POLYTECHNIC UNIVERSITY OF TURIN

MASTER's Degree in COMPUTER ENGINEERING (SOFTWARE)

MASTER's Degree Thesis

An Integrated Web Platform for Remote Control and Monitoring of Diverse Embedded Devices: A Comprehensive Approach to Secure Communication and Efficient Data Management

Supervisors

Prof. MAURIZIO REBAUDENGO

Dr. EDOARDO GIUSTO

Ing. PAOLO DOZ

Candidate

ROBERTO CASCHETTO

APRIL 2024

Summary

The project is oriented towards the development of a web platform designed for the remote control and monitoring of dispersed embedded devices across diverse geographical areas. The platform will enable the issuance of commands and retrieval of data from the device, collected by sensors connected to it.

Given the diverse nature of the devices, operating on different systems with also different Operating Systems, the client component is envisioned to be cross-platform, implemented using the C programming language. This client will play a crucial role in authenticating the devices trying to communicate with the server, in transmitting sensor data, requesting and receiving update packages, and executing commands received from the server.

On the server side, the backend will be developed using Node.js, with a bifurcated structure. The first segment will manage the REST API, handling calls for the web frontend and collecting and managing information from the individual embedded devices. The second segment will focus on maintaining open connections with the various active embedded devices, enabling the exchange of specific information and commands. The frontend, responsible for user interaction, will be developed using React.js.

Data storage will involve the strategic integration of both SQL and noSQL databases to efficiently manage the diverse datasets and information collected. The SQL database will be used to store basic structured information. Meanwhile, the noSQL database will be employed to store complex and unstructured information such as the data collected by individual devices. Additionally, an in-memory RAM database will be utilized to cache the latest update packages. This holistic approach ensures the creation of a sophisticated web platform tailored for comprehensive remote device management and monitoring, with a specific focus on data security, reliability, and efficient communication.

Acknowledgements

ACKNOWLEDGMENTS

I would like to express my sincere gratitude to all those who have contributed to the realization of this project. Firstly, we extend our appreciation to the team at Abinsula S.R.L. for their invaluable support, guidance, and expertise throughout the development process. Their dedication and collaboration were instrumental in bringing this project to fruition. I would also like to thank my academic advisors for their insightful feedback, encouragement, and mentorship. Their guidance has been invaluable in shaping the direction of this development and ensuring its successful completion.

Furthermore, I would like to express my heartfelt thanks to my family for their unwavering belief in me and their constant support. Their encouragement has been a source of strength and motivation, enabling me to reach this significant milestone.

I am deeply grateful to all those who have played a part in this endeavor, and I sincerely appreciate their contributions to its accomplishment.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

In the dynamic landscape of contemporary technology, the need for efficient management and monitoring of dispersed and diverse embedded devices has become essential. This thesis embarks on an exploration of an architecture centered on the development of a web interface realized to oversee the challenges of remote control and monitoring of these devices across diverse geographical areas.

The core essence of the initiative lies in empowering the platform to issue commands and retrieve data from a myriad of devices, each equipped with sensors collecting different valuable information. This intrinsic capability is designed to transcend the limitations of geographical, connectivity, maintainability, versioning, and control constraints, offering a comprehensive solution for effective device management to handle and solve all these difficulties.

Given the diverse array of devices, each working on different boards and various operating systems, the client component of this architecture stands as a testament to versatility. Implemented in the universally compatible C programming language, the cross-platform client assumes an important role. It not only authenticates devices seeking communication with the backend but also facilitates the seamless transmission of sensor data, the request and receipt of update packages, and the execution of commands received from the server, issued by the users in the web dashboard.

The server-side architecture adopts a bifurcated structure, with the back-end entirely developed using Node.js. The first segment takes charge of managing the REST API requests, orchestrating calls for the web front end, and adeptly collecting, processing, and handling information from individual devices. In tandem, the second segment focuses on the dynamic task of maintaining open connections with active embedded system and transmitting custom commands. This facilitates the communication, fostering a responsive and interconnected architecture.

The front end, the user's interface with the intricate infrastructure, is instead developed with React.js, ensuring a seamless and intuitive user experience.

Integral to the success of this project is a robust data storage infrastructure. A strategic integration of both SQL and noSQL databases ensures the management of diverse information of any possible structure granting optimal performances. The SQL database is dedicated to storing the basic structured data, such as device characteristics, users, etc. while the noSQL counterpart handles the more complex and unstructured information like the sensor data collected by the individual devices. Complementing this, an in-memory RAM database is employed to cache the latest update packages that are uploaded by the users from the front end, keeping the device's connection statuses, and other simple but frequently accessed data. In this way it is optimizing response times and alleviating the load on the different databases during mass updates or information storage/retrieval.

This project adopts a holistic approach, promising the creation of a sophisticated web platform tailored for comprehensive remote device management and monitoring. It stands as a testament to the commitment to data security, reliability, and efficient communication in an ever-evolving technological landscape.

1.1 Background

The genesis of this project is rooted in the strategic vision of the Abinsula S.R.L. company that has initiated its development. Originating from the essential need to establish a unified platform for streamlined collaboration with partners, the primary objective is to furnish a singular web interface accessible to the company and its associates. This collaborative ecosystem is designed to provide visibility into the diverse array of devices connected to the network, ensuring a cohesive and interconnected operational environment.

The conceptualization of this project was not without its complexities, as it grappled with multifaceted challenges inherent in the identification of devices across the network, the formulation of robust communication protocols, and the imperative aspect of securing communications between the client and server. To systematically address these challenges and potential roadblocks and bottlenecks, a rigorous requirement engineering phase was deemed indispensable. This additional step served as a comprehensive forum for the exhaustive deliberation and resolution of issues related to device discovery, communication protocols, data storage, and communication security, among other pertinent considerations.

With the architecture carefully defined and a comprehensive roadmap comprising technologies, procedural steps, and operational protocols firmly established, the project seamlessly transitioned from conceptualization to actualization. This step marked the commencement of active development, guided by a well-defined strategy crafted during the requirement engineering phase.

The background of this project underscores not only its origin within the corporate strategy but also the meticulous planning and foresight applied to overcome the intricacies and challenges that define its developmental landscape.

1.2 Objectives

The overarching objectives of this project are meticulously designed to encompass the intricacies of creating a comprehensive system for the remote control and monitoring of dispersed embedded devices across diverse geographical areas. The multifaceted objectives can be distilled into several key components, each contributing indispensably to the holistic success of the entire architecture.

1. Server Infrastructure Creation: An important aspect of the project involves the establishment of robust backend infrastructures. This encompasses the realization of two dedicated servers, each wielding distinct functions. The first part is specifically engineered to adeptly manage TCP connections with a myriad of devices dispersed across the network. Concurrently, the second one is tasked with intricately handling REST API requests. This dual-server architecture is specifically structured to accommodate API calls from both client instances – the ones resident on the devices themselves - and those constituting the frontend clientele. The servers, as bastions of computational power, bear the onus of carefully validating incoming requests and, where requisite, persisting pertinent information into designated databases, contingent on the nature of the processed data.

2. User-Friendly Frontend Interface: In tandem with the server infrastructure, the project envisages the creation of an intuitively designed web page. This interface is crafted to offer users a seamless and comprehensible view of diverse information sourced from the networked systems. Beyond data visualization, the frontend empowers users to issue specific commands to devices and facilitate the uploading of update packages earmarked for the enhancement of particular devices.

3. C Client Development: Complementing the server and frontend components, the project extends its purview to include the development of a C client tailored explicitly for the devices. This client assumes the critical role of establishing a secure connection with the servers, transmitting different kind of data, and also receiving commands, updates and ancillary information from the server.

4. Database Architecture Design: A concurrent objective involves the creation of a robust architecture for the three distinct databases that will be integral to the system. These databases, meticulously defined, are geared too cooperate

for handling and storing the diverse information that will be transmitted to the different servers.

The project's multifaceted objectives converge toward the creation of a sophisticated system tailored for the comprehensive remote management and monitoring of devices. This approach ensures a holistic solution, addressing challenges related to communication protocols, device discovery, security, and effective data handling granting at the same time optimal performances under different load levels.

In cognizance of the project's complexity and to ensure a robust foundation for future endeavors, a lot of emphasis was also placed on the meticulous creation of documentation. This documentation initiative encompassed every facet of the project, intending to furnish a detailed account of each component's functionalities and requisite information. Such a strategic documentation approach serves as a cornerstone for heightened maintainability, systematic analysis of requirements, and continuous improvement of the project's intricacies.

Component-Centric Documentation. Every integral component constituting the project underwent a meticulous documentation process. This entailed the creation of dedicated documents elucidating:

- Functionalities: A comprehensive breakdown of each component's functionalities was documented. This includes detailed information of the operations it performs, the services it provides, and its role in the overall system architecture.
- Information Requirements: The documentation outlines the specific information that each component requires for optimal performance. Understanding these dependencies is vital for seamless integration and interoperability within the system.
- Interfaces and Interactions: The documentation delineates how each component interfaces with others and the nature of their interactions. This is critical for comprehending the flow of information and ensuring coherent collaboration among different project elements.

Maintenance Facilitation. The primary goal of this documentation initiative is to streamline project maintenance. A well-documented system:

- Facilitates Debugging: In the event of issues or bugs, detailed documentation serves as a valuable resource for swift and accurate debugging. Developers can refer to the documentation to understand the intricacies of each component, aiding in effective issue resolution.
- Simplifies Updates and Enhancements: When it comes to refining or expanding functionalities, a clear understanding of each component expedites the process. Developers can confidently make updates or enhancements, armed with a thorough comprehension of existing structures and operations.

Requirement Analysis and Improvement. Documentation serves as a foundational artifact for ongoing improvement. It enables:

- Requirement Analysis: By scrutinizing the documented functionalities and information requirements, stakeholders can conduct in-depth requirement analyses. This, in turn, provides insights into areas for optimization or expansion.
- Continuous Improvement: Armed with a detailed understanding of each component, teams can iteratively refine and improve functionalities. This aligns with the project's evolution and ensures it stays adaptive to changing needs.

In essence, the commitment to comprehensive documentation emerges as a pivotal pillar of project success. Beyond the immediate advantages of maintenance facilitation, it acts as a reservoir of insights for continuous improvement and a testament to the project's commitment to systematic development practices.

Chapter 2

Requirements Gathering

The development trajectory of this project has diligently adhered to the tenets of the Software Engineering process, navigating through each essential phase. Fundamental to this process is the imperative step of Requirement Engineering, a critical phase wherein the intricate functionalities and operations of the platform are exhaustively studied and engineered. This phase is strategically designed to transcend potential limitations, creating a robust development plan.

During the Requirement Engineering phase, a comprehensive examination is undertaken, and a list of crucial requirements is curated. Each requirement is documented, and priorities are judiciously assigned to ensure a methodical and strategic approach to development. This phase assumes paramount importance as it forms the bedrock upon which the entire project is constructed. Any lapses or oversights during this stage have cascading effects throughout the project, potentially leading to complexities that are challenging to rectify in subsequent phases.

The Development phase is executed next in strict adherence to the prioritized list of requirements, commencing with those deemed of the highest priority and cascading down to those of lower precedence. This phase, albeit seemingly straightforward, assumes a critical role in the project's integrity. Any error or oversight at this juncture reverberates throughout the project, necessitating particular attention to detail and precision in implementation.

A SCRUM-like iterative approach has been incorporated into the development process, entailing periodic deliveries in conjunction with a small presentation every few weeks. This cyclic delivery framework facilitates ongoing assessments of the project's alignment with client requirements. Additionally, it offers a pragmatic mechanism for the reevaluation of inserted and omitted requirements. This iterative refinement process ensures the project's adaptability to evolving client needs and provides a mechanism for potential adjustments.

During the requirement engineering phase, a special focus was directed towards the formulation of an efficient and resilient database structure. Recognizing the enduring nature of the chosen schema, which was intended to persist throughout the project's lifecycle, the in-depth analysis of database design became a linchpin in the entire process. The rationale behind this scrutiny was to establish a schema that would endure deployment with minimal alterations, mitigating the risk of data loss or system disruptions although granting also optimal performances.

1. Enduring Database Schema Design: The selected schema for both the SQL and NoSQL databases was envisaged to be a stalwart foundation for the project. The longevity of this schema, intended to withstand the entire project duration, underscored the need for meticulous planning and strategic decisions during the requirement engineering process.

2. Data Allocation Strategy: A critical facet of the requirement engineering process involved judiciously segregating the data into the SQL and NoSQL databases. This allocation strategy was not only influenced by the nature of the data but also by the performance requirements and scalability considerations of the project. Key considerations included:

    - SQL Database: Data that adhered to a structured format and necessitated complex querying found its home in the SQL database. This included information critical for relational operations and intricate transactional processes.
    - NoSQL Database: Unstructured or semi-structured data, requiring flexible storage and rapid retrieval, was designated to the NoSQL database. This encompassed datasets associated with device information, sensor data, and other non-relational entities.

3. Long-Term Stability and Deployment Resilience: The decision-making process in database design was profoundly influenced by the project's commitment to long-term stability and deployment resilience. The objective was to minimize schema modifications post-deployment, ensuring a stable environment for the entire project lifecycle.

4. Risk Mitigation through Minimal Schema Modifications: Recognizing the inherent risks associated with altering the database schema during deployment, a deliberate effort was made to keep modifications at a minimum. This strategic approach sought to preclude the potential loss of data or disruptions that could arise from extensive schema changes post-deployment.

5. Continuous Evaluation and Adaptation: While the initial database schema provided a robust foundation, it was not static. The requirement engineering process was designed to accommodate continuous evaluation and adaptation.

This flexibility allows for refinement based on evolving project needs, with an emphasis on minimal intrusion into the established database structure.

The processes of Requirement Engineering and Development coalesce to form a robust and agile methodology. This approach not only safeguards against potential pitfalls but also integrates flexibility to accommodate evolving project dynamics. The planning and strategic decisions invested in the phase regarding database design are emblematic of the project's commitment to longevity, stability and performances. By thoughtfully allocating data and minimizing schema modifications, the project lays a resilient foundation, poised to endure the dynamic challenges of deployment and beyond.

2.1 Requirement Engineering: A Comprehensive Insight

1. Initiation and Exploration:
   - The Requirement Engineering process commences with a comprehensive initiation, where project stakeholders collaboratively explore the fundamental objectives and aspirations of the platform.
   - A detailed analysis of both functional and non-functional requirements is conducted, aiming to capture the breadth and depth of the envisioned system.
2. Stakeholder Collaboration:
   - Stakeholder engagement is important, involving constant collaboration to gain insights into the nuanced requirements and expectations of end-users, administrators, and other involved parties.
   - Iterative workshops and meetings are conducted to foster a dynamic exchange of ideas and to ensure a holistic understanding of diverse perspectives.
3. Requirement Elicitation Techniques:
   - A myriad of elicitation techniques, including interviews, surveys, document analysis, and brainstorming sessions, are judiciously employed to extract implicit and explicit requirements.
   - Prototyping and scenario-based discussions may be utilized to envision user interactions and system responses.
4. Requirement Documentation:

- The identified requirements are meticulously documented, employing standardized templates to ensure clarity, coherence, and traceability.
- Clear categorization of requirements into functional, non-functional, and domain-specific aspects is a hallmark of a well-structured requirement document.

5. Prioritization and Validation:
    - Prioritization mechanisms are employed to assign significance to each requirement, facilitating a phased development approach that addresses high-priority features first.
    - Continuous validation with stakeholders ensures that the documented requirements align with their expectations and the broader project goals.
6. Requirement Traceability:
    - Robust traceability matrices are established to map requirements through various stages of development, enabling systematic monitoring and verification of their implementation.

In essence, the Requirement Engineering process unfolds as an intricate dance between stakeholders and development teams, fostering a shared understanding of the project's objectives. This collaborative, iterative, and systematic approach lays the groundwork for a development phase that is not just efficient but adaptive to evolving project dynamics.

The subsequent Development phase, executed with the refined and prioritized requirements in mind, builds upon this robust foundation, propelling the project towards its envisioned fruition.

2.2 Initial Phase Gathering

During the initial requirements gathering phase of the project, a multitude of considerations were addressed. The project's scope can be broadly categorized into four primary areas: frontend, backend (comprising TCP and REST API components), devices, and databases (SQL, NoSQL, and In-RAM databases).

Frontend considerations involved the definition of main screens and functionalities aimed at providing end-users with a meaningful and seamless experience. Noteworthy pages include a Dashboard consolidating device information, a page for viewing and creating devices, models, and products, a space for managing device groups, and a dedicated area for creating, viewing, and modifying device updates.

Backend functionalities were carefully analyzed and specified, covering essential aspects such as API calls for frontend management, data delivery to the frontend, API calls for data storage in databases, and coordination of communication with diverse devices. Specifics for TCP servers included executed commands, data transmission details, authentication protocols, and mechanisms for customizing device parameters. Overall, also some performance requirements were specified in order to guarantee a correct and fluid experience through the entire platform under the possible stress loads it will have to face.

Device requirements were outlined comprehensively, encompassing communication protocols, stringent security measures to prevent data compromise, precautions against impersonation risks, and custom command's execution and proper authentication. Detailed specifications were provided for sending and receiving data, authentication methodologies and procedures for devices, and the specific data to be transmitted from sensors and other pertinent features.

The database architecture was strategically planned, involving the definition of three distinct databases - SQL (postgres), NoSQL (MongoDB), and In-RAM (Redis) - each allocated for specific activities. The specifics of what data to store, how to store it, and all relevant activities integral to effective database utilization were specifically outlined.

This holistic approach during the requirements-gathering phase ensures a comprehensive blueprint for subsequent project development stages. By strategically dissecting each facet, from user interface considerations to intricate backend and database specifications, the project is poised for a streamlined and efficient execution aligned with organizational objectives and end-user expectations.

The Requirements Engineering process was systematically conducted at various stages, aligning with the commencement of new development phases. Prior to initiating each phase, an assessment of existing requirements took place, ensuring a comprehensive understanding of the accomplished work. Subsequently, new requirements were formulated in consideration of the forthcoming objectives. This iterative approach allowed for concise refinement and adaptation of project requirements.

The initial phase of requirements discussion was centered around the frontend development. Following the completion of frontend development, an evaluation of these requirements occurred, accompanied by potential additions. Simultaneously, discussions unfolded concerning the requirements for subsequent phases, such as server development, database design, etc. A structured documentation approach was adopted, utilizing a document to capture requirements. Due to the project's manageable scale, an organizational hierarchy was implemented, encompassing macro categories, subcategories, and micro categories.

Macro categories served as overarching themes, for instance, "Frontend". Subsequently, subcategories, like "Web Pages" and "API Management", etc. facilitated a more granular breakdown. Further classification into micro categories ensued, addressing specific elements like the "Login Page Interface" and "Dashboard Interface". This classification framework obviated the need for numerical enumeration, promoting a clear and intuitive categorization.

Priority discussions ensued after requirement formulation, elucidating the significance of each requirement within the broader context. Concurrently, preliminary architectural sketches were crafted, offering a visual representation of the proposed system's structure and interactions between the different components. Also some sketch of the anticipated frontend aesthetics has been generated for a complete overview over the entire platform. Once the requirements engineering phase concluded, development activities could commence, guided by the established and prioritized set of requirements.

2.3 Protocol Definition

In delineating the communication protocols for the project, a crucial consideration was the diverse array of devices anticipated to utilize the C client. These devices inherently possessed distinctive characteristics, including varying resources, operating systems, and hardware capabilities. Consequently, the chosen communication protocol needed to strike a delicate balance between ease of use and minimal imposition on device resources during encryption and decryption processes.

Given the need for a secure yet resource-efficient solution, the optimal approach involved steering clear of on-device encryption and decryption overhead. To enhance security and minimize resource utilization, a pragmatic choice was the adoption of an SSL protocol in conjunction with a TCP connection. This method preemptively established a secure and encrypted channel, obviating the necessity for on-device encryption and decryption, thereby mitigating resource strain. This configuration ensured a seamless and secure data exchange between the client and server. To enhance security measures and preempt potential external threats, a custom certificate was integrated into the system. This certificate, generated beforehand, was pre-loaded onto the device along with the C client before delivery. Additionally, a custom certification authority is installed on the devices and undertakes the authentication of the certificate's validity before its usage. This proactive step not only ensured the certificate's trustworthiness but also guarded against potential issues such as expiration. In scenarios where the device required a new certificate and requested one from the server, the certification authority carefully verified its validity. This comprehensive validation process aimed to thwart any attempts at intercepting communication and modifying the certificate to gain unauthorized control. Such robust security measures were implemented to fortify the integrity of the communication channel and safeguard against external tampering.

Simultaneously, the formatting of messages for transmission over the network emerged as another critical concern. Deliberating on resource efficiency, a thorough exploration of various alternatives led to the endorsement of MessagePack, a lightweight library facilitating the transmission of JSON-formatted files. Notably, this approach yielded a remarkable reduction, approximately 40%, in the overall data size. This optimization proved especially beneficial for devices with limited resources, as it curtailed the volume of packets to be transmitted. Consequently, the chosen protocol and formatting strategy not only upheld security standards but also demonstrated a judicious consideration of resource constraints, ensuring optimal performance across a spectrum of devices. Furthermore, the selected library not only prioritized efficiency during data encoding and transmission but also facilitated a streamlined decode process. This attribute allowed for the easy re-composition of the JSON file during decoding, enabling the server to effortlessly save data in the database with minimal processing. This capability significantly enhanced overall performance, particularly in scenarios involving multiple devices concurrently communicating with the server, each sending diverse data for storage. The result was a reduction in server overhead, as the unpacked data could be swiftly saved directly to the databases without the need for additional processing. This dual focus on security and performance underscored the project's commitment to optimizing communication processes across varied devices.

Chapter 3

Communication Protocols

In the context of the project, the selection of a communication protocol held important significance, especially considering the diverse nature of the targeted embedded devices. Ranging from Raspberry Pis to other embedded systems characterized by limited resources and simplistic operating systems, the paramount need was to identify a protocol that struck a harmonious balance between simplicity, speed, and lightweight operation.

The evaluation process involved a meticulous analysis of various libraries, with a primary focus on achieving an equilibrium between performance, reliability, and security. A critical consideration in this assessment was the imperative of ensuring secure communication without unduly burdening devices with excessive overhead. The overarching objective was to identify a protocol adept at transmitting data reliably while maintaining optimal performances and resource utilization, particularly for devices with constrained resources.

The best solution emerged through the amalgamation of two distinct solutions: SSL for security, creating an encrypted channel for secure data transmission, and MessagePack for data encoding. This dual-protocol approach not only ensured the establishment of a secure channel but also facilitated the encoding of data in a manner that minimized packet sizes, thereby enhancing the overall performances of data transmission. This strategic combination addressed the project's fundamental requirements for secure and resource-efficient communication across a spectrum of different embedded devices.

3.1 Reliability and Security

Transport Layer Security (TLS) stands as a cryptographic protocol designed to secure a communication channel over computer networks. It primarily ensures privacy and data integrity between two applications, preventing potential eavesdropping and tampering.

Key Features of TLS:

1. Encryption: TLS employs advanced encryption algorithms to protect data during transit. This ensures that even if intercepted, the data remains unreadable without the appropriate decryption key.
2. Authentication: TLS facilitates mutual authentication between the client and server, assuring both parties of each other's legitimacy. This process involves the exchange of digital certificates to verify identity.
3. Data Integrity: TLS guarantees the integrity of transmitted data. Through cryptographic hash functions, it detects any unauthorized modifications, ensuring that the received data is identical to what was sent.
4. Forward Secrecy: TLS supports forward secrecy, a feature that generates unique session keys for each connection. Even if one session key is compromised, past and future communications remain secure.
5. Compatibility: TLS is widely supported and integrated into various web browsers, servers, and applications, making it a versatile and interoperable choice for secure communication.

In the realm of secure communication between an embedded C client and a Node.js server, the choice of the underlying protocol plays a pivotal role. The selected protocol must not only ensure the confidentiality and integrity of data but also guard against various malicious activities such as impersonation, data modification, or deletion. TLS (Transport Layer Security) stands out as a robust choice, providing encryption to ensure data privacy, strong mutual authentication between the client and server, broad support, and compatibility. Its established protocols and community scrutiny contribute to its reliability. While TLS may introduce a slightly higher computational overhead, the trade-off in terms of enhanced security makes it a favorable option. Other alternatives, such as SSH (Secure Shell) and DTLS (Datagram Transport Layer Security), offer their own set of advantages but may come with considerations like compatibility issues and limited support in specific environments. In the rapidly evolving landscape, newer protocols like QUIC (Quick UDP Internet Connections) bring features such as low-latency connections but may face challenges related to compatibility. Overall, the careful consideration of the strengths and limitations of each protocol is crucial in ensuring a secure and efficient framework.

- **TLS (Transport Layer Security)**
  - **Advantages:**

    * Robust encryption ensuring data privacy.
    * Strong mutual authentication between client and server.
    * Broad support and compatibility.
    * Established protocols and community scrutiny.
  - **Considerations:**
    * Slightly higher computational overhead.

- **SSH (Secure Shell)**
  - **Advantages:**
    * Secure remote access and encrypted communication.
    * Strong encryption and user authentication.
  - **Considerations:**
    * Compatibility issues with specific applications.
    * Primarily designed for terminal-based interactions.

- **DTLS (Datagram Transport Layer Security)**
  - **Advantages:**
    * TLS adapted for datagram-based communications.
    * Ideal for real-time applications.
  - **Considerations:**
    * Not as widely supported as traditional TLS.
    * Potential challenges in non-TCP environments.

- **QUIC (Quick UDP Internet Connections)**
  - **Advantages:**
    * Designed for low-latency connections.
    * Built-in multiplexing and encryption.
  - **Considerations:**
    * Relatively newer protocol.
    * May face compatibility challenges.

Selection considerations taken into account:

1. Security: Prioritize protocols offering robust encryption and authentication.
2. Compatibility: Assess the compatibility with the specific requirements of the C client and Node.js server.

3. Community Support: Consider protocols with a strong community for ongoing scrutiny and updates.

4. Resource Constraints: Evaluate the computational overhead and resource requirements, particularly for embedded systems.

In conclusion, while TLS remains a strong contender for its established security features and widespread support, the choice should align with the project's unique demands, taking into account compatibility, resource constraints, and specific communication requirements between the C client and Node.js server.

While it's possible to implement encryption methods at the client level, particularly in the C client for embedded systems, this practice is discouraged for several reasons:

1. Security Risks: Implementing encryption at the client level introduces the risk of weaker, less-vetted encryption methods. TLS benefits from extensive community scrutiny and standardized protocols, ensuring a higher level of security.

2. Limited Customization: Client-level encryption may lack the customization options provided by widely used protocols like TLS. This could lead to suboptimal security configurations tailored to the specific needs of the embedded system.

3. Maintenance Challenges: Client-level encryption often requires manual updates and maintenance. TLS, being a standardized protocol, is easier to maintain and update across various devices.

4. Interoperability Concerns: Implementing custom encryption methods may result in compatibility issues with other systems and services, hindering interoperability.

5. Community Auditability: TLS is widely used and subjected to continuous community auditability, ensuring vulnerabilities are promptly identified and addressed. Custom encryption methods lack this level of scrutiny.

In conclusion, despite the computational overhead associated with TLS, its widespread adoption, community auditability, and adaptability make it a superior choice over client-level encryption. The standardized nature of TLS, along with its security features and community support, ensures a more robust and secure protocol for embedded systems, even those with limited resources.

3.2 Command Execution and Data Retrieval

In the context of communication between a server and an embedded system, the choice of a suitable data serialization format is crucial, particularly when dealing with resource-constrained environments. MessagePack stands as a compelling solution, providing a lightweight binary format for serializing data. It is designed to be both simple and fast, resulting in reduced payload sizes and improved transmission speeds. The binary format of MessagePack allows for more efficient encoding and decoding, making it especially advantageous in scenarios where resources are limited. Compared to not using any specific serialization format, MessagePack introduces structure to the data, aiding in its interpretation on both ends of the communication.

Below is a table comparing MessagePack with other alternatives:

- **libcurl**
  - **Pros:**
    - Wide protocol support.
    - Mature and widely used.
    - Cross-platform.
  - **Cons:**
    - Can be perceived as heavyweight.
- **cJSON**
  - **Pros:**
    - Lightweight JSON parsing.
    - Simple API.
  - **Cons:**
    - Limited to JSON handling.
- **Mongoose**
  - **Pros:**
    - Networking library.
    - Lightweight and easy to use.
  - **Cons:**
    - May be less feature-rich.
- **Jansson**

  - **Pros:**
    - C library for JSON.
    - Well-documented.
  - **Cons:**
    - Focuses solely on JSON.

- **libmicrohttpd**
  - **Pros:**
    - Embeddable HTTP server.
    - Lightweight.
  - **Cons:**
    - May not be feature-rich.
- **ZeroMQ**
  - **Pros:**
    - High-performance messaging.
    - Supports various patterns.
  - **Cons:**
    - Requires understanding of patterns.
- **nghttp2**
  - **Pros:**
    - Implements HTTP/2 and HTTP/1.1.
    - Efficient communication.
  - **Cons:**
    - Focused on HTTP protocols.
- **JSON**
  - **Pros:**
    - Human-Readable.
  - **Cons:**
    - Larger Payload.
    - Additional data overhead.
- **XML**

  - **Pros:**
    * Standardized.
    * Human-Readable.
  - **Cons:**
    * Verbosity.
    * Resource-intensive.

- **MessagePack**
  - **Pros:**
    * Efficient binary serialization.
    * Compact data representation.
    * Supports multiple languages.
  - **Cons:**
    * May require additional setup.

In addition to its advantages in compactness and speed, MessagePack introduces ease of use in the decoding process, particularly beneficial for server-side operations. Its binary format facilitates straightforward decoding into a JSON-like structure, streamlining the process of interpreting and storing data in a NoSQL database. This inherent compatibility simplifies the integration of MessagePack into existing systems, as the server can effortlessly unpack the data and directly use the resulting JSON-like object for storage or further processing. This seamless decoding capability enhances the overall performances of data processing workflows, making MessagePack an even more attractive choice for scenarios where simplicity and resource optimization are paramount concerns.

Chapter 4

Architecture Development

In the ever-evolving landscape of software development, crafting a robust and scalable infrastructure is pivotal for the success of an application. This document delineates the architectural blueprint of a sophisticated software application, leveraging React.js for the frontend and Node.js for the backend, specifically divided into two distinct parts – a RESTful API employing Express.js and Passport for authentication, and a TCP connection handler for real-time communication.

The frontend of the application is developed using React.js, a widely adopted JavaScript library renowned for building responsive and dynamic user interfaces. React's component-based architecture facilitates the creation of reusable UI elements, promoting modularity and maintainability. The virtual DOM mechanism ensures optimal performance by minimizing unnecessary re-rendering. Key Features of React.js Frontend:

1. Component-Based Structure: Dividing the UI into self-contained components allows for better organization, reusability, and easier debugging.

2. State Management: Utilizing React's state management ensures efficient handling of the application's state, leading to seamless data flow and synchronization.

3. Virtual DOM Optimization: React's virtual DOM minimizes DOM manipulation, enhancing performance and delivering a smoother user experience.

4. Reusable Components: Component reusability reduces redundancy in code, promoting a cleaner and more maintainable codebase.

5. Declarative Syntax: React's declarative syntax simplifies the process of understanding and maintaining the code, contributing to improved developer productivity.

6. Community Support: The extensive React.js community provides a plethora of libraries, tools, and resources, facilitating rapid development and issue resolution.

The backend architecture is powered by Node.js, offering a scalable and efficient runtime environment. The backend is subdivided into two components, catering to RESTful API requirements and real-time TCP connections.

RESTful API with Express.js and Passport features:

1. Express.js Framework: Express.js is employed to establish a robust RESTful API, simplifying route handling, middleware integration, and request/response processing.

2. Passport for Authentication: Passport.js, a versatile authentication middleware, ensures secure and customizable user authentication strategies, accommodating various authentication methods.

3. Modular Routing: The API is structured with modular routing, enhancing code readability and facilitating the addition of new features without disrupting existing functionality.

TCP connection handling features:

1. Scalable TCP Server: The TCP connection handler is designed to manage real-time communication efficiently, providing a scalable solution for handling concurrent connections.

2. Event-Driven Architecture: Leveraging Node.js's event-driven architecture ensures a non-blocking, highly responsive system for handling asynchronous TCP communication.

3. WebSocket Integration: For bidirectional communication, WebSocket integration is implemented to facilitate real-time updates and notifications.

Architecture's advantages:

1. Full Stack JavaScript: Utilizing JavaScript across the entire stack ensures consistency, streamlining development, and promoting code sharing.

2. Real-Time Capabilities: The integration of a TCP connection handler enables real-time features, fostering responsive and interactive user experiences.

3. Modularity and Reusability: The use of React.js and Node.js encourages modular development, enhancing code maintainability, and supporting future scalability.

4. Community and Ecosystem: Both React.js and Node.js boast vibrant communities, providing access to an extensive ecosystem of libraries, tools, and best practices.

5. Performance Optimization: React.js's virtual DOM and Node.js's non-blocking I/O contribute to a performant application, ensuring responsiveness under varying workloads.

Overall, the proposed architecture leverages the strengths of React.js and Node.js to deliver a powerful, scalable, and maintainable software application. By carefully balancing frontend and backend technologies, the architecture is poised to meet the challenges of modern web development and provide a foundation for future enhancements.

The integration of frontend and backend components is a critical aspect of developing a cohesive and efficient web application. In this section, we delve into the formal process of linking the frontend, developed using React.js, with the backend, implemented in Node.js, Express.js, and Passport.

1. RESTful API Endpoint Consumption: The frontend, powered by React.js, communicates with the backend through HTTP requests to the RESTful API endpoints. These endpoints act as gateways, allowing the frontend to interact with the server's resources and functionalities.

2. Axios for HTTP Requests: To facilitate seamless communication, Axios, a promise-based HTTP client, is often employed. It simplifies the process of making asynchronous requests to the backend, handling responses and errors elegantly.

3. State Management: React's state management, including the use of hooks like useState and useEffect, allows the frontend to maintain dynamic content and respond to changes in data received from the backend. Stateful components are updated based on the information fetched from the server.

4. Authentication Flow: For secure interactions, the frontend follows a well-defined authentication flow. User authentication, managed by Passport.js on the backend, involves the issuance and verification of tokens, typically JWTs (JSON Web Tokens). These tokens are sent with each request to authenticated endpoints, ensuring secure communication.

5. Handling Backend Responses: Responses from the backend are processed within the frontend components. This includes handling successful data retrievals and managing errors gracefully. Feedback to the user is provided, ensuring a seamless and transparent user experience.

The integration of frontend and backend components forms a unified ecosystem where user interactions seamlessly translate into server actions, and real-time updates enhance the overall user experience. The linkage of React.js with Node.js, Express.js, and Passport ensures a responsive, secure, and feature-rich web application that meets the demands of modern user expectations.

The deployment strategy for the project encompasses the utilization of Docker containers to run distinct instances of the backend, frontend and the three databases. This section delineates the rationale behind containerization, its advantages, and how Docker facilitates the swift and efficient orchestration of services.

1. Containerization Strategy.

   - Purposeful Separation: Each core component of the system, including the backend, frontend, and databases, operates within dedicated Docker containers. This purposeful separation enhances maintainability, scalability, and facilitates independent updates and scaling for each service.
   - Scalability and Resource Allocation: Containerization enables a modular approach to scalability. Services can be independently scaled based on demand, and resources can be efficiently allocated to each container, preventing resource contention and optimizing overall system performance.

2. Docker: A Brief Overview.

   - Containerization with Docker: Docker provides a robust containerization platform that encapsulates applications and their dependencies into isolated units called containers. These containers ensure consistency across different environments, mitigating the infamous "it works on my machine" problem.
   - Advantages of Docker:
     - Portability: Docker containers can run consistently across various environments, mitigating compatibility issues.
     - Isolation: Each container operates independently, preventing interference and ensuring the encapsulation of dependencies.
     - Efficiency: Containers share the host OS kernel, optimizing resource utilization and reducing overhead.
     - Rapid Deployment: Docker's lightweight nature enables swift deployment, configuration, and scaling of services.

3. Docker Compose for Service Orchestration.

   - Streamlined Configuration: Docker Compose, a tool for defining and running multi-container Docker applications, is employed for orchestrating the simultaneous deployment of multiple containers. This streamlines the configuration process and ensures a synchronized launch of services.

- Simplified Management: Docker Compose simplifies the management of interconnected services. The declarative YAML syntax allows for the definition of service dependencies, easing the coordination of the backend, frontend, databases, and Grafana instances.

4. Benefits of Containerized Deployment.

   - Rapid Development Cycles: Containerization promotes an agile development cycle. Changes to one service can be implemented and tested independently, accelerating the overall development process.
   - Resource Efficiency: By sharing the host OS kernel, Docker containers consume fewer resources compared to traditional virtualization methods. This enhances overall system efficiency and resource utilization.
   - Scalability and Elasticity: Docker's container-centric architecture facilitates seamless scalability. Services can be scaled horizontally or vertically based on demand, ensuring optimal performance under varying workloads.

In conclusion, the adoption of Docker containers for the deployment of backend, frontend, and databases aligns with the project's commitment to agility, efficiency, and scalability. Docker's portability, isolation, and rapid deployment capabilities contribute to a streamlined development and operational process. The use of Docker Compose further enhances the orchestration and synchronization of services, simplifying the management of interconnected components. Embracing containerization stands as a strategic decision to foster an agile and efficient system deployment, ensuring the project's adaptability and responsiveness to evolving requirements.

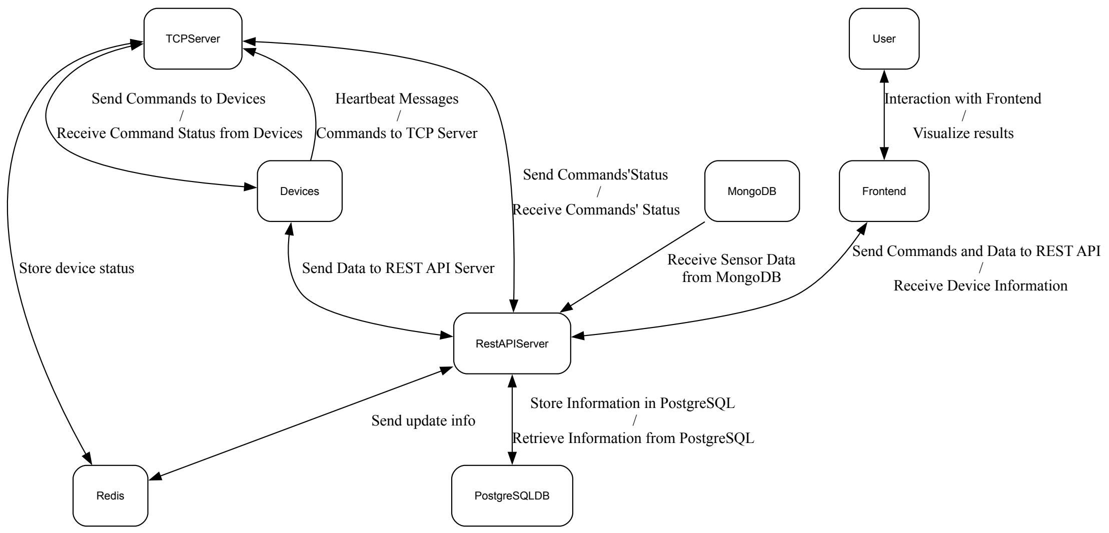

TCPServer
User
Send Commands to Devices / Receive Command Status from Devices
Heartbeat Messages / Commands to TCP Server
Interaction with Frontend / Visualize results
Send Commands'Status / Receive Commands' Status
Devices
MongoDB
Frontend
Store device status
Send Data to REST API Server
Receive Sensor Data from MongoDB
Send Commands and Data to REST API / Receive Device Information
RestAPIServer
Send update info
Store Information in PostgreSQL / Retrieve Information from PostgreSQL
Redis
PostgreSQLDB

Figure 4.1: Architecture Interactions' graph

As depicted in Figure 4.1, the user engages directly with the frontend, navigating through various screens to visualize data. From the web interface, users can interact with different components through API calls, enabling actions such as reviewing statistics, uploading updates, and checking available updates. The frontend, in turn, communicates with the REST API server to send requests for data retrieval or storage.

The REST API server serves as an essential point in the architecture. It establishes communication with the TCP server, facilitating the transmission of commands. This connection, although not direct, was made utilizing the Redis database, however this approach was not ideal due to the implementation and timing requirements, so a different approach was used, the event emitter module provided by Nodejs itself. Then the REST API server also interfaces with the databases, including MongoDB, to retrieve sensor data for presentation to the user, and PostgreSQL for managing information related to devices, users, and more. Moreover, the REST API server accepts API calls from devices, enabling the storage of data in the MongoDB database.

Concurrently, the TCP server manages direct communication with devices. It is responsible for sending commands and receiving back the execution statuses. Lastly, the devices play a crucial role in the system. They can transmit the data collected by the sensors to the REST API server for storing in the persistent database, execute the commands received, and periodically dispatch heartbeat messages to update their online and activity status to the TCP server.

This architectural arrangement establishes a robust and interconnected system, enabling seamless user interaction, data transmission, and command execution across different components of the platform.

4.1 Backend with Node.js

The backend engineering phase is pivotal in the development of a robust and scalable web application. This document outlines the meticulous engineering process using Node.js, Express.js, and Passport, focusing on the creation of a RESTful API and a TCP server for handling device connections.

RestAPI development:

1. Project Setup: The engineering process begins with setting up a Node.js project, utilizing tools like npm for package management. Express.js is integrated to facilitate the rapid creation of robust and scalable server-side applications.

2. Routing and Endpoint Design: RESTful API routes are meticulously designed to align with the application's functionalities. Each endpoint corresponds to a specific action or resource, providing a clear and logical structure to the API.

3. Middleware Integration: Express.js middleware, including those for error handling, authentication, and request parsing, are seamlessly integrated. Passport.js, a powerful authentication middleware, is employed to secure API endpoints through strategies such as JWT (JSON Web Token).

4. Database Integration: Database connections, often using popular databases like MongoDB or PostgreSQL, are established. Mongoose or Sequelize, respective Node.js Object Data Modeling (ODM) and Object-Relational Mapping (ORM) libraries, assist in defining data models and interacting with the database.

5. Security Measures: Security considerations, including input validation, encryption, and adherence to best practices like the principle of least privilege, are integrated to fortify the API against common vulnerabilities.

6. Testing and Documentation: The API is rigorously tested using tools like Mocha or Jest, ensuring the correctness of each endpoint. Documentation, often generated using tools like Swagger, is created to aid developers in understanding and utilizing the API effectively.

TCP connection handling:

1. Socket Management: The TCP server is engineered to manage socket connections efficiently. Node.js's native net module is commonly used for handling raw TCP connections.

2. Communication Protocols: Custom communication protocols are established for seamless interaction between devices and the server. JSON-based or binary protocols, depending on the application requirements, are implemented.

3. Event-Driven Architecture: The server adopts an event-driven architecture, leveraging Node.js's asynchronous capabilities to handle multiple device connections simultaneously. This ensures optimal performance even under heavy loads.

4. Error Handling and Recovery: Comprehensive error handling mechanisms are implemented to identify and address issues with device connections promptly. Automatic recovery mechanisms are designed to maintain continuous operation.

5. Security Measures: Security considerations extend to the TCP server, including encryption of data in transit and strict validation of incoming connections to prevent unauthorized access.

6. Scalability and Load Balancing: The server architecture is designed with scalability in mind. Load balancing strategies, possibly utilizing tools like nginx or built-in clustering in Node.js, are considered for even distribution of incoming device connections.

The backend engineering phase, utilizing Node.js, Express.js, and Passport, ensures the creation of a powerful and secure RESTful API and a responsive TCP server. Meticulous planning and adherence to best practices contribute to the development of a backend system that is reliable, scalable, and capable of handling the diverse demands of a modern web application.

4.2 Frontend with React.js

The frontend engineering process plays a crucial role in the development of a modern application, and React.js stands out as a key choice for creating a dynamic and highly responsive user interface. This document focuses on the component development phase, highlighting best practices and methodologies adopted in the engineering context.

1. Conceptual Decomposition: The initial phase involves the conceptual decomposition of the user interface into distinct elements, reflecting the modular nature of React.js. This process enables a clear understanding of functionalities and data flows within the application.

2. Component Design: Each component is designed independently, considering specific responsibilities and responsiveness to user inputs. Design is carried out with reusability in mind, promoting a scalable and maintainable structure.

3. Style and Layout: The application of style rules and layout design is integrated during the engineering phase. The use of methodologies such as CSS-in-JS or the adoption of CSS preprocessors contributes to maintaining a cohesive style and facilitates the management of layout dynamics.

4. State Management: The implementation of state management through React features, such as useState or useReducer, is integrated to ensure efficient and predictable handling of interface updates.

5. Navigation and Routing: For complex applications, the navigation and routing system is designed using common libraries like React Router. This allows for clean and intuitive management of navigation between different views of the application.

6. Testing and Validation: Each component undergoes targeted testing procedures to ensure its robustness and compliance with specifications. The adoption of testing frameworks like Jest and the integration of static analysis tools contribute to reducing the presence of bugs and improving overall code quality.

The main advantages of using React.js are:

1. Modularity and Reusability: React's composable nature allows the creation of standalone components, fostering modularity and code reuse.

2. Development Efficiency: The state management system and the presence of a virtual DOM enhance development efficiency, allowing developers to focus on application logic.

3. Active Community and Resources: React's extensive developer community provides a rich source of resources, documentation, and support, facilitating issue resolution and the adoption of best practices.

4. Agility in Change: The modular structure simplifies the introduction of changes and adaptation to new features without compromising system integrity.

The frontend engineering phase with React.js is a structured and methodical process aimed at realizing a sophisticated and highly performant user interface. Adopting the outlined best practices will contribute to ensuring code quality, long-term maintainability, and user satisfaction.

Chapter 5

Backend Architecture

The backend infrastructure of the project plays a primary role in orchestrating seamless communication between the frontend and connected devices. This section delves into a detailed overview of the backend architecture, adopting a bifurcated approach to enhance performance and streamline functionalities.

1. RESTful API Server: Managing Frontend and Device Interactions. The backend is specifically segregated into two distinctive servers, each assigned a specific role. The first, tasked with handling RESTful API requests, serves a dual purpose. It manages interactions with the frontend, addressing user-driven actions like login, device information display, and the creation of device groups. Simultaneously, it administers device-related tasks such as authenticating devices during the initial connection, negotiating configuration details for first-time setup, and processing incoming sensor information.

2. TCP Server: Dedicated Device Communication Hub. In contrast, the second server specializes in managing TCP connections with individual devices. This server maintains persistent connections with each device, facilitating the exchange of custom commands, periodic ping commands to ascertain device status, and other device-specific communications.

The decision to adopt a dual-server architecture stems from the need to prevent functional overlap and potential server overload caused by diverse request types. This approach ensures clarity in managing traffic patterns, distinguishing between user-oriented requests and device-specific commands.

Both servers are implemented in Javascript, fostering a unified coding structure. This strategic decision not only streamlines the development process but also contributes to a cohesive system architecture. In particular, Javascript is employed for TCP connections to harness the advantages of single-threaded execution.

In the TCP server handling device connections, the use of a single thread emerges as a crucial optimization strategy. This singular thread manages all requests emanating from various devices, eliminating the overhead associated with context switching. This proves especially vital in scenarios with a high influx of requests, preventing potential bottlenecks, server crashes, or unintended errors.

By distributing functionalities across two servers, the backend can comprehensively manage the diverse nature of incoming and outgoing traffic. The API server focuses on generalized data and information processing, while the TCP server excels in handling real-time, device-specific interactions.

The backend architecture, with its dual-server model and strategic use of Javascript, epitomizes a commitment to performance optimization. This specialized approach ensures that each server excels in its designated role, fostering efficient traffic management, reducing potential points of failure, and enhancing the overall robustness of the backend infrastructure.

Here is a little visualization of the interactions between the different components involved:

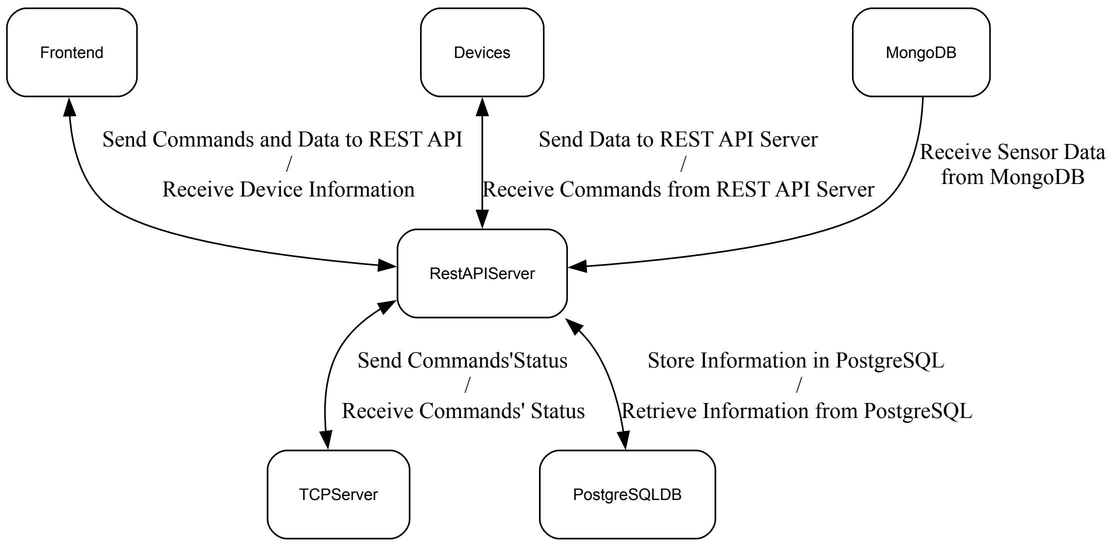

Frontend
Devices
MongoDB
Send Commands and Data to REST API / Receive Device Information
Send Data to REST API Server / Receive Commands from REST API Server
Receive Sensor Data from MongoDB
RestAPIServer
Send Commands'Status / Receive Commands' Status
Store Information in PostgreSQL / Retrieve Information from PostgreSQL
TCPServer
PostgreSQLDB

Figure 5.1: Architecture REST API Interactions' graph

As illustrated in Figure 5.1, the REST API server assumes a central role within the system, serving as a primary intermediary among various components. Its primary responsibility lies in handling requests originating from the frontend, facilitating the visualization of required data. Upon receiving commands, the REST API server orchestrates the forwarding of requests to either the MongoDB database for sensor data statistics or the PostgreSQL database for device or user information. Subsequently, it returns the retrieved data to the frontend for user consumption.

Furthermore, the REST API server takes charge of device authentication through API calls. It acts as the recipient of sensor data from devices, facilitated by additional API calls. Notably, the server was used to engage in interactions with the Redis database to transmit the commands from the Rest API server, however this was no longer used due to some constraints, so another approach was used, the event emitter approach provided by Nodejs. This approach allow the TCP server to listen for events coming from the Rest API server and forward the commands in real time to the devices allowing the immediate execution of the command on the device itself. In this context, the Redis database functions as a repository for update packages, specifically those requiring frequent downloads by numerous devices and other transitive information.

This comprehensive interaction framework positions the REST API server as a pivotal component in managing data flow, authentication processes, and communication across the various segments of the platform.

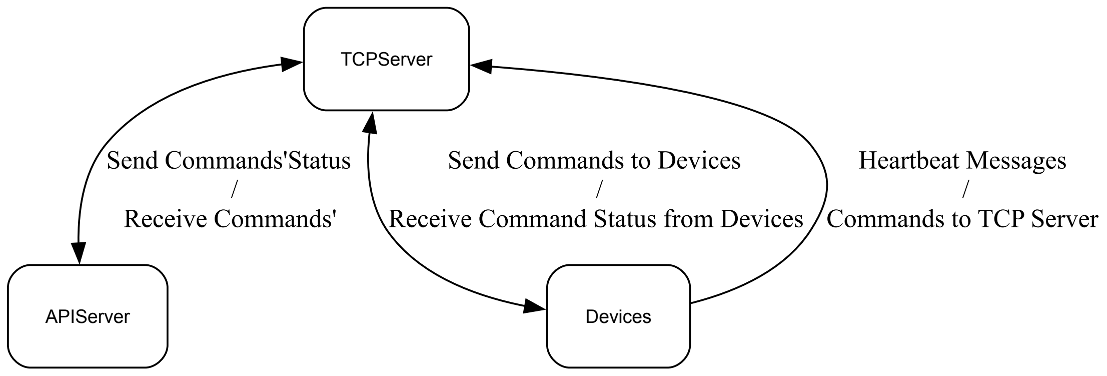

TCPServer
Send Commands'Status / Receive Commands'
Send Commands to Devices / Receive Command Status from Devices
Heartbeat Messages / Commands to TCP Server
APIServer
Devices

Figure 5.2: Architecture TCP Server Interactions' graph

As depicted in Figure 5.2, the TCP server stands as a fundamental component within the system, designated with the task of managing interactions with the connected devices. Its primary responsibilities encompass the retrieval and transmission of commands to be executed by the devices. The TCP server acts as an intermediary for forwarding commands to devices and subsequently storing the status of command executions if requisite.

In addition to its role in command propagation, the TCP server plays an important role in monitoring the vitality of connected devices. It achieves this through the reception of heartbeat packets from devices, signifying their operational status. This mechanism ensures the continuous awareness of the server regarding the status of individual devices, allowing for timely responsiveness and intervention as needed.

Moreover, the TCP server is intricately involved in the orchestration of bidirectional communication. It manages the transmission of commands to devices and concurrently oversees the retrieval of status updates concerning the execution of these commands. This dual functionality positions the TCP server as a critical element in maintaining the real-time synchronization and operational integrity of the system.

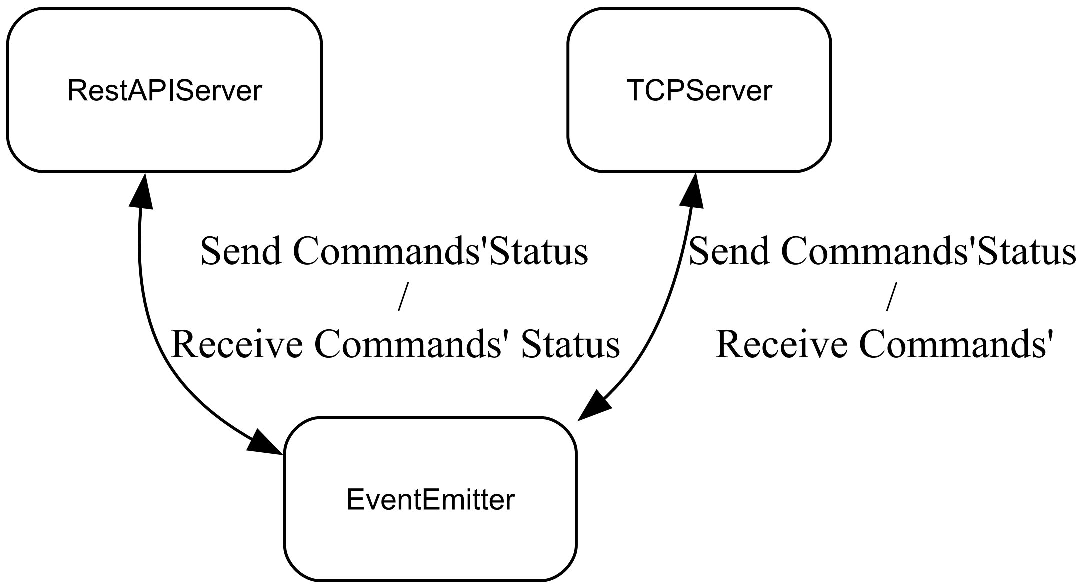

RestAPIServer
TCPServer
Send Commands'Status / Receive Commands' Status
Send Commands'Status / Receive Commands'
EventEmitter

Figure 5.3: Architecture Backend's graph

Within the illustrated architectural framework, as depicted in Figure 5.3, a pivotal facet revolves around the intricate interplay between the TCP server and the REST API server. Their communication is specialized, primarily oriented towards the exchange of commands and the subsequent receipt of command results. Notably, this interaction does not follow a direct trajectory; instead, it is mediated through the adept utilization of the event emitter library inherent to Node.js.

In this orchestrated communication model, a dedicated listener is established on the TCP server, perpetually attuned to potential commands emanating from the REST API server. Upon receipt of a command, the TCP server promptly relays it to the client, facilitating immediate execution. Simultaneously, the result of the operation is conveyed back to the TCP server, streamlining the entire process. This methodology circumvents the need for polling a predetermined list of commands, thereby enhancing operational efficiency and promoting instantaneous command execution.

Crucially, this communication paradigm eschews the initial employment of Redis as an intermediary communication bridge. Instead, it adopts an event-driven approach, leveraging the event emitter library. This strategic choice is poised to further augment the platform's performance, offering the distinct advantage of executing commands promptly, devoid of any latency associated with waiting periods.

This refined event-driven mechanism not only expedites command execution but also contributes to the optimization of platform performance. The elimination of polling-related delays ensures that commands are processed immediately, aligning seamlessly with the real-time operational demands of the system.

In essence, the adoption of this event-driven architecture stands as a testament to the platform's commitment to optimal performance and responsiveness. The deliberate choice to forego traditional intermediary solutions in favor of a more direct, event-based approach underscores the platform's agility and adaptability within a dynamic operational context.

5.1 REST API Management

In the foundational phase of creating the backend server dedicated to handling RESTful API requests, meticulous consideration is given to the selection of key Javascript libraries. Express, Morgan, CORS, and Passport collectively form a robust ensemble to rapidly process a spectrum of incoming requests, ensuring seamless interactions not only with the frontend but also with other pertinent components of the system.

Components overview:

1. Express: A Foundation for Web Applications. At the core of the REST API server lies Express, a powerful and widely adopted web application framework for Node.js. Express simplifies the process of defining routes, handling HTTP requests, and structuring responses. Its minimalist design fosters flexibility, enabling developers to construct a tailored architecture that aligns precisely with project requirements.

2. Morgan: Streamlining Logging Capabilities. Augmenting the robustness of the server, Morgan is incorporated to provide a streamlined logging mechanism. This library facilitates the generation of log outputs, aiding in the systematic tracking of incoming requests. Morgan's customizable logging capabilities prove invaluable for debugging, performance monitoring, and the generation of insightful analytics.

3. CORS: Mitigating Cross-Origin Resource Sharing Concerns. Cross-Origin Resource Sharing (CORS) intricacies are effectively addressed through the integration of the CORS library. CORS facilitates secure communication between the frontend and backend, overcoming browser restrictions related

to cross-origin requests. By strategically configuring CORS, the server ensures that only authorized origins can access its resources, fortifying security protocols.

4. Passport: Enabling Authentication Strategies. Passport.js emerges as a pivotal component for implementing authentication strategies. Its modular and adaptable architecture supports the integration of various authentication mechanisms, including username/password, social logins, and more. Passport seamlessly integrates with Express, streamlining the process of authenticating users and safeguarding access to sensitive resources.

Strategic implementation:

1. Comprehensive Request Handling: The amalgamation of Express, Morgan, CORS, and Passport empowers the server to comprehensively handle incoming requests. Whether originating from the frontend or other system components, the server processes requests optimally, orchestrating appropriate responses based on the nature of the user's needs.

2. Modular Extensibility: The modular nature of the selected libraries enhances the server's extensibility. Future adaptations and integrations can be seamlessly implemented without compromising the existing architecture. This flexibility is pivotal for accommodating evolving project requirements and ensuring scalability.

The meticulous selection of Express, Morgan, CORS, and Passport for the RESTful API server reflects a commitment to establishing a synergistic and resilient foundation. This framework not only ensures effective communication with the frontend but also establishes a modular infrastructure poised for adaptability and growth. By strategically leveraging these libraries, the server stands prepared to navigate the intricacies of API management with precision and reliability.

In tandem with the communication orchestration handled by Express, Morgan, CORS, and Passport, the backend architecture incorporates additional libraries to seamlessly integrate with SQL and MongoDB databases. This strategic integration is essential for robust data management, encompassing operations such as retrieval, storage, and processing.

In tandem with the communication orchestration handled by Express, Morgan, CORS, and Passport, the backend architecture incorporates additional libraries to seamlessly integrate with SQL and MongoDB databases. This strategic integration is essential for robust data management, encompassing operations such as retrieval, storage, and processing.

Knex: SQL Database Interaction. The utilization of Knex as a SQL query builder and migrator underscores its pivotal role in ensuring the coherence and integrity of the SQL database. Key functionalities of Knex within this context include:

- Database Schema Management: Knex plays a critical role in guaranteeing the existence of requisite tables within the SQL database. Upon receiving requests, it verifies the presence of necessary tables and dynamically generates them if absent. This proactive approach precludes potential issues during data retrieval or storage operations.

- Request Processing: Incoming requests directed at the SQL database are meticulously processed by Knex. It extracts pertinent information from requests, optimizing the data for seamless interaction with the database. This intermediate processing step contributes to the refinement of subsequent operations.

- Data Retrieval and Storage: Knex acts as the bridge between the backend server and the SQL database, facilitating the retrieval and storage of data. By interfacing with the database, Knex ensures that requested information is accurately retrieved and appropriately stored, maintaining the integrity of the database.

- User Authentication Support: User details, required for Passport library authentication, find a secure repository within the SQL database. Knex oversees the storage and retrieval of user-related data, enhancing the security framework by ensuring that only authorized users gain access to sensitive functionalities.

MongoDB Library: Managing NoSQL Data Flow. In parallel to Knex, the MongoDB library assumes a central role in handling NoSQL data flow, primarily interfacing with MongoDB. Its functionalities encompass:

- Data Storage for Devices: Incoming data from devices, encoded in MessagePack format, is decoded and seamlessly stored in the MongoDB database. This process ensures the persistence of device-generated information, contributing to a comprehensive historical record.

- Efficient Query Processing: MongoDB's query capabilities are harnessed to promptly process requests related to device data. The library navigates the intricacies of MongoDB, providing optimal query execution and enhancing the backend's responsiveness to requests.

The integration of Knex for SQL database operations and the MongoDB library for NoSQL data flow establishes a cohesive and versatile data management infrastructure. This dual approach not only ensures the reliability and security of SQL database interactions but also enables quick handling of device-generated data in the MongoDB database. By strategically employing these libraries, the backend not only manages the present data needs but also lays the groundwork for scalable and adaptive data operations in the future.

5.2 Device-Specific Information Handling

The TCP server, constituting the second pivotal component of the backend infrastructure, delineates a distinct set of functionalities compared to its REST API counterpart. Tailored for handling individual device connections, this segment of the server relies on a unique set of libraries, namely TLS, FS, and MessagePack-Lite, each playing a specialized role to ensure secure and efficient communication.

TLS Library: Fortifying Communication Channels. At the heart of securing communication between the server and individual devices lies the TLS (Transport Layer Security) library. Its multifaceted role encompasses:

- Secure Channel Establishment: TLS, coupled with the server's certificate and key files, orchestrates the establishment of a secure channel. This secure channel is instrumental in fostering encrypted communication between the TCP server and connected devices, safeguarding data against unauthorized access or interference.
- Data Encryption: Leveraging TLS, the TCP server encrypts the data exchanged between itself and the connected devices. This encryption ensures that the information traversing the communication channel remains confidential and immune to eavesdropping or tampering attempts.

FS Library: Facilitating Key and Certificate File Handling. The FS (File System) library assumes a required role in handling the key and certificate files indispensable for TLS-based secure connections. Key functionalities include:

- Key and Certificate Access: FS facilitates the seamless access and retrieval of the server's key and certificate files. These files are imperative for initiating secure connections with devices, and FS ensures their availability for TLS library integration.
- Dynamic File Handling: As the key and certificate files are dynamically generated, FS ensures their availability and responsiveness to changes. This dynamic handling is vital for maintaining the server's security credentials and adaptability to evolving security requirements.

MessagePack-Lite: Efficient Data Serialization. While TLS and FS focus on secure communication establishment, MessagePack-Lite addresses the efficient serialization and deserialization of data exchanged with connected devices. Key functionalities include:

- Binary Data Serialization: MessagePack-Lite excels in the serialization of data into binary format. This binary serialization enhances data transfer spedds, minimizing overhead and ensuring swift communication between the TCP server and devices.

- Compact Data Representation: The compact representation of data facilitated by MessagePack-Lite optimizes the payload transmitted between the server and devices. This improvement is particularly important in the context of real-time data exchange and responsiveness.

The integration of TLS, FS, and MessagePack-Lite into the TCP server establishes a robust foundation for secure and efficient communication with individual devices. By prioritizing secure channel establishment, key and certificate handling, and optimized data serialization, this component ensures a reliable and resilient architecture for handling the intricacies of device connections. The implementation not only addresses immediate security concerns but also positions the system for scalability and adaptability in accommodating future security enhancements.

Once a secure connection is established between the TCP server and an individual device, a comprehensive device management system comes into play. This system is meticulously designed to track, authenticate, and interact with connected devices in a secure and meaningful manner.

Device Identification and Tracking. Upon successful connection establishment, the TCP server diligently records the IP address of the connected device. This meticulous tracking mechanism serves a dual purpose:

- Device Inventory: By storing IP addresses, the server maintains a real-time inventory of all connected devices. This inventory becomes instrumental in uniquely identifying each device, ensuring that communications are correctly routed to the intended recipient.

- Connection Monitoring: Continuous tracking allows the server to monitor the status of each connection. It ensures that the server remains aware of active devices, facilitating dynamic responsiveness to changes in the device landscape.

Data Reception and Decoding. Subsequent to the device's registration, the TCP server transitions into a listening state, eagerly awaiting incoming data. Upon data reception, the server initiates a series of operations:

- Message Decoding: The received data, encapsulated in MessagePack format, undergoes a decoding process. This process involves unpacking the binary data and converting it into a structured format that the server can comprehend.

- Padding for Data Structure: To ensure uniformity in data structures, the server may introduce padding bytes to complete any missing elements. This step is crucial for maintaining consistency in data processing and interpretation.

Varied Communication Types. The nature of communication between the server and devices spans multiple types, each serving a distinct purpose:

- Authentication: Authentication messages validate the legitimacy of the connected device. This step ensures that the communicating entity is a recognized and authorized device, mitigating the risk of unauthorized access.
- Heartbeat: Periodic heartbeat messages signify the vitality of the connected device. By acknowledging regular heartbeats, the server confirms the continuous presence and functionality of the device, thereby sustaining the TCP connection.
- Custom Commands: The server can transmit custom commands to connected devices. These commands, ranging from system operations like reboot or shutdown, enable remote management of devices, enhancing operational control.

The TCP server's role extends beyond mere connection establishment. Through meticulous tracking, data reception, and varied communication types, it ensures a holistic and secure management system for all connected devices. This comprehensive approach not only fortifies the system against unauthorized access but also empowers seamless communication and control over the network of devices.

Chapter 6

Frontend Architecture

The frontend, serving as the user's gateway to seamless interaction with the server and devices, plays an important role in facilitating a rich and intuitive user-experience. Specifically designed and crafted, the frontend encompasses an array of functionalities designed to provide comprehensive insights and control over the connected ecosystem.

1. React: A Paradigm Shift in User-Experience. The decision to employ React as the framework for the frontend heralds a paradigm shift from a static multipage application to an agile single-page application (SPA). This shift is rooted in the inherent advantages that React brings to the table, promising a more responsive and dynamic user interface.

2. Unveiling the Power of Single-Page Applications. React, renowned for its virtual DOM rendering and component-based architecture, facilitates the creation of a SPA. This departure from a traditional multipage structure bestows several advantages:

   - Enhanced Responsiveness: The SPA model ensures a more fluid and responsive user-experience. By dynamically updating only the necessary components, React minimizes page reloads, resulting in quicker transitions between different views.
   - Streamlined User Navigation: With React, navigation within the application becomes more streamlined. Users can seamlessly traverse between different views without the jarring interruptions associated with traditional multipage applications.
   - Optimized Resource Utilization: The modular nature of React components allows for optimized resource utilization. Components are loaded on-demand, reducing initial page load times and enhancing overall performance.

3. Feature-Rich User Interface. The frontend unfolds a feature-rich interface, offering an array of pages tailored to the different needs:
   - Device Registry: Users can peruse a comprehensive list of registered devices, accessing vital details and configurations associated with each.
   - Active Devices Dashboard: A dynamic dashboard provides real-time insights into active devices, their status, and pertinent information.
   - Statistical Analytics: Users can delve into detailed statistics, ranging from telemetry data to device positions, empowering them with a holistic view of the connected ecosystem.
4. User-Centric Design Philosophy. The design of the frontend is underpinned by a user-centric philosophy. The user interface is intuitive, ensuring a seamless interaction that aligns with the diverse needs of both novice and seasoned users.
5. Future-Proofing with React. By embracing React, the company not only addresses the immediate requirements but also future-proofs the frontend against evolving technological landscapes. React's robust community support and continual evolution position the frontend for adaptability and scalability.

In conclusion, the frontend architecture, anchored by React, stands as a testament to the company's commitment to transformative user-experiences. The migration to a SPA model heralds a new era of responsiveness and agility, positioning the frontend as a dynamic interface empowering users in their interactions with the server and connected devices.

ReactJS: A Key Enabler of Efficient Development.

- Component-Based Architecture: React's component-based architecture dovetails seamlessly with Agile principles. The modularity allows for the creation of reusable components, promoting efficiency in development and ensuring a consistent design language across the application.
- Virtual DOM Rendering: React's virtual DOM rendering minimizes page reloads, optimizing the performance of the frontend. This aligns with Agile's emphasis on delivering increments of functionality rapidly.
- Continuous Integration and Deployment (CI/CD): Implementing CI/CD practices ensures that code changes are swiftly integrated, tested, and deployed. This fosters a continuous flow of deliverables, aligning with Agile's commitment to frequent, incremental releases.

User Feedback: An Iterative Catalyst.

- User Stories and Acceptance Criteria: User stories, encapsulating user requirements, and acceptance criteria, defining the conditions for feature acceptance, form the backbone of development. This user-centric approach ensures that the frontend aligns precisely with user expectations, that are the expectations of the client in this case.
- Continuous User Feedback: A central tenet of Agile methodologies involves continual user feedback. Regular user testing and feedback loops enable the team to make timely adjustments, refining the frontend in response to evolving user needs.

The Agile methodology, complemented by the Scrum framework, proves instrumental in steering the ReactJS frontend development. The iterative nature of Agile, coupled with React's efficiency-enhancing features, ensures that the frontend remains responsive, adaptable, and aligned with user expectations throughout the development lifecycle.

The creation of a user-friendly web frontend necessitates the strategic incorporation of various libraries and components. This section delineates the carefully chosen components and libraries employed, underscoring their role in fashioning an accessible and seamless user interface.

1. Navigating Seamlessly with React Router DOM.
   - Purposeful Navigation: The integration of React Router DOM is important for seamless navigation between distinct pages. This library allows the navigation of the application effortlessly, ensuring a cohesive and intuitive user experience.
   - Multi-Page Interaction: React Router DOM facilitates the realization of a multi-page application structure. This capability is fundamental for accommodating diverse functionalities and presenting them in an organized manner, optimizing user interaction.
2. Authentication Context for Session Persistence
   - Contextual Session Management: The implementation of the Auth Context library addresses the imperative of session persistence. This ensures that the user's context persists across various pages, offering a continuous and secure experience throughout the browsing session.
   - Enhanced Security: By securely managing session data, Auth Context contributes to the overall security of the application. User authentication and authorization processes are streamlined, fortifying the frontend against unauthorized access.

3. Tailwind CSS: A Framework for Graphical Finesse.
    - Streamlined Styling: Tailwind CSS emerges as a foundational framework for crafting the graphical interface. Its utility lies in the expedited styling of components, ensuring a visually cohesive and aesthetically pleasing frontend.
    - Customizable Design: Tailwind's modular approach allows for extensive customization. This adaptability is instrumental in tailoring the frontend's visual elements to align precisely with the desired user interface aesthetics.
4. Custom Components for Bespoke Functionalities.
    - Tailored User Experience: In conjunction with the aforementioned libraries, custom components have been specifically designed and implemented. These components are purpose-built to deliver bespoke functionalities, enriching the user-experience and catering to specific user requirements.
    - Functionality-Driven Design: Each custom component encapsulates specific functionalities, ensuring a function-driven design philosophy. This approach enhances the overall user interaction, providing a responsive and tailored experience.

In conclusion, the amalgamation of React Router DOM, Auth Context, Tailwind CSS, and custom components engenders an aesthetic and functional frontend ecosystem. This ecosystem is not only adept at facilitating seamless navigation and session persistence but also showcases a visually appealing and functional interface. The careful selection and integration of these libraries and components are important in realizing an intuitive and user-centric ReactJS frontend.

The integration of numerous APIs stands as a cornerstone in establishing robust communication channels between the frontend and the backend server. These APIs facilitate the seamless exchange of information, encompassing data retrieval and storage operations within the database. This section elucidates the purposeful implementation of APIs to orchestrate efficient interactions with the backend server.

1. Purposeful API Integration.
    - Establishing Communication: The implementation of APIs serves as a conduit for establishing clear and purposeful communication channels between the frontend and the backend server. This communication is foundational for the exchange of data, enabling a dynamic and responsive user interface.
    - Data Retrieval and Storage: APIs are instrumental in the retrieval of information from the server as well as the storage of data within the

database. These operations are executed through meticulously crafted API endpoints, ensuring a streamlined and secure data flow between the frontend and backend components.

2. Efficient Data Transmission.
   - Seamless Information Exchange: APIs are designed to facilitate seamless information exchange between the frontend and backend. By adhering to established protocols and endpoints, data is transmitted efficiently, optimizing the overall performance of the application.
   - Minimized Latency: The architecture of these APIs prioritizes minimized latency, ensuring that data retrieval or storage processes occur with optimal speed. This focus on response times contributes to a rapid and user-friendly frontend experience.
3. Robust Processing and Validation.
   - Data Processing: APIs are endowed with robust data processing capabilities. This includes parsing incoming data, executing necessary computations, and formatting information before presenting it to the user. Such processing enhances the quality and relevance of the data exchanged.
   - Validation Mechanisms: Implemented APIs feature validation mechanisms that guarantee the integrity and security of the transmitted data. These mechanisms serve as a protective barrier, preventing the ingestion of erroneous or malicious data into the system.
4. Scalability and Adaptability.
   - Scalable Architecture: The API architecture is conceived with scalability in mind. This ensures that the application can accommodate increased data flow and user interactions without compromising performance. The modular design allows for the seamless addition of new functionalities.
   - Adaptive Response Handling: APIs are equipped with adaptive response handling mechanisms. This ensures that the frontend can gracefully handle diverse responses from the backend server, fostering a resilient and fault-tolerant system.

In conclusion, the careful implementation of APIs forms a critical juncture in establishing a synergistic interaction between the frontend and backend components. These APIs serve as conduits for efficient data transmission, robust processing, and secure validation. The commitment to minimizing latency and fostering scalability ensures a responsive and adaptable architecture. The integration of APIs stands as a testament to the commitment to crafting a dynamic and user-centric application architecture.

6.1 React.js Implementation

The frontend serves as the primary interface facilitating user interaction with various project components, including servers and devices. This interface operates in an indirect and abstract manner, shielding users from the inherent complexity associated with communication among diverse project elements. The web interface is strategically designed to streamline user comprehension and simplify the execution of intricate tasks.

The web interface encapsulates the intricacies of interactions between distinct components, rendering it user-friendly and accessible. Through a series of web pages, it is possible to seamlessly perform a spectrum of operations essential to the project's functionality. It serves as a gateway to engage with the system's diverse functionalities, providing an intuitive and efficient means of task execution.

6.1.1 Login

To access the functionalities of the web interface, user authentication is imperative. Account creation is exclusively permitted by system administrators, ensuring that only company employees and authorized associates possess access privileges. Registration is not open to direct user initiation; rather, accounts are specifically curated, created, and approved by administrators. This stringent control mechanism ensures the system's security and restricts access to authorized personnel.

Post-administrator approval and creation of user accounts, the login process becomes the gateway to the comprehensive suite of functionalities. Users gain entry into the system through the dedicated login page, leveraging credentials established during the account creation process.

So, the web interface stands as a primary component in facilitating user engagement with the architecture's intricate ecosystem. Its design prioritizes clarity, accessibility, and security, ensuring also that everyone can efficiently navigate and utilize the platform's diverse capabilities in a easy way without sacrificing features, functionalities nor security.

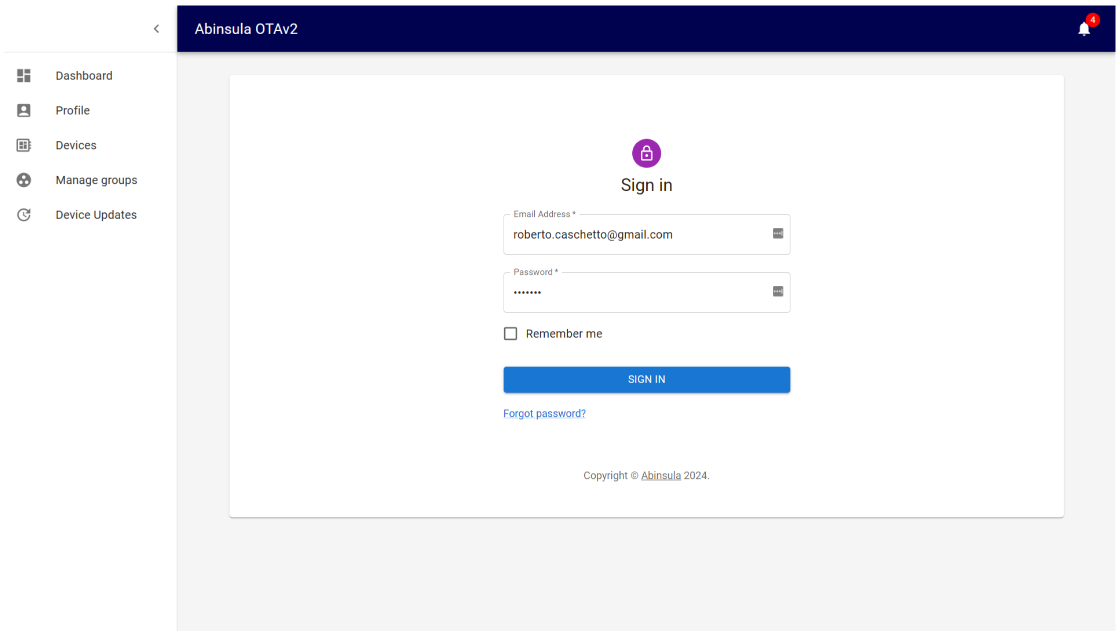


Figure 6.1: Login page

6.1.2 Dashboard

Upon successful user authentication, the system automatically redirects the user to the initial landing page, known as the dashboard. The dashboard serves as a comprehensive information hub, presenting a wealth of statistics and insights pertaining to registered devices and the accumulated data. This centralized interface offers a real-time overview of the project's ecosystem, providing at-a-glance information on device statuses, such as online or offline indicators in proportion.

The dashboard's functionality extends beyond mere data presentation; it serves as a dynamic platform for users to glean meaningful insights into the operational status of the registered devices. From there it is possible to efficiently monitor device status, collected data, and recent activities through intuitive visualizations and status indicators. These may include graphical representations of data trends, network connectivity, and temporal markers indicating the recency of device updates.

Moreover, the dashboard's design emphasizes user-friendly navigation, allowing to seamless drill down into specific device categories or data subsets for more detailed analyses. This hierarchical approach allows the explororation in granular details while maintaining an overarching perspective of the project landscape.

The incorporation of interactive elements further enhances user experience, enabling users to tailor the dashboard to their specific preferences and monitoring needs. For instance, it may be possible to choose to visualize only certain devices, data types, or performance metrics, fostering a personalized and adaptive dashboard environment.

The dashboard stands as the primary interface post-login, offering a sophisticated and informative gateway to the project's intricate data landscape. Its multifaceted capabilities empower users to not only stay informed about the current project state but also to delve deeper into the nuances of device performance and data dynamics.

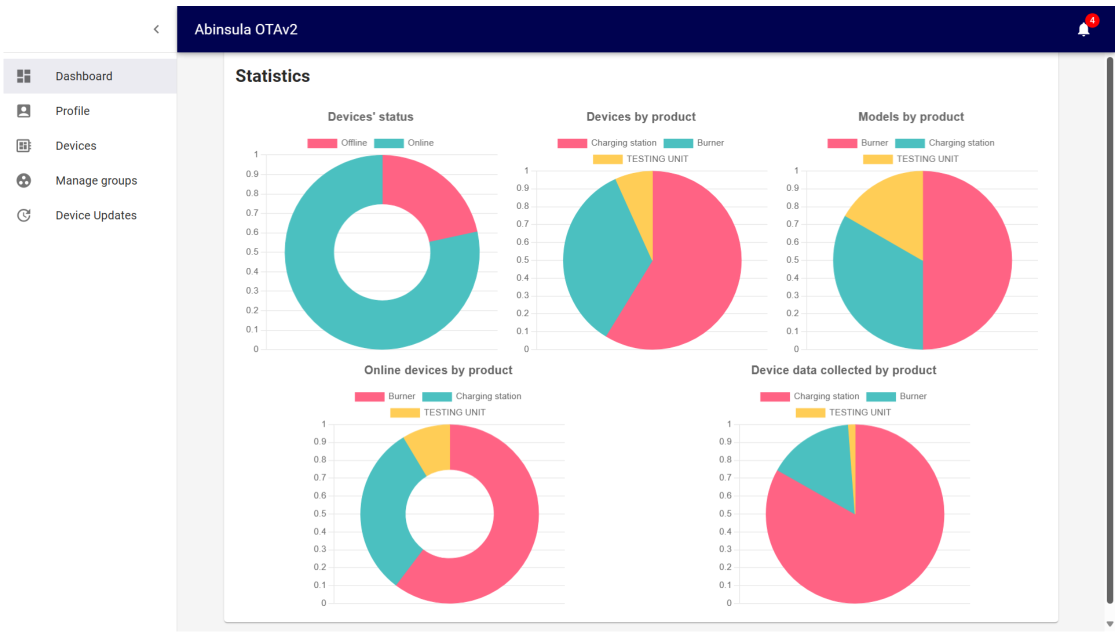


Figure 6.2: Dashboard Page

The dashboard is designed to furnish detailed information with a comprehensive snapshot of the diverse array of devices affiliated with the system. This interface serves as main hub, providing a broad understanding of the available data, the status of connected devices, and other pertinent details encompassing the entirety of devices associated with the platform. It is imperative to note that a device's initial connection triggers a registration process, rendering it traceable within the platform's ecosystem.

The dashboard, depicted in Figure 6.2, serves as a comprehensive informational hub, offering an overview of critical metrics related to the connected devices. It furnishes essential insights, including the total count of connected devices, categorization of connected devices based on models, the distribution of devices across products, the aggregation of models per product, and the cumulative volume of data collected by all devices categorized by product. This general presentation facilitates a high-level understanding of the company's device landscape, encompassing aspects such as device statuses (online/offline) and the aggregate data accumulation across various products.

While the dashboard provides a bird's-eye view of the device ecosystem, more detailed and specific information is available on dedicated pages tailored for distinct functionalities. These granular pages allows to delve deeper into specific aspects of the platform. Nevertheless, the dashboard stands as the primary interface, offering a convenient way to gain a comprehensive understanding of the company's diverse array of devices.

Moreover, the dashboard serves as an instrumental tool for monitoring the performance of devices, highlighting crucial indicators such as online device counts, model-specific distributions, and the overall data collection metrics. This strategic presentation aids in quickly grasping the current state of the device network without delving into more specialized pages.

The charts employs an interactive approach to enhance the comprehensibility of the presented data. The graphical representations incorporated into the dashboard are designed to be interactive, providing an enriched experience in data exploration. This interactive capability is particularly exemplified in the utilization of graphs where users can employ mouse hover actions to reveal detailed labels associated with specific data points. These labels not only elucidate the nature of the data but also furnish with precise numerical values, fostering a deeper understanding of the information being conveyed.

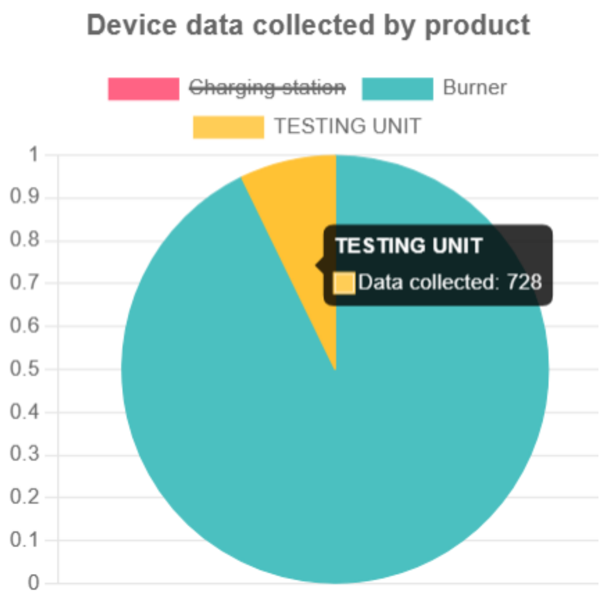


Figure 6.3: Dashboard's chart

Moreover, the interactive functionality extends to the comparative analysis of data. By clicking on specific labels within the graph, it is possible to selectively include or exclude them from the visualization. This feature facilitates a dynamic exploration of the data, allowing the focus on specific elements of interest while temporarily removing others for a clearer perspective. Such a nuanced approach to data visualization empowers users to tailor their analytical process, enhancing their ability to discern patterns and trends.

The decision to integrate interactivity into the graphical elements of the dashboard is rooted in the commitment to user-centric design principles. By providing these tools for dynamic exploration and customization, the platform seeks to optimize the clarity and relevance of the presented data. This approach aligns with the overarching goal of making complex datasets more accessible and user-friendly, ensuring a meaningful insights from the information at viewer's disposal.

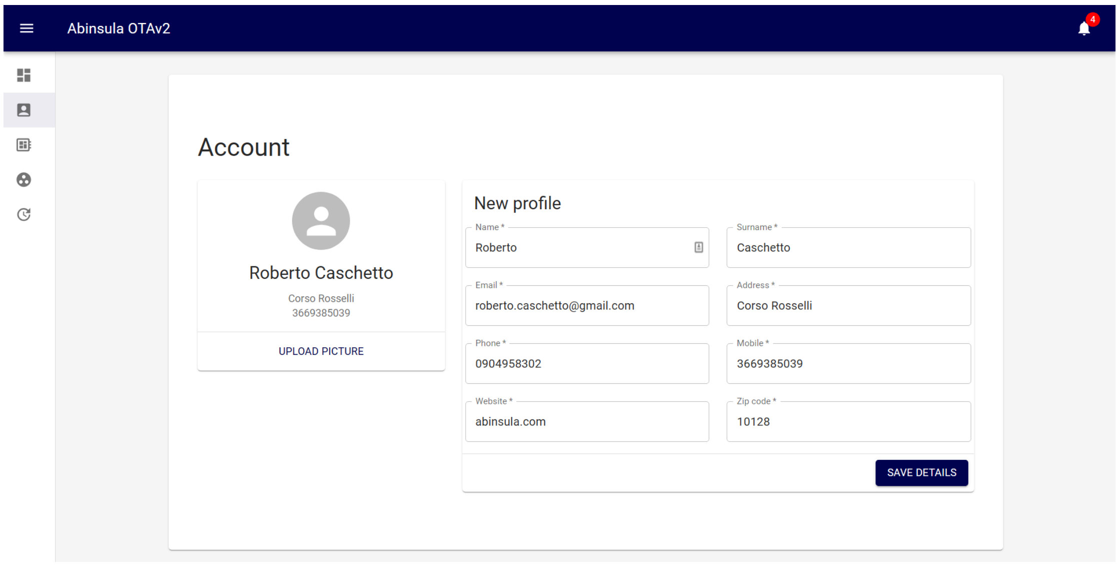
In summary, the dashboard acts as a strategic control center, providing at-a-glance insights into the company's device landscape. Its role extends beyond basic monitoring, offering a nuanced perspective on device connectivity, model distribution, and data accumulation across various products. This centralized interface is pivotal for users seeking a swift and comprehensive overview of the platform's operational dynamics. Also, the interactive features embedded in the dashboard's graphical representations serve to elevate the user experience by offering detailed insights and customization options. This deliberate design choice contributes to the platform's commitment to providing a robust and user-friendly environment for data analysis and interpretation.

6.1.3 User's profile

Subsequently, the platform incorporates a Profile Page, allowing the autonomy to personalize and manage the individual information of the account. Within this domain, it is possible to effectually tailor the profile details, including but not limited to email addresses, passwords, and other pertinent personal information. This feature is particularly advantageous for ensuring that user accounts are kept current and aligned with any changes in personal or professional details.

An additional facet of the Profile Page involves the option to upload a personal customized profile picture. This functionality serves a dual purpose, contributing to both user identification and the establishment of a more personalized and recognizable user interface. By allowing users to associate a visual representation with their profiles, the platform fosters a sense of individualization and facilitates ease of recognition, thereby enhancing the overall user experience.

Moreover, the Profile Page stands as a secure and user-centric space, ensuring that modifications to sensitive account information adhere to robust security protocols. The emphasis on user customization and self-management underscores the platform's commitment to providing a user-friendly and adaptable environment, catering to the individual preferences and requirements of each user within the system.



Account

Roberto Caschetto
Corso Rosselli
3669385039

New profile

Name * Roberto	Surname * Caschetto
Email * roberto.caschetto@gmail.com	Address * Corso Rosselli
Phone * 0904958302	Mobile * 3669385039
Website * abinsula.com	Zip code * 10128

Figure 6.4: Edit Profile Information

The profile page, as illustrated in Figure 6.4, offers a streamlined and user-friendly interface for the modification of user-specific information. The simplicity of the design is evident in the intuitive interaction model, where the user can effortlessly edit data fields through straightforward input textboxes. Modification of the data is facilitated by entering the desired changes and subsequently executing the save operation by selecting the save button. This straightforward process ensures a seamless and efficient means of updating user information.

Notably, the ease of use extends to the modification of the user's profile image. The user is presented with the option to upload a custom image directly from their local drive. This feature enhances personalization and recognition, as users can tailor their profile visuals to suit their preferences. Upon selecting and uploading the desired image, the save operation finalizes the update, encompassing both textual data modifications and the newly chosen profile image.

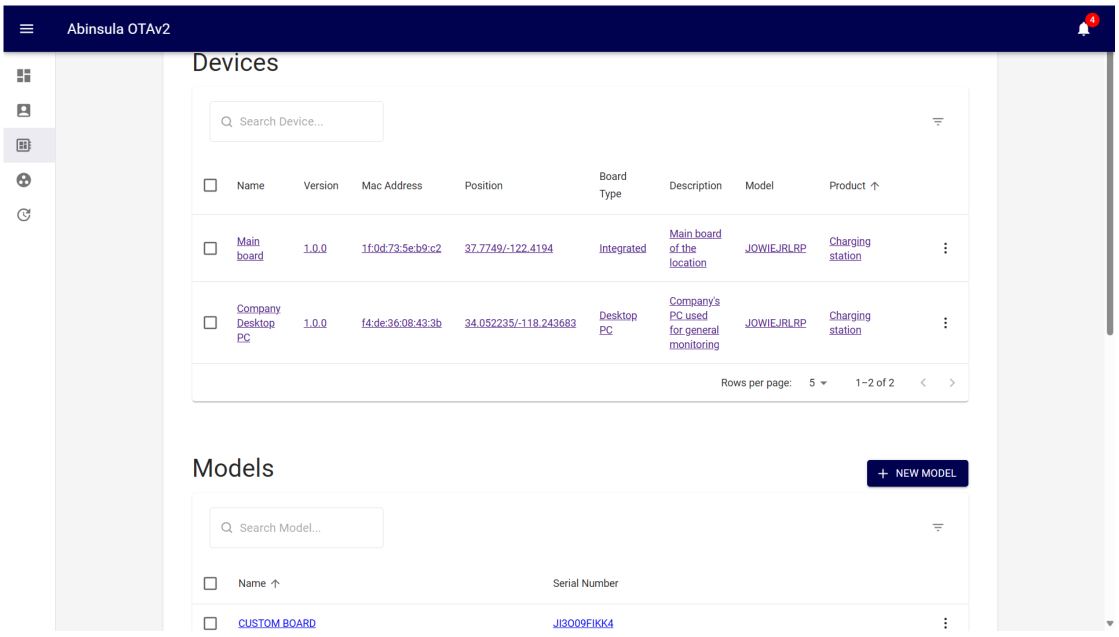
This user-centric approach to profile management not only prioritizes simplicity but also emphasizes accessibility. The straightforward modification process aligns with user expectations, minimizing the learning curve associated with updating personal information. This commitment to user-experience is pivotal in ensuring that users can seamlessly navigate and personalize their profiles without unnecessary complexity.

6.1.4 Device's info

The platform encompasses a dedicated segment known as the Devices Page, strategically designed to facilitate comprehensive management of products, models, and devices within the system. Within this interface, users are afforded the capability to not only peruse the catalog of created products but also to institute new product entries. This foundational functionality is instrumental in allowing users to maintain an organized inventory of distinct products associated with their operational scope.

Furthermore, the platform extends its utility by incorporating a module for the delineation and establishment of models. It is imperative to note that models are intricately linked to specific products, thereby exemplifying a form of specialization inherent to the overall product hierarchy. Users, through the Devices Page, are empowered to both visualize existing models and institute novel ones, thereby contributing to the systematic categorization and management of devices within the platform.

In this paradigm, the inherent relationship between models and products underscores the hierarchical structuring of devices, promoting an organized and logical framework. The entwined nature of models and products serves as a pivotal attribute, contributing to the streamlined management of devices and ensuring coherence in the representation of the overall system architecture.



Devices

Name	Version	Mac Address	Position	Board Type	Description	Model	Product ↑
Main board	1.0.0	1f:0d:73:5e:b9:c2	37.7749/-122.4194	Integrated	Main board of the location	JOWIEJRLRP	Charging station
Company Desktop PC	1.0.0	f4:de:36:08:43:3b	34.052235/-118.243683	Desktop PC	Company's PC used for general monitoring	JOWIEJRLRP	Charging station

Rows per page: 5 1–2 of 2

Models

Name ↑	Serial Number
CUSTOM BOARD	JI3O09FIKK4

Figure 6.5: Device list page

As depicted in Figure 6.5, the dedicated device list page offers a comprehensive overview of registered devices, presenting key details that provide a foundational understanding of the device landscape. The presented information encompasses crucial attributes related to each device, offering insights into their status, connectivity, and other relevant particulars.

The design of this page is meticulously crafted to furnish users with a succinct yet informative snapshot of the registered devices. By centralizing essential information in a cohesive manner, users can efficiently survey the array of devices under consideration. This approach not only streamlines the user experience but also lays the groundwork for more detailed exploration.

Each device entry on this page serves as a gateway to more in-depth insights. Upon selecting a specific device, users are seamlessly directed to dedicated pages that furnish a wealth of additional information. These individual device pages act as comprehensive repositories, providing intricate details about the selected device, its operational status, and associated statistics.

This hierarchical structure, where a general overview is accessible from the device list page, and detailed specifics are further unveiled on dedicated pages, is purposefully designed to balance accessibility and depth. Users can tailor their engagement with the platform based on their informational needs, navigating effortlessly between the broader device landscape and the granular intricacies of individual devices.

Furthermore, the user-centric approach extends to the responsive design of these pages, ensuring a seamless transition between different levels of information. The aim is to empower users with a flexible and intuitive interface that adapts to their preferences and exploration patterns.

The device list page stands as an important component within the platform's architecture, providing users with a centralized view of registered devices and serving as a launching point for more detailed investigations. This design philosophy aligns with the overarching objective of delivering a user-friendly and informative environment for effective device management.

A notable feature within the Devices Page is the comprehensive presentation of available devices, each intricately linked to a specific model. It is noteworthy that the creation of devices is inherently automated and contingent upon successful authentication during the initial connection to the platform. Devices, upon undergoing a seamless authentication procedure with the server, are seamlessly incorporated into the devices list, thereby attaining visibility within the system.

The authentication process serves as a critical security measure, warranting that only authenticated devices are afforded access to the panoply of operations permissible within the platform. Once a device successfully authenticates, it assumes its designated place within the devices list, whereupon it gains unfettered access to the spectrum of operations sanctioned for devices within the platform's operational milieu. This judicious approach to device management ensures a secure and controlled ecosystem, aligning with industry best practices for device-centric platforms.

Within the platform's user interface, the Devices Page serves as a pivotal hub for interacting with and obtaining detailed insights into individual products, models, and devices. A user can navigate through the comprehensive array of items by selecting them, prompting the system to dynamically load a dedicated page containing exhaustive information and pertinent statistics. This user-friendly approach streamlines the process of accessing critical data associated with each element.

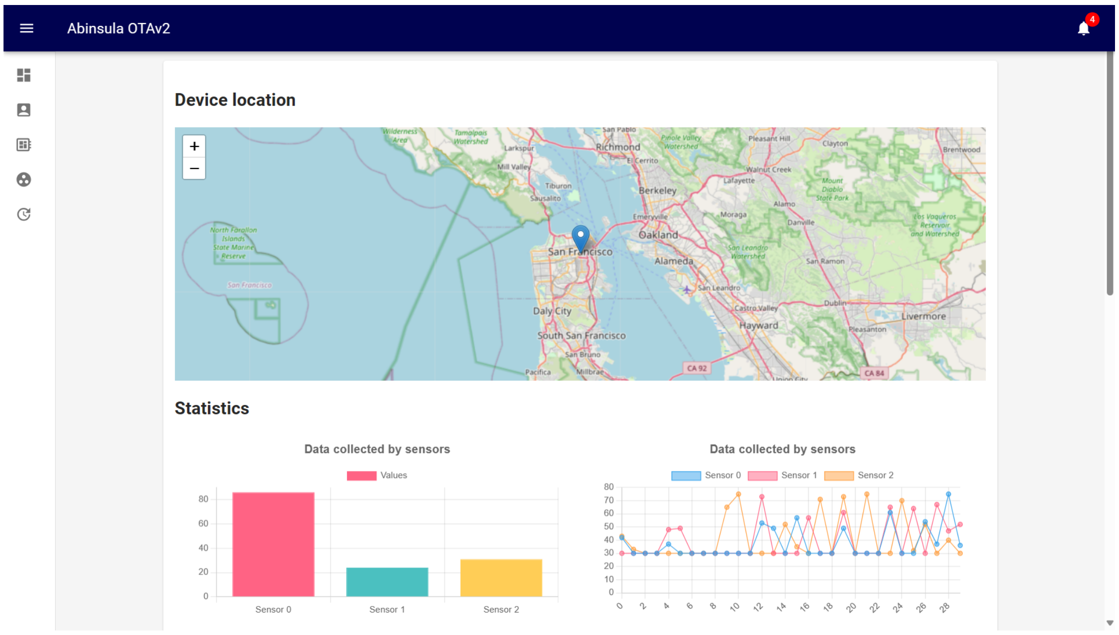


Figure 6.6: Device Info Page

Upon clicking a specific product, model, or device, a distinct page is presented, meticulously designed to showcase not only the intrinsic details of the selected entity but also to furnish users with pertinent statistics reflective of its operational parameters. It is noteworthy that the depth of information varies between models and devices, with the former offering insights into both the model itself and the encompassing product to which it is linked. Similarly, devices provide a holistic view by presenting not only their individual attributes but also incorporating details from the associated model and overarching product.

As illustrated in Figure 6.6, the device information page serves as a comprehensive repository of detailed insights into a specific device within the platform. This dedicated page is designed to offer users a nuanced understanding of the device's operational status, geographical location (when available), and an in-depth analysis of the collected sensor data.

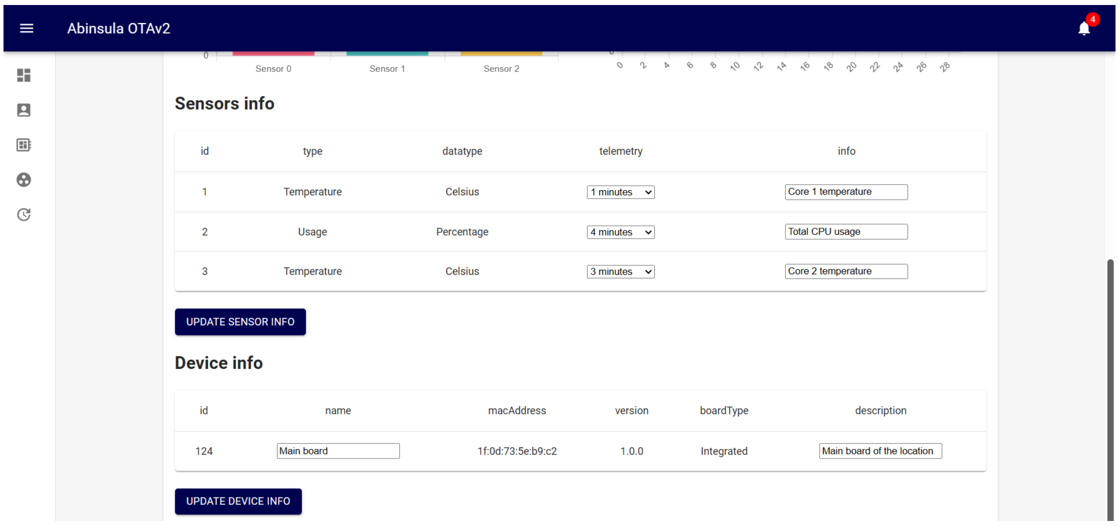
The initial segment of the page is devoted to geographical representation. A map interface is employed to visualize the device's location, leveraging coordinates sourced from either an embedded GPS module or stored within the device's configuration file. This spatial contextualization provides users with a tangible reference point, especially valuable for scenarios where the physical placement of devices holds significance.

Continuing with the commitment to user interactivity, the subsequent sections of the page incorporate two dynamic graphs. On the left, a graph delineates the volume of data amassed by each distinct sensor integrated into the device. This visual representation aids in quickly assessing the data contribution of individual sensors, contributing to a holistic understanding of the device's data generation dynamics.

Simultaneously, the graph on the right offers a temporal dimension by showcasing the last 30 data samples collected by each sensor. The overlay of each sensor's data facilitates comparative analysis, enabling users to discern patterns, anomalies, or correlations in the sensor data. This interactive graph empowers users to tailor their exploration, allowing them to focus on specific sensors or exclude certain data labels for a refined visual examination.

Moreover, the responsive design of this page ensures an optimal viewing experience, adapting to various screen sizes and orientations. This adaptability enhances the accessibility of detailed device information across different devices and usage scenarios.

The device information page, represented in Figure 6.7, extends its functionality to empower users not only with insights into the device's status and sensor data but also with the capability to modify pertinent information. This multifaceted feature set enhances the platform's user-centric approach, allowing for a more tailored and responsive management of individual devices and associated sensors.



Sensors info

id	type	datatype	telemetry	info
1	Temperature	Celsius	1 minutes	Core 1 temperature
2	Usage	Percentage	4 minutes	Total CPU usage
3	Temperature	Celsius	3 minutes	Core 2 temperature

UPDATE SENSOR INFO

Device info

id	name	macAddress	version	boardType	description
124	Main board	1f:0d:73:5e:b9:c2	1.0.0	Integrated	Main board of the location

UPDATE DEVICE INFO

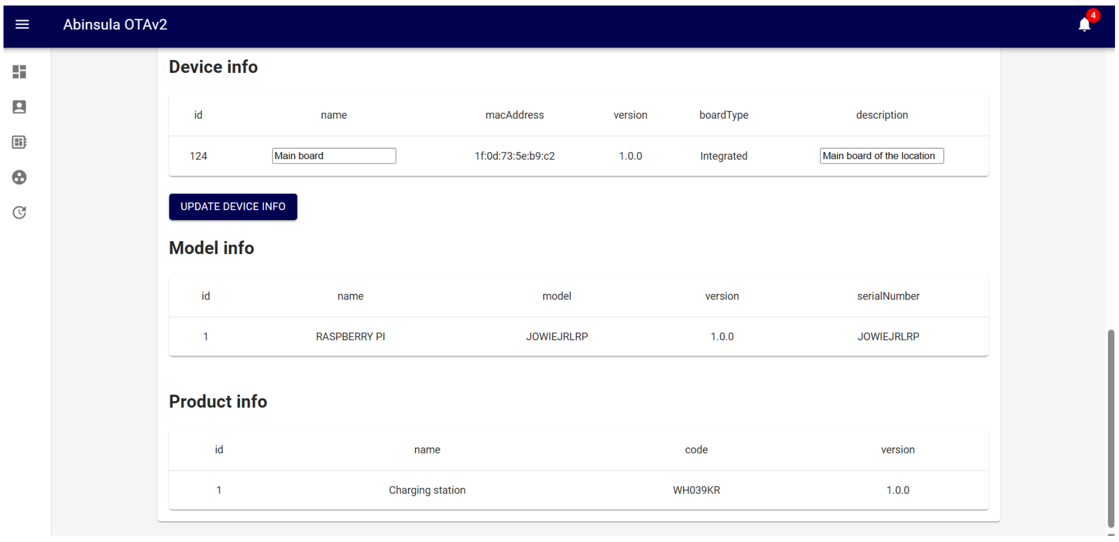
Figure 6.7: Device and sensor Info

Within the depicted interface shown in figure 6.7, users gain access to a granular view of the sensors installed on the device, accompanied by details about the device itself. This includes crucial information about the sensors' characteristics and, notably, the device's telemetry settings. The platform accommodates a dynamic interaction model, enabling users to not only peruse this information but also to effect changes as needed.

One key facet of customization lies in the ability to adjust the telemetry value, essentially determining the frequency at which the device transmits data to the server for storage. This feature is presented with a range of values, from 0 (indicating the cessation of data collection) to the maximum frequency allowable, as specified in the device configuration file. This flexibility in telemetry adjustment empowers users to align data transmission with specific operational requirements, optimizing both resource utilization and data currency.

Furthermore, the device information page facilitates modifications to the sensor attributes. Users can readily modify sensor names, providing a more intuitive and comprehensible identifier than the default ID. This customization is particularly valuable in scenarios where a user-friendly nomenclature enhances the clarity of device management. Additionally, a brief description field is provided, allowing users to append contextual information about the sensor's purpose or specific functionalities. This descriptive layer contributes to a more comprehensive understanding of the sensor's role within the device ecosystem.

The user-friendly interface is designed for ease of interaction, ensuring that modifications are intuitive and efficient. Upon effecting changes, users can seamlessly save the updated information by clicking on the corresponding button. This iterative and responsive design aligns with the broader platform philosophy of simplifying complex tasks and promoting user autonomy.



Device info

id	name	macAddress	version	boardType	description
124	Main board	1f:0d:73:5e:b9:c2	1.0.0	Integrated	Main board of the location

Model info

id	name	model	version	serialNumber
1	RASPBERRY PI	JOWIEJRLRP	1.0.0	JOWIEJRLRP

Product info

id	name	code	version
1	Charging station	WH039KR	1.0.0

Figure 6.8: Model and Product Information

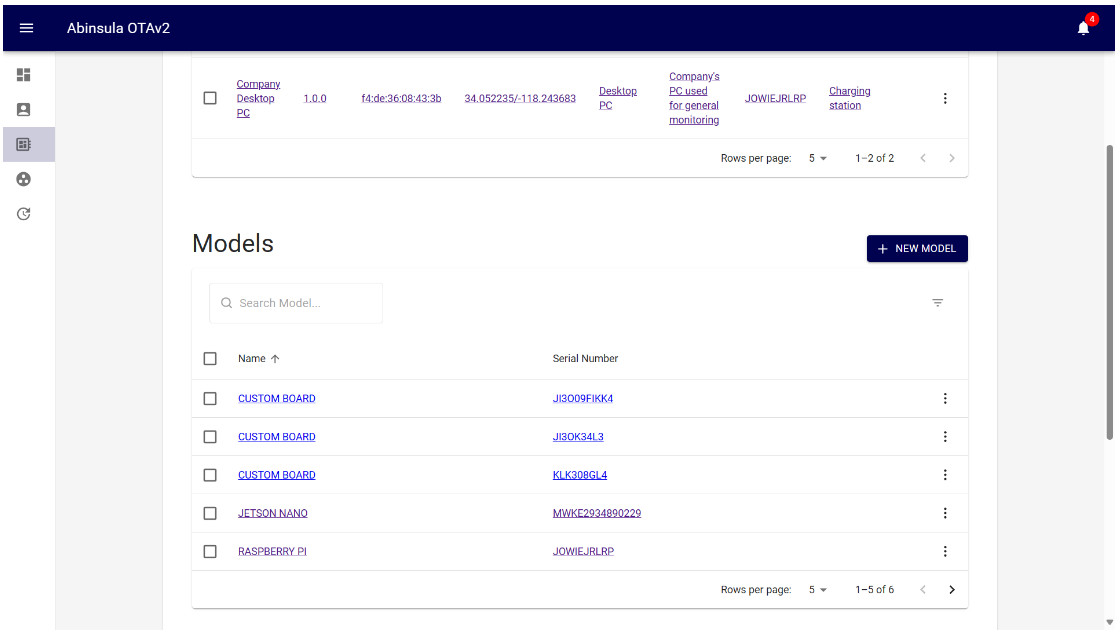
In the end of the page, shown in Figure 6.8, it encapsulates a comprehensive display of essential details concerning both the model and the associated product. This amalgamation of information serves as a centralized repository, offering users a holistic view of pertinent attributes associated with the specific product.

Within this interface, users can readily access a wealth of information pertaining to the model. This includes, but is not limited to, details such as the model's serial number and its given name. This foundational data provides users with a clear and unambiguous identifier for the model, facilitating streamlined navigation and management within the platform.

Complementing the model information, the interface also incorporates key insights into the associated product. This encompasses fundamental details such as the product's identification number, often referred to as its ID, and the product's name. The integration of these details ensures that users are not only apprised of the model specifics but also gain a contextual understanding of the broader product to which it is tethered.

In a broader context, this unified presentation of model and product information aligns with the platform's overarching philosophy of providing users with comprehensive insights to facilitate informed decision-making. By offering a consolidated view, the platform seeks to enhance user efficiency and effectiveness in managing and understanding the intricacies of their device inventory.

In essence, the device information page epitomizes the platform's commitment to delivering detailed and accessible insights. By integrating geographical context and interactive data visualizations, this page serves as a focal point for users aiming to delve into the intricacies of individual devices, fostering informed decision-making and analysis. The page not only serves as an informational hub but also as an interactive platform for users to fine-tune and customize device and sensor parameters. This blend of insight and control enhances the adaptability and user empowerment aspects of the platform, fostering a more robust and user-centric device management experience. This user interface serves as a testament to the commitment to user-centric design principles. It not only disseminates information but does so in a manner that prioritizes clarity, coherence, and ease of use. The model and product information interface stands as a pivotal component in the platform's architecture, enriching user interactions and contributing to a more nuanced understanding of the devices and models in their purview.



Company Desktop PC	1.0.0	f4:de:36:08:43:3b	34.052235/-118.243683	Desktop PC	Company's PC used for general monitoring	JOWIEJRLRP	Charging station

Rows per page: 5 1–2 of 2

Models

Name	Serial Number
CUSTOM BOARD	JI3O09FIKK4
CUSTOM BOARD	JI3OK34L3
CUSTOM BOARD	KLK308GL4
JETSON NANO	MWKE2934890229
RASPBERRY PI	JOWIEJRLRP

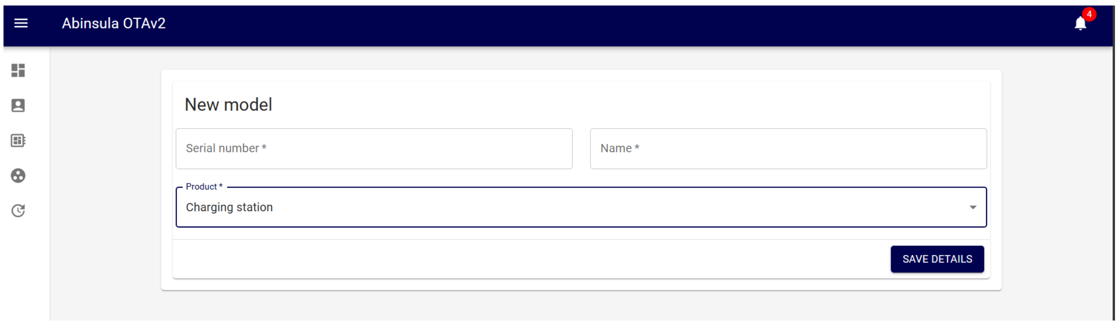
Rows per page: 5 1–5 of 6

Figure 6.9: Models' Information List

Returning to the main page, as depicted in Figure 6.5, a dedicated section is allocated to the management of various device models, as illustrated in Figure 6.9. This segment is instrumental in facilitating the creation and administration of distinct models, a prerequisite for the successful connection and registration of devices within the platform.

Within the platform's operational framework, the creation of models holds paramount significance. These models serve as a foundational layer, forming the basis upon which individual devices establish their identity and functionality. The process involves the manual creation of models, wherein users define and configure specific attributes that characterize a particular model. This comprehensive detailing encompasses essential parameters that delineate the operational characteristics of devices associated with the model.

The creation of models linked to products is a mandatory precursor to the seamless integration of devices into the platform. Each device, upon connection, aligns itself with a predefined model, establishing a standardized framework for data transmission, device behavior, and compatibility with updates. This systematic approach streamlines device management, ensuring a coherent and structured environment within the platform.



New model

Serial number *

Name *

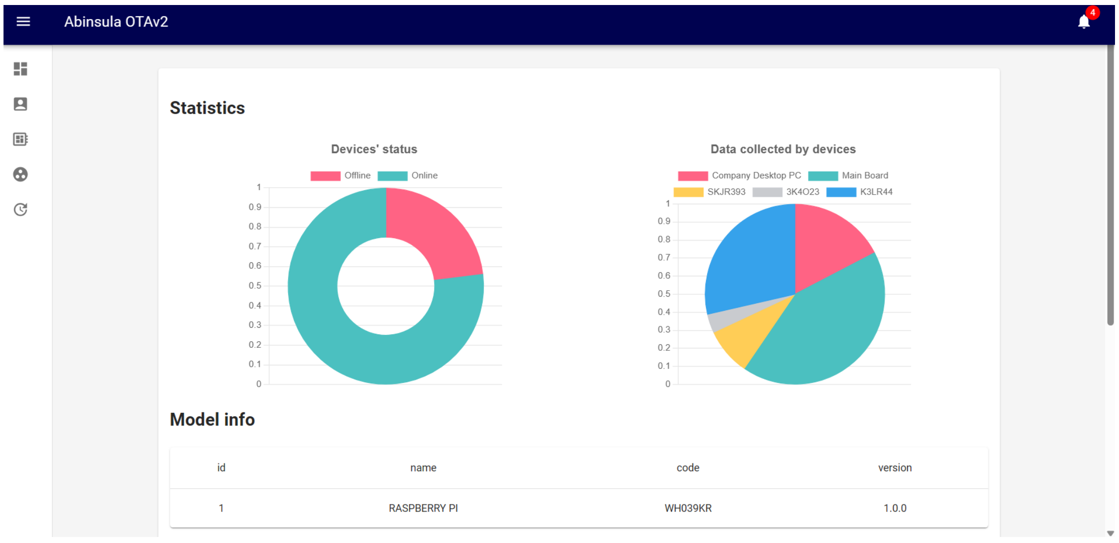
Product *
Charging station

Figure 6.10: Model Creation Page

The interface presented in Figure 6.9 not only allows the manual creation of new models specifically linked to existing products, shown in figure 6.10, but also provides a holistic view of existing models. Users can peruse a comprehensive list of models, each encapsulating vital information such as the model name and associated product. This consolidated representation affords users a quick overview of the existing models within the platform.

Moreover, the user interface is designed to foster an intuitive and user-friendly experience. The process of creating a new model involves specifying attributes and configurations relevant to the intended functionality of devices aligned with that model. This includes associating the model with a specific product, establishing a hierarchical relationship that contributes to the organized structuring of devices within the platform.

In essence, the dedicated models section within the platform serves as a pivotal control point, enabling users to define, manage, and organize the diverse array of devices based on standardized models. This strategic layer of abstraction not only enhances the platform's operational efficiency but also contributes to the establishment of a coherent and scalable framework for device integration and management.



Statistics

Model info

id	name	code	version
1	RASPBERRY PI	WH039KR	1.0.0

Figure 6.11: Model information

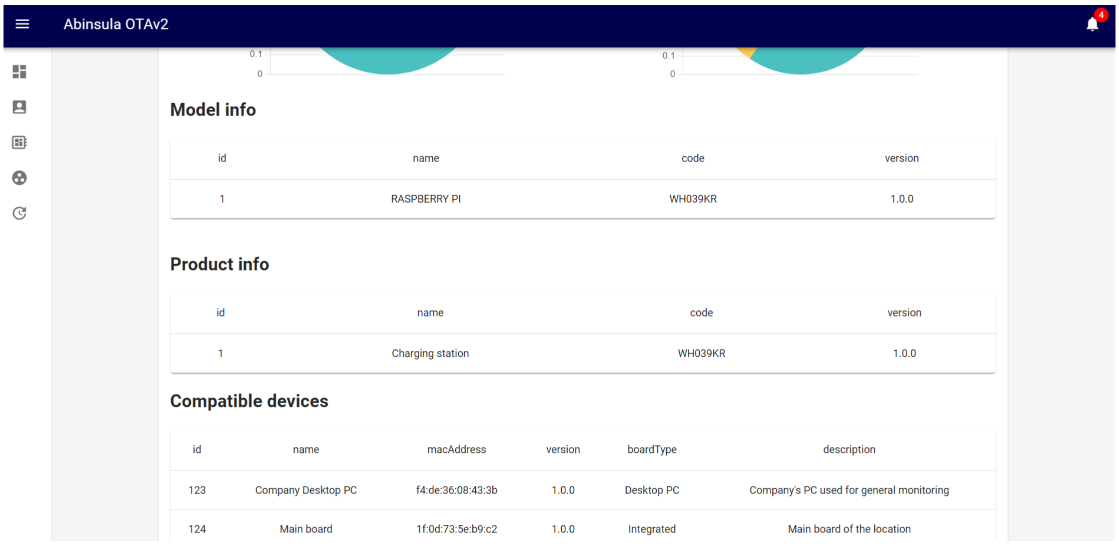
Upon selecting a specific model from the provided list, users are directed to a dedicated page, akin to those illustrated in Figures 6.6, 6.7, and 6.8. However, the information presented on this page is subtly distinct, as exemplified in Figure 6.11. This tailored section furnishes users with a nuanced perspective, offering insights into the operational dynamics of the selected model.

In the upper segment of the page, a series of interactive charts unfolds, providing a visual representation of pertinent data related to the model's ecosystem. These charts specifically illuminate details regarding the online status of devices affiliated with the model, coupled with an in-depth exploration of data collected by the top five devices within the same model category. This visual analytics approach empowers users with a comprehensive understanding of the model's performance metrics and the contribution of individual devices towards data generation.

Furthermore, the interface extends beyond graphical representations to furnish users with explicit details pertaining to the selected model. This includes foundational information such as the model's name, associated product, and a succinct description elucidating its intended purpose and functionality within the platform. This consolidated display encapsulates essential metadata, providing users with a holistic overview of the model's attributes and its significance within the broader context of the platform.

The user interface is intuitively designed to facilitate seamless navigation and comprehension. Users can effortlessly traverse through the presented information, leveraging the interactive charts for dynamic insights and scrolling through detailed textual information for a comprehensive understanding of the model's characteristics.

In essence, the model-specific page serves as a centralized repository of actionable intelligence. It amalgamates visual analytics with textual insights, empowering users to make informed decisions regarding the operational status and relevance of a particular model within the broader spectrum of the platform. This strategic interface not only enriches the user experience but also contributes to the platform's overarching goal of providing a robust and intuitive environment for the management and analysis of diverse device models.



Model info

id	name	code	version
1	RASPBERRY PI	WH039KR	1.0.0

Product info

id	name	code	version
1	Charging station	WH039KR	1.0.0

Compatible devices

id	name	macAddress	version	boardType	description
123	Company Desktop PC	f4:de:36:08:43:3b	1.0.0	Desktop PC	Company's PC used for general monitoring
124	Main board	1f:0d:73:5e:b9:c2	1.0.0	Integrated	Main board of the location

Figure 6.12: Model, Product and Devices' information

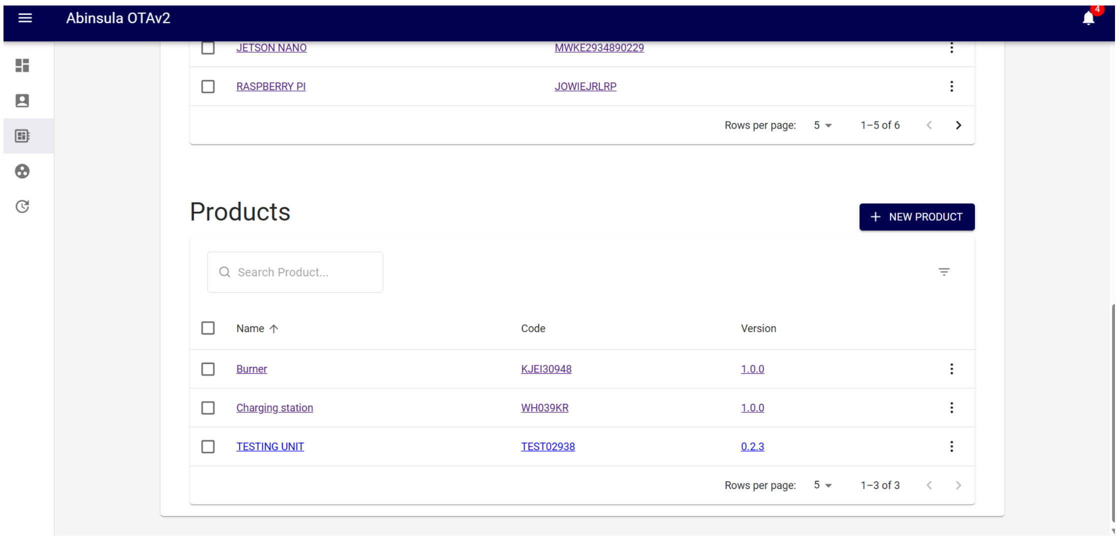
Continuing the exploration of the model-centric page, a detailed exposition unfolds, presenting a layered comprehension of crucial elements encapsulated by the selected model. This comprehensive representation, delineated in Figure 6.12, not only encompasses the model's intrinsic attributes but also extends its purview to encompass vital details regarding the associated product and the constellation of devices tethered to the model.

In the upper echelon of this interface, an amalgamation of graphical and textual elements converges to offer users a multifaceted understanding. The graphical segment, characterized by interactive charts, delves into the operational dynamics of the model. These charts, as previously delineated, furnish users with real-time insights into the online status of devices within the model and the granular specifics of data contributions from the top five devices. This visual analytics layer serves as an invaluable tool for users seeking to discern performance trends and hierarchies within the model's device landscape.

Simultaneously, the interface unfurls textual information pertaining to both the model and its associated product. This includes fundamental attributes such as the model's nomenclature, the linked product, and a concise yet elucidative description outlining the model's intended utility and functionality within the broader platform ecosystem. This synthesis of graphical representation and textual exposition contributes to a holistic comprehension of the model's significance and operational nuances.

Navigating further down the page, a tableau of information unfolds, shedding light on all devices interlinked with the selected model. This comprehensive listing facilitates a panoramic view of the model's device ensemble, offering a nuanced understanding of the diverse devices tethered to its framework. The synergy between the top-tier device performance charts and the inclusive device registry affords users a contextualized appraisal of the model's overall device landscape.

In essence, this model-centric interface serves as a pivotal nexus, uniting disparate yet interrelated facets of the platform's ecosystem. It not only empowers users with actionable insights into the model's real-time performance but also provides a panoramic view of its associated product and the ensemble of devices contributing to its operational footprint. This strategic amalgamation of graphical analytics and textual elucidation aligns with the platform's overarching ethos of furnishing users with an intuitive and comprehensive toolset for effective model management and analysis.


Abinsula OTAv2

JETSON NANO	MWKE2934890229
RASPBERRY PI	JOWIEJRLRP

Rows per page: 5 1–5 of 6

Products

NEW PRODUCT

Search Product...

Name	Code	Version
Burner	KJEI30948	1.0.0
Charging station	WH039KR	1.0.0
TESTING UNIT	TEST02938	0.2.3

Rows per page: 5 1–3 of 3

Figure 6.13: products' Information List

Reverting to the primary interface showcased in Figure 6.9, a designated section is expressly earmarked for the oversight of diverse products, as exemplified in Figure 6.13. This sector assumes a pivotal role in expediting the origination and oversight of discrete products, an imperative prelude to the seamless integration and management of devices within the platform.

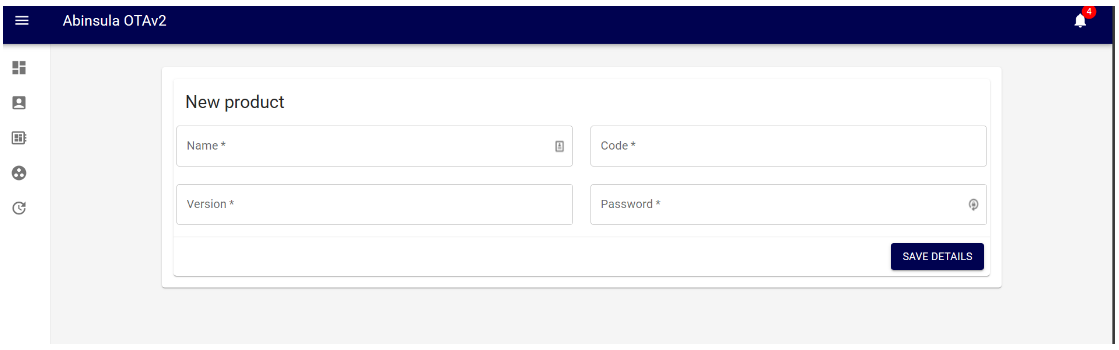
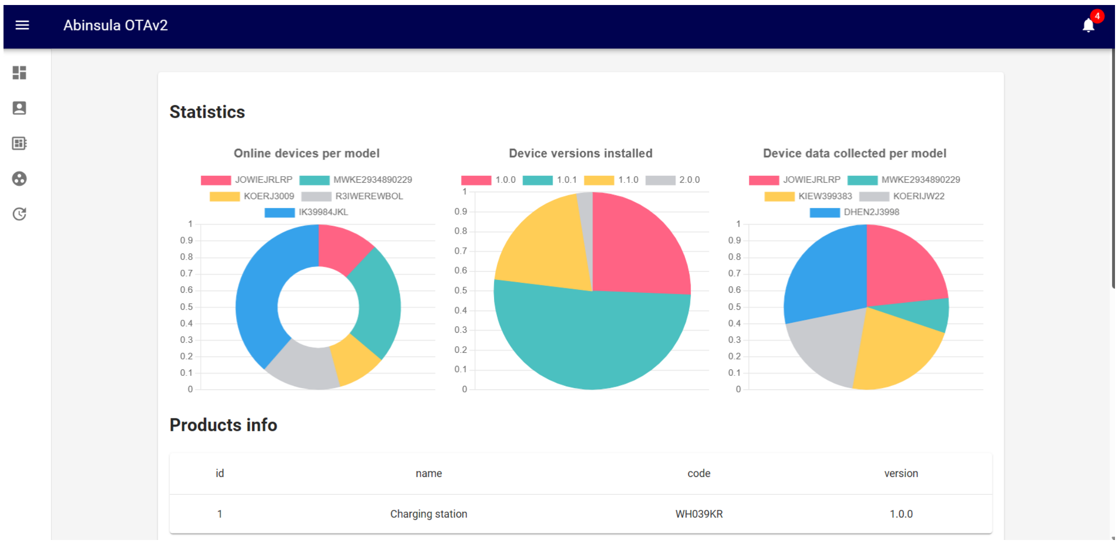


Figure 6.14: Product Creation Page

Within this segment, users are bestowed with the functionality to meticulously create new products manually, as shown in the figure 6.14. This procedural prerequisite is indispensable to facilitate the unimpeded connection and enrollment of devices into the platform's fold. The user interface here, while intuitive, underpins the foundational operational paradigm where the genesis of distinct products is an essential precursor to the broader orchestration of the interconnected device ecosystem.

This purposeful delineation aligns with the platform's architectural underpinning, emphasizing a modular and hierarchical structure. The ability to create and delineate products serves as a linchpin, fostering an environment where devices, models, and overarching system dynamics are seamlessly interwoven. In essence, this interface encapsulates the foundational stratum wherein the strategic organization of products not only begets a framework for device integration but also lays the groundwork for subsequent model-product interlinkages.

The platform's user-centric design underscores the importance of rendering ostensibly complex operations in a user-friendly and comprehensible manner. The dedicated section for product management, as exemplified by the graphical user interface, encapsulates this ethos by affording users an accessible means to wield control over the initiation and configuration of diverse products, thereby catalyzing the subsequent harmonization of devices within the platform's overarching infrastructure.



id	name	code	version
1	Charging station	WH039KR	1.0.0

Figure 6.15: Product Information Page

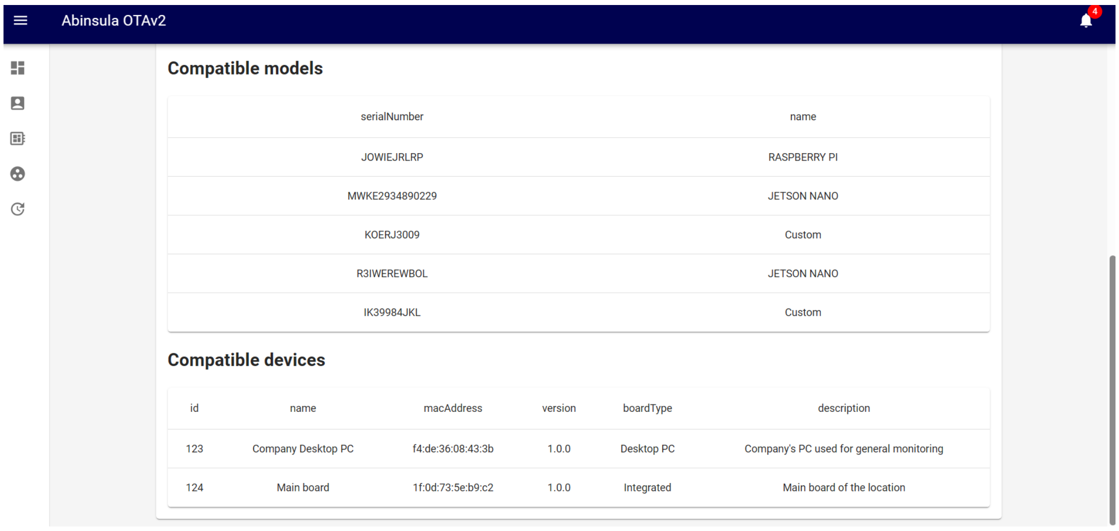
Upon selecting any element within the list, a wealth of additional insights becomes accessible, as elucidated in Figure 6.15. The standard array of charts is prominently displayed atop the interface. This suite of visual representations, meticulously crafted for optimal user comprehension, encompasses a diverse spectrum of critical metrics.

The initial chart furnishes a comprehensive breakdown of online devices stratified by models. This stratification serves to spotlight the top five models that boast the highest online device counts. Such nuanced visibility into the distribution of devices across distinct models offers a pivotal vantage point for assessing the comparative performance of different models within the product ecosystem.

Segueing to the subsequent chart, the interface offers a succinct portrayal of device version distribution. Specifically, it delineates the count of devices associated with a specific version. This insightful presentation discerns the top five versions that command the highest installation frequencies. This nuanced metric holds intrinsic value in deciphering the prevalence of specific software iterations across the device spectrum.

The final chart within this interface paradigm casts a spotlight on the volume of data amassed by devices, delineated by distinct model types. By presenting the top five models in terms of data collection, the interface bequeaths a consolidated overview of the data generation landscape. This delineation proves instrumental in gauging the efficacy of different models in contributing to the overarching data repository.

In essence, this interface amalgamates intricate datasets into visually accessible formats, empowering users to extract actionable insights effortlessly. The strategic placement of these charts within the product information interface aligns with the broader user-centric design philosophy of the platform. By affording users the ability to drill down into granular product-level analytics, the interface serves as a linchpin for strategic decision-making and performance optimization.



Compatible models

serialNumber	name
JOWIEJRLRP	RASPBERRY PI
MWKE2934890229	JETSON NANO
KOERJ3009	Custom
R3IWEREWBOL	JETSON NANO
IK39984JKL	Custom

Compatible devices

id	name	macAddress	version	boardType	description
123	Company Desktop PC	f4:de:36:08:43:3b	1.0.0	Desktop PC	Company's PC used for general monitoring
124	Main board	1f:0d:73:5e:b9:c2	1.0.0	Integrated	Main board of the location

Figure 6.16: Product, Model and Devices' Information

Finally, the Figure 6.16 encapsulates a holistic portrayal of a product's ecosystem. This comprehensive interface delivers pivotal insights into the product's foundational elements, interweaving crucial details about the product itself, its associated models, and the myriad devices tethered to these models.

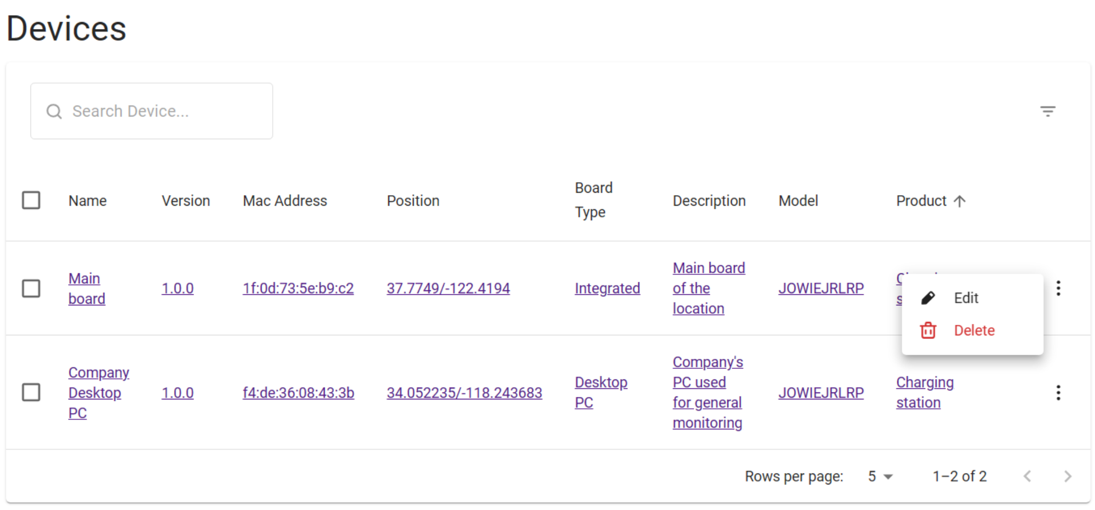
At the interface's zenith, a meticulous arrangement of charts meticulously unfolds. These visualizations, emblematic of the platform's commitment to user-centric data interpretation, crystallize intricate datasets into cogent presentations. The initial chart navigates the landscape of device distribution across associated models, spotlighting the top-performing models within the product ambit. This stratification serves as a linchpin for comprehending the distributional dynamics of devices within the broader product ecosystem.

Segueing to the subsequent layer of insights, the interface delineates the exhaustive spectrum of models associated with the product. Each model is presented with its salient information, furnishing users with a nuanced understanding of the compositional nuances within the product's repertoire. This strategic arrangement aids users in discerning the individual characteristics and performance attributes of each model, fostering informed decision-making.

The interface reaches its zenith by presenting a detailed inventory of devices linked to the diverse models associated with the product. This comprehensive device registry affords users an unobstructed view of the entire device landscape within the product domain. From here, users can seamlessly pivot to individual device profiles for deeper dives into device-specific metrics and functionalities.

In summation, this interface represents the culmination of the platform's design ethos — an amalgamation of user-friendly data visualizations and exhaustive datasets. It serves as an instrumental compass for users navigating the intricate nexus of products, models, and devices, empowering them with the clarity needed for strategic decision-making and system optimization.

This strategic presentation strategy augments the user experience, offering a consolidated overview that aids in informed decision-making and facilitates a nuanced understanding of the interconnected elements within the system. By encapsulating relevant data hierarchies, users can seamlessly traverse through the structural layers of products, models, and devices, gaining comprehensive insights at each level.



Name	Version	Mac Address	Position	Board Type	Description	Model	Product ↑
Main board	1.0.0	1f:0d:73:5e:b9:c2	37.7749/-122.4194	Integrated	Main board of the location	JOWIEJRLRP	[illegible]
Company Desktop PC	1.0.0	f4:de:36:08:43:3b	34.052235/-118.243683	Desktop PC	Company's PC used for general monitoring	JOWIEJRLRP	Charging station

Rows per page: 5 1–2 of 2

Figure 6.17: Tabular Data

Across the diverse tables within the platform's interfaces, a spectrum of versatile operations awaits users, designed to augment the accessibility and manipulability of data. One such operation is the search functionality, affording users the capability to pinpoint specific information by querying entries based on their names. This feature enhances the precision and efficiency of information retrieval within the expansive datasets encapsulated by the platform, as we can see in figure 6.17.

Furthermore, the capacity to orchestrate the display of tabular data is enriched through the implementation of sorting mechanisms. Users are empowered to arrange data entries in ascending or descending order based on various attributes, thereby facilitating a structured and customizable view of the information landscape.

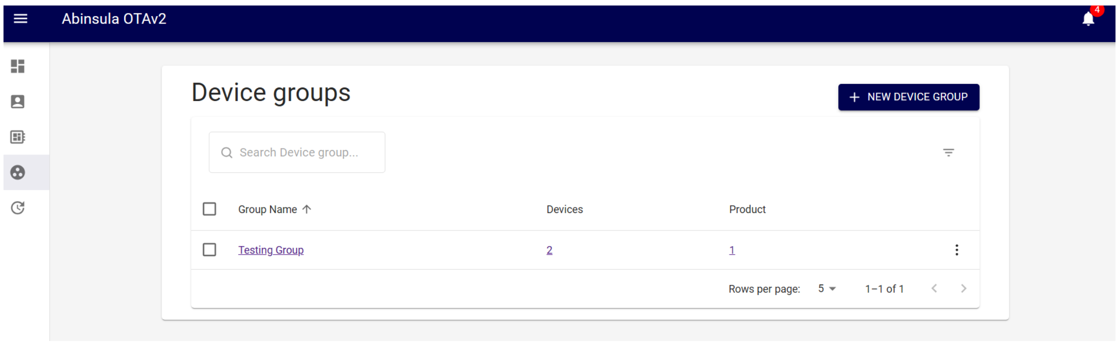
In pursuit of an interface that harmonizes with diverse user preferences, the platform introduces a pagination system. This feature allows users to tailor the number of displayed items per page according to their preference. While the default setting is configured to exhibit five entries, users can opt for increased density, with options to display 10 or 25 items per page. This adaptive design caters to varying user needs, balancing between a concise overview and in-depth perusal of data.

Furthermore, the platform augments user control by incorporating a concise menu accessible through three dots adjacent to each entry. This menu serves as a gateway to pivotal operations, notably enabling users to modify existing entries or effectuate their removal from the dataset. This intuitive mechanism streamlines the data management process, ensuring that users wield a seamless and efficient means to refine and curate the information repository.

In essence, these features collectively epitomize the platform's commitment to user-centricity, ensuring that the manipulation and interpretation of data are not only comprehensive but also tailored to individual user preferences and requirements.

6.1.5 Devices' groups

The Groups Page within the platform furnishes users with a strategic interface to oversee and manage distinct collections of devices, facilitating a nuanced organizational framework. Users, empowered by this feature, can create and designate various groups to serve specific operational purposes. These groups may be tailored to streamline the management of updates, categorizing devices based on deployment scenarios such as production or testing phases, among other customizable parameters.



Group Name ↑	Devices	Product
Testing Group	2	1

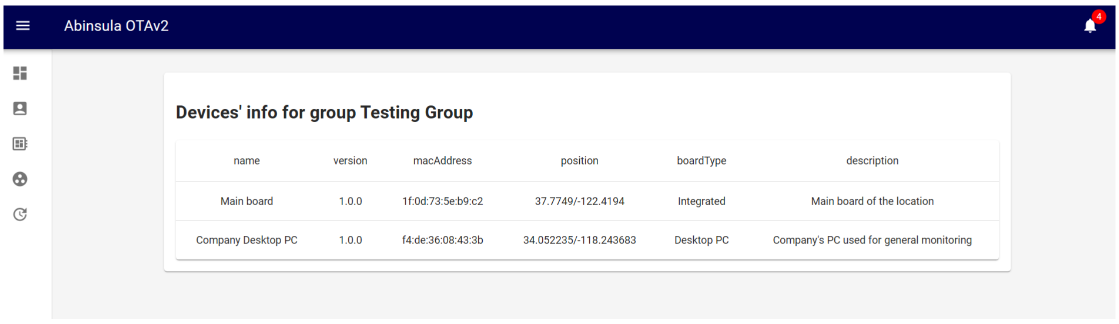
Figure 6.18: Group's page

Upon accessing the Groups Page, a user is presented with an overview of the diverse groups they have created, as we can see in Figure 6.18. This feature enhances the platform's adaptability to diverse use cases, allowing users to create bespoke organizational structures that align with their operational objectives. For instance, the creation of distinct groups for devices deployed in production, those designated for testing, or other customized classifications ensures a systematic approach to device management, as shown in Figure 6.20.

Each created group is interactive, affording users the capability to click on individual groups to access a more detailed view. This expanded view provides comprehensive insights into all devices contained within the selected group. Such granular information equips users with a holistic understanding of the devices affiliated with specific operational contexts, thereby supporting informed decision-making, as represented in Figure 6.19.

This strategic organization of devices into groups optimizes operational workflows, especially in scenarios where devices serve varied roles or undergo distinct phases within the overall operational lifecycle. The ability to customize groups based on specific criteria enhances the adaptability of the platform, catering to the dynamic and diverse needs of users.

The Groups Page offers users a sophisticated toolset for structured device management. This feature not only contributes to enhanced operational efficiency but also reflects the platform's commitment to providing users with a flexible and user-centric environment for navigating and managing their device ecosystem.



Devices' info for group Testing Group

name	version	macAddress	position	boardType	description
Main board	1.0.0	1f:0d:73:5e:b9:c2	37.7749/-122.4194	Integrated	Main board of the location
Company Desktop PC	1.0.0	f4:de:36:08:43:3b	34.052235/-118.243683	Desktop PC	Company's PC used for general monitoring

Figure 6.19: Group Info Page

The Groups Page facilitates the organized categorization and management of devices based on predefined criteria. Users are empowered to create new groups with a seamless and intuitive process, enhancing the platform's flexibility and adaptability to diverse organizational needs.

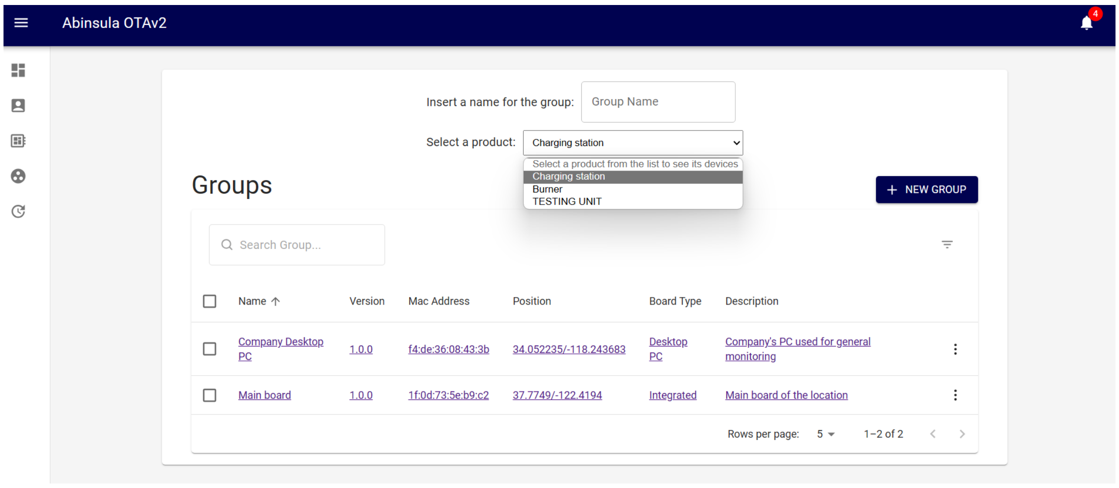
Upon accessing the Groups Page, users encounter a user-friendly interface designed for the creation and administration of device groups. The initiation of a new group is streamlined through an easily identifiable button, providing users with a straightforward starting point for the grouping process. This intentional design ensures accessibility and efficiency in the utilization of group creation features.

Within the group creation workflow, users navigate a systematic process that begins with the selection of models of interest. This step allows users to delineate the scope of the group by specifying the particular device models they intend to include. By adhering to a model-centric grouping strategy, the platform upholds safety protocols associated with updates and ensures compatibility within each device grouping.

Following model selection, users proceed to curate the group by handpicking individual devices from the available pool. This meticulous approach enables users to tailor each group composition according to specific operational requirements. Notably, the platform imposes constraints that prohibit the creation of groups comprising devices of mixed models. This deliberate limitation is grounded in safety considerations related to updates and broader compatibility concerns.

The platform's commitment to safety and precision is underscored by these design choices, as it prioritizes the creation of logically consistent and operationally secure device groups. This strategic approach aligns with industry best practices and underscores the platform's dedication to providing users with a robust and secure environment for managing their device ecosystem.

The Groups Page emerges as a feature-rich component, embodying the platform's commitment to user-centric design and operational integrity. By facilitating the creation of model-specific device groups, the platform ensures that users can confidently organize their devices in a manner that aligns with stringent safety standards and operational efficiency.



Insert a name for the group: Group Name

Select a product: Charging station

Select a product from the list to see its devices
Charging station
Burner
TESTING UNIT

Name	Version	Mac Address	Position	Board Type	Description
Company Desktop PC	1.0.0	f4:de:36:08:43:3b	34.052235/-118.243683	Desktop PC	Company's PC used for general monitoring
Main board	1.0.0	1f:0d:73:5e:b9:c2	37.7749/-122.4194	Integrated	Main board of the location

Rows per page: 5 1–2 of 2

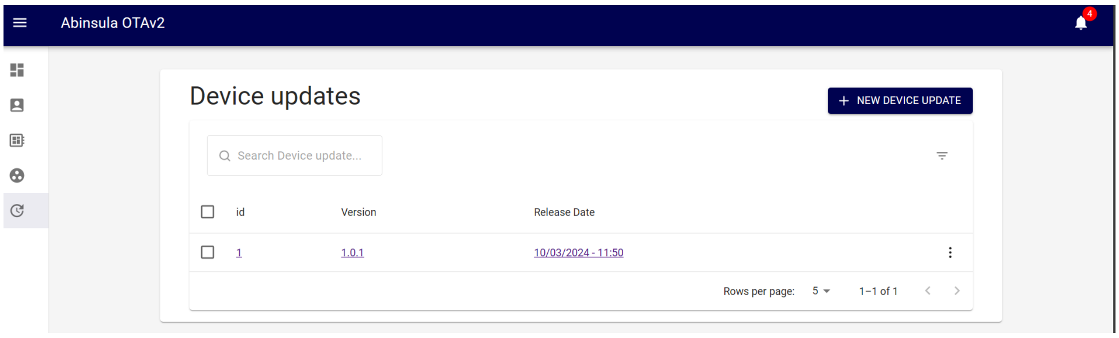
Figure 6.20: Group Creation Page

The Updates Section constitutes a crucial segment of the platform, serving as a centralized hub for overseeing and managing the update processes. Within this section, users gain access to pertinent information regarding existing updates, the ability to initiate the creation of new updates, and a comprehensive overview of update-related data.

Upon entry into the Updates Section, users are presented with a summary of ongoing and completed updates. This feature provides a bird's-eye view of the update landscape, offering insights into the status, versioning, and deployment particulars of each update. The clarity afforded by this overview ensures that users are well-informed about the update history, fostering an environment conducive to strategic decision-making.

6.1.6 Devices' updates

The platform extends functionality to create new updates, allowing users to seamlessly initiate and configure updates for their connected devices. This process involves specifying essential parameters such as version details, scheduled deployment times, and associated devices. The intuitive interface guides users through the update creation process, maintaining a user-friendly experience while ensuring precision in update management.



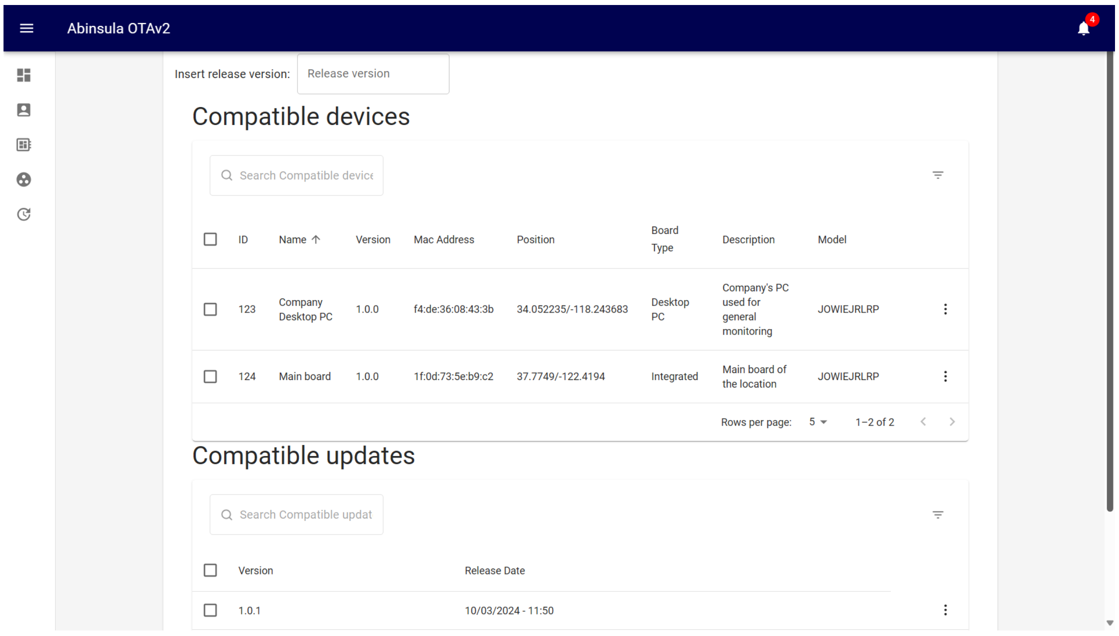
id	Version	Release Date
1	1.0.1	10/03/2024 - 11:50

Figure 6.21: Update List

Furthermore, the Updates Section offers an expansive array of information related to each update. Users can delve into specific updates to access detailed statistics, deployment timelines, and any associated success or error metrics. This granularity empowers users with a profound understanding of the impact and efficacy of individual updates, contributing to a robust feedback loop for continuous improvement.

Incorporating features such as sorting and filtering within the update information table enhances user functionality. Users can conveniently organize updates based on various parameters, facilitating efficient scrutiny of pertinent data. The platform's commitment to user-centric design is evident in these provisions, enabling users to tailor their interactions with update-related information to suit their specific requirements.

In summary, the Updates Section emerges as a pivotal element in the platform's architecture, designed to offer users a comprehensive suite of tools for managing and monitoring updates. Its multifaceted capabilities contribute to the platform's overarching goal of providing users with a sophisticated yet accessible environment for orchestrating seamless and effective updates across their device ecosystem.


Abinsula OTAv2

Insert release version: Release version

Compatible devices

Search Compatible device

ID	Name ↑	Version	Mac Address	Position	Board Type	Description	Model
123	Company Desktop PC	1.0.0	f4:de:36:08:43:3b	34.052235/-118.243683	Desktop PC	Company's PC used for general monitoring	JOWIEJRLRP
124	Main board	1.0.0	1f:0d:73:5e:b9:c2	37.7749/-122.4194	Integrated	Main board of the location	JOWIEJRLRP

Rows per page: 5 1–2 of 2

Compatible updates

Search Compatible updat

Version	Release Date
1.0.1	10/03/2024 - 11:50

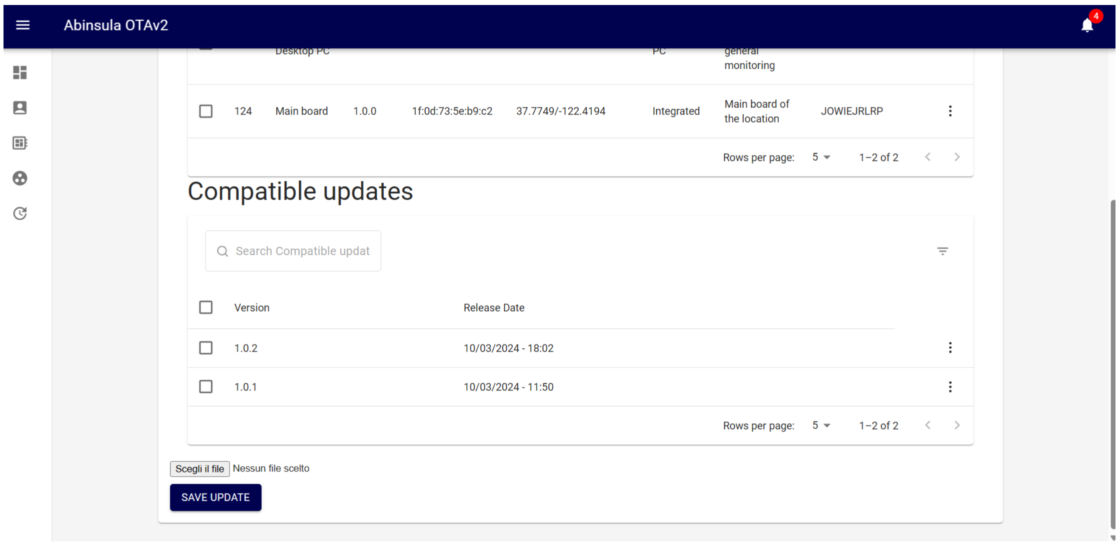
Figure 6.22: Update Creation Page

Within the platform's update management functionality, users are afforded a comprehensive suite of features when initiating the creation of a new update. This process is meticulously structured to encompass key parameters, ensuring precision and flexibility in the deployment of updates across the device ecosystem.

A fundamental aspect of the update creation process is the specification of the version for the impending update. This allows users to clearly define the iteration and progression of the software or firmware being deployed, thereby contributing to a systematic and traceable update history.

The platform's update management extends its functionality to include the explicit designation of compatible devices for the newly created update. Users can judiciously select the devices within their ecosystem that are slated to receive the update. This granular control over compatibility ensures that updates are strategically rolled out to devices, promoting a targeted and efficient update deployment strategy.

Furthermore, users have the capability to establish compatibility relationships with prior updates. This involves specifying from which previous update versions the current one can be seamlessly upgraded. This inter-update compatibility feature enhances the platform's versatility, accommodating diverse scenarios and update scenarios within the device environment.



	Desktop PC				PC	general monitoring		
124	Main board	1.0.0	1f:0d:73:5e:b9:c2	37.7749/-122.4194	Integrated	Main board of the location	JOWIEJRLRP	

Rows per page: 5 1–2 of 2

Compatible updates

Search Compatible updat

Version	Release Date
1.0.2	10/03/2024 - 18:02
1.0.1	10/03/2024 - 11:50

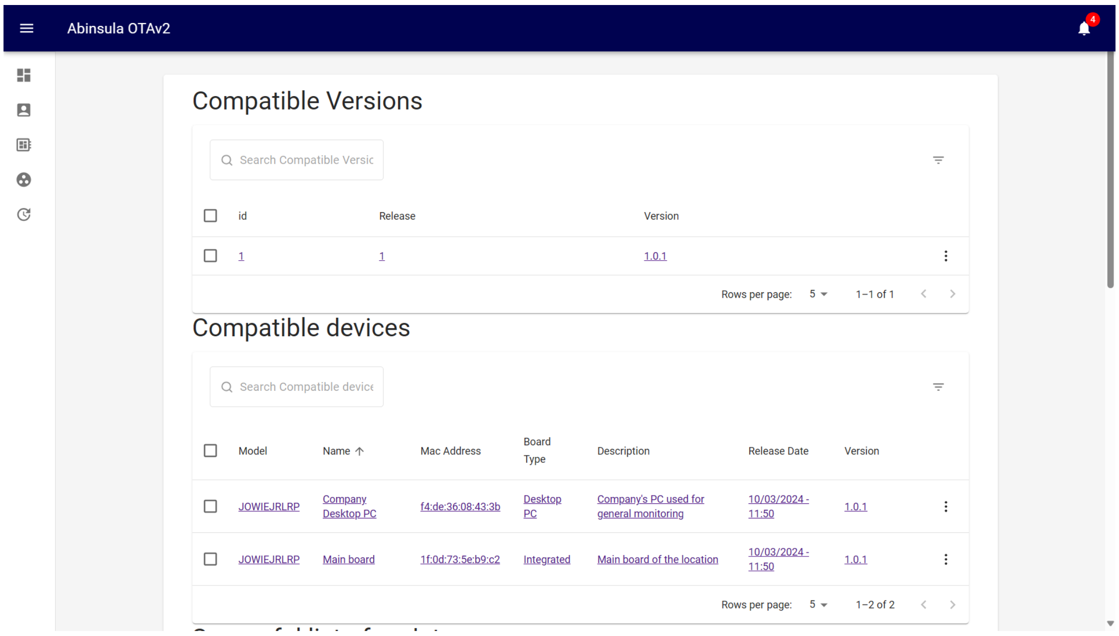
Rows per page: 5 1–2 of 2

Figure 6.23: Update Creation Page

A pivotal aspect of the update creation process is the selection of the update package. Users are provided with a streamlined interface to upload and attach the update package to the update creation workflow. This entails a meticulous selection process, ensuring that the correct and validated update package is associated with the corresponding update version.

Once these parameters are meticulously configured, users are presented with the option to save the update package. This critical step finalizes the update creation process, preserving the configured parameters and readying the update for subsequent deployment to the specified devices within the ecosystem.

In summary, the update creation process within the platform epitomizes a user-centric approach, providing administrators with a robust set of tools to orchestrate precise, versioned, and compatible updates across their device landscape. This feature aligns with industry best practices in software and firmware management, fostering an environment of controlled and secure updates within the organizational infrastructure.



Compatible Versions

Search Compatible Versic

id	Release	Version
1	1	1.0.1

Rows per page: 5 1–1 of 1

Compatible devices

Search Compatible device

Model	Name	Mac Address	Board Type	Description	Release Date	Version
JOWIEJRLRP	Company Desktop PC	f4:de:36:08:43:3b	Desktop PC	Company's PC used for general monitoring	10/03/2024 - 11:50	1.0.1
JOWIEJRLRP	Main board	1f:0d:73:5e:b9:c2	Integrated	Main board of the location	10/03/2024 - 11:50	1.0.1

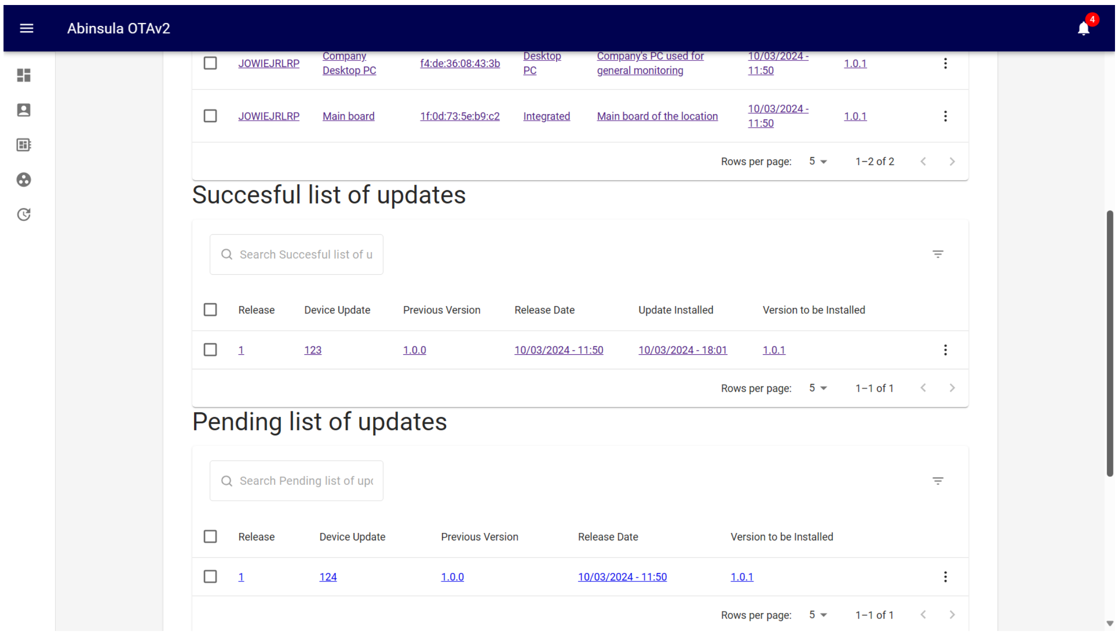
Rows per page: 5 1–2 of 2

Figure 6.24: Update Info Page

Upon accessing the detailed information page of a particular update within the platform, users are presented with a comprehensive array of data encapsulating various facets of the update's lifecycle. This informative display is strategically designed to furnish administrators with a holistic understanding of the update's scope and impact on the device ecosystem.

Key elements of the update information page include insights into the compatible versions. This feature provides clarity on the range of software or firmware versions with which the update harmoniously aligns. Such transparency is crucial for administrators to make informed decisions regarding versioning strategies and ensures compatibility adherence across the device landscape.

The update information page further delineates the list of compatible devices, offering a granular view of the devices earmarked to receive the update. This detailed enumeration aids administrators in conducting a precise assessment of the update's distribution across the device inventory, fostering strategic and targeted deployment.


Abinsula OTAv2

JOWIEJRLRP	Company Desktop PC	f4:de:36:08:43:3b	Desktop PC	Company's PC used for general monitoring	10/03/2024 - 11:50	1.0.1	⋮
JOWIEJRLRP	Main board	1f:0d:73:5e:b9:c2	Integrated	Main board of the location	10/03/2024 - 11:50	1.0.1	⋮

Rows per page: 5 1–2 of 2

Succesful list of updates

Search Succesful list of u

Release	Device Update	Previous Version	Release Date	Update Installed	Version to be Installed	
1	123	1.0.0	10/03/2024 - 11:50	10/03/2024 - 18:01	1.0.1	⋮

Rows per page: 5 1–1 of 1

Pending list of updates

Search Pending list of upc

Release	Device Update	Previous Version	Release Date	Version to be Installed	
1	124	1.0.0	10/03/2024 - 11:50	1.0.1	⋮

Rows per page: 5 1–1 of 1

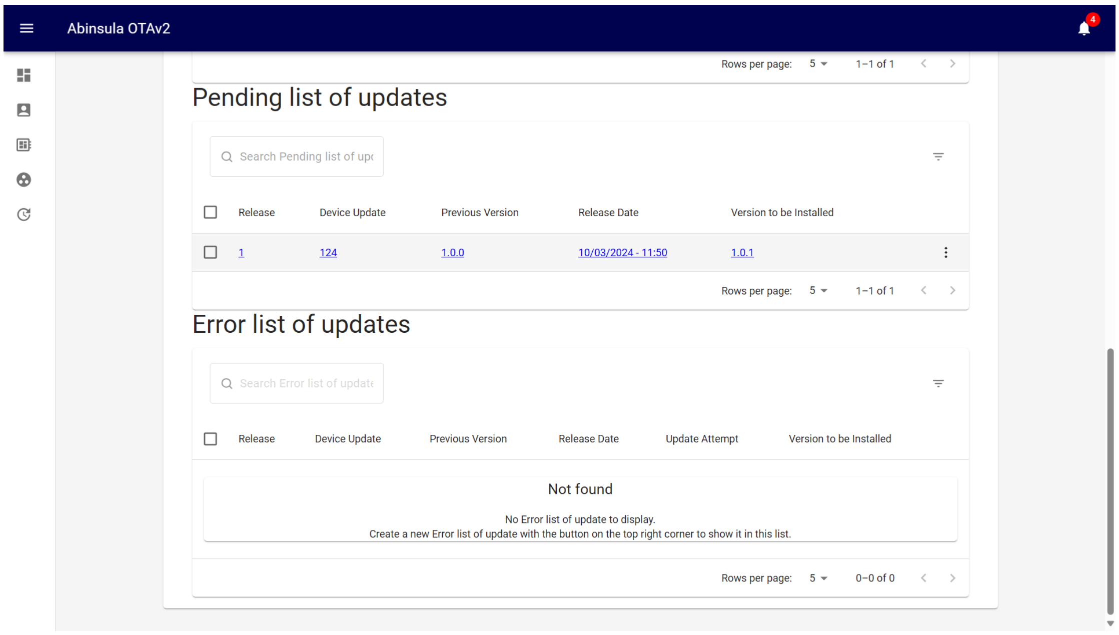
Figure 6.25: Update Info Page

A pivotal aspect of the update information page is the documentation of updates that have been successfully executed on devices. This historical record offers valuable insights into the performance and efficacy of past update deployments. Conversely, the page also provides visibility into updates that encountered issues during deployment, furnishing administrators with critical information for diagnostic and remedial purposes.

Additionally, the update information page meticulously catalogues devices awaiting the impending update in the pending list. This proactive visibility empowers administrators to anticipate forthcoming changes within the device ecosystem, facilitating strategic planning and resource allocation.

An intuitive feature embedded within the update information page is the seamless navigation to additional layers of information. For instance, users can expeditiously access detailed information about individual devices, compatible updates, and other pertinent data points directly from this centralized page.

In conclusion, the update information page stands as a robust and user-friendly interface, consolidating multifaceted data related to each update. This consolidated view not only streamlines administrative tasks but also fosters an environment of transparency and informed decision-making within the platform's update management framework.


Abinsula OTAv2

Rows per page: 5 1–1 of 1

Pending list of updates

Search Pending list of upc

Release	Device Update	Previous Version	Release Date	Version to be Installed
1	124	1.0.0	10/03/2024 - 11:50	1.0.1

Rows per page: 5 1–1 of 1

Error list of updates

Search Error list of update

Release	Device Update	Previous Version	Release Date	Update Attempt	Version to be Installed

Not found

No Error list of update to display.
Create a new Error list of update with the button on the top right corner to show it in this list.

Rows per page: 5 0–0 of 0

Figure 6.26: Update Info Page

The platform is meticulously crafted to present a user-friendly interface that effectively conceals the inherent complexity within the underlying project infrastructure. This strategic design philosophy aims to streamline and simplify the intricate communication processes between the two servers and the client, thereby affording users the ability to execute complex tasks with ease. It is noteworthy that this accessibility is extended to users irrespective of their proficiency levels, ensuring a seamless user experience even for individuals without advanced skills in the operational environment.

The emphasis on intuitive design principles becomes evident in the platform's architecture, which prioritizes clarity and simplicity. By abstracting the intricacies of the underlying project, the platform serves as a conduit for users to engage effortlessly with the system, obviating the need for specialized or advanced technical competencies. This deliberate simplification enhances usability and empowers users to navigate the platform confidently, undertaking tasks of considerable complexity without being encumbered by the technical nuances of the underlying infrastructure.

Moreover, the platform's user interface is strategically tailored to facilitate the execution of intricate operations through a straightforward and accessible framework. This design philosophy aligns with the overarching goal of democratizing access to the platform's functionalities, ensuring that users can perform sophisticated tasks without grappling with the complexities inherent in the communication protocols between servers.

By providing users with an environment where the complexity is elegantly abstracted, the platform aligns with contemporary standards of user-centric design. This approach not only enhances the overall user experience but also contributes to increased efficiency, as users can focus on the tasks at hand rather than navigating through convoluted technical intricacies.

In summation, the platform's design ethos centers on user empowerment through simplicity. This intentional strategy ensures that users, regardless of their technical acumen, can leverage the platform's capabilities efficiently, fostering a productive and inclusive user experience within the dynamic landscape of the project.

Chapter 7

Data Storage

The backbone of any robust system lies in its ability to efficiently handle, store, and manage information. This section outlines the comprehensive data storage strategy employed in the project, emphasizing the need for a meticulous approach to information organization and management. The server plays a pivotal role in managing a diverse range of information, including user requests, device authentication, telemetry, and system commands. Efficient data handling is critical to ensure seamless communication between the frontend, backend, and connected devices. The need to store data arises from various sources, such as user interactions with the frontend, device registrations, telemetry from connected devices, and system statistics. Effective data storage is imperative to maintain a record of user activities, device statuses, and facilitate data-driven decision-making.

Recognizing the diversity in the nature of information, the project adopts a hybrid approach utilizing both SQL and NoSQL databases. This ensures a balanced and efficient storage mechanism for both structured and unstructured data. Dividing the handling of data into two separate backend components facilitates streamlined processing. The REST API server manages requests from the frontend, including user-related actions and device information. Simultaneously, the TCP server specializes in handling real-time connections with individual devices, focusing on authentication, command execution, and data exchange.

During the requirement engineering phase, particular attention was given to designing a database schema that aligns with the project's long-term objectives. This step is important to maintain schema consistency, minimize modifications during deployment, and avoid data loss. The strategic use of SQL databases for structured data and NoSQL databases for unstructured data offers a synergy that provides the project with the best of both worlds. This approach balances the load of information storage and retrieval, ensuring optimal performance.

To further optimize data retrieval and reduce the load on databases, Redis is incorporated as an in-memory caching solution. It strategically stores critical information, ensuring swift responses during scenarios requiring frequent updates. While the deployment of multiple databases and caching mechanisms introduces complexity, the resulting gains in performance, scalability, and system robustness justify the architectural choice.

In conclusion, the project's data storage strategy is intricately designed to address the diverse nature of data structures while ensuring long-term stability, schema consistency, and efficient handling. By adopting a hybrid database approach, balancing SQL and NoSQL databases, and integrating Redis for intelligent caching, the system is poised to deliver optimal performance and scalability, meeting the demands of a dynamic and data-intensive environment.

7.1 SQL/NoSQL Integration

In the realm of modern software development, the choice between SQL (Structured Query Language) and NoSQL databases is often a critical decision influenced by diverse data management needs. This section outlines the strategic decision to leverage the strengths of both SQL and NoSQL databases within the project, elucidating the rationale, benefits, and the bifurcated backend architecture designed for optimal data handling.

SQL databases excel in managing structured data through predefined schemas, while NoSQL databases offer flexibility, particularly with unstructured or semi-structured data. Recognizing the complementary strengths of these databases, the project adopts a hybrid approach. By integrating SQL and NoSQL databases, the project aims to capitalize on the advantages of each paradigm. This amalgamation facilitates the storage of both structured and unstructured data, striking a balance between the rigidity of SQL and the flexibility of NoSQL.

The benefits of this Hybrid Database Architecture are:

- Load Balancing: Utilizing two databases distributes the load of data storage and retrieval, preventing bottlenecks and ensuring optimal performance. Each database is designated to handle specific types of data, contributing to efficient load balancing.
- Schema Simplification: The hybrid approach simplifies the complexity associated with database schema creation. SQL databases manage structured data, while NoSQL databases handle unstructured data, streamlining schema design and enhancing overall system maintainability.
- Backend Differentiation: To effectively manage data in the hybrid environment, distinct backends are implemented for SQL and NoSQL databases. This demarcation allows tailored handling of data, optimizing processing and retrieval operations for each database type.

Two separate backends are developed, each dedicated to interfacing with different database types. This clear separation ensures that the unique characteristics and requirements of SQL and NoSQL databases are accommodated with precision. Distinct libraries are employed for interfacing with SQL and NoSQL databases, acknowledging the disparate nature of these data storage mechanisms. This tailored approach optimizes data retrieval, storage, and processing for each database type. Data processing within the APIs' backend is customized to align with the specific needs of SQL and NoSQL databases. This involves nuanced handling of queries, transactions, and data manipulation to leverage the unique strengths of each database paradigm. While the hybrid database architecture introduces additional layers of complexity, the strategic advantages gained in terms of performance, scalability, and adaptability outweigh the inherent intricacies. The overhead introduced by managing two distinct databases is justified by the performance gains, improved readability, and scalability achieved through this hybrid approach.

In conclusion, the decision to adopt a hybrid database architecture reflects a nuanced understanding of the project's data management requirements. By harmonizing SQL and NoSQL databases, the project maximizes the benefits of both paradigms, ensuring efficient load balancing, simplified schema management, and optimized backend processes. The dual backend architecture, tailored database access libraries, and customized data processing in APIs demonstrate a meticulous approach to harnessing the strengths of SQL and NoSQL databases. While acknowledging the elevated complexity, the resultant advantages position the project for enhanced performance, adaptability, and sustained scalability.

Aspect	**SQL Databases**	**NoSQL Databases**
Data Structure	Structured	Flexible (JSON, BSON, etc.)
Schema	Fixed Schema	Dynamic Schema
Scalability	Vertical (Scaling Up)	Horizontal (Scaling Out)
Complex Queries	Excellent	Limited (Simple queries)
Transaction Support	ACID Properties	Limited (BASE properties)
Consistency	Strong	Eventual
Community Support	Strong	Varied
Use Cases	Complex Queries, Transactions	Flexible Schema, High Volume

Table 7.1: Comparison of SQL and NoSQL Databases

Strengths of both worlds:

- **SQL Databases:**

1. Well-suited for structured data like user information, device models, and update packages.
2. Excellent for complex queries and transactions, beneficial for handling structured information efficiently.

- **NoSQL Databases:**

  1. Suited for unstructured or semi-structured data, making it ideal for telemetry data, sensor information, etc.
  2. Scalability is a significant advantage, especially when dealing with a high volume of data from various devices.
  3. Flexible schema accommodates the dynamic nature of certain types of data.

7.2 SQL for Structured Information

This section delves into the rationale and approach behind leveraging a SQL database to store structured information in the project. The focus is on harnessing the efficiency and organization offered by SQL databases, with a particular emphasis on storing critical data related to users, devices, models, products, and device update packages.

SQL databases excel in structuring and organizing data efficiently. The tabular format and predefined schema allow for a systematic arrangement of information, which is crucial when dealing with well-defined and interrelated data entities. The relational model of SQL databases facilitates the establishment of relationships between different entities. This relational nature is advantageous when dealing with complex data structures, such as associating users with registered devices, device models, and product information.

User-related data, including user profiles, authentication details, and access permissions, are stored in the SQL database. This structured storage ensures easy retrieval and management of user-centric information. Details about registered devices, their specifications, and configurations find a home in the SQL database. The relational capabilities allow for linking devices to specific users, creating a comprehensive overview of device ownership. Information pertaining to the various models and products offered by the company is systematically stored. The SQL database structure supports the classification and categorization of products, aiding in efficient retrieval. Structured information about update packages for devices is a critical aspect of the SQL database. The ability to organize and relate update packages to specific devices is crucial for seamless update management.

The structured data entities stored in the SQL database can be seamlessly combined and queried to derive more complex and detailed information. This synergy enhances the analytical capabilities of the system. While introducing a level of complexity, the use of a SQL database for structured information is a trade-off for the immense advantages gained in terms of query efficiency, data integrity, and the ability to manage intricate relationships.

In conclusion, the decision to store structured information in a SQL database is rooted in the unparalleled advantages offered by the SQL model. By capitalizing on its organizational prowess, relationship management capabilities, and efficient data retrieval, the project ensures a robust foundation for handling critical data entities such as user information, device details, company products, and update packages. This approach enhances the system's ability to manage, analyze, and derive valuable insights from structured data.

Criteria	**MySQL**	**Microsoft SQL Server**	**PostgreSQL**
Open Source	Yes	No	Yes
License Cost	Free	Paid	Free
Community Support	Strong	Strong	Very Strong
Advanced Features	Limited	Extensive	Extensive (JSON, GIS)
Scalability	Good	Excellent	Excellent
Performance	Fast	Very Fast	Very Fast
ACID Compliance	Yes	Yes	Yes
Ease of Administration	Moderate	Moderate	Very Easy
Extensibility	Good	Limited	Excellent
Concurrency Control	Good	Excellent	Excellent
Suitability for Complex Queries	Good	Excellent	Excellent
Data Integrity	Good	Excellent	Excellent
Compatibility with GIS	Limited	Limited	Excellent (PostGIS)
Why Suitable for Project	Good balance	Strong enterprise	Best fit (advanced features, scalability, extensibility)

Table 7.2: Comparison of SQL Databases for the Project

A brief description of the different fields:

- Open Source: PostgreSQL and MySQL are open source, while Microsoft SQL Server is not.

- License Cost: PostgreSQL and MySQL are free, while Microsoft SQL Server is a paid solution.
- Community Support: PostgreSQL has a very strong and active community, ensuring good support.
- Advanced Features: PostgreSQL supports advanced features like JSON, GIS (Geographic Information System), making it versatile.
- Scalability: All three databases are scalable, but PostgreSQL excels in this aspect.
- Performance: All three databases perform well, with PostgreSQL and Microsoft SQL Server being very fast.
- ACID Compliance: All three databases adhere to ACID properties for ensuring data integrity.
- Ease of Administration: PostgreSQL is known for its ease of administration.
- Extensibility: PostgreSQL supports custom functions and types, making it highly extensible.
- Concurrency Control: PostgreSQL and Microsoft SQL Server excel in managing concurrent transactions.
- Suitability for Complex Queries: PostgreSQL is well-suited for handling complex queries.
- Data Integrity: All three databases provide strong data integrity.
- Compatibility with GIS: PostgreSQL stands out with its excellent support for GIS through the PostGIS extension.

7.3 Database Schema

The architectural underpinning of the SQL database is characterized by a deliberate simplicity and adaptability, engineered to facilitate seamless modifications to the database structure in response to evolving requirements. This foundational approach ensures that future alterations to the database schema are executed with optimal ease, minimizing data loss to the greatest extent possible. The schema, delineated into three principal domains, encompasses user-related facets, components associated with products, models, and devices, and elements pertaining to devices and updates.

Within the realm of user-related data, the schema encapsulates essential information germane to user profiles, authentication credentials, and authorization attributes. This segment serves as the bedrock for user management, authentication processes, and access control mechanisms, fostering a secure and personalized interaction between users and the platform.

The facet dedicated to products, models, and devices constitutes a primary component of the schema, orchestrating the relational dynamics between these entities. Product-centric data includes descriptors such as product names and overarching information, while models are intricately linked to products, representing a specialized categorization within the product taxonomy. Devices, forming the operational nodes of the system, possess a dedicated schema section capturing vital attributes and configurations. This structured approach provides a coherent representation of the hierarchical relationships between products, models, and devices.

The third facet of the schema is geared towards devices and updates, encapsulating data pertinent to device interactions and the lifecycle of updates. Device-specific details, including telemetry configurations and operational parameters, find a repository in this segment. Concurrently, the schema accommodates the nuances of update management, encompassing versioning, compatibility considerations, and the status of update installations. This section intricately choreographs the communication and synchronization between devices and the overarching update infrastructure.

In embracing such a tripartite schema, the database architecture not only ensures the efficient management of user-related data and hierarchical product-device relationships but also furnishes a robust foundation for overseeing the dynamic evolution of devices through systematic update management. This intentional design philosophy positions the database as a resilient and adaptable repository, primed for seamless scalability and evolution in tandem with the platform's functional evolution.

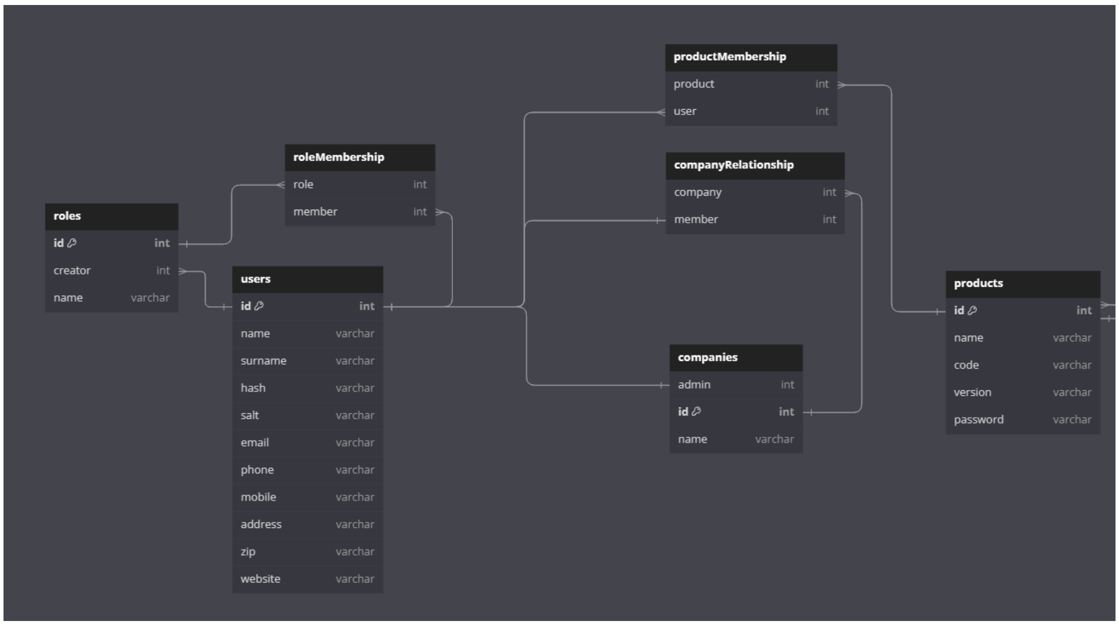


Figure 7.1: Database schema - User's related information

The foundational segment of the database schema pertains to user-related entities, as delineated in Figure 7.1. Within this domain, user information is systematically organized, featuring a role-based system that categorizes users into distinct hierarchical roles within the organizational structure. This role system serves as a mechanism for stratifying user privileges and responsibilities, ensuring a nuanced and controlled access environment.

A crucial component within this user-centric schema is the role assignment, where users are assigned specific roles indicative of their position and authority levels within the organizational hierarchy. This hierarchical arrangement contributes to the establishment of a well-defined user structure, promoting efficient role-based access control across the platform.

Complementing the role system, the inclusion of company specifications within the schema is notable. Each user is associated with a specific company, delineating their organizational affiliation. This affiliation is crucial in establishing the contextual framework for user roles, as the combination of role and company attributes affords a granular understanding of the user's rank and standing within the organizational landscape.

Moreover, the schema accommodates the linkage between products and companies, facilitating a seamless association between the products offered by a company and the users affiliated with that particular company. This linkage streamlines the delineation of product access and management responsibilities among users, reinforcing the cohesive integration of user roles, companies, and associated products.

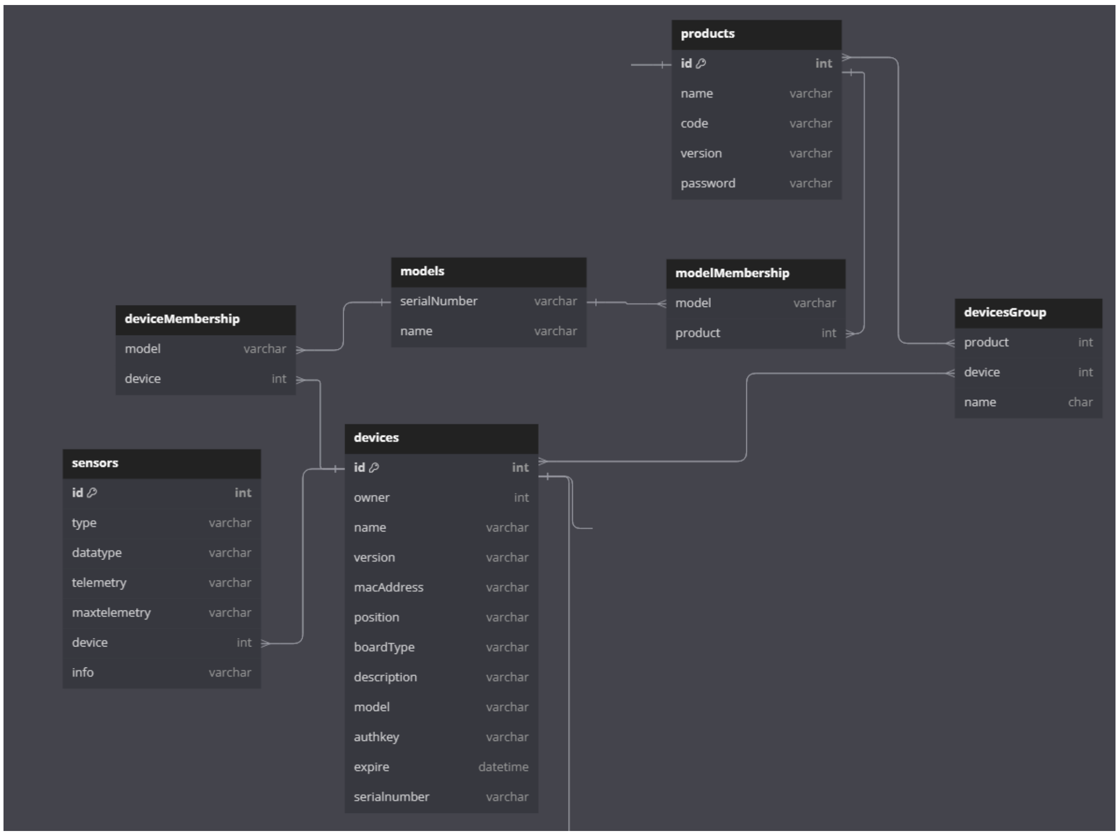


Figure 7.2: Database Schema - Product, Model and Device Related Information

The subsequent segment of the database schema is dedicated to the intricacies of products, models, and devices, as elucidated in Figure 7.2. This section encapsulates the hierarchical relationships and informational linkages that underpin the composition of devices and sensors within the platform.

At the core of this schema lies the concept of products, serving as foundational entities that encapsulate distinct offerings or solutions within the platform. These products, delineated by pertinent information, form a crucial link to subsequent layers of the schema, notably models. Models, as represented in the schema, encapsulate comprehensive information pertinent to their nature and function. This linkage between products and models establishes a structured framework for organizing and categorizing the diverse models associated with a particular product.

The crux of this section resides in the representation of devices. Devices, pivotal entities within the platform's ecosystem, harbor comprehensive information stored within their configuration files. The device-centric schema extends to include details about sensors connected to each device. These sensors, integral components of the platform's functionality, contribute to the diverse data collection capabilities of devices. Each sensor's relevant information, also stored in the device's configuration file, enriches the platform's understanding of the sensor landscape.

Furthermore, the schema incorporates the concept of device groups, providing a mechanism for aggregating and managing devices based on specified criteria. These groups facilitate streamlined device management, particularly in scenarios where operational coherence or specific functionalities demand collective attention to a subset of devices.

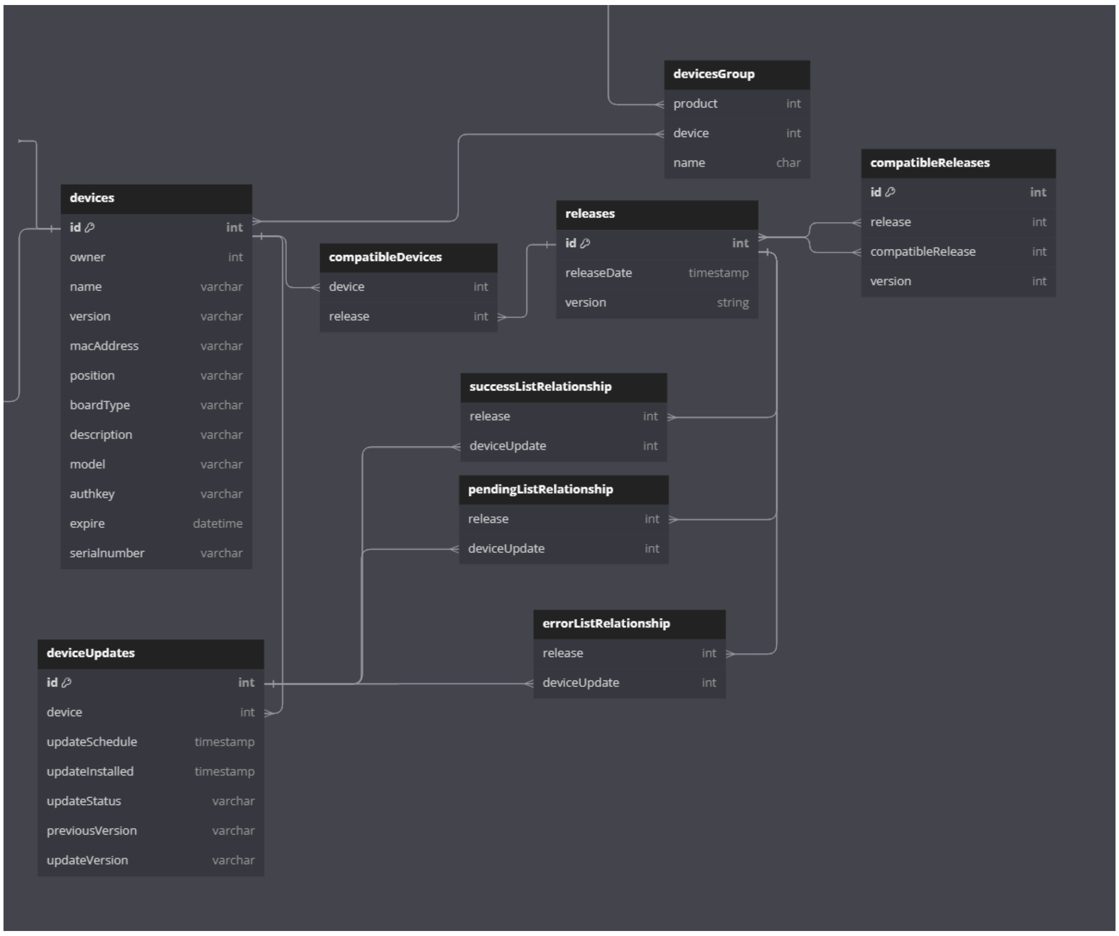


Figure 7.3: Database Schema - Devices and Updates Related Information

Concluding the database schema exploration, the final segment, illustrated in Figure 7.3, revolves around the integral connection between devices and updates within the platform's architecture. In this schema, devices are once again positioned at the epicenter, intricately linked with updates, thereby delineating the compatibility and evolution of the platform's firmware.

An important aspect of this schema is the establishment of a direct association between devices and software releases. This linkage serves to explicitly identify devices compatible with specific releases. Upon defining compatibility, a dedicated table, denoted as deviceUpdates, is instantiated to meticulously store pertinent information germane to the update process. This granular repository ensures a nuanced comprehension of the unfolding events during the update progression.

Moreover, the release creation process extends beyond device compatibility to encompass a delineation of compatible releases for a given device. This reciprocal linkage between devices and releases augments the platform's versatility in managing updates and accommodating diverse release scenarios.

The schema culminates in the manifestation of three distinct lists — success, pending, and error lists — derived from the information encapsulated within the deviceUpdates table. These lists provide a detailed breakdown of the update process, offering insights into the current status (success, pending, error) and supplementary details, including installation timestamps and related information. This meticulous categorization ensures an exhaustive and comprehensible representation of the dynamic update landscape.

In summary, The user-centric segment of the database schema not only orchestrates a sophisticated role-based access control system but also intricately weaves together user roles, company affiliations, and product linkages. This holistic design ensures a nuanced representation of the organizational structure, enabling effective user management, access control, and contextualized association with products offered by the respective company entities. The products, models, and devices-centric segment of the database schema establishes a hierarchical and interconnected framework. It seamlessly navigates from products to models and, ultimately, to devices, encompassing sensors and device groups along the way. This structured representation not only provides a comprehensive view of the platform's hardware landscape but also facilitates effective management and organization of devices, models, and associated sensors. The PostgreSQL database schema intricately captures also the interplay between devices and updates within the platform. By delineating compatibility, recording update processes, and categorizing outcomes, this schema provides a robust foundation for effective update management, contributing to the overall resilience and adaptability of the platform.

7.4 NoSQL for Unstructured Device Data

The choice of MongoDB as the NoSQL database for the project is underpinned by its inherent flexibility in handling diverse data types without the rigid structural constraints imposed by traditional SQL databases. This characteristic is particularly advantageous in managing data from devices with varying sensor configurations and data formats.

In MongoDB, data is organized into collections, providing a natural and scalable way to accommodate the dynamic nature of the information received from devices. Each registered device in the SQL database corresponds to a distinct collection in MongoDB. Within these collections, sensor data is stored with an additional field denoting the sensor's unique identifier. This design facilitates the representation of disparate sensor outputs within a single entry.

The unstructured nature of the data becomes evident as different sensors associated with a device can report distinct sets of parameters. For instance, one sensor may transmit temperature readings, while another might relay information on load, stress, and usage percentage in a unified dataset. The flexibility extends to the ability to introduce new sensors without necessitating alterations to the existing codebase. Additionally, modifications to the sensors, such as changes in the monitored parameters or the introduction of new sensors, can be seamlessly accommodated without causing disruptions to the system.

This adaptability ensures that the system can readily assimilate changes in the device ecosystem, fostering scalability and future-proofing against evolutions in sensor technologies.

A brief description of the different fields:

- Data Structure: Defines the fundamental organization of data in the database.
- Schema Flexibility: Indicates the degree to which the database accommodates changes in the data model.
- Scalability: Describes the approach to handling increasing amounts of data or traffic.
- Query Language: Specifies the language or method used for querying the database.
- Consistency Model: Defines how the database ensures consistency of data across nodes.
- Use Case Suitability: Highlights the types of applications or scenarios for which the database is well-suited.

Feature	**MongoDB**	**CouchDB**	**Cassandra**
Data Structure	Document-oriented	Document-oriented	Column-family
Schema Flexibility	Dynamic and flexible schema	Schema-free	Dynamic and flexible schema
Scalability	Horizontal scaling with sharding	Horizontal scaling	Linear scalability
Query Language	Rich and expressive queries	MapReduce queries	CQL (Cassandra Query Language)
Consistency Model	Strong consistency	Eventual consistency	Tunable consistency levels
Use Case Suitability	Versatile, suitable for diverse data types and applications	Document-oriented, suitable for applications with rapidly changing data	Wide-column store, suitable for time-series data and high-write workloads

Table 7.3: Comparison of NoSQL Databases

7.5 Redis for Caching Updates

In the pursuit of optimizing database performance and mitigating potential bottlenecks, the project incorporates Redis as an in-memory caching solution. This section elucidates the strategic utilization of Redis, detailing its role in alleviating the load on databases during intensive data requests and updates.

Redis serves as a dedicated in-memory cache to store critical information intelligently. The primary objective is to alleviate the strain on databases, particularly during scenarios involving mass updates or frequent retrieval of data. During instances where a bulk update of devices necessitates fetching the latest version of an update, Redis steps in to provide a rapid response. By caching frequently requested data, Redis significantly reduces the number of queries made to the SQL and NoSQL databases, optimizing response times and enhancing overall system performance. The strategic use of Redis ensures that high-impact, frequently accessed data is readily available in-memory. This allocation minimizes the strain on NoSQL databases, allowing them to focus on storing and retrieving other essential information, such as the plethora of sensor data generated by connected devices and retrieved by frontend clients. By caching pertinent information in Redis, the system achieves a delicate balance. It reduces the number of queries directed at the NoSQL database, preventing query overload and maintaining sufficient resources for handling the influx of sensor data—a crucial aspect considering the anticipated high volume of data generated by connected devices.

Being an in-memory database, Redis ensures lightning-fast data retrieval. This is particularly advantageous for scenarios where quick responses are imperative, such as fetching updates during mass device updates. Redis excels in resource efficiency by storing data in RAM. This minimizes the time spent on disk I/O operations, contributing to a more responsive and resource-optimized system.

In conclusion, the integration of Redis as an in-memory caching solution presents a strategic enhancement to the project's data management architecture. By judiciously caching frequently requested information, Redis effectively reduces the load on the databases during intensive data requests. The advantages, including accelerated response times, minimized query overhead, efficient resource allocation, and the overall resource efficiency of Redis, position it as a important additional component to ensure a high-performance and scalable system.

Chapter 8

Authentication and Device Management

The creation of the client software represented a pivotal phase in the project, as the client was intended to be pre-installed on the devices prior to deployment. Given the inherent challenges associated with updating or modifying the client post-deployment — a process that would necessitate device disconnection and potentially disrupt services — the imperative was to develop a robust and reliable client. This client was expected to operate seamlessly over extended durations without encountering significant issues.

This emphasis on robustness and reliability was driven by the need for the client to function continuously, minimizing the requirement for future interventions or updates that could potentially disrupt the normal operation of the deployed devices. The aim was to establish a client architecture that could withstand the rigors of long-term deployment and ensure sustained performance without compromising the integrity of the devices and the services they provide.

Key considerations in the client development included ensuring the ability to handle potential challenges, adapt to various operating conditions, and facilitate seamless operation without compromising the overall system's functionality. Additionally, the client needed to incorporate mechanisms for error handling, robust data transmission, and effective communication with the backend infrastructure.

In essence, the creation of the client software was a critical undertaking, requiring careful consideration of various factors to ensure its resilience, reliability, and ability to operate autonomously for extended periods, thereby contributing to the overall success and sustainability of the deployed system. An example of the interaction between the server and the client has been already shown in the Figure 5.2.

8.1 OS Diversity/Multiplatform Client (C)

The selection of the C programming language for the client software was driven by the diverse nature of the devices within the deployment environment, encompassing varying operating systems. Opting for C facilitated the creation of a compiled, executable program that could seamlessly run on devices across different platforms. This decision provided several advantages, including the ability to create a low-level client capable of executing on a diverse range of devices.

One of the primary benefits of employing C for the client software was the potential for cross-platform compatibility. By compiling the code into machine-level instructions specific to each device architecture, the resultant program could be executed on devices running different operating systems. This approach ensured a standardized client implementation, streamlining the deployment process across a heterogeneous device landscape.

Moreover, the use of a low-level language like C afforded the opportunity for fine-tuned optimization, particularly crucial when dealing with devices with limited computational resources. The inherent performance efficiency of C allowed for the implementation of resource-intensive operations at a low level, contributing to enhanced overall client performance. This was particularly advantageous for devices with constrained resources, where efficiency was paramount.

In summary, the adoption of the C programming language for the client software was a strategic decision emerged from the requirement engineering phase, rooted in the need for cross-platform compatibility and optimal performance across diverse devices. This choice aligns with the project's goal of developing a robust, adaptable client capable of functioning seamlessly in varied operational environments.

8.2 Client Authentication

Security is a fundamental concern within the project, given the multifaceted nature of potential threats, ranging from data manipulation to unauthorized access and cyber attacks. To fortify the platform against these threats, a comprehensive set of security measures has been meticulously devised.

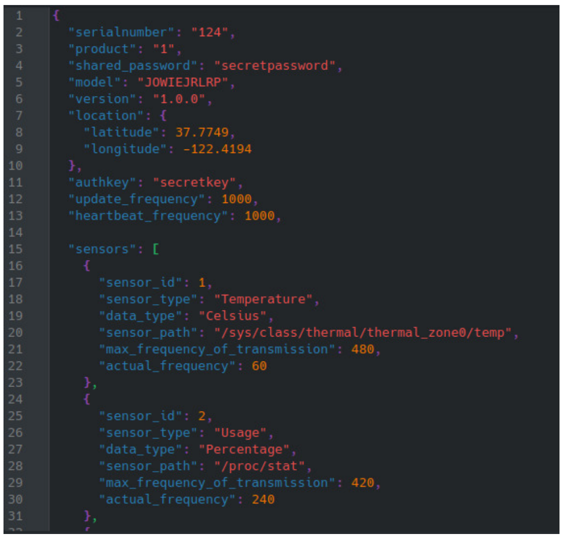
A pivotal step in enhancing security involves the implementation of robust authentication mechanisms for each connecting device. This authentication process is strategically designed to validate the identity of devices seeking connection to the server. By mandating device authentication, the system ensures that only legitimate and recognized devices are permitted to interact with the server. This stringent verification mechanism serves as a crucial deterrent against the introduction of counterfeit devices, preempting the transmission of erroneous data, potential cyber attacks, or any other malicious attempts aimed at disrupting the seamless operation of the entire platform.

The authentication protocol acts as a gatekeeper, effectively mitigating the risk of unauthorized access and reinforcing the integrity of the platform. This foundational security measure contributes significantly to the overall resilience of the system, creating a secure environment for the exchange of data and communication between authenticated devices and the server.

The initiation of the authentication phase commences with the systematic reading of the 'config.json' file. This configuration file, an integral component uploaded alongside the C client to all devices, encapsulates comprehensive information pertaining to the device. This information encompasses various key attributes essential for the device's interaction with the server.

Within the confines of the 'config.json' file, the device is endowed with a unique identifier, a product identifier, a model identifier, the transmission frequency for relaying sensor data to the server, heartbeat frequency to update the device's status, details about individual sensors, including their identifiers, sensor types, and the nature of the data they transmit. Additionally, an array housing ten distinct passwords, designated for authenticating communication with the server, is stipulated within the configuration. Furthermore, the provision is made for recording the device's spatial coordinates if such positional information is available.

This meticulous compilation of device-specific data within the 'config.json' file lays the groundwork for subsequent authentication procedures. The information encapsulated within this configuration file plays a pivotal role in orchestrating a secure and validated interaction between the device and the server during the authentication phase.



```json
{
  "serialnumber": "124",
  "product": "1",
  "shared_password": "secretpassword",
  "model": "JOWIEJRLRP",
  "version": "1.0.0",
  "location": {
    "latitude": 37.7749,
    "longitude": -122.4194
  },
  "authkey": "secretkey",
  "update_frequency": 1000,
  "heartbeat_frequency": 1000,

  "sensors": [
    {
      "sensor_id": 1,
      "sensor_type": "Temperature",
      "data_type": "Celsius",
      "sensor_path": "/sys/class/thermal/thermal_zone0/temp",
      "max_frequency_of_transmission": 480,
      "actual_frequency": 60
    },
    {
      "sensor_id": 2,
      "sensor_type": "Usage",
      "data_type": "Percentage",
      "sensor_path": "/proc/stat",
      "max_frequency_of_transmission": 420,
      "actual_frequency": 240
    },
```

Figure 8.1: Example of a possible config.json file.

Upon successful extraction of device-specific information from the ‘config.json‘ file, the client proceeds to establish a secure TCP connection with the server, utilizing the uploaded certificate in conjunction with the client and ‘config.json‘ files.

With a secure communication channel now in place, the client enters the preparatory phase before transmitting data across the network. In this preparatory step, the client encapsulates its data using the MessagePack format, resulting in a notable data size reduction of approximately 40%. This reduction is achieved through an encoding operation, converting the device-specific data into a binary representation. The binary data is primed for efficient transmission to the server.

Subsequently, the encoded, packed data is dispatched to the server, facilitating streamlined communication between the device and the server. This meticulous process, involving secure connections and optimized data packaging, forms a robust foundation for the secure and efficient exchange of information within the system.

Upon receipt of the client’s transmitted data, the server analyze the information sent by the client to verify its alignment with the records stored in its database. If the validation process confirms the validity of the client’s data and the device is successfully recognized, the server initiates a response sequence.

The server dispatches an acknowledgment signal (Return code 200 which implies ’Success’ plus a success field in the returned data structure) to the client, accompanied by a temporary key designated for secure communication between the client and the server. Additionally, the server may furnish optional information intended for updating the client’s configuration. This optional data transfer serves to synchronize any modifications made in the server’s database with the corresponding information on the client side. This meticulous synchronization ensures that both the server and the client possess congruent and up-to-date datasets, facilitating seamless and accurate interactions within the system.

Following the successful authentication of the device, substantiated by the confirmation of its authenticity as a genuine component within the system, the server facilitates communication by assigning a temporary key unique to that specific device.

With the allocated temporary key, the client device gains the ability to query the server for available updates. In the event of updates being identified, the client device initiates the update procedure. Conversely, if no updates are detected, the client device proceeds with the routine execution of its program and check again later for possible updates. Consequently, the authenticated device stands prepared to engage in various operations, including the transmission of sensor data to the server and the initiation of permissible actions such as update requests, reflecting the seamless and secure communication established between the client and the server.

8.3 Multipurpose Client Functionalities

Upon successful authentication, the client seamlessly transitions into a continuous operational loop, executing periodic interactions with the server. Each interaction leverages the pre-established secure channel, and data is meticulously packaged into message packs, adhering to the security and efficiency considerations previously elucidated.

Key among the client's functionalities is its robust handling of reconnection in the event of a lost connection with the server. This mechanism ensures the client's persistent connectivity, mitigating potential disruptions in its operation. Furthermore, the client incorporates a heartbeat functionality, implemented through a dedicated external thread. At regular intervals, typically every X minutes (configurable but set to 5 minutes by default), the client dispatches a heartbeat message to the server. This simple yet crucial message serves the purpose of notifying the server of the client's online status, connection integrity, and readiness to receive messages.

It is imperative to note that the heartbeat message is unidirectional, and the client does not await a response, while the server will just record the activity of the device and update its status. This design choice optimizes the efficiency of the process, as the primary objective is to provide the server with periodic updates on the client's operational status without introducing unnecessary delays.

A pivotal aspect of the client's functionality revolves around the management of updates. A dedicated thread, running approximately 1 to 2 times a day, diligently checks the REST API server for the availability of new updates. This process involves querying the server to ascertain if any updates are awaiting deployment. If updates are identified, the client proceeds to download the requisite update versions, prioritizing the acquisition of the latest available version. In the absence of updates, the thread will go into sleep and the client continues its routine operations seamlessly.

Notably, the client exhibits versatility in receiving updates through two distinct channels. The first involves the periodic querying of the REST API server, ensuring that the client stays abreast of the latest developments and security patches. In this scenario, the client autonomously initiates the update process upon detecting an available update.

The second channel involves the receipt of update commands directly via the TCP connection with the TCP server. In response to a custom command from the server signaling the availability of an update, the client promptly undertakes the download and installation process. This capability proves invaluable in scenarios demanding swift update deployment, especially in cases of security vulnerabilities requiring immediate patching.

Upon completion of the download, the client verifies the integrity of the update package using a checksum. If the checksum validation succeeds, indicating the integrity of the update, the client proceeds with the installation process. Following a successful installation, the client orchestrates an automatic restart of the device, ensuring the seamless application of the update.

In essence, the client's update management functionality encompasses automated periodic checks, autonomous update initiation based on REST API queries, and responsiveness to direct update commands via the TCP connection. This multifaceted approach ensures the timely and secure deployment of updates, contributing to the overall robustness and security of the deployed devices. The client's post-authentication phase, instead, involves continuous, secure communication with the server, incorporating reconnection strategies and periodic heartbeat messages to maintain a resilient and uninterrupted connection.

8.4 Sensor Data Transmission

Subsequent to the establishment of a secure connection and the successful completion of the authentication process, the client is poised to commence the periodic transmission of sensor data to the server. This involves sending data at intervals determined by the telemetry parameter configured in the client's associated configuration file or updated dynamically by the server at the user's request.

To facilitate this process, a dedicated thread operates periodically, reading data from the sensors. The sensor-generated data is automatically stored in the non-volatile memory of the device. The client accesses this information by reading corresponding JSON files, each file corresponding to a specific sensor and containing the data recorded by that sensor. Subsequently, the client compiles the data into a packed format using MessagePack and transmits it to the server. Given the inherent characteristics of the TCP protocol, any transmission issues, such as data loss or synchronization discrepancies, are addressed through automatic retransmission of the requisite packets until successful delivery is ensured. This mechanism guarantees the integrity and reliability of the transmitted sensor data.

Chapter 9

Real-Time Connection Maintenance

Upon successful connection to the server, a device is granted a limited number of authentication attempts. If authentication is successful, a persistent TCP connection is established, enabling direct communication between the device and the server. While the transmission of sensor data follows the route of the REST API server, the TCP connection remains inactive until a command from the frontend is directed to the device through the server. Maintaining an open connection is crucial to prevent disruptions and ensure the ability to reconnect with individual devices.

TCP is inherently capable of sustaining open connections for extended durations. However, to guarantee seamless functionality, a heartbeat mechanism is implemented. At regular intervals, typically every 5 minutes (configurable), the client sends a heartbeat message to the server. This proactive measure aims to uphold the integrity of the connection and serves as a confirmation of the device's ongoing activity. Upon receiving the heartbeat packet, the server updates the last activity date of the device, providing a timestamp to the frontend indicating the most recent activity and affirming the device's continued operational status.

9.1 Server-Side Separation for Active Devices

In the architectural design of the system, an important consideration is the server-side separation for active devices. This delineation involves a thoughtful partitioning of server responsibilities to efficiently handle various functionalities and data flows, particularly catering to the diverse needs of active devices within the network.

The server exhibits a bifurcated structure, with one facet devoted to the management of RESTful API requests. This segment is instrumental in servicing diverse frontend requests, including user authentication, device information display, device group creation, and more. Additionally, it plays a crucial role in managing device-related tasks, such as authenticating newly connected devices, facilitating initial configurations, and handling the exchange of sensor data. The decision to bifurcate these functionalities into a dedicated server component is strategic, aiming to streamline and optimize the handling of varied and concurrent requests. By keeping this server singularly focused on data and information management, potential pitfalls related to overlapping functionalities and overwhelming request loads are mitigated.

JavaScript, the language of choice for both segments of the server, is employed to maintain a unified approach. This design choice not only ensures a cohesive structure but is particularly advantageous for the component responsible for handling TCP connections.

The second facet of the server is entrusted with managing TCP connections with individual devices. Unlike the RESTful API server, this section maintains open connections with each device, orchestrating custom commands, exchanging ping commands for real-time device status checks, and handling device-specific instructions. This architectural decision is deliberate, aiming to segregate the handling of general data and information from the intricacies of device communication.

By isolating TCP connection management, the server can effectively discern and address the unique traffic patterns associated with device-specific communications. This tailored approach enhances the server's ability to handle diverse device requests efficiently. A single-threaded JavaScript execution is employed for TCP connections to facilitate streamlined request processing, avoiding potential overhead from context switching and ensuring a seamless experience even during high-demand scenarios.

The rationale behind this bifurcated server architecture is rooted in optimizing performance, maintaining code clarity, and proactively addressing potential scalability challenges. By allocating distinct servers for data-centric and device-specific operations, the system can intelligently distribute and prioritize tasks, providing a responsive and robust environment for both frontend interactions and device communications.

The separation of concerns also simplifies system monitoring, debugging, and maintenance. The distinct servers enable granular tracking of data traffic and device interactions, facilitating a more nuanced understanding of system behavior and performance.

In conclusion, the server-side separation for active devices reflects a meticulous design approach. It not only ensures optimal performance and responsiveness but also lays the groundwork for scalability and maintainability, essential attributes for a dynamic and evolving system landscape.

9.2 Information and Command Exchange

One of the primary aspects of the IoT platform is the seamless exchange of information and commands between the server and the client. This intricate process involves several phases, including the establishment of a secure channel, device authentication, and the routine transmission of data. Beyond the fundamental data flow, other critical components, such as heartbeat information and the execution of commands from the server, add layers of complexity and significance to the system.

1. Secure Channel Creation: The communication process initiates with the creation of a secure channel between the server and the client. This phase is integral to ensuring that data transmission occurs in a protected environment, safeguarding against unauthorized access and potential security breaches. The utilization of TLS (Transport Layer Security) combined with certificate-based authentication enhances the integrity of the communication channel, establishing a foundation for secure and encrypted data transfer.

2. Device Authentication: Upon the establishment of the secure channel, the next crucial step involves device authentication. The client undergoes a meticulous authentication process, presenting its credentials to the server. These credentials are typically stored in a configuration file, containing vital information about the device, including its ID, product ID, model ID, authentication passwords, and other relevant parameters. The server cross-verifies these credentials with its database, ensuring the legitimacy of the connecting device. Successful authentication results in the issuance of a temporary key, enabling further secure communication.

3. Routine Data Transmission: With the secure channel in place and device authentication completed, the client is poised to transmit data periodically to the server. This routine data transmission is orchestrated based on the telemetry parameters specified in the configuration file. A dedicated thread within the client periodically reads sensor data, encapsulates it using the MessagePack format for efficiency, and transmits it to the server. The reliability of the TCP protocol ensures that, in case of transmission issues, packets are retransmitted until successful delivery is confirmed.

4. Heartbeat Mechanism: To maintain the longevity of the connection and keep the server informed of the client's active status, a heartbeat mechanism comes into play. At regular intervals, the client sends a heartbeat message to the server, indicating its operational status. This heartbeat message not only serves as an indicator of device activity but also assists in updating the last activity date of the device in the server's records. This proactive

approach ensures that the server and client remain in sync regarding the device's operational status.

5. Command Execution: Beyond routine data transmission, the server has the capability to send commands to the client, initiating specific actions. These commands, ranging from software updates to custom operations like reboot or shutdown, demonstrate the bidirectional nature of the communication process. The client, upon receiving a command, interprets and executes it as instructed by the server, contributing to the dynamic and responsive nature of the entire IoT ecosystem.

In essence, the data exchange mechanism serves as the lifeblood of the IoT platform, orchestrating a symphony of secure, authenticated, and routine communications. Each component, from secure channel creation to command execution, plays a fundamental role in ensuring the efficiency, reliability, and security of the overarching IoT infrastructure.

Chapter 10

Remote Commands

The command execution feature is a robust implementation across the entire platform, offering users the capability to perform predefined operations on the connected devices. This functionality initiates in the frontend, providing users with an intuitive interface to interact with their devices comprehensively. Users can access a list of all connected devices, select specific devices, and execute various operations through a popup menu. Additionally, the system allows for the execution of custom commands, predefined actions designed to facilitate straightforward functions on the available devices.

- Frontend Interface: The user interface serves as the gateway for commanding devices. In the frontend, users can effortlessly navigate through a list of connected devices. A user-friendly popup menu facilitates the selection of different operations, empowering users to interact seamlessly with individual devices or groups of devices. This intuitive approach simplifies the user experience and enhances the accessibility of device management.

- Custom Commands: Custom commands, although predetermined and not user-writable, offer a spectrum of convenient functionalities. These commands are crafted in advance to provide users with a set of predefined operations that cater to common requirements. While it is not possible to write custom scripts commands, the predefined options encompass diverse actions, ensuring a versatile range of operations. These can include simple commands like device shutdown, reboot, or requesting the device to check for available updates.

- Command Execution Workflow: Upon user selection of a specific command, the server orchestrates the transmission of the command to the targeted device for execution. The command types are diverse, catering to various scenarios and operational needs. Examples range from elementary commands such as turning off a device to more intricate operations like initiating a device reboot or prompting the device to search for and install available updates.

- Versatility of Commands: The array of available commands contributes to the versatility of the platform. Users can seamlessly perform routine actions or respond to specific device requirements through a unified interface. This versatility aligns with the platform's commitment to user-centric control and ensures that users can dynamically engage with their devices to meet changing operational needs.
- Operational Impact: The execution of commands directly influences the operational state of devices. Whether it's for routine maintenance, troubleshooting, or responding to specific requirements, users can promptly initiate actions from the frontend. This instantaneous responsiveness enhances the user experience and facilitates efficient device management.

In essence, the command execution feature augments the user's ability to exert control and influence over connected devices. By providing a range of predefined operations and a user-friendly interface, the platform ensures that users can interact seamlessly with their devices, contributing to a robust and responsive IoT ecosystem.

10.1 Command Transmission from Server

The execution of commands within the platform is a sophisticated yet streamlined process, designed to seamlessly transition from user interaction on the frontend to the actual execution of commands on connected devices, abstracting any complexity present in the operations to be executed between the different components involved. In this detailed exploration, we delve into the intricate interplay between the REST API server and the TCP server, elucidating each step of the command execution flow.

- User Interaction and Command Selection: The process begins with user interaction on the frontend, where a diverse array of commands is made available. Users can effortlessly select commands from a popup menu associated with each device or even execute custom commands predefined by the system. This user-friendly interface serves as the entry point for initiating various operations on connected devices.
- Generation of REST API Call: Upon the user's command selection, a REST API call is dynamically generated. This call encapsulates a wealth of information, including specifics about the command to be executed, the target device, and any additional parameters essential for the successful execution of the command.

- Transmission to REST API Server: The REST API server, acting as the central orchestrator, receives the user-initiated REST API call. This server is instrumental in facilitating communication between the frontend and the underlying infrastructure. It processes the received command-related information and prepares it for transmission to the TCP server.

- Efficient Communication Between REST API and TCP Servers: A robust communication channel, established between the REST API server and the TCP server, ensures efficient data transmission. This direct linkage guarantees that the command-related information swiftly reaches the TCP server, minimizing latency and maximizing reliability.

- TCP Server's Role in Device Identification: The TCP server, upon receipt of the command-related information, undertakes the crucial task of identifying the target device within its list of actively connected devices. In instances where the device is not found, indicating a lack of current connectivity, the TCP server generates an error response. However, when the device is identified successfully, it is primed to receive the impending command.

- Secure Command Transmission to Device: With the target device identified, the TCP server initiates the secure transmission of the command to the associated TCP connection linked to that specific device. This secure communication pathway ensures the confidentiality and integrity of the command during transit.

- Device Execution and Potential Response: The identified device, having received the command, commences its execution. If the execution necessitates an acknowledgment or response from the device, the TCP server patiently awaits the device's reply. This meticulous waiting period ensures that the server remains synchronized with the real-time status of the device.

- Transmission of Device Response to REST API Server: Upon receiving the device's response or acknowledgment, the TCP server relays this vital information back to the REST API server. This bidirectional communication guarantees a cohesive and synchronous exchange of information, maintaining the integrity of the entire system.

- Device Status Update on the REST API Server: The REST API server, now armed with the response from the device, orchestrates the update of the device's status. This update encompasses the outcome of the command execution, whether successful or encountering an error. The device's status is promptly adjusted to reflect the most recent state.

This comprehensive and systematic flow ensures not only the accurate conveyance of user-initiated commands to respective devices but also provides real-time updates on the execution outcome. The seamless integration between the REST API server and the TCP server establishes a responsive and reliable command execution mechanism within the platform, fostering a user experience characterized by efficiency and transparency.

10.2 Execution on Receiving End

The process of command execution is a pivotal aspect of the platform's functionality, ensuring that user-initiated commands are seamlessly carried out on connected devices. This section outlines the intricate workflow involved in executing commands and managing device responses.

- Command Reception and Device State Evaluation: Upon receiving a command from the TCP server, the device assesses its current state. Critical operations, such as installing updates or transmitting data to the server, may be in progress. In such cases, the device intelligently waits for these activities to conclude before proceeding with the execution of the command. This careful evaluation prevents potential conflicts and ensures the smooth execution of commands.

- Command Execution Logic: The execution of commands is orchestrated using the C programming constructs. The system() function call becomes instrumental in carrying out the requested command on the device. This simplicity in execution enhances the device's ability to swiftly respond to diverse commands issued by the server.

- Handling Commands Requiring Device Shutdown: For commands necessitating a device shutdown, the server dispatches the command without waiting for an immediate acknowledgment. The device, upon receiving such a command, gracefully shuts down and subsequently reconnects to the server upon restarting. This asynchronous handling of shutdown commands ensures a streamlined process without unnecessary delays.

- Acknowledgment and Error Handling: Upon completion of a command, the device sends the command execution results. In case no results are expected from the command's execution, an acknowledgment (sort of ack) message is sent back to the server. This acknowledgment serves as confirmation that the command was executed successfully. The server awaits this response and, based on its receipt, adjusts the device's status accordingly. In the event of an issue during command execution, the server promptly updates the device's status to reflect the encountered error.

- Real-time Status Updates: The platform emphasizes real-time communication, and this extends to the immediate updating of device statuses. Whether it's a routine command execution or the resolution of an encountered error, the server maintains an accurate and up-to-date record of each connected device's status. This real-time feedback loop contributes to the overall responsiveness and reliability of the platform.

- Ensuring Command Integrity: To guarantee the integrity of command execution, the platform relies on robust communication protocols. The TCP nature of the connection ensures reliable data transmission, and the waiting mechanism for acknowledgments enhances the overall reliability of command execution.

In essence, the command execution workflow is designed to be dynamic and adaptive, allowing also the addition of new custom commands in the future. It intelligently manages ongoing device operations, executes commands efficiently, and maintains real-time synchronization between devices and the server. This meticulous approach contributes to a responsive and dependable ecosystem, empowering users with effective control over their connected devices.

Chapter 11

Performance Evaluation

To validate the robustness and scalability of the entire architecture, a comprehensive series of stress tests was conducted. The primary objective was to assess the system's performance under various loads and ensure its capability to effectively manage numerous concurrent connections. These stress tests involved subjecting the platform to increasing levels of load to systematically evaluate its responsiveness and stability.

The methodology employed in these stress tests encompassed a spectrum of scenarios, by simulating realistic usage patterns and progressively increasing the load, the tests aimed to identify potential bottlenecks, gauge the system's capacity, and ascertain its overall reliability.

The stress tests were designed to emulate conditions that the platform might encounter in real-world scenarios, considering factors such as the number of simultaneous user connections, data processing intensity, and the frequency of device interactions. This comprehensive approach provided a holistic perspective on the platform's performance metrics and its ability to sustain optimal functionality under diverse operational conditions.

Furthermore, the stress testing phase included the monitoring and analysis of key performance indicators, such as response times and resource utilization. This examination allowed for the identification of potential weaknesses or areas for improvement within the architecture.

The outcomes of these stress tests not only verified the platform's adherence to performance expectations but also facilitated the refinement of the system by addressing any identified limitations. This iterative process of testing and enhancement ensured that the architecture could reliably and efficiently support the anticipated workload, laying a solid foundation for its deployment in real-world scenarios with confidence in its scalability and responsiveness.

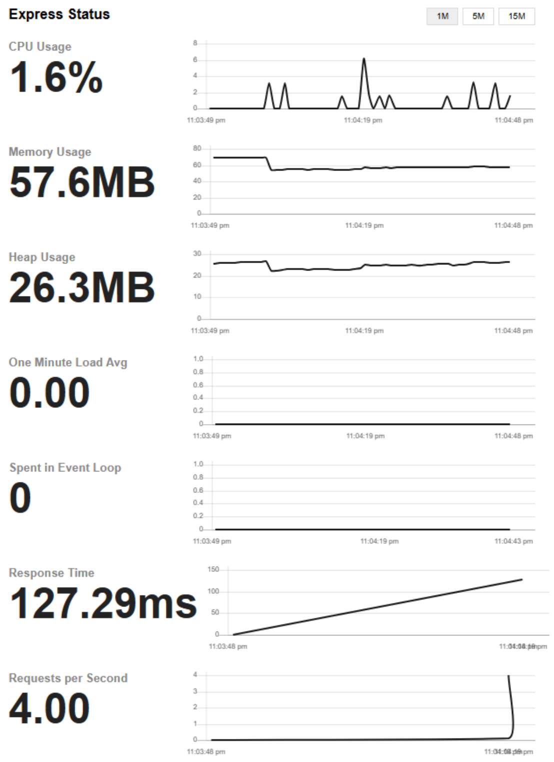


Figure 11.1: Load with only one device sending sensor data

As depicted in Figure 11.1, a meticulous examination of resource utilization during the initial phase of testing reveals commendably low resource consumption. Notably, this observation is underscored by the modest response times recorded, with any discernible delays primarily attributed to external connectivity issues. The results illustrate that, under minimal load conditions, the architecture exhibits an impressive capacity to handle interactions, as evidenced by the nominal strain on system resources.

Particularly noteworthy is the analysis of resource usage with the engagement of a solitary device communicating with the server. This scenario showcases the architecture's adeptness in efficiently managing a connection, as evidenced by the notably low levels of resource utilization. The system demonstrates resilience and proficiency even in this rudimentary testing scenario, setting a promising foundation for more comprehensive evaluations.

This initial phase of testing lays the groundwork for subsequent assessments under increased load scenarios, where the architecture's robustness and scalability will be systematically scrutinized. The forthcoming sections of the stress testing regimen will provide more nuanced insights into the platform's performance characteristics and offer a basis for continuous refinement and optimization.

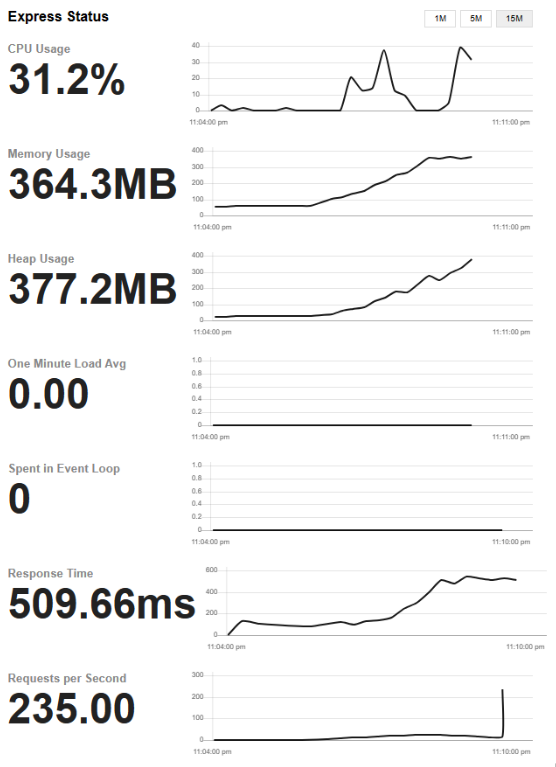


Figure 11.2: Load with one hundred devices sending sensors data

Elevating the operational load to accommodate 100 devices introduces a scenario analogous to the one illustrated in Figure 11.2. In this context, a discernible augmentation in both resource utilization and response times is observed, albeit not of a drastic magnitude. It is imperative to acknowledge the intrinsic challenge posed by the simultaneous communication of 100 devices with the server. Despite this heightened demand, the overall resources utilized remain within acceptable limits, particularly when considering the hardware configuration employed for the testing procedures.

The nuanced increase in resource consumption and response times under this elevated load underscores the architecture's ability to manage a substantial number of concurrent connections without a precipitous decline in performance. This observation is indicative of the platform's commendable scalability and resilience in handling a considerably larger device cohort.

A critical aspect to note is the ongoing optimization efforts and refinements being applied to further enhance the platform's efficiency under varying loads. The identified resource utilization metrics, while acceptable, prompt a commitment to continuous improvement to fortify the platform's performance parameters.

These subsequent phases will contribute to a comprehensive understanding of the platform's scalability, identifying potential areas for refinement, and reinforcing its robustness in accommodating diverse operational demands.

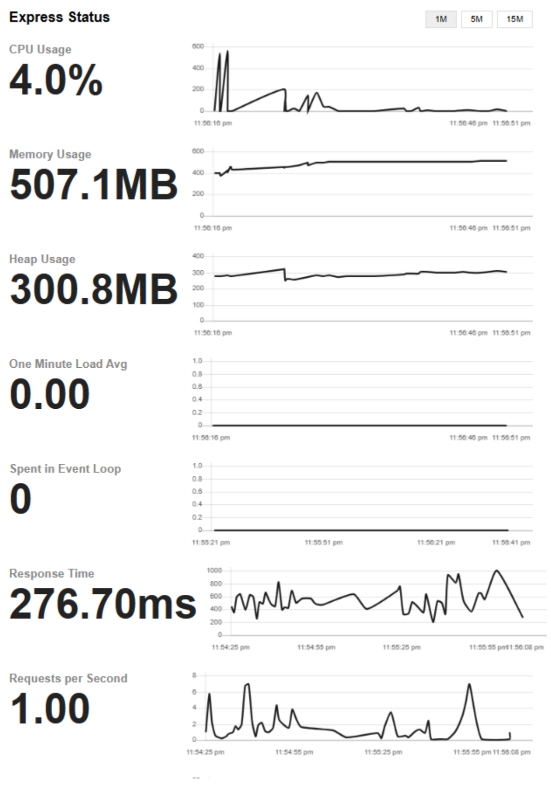


Figure 11.3: Load with five hundred devices sending sensors data

Subsequently, with a further augmentation in the device count to 500, the ensuing scenario aligns with the depiction in Figure 11.3. Evidently, both the response time and the volume of requests per second exhibit fluctuations, contingent upon the synchronization of devices transmitting data concurrently. Notably, the resource utilization demonstrates a subtle escalation compared to the precedent case involving 100 devices, as illustrated in Figure 11.2.

It is discerned from the presented scenario that as the number of devices is increased, even markedly, there is a proportional increment in resource utilization. This incremental augmentation, however, remains within reasonable limits, not even doubling in size, with five times more devices. Importantly, the platform's inherent capability to effectively manage a substantial influx of devices concurrently is manifest. The observed fluctuations in response time and request throughput underscore the dynamic nature of the platform's operation, adapting to the intricacies of simultaneous data transmission from a considerable device cohort.

This assessment emphasizes the platform's commendable scalability, ensuring efficient operation and responsiveness even under the heightened stress of 500 devices. It is pivotal to acknowledge the nuanced dependencies on device synchronization and the resultant impact on response metrics, which is inherent to scenarios involving a large number of concurrently communicating devices.

As part of the ongoing evaluation, further refinements will be considered to mitigate the observed fluctuations, enhancing the platform's resilience and fortifying its capacity to seamlessly accommodate the demands imposed by a significantly increased device load. Subsequent phases of stress testing will delve into more intricate scenarios, providing a comprehensive evaluation of the platform's performance across diverse operational scenarios.

Chapter 12

Conclusion

As we draw the curtain on the exploration and documentation of this comprehensive IoT project, it's vital to reflect on the journey, achievements, and the path forward.

- Project Recap: The inception of this project aimed to create a robust and scalable IoT ecosystem, seamlessly connecting devices, servers, and end-users. Through particular attention into planning and strategic implementation, each phase unfolded with a commitment to quality, security, and efficiency.
- Achievements: The project has achieved significant milestones, including the successful deployment of backend services, the development of a feature-rich frontend, and the integration of multiple databases to handle diverse data types efficiently. The implementation of secure communication protocols, client authentication mechanisms, and the ability to handle frequent updates showcases the project's versatility.
- Infrastructure and Architecture: The chosen infrastructure, consisting of Docker containers for databases and services, demonstrates a commitment to scalability and ease of deployment. The separation of backend and frontend components, along with the implementation of custom made graphs for statistics, ensures a well-organized and maintainable system.
- Database Strategies: The strategic use of both SQL and NoSQL databases leverages the strengths of each for structured and unstructured data, enhancing overall performance, scalability, and data management. The incorporation of Redis as an in-memory cache further optimizes data retrieval processes.
- Security Measures: The project places particular emphasis on security, implementing measures such as device authentication, secure communication channels, and periodic heartbeat checks. This proactive approach mitigates potential threats and ensures the integrity of the entire IoT ecosystem.

- Client-Side Development: The client-side development, crafted in the C programming language, exemplifies a commitment to efficiency and versatility. The client's ability to handle periodic data transmissions, manage updates seamlessly, and maintain a persistent connection underscores its robustness.
- User Interface and Experience: The frontend, developed with React, not only provides essential functionalities like device registration and statistics display but also ensures a responsive and user-friendly experience. The transition from a multi-page to a single-page application enhances overall responsiveness.
- Future Directions: Looking ahead, the project is poised for continued growth and innovation. The proposed future directions encompass machine learning integration, enhanced frontend features, and the exploration of emerging technologies such as edge computing and blockchain for heightened performance and security.
- Final Thoughts: In conclusion, this IoT project stands as a testament to meticulous planning, technical prowess, and a forward-thinking approach. Its impact extends beyond the codebase, reaching into the realms of scalability, security, and user experience. As we navigate the future, the project is well-equipped to embrace emerging technologies, meet evolving user needs, and continue its journey as a beacon of innovation in the IoT domain.

12.1 Achievements

Over the course of the project development, several key milestones and accomplishments have been achieved, contributing to the robustness, efficiency, and security of the overall system.

- REST API Management Implementation: The integration of Express, Morgan, Cors, and Passport libraries for handling REST API requests has been a pivotal achievement. This set of libraries facilitates seamless communication with the server, allowing efficient processing of requests and delivering appropriate responses. The use of Passport for user authentication enhances security, mitigating unauthorized access risks.
- Database Management with Knex and MongoDB: The utilization of Knex for SQL database interactions and MongoDB for NoSQL database operations is a significant achievement. Knex ensures the SQL database's integrity by handling table creation and data processing efficiently. Meanwhile, MongoDB caters to the flexibility required for handling diverse data structures, crucial for accommodating variable sensor data from different devices.

- TCP Server Implementation for Device Communication: The development of the TCP server component stands out as a critical achievement. Leveraging TLS, FS, and MessagePack-Lite libraries ensures secure communication channels with individual devices. The server's ability to log device IP addresses, decode incoming data, and perform various operations based on communication types (e.g., authentication, heartbeat, custom commands) demonstrates a comprehensive and tailored approach to managing device-server interactions.
- Documentation Strategy: A strong emphasis on comprehensive documentation has been instrumental in ensuring project maintainability. Each project component has dedicated documentation, detailing functionalities, requirements, and interactions. This approach enhances transparency, facilitates analysis, and simplifies future improvements.
- Scalable Frontend with React: The decision to migrate from a static multi-page application to a dynamic single-page application using React represents a noteworthy achievement. This transition not only enhances user experience but also lays the foundation for a more responsive and scalable frontend.
- Effective Use of Docker for Service Deployment: Deploying databases, backend, frontend, and statistics framework in separate Docker containers is a strategic achievement. Docker's containerization offers a streamlined, configurable, and easily deployable solution, enhancing project scalability and maintainability.
- Hybrid SQL and NoSQL Data Storage Strategy: The thoughtful integration of both SQL and NoSQL databases showcases a nuanced approach to data storage. Structured data finds its place in the SQL database, while unstructured sensor data benefits from the flexibility of MongoDB. This hybrid approach optimizes performance, readability, and scalability.
- Efficient Data Caching with Redis: The decision to implement Redis for data caching is an achievement that addresses the challenge of frequent data requests. By caching crucial information, Redis reduces the load on databases during mass updates, ensuring faster response times and improved overall system performance.
- Client Development in C for Cross-Platform Compatibility: Choosing the C programming language for client development is a notable achievement, considering the diverse nature of devices running different operating systems. This decision allows for compiled, runnable programs that are universally compatible, offering high performance and low-level control.
- Security Measures for Device Authentication: Implementing robust security measures, especially in the authentication process, is a noteworthy achievement.

By authenticating each device before establishing communication, the system ensures the integrity of incoming data, mitigates cyber threats, and safeguards against malicious activities.

These achievements collectively contribute to the success of the project, providing a solid foundation for further enhancements and ensuring a resilient and scalable IoT platform.

12.2 Future Directions

As we reflect on the current state of the project, it's essential to consider potential avenues for growth, enhancement, and adaptation to emerging technologies. The following outlines some key areas and future directions that could propel the project towards continued success.

- Machine Learning Integration for Predictive Analytics: Explore the incorporation of machine learning algorithms to analyze historical data and predict device behavior. This could contribute to proactive maintenance, identifying potential issues before they escalate, and optimizing the overall system's performance.
- Enhanced Frontend Features: Expand the frontend's capabilities by integrating more advanced features. This could include customizable dashboards and user-specific settings. Additionally, incorporating progressive web app (PWA) features for offline functionality would improve user experience.
- Continuous Security Audits and Updates: Establish a routine for continuous security audits and updates to stay ahead of potential vulnerabilities. Regularly review and enhance security protocols, considering the evolving nature of cyber threats and the importance of maintaining a secure IoT ecosystem.
- Integration with External Services: Explore integrations with external services and APIs to enrich the functionality of the platform. This could include weather data integration, third-party analytics tools, or collaboration with other IoT platforms for interoperability.
- User Feedback Mechanism: Implement a robust mechanism for collecting user feedback and feature requests. This can provide valuable insights into user needs and expectations, guiding the development team in prioritizing enhancements and improvements.
- Implementation of Digital Twins: Consider incorporating the concept of digital twins, creating virtual replicas of physical devices. This allows for

in-depth monitoring, simulation, and analysis of device behavior, enabling more accurate predictions and optimizations.

These future directions represent exciting possibilities for advancing the project's capabilities, staying at the forefront of IoT innovation, and addressing the evolving needs of users and stakeholders. The roadmap for future development should be flexible, allowing for adaptation to emerging technologies and industry trends.