam dificuldades ao solicitar novas licenças, o que afeta a sua produtividade e, em alguns casos, o desempenho geral da organização.

Para enfrentar estes desafios, este projeto visa desenvolver uma plataforma centralizada e automatizada para a gestão de licenças de software na Mindera. A plataforma irá simplificar o rastreamento de licenças, facilitar a revogação e automatizar a comunicação com os fornecedores de software, oferecendo uma solução abrangente para as necessidades de Software Asset Management (SAM) da empresa. O sistema permitirá a definição de políticas dinâmicas, possibilitando ações como a emissão ou revogação automática de licenças com base em regras predefinidas. Além disso, integrará ferramentas de suporte ao utilizador para facilitar a solicitação de licenças pelos colaboradores e manterá registos históricos para permitir melhores insights, controlo de custos e geração de relatórios.

A solução desenhada implementa dois serviços: o *Core Service*, que irá lidar com todas as informações relacionadas às licenças, e o *Frontend Web*, que será um serviço disponibilizado ao utilizador, permitindo que os dados sejam devidamente geridos.

Contents

List of Figures

List of Tables

List of Source Code

List of Acronyms

AD	Active Directory.
AI	Artificial Intelligence.
API	Application Programming Interface.
B2B	Business to Business.
CORS	Cross-Origin Resource Sharing.
CRUD	Create, Read, Update and Delete.
DSR	Design Science Research.
GCP	Google Cloud Platform.
GPT	Generative Pre-Trained Transformers.
HMR	Hot Module Replacement.
HTTP	Hypertext Transfer Protocol.
IDE	Integrated Development Environments.
ISV	Independent Software Vendor.
IT	Information Technology.
JSON	JavaScript Object Notation.
LDAP	Lightweight Directory Access Protocol.
LLM	Large Language Model.
RE	Requirements Engineering.
REST	Representational State Transfer.
SaaS	Software as a Service.
SAM	Software Asset Manager.
SDK	Software Development Kit.
UI	User Interface.
URL	Uniform Resource Locator.

Chapter 1

Introduction

Over the years, software development has been riddled with multiple products and services that require licenses, with the number ever-increasing as new tools are created.

As the software landscape evolves, managing the growing amount of licenses for various products and services becomes increasingly complex. Organizations often face challenges in keeping track of licensing agreements, ensuring compliance with diverse terms and conditions, and avoiding legal or financial penalties. The proliferation of tools, each with its own licensing model, necessitates robust software asset management practices to maintain an accurate inventory and manage usage effectively. This complexity can strain resources and lead to inefficiencies if not carefully managed.

In a medium-sized company like Mindera, where self-organization is a key value, the responsibility for managing software licenses often falls on individual employees. This decentralized approach can lead to increased stress for these employees, as they try to manage their primary client-focused roles with the additional task of overseeing license management. The dual demands can affect negatively their productivity and efficiency on client projects, potentially affecting overall project timelines and quality. Balancing these responsibilities requires careful time management and prioritization, which can strain their ability to deliver high-quality work while maintaining compliance with the licensing agreements and trying to keep the license costs controlled. Implementing centralized or automated solutions for license management might alleviate some of this burden, allowing employees to focus more effectively on their core project responsibilities.

This, however, doesn't happen only in Mindera. Technological companies have grown at a steady rate and so has their spending. A report from Gartner on Software Asset Manager (SAM) shows that, in 2024, software development companies will have an estimated spending of $1 trillion. With such large sums of money being allocated between different areas, these companies have started looking for places where they can avoid overspending. One of the areas that end up being scrutinized the most is software licensing with the help of SAM tools [1]. The same report from Gartner shows that SAM, from 2021 to 2022, have grown about 17.8%, while having a current market share of $1.7 billion.

1.1 Problem

Mindera currently provides over twenty different licenses to its employees, ranging from Integrated Development Environments (IDE) to design tools licenses, which are used by over a thousand people every day. Along with this, every month some people join and leave the company, which forces each license manager to regularly update a spreadsheet with the

people who currently have licenses. Since this is a manual process, it is prone to human error, which sometimes happens when an employee leaves the company and the license manager doesn't terminate the ex-employee license key.

Since most licenses are managed by different groups of people, it is also extremely difficult to properly manage the aggregate costs of these licenses, as well as know which licenses are the most used and where it is possible to save more money, which ends up being a pain point in the financial department. This is mainly due to having multiple people handling the management of licenses, as some license managers may be busier than others, it may be the case that the finance team needs to wait for the license manager to finish their client's work to shift their attention to the licenses assigned to them, and, while doing so, it may happen that employees don't give their full attention to this work, which ends up hurting the company financially, even if that is not their intention. Overall, having multiple people in charge of managing licenses makes it increasingly difficult to optimize the processes applied in the company.

Despite one of the largest pain points being the financial aspect, it is also frustrating, as an employee, to be kept on hold for indefinite periods to have access to specific licenses. Since this is a manual process, as mentioned previously it may be the case that the license manager simply doesn't have the time to invest in providing licenses to newly interested employees. This, however, can even end up affecting other employees' performance, as they may need the requested license to improve their work. This can even be the case for newly integrated employees, as there isn't an automated pipeline to distribute them the licenses they require once they join the company.

Lastly, as an employee who wants to request a new license, it's somewhat difficult to know how to request a certain license. This happens mainly due to having a decentralized system, as you never know who you need to ask, if you're eligible to receive the license, or if there are any licenses available at that time.

1.2 Objectives

The goal of this project is to develop a comprehensive platform that eases the management of licenses issued to employees within Mindera. Doing this will enhance license management processes, improve administrative efficiency, and ensure compliance within the company by providing a complete and user-friendly software solution.

More specifically, this software solution aims to:

- **Streamline license tracking** - by enabling the registration and tracking of licenses assigned to employees, ensuring a centralized and organized system.
- **Simplify license revocation** - by providing a clear understanding of which licenses need to be revoked when an employee leaves the company or their current project, ensuring efficient management of license usage. There should be a set of rules and actions that can be applied to each software license.
- **Enable communication with license providers** - by establishing effective automated communication with license providers, through the provided Application Programming Interface (API) or alternative methods, to facilitate license issuance and revocation.
- **Define dynamic policies** - given a certain set of dynamically defined rules, perform actions that trigger certain workflows, which can be the automatic issuing of a license

to a certain group of people, the revocation of licenses from a calculated group of people, among other actions to be defined.

- **Integrate with user support tools** - for easy license requests, there should be a way for employees to request licenses through, for example, support channels in the company's communication platform.
- **Maintain historical records and reporting** - by storing a history of issued and revoked licenses, it should be possible to reference and generate detailed reports for better insights, analysis, and cost control.

1.3 Research questions

This research project aimed to answer a very objective question: what is needed to build a system to manage software licenses and what is the best way to integrate it within a software development company?

Despite how straightforward this question may look, it brings other topics that need to be investigated. These are topics related to the system itself, such as the objectives of a SAM and the processes that companies normally use to manage these licenses. Then, it is needed to investigate other products that are available in the market and gather information on the features they provide. Therefore, with this, some minor questions also need to be researched, which are the following:

- What is a SAM and what are its key features?
- What do companies look for when searching for a SAM?
- What are the most common features among SAM tools across the market?

1.4 Research method

Throughout the research process, the Design Science Research (DSR) method will be utilized. This method "seeks to enhance technology and science knowledge bases that solve problems and improve the environment in which they are instantiated" [2].

This process is composed of six different activities [2]:

1. "Problem identification and motivation" - As it is the first step of the process, it starts with the identification of the problem at hand and the solution to solve it.
2. "Define the objectives for a solution" - Based on the first activity, the objectives can be gathered and specified. According to Brocke, Hevner, and Maedche [2], they can be either "quantitative, e.g., terms in which a desirable solution would be better than current ones, or qualitative, e.g., a description of how a new artifact is expected to support solutions to problems".
3. "Design and development" - This includes the creation of artifacts that determine the functionality to be developed, its architecture and then developing the artifact itself.
4. "Demonstration" - In this activity, the artifacts are used to demonstrate how they solve one or more of the problems.

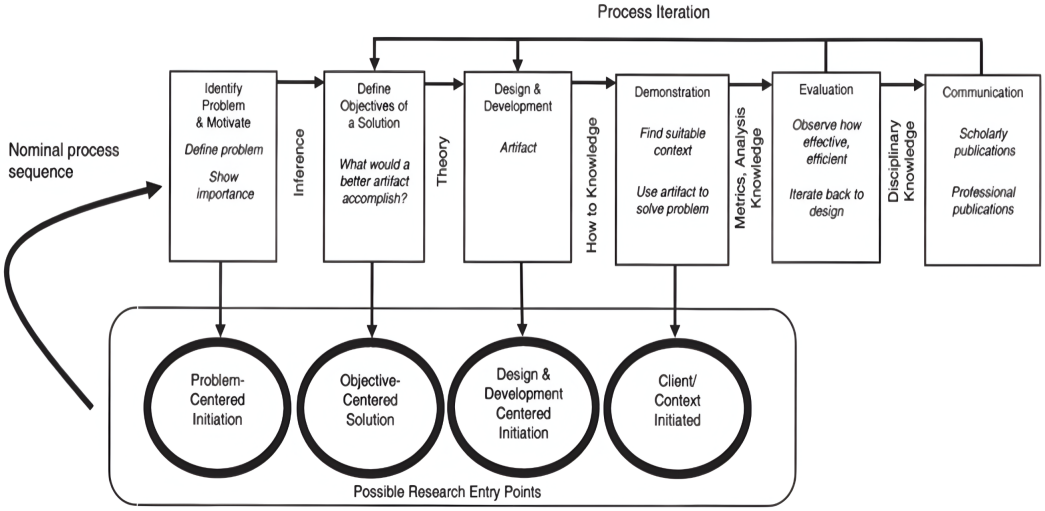

Figure 1.1: DSR Methodology Process Model Brocke, Hevner, and Maedche [2].

5. "Evaluation" - It "measures how well the artifact supports a solution to the problem", involving a comparison of the objectives to the results that were achieved. At the end of this activity, a decision is made on whether the involved researchers iterate to step 3, or if they continue to the following activity.
6. "Communication" - The stakeholders involved in the aspects of the problem are notified with every artifact created throughout the research process.

This research process, as mentioned in the *Evaluation* activity, is iterative and, therefore, it is constantly evolving. This can also be observed in Figure 1.1, shown in Brocke, Hevner, and Maedche [2]. This diagram shows us that the first four steps are sequential. These are followed by the last two steps, where it is possible to re-iterate the process, either after conducting the evaluation or after presenting the artifacts to the stakeholders, on the *Communication* activity. This re-iteration is then started on either activity two, or activity three, depending on the reasons that lead to this decision.

Since this process is constantly evolving, it is not possible to provide correct planning on the work that is going to be developed, as it is always subject to changes.

1.5 Document structure

After this introduction, the document will start with a study on the state of the art of current SAM tools (2), as well as a study on the architecture and technologies that will be needed to build a piece of software to manage software licensing.

This document will be composed of seven different chapters, each divided into sections and subsections, and have been chosen based on important steps that are essential to software engineering. These chapters are:

1. Introduction - The introduction chapter provides an overview of the project's problem, outlines the objectives, and establishes the research question of the thesis. Then, it explains the research method utilized, and, lastly, the document's structure.
2. State of the Art - This chapter aims to analyze available literature and available reports to answer the questions raised in 1.3. Then, a comparison between different similar products will be presented and assessed.
3. Analysis and Requirements - In this chapter, the research problem is thoroughly analyzed, and the functional and non-functional requirements of the proposed solution are clearly defined. It includes actors analysis, use cases, the business's domain model, and the glossary.
4. Design - The design chapter outlines the architecture and detailed design of the proposed solution. It includes diagrams, models, and descriptions of key components, explaining how they interact to meet the specified requirements. This chapter translates the theoretical analysis into a practical plan for implementation.
5. Implementation - This chapter details the process of developing the proposed solution, describing the tools, technologies, and frameworks used.
6. - Evaluation - TODO
7. - Conclusion - The final chapter outlines a possible plan for Mindera to adopt the system that was developed, as well as an evaluation of the achieved results, the challenges that were faced, potential improvements, and, lastly, the final reflections about the thesis.

Chapter 2

State of the art

This chapter aims to analyze what a software license is, as well as the standards for managing software assets. Lastly, a comparison between multiple software license management tools and services will be assessed. These have been carefully chosen from a vast array of products distributed throughout the internet to gather information on what the products on the market are currently offering.

2.1 Software Licenses

Software licenses are temporary or permanent grants of the rights to use a computer program. Multiple restraints can be applied to a software license, including geographical, temporal, or role-based restrictions [3]. Normally, software licenses inside companies have a defined life cycle, composed of 5 different steps [4]:

1. License request - an employee submits a request to have a license created and assigned to them. Then, the license manager verifies if there is a license currently available to assign to the employee. If there is a license, then the cycle skips to step 4.
2. Buying the license - the license manager buys the software license, either by directly contacting the license provider, by using the provider's website, or through any other means.
3. Registering the license - the license manager registers the software license that has just been bought in the company's records.
4. License assignment - the license manager assigns the software license requested by the employee to them.
5. Continuous validation checks - periodically verify if the employee is using the license they requested to make sure the company isn't losing money.

2.2 Software Asset Management Tools

According to the *ISO/IEC 19770-5:2015* norm, Software Asset Management is defined as the "control and protection of software and related assets within an organization, and control and protection of information about related assets which are needed in order to control and protect software assets" [5].

A SAM tool is defined as an automated solution designed to assist with tasks necessary for guaranteeing compliance with the licensing requirements of an Independent Software

Vendor (ISV). These tools also enhance an organization's capability to proactively identify and manage software-related risks and optimize spending [1].

To effectively implement Software Asset Management, organizations must integrate SAM tools into their broader IT asset management strategy. This integration ensures a comprehensive approach to managing software assets throughout their lifecycle, from when they are first needed until they aren't used. By leveraging SAM tools, organizations can achieve greater visibility and control over their software inventory, avoid potential legal and financial penalties for non-compliance, and make decisions based on the data they possess to optimize software usage and reduce unnecessary costs.

2.2.1 Key Activities

According to *Gartner*, there are ten key activities that a SAM should have. These are represented in Figure 2.1 as a cyclical diagram and are the following [1]:

- Ability to perform platform discovery - it should query all platforms linked to the company and find the platforms where software is executed.
- Ability to perform entitlement discovery - by using different systems provided by the license providers, it should obtain all data regarding contracts, purchase, and procurement records.
- Identify license usage regardless of platform - using the same systems mentioned previously, it should be able to verify when a certain license was used, and, if needed, where it was used.
- Identify entitlements - by analyzing the contracts and purchases of software licenses, it should correctly identify the entitlements presented by a given license contract.
- Normalize consumption data - by using different data sources provided by license providers, it should consolidate this data into a single, organized inventory of software consumption.
- Normalize entitlement data - using the entitlement records of each license agreement, it should consolidate the data into a single, organized inventory of entitlements for each license provider.
- Reconcile entitlements with consumption data - Correlate data to create a basis for different operations on the software licenses, such as contract negotiations, spending optimization, and risk reduction.
- Operate on license providers - it should allow the proper allocation of software licenses, identify incorrect data or noncompliance of entitlements, and automate the workflows in place in companies.
- Optimization - it should optimize the overall spending of a company, by taking leverage of changes to license structures and by forecasting the demand for the specific type of license.
- Share information - it should be possible to share the information produced by the tool with other teams inside the company.

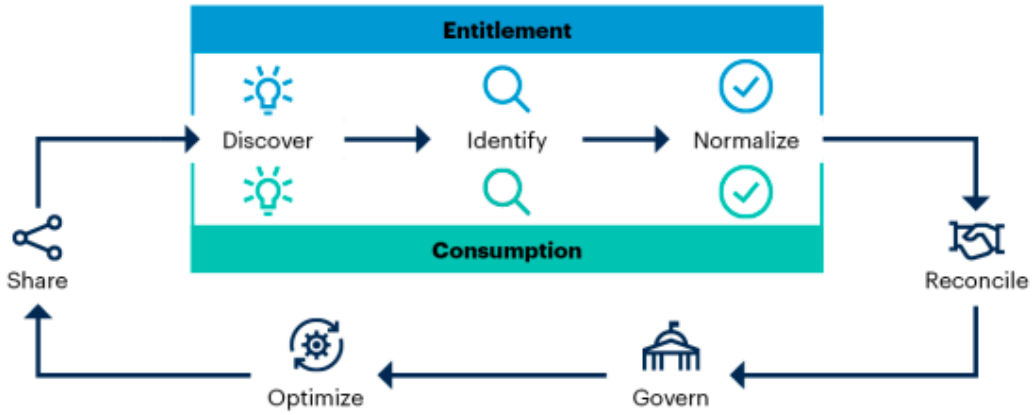

Figure 2.1: The 10 key activities of a SAM tool [1]

2.2.2 Benefits

There are multiple benefits of finding a high-quality SAM tailored to the needs of each company. It has a clear impact on companies' finances, as it can drastically reduce costs due to high scrutiny of the licenses that employees use. By using these tools, people responsible for each license buy only the needed quantity, without overspending by mistake [4]. However, even though these types of systems have multiple benefits, they sometimes become unused, mainly due to having some of the metrics or licenses missing from the SAM, as it becomes harder to aggregate all of the data into one place. Therefore, to properly improve spending and facilitate the work of employees, it is of the highest importance to have a clear notion of the processes and needs of the companies that a SAM is integrated into [1].

Despite some of these features involving the automation of processes, this same study has shown that not all of the work should be handled by the tool used. There needs to be a connection between the tool and the financial teams in the company [1]. However, with current advancements in Artificial Intelligence (AI), some of this work can be aided by using a Large Language Model (LLM), such as OpenAI's Generative Pre-Trained Transformers (GPT) [6]. By using these technologies and feeding them with the SAM data, companies could have clear paths on cost reduction by asking the GPT for ways to plan these reductions in expenses.

2.3 Existing products comparison

During this section, there will be an overview of some of the SAM tools existing in the market. These have been selected based on the products displayed in the Gartner report that was previously mentioned [1]. These tools are categorized based on four different types of SAM tools: Traditional SAM tools, which help manage licenses from multiple publishers, across on-premises infrastructure and cloud platforms; SAM tools for SAP, which focus specifically on software provided by SAP; SAM tools for engineering and specialty software, which focus on specialty engineering software, like civil engineering software or electrical engineering software, for example; SAM tools for Software as a Service (SaaS), which focus on the usage and optimization of SaaS, based on the data distributed by the provider's API. Based on Mindera's needs, this comparison will be focused on products that belong in two of these four categories: Traditional SAM tools and SAM tools for SaaS.

After the categories have been selected, four tools that had both these categories marked were selected: *Snow License Manager*, *Certero SAM*, *Service Now*, and *Flexera*. Along with these, a tool called *Snipe-IT* was also selected, since it is already used by the security department in Mindera.

To compare these products fairly, a set of features has been selected. These include some of the points previously mentioned in the reports, as well as some capabilities that Mindera considers essential to the software that the company needs to use:

1. Possibility of automating processes - issuing or revoking licenses automatically.
2. Integration with license providers - the products that integrate with Mindera's most commonly used license providers will be favored (for example, *Figma* or *JetBrains* licenses).
3. Integration with external authentication providers - possibility to integrate with authentication services such as *Google Identity* [7], with the possibility of having roles for specific users.
4. Ability to integrate with external services - should be able to integrate with companies' human resources platforms to automatically trigger license changes based on actions done in these platforms.
5. Dashboard - should be able to visualize overall spending and filter by licenses and/or their types.
6. Pricing
7. Integration with AI - should be able to optimize costs and/or provide a help desk for employees to use whenever they need to clarify any questions regarding licensing. This feature isn't mandatory, but is nice to have.

2.3.1 Snow License Manager

Snow Software is a unified, cloud-native platform that delivers actionable insights on Information Technology (IT) companies. It is used by some of the world's largest enterprises and currently has over 2500 customers [1, 8].

This software offers many different tools, with the one most relevant for this study being the *Snow Software SAM*. This tool provides an inventory dashboard with usage data for some digital assets while reconciling this information against discovered software, giving an up-to-date view of the data [8].

This particular SAM also provides a tool called *Snow Spend Optimizer*, which identifies savings opportunities from unused accounts, duplicate users, or redundant applications. It also helps companies buy and renew only the licenses they need, by identifying overlapping technologies used in the organization [8].

Regarding authentication, *Snow Software* provides integration with existing Active Directory (AD) groups to provide a seamless single sign-on experience [8].

Lastly, this tool also provides SaaS management, providing a comprehensive view of the organization's SaaS environment, while providing all of the features previously mentioned to over 23 000 license providers and providing scripting languages to automate workflows [8].

Regarding this software's pricing, *Snow Software* doesn't disclose a base amount for this service, as they require companies to contact them to find the most suitable monthly rate for these services.

2.3.2 Certero SAM

Certero provides software that presents a unified platform to manage hardware, software, SaaS, among other things. It provides a dashboard to easily track software license costs, as well as a tool to automatically optimize a company's spending, stating that, in some cases, it optimizes 20% to 30% of unused software licenses [1, 9].

It provides management for different types of SaaS licenses, for example, *Slack*, *Zoom*, *Google Workspace*, among others. For these licenses, it provides integrations with the tools' own API, to automate some of the workflows [9].

There are, however, multiple important topics that aren't mentioned throughout *Certero's* website. For instance, it isn't explicit if it is possible to have external integrations to remotely trigger certain processes, if it's possible to integrate with external authentication providers, or if it is possible to integrate this data with AI tools [9].

Lastly, pricing is never mentioned throughout the website, as it is needed to contact the company directly to have a monthly rate established depending on the size of the company and the features that are needed.

2.3.3 ServiceNow

ServiceNow is a cloud platform company with a heavy focus on enterprise IT management. Its IT asset management services offer four different products, with one of being its own SAM. It provides automated workflows, software compliance, optimization, and cloud discovery. Along with this, one of the plugins available allows for SaaS license management, integrating with over 35 SaaS API integrations, with some of the most notable ones being *Confluence Cloud*, *GitHub*, *Jira Software*, *Slack*, and *Zoom* [1].

To automatically process some of the workflows, *ServiceNow* provides a scripting language that uses the API integrations to trigger these specific actions, which include creating licenses, revoking licenses, fetching a specific license usage, among other features [10].

Despite having many of the wanted features, it does not provide a way to integrate with external processes to trigger certain workflows automatically. It also doesn't allow integration with external authentication providers and doesn't provide a tool to automatically optimize a company's licensing costs, as it only provides a dashboard to analyze the costs that these licenses bring to companies [10].

As for the pricing, there isn't any publicly released information on it, as *ServiceNow* advocates for personalized pricing for each customer, needing to contact them to know what this amount totals to.

2.3.4 Flexera

Flexera provides many different products regarding IT asset management, with two of them being its Software Asset Manager and their SaaS manager. These two products offer license

discovery, inventory, normalization, and optimization for both traditional and SaaS licenses [1, 11].

This platform provides multiple dashboards where data is transformed into actionable intelligence, reducing risk and optimizing how much companies spend. It automatically removes unused software and automates the asset life cycle, giving companies the leverage to negotiate contracts and control how much they spend on software based on actual usage [11].

Specifically for SaaS management, it claims to support over 32 000 SaaS applications, with API level integration with the major ones, without specifying which products it does support. By using the available plans of each vendor and correlating it with the actual usage of the licenses, *Flexera* recalculates subscriptions to make companies pay as little as possible, giving suggestions on what needs to be renegotiated. One key feature it provides is the tracking of SaaS renewals, even enabling auto-renewals for some products, creating optimal renewal strategies [11].

It allows for the automation of workflows, such as license creation, through their license provider integrations, and it also provides access to an internal API, used to gather metrics and data related to assets for companies that want to build custom reports with the data provided. It does not, however, provide integration with Google's Lightweight Directory Access Protocol (LDAP), making it harder to integrate employees into the product's user list [11].

Regarding pricing, *Flexera* remains quite vague throughout its website, only specifying how much its clients can save instead of stating how much their product costs. Therefore, their price is only made known after a meeting is scheduled with them.

2.3.5 Snipe-IT

Snipe-IT is an open-source web-based software, possible to run on any web server [12]. It is a full-fetched asset manager, so some features won't be included in this study, such as device management-related features.

Regarding *Snipe-IT's* license management functionalities, it provides a dashboard with some very basic metrics, such as the total number of licenses registered in the system, and various groupings by categories, status, or manufacturer. Despite this, it does not store any information related to the license price, and, therefore, does not provide metrics to see how much the company is spending and how it may be optimized. It provides a JavaScript Object Notation (JSON) Representational State Transfer (REST) API so that its workflows are integrated with companies' systems and, since this tool is open-source, there are multiple Software Development Kit (SDK) being developed by the community to ease integrations even more.

Regarding user management, it allows companies to synchronize their Google Secure LDAP users with the tool, making it so that everyone can access the software, with each of these users having different policies associated with them, which means that every person can have customized roles depending on the features they can access.

To notify users whenever a license is requested or returned, it provides different notification methods whenever a license is requested or returned, such as *Slack* notifications or emails.

Despite having specific areas for license management, it does not perform automatic actions directly on the provider level, as it does not have any kind of integration with license providers. It also does not have any specific AI related features.

Regarding pricing, since it is open-source software, it allows for self-hosting the tool for free, with the only costs being the companies' infrastructure costs needed to properly run the software. However, it also offers some cloud-hosted plans, with prices ranging from $399.99 to $2499.99 billed annually. These paid plans also offer some specific features, such as automatic backups, automatic upgrades whenever a new version is released, and automatic server maintenance, among other features.

2.3.6 Comparison conclusions

In Table 2.1, we can see a table showing which features each of the products investigated currently provides. These features are the same as the ones listed in section 2.3, with the following correlation:

- Feature 1 - Possibility of automating processes
- Feature 2 - Integration with license providers
- Feature 3 - Integration with external authentication providers
- Feature 4 - Ability to integrate with external services
- Feature 5 - Dashboards

After researching the listed products, it is clear that they all have their similarities. All of them allow the automatic creation and assigning of licenses and, some of them, perform the process end to end. That is, they register new licenses directly on the license provider, registering more or deleting them whenever it is needed.

Regarding integration with *Google's* LDAP, only two of them provide this feature, which is somewhat important for some companies as they increase in size.

One feature that, surprisingly, is lacking in most of these products is the ability to remotely trigger workflows through, for example, provided APIs. This makes it increasingly harder to integrate into companies that already have some sort of management software in place.

In the topic of data visualization dashboards, all of the researched products provide their versions of data dashboards, some with very complete data, and others with very little information regarding the stored data.

Pricing is, surprisingly, a topic that most of these products tend to avoid at all costs. Most of them never mention it unless companies contact them directly, which makes it hard to know if a company is being offered a fair price for the service they are going to pay for. The only product that was transparent in this aspect was the open-source software, *Snipe-IT*.

Lastly, AI integration seems to be an area that none of these products seem to have dwelled into yet. However, despite not being available in any of these tools, it may be something worth investigating for the solution to be developed, since it is constantly improving and providing better results as time goes on.

	Feature 1	Feature 2	Feature 3	Feature 4	Feature 5	Pricing	AI Integration
Snow License Manager	Yes	Yes, but not all used by Mindera	Yes	No	Yes	Unknown	No
Certero SAM	Yes	Yes, but not all used by Mindera	No	No	Yes	Unknown	No
ServiceNow	Yes	Yes, but not all used by Mindera	No	No	Yes	Unknown	No
Flexera	Yes	Maybe	No	Maybe	Yes	Unknown	No
Snipe-IT	Yes	No	Yes	Yes	Yes	$399.99/year	No

Table 2.1: Product comparison using the features described in Section 2.3

2.4 Conclusion

As *Gartner* reported, it is extremely hard to find an existing solution that encompasses the vast array of licenses that companies support. That is even more evident after observing Table 2.1, as none of the researched products provided all of the features that *Mindera* was looking for. However, this study provided crucial information on how to maximize the benefits provided by SAM systems, and which main features they should have, which will be essential for the next chapters in this report, where a system of this kind will be created and integrated with *Mindera's* systems.

Chapter 3

Analysis and Requirements

This chapter will present the requirements for this project based on the objectives traced in section 1.2. To do so, multiple aspects of the system will be analyzed, such as its: actors, functional requirements, and non-functional requirements. Lastly, the domain model representing the problem at hand will be presented based on the previously listed requirements.

3.1 Requirements Engineering

Requirements Engineering (RE) is a crucial phase in the software development life cycle. It encompasses the process of identifying and understanding the needs and requirements of stakeholders, and then documenting them in a manner that provides a foundation for all subsequent system development activities. This process involves gathering, analyzing, documenting, validating, and managing the requirements of the proposed software system as expressed by the relevant stakeholders [13].

3.1.1 Actors

Actors are individuals, groups of people, or systems that interact with the described system. Therefore, this subsection will identify the actors interacting with at least one part of the system. These are:

Name	Type	Description
License Manager	User	A person who manages one or more software licenses used by Mindera's employees
Employee	User	An employee of Mindera
System Administrator	User	User that controls the system
License Manager System	System	The system to be developed
Mindera People	System	Internal software developed at Mindera which manages employee's information

Table 3.1: Actors that interact with the system

3.1.2 Functional Requirements

After analyzing the objectives defined in 1.2 and the actors listed in 3.1.1, Figure 3.1 displays the functional requirements directly related to the main actors. These will aid in the design

and implementation phases, as they will make sure the stakeholders' expectations are aligned with this system's developments.

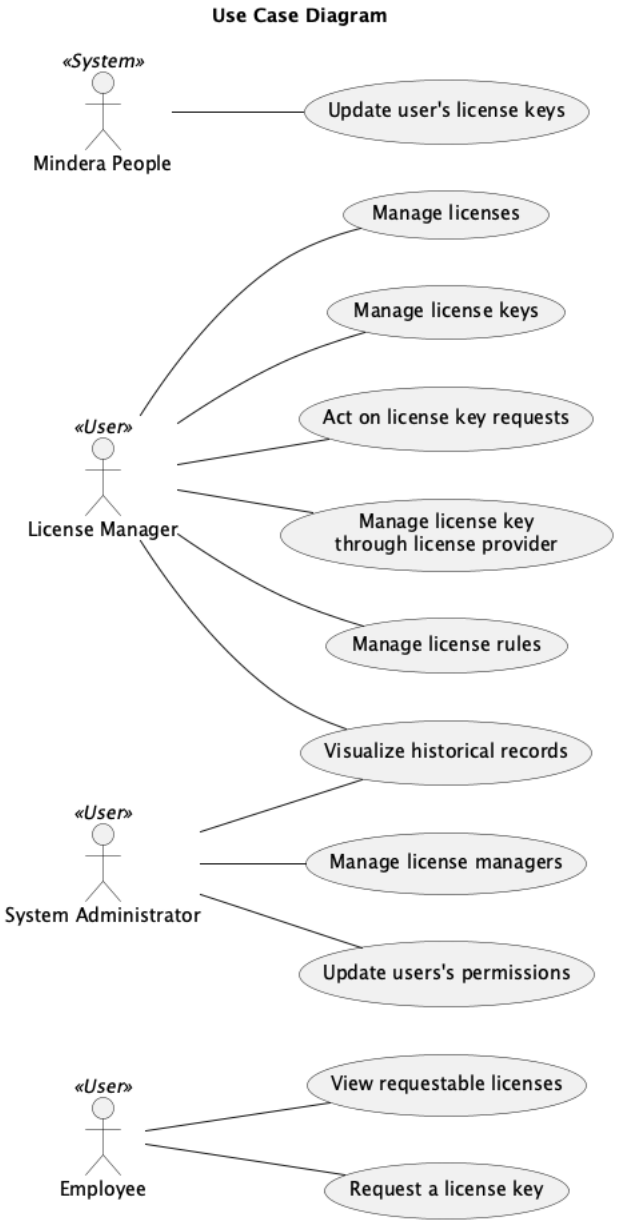

Figure 3.1: Functional requirements and stakeholders

Table 3.2 showcases these same functional requirements in a more detailed overview and with a brief description of each requirement.

Identifier	Actor	Title	Description
FR1	License Manager	Manage licenses	Manage licenses information.
FR2	System Administrator	Manage license managers	Add or revoke users' management rights to a license
FR3	License Manager	Manage license keys	Handles the distribution and revocation of license keys.
FR4	Employee	View requestable licenses	Displays a list of licenses available for request.
FR5	Employee	Request a license key	Submits a request for a new license key.
FR6	System Administrator	Update users' permissions	Change users' type.
FR7	License Manager	Act on license key requests	Responds to requests for license keys.
FR8	Mindera People	Update user's license keys	Automatically create and/or assign a license key to a user, based on messages from Mindera People.
FR9	License Manager	Manage license key through license provider	Requests, revokes or requests usage of a license key from an external license provider.
FR10	System Administrator and License Manager	Visualize historical records	Allows administrators to view past license management actions.
FR11	License Manager	Manage license rules	Modifies rules governing the use and distribution of license keys.

Table 3.2: Functional requirements with their description

3.1.3 Non-Functional Requirements

Non-functional requirements are considered some of the most important and critical requirements for software development. If these are not considered at early stages of software development, they may become very complex to address in later stages [14].

The FURPS+ model, a widely recognized framework, will be utilized to identify the non-functional requirements. FURPS+ stands for Functionality, Usability, Reliability, Performance, and Suportability. The "+" symbol is included to help remember additional considerations, such as: Design Requirements, Implementation Requirements, Interface Requirements, and Physical Requirements [15].

Using this framework ensures that not only are the functional requirements outlined in the project's objectives met, but it also clarifies the additional aspects the framework addresses.

This approach guarantees the delivery of a robust system that aligns with stakeholder perspectives while using the industry's best practices.

These requirements originated from stakeholder meetings based on the system's needs and are detailed in table 3.3.

Identifier	Category	Description
NFR1	Functionality	Every technological dependency the system uses must be free and open-source.
NFR2	Functionality	Sensitive data, such as license keys, should be encrypted before being stored in the database.
NFR3	Usability	The user interface should be responsive and accessible on various devices.
NFR4	Performance	The system's backend should return responses to any requests in less than 2 seconds.
NFR5	Implementation Requirement	The system should integrate with Mindera People's message broker and listen to messages sent to employee-related topics.
NFR6	Implementation Requirement	The system should be integrated with Slack's API to be able to send automated messages to its users.

Table 3.3: Non-functional requirements and their descriptions

3.1.4 Domain Model

Following the requirements previously listed, a comprehensive domain model detailing the key concepts of the business was conceived. This model outlines essential concepts and their relationships, which are necessary to accurately represent the business's operations and processes, making sure there is a robust foundation for the system design and implementation phases. This diagram is represented in figure 3.2.

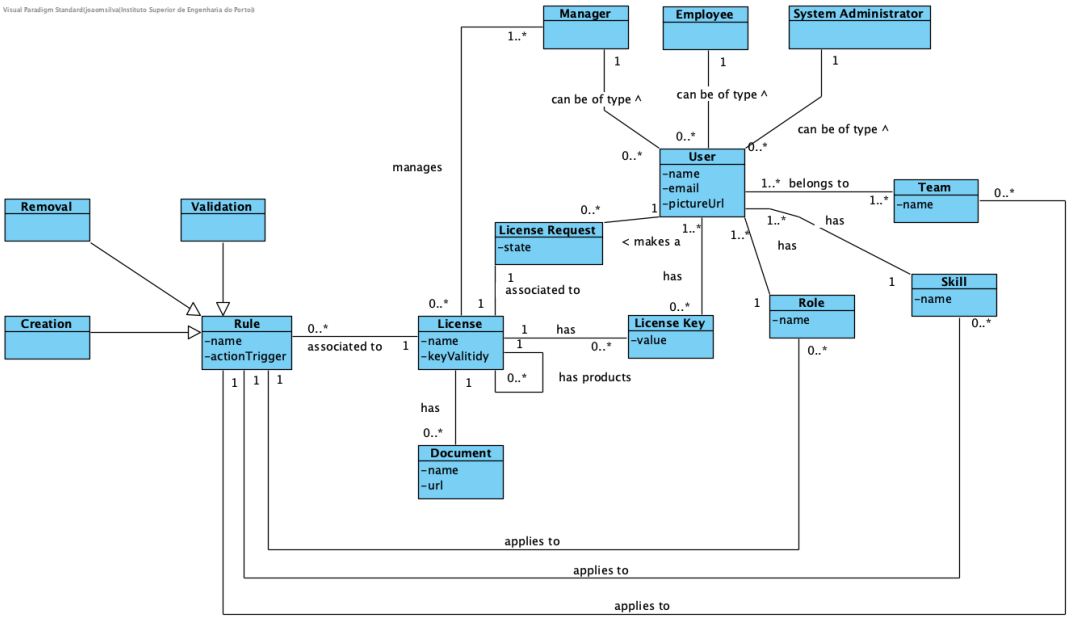

Figure 3.2: Functional requirements and stakeholders

To help understand the domain model, a glossary was elaborated to describe each concept mentioned. This glossary can be consulted in table 3.4

Concept	Description
User	An individual who interacts with the system.
Manager	A user with the authority to oversee and manage licenses, typically responsible for approving or rejecting license requests.
Employee	A user who requests licenses for the software they need to perform their job duties.
System Administrator	A user with administrative privileges, responsible for maintaining the system, managing users, managing rules, and handling overall license management.
License	Represents a software license and all the details related to it. Can have children licenses (products).
License Key	A unique key associated with the license, often required to activate the software.
License Request	A formal request made by an employee to obtain a software license.
Rule	A set of periodic actions that govern the issuance, revocation, and validation of software licenses.
Creation	Rules responsible for the creation of licenses.
Removal	Rules responsible for the removal of licenses.
Validation	Rules that validate if a user still uses a license key.
Team	A software development team to which a user belongs.
Role	An employee's area of expertise. (e.g. Backend Developer)
Skill	An employee's specific skill. (e.g. Java Developer)
Document	Represents any relevant documentation or files related to licenses, such as contracts or terms of service.

Table 3.4: Glossary of the project's business concepts

Chapter 4

Design

In this chapter, the insights gathered from the previous chapter *3 - Analysis and Requirements* will be used to outline a detailed implementation solution. Additionally, where applicable, alternative approaches will be discussed, with explanations provided for the selection of the chosen approach.

4.1 Level 1 (Context)

This view shows the system in little granularity, presenting it as a whole.

4.1.1 Logical View

In the view in figure 4.1, the system is represented as a single component, *MinderaSAM*, which exposes a single interface: *Frontend Web UI*. This single component also consumes at least three APIs, *Mindera People Google Cloud Platform (GCP) Pub/Sub*, *Slack Hypertext Transfer Protocol (HTTP) API* and *LicenseProvider1 HTTP API*, thus satisfying both NFR5 and NFR6. This diagram, however, shows *MinderaSAM* consuming more than one license provider API, as the system will consume a variable number of this type of APIs, depending on how many are configured in the system.

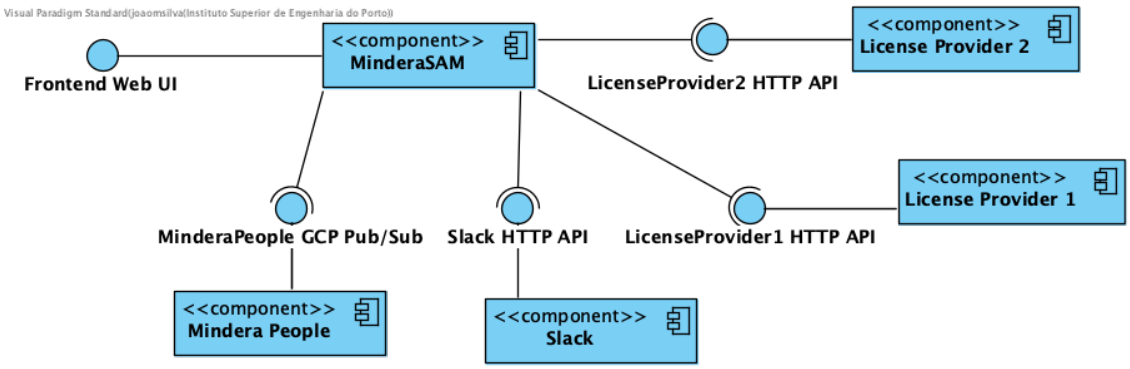

Figure 4.1: Level 1 logical view

4.1.2 Process View

The view in figure 4.2 shows the interactions made between a user and the system as a whole. These will be the only actions presented in the process views, as they make for most of the system's interactions.

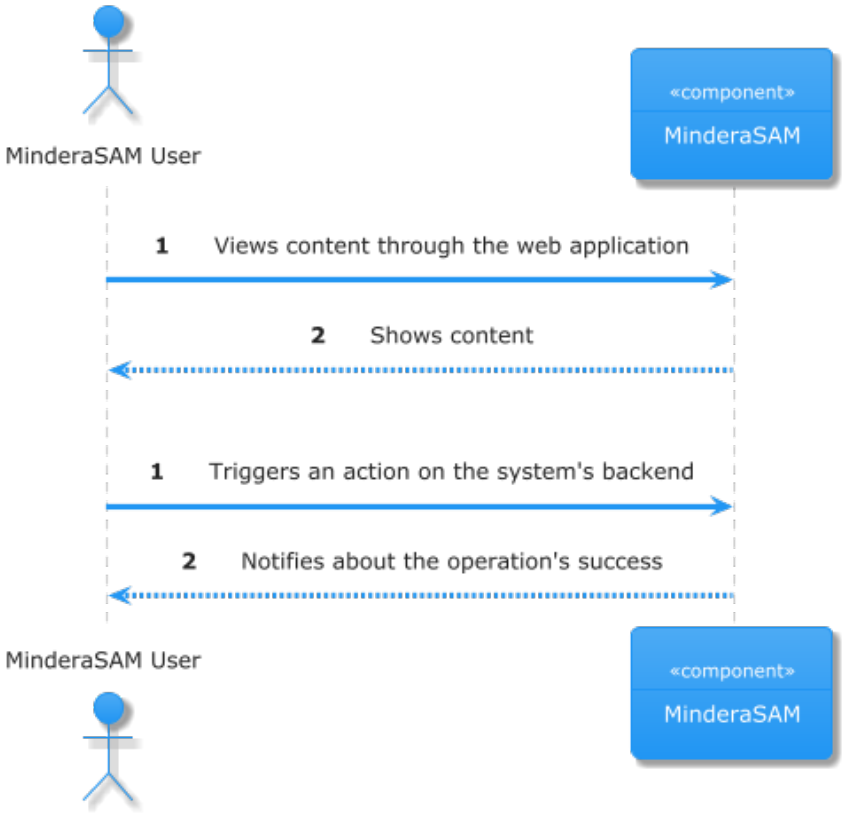

Figure 4.2: Level 1 process view

4.2 Level 2 (Container)

The level in this section explores a finer granularity within the system, showing how the different components within the system interact with each other and with the external components consumed by the system.

4.2.1 Logical View

For this view, two versions were elaborated, firstly an alternative which divided the system into microservices, and the final one where a service approach was followed.

First Iteration

At first, it was considered that a microservices approach would be beneficial for the system. This way, the core of the system, the software license management, would be separated from the rest of the responsibilities of the system, which, in this case, are the reporting functionalities and the communication with external license providers. In this approach, there would be three backend services: *Core Service*, *Provider Service*, and *Reporting Service*. It could be possible to separate the first service into even more microservices, but since the data handled by this service is very dependent on relations between the concepts, it would require a lot of maintenance to ensure the data is correctly synchronized between the different microservices.

In this approach, the *Core Service* would be responsible for most of the actions done by the system, such as license management, and user management, among others. This system would then communicate with *Provider Service*, acting as an adapter between the system and the license providers. Lastly, *Reporting Service* would be utilized mainly for the generation of reports, as the name suggests. This service would consume a message broker fed with messages created by *Core Service* and would have most of the historical data needed to build current reports, as well as previous ones, thus reducing the load on *Core Service*. For the user interface, there would be a web app that would consume the API on *Core Service* to perform any actions that the user might want to do.

One of the benefits of this approach is that, in this way, certain changes, such as integrating new license providers or changing the way reports are generated, could be developed without affecting the main functionalities of the system. It would also benefit the scalability of the system, as certain microservices could be horizontally scaled if they had high loads of volume, while other microservices that had less active times could be kept scaled down, optimizing the total resources needed to run the entire system. The level 2 logical view diagram representing this architecture is present in figure 4.3.

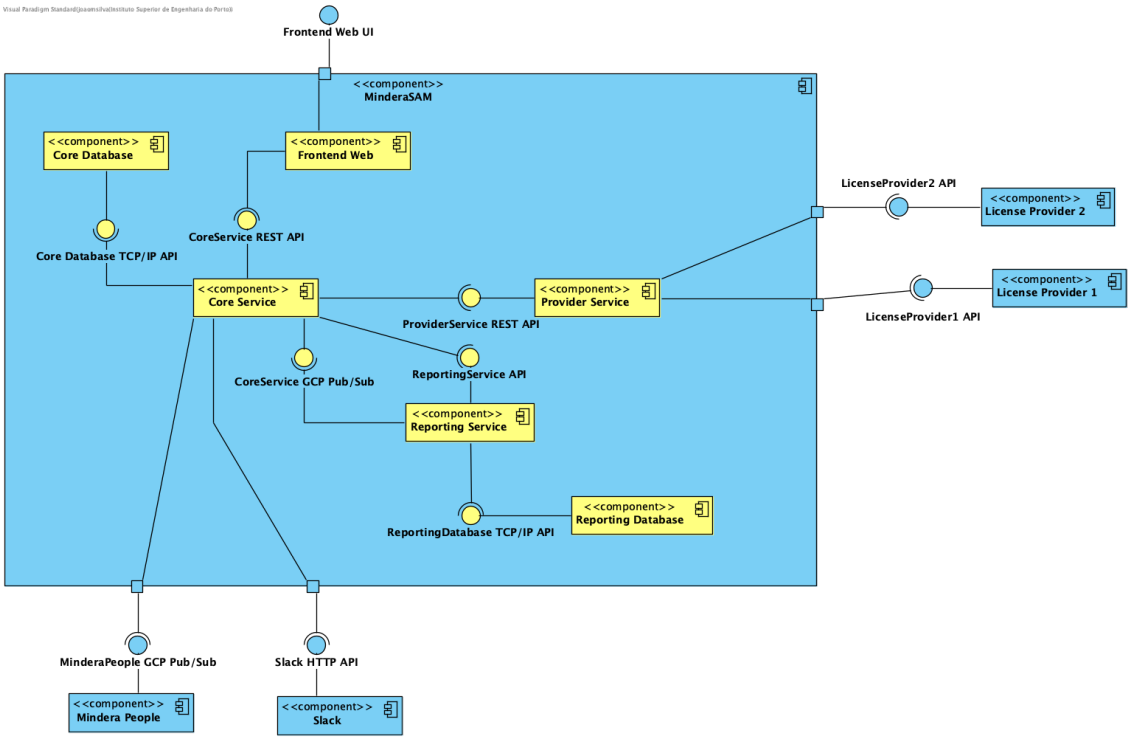

Figure 4.3: First iteration of the level 2 logical view

Final Iteration

For the second, and final, alternative, it was preferred to follow a monolithic approach. This was mainly due to the complexity and the amount of people that are needed to correctly follow a microservices approach within a company. Since there wouldn't be an entire team dedicated to this project at Mindera, and since this would be, for the foreseeable future, a system only used within Mindera, it was decided that the best thing would be to keep it as simple as possible and, in the future, if needed, the system would be split into microservices. With this, the system would be faster to develop for a smaller team, faster to build, easier

to debug and test, and would cost less to the company, as there would be less infrastructure to support the system.

In sum, the *Provider Service* and the *Reporting Service* were removed, with their responsibilities being handled by the *Core Service*. Regarding the user interface, it remained the same as the previous iteration. This new approach is presented in figure 4.4.

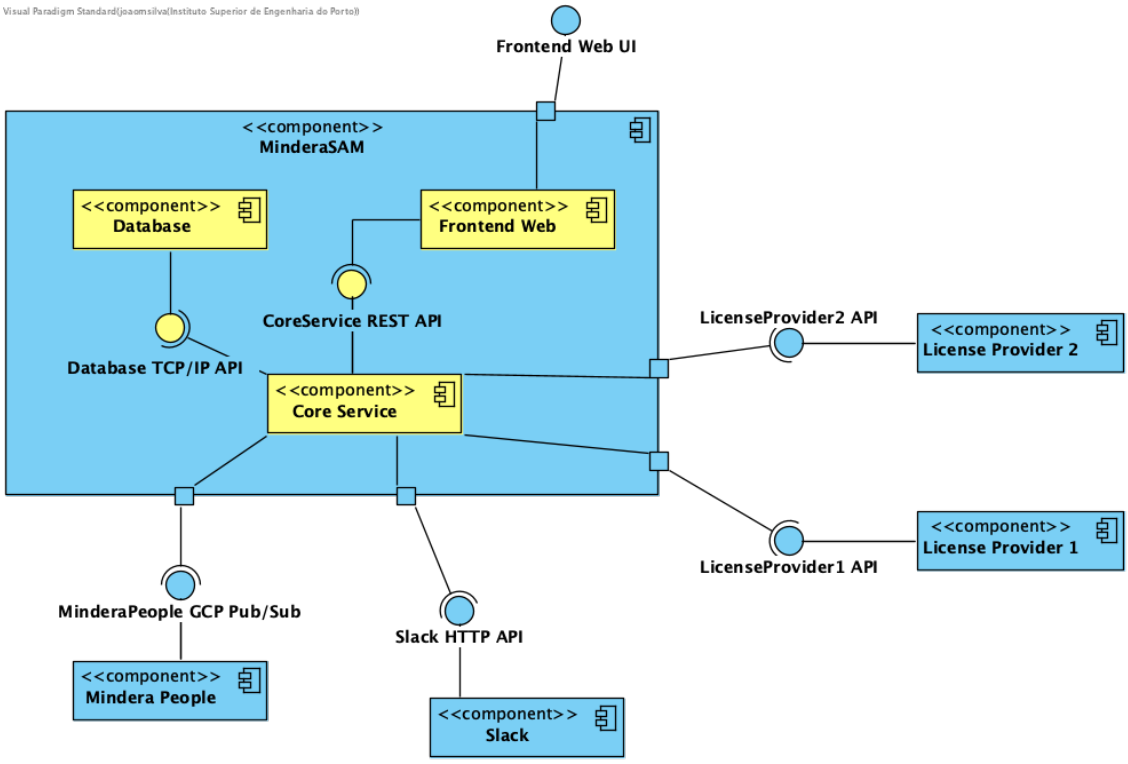

Figure 4.4: Final alternative of the level 2 logical view

4.2.2 Process View

In figure 4.5, the actions between the system's components are detailed, giving more insight on how the system will function. In this diagram, there are two types of flows detailed. The first one is where a user visualizes data on the system, while the second one details how inserting or updating data will work.

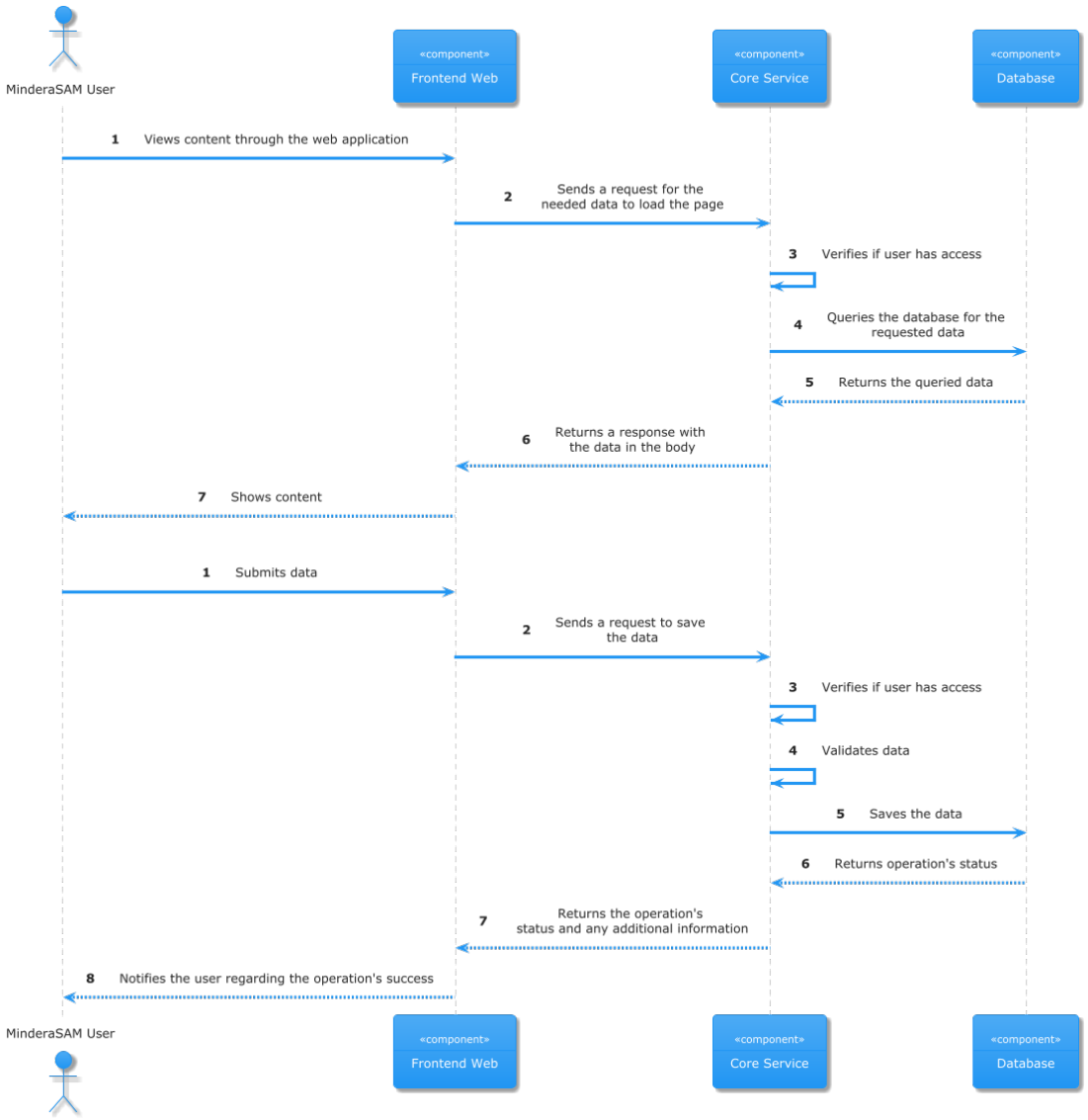

Figure 4.5: Level 2 process view

4.2.3 Implementation View

The implementation view showcases how the different modules within the system relate with each other. This diagram can be seen in figure 4.6

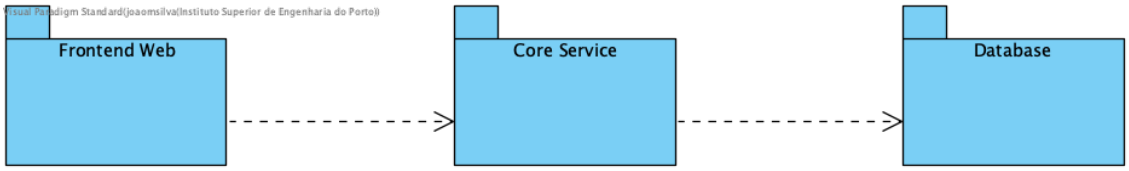

Figure 4.6: Level 2 implementation view

4.2.4 Deployment View

This view illustrates the physical architecture of the system, showing how its components are deployed onto hardware nodes, while also highlighting communication between the different nodes. This diagram is shown in figure 4.7.

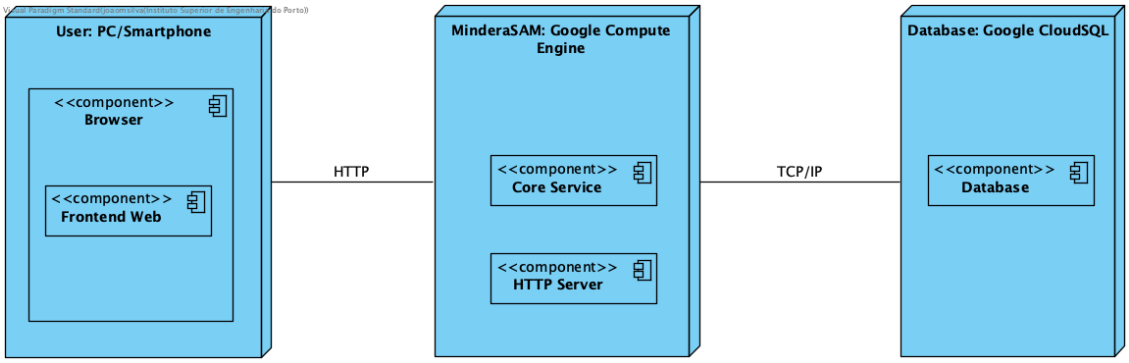

Figure 4.7: Level 2 deployment view

4.3 Level 3 (Component)

This view will focus on each of the components that need to be implemented, detailing them in a finer granularity.

4.3.1 Logical Views

Core Service

As described previously, the backend is composed of a single service: *Core Service*. To describe this service in more detail, the diagram in 4.8 was elaborated, which showcases how the different sub-components interact with each other and the external services. Here, the controllers create an API to be accessed by clients. These controllers consume the services within the system which, in turn, consume either the models developed or the clients that consume the information served by the external APIs. The services also consume the *Message Listeners*, a component that listens for any message published in the Mindera People's topics in GCP Pub/Sub module. Lastly, the models component consumes the database, either to fetch information or to save and update.

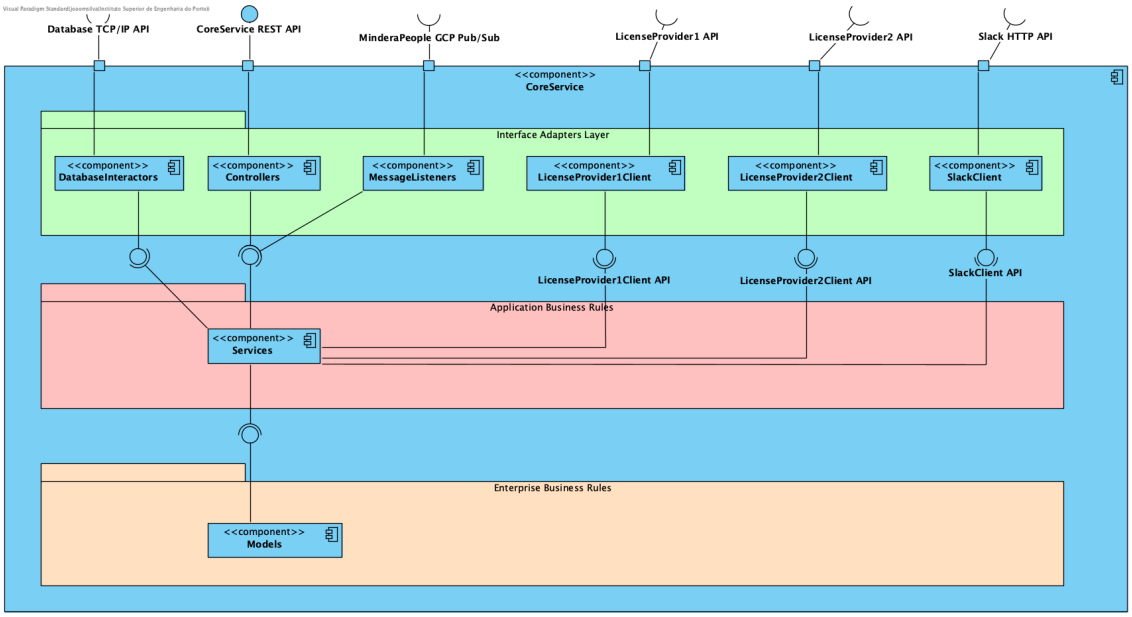

Figure 4.8: Core Service's level 3 logical view

Frontend Web

For the client-side part of this project, the diagram in figure 4.9 shows the *Frontend Web* component, which exposes a Router API which will be consumed by users on a browser, for example. This component will then consume the data provided by the *Core Service* API. Within the component, it has four different sub-components. Firstly, the *Router*, which loads a certain page depending on the Uniform Resource Locator (URL) the user visits within the system. The *Pages* sub-component then consumes two APIs, *Services*, to fetch the needed data to load the page correctly, and *Components*, which are the elements displayed within the page. These can also interact with the *Services* sub-component, in order to delegate actions to the *CoreService*, like saving or updating entities.

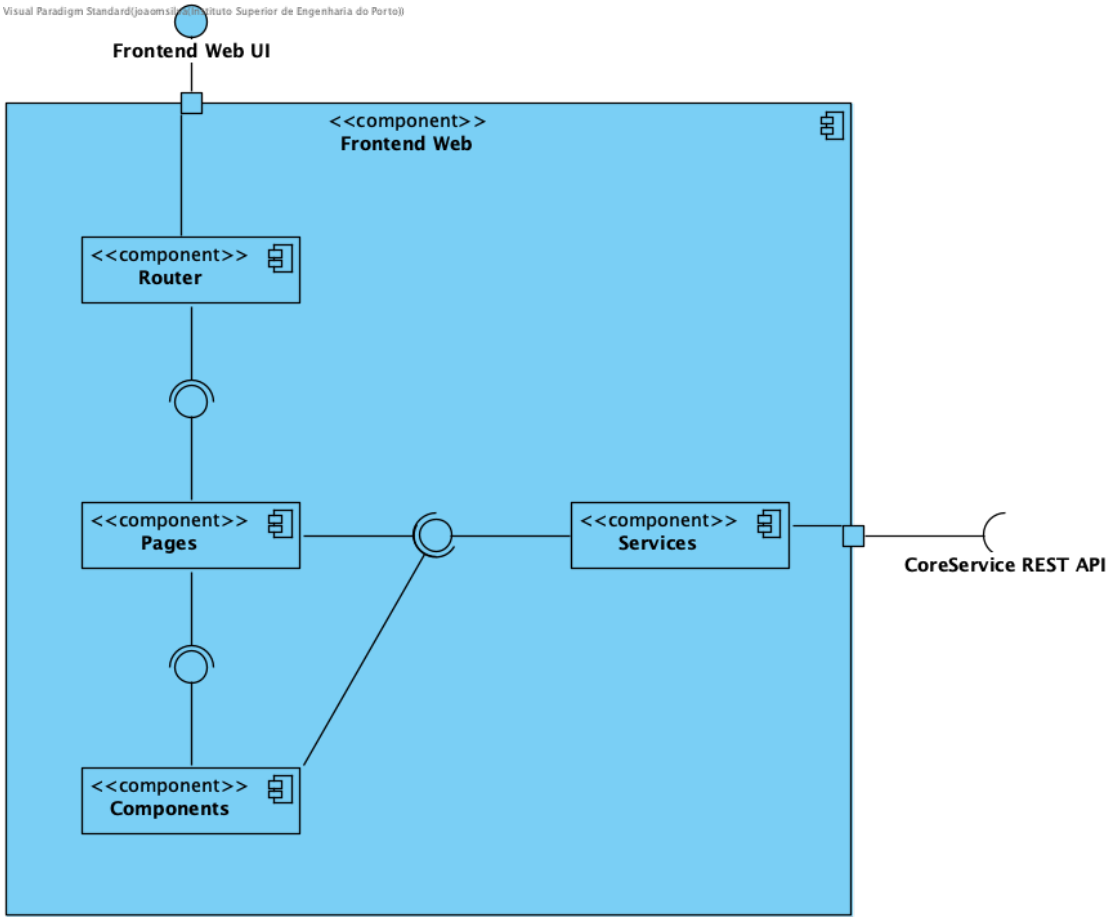

Figure 4.9: Frontend Web's level 3 logical view

4.3.2 Process Views

Core Service

In the diagram in figure 4.10, it is possible to see how the different components will interact with each other in most of the features that will be implemented.

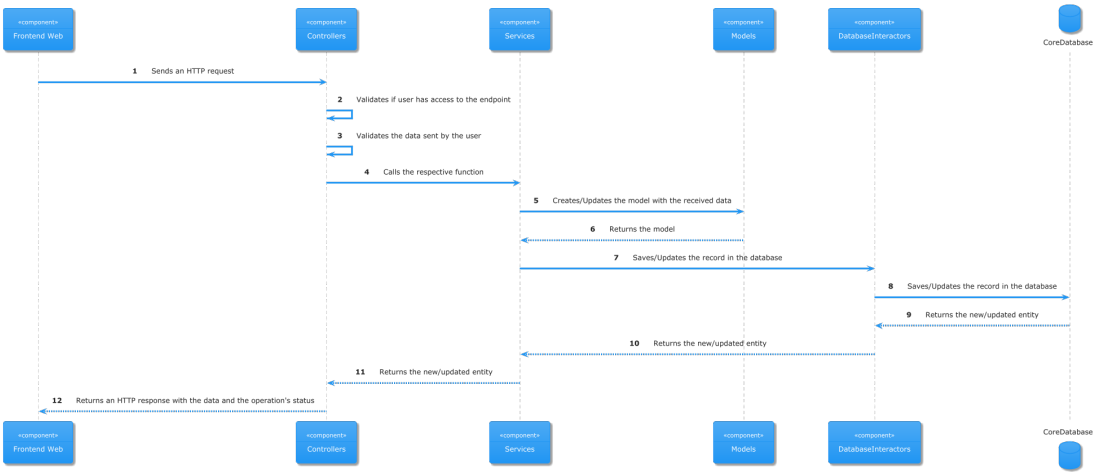

Figure 4.10: Core service's level 3 process view

Frontend Web

The diagram in figure 4.11 shows how a page is loaded and presented to the user, including the calls to other components within the system.

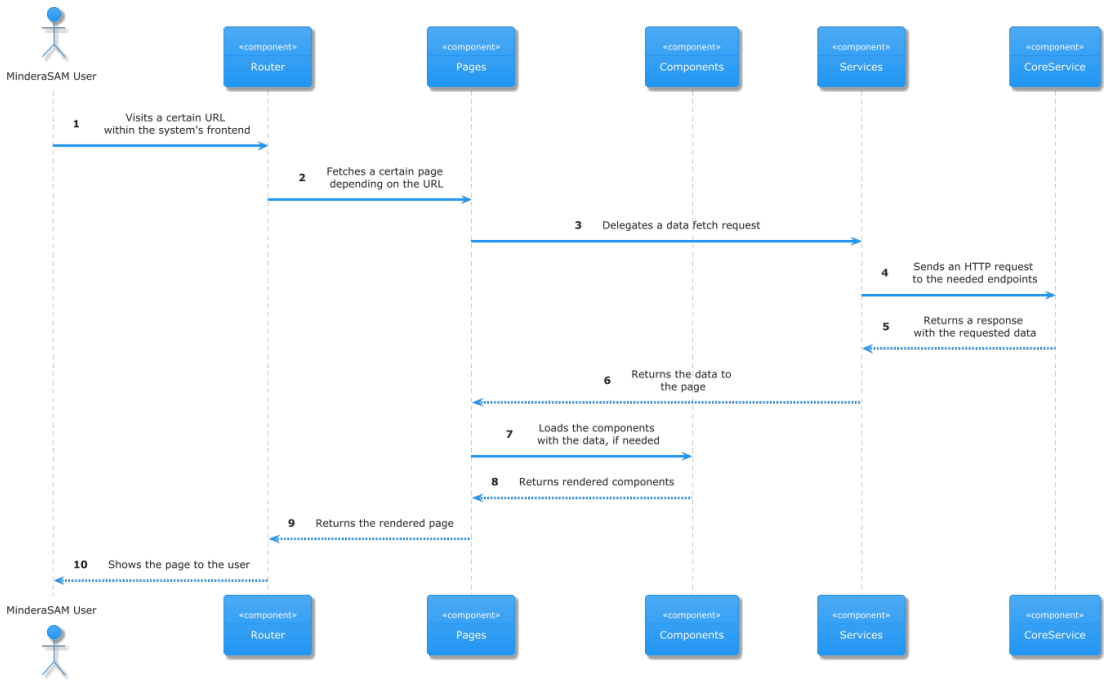

Figure 4.11: Frontend Web's level 3 process view

4.4 Level 4 (Code)

In this section, code-level structures will be detailed, giving a deep understanding about how the components will be implemented and how they interact with each other.

4.4.1 Process Views

This sub-section will showcase two important and complex cases within the system, highlighting how the different classes communicate with each other.

Consume "New Employee" Message

The case showcased in figure 4.12 is directly related to FR1, showcased in 3.1.2. Here, the flow starts with a message being published on the MinderaPeople's GCP Pub/Sub queue. Then, after receiving the message, the *MessageListener* communicates with the *UserService*, which applies rules via the *RuleService* to determine the appropriate actions. For each rule, a request to add a key is sent to the *LicenseKeyService*. If the action is manual, a license request is created and saved. However, if it is automatic, then the correct license provider client is fetched from *LicenseProviderClientStrategy*, which is then used to create a license key on the provider. Is is then followed by encryption of the generated key using the *EncryptionService*, which would satisfy NFR2. The key is then assigned to the user and saved in the database.

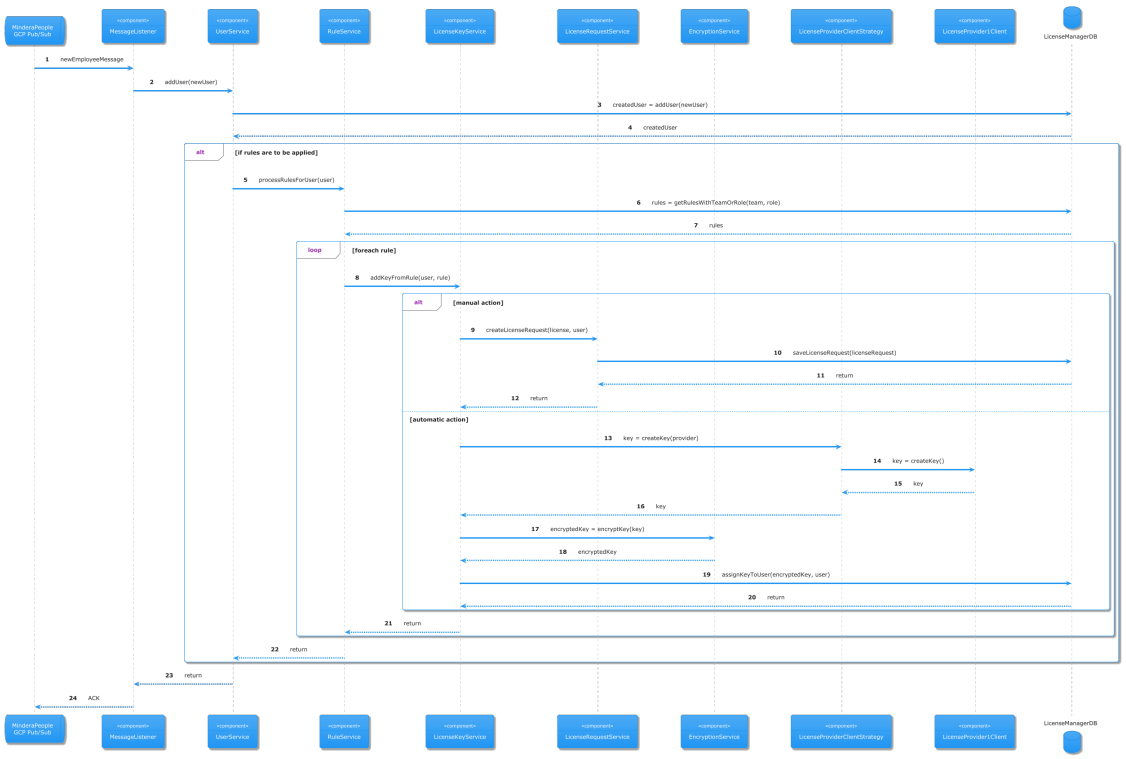

Figure 4.12: FR1 sequence diagram

Consume "Delete Employee" Message

The case showcased in figure 4.13 is directly related to FR2, showcased in 3.1.2. The flow begins with a message from Mindera People's GCP Pub/Sub, processed by the *MessageListener*, which triggers the *MessageListener* to update a user's information. The *RuleService* then checks for any applicable rules. If rules exist, a loop is initiated where the system removes the license key for each rule through the *LicenseKeyService*. Depending on whether the removal requires manual or automatic actions, the key removal process either notifies a manager that a license key should be unassigned and deleted, or directly proceeds with actions such as unassigning the key, updating the records in the database, and decrypting the key using the EncryptionService so that it can be sent to the license provider in order to be deleted. This is done by fetching the correct provider via the *LicenseProviderClientStrategy* and using the client to delegate the information.

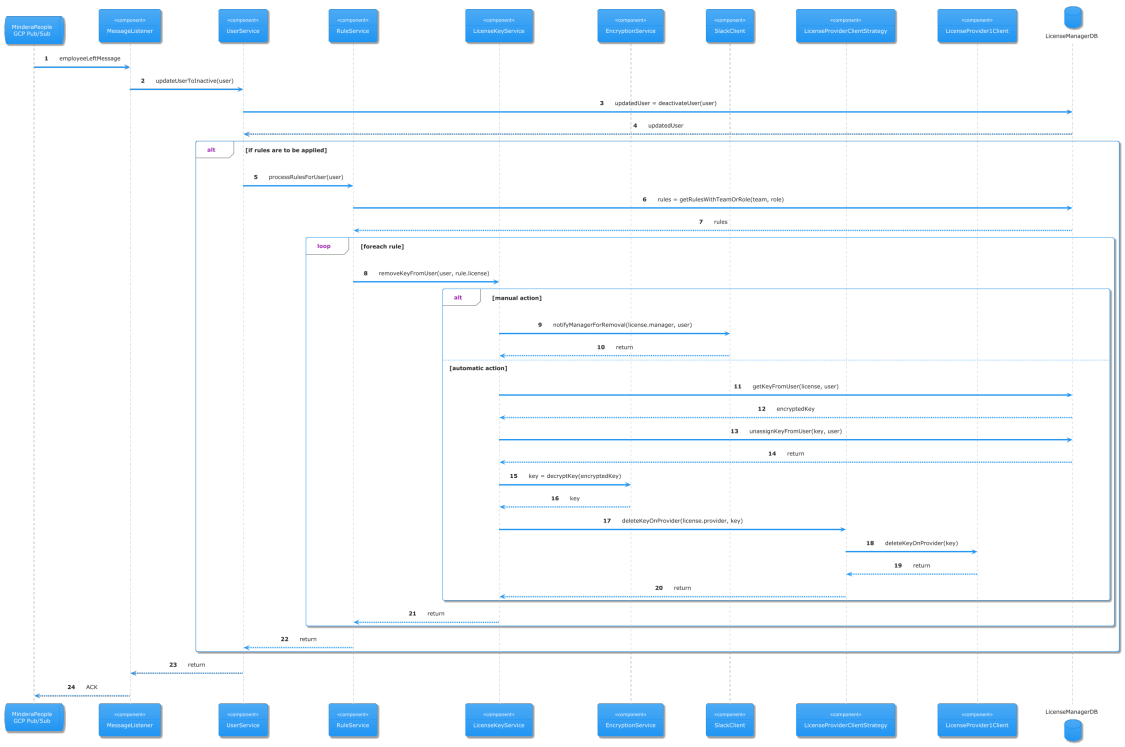

Figure 4.13: FR2 sequence diagram

4.4.2 Database Data Model

With the help of the domain model made in figure 3.2, an entity relationship diagram was developed, detailing every entity that will exist in the system, how they relate with each other, and which tables are needed to do the database normalization. The diagram constructed can be consulted in figure 4.14. This diagram shows that in this system, users, licenses, and rules interact in a structured manner. At its core are users, who are associated with teams, roles, and skills. Users can manage licenses, request them, and have specific license keys assigned to them. The license entity plays a central role, linking to other components like license keys, documents, and license requests. The rules system is modular, associating rules with different teams, roles, and skills, thus enabling configurable behaviors within the system. Additionally, the system logs user actions through an events entity.

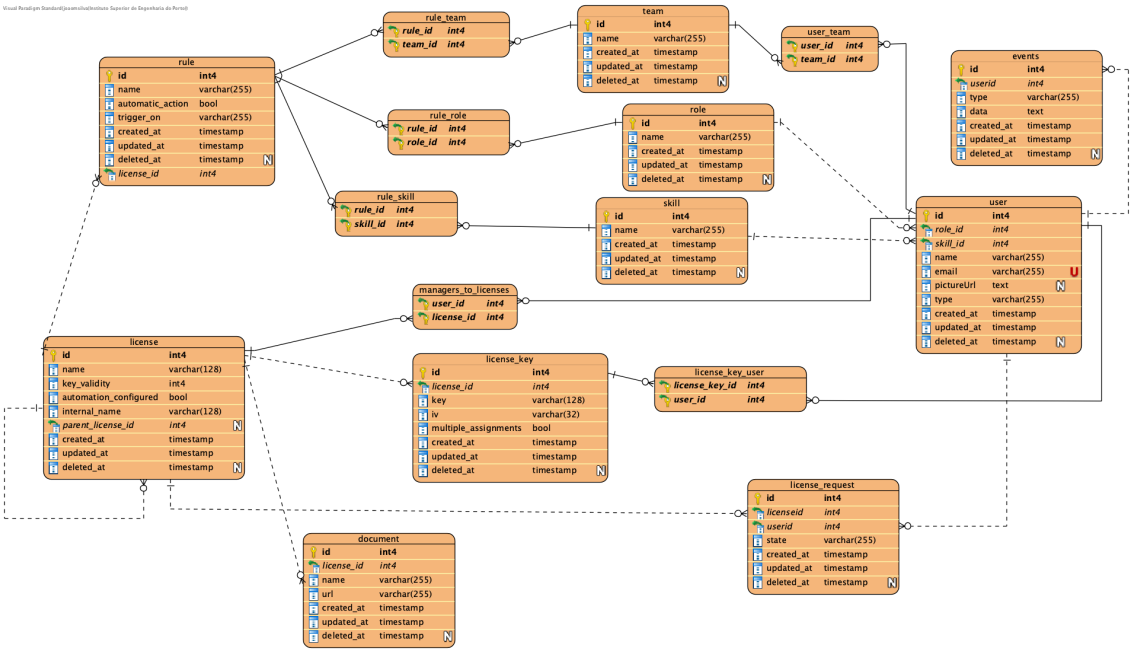

Figure 4.14: System's entity relationship diagram

Chapter 5

Implementation

This chapter will focus on detailing and following the implementation of a list of tasks that need to be done to start feature development.

5.1 Tasks

In table 5.1, the tasks needed to start the development of user stories are presented. These aren't directly related to a user but provide direct value to the development of the system.

Identifier	**Name**	**Description**
T1	Create HTTP server	Implement an HTTP server and create the base structure for endpoints to be created and implemented.
T2	Create entity model	Configure the database connection and create the files needed to properly create the entity model.
T3	Implement strategy design pattern for license provider clients	Implement the needed functions to have a working strategy to fetch license provider clients based on the license internal name.
T4	Implement license key encryption	Implement and configure a service to encrypt and decrypt strings.
T5	Configure frontend application	Bootstrap a frontend application to serve the system.
T6	Configure communication between frontend and backend	Create a HTTP client interface for the communication between the two components.
T7	Add authentication to the backend	Add authentication and authorization to router.

Table 5.1: List of Tasks

5.2 Technological Stack

This section will provide a detail and reasoning for the choices that were made for each technology chosen to implement the services previously mentioned in chapter 4.

5.2.1 Programming Languages, Libraries, and Frameworks

This subsection will start with listing some of the tools used to develop this system, followed by component-specific libraries and frameworks.

To build the services involved in this project, TypeScript was the chosen programming language, mainly due to its powerful static typing system, which significantly enhances code reliability and maintainability, as well as its ability to be both a server-side and a client-side programming language. With this, developers are able to have a much more integrated full-stack experience, as they don't have to keep switching languages between components. Along with this, in a system like a software asset management tool, where managing sensitive data such as license keys is crucial, TypeScript's type safety helps to prevent common runtime errors by identifying issues during development. This leads to a more predictable and secure development process, which is especially important in systems that involve numerous interactions between components [16].

Bun was selected to act as the JavaScript runtime, to optimize performance across the system. One of Bun's key advantages is having native support for TypeScript and JSX, which simplifies the development process by eliminating the need for separate compilation steps, thereby reducing build times and enhancing developer productivity. Bun's performance ensures that tasks are handled with minimal latency, resulting in a responsive and smooth user experience. Benchmarks show that Bun operates over 4x as fast as it's more commonly used competitor, NodeJs [17].

The combination of TypeScript and Bun provides the flexibility and scalability necessary for the project's long-term success. With Bun's built-in tools, including a fast bundler and native support for various web APIs, it is possible to achieve a much faster development as well as higher performance which, when combined with Typescript's typing system, results in a strong foundation for any kind of project. With these, the system's objectives are easier to achieve while allowing the system to be adaptable for any needed changes or expansions in the future.

5.3 Implementation Details

In this section, the implementation of the different tasks and functional requirements, defined in subsection 3.1.2, will be described in detail, as well as any external libraries that might have been used to ease the development process. There are, however, some functional requirements that weren't implemented due to time constraints and, as such, will be omitted from this section.

5.3.1 T1 - Create an HTTP Server

To accelerate development and reduce the possibility of runtime errors happening, Elysia was selected as the framework to be used alongside Bun. This decision was made due to its lightweight, performance-driven nature, making it well-suited for building fast and scalable applications. Elysia's minimal overhead allows for efficient processing of high volumes of API calls and its focus on performance ensures that operations can be executed swiftly, which is critical in maintaining a responsive system with low latency. Another key advantage of Elysia is its developer-friendly design, which simplifies the process of setting up routes, handling middleware, and managing dependencies. This allows for a clean, modular architecture that is easy to maintain and scale as the system grows. Its flexibility enables continuous

integration with TypeScript, ensuring type safety and clear structure throughout the backend development process [18].

To create an HTTP server with Elysia, the code in listing 5.1 needs to be implemented. With this simple snippet, a route for a **GET** endpoint is defined on the path *"/v1/__health"*. This structure, in the future, can be used to implement the other endpoints needed for the system to fully operate and fulfilling all functional requirements. To add more capabilities to the framework, Elysia allows the developer to use plugins which, generally, add more functionalities to the framework.

```
1  import { Elysia } from "elysia";
2  import { setup } from "./setup";
3
4  const app = new Elysia({ prefix: "/v1" })
5    .use(setup)
6    .get("/__health", () => {
7      return {"message": "Alive!"}
8    })
9    .listen(3000);
```

Listing 5.1: Creation of a HTTP server with Elysia - index.ts

To reduce the amount of code written in the server declaration file, the plugin *setup* was created. This will house all the shared configurations that will be needed throughout the service. Additionally, the *cors* plugin was also used and configured in the *setup* plugin, to allow for the customization of Cross-Origin Resource Sharing (CORS) within the service. Both of the plugins' definitions can be seen in listing 5.2.

```
1  import cors from "@elysiajs/cors";
2  import jwt from "@elysiajs/jwt";
3  import Elysia from "elysia";
4  import type { z } from "zod";
5  import type { selectUserSchema } from "@core/domain/user.schema";
6  import { env } from "./env";
7
8  export const setup = new Elysia({ name: "setup" })
9    .use(cors())
10   );
```

Listing 5.2: Setup Plugin declaration - setup.ts

5.3.2 T2 - Create entity model

To manage the connection between the service and the database, Drizzle-ORM was selected due to its modern approach to handling database operations with a focus on performance and adaptability. It provides a clean, type-safe interface that integrates continuously with TypeScript, ensuring that database queries are free from common runtime errors. One of the key advantages of Drizzle-ORM is its support for a wide range of database drivers, allowing the system to easily adapt to different databases, if there is ever a need to do so. Drizzle-ORM configures these drivers' language to be as close as the database itself, providing an interface for defining the domain that, while being code-first, gives developers the closest database-first approach, without operating the database directly. Drizzle-ORM also offers some of the best query processing available in the market, making it easy for the system to process high volumes of data.

To complement Drizzle-ORM, the same team also develops Drizzle-Kit, which provides powerful migration tools that simplify database management and ensure consistency across environments. It automatically generates migration files based on schema changes, making it easier for developers to maintain database structures and apply updates without manual intervention. This feature enhances the workflow by reducing the risk of errors during schema evolution, ensuring that changes can be tracked and applied systematically [19].

To implement Drizzle in the project, firstly it is necessary to create the schema of each entity that will be used in the system, as well as the connections between entities. Listing 5.3 shows one of the entities that was implemented, with the relations that were created. With Drizzle, *schemas* are defined using the same data type names that the database driver uses, reducing the room for error. To declare the relations, it is as simple as importing the schema that is needed to be referenced and selecting the foreign key's field. Lastly, there are two lines of code that are present in every schema, which create types for the database operations with the help of Zod. Zod is a schema validation tool with static type inference, which leverages TypeScript's type capabilities to validate objects at compile time, thus reducing possible runtime errors [20].

```
export const licenses = pgTable("licenses", {
  id: serial("id").primaryKey(),
  name: varchar("name", { length: 128 }).notNull(),
  internalName: varchar("internal_name", {})
    .generatedAlwaysAs((): SQL => sql`replace(upper(${licenses.name}), ' ', '_')`)
    .notNull(),
  keyValidity: integer("key_validity").notNull(),
  automationConfigured: boolean("automation_configured").default(false).notNull(),
  parentLicenseId: integer("parent_license_id").references((): AnyPgColumn => licenses.id),
  ...timestamps,
});

export const licensesRelations = relations(licenses, ({ many, one }) => ({
  documents: many(documents),
  managersToLicenses: many(managersToLicenses),
  licenseKeys: many(licenseKeys),
  parentLicense: one(licenses, {
    fields: [licenses.parentLicenseId],
    references: [licenses.id],
  }),
}));

export const createLicenseSchema = createInsertSchema(licenses);
export const selectLicenseSchema = createSelectSchema(licenses);
```

Listing 5.3: License schema - domain/license.schema.ts

In the case of many-to-many relationships, auxiliary tables need to be configured. This can be seen in listing 5.4, where the table *managers_to_licenses* was created. This table follows the same principles as the one mentioned previously, with the only exception being the creation of a composite primary key.

```
export const managersToLicenses = pgTable(
  "managers_to_licenses",
  {
    userId: integer("user_id")
      .notNull()
```

```
      .references(() => users.id)
      .notNull(),
    licenseId: integer("license_id")
      .notNull()
      .references(() => licenses.id)
      .notNull(),
  },
  (table) => {
    return {
      pk: primaryKey({ columns: [table.userId, table.licenseId] }),
    };
  },
);

export const managersToLicensesRelations = relations(managersToLicenses, ({ one }) => ({
  user: one(users, {
    fields: [managersToLicenses.userId],
    references: [users.id],
  }),
  license: one(licenses, {
    fields: [managersToLicenses.licenseId],
    references: [licenses.id],
  }),
}));

export const createManagersToLicenseSchema = createInsertSchema(managersToLicenses);
export const selectManagersToLicenseSchema = createSelectSchema(managersToLicenses);
```

Listing 5.4: License schema - domain/managersToLicenses.schema.ts

After these steps, to fully integrate the service with a database, it is necessary to configure Drizzle with a driver, which, in the case of this project, was *pg* in most cases, and *PGlite* the integration tests are being executed. The benefit of having a different driver for tests is that *PGlite* provides an in-memory *PostgreSQL* database, which significantly reduces the infrastructure that would be needed to setup a test container with a database to run the tests. To do so, in the first case, it is as simple as creating a *postgres* connection pool and passing it to the drizzle initiation function, along with all of the schema objects created previously. For the driver used for the tests, after creating the database client, it is necessary to execute the migrations, as this in-memory database won't persist any of its values. This code can be seen in listing 5.5.

```
const driverOptions = {
  logger: false,
  schema: {
    ...licenses,
    ...documents,
    ...events,
    ...licenseKeys,
    ...managersToLicenses,
    ...users,
    ...usersToLicenseKeys,
    ...licenseRequests,
  },
};

const getDriverFromEnv = async () => {
  if (env.NODE_ENV === "test") {
```

```
    const db = drizzlePgLite(new PGlite(), driverOptions);
    await migrate(db, { migrationsFolder: "./drizzle" });
    return db;
  }
  const pool = new Pool({
    host: env.DATABASE_HOST,
    port: env.DATABASE_PORT,
    user: env.DATABASE_USER,
    password: env.DATABASE_PASSWORD,
    database: env.DATABASE_NAME,
  });
  return drizzlePg(pool, driverOptions);
};

export const db = await getDriverFromEnv();

//postgres://root:license-manager@localhost:5432/license-manager-core
//postgres://user:password@host:port/db
```

Listing 5.5: Database connection configuration - config/database.ts

5.3.3 T3 - Implement strategy design pattern for license provider clients

To create the strategy design pattern, instead of using TypeScript classes, it was decided that it was best to leverage the object interface concept present in the language. To do so, firstly it is needed to create the constants present in listing 5.6, which define every operation these clients will perform.

```
export const CREATE_KEY_OPERATION = "CREATE_KEY_OPERATION";
export const DELETE_KEY_OPERATION = "DELETE_KEY_OPERATION";
```

Listing 5.6: Operation constants - constants/licenseStrategy.ts

Then, the type interfaces need to be created, which happens in listings 5.7 and 5.8. The first one defines the body and return types of functions that will be implemented for that license provider client. Then, the second listing defines the types needed to create the strategy design pattern. Here, a lookup map type is created, with its keys being every *LICENSE_NAME* defined and its values the clients to return, to fetch the correct strategy dynamically. Then, the same logic is applied to each client, but a lookup map type is created for the operations that each of these clients implement.

```
export type IntelliJCreateKeyBody = {};
export type IntelliJDeleteKeyBody = {};
export type IntelliJCreateKeyResponse = null | undefined;
export type IntelliJDeleteKeyResponse = null | undefined;
```

Listing 5.7: Placeholder type definitions for the IntelliJ client - type/license-ProviderClients/intellijClient.type.ts

```
type LicenseProviders = typeof INTELLIJ_INTERNAL_NAME | typeof ZOOM_INTERNAL_NAME;

type CreateKeyBodyTypes = ZoomCreateKeyBody | IntelliJCreateKeyBody;
type CreateKeyResponseTypes = ZoomCreateKeyResponse | IntelliJCreateKeyResponse;

type DeleteKeyBodyTypes = ZoomDeleteKeyBody | IntelliJDeleteKeyBody;
```

```
type DeleteKeyResponseTypes = ZoomDeleteKeyResponse | IntelliJDeleteKeyResponse;

export type LicenseStrategyProviders = {
  [key in LicenseProviders]: LicenseProvider;
};

export type LicenseProvider = {
  [CREATE_KEY_OPERATION]: (
    payload: CreateKeyBodyTypes,
  ) => Promise<CreateKeyResponseTypes>;
  [DELETE_KEY_OPERATION]: (
    payload: DeleteKeyBodyTypes,
  ) => Promise<DeleteKeyResponseTypes>;
};
```

Listing 5.8: License client strategy type interfaces - type/licenseClientStrategy.ts

For the implementation of the client, the functions are created and the previously defined types are assigned to the parameters and return types. Then, it is necessary to export a lookup map object with the functions defined for the strategy to be correctly applied, as shown in the placeholder implementation of one of the license provider clients, in listing 5.9.

```
export const INTELLIJ_INTERNAL_NAME = "INTELLIJ";

const createKeyOnProvider = async (
  body: IntelliJCreateKeyBody,
): Promise<IntelliJCreateKeyResponse> => {
  return null;
};

const deleteKeyOnProvider = async (
  key: IntelliJDeleteKeyBody,
): Promise<IntelliJDeleteKeyResponse> => {
  return null;
};

export const intellijClient = {
  [CREATE_KEY_OPERATION]: createKeyOnProvider,
  [DELETE_KEY_OPERATION]: deleteKeyOnProvider,
};
```

Listing 5.9: Placeholder for a client implementation - service/licenseClients/intelliJClient.ts

Lastly, the strategy lookup map object is created, where each license internal name is associated to its corresponding client, shown in listing 5.10.

```
export const licenseClientStrategy: LicenseStrategyProviders = {
    [INTELLIJ_LICENSE_NAME]: intelliJClient,
    [ZOOM_LICENSE_NAME]: zoomClient
}
```

Listing 5.10: License client strategy object - service/licenseClients/index.ts

5.3.4 T4 - Implement license key encryption

To encrypt a string, firstly it is needed to choose an encryption algorithm. For this use case, AES-256-CTR was selected, which is a symmetric encryption method using a 256-bit key in counter mode. It encrypts data by XORing a string with a generated key stream, offering fast and secure encryption without padding.

To implement this encryption, and, consequently, decryption, the code in listing 5.11 was written. The encryption function starts by generating a random array of 16 bytes, which will be used alongside a 256-bit secret value to create a cipher. This cipher is then used to update the plain string passed to the function to create the encrypted value, which is returned with the initial random array of 16 bytes. For decryption, both the encrypted value and the 16-byte array are needed. Then, using both values, a decryption cipher is created and utilized to decrypt the encrypted string.

```
1  export const encryptString = (plainString: string) => {
2    const iv = randomBytes(16);
3    const cipher = createCipheriv("aes-256-ctr", env.ENCRYPT_SECRET, iv);
4    const encrypted = Buffer.concat([cipher.update(plainString), cipher.final()]);
5
6    return {
7      iv: iv.toString("hex") as string,
8      encryptedString: encrypted.toString("hex"),
9    };
10 };
11
12 export const decryptString = (encryptedString: string, iv: string) => {
13   console.log("encryptString", encryptString);
14   const decipher = createDecipheriv(
15     "aes-256-ctr",
16     env.ENCRYPT_SECRET,
17     Buffer.from(iv, "hex"),
18   );
19   const decrypted = Buffer.concat([
20     decipher.update(Buffer.from(encryptedString, "hex")),
21     decipher.final(),
22   ]);
23
24   return decrypted.toString();
25 };
```

Listing 5.11: Encryption and decryption functions - utils/security.utils.ts

5.3.5 T5 - Configure frontend application

To build the *Frontend Web* component, React was chosen as the framework due to its component-based architecture. It allows the creation of reusable, modular components that can be easily maintained and extended, such as forms, user dashboards, or any interface that the developer might want to create. The use of React hooks and context API makes it easier to manage the state of the application, particularly when dealing with user interactions and authentication. React's virtual DOM optimizes rendering, ensuring that the interface remains performant even as the complexity of the system grows, providing a seamless user experience for managing and interacting with the system [21].

To build the code written in React, Vite was chosen as the build tool due to its speed and efficiency in setting up development environments. Unlike traditional bundlers, Vite leverages native ES modules in modern browsers and performs Hot Module Replacement (HMR), which significantly reduces the time it takes to start a project locally and implement changes. This is particularly useful in a user-facing system where the user interface may require frequent updates [22].

To reduce the complexity of writing styles for the component, Tailwind CSS was chosen because it provides a utility-first approach to styling, which enhances both development speed and maintainability. Instead of writing custom CSS, Tailwind allows developers to apply pre-defined utility classes directly within the JSX, improving the process of building responsive and visually consistent UI components. This integration is especially valuable in a React app where components are modular. With Tailwind, developers can quickly style each component without worrying about conflicts or the need to manage separate CSS files. Tailwind's flexibility in handling responsive design and theming makes it easier to create a cohesive and adaptive user interface, which is crucial for a system where clear and intuitive visual feedback is important for users navigating features [23].

Despite having a good development foundation with the previously listed tools, there are some additional features that are needed that these libraries don't offer. In this case, it is a router for the pages that will be implemented. For this purpose, TanStack Router was chosen for the service. This library is a lightweight and powerful routing solution for React applications that offers fine-grained control over route matching, data fetching, and rendering. It stands out for its declarative approach to defining routes, which integrates perfectly with React's component-based architecture and with Typescript, as it creates type-safe routing mechanisms that reduce runtime errors. This makes it ideal for managing complex navigation scenarios in dynamic applications where users might need to switch between different views. It also handles nested and asynchronous routes, allowing developers to load data before rendering components or handle side effects as part of the routing process [24].

To configure the router, the code in listing 5.12 was implemented. Here, a router is defined and linked to the *routeTree.gen.ts* file, which is automatically generated by TanStack depending on the files that exist in the *routes* folder. With this, developers don't need to keep changing the routing configuration, it is only required to create files and directories equivalent to the route they wish to create. For example, the file *routes/licenses/$licenseId.tsx* automatically routes the component in this file to the path */licenses:licenseId*, where *licenseId* is any value that can be fetched in the page for dynamically loading content depending on the URL specified.

```
1  const router = createRouter({ routeTree, context: { auth: undefined } });
2
3  declare module "@tanstack/react-router" {
4    interface Register {
5      router: typeof router;
6    }
7  }
8
9  const TanStackRouterDevtools =
10   process.env.NODE_ENV !== "dev"
11     ? () => null // Render nothing in production
12     : React.lazy(() =>
13         // Lazy load in development
```

```
        import("@tanstack/router-devtools").then((res) => ({
          default: res.TanStackRouterDevtools,
        })),
      );

// biome-ignore lint/style/noNonNullAssertion: <explanation>
ReactDOM.createRoot(document.getElementById("root")!).render(
  <React.StrictMode>
    <AuthProvider>
      <InnerApp />
    </AuthProvider>
  </React.StrictMode>,
);

function InnerApp() {
  const auth = useAuth();
  return (
    <>
      <RouterProvider router={router} context={{ auth }} />
      <TanStackRouterDevtools router={router} />
    </>
  );
}
```

Listing 5.12: React and TanStack Router definition - main.tsx

Lastly, the final library added to the service was Shadcn/UI, a component library that leverages the power of React components and the utility-first approach of Tailwind CSS to offer an effortless way to build user interfaces. By combining pre-built, customizable React components with Tailwind's extensive utility classes, Shadcn/UI allows developers to create responsive, visually consistent, and accessible User Interface (UI) elements with minimal effort. Each component in the library is designed with flexibility in mind, allowing developers to easily change the design and behavior to fit the needs of the application without writing extensive custom CSS [25].

Using all of the libraries described previously, a root route was defined to be the skeleton of the entire frontend application, as seen in listing 5.13. It displays a lateral *Sidebar* and a *Navbar* on top, with the rest of the screen being reserved for the pages that will be implemented in future user stories.

```
export const Route = createRootRouteWithContext<RouterContext>()({
  component: () => (
    <>
      <div className="grid min-h-screen w-full md:grid-cols-[220px_1fr] lg:grid-cols-[280px_1fr]">
        <Sidebar routes={routes} />
        <div className="flex flex-col">
          <Navbar routes={routes} />
          <main className="flex flex-1 flex-col gap-4 p-4 lg:gap-6 lg:p-6">
            <Outlet />
          </main>
        </div>
      </div>
    </>
  ),
});
```

Listing 5.13: Base route of the frontend application - routes/root.tsx

5.3.6 T6 - Configure communication between frontend and back-end

To implement this task, a tool from Elysia was used, which was Eden. Eden provides end-to-end type safety between the frontend and the backend, all by using the types generated by Elysia's router. To configure this, all that is needed to be done is call the *treaty* function, specify the router's type, which is the *App* created in listing 5.1, and specify the backend's URL. This code is present in listing 5.14.

```
import type { App } from "@core/index";

export const api = treaty<App>(env.VITE_BACKEND_URL);
```

Listing 5.14: Elysia's eden treaty definition - services/config.ts

5.3.7 T7 - Add authentication to the backend

To allow for a user to perform authenticated requests, a *derive* was added to the *setup* middleware that is present in every controller. This decorator, in listing 5.15, replacing listing 5.2, fetches the bearer token from the Authorization header and decodes it into a user schema object, which can then be used to verify all sorts of information, such as the user's type, which will be done during the development of the functional requirements.

```
export const setup = new Elysia({ name: "setup" })
  .use(cors())
  .use(
    jwt({
      name: "jwt",
      secret: env.JWT_SECRET,
    }),
  )
  .derive({ as: "scoped" }, async ({ headers, jwt }) => {
    const token = headers.authorization?.startsWith("Bearer ")
      ? headers.authorization.slice(7)
      : null;
    if (!token) {
      return;
    }
    const user = (await jwt.verify(token)) as unknown as z.infer<typeof selectUserSchema>;
    if (!user) {
      return;
    }
    return {
      user,
    };
  });
```

Listing 5.15: *user* Elysia derive - setup.ts

5.3.8 FR1 - Manage licenses

This functional requirement's goal is to allow a license manager to add licenses to the system, as well as products directly related to existing licenses.

This requirement starts in the *Core Service* component, with the implementation of listing 5.16, which begins with calling the function in listing 5.17 if the user tries to add a product

instead of a license. In this case, it cannot save if a parent license doesn't exist in the system and, therefore, throws a custom error if the parent license isn't found. After this is checked, the first function saves the new license or product on the database. Since the *insert* function returns an array, the generic function in listing 5.18 was created that returns the first entry of this array, if there is one. If there isn't a value on this array, it throws an error.

```
export const createLicenseAndReturn = async (license: z.infer<typeof createLicenseSchema>) => {
  if (license.parentLicenseId) {
    await getLicenseOrThrow(license.parentLicenseId);
  }
  return returnFirst(await db.insert(licenses).values(license).returning());
}
```

Listing 5.16: createLicenseAndReturn function definition - database/license.db.ts

```
export const getLicenseOrThrow = async (id: number) => {
  const license = await db.query.licenses.findFirst({
    where: (licenses, { eq, isNull }) =>
      and(eq(licenses.id, id), isNull(licenses.deletedAt)),
    with: {
      managersToLicenses: {
        with: {
          user: true,
        },
      },
    },
  });
  if (!license) {
    throw new EntryNotFoundError("License wasn't found");
  }
  return license;
};
```

Listing 5.17: getLicenseOrThrow function definition - database/license.db.ts

```
export function returnFirst<T>(array: Array<T>): T {
  if (array.length === 0) {
    throw new EntryNotFoundError();
  }
  return array[0];
}
```

Listing 5.18: returnFirst function definition - utils/db.utils.ts

This function is then called by the service function in listing 5.20, which simply returns the value returned by the database function.

```
export const createLicense = async (
  license: z.infer<typeof createLicenseSchema>,
) => {
  return await createLicenseAndReturn(license);
};
```

Listing 5.19: createLicense function definition - service/license.service.ts

Lastly, the route in listing 5.20 was defined on the *LicenseController*, which specifies the valid input, in this case, the date needed to create the license. Then, a handler function is assigned to the route, which simply returns the value obtained from the service function. The function is then available through the router on the path */v1/license*, using a POST HTTP method.

```
export const licenseRoutes = new Elysia({ prefix: "/license" })
  .use(setup)
  .post("", async ({ body }) => await createLicense(body), {
    body: t.Object({
      name: t.String(),
      keyValidity: t.Integer(),
      parentLicenseId: t.Optional(t.Union([t.Number(), t.Undefined(), t.Null()])),
    }),
  })
```

Listing 5.20: POST /v1/license route definition - controller/license.controller.ts

For the client-side implementation, in the *Frontend Web* component, the route in listing 5.21, explained in subsection 5.3.11, and the component in listing 5.22 were created. Here, *CreateLicenseDialog* is invoked, with a button to create a license. The remainder of the page will be fully explained in subsection 5.3.11, as it is related to the mentioned functional requirement.

```
export const Route = createFileRoute("/licenses/")({
  validateSearch: z.object({
    page: z.number().default(1).catch(1),
  }),
  component: Licenses,
  loaderDeps: ({ search: { page } }) => ({ page }),
  loader: async ({ deps: { page }, context }) => {
    if (context.auth.isAuthenticated) {
      return (await loadActiveLicenses(context.auth.jwt, page)).data;
    }
  },
});
```

Listing 5.21: /licenses/ route definition - routes/licenses/index.tsx

```
function Licenses() {
  const data = Route.useLoaderData();
  const search = Route.useSearch();
  const pageSize = 10;

  return (
    <>
      <div className="flex items-center">
        <h1 className="text-lg font-semibold md:text-2xl">Licenses</h1>
        <div className="ml-auto">
          {data && data.entries.length !== 0 && (
            <CreateLicenseDialog>
              <Button variant="outline">Create License</Button>
            </CreateLicenseDialog>
          )}
        </div>
      </div>
```

```
18        <div
19          className="flex flex-1 justify-center rounded-lg border border-dashed shadow-sm"
20          x-chunk="dashboard-02-chunk-1"
21        >
22          {data && data.entries.length !== 0 ? (
23            <LicenseTable licenses={data} page={search.page} size={pageSize} />
24          ) : (
25            <EmptyLicensesPage />
26          )}
27        </div>
28      </>
29    );
30  }
```

Listing 5.22: Licenses page definition - routes/licenses/index.tsx

The implementation of the *CreateLicenseDialog* component, in listing 5.23, is composed of two main components, *Dialog* and *Toaster*. When the *children* is triggered, in this case, the *Create License* button in listing 5.22, a modal pops up with a form for the user to input all the data that is needed to create a license or a product. If the user is creating a product, the parent license identification is passed to the component and used in the form, without the need for the user to input it. Once the user writes all the information and the submit button is pressed, a function call to the service function in listing 5.24 happens, which sends an HTTP request to the backend to create the license. If the license is created with success, then the user is forwarded to a page with all the details about the license that was just created. However, if the request ends with an error, a toast is shown in the bottom right of the user's screen, informing that the operation ended with an error.

```
1   const formSchema = z.object({
2     name: z.string(),
3     keyValidity: z.number().positive(),
4     parentLicenseId: z.number().optional(),
5   });
6
7   export function CreateLicenseDialog({
8     parentLicenseId,
9     children,
10  }: { parentLicenseId?: number; children: React.ReactNode }) {
11    const form = useForm<z.infer<typeof formSchema>>({
12      resolver: zodResolver(formSchema),
13      defaultValues: {
14        name: "",
15        keyValidity: 1,
16        ...(parentLicenseId && { parentLicenseId }),
17      },
18    });
19
20    const navigate = useNavigate({ from: "/licenses/$licenseId" });
21    const auth = useAuth();
22    const { toast } = useToast();
23    const [open, setOpen] = useState<boolean>(false);
24
25    async function onSubmit(values: z.infer<typeof formSchema>) {
26      console.log(values);
27      const res = await createLicense(auth.jwt, values);
28      if (res.error) {
29        toast({
```

```
30          title: "Something went wrong, please try again.",
31          description: JSON.parse(res.error.value as string).message,
32          variant: "destructive",
33        });
34        return;
35      }
36
37      navigate({
38        to: "/licenses/$licenseId",
39        params: { licenseId: String(res.data.id) },
40        search: { page: 1, licenseKeysPage: 1 },
41      });
42    }
43
44    return (
45      <>
46      <Toaster />
47      <Dialog open={open} onOpenChange={setOpen}>
48        <DialogTrigger asChild>{children}</DialogTrigger>
49        <DialogContent className="sm:max-w-[425px]">
50          <DialogHeader>
51            <DialogTitle>
52              {parentLicenseId ? "Create Product" : "Create license"}
53            </DialogTitle>
54          </DialogHeader>
55          <Form {...form}>
56            <form onSubmit={form.handleSubmit(onSubmit)}>
57              <FormField
58                control={form.control}
59                name="name"
60                render={({ field }) => (
61                  <FormItem className="mb-4">
62                    <FormLabel>Name</FormLabel>
63                    <FormControl>
64                      <Input placeholder="JetBrains" {...field} />
65                    </FormControl>
66                  </FormItem>
67                )}
68              />
69              <FormField
70                control={form.control}
71                name="keyValidity"
72                render={({ field }) => (
73                  <FormItem>
74                    <FormLabel className="">Key Validity (months)</FormLabel>
75                    <FormControl>
76                      <Input type="number" placeholder="1" {...field} />
77                    </FormControl>
78                  </FormItem>
79                )}
80              />
81              <Button className="mt-6" type="submit">
82                Submit
83              </Button>
84            </form>
85          </Form>
86        </DialogContent>
87      </Dialog>
88      </>
```

```
  );
}
```

Listing 5.23: CreateLicenseDialog component definition - components/create-license-dialog.tsx

```
export async function createLicense(
  token: string,
  license: z.infer<typeof createLicenseSchema>,
) {
  return await api.v1.license.post(license, {
    headers: { authorization: `Bearer ${token}` },
  });
}
```

Listing 5.24: createLicense function definition - services/licenses.ts

To add a product to a license, the user needs to access the *SpecificLicensePage*, in listing 5.25, provided by the route in listing 5.26. This page displays all information related to a license, a button to create a product, that belongs to the same component in listing 5.23, with the only difference being that the parent license identification is sent to the component, which will be sent with the rest of the information, and a *Tabs* component with three tabs that will be detailed in future subsections, including the cases in which they are enabled to the user.

```
function SpecificLicensePage() {
  const data = Route.useLoaderData();
  const search = Route.useSearch();

  return (
    <>
      {data?.license ? (
        <>
          <GoBackButton parentLicenseId={data.license.parentLicenseId ?? undefined} />
          <div className="grid gap-8 max-w-6xl mx-auto py-8 px-4 md:px-6">
            <div className="grid gap-4">
              <div className="flex items-center">
                <span className="text-3xl font-bold">{data.license.name}</span>
                <div className="ml-auto flex space-x-1">
                  {!data.license.parentLicenseId && (
                    <CreateLicenseDialog parentLicenseId={data.license.id}>
                      <Button variant="outline">Create Product</Button>
                    </CreateLicenseDialog>
                  )}

                  {(data.license.parentLicenseId || data.products) && (
                    <RequestLicenseButton />
                  )}
                </div>
              </div>
              <div className="grid gap-4">
                <p className="mb-2 text-muted-foreground">
                  Lorem ipsum dolor sit amet, consectetur adipiscing elit. Donec vel augue
                  vitae dui viverra gravida. Sed interdum tincidunt aliquam. Mauris in
                  congue metus, et ultrices magna. Phasellus fringilla metus arcu, id
                  bibendum enim volutpat at. Vivamus aliquet sagittis semper. Mauris eu
                  ornare turpis, vel ornare urna. Vestibulum vel tortor eget orci congue
```

```
33                  luctus. Mauris egestas eget sem ac ultricies. Nullam eleifend eget leo
34                  quis dictum. Aenean est magna, convallis id semper eget, dapibus et
35                  sapien. Vestibulum bibendum justo a lacinia consectetur. Mauris sit amet
36                  risus eget libero pharetra sollicitudin eget sed nisl. Suspendisse
37                  tincidunt magna luctus tempus blandit. Etiam mollis ex ut malesuada
38                  aliquam.
39                </p>
40                <p>
41                  <span className="font-bold">Key validity: </span>
42                  {data.license.keyValidity} month
43                  {data.license.keyValidity > 1 ? "s" : ""}
44                </p>
45                <div className="flex items-center gap-2">
46                  {data.license.automationConfigured ? (
47                    <>
48                      <BadgeCheck />
49                      <span>Automation configured</span>
50                    </>
51                  ) : (
52                    <>
53                      <BadgeX />
54                      Automation not configured
55                    </>
56                  )}
57                </div>
58              </div>
59            </div>
60            <Tabs
61              defaultValue={
62                data.license.parentLicenseId !== null ? "licenseKeys" : "products"
63              }
64            >
65              <TabsList className="grid grid-cols-3 divide-x">
66                <TabsTrigger
67                  disabled={data.license.parentLicenseId !== null}
68                  value="products"
69                >
70                  Products
71                </TabsTrigger>
72                <TabsTrigger value="licenseKeys">License Keys</TabsTrigger>
73                <TabsTrigger
74                  disabled={data.license.parentLicenseId !== null}
75                  value="managers"
76                >
77                  Managers
78                </TabsTrigger>
79              </TabsList>
80              <TabsContent value="products">
81                <div className="border shadow-sm rounded-lg overflow-hidden">
82                  <LicenseTable licenses={data.products} page={search.page} size={10} />
83                </div>
84              </TabsContent>
85              <TabsContent value="licenseKeys">
86                {data.licenseKeys && (
87                  <LicenseKeyTable
88                    licenseKeys={data.licenseKeys}
89                    page={search.licenseKeysPage}
90                    size={10}
91                  />
```

```
                )}
              </TabsContent>
              <TabsContent value="managers">
                {data.managers && !data.license.parentLicenseId && (
                  <ManagersCard
                    managers={data.managers}
                    licenseManagers={data.license.managersToLicenses}
                    licenseId={data.license.id}
                  />
                )}
              </TabsContent>
            </Tabs>
          </div>
        </>
      ) : (
        <NotFoundComponent />
      )}
    </>
  );
}
```

Listing 5.25: SpecificLicensePage page definition - routes/licenses/$licenseId.tsx

```
export const Route = createFileRoute("/licenses/$licenseId")({
  validateSearch: licenseSearchSchema,
  component: SpecificLicensePage,
  loaderDeps: ({ search: { page } }) => ({ page }),
  loader: async ({ params, deps, context }) => {
    if (context.auth.isAuthenticated) {
      if (!Number.isNaN(params.licenseId)) {
        const license = (await getLicenseById(context.auth.jwt, Number(params.licenseId)))
          .data;
        const products = await getProductsFromLicense(
          context.auth.jwt,
          Number(params.licenseId),
          deps.page,
        );
        return {
          license,
          products: products,
        };
      }
      return null;
    }
  },
});
```

Listing 5.26: /licenses/$licenseId route definition - routes/licenses/$licenseId.tsx

5.3.9 FR2 - Manage license managers

The objective of this functional requirement is to add a view that allows a system administrator to see who are the current managers of a license, as well as a way to add to these managers, or remove any of them.

Starting with the *Core Service* implementation, the database functions were introduced, in listing 5.27. The first function starts with verifying if a license exists within the system. If it does, then it verifies, with the function in listing 5.28, if the user is a manager. If both of these conditions are fulfilled, then the data is saved onto the database, and the entry is returned.

```
export const addManagerToLicenseAndReturn = async (
  dto: z.infer<typeof createManagersToLicenseSchema>,
) => {
  const license = await getLicenseOrThrow(dto.licenseId);
  if (license.parentLicenseId !== null) {
    throw new NotAParentLicenseError();
  }

  if (!(await isManager(dto.userId))) {
    throw new NotAManagerError();
  }

  return returnFirst(await db.insert(managersToLicenses).values(dto).returning());
};

export const removeManagerFromLicenseAndReturn = async (
  userId: number,
  licenseId: number,
) => {
  return returnFirst(
    await db
      .delete(managersToLicenses)
      .where(
        and(
          eq(managersToLicenses.userId, userId),
          eq(managersToLicenses.licenseId, licenseId),
        ),
      )
      .returning(),
  );
};
```

Listing 5.27: addManagerToLicenseAndReturn and removeManagerFromLicenseAndReturn functions definition - database/managersToLicenses.db.ts

```
export const isManager = async (id: number): Promise<boolean> => {
  const res = await db.query.users.findFirst({
    where: (users, { eq }) => eq(users.id, id) && eq(users.type, UserTypes.MANAGER),
  });
  return !!res;
};
```

Listing 5.28: isManager function definition - database/user.db.ts

Listing 5.29 shows the implementation of the service methods, which simply return the values returned by the database functions.

```
export const addManagerToLicense = async (
  dto: z.infer<typeof createManagersToLicenseSchema>,
) => {
  return await addManagerToLicenseAndReturn(dto);
```

```
};

export const removeManagerFromLicense = async (userId: number, licenseId: number) => {
  return await removeManagerFromLicenseAndReturn(userId, licenseId);
};
```

Listing 5.29: addManagerToLicense and removeManagerFromLicense functions definition - service/managersToLicenses.service.ts

The routes in listing 5.30 were added to the *LicenseController*, as it directly relates to a license. These specify the valid inputs and a handler function is assigned to the routes, which simply returns the value obtained from the service function. The function is then available through the router on the path */v1/license/:licenseId/manager/:userId*, using a POST or a DELETE HTTP method, respectively.

```
.post(
  "/:licenseId/manager/:userId",
  async ({ params }) => await addManagerToLicense(params),
  {
    params: t.Object({
      licenseId: t.Numeric(),
      userId: t.Numeric(),
    }),
  },
)
.delete(
  "/:licenseId/manager/:userId",
  async ({ params }) => await removeManagerFromLicense(params.userId, params.licenseId),
  {
    params: t.Object({
      licenseId: t.Numeric(),
      userId: t.Numeric(),
    }),
  },
)
```

Listing 5.30: POST and DELETE /v1/license/:licenseId/manager/:userId route definition - controller/license.controller.ts

The client-side implementation involved the creation of the *ManagersCard* component in listing 5.31, which was first seen in a *Tab* in listing 5.25. This tab is only available to be accessed when the user is browsing through a parent license. The component is composed of one main component *Card*, which contains two sections. The first one displays all the users that are assigned as managers to this license and, for each one, a component to remove them from the license. The component in listing 5.32 provides a button that, when clicked, opens a modal for the user to confirm his action. Once the action is confirmed, a function call, seen in listing 5.33 is made that sends a request to the backend to complete the action. If the request is successful, the page is refreshed. If not, a toast is shown on the user's screen saying that an error occurred.

```
function ManagersCard({
  licenseManagers,
  managers,
  licenseId,
}: {
  licenseManagers: Array<{ user: z.infer<typeof selectUserSchema> }>;
```

```
7     managers: PaginatedQuery<Array<z.infer<typeof selectUserSchema>>>;
8     licenseId: number;
9   }) {
10    const [selectedManager, setSelectedManager] = useState<number>(0);
11
12    return (
13      <Card>
14        <CardHeader>
15          <CardTitle>Current license managers</CardTitle>
16        </CardHeader>
17        <CardContent>
18          <div className="grid grid-cols-2 space-x-5">
19            <div className="border-2 rounded-md p-4">
20              {licenseManagers.map((manager) => (
21                <div className="grid  grid-cols-2 divide-x my-2" key={manager.user.id}>
22                  <div className=" content-center">
23                    <ShowUsers users={[manager]} />
24                  </div>
25                  <div className="content-center ml-auto">
26                    <RemoveManagerButton userId={manager.user.id} licenseId={licenseId} />
27                  </div>
28                  <Separator className="mt-2 col-span-2" />
29                </div>
30              ))}
31            </div>
32            <div className="border-2 rounded-md p-4">
33              <Select onValueChange={(e) => setSelectedManager(Number(e))}>
34                <SelectTrigger className="col-span-3 text-left" id="user">
35                  <SelectValue placeholder="Select a user" />
36                </SelectTrigger>
37                <SelectContent>
38                  <SelectGroup>
39                    {managers.entries
40                      .filter((user) =>
41                        licenseManagers.every(
42                          (managerToLicense) => managerToLicense.user.id !== user.id,
43                        ),
44                      )
45                      .map((user) => (
46                        <SelectItem key={"user.id"} value={String(user.id)}>
47                          {user.name} | {user.type}
48                        </SelectItem>
49                      ))}
50                  </SelectGroup>
51                </SelectContent>
52              </Select>
53              <div className="mt-4">
54                <AddManagerButton userId={selectedManager} licenseId={licenseId} />
55              </div>
56            </div>
57          </div>
58        </CardContent>
59      </Card>
60    );
61  }
```

Listing 5.31: ManagersCard component definition - routes/license/$licenseId.tsx

Listing 5.32: RemoveManagersButton component definition - routes/license/$licenseId.tsx

```tsx
export async function removeLicenseManager(
  token: string,
  userId: number,
  licenseId: number,
) {
  return await api.v1
    .license({ licenseId })
    .manager({ userId })
    .delete(null, { headers: { authorization: `Bearer ${token}` } });
}
```

Listing 5.33: removeLicenseManager function definition - services/licenses.ts

The second section shows a *Select* component with all managers that aren't assigned to the license, as well as the component in listing 5.34. This component provides a button that, when clicked, shows a modal for the user to confirm the action. When the action is confirmed, the function in listing 5.35 is triggered, and the request is sent to the backend. If the request is successful, the page is refreshed to show the new data. If not, a toast is shown on the user's screen with information about the error.

```tsx
function AddManagerButton({ userId, licenseId }: { userId: number; licenseId: number }) {
  const { toast } = useToast();
  const auth = useAuth();
  const navigate = useNavigate({ from: "/licenses/$licenseId" });
  const search = Route.useSearch();
  const [open, setOpen] = useState<boolean>(false);

  return (
    <>
      <Toaster />
      <AlertDialog open={open} onOpenChange={setOpen}>
        <AlertDialogTrigger asChild>
          <Button disabled={userId === 0} className="ml-auto" variant="default">
            Add Manager
          </Button>
        </AlertDialogTrigger>
        <AlertDialogContent>
          <AlertDialogHeader>
            <AlertDialogTitle>Please confirm your action.</AlertDialogTitle>
            <AlertDialogDescription>
              Are you sure you want this user to be one of this license's managers?
            </AlertDialogDescription>
          </AlertDialogHeader>
          <AlertDialogFooter>
            <AlertDialogCancel>Cancel</AlertDialogCancel>
            <AlertDialogAction
              onClick={async () => {
                const res = await addNewLicenseManager(auth.jwt, userId, licenseId);
                if (res.error) {
                  toast({
                    title: "Something went wrong, please try again.",
                    description: JSON.parse(res.error.value as string).message,
                    variant: "destructive",
                  });
                  return;
```

```
                }
                setOpen(false);
                navigate({
                  to: "/licenses/$licenseId",
                  params: { licenseId: String(licenseId) },
                  search,
                  replace: true,
                });
              }}
            >
              Confirm
            </AlertDialogAction>
          </AlertDialogFooter>
        </AlertDialogContent>
      </AlertDialog>
    </>
  );
}
```

Listing 5.34: AddManagerButton component definition - routes/license/$licenseId.tsx

```
export async function addNewLicenseManager(
  token: string,
  userId: number,
  licenseId: number,
) {
  return await api.v1
    .license({ licenseId })
    .manager({ userId })
    .post(null, { headers: { authorization: `Bearer ${token}` } });
}
```

Listing 5.35: addNewLicenseManager function definition - services/licenses.ts

5.3.10 FR3 - Manage license keys

For this functional requirement to be considered complete, it is necessary to provide a license manager with a way to visualize a license's keys, as well as assign them to a user, or unassign them from the current user that is using it.

Starting the implementation with the *Core Service* component, in listing 5.36 there are the functions that communicate with the database. The first one, simply saves a new entry on the key-to-user relationship table, returning the saved value, while the second one deletes an entry with the specified license key identification and user identification. Another function was also made to retrieve the license keys associated with a specific license, which is in listing 5.37. This function starts by retrieving the keys, based on the page and page size passed as parameters, then queries the database to get the total amount of license keys stored in the database, returning the aggregated value of these two queries in the end.

```
export const assignKeyToUserAndReturn = async (licenseKeyId: number, userId: number) => {
  return returnFirst(
    await db
      .insert(usersToLicenseKeys)
      .values({ userId: userId, licenseKeyId: licenseKeyId })
```

```
      .returning(),
  );
};

export const unassignKeyFromUserAndReturn = async (
  licenseKeyId: number,
  userId: number,
) => {
  return returnFirst(
    await db
      .delete(usersToLicenseKeys)
      .where(
        and(
          eq(usersToLicenseKeys.licenseKeyId, licenseKeyId),
          eq(usersToLicenseKeys.userId, userId),
        ),
      )
      .returning(),
  );
};
```

Listing 5.36: assignKeyToUserAndReturn and unassignKeyFromUserAndReturn functions definition - database/usersToLicenseKeys.db.ts

```
export const getLicenseKeysFromLicensePaginated = async (
  licenseId: number,
  page: number,
  size: number,
) => {
  const keys = await db.query.licenseKeys.findMany({
    where: (licenseKeys, { eq }) => eq(licenseKeys.licenseId, licenseId),
    with: {
      usersToKeys: {
        with: {
          user: {},
        },
      },
    },
    limit: size,
    offset: (page - 1) * size,
  });
  const amount = await db
    .select({ count: count() })
    .from(licenseKeys)
    .where(eq(licenseKeys.licenseId, licenseId));
  return {
    entries: keys,
    totalAmount: amount[0].count,
    page: page,
    size: size,
  };
};
```

Listing 5.37: getLicenseKeysFromLicensePaginated function definition - database/licenseKey.db.ts

Listing 5.38 shows the service implementation where, on the first function, the license key and user are fetched and an error is thrown if they don't exist within the system, and

then the previously mentioned function is called. The second one, however, only calls the *unassignKeyFromUserAndReturn* function, as it will throw an error if data isn't deleted. Both of these functions return *void* as its information isn't needed in the controller. For the retrieval of keys, listing 5.39 was implemented, and starts by retrieving the license and throwing if it doesn't exist. Then, the function verifies if the user attempting to complete the request is a manager of the license that was fetched. After this is verified, it calls the database function to retrieve the keys. The next step of this function is to filter the keys that can be assigned to multiple users or that don't have users assigned to them if the *assignable* flag is active. Lastly, the key values are decrypted and returned out of the function.

```
export const assignKeyToUser = async (
  token: string,
  licenseKeyId: number,
  userId: number,
) => {
  return await api.v1
    .licenseKey({ licenseKeyId })
    .user({ userId })
    .post(null, { headers: { authorization: `Bearer ${token}` } });
};

export const unassignKeyFromUser = async (
  token: string,
  licenseKeyId: number,
  userId: number,
) => {
  return await api.v1
    .licenseKey({ licenseKeyId })
    .user({ userId })
    .delete(null, { headers: { authorization: `Bearer ${token}` } });
};
```

Listing 5.38: assignKeyToUser and unassignKeyFromUser functions definition - service/usersToLicenseKeys.service.ts

```
export const getAllLicenseKeysFromLicensePaged = async (
  userId: number,
  licenseId: number,
  assignable: boolean,
  page: number,
  size = 10,
) => {
  const licenseWithManagers = await getLicenseOrThrow(licenseId);

  if (
    !licenseWithManagers.managersToLicenses.filter((entry) => entry.userId === userId)
  ) {
    throw new AuthenticationError();
  }

  const keys = await getLicenseKeysFromLicensePaginated(licenseId, page, size);
  if (assignable) {
    keys.entries = keys.entries.filter(
      (key) => key.multipleAssignments || key.usersToKeys.length === 0,
    );
  }

```

```
23    for (const licenseKey of keys.entries) {
24      licenseKey.key = decryptString(licenseKey.key, licenseKey.iv);
25    }
26    return keys;
27  };
```

Listing 5.39: getAllLicenseKeysFromLicensePaged function definition - service/licenseKey.service.ts

In listing 5.40 the routes are defined on path */v1/licenseKey/:licenseKeyId/user/:userId*, with the POST method calling the assign function and the DELETE method calling the unassign function. Lastly, both of these are provided with type validation for the specified parameters. Listing 5.41 defines a route for path */v1/license/:licenseId/key/paged* on the method GET. This route calls the previously defined function after verifying if the user that sent the request has the *MANAGER* type.

```
1   export const assignKeyToUserAndReturn = async (licenseKeyId: number, userId: number) => {
2     return returnFirst(
3       await db
4         .insert(usersToLicenseKeys)
5         .values({ userId: userId, licenseKeyId: licenseKeyId })
6         .returning(),
7     );
8   };
9
10  export const unassignKeyFromUserAndReturn = async (
11    licenseKeyId: number,
12    userId: number,
13  ) => {
14    return returnFirst(
15      await db
16        .delete(usersToLicenseKeys)
17        .where(
18          and(
19            eq(usersToLicenseKeys.licenseKeyId, licenseKeyId),
20            eq(usersToLicenseKeys.userId, userId),
21          ),
22        )
23        .returning(),
24    );
25  };
```

Listing 5.40: POST & DELETE /v1/licenseKey/:licenseKeyId/user/:userId route definition - controller/licenseKey.controller.ts

```
1   .get(
2     "/:licenseId/key/paged",
3     async ({ params, query, user }) => {
4       if (user && hasRole(user, UserTypes.MANAGER)) {
5         return await getAllLicenseKeysFromLicensePaged(
6           user.id,
7           params.licenseId,
8           query.assignable,
9           query.page,
10          query.size,
11        );
12      }
```

```
13
14      throw new AuthenticationError();
15    },
16    {
17      params: t.Object({
18        licenseId: t.Numeric(),
19      }),
20      query: t.Object({
21        assignable: t.Boolean(),
22        page: t.Number(),
23        size: t.Optional(t.Number()),
24      }),
25    },
26  )
```

Listing 5.41: GET /v1/license/:licenseId/key/paged route definition - controller/license.controller.ts

Moving to the implementation on the *Frontend Web* component, these requests can be called from the component in listing 5.42, which resides on the page mentioned in listing 5.25. This component lists all of the keys and their information, divided by pages where the current page is defined as a query parameter on the route (defined in listing **??**). Two buttons are defined for each license key: one that allows a manager to assign an available key to a user, which is only available when the license key doesn't have a user assigned to it, or it can have multiple users assigned to itself; and one that allows a user to remove a license key from a user, which is only available when there is at least one user assigned to it.

```
1   function LicenseKeyTable({
2     licenseKeys,
3     page,
4     size,
5   }: {
6     licenseKeys: PaginatedQuery<Array<licenseKeysWithUsers>>;
7     page: number;
8     size: number;
9   }) {
10    return (
11      <Table>
12        <TableHeader>
13          <TableRow>
14            <TableHead className="w-[10%]">Id</TableHead>
15            <TableHead className="w-[25%]">Key</TableHead>
16            <TableHead className="w-[25%] text-center">Assigned To</TableHead>
17            <TableHead className="w-[30%] text-center">Multiple Assignments</TableHead>
18            <TableHead className="w-[10%]" />
19          </TableRow>
20        </TableHeader>
21        <TableBody>
22          {!!licenseKeys &&
23            licenseKeys.entries.map((licenseKey) => (
24              <TableRow key={licenseKey.id}>
25                <TableCell className="font-medium">{licenseKey.id}</TableCell>
26                <TableCell>{licenseKey.key}</TableCell>
27                <TableCell className="flex justify-center">
28                  {licenseKey.usersToKeys.length === 0 ? (
29                    "---"
```

```
30                ) : (
31                  <ShowUsers users={licenseKey.usersToKeys} />
32                )}
33              </TableCell>
34              <TableCell>
35                {licenseKey.multipleAssignments ? (
36                  <Check className="m-auto" />
37                ) : (
38                  <X className="m-auto" />
39                )}
40              </TableCell>
41              <TableCell className="grid grid-flow-col divide-y">
42                {(licenseKey.usersToKeys.length === 0 ||
43                  licenseKey.multipleAssignments) && (
44                  <AssignLicenseKeyModal licenseKey={licenseKey} />
45                )}
46                {licenseKey.usersToKeys.length !== 0 && (
47                  <RemoveLicenseKeyModal licenseKey={licenseKey} />
48                )}
49              </TableCell>
50            </TableRow>
51          ))}
52        </TableBody>
53        <TableFooter>
54          <TableRow>
55            {!!licenseKeys && (
56              <>
57                <TableCell colSpan={4} className="text-right">
58                  {licenseKeys.totalAmount === 0 ? 0 : (page - 1) * size + 1}-
59                  {Math.min(size * page, licenseKeys.totalAmount)} of{" "}
60                  {licenseKeys.totalAmount}
61                </TableCell>
62                <TableCell className="flex justify-center">
63                  <Pagination
64                    currentPage={page}
65                    pageSize={size}
66                    totalAmount={licenseKeys.totalAmount}
67                  />
68                </TableCell>
69              </>
70            )}
71          </TableRow>
72        </TableFooter>
73      </Table>
74    );
75  }
```

Listing 5.42: LicenseKeyTable component - routes/licenses/$licenseId.tsx

The first button that was mentioned previously, its component is in listing 5.43, and it is composed of two main components, a *Dialog* and a *Toaster*. The first one provides a button that, when clicked, fetches all the users in the system, using the service function in listing 5.44. After this happens, a modal is shown to the user. The users are then filtered, removing users that are already assigned to that specific key, and used to populate a *Select* component. Then, there is a *Button* component, only enabled when a user is selected, that when pressed, sends a request to the backend, using the function defined in listing 5.45. If this request is successful, the page is refreshed to show the new data to the user. If not, a toast is shown in the bottom right of the user's screen, manifesting the error that occurred

with the action.

```
export default function AssignLicenseKeyModal({
  licenseKey,
}: {
  licenseKey: licenseKeysWithUsers;
}) {
  const { jwt } = useAuth();

  const [isOpen, setIsOpen] = useState<boolean>(false);
  const [users, setUsers] = useState<Array<z.infer<typeof selectUserSchema>>>([]);
  const [selectedUser, setSelectedUser] = useState<z.infer<typeof selectUserSchema>>();
  const { toast } = useToast();

  return (
    <>
      <Toaster />
      <Dialog open={isOpen} onOpenChange={setIsOpen}>
        <DialogTrigger asChild>
          <Button
            size="sm"
            variant="ghost"
            onClick={() => {
              getUsers(jwt, []).then((res) => {
                if (res.data) {
                  setUsers(res.data.entries);
                }
              });
            }}
          >
            <HandHelping className="size-5 text-green-600" />
          </Button>
        </DialogTrigger>
        <DialogContent className="sm:max-w-[450px]">
          <DialogHeader>
            <DialogTitle>Select a user</DialogTitle>
          </DialogHeader>
          <Card>
            <CardHeader>
              <CardTitle>Please choose an user to assign the key to</CardTitle>
            </CardHeader>
            <CardContent className="space-y-2">
              <div className="grid gap-4 py-4">
                <div className="grid grid-cols-4 items-center gap-4">
                  <Label htmlFor="user" className="text-right">
                    Users
                  </Label>
                  <Select onValueChange={(val) => setSelectedUser(users[Number(val)])}>
                    <SelectTrigger className="col-span-3 text-left" id="user">
                      <SelectValue placeholder="Select a user" />
                    </SelectTrigger>
                    <SelectContent>
                      <SelectGroup>
                        {users
                          .filter(
                            (user) =>
                              !licenseKey.usersToKeys.some(
                                (userToKey) => userToKey.userId === user.id,
                              ),
```

```
                        )
                        .map((user, idx) => (
                          <SelectItem key={user.id} value={String(idx)}>
                            {user.name} | {user.type}
                          </SelectItem>
                        ))}
                      </SelectGroup>
                    </SelectContent>
                  </Select>
                </div>
              </div>
            </CardContent>
            <CardFooter>
              <Button
                disabled={!selectedUser}
                onClick={async () => {
                  if (selectedUser) {
                    const res = await assignKeyToUser(
                      jwt,
                      licenseKey.id,
                      selectedUser.id,
                    );

                    if (res.error) {
                      toast({
                        title: "Something went wrong, please try again.",
                        description: JSON.parse(res.error.value as string).message,
                        variant: "destructive",
                      });
                      return;
                    }
                    setIsOpen(false);
                    location.reload();
                  }
                }}
              >
                Confirm
              </Button>
            </CardFooter>
          </Card>
        </DialogContent>
      </Dialog>
    </>
  );
}
```

Listing 5.43: AssignLicenseKeyModal component - components/AssignLicenseKeyModal.tsx

```
export const getUsers = async (token: string, type?: UserTypes[]) => {
  return await api.v1.user.get({
    query: { type },
    headers: { authorization: `Bearer ${token}` },
  });
};
```

Listing 5.44: getUsers service function - services/user.ts

```
export const getUsers = async (token: string, type?: UserTypes[]) => {
  return await api.v1.user.get({
    query: { type },
    headers: { authorization: `Bearer ${token}` },
  });
};
```

Listing 5.45: assignKeyToUser service function - services/licenseKey.ts

For the second button, the component can be seen in listing 5.46. This component is, again, composed by two main components: a *Dialog* and a *Toaster*. When the button that the first component provides is clicked, a modal is shown to the user with a *Select* of the users that are assigned to the license key. The user can then choose one of them and confirm the action, which triggers a function, shown in listing 5.47, that sends a request to the backend to remove the license key from the user. If this request ends with an error, a toast is shown in the bottom right of the screen. If it ends successfully, the page is refreshed and the data is updated on the frontend.

```

```

Listing 5.46: RemoveKeyModal component - components/RemoveKeyModal.tsx

```
export const unassignKeyFromUser = async (
  token: string,
  licenseKeyId: number,
  userId: number,
) => {
  return await api.v1
    .licenseKey({ licenseKeyId })
    .unassign({ userId })
    .delete(null, { headers: { authorization: `Bearer ${token}` } });
};
```

Listing 5.47: unassignKeyFromUser service function - services/licenseKey.ts

5.3.11 FR4 - View requestable licenses

This functional requirement, albeit simple, is very important for the user's experience. The purpose is for any type of user to visit a page where they can visualize all of the licenses that can be requested, as well details of specific licenses and their products (a license's sub-licenses).

To implement this functional requirement, firstly it was needed to create the Drizzle query functions to fetch the licenses from the database. To do so, in listing 5.48, the function declared receives a page, a page size, and an optional *parentLicenseId*. Adding this last parameter allows for this function to be used not only to fetch all of the top-level licenses but also to query the product licenses, which happens in lines 8 to 10, depending on whether or not the *parentLicenseId* is present. Then, leveraging Drizzle's capabilities, two table joins are created, one for the *managersToLicenses* table and the other for the *user* table. Lastly, the query is paginated depending on the values passed to the function.

```
1  export const getActiveLicensesPaginated = async (
2    page = 1,
3    size = 10,
4    parentId?: number,
5  ): Promise<PaginatedQuery<Array<licenseWithManagers>>> => {
6    const data = await db.query.licenses.findMany({
7      where: (licenses, { and, isNull, eq }) =>
8        parentId
9          ? and(isNull(licenses.deletedAt), eq(licenses.parentLicenseId, parentId))
10         : and(isNull(licenses.deletedAt), isNull(licenses.parentLicenseId)),
11     with: {
12       managersToLicenses: {
13         with: {
14           user: true,
15         },
16       },
17     },
18     limit: size,
19     offset: (page - 1) * size,
20   });
21
22   const amount = await db
23     .select({ count: count() })
24     .from(licenses)
25     .where(
26       parentId
27         ? and(isNull(licenses.deletedAt), eq(licenses.parentLicenseId, parentId))
28         : and(isNull(licenses.deletedAt), isNull(licenses.parentLicenseId)),
29     );
30   return {
31     entries: data,
32     totalAmount: amount[0].count,
33     page: page,
34     size: size,
35   };
36 };
```

Listing 5.48: getActiveLicensesPaginated function definition - database/license.db.ts

To allow the client to elaborate proper pagination, the total amount of entities need to be returned as well, which happens in lines 22 to 29, where the condition, which was previously used, is applied to the select operation to only count records that are directly related to the operation that is currently being made.

This function is then called by the service function in listing 5.49, which, in this case, only returns its result. This result is then returned to the controller, where two routes are defined (one for the top-level licenses and another one for the license products), as well as any needed input validations, which happens in listing 5.50.

```
1  export const getActiveLicensesPaginated = async (
2    page = 1,
3    size = 10,
4    parentId?: number,
5  ): Promise<PaginatedQuery<Array<licenseWithManagers>>> => {
6    return await getActiveLicensesPaginated(page, size, parentId);
7  };
```

Listing 5.49: getLicensesPaginated function definition - service/license.service.ts

```
.get("", async ({ query }) => await getActiveLicensesPaginated(query.page, query.size), {
    query: t.Object({
      page: t.Optional(t.Numeric()),
    size: t.Optional(t.Numeric()),
  }),
})
.get(
  "/:licenseId/product",
  async ({ params, query }) => await getActiveLicensesPaginated(query.page, query.size, params.licenseId),
  {
    params: t.Object({
      licenseId: t.Numeric(),
    }),
    query: t.Object({
      page: t.Optional(t.Numeric()),
      size: t.Optional(t.Numeric()),
    }),
  },
)
```

Listing 5.50: GET /v1/licenses/ & GET /v1/licenses/:licenseId/products routes definition - service/license.service.ts

On the frontend, as mentioned previously in subsection 5.3.8, routes in listing 5.21, and the component in 5.22 were created, and a subsequent route, in listing 5.26. The query parameters defined for the route allow for the frontend to retrieve the license data based on its page value, and if the user goes to the next page, the whole UI isn't updated, but only the components that directly depend on it. The specific license route, in listing 5.26, had to be configured using a dynamic value on the route (in this case its license identification), which allows the user to visit the specific license page he wants, with the data being loaded depending on the visited URL. This component is presented in listing 5.25 and both of the pages render the component *LicensesTable*, shown in listing 5.51. This component follows a structure that is common throughout the application. Firstly the headers of the table are defined. Then, using the values retrieved from the backend, the cells are populated with the licenses' information, as well as a button to view more details about each of the licenses and to further inspect any of its products.

```
export default function LicenseTable({
  licenses,
  page,
  size,
}: {
  licenses: PaginatedQuery<Array<licenseWithManagers>>;
  page: number;
  size: number;
}) {
  return (
    <Table>
      <TableHeader>
        <TableRow>
          <TableHead className="w-[10%]">Id</TableHead>
```

```
15            <TableHead className="w-[45%]">Name</TableHead>
16            <TableHead className="w-[10%] text-center">Key Validity (months)</TableHead>
17            <TableHead className="w-[15%] text-center">Managers</TableHead>
18            <TableHead className="w-[10%] text-center">Automatable</TableHead>
19            <TableHead className="w-[10%]" />
20          </TableRow>
21        </TableHeader>
22        <TableBody>
23          {licenses.entries.map((license: licenseWithManagers) => (
24            <TableRow key={license.id}>
25              <TableCell className="font-medium">{license.id}</TableCell>
26              <TableCell>{license.name}</TableCell>
27              <TableCell className="text-center">{license.keyValidity}</TableCell>
28              <TableCell className="text-center">
29                {license.managersToLicenses.length === 0 ? (
30                  "---"
31                ) : (
32                  <ShowUsers users={license.managersToLicenses} />
33                )}
34              </TableCell>
35              <TableCell>
36                {license.automationConfigured ? (
37                  <Check className="m-auto" />
38                ) : (
39                  <X className="m-auto" />
40                )}
41              </TableCell>
42              <TableCell>
43                <Link
44                  to="/licenses/$licenseId"
45                  params={{ licenseId: String(license.id) }}
46                  search={{ page: 1 }}
47                >
48                  <Button variant="secondary">More info</Button>
49                </Link>
50              </TableCell>
51            </TableRow>
52          ))}
53        </TableBody>
54        <TableFooter>
55          <TableRow>
56            <TableCell colSpan={5} className="text-right">
57              {(page - 1) * size + 1}-{Math.min(size * page, licenses.totalAmount)} of{" "}
58              {licenses.totalAmount}
59            </TableCell>
60            <TableCell className="flex justify-center">
61              <Pagination
62                currentPage={page}
63                pageSize={size}
64                totalAmount={licenses.totalAmount}
65              />
66            </TableCell>
67          </TableRow>
68        </TableFooter>
69      </Table>
70    );
71  }
```

Listing 5.51: LicenseTable component - components/license-table.tsx

On the page in listing 5.25, on the Tab component mentioned in subsection 5.3.8, the products are presented to the user, and they can be further inspected. When the user enters this level of inspection, he cannot go further, as products cannot be parent licenses of other products.

teste 5.25

5.3.12 FR5 - Request a license key & FR7 - Act on license key requests

This sub-section will go over two different functional requirements that, despite being different, overlap in some areas, specially in the component. Both of the functional requirements have, as its clear purpose, the goal of managing information regarding licecnse requests. The first one is destined to regular users of the application that, if needed, will visit the frontend of the system, browse through the available software licenses and request one of the licenses they might need for their work. Then, the second functional requirement is applied, where a license manager will visit the same frontend, but to a different page, to view all of the license requests and either approve or deny them.

The implementation starts on the *Core Service* component, the database functions in listings 5.52 were defined, which will be used for the license request creation.

```
export const getLicenseRequestFromUserToLicenseOrThrow = async (
  userId: number,
  licenseId: number,
) => {
  const request = await db.query.licenseRequests.findFirst({
    where: (licenseRequests, { eq, and }) =>
      and(eq(licenseRequests.userId, userId), eq(licenseRequests.licenseId, licenseId)),
  });

  if (!request) {
    throw new EntryNotFoundError("License key wasn't found");
  }
  return request;
};

export const createLicenseRequestAndReturn = async (
  userId: number,
  licenseId: number,
) => {
  return returnFirst(await db
    .insert(licenseRequests)
    .values({ licenseId: licenseId, userId: userId })
    .returning())
}
```

Listing 5.52: getLicenseRequestFromUserToLicenseOrThrow and createLicenseRequestAndReturn function definition - database/licenseRequest.db.ts

To act on requests, either by approving or denying, the functions in listing 5.53 were created.

```
export const getLicenseRequestOrThrow = async (id: number) => {
  const request = await db.query.licenseRequests.findFirst({
    where: (licenseRequests, { eq }) => eq(licenseRequests.id, id),
    with: {
```

```
      license: {
        with: {
          managersToLicenses: {},
          licenseKeys: {},
        },
      },
    },
  });

  if (!request) {
    throw new EntryNotFoundError();
  }
  return request;
};

export const updateLicenseRequestAndReturn = async (
  id: number,
  state: LicenseRequestState,
) => {
  return returnFirst(
    await db
      .update(licenseRequests)
      .set({ state })
      .where(eq(licenseRequests.id, id))
      .returning(),
  );
};
```

Listing 5.53: getLicenseRequestOrThrow and updateLicenseRequestAndReturn function definition - database/licenseRequest.db.ts

Fetching the requests from the database involves a complex query, defined in listing 5.54, which starts by defining two sub-queries: the first one to get all the license identifications from the license that the user currently manages; and the second one to get all the products identifications that will be retrieved from the first sub-query. These sub-queries are then used to find all the license requests that were made concerning those entries, filtered by their state.

```
export const getLicenseRequestsForManagerPaginated = async (
  managerId: number,
  page: number,
  size: number,
  state?: LicenseRequestState,
) => {
  const managedLicenses = db
    .select({ licenseId: managersToLicenses.licenseId })
    .from(managersToLicenses)
    .where(eq(managersToLicenses.userId, managerId));
  const productChildren = db
    .select({ licenseId: licenses.id })
    .from(licenses)
    .where(inArray(licenses.parentLicenseId, managedLicenses));

  const data = await db.query.licenseRequests.findMany({
    where: (licenseRequests, { inArray, eq, and }) => {
      console.log(state);
      if (state) {
        return and(
```

```
            or(
              inArray(licenseRequests.licenseId, managedLicenses),
              inArray(licenseRequests.licenseId, productChildren),
            ),
            eq(licenseRequests.state, state),
          );
        }
        return or(
          inArray(licenseRequests.licenseId, managedLicenses),
          inArray(licenseRequests.licenseId, productChildren),
        );
      },
      with: {
        license: {},
        user: {},
      },
      limit: size,
      offset: (page - 1) * size,
    });

    const amount = await db
      .select({ count: count() })
      .from(licenseRequests)
      .where(
        state
          ? and(
              or(
                inArray(licenseRequests.licenseId, managedLicenses),
                inArray(licenseRequests.licenseId, productChildren),
              ),
              eq(licenseRequests.state, state),
            )
          : or(
              inArray(licenseRequests.licenseId, managedLicenses),
              inArray(licenseRequests.licenseId, productChildren),
            ),
      );
    return {
      entries: data,
      totalAmount: amount[0].count,
      page: page,
      size: size,
    };
};
```

Listing 5.54: getLicenseRequestsForManagerPaginated function definition - database/licenseRequest.db.ts

Lastly, a function to create a new key and assign it to a user was defined in 5.55.

```
export const createKeyAndReturn = async(encryptedString: string, iv: string, licenseId: number, multipleAs
  return returnFirst(
    await db
      .insert(licenseKeys)
      .values({
        key: encryptedString,
        iv: iv,
        licenseId: licenseId,
        multipleAssignments: multipleAssignments,
```

```
10        })
11        .returning(),
12    );
13  }
```

Listing 5.55: createKeyAndReturn function definition - database/licenseKey.db.ts

Then, listing 5.56 shows the function that was implemented for the service to create a license request, where it is verified if the user already has the license that is being requested assigned to himself, or if he has any pending requests. If not, the request is created and returned.

Listing 5.56: createLicenseRequest function definition - service/licenseRequest.service.ts

Listing 5.57, however, simply returns the fetched license requests from the database function to get the manager's license requests.

```
1  export const getRequestsForManagerPaginated = async (
2    managerId: number,
3    state?: LicenseRequestState,
4    page = 1,
5    size = 10,
6  ) => {
7      return await getLicenseRequestsForManagerPaginated(managerId, page, size, state);
8  };
```

Listing 5.57: getRequestsForManagerPaginated function definition - service/licenseRequest.service.ts

For the functions to act on license requests, listing 5.58 shows the functions that were created, where validations occur before updating the database entry.

```
1  export const approveLicenseRequest = async (
2    managerId: number,
3    requestId: number,
4    createInProvider: boolean,
5  ) => {
6    const licenseRequestWithManagers = await getLicenseRequestOrThrow(requestId);
7
8    if (
9      !licenseRequestWithManagers.license.managersToLicenses.every(
10       (entry) => entry.userId === managerId,
11     )
12   ) {
13     throw new NotAManagerError();
14   }
15
16   return await updateLicenseRequestAndReturn(requestId, LicenseRequestState.FULFILLED);
17 };
18
19 export const denyLicenseRequest = async (managerId: number, requestId: number) => {
20   const licenseRequestWithManagers = await getLicenseRequestOrThrow(requestId);
21   if (
22     !licenseRequestWithManagers.license.managersToLicenses.every(
23       (entry) => entry.userId === managerId,
24     )
```

```
  ) {
    throw new NotAManagerError();
  }

  return await updateLicenseRequestAndReturn(requestId, LicenseRequestState.DENIED);
};
```

Listing 5.58: approveLicenseRequest and denyLicenseRequest function definition - service/licenseRequest.service.ts

The service function to create a key, defined in listing 5.59, encrypts the received key value, after performing the validations. It then saves the key to the database and assigns it to the user received as a parameter.

```
export const createKey = async (
  createdById: number,
  userId: number,
  createKeyDto: Omit<z.infer<typeof createLicenseKeysSchema>, "iv">,
) => {
  const licenseWithManagers = await getLicenseOrThrow(createKeyDto.licenseId);

  if (
    !licenseWithManagers.managersToLicenses.every(
      (entry) => entry.userId === createdById,
    )
  ) {
    throw new AuthenticationError();
  }

  const { encryptedString, iv } = encryptString(createKeyDto.key);

  const key = await createKeyAndReturn(encryptedString, iv, createKeyDto.licenseId, createKeyDto.multipleA
  await assignKeyToUserAndReturn(key.id, userId);
};
```

Listing 5.59: createKey function definition - service/licenseKey.service.ts

For the controller routes, to create a license request, the route on path /v1/license/:licenseId/request with the method POST was defined in listing 5.60.

```
.post(
  "/:licenseId/request",
  async ({ params, user }) => {
    if (!user) {
      throw new AuthenticationError();
    }

    return await requestLicense(params.licenseId, user.id);
  },
  {
    params: t.Object({
      licenseId: t.Numeric(),
    }),
  },
)
```

Listing 5.60: POST /v1/license/:licenseId/request route definition - controller/license.controller.ts

To either approve or deny license requests, routes /v1/request/approve and /v1/request/deny, both with the method PATCH, were created in listing **??**.

```
.patch(
  "/:requestId/approve",
  async ({ params, user }) => {
    if (!user || !hasRole(user, UserTypes.MANAGER)) {
      throw new AuthenticationError();
    }

    return await approveLicenseRequest(user.id, params.requestId, false);
  },
  {
    params: t.Object({
      requestId: t.Numeric(),
    }),
  },
)
.patch(
  "/:requestId/refuse",
  async ({ params, user }) => {
    if (!user || !hasRole(user, UserTypes.MANAGER)) {
      throw new AuthenticationError();
    }

    return await denyLicenseRequest(user.id, params.requestId);
  },
  {
    params: t.Object({
      requestId: t.Numeric(),
    }),
  },
)
```

Listing 5.61: PATCH /v1/request/:requestId/approve & /v1/request/:requestId/refuse route definition - controller/licenseRequest.controller.ts

Listing 5.62 defines a route to get the license requests related to a manager on path /v1/request with the method.

```
.get(
  "",
  async ({ query, user }) => {
    if (!user || !hasRole(user, UserTypes.MANAGER)) {
      throw new AuthenticationError();
    }
    return await getRequestsForManagerPaginated(user.id, query.state, query.page);
  },
  {
    query: t.Object({
      page: t.Numeric(),
      state: t.Optional(t.Enum(LicenseRequestState)),
    }),
  },
);
```

Listing 5.62: GET /v1/request route definition - controller/licenseRequest.controller.ts

Lastly, route /v1/license/:licenseId/key with the method POST was created to create a license key and assign it directly to a user, and can be seen in listing 5.63.

```
.post(
  "/:licenseId/key",
  async ({ params, user, body }) => {
    if (!user) {
      throw new AuthenticationError();
    }
    return createKey(user.id, body.userId, { ...body, licenseId: params.licenseId });
  },
  {
    params: t.Object({
      licenseId: t.Numeric(),
    }),
    body: t.Object({
      userId: t.Numeric(),
      key: t.String(),
      multipleAssignments: t.Boolean(),
    }),
  },
);
```

Listing 5.63: POST /v1/license/:licenseId/key route definition - controller/license.controller.ts

For the *Frontend Web* component, a new route, with the purpose of handling the requests data, was created and is shown in listing 5.64. This is similar to the routes defined previously but has a new query parameter: *state*. This, as well as the *page* will be used to filter the data shown in the page rendered by component *Requests*, in listing 5.65. This component is composed of two components: the *ToggleGroup* and the *RequestTable*. The *ToggleGroup* is used to change the *state* query parameter to fetch filtered requests on demand.

```
export const Route = createFileRoute("/requests")({
  validateSearch: z.object({
    page: z.number().default(1).catch(1),
    state: z.union([z.nativeEnum(LicenseRequestState), z.literal("")]).default(""),
  }),
  component: Requests,
  loaderDeps: ({ search: { page, state } }) => ({ page, state }),
  loader: async ({ deps: { page, state }, context }) => {
    if (context.auth.isAuthenticated) {
      return (await loadActiveRequestsPaginated(context.auth.jwt, page, state)).data;
    }
  },
});
```

Listing 5.64: /requests route definition - routes/requests.tsx

```
function Requests() {
  const data = Route.useLoaderData();
  const search = Route.useSearch();
```

```
4   const navigate = useNavigate({ from: "/requests" });
5   const pageSize = 10;
6
7   return (
8     <>
9       <div className="flex items-center">
10        <h1 className="text-lg font-semibold md:text-2xl">License Requests</h1>
11        <div className="flex ml-auto items-center">
12          <span className="mr-4">Filter:</span>
13          <ToggleGroup
14            variant="outline"
15            onValueChange={(e) =>
16              navigate({
17                to: "/requests",
18                search: { page: 1, state: e as LicenseRequestState },
19              })
20            }
21            value={search.state}
22            type="single"
23          >
24            <ToggleGroupItem value={LicenseRequestState.PENDING}>Pending</ToggleGroupItem>
25            <ToggleGroupItem value={LicenseRequestState.FULFILLED}>
26              Fulfilled
27            </ToggleGroupItem>
28            <ToggleGroupItem value={LicenseRequestState.DENIED}>Denied</ToggleGroupItem>
29          </ToggleGroup>
30        </div>
31      </div>
32      <div
33        className="flex flex-1 justify-center rounded-lg border border-dashed shadow-sm"
34        x-chunk="dashboard-02-chunk-1"
35      >
36        {data && data.entries.length !== 0 ? (
37          <RequestTable requests={data} page={search.page} size={pageSize} />
38        ) : (
39          <EmptyRequestsPage />
40        )}
41      </div>
42    </>
43  );
44  }
```

Listing 5.65: Requests page definition - routes/requests.tsx

Regarding the *RequestTable* component, it follows the same pattern as the previous tables, with the only difference being the two buttons presented at the end of each entry: *ApproveRequestModal* and *DenyLicenseRequest*.

The *ApproveRequestModal* component, shown in listing 5.66, presents a modal to the user where it displays two *Tabs*, one to assign an existing license key to the user, and another to create a new license key and assign it to the user. In the first case, the component *AssignAvailableKeyToUserCard* in listing 5.67 is invoked, and, when clicked, the user is presented with a *Select* component with all the assignable license keys, fetched using the service method *getLicenseKeysFromLicense*, with the *assignable* flag set to true. This will request the backend to respond with all the license keys that are available to be assigned to this license request. Then, with the license keys ready, it will fetch a *Select* component with a list of licenses that the manager can assign to the user. After one is selected and the

submit button is pressed, function *assignKeyToUser* is called, which sends a request to the backend to save the key's assignment. If everything resolves correctly, the page is refreshed and the user can see the newly input data. If it isn't, then a toast is shown, saying the request is finished with an error.

```
function ApproveRequestModal({
  isOpen,
  setIsOpen,
  request,
}: {
  isOpen: boolean;
  setIsOpen: Dispatch<SetStateAction<boolean>>;
  request: requestsWithUserAndLicense;
}) {
  const { jwt } = useAuth();
  const [licenseKeys, setLicenseKeys] = useState<
    Array<
      z.infer<typeof selectLicenseKeysSchema> & {
        usersToKeys: Array<z.infer<typeof selectUsersToLicenseKeysSchema>>;
      }
    >
  >([]);

  return (
    <>
      <Toaster />
      <Dialog open={isOpen} onOpenChange={setIsOpen}>
        <DialogTrigger asChild>
          <Button
            size="sm"
            variant="ghost"
            onClick={() => {
              if (request.licenseId > 0) {
                getLicenseKeysFromLicense(jwt, request.licenseId, true).then((res) => {
                  if (res.data) {
                    setLicenseKeys(res.data);
                  }
                });
              }
            }}
          >
            <Check className="size-5 text-green-600" />
          </Button>
        </DialogTrigger>
        <DialogContent className="sm:max-w-[450px]">
          <DialogHeader>
            <DialogTitle>Assign a license key to the user</DialogTitle>
          </DialogHeader>
          <Tabs defaultValue="existingKey" className="w-[400px] mt-5">
            <TabsList className="grid w-full grid-cols-2">
              <TabsTrigger value="existingKey">Use an available key</TabsTrigger>
              <TabsTrigger value="newKey">Create a new key</TabsTrigger>
            </TabsList>
            <TabsContent value="existingKey">
              <AssignAvailableKeyToUserCard licenseKeys={licenseKeys} request={request} />
            </TabsContent>
            <TabsContent value="newKey">
              <CreateLicenseKeyCard
```

```
                selectedUserId={request.userId}
                licenseId={request.licenseId}
                users={[request.user]}
              />
            </TabsContent>
          </Tabs>
        </DialogContent>
      </Dialog>
    </>
  );
}
```

Listing 5.66: ApproveRequestModal component definition - routes/requests.tsx

```
function AssignAvailableKeyToUserCard({
  request,
  licenseKeys,
}: {
  request: requestsWithUserAndLicense;
  licenseKeys: Array<
    z.infer<typeof selectLicenseKeysSchema> & {
      usersToKeys: Array<z.infer<typeof selectUsersToLicenseKeysSchema>>;
    }
  >;
}) {
  const [selectedKey, setSelectedKey] = useState<
    z.infer<typeof selectLicenseKeysSchema> & {
      usersToKeys: Array<z.infer<typeof selectUsersToLicenseKeysSchema>>;
    }
  >();

  const { toast } = useToast();
  const { jwt } = useAuth();

  return (
    <>
      <Toaster />
      <Card>
        <CardHeader>
          <CardTitle>Use an available key</CardTitle>
          <CardDescription>
            Choose an available key from the list below. If there aren't any available,
            please consider creating one.
          </CardDescription>
        </CardHeader>
        <CardContent className="space-y-2">
          <div className="grid gap-4 py-4">
            <div className="grid grid-cols-4 items-center gap-4">
              <Label htmlFor="user" className="text-right">
                Keys
              </Label>
              <Select onValueChange={(val) => setSelectedKey(licenseKeys[Number(val)])}>
                <SelectTrigger className="col-span-3 text-left" id="user">
                  <SelectValue placeholder="Select a license key" />
                </SelectTrigger>
                <SelectContent>
                  <SelectGroup>
                    {!!licenseKeys &&
```

```
45                      licenseKeys.map((licenseKey, idx) => (
46                        <SelectItem key={licenseKey.id} value={String(idx)}>
47                          {licenseKey.key}
48                          {licenseKey.multipleAssignments
49                            ? ` | Used by ${licenseKey.usersToKeys.length}`
50                            : ""}
51                        </SelectItem>
52                      ))}
53                    </SelectGroup>
54                  </SelectContent>
55                </Select>
56              </div>
57            </div>
58          </CardContent>
59          <CardFooter>
60            <Button
61              disabled={!selectedKey}
62              onClick={async () => {
63                if (selectedKey) {
64                  const res = await assignKeyToUser(jwt, selectedKey?.id, request.userId);
65
66                  if (res.error) {
67                    toast({
68                      title: "Something went wrong, please try again.",
69                      description: JSON.parse(res.error.value as string).message,
70                      variant: "destructive",
71                    });
72                    return;
73                  }
74
75                  const res2 = await approveRequest(jwt, request.id);
76
77                  if (res2.error) {
78                    toast({
79                      title: "Something went wrong, please try again.",
80                      description: JSON.parse(res2.error.value as string).message,
81                      variant: "destructive",
82                    });
83                    return;
84                  }
85
86                  location.reload();
87                }
88              }}
89            >
90              Assign selected key
91            </Button>
92          </CardFooter>
93        </Card>
94      </>
95    );
96  }
```

Listing 5.67: AssignAvailableKeyToUserCard component definition - routes/requests.tsx

Regarding the second tab, the component *CreateLicenseKeyCard*, shown in listing 5.68, displays a form with one *Select* component and one *Input* component, the first one being pre-selected with the user that created the license request. After the manager inputs the

license key to create and clicks on the submit button, the *createLicenseKey* function is called, and the request is sent to the backend. Once the response is back, the behavior is the same as the component explained previously.

```
const formSchema = z.object({
  key: z.string(),
  multipleAssignments: z.boolean(),
  userId: z.number(),
});

export function CreateLicenseKeyCard({
  selectedUserId,
  users,
  licenseId,
}: {
  selectedUserId?: number;
  licenseId: number;
  users: Array<z.infer<typeof selectUserSchema>>;
}) {
  const { jwt } = useAuth();
  const form = useForm<z.infer<typeof formSchema>>({
    resolver: zodResolver(formSchema),
    defaultValues: {
      key: "",
      multipleAssignments: false,
      userId: selectedUserId || -1,
    },
  });

  async function onSubmit(values: z.infer<typeof formSchema>) {
    await createLicenseKey(jwt, licenseId, values);
  }

  return (
    <Card>
      <CardHeader>
        <CardTitle>Create a new key</CardTitle>
        <CardDescription>
          Enter the key information and assign it to the selected user.
        </CardDescription>
      </CardHeader>
      <CardContent className="space-y-2">
        <Form {...form}>
          <form onSubmit={form.handleSubmit(onSubmit)}>
            <FormField
              control={form.control}
              name="userId"
              render={({ field }) => (
                <FormItem>
                  <FormLabel>User</FormLabel>
                  <FormControl>
                    <Select
                      disabled={!!selectedUserId}
                      onValueChange={field.onChange}
                      defaultValue={String(selectedUserId)}
                    >
                      <FormControl>
                        <SelectTrigger>
                          <SelectValue placeholder="Select a user to assign the key to" />
```

```
                        </SelectTrigger>
                      </FormControl>
                      <SelectContent>
                        {users.map((user) => (
                          <SelectItem key={user.id} value={String(user.id)}>
                            {user.name}
                          </SelectItem>
                        ))}
                      </SelectContent>
                    </Select>
                  </FormControl>
                </FormItem>
              )}
            />
            <FormField
              control={form.control}
              name="key"
              render={({ field }) => (
                <FormItem>
                  <FormLabel>Key Value</FormLabel>
                  <FormControl>
                    <Input placeholder="XXXX-XXXX-XXXX-XXXX" {...field} />
                  </FormControl>
                </FormItem>
              )}
            />
            <FormField
              control={form.control}
              name="multipleAssignments"
              render={({ field }) => (
                <FormItem className="flex items-end">
                  <FormLabel className="">Multiple Assignments</FormLabel>
                  <FormControl>
                    <Checkbox
                      className="ml-2"
                      checked={field.value}
                      onCheckedChange={field.onChange}
                    />
                  </FormControl>
                </FormItem>
              )}
            />
            <Button className="mt-6" type="submit">
              Submit
            </Button>
          </form>
        </Form>
      </CardContent>
    </Card>
  );
}
```

Listing 5.68: CreateLicenseKeyCard component definition - components/create-license-key-card.tsx

Lastly, the component *DenyLicenseRequest*, shown in listing 5.69, which is displayed inside the *newKey* tab, simply shows an alert dialog asking for the user's confirmation on the action. Once it is confirmed, the backend is requested regarding the update of the request's state and the same response behavior happens.

```tsx
function DenyLicenseRequest({ requestId }: { requestId: number }) {
  const { toast } = useToast();
  const { jwt } = useAuth();
  const navigate = useNavigate({ from: "/requests" });
  const search = Route.useSearch();
  return (
    <>
      <Toaster />
      <AlertDialog>
        <AlertDialogTrigger asChild>
          <Button size="sm" variant="ghost">
            <X className="size-5 text-red-600" />
          </Button>
        </AlertDialogTrigger>
        <AlertDialogContent>
          <AlertDialogHeader>
            <AlertDialogTitle>Please confirm your action.</AlertDialogTitle>
            <AlertDialogDescription>
              Are you sure you want to deny this request?
            </AlertDialogDescription>
          </AlertDialogHeader>
          <AlertDialogFooter>
            <AlertDialogCancel>Cancel</AlertDialogCancel>
            <AlertDialogAction
              onClick={async () => {
                const res = await denyRequest(jwt, requestId);
                if (res.error) {
                  toast({
                    title: "Something went wrong, please try again.",
                    description: JSON.parse(res.error.value as string).message,
                    variant: "destructive",
                  });
                  return;
                }
                navigate({
                  to: "/requests",
                  search: { page: search.page, state: search.state },
                  replace: true,
                });
              }}
            >
              Deny
            </AlertDialogAction>
          </AlertDialogFooter>
        </AlertDialogContent>
      </AlertDialog>
    </>
  );
}
```

Listing 5.69: DenyLicenseRequest component definition - components/create-license-key-card.tsx

5.3.13 FR6 - Update users' permissions

The objective of this functional requirement is for a system administrator to update a user's permissions. To do so, a page will be implemented on the client-side, where the system

administrator can visualize every user that exists in the system, and update the type of any user available in the table.

To do so, the first step was to implement this functional requirement was to define a database function to update a user's type, shown in listing 5.70.

```
export const changeUserTypeAndReturn = async (id: number, type: UserTypes) => {
  return returnFirst(
    await db.update(users).set({ type }).where(eq(users.id, id)).returning(),
  );
};
```

Listing 5.70: getUsersPaginated function definition - database/user.db.ts

Then, the service function to return the content from the database was created, which is used by the route to path /v1/user/:userId/type with the PATCH method, defined in listing 5.71

```
.patch(
  "/:userId/type",
  async ({ params, body }) => await changeUserType(params.userId, body.type),
  {
    params: t.Object({
      userId: t.Numeric(),
    }),
    body: t.Object({
      type: t.Enum(UserTypes),
    }),
  },
)
```

Listing 5.71: PATCH /v1/user/:userId route definition - controller/user.controller.ts

Now, for the user to visualize the data on the client-side, a function to retrieve every user from the database is needed. Therefore, in listing 5.72 a function to do so was implemented.

```
export const getUsersPaginated = async (
  page: number,
  size: number,
  types?: UserTypes[],
) => {
  const data = await db.query.users.findMany({
    where: (users, { and, inArray, isNull }) =>
      types
        ? and(isNull(users.deletedAt), inArray(users.type, types))
        : isNull(users.deletedAt),
    limit: size,
    offset: (page - 1) * size,
  });
  const amount = await db
    .select({ count: count() })
    .from(users)
    .where(
      types
        ? and(isNull(users.deletedAt), inArray(users.type, types))
        : isNull(users.deletedAt),
    );
```

```
22    return {
23      entries: data,
24      totalAmount: amount[0].count,
25      page: page,
26      size: size,
27    };
28  };
```

Listing 5.72: getUsersPaginated function definition - database/user.db.ts

Then, a service function that returns the value from the previous function was defined, as well as the route to path /v1/user with the GET method, seen in listing 5.73. Here, two functions are defined, with the only difference between them being the fact that one of them returns paginated users, while the other one returns all users in the system, filtered by the type sent to the route.

```
1   .get(
2     "",
3     async ({ query }) => {
4       if (query.page || query.size) {
5         return await getActiveUsersPaginated(query.type, query.page, query.size);
6       }
7       return await getActiveUsers(query.type);
8     },
9     {
10      query: t.Object({
11        page: t.Optional(t.Numeric()),
12        size: t.Optional(t.Numeric()),
13        type: t.Optional(t.Array(t.Enum(UserTypes))),
14      }),
15    },
16  )
```

Listing 5.73: GET /v1/user route definition - controller/user.controller.ts

Once the backend was implemented, a new route was created in the *Frontend Web* component, which can be seen in 5.74. This router defines that the query parameters for this route are: *page*, and *type*. The first one will be used to build the pagination in the table to present the data, and the last one will be to filter the data shown to the user, maintaining the state in the URL.

```
1   export const Route = createFileRoute("/users/")({
2     validateSearch: z.object({
3       page: z.number().default(1).catch(1),
4       type: z.array(z.nativeEnum(UserTypes)).default([]),
5     }),
6     component: Users,
7     loaderDeps: ({ search: { page, type } }) => ({ page, type }),
8     loader: async ({ deps: { page, type }, context }) => {
9       if (context.auth.isAuthenticated) {
10        return (await getUsersPaginated(context.auth.jwt, type, page)).data;
11      }
12    },
13  });
```

Listing 5.74: /users/ route definition - routes/users/index.tsx

The page that will be shown in this route is the *Users* page, implemented in listing 5.75, which is very similar to the *Requests* page, presented in 5.3.12. It defined a table to show the users' data and a filter for the users' type.

```
function Users() {
  const data = Route.useLoaderData();
  const search = Route.useSearch();
  const navigate = useNavigate({ from: "/users" });
  const pageSize = 10;

  return (
    <>
      <div className="flex items-center">
        <h1 className="text-lg font-semibold md:text-2xl">Users</h1>
        <div className="flex ml-auto items-center">
          <span className="mr-4">Filter:</span>
          <ToggleGroup
            variant="outline"
            onValueChange={(e) =>
              navigate({
                to: "/users",
                search: { page: 1, type: e as UserTypes[] },
              })
            }
            value={search.type}
            type="multiple"
          >
            <ToggleGroupItem value={UserTypes.USER}>User</ToggleGroupItem>
            <ToggleGroupItem value={UserTypes.MANAGER}>Manager</ToggleGroupItem>
            <ToggleGroupItem value={UserTypes.ADMIN}>Admin</ToggleGroupItem>
          </ToggleGroup>
        </div>
      </div>
      <div
        className="flex flex-1 justify-center rounded-lg border border-dashed shadow-sm"
        x-chunk="dashboard-02-chunk-1"
      >
        {data && data.entries.length !== 0 ? (
          <UserTable users={data} page={search.page} size={pageSize} />
        ) : (
          <EmptyUsersPage />
        )}
      </div>
    </>
  );
}
```

Listing 5.75: Users page definition - routes/users/index.tsx

Inside the *UserTable*, shown in 5.76, next to each entry in the table, there is a button, *ChangeUserTypeDialog* which, similarly to the previous *Dialog* components, opens a modal for the user to choose the new type to be assigned to the selected user. The behavior after clicking the submit button is also the same as previously described.

```
function UserTable({
  users,
  page,
  size,
```

```
}: {
  users: PaginatedQuery<Array<z.infer<typeof selectUserSchema>>>;
  page: number;
  size: number;
}) {
  return (
    <Table>
      <TableHeader>
        <TableRow>
          <TableHead className="w-[10%]">Id</TableHead>
          <TableHead className="w-[35%]">Name</TableHead>
          <TableHead className="w-[35%]">Email</TableHead>
          <TableHead className="w-[10%]">Type </TableHead>
          <TableHead className="w-[10%]" />
        </TableRow>
      </TableHeader>
      <TableBody>
        {users.entries.map((user: z.infer<typeof selectUserSchema>) => (
          <TableRow key={user.id}>
            <TableCell className="font-medium">{user.id}</TableCell>
            <TableCell>
              <ShowUsers users={[{ user: user }]} />
            </TableCell>
            <TableCell className="text-left">{user.email}</TableCell>
            <TableCell className="text-left">{user.type}</TableCell>
            <TableCell>
              <ChangeUserTypeDialog userId={user.id} />
            </TableCell>
          </TableRow>
        ))}
      </TableBody>
      <TableFooter>
        <TableRow>
          <TableCell colSpan={4} className="text-right">
            {(page - 1) * size + 1}-{Math.min(size * page, users.totalAmount)} of{" "}
            {users.totalAmount}
          </TableCell>
          <TableCell className="flex justify-center">
            <Pagination
              currentPage={page}
              pageSize={size}
              totalAmount={users.totalAmount}
            />
          </TableCell>
        </TableRow>
      </TableFooter>
    </Table>
  );
}
```

Listing 5.76: UserTable component definition - routes/users/index.tsx

5.3.14 FR8 - Update user's license keys

The purpose of this functional requirement is to consume the messages published by the Mindera People's message broker, in this case, GCP Pub/Sub module, and to assign or remove keys based on these messages and on the dynamic license rules. To implement this, a SDK maintained by Google was used, which was *@google-cloud/pubsub*. To use this SDK,

it is necessary to connect to it using the code in listing 5.77. Here an auxiliary function was declared, where, given the *PubSub* client, a topic identification, a subscription identification, and a handler function, a connection to a subscription is established and the service starts to actively listen for messages and, when the service receives them, applying the handler function defined.

```
export const handleNewUserMessage = async (message: Message) => {
  const parsedMessage: NewUserMessage = JSON.parse(message.data.toString());
  await createUser(parsedMessage);

  //TODO: Rule Handling
};

export const handleUpdatedUserMessage = async (message: Message) => {
  //TODO: Implement user ROLES/SKILLS/TEAMS when implementing rules
};

export const handleRemoveUserMessage = async (message: Message) => {
  const parsedMessage: RemoveUserMessage = JSON.parse(message.data.toString());
  const user = await getUserByEmail(parsedMessage.email);
  const removedKeys = await unassignAllKeysFromUser(user.id);
  const managersToKeys: {[key: string]: number[]} = {};
  for (const removedKey of removedKeys) {
    const key = await getLicenseKey(removedKey.licenseKeyId);
    const licenseWithManagers = await getLicenseOrThrow(key.licenseId);
    for (const manager of licenseWithManagers.managersToLicenses) {
      const entry = managersToKeys[manager.user.slackId]
      if (!entry) {
        managersToKeys[manager.user.slackId] = [];
      }
      managersToKeys[manager.user.slackId].push(key.id);
    }

    for (const entry of Object.entries(managersToKeys)) {
      sendSlackMessageToUser(
      entry[0],
        `Hello! I just freed license keys with IDs ${entry[1].join(", ")}. Thank you!`,
      );
    }

    await deactivateUser(user.id);
  }
};
```

Listing 5.77: Initiating the PubSub client - config/gcpPubSub.ts

The handler functions were defined in listing 5.78, and the types used are present in listing 5.79. Some of them aren't complete, due to not having the dynamic rules implemented. However, to fully implement these handlers, starting by the case of *handleNewUserMessage*, after creating the user in the database, it would be needed to loop through all of the rules and, for each of them, verify if the user has the skill, role, or belongs to a team which the rule specifies. If this happens, then a license key should be created and assigned to the user. For *handleUpdatedUserMessage*, it would be necessary to verify which teams, roles, or skills the user stopped having, and if there were rules regarding these, then the license key should be unassigned from the user and, possibly, deleted. The same would have to happen for new roles, skills, or teams, but, in this case, license keys would have to be created and

assigned for each rule that would be applied. Lastly, *handleRemoveUserMessage*, which is fully implemented, starts by parsing the message received and fetching the user from the database. It then unassigns all of the keys that were previously assigned to the user and, for each key, and for each manager, a map is constructed that has all keys that were removed from licenses managed by a user, to prevent spamming the license managers with messages. Finally, a message is sent through *Slack*, implemented with the resource of a library maintained by the Slack team called *@slack/web-api*, in listing 5.80, to each license manager with the presented message.

```
export const handleNewUserMessage = async (message: Message) => {
  const parsedMessage: NewUserMessage = JSON.parse(message.data.toString());
  await createUser(parsedMessage);

  //TODO: Rule Handling
};

export const handleUpdatedUserMessage = async (message: Message) => {
  //TODO: Implement user ROLES/SKILLS/TEAMS when implementing rules
};

export const handleRemoveUserMessage = async (message: Message) => {
  const parsedMessage: RemoveUserMessage = JSON.parse(message.data.toString());
  const user = await getUserByEmail(parsedMessage.email);
  const removedKeys = await unassignAllKeysFromUser(user.id);
  for (const removedKey of removedKeys) {
    const key = await getLicenseKey(removedKey.licenseKeyId);
    const licenseWithManagers = await getLicenseOrThrow(key.licenseId);
    for (const manager of licenseWithManagers.managersToLicenses) {
      sendSlackMessageToUser(
        manager.user.slackId,
        `Hello! Unassigned license key with ID ${key.id} from ${user.name}. Thank you!`,
      );
    }
    await deactivateUser(user.id);
  }
};
```

Listing 5.78: Handler functions for user messages - listeners/user.listeners.ts

```
type NewUserMessage = {
  name: string;
  email: string;
  slackId: string;
  pictureUrl: string;
  teams: string[];
  roles: string[];
  skills: string[];
};

type UpdatedUserMessage = {
  name: string;
  email: string;
  slackId: string;
  pictureUrl: string;
  teams: string[];
  roles: string[];
  skills: string[];
```

```
};

type RemoveUserMessage = {
  name: string;
  email: string;
  slackId: string;
};
```

Listing 5.79: Types of messages received through the message broker - type-/user.listener.ts

```
const slackClient = new WebClient(env.SLACK_TOKEN);

export const sendSlackMessageToUser = async (userSlackId: string, message: string) => {
  try {
    const conversationResponse = await slackClient.conversations.open({
      users: userSlackId,
    });
    if (conversationResponse.ok) {
      const channel = conversationResponse.channel;
      if (channel?.id) {
        slackClient.chat.postMessage({ channel: channel.id, text: message });
      }
    }
  } catch (e) {
    console.error(e);
  }
};
```

Listing 5.80: Slack client initiation and functions - config/slack.ts

5.3.15 FR9 - Manage license key through license provider

This functional requirement was, unfortunately, not implemented in time for this report. However, to implement it would be needed to implement, some of the following functions on each license provider client that relate to an automated license: create a license key; assign a license key; unassign a license key from an user; delete license key; check a license key's usage, so that it could be removed, or even deleted, if the user wasn't using it anymore.

This would be implemented using the work presented in subsection 5.3.3. The implementation would start with the addition of all the license operations that would be implemented. Then, each of the client types would have to be updated, depending on the data the external API received and returned to the system. After this, using the types created, all of the operations that were defined would have to be implemented to minimize errors within the system. With this work done, any of the clients could be fetched anywhere that might be necessary, for example, in the user messages listener.

5.3.16 FR10 - Visualize historical records

To implement this functional requirement, based on the data model in sub-section 4.4.2, there would be an events table, linked to a user. This entity would store all actions made by this user, having a different *type* for each action and, with the *data* field storing the JSON object containing everything to retrace the event that was realized, it would be possible to trace every single action made in the system. Every time data was inserted, updated, or

deleted, a new entry in this table would have to be created. Based on this data, it would then be possible to create global, per-license, or per-user dashboards or reports.

5.3.17 FR11 - Manage license rules

For this functional requirement to be implemented, apart from having the Create, Read, Update and Delete (CRUD) endpoints defined with both the service and database layers implemented in *Core Service*, there would have to be a new page developed on the *Frontend Web* component, as the license managers would need an UI to manage these rules. These would be created per license, if needed, and related to any amount of roles, skills, or teams. Then, when these are available, they could be applied as described in sub-section 5.3.14 and thus automating most of the processes in this system.

In sum, at least 5 database functions would be needed: *createRuleAndReturn*, *updateRuleAndReturn*, *deleteRuleOrThrow*, *getRuleOrThrow*, and *getRulesPaginated*, where a manager, similarly to what happens in the license requests, can retrieve all rules related to the licenses and products they manage. Then, the corresponding service functions would also be implemented, with similar names, and with similar implementations to what was presented in previous sub-sections, and five routes defined: a GET and POST route on /v1/rule, and a PUT, GET, and DELETE route on /v1/rule/:ruleId. For the *Frontend Web* component, a similar structure to the *License* page component, with the only thing different being the fields that each component would show, as they are different entities.

Chapter 6

Solution Evaluation

This chapter will start by exploring the continuous testing methods that were applied throughout the development process. Then, the focus will be on presenting the pages developed for fulfilling the functional requirements defined in 3.1.2, exploring all the components that were introduced to create the UI and, lastly, the non-functional requirements in which there was no evidence throughout the development process will be reviewed.

6.1 Testing

Throughout the development of this system, three types of testing were utilized: Unit testing, and Integration testing. This section will provide examples of each of the types of tests used.

6.1.1 Unit Tests

The unit test's focus is to test small pieces, or units, of software. In this system, they were used to test each of the functions defined in services, and any additional function that seemed critical to the system. In listing 6.81 there is an example of a unit test implemented to verify if a license request is created when there are no license keys assigned to the user, and there are no active license requests. Since the objective of this test is to verify the behavior of this specific unit, all of the function calls to the database were mocked.

```
describe("licenseRequest.service.ts tests", () => {
  describe("requestLicense test", () => {
    test("pass when there are no keys and no license requests pending", async () => {
      const licenseId = 1;
      const userId = 1;

      mock.module("@core/database/licenseKey.db", () => ({
        getLicenseKeysFromLicense: () => [],
      }));
      mock.module("@core/database/licenseRequest.db", () => ({
        getLicenseRequestFromUserToLicense: () => ({
          state: LicenseRequestState.FULFILLED,
          id: 1,
          licenseId: licenseId,
          userId: userId,
        }),
        createLicenseRequestAndReturn: () => ({
          state: LicenseRequestState.PENDING,
          id: 2,
          licenseId: licenseId,
          userId: userId,
        }),
      }));

      const gottenRequest = await requestLicense(licenseId, userId);
      expect(gottenRequest).toBeDefined();
      expect(gottenRequest.state).toBe(LicenseRequestState.PENDING);
      expect(gottenRequest.id).toBe(2);
    });
  });
});
```

Listing 6.81: Example of the implementation of a unit test

6.1.2 Integration Tests

The purpose of integration tests is to test the entire flow of the system, from the controller to the database. Due to testing the database itself, it was needed to configure a database to be available every time these tests were run. This was solved by using the code in listing 5.5, where an in-memory database is used whenever the integration tests are running. Listing 6.82 provides an example of one of the integration tests that were developed. Here, the data that will be required by the routes that were tested is inserted into the database and, with the help of Elysia's *treaty* function, which provides a type-safe client to the controller, the routes that were being tested were called and its response was validated.

```
describe("licenseRequest routes tests", () => {
  const api = treaty(app);
  const userToken = "PLACEHOLDER";

  describe("PATCH /request/:requestId/approve", () => {
    beforeAll(async () => {
      await db
        .insert(users)
        .values({ name: "user1", email: "user1@mindera.com", type: UserTypes.MANAGER });
      await db
        .insert(users)
        .values({ name: "user2", email: "user2@mindera.com", type: UserTypes.USER });
      await db.insert(licenses).values({ name: "license1", keyValidity: 1 });
    });
    beforeEach(async () => {
      await db.delete(licenseRequests);
    });

    test("success if the user doesn't have an active license request", async () => {
      const { data, error } = await api.v1
        .license({ licenseId: 1 })
        .request.post(null, { headers: { authorization: `Bearer ${userToken}` } });
      expect(error).toBeNull();
      expect(data).toBeDefined();
      expect(data?.state).toBe(LicenseRequestState.PENDING);
    });

    test("error if user already has a pending license request", async () => {
      const request = await db
        .insert(licenseRequests)
        .values({ licenseId: 1, userId: 2 })
        .returning();
      expect(request.length).toBe(1);

      const { data, error } = await api.v1
        .license({ licenseId: 1 })
        .request.post(null, { headers: { authorization: `Bearer ${userToken}` } });
      expect(data).toBeNull();
      expect(error).toBeDefined();
      const errorJson = JSON.parse(error?.value as string);
      expect(errorJson).toHaveProperty("message");
      expect(errorJson.message).toBe(errorCauses.LICENSE_REQUEST_ALREADY_MADE);
    });
  });
});
```

Listing 6.82: Example of the implementation of an integration test

6.2 Implementation Evidence

In this section, all of the pages and components developed will be listed, showing how these are presented to the user and how they interact with each other.

In figure 6.1 the first page displays the users' information, with a filter being available to filter between different user types. When the filter is used and there are no users to display, figure 6.2 is shown to the user.

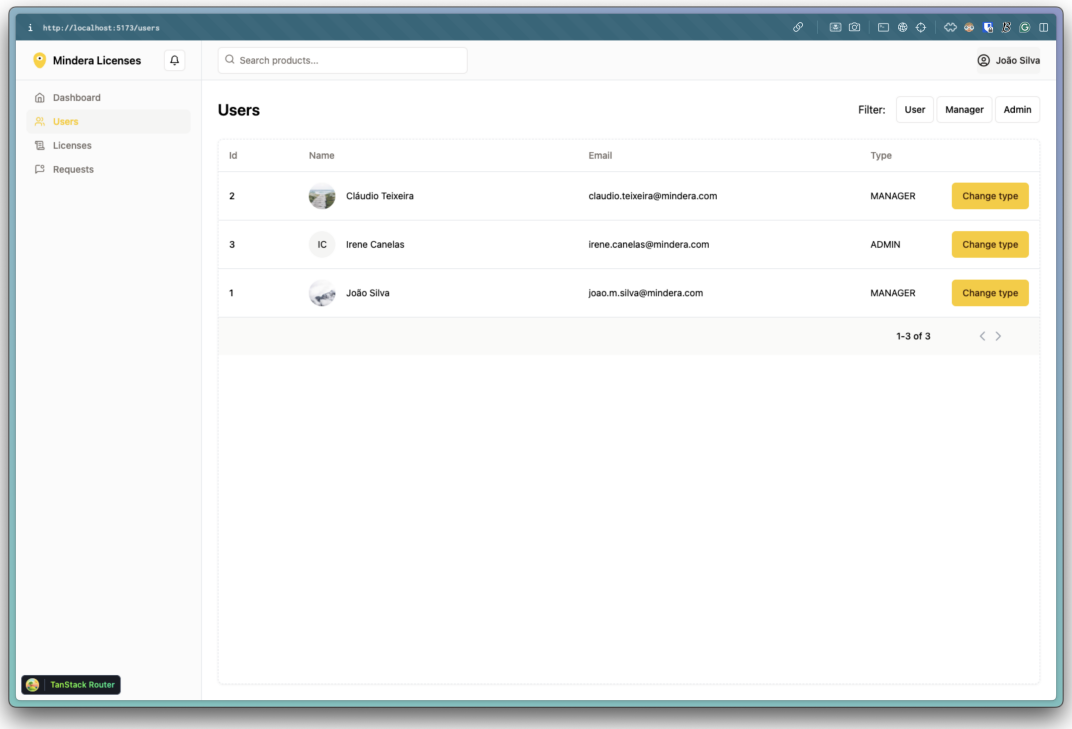

Figure 6.1: Table with the users' information, and a button to change permissions

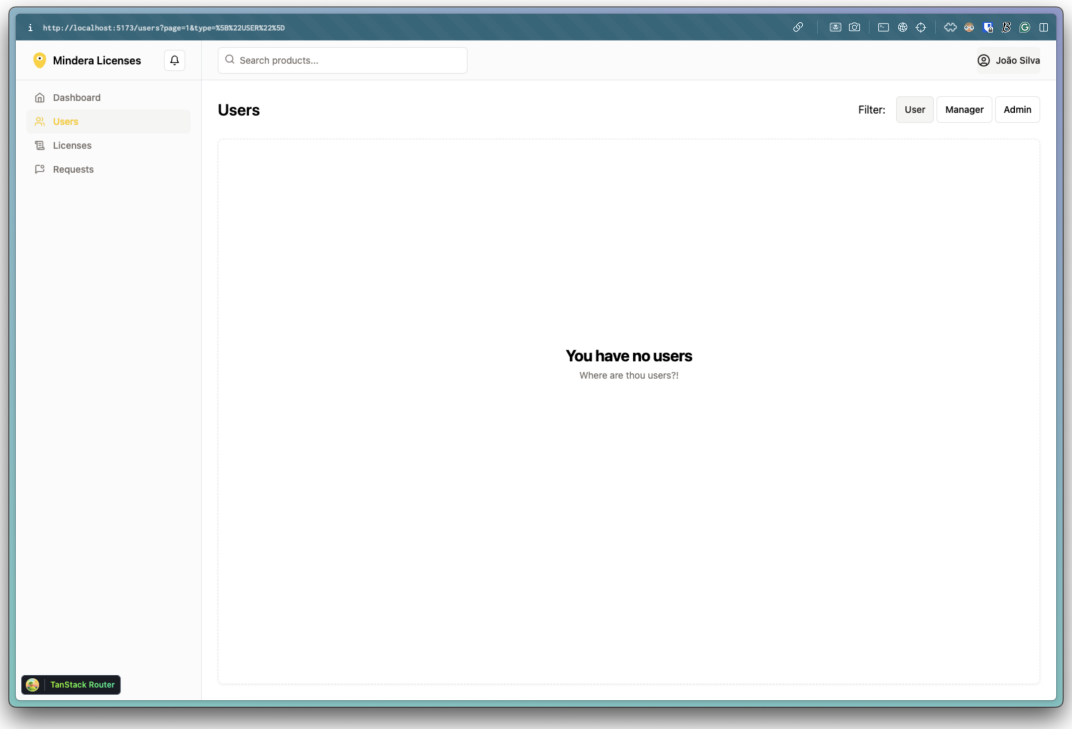

Figure 6.2: Table with a filter applied, showing there are no users with that type

In the table, each entry has a button to change the user's type, which, when clicked, pops up the modal in figure 6.3. Here, the user can select one of the types in the drop-down presented and, once the submit button is clicked, the data is updated.

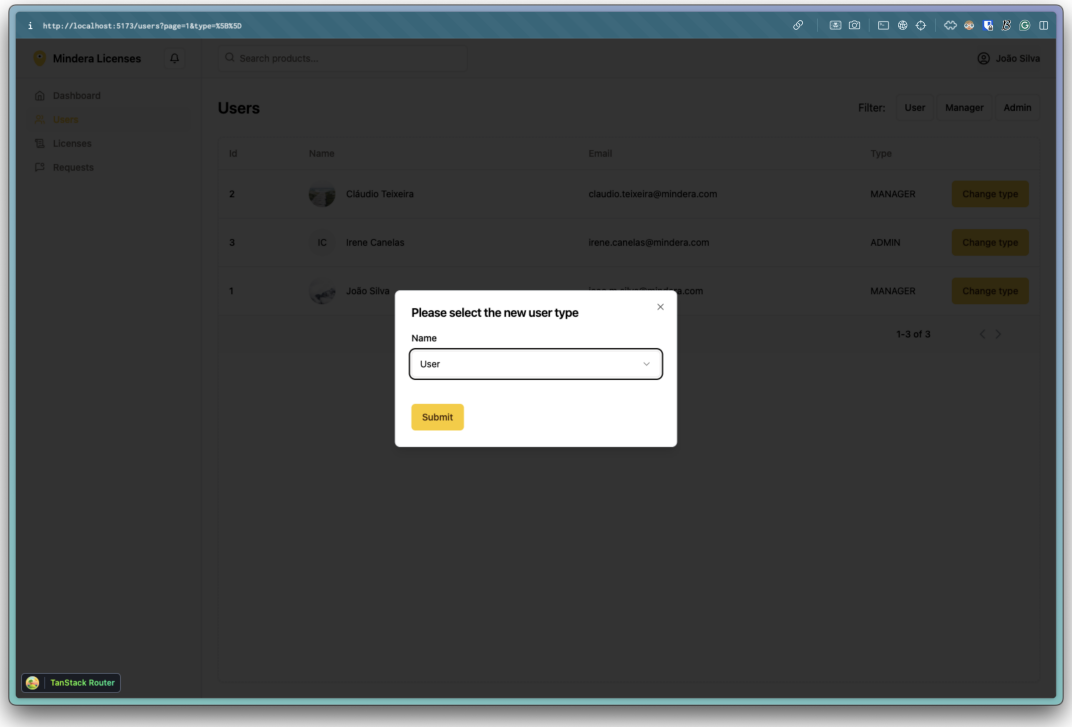

Figure 6.3: Modal displaying a drop-down with the various user types

Next, we have the page showing the licenses' information in figure 6.4. In this table, apart from the data, there are two different components: the first one has the purpose of stacking the license managers' pictures, to save on screen space; the second one is a button that takes the user to a specific license's page.

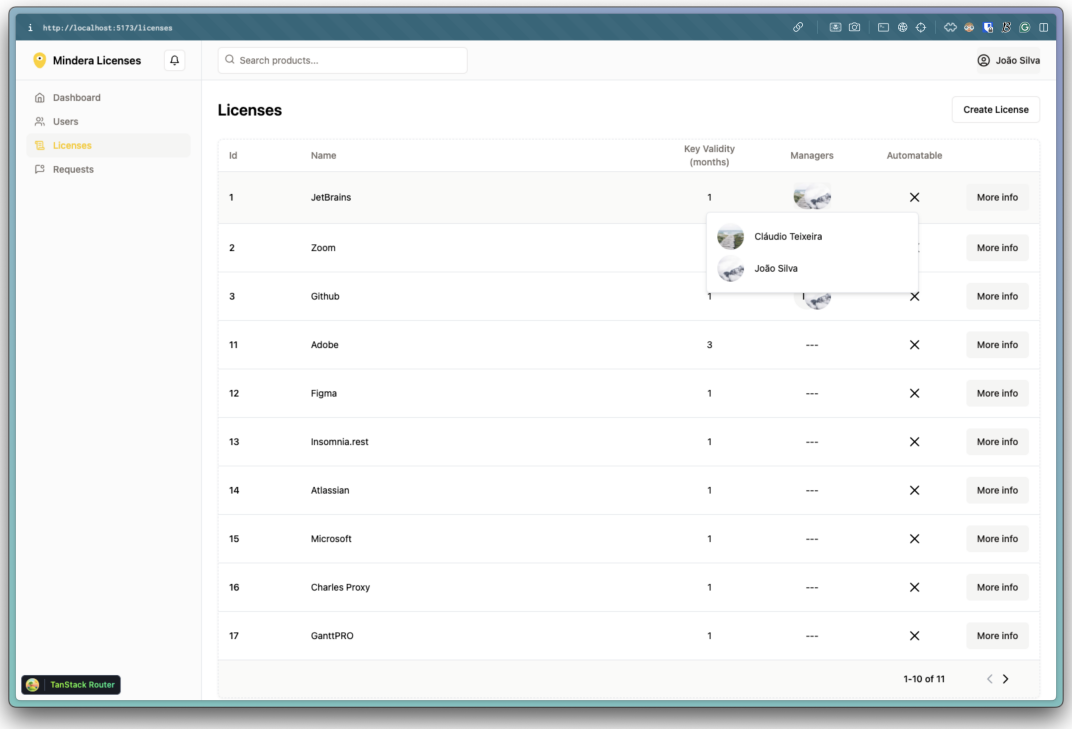

Figure 6.4: Table with the licenses' data, with the assigned managers' information expanded

On the same page as previously, there is also a button to create a license, which, when clicked, shows the modal in figure 6.5. This modal presents the user with a form, in which they can insert the information they wish to use to create a new license.

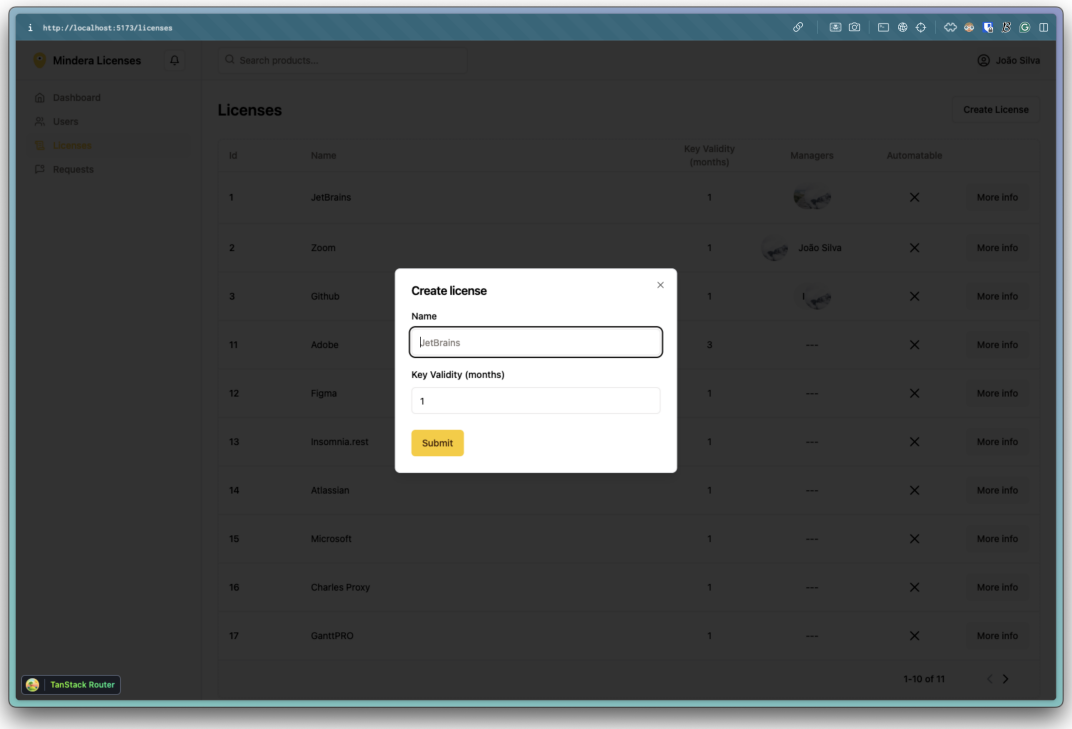

Figure 6.5: Modal with a form to create a license

Once the user visits a specific license's page in figure 6.6, he is greeted with the license's information, as well as three tabs at the bottom of the page. The first one, regarding the license's products, can be seen in figure 6.6 and displays the same information as the table in figure 6.4.

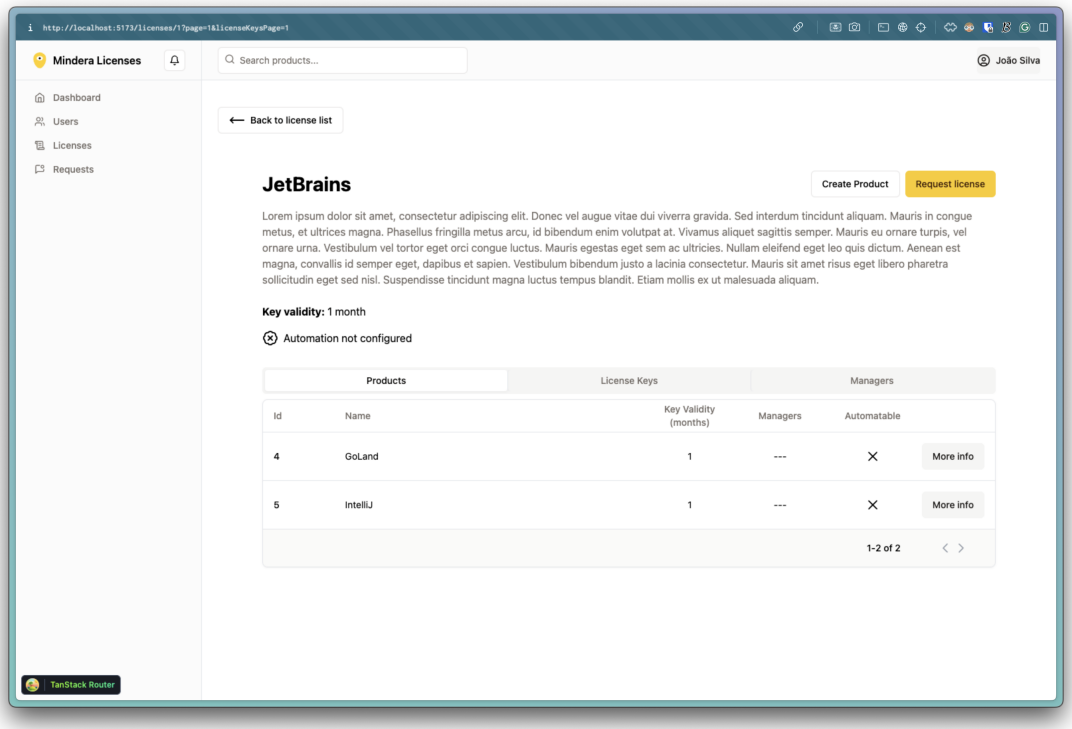

Figure 6.6: Page with a specific license's information, with the products tab expanded

Figure 6.7 shows the license keys tab as active. In this tab, it is possible to see every license key that belongs to the license being inspected, as well as two buttons, one to assign a key to a user, shown in figure 6.8, and another to remove the key from a user, shown in figure 6.9

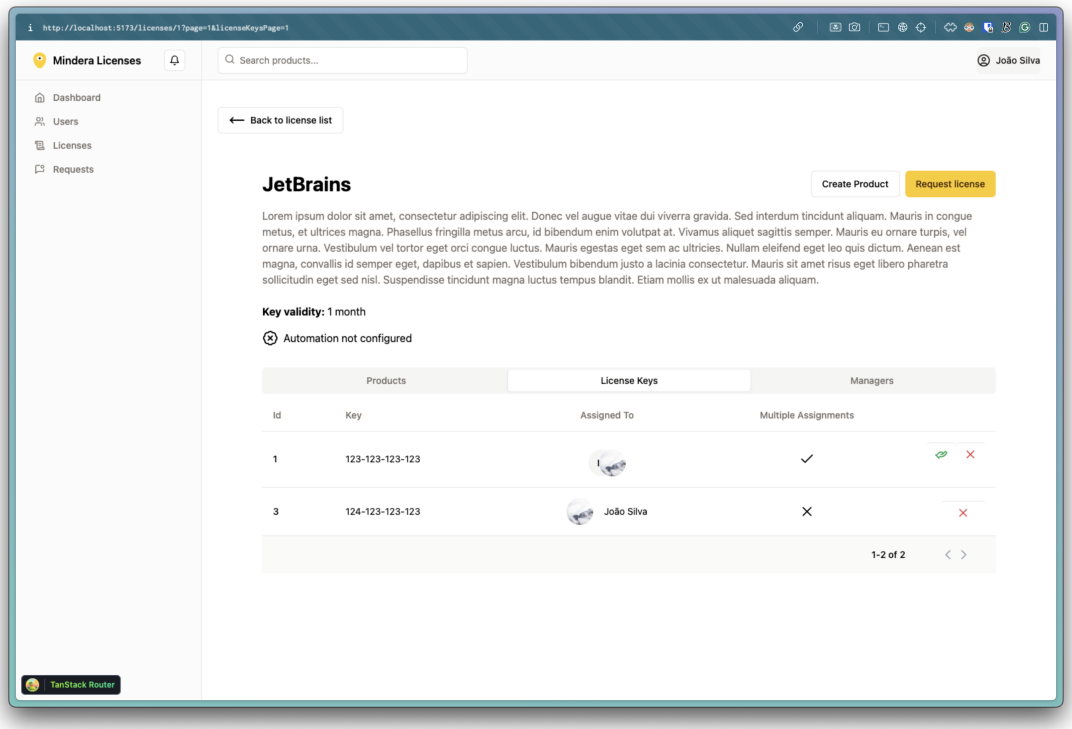

Figure 6.7: Page with a specific license's information, with the license keys tab expanded

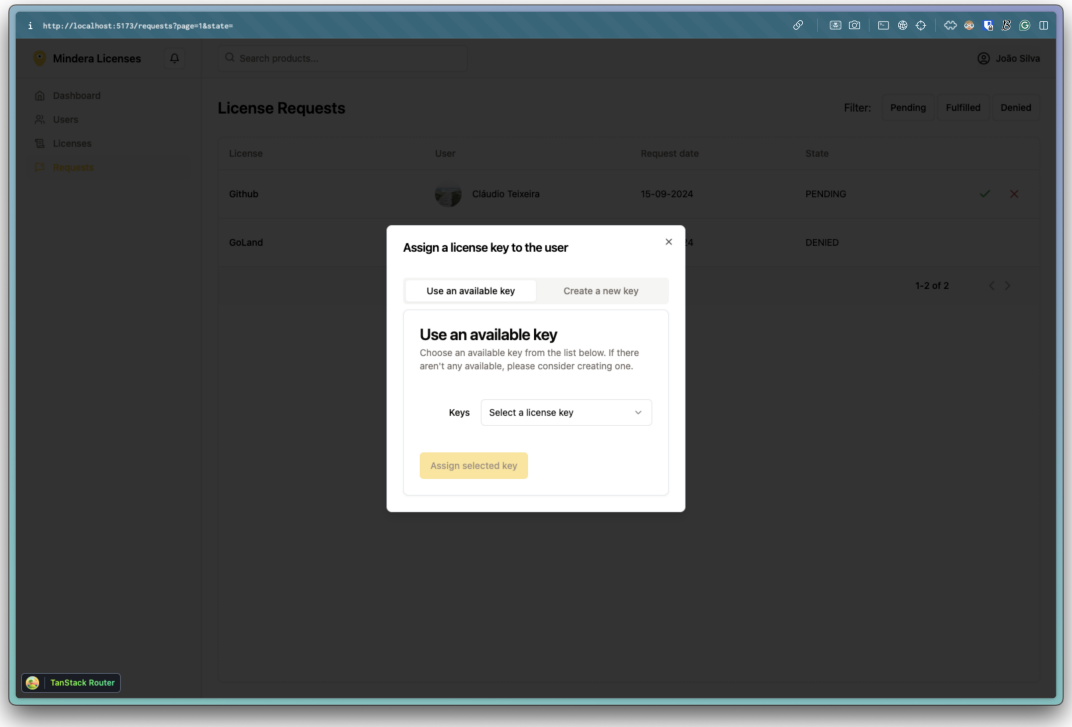

Figure 6.8: Modal to assign a key to the user, with the "Use an available key" tab open

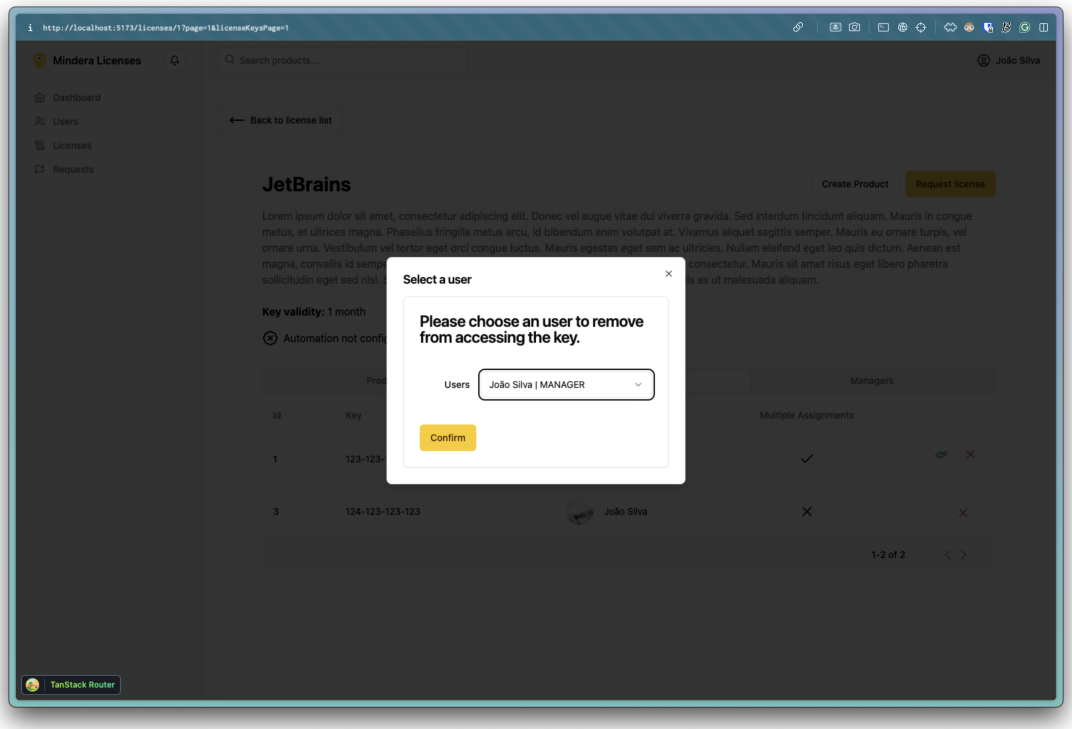

Figure 6.9: Modal showing a drop-down with the users that currently have the key assigned to them

On the managers tab, shown in figure 6.10, it is possible to see who are the managers that are assigned to the license, as well as remove them and assign another user to also be the license manager.

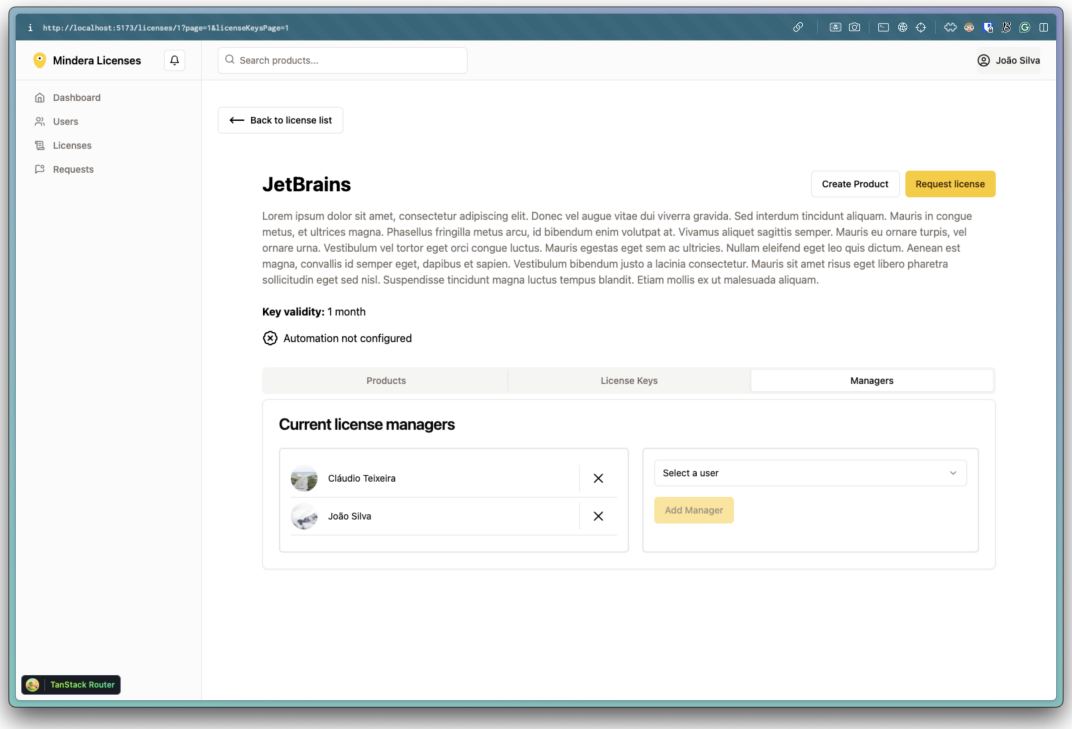

Figure 6.10: Page with a specific license's information, with the managers tab expanded

On a specific license page, a button to request a license is present at all times. When clicked, this button shows the modal in figure 6.11, to ensure the user really wants to perform the action.

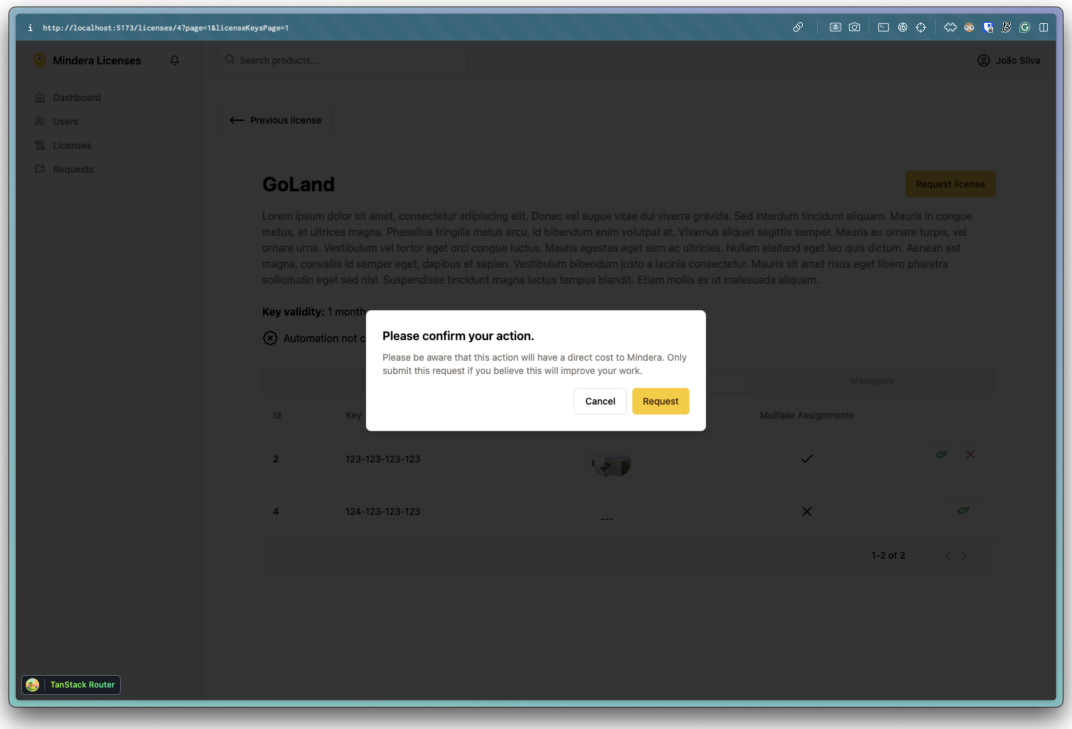

Figure 6.11: Modal shown when a user clicks the request license button

Figure 6.12 inspects a product, or, in other words, the child of a license. This has some different behaviors, as the only tab available is the license keys tab, and here it is also not possible to create a product.

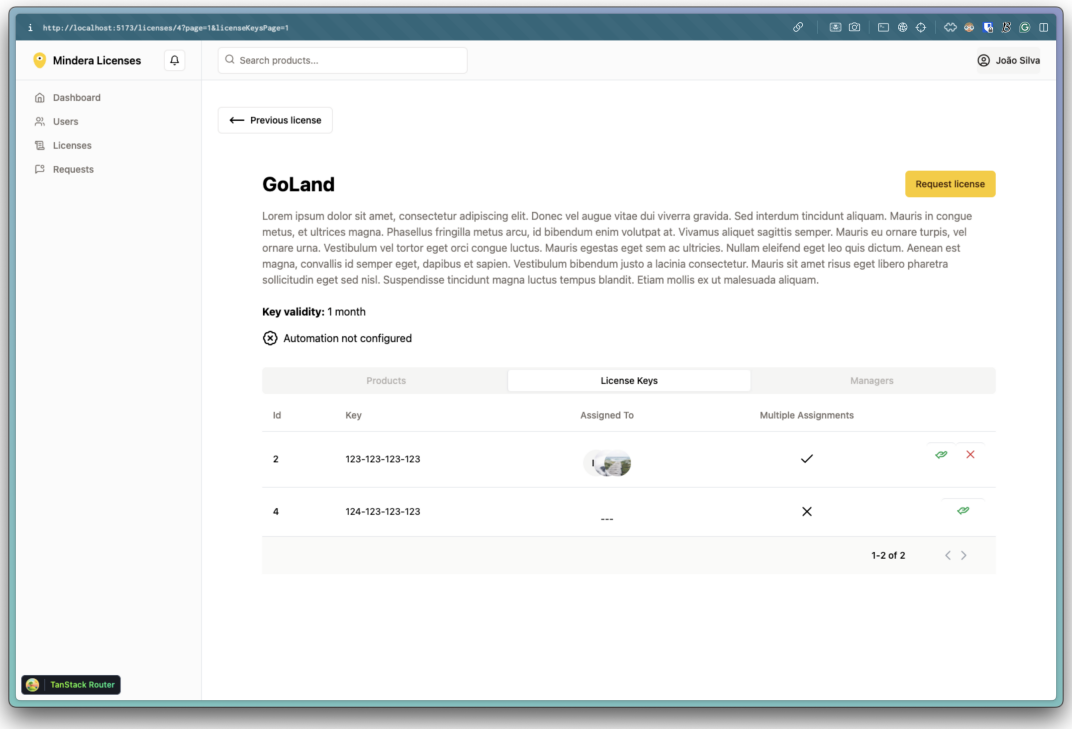

Figure 6.12: The page for a specific license, in the case a product, with the license keys tab as the only active tab

The license requests' data is presented in the table in figure 6.13. Each entry in the table, if it is in the pending state, displays two buttons, one to approve the request, and the other to deny. For the first case it gives the user two options: to assign a key, shown in figure 6.8, or to create a new key and assign it to the user, shown in figure 6.14. In the eventuality the user clicks the deny button, the modal in 6.15 is shown, which requires users to confirm their actions.

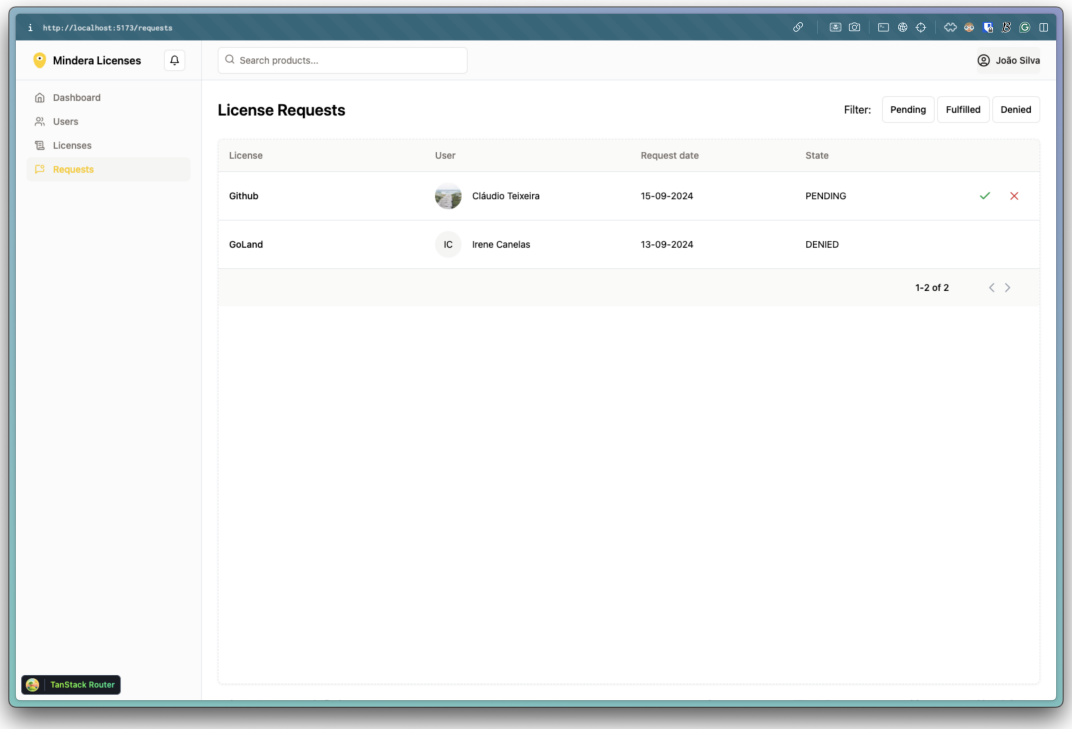

Figure 6.13: Page with a table showing all the license requests' data

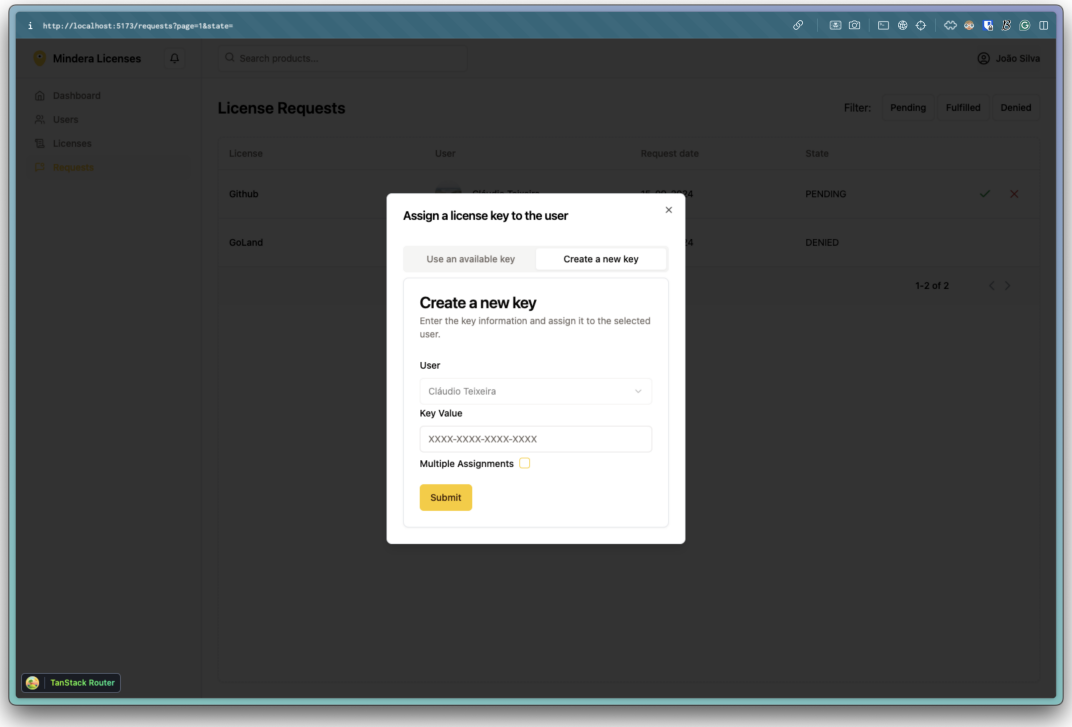

Figure 6.14: Modal to assign a key to the user, with the "Create a new key" tab open

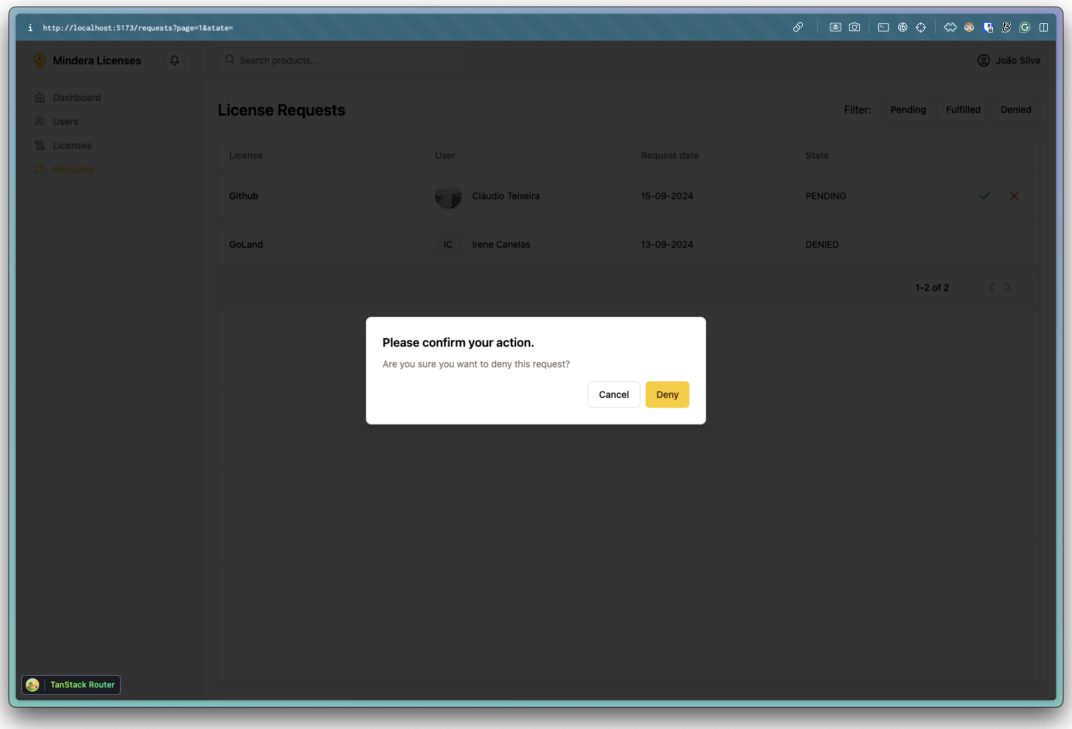

Figure 6.15: Modal shown when the license manager clicks the deny button

6.3 Non-Functional Requirements Review

This section will go through non-functional requirements that weren't covered during chapter 5, and verify if they were achieved by providing the needed evidences.

6.3.1 NFR3 - The user interface should be responsive

This non-functional requirement can be quickly validated by using Google Chrome's Dev-Tools, which loads a website in the wanted dimensions. After doing this, it is possible to evaluate what a user would see in a mobile phone, for example.

Figure 6.16 and 6.17 show that, when accessed through devices with dimensions smaller than a computer, the frontend remains responsive and accessible for the users.

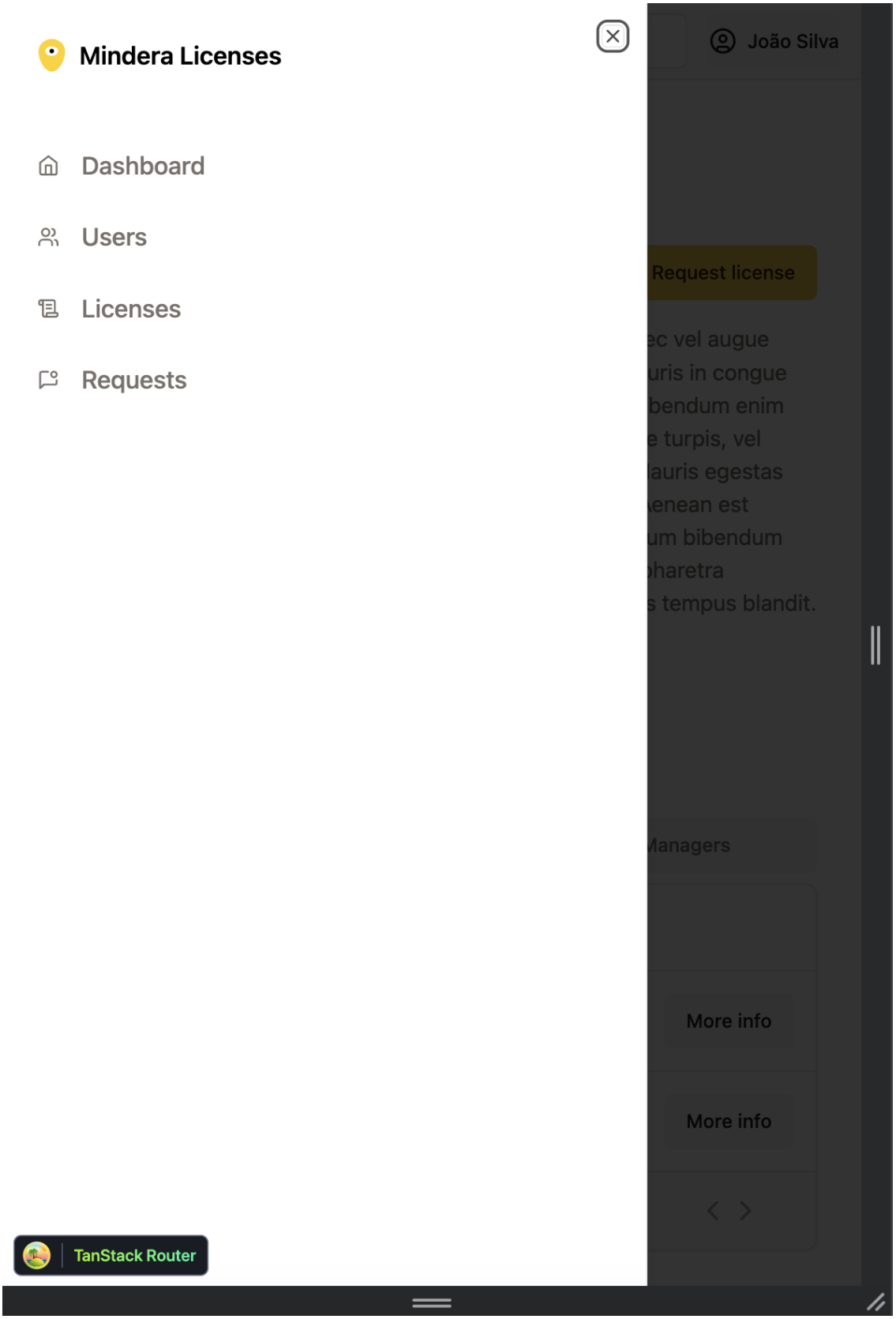

Figure 6.16: Navigation bar being loaded on the dimensions of a mobile phone

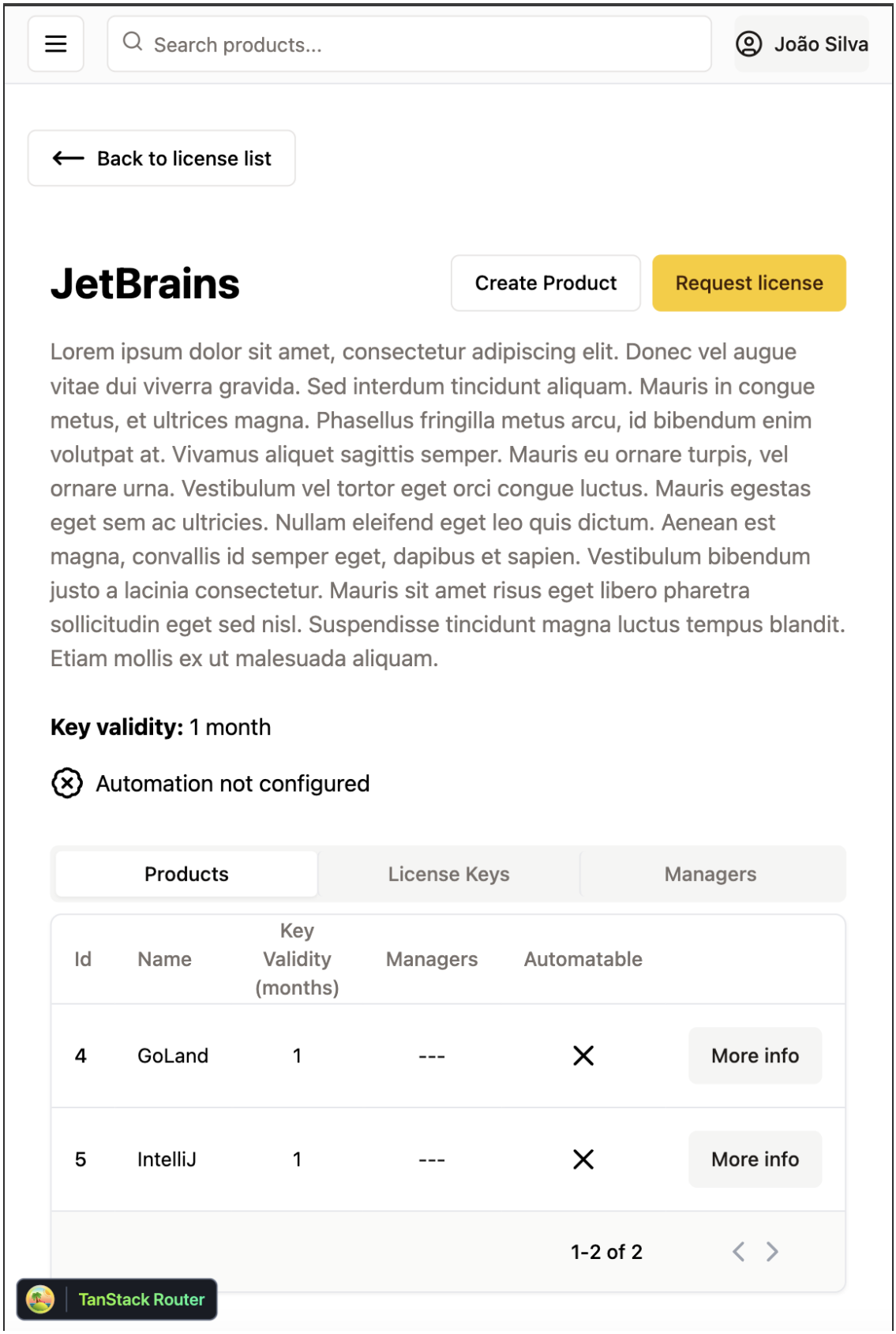

Figure 6.17: Specific license page being loaded on the dimensions of a mobile phone

6.3.2 NFR4 - Backend requests should take less than 2 seconds

During development, all of the routes that were defined had the thought of being as optimized as possible, which originated the pagination features in each of the pages present in the fronted. Because of this, the most resource-heavy request will be, most likely, the request to get the paginated licenses in the system, due to the amount of subqueries being used. Because of this, this specific endpoint will be performance tested, to see how the API handles while being on load.

To perform the performance test, JMeter was used. In figure 6.18, the route that was going to be tested was defined. To run the tests, 250 threads were used to send 50 requests each, totaling 12500 requests. The results can be seen in figure 6.19 and these show that the endpoint has a throughput of around 850 requests per second, with an average response time of 287ms. Due to this evidence, it is safe to say that the endpoints defined will have a latency of less than 2 seconds.

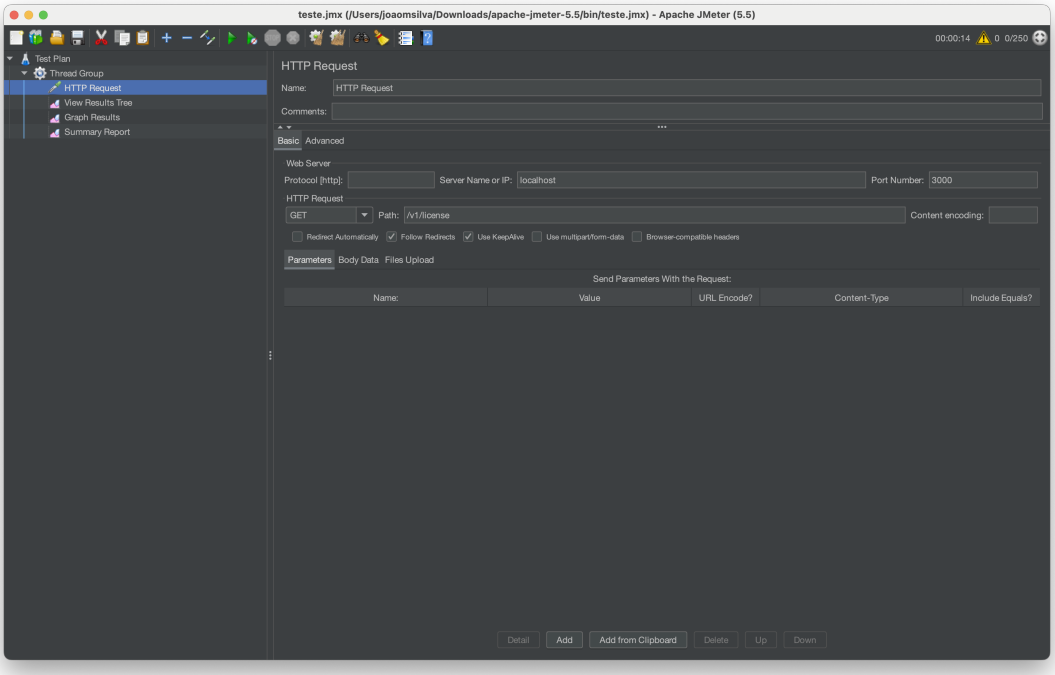

Figure 6.18: JMeter HTTP Request definition

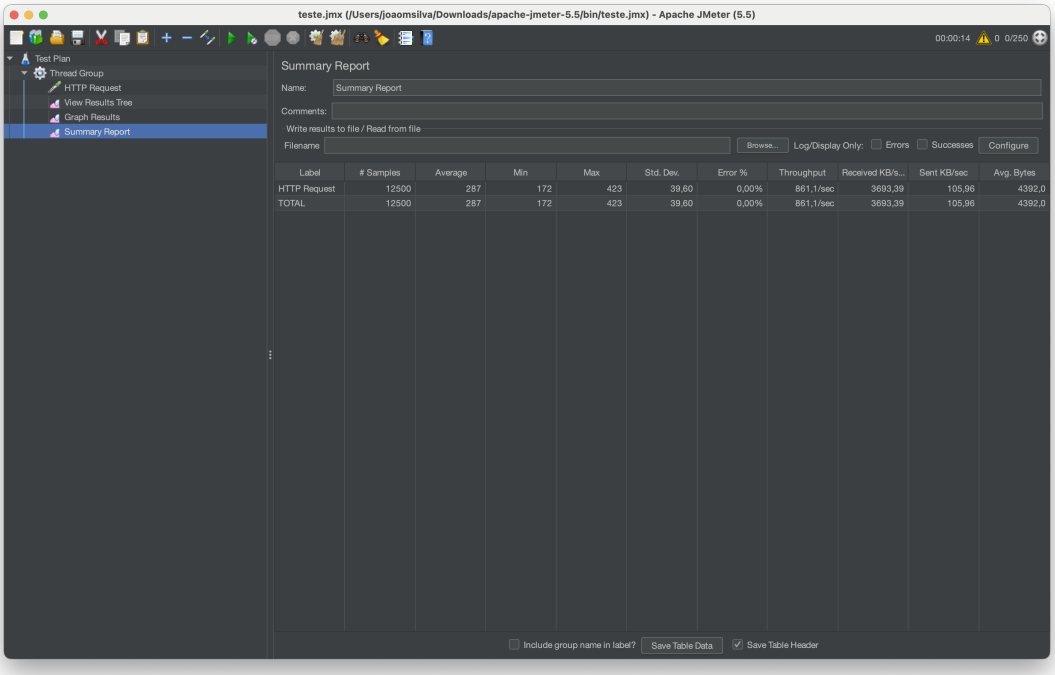

Figure 6.19: JMeter performance test results

Chapter 7

Conclusion

This chapter will summarize the project, comparing what was achieved with its expectations. It also will critique the work that was implemented, showcasing the limitations encountered and possible improvements for this system. Lastly, some final considerations by the author will be presented.

7.1 Achieved Objectives

This document started with providing context to the problem and defining some objectives to tackle it, leaving the author with a single research question: "What is needed to build a system to manage software licenses and what is the best way to integrate it within a software development company?". Chapter 2 addressed this question, which guided the analysis and design process in chapters 3 and 4, respectively. Chapter 5 dived deep into the implementation of most of the functional requirements, despite not being able to fully complete these, and non-functional that were defined in earlier stages of this paper. Chapter 6 provided a better grasp of what was actually achieved, addressing some non-functional requirements that weren't shown in earlier stages of the paper.

All of this culminated in, unfortunately, not all of the project's objectives being met. These were already mentioned previously, and are the dynamic rules, the communication with external providers, and the full automation of the license assignment and removal process. This occurred mainly due to time constraints while writing this paper, and should be addressed in the future to improve this system even further.

7.2 Future Work

Other than the functional requirements that weren't achieved, the system could benefit from some additional features that would elevate this work even further. Firstly, integrating the frontend with Google authentication is a must if this system is ever going to be handed to users, as they need to authenticate with their own accounts. Then, this could be turned into a Business to Business (B2B) solution, with the changes that would need to happen to be the addition of one concept, the workspace. With this, information could be separated into different workspaces (where each company is assigned one), and the different workspaces' data could possibly be separated into different databases, ensuring that critical information isn't leaked to other companies by mistake.

Lastly, despite the author being pleased with the UI that was elaborated, it could become slightly better and more user friendly with the elaboration of a design for every feature in the application, to provide consistent pages and components throughout the client-side.

7.3 Final consideration

This project proved to be a valuable learning experience, mainly due to the humbling effect that not being able to completely deliver it gave the author. Despite being a great idea with a lot of room to grow, it wasn't possible to deliver it in it's full glory.

This project, however, also gave an immeasurable amount of experience to the author, from the planning to the design phases, trying to sketch the best possible version of this system, to the implementation and testing of a full-blown fullstack application gave the author a very large room to experiment with technologies and tools that were less than a year old at the time of this paper.

Bibliography

[1] *Market Guide for Software Asset Management Tools*. Tech. rep. [Online; accessed 25. Dec. 2023]. Oct. 2023. url: https://www.gartner.com/en/documents/4801731.

[2] Jan vom Brocke, Alan Hevner, and Alexander Maedche. *Introduction to Design Science Research*. Sept. 2020. isbn: 978-3-030-46780-7. doi: 10.1007/978-3-030-46781-4_1.

[3] Miriam Ballhausen. "Free and Open Source Software Licenses Explained". In: *Computer* 52.6 (June 2019), pp. 82–86. doi: 10.1109/MC.2019.2907766.

[4] Martin Jakubička. "Software asset management". In: *2010 IEEE International Conference on Software Maintenance*. 2010, pp. 1–2. doi: 10.1109/ICSM.2010.5609662.

[5] *ISO/IEC 19770-5:2015(en), Information technology IT asset management: Overview and vocabulary - Part 5*. [Online; accessed 29. Aug. 2024]. Aug. 2024. url: https://www.iso.org/obp/ui/en/#iso:std:iso-iec:19770:-5:ed-2:v1:en.

[6] *GPT-4*. [Online; accessed 27. Dec. 2023]. Dec. 2023. url: https://openai.com/gpt-4.

[7] *Google Identity | Google for Developers*. [Online; accessed 27. Dec. 2023]. Nov. 2023. url: https://developers.google.com/identity.

[8] *Snow License Manager*. [Online; accessed 2. Jan. 2024]. Nov. 2023. url: https://www.snowsoftware.com/products/snow-license-manager.

[9] *Software Asset Management (SAM) | Certero Software & Services*. [Online; accessed 2. Jan. 2024]. June 2023. url: https://www.certero.com/software-asset-management.

[10] *ServiceNow – The world works with ServiceNow™*. [Online; accessed 28. Dec. 2023]. Dec. 2023. url: https://www.servicenow.com.

[11] *IT and Cloud Management, Optimization and Solutions | Flexera*. [Online; accessed 2. Jan. 2024]. Dec. 2023. url: https://www.flexera.com.

[12] *Snipe-IT*. [Online; accessed 2. Jan. 2024]. Jan. 2024. url: https://snipeitapp.com.

[13] Muhammad Tukur, Sani Umar, and Jameleddine Hassine. "Requirement Engineering Challenges: A Systematic Mapping Study on the Academic and the Industrial Perspective". In: *Arab. J. Sci. Eng.* 46.4 (Apr. 2021), pp. 3723–3748. issn: 2191-4281. doi: 10.1007/s13369-020-05159-1.

[14] Mahrukh Umar and Naeem Ahmed Khan. "Analyzing Non-Functional Requirements (NFRs) for software development". In: *2011 IEEE 2nd International Conference on Software Engineering and Service Science*. IEEE, 2011, pp. 15–17. doi: 10.1109/ICSESS.2011.5982328.

[15] [Online; accessed 27. Aug. 2024]. Oct. 2014. url: https://www.imeko.org/publications/tc4-2014/IMEKO-TC4-2014-396.pdf.

[16] *Typescript - JavaScript With Syntax For Types*. [Online; accessed 10. Sep. 2024]. Sept. 2024. url: https://www.typescriptlang.org.

[17] *Bun - All in one Javascript runtime and toolkit*. [Online; accessed 10. Sep. 2024]. Sept. 2024. url: https://bun.sh.

[18] *Elysia - Ergonomic Framework for Humans*. [Online; accessed 10. Sep. 2024]. Sept. 2024. url: `https://elysiajs.com`.

[19] *Drizzle ORM - next gen TypeScript ORM.* [Online; accessed 10. Sep. 2024]. Sept. 2024. url: `https://orm.drizzle.team`.

[20] *Zod - TypeScript-first schema validation with static type inference*. [Online; accessed 12. Sep. 2024]. Sept. 2024. url: `https://zod.dev`.

[21] *React - Library for web user interfaces*. [Online; accessed 10. Sep. 2024]. Sept. 2024. url: `https://react.dev`.

[22] *Vite - Frontend Tooling*. [Online; accessed 10. Sep. 2024]. Sept. 2024. url: `https://vitejs.dev`.

[23] *Tailwind CSS - Rapidly build modern websites without ever leaving your HTML.* [Online; accessed 11. Sep. 2024]. Sept. 2024. url: `https://tailwindcss.com`.

[24] *TanStack Router*. [Online; accessed 14. Sep. 2024]. Sept. 2024. url: `https://tanstack.com/router/latest`.

[25] shadcn. *shadcn/ui - Component Library*. [Online; accessed 11. Sep. 2024]. Sept. 2024. url: `https://ui.shadcn.com`.