Data lake for aggregation of production data and visualization tools in the stamping industry

Leonardo Leite Meira dos Santos - 54363

Dissertation presented to the School of Technology and Management of Bragança for the attainment of the Master's Degree in Sistemas de Informação.

Work guided by:

Prof. Paulo Alves

Prof. Kecia Marques

This dissertation does not include the criticisms and suggestions made by the Jury.

Bragança

2022-2023

Data lake for aggregation of production data and visualization tools in the stamping industry

Leonardo Leite Meira dos Santos - 54363

Dissertation presented to the School of Technology and Management of Bragança for the attainment of the Master's Degree in Sistemas de Informação.

Work guided by:

Prof. Paulo Alves

Prof. Kecia Marques

This dissertation does not include the criticisms and suggestions made by the Jury.

Bragança

2022-2023

Abstract

In this master's dissertation, the main objective is to develop a system for monitoring industrial sensors, specifically for the stamping industry. The work is motivated by the need for real-time monitoring systems to track machine operation in production, thus enabling data-driven management. The system was implemented using Python on the backend with FastAPI for the API, MongoDB for data storage, and NextJs for the dashboard where the information is displayed.

The results indicate that the system is capable of monitoring, analyzing, and presenting sensor data in real-time, with an alert mechanism that triggers notifications based on predefined parameters. The state of the art was reviewed to better understand emerging techniques and technologies in related areas, such as industrial Internet of Things (IoT), big data, and real-time data analysis.

The current implementation, although simpler compared to the solutions found in the literature, served as a valid proof of concept and is highly adaptable for future iterations based on feedback from the real production environment. It is concluded that the developed system meets the initial requirements and offers a certain degree of flexibility to adapt to other contexts and make future enhancements.

The application of Artificial Intelligence (AI) for data analysis and predictive maintenance of machines are likely directions for research and development of future work.

Keywords: Industrial Internet of Things (IIoT), Real-time Monitoring, Sensor Data Analytics, Data Lake

Contents

List of Figures

Chapter 1

Introduction

1.1 Framework

In the contemporary industrial scenario, the constant pursuit of efficiency and innovation has become an essential pillar for the competitiveness and financial sustainability of companies **Bonilla2018**. As technology advances, companies face pressure to remain competitive in the market **ashraf2020optimization**. In this context, monitoring and optimizing machines in production lines become crucial to ensure effective operation and prevent potential downtime or operational failures.

However, the tradition of industrial practices is often characterized by manual inspections and outdated monitoring systems that fail to provide real-time information or in-depth analysis of machine performance. This technological gap can result in significant losses in terms of production, financial resources, and machine maintenance, considering that investment in technology and monitoring can make companies more financially efficient **mabad2021rfid**.

In addition, with the increasing integration of IoT systems and the proliferation of advanced sensors, there is an immense amount of data being continuously generated, demanding more efficient processing **bajaj2021iot**. However, without the proper infrastructure to store and analyze this data, companies may find themselves overwhelmed,

unable to extract meaningful insights that could inform strategic and operational decisions.

In this market context, a stamping industry company sought to build projects that enable the sensorization of their machines, storage and processing of data, and visualization of this information, in order to become more competitive, efficient, and profitable. These projects were grouped within the "ATTRACT - Digital Innovation Hub for Artificial Intelligence and High-Performance Computing - Project: 101083770 — ATTRACT — DIGITAL-2021-EDIH-01" context, with funding from the "Digital Europe Programme (DIGITAL) - DIGITAL-2021-EDIH-INITIAL-01 — Initial Network of European Digital Innovation Hubs", referred to as the `Attract Project` in this document.

1.2 Objectives

Given the previous framework, the identified need is to develop a robust system that can receive data from sensors that collect real-time data from machines, store them efficiently in a data lake, and present them through a dashboard, transforming the way the company monitors and optimizes its production line, ensuring not only efficiency but also a proactive approach to maintenance and industrial management. This system would not only provide real-time information about the status and performance of the machines, but it would also allow for historical analyses, helping managers and technicians to identify trends and failures, as well as optimize production, minimizing production losses and maximizing financial gains.

1.3 Document Structure

This master's dissertation is organized starting with the Introduction, where the problem and scope are presented. This section contextualizes the need for industrial modernization, highlighting the importance of machine monitoring and optimization in the industry. The relevance of the study is then justified, focusing on the increasing demand for data analysis

in the market.

This is followed by the Literature Review, which presents an analysis of works and concepts related to the project. This section encompasses similar works already carried out in the parts of real-time data reception, processing, and alerts.

The Methodology discusses the choice of specific technologies used in the project, detailing the data storage method, and the process adopted for the software development, and also describes the strategies for managing the project activities.

In the chapter on System Architecture, the technical specifics of the proposed system are discussed. From the general system diagram to the way each part of the system is organized, making clear how each part is organized and how the interaction between them occurs.

In the Implementation chapter, the technical aspects of the system are delved into. Here, each component of the database, the API, the data processing module, the data receiving module, and the frontend, are detailed. This chapter aims to show in a practical way how the architecture detailed in the previous chapter was implemented.

In regard to the System Characteristics from a functional point of view, the dissertation has a section that focuses on demonstrating the practical application of the software, showing how each feature works and how user interactions occur.

In Results and Evaluation, the results obtained during the project are presented, highlighting the main achievements and benefits observed in the implementation of the proposed system. The Conclusion and Future Works section summarizes the key points of the project, identifies the limitations of the current system, and suggests possible directions for continuity and future implementations.

Chapter 2

State of the art

The literature review for the development of this project was carried out by researching projects that were similar to this one in some aspects, such as the receipt, storage, and processing of Internet of Things (IoT) sensor data, and real-time information transmission.

2.1 Data storage and big data

In the context of data storage and big data, the importance of managing and analyzing large data sets has been emphasized in various studies. One of these studies is the work on the FastQ Quality Control (FQC), a software designed to manage information from FASTQ files **fqc2017**. Developed in Python and JavaScript, the FQC aggregates data and generates metrics that are displayed on a dashboard. The software is capable of processing single-end or paired data, and can process batch files based on a specified directory.

It is important to note that the software allows customization, users can configure Key Performance Indicator (KPI), charts, and other dashboard elements. The data processing layer of the FQC bears similarities with industrial sensor monitoring systems of this dissertation, mainly in aspects of data access and processing. It operates by accessing a directory, processing the information, and making it available for later use, and also supports the execution of small batch functions.

The dashboard built by the FQC offers a variety of visualizations, including line charts, bar charts, and heat maps, among others. These visualizations are dynamically generated and can be configured using JavaScript Object Notation (JSON) files **mdnJson**, providing a flexible and user-friendly interface for data management and analysis.

Another contribution to the field of data storage and big data is the work of Ren et al. **ren2021data**, `Quality Mining in a Continuous Production Line based on an Improved Genetic Algorithm Fuzzy Support Vector Machine`, which focuses on predicting product quality through a data-driven approach. The Indicadores Chave de Desempenho (KPIs) are identified to serve as highly relevant state variables for product quality. Traditionally, these variables are measured through offline laboratory analyses, which introduces latency into the system.

The AI component of the system is layered, where the lower layer deals with both categorized and uncategorized data to train and test the AI models. A Gaussian distribution is applied in the second layer to process the data, which are then fed into a semi-supervised training layer. The final layer provides the results of the predictions, thus closing the cycle.

Case studies presented in the article demonstrate the implementation of this system in industrial mineral processing processes. Although the method successfully addresses the issue of unmarked records, it requires a high degree of continuity in industrial systems, which is identified as a limitation.

The work of Ren et al. provides insights into the use of data for predictive quality control in industrial environments. The layered AI model and the focus on KPIs are especially relevant for the future development of the system, which can use the analyses carried out over time as training data for the AI.

Still within the scope of data storage and big data, Predictive Maintenance (PdM) becomes a very relevant strategy, both in the context of this project and in semiconductor manufacturing **susto2015machine**, as explained in `Machine Learning for Predictive Maintenance: A Multiple Classifier Approach`. The article by Susto et al.**susto2015machine**

presents a multiple classifier approach to PdM, aiming to minimize downtime and associated costs. Three main categories for maintenance management are identified: Run to Failure, Preventive Maintenance, and Predictive Maintenance. The latter is emphasized for its ability to leverage historical data, forecasting algorithms, statistics, and engineering methods.

The paper uses several trained classification modules with different forecasting horizons to offer various performance trade-offs. Two main indicators are identified to reduce total operational costs: the frequency of unexpected breakdowns and the amount of unutilized lifespan. Linear regression is used as a statistical method for forecasting.

The approach is particularly relevant for systems that require real-time data analysis and efficient data storage, such as industrial sensor monitoring systems. It addresses the limitations associated with the lack of continuous data feed in industrial systems and offers a cost-based decision-making system for maintenance management.

Therefore, the work of Susto et al. provides insights into the application of machine learning for predictive maintenance, especially in semiconductor manufacturing. The methodology can be particularly beneficial for industrial environments where minimizing downtime and operational costs are essential, and could be a logical evolution of this project, given the storage of historical data received in the system that can be used for predictive maintenance.

2.2 Real-time Monitoring

In the context of industrial sensor monitoring and real-time data analysis, the paper by Shafi et al. **shafi2019precision** `Precision agriculture techniques and practices: From considerations to applications` is relevant as it offers a comprehensive exploration of precision agriculture techniques, with a particular focus on IoT-based intelligent irrigation systems. These systems face similar challenges to those in industrial environments, such as latency, bandwidth limitations, and intermittent internet connectivity.

Edge computing (fog computing), as discussed in the article, emerges as a cutting-edge

solution to these challenges. It aims to save energy and bandwidth, reducing failure rates and delays. This is particularly relevant for real-time data analysis and alert systems in industrial sensor monitoring. The Fog of Everything architecture, introduced in the article, offers a multi-layered approach that could be adapted for industrial applications aiming to improve service quality and efficiency in data storage. The methodologies and technologies discussed in the article provide ways of system organization for data storage, big data, and sensor monitoring systems.

The article `The Role of Big Data Analytics in Industrial Internet of Things` by Rehman et al. **REHMAN** explores the integration of Big Data Analysis (BDA) with Industrial Internet of Things (IIoT), focusing on real-time data analysis, data management and storage, aspects that connect with the aim of the dissertation to develop a robust system for monitoring industrial sensors.

The article's discussion on real-time analysis can guide the development of the Data Reception Module, and the data processing module for historical data analysis in this project. In addition, the article's insights on data management and storage can offer paths to optimize the performance of the MongoDB database if necessary.

Although the article does not specifically discuss alert systems, its categorization of analysis techniques into descriptive, prescriptive, predictive, and preventive procedures can provide a framework for generating alerts based on predefined parameters. Moreover, the article's focus on interoperability and integration in IIoT systems may offer guidelines for the effective design and integration of the various modules in this project, such as the Database, Data Receiving Module, and API.

An important project to highlight in real-time data monitoring is presented in the article `Internet of Things in Vehicle Safety - Obstacle Detection and Alert System` by Umakirthika et al. **Umakirthika2018**. This extensively explores the use of Internet of Things (IoT) technologies to enhance vehicle safety. The article introduces an Obstacle Detection and Alert System (ODAS) System designed to identify obstacles on the road and alert drivers in real-time. The system uses embedded algorithms that detect obstacles based on various vehicle parameters, such as speed and steering angle. Once an obstacle

is detected, its location is stored locally and sent to a cloud server periodically. The cloud server processes these data, confirms the presence of a real obstacle, and sends this information back to the vehicle's alert system, which provides audible and visual alerts to the driver.

This article's approach can provide important information for the design and implementation of real-time alert systems in an industrial environment, specifically, the use of IoT for data collection and cloud processing can be adapted to enhance the real-time analysis capabilities of this system. The article also discusses the challenges associated with implementing such a system, including data security and latency, which are important points to consider when transmitting real-time data and generating alerts about machine operation.

In the field of data storage and big data, machine learning algorithms have been widely studied for their ability to analyze and interpret large data sets. A review conducted by Sarker **sarker2021machine**, in `Machine Learning: Algorithms, Real-World Applications and Research Directions`, elucidates various statistical and machine learning techniques pertinent to feature selection and data analysis. Methods such as Variance Analysis and Chi-Square tests are highlighted for their utility in identifying statistically significant features in data sets. These techniques are particularly relevant for systems that require real-time data analysis and decision-making, such as industrial sensor monitoring systems.

In addition, the article discusses the application of machine learning algorithms in various areas, potentially including industrial settings and the Internet of Things (IoT). Although the article does not specifically delve into real-time data analysis, the algorithms and methods presented can be adapted for such purposes. For instance, machine learning algorithms can be employed to predict sensor failures or other anomalies based on historical data, thereby enhancing the robustness and reliability of industrial monitoring systems.

In this way, the methodologies and algorithms discussed by Sarker **sarker2021machine** provide guidance for the development of systems that require efficient data storage and real-time analysis capabilities.

2.3 Data Processing and Analysis

The article `Next-Generation Big Data Analytics: State of the Art, Challenges, and Future Research Topics` by Lv et al. **Lv2017** provides a comprehensive review of the current landscape of BDA, covering various types of data, storage models, and analysis methods. The article also addresses the challenges and future research directions in BDA, emphasizing the role of emerging technologies such as edge computing, machine learning, and blockchain.

In the context of this dissertation, the article's comprehensive treatment of data storage models and analysis methods is particularly relevant when it discusses storage models, including NoSQL databases like MongoDB, which can provide insights to optimize the database component of this project.

Furthermore, the exploration of various analysis methods by the article, such as machine learning algorithms and real-time analysis, can serve as a guide for future improvements in data analysis by the Data Processing Module in this project. The article also discusses the integration of edge computing for real-time analysis, which could be a future direction for this dissertation in order to make the system more scalable and efficient in an industrial environment.

Although the article covers a wide range of topics, its general principles and methodologies can be adapted to the industrial context of this dissertation. The article not only describes the practical applications of BDA, but also discusses challenges, such as data security and privacy, and future research directions.

Another relevant article is `Big data analytics in smart grids: a review` by Zhang et al. **Zhang2018**, which provides a comprehensive review of the role of BDA in the context of smart grids. The article explores various analysis techniques, including machine learning algorithms, statistical methods, and data mining, and their applications in smart grid systems. It also discusses the challenges and future directions in the field, such as data security, real-time analysis, and integration of renewable energy sources.

The exploration of machine learning algorithms and statistical methods by the article

can be especially informative to enhance the real-time data analysis performed by the Data Processing Module in this project. Moreover, the article's discussion on real-time analysis is directly relevant to the focus of this dissertation, as it explores the challenges and solutions in implementing real-time analysis in smart grids, which could offer options for the development of this system's real-time analysis capabilities.

The paper also discusses data security challenges, which could be pertinent when considering the secure API for managing access to sensor data in an industrial environment. Although the paper is focused on smart grids, its general principles and methodologies can be adapted to the context of industrial sensor monitoring of this dissertation.

An important work that discusses data structuring is presented in the paper `Advanced data analytics for enhancing building performances: From data-driven to big data-driven approaches` by Fan et al. **Fan2021**. This paper presents a critical review of data-based methods for building energy modeling and their practical applications for improving building performance. Although the main focus of the paper is on building energy systems, its methodological approach and findings provide important insights for this project. The paper categorizes analysis techniques into descriptive, prescriptive, predictive, and preventive procedures, and discusses the transition from traditional data-based methods to big data-based approaches.

The detailed discussion of the article on data-based methods can guide the statistical data analysis carried out by the Data Processing Module. Moreover, the exploration of big data-based approaches by the article may offer new perspectives to optimize the MongoDB database used in this system. The article discusses the challenges and opportunities in transitioning from traditional data storage and analysis methods to big data-based approaches, which may be relevant to enhance the system's adaptability and performance in an industrial environment.

Although the article focuses on building performance, its general principles and methodologies on data analysis can be adapted to the context of industrial sensor monitoring of this dissertation. The article also goes through the practical applications and challenges of data-based methods, which can serve as a guide for future improvements and adaptations

of the system based on feedback from the real production environment.

2.4 Development Methodology

In the field of software development methodologies, the article by Radha Shankarmani titled "Agile Methodology Adoption: Benefits and Constraints" **shankarmani2012agile** provides valuable insights into the iterative nature of agile processes. The article emphasizes the importance of iterative cycles that deliver tangible and usable results after each iteration. Furthermore, it discusses the role of customer checkpoint reviews for quality assurance at the end of each iteration, ensuring that the work aligns with predefined quality standards. The article also highlights the importance of post-release reviews that generate feedback for improvement plans, which is of utmost importance for the process.

These agile practices are particularly relevant to the current project. Given the focus on real-time data analysis and alert systems, the iterative approach and feedback mechanisms discussed in Shankarmani's article can be instrumental. Specifically, iterative cycles can facilitate incremental development and refinement of system modules to better meet needs according to received feedback. Furthermore, the feedback generated by post-version reviews can be invaluable for improving the overall quality and performance of the system, especially when the software is put into production.

Thus, the agile methodologies discussed in the article can serve as a guiding framework for managing the evolutionary process of the system over time, ensuring its adaptability and adequate responsiveness to the context and needs of the company.

2.5 Conclusion

In the review of the state of the art, various approaches were analyzed, covering areas such as data storage and big data, real-time monitoring, data analysis processes, and methodology for software products. These topics are crucial for understanding and developing a robust system for monitoring industrial sensors. The reviewed academic articles

provided important information in both technical aspects and broader context considerations. From a technical standpoint, the methods and technologies explored in the consulted works helped to understand issues related to efficiency in real-time communication between sensors and servers, as well as the structuring of databases to accommodate and retrieve large volumes of information.

Furthermore, the state of the art provided guidelines on advancements in the application of AI for data analysis. These applications prove to be especially relevant for areas such as predictive maintenance, which not only benefit from real-time data analysis, but also from the ability to make reliable predictions about future system states. This integration of AI can open doors for more proactive and less reactive monitoring, contributing to the overall increase in efficiency and reduction of operational costs.

Regarding the context, the review of the state of the art also helped to identify the challenges and opportunities that may shape future versions of this monitoring system. Trends in emerging technologies and industrial practices can be informed by these academic reviews, allowing the project to stay aligned with the most recent advancements in the field and prepared to meet new demands that may arise. Regarding the current implementation, it is pertinent to mention that it presents a lower level of complexity compared to the solutions detailed in the discussed literature since the goal is to develop a first functional version of the system, which can be iteratively improved. This approach allows for faster validation in real production environments, paving the way for refinements based on practical feedback and observed performance.

Chapter 3

Methodology

The methodology section explores an approach to software development, starting with the definition of requirements, moving on to the method of data reception, and the selection of technologies. The development process incorporates the creation of user stories, organized on a Kanban board for task management, and the configuration of repositories. Documentation and regular meetings that track progress are a constant in all phases to ensure that the project is proceeding as expected.

3.1 Requirements Definition

The precise definition of requirements is crucial to ensure that the developed system meets the project's needs and objectives in an agile way **asghar2016role**. The requirements of this project were classified into functional and non-functional categories, to ensure a complete understanding of what is expected from the system.

3.1.1 Functional Requirements

Functional requirements play a fundamental role in system development, defining the functions that a system or software component should be able to perform. Essentially, they provide a description of the interactions the system will have with its users or other

systems, specifying the services the system should provide.

To ensure effectiveness, functional requirements must be clearly defined, unambiguous, and measurable, traceable, complete, and consistent. Moreover, they should be defined considering the needs and objectives of the project, ensuring that the developed system is not only technically sound but also useful and relevant to its end users.

Within the context of the developed system, the functional requirements of the system and their description are listed below, to clearly state, functionally, what the system should do.

FR1 - The system must allow a user to securely access the system with an email and password

Given that the system is for viewing the operational data of a stamping industry, the information provided should only be accessed by previously authorized users.

FR2 - The system must allow a user to view their personal information that is stored in the system

Each user who has access to the system will have some of their personal data registered in it, such as email, position, and type of access. Therefore, each user should have access to their personal information that is saved in the system.

FR3 - The system must display in real time the values read by the sensors in each of the machines

Upon receiving the data sent by the sensors, the system should display on screen the read values, separated by sensor type and machine.

FR4 - The system must store an ideal maximum value for each type of sensor used

Each sensor should have an ideal maximum value for operation. It will serve as a parameter to understand whether the value read by the sensor indicates good or poor machine performance.

FR5 - The system must identify whenever a value read by the sensor is not below the ideal value

This requirement refers to the system's ability to automatically detect every time the sensor indicates a value that is not below the pre-defined limit. That is, if the ideal value is X, and the sensor reads a value greater than or equal to X, the system will recognize this situation.

FR6 - The system must always register when a value read by the sensor is not in accordance with the ideal value

This requirement implies that the system must keep a record of all times when the value detected by the sensor is not below the stored ideal value.

FR7 - The system must display on screen when a value read by the sensor is not below the ideal

Whenever the sensor detects a value below the ideal standard, the system should display an alert on the interface so that it is always visible to the user.

FR8 - The system must display in notification format the records of non-operation below the ideal value

This requirement establishes that the system should present to users in the form of notifications when the sensor reads a value above the ideal, to enable users to be informed, even if later, whenever an alert is identified.

FR9 - The system should allow the user to mark a notification as read, so that it does not appear again

After being notified, users should have the ability to mark this notification as "read", ensuring that the same information does not continue to be displayed repeatedly.

FR10 - The system should display graphs showing the values read by the sensors on previous days in an aggregated manner, separating by machines

This requirement ensures that users can view, through graphical representations, the readings of sensors from previous days in an aggregated manner. These graphs should be categorized by machine, providing a detailed analysis of the performance of each piece of equipment over time.

FR11 - The system must display in the graphs a statistical analysis of the machines' operation, along with the maximum ideal operating value

The graphs should provide a statistical analysis, showing the statistical indicators of the aggregated data average, median, 75th percentile, and average removing outliers. Along with this, the graph will also show the ideal value, serving as a reference for evaluating performance.

FR12 - The system must allow filtering the information displayed on screen by machines

Users should have the flexibility to select and view specific information for certain machines, allowing them to focus on specific equipment as needed.

FR13 - The system must allow filtering the charts displayed on screen by date

The system should offer the ability for users to filter graphic displays by specific dates, allowing for detailed temporal analyses and comparisons between different periods.

FR14 - The system must display the machine stoppage charts in a way that exemplifies the display of this data

The system should display machine stops according to the data passed by the spreadsheets with the data. In this way, it can be exemplified how the machine stop information would look if the system received this information.

3.1.2 Non-Functional Requirements

Non-functional requirements are specifications that determine the performance characteristics, usability, reliability, and other properties that the system must possess, rather than specific behaviors it should demonstrate. While functional requirements describe what a system should do, non-functional requirements specify how the system should perform these functions.

These requirements are crucial to ensure user satisfaction and the operational effectiveness of the system, playing a fundamental role in the quality and overall operation of a software product.

Non-functional requirements can be of various types, such as usability, performance, security, availability, maintenance, and reliability. Within the context of the developed system, the non-functional requirements of the system and their description are listed below, to make clear what was taken into account when developing each of the system's functionalities.

NFR1 - Availability

The system must have automatic reconnection mechanisms that activate when connection problems or data reception from sensors are detected, thus ensuring the continuity in data reception.

NFR2 - Access Security

The system must implement access controls so that only authorized employees have permission to access data and functionalities relevant to their role.

NFR3 - Network Security

To ensure the security of data transmission, the connection to the system must be established using the HyperText Transfer Protocol Secure (HTTPS) protocol, which incorporates the Transport Layer Security (TLS) security layer, thus protecting the data against interceptions and alterations.

NFR4 - Real-time Transmission

The system must process and transmit the data sent by the sensors in a streaming-based architecture. The delay between the sensor sending the data and its visualization by the end user should be less than three seconds.

NFR5 - Modularity

The system's architecture should be modular, allowing for the integration and addition of new components or functionalities in an efficient manner without compromising the operation of the existing parts.

NFR6 - Maintainability

Prioritizing longevity and ease of maintenance, the system should be developed following good programming practices and system modularization. This will facilitate future modifications, expansions, and the correction of any potential problems.

NFR7 - Scalability of sensors and machines

The system design must be able to handle an increasing volume of sensors and machines, ensuring that there is no performance degradation or failures when the demand for resources increases.

NFR8 - Portability

The system must ensure compatibility with the main web browsers available on the market. In addition, the user interface should adapt well on larger screens such as televisions, allowing the dashboard to be clearly viewed in different factory environments.

NFR9 - Usability

The system interface and its components should be designed considering fundamental principles of interaction design, ensuring that users can understand and interact with the system in an intuitive and efficient manner.

3.2 Data Collection and Storage Method

Within the project context, the way sensor data is collected and stored greatly influences the system's operation, as it is from them that the entire system is structured. Thus, a protocol developed in another project within the same context of the `Attract Project` was used as a basis, which transmits all the necessary information for the context of this project. Within the system in question, the responsibility of implementing the decoder for the given protocol was assigned.

The protocol format is structured to represent the information pertinent to the machine, the type of communication, the sensor, and the meaning of the transmitted data, following the format below:

Machine ID (2 bytes)

The *Machine ID* field is responsible for identifying the machine in question and is divided into two subfields:

- **High (higher order bytes)**: Represents the type of machine. Possible values include: press, lathe, robot, conveyor, among others.
- **Low (lower order bytes)**: Identifies the machine number.

Type (1 byte)

The *Type* field indicates the type of message and can assume the following values:

1. Publish
2. Request to publish

Sensor ID (2 bytes)

The *Sensor ID* field provides details about the sensor that is transmitting the data:

- **High (higher order bytes)**: Represents the physical quantity being measured, such as temperature, speed, pressure, force, among others.
- **Low (lower order bytes)**: Indicates the sensor number.

Meaning of Data (2 bytes)

The *Meaning of Data* field provides information about the type and meaning of the data:

- **High (higher order bytes)**: Data type:
    1. Not defined
    2. Normal
    3. Raw data

4. Alarm

- **Low (lower order bytes)**: Meaning of the data, which varies according to the equipment. Examples include:
  - Oil critical temperature
  - Check oil temperature
  - Oil pressure

Length (2 bytes)

The *Length* field indicates the number of subsequent bytes in the package.

Data

This field represents the data transmitted by the sensor. The exact specification of what the data represents should be defined and standardized, as indicated by the notation (*).

3.3 Software development process

With a clear definition of the system requirements and how the data read by the sensors are transmitted, the process of how the project would be developed was defined. This development process involves defining user stories, organizing activities, organizing documentation, setting up repositories on GitHub, and holding regular meetings with the supervising professor and the company to discuss progress.

3.3.1 User Stories

In agile software development, one of the most user-centered approaches to understanding system features and requirements is the use of *user stories*. These are short, simple, and informal descriptions from the perspective of an end user, capturing what they need or want to do in the software **lucassen2015forging**.

The typical structure of a user story is: "As a [type of user], I want [an action] so that [a benefit/outcome]". This structure helps to keep the focus on the user's needs and desires, rather than prematurely diving into technical solutions or implementation details.

In addition to being a communication tool between developers and stakeholders, user stories facilitate task prioritization, assist in creating acceptance criteria, and provide a basis for interactive discussions during review and planning meetings.

In short, user stories serve as an effective means of translating complex requirements into manageable, user-centered tasks, ensuring that the final product meets the expectations and needs of its users.

With this definition of user stories, the following items were defined that translate the requirements into tasks for the project development:

1. As a user, I must be able to log in with my credentials to use the system.
2. As a user, I should be able to view my personal information on a profile page to manage the data the system holds about me.
3. As a user, I should be able to view in real-time values of the machines to detect relevant variations in operation more quickly.
4. As a user, I should be able to view the historical values of the sensors aggregated in charts of the machines to monitor the status over time.
5. As a user, I should be able to filter the dashboard information by machine and by date to view data according to my needs.
6. As a user, I should be alerted when a sensor reads a value that exceeds the ideal parameter so I can take necessary actions as quickly as possible.
7. As a user, I should be able to view system notifications to be alerted about machine operation alerts.
8. As a user, I should be able to view the machine downtime records for a better and more organized view of the recorded machine downtimes.

9. As a user, I should be able to view the system information (dashboards) on different screen sizes so that I can display the information in different contexts.

Each of these user stories was further detailed in the task organization, including a more complete description, possible business rules, which requirements, functional and non-functional, it refers to, and also acceptance criteria. The organization of activities is detailed in section 3.3.2.

3.3.2 Task Organization

The Kanban method, originating from the Toyota production system, has become a popular and effective tool for managing and organizing workflows. The word "Kanban" is of Japanese origin and can be translated as "visual card" or "signage". In the context of project management, Kanban refers to a visual management system that highlights the workflow and tasks at different stages of the process. The essence of Kanban is to visualize the entire workflow, from tasks that have not yet been started to those that have been completed. This visualization allows the identification of bottlenecks and inefficiencies, thus optimizing the process **ghani2015agile**. For the organization of the activities of this project, the Kanban method was adopted as a strategy to ensure an efficient and systematic progression of work.

For the implementation of the Kanban method, Notion was chosen as the tool. The choice of this software is due to its flexibility and customization capacity, allowing the creation of a Kanban board that specifically adapts to the project's needs **notionProjectManagement**. In addition, Notion offers an intuitive interface for building documentation, which is further detailed in section 3.3.3.

The Kanban board was structured into five columns, each representing a distinct stage in the workflow:

- **Backlog:** This column contains all identified tasks and activities that have not yet been started. It is a repository of everything that needs to be done, but does not yet have a defined start date.

- **To Do:** The tasks in this column are ready to be started. They have been taken from the Backlog, detailed, and have priority to be started soon.
- **Stopped:** Here are the tasks that have been started, but for some reason had to be interrupted, from changes in priority, the need for some validation, or technical limitation.
- **In Progress:** This column contains the tasks that are currently underway. The move to this column indicates that work is actively being done on the task.
- **Done:** As soon as the development of a task is completed, it is marked as completed, and therefore, moved to this column. It represents the success in completing the activity, and serves as a record of all completed items.

The structuring of these columns provides a clear view of the status of each activity and helps to quickly identify where the bottlenecks are, facilitating decision-making and task prioritization.

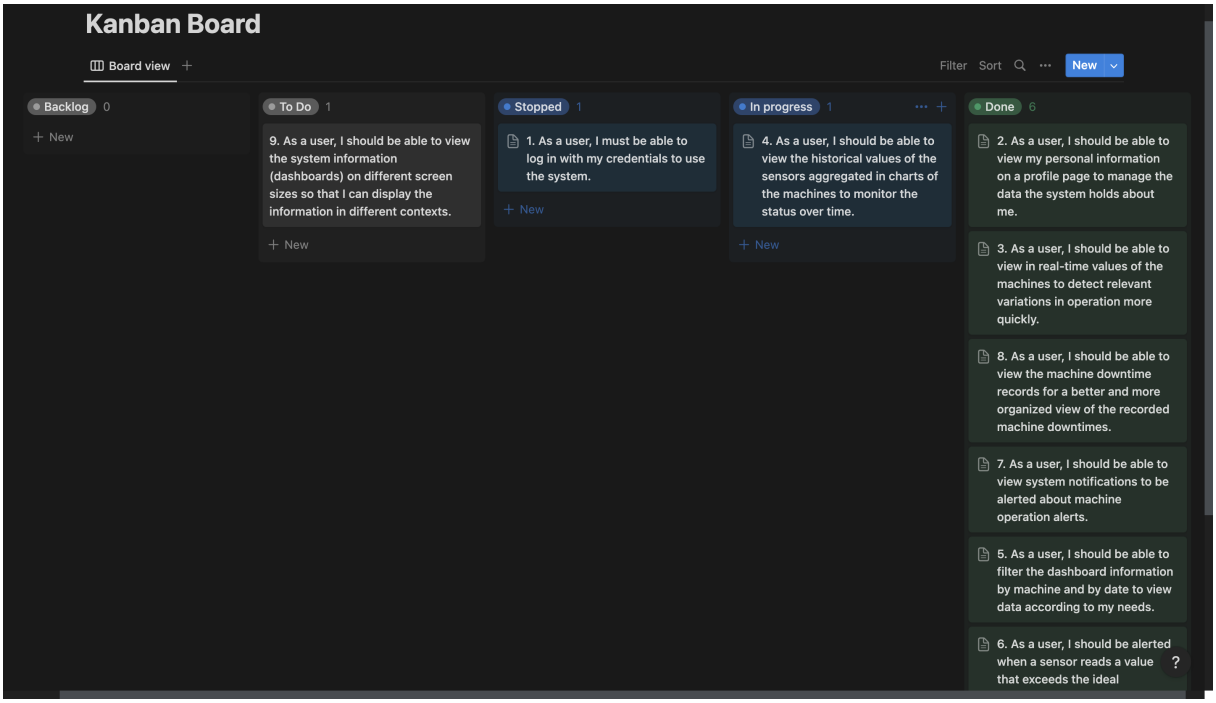

Figure 3.1: Kanban board used to manage the tasks.

In order to better define the implementation of each user story, each card on the Kanban board was detailed with a description that includes relevant business rules, references to functional and non-functional requirements, acceptance criteria, and sub-tasks. Before a user story can be moved to the "In Progress" column, it is essential that these fields are evaluated to ensure an understanding of the task scope. The acceptance criteria play an important role in verifying that a story meets all the established requirements before it is marked as completed.

To illustrate this process, user story number 1 is detailed.

1. As a user, I must be able to log in with my credentials to use the system.

- **Description:** To ensure security and customization of the user experience, the system must have an authentication feature. The user must enter their credentials - usually a username or email address and a password - to access their account and the functionalities associated with it.
- **Relevant Business Rules:**
  - Users cannot access the system without authentication.
  - Attempts to log into the system should be stored in the database.
  - Passwords should be stored securely, using techniques such as hashing.
- **Requirement References:**
  - **Functional:** FR1
  - **Non-Functional:** NFR2, NFR3, NFR6, NFR9
- **Acceptance Criteria:**
  - The system must present a clear and intuitive login screen.
  - After correctly entering the credentials, the user should be redirected to the system's home page (/dashboard).
  - If the credentials are incorrect, the user should receive a clear error message.

- **Sub-tasks:**
  - Design the login screen interface.
  - Build the login interface in the repository.
  - Implement the user creation logic with password hash.
  - Implement the authentication logic in the backend with JSON Web Token (JWT) Token.
  - Test the security and effectiveness of the login functionality.
  - Test if the API is returning the HyperText Transfer Protocol (HTTP) codes correctly.

3.3.3 Project Documentation

The project documentation was created using the Notion tool. The choice for this project was based on the fact that Notion stands out for its flexibility, allowing an adaptable configuration of the text to meet various types of needs. The intuitive interface makes the insertion and updating of information a simple process, which can be executed as it is a platform accessible on the internet by any browser **notionProjectManagement**.

In this way, the documentation was elaborated adopting a structured approach to ensure that all features and functionalities were properly cataloged. The basis of this documentation is a table, where each entry corresponds to a functionality or a specific part of the system, being named according to the name of the feature in question.

Each item in the table expands into an independent page, with descriptive details. This organization allows understanding in each aspect of the system, as it was being built as the system was developed, thus reflecting the most recent additions and changes.

To facilitate navigation and search for specific information, a tag column was incorporated into the table. These tags serve to categorize functionalities, and also an efficient search mechanism, allowing users to quickly identify the relevant aspects of the system.

In addition, the internal design of each page is based on the structure of markdown, explained in **markdownguide**, to ensure that the content of the documentation is presented in a clear, structured, and aesthetically pleasing way, facilitating the understanding and absorption of information by the reader.

3.3.4 Repository Configuration

The platform chosen for managing the project's repositories was GitHub **github**, one of the most renowned and widely adopted Git-based version control platforms currently available. The decision to use GitHub was based on several factors. Firstly, the platform offers an intuitive interface and a robust set of tools that facilitate tracking code progress, as well as collaboration among different team members. In addition, GitHub is widely recognized for its active community, which translates into a vast range of resources, tutorials, and available support, essential for resolving possible doubts and challenges. Furthermore, integration with other tools and platforms is easily achievable if necessary, allowing for a continuous and optimized workflow. Finally, GitHub's commitment to security, ensuring the integrity of the project's code and data, reinforced our decision to adopt it as the version control solution for the developed project.

In this context, the repositories created for this project were `backend`, which stores the code related to the system's API, another called `frontend`, which stores the code of the dashboard that is displayed on the web, and another called `iot_sensors_data_aggregation`, responsible for storing the code that performs the aggregation of the data received by the sensors, the Data Processing Module.

3.3.5 Periodic meetings

The development of the project was accompanied by meetings to ensure its alignment with the project objectives. Weekly meetings with the supervising professor were established, ensuring a constant review, time to ask questions, and to detail activities. These meetings provided continuous feedback, allowing for trajectory correction and focus on the desired

progress for the project. In parallel, monthly meetings were conducted with the interested company, to present what was being built, gather feedback, guidance on desired features, and also to better understand how the company operates and its needs.

These meetings played a fundamental role in integrating academic research and the practical needs of the industry, ensuring that the developed solutions remained relevant and applicable to the business context. This supervision system was crucial in keeping the project on track, balancing academic needs with industrial applicability within the company's context.

3.4 Technologies

The selection of technologies was carried out to meet specific requirements of scalability and long-term sustainability. Given that the system is primarily aimed at data storage and management, its use as a reference for future projects with similar characteristics was anticipated, therefore, the emphasis was placed on modern technologies, widely recognized and with robust support in the development community.

In this context, MongoDB was chosen as our non-relational database solution, due to its flexibility and performance. Python was adopted as the language for the backend, due to its versatility and extensive library. The FastAPI framework, in turn, was employed for the development of the API, thanks to its efficiency and ease of integration. In the frontend scope, the JavaScript language was complemented by the NextJs framework, recognized for its optimization and advanced features. To ensure a fluid and modular integration of the system components, Docker containers were used, while efficient web server management was ensured by NGINX.

3.4.1 MongoDB

When selecting a database platform, MongoDB **mongodbDocs** was chosen, a non-relational database system designed to flexibly adapt to changes in the format of the

data that is stored. MongoDB provides ease in manipulating the data lake in various contexts. Its meticulously structured documentation, coupled with a vast range of content available online, proved to be invaluable for knowledge acquisition.

Distinctively, MongoDB presents advantages such as the ability to support distributed and parallel queries, optimizing the processing of intricate requests in scenarios with significant data density. Additionally, its compatibility with a wide variety of data analysis tools sets a promising precedent for future evolutions of the project.

3.4.2 Python

In the development of the backend, the Python language **pythonOfficialDocs** was chosen, widely recognized for its versatility, readability, and adaptability in various application contexts. Python, being one of the most popular and widely accepted languages in the academic and industrial world, presents a vast standard library and robust community support. The rich range of educational materials, which spans from detailed tutorials to extensive discussion forums, was essential for learning.

Python's intuitive syntax favors rapid prototyping and development, while the wide range of available frameworks and libraries enhances its application in various aspects, from data analysis to web development. These intrinsic characteristics, combined with the language's flexibility and efficiency, consolidate the decision to adopt Python as the central language for the backend in this master's project.

3.4.3 FastAPI

In the backend implementation phase, it was decided to use the Python language, combined with the FastAPI framework **fastapiDocs**. This choice was largely motivated by the efficiency and high performance offered by FastAPI. This framework stands out for incorporating the Asynchronous Server Gateway Interface (ASGI) library, an interface that optimizes request management, by fully leveraging asynchronous execution, ensuring more agile and accurate responses. An important feature of FastAPI is its comprehensive

and well-crafted documentation, which serves as a fundamental tool in the learning and development process.

3.4.4 NextJs

For the frontend architecture, the choice was made to use Next.js **nextjsDocs**, a framework that significantly enhances the interaction with the JavaScript React library **reactDocs**, as emphasized in the official React documentation itself. Community support is one of its main features, being widely complemented by a range of educational materials available on the internet - from tutorials to blog articles and instructional videos - which contributed essentially to the learning process.

In the scope of this project, alongside Next.js, TypeScript **typescriptLang** was incorporated, which, due to its static typing nature, provides a more intuitive code maintenance, increasing its readability, simplifying its understanding and management.

The coherence between Next.js and TypeScript establishes a highly effective development environment. While Next.js promotes a more fluid and high-performance development experience, TypeScript strengthens security and productivity, thanks to its rigorous typing. These factors justify the choice of the combination of Next.js and TypeScript for the realization of this project.

In addition to Next.js and TypeScript, Material UI 5 **muiDocs** was integrated into the project as a user interface design library. This set of React components, based on Google's Material Design standard **m3Docs**, offers a wide range of already stylized and easy-to-implement interface elements. Besides saving time in developing components from scratch, the library provides a cohesive and modern user experience. The use of Material UI 5 also contributes to the standardization of design throughout the project, ensuring a more intuitive and pleasant user experience. Therefore, the addition of this feature effectively complements the already robust capabilities offered by the combination of Next.js and TypeScript, making the development environment even richer and more productive.

3.4.5 Docker

For the orchestration and management of the development and production environment, Docker **dockerDocs** was adopted as the containerization tool. Docker, widely recognized in the software development universe, allows encapsulating applications and their dependencies in containers, ensuring uniformity, reproducibility, and isolation among environments **dockerOverview**. This approach significantly simplifies integration, testing, and deployment processes, as containers can be transparently moved between different environments and platforms.

The extensive documentation available, along with an active community, provided a clear understanding and facilitated the adoption of this technology. In addition, the flexibility and efficiency provided by Docker, by minimizing dependency conflicts and ensuring that the application works consistently in various contexts, were decisive factors for its choice in this project.

3.4.6 NGINX

For the part of managing web requests, NGINX **nginxDocs** was adopted as the web server. NGINX is recognized for its high performance, reliability, and flexibility, making it a suitable choice in production environments that demand low latency, efficient handling of a large number of simultaneous connections, and the ability to serve static content extremely quickly. These characteristics make it particularly suitable for systems aiming for scalability and robustness.

The extensive documentation and the vast community resources were essential to deepen the understanding and apply best practices in the project context. Considering the need for a consistent and optimized delivery of content to the end user, as well as a secure and effective proxy configuration, NGINX proved to be the preeminent choice for this master's dissertation.

Chapter 4

System Architecture

In this chapter, the architecture and structure adopted for the construction of the system components are discussed in detail. It is important to understand that the system was conceived as a set of modules, with each one performing specific functions, and when operated together, these modules result in the achievement of the purposes intended for the system.

The system is fundamentally structured in distinct layers, the backend, the frontend, and the database.

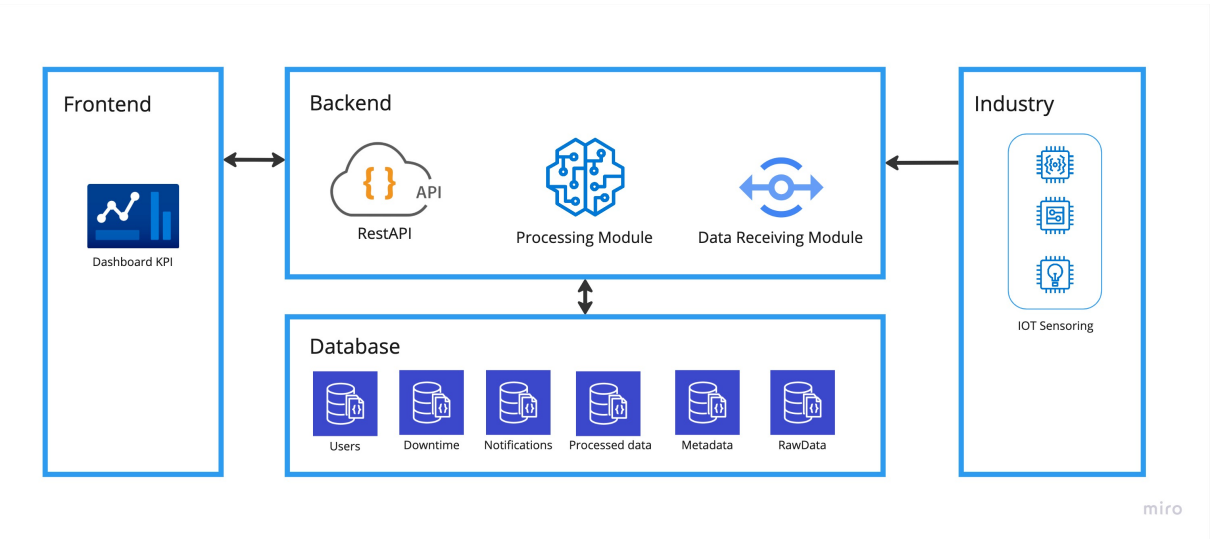

Figure 4.1: System architecture.

The backend functions as the core of the system. Its main role is to receive the data, process it according to the rules established in the requirements and user stories, and store it securely in the database. In addition to storage and processing functions, the backend is also assigned the responsibility of making these data available through an API, which can be accessed using HTTP methods. This API acts as an intermediary between the central logic of the system and the interfaces with which the end user interacts, the frontend. On the other hand, the frontend is characterized as the visual interface that the end user accesses. It serves as a means by which users interact with the system, sending and receiving information. This layer is designed to access, retrieve, and present the data processed and stored by the backend according to design principles and data visualization **barbosa2019introduction**.

As can be seen in figure 4.1, in the industrial plant, where the machines with the sensors are located, the sensors send the data to the system, which receives them through the data receiving module and stores them in the database. The processing module accesses the stored data to perform aggregation, and the API manages access to the database, making the information available to users on the frontend.

In the following sections, each of these components will be explored more deeply, going through their specificities and interactions.

4.1 Backend Architecture

This section addresses the operation of the backend. It is divided into three parts, the module for receiving data from the sensors, the processing module where the aggregation and statistical analysis of the data is done, and the API that manages access to information through HTTP requests.

Regarding the organization of the repositories, the API and the data receiving module are in the same repository, facilitating communication between them. The processing module, on the other hand, is in a separate repository, its only function being to read the database, process the data, and store the results.

4.1.1 Data Receiving Module

For the data receiving module, initially, the `SensorConnection` class stands out, whose function is to manage and maintain the connection with the sensor network. This class transmits the received data to a designated function `save_data_func`, ensuring that the data is forwarded for appropriate handling.

In the next part of the architecture, the `IotSensorConnection` class is used, which originates from the `IotSensorConnectionInterface` interface. This interface was created to ensure the system's adaptability, facilitating the integration of different types of data receipts, such as a class intended to generate sensor data in a development environment, where there is no access to the real sensor. The `IotSensorConnection` class, when instantiated, is responsible for establishing the connection, and creating a new thread that operates as an active listener, monitoring the arrival of new information. Upon noticing the reception of new data, the class directs this information to a third entity, which holds the responsibility of applying the business rules.

This third entity is the `SensorsRepository` class, which, when triggered with data from the sensors, has the responsibility to evaluate the information based on the established parameters, deciding whether it is necessary to trigger an alert, and make the sensor data accessible via API, ensuring that these data are available to be transmitted in real time, via stream, to all connected users. Furthermore, the data is saved in the database, specifically in the raw data collection of the data lake, `Raw Data`. Once saved in the database, these raw data are available to be processed by the processing module.

The data provision by the `SensorsRepository` class occurs through the instance of the `SensorValue` class, which with the `update_current_sensor_value` method updates the data in memory that are accessed by the connected users.

The diagram showing the organization of these classes can be seen in figure 4.2. This design ensures that the raw data from the sensors are effectively received, evaluated, and stored.

In chapter 5, the implementation details of this module are further elaborated.

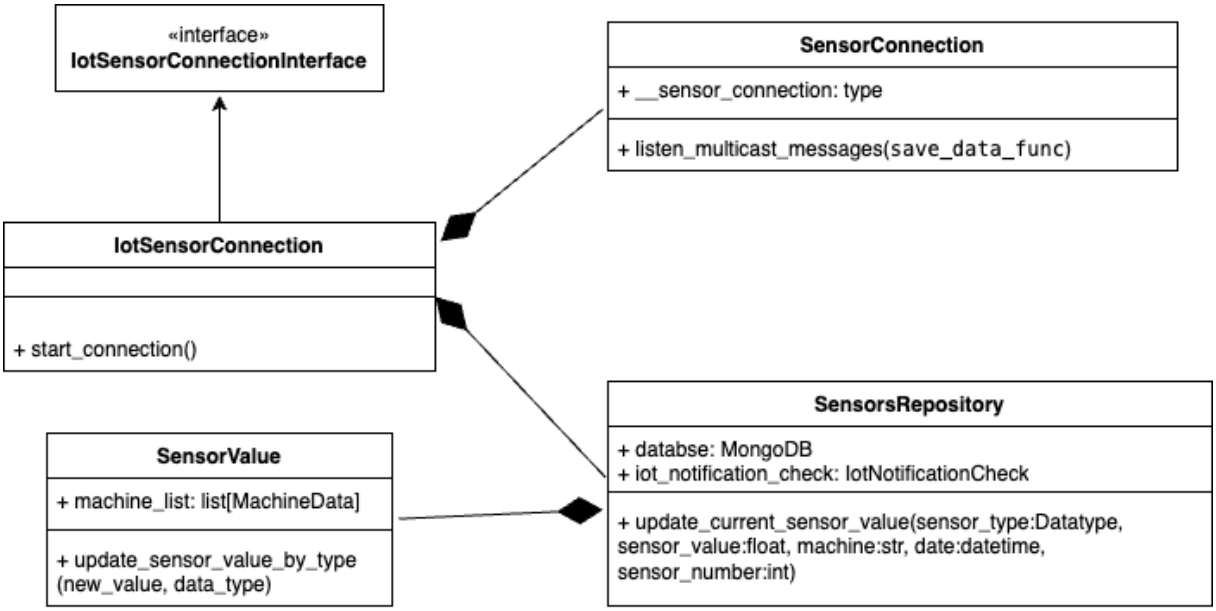

Figure 4.2: Module to receive sensor data.

4.1.2 Data Processing Module

The data processing module was developed to ensure that the raw data collected are processed, providing the statistical analysis that is displayed to the users.

The code is executed when a specific function is invoked, which is responsible for performing a series of operations. Firstly, a list is constituted containing the database collections responsible for storing both the raw data and the already processed data. Simultaneously, a second list is generated, representing the machines that sent information to the system.

With these lists, an iterative procedure begins, in which, for each identified machine, the available data are read, submitted to a statistical analysis process, after which the results obtained are recorded in the processed data collection. This statistical analysis adopts the Box Plot method.

The Box Plot, also known as a box diagram, is a graphical tool used to represent the variation of observed data from a numerical variable through quartiles. In figure 4.3, the rectangle formed by the first quartile (Q1), median, and third quartile (Q3) can be seen, which provide a notion about the centrality and dispersion of the data, while the

"antennas" extend to show the full range of the data, thus helping in the identification of possible outliers.

By adopting the Box Plot, the system ensures a robust understanding of the data distribution, identifying not only central trends but also variations and potential anomalies **marmolejo2010shifting**.

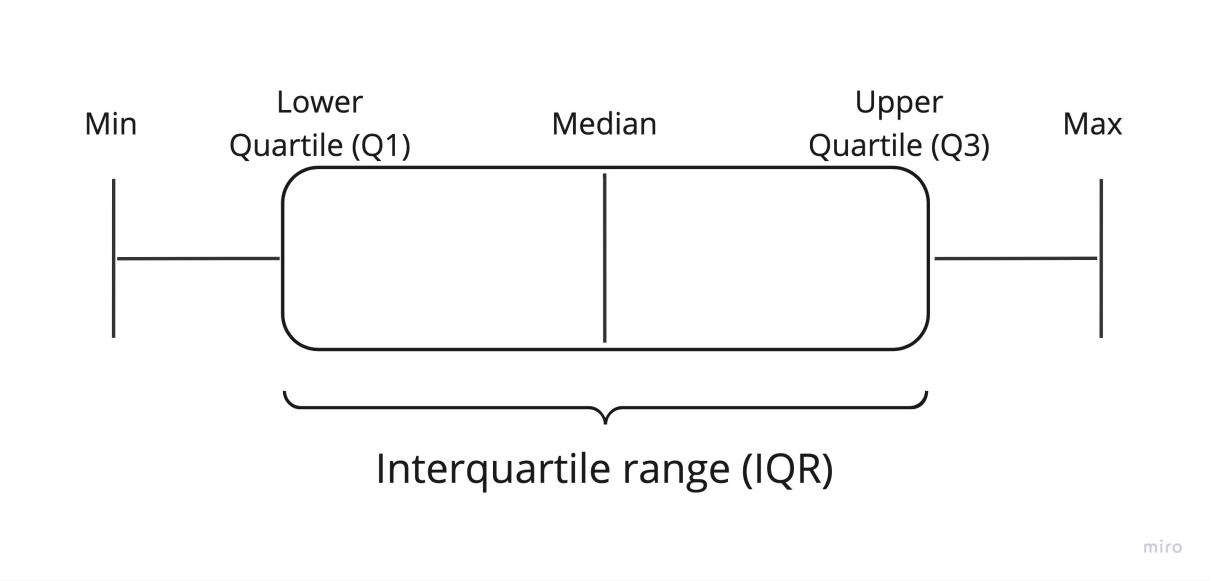

Figure 4.3: BoxPlot.

4.1.3 API

The API was structured into small sub-modules, each focused on a specific context. This modularization ensures that each part of the API has a single responsibility. In each module, there is a segmentation composed of: the *controller* layer, intended to receive and manage HTTP requests; the service layer, which serves to process the information and apply the respective business rules; and the repository layer, whose role is to establish a bridge with the database, accessing and providing the necessary data. Figure 4.4 represents the folder organization that was used for this architecture.

When a request is sent to the API, the first interaction occurs with the controller layer. Once received, this request is directed to the service layer, where business rules are

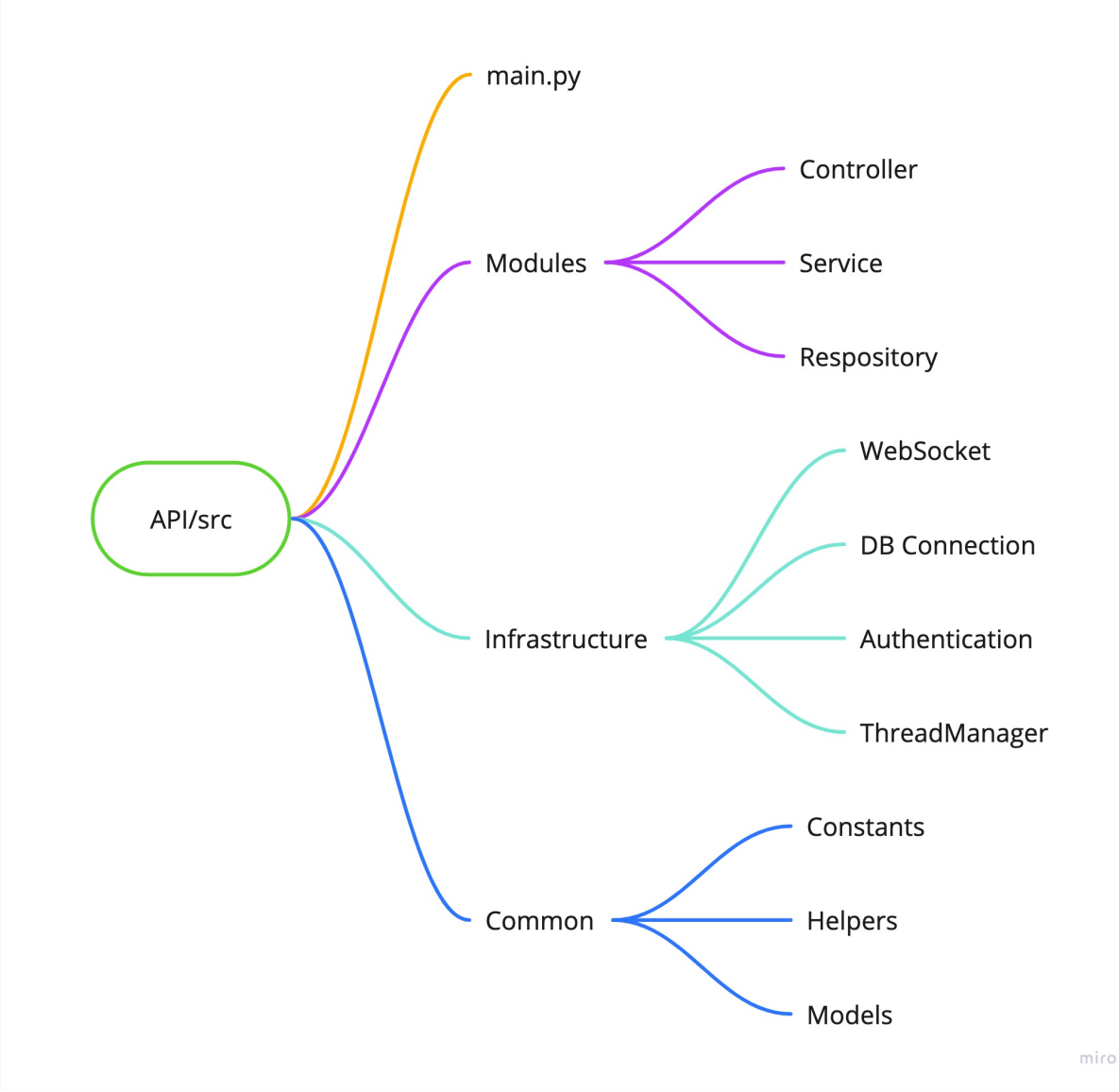

Figure 4.4: API Organization.

applied. The service layer communicates closely with the repository layer, which holds the responsibility of accessing the database and bringing accurate information, suitable for the demands of the module in question.

The modules implemented in the API with the described logic are:

- **Downtime:** Responsible for managing access to downtime data stored for testing in the system.
- **IOT Sensors:** Responsible for managing access to data related to the factory machine sensors.
- **Notification:** Responsible for managing access to notifications generated by the system, as well as web socket connections for sending notifications.
- **User:** Responsible for managing access to user data, as well as performing login and logout operations.

In addition to these modular layers, there is a specific area in the API for storing codes common to all modules. This section encompasses various useful functions, class templates, constant values, and default settings. These elements ensure greater cohesion and reduce code repetition, optimizing overall performance. Among the default settings, the initializer that establishes access to the database, authentication middleware, web socket connection for sending notifications, and the initializer of new *threads* deserve special mention. The latter is used for asynchronous operations that are executed in parallel to the operation of the API, such as those executed by the data reception module.

4.2 Frontend Architecture

Using *Next.js* **nextjsDocs** as a framework, the frontend follows a basic structure already established by it.

The system's routes reside in the `pages` folder, aligned with the framework's guidelines. The layouts that serve as a base for each page are located in the `layouts` folder.

React components **reactDocs** are the foundation of each page and layout and are organized in a specific layer, allowing them to be reused in various parts of the application.

With the adoption of *Typescript* **typescriptLang**, models define the types of data structure used. These are kept in the `types` folder, establishing data format contracts for the frontend. This minimizes errors and enhances efficiency in development.

The React *Context API* is used to manage data in components, allowing for centralized sharing of information, as can be seen in figure 4.5. This approach optimizes the way data is accessed and distributed in the system, enhancing the organization of the architecture.

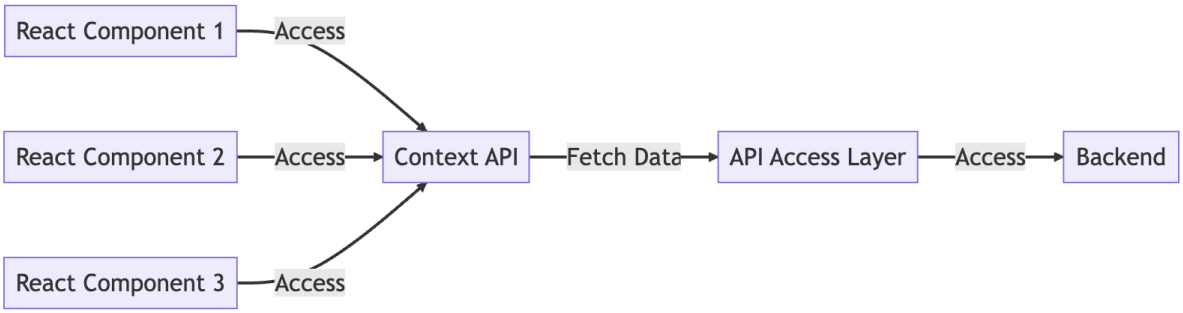

Figure 4.5: Frontend organization.

There is a specific layer for external access, which manages communication with the API and WebSocket connections. This is accessed only by contexts for updating and retrieving data.

Finally, there is a dedicated folder to store recurring codes, containing helper functions, themes, and assets, facilitating development and maintenance by providing a clear and cohesive structure.

4.3 Containers

Containers are technologies that allow applications to be isolated in specific environments with all their dependencies, libraries, and necessary configurations, without the overhead of full virtual machines. This ensures that the application works identically in different environments, from development to production **paraiso2016model**. In figure 4.6, it is

possible to visualize how containers work within the host operating system.

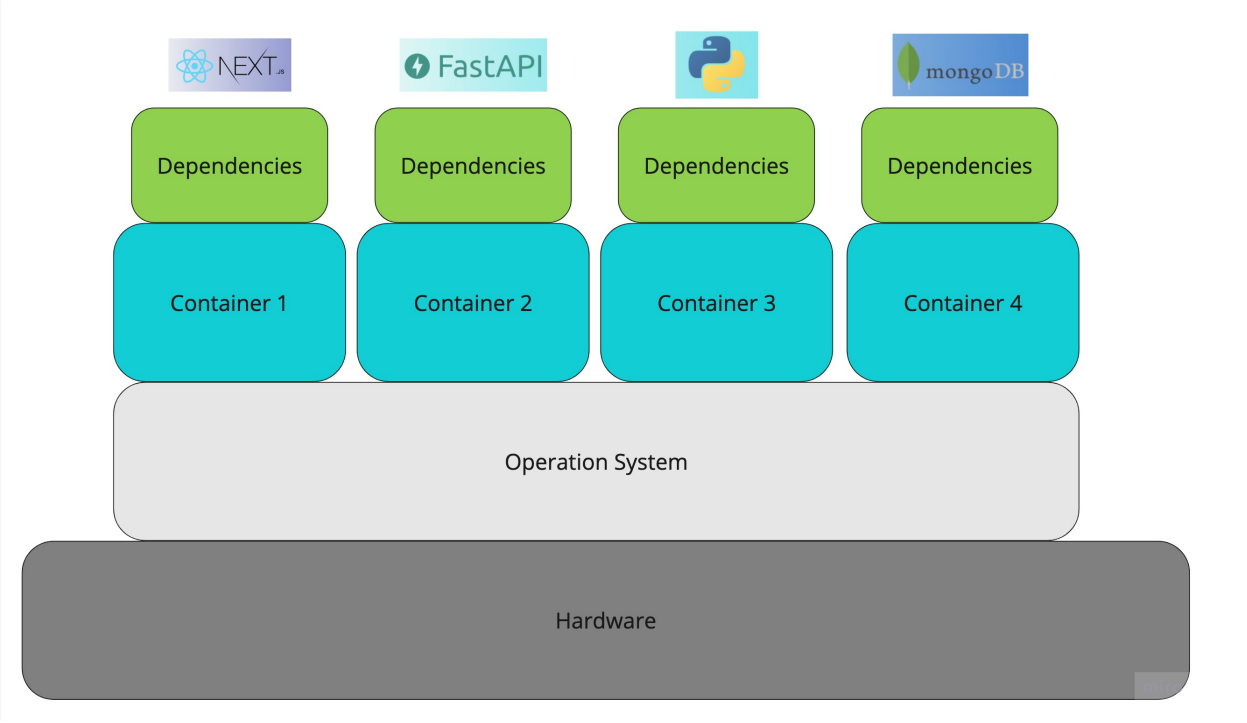

Figure 4.6: How container works.

Within the universe of containers, *Docker* **dockerDocs** was the tool selected for this project. Several factors influenced this decision, including comprehensive documentation, an active community, and the presence of a wide variety of available content. In addition, Docker simplifies the definition, creation, and execution of containers, making it a robust solution for application deployment.

The system adopts some containers to organize and manage the various parts of the application:

- **Frontend**: A container dedicated to the frontend, built with *NextJs*.
- **Backend**: Divided into two distinct containers:

  One that encompasses the API and the data receiving module;

  And another one specifically aimed at the data processing module;

- **Database**: A container for the MongoDB database, ensuring isolation and efficiency in data management.

The communication between the containers is made possible through a bridge network provided by Docker **dockerNetwork**. This network is a software interface created on the host that allows containers to communicate with each other and with the host, ensuring the necessary connectivity between the different modules of the application. This adds an extra layer of security to the application, as all external connections must be made through this network. The connection with the external network is made through a web server, explained in section 4.4.

To ensure data persistence and prevent the loss of vital information, the concept of Docker *volumes* **dockerVolumes** was employed in the system architecture. Volumes are designated spaces in the host system that can be accessed and used by containers. In the context of this project, a volume was specifically configured for the MongoDB database. Thus, even if the database container is restarted or removed, the database remains intact and available, due to its storage in the volume, which operates independently of the container's lifecycle.

With the need to manage multiple containers, network configurations, and volumes in a cohesive and simplified manner, *Docker Compose* **dockerCompose** was adopted in the system architecture. Docker Compose allows the definition and execution of multi-container applications using a YAML file **yamlOrg**. This file contains all the necessary configurations to initialize and interconnect the containers. Thus, instead of executing a series of commands to start each container individually, it is possible, through Docker Compose, to start the entire system with a single command. This approach not only simplifies the deployment and development process but also ensures that network and volume configurations are consistently applied in each execution.

The use of containers in the project brought advantages. Firstly, it ensured consistency between development and production environments. Additionally, the modularization provided by the containers facilitates the scalability and maintenance of the system, allowing updates and changes to be made quickly and safely as the system grows. Lastly,

the use of containers facilitates the portability of the system, which can be run on various types of servers and systems, provided that docker is installed.

4.4 Web Server

Within the proposed architecture, with containers running different parts of the application, *NGINX* **nginxDocs** was used to intermediate the traffic of requests, ensuring the correct distribution of requests to each container.

The method employed for this is the reverse proxy. In simple terms, the reverse proxy acts as an interface between the client and several servers, directing client requests to the appropriate server (in the context of this project, the containers), thus optimizing resource use and ensuring a faster and more efficient response.

Regarding specific requests, those that involve return in stream format or establish a *WebSocket* connection, specific settings were made in the *NGINX* configuration, these are detailed in chapter 5, dedicated to implementation. Upon receiving a request, the *NGINX* server identifies, based on it, which container is responsible for the service. After this identification, the appropriate settings are applied, and the request is directed to the corresponding container to obtain the response. This workflow can be seen in figure 4.7.

The incorporation of *NGINX* brought some benefits to the project. One of them is the additional layer of security: *NGINX* limits direct access to the containers, serving as a barrier against unauthorized access attempts. Additionally, with NGINX, the scalability process becomes simpler and more efficient, thanks to the server's inherent ability to act as a load balancer. This load balancer distributes incoming traffic among several servers, ensuring that no server becomes overloaded. This feature not only improves overall performance but also provides greater system availability, as in the event of a server failure, traffic can be directed to another operational one.

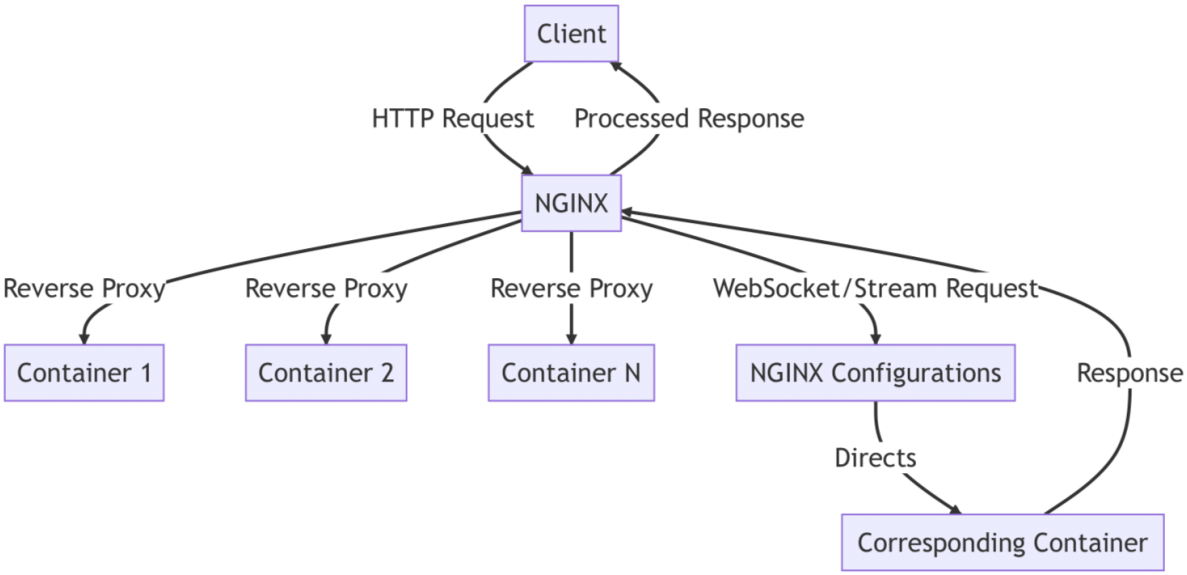

Figure 4.7: NGINX workflow.

Chapter 5

Implementation

After the chapter where the software architecture was detailed, this chapter is focused on explaining how such architecture was implemented, as while the first describes the structure and organization, this one focuses on the technical actions adopted to make this structure work.

For a more structured and detailed analysis, this chapter has been divided into specific sections for each system component. They are:

- **Database implementation**: This section will address the technical details of the database design, adopted schemas, and how information is stored and retrieved.
- **Implementation of the data receiving module**: This section will detail how data is received, validated, and processed before being stored and made available to users.
- **API Implementation**: Here, the structure of the API will be discussed, going through the provided endpoints, the logic behind each one, and the layers used.
- **Implementation of the data processing module**: The handling of data received from sensors is addressed, and how the statistical analysis and aggregation that generates the information presented in the charts is done.

- **Frontend Implementation**: Finally, the user interface will be discussed, explaining how data is structured and displayed on screen.

5.1 Database Implementation

Within the system implementation, MongoDB was used to store all system information. This non-relational database allowed for flexible data organization, facilitating the storage of different data that can be received by the data receiving module, and facilitating the creation of processing layers. The structuring of the databases and their respective collections was designed to facilitate both the insertion and the consultation of information.

Regarding data organization, the following databases were created:

- **Users**: Stores information related to users. It has collections that record login attempts, users' personal details, and tokens associated with them.

- **Notification**: Intended for system notifications. Currently, this database only contains notifications associated with machine alerts, generated by the data received from the sensors along with the stored parameters.

- **Downtime**: Stores two collections, one with the data read from the machine downtime spreadsheets, and another with this processed data. This database with these collections is only to simulate what the machine downtime data would look like if they were inserted into the system.

- **Raw Data**: This database is dedicated to storing raw data from different sensors. Each type of sensor, such as pressure sensors, has its own collection, ensuring a grouping of information that facilitates analysis.

- **Processed Data**: As the name suggests, it stores data that has already undergone a processing stage. Thus, interpreted data from different sensors are separated into specific collections, such as pressure in one and voltage in another.

- **Metadata**: Dedicated to storing system metadata. So far, the only collection present is "AlertParameter", which gathers parameters used to generate alerts associated with each sensor.

With this structuring, the aim is not only to logically organize the data, but also to optimize query operations and ensure simplified expansion as it becomes necessary to store new data in the system.

The implementation of database access is detailed in the implementation section of the API, in 5.4.2.

5.2 Implementation of the data receiving module

In the process of implementing the system, one of the essential steps was the development of a module intended for receiving data from IoT sensors. This reception is carried out via a multicast connection, an efficient approach to handle the transmission of messages to multiple recipients simultaneously.

This module is responsible for establishing the multicast connection to receive the data, performing the conversion of the received data according to the predefined protocol, making the data available to be displayed in real time for connected users, checking if it generates any type of alert (and if it does, notifying users about it by creating a notification), and saving the generated information in the database.

5.2.1 Connection and data reception

The `SensorConnection` class has the main responsibility of creating a socket, staying connected to receive messages and interpreting them. The structure and operation of this class are detailed below.

The `SensorConnection` class is initiated with the creation of an IPv4 and UDP socket:

```
class SensorConnection:
    def __init__(self):
```

```
self.sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
```

To ensure that the system is constantly listening to multicast messages from the sensors, the `listen_multicast_messages` method was defined within this class. It starts with the creation of the connection and initiates the message reading process, also managing possible disconnections and reestablishing the connection when necessary:

```
async def listen_multicast_messages(self, save_data_func):
    self.__create_connection()
    while True:
        await self.__start_read_messages(save_data_func)
        self.sock.close()
        time.sleep(1)
        self.__reconnect()
```

The function `__create_connection` is responsible for establishing and configuring the initial connection with the multicast group, and within the infinite loop, the reception of messages is initiated with the `__start_read_messages` method. When this method is finished, the socket connection is closed, and then reconnected to resume reading the messages. The call to the `time.sleep(1)` function is used to have a small interval between one call and another so as not to make a very large number of calls in case there is some kind of problem.

Below, each of the functions called within this method is detailed.

Create connection method

```
def __create_connection(self):
    self.sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)

    server_address = ('', SENSOR_MULTICAST_PORT)
    self.sock.bind(server_address)
```

```
multicast_group = SENSOR_MULTICAST
group = socket.inet_aton(multicast_group)
mreq = struct.pack('4sL', group, socket.INADDR_ANY)
self.sock.setsockopt(socket.IPPROTO_IP,
    socket.IP_ADD_MEMBERSHIP,
    mreq)
```

Initially, the socket is configured to allow multiple connections on a single address. The `SO_REUSEADDR` option is set to the value 1, allowing more than one socket to bind to the same address, which is especially useful in multicast connection contexts:

```
self.sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
```

After this, the socket is bound to a specific multicast address and port. It is important to note that the first argument in the server address definition is left blank. This approach ensures that the system is connecting to all available network interfaces, providing broad connection coverage:

```
server_address = ('', SENSOR_MULTICAST_PORT)
self.sock.bind(server_address)
```

Finally, to effectively join the multicast group, some steps are performed. The multicast IP address is first converted to binary format with the call to `socket.inet_aton`. Then, this address and the local address (represented by `socket.INADDR_ANY`) are packed into a data structure by `struct.pack`. This structure is used to specify to the socket that it should join a multicast group in `self.sock.setsockopt`. The `IP_ADD_MEMBERSHIP` option is set and the previously created structure is passed as an argument, concluding the connection to the multicast group:

```
multicast_group = SENSOR_MULTICAST
group = socket.inet_aton(multicast_group)
```

```
    mreq = struct.pack('4sL', group, socket.INADDR_ANY)
    self.sock.setsockopt(socket.IPPROTO_IP, socket.IP_ADD_MEMBERSHIP, mreq)
```

These operations ensure that the socket is configured and connected to the multicast group, ready to receive messages from multiple sources simultaneously.

Reconnect Method

```
def __reconnect(self):
    try:
        self.sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
        self.__create_connection()
    except Exception as e:
        print(f"Error to reconnect: {e}")
```

In situations where the connection to the sensors is interrupted, the `__reconnect` method is called to try to reestablish the connection, creating a new instance of the socket and calling the `__create_connection` function again, detailed earlier.

Method start read messages

```
async def __start_read_messages(self, save_data_func):
    while True:
        try:
            data, address = self.sock.recvfrom(1024)
            result = self.__parse_multicast_message(data)
            if not type(result) == str:
                await save_data_func(result)
        except Exception as e:
            print(f"Error: {e}")
            break
```

After the settings are made, the messages are continuously read and processed by the `__start_read_messages` function. During this process, each message is processed by the `__parse_multicast_message` method, and if it is in the correct format, it is passed to a function that will save and make it available for the API to stream to connected users.

If there is any problem in the execution of this method, it is terminated and returns to the `listen_multicast_messages`, where the socket is closed and a new connection is established by the `__reconnect` method.

Parse Multicast Messages Method

```
def __parse_multicast_message(self, data):
        (machine_type_high, machine_number_low) =
            self.__parse_bytes(data[:2])

        message_type = data[2]

        if message_type == 2:
            return "Request to publish..."

        (physical_quantity_high, sensor_number_low) =
            self.__parse_bytes(data[3:5])

        (data_type_high, meaning_low) =
            self.__parse_bytes(data[5:7])

        message_dict = {
            'Machine': {
                'Type': str(machine_type_high)+". "+MACHINE_TYPE[machine_type_high],
                'Number': machine_number_low
            },
```

```
            'Type': str(message_type)+". "+ MESSAGE_TYPE[message_type],
            'Sensor': {
                'PhysicalQuantity': PHYSICAL_QUANTITY[physical_quantity_high],
                'Number': sensor_number_low
            },
            'MeaningOfData': {
                'DataType': str(data_type_high)+". "+DATA_TYPE[data_type_high],
                'Meaning': str(meaning_low)+". "+DATA_MEANING[meaning_low]
            }
        }

        return message_dict
```

To interpret and extract information from the message received from the multicast, it is crucial to properly decode the message according to the protocol defined earlier. The implementation of this decoding is done by the `__parse_multicast_message` method. The helper function `__parse_bytes` is used for this task, given a sequence of bytes, the function interprets the bytes using the big-endian order (where the most significant bytes come first).

```
def __parse_bytes(self, bytes):
    data = int.from_bytes(bytes, byteorder='big')
    high_data = (data >> 8) & 0xFF
    low_data = data & 0xFF

    return (high_data,low_data)
```

Here, `data` contains the integer value of the provided bytes. The high-order byte is extracted by shifting the value 8 bits to the right and applying an "AND" operation (&), and the low-order byte is simply obtained by applying the "AND" operation with `0xFF`.

With the ability to interpret the bytes, the main function `__parse_multicast_message` can begin the decoding:

- First, it extracts the machine type and the machine number from the first two bytes of the message.
- The third byte of the message is then interpreted as the message type. If the message type is `2`, the function will directly return a request to publish.
- Bytes 4 and 5 are interpreted as the sensor ID, which contains the physical quantity being measured and the sensor number.
- Bytes 6 and 7 are used to extract the data type and its meaning.

The extracted information is then organized into a dictionary for clear representation and easy access to the components individually:

```
message_dict = {
    'Machine': {
        ...
    },
    'Type': ...,
    'Sensor': {
        ...
    },
    'MeaningOfData': {
        ...
    }
}
```

This structure allows a clear and modular representation of the decoded message, making it easy to integrate and use in other parts of the system. Therefore, the return of the `__parse_multicast_message` method is used as the result of the interpretation of the multicast message, and sent to the function received as a parameter, `save_data_func`.

5.2.2 Verification and provision of data

In the data reception process, after opening the connection and the data, it is necessary to check if they are in the correct format, if it generates any alert, insert it into the database and make it available to the users connected to the system.

The `IotSensorConnection` class, which implements the `IotSensorConnectionInterface` interface, plays a key role in this module. Upon its initialization, a connection with the repository is established through the `self.__repository` variable. In addition, it is responsible for the connection with the sensor, which is established through the `self.__sensor_connection`, explained earlier in section 5.2.1.

```
class IotSensorConnection(IotSensorConnectionInterface):
    def __init__(self, respository:SensorsRepository):
        self.__repository = respository
        self.__sensor_connection = SensorConnection()

    def start_connection(self):
        threadManager = ThreadManager()
        threadManager.start_async_thread(self.__start_connection)

    async def __start_connection(self):
        await self.__sensor_connection.
            listen_multicast_messages(self.__handle_iot_data)
```

Upon initiating the connection, using the `start_connection` method, a new thread is created through the `ThreadManager` class, explained in 5.4.2. This thread invokes the `listen_multicast_messages` method from the `SensorConnection` class that was detailed in section **??**. It is necessary to create a new thread because this module is together with the API, and it is necessary for both processes to function at the same time, a new thread was necessary for the parallel operation of both.

Data Format Verification

To handle the received data, the `__handle_iot_data` method is passed as an argument to `listen_multicast_messages` (as the save_func argument that exists in the `SensorConnection` class).

```
async def __handle_iot_data(self, sensor_data:dict):
    sensor_model = self.__parse_sensor_data_to_sensor_model(sensor_data)
    await self.__repository.update_current_sensor_value(
        sensor_value = sensor_model.value,
        machine = sensor_model.machine,
        date = sensor_model.date,
        sensor_type = sensor_model.type,
        sensor_number = sensor_model.sensor_number
    )

def __parse_sensor_data_to_sensor_model(self, sensor_data:dict):
    value = sensor_data["value"]
    machine = str(sensor_data["Machine"]['Number'])
        + sensor_data["Machine"]['Type']
    date = datetime.now()
    data_type = sensor_data["Sensor"]["PhysicalQuantity"]
    sensor_number = sensor_data["Sensor"]["Number"]
    return ConnectionModelToParse(
        date=date,
        machine=machine,
        sensor_number=sensor_number,
        type=data_type,
        value=value
    )
```

This method is responsible for receiving the sensor data and converting it into a model class, named `ConnectionModelToParse`, which uses `Pydantic` to validate the information. The explanation of `Pydantic` is given in section 5.4.3.

```
from datetime import datetime
class ConnectionModelToParse:
    def __init__(self,value:float,machine:str,
        date:datetime,type:Datatype,
        sensor_number:int):

        self.value = value
        self.machine = machine
        self.date = date
        self.type = type
        self.sensor_number = sensor_number
```

After this transformation, the data is forwarded to the repository. The `update_current_sensor_value` method from the repository is called to check if the received data triggers any type of alert, save it in the database, update the data in memory, and perform notification checks.

```
class SensorsRepository:
    def __init__(self):
        self.database = MongoDBIOT()
        self.iot_notification_check = IotNotificationCheck()
        self.__sensor_value = SensorValue()

    async def update_current_sensor_value(self, sensor_type:Datatype,
        sensor_value:float, machine:str, date:datetime,
        sensor_number:int):
        alert_type = await self.__get_alert_type(sensor_value, sensor_type)
```

```
current_value = {"machine":machine,
    "value":sensor_value, "timestamp": date,
    "alert_type":alert_type.value,
    "sensor_number":sensor_number}
result = await self.insert_value_into_database(current_value,
    sensor_type)
new_id = result.inserted_id
iot_data = IotData(
    alert_type=current_value["alert_type"],
    machine=current_value["machine"],
    timestamp=current_value["timestamp"],
    value=current_value["value"],
    id=PyObjectId(new_id),
    datatype=sensor_type,
    sensor_number=sensor_number
)

self.__sensor_value.update_sensor_value_by_type(
    iot_data,sensor_type)

await self.iot_notification_check.check_iot_notification(
    iot_data)
```

Alert Verification

Within the `update_current_sensor_value` method, the type of alert generated is first verified with the `__get_alert_type` method. This method reads the parameter according to the sensor type within the system metadata, where access is explained in 5.4.1, and with it checks the alert status.

The alert status, defined by the `get_alert_status` function, returns as `OK` if the

sensor value is less than 90% of the value defined as a parameter, returns as `WARNING` if this value is between 90% and 100%, and returns as `PROBLEM` if the value returned by the sensor is greater than 100% of the value defined as a parameter.

```
def get_alert_status(self,sensor_value:int,
    alert_parameter:int)->AlertTypes:

    parameter = ((sensor_value/alert_parameter)*100)
    if parameter < 90:
        return AlertTypes.OK
    if parameter >= 90 and parameter < 100:
        return AlertTypes.WARNING
    if parameter >= 100:
        return AlertTypes.PROBLEM

async def __get_alert_type(self, sensor_value:float,
    sensor_type:Datatype)->AlertTypes:
    alert_parameter = await MetadataRepository()
        .get_sensor_alert_value(sensor_type)
    alert_type = self.get_alert_status(sensor_value,alert_parameter)
    return alert_type
```

Database Registration

With the verification of the alerts, all information has been generated, so it can now be registered in the database. The `insert_value_into_database` method is used for this registration.

```
async def insert_value_into_database(self, value:BaseIotData, type:Datatype):
    try:
        collection = sensor_name_to_raw_data_collection(type)
```

```
        return await self.database.insert_one(IOT_DATABASE,collection,value)
    except Exception as ex:
        print(ex)
        raise ex
```

This method uses the base class of the database with already defined operations to perform the registration. Within the `update_current_sensor_value` method of the repository, the return is used to keep the registered ID in memory, important for creating the `IotData` object, which is sent to the connected users, via stream, in the next step.

```
current_value = {"machine":machine,
    "value":sensor_value,
    "timestamp": date,
    "alert_type":alert_type.value,
    "sensor_number":sensor_number}
result = await self.insert_value_into_database(current_value, sensor_type)
new_id = result.inserted_id
iot_data = IotData(
    alert_type=current_value["alert_type"],
    machine=current_value["machine"],
    timestamp=current_value["timestamp"],
    value=current_value["value"],
    id=PyObjectId(new_id),
    datatype=sensor_type,
    sensor_number=sensor_number
)
```

An important piece of information to highlight is that the name of the collection used by the `insert_value_into_database` method is defined according to the established data type, using the helper function `sensor_name_to_raw_data_collection`, explained in 5.4.3.

Data update in memory

With the alert type defined and the data registered in the database, the `SensorValue` class is used to update the information in memory. This process is done through the call `__sensor_value.update_sensor_value_by_type (iot_data,sensor_type)` in the `update_current_sensor_value` method of the repository.

The `SensorValue` class is responsible for managing and updating the values in memory. It is noted that it uses the `Singleton` design pattern, ensuring the existence of only one instance of this class throughout the application's lifecycle, guaranteeing that there is only one instance storing the sensor information.

```
class SensorValue(metaclass=Singleton):
    def __init__(self) -> None:
        self.machine_list:list[MachineData] = []

    def update_sensor_value_by_type(self, new_value: IotData, data_type: Datatype):
        is_new_machine = True

        for machine in self.machine_list:
            if machine.name == new_value.machine:
                is_new_machine = False
                is_new_sensor = True

                for index, sensor in enumerate(machine.sensor_data):
                    if sensor.datatype == data_type:
                        machine.sensor_data[index] = new_value
                        is_new_sensor = False
                        break

                if is_new_sensor:
```

```
                machine.sensor_data.append(new_value)
                break

        if is_new_machine:
            new_machine = MachineData(name=new_value.machine,sensor_data=[new_value])
            self.machine_list.append(new_machine)
```

At the time of its initialization, the `SensorValue` class initializes an empty list, `machine_list`, which will be responsible for storing the sensor values organized by machine.

The update occurs through the `update_sensor_value_by_type` method. This method updates the sensor value in memory according to its type (`data_type`). The update process first checks if the machine associated with the sensor already exists in the list. If so, it searches for the specific sensor within the machine's data and updates its value. If the sensor is not found, a new one is added to the list of sensors of the corresponding machine.

On the other hand, if the machine is not found in the `machine_list`, a new instance of `MachineData` is created and added to the list, containing the machine's information and the received sensor data.

```
class MachineData(BaseModel):
    name:str = Field(...)
    sensor_data:list[IotData] = Field([])
```

In this way, the repository sends the information to this method, and with the appropriate verification, the most updated data is kept in memory, and available to be used by the API, enabling real-time access to sensor data.

Notification Verification

With the alert type verified, the information saved in the database, and the `IotData` object assembled, the last task of the `update_current_sensor_value` method in the

repository is to use the IotNotificationCheck singleton to verify the notifications regarding the operation of the machines.

The `IotNotificationCheck` class acts as an alert controller for IoT data. Upon receiving IoT data, it checks the alert status and takes appropriate measures, whether adding or removing machines or sensors from the alert list. This class is essential for monitoring and responding to real-time alert events, ensuring that associated users are notified of any abnormalities or important events detected by the IoT sensors.

Through the `check_iot_notification` method, the class verifies the type of alert received by the IotData object, whether the machine is in an alert state, and whether the machine's specific sensor is in an alert state. Based on this verification, the method takes one of the following actions:

1. Puts a new machine in an alert state.
2. Puts a new sensor of the machine in an alert state.
3. Removes a sensor from the machine from the alert state. If the machine has only a single sensor in an alert state, the machine is removed from the alert state.

```
async def check_iot_notification(self, iot_data:IotData):
    is_alert_value = self.__is_alert_type_a_new_alert(
        iot_data.alert_type)
    machine_in_alert = self.__is_machine_in_alert_state(
        machine_name=iot_data.machine)
    machine_sensor_in_alert = self.__is_machine_sensor_in_alert_state(
        machine_in_alert,
        iot_data.datatype)

    is_machine_in_alert = machine_in_alert!=None

    if is_alert_value and is_machine_in_alert and (not machine_sensor_in_alert):
```

```
        await self.__put_new_machine_sensor_in_alert_state(
            machine_in_alert,
            iot_data.datatype)

    if is_alert_value and (not is_machine_in_alert):
        await self.__put_new_machine_in_alert_state(
            iot_data.machine,
            iot_data.datatype,
            iot_data.timestamp,
            iot_data.alert_type)

    if (not is_alert_value) and is_machine_in_alert and machine_sensor_in_alert:
        await self.__remove_machine_sensor_from_alert_state(
            machine_in_alert,
            iot_data.datatype,
            iot_data.timestamp)
```

The method `__put_new_machine_sensor_in_alert_state` is a private asynchronous method that is responsible for adding a new sensor to the alert state for a specific machine. It receives two parameters: `machine_in_alert`, which is an instance of the `MachinesSensorAlert` class representing the machine in question, and `sensor_type`, which is an instance of the `Datatype` type, shown in 5.4.3, representing the type of sensor that should be put on alert.

```
class MachinesSensorAlert(BaseModel):
    id: PyObjectId = Field(default_factory=PyObjectId, alias="_id")
    machine:str = Field(...)
    sensors:list[str] = Field([])
    alert_type:str = Field(...)
    start_time:datetime = Field(...)
```

```
    sensors_historical:list[str] = Field([])
    is_in_alert:bool = Field(True)

    end_time:Optional[datetime|None] = Field(None)
    read_by:Optional[list[str]] = Field([])
```

It is important to highlight that within this instance that is kept in memory, the `read_by` attribute is not filled. This happens because this attribute is used to control the users who marked the notification as read, and thus identify notifications read by the user. Therefore, this attribute is filled only in the database, by the notifications module of the API, shown in 5.4.4.

The first step performed by this method is to identify the position (or index) of the machine within the `machines_alert` list using the `index` method. Once the index is obtained, the sensor type is added to the machine's alert state sensor list, represented by the `sensors` attribute. In addition, this sensor is also added to the machine's alert state sensor history, indicated by the `sensors_historical` attribute. Finally, the updated machine (with the new sensor added to its alert and history lists) is reinserted into the main `machines_alert` list at the same position identified earlier.

This method ensures that whenever a new sensor enters an alert state for a machine that already had a sensor in alert, the relevant information is properly updated and kept in memory, allowing real-time monitoring of the alert conditions of all monitored machines.

```
async def __put_new_machine_sensor_in_alert_state(
        self,
        machine_in_alert: MachinesSensorAlert,
        sensor_type:Datatype):
    index = self.machines_alert.index(machine_in_alert)
    machine_in_alert.sensors.append(sensor_type.value)
    machine_in_alert.sensors_historical.append(sensor_type.value)
    self.machines_alert[index] = machine_in_alert
```

The `__put_new_machine_in_alert_state` method is a private asynchronous method whose main function is to create and register a new alert state for a specific machine. This method is invoked when a machine enters an alert state for the first time, which means it is not yet present in the `machines_alert` list of the class.

It receives four parameters: `machine_name`, which is a string representing the machine's name; `sensor_type`, which is an instance of the `Datatype` type, shown in 5.4.3, denoting the type of sensor that triggered the alert; `start_time`, an instance of `datetime` indicating the start of the alert; and `alert_type`, which is a string representing the type of alert.

Initially, the method creates a new instance of the `MachinesSensorAlert` class. This new instance represents the machine's alert state. The instance is initialized with the machine's name, the type of sensor that triggered the alert, a timestamp of the alert's start, and the type of alert. In addition, the machine is marked as being in an alert state through the `is_in_alert` attribute, which is set to `True`.

Finally, the machine's new alert state, represented by the newly created `MachinesSensorAlert` instance, is added to the `machines_alert` list.

```
async def __put_new_machine_in_alert_state(self,
    machine_name:str,
    sensor_type:Datatype,
    start_time:datetime,
    alert_type:str):

    new_machine_alert = MachinesSensorAlert(
        machine=machine_name,
        sensors=[sensor_type.value],
        sensors_historical=[sensor_type.value],
        is_in_alert=True,
        start_time=start_time,
```

```
        alert_type=alert_type)

    self.machines_alert.append(new_machine_alert)
```

The method `__remove_machine_sensor_from_alert_state` is a private asynchronous function designed to remove a specific sensor from a machine's alert state. It takes three parameters: `machine_in_alert`, which is an instance of the `MachinesSensorAlert` class representing the machine in question; `sensor_to_remove`, which is of the `Datatype` type 5.4.3, and identifies the sensor to be removed; and `end_time`, an instance of `datetime` that indicates the moment when the sensor was removed from the alert state. Within this method, initially, the positions of the sensor and the machine are identified in the appropriate lists. The sensor is then removed from the machine's list of sensors in alert state. If, after removal, the machine no longer has sensors in alert state, it will be removed from the alert state, by calling the method `__remove_machine_from_alert` otherwise, only the sensor's state is updated, by calling another method, `__remove_sensor_from_alert_state`.

```
async def __remove_machine_sensor_from_alert_state(self,
    machine_in_alert: MachinesSensorAlert,
    sensor_to_remove:Datatype,
    end_time:datetime):
    index_of_machine = self.machines_alert.index(machine_in_alert)
    index_of_sensor = machine_in_alert.sensors.index(sensor_to_remove.value)

    machine_in_alert.sensors.pop(index_of_sensor)

    if len(machine_in_alert.sensors) == 0:
    await self.__remove_machine_from_alert(index_of_machine, end_time)
    else:
    await self.__remove_sensor_from_alert_state(index_of_machine,machine_in_alert)
```

The `__remove_machine_from_alert` method is another private asynchronous function, which is responsible for completely removing a machine from the alert state. It accepts two parameters: `index_of_machine`, the index of the machine in question in the list, and `end_time`, the time when the machine was removed from the alert. Within this method, the machine is first marked as not being on alert and then it is removed from the `machines_alert` list. The machine is then stored in the database with a record of its final state and the end time. Finally, a notification is sent through a websocket to inform the user interface about the change in the machine's state. The details of how the notification is sent are explained in 5.4.2.

```
async def __remove_machine_from_alert(self,
    index_of_machine:int,
    end_time:datetime):
    machine_in_alert = self.machines_alert[index_of_machine]
    machine_in_alert.is_in_alert = False
    machineNotification = self.machines_alert.pop(index_of_machine)
    machineNotification.end_time = end_time
    await self.iot_database.insert_one(
    NOTIFICATION_DATABASE,
    IOT_MACHINE_ALERTS,
    machineNotification.to_bson())
    await self.websocket.send_notification(machineNotification)
```

The method `__remove_sensor_from_alert_state` is a simple asynchronous function that updates the state of a machine's sensor in the alert list. It receives two parameters: `index_of_machine`, which is the index of the machine in the `machines_alert` list, and `machine_alert_updated`, which is the updated instance of the machine in alert. Essentially, this method replaces the existing machine in the list with the updated object provided as a parameter by the `__remove_machine_sensor_from_alert_state` method.

```
async def __remove_sensor_from_alert_state(self,
```

```
    index_of_machine:int,
    machine_alert_updated:MachinesSensorAlert):
    self.machines_alert[index_of_machine] = machine_alert_updated
```

5.3 Implementation of the data processing module

As explained in 4.1.2, the data processing module reads the raw data from the system, applies the `boxplot` calculation, and stores the result in the database.

5.3.1 Scheduling for periodic execution

The data processing needs to occur periodically, in this case it was initially defined once a day. To execute the processing function call once a day, the `schedule` library was used **scheduleDocs**. With this library, the execution of the data aggregation function was scheduled for every day at midnight. An infinite loop was created to keep the code running, checking whether the function should be executed or not.

```
import schedule
schedule.every().day.at("00:00").do(aggregation_init)
print(datetime.now(), flush=True)
while True:
    schedule.run_pending()
    time.sleep(1)
```

5.3.2 Identifying the origin of the data

Within this structure, it is necessary to identify the correct collections from which the data should be retrieved before processing. This identification begins with the function `get_tuples_with_raw_data_collections_and_processed_collections()`. This function, as the name suggests, is responsible for retrieving tuples relating the raw data collections with their respective processed collections. It iterates over all types of sensors,

represented by the enumerator `Datatype`, explained in 5.4.3, and for each type of sensor, the respective raw and processed data collections are identified, resulting in a list of tuples.

It is important to highlight that the names of the collections are retrieved by the helper functions, explained in 5.4.3.

```
def get_tuples_with_raw_data_collections_and_processed_collections():
    result:list[tuple] = []
    for sensor_type in Datatype:
        raw_collection = sensor_name_to_raw_data_collection(
        sensor_type)
        processed_collection = sensor_name_to_processed_collection(
        sensor_type)
        result.append((raw_collection,processed_collection))
    return result
```

To initiate the data processing, the function `aggregation_init()` is called, first obtaining the list of tuples that relate the raw data collections with the processed ones. After retrieving this list, it initializes an asynchronous loop, whose aim is to execute an aggregation function until its completion. This asynchronous design is necessary to ensure that the processing can make asynchronous function calls, given that this module is separate from the API.

```
def aggregation_init():
    tuples_list =
    get_tuples_with_raw_data_collections_and_processed_collections()
    loop = asyncio.new_event_loop()
    loop.run_until_complete(aggregation(tuples_list))
    loop.close()
```

In this way, the origin of the data is identified, and where they should be inserted after

being processed. This information is passed to the aggregation function so that it can be executed for any stored data.

5.3.3 Starting the aggregation

Once the data source is defined through the identified collections, the data aggregation phase is initiated. The function responsible for this task is `aggregation()`, which accepts a list of tuples representing the sensor collections.

```
async def aggregation(sensors_collection_list:list[tuple]):
    database = BaseDB()
    for collection_tuple in sensors_collection_list:
        (raw_data_collection, processed_data_collection) = collection_tuple
        machine_list = await database.read_machines_list(raw_data_collection)

        for machine in machine_list:
            await aggregate_data(
                database,
                raw_data_collection,
                processed_data_collection,
                machine)
```

Within this function, firstly, a database instance is initialized using the `BaseDB()` class, explained in 5.4.2. Then, the function iterates over each tuple in the provided list. For each tuple, the raw and processed data collections are extracted. Using the raw data collection as a reference, a reading of the list of machines associated with this collection is made through the `read_machines_list()` method.

```
async def read_machines_list(self, collection:str):
    temp_client = self.client
    return await temp_client[IOT_DATABASE][collection].distinct('machine')
```

For each identified machine, the data is then aggregated. The function `aggregate_data()` is called, passing the database, the raw data collection, the processed data collection, and the specific machine in question as arguments. This function, in turn, is responsible for effectively aggregating the machine's data, transforming raw data into processed data that will be stored in the respective processed data collection.

Searching for data to be aggregated

Initially, a `query` is generated using the `get_aggregation_query()` function, which uses the information from the aggregated collection and the machine in question. With this `query`, the raw data is then read from the raw data collection using the `read_raw_data()` method.

The `get_aggregation_query()` function is responsible for generating the query that searches for the information to be aggregated by the processing module. Its goal is that only the raw data not yet processed are considered for aggregation, optimizing the process and avoiding unnecessary reprocessing.

This function requires an instance of the database, the name of the collection where the aggregated data is stored, and the specific machine for which the aggregation is needed.

```
async def get_aggregation_query(
    database:BaseDB,
    collection:str,
    machine:str)->dict:
    field_to_aggregate = "more_recent_register"
    more_recent_processed_data:BoxPlotData|None =
    await database.read_more_recent_data(
        collection,
        machine,
        field_to_aggregate)
```

```
    if more_recent_processed_data is None:
        return __build_query_with_limit_of_data(machine)
    else:
        return __build_query_with_range_of_data(
        more_recent_processed_data,
        machine,
        field_to_aggregate)
```

The construction of the *query* uses two important constants. `MAX_VALUE_BY_PERIOD` stores the maximum number of records that can be read for aggregation, which in this case is the equivalent amount to 24 hours of reading considering the arrival of data every second, or 86400 records. Meanwhile, `AGGREGATION_PERIOD_IN_HOURS` stores the number of hours between one aggregation and another, in this case 24 hours, being consistent with the previous constant.

Initially, the field `more_recent_register` is set as the attribute to be searched for. The function `read_more_recente_data()` is then called to obtain the most recent processed data for the machine and collection in question.

```
async def read_more_recente_data(self,
    collection:str,
    machine:str,
    date_time_field:str):
    try:
        temp_client = self.client
        cursor = temp_client[IOT_PROCESSED_DATA][collection]
            .find({"machine":machine})
            .sort([(date_time_field,pymongo.DESCENDING)])
        result:list = await cursor.to_list(None)
        return result[0] if len(result)!= 0 else None
    except Exception as ex:
```

```
        print(ex)
        raise ex
```

If no recently processed data is found, the *query* is built using the `__build_query_with_limit_of_data` function. This function simply limits the amount of data retrieved to `MAX_VALUE_BY_PERIOD` and searches for records that match the specified machine.

```
def __build_query_with_limit_of_data(machine:str)->dict:
    return {
        "limit":MAX_VALUE_BY_PERIOD,
        "query":{"machine":machine}
    }
```

However, if recent processed data is found, which is expected, the function to be used is `__build_query_with_range_of_data()`. This function considers the most recent processed record and calculates a time range (`date_limit_to_process_data`) by adding the aggregation period, defined by `AGGREGATION_PERIOD_IN_HOURS`, to the date of this most recent record. The generated query searches for records with timestamps within this time range and that match the specified machine, with a maximum limit of records defined by `MAX_VALUE_BY_PERIOD`.

```
def __build_query_with_range_of_data(more_recent_processed_data:BoxPlotData,
machine:str,
field_to_aggregate:str)->dict:
    date_of_more_recent:datetime =
    more_recent_processed_data[field_to_aggregate]

    date_limit_to_process_data = date_of_more_recent +
    timedelta(hours = AGGREGATION_PERIOD_IN_HOURS)

    return {
```

```
        "query":{
            "timestamp": {
                "$gt": date_of_more_recent,
                "$lte": date_limit_to_process_data
            },
            "machine":machine
        },
        "limit":MAX_VALUE_BY_PERIOD
    }
```

BoxPlot Calculation

With the query assembled, the data is retrieved using the `read_raw_data` function.

```
async def read_raw_data(self, collection:str, query:dict):
    try:
        temp_client = self.client
        cursor = temp_client[IOT_DATABASE][collection].find(
            query["query"])
            .sort([("timestamp",pymongo.ASCENDING)])
            .limit(query["limit"])
        return await cursor.to_list(None)
    except Exception as ex:
        print(ex)
        raise ex
```

The amount of data retrieved is calculated and, if this amount exceeds a predefined minimum value `MINIMUM_DATA_TO_AGGREGATE`, the aggregation proceeds. If the minimum amount is not reached, the recursive function is terminated, concluding the processing of the data from that collection.

`MINIMUM_DATA_TO_AGGREGATE` is a constant value defined as 100, which ensures that there are enough data to be aggregated, avoiding the aggregation of a small amount of data, which could compromise the analysis.

After the query search, a `logger` object is initialized to keep records of the aggregation process.

In this implementation, the log is used only to display information on the console, but a future implementation may add a more comprehensive form of logs, as detailed in 8.3.2.

```
class Logger(metaclass=Singleton):
    async def store_aggregation_log(self,
        box_plot_data:BoxPlotData,
        collection:str):
        print("++++++++++++++++++++++++++++++")
        print("Collection {}".format(collection))
        print("more_recent_register {}".format(box_plot_data.more_recent_register))
        print("median {}".format(box_plot_data.median))
        print("mean {}".format(box_plot_data.mean))
        print("q1 {}".format(box_plot_data.q1))
        print("q3 {}".format(box_plot_data.q3))
        print("lower_quartile {}".format(box_plot_data.lower_quartile))
        print("upper_quartile {}".format(box_plot_data.upper_quartile))
        print("mean_with_selection {}".format(box_plot_data.mean_with_selection))
        print("amount_of_data {}".format(box_plot_data.amount_of_data))
        print("++++++++++++++++++++++++++++++")

    async def not_aggregated_data(self, amount:int, collection:str):
        print("============================")
        print("")
        print("Amount data not aggregated {} - {}".format(str(amount),collection))
```

```
        print("============================")
```

The raw data read from the database is converted into a DataFrame, from the `pandas` library **pandasDocs** (a python language library used for data manipulation), after which the relevant aggregated data is calculated using the `calc_box_plot()` function. This function returns the data in a structured form suitable for graphical representations, such as a box plot.

To perform the calculation, various functions from the pandas library are used, such as `median`, `quartile`, `mean`, and `shape`, which facilitate the understanding and execution of the calculation.

```
def calc_box_plot(df:pd.DataFrame, machine:str):
    values = df["value"]

    median = values.median()
    mean = values.mean()

    Q1 = values.quantile(.25)
    Q3 = values.quantile(.75)

    IIQ = Q3 - Q1

    lower_quartile = Q1 - 1.5 * IIQ
    upper_quartile = Q3 + 1.5 * IIQ

    selection = (df["value"]>=lower_quartile) & (df["value"]<=upper_quartile)

    values_selected = values[selection]

    mean_with_selection = values_selected.mean()
```

```
df['timestamp'] = pd.to_datetime(df['timestamp'])
date_of_more_recent:datetime|str = df['timestamp'].max()

amount_of_data = df.shape[0]

box_plot = BoxPlotData()

box_plot.more_recent_register:datetime = date_of_more_recent

box_plot.lower_quartile=lower_quartile
box_plot.upper_quartile=upper_quartile
box_plot.median=median
box_plot.mean=mean
box_plot.mean_with_selection=mean_with_selection
box_plot.q1=Q1
box_plot.q3=Q3
box_plot.amount_of_data=amount_of_data
box_plot.machine=machine

return box_plot
```

In this function, the received dataframe contains a series of values that will be used to calculate the components of the Box Plot. First, the median and mean values of the data are determined. The quartiles Q1 (first quartile) and Q3 (third quartile) are calculated using the `quantile()` function from the pandas library. From these quartiles, the Interquartile Range (IQR) is determined as the difference between Q3 and Q1.

To identify the outlier values, the lower and upper limits are calculated. The lower limit is obtained by subtracting $1.5 \times$ `IQR` from Q1 and the upper limit is obtained by

adding 1.5 × `IQR` to Q3. Subsequently, a selection of values that are between the lower and upper limits is made. The mean of these selected values is then calculated, resulting in `mean_with_selection`.

The function also takes care of converting the `timestamp` column to datetime type and identifying the most recent *timestamp*, which will be crucial for assembling searches in the following aggregations.

With all the calculated values, a `BoxPlotData` object is instantiated and populated with the Box Plot components, along with additional information, such as the total number of data and the corresponding machine.

Recording of processed data

After the entire process described, the data is converted into JSON format and inserted into the collection of aggregated data by the `insert_processed_data` function.

```
async def insert_processed_data(self, collection:str, data):
    try:
        temp_client = self.client
        await temp_client[IOT_PROCESSED_DATA][collection]
            .insert_one(data)
    except Exception as ex:
        print(ex)
        raise ex
```

After a successful insertion, the function `aggregate_data()` is recursively called, ensuring that all relevant raw data are aggregated.

However, if the amount of raw data does not reach the minimum limit, the function records this occurrence using the `not_aggregated_data()` method, indicating that the data were not aggregated due to their lack, and ends the recursion.

5.4 API Implementation

As explained in the section about the architecture 4.1.3, the API has a division by modules, and each module follows a predefined structure, with a controller layer, responsible for receiving HTTP requests, a service layer, responsible for handling data and business rules, and a repository layer, responsible for managing access to the database of that module.

In addition, the API also has parts that are common to all modules. The infrastructure that has the function of providing a database access interface, a web socket message sending access interface, authentication means, and the thread manager. The common codes, which store constants, common functions that need to be standardized, and data models. Therefore, this section will address each of these parts, first going through the common parts of the system and then showing how a complete module was developed.

5.4.1 Initialization

The system initialization occurs through the `main.py` file, which serves as the entry point to initialize the API and the data processing module.

The `FastAPI` library **fastapiDocs** is used to create the main application, here referred to as `app`. The `CORSMiddleware` middleware is added to the `FastAPI` application, allowing for comprehensive Cross-Origin Resource Sharing (CORS) (Cross-Origin Resource Sharing) configuration. This configuration allows the API to be accessed from different origins.

The import of the `API_data_layer` module not only incorporates the routes related to this module, but also initializes the data receiving module. This implies that the initialization of this module occurs simultaneously with the loading of the API, but on a separate thread, as detailed in 5.3.

For metadata management, an instance of the `MetadataRepository` is created during initialization. This component is essential for loading the metadata that are used in different parts of the system, such as constants and alarm parameters.

The API routes are then included in the main application through the `include_router` method for different modules, such as authentication, API analysis, data layer, notifications, and users.

Additionally, the WebSocket is mounted at the root of the application through the `socketio_app` object, enabling real-time communication between the server and the clients. The implementation of the websocket connection can be seen in 5.4.2.

The execution of the file concludes with the initialization of the `Uvicorn` server **uvicornOfficialDocs**, setting the host and port for listening. `Uvicorn` is an ASGI server that serves as the interface between the application code and the web server. It is responsible for hosting the `FastAPI` application and listening for incoming connections on the specified host and port. The choice of this server was based on the recommendation from the FastAPI documentation **fastapiTutorial**.

```
from fastapi import FastAPI
import uvicorn
from fastapi.middleware.cors import CORSMiddleware
from src.infrastructure.database.metadata.metadata_repository import (
MetadataRepository)
from src.modules.api_analytics import api_analytics_router
from src.modules.api_data_layer import api_data_layer_router
from src.modules.notifications import notification_module_router
from src.modules.user import user_module_router, auth_router
from src.infrastructure.websocket import socketio_app
from src.infrastructure.websocket import socket_dispacher
from dotenv import load_dotenv

load_dotenv()
MetadataRepository()
socket_dispacher
```

```
app = FastAPI()
app.add_middleware(
    CORSMiddleware,
    allow_origins=["*"],
    allow_credentials=True,
    allow_methods=["*"],
    allow_headers=["*"]
)
@app.get("/")
async def health_check():
    return {
        "Status":"OK",
        "Message":"Access /docs for more information"
    }
app.include_router(auth_router)
app.include_router(api_analytics_router)
app.include_router(api_data_layer_router)
app.include_router(notification_module_router)
app.include_router(user_module_router)

app.mount("/",socketio_app)

if __name__ == "__main__":
    uvicorn.run(app, host="0.0.0.0", port=8000)
```

5.4.2 Infrastructure

The infrastructure is composed of 4 sub-modules, Authentication, WebSocket, Database Connection, and Thread Management.

Authentication

The system's authentication implementation was based on the official FastAPI documentation **fastapiSecurity**, therefore a token-based approach was adopted using JWT. The JWT is a widely accepted standard for securely transmitting information between parties. The structure of a JWT is encoded and can be verified to ensure that the data has not been altered during transmission.

The entry point for authentication is the `o_auth2_password_bearer`, an instance of OAuth2PasswordBearer that is designed to obtain the token from the request header. The `auth_middleware` method was defined as an asynchronous middleware, which depends on this bearer token. This middleware is used in the controllers to verify whether the received request has permission to access the information or not.

Within this middleware, the `decode_jwt_token` function is invoked to decode and validate the provided JWT token.

```
o_auth2_password_bearer = OAuth2PasswordBearer(tokenUrl="/user/login")

async def auth_middleware(token:str = Depends(o_auth2_password_bearer))-> TokenPayload:
    try:
        result = decode_jwt_token(token)
        if result.status:
            return result.data
        else:
            raise HTTPException(status_code=401, detail=result.exception.message)
    except JWSError as jwt_err:
        print(jwt_err)
        raise HTTPException(status_code=401, detail=Unauthorized().message)
```

The function `decode_jwt_token` receives a token as an argument and attempts to decode it using the specified secret key and algorithm. If the token is successfully decoded

and is of the type "access_token". Otherwise, different types of exceptions can be raised, for example, if the token has expired or if there is some error in the JWT operations.

```
def decode_jwt_token(token:str)->Result[TokenPayload|None]:
    try:
        token_dict = jwt.decode(token,key=SECRET_KEY, algorithms=ALGORITHM)
        if token_dict["type"] != "access_token":
            return Result(status=False, exception=WrongTokenType(), data=None)
        token_payload = TokenPayload(**token_dict)
        return Result(status=True, data=token_payload, exception=None)

    except ExpiredSignatureError as invalid_token:
        return Result(status=False, exception=Unauthorized(exception=invalid_token), dat

    except (JWSError, JOSEError, JWTError, JWEError) as ex:
        return Result(status=False, exception=GenericException(message="Authorization er

    except Exception as ex:
        print(ex)
        raise ex
```

The model class for the token payload is as follows:

```
class TokenPayload(BaseModel):
    name:str
    exp:int|None = None
    sub:str|None = None
    user_id:str
    type:str = "access_token"
```

The AuthService class is where the main authentication logic is implemented. This

class follows the *Singleton* pattern to ensure that only one instance is created and used throughout the program execution.

Within `AuthService`, the `verify_password` method is used to check if a provided password matches an encrypted password, while the `hash_password` is responsible for encrypting a provided password.

The `create_user_tokens` method generates a pair of tokens (access and refresh) for a user, where the access token is valid for 4 hours and the refresh token for 168 hours. The refresh token is especially important to allow users to obtain new access tokens without having to enter their credentials again. If the access token expires, the refresh token can be used to obtain a new pair of tokens, using the `get_new_user_tokens` method. It is important to note that the frontend does not yet make use of the refresh token, leaving this functionality for a future implementation.

```
class AuthService(metaclass=Singleton):
    def __init__(self):
        self.__database__ = MongoDB()
        self.__user_repository = UserRepository()

        self.__ACCESS_TOKEN_EXPIRE_HOURS__ = 4
        self.__REFRESH_TOKEN_EXPIRE_HOURS__ = 168
        self.__SECRET_KEY__ = SECRET_KEY
        self.__ALGORITHM__ = ALGORITHM
        self.__pwd_context__ = CryptContext(
            schemes=["bcrypt"],
            deprecated="auto")

    def verify_password(self,plain_text_password:str, hashed_password:str):
        return self.__pwd_context__.verify(plain_text_password,hashed_password)
```

```
def hash_password(self,password:str):
    return self.__pwd_context__.hash(password)

async def create_user_tokens(self, user:User)->tuple[str,str]:
    payload = TokenPayload(name=user.name,user_id=str(user.id))
    access_token = self.__create_bearer_token(
        user_id=user.id,
        data=payload.__dict__,
        expire_hours=self.__ACCESS_TOKEN_EXPIRE_HOURS__)
    refresh_token = await self.__create_refresh_token(str(user.id))
    return (access_token, refresh_token)

async def get_new_user_tokens(self,
    refresh_token:str) -> Result[tuple[str, str]]:
    result = self.__decode_jwt_refresh_token(refresh_token)
    if not result.status:
        return Result(status=False, exception=result.exception, data=None)

    token_payload = result.data
    is_valid = await self.__check_if_refresh_token_is_valid(token_payload)
    if not is_valid:
        return Result(status=False, data=None, exception=Unauthorized())

    user = await self.__user_repository.read_user_by_id(token_payload.user)
    payload = TokenPayload(name=user.name,user_id=str(user.id))
    new_access_token = self.__create_bearer_token(
        user_id=token_payload.user,
        data=payload.__dict__,
        expire_hours=self.__ACCESS_TOKEN_EXPIRE_HOURS__)
```

```
        return Result(status=True, data=(new_access_token, refresh_token), exception=Non
```

The methods `__create_bearer_token` and `__decode_token` are auxiliary functions used to create, decode, and verify tokens, respectively.

```
def __create_bearer_token(self,user_id:int, data:dict, expire_hours):
    data_to_enconde = data.copy()
    expire = datetime.now()+timedelta(hours=expire_hours)
    data_to_enconde["exp"] = expire
    data_to_enconde["sub"] = str(user_id)
    return jwt.encode(
        claims=data_to_enconde,
        key=self.__SECRET_KEY__,
        algorithm=self.__ALGORITHM__)

def __decode_token(self,token:str)->dict:
    return jwt.decode(
        token,
        key=self.__SECRET_KEY__,
        algorithms=self.__ALGORITHM__)
```

WebSocket

The implementation of the connection via WebSocket was done using the `socket.io` library **socketIoDocs**, which has various ready-to-use features that facilitate the management of Web Socket connections, such as the creation of rooms for firing notifications.

The structure adopted for managing Web Socket connections was designed in such a way that the client must make a websocket request to the root *endpoint* of the API to be registered in a specific virtual room. After token validation and successful completion of

this request, the client begins to receive all messages directed to the room in which it was registered.

The current implementation contemplates only one room, specifically intended for sending notifications related to the operation of the machines. This room is identified by the identifier `NOTIFICATION_ROOM`. This constant stores the value `"Notification"`, which is the name of the room to be connected, stored along with the system constants described in 5.4.3.

The asynchronous mode ASGI selected for the server creation, and the allowed origins for *CORS* are set as empty.

```
socket_io_server = AsyncServer(async_mode="asgi",
    cors_allowed_origins=[])

socketio_app = ASGIApp(socketio_server=socket_io_server,
    socketio_path="")

socket_dispacher = WebSocketDispatcher(socket_io_server)

@socket_io_server.event
async def connect(sid, environ, auth):
    token = auth["Authorization"]
    await auth_middleware(token)
    socket_io_server.enter_room(sid, NOTIFICATION_ROOM)
```

The `socket_io_server` object is responsible for managing the *WebSocket* communication, while the `socketio_app` creates an ASGI application that interacts with the *WebSocket* server, and is added to the FastAPI server. Additionally, an instance of the `WebSocketDispatcher` class was created to facilitate the sending of notifications through the *WebSocket*. This configuration is done at system initialization, explained in 5.4.1

In the connection event, named `connect`, a client is automatically added to the `NOTIFICATION_ROOM`.

Finally, the `WebSocketDispatcher` class has a `send_notification` method, which is used to send notifications. When calling this method, the notification is converted to the JSON format and sent to all clients in the `NOTIFICATION_ROOM` through the `emit` method.

```
class WebSocketDispatcher:
    def __init__(self,socket_io_server: AsyncServer):
        self.__socket_io_server = socket_io_server

    async def send_notification(self,
        machine_sensor_notification:MachinesSensorAlert):
        machine_sensors_dict = machine_sensor_notification.to_json()
        await self.__socket_io_server.emit(
            NOTIFICATION_ROOM,
            machine_sensors_dict,
            room=NOTIFICATION_ROOM)
```

The `WebSocketDispatcher` class is used by the data receiving module, detailed in 5.2.2, to trigger notifications when an improper operation of the machines is identified.

Thread Management for Asynchronous Tasks

In the system architecture, the need to perform tasks concurrently, without blocking the execution of the API, was identified. These tasks are the execution of the data reception module, and the checking of the system's metadata. Both tasks must be executed in parallel with the execution of the API, without influencing its execution. Therefore, to achieve this goal, a thread manager, called `ThreadManager`, was implemented.

The `ThreadManager` class is designed following the Singleton pattern, ensuring that

only one instance is created, thus avoiding conflicts or redundancies in thread management. A list called `threads` is initialized to store all created threads, while an asynchronous event loop, assigned to the `loop` variable, is created using the `asyncio` library **pythonAsyncio**.

The `start_async_thread` method was introduced to facilitate the creation and management of asynchronous tasks. This method accepts an asynchronous function, `func`, as an argument and performs the following operations:

1. An internal function `start_function` is defined. This function is responsible for starting the execution of the asynchronous task.

2. Within `start_function`, the boolean variable `isSeted` is checked to determine if the event loop has already been set up. Otherwise, the event loop is set up and the asynchronous task is executed until completion through the `run_until_complete` method.

3. If the event loop is already set up (`isSeted = True`), the asynchronous task is simply added to the existing loop using `create_task`.

4. Finally, a new thread is created with `start_function` as the target and added to the `threads` list. The thread is then started, executing the asynchronous task.

```
import asyncio
from threading import Thread


class ThreadManager(metaclass=Singleton):
    def __init__(self):
        self.threads = []
        self.loop = asyncio.new_event_loop()
        self.isSeted = False


    def start_async_thread(self,func):
```

```
        def start_function():
            if not self.isSeted:
                asyncio.set_event_loop(self.loop)
                self.isSeted = True
                self.loop.run_until_complete(func())
            else:
                self.loop.create_task(func())

        new_thread = Thread(target = start_function)
        self.threads.append(new_thread)
        new_thread.start()
```

This implementation allows the execution of multiple asynchronous tasks in parallel, each in its own thread, all managed by the same asynchronous event loop.

Database

In the process of implementing the system, to establish an efficient connection with the database, the `motor` library **motorDocs** was adopted as the mechanism.

At the heart of the connection strategy is a base class, named `BaseDB`, which is responsible not only for establishing the connection with MongoDB, but also for defining a series of basic operations for the manipulation of stored data. The structure of this class is presented below:

```
import motor
class BaseDB:
    def __init__(self):
        self.client = motor.motor_tornado.MotorClient(url, port)
```

Some of the fundamental operations implemented by `BaseDB` include:

- `insert_one`: Receives as parameters the corresponding *database* and *collection* in text format, and the *data* to be inserted. It inserts a document into the specified collection.

- `insert_many`: Receives as parameters the corresponding *database* and *collection* in text format, and the *data* containing several documents to be inserted. It inserts several documents into the specified collection.

- `read_data_with_pagination`: Receives as parameters the *database*, the *collection*, the *query*, the *page_number*, the *limit*, the *sort_descending_field*, and the *projection*. Retrieves data with pagination, allowing for a more organized reading.

- `read_data_with_limit`: Receives as parameters the *database*, the *collection*, the *query*, and the *limit*. Reads data with a predefined limit of returned documents.

- `read_data`: Receives as parameters the *database*, the *collection*, and the *query*. Performs a simple data reading based on a query.

- `get_distinct_property`: Receives as parameters the *database*, the *collection*, and the *property*. It obtains distinct properties from a collection, checking all the present documents.

- `list_collections_by_db`: Receives as a parameter the *database*. It lists all the collections present in a specific database.

- `add_item_into_lists_by_filter`: Receives as parameters the *database*, the *collection*, the *filter*, the *list_properties*, and the *new_data*. It adds an item into specific lists based on a filter.

- `update_item`: Receives as parameters the *database*, the *collection*, the *data* to be updated, and the *filter*. It updates a specific document.

- `update_many_items`: Receives as parameters the *database*, the *collection*, the *data* to be updated, and the *filter*. It updates several documents that meet a filter.

- `count_documents`: Receives as parameters the *database*, the *collection*, and the *query*. It counts the number of documents in a collection that meet a query.
- `get_data_between_dates`: Receives as parameters the *database*, the *collection*, and the *query*. Retrieves data between two specific dates.

With the access base established, other classes were developed, inherited from `BaseDB`, to meet specific system contexts. These classes follow the singleton pattern, which ensures that only one instance of the connection is created for a specific context, optimizing resource management. An example is the `MongoDBIOT` class intended for the data reception module:

```
class MongoDBIOT(BaseDB, metaclass=Singleton):
    def __init__(self):
        super().__init__()
```

Similar classes, following the same format, were created for other contexts, such as database access through the API, ensuring an organized and efficient structure for connection and data manipulation.

5.4.3 Common Files

Within the structure of the API, a folder named `common` was implemented with the aim of centralizing reusable components, covering multiple modules and layers. This organization was established to maximize development efficiency and code consistency.

Data Models

The data models section in the `common` folder houses various classes that define the structure of the used data. Classes specifying users and sensor data are present, and they make use of the `Pydantic` library to define the models and create data validations.

`Pydantic` **pydanticDocs** is a data validation library that adds static typing in Python to validate that the received data matches a certain format or schema. When used for the

construction of the API, `Pydantic` contributes to the automatic and consistent verification of data sent through HTTP requests, and manipulations performed in the database. This approach reduces the need for manual coding for data validations, thus speeding up development time and increasing code robustness.

To illustrate, we have the NotificationSensorResponse class, which is responsible for defining the data model that is returned when the API receives a request for a specific user's notifications. The use of `BaseModel` in the inheritance system, from `Pydantic`, necessary to define the return types within FastAPI, is highlighted. In addition, the `Field` function, also from `Pydantic`, is used to indicate with three points that it is a mandatory attribute to be informed in the class construction.

```
class NotificationSensorResponse(BaseModel):
    data:list[dict] = Field(...)
    total_count:int = Field(...)
```

Other important data models are the exception classes and a class called `Result`, responsible for data traffic between the different layers of the application, shown in the implementation of the API module in 5.4.4. Example of a class that defines a system exception.

```
class CustomBaseException(Exception):
    def __init__(self, message:str, exception, *args: object) -> None:
        self.message = message,
        self.exception = exception
        super().__init__(*args)


class GenericException(CustomBaseException):
    def __init__(self,
        message:str =   "An error has occurred",
        exception = None) -> None:
        super().__init__(message, exception)
```

Class `Result` used for communication between layers of the system modules. The `TypeVar` is used to indicate a generic type for the `data` attribute of the class.

```
from typing import TypeVar, Generic
from src.common.models.exceptions.unauthorized import CustomBaseException

T = TypeVar('T')
class Result(Generic[T]):
    def __init__(self, status:bool, data:T|None, exception:CustomBaseException|None):
        self.status = status
        self.data = data
        self.exception = exception
```

The use of the `Singleton` class is also highlighted, used when there is a need to ensure that only one instance of a certain class will be used.

```
class Singleton(type):
    _instances = {}
    def __call__(cls, *args, **kwargs):
        if cls not in cls._instances:
            cls._instances[cls] = super(Singleton, cls).__call__(*args, **kwargs)
        return cls._instances[cls]
```

Finally, the `PyObjectId` class is highlighted, used to define a type for the ID attribute of classes that represent database models. The methods defined for this class are used internally by `FastAPI` and `Pydantic`.

This class is necessary so that the ID can be correctly converted to text, and returned in the requests. Furthermore, the use of this class enables the manipulation of the ID when necessary, leaving the management of unique identifiers to the application and not to the database.

```
class PyObjectId(ObjectId):
    @classmethod
    def __get_validators__(cls):
        yield cls.validate

    @classmethod
    def validate(cls, v):
        if not ObjectId.is_valid(v):
            raise ValueError("Invalid object id")
        return ObjectId(v)

    @classmethod
    def __modify_schema__(cls, field_schema):
        field_schema.update(type="string")
```

Helper Functions

The section of *helper* functions in the `common` folder was built to contain methods that are common in different modules of the system, avoiding code duplication and standardizing operation.

An example of these functions is the conversion of `Datatype`, shown in the section on constants 5.4.3, to MongoDB collections. The following code illustrates one of these processes:

```
def sensor_name_to_processed_collection(
    sensor_name:Datatype)->str:
    sensor = (sensor_name.name).capitalize()
    return IOT_AGGREGATION_COLLECTION.replace("NAME", sensor)
```

These functions are used to determine the correct names of the collections according to the type of data to be processed and manipulated by the modules. Functions such as

`sensor_name_to_processed_collection` and `sensor_name_to_raw_data_collection` convert between sensor names and collection names.

Additionally, a series of functions was developed to map the processed collection name back to the corresponding `Datatype`, shown in the section on constants 5.4.3, ensuring safer and more consistent data manipulation.

Constants

Finally, the constants section stores a series of fixed values that are used in various parts of the system. This includes database names, alert types, websocket rooms, and others as needed.

Among the constants, `DataType` stands out, which is an enum that standardizes the types of data that can be received from the sensors. This standardization is employed in various modules to ensure that data is received, processed, stored, and returned in a consistent and correct manner.

```
from enum import Enum

class Datatype(Enum):
    PRESSURE = "PRESSURE"
    TEMPERATURE = "TEMPERATURE"
    VOLTAGE = "VOLTAGE"
    CURRENT = "CURRENT"
    SPEED = "SPEED"
    ACCELERATION = "ACCELERATION"
    DISTANCE = "DISTANCE"
    HUMIDITY = "HUMIDITY"
    FORCE = "FORCE"
    PRODUCTION_COUNTER = "PRODUCTION_COUNTER"
```

5.4.4 Modules

The API was structured into modules, each one responsible for managing a certain context. The developed modules are listed below.

1. **IOT Analytics**: Module responsible for managing access to sensor information, both real-time data via stream, and the information processed by the data processing module, explained in 5.3.

2. **Notifications**: Module responsible for managing access to notifications.

3. **User**: Module responsible for managing the system users' information, and performing the authentication explained in 5.4.2.

4. **Downtime Analytics**: Module used to provide test data on machine downtime. This module feeds the screen that displays information about machine downtime.

Each module followed a pattern of having one layer for receiving HTTP requests, the `controller`, another to handle the request according to business rules, the `service`, and the `repository`, to provide methods for accessing and manipulating information in the database.

Controller

The controller has the function of receiving HTTP requests, sending the received information to the service layer, and making the appropriate return, with the formatted information, and correct HTTP code.

To exemplify the operation of the controller, a specific example will be presented. The following code snippet represents the router of the API for the IoT sensor data:

```
iot_data_router = APIRouter(tags=["IOT Data"], dependencies=[Depends(auth_middleware)])


service = ServiceIOT()
```

```
@iot_data_router.get("/realtime")
async def real_time_iot():
    def get_real_time_data():
        sensor = SensorValue()
        while True:
            time.sleep(1)
            lista_json = [machine.to_json() for machine in sensor.machine_list]
            last_data = json.dumps(lista_json)
            yield bytes(last_data, "utf-8")

    return StreamingResponse(
        get_real_time_data(),
        media_type="application/octet-stream")
```

In this example, the controller uses the `APIRouter` to create routes associated with IoT data. An authentication middleware is applied as a dependency, as shown in the authentication detail in 5.4.2, ensuring that only authenticated users can access these routes. In this case, by applying the middleware in the creation of the `iot_data_router`, it is guaranteed that all `endpoints` created from it require authentication.

Below, an instance of the service to be used by the `endpoints` to respond to the received requests is created.

The `/realtime` route is designated to provide real-time data. An internal function, `get_real_time_data`, is responsible for collecting this data from the `SensorValue`, explained in **??**. In this endpoint, the API takes the current values and returns a `StreamingResponse`, which sends real-time data as a continuous stream.

The other routes, such as `/machines_in_sensor` and `/graph_info`, operate in a similar way, but with different responsibilities. They make calls to the `ServiceIOT` instance to retrieve specific information and return it to the client. If an exception or error occurs,

an `HTTPException` is thrown with an appropriate HTTP status code and a detailed error message.

In these two endpoints, it is important to highlight the specification of the `response_model`, so that automatic validations by the framework occur before the data is returned, ensuring consistency in the returned data.

```
@iot_data_router.get("/machines_in_sensor", response_model=list[MachinesSensor])
async def get_machines_in_sensor():
    result = await service.get_all_machines_processed_info()
    if result.status:
        return result.data

    raise HTTPException(status_code=500, detail=result.exception.message)

@iot_data_router.get("/graph_info", response_model=list[ProcessedData])

\begin{verbatim}
async def get_graph_info(machine:str, sensor:Datatype, initial_date:datetime, end_date:d
    result = await service.get_processed_data(
        machine=machine,
        data_type=sensor,
        initial_date=initial_date,
        end_date=end_date)

    if result.status:
        return result.data

    raise HTTPException(status_code=500, detail=result.exception.message)
```

Service

The service layer is used to apply the appropriate rules before returning the data to the control layer 5.4.4. It can access the repository layer to read or write data, and it should return the data to the control layer using the `Result` class, shown in section 5.4.3, so that the result of the processing can be correctly identified, and the appropriate return can be given to the client who made the request.

For a deeper understanding, a specific example of the service layer implementation follows below:

```
class ServiceIOT:
    def __init__(self):
        self.__repository = RepositoryIOT()
        self.__database__ = MongoDB()
        self.appMetadata = MetadataRepository()

    async def get_processed_data(self,
        machine:str,
        data_type:Datatype,
        initial_date:datetime,
        end_date:datetime)-> Result[list[ProcessedData]]:
        try:
            alert_parameter = await self.appMetadata.get_sensor_alert_value(
            data_type)
            processed_data = await self.__repository.read_iot_processed_data__(
                machine=machine,
                datatype=data_type,
                initial_date=initial_date,
                end_date=end_date,
                sort_by_field="more_recent_register")
```

```
            for data in processed_data:
                data["alert_parameter"] = alert_parameter
            return Result[list[MachinesSensor]](
                status=True,
                data=processed_data,
                exception=None)
        except Exception as ex:
            return Result[list[MachinesSensor]](
                status=False,
                data=None,
                exception=GenericException(exception=ex))
```

In this implementation, the `ServiceIOT` class is initialized with instances of `RepositoryIOT` and `MetadataRepository`, allowing the service to access the corresponding repository and metadata layers.

The `get_processed_data` method is used to obtain processed data based on various parameters such as machine, data type, and date range. Initially, an alert value is retrieved from the sensor metadata for the provided data type. Subsequently, the processed data is read from the repository. For each retrieved data entry, the alert value is added as a new field. The method returns a `Result` object encapsulating these data. The `Result` object is detailed in 5.4.3.

The `get_all_machines_processed_info` method retrieves information about all processed machines and sensors. It iterates through the processed data collections, aggregating machine and sensor information. In case of success, it returns a `Result` object containing a list of machines and their corresponding sensors.

```
async def get_all_machines_processed_info(
    self)-> Result[list[MachinesSensor]]:
    try:
```

```
        collection_list =
            await self.__repository.get_processed_data_collections()
        machine_sensors: list[MachinesSensor] = []
        for collection in collection_list:
            machine_list = await self.__repository
                .get_distinct_machines_by_collection(collection)
            sensor = processed_collection_to_sensor_name(collection)

            for machine in machine_list:
                matching_sensor =
                next((data for data in machine_sensors
                    if data.machine == machine), None)
                if matching_sensor:
                    matching_sensor.sensors.append(sensor)
                else:
                    machine_sensors.append(MachinesSensor(
                        machine=machine,
                        sensors=[sensor]))

        return Result[list[MachinesSensor]](
            status=True,
            data=machine_sensors,
            exception=None)
    except Exception as ex:
        print(ex)
        return Result[bool](
            status=False,
            data=None,
            exception=GenericException(exception=ex))
```

The last method, `read_raw_data_by_id`, is used to read raw data based on an identifier and data type. It accesses the corresponding repository to retrieve the data and returns a `Result` object containing these data or an exception, if applicable.

```
async def read_raw_data_by_id(self,
    raw_data_id:str,
    datatype:Datatype) -> Result:
    try:
        collection = sensor_name_to_raw_data_collection(datatype)
        result = await self.__repository.read_raw_data(collection,raw_data_id)
        return Result(status=True, data=result, exception=None)
    except Exception as ex:
        return Result[bool](
            status=False,
            data=None,
            exception=GenericException(exception=ex))
```

This implementation exemplifies how the service layer interacts with the repository layers and accesses metadata, and how it prepares the data to be sent back to the controller, thus ensuring a cohesive and efficient data flow through the various layers of the application.

Respository

Finally, the repository layer is responsible for accessing the database and performing read and write operations as needed. This layer initiates a connection with the database, and its methods use the base methods defined by the database connection infrastructure, in 5.4.2, to perform operations according to the context of that module.

The following code provides an example of the implementation of this layer:

```
class RepositoryIOT:
    def __init__(self):
```

```
        self.__database__ = MongoDB()

    async def get_processed_data_collections(self):
        collection_list =
            await self.__database__.list_collections_by_db(
            IOT_PROCESSED_DATA)

        return collection_list

    async def get_distinct_machines_by_collection(self, collection:str):
        machine_list =
            await self.__database__.get_distinct_property(
                IOT_PROCESSED_DATA,
                collection,
                "machine")

        return machine_list

    async def read_raw_data(self, collection:str, raw_data_id:str):
        result = await self.__database__.read_data(
            IOT_DATABASE,
            collection,
            {"_id":ObjectId(raw_data_id)})

        return result

    async def read_iot_processed_data__(self, machine:str, datatype:Datatype,
        initial_date:datetime, end_date:datetime,
        sort_by_field:str) -> list[ProcessedData]:
```

```
try:
    collection = sensor_name_to_processed_collection(datatype)
    query = {
        "machine":machine,
        sort_by_field:{
            "$gte":initial_date,
            "$lte":end_date
        }
    }
    result = await self.__database__.get_data_between_dates(
        IOT_PROCESSED_DATA,
        collection,query)
    return result
except Exception as ex:
    print(ex)
    raise ex
```

Upon initializing the `RepositoryIOT` class, a connection with MongoDB is instantiated. The `get_processed_data_collections` method is used to list all the processed data collections from the `IOT_PROCESSED_DATA` database. This method makes a direct call to the collection listing method provided by the `MongoDB` class.

The `get_distinct_machines_by_collection` method is responsible for retrieving a list of distinct machines for a given collection. It does this through the `get_distinct_property` method of the database.

The `read_raw_data` method is used to read raw data from a specific collection, using the data ID as a search parameter. It makes a call to the `read_data` method of the `MongoDB` class, providing the necessary parameters for data reading.

Finally, the `read_iot_processed_data__` method is used to read processed data

based on various criteria such as machine, data type, and date range. A query is built for this purpose and passed to the `get_data_between_dates` method of the `MongoDB` class.

Each of these methods assists in maintaining a clear separation of responsibilities, allowing the service layer to maintain a strict focus on business logic, while the repository layer manages database operations.

5.5 Frontend Implementation

The implementation of the user interface was developed according to the architecture exposed in section 4.2. The development of the *frontend* is segmented into several parts, which include the organization of the system pages according to the pre-defined structure of Next.js **nextjsDocs**, the management of data accessed by the components, the configuration for external access, and the construction of individual components.

It is relevant to note that, to maintain compliance with best practices and simplify development, the default settings of Next.js were maintained.

5.5.1 Pages

Being a framework, NextJs has a predefined structure for creating system pages as well as their routes **nextjsDefiningRoutes**. Within the framework files, the *pages* folder is used to store each of the system's pages, with each file being a page, and the file name being the route for access. The configuration of the pages occurs through files with specific names, in this case *_app.tsx* and *_document.tsx*.

Configuration pages

Regarding the configuration pages, we first have the *_app.tsx*. This file is responsible for configuring and managing contexts, global styling, and date localization, in other words, global aspects of the entire application.

The code begins with the import of various modules and libraries, which includes

specific contexts such as `OpenContext` and `PrivateContext`, which are better explained in 5.5.2, and support for date localization with `AdapterDayjs` **dayJsInstallation**.

The `LocalizationProvider` and `AdapterDayjs` are from libraries that aim to provide localization and date formatting functionalities. The `LocalizationProvider` acts as a wrapper for the date system, allowing integration with different date management libraries. In this case, the `AdapterDayjs` is used as the adapter for the Day.js library, allowing dates to be manipulated and formatted efficiently and compatible with various geographical locations and formats. With these libraries, it becomes easier to manage dates for the construction of the dashboard filter that manages the date period displayed in the charts, explained in 6.3.

The type `NextPageWithLayout` was defined to enrich the page properties with information about the layout. This allows each page to have a custom layout if necessary, offering great flexibility in interface design.

The main function `App`, which receives `Component` and `pageProps` as arguments, is responsible for setting up the layout and rendering the page components. The logic within this function checks the current route using `useRouter` **nextjsUseRouter** to determine if the user is on the login page.

The content is then encapsulated within the relevant contexts. If the user is on the login page, only the `OpenContext` is applied. For all other pages, the `PrivateContext` is additionally applied, ensuring that sensitive information is accessed only by authenticated users. The contexts used are detailed in 5.5.2.

Within the `LocalizationProvider`, the `AdapterDayjs` adapter is used to provide date localization functionalities, making the application more versatile in different locations.

```
import {OpenContext, PrivateContext} from '@/context'
import '@/styles/globals.css'
import { LocalizationProvider } from '@mui/x-date-pickers'
import { AdapterDayjs } from '@mui/x-date-pickers/AdapterDayjs'
import { NextPage } from 'next'
```

```
import type { AppProps } from 'next/app'
import { useRouter } from 'next/router'
import { ReactElement, ReactNode } from 'react'

export type NextPageWithLayout<P = {}, IP = P> = NextPage<P, IP> & {
    getLayout?: (page: ReactElement) => ReactNode
}

type AppPropsWithLayout = AppProps & {
    Component: NextPageWithLayout
}

export default function App({ Component, pageProps }:
AppPropsWithLayout) {

    const getLayout = Component.getLayout || ((page) => page)
    const router = useRouter()
    const isLoginPage = router.pathname === "/"

    const componentWithProps = <Component {...pageProps} />

    return getLayout(
    <LocalizationProvider dateAdapter={AdapterDayjs}>
        <OpenContext>

        {isLoginPage?
            <>{componentWithProps}</>
            :<PrivateContext>
                {componentWithProps}
```

```
            </PrivateContext>
        }

        </OpenContext>
    </LocalizationProvider>
    )
}
```

Although it is a simpler file compared to `_app.tsx`, the `_document.tsx` has the responsibility of defining the global HTML structure of the application.

In the file, the components `Html`, `Head`, `Main`, and `NextScript` from the `next/document` library were imported. These components are used to create the basic structure of the HTML page within NextJs.

The `Html` component is used to encapsulate all HTML content and includes the attribute `lang="en"`, which sets the page language to English. The `Head` component **nextjsHeadComponent** is employed to add elements to the HTML page header. In this case, the page title is set to 'Dashboard'.

The body of the HTML page is composed of the `Main` and `NextScript` components. The `Main` is where the main content of the page is inserted, while the `NextScript` is responsible for including the necessary scripts for Next.js to operate.

It is worth noting that the `_document.tsx` does not have access to specific page features such as the `getInitialProps` **nextjsInitialProps**, `getStaticProps` **nextjsGetStaticProps**, or `getServerSideProps` **nextjsGetServerSideProps** methods (NextJs functions for server-side data loading). This implies that this file is ideal for configurations that are common to all pages and do not require dynamic information.

```
import { Html, Head, Main, NextScript } from 'next/document'

export default function Document() {
  return (
```

```
    <Html lang="en">
      <Head title='Dashboard'/>
      <body>
        <Main />
        <NextScript />
      </body>
    </Html>
  )
}
```

System Pages

The system pages are divided into two types, private and public, with the public one being only the login page. This public page is in the `index.tsx` file, being the root route of the system.

In this file, only `meta` settings and the `Login` component are invoked. The `Head` element **nextjsHeadComponent** is used to define global HTML settings, such as the page title and metadata.

The `Login` component is called within the `main` tag, which serves as the main content of the page. This design approach keeps the `index.tsx` page lean, transferring most of the logic and visual presentation to the `Login` component. This is an example of the principle of separation of concerns, where each file or component has a single clearly defined responsibility.

```
export default function Home() {
    return (
      <>
        <Head>
          <title>Catraport Dashboard</title>
```

```
        <meta name="description"
          content="Generated by create next app" />
        <meta name="viewport"
          content="width=device-width,
          initial-scale=1" />
        <link rel="icon" href="/favicon.ico" />
      </Head>
      <main>
        <Login/>
      </main>
    </>
  )
}
```

The other system pages are located within the `dashboard` folder, which is also inside the `pages` folder. This implies that all pages within this folder should be accessed at the `/dashboard` route **nextjsDefiningRoutes**.

Within the dashboard, there is the main page, at `/index.tsx`, with the dashboard page that displays real-time data and charts with processed historical data. The functionalities of this page are detailed in 6.

This file follows the same design logic observed on the login page, maintaining the separation between the page settings and the logic of the invoked components.

The `Dashboard` component is based on composition, delegating various responsibilities to individual components. The `DashboardLayout` component is used as a container that defines the global structure of the page, providing a consistent layout for the other dashboard pages as well. Within this component, several others are called to perform specific functions.

The `DashboardHeader` is responsible for displaying the page header, providing access to filters for viewing information. Following this is the `SensorsValues` component, which

is designated to display the sensor values in real time.

A `Divider` element, from the `Material UI 5` library **muiDocs**, is inserted to provide a visual separation between the different sections of the page. Finally, the `SensorsGraphs` component is invoked to display graphs related to the historical data of the sensors in an aggregated manner.

```
export default function Dashboard() {
    return (
        <DashboardLayout>
            <DashboardHeader/>
            <SensorsValues/>
            <Divider/>
            <SensorsGraphs/>
        </DashboardLayout>
    )
}
```

The other pages of the dashboard were also built using the demonstrated composition logic and using the same base component for the layout, `DashboardLayout`. These pages are:

1. **Maintenace**: Responsible for displaying machine downtime data in the form of graphs to demonstrate the visualization of this information within the system, 6.4.
2. **Profile**: Responsible for displaying the information of the user who is logged into the system, as well as allowing changes to the data, 6.5.

5.5.2 System data management

One of the important points in frontend development is the management of global states, which are information or behaviors shared among unrelated components. Within this project, the `Context API` **reactCreateContext** was used for this purpose, being a native

React solution that stands out for its ease of implementation and use. This feature allows data to be efficiently passed throughout the component tree, eliminating the need to manually pass properties through intermediate levels.

Not only data states, but also behavioral states, such as the user's login status and the side menu status (open or closed), were managed through the `Context API`. The data layer was constructed in such a way that all necessary information for the system's operation, which depends on a global value, was included.

The contexts created for state management are as follows:

1. `SnackbarContext`: Used for managing messages and alerts in the system, facilitating access to the function that displays alerts and its configuration throughout the system.

2. `AuthContext`: Responsible for managing the user's authentication state, 5.4.2.

3. `ThemeContext`: Responsible for managing the visual theme of the application **muiDefaultTheme**.

4. `NotificationContext`: Used for the management, reading, and receiving of notifications in the system.

5. `LegacyContext`: Manages machine stoppage information after being read from the backend.

6. `IotContext`: Used for managing states related to IoT devices in the system.

7. `DrawerContext`: Responsible for managing the state of the side menu (open or closed), and making it available on all system pages that use the side menu.

Each context was designed with a specific function in order to allow a clear separation of responsibilities. Using the architecture shown in X, this frontend data management model becomes simple and scalable, as specified in the requirements, in 3.1.

Management of different contexts

To deal with the complexity generated by the variety of contexts needed in the system, it was chosen to instantiate these contexts through specific components. These components were designed to encapsulate different groups of contexts, according to access needs.

Two main components were developed: `OpenContext` and `PrivateContext`. The former is responsible for instantiating contexts that are available to any individual who accesses the system. The latter is in charge of instantiating contexts that can only be accessed by authenticated users. The creation of a context is better exemplified in 5.5.2.

These components are used in the `__app.tsx` file, which is the initial configuration file of Next.js, as explained in 5.5.1, in order to make the contexts accessible throughout the application's component tree.

Below is the code that illustrates how these components were implemented:

```
interface Props{
    children:React.ReactNode
}

function OpenContext({children}:Props){
    return (
        <SnackbarContextProvider>
            <AuthContextProvider>
                <ThemeContextProvider>
                        {children}
                </ThemeContextProvider>
            </AuthContextProvider>
        </SnackbarContextProvider>
    )
}
```

```
function PrivateContext({children}:Props){
    return(
        <>
            <NotificationProvider>
                <LegacyContext>
                    <IotContext>
                        <DrawerContextProvider>
                            {children}
                        </DrawerContextProvider>
                    </IotContext>
                </LegacyContext>
            </NotificationProvider>
        </>
    )
}
```

In this way, contexts are properly isolated and managed, ensuring that the correct data and functionalities are available to users, according to their level of access.

Creating a context

For the creation of a context, the correct definition of types is first necessary, mandatory due to the use of `Typescript`. For each context, a properties interface (`Props`) and a default value (`DEFAULT_VALUE`) are created.

To demonstrate the creation of a context within the project, the `DrawerContext` will be used as an example, which is responsible for managing the state of the application's drawer. The following code exemplifies how this context was created:

```
import { createContext, useContext, useState } from "react"


interface Props {
```

```
    open:boolean
    setOpen:React.Dispatch<React.SetStateAction<boolean>>
}

const DEFAULT_VALUE = {
    open:false,
    setOpen:()=>{}
}

const DrawerContext = createContext<Props>(DEFAULT_VALUE)

function DrawerContextProvider({ children }:{children:React.ReactNode}){
    const [open, setOpen] = useState<boolean>(false)

    return (
        <DrawerContext.Provider value={{open,setOpen}}>
            {children}
        </DrawerContext.Provider>
    )
}

export default function useDrawer(){
    return useContext(DrawerContext)
}

export {DrawerContextProvider};
```

In this example, the `DrawerContext` context is created using the `createContext` function from React **reactCreateContext**. The `Props` interface defines the types for the open

state of the drawer (`open`) and the function to set this state (`setOpen`). A default value (`DEFAULT_VALUE`) is established to initialize the context.

The `DrawerContextProvider` component uses local React state to manage the `open` state value. This value and the `setOpen` function are then made available to all child components through the `DrawerContext.Provider`. In this way, the `DrawerContextProvider` component is used in context management, as explained in 5.5.2, to make all information accessible to the entire component tree.

The `useDrawer` function is a custom function that facilitates access to the `DrawerContext` in any part of the application. Within React, functions with 'use' in front are called `hooks`, as can be seen in the official documentation at **reactHooksReference**.

Thus, the context was created and can be used to manage the open and closed state of the side menu throughout the application.

This context creation model is repeated for all other contexts, with the addition of access to the backend, which is explained in 5.5.3.

5.5.3 External Access

For interaction with data stored and managed by the backend, an external access layer was established in the frontend. This layer serves as a centralized point for all network requests and is necessary for reading and manipulating data that is outside the scope of the frontend, therefore it deals with HTTP requests, WebSocket connections, and data reception via stream.

The various contexts created in the system, as described in 5.5.2, use this external access layer to load the necessary data. At the initialization of each context, function calls to this layer are made, if necessary. These calls are responsible for making requests to the backend and for receiving the returned information.

In the example below, the hook `useEffect` **reactUseEffect** is used to call a function that accesses the external access layer to load data related to the sensors, being them the real-time data, and chart data, as soon as the context is initialized.

```
const fetchSensorData = useCallback(async()=>{
    await Promise.all([
        getRealTimeDataData(reciveRealTimeData),
        fetchGraphData(),
    ])
},[])

useEffect(()=>{
    fetchSensorData()
},[])
```

HTTP Requests

The `Axios` library **axiosIntro** was used to facilitate the making of requests to the backend. This library provides a simple and efficient interface for creating HTTP requests, and it was integrated into the functions of the external access layer. This library allows an initial configuration to be used in all the requests made.

In the configuration used, the backend address and the URL to add the settings to the base route, where all requests should go, are read from the system's environment variables. Another applied configuration is the addition of the interceptor **axiosInterceptors** at the moment the request is made, adding the necessary configuration for authentication to the header, reading the access token from the local storage **mdnLocalStorage**, and adding it in the correct format for backend reading in requests that require authentication.

```
const baseUrl = process.env.NEXT_PUBLIC_API_URL
const apiRoute = process.env.NEXT_PUBLIC_API_ROUTE

const baseApi = axios.create({
  baseURL: `http://${baseUrl}${apiRoute}`
});
```

```
baseApi.interceptors.request.use(function (config) {
    let token = localStorage.getItem("access_token")
    if (token) {
      config.headers['Authorization'] = `Bearer ${token}`;
    }
    return config;
  }, function (error) {
    return Promise.reject(error);
});
```

The functions of this layer use this base configuration to fetch data from the backend. The function below exemplifies one of these external access functions to fetch data for a specific chart. The function uses the base configuration of `axios` along with a specific url to access the desired endpoint to fetch the information. If the return has a `status code` equal to 200, it means that the request was successful, then the data is returned to the context that made the call of this function. The context receives the data and makes it available to the entire application, as explained in 5.5.2.

```
async function getGraphData(
  machine:string,
  type:SensorType,
  startDate:Dayjs,
  endDate:Dayjs):Promise<Array<MachineGraphAggregateData>>{
  try{
    let url = "/iot/graph_info"+get_graph_query(
      machine,
      type,
      startDate,
      endDate)
```

```
    let response = await baseApi.get(url)
    if(response.status === 200){
      return response.data
    }
    throw("Error to access graph - "+response.status)
  }catch(ex){
    throw("Error to access graph - "+ex)
  }
}
```

WebSocket Connection

In addition to HTTP requests, the external access layer also manages the WebSocket connection. Specifically, the notification context uses this connection to receive and manage real-time notifications.

The management of the WebSocket connection is carried out through the *CustomSocketConnection* class. This class is designed as a singleton, ensuring that a single instance is created and reused throughout the application. It is responsible for initializing and maintaining the `Socket` object, which is part of the *socket.io-client* library **socketIoClientApi**.

```
class CustomSocketConnection{
  private static _instance:CustomSocketConnection|null = null
  private _socketio: Socket|null = null

  private constructor() {
      if (CustomSocketConnection._instance === null) {
          CustomSocketConnection._instance = this;
      }
  }
```

```
  public static getInstance(): CustomSocketConnection {
      if (!CustomSocketConnection._instance) {
          CustomSocketConnection._instance = new CustomSocketConnection();
      }
      return CustomSocketConnection._instance;
  }

  get socketio(){
    if(this._socketio===null){
      let token = localStorage.getItem("access_token")
      let url = process.env.NEXT_PUBLIC_API_URL??"localhost"
      this._socketio = io(url, { autoConnect: false, auth: { "Authorization": token } })
    }
    return this._socketio
  }
}
```

The `getInstance()` method ensures that only one instance of the class is created. This instance is stored as a static attribute and is returned whenever requested.

An instance of the *CustomSocketConnection* class is used in the notification context. Through this instance, the context is able to receive messages from the server, manipulate them, and then make them available to the entire application.

The class also includes an authentication mechanism. The access token is retrieved from local storage **mdnLocalStorage** and is used as part of the authentication header during the connection process.

Within the notification context, the Socket attribute of the class is used in the context initialization to establish the connection. The instance of the class is stored in a constant, and then three functions are registered for connection events, notification receipt, and

disconnection. The function that handles the event of receiving a new notification sends the data to a helper function whose purpose is to analyze the received data and update the list of notifications that appear on the screen.

```
useEffect(() => {
  const socket = CustomSocketConnection.getInstance()
  function onConnect () {
      console.log(‘Connected with id: ${socket.socketio.id}‘);
      console.log(‘Connection Status: ${socket.socketio.connected}‘);
  }

  function newNotification(data:any){
      console.log("Socket Io Notification", data);
      checkNewNotification(data);
  }

  function disconnect(data:any) {
      console.log("Socket Io disconnect", data);
      console.log("Connection Status");
  }

  socket.socketio.on(’connect’, onConnect);
  socket.socketio.on(’Notification’, newNotification);
  socket.socketio.on(’disconnect’, disconnect);

  return () => {
      socket.socketio.off(’connect’, onConnect);
      socket.socketio.off(’Notification’, newNotification);
      socket.socketio.off(’disconnect’, disconnect);
```

```
  };
}, [notifications]);
```

In this way, the external access layer provides a Web Socket connection method to be used to receive new notifications while the user is connected to the system.

Receiving data via Stream

Regarding the external access layer, the last type of connection is the receipt of data via stream. This mechanism allows for the constant updating of received data, thus ensuring a continuous flow of updated information for the application. Specifically, this connection is used to provide the data from the sensors that are received in the backend, as explained in **??**.

The receipt of data via stream is implemented using the *fetch* function **mdnFetchAPI**, native to JavaScript. This function is responsible for making the request to the corresponding endpoint and obtaining the data stream in real time.

The `readStream` function is used to interpret the data from the stream. This function receives the reader of the response body, and returns the data converted to the JSON format.

```
async function readStream(
  reader:ReadableStreamDefaultReader<Uint8Array> | undefined) {
  let result = await reader?.read()
  if (!result?.done) {
      let value = result?.value
      if(value){
          const jsonData = parseBytesToJson(value)
          return jsonData
      }
  }
  return false
```

```
}
```

The function `getRealTimeDataData` is responsible for initiating the data streaming process. It is in this function that the IoT sensor data context makes the call and, consequently, receives and sends the data to the function received as a parameter.

Since the function does not use the axios structure explained in 5.5.3, it is necessary to perform the same authentication configuration that was previously explained in the structure.

After making the request, the data reader is read from the body of the request response, and passed to the `readStream` function, explained earlier, to be interpreted and converted to JSON.

```
async function getRealTimeDataData(setData: UpdateDataFromStream) {
  try{
    let token = localStorage.getItem("access_token")
    let url = baseApi.getUri()
    let response = await fetch(`${url}/iot/realtime`,{headers:{
        Authorization: `Bearer ${token}`,
        'Content-Type': 'application/json',
    },});

    const reader = response?.body?.getReader();
    let result:Array<MachineRealTimeData>|boolean = await readStream(reader)
    if(typeof(result) === "boolean"){
        throw "Error to read stream data"
    }
    do{
      if (Array.isArray(result)) {
          const typedResult = result as MachineRealTimeData[];
```

```
            setData(typedResult);
        }
        result = await readStream(reader)
      }while(result!==false)
    }catch(ex){
      console.error(ex)
      throw ex
    }
}
```

The function `getRealTimeDataData` continues to run as long as the connection is open and no errors occur. The `setData` method is then called to send the reading result to the context that called the function.

Within the context of IOT sensor data, the `setData` function passed as a parameter only updates the global state to update the information in all components that make use of it.

5.5.4 Component construction

In the system implementation, the logic of component construction **reactFirstComponent** was used to compose the screens. Modularity and code reuse are critical factors that motivate this choice.

Components access data directly from contexts, as I discuss in 5.5.2. This method facilitates the passage of data and allows the components to be more specific in their function, in addition to avoiding the passage of many properties in the component tree.

The basis for the smaller components was extracted from the Material-UI Version 5 (MUI5) library **muiDocs**. This includes elements such as buttons, containers, text boxes, and others. A practical example is the sidebar menu on the dashboard pages, where the Drawer component from MUI5 was employed.

A component that deserves highlight is the `Grid` from MUI5 **muiReactGrid**. The

use of Grid allowed a simple spatial organization of the user interface elements. This grid system offers a flexible approach to allocate space, align content, and handle screen variations, which is especially useful in web applications with a lot of information and different components on screen. In the sidebar menu, for example, the `Grid` component was used to organize the menu information and define the positioning according to the open or closed state.

```
<Grid
  container
  direction="column"
  justifyContent="space-between"
  alignItems={open?"center":"start"}
  height={"97vh"}
>
  <Grid
    container
    item
    direction="column"
    justifyContent="space-between"
    alignItems={open?"center":"start"}
  >
    // Content
  </Grid>
</Grid>
```

The use of Grid, therefore, contributed to the cohesion of the layout, providing a solid structure upon which other components could be organized in a simple and easy-to-understand manner, fulfilling requirements to facilitate maintenance specified in 3.1.2.

On the other hand, for the construction of the graphs, the Recharts library was used. The graph shown on the dashboard with the information generated by the processing

module, detailed REF 5.3, is composed of Scatter, Area, Bar, and Line charts, thus allowing a multifaceted analysis of the generated information.

For components that require date manipulation, the `Days Js` library **dayJsInstallation** was integrated. A use case is the date filter component for displaying graphs, which also employs the DatePicker **muiDatePickerValidation** from MUI5 for a more intuitive user interface. The Days Js library facilitates the manipulation of date data input and output.

```
<DatePicker
  value={dateFilter.startDate}
  onChange={(value)=>onChangeDate(value, "startDate")}
  label="Data inicial"
/>

const onChangeDate = (newDate:Dayjs|null,
  dateField:"startDate"|"endDate")=>{
  if(newDate!==null){
    setIsDataUpdated(false)
    if(dateField==="endDate"){
      setDateFilter(oldValue=>({...oldValue,"endDate": newDate}))
    }else{
      setDateFilter(oldValue=>({...oldValue,"startDate": newDate}))
    }
  }
}
```

Chapter 6

System Characteristics from a Functional Point of View

After detailing the requirements, the chosen architecture for the system, the technologies used in the project, and the detailing of the implementation of each system component, this chapter is focused on the functionalities that make up the application. The project encompasses several components, including a frontend layer, a backend layer, a data receiving module, a data processing module, and the database. Therefore, this chapter aims to detail the operation of each system functionality, providing a comprehensive view of how each component interacts and contributes to the operation of the system as a whole.

Each section of this chapter will be dedicated to a specific functionality, examining its role and operation in depth, as well as the interaction between different system components for its realization.

6.1 Real-time Monitoring

On the user interface dashboard, individual cards corresponding to each monitored machine are presented. In figure 6.1, this page can be seen with a card for the machine `5. MACHINE_STAMPING`. On each card, sensor information is displayed with the possibility of

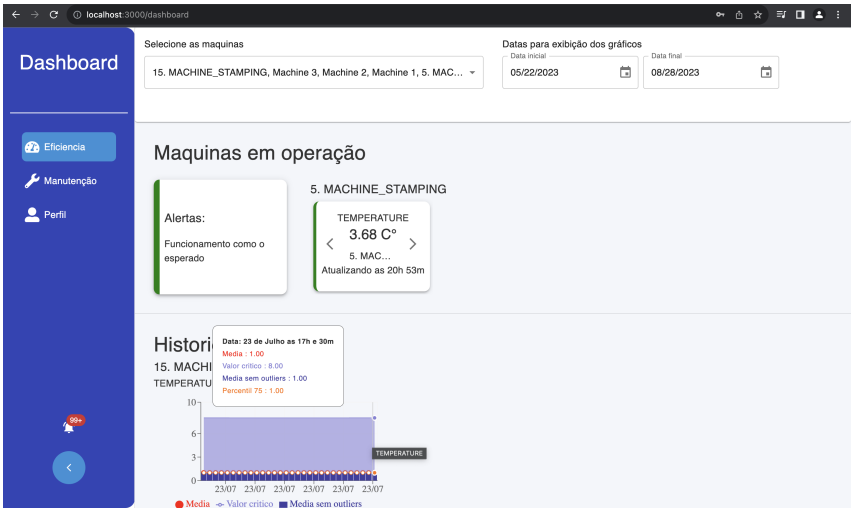

Figure 6.1: Dashboard with real time and graph data.

navigating between them with a directional arrow. The card can be seen in figure 6.2.

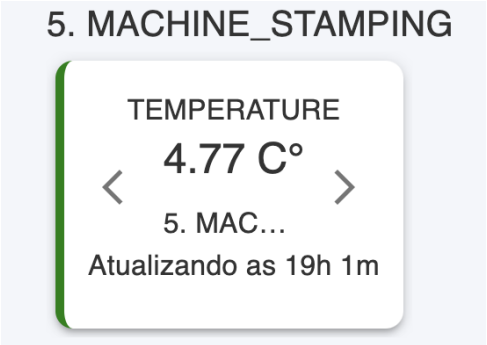

Figure 6.2: Card with machine data.

The acquisition of this real-time data is carried out through a continuous data stream. The frontend of the application makes a request to the endpoint `iot/realtime`, which returns this real-time data stream. The provision of this information occurs as explained in 5.5.3.

In the backend layer, the data stream is constantly fed by the `SensorValue` class, which in turn, is updated by the data receiving module. Upon receiving new sensor readings, the module performs appropriate validations before updating the values in the `SensorValue` class, as explained in 5.2. Once updated, the API accesses these new values and inserts them into the data stream transmitted to the connected user.

6.2 Alerts and notifications

The main objective of this feature is to monitor the performance of the machines in real time and issue alerts and notifications to users if inappropriate operating conditions are detected. This allows corrective actions to be taken immediately.

Whenever a new sensor reading is received by the data receiving module, a validation is performed to check if the machine is operating within acceptable parameters. These parameters are obtained through the system metadata, loaded at the initialization of the API, explained in 5.4.1.

If a value outside the acceptable range is identified during validation, the data receiving module inserts a special marking on this reading within the `SensorValue` class. This marking is later transmitted to the frontend during the data streaming process by the **IOT Analytics** module of the API, as explained in 5.4.4.

Upon receiving a marked reading, the frontend updates the corresponding information card to reflect the anomalous state. Specifically, within the dashboard shown in figure 6.1, the color of the card is changed to red or yellow, both on individual cards, figure 6.2, and on the general card that is used to show the overall status of the machines, figure 6.3, thus serving as an immediate visual alert for the user.

The data receiving module, **??**, keeps track of the operational state of each machine that is sending information. When a machine exits an alert state, the occurrence is recorded in the database, with the start and end date of the alert state, along with information related to the sensor and the involved machine, as explained in 5.4.2. Subsequently, a message is sent to the connected users using the *WebSocket* interface, informing them of the machine's malfunction during the period recorded by the system.

Information about alert completion can be accessed by users through notifications in the system interface, as can be seen in figure 6.4. By clicking on the notification icon in the dashboard's side menu, the user can view these notifications.

Therefore, when starting a session in the system, previous notifications are loaded for the user. In addition, any new notification generated during the user's session is

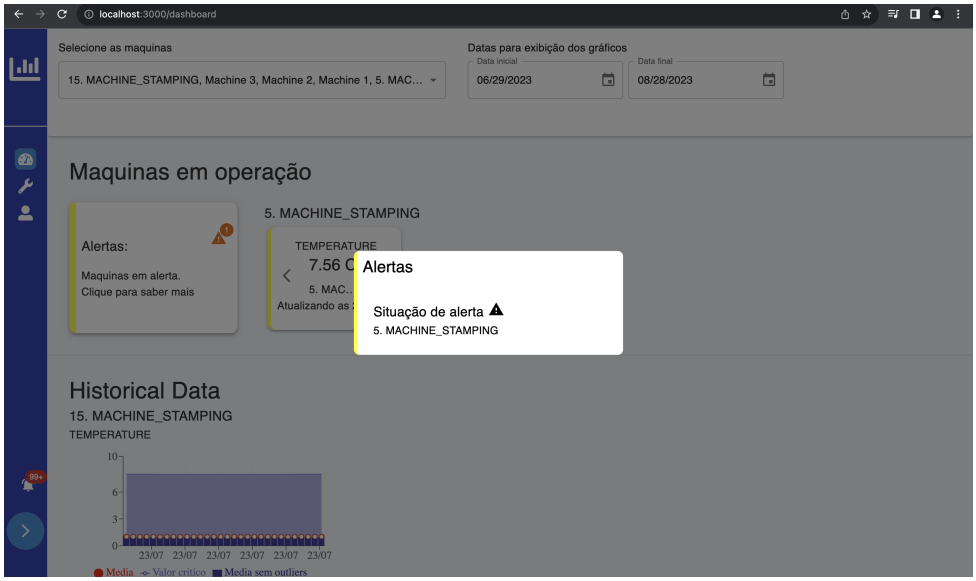

Figure 6.3: General Notifications.

Figure 6.4: Notification drawer.

transmitted in real time through a WebSocket connection established between the frontend and the backend, as explained in 5.4.2.

6.3 Statistical Analysis of Historical Data

The data processing module is responsible for the statistical analysis of historical data. Following the system's standard parameter, every day at midnight, the raw unprocessed data is read and analyzed. The result of the analysis is then stored in the database for future queries, as explained in 5.3.

The possibility to access these analytical data is provided by the API, specifically by the "IOT Analytics" module. The request to obtain this information must be authenticated, as described in section 5.4.4.

The visual representation of these analyses is carried out through graphs that aggregate four main metrics resulting from the processing. Thus, the graph displays:

- *Ideal Value*: Represented in an area chart, it serves as an ideal operating parameter to provide a perspective relative to the other data.
- *75th Percentile*: Displayed in a line chart, this metric offers a view on the distribution of values during the aggregation period and its evolution over time.
- *Aggregation Average*: Represented in a scatter chart, this metric provides the average value of the aggregated data.
- *Average with Outlier Removal*: Illustrated in a bar chart, this metric is calculated after the removal of outlier values, as determined by the boxplot construction method.

Therefore, image 6.5 shows how these information are displayed within the dashboard.

The visualization of these graphs can be filtered by date, allowing the selection of the start date and the end date for the data to be displayed. When these fields are changed and the "apply filter" button is clicked, a new request is sent and the data is loaded according to the specified period. By default, when the page is first loaded, the system

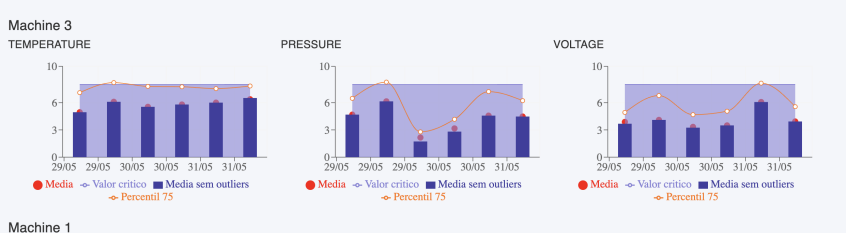

Figure 6.5: Graph example.

sends a request seeking data from the last 30 days, and it is up to the user to use the filter to view a different period. The date filter can be seen in figure 6.6.

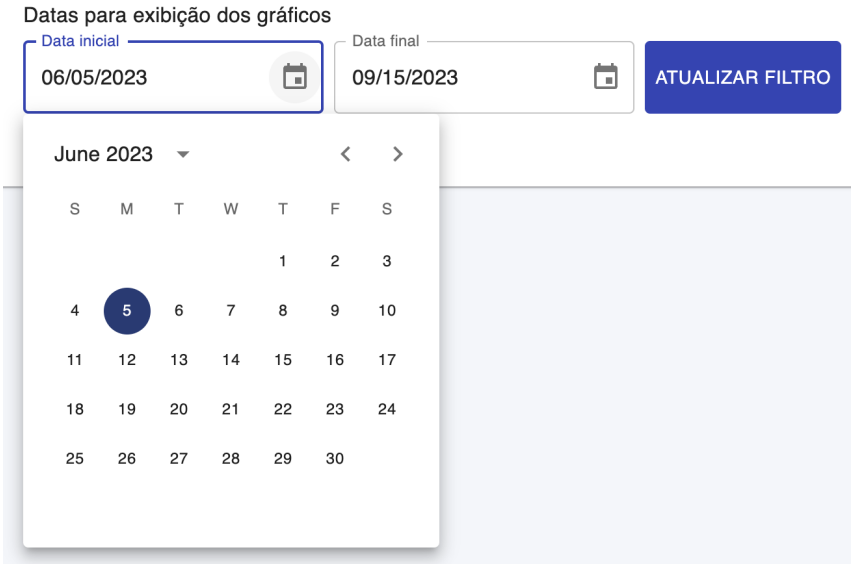

Figure 6.6: Date filter.

6.4 Display of data related to machine downtime

This feature is dedicated to the display of data related to machine downtime, previously stored in the database. The functionality aims to demonstrate how this data would be presented if it were received by the system in a manner similar to sensor data. The source of the data for the preparation of these graphs comes from three spreadsheets received at the beginning of the project development, where the information was saved

in the database. This data is accessed by the frontend through the `downtime_analytics` module of the API.

The graphs generated to represent these data are presented in the form of column charts. Each chart displays a title corresponding to the spreadsheet from which the data were extracted. The chart legend indicates the percentage that each specific stop represents in relation to the total stops.

Column charts provide an effective way to compare the different machine stops, as can be seen in figure 6.7. This visualization offers a means to analyze operational efficiency, identify possible areas for improvement, and track the outcome of actions taken in relation to machine downtime and predictive maintenance.

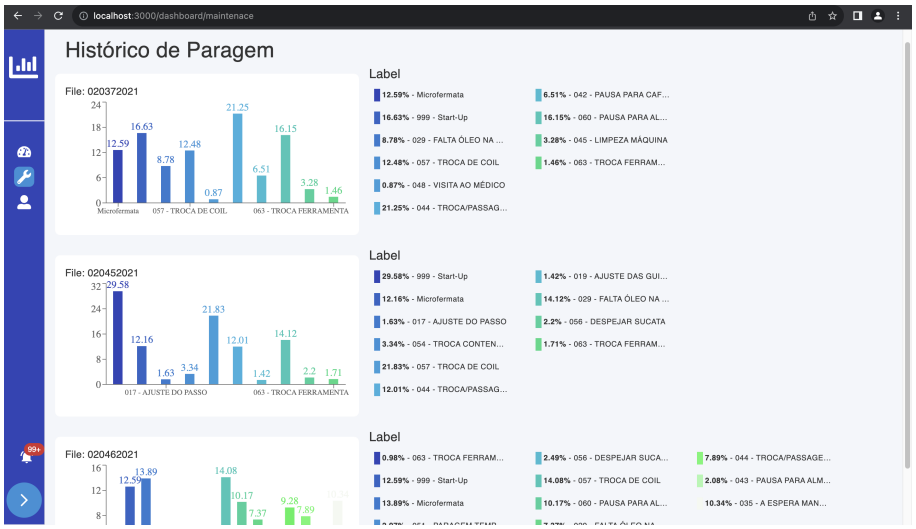

Figure 6.7: Downtime graph.

6.5 User Profile

The user profile functionality was developed with the aim of providing users with access to their personal data stored in the system. In addition to viewing this information, this screen also allows for modifications, including password changes, email, and other personal data such as name, surname, and description.

To read and modify the displayed data, the *User* module of the API is used. This module provides endpoints that allow access and modification of the stored data, ensuring that the information is updated according to the user's interactions, as can be seen in figure 6.8.

An additional feature provided by this screen is the logout option. By selecting this option, all client-side stored information is erased, thus ensuring the user's secure exit from the system.

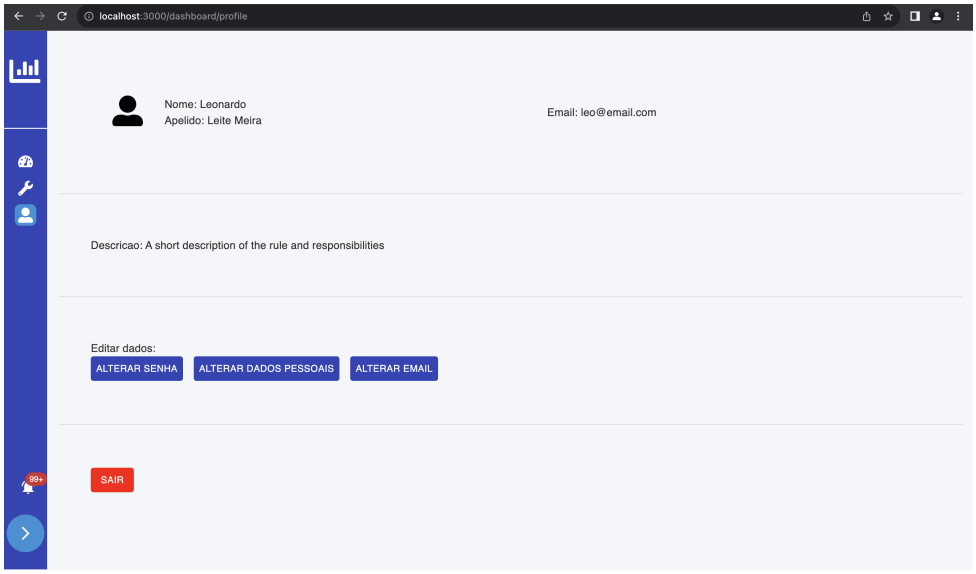

Figure 6.8: Profile page.

Chapter 7

Results and Evaluation

In this chapter, an evaluation of the developed system will be presented, focusing on its benefits and competitive advantages, as well as the results of tests and validations carried out. The system can be modified in such a way that it can be adapted to other contexts, where information generated by sensors needs to be visualized in real time, and where the processing and analysis of historical data are required.

The implementation of this system can assist in the quicker identification of production problems, thanks to real-time monitoring. In addition, the insights generated by the historical data analysis can be instrumentalized to optimize production processes **raczSzabo2020realTime**.

The adoption of a system like this represents a competitive advantage, potentially raising the company's efficiency in relation to the competition **ng2011realTime**. However, it is important to note that the software has not yet been implemented in a production environment, and therefore, there is an unmapped set of improvements that must be made for the system to operate as effectively as possible.

7.1 Benefits

A series of benefits is offered by the developed system. Firstly, comprehensive monitoring of machines throughout the industrial plant is facilitated. Through this monitoring, crucial information about the operational status of each machine is displayed to employees in real time.

Additionally, the system provides the ability to take quicker actions in case of operational issues, and as a result, the downtime of machines can be reduced, mitigating losses associated with production.

Another advantageous aspect lies in the evaluation of the effectiveness of the implemented maintenance measures. Through historical tracking, the system allows for the visualization of the effects of actions taken for machine maintenance, thus, more informed and effective decisions can be made quickly, extending the benefits to other machines in the plant.

Finally, the system's historical data also contributes to the generation of insights from its analysis. This analysis can provide a new perspective for monitoring machine performance, and operational anomalies can be identified and specific corrective measures can be implemented to improve certain indicators.

7.2 Competitive Advantages

The developed system offers a set of competitive advantages that are mainly manifested when contrasted with companies that do not implement a similar solution. Initially, the lack of an adequate monitoring system can result in operations below the efficient potential for a company. This scenario generates additional costs in maintenance, equipment loss, and even waste of raw materials.

On the other hand, the implementation of a robust monitoring system provides benefits to the efficiency of production, maintenance, and development of products and services. Real-time monitoring and analysis of historical data allow the optimization of various

operations, from the quick identification of problems to the implementation of corrective and preventive actions.

Thus, the quality of products or services is significantly improved, as the information provides data-driven management, thereby facilitating a leaner and more efficient production, which reduces overall costs and, consequently, can make the product or service more competitive in terms of price **glowalla2014processDriven**.

Therefore, the competitive advantages generated by the use of this system are multifaceted, encompassing not only operational efficiency, but also the quality and cost of products and services. These joint improvements enable the company to acquire a more solid and advantageous position in the market in which it operates.

7.3 Conducting tests and validations

Although all the requirements raised for the system have been met and the detailed user stories have been completed, a significant evaluation of the system in a production environment has not yet been carried out. The importance of testing and validating a software system in a real environment cannot be underestimated **leTraon1999selfTestable**, as conducting tests is fundamental to assess the system's suitability to practical needs, while validation ensures that the system meets the established requirements.

The development of software systems is an iterative process that involves a series of steps, including requirements, design, implementation, testing, and maintenance. Each of these steps requires review and adjustments based on the results of tests and validations **coleman2006softwareProcess**. The lack of a rigorous testing and validation stage can result in various deficiencies, both technical and functional, which can not only affect the system's performance but also render it impractical for use in a production environment.

Without a proper series of tests, the system is susceptible to failures that can be both technical and related to functional requirements. These failures may be minor, but they have the potential to escalate and compromise the system's integrity. Thus, the absence of tests and validations in a real environment represents a significant gap that must be

addressed to ensure the system's robustness and effectiveness.

Chapter 8

Conclusion and Future Work

This chapter is structured into the sections "Project Summary", a discussion on the "System Limitations", and finally, "Suggestions for Future Work".

The system construction is considered successful, serving as an initial milestone for future implementations and adaptations. The next steps for the advancement of this project include placing the system in a production environment, followed by the collection of user feedback. This approach will allow an incremental learning process, in which the system will be constantly improved based on the experiences gained and the needs identified.

8.1 Summary

In this dissertation, a multifunctional system was developed for the collection, storage, processing, and visualization of data generated by sensors. Python was used for the creation of the API and the processing module, MongoDB for database management, and NextJs for the construction of the control panel, known as *dashboard*. The system architecture can be seen in 4.

The data were received through a multicast connection established by the data receiving module, shown in 4.1.1. Once received, the data were immediately subjected to a preliminary analysis to identify any condition that could trigger an alert. If an alert was

generated, users were notified, allowing for quick and effective interventions.

For the analysis of historical data, the BoxPlot method was employed in the processing module, explained in 4.1.2. This analysis aimed at identifying patterns and anomalies in the data collected over time, providing valuable insights for the operation and maintenance of the monitored machines.

The API, described in 5.4, played a crucial role in the system, managing access to the data. Security was ensured through the implementation of JWT authentication, and several *endpoints* were developed to allow effective access to the data.

The *frontend*, described in 5.5, in turn, was responsible for displaying real-time information, generated alerts, and processed data in the form of charts.

8.2 System Limitations

Despite the advancements achieved with the development of the system in question, some limitations were identified that could influence its effectiveness and applicability in different contexts.

Firstly, it is identified that the data receiving module needs to be specifically adapted for each operational context. This requirement may compromise the system's portability, requiring manual adjustments whenever a new application is considered.

Secondly, there are restrictions regarding the structure of the received data, as for the effective operation of the data receiving module and the alert functionality, it is necessary that the received data have an identification field and a corresponding numerical value. The absence of the latter prevents alerts from being identified and makes the processing module ineffective for the analysis of these data.

Lastly, the system was designed and tested in an environment with a limited number of connected machines. No tests were carried out to evaluate the system's performance under the load of a large number of machines and sensors sending data simultaneously. Therefore, for larger scale scenarios, adaptations may be necessary to ensure the system's performance and effectiveness.

8.3 Suggestions for Future Work

Based on the observations and analyses carried out throughout this project, there are several works that can be done on the system for future research and development.

8.3.1 Use of artificial intelligence for prediction

Given that the data read by the sensors are stored in the system, they have the potential to reveal significant information about the operation of the machines. The application of AI techniques to the collected data was identified as the main functionality for the evolution of the system, which can become a robust tool for predictive maintenance. By applying machine learning algorithms to the stored data, predictive models can be generated that anticipate failures or inefficiencies in industrial equipment.

The implementation of a predictive maintenance system based on AI could confer a significant competitive advantage to the company, not only would it improve operational efficiency, but it would also optimize the allocation of resources for maintenance, resulting in cost reduction and increased productivity. Therefore, the future exploration of AI for the analysis of stored data is strongly recommended to enhance the effectiveness of the system under study.

8.3.2 Monitoring and Logs

Comprehensive system monitoring and activity log maintenance are crucial aspects for system sustainability and scalability. The absence of a well-structured log system can result in difficulties in identifying and resolving issues that may arise during real-time system operation. In this context, three main areas are identified where monitoring and logs could provide valuable insights for continuous improvement.

Moreover, it would be advantageous to maintain a comprehensive record of data transactions between the sensors and the data receiving module. These logs could include information such as the date and time of the transaction, sensor identification, and any anomaly or failure during the receiving process. This would facilitate the verification of

the integrity of the received data and assist in early detection of potential hardware or network connectivity issues.

Log for Statistical Analysis

The data processing module, responsible for statistical analysis, would also benefit from a log system. Details about the execution of the BoxPlot or any other statistical analysis could be recorded. This includes information such as the number of data points analyzed, any detected outliers, and the time taken for the analysis execution. With this information, it would be possible to further refine the analysis algorithm and identify areas for optimization.

8.3.3 Dynamic Parameters for Alerts

In the current version of the system, the parameters responsible for triggering alerts are defined statically, embedded directly in the source code. This approach, although functional, presents limitations in terms of flexibility and adaptability to different operational scenarios.

In the current configuration, any change in alert parameters requires a direct intervention in the code, followed by a testing and deployment process, which can be both time-consuming and prone to errors. Moreover, the lack of flexibility limits the company's ability to quickly adapt to new operational conditions.

It would be advantageous to allow alert parameters to be configured dynamically, through the system's user interface. A feature that allows users to adjust alert thresholds and other related parameters could be implemented. The possibility of making these adjustments in real time, without the need to interrupt the system operation, would represent a significant advancement in system usability and adaptability.

With the implementation of dynamic parameters, users could respond more quickly to changes in operational conditions, such as variations in machine workload or updates in security policies. In addition, this flexibility would increase the system's portability,

facilitating its deployment in various industrial environments with distinct requirements.

8.3.4 Generalization to Other Contexts

The developed system was initially designed for a specific industrial environment. Although effective in this context, the direct transfer of the system to other industrial areas may not be trivial. Therefore, the generalization of the system to other contexts is identified as an area of interest for future work.

The data receiving module currently requires specific customization for each industrial context. In addition, the system was designed to analyze data that contains an identification field and a numerical value. The lack of these fields could hinder or make it unfeasible to adapt the system in environments that require the handling of different types of data.

Future research could explore methods to make the data receiving module and the processing module more flexible and adaptable to different types of data and structures. Machine learning techniques or advanced statistical methods could be applied to automate the detection of anomalous events in different scenarios, without the need for extensive manual programming.

The ability to adapt the system to different industrial contexts would not only increase its applicability, but could also lead to improvements in the efficiency of industrial operations in a variety of sectors. This is particularly relevant in a scenario where Industry 4.0 and the Internet of Things are gaining momentum **nagy2018roleImpact**, and real-time data analysis is becoming increasingly critical **glowalla2014processDriven** for business competitiveness.

8.3.5 Performance Optimization

The system's ability to scale and operate efficiently under high load has not been extensively tested. In particular, there are concerns about performance when a large number of machines are connected and sending data simultaneously, as well as the frontend's ability to display multiple real-time data.

The system has not yet been put into production, so it has also not been evaluated in an environment with high traffic, both in terms of connected machines and users accessing the dashboard. Therefore, the challenges associated with scalability, such as latency in data receipt and potential bottlenecks in the database, are still unknown.

The frontend, built in Next.js, has the potential to become a bottleneck area, especially when displaying real-time data for multiple machines. The use of newer technologies, such as Server Components in newer versions of Next.js, could contribute to more efficient rendering and better performance.

Improvements in the API and processing module are also considered to enhance the overall efficiency of the system. Caching techniques, load balancing, and database query optimization are some of the strategies that can be explored if performance issues occur.

To validate any implemented improvement, performance tests, high traffic simulation, and real-time monitoring are required. These tests can provide objective metrics to assess the effectiveness of optimizations and identify new areas for improvement.

The system's performance optimization is a priority area for future work, aiming to ensure that it can operate effectively under various load conditions, both in terms of data input and user interaction.

8.3.6 Technology Updates

Given the ever-evolving nature of software development, upgrading to newer technologies is something that cannot be neglected. In particular, newer versions of Next.js, specifically versions 13 and 14, offer features that could substantially improve the performance and efficiency of the system.

One of the most promising features available in the newer versions of Next.js is the concept of Server Components **nextjsServerComponents**. These components allow for more efficient rendering of user interface elements, as they are processed on the server and sent to the client as pure HTML. This reduces the load on the browser and can significantly improve the speed and efficiency of the application.

In addition, the new architecture offers more robust opportunities for caching. This is particularly useful in the context of this system, where the processing module runs at specific intervals, therefore graphs and other visual elements can be cached on the server, optimizing the user experience when accessing updated data.

It is important to note that technology updates like these require a transition period and rigorous testing to ensure that compatibility between the different elements of the system is maintained. Therefore, a well-structured migration plan and testing phases are essential for successfully implementing any update.

8.3.7 Deployment and Feedback in the Factory

A critical step for the validation and continuous improvement of the system is its deployment in a real industrial environment, preferably a factory with operations that align with the context for which the system was designed.

Deployment in a factory environment offers the opportunity to collect direct feedback from end users and stakeholders. This feedback is not only instrumental in identifying areas for immediate improvement, but also provides insights into how the system fits into daily operations and the organization's long-term goals.

The advantage of collecting real feedback lies in the ability to make incremental adjustments to the system. These adjustments can range from fixing minor bugs to more significant modifications that can enhance the system's effectiveness. The tuning process is crucial to align the system with the users' needs and expectations, as well as to optimize performance.

In sum, the deployment of the system in a factory environment is not an end, but rather a vital step in a cycle of continuous development and improvement. Collecting real feedback and the ability to make incremental adjustments are key to ensuring that the system is efficient in a real industrial context.