Aurelio Menendez

DEVELOPING REMOTE DIAGNOSTICS PLATFORM FOR DEVICE VAULT

Building a Scalable Remote Diagnostics Demo

School of Technology and Communication

2025

VAASAN AMMATTIKORKEAKOULU
UNIVERSITY OF APPLIED SCIENCES
Information Technology

ABSTRACT

Author	Aurelio Menendez
Title	Developing Remote Diagnostics Platform for Device Vault : Building a Scalable Remote Diagnostics demo
Year	2025
Language	English
Pages	23
Name of Supervisor	Anna-Kaisa Saari

This thesis focuses on the design and implementation of a proof of concept of a modular remote diagnostics system that complements proprietary industrial software. The solution was built on a microservices architecture using Java Spring Boot, AMQP was used for asynchronous service communication.

In the developed system, real-time data is collected from sensor-equipped edge devices, that are connected via serial and TCP communication. Data is then analyzed within a centralized data-processing service. Device integration, fault simulation, and live heartbeat tracking are supported, allowing for dynamic device state management and notification system.

A React-based web dashboard provides real-time monitoring and control. The system enables secure, scalable, and maintainable diagnostics workflows across services such as monitoring, and notification handling. The proof-of-concept implementation demonstrates how standard open-source technologies can be designed to deliver reliable remote insights into operational equipment, making it a viable foundation for future development in industrial diagnostics.

Keywords cloud services, remote access, diagnostics, wireless data transmission

CONTENTS

FIGURES

ABBREVIATIONS

AMQP	Advanced Message Queueing Protocol
DV	Device Vault
JVM	Java Virtual Machine
TCP	Transmission Control Protocol
REST	Representational State Transfer
API	Application Programming Interface

1. INTRODUCTION

This chapter provides the necessary contexts to understand the initial background of the project, the problem it aims to address and how it intends to complement the current implementation of DeviceVault. DeviceVault is a cloud platform that provides seamless application testing across different hardware and software versions.

1.1 Project Background

The fast development and introduction of new technologies has significantly transformed our modern world. Throughout the last decades we have witnessed changes in various industries and sectors, and how the development of specialized software solutions has been a key driver of innovation in dozens of markets and industries, such as finance, healthcare, automotive, manufacturing and many others.

Across all the previously mentioned industries, companies share a common objective: maximizing their operating cash flow. To achieve this, they face continuous pressure to reduce costs without compromising operational efficiency. One of the major challenges they face is the risk of unexpected mechanical failures, which can result in unplanned downtime, increase of expenses and disruptions of operations. Failing to service equipment within the appropriate time frame only increases the likelihood of failures, resulting in higher repair costs and relevant losses in productivity. To mitigate these risks, business should adopt proactive maintenance that optimizes equipment reliability and minimizes unexpected downtime.

The main objective of this thesis is to develop a proof-of-concept that includes the implementation of a real-time monitoring system and data log retrieval. By developing this feature, the project aims to identify potential challenges and limitations related to remote diagnostics. The result of this project will be used for further feature development and gather feedback from potential users within DeviceVault.

1.2 Usage of AI

During the project AI tools such as ChatGPT and GitHub copilot were used to improve written sentences in situations where the input would use informal language. For example, the last paragraph of Chapter 6 was improved by the following prompt "Given the following paragraph - *Ultimately during the presentation of the project expected problems came out to the light and became evident, making evident the need of improvement and validation of the approach* – improve the use of informal language to be used in an academic context", resulting into the improved version. Additionally, code snippets used in Figures 6 and 8 were generated by GitHub Copilot with the prompt "Given the method/entity used in lines X and entity Y generate generic methods to be used in documentation". Finally, after a source was reviewed a prompt like "Given the following URL, generate a refence compliant with APA 7 specifications" resulting in the references used in the thesis work after verifying that the results were compliant with the guidelines.

2 THEORETICAL BACKGROUND

This chapter provides the necessary context to understand the relevance and use cases of the project presented in this thesis. It will provide an overview of the main objective of remote diagnostics and how this technology allows businesses to improve operational efficiency without needing to be physically present.

2.1 Current Market Situation

The demand for cost-effective solutions has driven companies to explore new technologies, including remote monitoring. However, cost reduction is not the only factor that is encouraging the rapid development of these types of solutions. The increasing availability of internet connectivity for industrial devices has played a crucial role in the expansion of this market. Figure 1 (Ritchie, Mathieu, Roser, & Ortiz-Ospina 2023) shows the drastic increase of users connected to the internet over the last two decades, highlighting the growing potential for remote monitoring solutions due to the availability of network infrastructure.

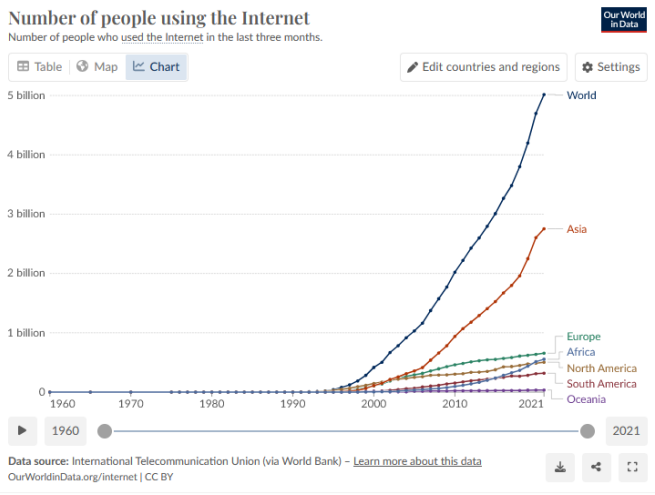

Figure 1. Number of people using the internet.

Additionally, the number of connected IoT devices was projected to grow by 13% to 18.8 billion by the end of 2024, reinforcing the scale and relevance of these technologies in modern monitoring systems (Sinha, 2024).

2.2 Remote Diagnostics

Remote diagnostics is a solution to monitor the status of equipment using a set of tools and communication protocols to enable the connection between remote devices and servers. These systems collect valuable information and operational data (Cummins, 2023). This data can be used for multiple purposes, such as creating systems that improve faster response time when equipment meets technical issues, predictive analysis about equipment health and condition and technical support without the waiting times for experts to evaluate the equipment on site.

Companies have developed various remote monitoring and diagnostic solutions for their products. These technologies help address the needs for gathering data. For example, Fortum offers remote monitoring software for their wind turbines, emphasizing the benefits of automated data collection and anomaly detection as a tool to decrease downtime and overhaul cost efficiently (Fortum, 2020).

This approach demonstrates how real-time monitoring can proactively address equipment failures, reducing the need for manual inspection and on-site interventions, especially when wind turbine clusters are in remote and difficult access locations. Similarly, Cummins Engines, a leading diesel engine and generator manufacturer, has successfully implemented remote diagnostics to enhance fleet engine management. Their system minimizes downtime by filtering data and identifying critical issues. For instance, in a fleet of 5000 vehicles, a customer might receive alerts for only 100, as the monitoring system can accurately diagnose and filter out the irrelevant information. This technology ensures consistently high level of diagnostic accuracy across the entire fleet (Cummins, 2023). By taking advantage of data analysis, Cummins ensures that fleet managers receive only relevant alerts, allowing their customers a better resource allocation and preventative maintenance planning.

3 USED TECHNOLOGIES

This chapter reviews the technologies used in the development of the project. The first section provides an overview of the software tools employed for both frontend and backend development. Additionally, the chapter presents the hardware components used to simulate live equipment behavior.

3.1 Software

The project consists of three main components of software: backend, frontend and edge device. The backend was implemented using Java with the Spring Boot framework. Java is a programming language widely used in enterprise applications and mobile applications. One important benefit that Java possess is that compiled code can be run on any device with the assistance of a Java Virtual Machine (Oracle, n.d.). The JVM can be described as a middleware that converts compiled byte code into native binary code. Spring boot is a framework designed to simplify the development of Java code (VMware, n.d.). This came useful when implementing an API Gateway service. When working with spring-based applications, automation tools are required. For that purpose, Maven was chosen. Maven is an open-source project that simplifies the project setup, dependency management and building process (The Apache Software Foundation, n.d.).

The front-end was built using React with TypeScript and Vite. React is an open-source JavaScript framework developed by Meta (Meta, n.d.). It is commonly used to create user interfaces using a modular component approach. Many leading industries use React framework for their applications, and since DeviceVault user interface is also based on React, it was the right choice to select this technology for developing the proof-of-concept.

Finally, the hardware or edge device, was developed using C++. C++ is a programming language widely used in embedded systems (Bloodshed Software, n.d.). It allows for precise control over system resources, which is important when working with microcontrollers with limited memory like the Arduino Uno.

3.2 Hardware

The hardware utilized for this project consists of primarily two components: the Raspberry-pi Zero and a Grove Beginner Kit for Arduino. These devices were chosen to replicate the behavior of edge devices and data collection of sensors. The Raspberry Pi Zero served as the intermediate gateway between the Arduino Uno and the cloud-based microservices. It provided the computational resources necessary to handle serial communication, data processing, and securely forward messages via TCP.

Finally, the Grove kit functioned as the primary data acquisition unit in the system. It interfaced directly with a variety of sensors including temperature, light, and sound modules as well as output components for signaling and control. The kit handled real-time measurements and low-level control actions such as triggering alarms or visual indicators in response to specific sensor readings or simulated failure events. Additionally, it maintained a consistent stream of heartbeat messages to signify operational status. The firmware was programmed to encode sensor data into compact, JSON-like strings, which were transmitted over the serial connection to the Raspberry Pi for further processing.

3.3 Additional Components

Other tools that played a crucial role during the implementation of the project were Docker, OpenZiti, RabbitMQ and PostgreSQL. These contributed to different aspects of system deployment, security, and data communication. The communication within the system was mostly done via AMQP. AMQP is a messaging protocol that allows the communication of client applications with via middleware brokers (RabbitMQ, n.d.). Middleware broker in the project is RabbitMQ.

Docker was extensively used to containerize the application. Each service was encapsulated, defined and built using dedicated Dockerfiles. Docker Compose files were utilized to orchestrate multi-container setups during development, allowing a configuration of environment variables, and restricted port access to enhance the security. This setup not only facilitated consistent builds and deployment across environments but also enabled the generation of Kubernetes YAML files directly from the Docker configurations. This setup can be used further on to create a solid and scalable foundation.

OpenZiti was incorporated to improve the security and resilience of network communications, especially in edge-to-cloud scenarios. Unlike traditional VPNs or firewalls, OpenZiti enables a zero-trust model (OpenZiti, n.d.), in which access is granted only to explicitly authorized devices and applications. The zero-trust approach eliminates the need to expose backend services to public networks providing secure communication channels without the need to use port forwarding. The use of OpenZiti in the project enhanced the overall security posture of the system, making it more suitable for real-world deployment in industrial or remote environments. Finally, PostgreSQL was used as the database system responsible for storing structured data such as device information, sensor readings and alerts. Together, these tools form the base of data handling flow, ensuring smooth interaction between components and long-term data storage.

4 REAL TIME MONITORING

This chapter provides a detailed description of each individual component, their roles, and how they interact within the system. It also explains how these interconnected services work together to fulfill the objectives defined in this thesis.

4.1 Architectural Design

In software development, an architectural pattern is a reusable solution to a recurring problem at the architecture level. These patterns provide a blueprint for organizing software systems and guide how components should interact to achieve desired attributes such as scalability, maintainability, fault tolerance, or separation of concerns.

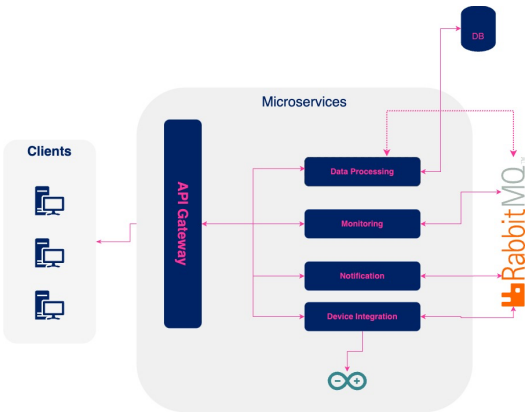

Figure 2. Initial architectural design

This project evolved through two relevant design changes: an initial solution, which is presented in Figure 3 where no edge devices were expected to be implemented, and a final solution that required the implementation of OT-EDGE service due to the need of demonstrating the full monitoring cycle, from data collection to display it remotely.

As requirements changed during the implementation of the project, the microservices architecture allowed changes to individual services based on the new needs. Functions could be replaced independently without affecting the entire project. For example, initially data gathering was done without the need of having an edge gateway service. Eventually when this requirement changed, the initial pipeline for monitoring evolved from data retrieval within device integration service to delegating that responsibility to external called OT-EDGE. This refactoring did not affect the other areas of the project and due to the separation of concerns that the microservices provided.

4.2 Final Solution

Initially the project focused mainly on the cloud application and did not prioritize the data collection of edge devices. Eventually the requirements switched focus to demonstrate the full pipeline of data collection of live information.

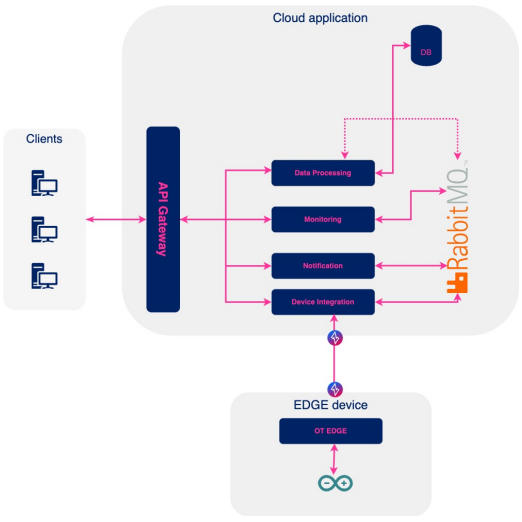

Figure 3. Final architecture

Requirements switch led to reevaluating the initial design, as seen in Figure 4. The main difference between the first and second design iteration is the addition of EDGE device and the OpenZiti tunnel to demonstrate secure data collection from a remote device.

4.3 Data Flow

Figure 4 presents a high-level overview of two principal processes executed by the application and their interaction with the microservices architecture. This figure provides a clearer understanding of how the application manages critical events, such as the triggering of an alarm by a device, where the communication flow traverses multiple microservices. Centralizing all processes within a single component would substantially increase the risk of systemic failure and data loss, as a single point of disruption could compromise the entire workflow and result in significant data loss. In contrast, distributing responsibilities across distinct microservices mitigates these risks, improving system resilience compared to a monolithic approach.

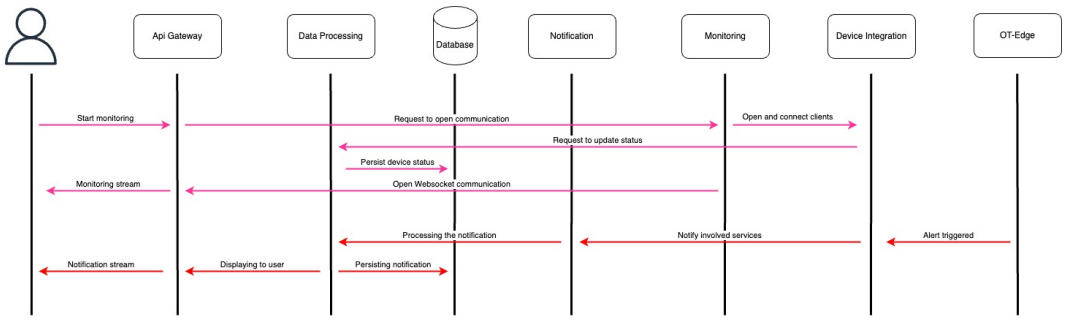

Figure 4. Data communication flow

4.4 Technical Requirements

Figure 3 depicts how the user interface, monitoring application and the edge device interact in the system. The user interface required the implementation of a simple dashboard that would be able to communicate with the monitoring application via REST API and WebSocket. The monitoring application is responsible for the processing, management and visualization of the incoming data from connected edge devices. This functionality was achieved through the implementation of dedicated microservices that oversee tasks such as monitoring, notification and data processing through the implementation of AMQP and database clients, WebSocket and TCP services.

Finally, the Edge device required the implementation of TCP and serial protocol communication services that would interact with each other to process the bidirectional communication of information between the Raspberry Pi and the Arduino Uno.

4.5 Microservice Overview

The project consists of several microservices that handle different aspects of the application. These microservices are OT-Edge, api-gateway and Dashboard UI. UI contains a simple dashboard that demonstrates the functionality of remote monitoring of equipment information in real-time by interacting with the REST API of the api-gateway microservice, data processing, device integration, monitoring, notifications and dashboard UI.

The purpose of creating API Gateway is to simplify and isolate the communication between clients and the monitoring application. By creating a single-entry point to the system. The gateway routes incoming requests to the appropriate services.

For API Gateway implementation Spring Boot provides the Spring Cloud library that contains all the required dependencies and methods for implementing a production ready service with minimal boilerplate code.

Figure 5 shows the configuration that the project used to define and route the requests to the correct services within the monitoring application.

Figure 5. API Gateway configuration

Once the monitoring services receive a request from the device or from the user interface, the data moves between the microservices via the listening AMQP clients subscribed to different topics and queues using JSON strings. JSON message is shown in Figure 6.

Figure 6. Sensor information message

The Monitoring Service is responsible for receiving sensor data that should be displayed in real-time. Service implements a WebSocket server with the help of spring-boot-WebSocket dependency to transmit the equipment information to the user interface. Additionally, it implements all necessary RabbitMQ clients that listen to incoming messages containing the monitoring data.

The Notification service contains a similar structure as the previously described monitoring service. It utilizes also the same code base and logic to gather its information. The main difference is that this service focus is to collect notifications or alerts and its lifecycle within the application. The process starts with the retrieval and ends with the persistence and acknowledgement of the notification.

5 IMPLEMENTATION

This project was primarily developed targeting edge devices capable of transmitting data via serial communication (e.g. UART via USB). This Decision was made to simplify the implementation, reduce development time and to validate core functionality of remote monitoring, data collection and processing of edge devices. Consequently, the system does not support legacy communication protocols such as RS-485, CAN bus, Modbus or I2C, which are commonly used within industrial and embedded systems. Supporting such protocols would require additional hardware interfaces and its corresponding drivers, even though implementing these additional communication protocols would not require a complete system overhaul. Thanks to the modular and flexible architecture of the project, such features could be integrated by extending the existing codebase.

Due to the development approach taken for this project, certain recurring patterns emerged across services. This section reviews common configurations, design patterns, and folder structures that offer valuable learning insights.

5.1 Project Setup and Version Control

The project followed a modular approach: the microservices were containerized using Docker and orchestrated with Docker Compose for local development. Version control was handled with Git, hosted on a private Gitea instance. Following this strategy promotes a maintainable configuration and development environment by isolating infrastructure components into dedicated directories and files.

5.2 Communication between Components

The communication between the application components is an important aspect for the microservices architecture. This section reviews some of the mechanisms and patterns that were implemented to enable interaction between services. RabbitMQ was used as the message broker for the application, all the services required to transmit or consume data from and to a client.

```java
import org.springframework.amqp.core.Binding;
import org.springframework.amqp.core.BindingBuilder;
import org.springframework.amqp.core.Queue;
import org.springframework.amqp.core.TopicExchange;
import org.springframework.amqp.rabbit.connection.ConnectionFactory;
import org.springframework.amqp.rabbit.core.RabbitTemplate;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class MessageBrokerConfig {

    // Defines a topic exchange for routing messages
    @Bean
    public TopicExchange topicExchange() {
        return new TopicExchange(name:"application.exchange");
    }

    // Defines a durable queue for device monitoring messages
    @Bean
    public Queue monitoringQueue() {
        return new Queue(name:"device.monitoring.queue", durable:true);
    }

    // Binds the queue to the exchange using a specific routing key
    @Bean
    public Binding monitoringBinding(TopicExchange topicExchange, Queue monitoringQueue) {
        return BindingBuilder
                .bind(monitoringQueue)
                .to(topicExchange)
                .with(routingKey:"device.monitoring.topic");
    }

    // Configures the RabbitTemplate used for sending messages
    @Bean
    public RabbitTemplate rabbitTemplate(ConnectionFactory connectionFactory) {
        return new RabbitTemplate(connectionFactory);
    }
}
```

Figure 7. Common RabbitMQ configuration file

Figure 7 display an example of a Java class that builds a RabbitMQ client, all microservices that required internal communication to the message broker contained a similar implementation of MessageBrokerConfig.

```java
import org.springframework.amqp.rabbit.core.RabbitTemplate;
import org.springframework.stereotype.Service;

@Service
public class SampleMessageSender {

    private final RabbitTemplate rabbitTemplate;

    public SampleMessageSender(RabbitTemplate rabbitTemplate) {
        this.rabbitTemplate = rabbitTemplate;
    }

    public void sendMessage(String message) {
        String routingKey = AmqpTopics.DEVICE_MONITORING_TOPIC.getTopicName();
        rabbitTemplate.convertAndSend(AmqpExchanges.APPLICATION_EXCHANGE.getExchangeName(), routingKey, message);
        System.out.println("Message sent: " + message);
    }
}

import org.springframework.amqp.rabbit.annotation.RabbitListener;
import org.springframework.stereotype.Service;

@Service
public class SampleMessageListener {

    @RabbitListener(queues = "#{deviceMonitoringQueue.name}")
    public void receiveMessage(String message) {
        System.out.println("Received message: " + message);
    }
}
```

Figure 8. AMQP publisher and consumer

Additionally, to receive and transmit information from the message broker the microservices required the implementation of service classes like the ones displayed in Figure 8, these services are responsible for sending and receiving messages using the RabbitMQ client resulting in a complete stack capable of internal communication between microservices.

5.3 Real-time Monitoring

The most important feature of this project relies on the ability to perform real time monitoring of connected devices. Central to this functionality is laying within the TcpClientService contained in the device integration service. Within this java class, that is presented in Figure 9, an important method was implemented to handle and process incoming messages from devices. Upon receiving a message, the handleMessage method determines its category such as monitoring data or event notifications and dispatches it accordingly. Monitoring-related messages are typically forwarded to topic dedicated to real-time telemetry, allowing other system services to react or store the data as needed.

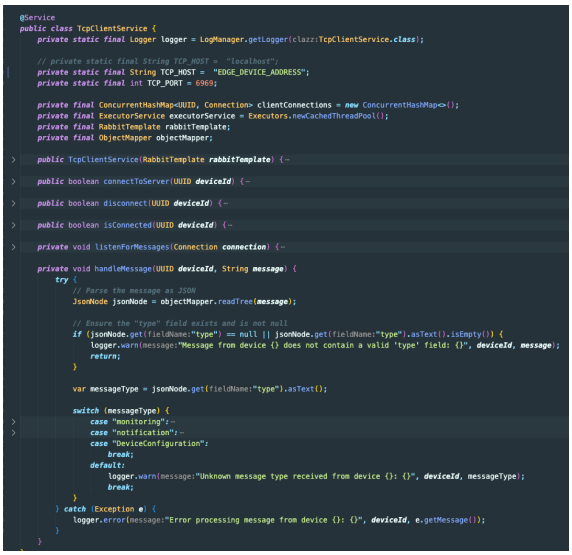

Figure 9. TcpClientService - device integration

For monitoring messages specifically, handleMessage function plays a critical role in extracting and forwarding sensor readings. These values provide up-to-date insights into device status, enabling the platform to maintain situational awareness across all connected devices.

By acting as an intelligent bridge between devices and internal services, this method ensures reliable and efficient data flow throughout the system. This architecture supports scalability, modular development, and ease of maintenance.

6 CONCLUSIONS

This project has been a deep dive into the complexities of building a real-world distributed monitoring system that bridges hardware constraints, real-time data processing, and secure communication between microservices. Dealing with the Arduino Uno memory limitations to designing and implementing a runtime configurable sensor tracking system with a lightweight JSON protocol not only kept the device stable but also allowed for a smarter, more adaptable architecture.

Ultimately, not everything worked perfectly, and rewriting parts of the system was required. But at the end a minimum viable product was achieved and subjected to further evaluation within the organization for further development. Figure 10 is a screenshot of the dashboard with an online device receiving a real-time information.

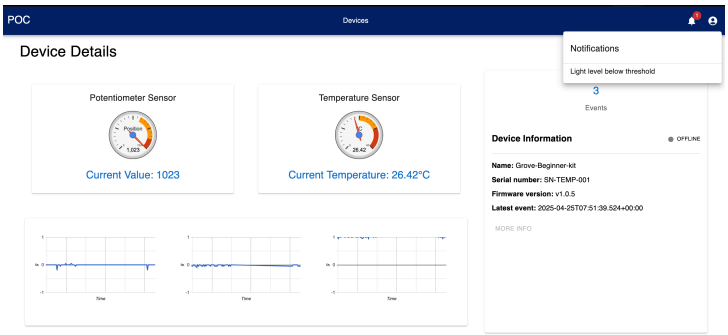

Figure 10. Application dashboard

Ultimately, during the presentation of the project, the previously identified issues resurfaced and became evident. However, these challenges were anticipated, and corresponding solutions had already been prepared as part of the system's design considerations, such as enabling a second communication channel.

REFERENCES

Bloodshed Software. (n.d.). *Dev-C++: Open Source C/C++ IDE for Windows*. Retrieved March 23, 2025. https://dev-cpp.com/.

Cummins. (2023, October 17). *What is remote diagnostics and how does it work for trucks and buses?* Cummins Inc. Retrieved February 18, 2025. https://www.cummins.com/news/2023/10/17/what-remote-diagnostics-and-how-does-it-work-trucks-and-buses-.

Fortum. (2020, April). *Remote monitoring service for turbines and generators enable data-driven plant management*. Retrieved February 18, 2025 https://www.fortum.com/media/2020/04/remote-monitoring-service-turbines-and-generators-enable-data-driven-plant-management.Meta. (n.d.). *Acknowledgements – React community*. Retrieved March 23, 2025.https://react.dev/community/acknowledgements.

OpenZiti. (n.d.). *OpenZiti project*. Retrieved February 18, 2025. https://openziti.io/.

Oracle. (n.d.). *What is Java technology and why do I need it?* https://www.java.com/en/download/help/whatis_java.htmlJava

RabbitMQ. (n.d.). *AMQP 0-9-1 model explained*. Retrieved March 23, 2025. https://www.rabbitmq.com/tutorials/amqp-concepts.

RabbitMQ. (n.d.). Queues. RabbitMQ. https://www.rabbitmq.com/docs/queues

Ritchie, H., Mathieu, E., Roser, M., & Ortiz-Ospina, E. (2023). *Internet*. Our World in Data. Retrieved January 18, 2025. https://ourworldindata.org/internet.

Sinha, S. (2024, September 3). *State of IoT 2024: Number of connected IoT devices growing 13% to 18.8 billion globally*. IoT Analytics. https://iot-analytics.com/number-connected-iot-devices/

The Apache Software Foundation. (n.d.). *Welcome to Apache Maven*. https://maven.apache.org/

VMware. (n.d.). *Why Spring*. Spring.io. https://spring.io/why-spring