Satakunnan ammattikorkeakoulu
Satakunta University of Applied Sciences

EMIL KHAIBRAKHMANOV

Remote monitoring of warehouse and distribution processing

Developing a mobile phone application for remote monitoring of warehouse and distribution processing.

DEGREE PROGRAMME IN DATA ENGINEERING
2024

ABSTRACT

Khaibrakhmanov, Emil: Remote monitoring of warehouse and distribution processing. Developing a mobile phone application for remote monitoring of warehouse and distribution processing.
Bachelor’s thesis
Degree programme in Data Engineering
August 2024
Number of pages: 49

In this thesis, I developed a mobile application for monitoring and management of automated warehouse systems. The application enables real-time tracking of system performance and failures, providing detailed information about the type, location, and nature of errors directly to employees' mobile devices. By utilizing features such as barcode scanner and error notifications, the application allows for seamless identification and resolution of issues. The development process focuses on selecting efficient technologies and programming languages that ensure high performance, reliability, and scalability. Additionally, the application is designed with user-friendly interfaces and security features to safeguard sensitive operational data. This mobile solution aims to reduce labor costs, minimize downtime, and improve overall operational efficiency within the warehouse environment.

Keywords: automation, software engineering, data collection, Rust, React Native, Axum, Expo, mobile application

Contents

1. INTRODUCTION.

1.1 Background information

Modern large-scale warehouse facilities are not operated manually. Instead, distribution processes are nearly fully automated and executed by robots. The performance of a modern warehouse hinges on two critical factors: the hardware machinery employed on-site and the advanced control software in use. The latter is crucial for guaranteeing seamless and reliable operations of all robotic components. The integration of high-quality control software can lead to substantial reductions in labor costs, reduced frequency of failures, and enhanced efficiency in inventory management.

Despite significant advancements in enhancing the reliability of automated warehouse distribution technologies, failures remain unavoidable. This can occur due to various factors on both the hardware and software sides, including failures in item identification, such as misplaced item identifiers (e.g., barcodes) or reader failures, mechanical damage to conveyor components, human errors, and software bugs, among others. Therefore, it is crucial to reliably track warehouse operations, collect error data, identify the sources of these errors, and swiftly resolve any issues.

In this Thesis work, I focus on improving the software component used to track and resolve possible failures of automated warehouses. I present a program in the form of a mobile application that allows to swiftly identify type, location, and other detailed information about the occurred error. In certain cases, the application even allows resolving the problem directly from the mobile device.

1.2 Problem statement.

Modern warehouses are large-scale, complex systems composed of large numbers of interacting components. Advanced software products are used to monitor and

store information about the status, performance, and failures in the distribution process. Typically, such information is collected and can be accessed via centralized consoles located at fixed locations at the warehouse premises. This solution is, however, sub-optimal in various scenarios. Firstly, it requires an operator to be always present at the console. The console can be located far away from the site where the failure actually took place, so the information has to be communicated manually to other employees across the warehouse. This process is centralized, includes a lot of human labor, and can be time-barring.

An alternative approach would be to provide all employees with immediate and detailed information about the status of the warehouse processes and about a failure if it occurs. This can be achieved via a portable device, such as a mobile phone. Moreover, smart features such as location services can be added, such that the error notifications are assigned to employees near to the site failure. Finally, certain functionality to resolve simple failures can be added directly to the mobile device.

1.3 Objectives of this Thesis

The objective of my Thesis project is to develop a prototype of software for tracking the status and failures taking place at warehouse distribution centers using mobile devices instead of stationary consoles. The software will come in the form of a mobile application that can be installed on mobile platforms such as iOS and Android, as well as accessed through a web-browser.

The software should provide the user with the following functionality:

- A barcode scanner. The application should have access to the device camera and use it as a barcode scanner. Upon scanning the barcode, the application should provide the user with a precise location of the scanned item.
- A unit data page. The application should provide the user with certain information about an item, such as its type, weight, and dimensions.

- A status page. The application should provide the user with detailed information about the status of certain devices used in the distribution process.
- Notifications functionality. The application should allow the user to see important events live, such as errors and other critical information. In addition, the application should notify the user about such critical events via push notifications.
- In addition to the mobile application, the software should be available through a web browser. The web-based version will have limited functionality due to the lack of certain features such as a barcode scanner.
- Access the core system to fetch data mentioned above.

In addition to the minimal requirements listed above, the following features can be added to the software:

- Global network access to ensure ease of access from any location with an internet connection.
- A built-in functionality to resolve specific issues directly from the application.

The rest of the Thesis is organized in the following way. In Section 2, I review the technologies enabling the development of the proposed software. and determine the high-level design of the software. Section 3 is devoted to the implementation of the mobile application.

2. Methodology

In this section, I provide a brief survey of the technological tools that will be employed in the development of the mobile application. The first critical decision

involves selecting the appropriate programming language, which will have a significant effect on the app performance, compatibility, and reliability. The second decision concerns the management of data storage and transfers, which involves choosing the right databases, ensuring data security, and optimizing data transmission efficiency.

2.1 Programming language and framework

The development process and the code architecture strongly depend on the programming language in use.

2.1.1 Frontend.

In the field of software engineering, the term "frontend" denotes the presentation layer. Essentially, it refers to the visual and interactive elements of a website or an app that clients interact directly with (Wikipedia, 2024a) .

I have considered a few candidates to be used in front-end development. The first option is to use platform-specific programming languages. A programming language that Google officially adopted in 2017 for the development of Android applications is Kotlin (Matias Martinez and Bruno Gois Mateus., 2021), while Swift is used exclusively for developing apps for iOS-based systems (Apple developer). This approach potentially offers better performance and lower energy consumption, and also provides less network consumption (Horn et al., 2023). Such a platform-specific approach, however, requires the development and maintenance of separate codebases for each platform, including separate Kotlin codebase for Android, Swift codebase for iOS, and JavaScript-based version for Web browsers. This approach hence demands significantly more effort.

Alternatively, an application can be designed using a cross-platform framework, which allows a single codebase to be used across all supported platforms. While being less optimized in terms of performance, this approach provides a great benefit when it comes to cross-platform development.

For the purposes of this Thesis, I decided to use a cross-platform framework. While being slightly less efficient in terms of performance and resource usage, this approach allows to run and test an application across various supported platforms without the need of developing a new code base, hence providing an optimal choice for developing the app prototype.

Next, I need to make a choice of a framework and programming language for the development process. A framework is a comprehensive collection of tools and libraries that serve as a fundamental basis for creating software applications. In essence, a programming language serves as a tool, whereas a framework functions as a work platform (GeekForGeeks, 2024).

Among many of available frameworks, I considered Flutter and React Native as two candidates. Since the two frameworks use different programming languages, the development process will differ a lot depending on the choice I make. Below, I provide a brief introduction to them and outline the advantages and disadvantages of each option.

Flutter, an innovation brought forth by Google, is constructed upon the Dart programming language (Flutter, 2024). Among its most noteworthy attributes is Flutter's capability to convert Dart into native code across a variety of platforms, thereby achieving performance that closely rivals that of native code. This proficiency facilitates the creation of mobile and web applications that are both high in performance and refined in execution. Moreover, the Hot-Reload feature offers developers the distinct advantage of witnessing their alterations in real-time, a boon that significantly boosts the debugging process and curtails the duration required for development (Marimuthu et al., 2023).

The other framework I considered was React Native. This Java Script-based framework was resealed by Meta in 2015 and maintained since then at (React Native, 2024). React Native has a big community support, which grants access to a large number of third-party libraries. Therefore, the majority of the features are

readily available and can be easily implemented. Just like Flutter, React Native includes a hot reload feature, which allows users to modify the code and observe the outcome without the need to stop the application (GeeksForGeeks, 2024). In my case, usage of React Native does not require the learning of a new programming language, as it bears close similarity to React, which I have prior experience with. React Native utilizes a bridge to communicate with the native platform, which results in a reduction in performance and an increase in overhead compared to Flutter. The bridge acts as translator by providing data exchange between the React Native and native platform (GeeksForGeeks, 2022).

Among the two frameworks, I have chosen to use React Native. This decision is based on my familiarity with TypeScript and the similarities between React Native and React, which makes the transition easier for developers experienced in React.

2.1.2 Backend.

Backend is a data access layer responsible for the application's functionality, logic, and database interactions. (Wikipedia, 2024b) When choosing the backend programming language, the main requirements taken into account are performance, security, and the possibility to deploy an app into a cloud.

For the backend, I have chosen Rust (Rust programming language, 2024) as a programming language. Rust has a steep learning curve, but as a trade off it offers everything that I need for implementing the backend. Rust gained prominence and support from a large number of developers. Figure 1 shows Google trend chart, demonstrating an increasing popularity of Rust in the past decade. In 2018, Rust was ranked as the fifth programming language in terms of rapidly growing popularity by GitHub, and it continues to be the most admirable language in 2024 according to Stack Overflow surveys (Qin et al., 2021) (Stackoverflow).

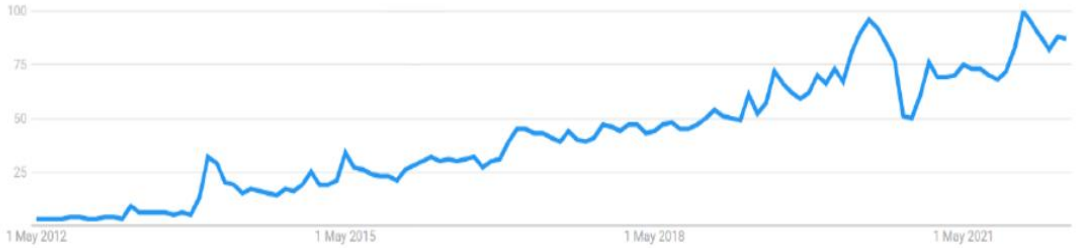


Figure 1. Popularity of Rust programming language over time according to Google trends (Bugden and Alahmar, 2022a) .

Performance—Rust was designed for performance, memory and concurrency safety. Many recent benchmarks indicate that Rust surpasses well-established languages in terms of performance. As such, Rust was found to be either on par, or faster than these languages in many cases. For instance, Fig. 2 provides benchmarks of three important algorithms. As one can see from the graph, Rust outperforms all the competing programming languages in all considered cases, except for being slightly slower in only one case. Figure 3 shows the comparison of simple web services implemented in Rust and Java. Java was outperformed by a huge margin in all aspects such as latency, memory, CPU utilization and program size.

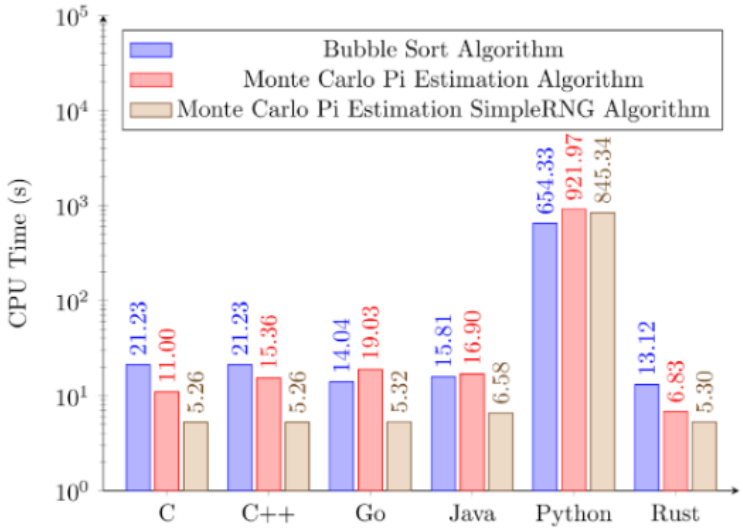


Figure 2. Average CPU time benchmarks in various algorithms (Bugden and Alahmar, 2022b).

Performance Test	Java	Rust	Improvement
Latency (Fibonacci)	1,900 ms	57.71 ms	~30x
Memory	1,498 Mb	16.94 Mb	~80x
Idle Memory	162 Mb	0.36 Mb	~450x
CPU Utilization	73%	24%	~3x
Program Size	27 Mb	3.7 Mb	~8x

Figure 3. Simple Web Service benchmark results (InfinyOn, 2024).

Memory and concurrency safety—Memory safety is the condition of being covered from a variety of software bugs and security vulnerabilities (Wikipedia, 2024). Rust is one of the best languages in preventing memory-safety bugs. In order to ensure memory-safe programming, Rust implements a set of strict semantic rules. As a result, Rust-based applications are well protected against safety bugs (Xu et al, 2021). The White House press release related to cybersecurity strategy has recently recommended the use of memory and type safety languages and provided Rust as an example of such language. (The White House, 2024)

Rust is designed to enforce memory safety (Wikipedia, 2024), meaning that the code will not compile if it violates memory safety rules defined by Rust, unless special *unsafe* labels are added in the code (Blog Rust, 2015a) (Documentation Rust, 2024). Memory safety violations are among the most prevalent causes of vulnerabilities. As reported by the Microsoft Security Response Center, memory safety issues continue to account for approximately 70% of the common vulnerabilities and exposures annually.

In the present day, memory safety is of such importance that I contemplated selecting Rust from a wide range of other languages.

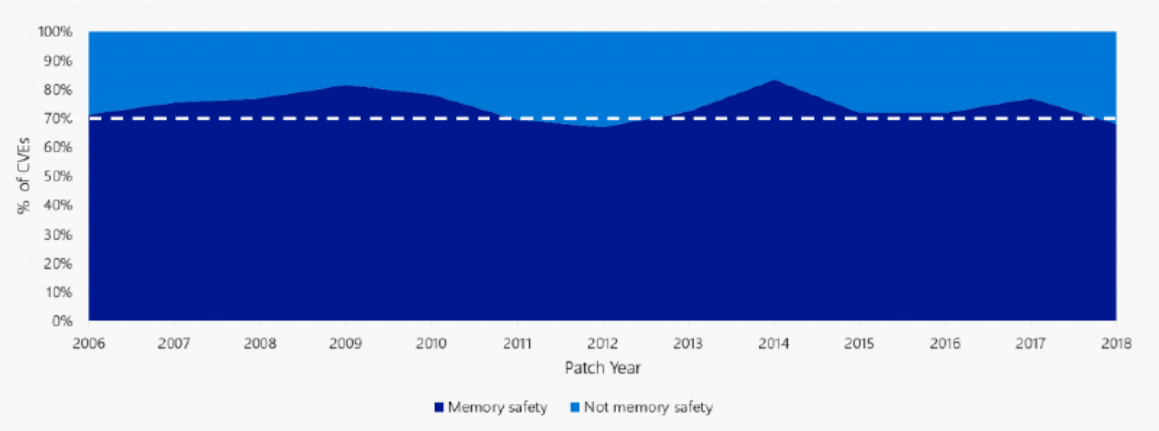


Figure 4. Memory safety issues continue to comprise approximately 70% of the vulnerabilities that Microsoft allocates a CVE each year (Microsoft Security Response Center, 2019).

Rust also offers strong benefits for concurrent task execution, which is vital for the efficiency of modern applications (Lenovo, 2024). Concurrency bugs are dangerous and can allow hackers to intervene the code operation by, e.g., hijacking code execution or bypassing security checks (Zhao et al., 2018). Rust offers a variety of paradigms to avoid concurrency bugs and permit the confident management of concurrency (Blog Rust, 2015b) (PullRequest, 2024).

Other benefits of Rust—In Europe, energy efficiency is crucial as it reduces energy expenses, safeguards Nature by minimizing carbon footprint, enhances quality of life, lessens the EU's dependency on external oil and gas suppliers, and fosters sustainable economic growth within the EU. (European Commission) Rust did show noteworthy results in this regard, as it was identified as one of the most energy-efficient programming languages during the research. Figure 5 shows that in energy efficiency Rust takes second place after C.

Total					
	Energy		Time		Mb
(c) C	1.00	(c) C	1.00	(c) Pascal	1.00
(c) Rust	1.03	(c) Rust	1.04	(c) Go	1.05
(c) C++	1.34	(c) C++	1.56	(c) C	1.17
(c) Ada	1.70	(c) Ada	1.85	(c) Fortran	1.24
(v) Java	1.98	(v) Java	1.89	(c) C++	1.34
(c) Pascal	2.14	(c) Chapel	2.14	(c) Ada	1.47
(c) Chapel	2.18	(c) Go	2.83	(c) Rust	1.54
(v) Lisp	2.27	(c) Pascal	3.02	(v) Lisp	1.92
(c) Ocaml	2.40	(c) Ocaml	3.09	(c) Haskell	2.45
(c) Fortran	2.52	(v) C#	3.14	(i) PHP	2.57
(c) Swift	2.79	(v) Lisp	3.40	(c) Swift	2.71
(c) Haskell	3.10	(c) Haskell	3.55	(i) Python	2.80
(v) C#	3.14	(c) Swift	4.20	(c) Ocaml	2.82
(c) Go	3.23	(c) Fortran	4.20	(v) C#	2.85
(i) Dart	3.83	(v) F#	6.30	(i) Hack	3.34
(v) F#	4.13	(i) JavaScript	6.52	(v) Racket	3.52
(i) JavaScript	4.45	(i) Dart	6.67	(i) Ruby	3.97
(v) Racket	7.91	(v) Racket	11.27	(c) Chapel	4.00
(i) TypeScript	21.50	(i) Hack	26.99	(v) F#	4.25
(i) Hack	24.02	(i) PHP	27.64	(i) JavaScript	4.59
(i) PHP	29.30	(v) Erlang	36.71	(i) TypeScript	4.69
(v) Erlang	42.23	(i) Jruby	43.44	(v) Java	6.01
(i) Lua	45.98	(i) TypeScript	46.20	(i) Perl	6.62
(i) Jruby	46.54	(i) Ruby	59.34	(i) Lua	6.72
(i) Ruby	69.91	(i) Perl	65.79	(v) Erlang	7.20
(i) Python	75.88	(i) Python	71.90	(i) Dart	8.64
(i) Perl	79.58	(i) Lua	82.91	(i) Jruby	19.84

Figure 5. Energy, Time and Memory benchmark results (Pereira et al, 2017).

Rust popularity is rapidly growing among Cloud providers due to fast, efficient, and safe memory management. Most cloud native applications have been written or rewritten in Rust (Intel). Applications developed in Rust are typically smaller in size and consume less RAM compared to languages that use higher level abstractionism, such as Java, Javascript. This results in a reduction in the cost of the cloud-hosted application.The cloud bill cost is reduced as a consequence of the application's reduced size and increased efficacy.

There are many excellent web frameworks developed for Rust. Below I provide a brief overview of two widely used frameworks: Actix-Web and Axum.

One of the first frameworks for Rust was actix-web. It has been widely used in production and has established a significant community and plugin ecosystem (Luca

Palmieri, 2022, p.16). It is also extremely fast, extensible (Actix Web), and type-safe. The latter ensures that data is used in accordance with its designated type, hence lowering the probability of bugs and boosts the program stability.

Axum is an easy to use, ergonomic and modular web framework. It gives users a clear and easy way to manage errors, making the process of creating responses in a software system much simpler. By reducing the amount of repetitive coding, it helps streamline the development process, so users can focus more on important tasks instead of dealing with the same code over and over again. This makes the system more efficient and easier to work with. (Tokio-rs blog, 2021). In this work, I chose to use Axum due to its relative simplicity and modular structure.

2.1.3 Database.

An electronic collection of structured data, usually kept in an ordered manner inside a computer system, is called a database (Oracle). The most widely used language for interacting with databases and obtaining data that users want is called Structured Querying Language (SQL) (Zhang et al., 2024). It is a standardized language for relational database management systems, which means it is widely supported across various database platforms. This approach allows to perform complex queries with simple commands, supports powerful and flexible querying capabilities, and can handle large volumes of data.

From a variety of SQL-based databases I must select one for my backend. The first option I considered is MySQL released in 1995. The reliability and efficiency of MySQL are widely recognized. It is one of the most frequently employed database management systems on the internet (Andjelic et al., 2008). However, after reviewing the MySQL license, I found that it may not be the most suitable option for my project due to the certain license limitations. In particular, the requirement to provide the source code might conflict with the company distribution model.

PostgreSQL was released in 1996 and, as the official site states, it was developed for over 35 years. Hence it too manages to stay afloat for many years (PostgreSQL,

2024). PostgreSQL is open-source, meaning that it is free and publicly accessible for potential modification and redistribution (Wikipedia, 2024).

I have extensive experience working with PostgreSQL. I am well-versed in the complexities of PostgreSQL, so usage of it is pretty straightforward for me. In scenarios that involve extensive write operations, PostgreSQL typically outperforms MySQL, despite the fact that MySQL is frequently acknowledged for its superior read performance (Amazon Web Services, 2024). PostgreSQL is particularly well-suited to satisfy my performance requirements due to the fact that my application emphasizes write operations.

I decided to continue using PostgreSQL in this Thesis due to its availability, free distribution model, and my prior experience with it.

2.2 Data integration.

Almost all data is moving through the control system backend and most of it is stored in the control system database. The backend needs to access the data from the control system database, in order to provide the user with the required information, such as location of the unit or product information of the selected unit.

I considered two possible approaches for data integration between the control system and the backend logic. The first approach which is shown in Fig. 6 involves implementing a communication interface, which would transmit the necessary data from the control system to the backend. Upon receiving the data, the backend code parses and stores it in the local database. Implementing the communication interface requires writing the code for both sides - the control system and the mobile application. This involves setting up how the two systems talk to each other, which can be time-consuming. Since this has to be done on both sides, it adds extra work to ensure the two parts communicate smoothly and reliably. This process is essential but requires careful attention to detail, as I need to make sure the control system and the mobile app are perfectly in sync. The Control System gets the necessary data

from the database and sends it to the backend using a data exchange mechanism. When data is received by the backend, it is parsed and saved in the backend database. An alternative approach shown in Fig. 7 is to establish a direct connection to the database, allowing data retrieval.

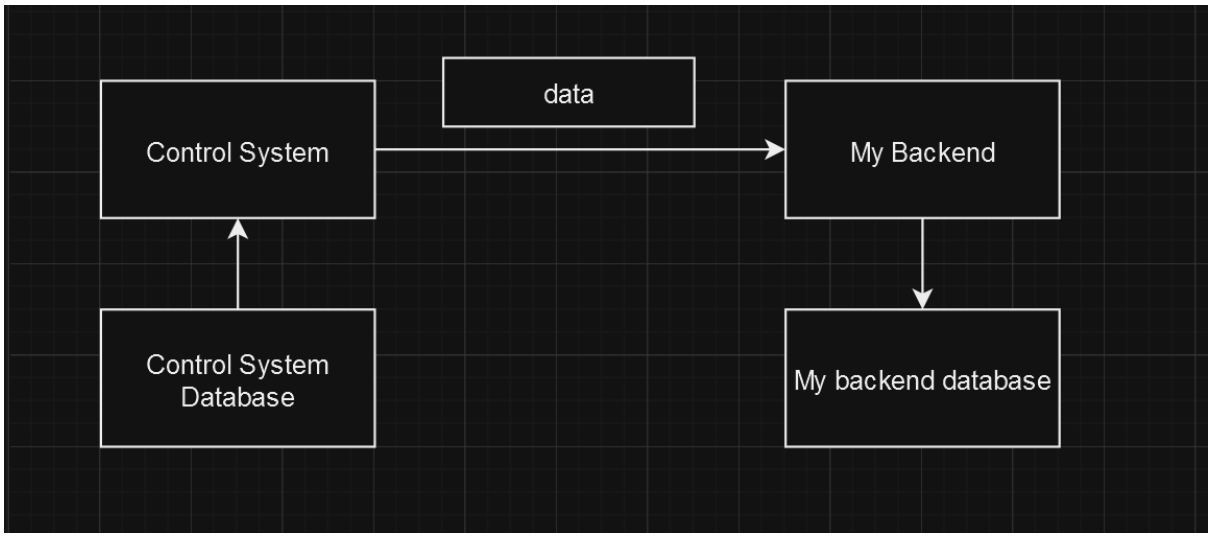


Figure 6. Communication interface usage approach.

For prototyping purposes in this Thesis, direct database access is faster and involves less overhead compared to setting up a communication service. This method simplifies the architecture, accelerates development, but decreases scalability and creates vulnerabilities in case the application will be hosted in the cloud and will be globally accessible. In addition, to reduce the control system database usage, it is possible to use multiple databases with a single backend. For example, data–such as user name–can be stored in a separate from control system database.

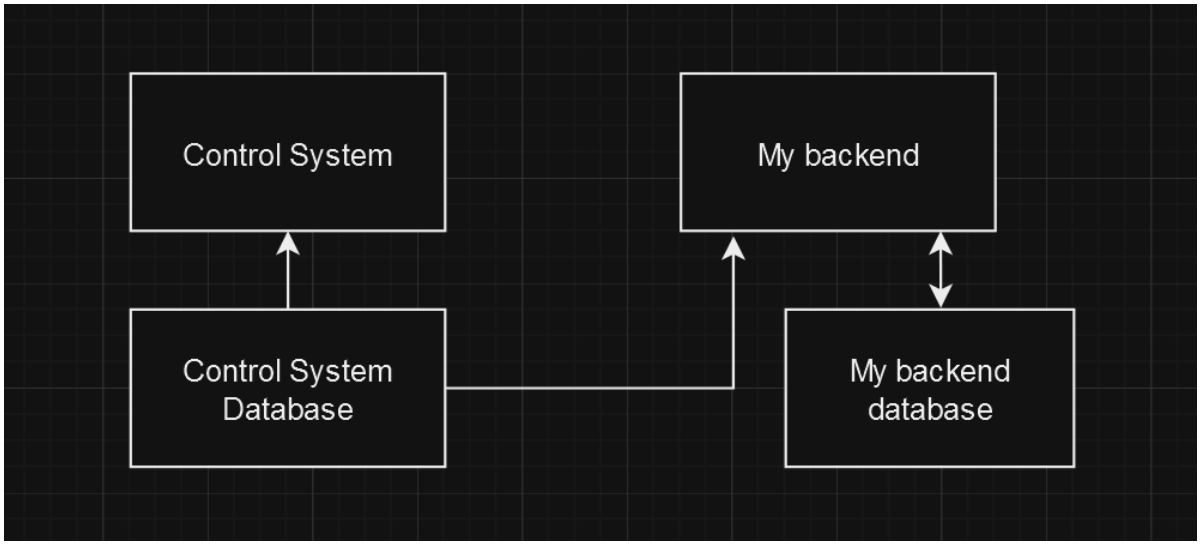


Figure 7. Fetching data directly from the database approach.

In my implementation, the backend inquires directly from the database about the availability of necessary data. If the data is available, it can be sent to the frontend or saved to the backend's own database.

To reduce the development time, I decided to stick with direct connection to the database. And to reduce the usage of the Control System database, I will store what is possible in a separate database, as described above (PostgreSQL).

2.3 General design of the software.

In this Section, I will elaborate more on the functionalities of the application and their implementation.

The main database stores essential information such as unit history, unit dimensions, product names, and the current status of devices (e.g., running or an error). My application requires access to this data, and below I outline the method used for fetching it.

For the backend to fetch the data from the Control System database, the database needs to establish a connection. I use Microsoft's Open Database Connectivity (ODBC) (Progress Software, 2024) driver to connect to the database. This driver interface will allow the backend of the application to connect to the Control System Database and read the data. To use the ODBC in Rust, I require to install a crate–a packaged Rust code which can be either a library or an application. Using such crates allows saving a significant amount of time by using off-the-shelf solutions, which is especially relevant for complex functionalities. This approach is not unique for Rust, and most programming languages use similar package libraries. Crate that I am going to use for the ODBC connection is named "odbc-api" (GitHub, 2024).

My application requires performing read operations on the database. However, since the Control System database already handles a significant number of reads from other applications, I aim to minimize any unnecessary read operations. This approach is intended to reduce the load on the database and prevent any potential performance issues that could impact the other applications relying on the Control System database. To avoid this, my application is going to use a second database. Rust has a crate that allows me to simplify implementation of a connection between the backend and PostgreSQL database. To fetch the data from the internal PostgreSQL database, I will use a crate called sqlx (GitHub, 2024).

Next, mobile applications need to display push notifications to the user, for example, in case of an error. Information should be delivered as soon as the device establishes a connection to the backend. To deliver and show information to the user, the backend should fetch and send data to the frontend. To do so, I require to implement communication between backend and frontend. Mainly, there are 4 possibilities on how to implement notification delivery to the user:

1. Polling. A mobile application sends check requests to the server every x number of seconds. This option is not suitable, because we do want to get error notification as soon as possible with minimal delay and not overheat the database with requests.

2. Server-side events (SSEs). An application makes a special connection to the server and whenever there will be a notification data ready, the server will send the data to the mobile application through this special connection. This option is not suitable also, because SSE does not guarantee the delivery. That means that critical events might be missed.
3. Push notification services such as Apple Push Notification Service or Firebase Cloud message are not suitable either, because they require connection to the global network, which the Automation factory can not always guarantee.
4. WebSockets. Mobile applications with WebSockets can communicate with a server in real time. This implies that notifications can be sent and received instantly without the need to refresh or wait. WebSockets, unlike SSE, guarantees the delivery.

In my application, I chose to use WebSockets, because I would prefer to deliver notifications instantly to the user and guarantee the delivery. However, one does not always need instant delivery and two-way delivery, hence the overhead can be reduced by adding a second communication protocol. I will use a common architecture style called Representational State Transfer (REST) (Karlsson et al, 2023). For example when a user requests information about the product or unit history, data will be transferred via REST. Axum has built-in WebSocket and REST support, same for React Native, so I can use such a functionality without writing complex functionalities.

In the warehouse, each unit has a barcode. By scanning the barcode it is possible to view the whole history of the unit and information about the product. Hence, one of the objectives is to implement the barcode scanner in application and by using a camera scan the barcode of the unit or allow the user to type barcode manually instead. Then show information about the product and unit history. Features like a barcode scanner are available in Expo, a framework and platform for creating React Native applications which offers a set of tools and services aimed at simplifying the mobile app development process. Hence I am going to use Expo framework within my frontend application codes to simplify the development process (Expo, 2024).

Figure 8 demonstrates the high-level design architecture and puts it into perspective.

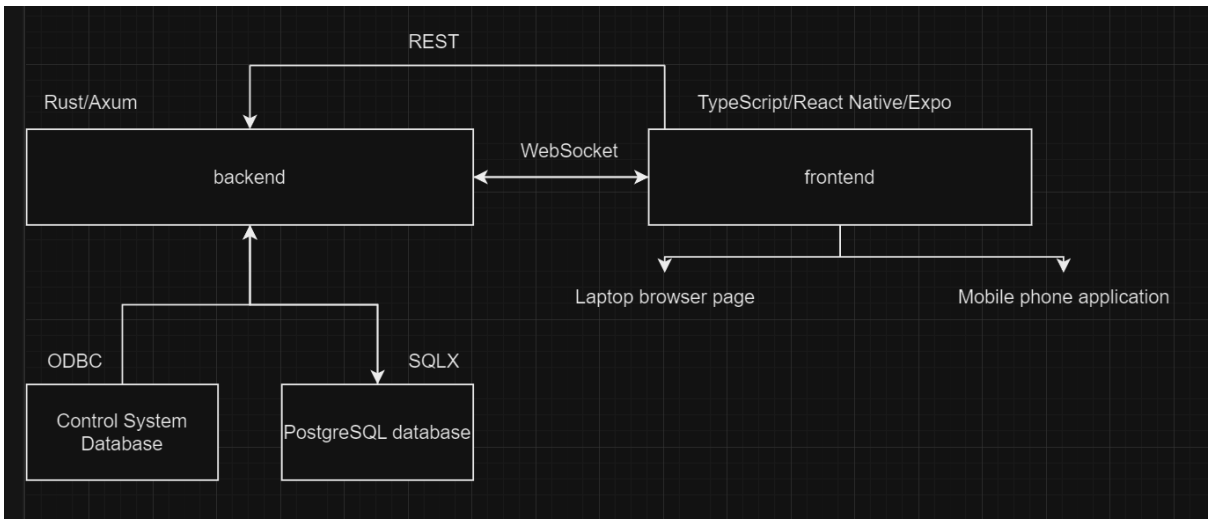


Figure 8. High level architecture design.

To explain the whole structure assume that the user scans the barcode from a mobile phone application. To do so, users are required to click the scan barcode button in mobile application, which will access the camera of the mobile phone and the frontend code of the application decodes the barcode to the numbers. As soon as barcode was decoded to the numbers, code will create the message in JSON format by attaching the received numbers to the key named ‘barcode’ and send the message via REST interface to the backend of the application.

Before continuing the triggered chain, I will explain what JSON is. JSON stands for JavaScript Object Notation and it was designed for the need for humans to easily read and write messages for lightweight data exchange. JSON is also considered as easy for machines to parse and generate. In my case JSON is going to look as follows: {"barcode":"1234567890"}. Left brace is the beginning of the object, the right one is the end of the object, the “barcode” is the name of the pair followed by the colon and the value - “123456789” (JSON, 2024). Backend will parse barcodes wrapped in JSON format, validate against the validity rules that I define later in the thesis, and check the information regarding requested barcodes from the database of the control system. If data is available, the backend structures, wraps the information to JSON format, and retrieves the information to the frontend. The frontend of the application will parse and arrange the data, and subsequently show it

to the user. In the example above we touched on the next sectors of the Figure 8: mobile application, frontend, REST, backend, Control System Database.

To elaborate the app functionality further, I provide one more example, which will include WebSocket and PostgreSQL database usage sectors. Every second, the backend will perform a verification of the status of the devices in the control system database. If any updates are identified, they will be saved in the PostgreSQL database for offline devices. The message will be received by the user at a later time when the user begins using the application. Nevertheless, the notification will be delivered immediately to the user via WebSockets if the user is online.

2.4 IDEs.

It is possible to write the code using classic notepad, but to ease the development I am going to use an Integrated Development Environment (IDE). It is a software application which improves the efficiency of software code development for programmers. It enhances developer efficiency by integrating functions including software editing, development and testing (Amazon Web Services). It contains such features as syntax highlighting, code completion and code navigation. Rust has extensive IDE support, in my case the choice is a personal preference, since I used IDEs created by JetBrains previously for other programming languages, I would stick with the IDE developed by JetBrains specifically for Rust, the name of this IDE is RustRover (JetBrains, 2024).

3. IMPLEMENTATION.

3.1 Unit history.

I begin with the implementation of the backend for my application. The first step involves creating a simple REST API with an endpoint that accepts GET requests containing a barcode. A GET request in REST is used to retrieve data from a server. In my case it includes the barcode parameter in the URL. Upon receiving a request, the backend retrieves information related to the unit from the database and returns this data to the user.

3.1.1 Unit history backend implementation.

My starting point will be printing the barcode from the database in the console in an application built with axum. In Fig. 9 I implemented the unit-history endpoint and functionality which prints a barcode to the console.

```rust
async fn handle_unit_history(Query(params): Query<BarcodeQuery>) -> impl IntoResponse {
    println!("Barcode received: {}", params.barcode);
    let out : Stdout = stdout();
    let mut writer : Writer<Stdout> = csv::Writer::from_writer(out);
}

async fn process_unit_history(params: BarcodeQuery) -> Result<(), Box<dyn std::error::Error>> {...}

#[tokio::main]
async fn main() {
    let app : Router = Router::new().route( path: "/unit-history", get(handle_unit_history));

    let addr : SocketAddr = SocketAddr::from( pieces: ([127, 0, 0, 1], 3000));
    let listener : TcpListener = tokio::net::TcpListener::bind(addr).await.unwrap();
    println!("Server listening on {}", addr);

    axum::serve(listener, app.into_make_service())
        .await : Result<()>
        .unwrap();
}
```

Figure 9. Initializing unit history endpoint.

I will attempt to run the application using the "cargo run" command and conduct a brief manual test using the API Client Insomnia. An API client is a tool that permits sending requests and receiving responses from backend, thereby simplifying the

testing and interaction with web services. Insomnia (Insomnia, 2024) is a widely used API client software that offers a user-friendly interface for the creation and testing of HTTP requests, thereby simplifying the process of debugging and developing APIs. When the application is launched, the console indicates that the server is listening on 127.0.0.1:3000. This means that the server is currently running on the local computer and is ready to receive connections at the IP address 127.0.0.1 on port 3000. To test the functionality, one can enter this address to Insomnia and make a test GET request to endpoint /unit-history with an attached barcode, as shown in Figure 10.

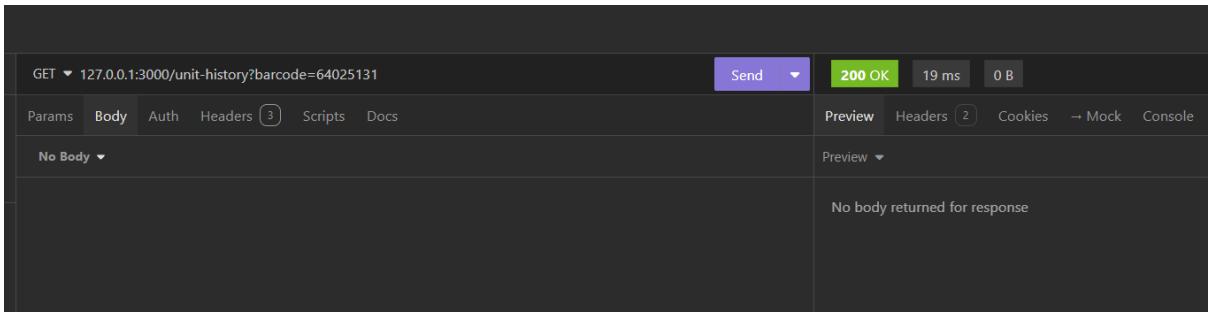


Figure 10. Testing endpoint using Insomnia.

```
Barcode received: 64025131
```

Figure 11. Console output after sending the request to the backend.

After the request is sent, we can see that the backend replied with an ‘OK’ message to Insomnia and the console is showing the barcode in the console that I just sent, as shown in Fig. 11. This means that the service is up and running, and next steps can be implemented.

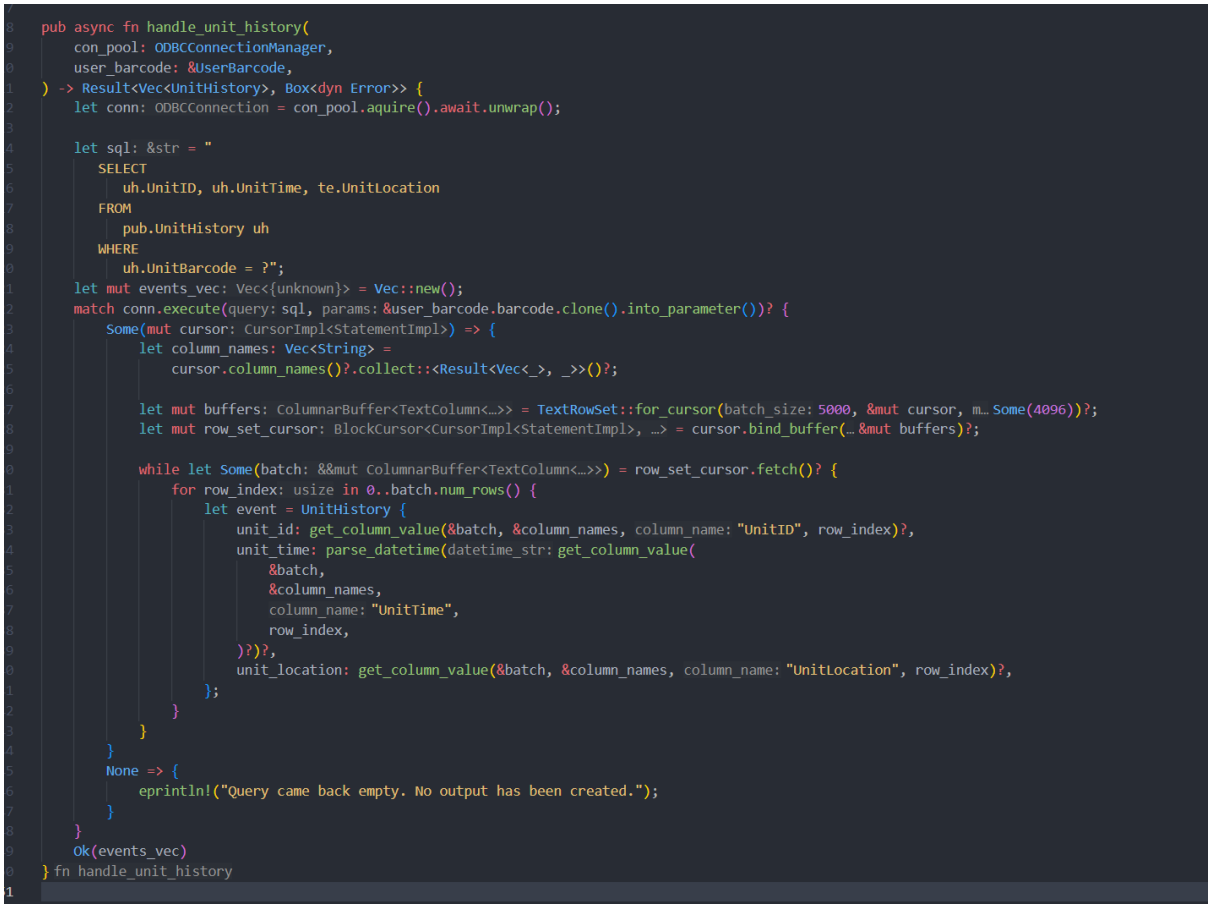
The second step involves utilizing the barcode to retrieve unit history data from the database. To accomplish this, a connection between the backend and the database using ODBC must be established, and logic that includes SQL queries must be developed. The connection string retrieves environment variables for DataSourceName, Username, and Password (Fig.12), ensuring that each of them is

set. Otherwise, a specific ‘panic’ error message will be received. These variables are subsequently used to format a connection string for a Control System database, specifying the driver, server, port, database name, username, and password. This connection string will be used for a database connection in handle_unit_history function, which is shown in Figure 13.

```
let connection_string : String = format!(
    "Driver={{ODBC Driver 18 for SQL Server}};Server=localhost;Port=5432;Database={};Uid={};Pwd={};",
    data_source, username, password
);
```

Figure 12. ODBC connection string.

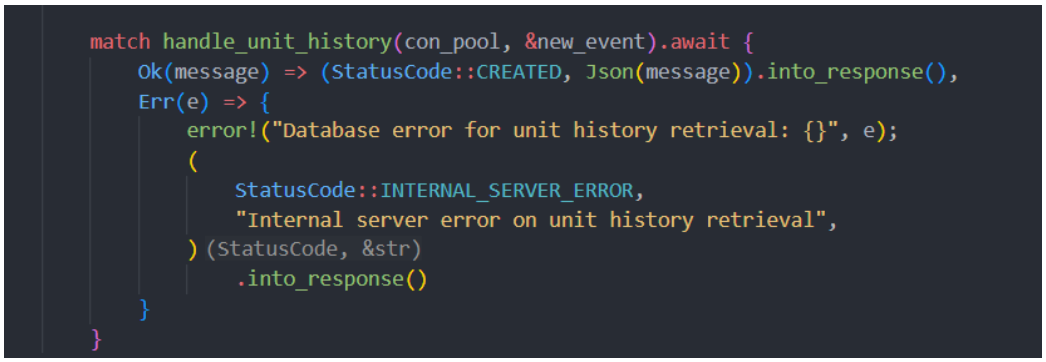
This connection is then used to fetch the history details of the unit from the database. It executes a SQL query with a user-provided barcode, processes the results, and extracts relevant information like UnitID, UnitTime, and UnitLocation from each row. The extracted data is then used to create UnitHistory objects, which are collected into a vector and returned.



```rust
pub async fn handle_unit_history(
    con_pool: ODBCConnectionManager,
    user_barcode: &UserBarcode,
) -> Result<Vec<UnitHistory>, Box<dyn Error>> {
    let conn: ODBCConnection = con_pool.aquire().await.unwrap();

    let sql: &str = "
       SELECT
          uh.UnitID, uh.UnitTime, te.UnitLocation
       FROM
          pub.UnitHistory uh
       WHERE
          uh.UnitBarcode = ?";
    let mut events_vec: Vec<{unknown}> = Vec::new();
    match conn.execute(query: sql, params: &user_barcode.barcode.clone().into_parameter())? {
        Some(mut cursor: CursorImpl<StatementImpl>) => {
            let column_names: Vec<String> =
                cursor.column_names()?.collect::<Result<Vec<_>, _>>()?;

            let mut buffers: ColumnarBuffer<TextColumn<…>> = TextRowSet::for_cursor(batch_size: 5000, &mut cursor, m… Some(4096))?;
            let mut row_set_cursor: BlockCursor<CursorImpl<StatementImpl>, …> = cursor.bind_buffer(… &mut buffers)?;

            while let Some(batch: &&mut ColumnarBuffer<TextColumn<…>>) = row_set_cursor.fetch()? {
                for row_index: usize in 0..batch.num_rows() {
                    let event = UnitHistory {
                        unit_id: get_column_value(&batch, &column_names, column_name: "UnitID", row_index)?,
                        unit_time: parse_datetime(datetime_str: get_column_value(
                            &batch,
                            &column_names,
                            column_name: "UnitTime",
                            row_index,
                        )?)?,
                        unit_location: get_column_value(&batch, &column_names, column_name: "UnitLocation", row_index)?,
                    };
                }
            }
        }
        None => {
            eprintln!("Query came back empty. No output has been created.");
        }
    }
    Ok(events_vec)
} fn handle_unit_history
```

Figure 13. Handling unit history.

If the SQL query returns no results, an error message is logged, and an empty vector is returned, leading to an error reply to the frontend. On successful query execution, the data is converted into UnitHistory objects, serialized into JSON, and returned with an HTTP status code - created as shown in Figure 14.



```rust
    match handle_unit_history(con_pool, &new_event).await {
        Ok(message) => (StatusCode::CREATED, Json(message)).into_response(),
        Err(e) => {
            error!("Database error for unit history retrieval: {}", e);
            (
                StatusCode::INTERNAL_SERVER_ERROR,
                "Internal server error on unit history retrieval",
            ).(StatusCode, &str)
                .into_response()
        }
    }
```

Figure 14. Response to the frontend.

I will now test the same endpoint using Insomnia to verify the response. The expected outcome is a status of “OK”, with the content of the reply containing data that is suitable for display on the frontend.

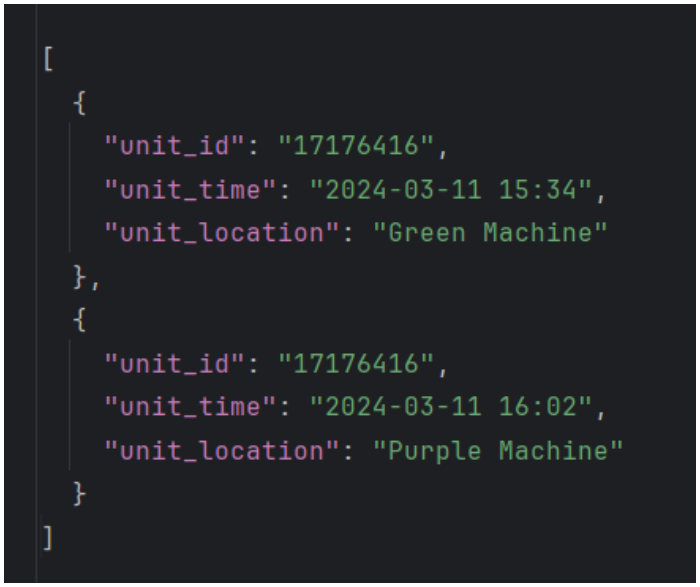


Figure 15. JSON with unit location data.

The data was returned as anticipated. The reply message contains the data which will be shown in frontend, as in Figure 15.

3.1.2 Unit history frontend implementation.

Before displaying the unit history, the barcode scanner must be implemented. A mobile phone camera can scan the barcode and decode it into a numeric format, which will then be wrapped into the JSON and sent to the backend. To implement this, I provide the user with an interface featuring a button that activates the camera for barcode scanning. When the user taps the "Click here to scan barcode" button, the camera activates, allowing them to scan a barcode. Alternatively, the user can manually enter the barcode into a text input field.

Once the barcode is scanned or manually entered, the user can tap the "Request the information" button to submit the barcode data. If the application is currently

processing a request, a loading indicator will be shown instead of the button text. The barcode data will be sent to the backend, where it will be processed to retrieve the unit history information.

The camera view is contained within an animated container, which adjusts its height dynamically based on user interaction. If the camera is active, this view will display the camera feed along with features like autofocus and tapping functionality, allowing the user to focus on a specific area for scanning. Once the barcode is scanned, the camera view will close, and the application will proceed with fetching the relevant data based on the barcode provided.

```
return (
  <View style={styles.container}>
    <View style={styles.sectionUnitHistory}>
      <Pressable style={styles.scanButton} onPress={toggleCamera}>
        <Text style={styles.buttonScanBarcodeText}>
          Click here to scan barcode {'   '}
        </Text>
        <AntDesign name="camera" size={15} color="black" />
      </Pressable>
      <View style={styles.inputRow}>
        <TextInput
          style={styles.input}
          value={inputBarcode}
          onChangeText={setBarcode}
          placeholder="Or type barcode manually"
        />
      </View>
      <Pressable style={styles.button} onPress={handleSubmit} disabled={status === 'loading'}>
        {status === 'loading' ? (
          <ActivityIndicator size="small" color="#FFFFFF" />
        ) : (
          <Text style={styles.buttonRequestTheInformationText}>
            Request the information
          </Text>
        )}
      </Pressable>
      <Animated.View style={[styles.cameraContainer, { height: cameraHeight }]}>
        {cameraVisible && (
          <CameraView
            isRefreshing={isRefreshing}
            onTap={onTap}
            onBarcodeScanned={handleBarcodeScanned}
            focusSquare={focusSquare}
          />
        )}
      </Animated.View>
    </View>
```

Figure 16. Scanner page code.

When the frontend application is started on a mobile phone, the screen waits for a barcode input, as shown in Figure 17.

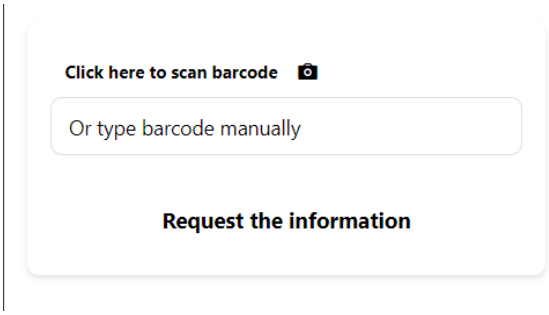


Figure 17. Section of a barcode scanner in a mobile application.

After a barcode is scanned and the "Request the information" button is clicked, the barcode data needs to be sent to the backend to retrieve relevant information. To accomplish this, the application uses a function fetchUnitHistory, which sends the scanned barcode to a specific (unit-history/) endpoint. This function is triggered when the handleSubmit function is called, which happens upon tapping the button. The fetchUnitHistory function [Figure 18] then processes the backend response, updating the application state with the retrieved data and marking the operation as successful. If the data retrieval is successful, the information related to the barcode is stored and can be displayed to the user.

```
export const fetchUnitHistory : AsyncThunk<EventData, string, ... = createAsyncThunk( Show usages
  typePrefix: 'api/fetchUnitHistory',
  payloadCreator: async (inputBarcode: string): Promise<EventData> => {
    const response : AxiosResponse<any, any> = await customFetch.post( url: 'unit-history/', data: {
      barcode: inputBarcode,
    });
    return response.data;
  },
);
```

Figure 18. Fetch unit history code sample.

Once the data is received from the backend, it needs to be displayed to the user in an organized manner. To do this, I implemented the ResultSection component (Figure 19), which checks if the data exists and then displays the unit's barcode and associated information. If the data is available, it is presented through the EventList

component (Figure 20), which renders the events in a scrollable list format. This ensures that users can easily view the detailed information related to the scanned barcode. If no data is available, a message is shown to prompt the user to scan a barcode.

```
<ResultSection barcodeData={show_event_data.barcode_data || 'N/A'} barcode={show_event_data.barcode || 'N/A'}/>
<View style={styles.section}>
  <Text style={styles.header}>Unit history</Text>
  <Text style={styles.barcodeText}>Unit barcode: {show_event_data.barcode || 'N/A'}</Text>
  <EventList barcodeData={show_event_data.barcode_data} />
</View>
```

Figure 19. ResultSection component.

```
<EventList barcodeData={barcodeData} />
```

Figure 20. EventList component

After implementing the above code, the view displays the data by showing the unit's barcode, followed by a list of details related to the unit, allowing users to easily access and review the relevant information retrieved from the backend, as shown in Fig. 21.

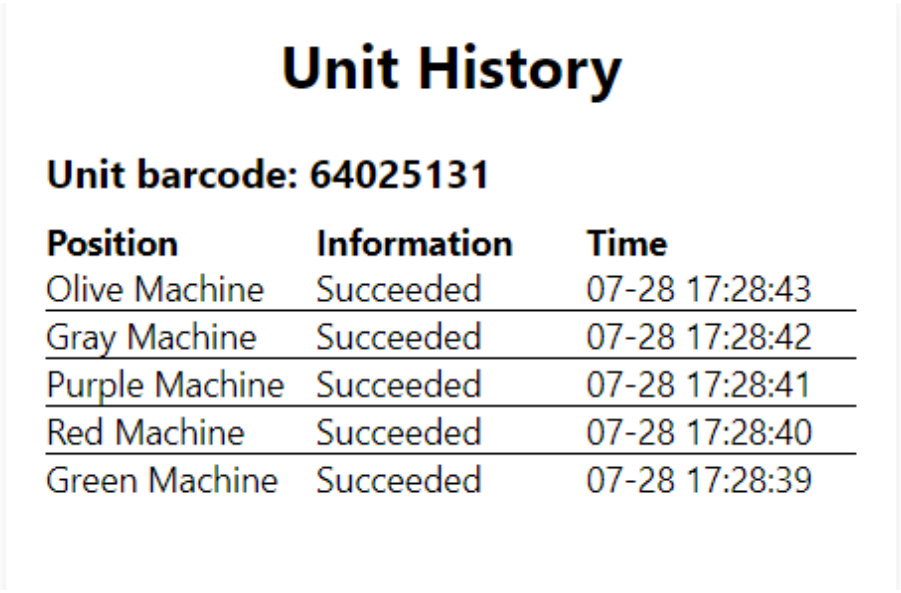


Figure 21. Unit history section in the mobile application.

3.2 Error and status notifications.

Next, I will move on to implementing the notification system. This will involve setting up WebSockets on both the backend and frontend to enable real-time communication. Additionally, I will implement a polling mechanism on the backend that will regularly check the Control System database if the status of a device has changed. If a change in status is detected, this information will be sent to the frontend through the WebSocket connection, where it will be displayed to the user.

3.2.1 Backend WebSocket and polling implementation.

I will implement a function that continuously monitors the status of certain devices in the system. This function will regularly check the database for any changes in the statuses of these devices, which are stored in the Control System Database. During each cycle, the code compares the current statuses with the previous ones [Figure 22]. If a change is detected in any device's status, the function will log this change and prepare a message containing details about the status update.

```
for row :StatusTable in rows {
    if let Some(&previous_status :i32 ) = previous_statuses.get(&row.statusid) {
        if row.status != previous_status {
            let status_msg :String =
                format!("ID: {}, Status changed to: {}", row.statusid, row.status);
            let default_ws_message = DefaultMessage {
                id: Uuid::new_v4(),
                msg: status_msg.clone(),
                createdat: Utc::now(),
                msgtype: "Status update".to_string(),
            };
```

Figure 22. Status checking.

Figure 23. Converting Status data to JSON and sending via WebSocket to the frontend.

The updated status information is then sent to users through WebSocket communication, ensuring they are immediately informed of any changes (Figure 23). The function also stores these status update messages in the PostgreSQL database for record-keeping. This allows users who were not connected to the WebSocket at the time of the update to be notified when they reconnect. This entire process repeats every second to ensure the system is always up-to-date with the latest device statuses.

3.2.2 Testing WebSocket connection using wscat.

To manually test the WebSocket connection, I will use a tool called wscat. Wscat is a command-line utility that allows interacting with backend WebSocket by sending and receiving messages directly in the terminal (Amazon Web Services, 2024). This tool is useful for verifying that the WebSocket communication is working correctly by simulating a frontend connection and observing how the server responds in real-time. Additionally, I will manually change the status values in the Control System database to trigger updates, allowing me to confirm that these changes are correctly detected and communicated through the WebSocket.

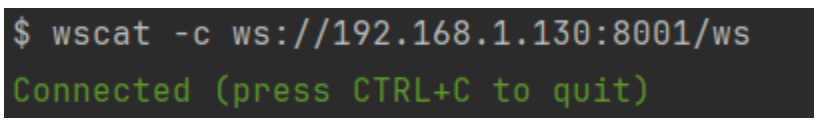


Figure 24. Command to connect to the backend WebSocket.

Figure 25. Console output in wscat.

The connection was successful (Figure 24), and when I manually changed the status, I was notified via WebSockets with a message, as shown in Figure 25.

3.2.3 Fontend notification implementation.

Notifications should be displayed in both the web browser and mobile applications. To accomplish this, I will implement a system that listens for real-time updates and sends notifications to users.

The WebSocketProvider component establishes the WebSocket connection and monitors its status, ensuring that the app knows whether the connection is active or not. The corresponding function is shown in Figure 26.

```
// Custom hooks
function useWebSocketConnection(  Show usages
  setIsConnected: React.Dispatch<React.SetStateAction<boolean>>,
) : void  {
  useEffect( effect: () => {
    const handleOpen = () => setIsConnected( value: true);  Show usages
    const handleClose = () => setIsConnected( value: false);  Show usages

    webSocketService.on( event: 'open', handleOpen);
    webSocketService.on( event: 'close', handleClose);

    webSocketService.connect( url: 'ws://192.168.1.130:8001/ws');

    return () : void  => {
      webSocketService.off( event: 'open', handleOpen);
      webSocketService.off( event: 'close', handleClose);
      if (webSocketService.socket) {
        webSocketService.socket.close();
      }
    };
  }, deps: [setIsConnected]);
}
```

Figure 26. Code for establishing WebSocket connection in the frontend.

When messages are received, the useWebSocketMessageHandler hook parses these messages and dispatches them as notifications, see Figure 27. Depending on the platform (web or mobile), the messages are either shown as native notifications on mobile devices or browser notifications on web platforms.

Figure 27. Code sample of useWebSocketMessageHandler hook.

The useNotificationHandler hook ensures that mobile notifications are handled correctly, including showing alerts, playing sounds, and updating badges, as implemented by a function shown in Figure 28.

Figure 28. Code sample of notification handler.

This setup allows the application to deliver real-time notifications seamlessly to users, whether they use a web browser or a mobile device. After this, the notifications appear on mobile phones, as shown in Figure 29, and in web browsers, as shown in Figure 30.

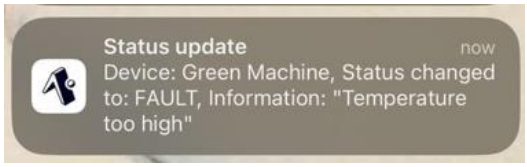


Figure 29. Notification in the iOS device.

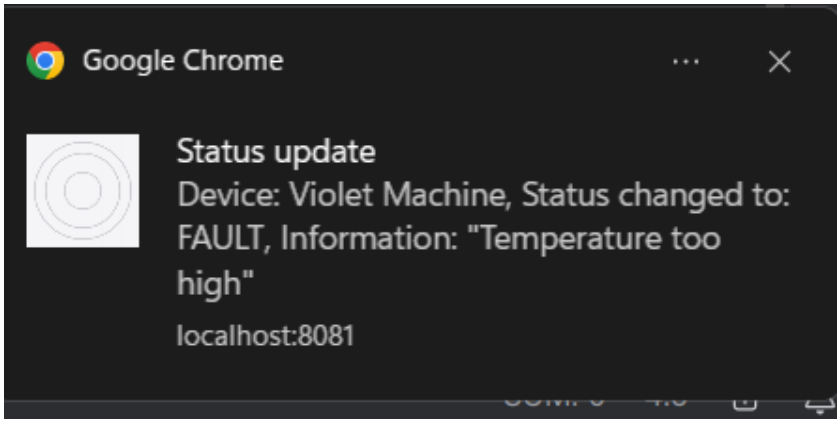


Figure 30. Notification in Google Chrome web browser.

3.3. Show product information.

Each unit is associated with a specific product, which may have attributes such as brand or type. For example, if the product is food, it could be a banana from a particular brand or an orange from another brand. This information is crucial for warehouse staff, especially when a unit fails to move through a designated position or triggers an error. By verifying the product details, such as the brand, type, and dimensions, the staff can ensure that the correct product is being handled and that its specifications meet the required standards. I will begin with the backend implementation.

3.3.1. Unit product backend implementation.

This section's backend implementation will be executed in a manner that is very similar to the manner in which I previously implemented the unit history feature. To guarantee consistency in how the system is built and functioning, I will adhere to the same procedures and methodologies that I have previously used. Since the unit history feature was already successfully implemented, I'll apply the same approach here to achieve similar results. I will need to modify the endpoint [Figure 31] and adjust the SQL query accordingly [Figure 32].

```
.route( path: "/product", get(handle_product))
```

Figure 31. Code sample of the route to product.

```
SELECT id, barcode, product_type, product_brand, product_size, weight, information
FROM product
WHERE barcode = $1
```

Figure 32. SQL Query to fetch the product data.

The backend implementation for the product feature is now complete, and I will proceed with the frontend development.

3.3.2 Unit product frontend implementation.

In the frontend, I will create a section that displays detailed product information, such as the product type, size, brand, and weight, based on the scanned barcode. The system will also check whether the product's weight falls within an acceptable range. If the weight exceeds the maximum limit, the product will be highlighted. In Figure 34, the ProductSection component is responsible for displaying detailed product information based on a scanned or manually entered barcode.

```
function ProductSection({ barcode, productData, onRequestProduct, isLoading }: ProductSectionProps) {
  const hasData = barcode && productData && Object.keys(productData).length > 0;
  const minWeight : 10  = 10;
  const maxWeight : 20  = 20;
  const getWeightColor = (weight: number) : string  => {  Show usages
    return weight > maxWeight ? 'red' : 'black';
  };
  return (
    <View style={styles.section}>
      <Text style={styles.header}>Product Information</Text>
      {hasData ? (
        <>
          <Text style={styles.barcodeText}>Product barcode: {barcode}</Text>
          <Text>Type: {productData.product_type}</Text>
          <Text>Size: {productData.product_size}</Text>
          <Text>Brand: {productData.product_brand}</Text>
          <Text style={{ color: getWeightColor(productData.weight) }}>
            Weight: {productData.weight}
          </Text>
          <Text>Acceptable weight range: {minWeight} - {maxWeight}</Text>
          {productData.weight > maxWeight && (
            <Text style={{ color: 'red' }}>Warning: Weight exceeds maximum limit!</Text>
          )}
```

Figure 34. ProductSection component.

If the product data is available, key details such as product type, size, brand, and weight will be shown, along with warnings if the weight exceeds a specified limit. If no product data is available, it provides an option for the user to request this information, displaying a loading indicator while the request is being processed. An example of unit's information is shown in Figure 35.

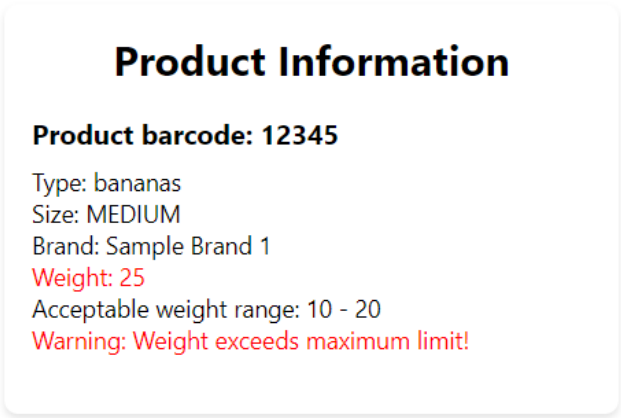


Figure 35. Product information section in the mobile application.

3.4. Header and menu tabs.

In the next part of the implementation, I add a header and menu tabs to the application. The header will allow users to easily identify the current page and monitor the connection status. The menu tabs will enable users to navigate across different sections of the app, such as the scanner page and the status page. This addition will enhance the user experience by providing clear navigation and real-time feedback on the app's status.

To implement the tab navigation, I will use the Tabs layout provided by Expo Router (Expo documentation). The (tabs) directory is a special directory that Expo Router recognizes as the layout for tab-based navigation. Within this directory, the main layout file (tabs)/_layout.tsx will define the appearance and behavior of the tab bar and its buttons. I configure the tabs to display different sections of the app, such as "Home" and "Settings," each represented by an icon and a title. By using the structure of Fig. 36, I ensure that the user interface is intuitive and allows users to easily navigate between different parts of the application.

```
<Tabs.Screen
  name="index"
  options={{
    title: 'Scanner',
    headerShown: true,
    headerTitleAlign: 'left',
    tabBarIcon: ({ color : string  }) => (
      <FontAwesome size={28} name="qrcode" color={color} />
    ),
  }}
/>
<Tabs.Screen
  name="statusesanderrors"
  options={{
    title: 'Statuses/Errors',
    headerShown: false,
    tabBarIcon: ({ color : string  }) => (
      <FontAwesome size={28} name="envelope" color={color} />
    ),
  }}
/>
```

Figure 36. Sample code of Tabs.

Figure 37 illustrates the code designed to visually indicate the status of a WebSocket connection. It evaluates the isConnected boolean to determine whether the user is connected or not. If the user is connected, a green status indicator with the text "Status: Connected" is displayed, signaling an active connection. If the user is not connected, a red status indicator with the text "Status: Disconnected" is shown. This visualization allows users to easily understand their connection status, and it can be implemented to provide real-time feedback in the user interface.

```
<View style={styles.statusContainer}>
  {isConnected ? (
    <LinearGradient
      colors={['#28a745', '#218838']}
      start={[0, 0]}
      end={[1, 0]}
      style={styles.statusIndicator}>
      <Text style={styles.statusText}>Status: Connected</Text>
    </LinearGradient>
  ) : (
    <LinearGradient
      colors={['#ae2121', '#ec0516']}
      start={[0, 0]}
      end={[1, 0]}
      style={styles.statusIndicator}>
      <Text style={styles.statusText}>Status: Disconnected</Text>
    </LinearGradient>
  )}
</View>
```

Figure 37. Code sample of the implementation of the status of a WebSocket connection.

The header with the connection status is illustrated in Figure 38. The tab bar is shown in Figure 39.

Figure 38. Status of the Websocket connection in the mobile application.

Figure 39. Menu tabs in the mobile application.

3.5. A status page.

The status page functions similarly to that of the popular messaging apps like WhatsApp. The page keeps users constantly informed about the latest updates and activities happening in the warehouse. Users will be able to open any chat and instantly check the latest data, ensuring they are always up to date with critical information. To achieve this, I integrated a chat interface that automatically loads and updates messages in real-time as they become available. This approach will make it easy for users to monitor the warehouse status and communicate effectively in a seamless and intuitive interface.

3.5.1. Backend implementation.

To achieve the result described above, I implemented a feature which automatically sends a request to the backend server whenever a user opens the chat application. This request retrieves the most recent messages stored in the internal PostgreSQL database. The backend queries the database for these messages and returns them to the user, ensuring that the latest communication history is displayed. In Fig. 40 the get_messages function retrieves messages from a PostgreSQL database and returns them to the client. The fetch_messages function queries the database for messages, ordering them by the creation date. In Fig. 41 the route .route("/messages/", get(get_messages)) connects the /messages/ endpoint to the get_messages function, ensuring that this function is executed when the endpoint is accessed.

Figure 40. Fetching messages function, which contains SQL query.

Figure 41. Messages endpoint route.

Furthermore, when the chat page is already open, any new messages will be delivered instantly through WebSocket connections. This ensures that users see new messages in real-time without needing to refresh the page.

3.5.2 Frontend implementation.

Instead of displaying individual contacts like in WhatsApp (WhatsApp, 2024) or other chat applications, this interface will present different chat channels corresponding to specific warehouse events, such as "broken products" or "failed products". When the user opens one of these chat channels, the frontend will send a request to the backend server to retrieve the latest messages stored in our PostgreSQL database, ensuring the user always sees the most recent updates.

Furthermore, if the user is already viewing a chat, any new messages will be delivered in real-time via WebSocket connections, allowing them to see updates instantly without needing to refresh the page. The chat interface will be styled to enhance user experience, with clear chat bubbles and formatted timestamps, making it easy to stay informed about important warehouse events through a familiar, easy-to-use chat format. In Fig. 42 the GiftedChat component is used to create a chat interface that displays messages dynamically. It leverages the Bubble component to

style individual chat messages. This setup provides a user-friendly and visually distinct chat experience, ensuring messages are clearly presented and easy to read.

```
<GiftedChat
  messages={imessages}
  user={{
    _id: 1,
  }}
  bottomOffset={insets.bottom}
  renderAvatar={null}
  maxComposerHeight={100}
  renderTime={renderCustomTime}
  renderBubble={props : Readonly<BubbleProps<IMessage>...  => (
    <Bubble
      {...props}
      textStyle={{
        right: {
          color: '#000',
        },
      }}
      wrapperStyle={{
        left: {
          backgroundColor: '#fff',
        },
        right: {
          backgroundColor: 'orange',
        },
      }}
    />
  )}
```

Figure 42. GiftedChat component code sample.

Figure 43 shows the status page that contains events grouped according to their status. Figure 44 shows an example of a chat window with status messages.

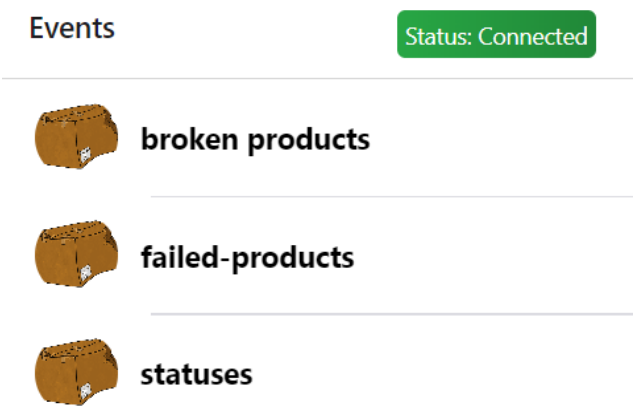


Figure 43. Status page in the mobile application.

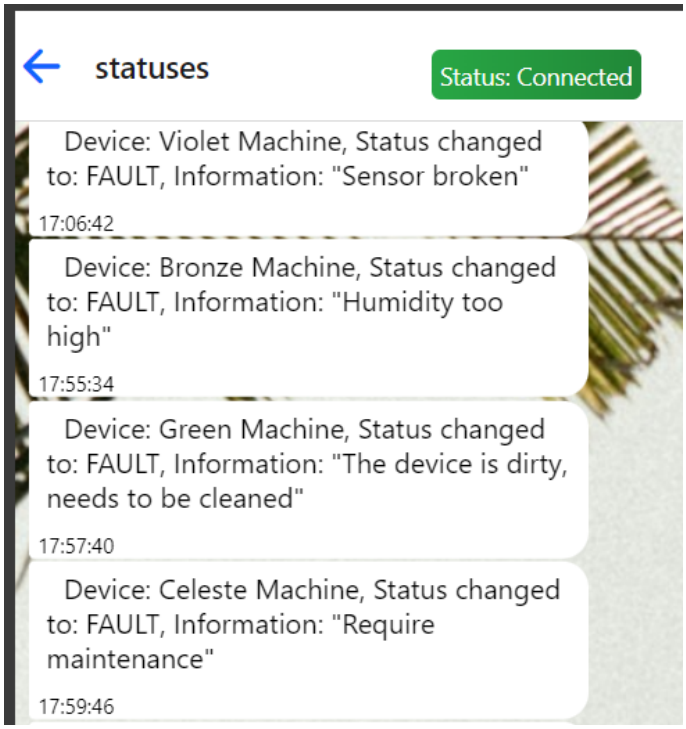


Figure 44. Status chat in the mobile application.

With this, the implementation of the basic functionality of the app is complete.

4. CONCLUSION.

This thesis project aimed to achieve several objectives specified in the introduction. The primary goal was to develop software to improve the efficiency of automated distribution processes in automated warehouses by using a mobile application, as a replacement or in combination with the stationary user interface (UI).

Developing the backend for the new application required a comprehensive understanding of the legacy software architecture, as the warehouse data is stored in a legacy system database. In addition, new data was integrated into this system to guarantee compatibility and efficient data retrieval by the newly implemented backend.

Subsequently, the application design phase commenced. New technologies that required learning were selected, including Expo and Rust with Axum. Expo was chosen to facilitate cross-platform development, while Rust with Axum was selected for backend implementation. Both TypeScript and PostgreSQL were utilized due to prior familiarity, which expedited the project's completion.

The objectives were achieved in this thesis. The implemented mobile application demonstrated that the chosen technology stack is entirely suitable for creating a larger, full-fledged application. Cross-platform development using Expo exceeded initial expectations, revealing that the platform's tools are easy to integrate and well-documented. Many layouts, such as tabs, had pre-existing implementations, requiring only the addition of specific details like titles and colors. Implementing REST and WebSocket interfaces using Axum and Rust was not overly complicated, owing to the availability of pre-existing implementations and examples.

Migrating from a stationary UI to a mobile application offers multiple benefits. The mobile software can collect important information about the warehouse and potential failures within the warehouse and immediately send any error messages via push notifications to the user.

Using a mobile application improves accessibility to the warehouse state, significantly speeding up issue recognition times and provides the delivery of more detailed information, as each operator can use their own device to observe relevant data.

Furthermore, role based information can be targeted to operators in specific areas who need to be aware of specific issues. This approach also unbinds operators from location constraints - they can be more mobile in their movements around the warehouse, as there is no longer a need to remain near a stationary computer.

REFERENCES

Website of Wikipedia, Frontend and backend. Referred 15.6.2024. https://en.wikipedia.org/wiki/Frontend_and_backend

Matias Martinez, Bruno Gois Mateus. (2021) Why did developers migrate Android applications from Java to Kotlin? https://arxiv.org/pdf/2003.12730

Website of Apple, Swift. Referred 15.6.2024. https://developer.apple.com/swift/

Ruben Horn, Abdellah Lahnaoui, Edgardo Reinoso, Sicheng Peng, Vadim Isakov, Tanjina Islam, Ivano Malavolta. 2023. Native vs Web Apps: Comparing the Energy Consumption and Performance of Android Apps and their Web Counterparts. Referred 16.6.2024. https://arxiv.org/pdf/2308.16734

Website of GeeksForGeeks. 2024. Referred 16.6.2024. https://www.geeksforgeeks.org/what-is-syntax-components-rules-and-common-mistakes/

Website of Flutter. 2024. Referred 16.6.2024. https://flutter.dev/

Kavitha Marimuthu, Arunkumar Panneerselvam, Senthilkumar Selvaraj, Lakshmi Praba Venkatesan, Vetriselvi Sivaganesan. 2023. Android Based College App Using Flutter Dart. Referred 17.6.2024. http://dx.doi.org/10.53623/gisa.v3i2.269

Website of React Native. 2024. Referred 17.6.2024. https://reactnative.dev/

Website of GeeksForGeeks. Difference between hot reloading and live reloading in React Native. 2024. Referred 17.6.2024. https://www.geeksforgeeks.org/difference-between-hot-reloading-and-live-reloading-in-react-native/

Website of GeeksForGeeks. What is a bridge in React Native. 2022. Referred 17.6.2024. https://www.geeksforgeeks.org/what-is-a-bridge-in-react-native/

Website of Rust programming language. 2024. Referred 17.6.2024. https://www.rust-lang.org/

Boqin Qin, Yilun Chen, Haopeng Liu, Hua Zhang, Qiaoyan Wen, Linhai Song, Yiying Zhang. Understanding and Detecting Real-World Safety Issues in Rust, 2021. Referred 18.6.2024. https://doi.org/10.1109/TSE.2024.3380393

Website of StackOverFlow. 2024 develop survey. 2024. Referred 18.6.2024. https://survey.stackoverflow.co/2024/technology/

William Bugden and Ayman Alahmar. Rust: The Programming Language for Safety and Performance. 2022. Referred 19.6.2024. http://dx.doi.org/10.48550/arXiv.2206.05503

Website of InfinyOn. Java vs. Rust Comparison. 2024. Referred 19.6.2024. https://www.infinyon.com/resources/files/java-vs-rust.pdf

Website of Wikipedia. Memory safety. 2024. Referred 19.6.2024. https://en.wikipedia.org/wiki/Memory_safety

Hui Xu, Zhuangbin Chen, Mingshen Sun, Michael R. Lyu. Memory-Safety Challenge Considered Solved? An In-Depth Study with All Rust CVEs. 2021. Referred 20.6.2024. https://arxiv.org/pdf/2003.03296

Website of The White House. Back to the building blocks: a path toward secure and measurable software. 2024. Referred 20.6.2024. https://www.whitehouse.gov/wp-content/uploads/2024/02/Final-ONCD-Technical-Report.pdf

Website of Wikipedia. Rust programming language. 2024. Referred 21.6.2024. https://en.wikipedia.org/wiki/Rust_(programming_language)

Website of Lenovo. Compilation. 2024. Referred 21.6.2024. https://www.lenovo.com/us/en/glossary/compilation/

Website of blogs for Rust programming language. Fearless concurrency. 2015. Referred 21.6.2024. https://blog.rust-lang.org/2015/04/10/Fearless-Concurrency,

Website of documentation for Rust programming language. Keyword unsafe. 2024. Referred 21.6.2024. https://doc.rust-lang.org/std/keyword.unsafe.html

Website of Microsoft Security Response Center. A proactive approach to more secure code. 2019. Referred 21.6.2024. https://msrc.microsoft.com/blog/2019/07/a-proactive-approach-to-more-secure-code/

Website of Lenovo. Concurrency. 2024. Referred 22.6.2024. https://www.lenovo.com/in/en/glossary/concurrency

Shixiong Zhao, Rui Gu, Haoran Qiu, Tsz On Li, Yuexuan Wang, Heming Cui, and Junfeng Yang. OWL: Understanding and Detecting Concurrency Attacks. 2018. Referred 22.6.2024. https://doi.org/10.1109/DSN.2018.00033

Website of PullRequest. Rust Safety: Writing Secure Concurrency without Fear. 2024. Referred 22.6.2024. https://www.pullrequest.com/blog/rust-safety-writing-secure-concurrency-without-fear/

Website of European Commission. Energy efficiency targets. Referred 23.6.2024. https://energy.ec.europa.eu/topics/energy-efficiency/energy-efficiency-targets-directive-and-rules/energy-efficiency-targets_en

Rui Pereira, Marco Couto, Francisco Ribeiro, Rui Rua, Jácome Cunha, João Paulo Fernandes, João Saraiva. Energy efficiency across programming languages: how do energy, time, and memory relate? 2017. Referred 23.6.2024. http://dx.doi.org/10.1145/3136014.3136031

Website of Intel. Accelerating Cloud Native Apps Written in Rust. Referred 23.6.2024. https://www.intel.com/content/www/us/en/developer/articles/technical/accelerating-cloud-native-apps-written-in-rust.html

Luca Palmieri. Zero to production in Rust. Referred 24.6.2024. https://www.zero2prod.com/

Website of Actix Web. 2024. Referred 24.6.2024. https://actix.rs/

Website of Tokio-rs blog. Announcing Axum. 2021. Referred 24.6.2024. https://tokio.rs/blog/2021-07-announcing-axum

Website of Oracle. What is a database? Referred 25.6.2024. https://www.oracle.com/database/what-is-database/

Qinggang Zhang, Junnan Dong, Hao Chen, Wentao Li, Feiran Huang, Xiao Huang. Structure Guided Large Language Model for SQL Generation. 2024. Referred 26.6.2024. https://arxiv.org/pdf/2402.13284

Svetlana Andjelic, Slobodan Obradovic, Branislav Gacesa. A Performance Analysis of the Dbms - MYSQL Vs POSTGRESQL. 2008. Referred 26.6.2024. http://dx.doi.org/10.26552/com.C.2008.4.53-57

Website of MySQL. Products. Referred 26.6.2024. https://www.mysql.com/products/

Website of PostgreSQL. Referred 26.6.2024. https://www.postgresql.org/

Website of Wikipedia. Open source. 2024. Referred 26.6.2024. https://en.wikipedia.org/wiki/Open_source

Web site of Amazon Web Services. What's the Difference Between MySQL and PostgreSQL? 2024. Referred 26.6.2024. https://aws.amazon.com/compare/the-difference-between-mysql-vs-postgresql/

Website of Progress Software. What is an ODBC Driver? 2024. Referred 27.6.2024. https://www.progress.com/faqs/datadirect-odbc-faqs/what-is-an-odbc-driver

Website of GitHub. ODBC-API repository. 2024. Referred 27.6.2024. https://github.com/pacman82/odbc-api

Website of GitHub. SQLX repository. 2024. Referred 27.6.2024. https://github.com/launchbadge/sqlx

Stefan Karlsson, Robbert Jongeling, Adnan Cauˇsevi´c, Daniel Sundmark. Exploring Behaviours of RESTful APIs in an Industrial Setting. 2023. Referred 28.6.2024. https://arxiv.org/pdf/2310.17318

Website of Expo. 2024. Referred 28.6.2024. https://expo.dev/

Website of JSON. 2024. Referred 29.6.2024. https://www.json.org/json-en.html

Website of Amazon Web Services. What is an IDE (Integrated Development Environment)? 2024. Referred 30.6.2024. https://aws.amazon.com/what-is/ide/

Website of Insomnia. 2024. Referred 30.6.2024. https://insomnia.rest/

Website of Expo documentation. Tabs. 2024. Referred 1.8.2024. https://docs.expo.dev/router/advanced/tabs/

Website of WhatsApp. 2024. Referred 2.8.2024. https://www.whatsapp.com/

Website of Amazon Web Services. What is Amazon API Gateway? 2024. Referred 2.8.2024. https://docs.aws.amazon.com/apigateway/latest/developerguide/apigateway-how-to-call-websocket-api-wscat.html

Website of Jetbrains. RustRover A powerhouse IDE for Rust developers. 2024. Referred 11.08.2024. https://www.jetbrains.com/rust/