UNIVERSITAT POLITÈCNICA DE VALÈNCIA

UNIVERSITAT POLITÈCNICA DE VALÈNCIA
CAMPUS D'ALCOI

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Higher Polytechnic School of Alcoi

Asset management in fieldwork

End of Degree Project

Bachelor's Degree in Informatics Engineering

AUTHOR: Takamäki, Johannes Viljami

Tutor: Esparza Peidro, Javier

ACADEMIC YEAR: 2023/2024

Resumen

El objetivo del proyecto final es definir los procesos necesarios y avanzar en el proyecto para crear una solución que pueda procesar alertas del sistema de gestión de energía de un cliente, crear un ticket de trabajo y ofrecer a un trabajador de campo información detallada sobre el dispositivo y su historial de mantenimiento. Para la empresa de mantenimiento, la solución ofrece un CMDB para obtener datos del trabajo de campo a su servicio de gestión de activos existente. El proyecto es un proyecto de investigación y se está desarrollando como prueba de concepto.

El trabajo involucra el desarrollo de portlets de Liferay y una base de datos PostgreSQL, utilizando React, TypeScript y Java EE.

La solución se está desarrollando continuamente para lograr una funcionalidad mejorada. El trabajo futuro se centrará en refinar la escalabilidad, la seguridad y la experiencia del usuario del sistema.

Palabras clave: Liferay, Java EE, React, API, Portlet, Servicio, Full stack, CMDB

Resum

L'objectiu del projecte final és definir els processos necessaris i avançar en el projecte per crear una solució que pugui processar alertes del sistema de gestió d'energia d'un client, crear un tiquet de treball i oferir a un treballador de camp informació detallada sobre el dispositiu i el seu historial de manteniment. Per a l'empresa de manteniment, la solució ofereix un CMDB per obtenir dades del treball de camp al seu servei de gestió d'actius existent. El projecte és un projecte de recerca i s'està desenvolupant com a prova de concepte.

El treball implica el desenvolupament de portlets de Liferay i una base de dades PostgreSQL, utilitzant React, TypeScript i Java EE.

La solució s'està desenvolupant contínuament per aconseguir una funcionalitat millorada. El treball futur se centrarà en refinar l'escalabilitat, la seguretat i l'experiència de l'usuari del sistema.

Paraules clau: Liferay, Java EE, React, API, Portlet, Servei, Full stack, CMDB

Abstract

The goal of the final project is to define the necessary processes and advance the project to create a solution that can process alerts from a client’s energy management system create a work ticket and offer a field worker detailed information about the device and its maintenance history. For the maintenance company, the solution offers a CMDB solution to get data from the fieldwork to their existing asset management service. The project is a research project and is being developed as proof of concept.

The work involves developing Liferay portlets and a PostgreSQL database, utilizing React, TypeScript, and Java EE.

The solution is being continuously developed to achieve improved functionality. Future work will focus on refining the system's scalability, security, and user experience.

Keywords: Liferay, Java EE, React, API, Portlet, Service, Full stack, CMDB

LIST OF FIGURES

Abbreviations

CMDB	Configuration management database
SmartE3	Smart edge-computing based building energy management enabling electric transportation
EMS	Energy Management System
DTO	Data Transfer Object
DAO	Data Access Object
JPA	Java Persistence API

1 Introduction

This chapter serves as the foundation for understanding the scope, objectives, and background of the project. It is divided into six subsections that provide a comprehensive overview of the motivation behind the project, the specific components and participants involved, the market research conducted, the objectives to be achieved, and the overall structure of the report.

1.1 Motivation

In an industrial setting, it's common for infrastructural building and maintenance, that devices and structural elements of the infrastructure are being deployed geographically in large areas. These devices have an owner, but most commonly, multiple entities are working with these devices. These entities might include the manufacturer of the device, the end-user or the owner, a service provider like an operator, and the companies providing installation and maintenance operations.

This final project might be of interest to companies working with Liferay Portal or considering Liferay for their company's needs. With this project, I'm covering portlet development with a full-stack approach, thus covering various parts of the portal development process. It needs to be said that there is a substantial amount of existing in-house developed software used, which may differ from existing specifications and frameworks.

1.2 SmartE3

SmartE3 or Smart edge-computing based building energy management enabling electric transportation is one of Nokia's many Veturi program research projects. Veturi projects aim to build sustainable solutions that leverage edge computing and edge intelligence to manage ever-growing data transfers and network capacity requirements. With 70 million EUR funding from Business Finland and the EU's Recovery and Resilience Facility (RRF) (Business Finland, 2022), the Nokia Veturi program: Competitive Edge, started in January 2022 and is ending in December 2024 (Nokia, n.d.).

The objective of SmartE3 is to develop technologies for controlling the energy usage of buildings by introducing an edge computing solution. One of the main goals of the project is to create a technology-neutral and replicable concept that can be deployed to any property in Finland or abroad. The project was kicked off at the end of the year 2022 and was originally scheduled to end in the fall of 2024 but was extended for the spring of 2024 for another six months.

1.3 Participants

SmartE3 is a collaboration of LUT University, LSK Group, and Rossum Oy, where LUT University provides an edge computing solution, data is gathered from LSK Group's contract property and Rossum Oy provides a service system that acts as a fail-safe, where if the automation fails, Rossum operation management service will send a human to tend the problem, which in place generates new data for the automation system.

1.3.1 LUT

LUT University or Lappeenranta-Lahti University of Technology, is a Finnish public science university in southern Finland, with campuses in Lappeenranta and Lahti. Founded in 1969, LUT University has a broad international impact as an international campus, consisting of 9000 members of more than 90 different nationalities (LUT University, n.d.-a). LUT University's strategy 2030: Trailblazers, focuses on four different themes: clean energy, water, air, and business and society (LUT University, n.d.-b). With these goals in mind, LUT University is dedicated to achieving carbon neutrality, by advancing the UN's Sustainable Development Goals (SDGs). As such, this project is right at the core of LUT's strategy.

LUT oversees the development of a control algorithm that can identify factors influencing the conditions and energy usage, thus enabling machine learning to be used in energy management for the LSK's test property.

1.3.2 LSK Group

LSK Group is a specialist in building and industrial technology. Operating throughout Finland, LSK's long expertise in the field of building and industrial technology. With the determination to develop energy-efficient smart solutions for the present and the future, LSK Technology was awarded an ISO 9001 quality management certification. (LSK Group, n.d.)

With an extensive list of contract properties, LSK provided a test property for the SmartE3 project which had an existing infrastructure from real-world test data that could be gathered.

1.3.3 Rossum Oy

Rossum is a Finnish software company that specializes in delivery processes involving fieldwork and the communication of work-related information. Founded in 1994, they provide tailored ERP or Enterprise Resource Planning solutions for companies, allowing them to select and configure the modules, thus addressing that company's particular needs. (Rossum Oy, n.d.-a)

Rossum’s main product is an ERP system called Rossum Works. It is built on Liferay Portal, which is an open-source web-based application framework. Rossum Works is designed to allow companies to assign and track work progress and status information in real time (Rossum Oy, n.d.-e).

Toolkit, as the name suggests, is a collection of different tools that aims to bring to bring together the tools needed for work in the field in a single interface. Toolkit is meant to work alongside fieldwork management systems and can be integrated into an existing ERP system, complementing the workflow of fieldworkers seamlessly. For the company, the toolkit offers a service that reaches the subcontractor's fieldworkers without data integrations, this has been an essential need with telecommunicators, as in their business, work is done by a subcontractor/contractor. From the many benefits to the fieldworker, the toolkit offers easy access to all the tools with a single sign-up, where all that is required is to log in to the company email. (Rossum Oy, n.d.-b, n.d.-c)

1.4 Market research

There isn't a comparable product for Rossum's asset management service as it targets a particular industry and provides solutions for extremely specific tasks. An asset management service itself is not that uncommon for companies, but a service where an asset management service is combined with field management for companies and subcontractors is aiming for a new niche market that has no real competition in Finland.

1.5 Objectives

The final project is about working with Finnish software company Rossum Oy, on a large-scale collaboration project called SmartE3. Rossum aims to create an asset management service or a CMDB, which for Rossum is meant to be a standalone module from the company’s existing ERP product but offers the possibility to work in conjunction with the existing system. For SmartE3, Rossum offers a solution where if the automation fails or a device malfunctions, a work ticket will be created and assigned to a company or a subcontractor, and ultimately a worker will be automatically sent to the location. The sent worker can then use the developed CMDB or Configuration Management Database to gain helpful information about the malfunctioned device.

As for clear objectives for the final project, the aim is to advance the development of the CMDB or Configuration Management Database, which has the functionality to store the

data that has been generated by the property's energy management system into a property's own asset management database.

Due to the time allocated to the research project SmartE3, the final project will not be the finished version of the software solution developed by Rossum but will act as a proof of concept. The goal of the final project is to identify potential challenges and areas for improvement, advance the development, and help define necessary functionalities and technologies to achieve a possible solution.

1.6 Structure

The final project report is divided into six main chapters. Each chapter is then divided into its own subsections to improve the chapter's general understanding. Below is a brief overview of each chapter.

Chapter 1: Introduction

This chapter provides an introduction to the project, detailing the motivation behind the work, the background of the project, the conducted market research, and the objectives of the project. It sets the settings for the subsequent chapters.

Chapter 2: Analysis

The second chapter gives an initial analysis of the project's required functionalities and challenges that need to be resolved. As I started working on the project midway through the development, analysis includes the starting point of the technical aspect.

Chapter 3: Design

This chapter provides a detailed overview of the project structure and its components. It is divided into three main sections: the first part gives insight into how the project's different components are connected to each other; the second part lists the technologies and tools used and the third part gives details of the various layers of the project.

Chapter 4: Results

This chapter presents the results of the final project, using a real-world alert and presenting it as a tour of the application.

Chapter 5: Conclusion

The final chapter provides a conclusion, summarizing the key findings and implications of the project. The chapter also outlines potential and needed developments and enhancements that could be made based on the project's stage.

2 Analysis

This chapter gives an initial analysis of the project and gives throughout insight into what was already in place at the start of the project.

2.1 Starting point

As I began working on the SmartE3 project, some specifications were already in place, but as the project developed, initial goals for the SmartE3 project got more focused, yet Rossum's goals remained largely unchanged from the initial specifications. Develop a solution that can read alerts and based on findings on maintenance needs, from the received alert a work ticket is created and sent to the manager of the property concerned or to the maintenance company of their choice. The conceptual outline is illustrated in Figure 2. In addition to the creation of these work tickets, the aim is to find a solution for giving the instructions and other necessary documents related to the maintenance task to the maintenance worker. These goals can be divided into two main parts: creating a service ticket in Rossum Works and developing a CMDB service called Device Registry.

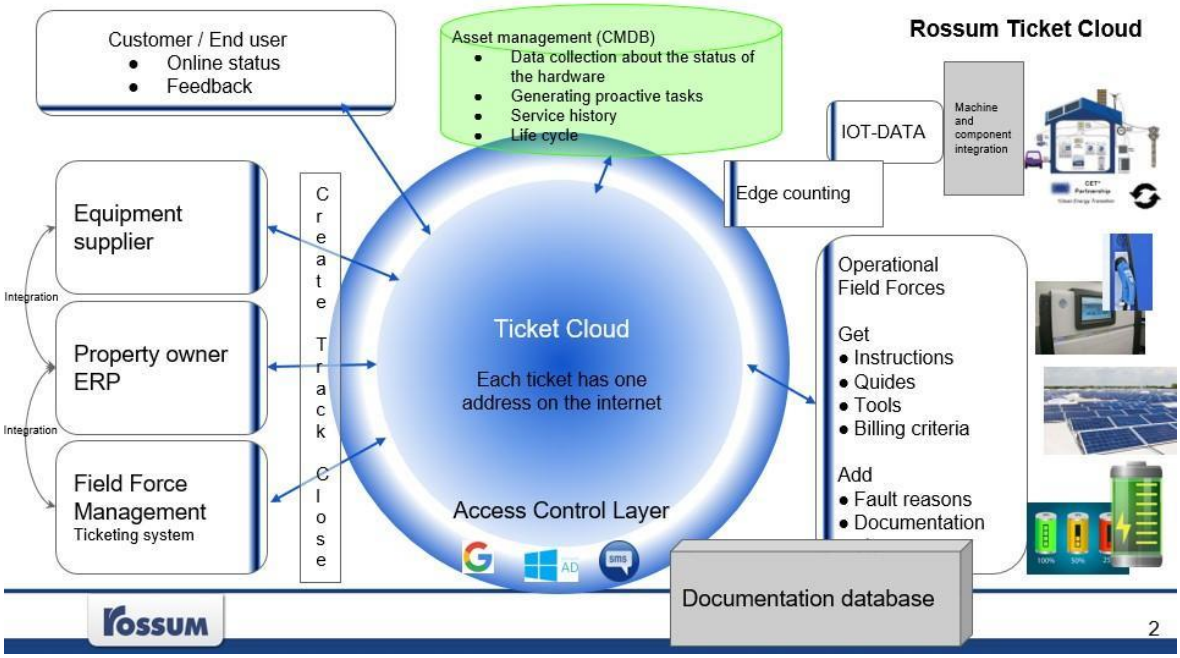


Figure 1: An outline of overall Rossum Ticket Cloud concept.

(Rossum Oy, n.d.-d)

2.1.1 Service Ticket to Rossum Works

The first goal involves creating the needed functionalities that have the capability to read alerts of faulty machinery or devices from the client's energy management system or EMS,

create tickets in Rossum's ticketing system called Rossum Works, direct maintenance and repair tasks created based on system observations to the desired company or individual and notify the client if the read operation was successful or not.

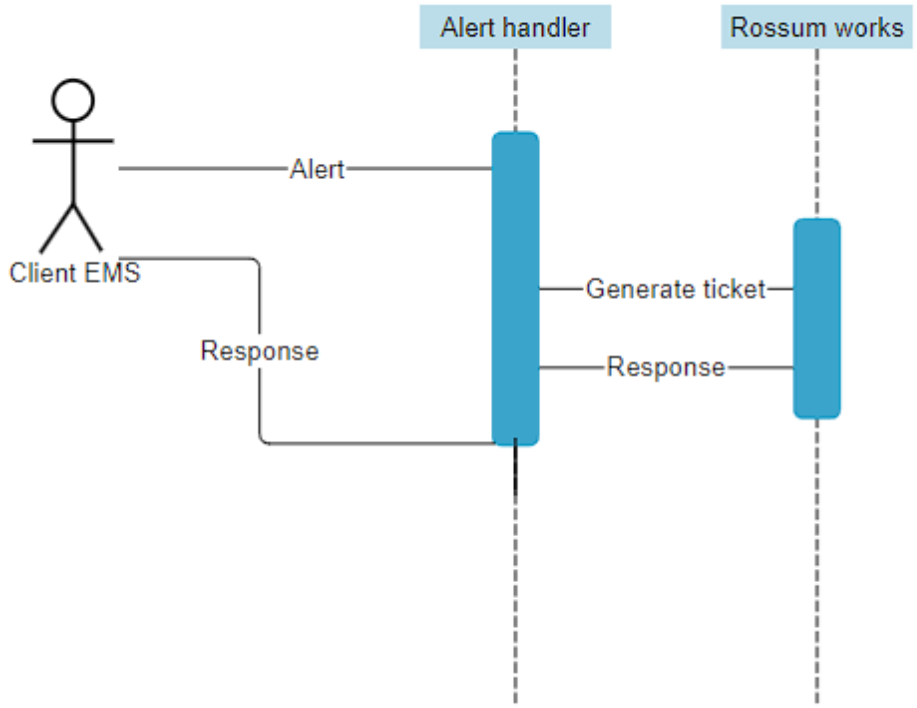


Figure 2: Alert reading

2.1.2 Device Registry (CMDB)

Secondly, the creation of a CMDB service called Device registry, which can be deployed independently from Rossum's existing products, but have the capability to work alongside them, complementing each other. The Device Registry should incorporate GUI or Graphical User Interface made with React, store basic data from the devices and their components, devices' service history, and offer an API from which an entity, like the property owner, can add, update, or retrieve data, based on that their privileges.

2.2 Technical starting point

From the start, there had already been some work done in development. There was only a mostly empty Liferay Portlet module called Device Registry. This portlet had been configured with Rossum's own template of front-end consisting of React with TypeScript. Back-

end with Java running with Spring Framework and a PostgreSQL database. All this running in a docker containers (Figure 3).

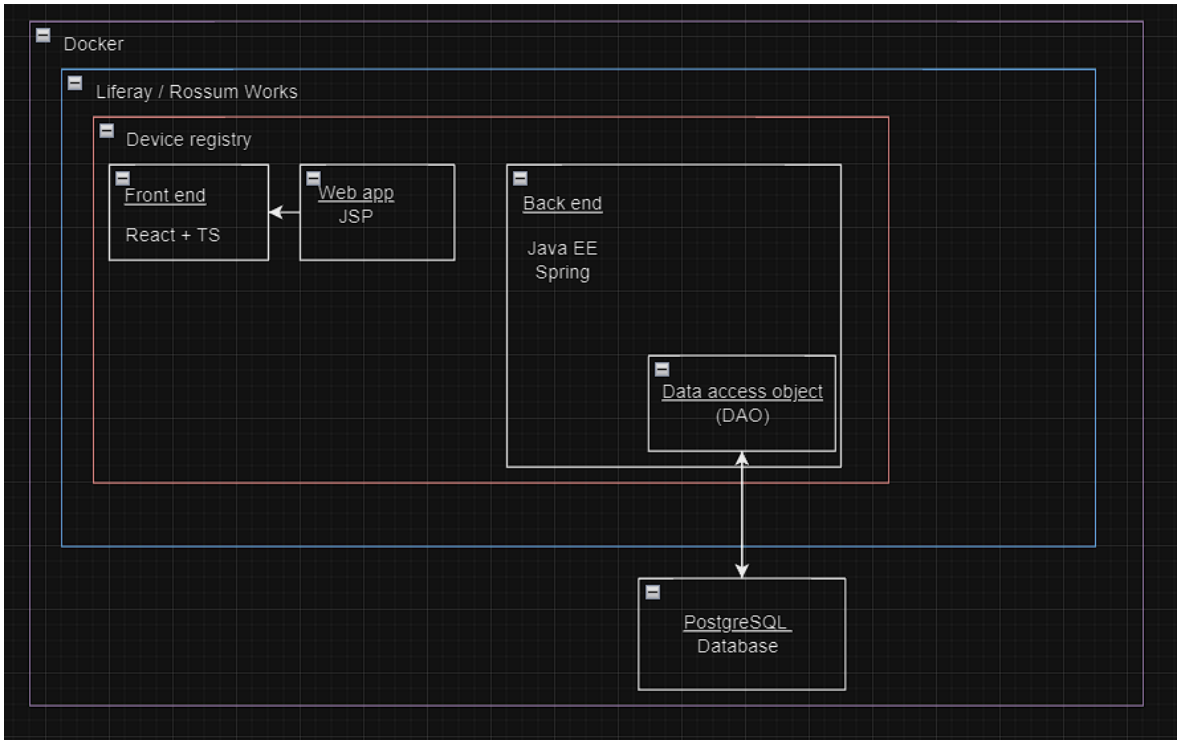


Figure 3: Project Start

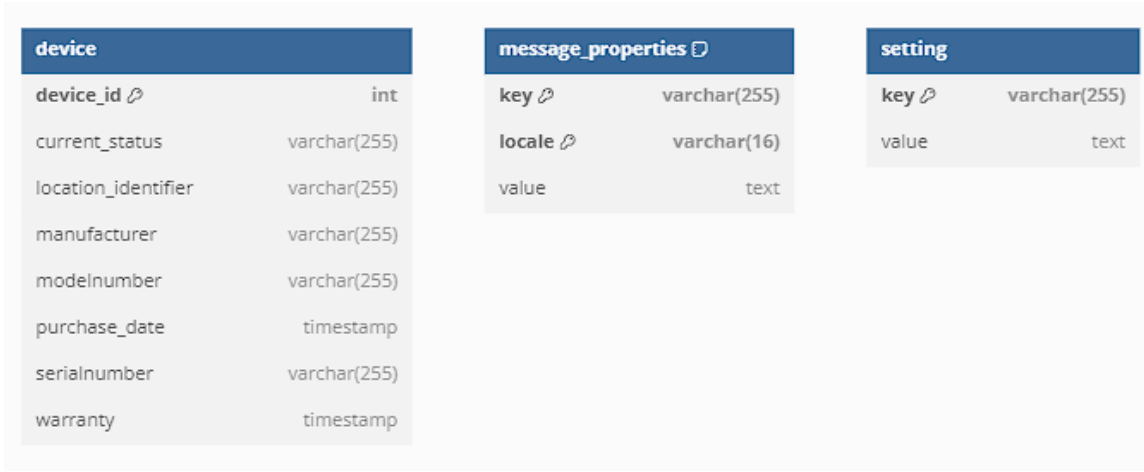


Figure 4: Database structure

The project had a simple GUI which had existing card components made for the different devices, but the data was dummy data, hard coded in the front end. At this point, there was no functionality to make a call to the back end to retrieve the data from the database and send it to the front end.

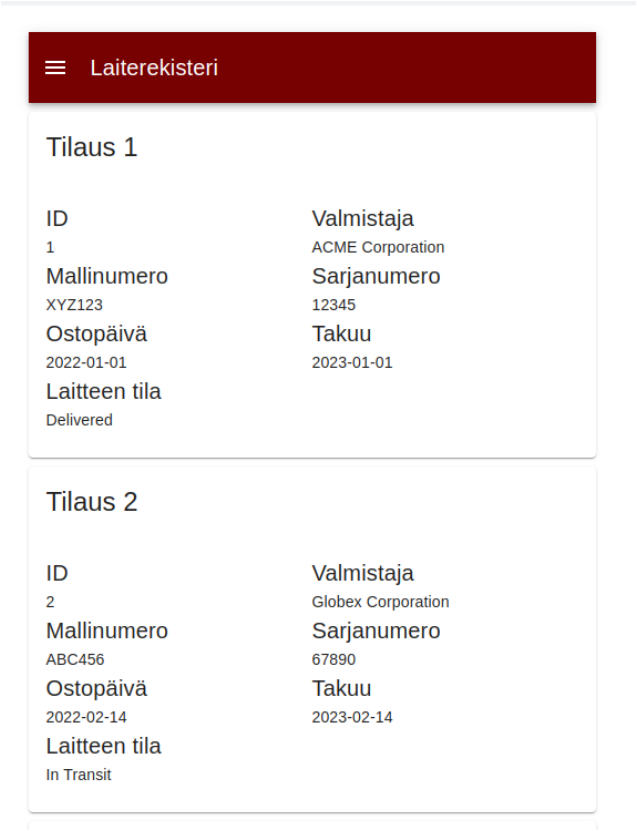


Figure 5. Starting point, GUI.

3 Design

This chapter explains the structure of the project and provides a detailed overview of the work done. It is divided into three main parts. Part, one explains how the different components of the project are connected. Part two provides insight into the various technologies and tools used to achieve the results. Part three describes the different layers or components of the project: Front end, Back end, Database solution, API, and Email reading functionality.

3.1 Application architecture

From the start, there was a clear structure in place for communication between layers, but they were not finalized. The front end was missing a way to communicate with the back end. As the project advanced and new functionalities were introduced with new components like APIs or data required to be retrieved from other portlets like Rossum Toolkit, a more structured approach for communication between layers became increasingly crucial.

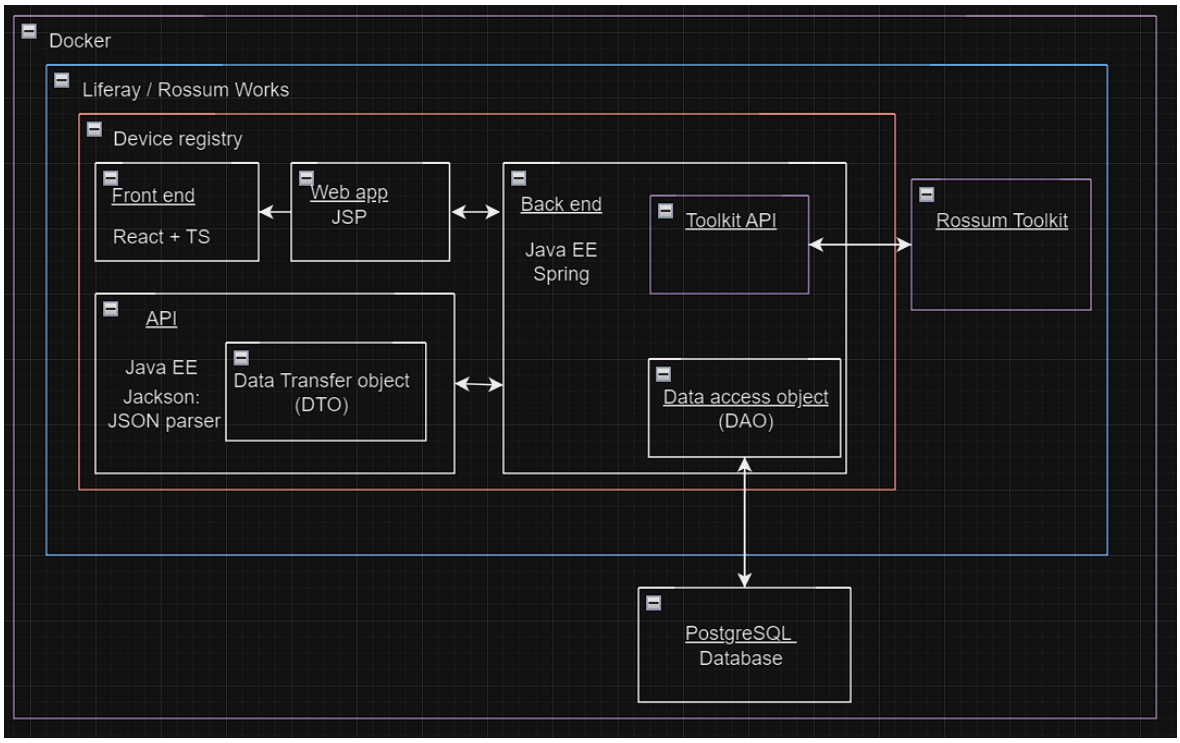


Figure 6: Project finish; Simplified

The basic communication chain between the front end, back end, and database are the following (Figure 7). From the start of the application, the JSP file retrieves the URLs of the back end and adds them into a global variable for the front end. The front end uses Fetch API to make Ajax (or Asynchronous JavaScript and XML) calls to the back-end controllers with the URLs fetched by the JSP. The back end will then call corresponding DAOs (or Data

Access Objects) which act as a transactional repository. The DAO will then make the required query to the database and send a response back to the communication chain where every participant processes the data accordingly.

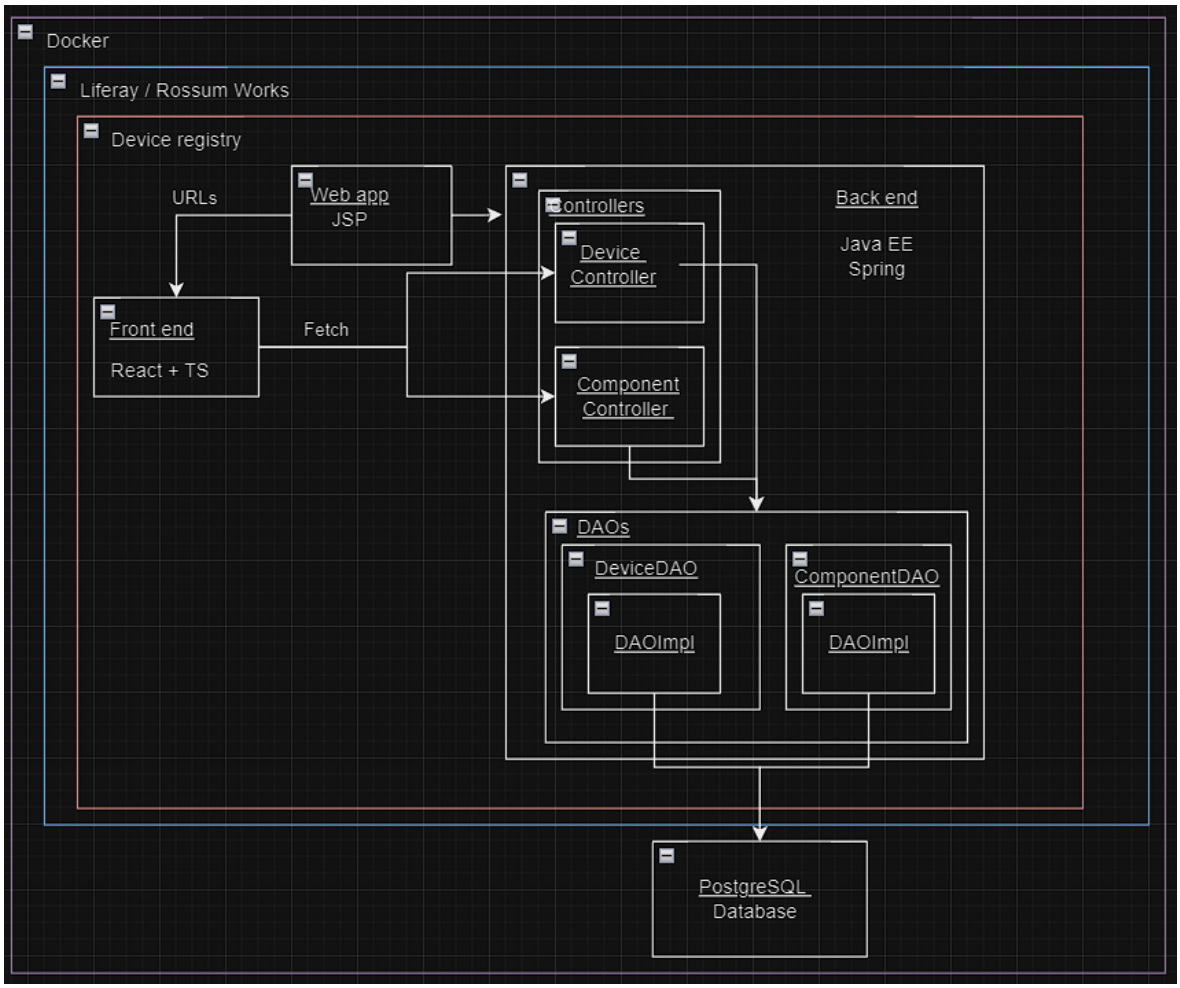


Figure 7: Communication chain

3.2 Technologies and tools used

As we're working with an existing project, a lot of the technical aspects were already predefined. For the front end, this meant working with React and typescript, and given that Liferay is inherently based on Java, this had already narrowed the backend to Java. Additionally, a PostgreSQL database has been implemented, and the entire setup is running in a docker container. For the actual integrated development environment or IDE, there were two possible options, Eclipse and Visual Studio Code, which in the end boiled down to personal preferences.

3.2.1 Liferay portal

Rossum's products are built upon Liferay's portal, which is an open-source Web application framework based on Java. Even if Liferay's portal is based on the Java platform, it can be extended and customized using any JVM-compatible programming language, such as Scala, JRuby, Jython, and others (Liferay Help Center, n.d.-a). It is a robust platform supported by extensive documentation, community support, and professional services.

3.2.1.1 Portlet

Portlets in Liferay are web-based components based on the Java Portlet Specification, a well-established standard that has evolved over time. Initial foundations of what a portal is, and its behaviour were defined in 2003 with JSR-168 and subsequently refined by JSR-268 in 2008, this version introduced features like inter-portlet communication, while ensuring backward compatibility (Liferay Help Center, n.d.-b). JSR-362, released in 2017, superseded earlier versions and brought significant enhancements to portlet technology (Liferay Help Center, n.d.-b). These enhancements include support for JEE7 or Java Platform, Enterprise Edition 7 features, improved resource sharing among portlets, and readiness for future Web Services for Remote Portlets or WSRP (Prince, 2024).

As the portlets in Liferay are built upon well-known specifications, the portlets are standards-compliant but there are differences in the APIs to make developers' lives easier (Liferay Help Center, n.d.-b). These differences, without modification, prevent portlets from running in non-Liferay portal containers. That being said, Liferay is not restricted to its own framework but allows portlets developed with strictly standards-compliant frameworks, to be deployed to a standards-compliant portal container (Liferay Help Center, n.d.-b).

3.2.1.2 Servlet

Servlets are Liferay's way of providing lightweight endpoints for users. They are Java programs that run on a server and handle requests and responses from clients. In the context of Liferay, servlets are used to extend the functionality of the portal and provide custom web services or RESTful endpoints. (Liferay Help Center, n.d.-c)

Servlets can be extremely useful when coupled with REST APIs in the context of isolating code and configuring it for specific tasks.

3.2.2 React

React is a JavaScript library for building user interfaces. One of the main advantages for developers is that it is composed of components, allowing you to divide UI into separate, reusable pieces, and consider each piece separately (React.org, n.d.-a). React uses its own syntax extension called JSX, resembling HTML, used to describe the UI in React (React.org, n.d.-b). While the syntax extension is not mandatory, it enhances readability and provides better error messages. React has been a recent addon to Rossum's products, where recently, plain JavaScript solutions in portlets have been replaced by more modern React solutions.

In our case, we are coupling React with TypeScript to take advantage of TypeScript's type-checking making the developed software less prone to error as they're identified in development. This change in general logic changes the syntax extension from Reacts normal JSX to TSX, which indicates that TypeScript is being used.

3.2.3 Programming languages

As the project involved both front-end and back-end development, multiple programming languages were used. These languages were predetermined based on the company's main platform and front-end requirements. The company's main platform, Liferay, is built using Java. For front-end development, TypeScript and React were chosen to modernize Rossum's software.

3.2.3.1 Java

Liferay's portlet is fundamentally created with Java EE 7 or Java Platform, Enterprise Edition. The difference between Java SE or standard edition and Java EE is that Java EE built on top of Java SE and is fundamentally more robust version of Java, offering for example additional API's and runtime environments.

3.2.3.2 Jakarta Server Pages

In addition to Java, Device Registry utilizes JSP or Jakarta Server Pages to make connecting the front end to the backend easier. JSP is a server-side technology used to create web applications. It consists of HTML and JSP tags, where JSP tags are used to insert Java code into HTML pages (Geeks for Geeks, n.d.). JSP is an excellent choice for Liferay because it provides direct access to the entire Java API, simplifying communication between the front end and back end. The use of JSP is quite common in Liferay projects. For this

project, JSP was predefined, and a base implementation was already in place before starting the work on the final project.

3.2.3.3 Typescript

Typescript was a predefined choice to modernize Rossum's products. It builds upon JavaScript by adding type-checking and optional static types. One of the main benefits of using TypeScript is its improved code quality and compatibility with existing JavaScript code. TypeScript catches errors at compile time rather than runtime, which leads to more robust and maintainable code.

3.2.4 Headless API

A headless API was chosen to give client entities the possibility to automatically update the data that belongs to them. The purpose of this was to provide client entities with the ability to update their own asset management database with information that contractors or subcontractors have completed or modified on the job site. The API structure was designed following OpenAPI specifications, ensuring a standardized and well-documented API that clients can easily reference when building their API endpoints. Additionally, the API was developed as a RESTful API using the JAX-RS specification included in Java EE (Java Enterprise Edition).

3.2.5 Postman

Postman is an API platform used for building and testing APIs. One advantage of using Postman in development is that it allows developers to simulate API endpoints with mock servers, which is particularly helpful for testing (Postman, 2024). Postman makes it easier for developers to manage their APIs more effectively by streamlining every stage of the API lifecycle, from design and testing to documentation and deployment (Postman, 2024). For this project, Postman's Visual Studio Code extension was used to test and validate that the API functions correctly.

3.2.6 DBeaver

DBeaver is a free open-source database management tool that supports a wide variety of databases. Known for its extensive compatibility and clear, easy-to-use user interface, DBeaver was a personal choice for this project due to my positive past experiences with the tool.

3.2.7 PostgreSQL

PostgreSQL is Rossum's preferred database and was already implemented for the project. It has been stated as the world's most advanced open-source relational database by postgresql.org (PostgreSQL, n.d.-a). PostgreSQL is an open-source, highly extensible database that was created in 1986 PostgreSQL, n.d.-b). This makes PostgreSQL a wonderful addon for enterprises seeking reliability, scalability, and extensibility in their database solutions.

3.2.8 IDE

For this project, the IDE or integrated development environment I had two personal options, in previous years when working for Rossum, I had used Eclipse as my development environment due to its status as an IDE primarily made for Java development. Another option for me was Visual Studio Code, which I also considered due to my positive past experiences with it. In the end, Visual Studio Code was chosen for development due to its fast and lightweight nature.

3.2.9 Docker

The Liferay environment had been containerized using Docker. Using Docker, we can package all of the required software like libraries, system tools, code, and runtime into a separate container, which enables easy deployment (AWS Amazon, n.d.). This is another new change in Rossum's products and a demo environment was used during the development of the Device Registry portlet.

3.2.10 Artificial Intelligence

During this project, multiple tools utilizing artificial intelligence have been heavily relied on. For coding, debugging, and understanding, ChatGPT was utilized due to its ability to understand and provide detailed explanations in natural human language. Another essential tool during the writing process of the final project report is Grammarly. Grammarly has been used to help to recognize grammatical mistakes when writing the final project report.

3.2.10.1 ChatGPT

During the development, OpenAI's artificial intelligence chatbot called ChatGPT was utilized in multiple aspects of the project. Due to its ability to take normal human language, for example, English, as an input and answer in multiple languages, giving relevant information that can help you identify and fix the issue. In the end, ChatGPT is a tool that you need to

know how to utilize. As the complexity of the code or the project progresses, ChatGPT's ability to generate coherent and fully functional code greatly diminishes.

Debugging has been the primary use of ChatGPT. Liferay and Rossum's own platforms might produce abnormally long and hard-to-understand stack traces when an error occurs. Using ChatGPT it's fairly simple to gain insight into what might be a probable cause of the error message.

The second relevant use case for ChatGPT has been its ability to quickly summarize and break down code and its relationships. Offering insights to mostly undocumented in-house developed software, like Rossum's Device registry portlet was at the start. This has helped me greatly to understand Liferay's portlet's functionality and how certain functionalities, like API should be implemented.

The third advantage that was already mentioned, is its ability to give suggestions or instructions on how functionalities like APIs should be implemented.

3.2.10.2 Grammarly

Grammarly utilizes the help of artificial intelligence techniques like machine learning, natural language processing, and deep learning to enhance writing by helping to identify misspellings and grammatical and punctuation mistakes (Grammarly, 2019). It also provides suggestions for enhancing the clarity and readability of the text, thus making it more polished. Integrating Grammarly into the writing process can significantly improve the quality of the report, ensuring that it meets academic standards.

As English is not my first language, during the writing process of the final project, Grammarly was used to help identify and mitigate possible errors in the writing.

3.3 GUI

The GUI or the front-end is built on React and coupled with TypeScript. The front end is practically a standard React application extended with TypeScript which uses Fetch API to make recourse calls to the back end. During the development the React application was extended with multiple card components corresponding to different data we wanted to show, forms were created to add data intuitively from the front end and to modify data, and logic was added to handle data when moving between different views.

The user's functionalities with the GUI can be restricted using Liferay's user's role, this is done by asking the back end if the user has a power user role. If the user is not a power

user, functionalities like edit buttons or the ability to access all devices are never rendered which with React means that the code for these is never created for the front end.

Overall, the look of the GUI was kept the same as it was deemed sufficiently clear and worked well with mobile devices, which was a priority during the design process.

3.4 JSP

In Liferay, it is common to use JSP (Jakarta Server Pages) to make communication between the front end and back end simpler. In the Device registry, JSP had been configured to dynamically retrieve URLs due to Liferay's URLs being long and including variables like portlet's id, portlet's mode, etc. For this project, every call we wanted to make into the back end had to be configured into a view.jps file.

3.5 Back end

In this project, most of the changes made have been in the back end. These changes include creating a communication path for front end to back end communication.

For resource requests from the front end, multiple Spring MVC controller classes were created, each corresponding to a specific data table. This approach enhances readability, maintainability, and modularity. These classes serve as the endpoints for the backend, responsible for handling requests and interfacing with the respective data tables.

Every database table requires the creation of a corresponding JPA (or Java Persistence API) entity in order to be accessed by the database; in other words, each corresponding Java class represents a table in the database.

For making database calls, Java DAOs (or Data Access Objects) were created. The idea of using DAOs is that it offers an extra layer to separate persistence logic from the business logic, making the service remain completely in the dark on how the access to the database is done (Shubham, 2022). DAO design pattern requires three things after the mentioned JPA (or Java Persistence API), DAO interface, and DAOImpl which implements the DAO interface and has the concrete implementation of the persistence logic.

As Device Registry was intended to be accessed throughout Rossum's Toolkit, a way to transfer data between portlets had to be created. As Rossum's Toolkit had a preexisting API for retrieving data associated with Toolkit's token which is generated when accessing a tool configured within the Toolkit, the process was fairly straightforward. Create an API in Device Registry and add credentials for it in the database. For Toolkit, the Device registry had to be registered as another tool. As Toolkit components or tools are created dynamically, all that was needed for this process was adding configuration to Toolkit's database,

such as the tool portlets base URL, whether or not we want to use tokens, and what data should be associated with the token.

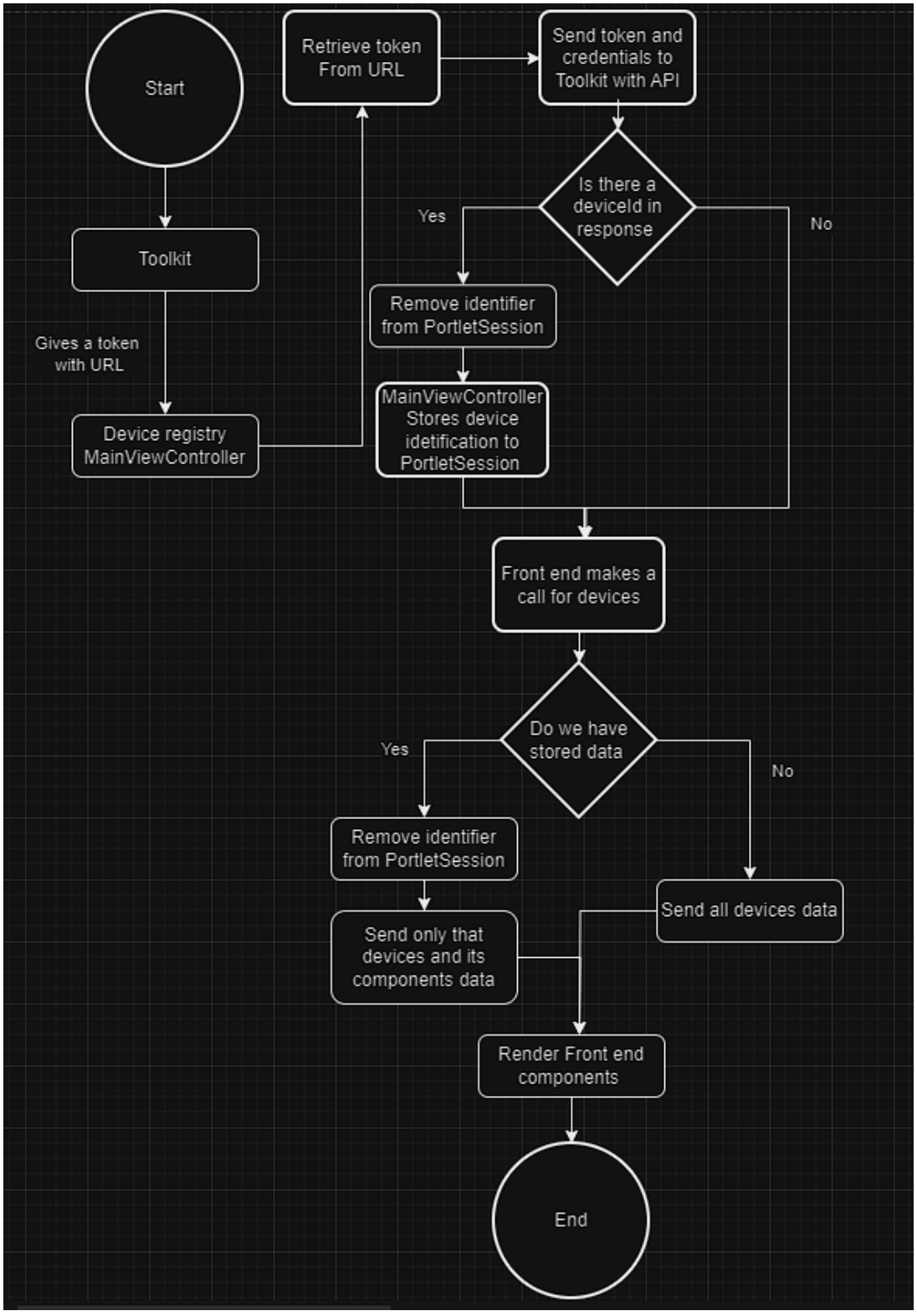


Figure 8: Flowchart of Toolkit in Device registry

3.6 API

The API functions separately from the Java backend. API was chosen to be made into a separate servlet as it provides several advantages, especially in the context of maintaining and configuring the actual application. Built with JAX-RS and Swagger.io annotations. JAX-RS, which is a Java EE's (or Java Enterprise Edition's) specification for creating RESTful web services in Java. Swagger.io annotations are used to document API endpoints, their operations, and responses.

The API URL consists of:

1. The base URL of Liferay's instance where the current portlet is being used.
    - Example: http://toiminnanohjaus-test.rossum.fi:18080
2. The legacy prefix from Liferay 7.
    - Prefix: /o
3. The name of the portlet.
    - Portlet name: /devicereg
4. The path configured in the servlet.
    - Path: /api/rest

Putting it all together, the complete API URL in the demo environment would be: http://toiminnanohjaus-test.rossum.fi:18080/o/devicereg/api/rest

Currently, the API endpoints include retrieving, updating, adding, and deleting entries in the database for devices, components, and device history. The API

A DTO or a Data Transfer Object was created for the API to offer an extra layer of security, as we are not exposing the whole database itself to clients but can limit the API's access to required fields. Another reason a DTO was created, is that we have defined a standardized format for how the data is transferred with the API. For Rossum this provides some flexibility as we have a defined data structure for the API, allowing developers to make changes to the actual database structure without affecting the clients. This means that even if the underlying database schema evolves, such as adding new columns, renaming existing fields, or modifying relationships, the DTO can remain consistent, ensuring a stable interface for clients to interact with.

For example, when we want to add data to the device table using the API. The API receives data as JSON or JavaScript Object Notation, with an array of objects (Figure ()). The

received data is converted into the actual domain object before calling the deviceDAO interface which handles storing data into the database. After every successful addition, DAO sends back the created database object which is then converted back to DTO and added to the response.

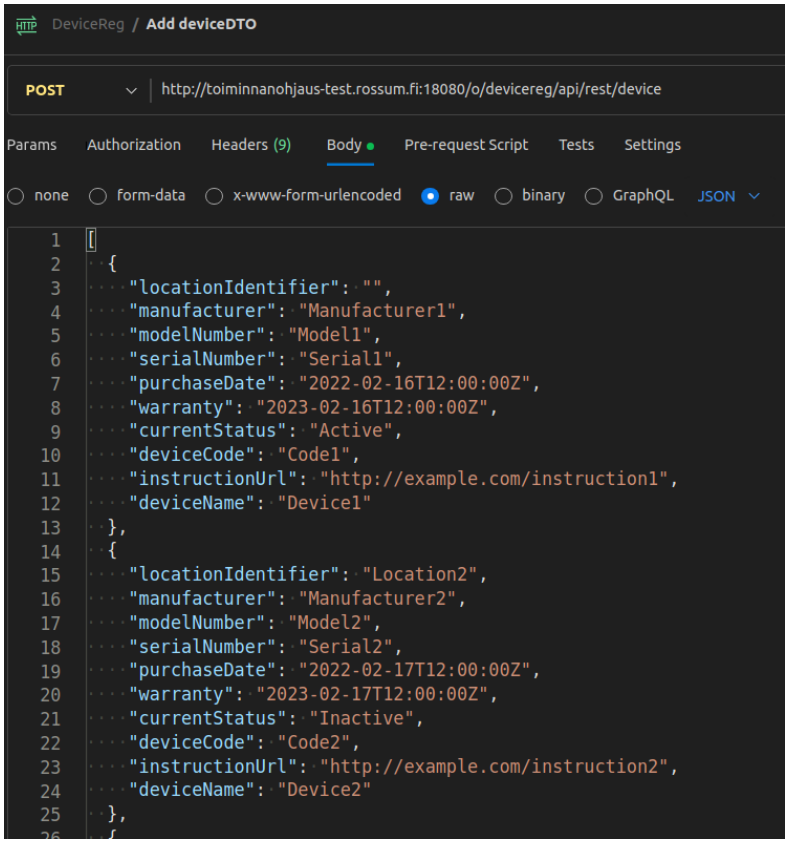


Figure 9: API call, add device using postman

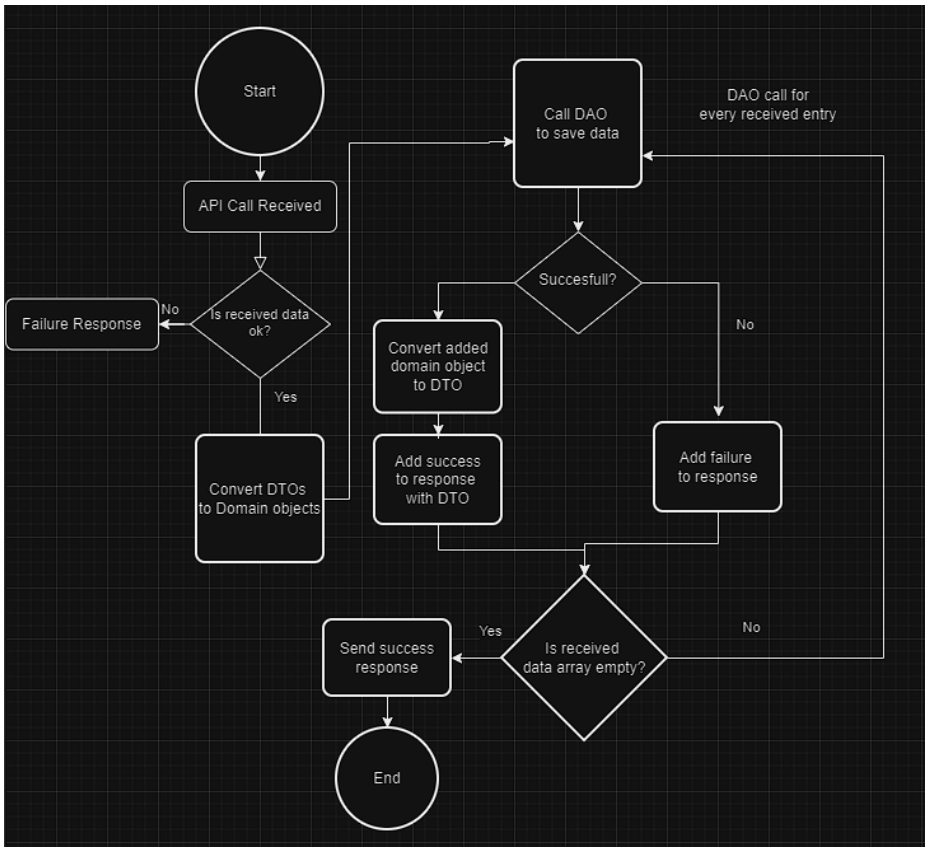


Figure 10: API call add, flowchart

3.7 Database

The database itself has been extended to align with the project's objectives (Figure 11). The device table itself has seen minimal changes, where the only new additional fields are image_binary to store image data in bytea form, device_code, and device_name for easier identification. Component and device history are two entirely new tables that have been added. They both function as many-to-one logic where a single device might have multiple entries in component or device_history tables.

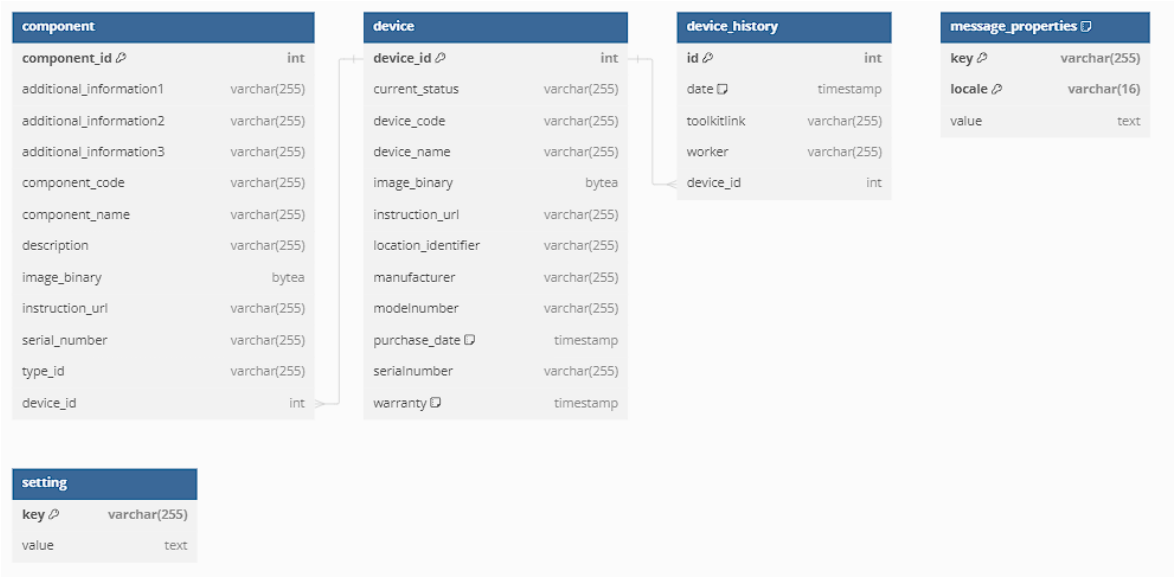


Figure 11: Final database structure

3.8 Email handler

For implementing an email handler, Rossum Works already had a configured system in place, but it was outdated and not in use. This preexisting system had every possible field mapped into an if-else checklist. For SmartE3 a more generalized system was created. Going with the name of General Email Handler, this new handler was designed to match the information received in the email into field names corresponding to the JPA entity to create a work ticket and send response emails to the sender. For this project, a new email was configured to be an inbox for incoming email alerts.

Email handler functionality was created in Rossum Works and not in the Device registry to achieve the goals of the project.

The benefit of creating the handler to be more generalized is that the database or JPA can be extended, and it requires no changes to the handler itself but creates the problem that possible clients need to be informed of these changes.

4 Result

This chapter shows what has been created. To make it easier to follow, the chapter has been divided into smaller subsections. The results are presented as a guided walkthrough of the application, showcasing it with a possible real-world scenario.

4.1 Service ticket to Rossum Works

An email-based system was chosen due to its simplicity and ease of implementation compared to alternative solutions like APIs. This simplicity allows quicker setup without the need for complex configurations or additional infrastructure.

The process begins when the client's Energy Management System (EMS) recognizes an error and sends an error email into a preconfigured inbox owned by Rossum (Figure 12).

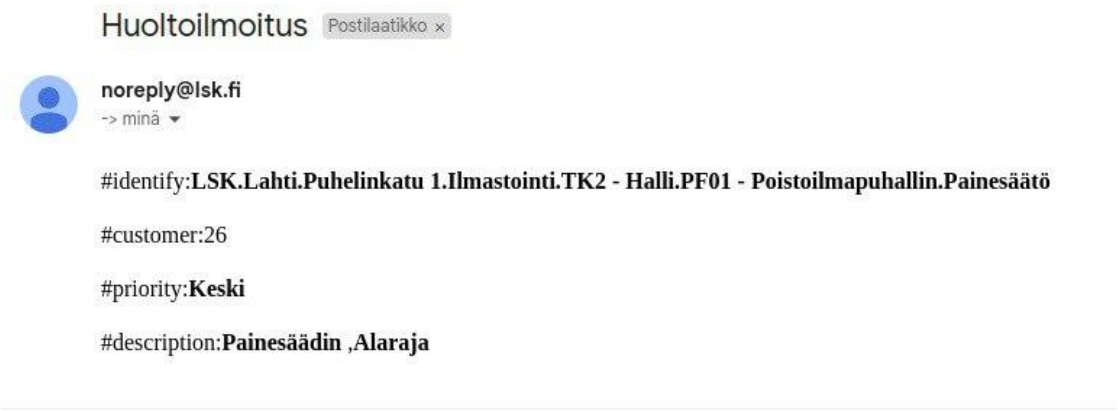


Figure 12: Error email

This email is then parsed into data complying with Job and JobRow database tables, effectively creating a new job ticket in the system. After the creation of a new job is successful, a success email is then sent to the same email address where the original error email

was received (Figure 13). This email has a Toolkit link attached to it, from where the user can directly access Rossum's toolkit without the need to first access Rossum Works.

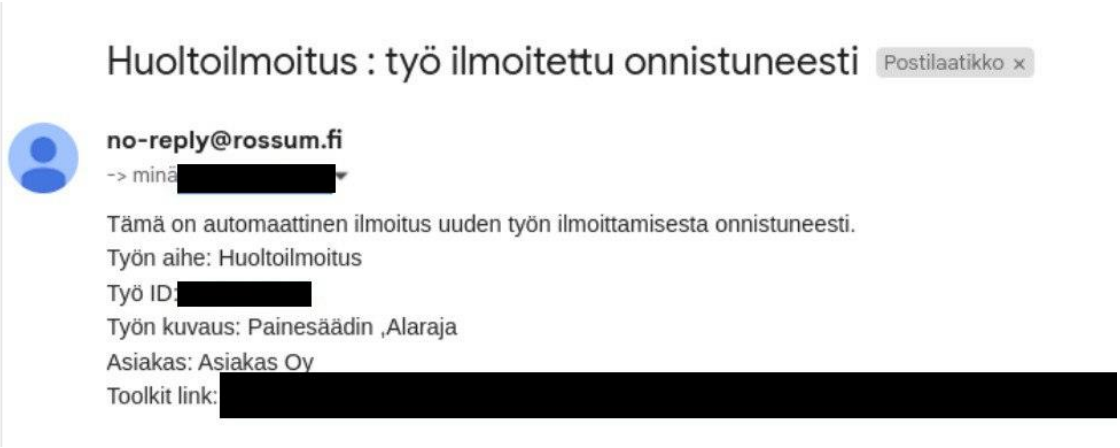


Figure 13: Success response email

If there was an error identified during the creation of the work ticket, a failed response email will be again sent to the same email address where the original error email was received (Figure 14).

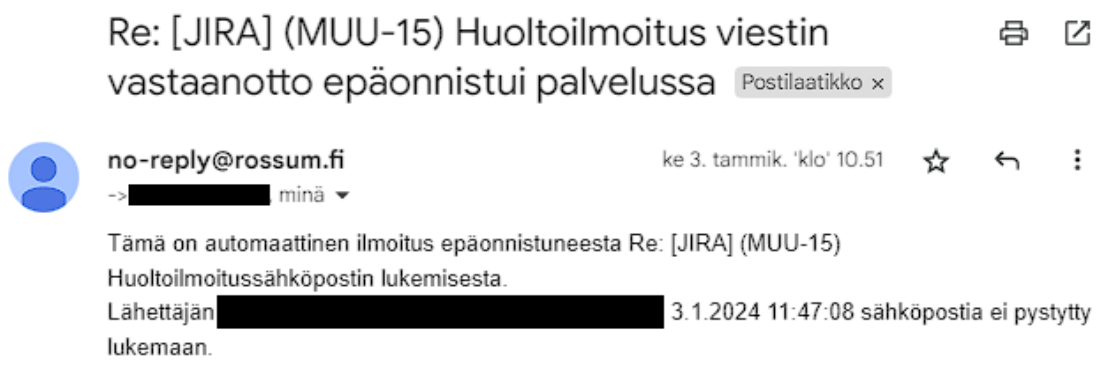


Figure 14: Fail response email

The email-generated ticket can then be viewed in the Rossum Works. This page also includes a button for Rossum's toolkit, seen in figure 15, a green button with a T, under Task data.

Figure 15: Created work ticket

In the case of the figure 12, the identifier is “TK2 - Halli”. This identifier will be used with Rossum’s toolkit (Figure 16) for connecting the ticket to its corresponding device in the Device registry portlet.

Figure 16: Toolkit view

4.2 Device Registry

The GUI or Graphical User Interface has remained fairly recognizable throughout the development, as most of the changes have happened in the backend of the project. As the Device registry portlet is made with the field worker in mind, it decided to keep the current design as it’s intuitive and the simple design works with different phones.

4.2.1 GUI: Power user

In the viewpoint shown to the user that has Liferay role of administrator or a power user. The displayed data cards have been restructured to first show simplified information with a possible link to the documentation of the said device (figure 17). Devices can be easily searched by using their code, serial number, status, or location identifier, making it easier to navigate. Then opening a said device, detailed information and components corresponding to the device will be shown.

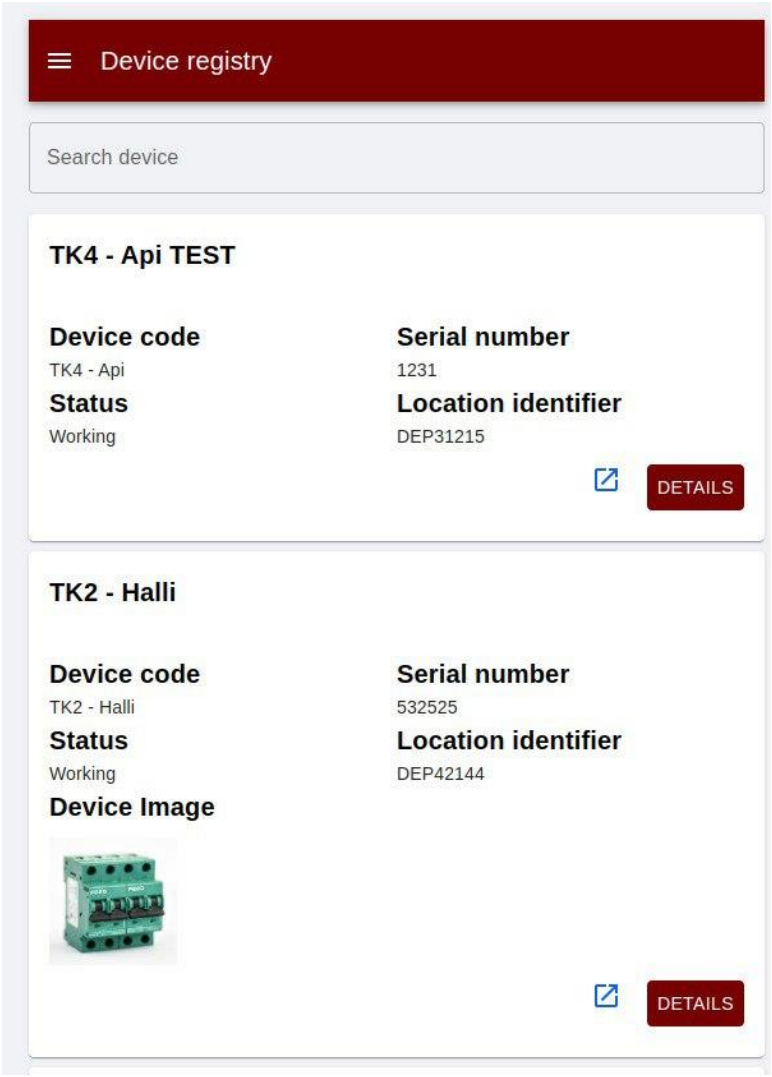


Figure 17: Updated GUI

Depending on the user's role within Liferay's system, from Figure 17's drop-down menu, the user can open a device form (Figure 18) to add new devices straight from the GUI, Where the only required fields are the Location Identifier, Device Code and Device Name.

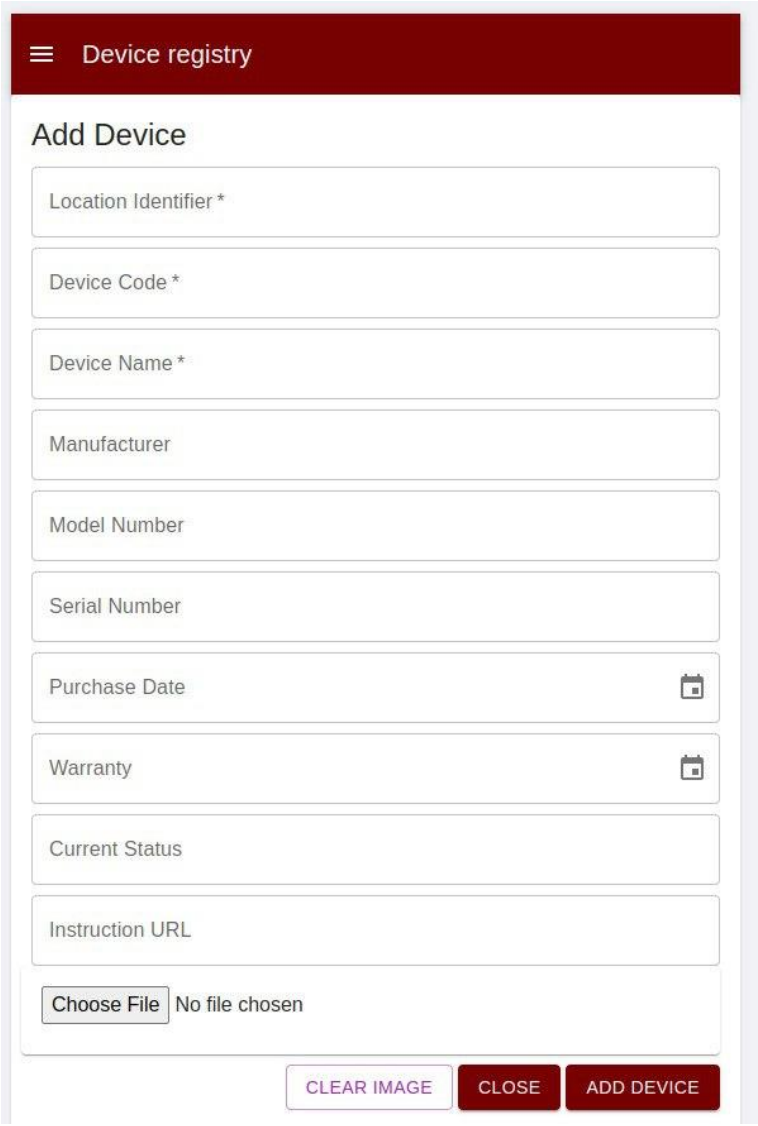


Figure 18: Device form

An equivalent form can be found when inspecting the device (Figure 19).

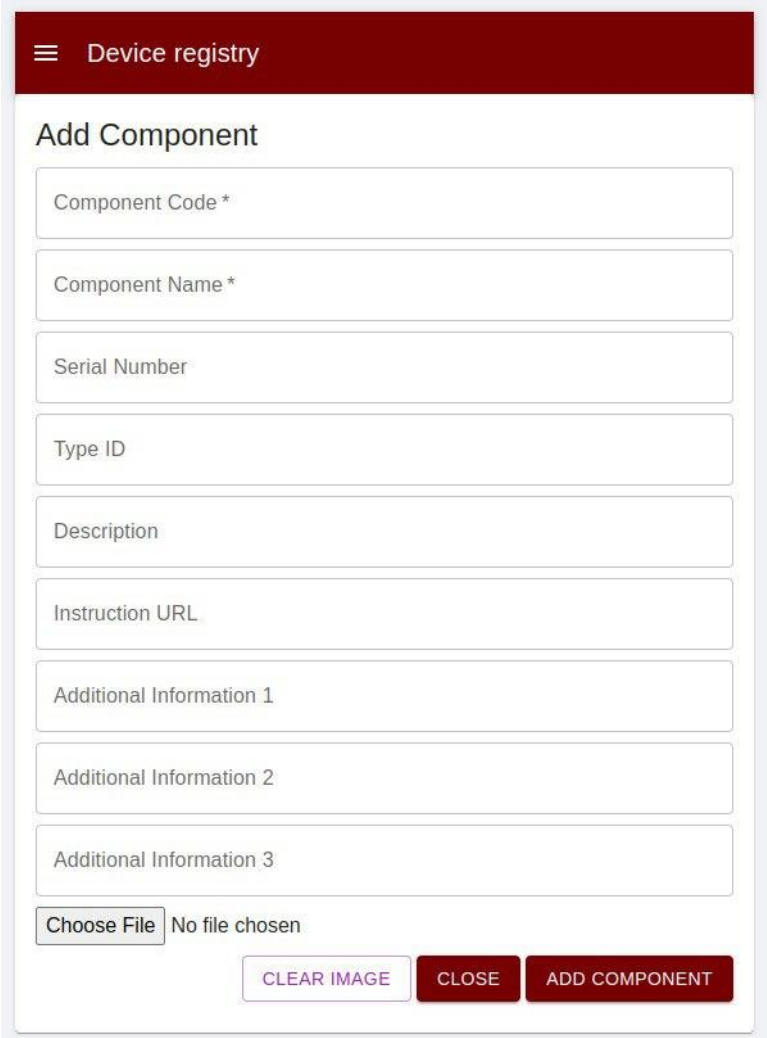


Figure 19: Component form

When inspecting a device, the user is shown more detailed information about the actual device itself and its corresponding components. To keep the GUI simple to navigate, a decision was made to limit the GUI to two layers. The first layer consists of a list of devices (Figure 17), while the second layer provides detailed information about each device and its components (Figure 20).

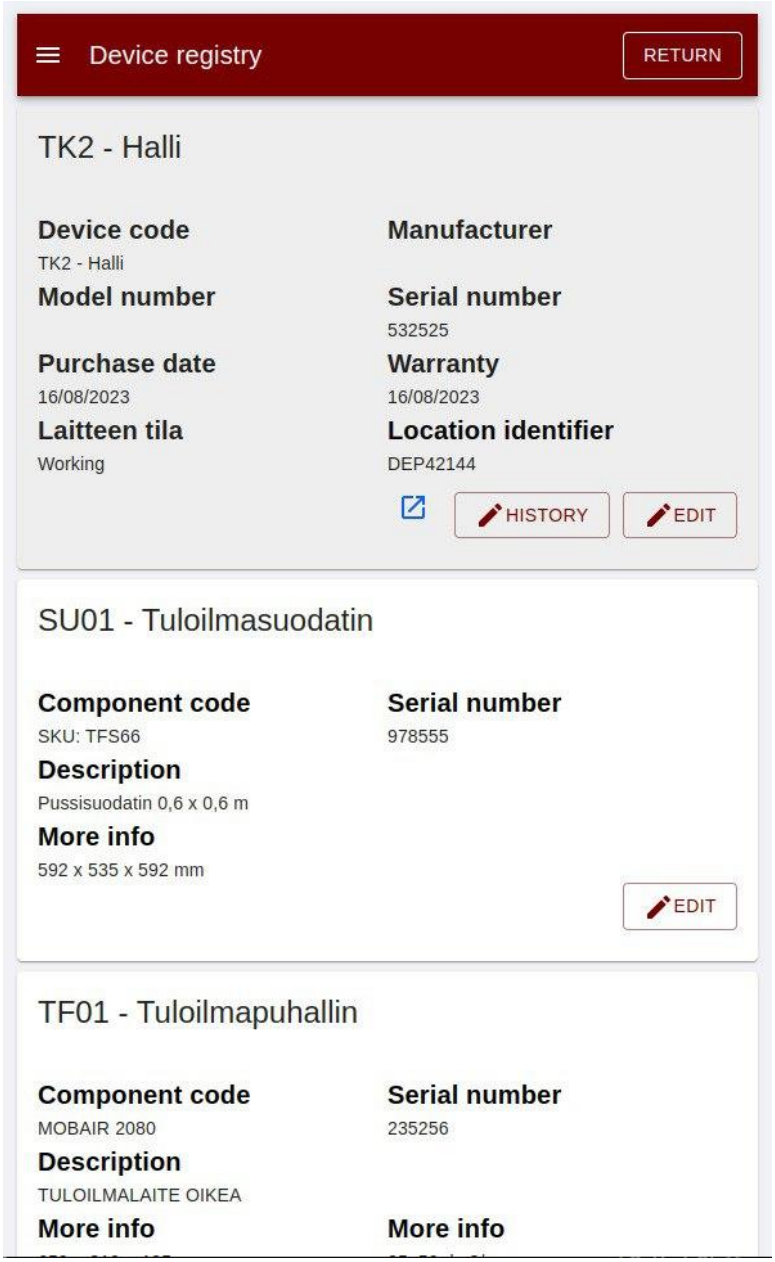


Figure 20: Detailed view

When inspecting the device, the user can then view the device history by clicking the history button (Figure 20). This will then switch out the component cards and show the history cards (Figure 21). The history cards consist of the date of creation, the name of the worker who had created it, and a link to the Toolkit where the worker can see documentation of what has been done for the device.

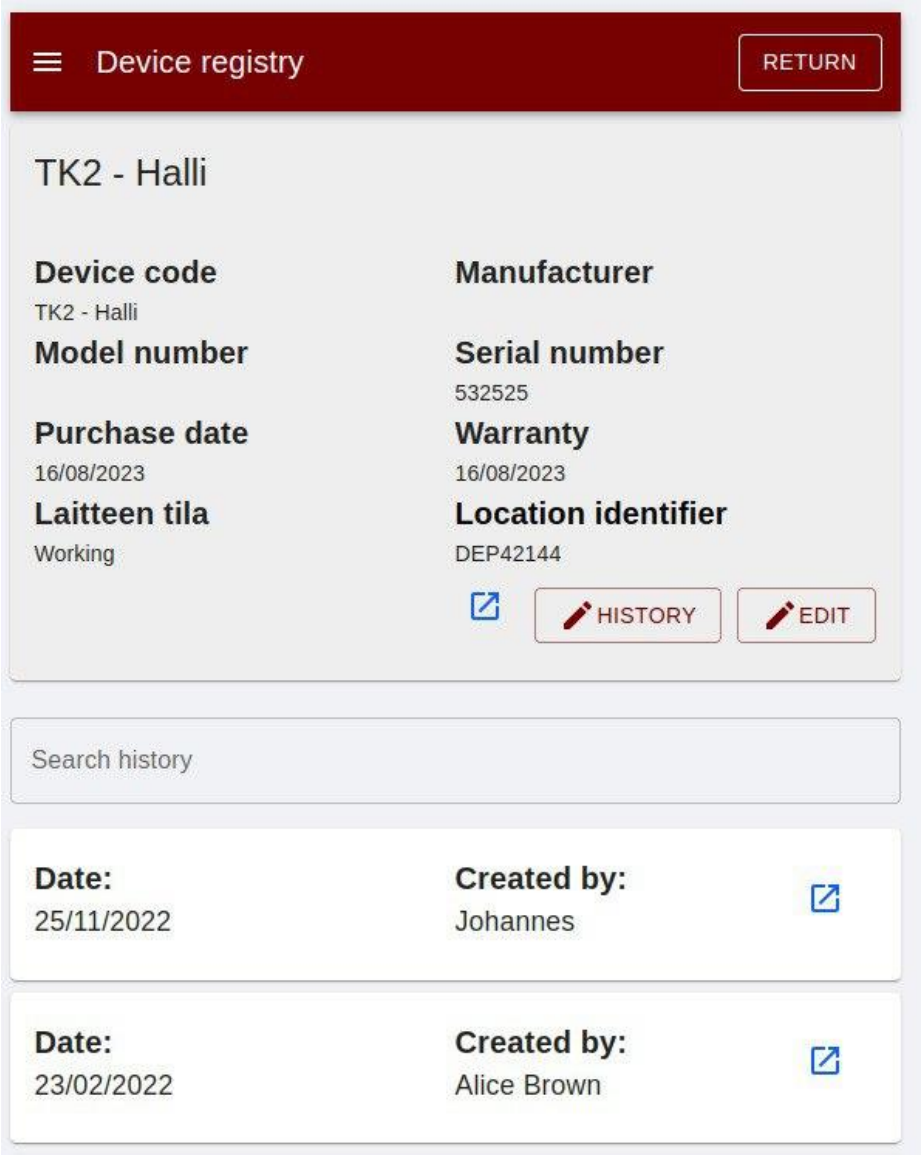


Figure 21: Device History

When inspecting a device (Figure 20), a user with a power user role can modify both the device and its components using corresponding edit buttons. The user will then be presented with an identical form used to create the device or component, filled with data of the device.

Figure 22: Device modify form

Corresponding form can be found for the components.

Figure 23: Modify component form

4.2.2 GUI: Normal user

The intended way for a worker to access the Device registry is through Toolkit (Figure 16). When accessed this way, the user will be immediately shown the problematic device and has no way to access other devices or modify directly the data. From this view, the worker can then inspect the device and plan ahead what might be the problem and what the completion of this work ticket would require.

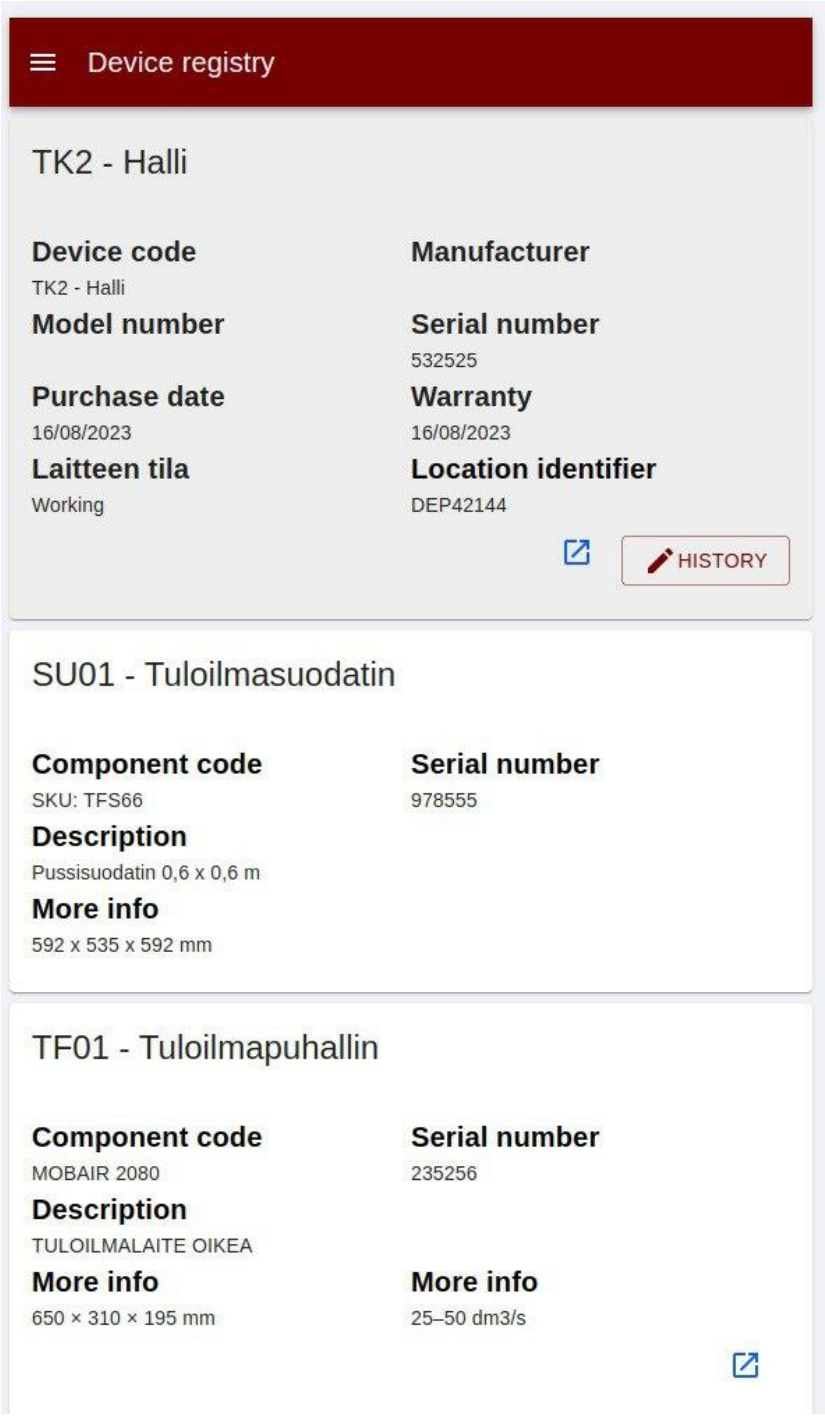


Figure 24: Detailed view for normal user

4.3 API

Portlet offers a headless RESTful API for entities like companies or end-users to access and update the data in the Device registry or retrieve data for their own asset management service.

API takes in JavaScript Objects shown in figure 9 and sends a response in a similar fashion. The response separates and shows which objects were not added successfully (Figure 25).

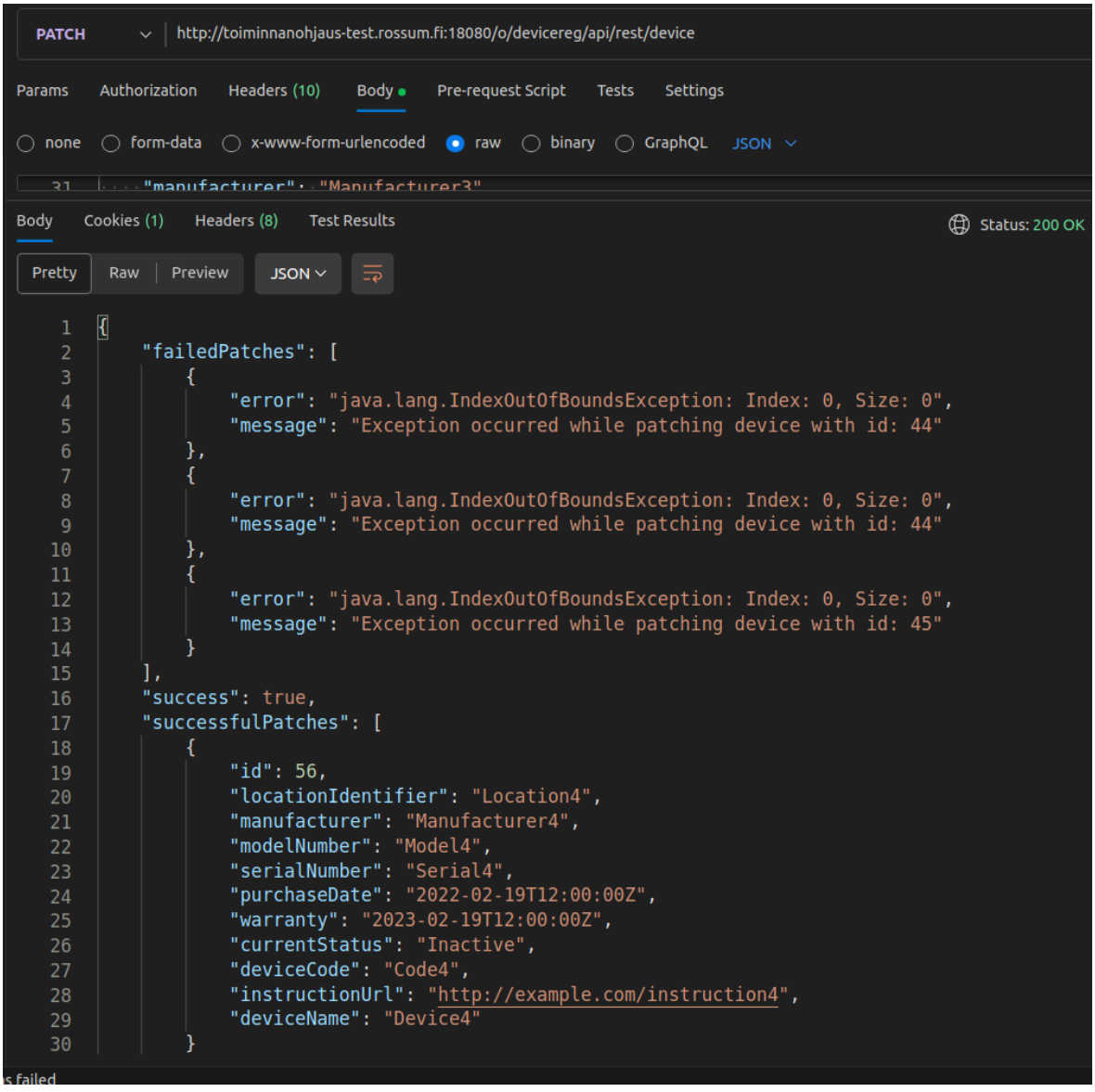
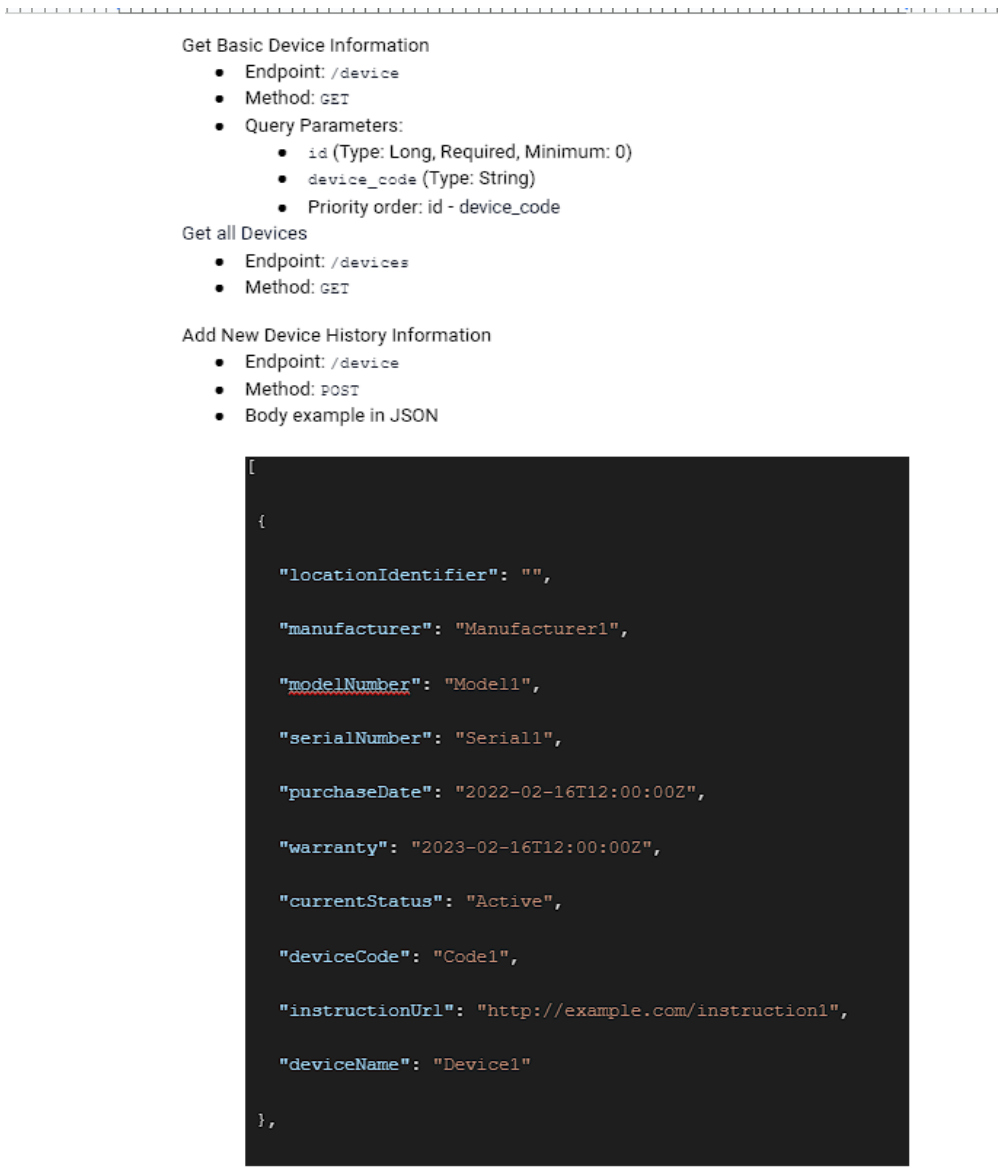


Figure 25: API call, patch

Documentation for the API (Figure 26) was made with clients in mind with use cases, which outlines all possible API calls and gives examples of how the data should be sent.



Get Basic Device Information

- Endpoint: `/device`
- Method: `GET`
- Query Parameters:
  - `id` (Type: Long, Required, Minimum: 0)
  - `device_code` (Type: String)
  - Priority order: id - device_code

Get all Devices

- Endpoint: `/devices`
- Method: `GET`

Add New Device History Information

- Endpoint: `/device`
- Method: `POST`
- Body example in JSON

```
[
 {
   "locationIdentifier": "",
   "manufacturer": "Manufacturer1",
   "modelNumber": "Model1",
   "serialNumber": "Serial1",
   "purchaseDate": "2022-02-16T12:00:00Z",
   "warranty": "2023-02-16T12:00:00Z",
   "currentStatus": "Active",
   "deviceCode": "Code1",
   "instructionUrl": "http://example.com/instruction1",
   "deviceName": "Device1"
 },
```

Figure 26: API use case documentation

5 Conclusion

Overall, the SmartE3 project on behalf of Rossum Oy has been a success. The goal of this project has been to advance the research project called SmartE3 on behalf of Rossum Oy. The goal for Rossum was to define and develop an asset management service or a CMDB and leverage existing solutions to create a system where if the automation fails, a real worker could be automatically assigned and sent to tend the problem.

The development of the project faced several challenges, particularly in working with enterprise-level applications and complying with industry standards. These challenges included highly complex workings of Liferay and Rossum's own software built upon after mentioned application, making sure that the solutions created were compliant with industry standards, and ensuring they could be effectively used.

The solutions developed have reached most of the required functionality, as the Device Registry portlet has reached a point in development where it can and has been used to make functional demonstrations to clients. Device registry has the required capability to function as a CMDB for the test property and the data can be accessed through API or GUI. The only key functionality that is missing from the goals, is incorporating Toolkit links for the Device registry's history.

As it was always a research project, it's still quite unpolished and requires a lot of modifications before the final implementation.

5.1 Future

As the final project was never intended to encapsulate the start and the finish of the project, there is still a lot of work to be done. Due to the project's research nature, a lot of the solutions created have been made with a single client and test property in mind to create proof-of-concept software.

5.1.1 Finalising the database

As it was never concluded what the actual data we want to store from the devices and components, the current database has development fields and is designed exclusively for the demo, with a single client and test property in mind. The database must be extended to be able to distinguish between information that belongs to different parties.

Another add-on to the database, that was proposed and specified by the end of the development, was a table, containing information from the API calls. This table would include the type of the API call, API endpoint, who made it, timestamp, the data sent, and whether

it was successful or not. This change would be used not only for monitoring but enhancing security. It would give Rossum a great way to audit, identify usage patterns, detect anomalies, and provide accountability.

In the current implementation, images are turned into byte arrays or into a bytea or byte array format and stored in the database. This approach simplifies storing the images as there is no need to set up an external drive or location in the cloud or a server. It offers streamlining the process at the expense of making the database unnecessarily big. Another downside that was noticed during testing was a significant hit on loading times of data as bytea needs to be converted back to a viewable image in the browser.

5.1.2 API

API itself and its use cases have been created manually using OpenAPI specifications. For future simplification, the API should be recreated and updated using tools like Swagger.io, where the code and use cases for API can be easily created. Creating the API with OpenAPI specification would allow the company to generate the use case documentation through Swagger.io. These documentations could then be shared with clients which gives clear instructions that the API is industry standard.

API calls should also be limited to x number of calls in x amount of time from a single source, as a countermeasure for a Dos attack or Denial of Service attack where an attacker tries to overwhelm a system by sending an excessive number of requests in a short period of time.

5.1.2.1 API authentication

After the smartE3 research project has been concluded and before the developed solution will be offered to new clients, the data handling must be reworked. Currently, the authentication is being handled by Apache configuration, where if an entity is added to the configuration list, it can send unrestricted API calls, meaning it potentially can retrieve and modify data that are not properly restricted. This setup lacks fine-grained access control, posing several security risks.

During the development of the SmartE3 demo, an OAuth2 service was proposed for authentication, but due to project research nature and surfaced consideration if the OAuth2 service could be utilized in other aspects of the business, it was decided to not be implemented now as Apache would fit the required criteria for authentication.

5.1.2.2 API ticket

Another potential enhancement would be to add functionality for creating a work ticket through API calls, similar to the current email-based system. The email-based approach was chosen for its easy implementation but should also be considered for API. This change would remove the need for third-party software used for sending emails and would be a vastly more scalable solution.

5.1.3 Email reading

The current alert email handling functionality creates a new work ticket every time an error occurs. Inherently this is the functionality we wanted to achieve, but in cases where a sensor hovers around the line it would be frequently creating a new error email and a new work ticket.

Now, a situation like this would fill up the ticketing system and would require manual work to be cleaned. For this reason, if multiple error emails are received corresponding to a single device or a sensor, a single work ticket could be created, and based on how many error emails have been received, the urgency of the ticket would be raised. This approach would streamline the ticketing process by reducing redundant tickets and ensuring that recurring issues are properly prioritized.

5.1.4 Toolkit link

Currently, there is no solution for retrieving Toolkit links for the maintenance history of the device. There are a few possible solutions. The current work ticket link can be brought with the identifier of the device when someone uses Toolkit to search for a device. The main problem encountered here is that there could be historical gaps as a result. Another option is to extend the Device Registry's Toolkit API, which is being used to retrieve token data, to systematically send calls to Toolkit for retrieving these links. Functionality like this is not currently found in Toolkit, which would require extending Toolkit's API specifically for Device Registry before extending Device Registry's own API.

5.1.5 Access

The current demo has been made as a proof of concept with a single client in mind. After finalizing the database to accommodate multiple clients, the after-mentioned authentication in API authentication (5.1.2.1) has to be resolved, but also the GUI has to be modified. Currently, we are only taking into account if the user's role is an administrator or a power user. This functionality has to be extended to take into account what the user should get access to.

For this, we could utilize readily available information in Liferay's of the user's organization. This information can be then brought to the attention of the Device Registry through the same methods as the role. This approach would solve the access problem with organizations, but in the original description, we wanted to give access to entities that might need access to the information of these devices. These entities included the manufacturer of the device, the end-user or the owner, and a service provider like operators.

Another option would be to create an access functionality separate from Rossum's existing platform, which would require possibly maintaining an exact copy of the Rossum Works users and extending it with individual entities. This implementation requires a way to synchronize changes in the Rossum Works database to Device Registry possibly throughout an API, which again requires first creating the functionality to Rossum Works.

5.1.6 Worker functionalities and capabilities

The developed solution currently only provides the field worker with the ability to examine and check a device's history; it lacks the functionality or protocol that specifies what should be done when a device needs a component changed. One of the benefits of creating a device registry was to provide companies with information on what has been done in the field. Now, it's clear that the worker must be able to modify the component data or at least be able to send a modification proposal.

To address this, we can utilize the developed functionality to add and modify components in the GUI. However, this approach would be prone to human error and if left unattended, could lead not only to the Device Registry database being faulty but also if a company wants to potentially synchronize changes happening to devices in the field, their own asset management database would get this faulty data.

This part will be always prone to human error and requires more attention for a suitable solution to be found.

5.1.7 Documentation

One of the main hardships of working on this project has been a lack of documentation. The portlet application itself was not completed and was missing the functionality to make calls to the back end. Combined with the company's in-house developed software which extends from Liferay's own documentation or purely disregards it, made troubleshooting and debugging extremely difficult, as it sometimes required aimlessly searching through code in hopes of finding something corresponding to what was trying to be implemented.

Documentation needs to be made into a more standard practice as it would simplify the development process and cut down the time required for implementation.

Bibliography

AIXTOR Technologies. (2023, Mar 1). What is Liferay? *Medium*. Retrieved from https://medium.com/@aixtor-technologies/what-is-liferay-f62a283256eb

AWS Amazon. (n.d.). What is Docker? Retrieved June 29, 2024, from https://aws.amazon.com/docker/

Business Finland. (2022). NOKIA'S COMPETITIVE EDGE PROJECT RESPONDS TO ONE OF THE HOTTEST TRENDS IN TECH: EDGE COMPUTING. Retrieved May 5, 2024, from https://www.businessfinland.fi/en/whats-new/cases/2022/nokias-competitive-edge-project

dbeaver.io. (n.d.). About. Retrieved June 28, 2024, from https://dbeaver.io/about/

Geeks for Geeks. (n.d.). Introduction to JSP. Retrieved July 2, 2024, from https://www.geeksforgeeks.org/introduction-to-jsp/

Grammarly. (2019, May 17). How We Use AI to Enhance Your Writing | Grammarly Spotlight. Retrieved June 28, 2024, from https://www.grammarly.com/blog/how-grammarly-uses-ai/

Gothe, D. (2010, Jan). Understanding the Java Portlet Specification 2.0 (JSR 286): Part 1, Overview and Coordination Between Portlets. Retrieved May 8, 2024, from https://www.oracle.com/java/technologies/portlet-specification1-v20.html

ISO.org. (n.d.). ISO/IEC 27001:2013. Retrieved May 7, 2024, from https://www.iso.org/contents/data/standard/05/45/54534.html

JCP.org. (n.d.). JSR 362: Portlet Specification 3.0. Retrieved May 8, 2024, from https://jcp.org/en/jsr/detail?id=362

Kangas, T. (n.d.). Lahti GEM -kumppanusten Business Finland -projektissa pilotoidaan kiinteistön ja sähköautojen älykästä energianhallintaratkaisua. *Lahti GEM*. Retrieved from https://lahtigem.fi/news/lahti-gem-kumppanusten-business-finland-projektissa-pilotoidaan-kiinteiston-ja-sahkoautojen-alykasta-energianhallintaratkaisua

Lakatos, D. (2024, Mar 7). Java EE vs Java SE: Navigating the Landscape of Java Platforms. *Medium*. Retrieved from https://medium.com/@lktsdvd/java-ee-vs-java-se-navigating-the-landscape-of-java-platforms-b6f8372362ba

Liferay. (2022). Liferay DXP Application Security Features. Retrieved May 7, 2024, from https://www.liferay.com/documents/10182/282340280/Liferay+DXP+Application+Security+Features.pdf/6df1c611-abbf-4f4f-a8cc-3800ef7199d3?t=1668615878845

Liferay. (n.d.). What is a Portal. Retrieved May 7, 2024, from https://www.liferay.com/resources/whitepapers/What+is+a+Portal

Liferay Help Center. (n.d.-a). Introduction to Liferay Development. Retrieved June 12, 2024, from https://help.liferay.com/hc/en-us/articles/360018156791-Introduction-to-Liferay-Development

Liferay Help Center. (n.d.-b). Introduction to Portlets. Retrieved May 8, 2024, from https://help.liferay.com/hc/en-us/articles/360018159431-Introduction-to-Portlets

Liferay Help Center. (n.d.-c). Servlets in a Module. Retrieved July 3, 2024, from https://help.liferay.com/hc/en-us/articles/360022829011-Servlets-in-a-Module

LSK Group. (n.d.). LSK yrityksenä. Retrieved May 8, 2024, from https://www.lsk.fi/lsk-yrityksena/

LUT University. (n.d.-a). INTRODUCING THE UNIVERSITY. Retrieved May 22, 2024, from https://www.lut.fi/en/about-us/introducing-university

LUT University. (n.d.-b). Strategy 2030: Trailblazers – Science with a Purpose. Retrieved June 6, 2024, from https://www.lut.fi/en/about-us/introducing-university/strategy

Nokia. (n.d.). Nokia Veturi program: Competitive Edge. Retrieved May 22, 2024, from https://www.nokia.com/innovation/veturi-programs/competitive-edge/#objectives

postgresql.org. (n.d.-a). PostgreSQL: The World's Most Advanced Open Source Relational Database. Retrieved June 28, 2024, from https://www.postgresql.org/

postgresql.org. (n.d.-b). About. Retrieved June 28, 2024, from https://www.postgresql.org/about/

Postman. (n.d.). What is Postman? Retrieved July 3, 2024, from https://www.postman.com/product/what-is-postman/

Prince, M. (2024, May 8). Introduction to JSR 362 / Portlet Specification 3.0. *Liferay Savvy*. Retrieved from http://www.liferaysavvy.com/2016/03/introduction-to-jsr-362-portlet.html

React.org. (n.d.-a). Components and Props. Retrieved June 26, 2024, from https://legacy.reactjs.org/docs/components-and-props.html

React.org. (n.d.-b). Introducing JSX. Retrieved July 1, 2024, from https://legacy.reactjs.org/docs/introducing-jsx.html

Rollbar editor team. (2023, Mar 13). How to Debug Code Using ChatGPT. *Rollbar*. Retrieved June 28, 2024, from https://rollbar.com/blog/how-to-debug-code-using-chatgpt/

Rossum Oy. (n.d.-a). Rossum Oy. Retrieved May 9, 2024, from https://rossum.fi/yritys_rossum/

Rossum Oy. (n.d.-b). teleoperaattoreiden pitkäaikainen kumppani. Retrieved July 1, 2024, from https://rossum.fi/teleoperaattoreiden-kumppani/

Rossum Oy. (n.d.-c). Rossum Toolkit. Retrieved July 1, 2024, from https://rossum.fi/toolkit_rossum/

Rossum Oy. (n.d.-d). Rossum Ticket Cloud [Infographic]. Internal Infographic image: unpublished.

Rossum Oy. (n.d.-e). Rossum Works. Retrieved July 1, 2024, from https://rossum.fi/works_rossum/

Shubham. (2022, Aug 3). DAO Design Pattern. *Digital Ocean*. Retrieved July 4, 2024, from https://www.digitalocean.com/community/tutorials/dao-design-pattern

typescriptlang.org. (n.d.). What is TypeScript? Retrieved June 28, 2024, from https://www.typescriptlang.org/