UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Escola Politècnica Superior d'Enginyeria de Manresa

MASTER THESIS

Incident validation and remote monitoring tools for a bioleaching plant.

Master in Natural Resources Engineering
2022/2023

Author: Rubén del Olmo Arjona.

Director: Marta Isabel Tarres Puertas.

Co Director: Albert Comerma Montells.

October 5th, 2023.

Acknowledgements

Special thanks to Jose Antonio Caro for providing technical support in areas where my expertise was limited but crucial to the project's success.

Abstract

An overview of the development process of a web application designed to facilitate remote monitoring and control of a bioleaching chemical plant is presented in this master's thesis. Copper is recovered from electrical devices by exposing them to oxidizing bacteria in this plant. The application was designed to meet the specific needs of plant technicians and managers, who wanted remote access to sensor data, control of actuators, and alerts for any process anomalies.

This system consists of a web application that can be accessed from any web browser, ensuring accessibility and flexibility across multiple platforms. An application programming interface (API) powered by Spring Boot (Java) serves as the core of the system, providing access to this valuable information from multiple user interfaces and for future applications. Communication is maintained between the server and the chemical plant controller, which orchestrates chemical processes, activates pumps and solenoid valves, transmits sensor data, and accepts remote commands. Furthermore, an intuitive user interface is provided by a React (JavaScript) front-end that enhances the system's accessibility.

An email notification system was implemented to meet the alerting requirements, allowing users to enable notifications according to their preferences. A system notification function was also added.

It is a significant step toward improving the efficiency and reliability of bioleaching chemical plant operations. In addition to addressing the immediate needs of plant personnel, this work lays the groundwork for future advancements in remote monitoring and control.

Resum

Aquesta tesi desenvolupa el procés de creació d'una aplicació web dissenyada per facilitar el seguiment remot i el control d'una planta química de biolixiviats. En aquest procés químic, el coure es recupera dels dispositius elèctrics mitjançant l'exposició d'aquests a bacteris oxidants. L'aplicació a set dissenyada per satisfer les necessitats específiques dels tècnics i gestors de plantes, que volien accés remot a les dades dels sensors, control dels actuadors i rebut d'alertes per a certes anomalies del procés.

Aquest sistema consisteix en una aplicació web a la qual es pot accedir des de qualsevol navegador web, garantint l'accessibilitat i la flexibilitat a través de múltiples plataformes. Una interfície de programació d'aplicacions (API) alimentada per Spring Boot (Java) serveix com el nucli del sistema, proporcionant una visió de l'estat de la planta des de múltiples interfícies d'usuari. La comunicació es manté entre el servidor, l' API i el controlador de la planta química. L'aplicació permet activar bombes i vàlvules de solenoides, rebre i transmetre dades al sensor. A més, una interfície d'usuari intuïtiva és proporcionada per React (JavaScript), millorant l'accessibilitat del sistema.

Tanmateix, s'ha implementat un sistema de notificació per correu electrònic per complir els requisits d'alerta, permetent als usuaris habilitar notificacions segons les seves preferències.

Es tracta d'un pas significatiu cap a la millora de l'eficiència i fiabilitat de les operacions de la planta química. A més d'abordar les necessitats immediates del personal de les plantes, aquest treball estableix les bases per a futurs avanços en la vigilància i control remots.

Table of Contents

Table of Figures

Part I.

THESIS

1. Introduction

As a transformative force in today's industrial landscape, cutting-edge information technologies have been integrated into traditional processes. Autonomous systems have become an important component of industrial progress as they strive for increased productivity, operational efficiency, and comprehensive control.

Developing an autonomous digital system is the focus of this master's thesis, which explores a technological frontier. We explore the development of a bioleaching plant, a dynamic entity that undergoes continuous improvement. Metals are cyclically extracted through biological processes and microorganisms, a more environmentally friendly method than conventional methods. In contrast to traditional methods, it operates at ambient temperatures, avoiding energy-intensive furnaces.

To bridge the information gap between laboratory technicians and bioleaching plant personnel, our digitalization effort is urgent. As a result of this challenge, this project develops a web application that will allow technicians to monitor the plant remotely through a web application. These professionals require real-time insights and comprehensive control over the chemical process.

This endeavor represents a prime example of a Cyber-Physical System (CPS), where the synergy of digital technologies and physical processes becomes apparent. As the basis of the system, a virtual model of the bioleaching plant is created. In addition to seamlessly communicating with a server, this digital twin also provides a versatile user interface. Additionally, it allows the server to transmit information and directives continuously to the chemical plant, enabling real-time monitoring and proactive intervention.

In addition to being a real-world application, this project is poised to redefine the trajectory of bioleaching plants. Laboratory technicians are motivated to pursue this pursuit to gain timely and comprehensive insight into the chemical process. This digital tool allows them to manage bioleaching operations more efficiently and with unprecedented control.

2. Objectives

The objectives aim to address the specific needs and concerns raised by lab technicians. A comprehensive digital solution will be built within the plant's operating framework to enhance user experience and informed decision-making while extending control management. To optimize plant performance and to ensure its safe and efficient operation, these objectives focus on secure access, real-time data monitoring, proactive event management, and data visualization.

The following is a detailed breakdown of each objective:

1. **Improve Control Management:** Develop and implement a comprehensive digital solution that extends the capabilities of a chemical plant's control management. Pump and solenoid valve control management will be improved, data will be securely stored, notifications of critical plant events will be sent out, and administrators with specific control privileges will be able to manage users.

2. **User-Centric Secure Access:** Develop a user-friendly web application that ensures secure and private access to plant management functions via user accounts. The plant's operational elements will be accessible to all users, but administrators will be able to take specific actions.

3. **Informed Decision-Making:** Facilitate real-time data monitoring by implementing a control mechanism that records data at one-minute intervals. A variety of data formats are captured, including real numbers (such as pH and temperature readings) and integers (such as color sensor values). Informed decisions can be made based on the performance of the plant with the help of this feature.

4. **Proactive Event Management:** Establish a system that proactively identifies and responds to potential emergency situations within the plant, such as high tank levels, out-of-range pH values or extended high-level sensor activation. In this way, the chemical process can be operated safely and efficiently.

5. **Comprehensive Visualization:** Enable users to visualize critical plant parameters for each station, including the bioreactor, leaching stage, and electrowinning stage. pH, redox, voltage, current, tank levels, pressure levels, color indicators, temperature, pump and solenoid valve operation status, and pressure sensor indicators will be displayed by the application. To improve plant oversight, all sensor data will be presented with graphical trends.

6. **Data Accessibility:** Empower users to download and share the stored information from the database in easily readable formats, such as spreadsheet files compatible with programs like Microsoft Excel or LibreOffice Calc. Data accessibility and collaboration are enhanced by this feature.

7. **Enhanced Visualization:** Utilize a variety of graphical resources within the application to enhance the visualization of the plant's status. Through intuitive graphics, users can interpret solenoid valve statuses, pump statuses, and color sensor values, improving their understanding of plant operations.

3. State of the art

Information and communication technology (ICT) is currently undergoing rapid development. Many disruptive technologies, such as cloud computing, Internet of Things (IoT), big data analytics, and artificial intelligence, have emerged. These technologies are permeating the manufacturing industry and make it smart and capable of addressing current challenges, such as increasingly customized requirements, improved quality, and reduced time to market [1]. An increasing number of sensors are being used in equipment to enable this equipment to self-sense, self-act, and communicate with one another [2].

Regarding information and communication technology, researchers are attracted to IoT [3]. By adopting this essential technology, companies have become smarter, more competitive, automated, and sustainable in the global supply chain. In today’s competitive marketplace, supply chains are struggling as they compete with each other. Therefore, IoT devices are an effective way to authenticate, monitor, and track products using GPS and many technologies [4,5]. Industry 4.0 stands for the fourth industrial revolution **(IR 4.0)** in the digital age, it is associated with virtualizing real-world scenarios of production and processing without human intervention. This virtual world is linked to IoT devices,allowing the creation of cyber–physical systems **(CPS)** to communicate and cooperate [6,7].

Generally speaking, CPS is referred to as the system that can efficiently integrate both cyber and physical components through the integration of the modern computing and communication technologies [8, 9], aiming to changing the method of interaction among the human, cyber and physical worlds. CPS emphasizes the interactions between cyber and physical components and has a goal of making the monitoring and control of physical components secure, efficient, and intelligent by leveraging cyber components [10]. In CPS, “cyber” means using the modern sensing, computing, and communication technologies to effectively monitor and control the physical components, while “physical” means the physical components in the real world, and “system” reflects complexity and diversity. Based on the clarification, we can see that a CPS consists of multiple heterogeneous distributed subsystems [11]. Similar to the development of IoT, CPS has been developed in numerous areas [11, 12, 13], including smart grid, smart transportation, etc. As shown in [14], the CPS is the integration of physical components, sensors, actuators, communication networks, and control centers, in which sensors are deployed to measure and monitor the status of physical components, actuators are deployed to ensure the desirable operations on physical components, and communication networks are used to deliver measured data and feedback comments among sensors, actuators, and control centers. The control centers are used to analyze measured data and send feedback commands to actuators, ensuring the system operates in desired states [14, 15].

As mentioned above, the essence of CPS is the system, and the main objective of CPS is to measure the state information of physical devices and ensure the secure, efficient, and intelligent operation on physical devices. In CPS, the sensor/actuator layer, communication layer, and application (control) layer are present. The sensor/actuator layer is used to collect real time data and execute commands, the communication layer is used to deliver data to the upper layer and commands to the lower layer, and application (control) layer is used to analyze data and make decisions. In contrast, IoT is a networking infrastructure to connect a massive number of devices and to monitor and control devices by using modern technologies in cyber space. Thus, the key of IoT is “interconnection.” The main objective of IoT is to interconnect various networks so that the data collection, resource sharing, analysis, and management can be carried out across heterogeneous networks. By doing so, reliable, efficient, and secure services can be provided. Thus, IoT is a horizontal architecture, which should integrate communication layers of all CPS applications to achieve interconnection.

CPS systems are key elements in the implementation of the fourth industrial revolution (IR 4.0) [16]. Industry 4.0 is the network-enabled entity that automates the whole process of manufacturing, connecting business and processes. Market demands and the advancements in new technologies are transforming manufacturing firms’ business operations into smart factories and warehouses. Due to this automation, IoT devices are producing a massive amount of data daily, known as big data [17,28]. Statistics show that, at the end of 2021, there were more than 10 billion active IoT devices globally [19]. By 2030, the number of active IoT devices is expected to exceed 10 billion to 25.4 billion. By 2025, the data created by IoT devices will reach 73.1 ZB (zetta bytes) [20]. In 2020, the IoT industry was predicted to generate more than USD 450 billion, including hardware, software, systems integration, and data services. By the end of 2021, it reached USD 520 billion. The IoT industry is predicted to grow to more than USD 2 trillion by 2027 [21,22]. The increasing number of devices and the usage by humans shows the importance of IoT devices; moreover, the industry is growing and gaining revenue.

In the context of Industry 4.0, the integration of Cyber-Physical Systems (CPS) and the Internet of Things (IoT) has far-reaching implications, not only for manufacturing processes but also for enhancing the user experience (UX) during product development [23]. ISO 9241-210 defines UX as "a person’s perceptions and responses resulting from the use and/or anticipated use of a product, system, or service." Within Industry 4.0, two notable trends emerge: First, customers are actively involved in co-creating personalized products, emphasizing improved UX and satisfaction—a phenomenon known as mass personalization [24]. Second, products themselves become intelligent and capable of communication throughout their life cycles, aligning with the IoT paradigm [25]. Both of these aspects converge to enhance UX during product development.

Moreover, monitoring plays a pivotal role in the operation, maintenance, and efficient scheduling of Industry 4.0 manufacturing systems [26]. The widespread deployment of sensors enables smart monitoring, providing real-time data on various manufacturing parameters such as temperature, electricity consumption, vibrations, and speed. These data are not only visualized graphically but also trigger alerts when anomalies occur in machines or tools [27, 28]. CPS and IoT technologies are instrumental in facilitating smart monitoring within Industry 4.0 smart manufacturing systems.

In the broader landscape of Industry 4.0, research spans several key dimensions:

1. Smart Design and Manufacturing: This encompasses areas such as smart design, prototyping, controllers, and sensors [29, 30]. Real-time control and monitoring are crucial for realizing smart manufacturing [31]. Supporting technologies include IoT, 3D printing, industrial robotics, and wireless communication [32], all of which contribute to the efficient and responsive design and production processes.

2. Smart Decision Making: At the heart of Industry 4.0 lies smart decision making. The widespread deployment of sensors aims to enable intelligent decision-making through comprehensive data collection [33]. Achieving this requires real-time information sharing and collaboration [33]. Technologies like CPS, big data analytics, cloud computing, modeling, and simulation are pivotal in enabling data-driven decision-making processes, including data-enabled predictive maintenance and data-driven modeling [34, 35, 36].

The integration of CPS and IoT technologies within Industry 4.0 not only revolutionizes manufacturing processes but also opens doors to enhanced user experiences during product development [23]. Moreover, these technologies facilitate real-time monitoring, smart design and manufacturing [29, 30], and data-driven decision-making [34, 35, 36], ultimately reshaping the industrial landscape for improved efficiency, productivity, and sustainability.

The development of a web application at the intersection of Cyber-Physical Systems (CPS) and the bioleaching plant represents a significant step in enhancing bioleaching operations. This innovative web app serves as a bridge between the digital and physical domains, inspired by CPS principles. By seamlessly integrating modern computing and communication technologies, this application empowers users to efficiently monitor and control the bioleaching plant's physical components. It acts as a conduit between the digital interface of the web app and the real-world physical components, where microorganisms, sensors, actuators, and chemical processes operate collaboratively. Through this integration, the web app enables real-time data collection, analysis, and decision-making, aligning with the core objectives of CPS while streamlining bioleaching processes. It demonstrates how the synergy between digital technology and physical systems can contribute to more effective and intelligent operations in bioleaching.

4. Case Study

There are two main dimensions to our journey through this case study: the physical plant and the digital infrastructure that enables it. In this section, we will provide a high-level overview of the bioleaching process, including its stages and critical components. Our next step will be to look at the existing software technological components that form the foundation of this operation.

Plant Overview

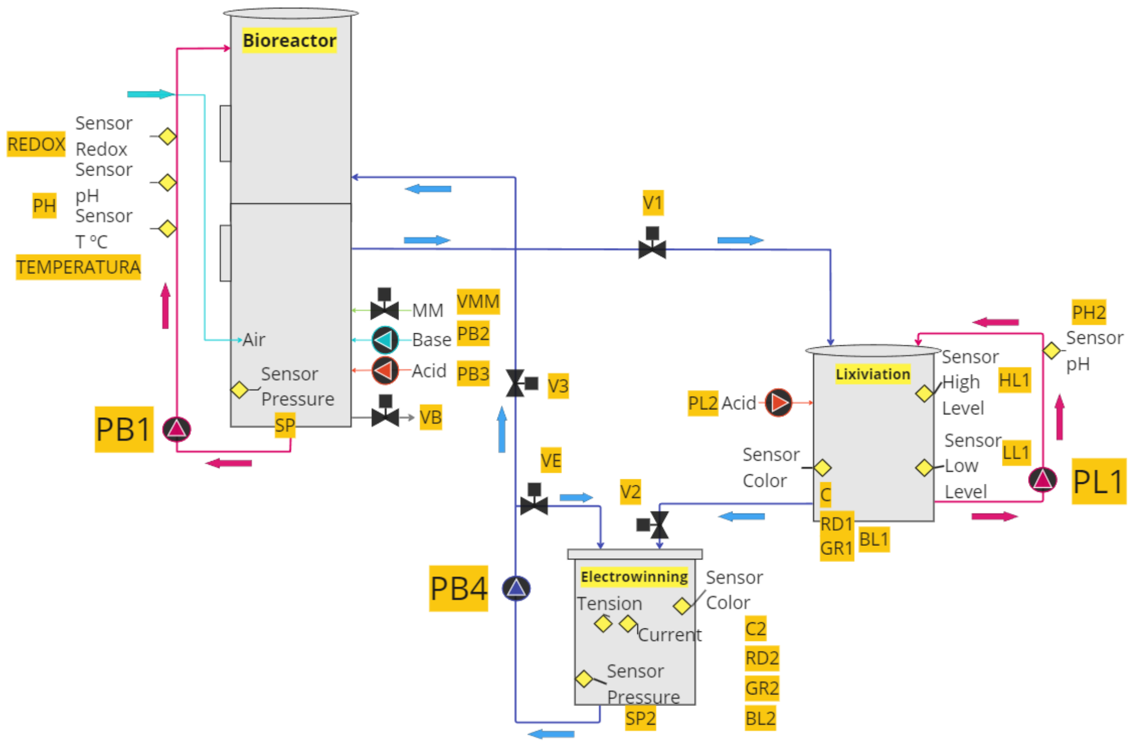


Figure 1: Plant overview. Own Source.

As part of the project plant, copper is recovered from electronic components using the catalytic action of oxidizing bacteria through bioleaching. As a foundation for the later chapters of this thesis, this section provides a high-level overview of the bioleaching process.

Plant Stations

Bioreactor

In the bioleaching process, the bioreactor is responsible for generating and maintaining the biological leaching agent. Through the use of pumps, solenoid valves, and sensors, it controls essential parameters, including pH, redox, pressure levels, and temperature.

Leaching

During the leaching stage, copper is dissolved. This is a crucial step in the extraction process. A color sensor indicates completion of this stage when the leaching agent changes color from orange to green.

Electrowinning

A color sensor monitors the process in the copper recovery stage, noting that the solution transforms from light green to dark green as the dissolved copper is reclaimed from the solution.

After applying low voltage / high current between the electrowinning electrodes, the metal is deposited, and the process is considered to be finished when a variation in color is detected. It is also under study if a variation in current could also trigger the end of the process.

Station Components

The bioreactor station consists of the following components:

- **Redox:** Redox Level Sensor
- **PH:** pH Level Sensor
- **Temperature**: Temperature Level Sensor
- **SP:** Pressure Sensor
- **PB1:** Recirculation Pump
- **PB2:** Supply Pump - pH Base
- **PB3:** Supply Pump - pH Acid
- **PB4:** Circulation Pump - Electrowinning Outlet / Bioreactor Inlet
- **VB:** Solenoid Valve - Bioreactor Reject
- **V3:** Solenoid Valve - Electrowinning Outlet / Bioreactor Inlet
- **V1:** Solenoid Valve - Bioreactor Outlet / Leaching Inlet
- **VMM:** Solenoid Valve - M. Medium

The leaching station consists of the following components:

- **PH2:** pH Level Sensor
- **C:** Composite number of all colors
- **RD1:** Red Color
- **GR1:** Green Color
- **BL1:** Blue Color
- **HL1:** High-Level Sensor
- **LL1:** Low-Level Sensor
- **PL1:** Recirculation Pump
- **PL2:** Supply Pump - pH Acid
- **V1:** Solenoid Valve - Bioreactor Outlet / Leaching Inlet
- **V2:** Solenoid Valve - Leaching Outlet / Electrowinning Inlet

An electrowinning station consists of the following components:

- **PB4:** Circulation Pump - Electrowinning Outlet / Bioreactor Inlet
- **VE:** Recirculation Solenoid Valve
- **V2:** Solenoid Valve - Leaching Outlet / Electrowinning Inlet
- **SP2:** Pressure Sensor
- **Tension:** Voltage Sensor
- **Current:** Current Sensor
- **C2:** Composite number of all colors
- **RD2:** Red Color
- **GR2:** Green Color
- **BL2:** Blue Color

The processes of the plant are controlled by authorized users and their designated controllers. This succinct description sets the stage for a more detailed examination of the bioleaching process in the following chapters.

Plant Existing Software

1. **Server:**
    - **Role:** Servers serve as intermediaries between the user and the plant. They save plant information and provide it to users. They also process administrative requests.
    - **Communication Protocol:** Communication with the mobile application occurs via the internet using HTTPS (Hypertext Transfer Protocol Secure).
    - **HTTPS Features:** HTTPS offers several key features:
        1. **Encryption:** It encrypts information exchanged between clients and the server, preventing eavesdroppers from understanding the content.

    2. **Integrity:** HTTPS ensures data integrity, preventing unauthorized modification during transmission.
    3. **Authentication:** Clients verify the server's identity using certificates, reducing the risk of man-in-the-middle attacks.
  - **Data Format:** The agreed format for information exchange is JavaScript Object Notation (JSON), a lightweight and easily readable format.
  - **JSON Structure:** JSON is structured around objects and vectors. An object consists of named sets of values enclosed in braces {}. A vector is an ordered collection of values enclosed in square brackets [].

2. **Plant Control Software Integration:**
   - **Current State:** The plant operates with its existing control software, responsible for collecting data from various sensors and controlling actuators.
   - **Independence:** The plant's software and the server can operate independently, allowing the plant's functionality to be maintained while the project's data can be accessed effectively
   - **Significance:** As a result of this integration, real-time data can be utilized for decision-making and analysis to achieve project goals.

By enhancing this structure, the plant software, server specifications, and communication protocols of the existing components are better understood, as are their critical roles, communication protocols, data format, and server integration.

The current existing server has Ubuntu 20.04 as the foundational layer of the system, a widely-used, stable operating system ideal for server environments. It provides a secure and performant foundation for all the other software. Additionally, Python 3.8.10 is used for certain operational purposes, offering a versatile and powerful programming language.NGINX, a high-performance web server and reverse proxy, version 1.18.0 its also part of the existing software and its used to serve static files to the browser efficiently and adeptly manages incoming HTTP requests, determining whether they should be handled internally (for static content) or proxied to another server (for dynamic content/API requests).

In the project architecture, NGINX serves the static HTML, CSS, and JavaScript files generated by the React build to the user interface, which is the front end library used in this project. In addition, the current project uses Spring Boot backend as the API server, Spring Boot manages business logic, database interactions, and other server-side functionality, simplifying the development of production-grade applications. By intelligently proxying API requests to Spring Boot's backend, NGINX ensures the decoupling of static and dynamic content delivery, increasing performance and scalability. The backend processes API requests, performs computations or data retrieval, and then sends the responses back through NGINX to the client-side React application.

Essentially, this architecture combines the rapid and efficient delivery of static files using NGINX with the dynamic and interactive data processing capabilities of Spring Boot. Through a unified access point managed by NGINX, the user interacts with a fast-loading, client-side React application, which communicates seamlessly with the backend via API requests. Separating concerns and optimizing static from dynamic content not only enhances the user experience by reducing load times and ensuring responsive interactions, but also facilitates scalability and maintainability.

5. Methodology

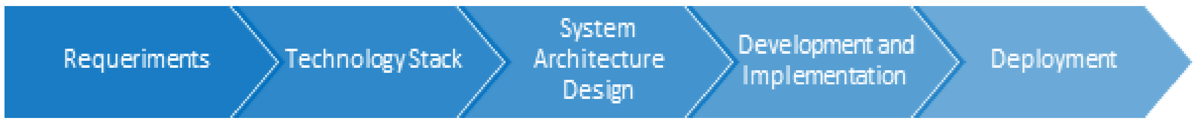


Figure 2*: Methodology process. Own Source.*

To achieve the objectives of this master's thesis, a structured methodology will be employed, encompassing the following key stages:

1. **Requirement Analysis and Specification:** The initial phase of the project involves an in-depth analysis of the requirements for the web application. Detailed insights into the monitoring and control of bioleaching processes are obtained through consultation with laboratory technicians and plant managers.

2. **Selection of technology stack:** The technology stack, including front-end and back-end frameworks and tools, will be chosen after an assessment of available options and their alignment with project goals**.** The front-end will be built with React, the back-end will be built with Java and Spring Boot, and the design will be built with Bootstrap.

3. **System Architecture Design:** The architecture of the web application will be meticulously designed to ensure scalability, efficiency, and reliability. Considering factors such as real-time data processing, data flow, and security, the system will be divided into modular components. This phase involves the design of the database and the architecture of the system.

4. **Development and Implementation:** The actual development of the web application will commence following established best practices and coding standards. To foster collaboration, adaptability to changing requirements, and iterative development, agile development methodologies will be used. To manage code changes efficiently, continuous integration and version control will be used.

5. **Deployment and Hosting:** The web application will be deployed to a hosting environment that ensures scalability and high availability.

Using this structured methodology, the web application is developed with a systematic approach, focusing on attaining the project goals while adhering to software engineering standards and best practices.

Requirements Specification

Requirements can be found in chapter 2: Objectives.

These include a wide range of features, such as secure access to data, proactive event management, real-time data monitoring, comprehensive data visualization, and enhanced data access. The requirements are meticulously crafted to provide plant technicians, laboratory professionals, and administrators with an intuitive, efficient, and user-friendly experience.

It is based on these customer requirements that the web application is developed, aligned with the core objectives of safety assurance, control management enhancement, and data enlightenment.

Technologies and Tools

This web application was built using the following technology stack:

- **Programming Languages**: Java is used for the backend and JavaScript is used for the frontend.
- **Frameworks:** Spring Boot on the backend, React and Bootstrap on the frontend.
- **Database:** SQLite is used for data storage.
- **Development Tools:** IntelliJ IDEA, Visual Studio Code, and Git for version control.

After carefully evaluating various development options, the decision was made to develop this project as a web application that meets the project's goals and requirements.

The web app is cross-platform, so it can be used on Android, iOS, and other devices and operating systems. By eliminating the need to maintain separate codebases for different platforms, development complexity and costs are reduced. In addition, web app development is more cost-effective than native app development, allowing for efficient resource allocation.

As changes made to the server are immediately visible to users, web apps can be updated easily. By doing so, updates can be deployed quickly, and users always have access to the latest version of the application. Moreover, web apps can reach a broader audience since they can be accessed directly through web browsers without downloading from app stores.

Performance constraints: Web applications, especially those built using heavy frameworks or that are not optimized, may experience slower performance than native applications**.** Code execution can be less efficient due to the reliance on browsers.

1. Although modern web APIs and frameworks have expanded the capabilities of web applications, they do not provide the same level of access to **device-specific**

features as native applications do. Advanced camera controls, geofencing, or seamless background operations can be difficult or cumbersome to implement.
2. Web applications are heavily **dependent on browsers** for security, which vary in terms of their robustness and frequency of updates.
3. For applications that require high-end graphics rendering, such as games and 3D modeling applications, web applications may not provide the same **level of performance** and fidelity as native applications.

The development of native mobile or desktop apps was considered as an alternative to building a web app. With native apps, users have direct access to device-specific features and could potentially enjoy a smoother experience, but maintaining separate codebases for different platforms can be time-consuming and costly. Users may need to install software to use desktop applications, though they provide robust functionality on specific platforms.

However, considering the project's primary objectives, particularly the requirement for cross-platform accessibility, the decision to build a web application was made. In choosing a web application, the project aims to manage development resources effectively while providing an accessible and versatile solution.

To better understand the inner workings of the web application, we can break it down into two main components: the front end and the back end:

Front End: It's the part of the web application that the user interacts with directly. The front end is responsible for presenting information to the user and collecting input from them. It is the interface that you see and interact with in the browser. It includes elements such as buttons and forms.

Back End: In contrast, the backend is the part of the web application that users do not directly see. The backend plays a crucial role in making the front end work.

1. **Server Interaction**: It connects to an already existing server that manages requests from the front end and processes them. The server acts as a bridge between the front end and the various data sources.

So, in simple terms, while the front-end handles what the user sees and interacts with on the web app, the back-end manages the communication with a server and a database, ensuring that the correct information is delivered to the front end and any necessary updates are made behind the scenes. Users benefit from this division of responsibilities by experiencing a smooth and efficient web experience.

Front End

Introduction to React and JavaScript

In addition to its component-based architecture and efficient rendering through the Virtual DOM, React is a widely recognized JavaScript library for creating user interfaces. It was developed by Facebook. It is an excellent choice for developing interactive and dynamic web applications because it enables developers to create reusable UI components.

Web development is based on JavaScript, which runs in browsers and provides client-side interactivity. It can also be used on the server using technologies like Node.js. In the development of modern web apps, JavaScript's versatility and React's powerful UI capabilities make them perfect partners.

Justification for React and Javascript in the Project

Several key advantages justify the use of React in conjunction with JavaScript for this project:

- **Component-Based Development:** It is easy to create self-contained UI components in JavaScript due to React's component-based architecture, which aligns well with JavaScript's modular nature.
- **Efficient Rendering:** In addition to reducing direct DOM manipulation, React's Virtual DOM makes it easier to manage the web page's elements and interactivity using JavaScript, the web's scripting language.
- **Ecosystem Compatibility:** The JavaScript ecosystem offers a wide range of libraries and tools that integrate seamlessly with React. These include routing solutions like React Router, and data visualization libraries, allowing React to be used to enhance other applications as well.
- **Community Support:** Developers can access extensive documentation, tutorials, and a wealth of knowledge through both React and JavaScript communities.

Facebook and the open-source community ensure that React remains current and adaptable to changing web development standards through active maintenance and updates.

A strong alignment with project objectives, such as cross-platform compatibility, efficient component management, and real-time data handling, drives the choice of React and JavaScript. A robust web application for monitoring and controlling plant processes can be developed with this combination of versatility, performance, and maintainability.

Additional Alternatives: Vue.js and Angular

Web applications can also be built using two other notable alternatives:

- **Vue.js:**
  Like React, Vue.js is a progressive JavaScript framework that provides a clear, concise syntax and allows for component-based development.

- **Angular:**
  It provides powerful features such as two-way data binding and dependency injection for building web applications. Angular is a comprehensive JavaScript framework developed by Google. Angular is suitable for large-scale applications and projects that have complex requirements.

It is our intention to gain proficiency in a widely used framework known for its scalability and versatility that led us to use React for this project. The choice of React was driven by the desire to gain hands-on experience with a framework that holds considerable relevance in the modern web development landscape, regardless of whether it is React, Vue.js, or Angular.

Introduction to CSS and Bootstrap

The visual and interactive aspects of web applications are shaped by CSS and Bootstrap. A web page's overall aesthetics, colors, fonts, and layout are defined by CSS, the web's core styling language. Using it, HTML content can be transformed into engaging user interfaces that are visually cohesive. On the other hand, Bootstrap is a popular front-end framework that streamlines and enhances the process of web design and development. It offers a robust set of pre-designed UI components, responsive grids, and CSS stylesheets, making it a go-to choice for creating visually appealing and user-friendly web applications.

Justification for CSS and Bootstrap in the Project

Here's why CSS, Bootstrap, and React are crucial to our current project, both for enhancing the user experience and simplifying development.

- **Visual Consistency:** By defining a unified style guide for the entire project, CSS maintains a professional and cohesive appearance across all application screens.
- **Responsive Design:** This is especially important for our project, since Bootstrap's responsive grid system enables users to access and control bioleaching processes from a wide range of devices.
- **Time Efficiency:** Using Bootstrap's library of pre-designed UI components, you can create interactive interfaces much faster thanks to elements such as navigation bars, forms, buttons, and modals.
- **Customization:** Use of styles and components with Bootstrap to align with the branding and design requirements of the project. It is possible to create a unique and visually pleasing application by balancing out-of-the-box functionality with customization options.
- **Scalability:** Bootstrap's modular approach allows application functionality to grow while maintaining a consistent design, ensuring a smooth user experience. React, JavaScript, CSS, and Bootstrap together will scale efficiently.

Ultimately, the strategic use of CSS and Bootstrap alongside React and JavaScript is based on the objective of creating an aesthetically attractive, responsive, and user-friendly web application that monitors and controls bioleaching processes. By combining these technologies, we can accomplish our project goals efficiently and effectively, enhancing both the application's visual appeal and functionality.

Back End

Introduction to Java and Spring Boot

For decades, Java has been a cornerstone of software development due to its robust and versatile programming language. Spring Boot emerges as a powerful and efficient framework for building web applications when used in conjunction with Java, due to its portability, security features, and extensive library and framework ecosystem. Developers can focus on application logic rather than infrastructure concerns with Spring Boot's streamlined, convention-over-configuration approach.

Justification for Using Java and Spring Boot in this Project

There are several compelling reasons for choosing Java and Spring Boot for this project's backend development:

- **Robustness and Reliability:** In a project like this, which involves monitoring and controlling bioleaching processes, reliability is essential. Java is widely regarded as one of the most reliable and stable programming languages available.
- **Security:** To ensure the integrity of an application and safeguard sensitive data, Java's bytecode verification and runtime security checks are crucial.
- **Portability:** Because Java is a "write once, run anywhere" platform, it is accessible and compatible across a wide range of platforms and operating systems.
- **Spring Boot's Efficiency:** Through its opinionated approach to application setup and configuration, Spring Boot simplifies the development process. As a result of its comprehensive ecosystem of tools and libraries, our project will be delivered within a reasonable timeframe with a robust backend.
- **Community and Ecosystem:** It is well known that Java and Spring Boot have large, active developer communities. This translates into extensive documentation, support, and a wide range of third-party libraries, which can accelerate development and enhance functionality.
- **Scalability:** Its modular and organized architecture allows Spring Boot to scale well. As our project grows, we may need additional features or to increase processing capabilities.

Additional Alternatives

In addition to Java and Spring Boot, there are other alternatives for backend development, each with its own strengths:

- **Python and Django:** Django is a high-level web framework that excels at projects that require a straightforward and quick backend setup. Python is known for its simplicity and readability.
- **Node.js and Express.js:** Using Node.js, you can develop JavaScript-based applications on the server, while Express.js allows you to create lightweight, real-time applications.
- **Ruby on Rails:** Known for its developer-friendly syntax and convention-over-configuration approach, Ruby on Rails is a popular framework for building web applications.
- **PHP and Laravel:** Its elegant syntax and ease of use make Laravel one of the most popular PHP frameworks for web development.

In the end, Java and Spring Boot were chosen due to their proven track record, robustness, and suitability for building a backend that can support our complex requirements.

Tools versions and browser specifications

For the technical justification of the tools used in the project we'll focus on the key technologies, their versions, browser requirements, and relevant Internet of Things (IoT) protocols leveraged, as inferred from the provided code snippets.

Front-end Technologies:

React (18.2.0):

Browser Compatibility: The `browserslist` has been configured to support the latest versions of Chrome, Firefox, and Safari in the development environment, and >0.2% of all browsers in the production environment, excluding Opera Mini and dead browsers.

In essence, this configuration means that during development, the setup has focused on a few, most recent browser versions to streamline debugging, whereas during production, the setup will cater to a much broader and slightly older browser range for a wider audience. As a result, developer convenience and user accessibility are maintained at a good balance.

Bootstrap (5.3.0-alpha1) & React-Bootstrap (2.7.2):

Recharts (2.8.0):

Recharts, utilized for crafting intuitive, interactive charts, leverages D3.js under the hood, offering both power and flexibility in visualizing data while maintaining compatibility with React’s component architecture.

React-Router-Dom (6.16.0):

It allows for dynamic, client-side routing, which enhances user experience by reducing page reloads and ensuring intuitive navigation throughout the application.

Back-end Technologies:

Spring Boot (3.0.6):

Spring Boot Starter WebSocket:

Using Spring for WebSocket handles message-handling between the server and subscribed clients in real-time, which is pivotal for IoT applications.

Spring Boot Starter Data JPA:

Spring Data JPA facilitates the implementation of JPA data layers, aiding in the simplification of data access within the H2 database. It ensures that the repository layer is easy to create and manipulate, supporting a wide array of database operations without the necessity for verbose code.

Java (17):

Java offers platform independence, robustness, and ease of use in enterprise environments, thus providing a stable and secure back-end platform.

JWT-Decode (3.1.2) and Java-JWT (3.18.1):

JSON Web Tokens (JWT) present a compact, URL-safe means of representing claims to be transferred between two parties. These tools allow for secure, token-based user authentication, thereby ensuring that users are accurately identified and validated across requests.

IoT Protocols

STOMP WebSocket:

Simple (or Streaming) Text Orientated Messaging Protocol (STOMP), used alongside WebSocket, provides a communication standard that allows the front-end and back-end of the application to communicate via two-way messaging. STOMP over WebSocket for IoT allows to push messages from the server to the client whenever an event occurs, thereby ensuring real-time updates and interactive user interfaces that can promptly reflect changes in underlying data or system state.

System Architecture Design

System Architecture Schema

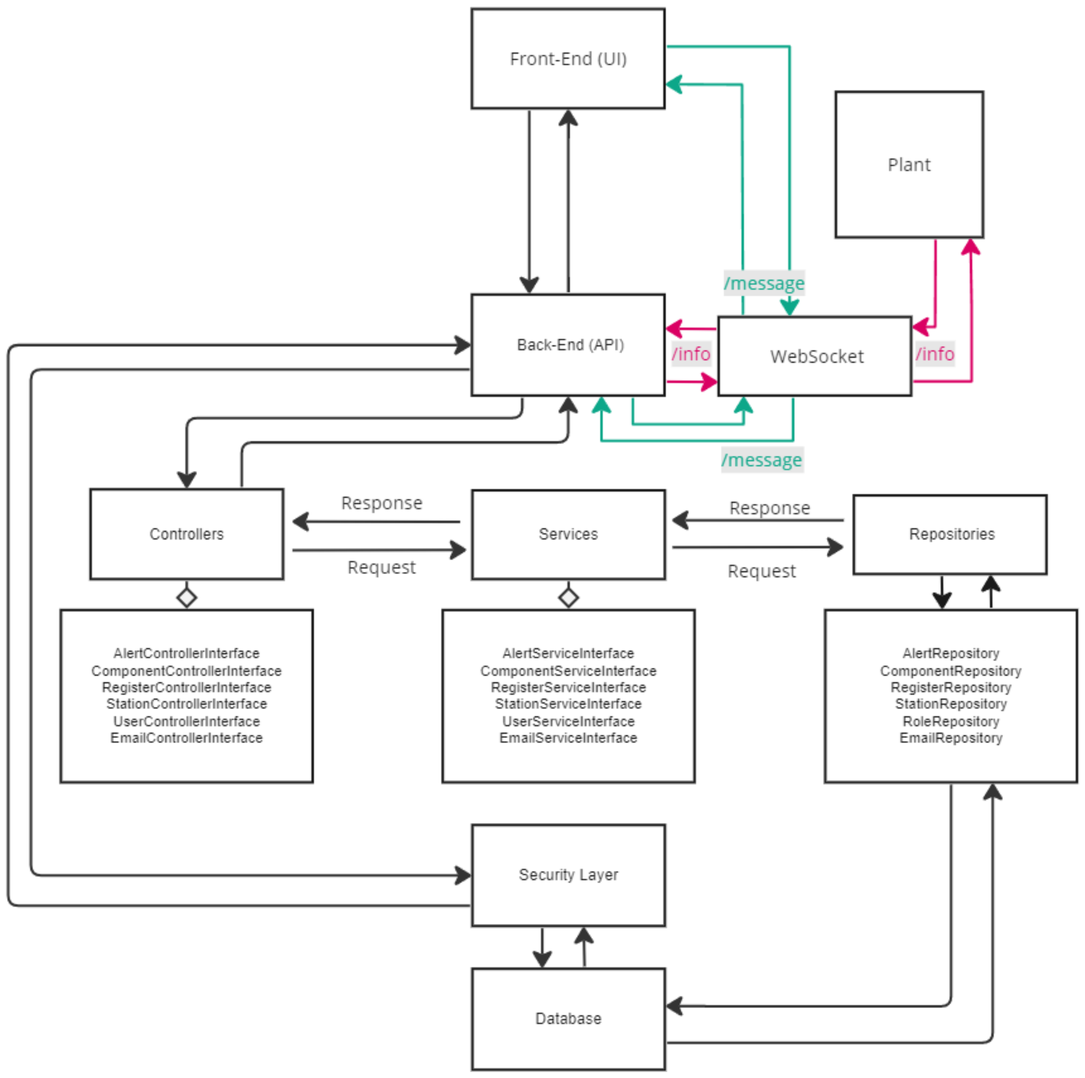


Figure 3*: System architecture schema. Own Source.*

- **Front-End (UI)**: The user interface, where users interact with the system.
- **WebSocket**: The server and the user interface communicate in real-time for live data updates. The server and the plant are also able to communicate in real-time. Websockets have been implemented to send alerts to the user interface from the back end, to send commands to the plant, and to receive status values of the components.
- **Back-End (API)**: Manages components, stations, users, alerts, and registers using controllers, services, repositories, and models.
- **Controllers**: Interact with services by handling incoming HTTP requests.
- **Services**: Develop both interfaces and implementations for the implementation of business logic and interaction with repositories.

- **Repositories**: Handle database operations. Implement both interfaces and implementations.
- **Security Layer**: Secures data transmission by authenticating and authorizing users.
- **Database**: User data, component data, alerts, and registrations are stored in a relational database.

Plant Organization

To understand how data is organized and managed within an application, it is important to understand the system architecture's hierarchical structure closely similar to the plant's physical layout and operational hierarchy. In this hierarchy, there are the following components:

Stations

A station, such as a bioreactor, a leaching tank or an electrowinning tank, is the highest level of the system.

Components

The system is composed of components that are the smallest units of operation. They each have their own unique parameters and functions that define their role within the system as a whole.

Components and Unique IDs

Unique component IDs play a key role in the application. By adopting this strategic approach, sensors in both bioreactors and leaching areas are associated precisely with specific components, enabling accurate data assignment and display. For example, pH sensors are assigned a unique ID, allowing accurate data assignment and display.
In order to ensure effective data management and intuitive user interaction, the web application's system architecture is hierarchical and component centric.

System Architecture Overview

The Model-View-Controller (MVC) pattern promotes a clean, organized codebase by separating concerns and ensuring modularity, scalability, and maintainability.

The system architecture comprises modules, services, repositories, and controllers that work together seamlessly to provide comprehensive control management for chemical plants.

1. Modules

Within a larger software framework, modules represent self-contained sets of features and functions.

As building blocks, these modules deliver different aspects of the functionality of the application. Each module serves a distinct purpose, and together they form the cohesive structure of the application.

The developer can concentrate on specific aspects of an application by encapsulating related functions in a module instead of feeling overwhelmed by the whole application.

In addition to simplifying development, this isolation enhances code readability and reduces the likelihood of unintended interactions among application components. Additionally, modules facilitate the organization of code, allowing specific functionality to be located and updated more easily. Modularity reduces the risk of introducing bugs and ensures that changes or enhancements to one part of the application will not disrupt the whole system.

With its unique purpose and functionality, each module of the web application contributes to its robustness, scalability, and maintainability. With these components, users can effectively monitor, manage, and control plant components and data as a cohesive and efficient framework.

Let's take a closer look at each module individually:

Alert Module: The alert module is responsible for managing alerts within the system. The alerts can be associated with specific components within a plant or with specific events within the plant.

Each alert instance includes the following attributes that are included in the Alert entity, which is the heart of the Alert Module.

- **id:** A unique identifier for each alert instance.
- **date:** Time and date when the alert was generated is represented by a timestamp.
- **message**: In the notification pop up and/or in the email alert content, this is the description of the alert that provides insights into its nature.
- **componentThing:** A relationship is established between this attribute and the ComponentThing entity to link alerts with specific plant components or events. This helps provide context about alerts and facilitate targeted responses.

Component Module
Components can include pH, agitation, color, solenoid valves, pumps, and many more. Furthermore, users with administrative roles can create new components and assign them seamlessly to existing stations, ensuring plant management flexibility and adaptability.

The Component Module revolves around the ComponentThing entity, which provides a structured approach to component management by encapsulating essential component information.

- **id:** Each component has a unique identification number that facilitates precise tracking and identification.
- **name:** Component names should provide clarity regarding their purpose or function.
- **description:** An explanation of the role or characteristics of a component in text format.
- **unit:** An attribute specifying the unit of measurement associated with the component's values, enhancing data consistency.

- **canAct:** This Boolean flag indicates whether the component can be activated or controlled within the plant.
- **valueType:** To ensure proper handling of data, a reference to the type of values associated with the component should be provided.
- **maxValue** and **minValue:** Operational thresholds are defined by the maximum and minimum permissible values for the component.
- **stationList:** It facilitates the organization and association of components with specific plant process areas by assigning them to specific stations.
- **registers**: Historical data tracking and analysis is enabled by a collection of registers associated with a component.
- **alerts:** Alerts associated with the component, notifying users of any issues or anomalies as soon as they arise.

Assigning new components to existing stations can be done by users with administrative roles, increasing the plant management system's flexibility and adaptability.

Register Module

Managing data registers associated with plant components, stations, or specified time intervals is the responsibility of the Register Module, which is a critical component of the web application. This is where the data will be placed and assigned to the component id in the register entity.

It is the Register entity that is at the core of the Register Module, capturing and managing critical data register information. It is composed of several attributes that all serve a particular purpose:

- **id:** This identifier allows precise tracking and reference of each data register.
- **value:** A component's status or performance is represented by the value in the data register.
- **date:** Tracks and analyzes data chronologically by capturing the timestamp when it was registered.
- **componentThing:** Connects data registers with specific plant components through the ComponentThing entity. This allows each data register to be attributed to a specific component, providing context to values recorded.

Station Module

The Station Module serves as a fundamental component of the web application, facilitating the management of various stations within the chemical plant, including the bioreactor, leaching and electrowinning. Users can efficiently interact with stations with this module, including retrieving station details, including associated components.

Each station in the chemical plant is represented by an entity called Station, which includes the following attributes:

- **id:** Each station is identified by a unique identifier, allowing precise identification and reference.
- **name:** Describes the location or function of a station within the chemical plant.
- **componentList:** There are many-to-many relationships between stations and specific components within the plant, ensuring that each station is linked to a relevant component. This attribute represents a list of associated components. With this association, users can understand the components associated with each station, making monitoring and management more efficient.

User Module

An extensive range of user-related operations is handled by the User Module, which plays a central role in user management within the web application. With this module, users are created, roles are assigned, passwords are changed, user notifications are sent, and emails are managed effectively. A user's privileges and responsibilities are defined by their role: "admin", "user" or "view".

As the core of the User Module, the User entity provides essential information about users, including the following attributes:

- **username(id):** User identifiers are typically used for authentication and user-specific actions.
- **password:** The user's login password, securely stored in the database.
- **email:** Communication and notifications are enabled by a unique email address associated with each user.
- **roles:** Each user's permissions and access levels are determined by the Role attribute, which establishes a many-to-many relationship between the User attribute and the Role entity.
- **listenNotification:** Notifications enable users to stay on top of important system updates and events. This feature can be toggled on and off by the user.

Messaging Module

This module encompasses a range of services that facilitate the sending of email notifications, especially when critical alerts are triggered. To ensure the existence of a single instance of the EmailService, the Messaging Module leverages the Singleton pattern.
It is designed according to the Singleton pattern to ensure that only one instance of the EmailService exists throughout the application's lifecycle. At the core of the Messaging Module lies the EmailService. In this way, resource efficiency is increased, and email notifications are handled consistently.
The module sends email notifications to users when alerts are triggered, keeping them informed about important events and issues within the bioleaching plant as soon as they occur.

2. Services

Application services perform specific functions within the application, ensuring the efficient execution of critical functions.

A crucial element of increasing modularity, maintainability, and overall system efficiency are the services that encapsulate the business logic that drives the various modules and ultimately enable users to interact with, manage, and monitor the bioleaching plant seamlessly.

It is evident that services play an important role in centralizing and streamlining application operations. As a result, each service is crafted to handle a particular set of tasks, such as handling alerts, managing plant components, processing data registers, overseeing stations, and managing user information. Services ensure code maintainability and reuse, facilitating agility for changing application requirements by compartmentalizing these functionalities.

As a result, services simplify and clarify code by abstracting the complexities of data manipulation and repository communication. By providing a unified interface for interaction between the application logic and the underlying data repositories, they act as intermediaries between them.

By enforcing business rules and validating data, the services ensure that our application is secure and compliant. They protect sensitive information and ensure that it is processed and accessed securely.

When conflicts occur or operations are attempted on nonexistent alerts, the service methods provide meaningful error responses through exception handling. When conflicts or not found scenarios occur, it throws a ResponseStatusException with a suitable HTTP status code and message.

Let's examine each service's unique features and capabilities:

Component Service

A wide range of component-related operations are handled by the Component Service, a critical component within the web application.

Component Service Methods:

- **getAllComponent():** This method retrieves a list of all components stored in the data repository, providing users with comprehensive access to component information.
- **getComponentByName(String name):** When called, this method retrieves a component by its name. It includes validation to ensure that the name provided is not empty.
- **insertComponent(ComponentThingDTO componentThingDTO):** This method inserts a new component into the data repository, including necessary validation checks. It also associates the component with a specific station.
- **updateComponent(ComponentThing componentThing):**

- **deleteComponent(Long id):** Components can be removed from the data repository based on their ID. They can also be removed from associated stations and disassociated from registers using this method.
- **changeComponentName(Long id, String name):** Allows users to change the name of a component identified by its ID.
- **changeComponentUnit(Long id, String unit):** This method enables users to modify the unit associated with a component.
- **changeComponentMinValue(Long id, Float minValue):** Facilitates the modification of a component's minimum value.
- **changeComponentMaxValue(Long id, Float maxValue**): Allows users to adjust the maximum value associated with a component.
- **pumpCommand(Long id, boolean value):** This method handles enabling or disabling components for action, particularly for pumping.
- **getAllComponentsWithLatestRegisterByStationId(Long id):** Provides real-time monitoring and reporting capabilities by retrieving a list of all components associated with a specified station.

Register Service

The Register Service retrieves data registers and inserts new registers.

Register Service Methods:

- **getAllRegister():** As part of this method, all registers in the repository are retrieved, as well as the raw register data is transformed into a more accessible format represented by RegisterDTO objects.
- **getRegisterByStationId(Long id):** This method retrieves registers associated with a specific station based on the station's ID. After obtaining the station from the repository, it retrieves all components linked to the station. In order to increase accessibility, it iterates through these components to retrieve registers. As with the previous method, the data is transformed into RegisterDTO objects.
- **insertRegister(Register register):** Ensures there is not already a register with the same ID in the data repository, preventing duplication by inserting a new register.

In order to transform raw register data into a more consumable format, the service methods use Data Transfer Objects (DTOs), specifically RegisterDTO, which encapsulates ID, value, date, component ID, component name, and component unit.

Station Service

Station Services support the efficient operation and administration of various stations within the bioleaching plant by managing station-related operations within the web application.

Station Service Methods:

- **getAllStation():** Using this method, you can retrieve a list of all data stations stored in the repository. You can also access the ID and name of each station.

- **updateStation(Station station):** This function updates station details, such as the added new components. It verifies the existence of the station based on its ID before updating it.

- **getStationById(Long id):** The ResponseStatusException throws a ResponseStatusException with an appropriate HTTP status code and message if the requested station does not exist.

User Service

User Service manages user-related tasks such as user creation, role assignment, password changes, email retrieval, role changes, email updates, and user deletion.

User Service Methods:

- **loadUserByUsername(String username):** It constructs a collection of SimpleGrantedAuthority objects from the user's roles and returns user details for authentication based on the provided username.
- **saveUser(User user):** Securely stores user information, including passwords, in the data repository.
- **saveRole(Role role):** Allows for the creation or saving of user roles into the data repository.
- **addRoleToUser(String username, String roleName):** The function assigns a specific role to a user by retrieving the user by their username and the role by its name.
- **getUser(String username):** Based on the username provided, retrieves user details.
- **getUsers():** Lists all users stored in the data repository.
- **getEmail(String username):** This method returns a user's email address based on their username. If the user cannot be found, the method throws a ResponseStatusException.
- **changePassword(String username, String password):** Encodes the new password and updates it in the data repository securely.
- **changeRole(String username, String role):** Changes a user's role based on whether the user and role exist.
- **changeEmail(String username, String email):** Provides a means of updating a user's email address in the data repository.
- **deleteUser(String username):** Removes a user account based on their username.
- **changeUserReceiveNotications(String username, boolean value):** Provides users with the ability to set their notification preferences (listening or not listening to notifications).

Messaging Service

Messaging Service sends notifications to users, interacting with Email Service to do so.

Service Method:

- **sendEmailForTheAlert(ComponentThing componentThing, String messageText, User user):** This method is used to send an email notification for a specific alert. It takes three main parameters:
    - **componentThing:** The component associated with the alert.
    - **messageText:** The text of the alert message.
    - **user:** The user who will receive the email notification.

As a result of the method, the following steps are performed:

1. To ensure that email notifications are only sent to users with valid email addresses, it checks if the user has an email address (user.getEmail() != null).
2. Using the JavaMailSender, it creates a MimeMessage object.
3. Configures the email message by setting the sender, recipient, subject, and content.
4. The MessageFormatterImpl is used to create a formatted message, combining the component information with the message text.
5. The email message is then sent using the emailSender.

3. Repositories

In the present web application's architectural framework, the Repository Chapter plays a pivotal role in database management. This chapter contains a collection of repositories, each meticulously tailored to handle specific domain entities. With these repositories, we can perform CRUD operations on our application's critical data with ease; Create, Read, Update, and Delete.

Streamlined CRUD Operations:

It is crucial to the design of an application repository that CRUD operations are executed seamlessly. In addition to creating new records, retrieving existing ones, updating their attributes, and removing records as necessary, these operations cover the entire spectrum of data interaction. A repository orchestrates these operations efficiently and precisely.

Stereotype Annotations:

Spring stereotype annotations, including @Repository, serve as markers for these repositories, signaling to Spring that they serve as repositories. Spring, in turn, detects and manages these repositories seamlessly within the application context.

Essentially, well-crafted repositories ensure data integrity, maintain consistency, and guarantee accessibility across the entire application. Our web application is resilient and dependable because each repository chapter caters to the unique needs of its corresponding domain entity.

Taking a closer look at each repository chapter, let's examine its specific usage:

Component Repository: Managing components, critical elements in the application, such as pH levels, colors, and pumps, is the focus of this repository chapter. In this way, these essential elements are accurately and efficiently managed. It provides a comprehensive suite of methods to create, read, update, and delete component data.

Register Repository: In addition to storing historical data, the Register Repository also manages data registers associated with components, stations, or specific time intervals. The system ensures that registers can be retrieved, inserted, updated, and deleted seamlessly, while maintaining an exhaustive history.

Station Repository: In the chemical plant application, stations are integral components. The Station Repository manages station-related database operations. It allows us to retrieve station information, update station names, and manage associated components efficiently.

User Repository: In addition to creating users, assigning roles, changing passwords, managing email addresses, and deleting users, the User Repository is the cornerstone of our user management system, handling a wide range of user-related tasks.

4. Controllers

In our application's architectural framework, the Controller Chapter takes the reins when it comes to handling HTTP requests from the front-end or external clients. Each controller in this chapter is designed to cater to a different aspect of the application's functionality.

To fulfill requests effectively, these controllers act as intermediaries, seamlessly receiving and processing HTTP requests.

Specialized Controllers:

- **Component Controller:** To provide a comprehensive set of operations for HTTP requests related to components, this controller interfaces with the Component Service.
- **Register Controller:** Data registers are managed by this controller, which manages HTTP requests for data retrieval, updates, and more. It communicates with the Register Service to ensure data integrity.
- **Station Controller:** This controller is at the forefront of all station-related operations. It receives HTTP requests and collaborates with the Station Service to process them.
- **User Controller:** Managing user actions is the core responsibility of this controller. It works in conjunction with the User Service to fulfill user-related requests.
- **Messaging Controller:** In addition to providing WebSocket functionality, it enables real-time messaging and notifications within the application. This feature significantly enhances the user experience by providing instant notifications and critical alert notifications in real time.

- **PlantWebSocketController:** Provides websocket functionality for plant communication. It provides the logic to handle user commands and plant component status, together with register creation.

Development and Implementation

This chapter delves into the development and implementation of the web application. It explores the methodologies and challenges encountered during the development process. Additionally, it details the architecture, database design, user interface, and core functionalities of the application, providing an in-depth view of its construction.

Development Methodology

Projects were broken down into manageable sprints using agile practices such as Scrum.

Managing source code efficiently, tracking changes, and collaborating smoothly was made possible by Git, a distributed version control system. Git enabled Agile development. In addition to maintaining code integrity, Git branching and merging capabilities allowed features and bug fixes to be developed simultaneously.

Git repositories, hosted on GitHub, provided a centralized location for code storage and version history.

Database Implementation

A normalized schema was carefully crafted consisting of tables for users, components, stations, registers, and roles. The schema reflects the logical structure of our application's data as part of the initial and critical phase of database implementation. The schema consisted of several tables that served a specific purpose in the application. The major tables included:

- **Users:** Managing user-related information such as usernames, passwords, email addresses, and roles.
- **Components:** Information about plant components, such as names, descriptions, units, and operational thresholds.
- **Stations:** Data collection related to the chemical plant's various stations, including station names and components.
- **Registers:** Historical data records, including values, timestamps, and component associations.

A key measure within the database ensures data integrity and consistency:

- **Foreign Key Constraints:** The application uses foreign key constraints to establish relationships between tables. For instance, the 'Registers' table had foreign key references to both the 'Components' and the 'Stations' tables. As a result, data anomalies were prevented because registers were always associated with valid components and stations.

SQLITE Database

The app relies on SQLite database to store its data. The application properties file (application.properties) contains the configuration parameters for connecting to the database. As part of these configurations, Spring Boot applications are able to access and control SQLite databases and send emails using Gmail, providing connectivity to SQLite databases. In production-grade applications, be sure to secure sensitive data adequately.

Here are the configuration parameters to match the application backend with the server's database:

spring.jpa.database-platform=org.hibernate.community.dialect.SQLiteDialect

Specifies the dialect for Hibernate, indicating that SQLite will be used, and helps Hibernate generate SQL optimized for the SQLite database. This is necessary because spring boot does not support sqlite.

spring.jpa.hibernate.ddl-auto=update

Configures Hibernate's behavior with regards to database schema updates. Here, `update` means that Hibernate will update the schema whenever it finds a discrepancy.

spring.datasource.url = jdbc:sqlite:****

Specifies the JDBC URL for the SQLite database.

spring.datasource.driver-class-name = org.sqlite.JDBC

Specifies the JDBC driver to be used for connecting to SQLite.

spring.datasource.initialization-mode=never

Spring.datasource.initialization-mode can be used in the following ways:

- ***always***: Useful in development or testing environments, where you want to start with a known database state each time the application starts. This could include creating the schema, initializing or re-initializing data, etc.
- ***never***: More common in production environments to avoid any risk of changing the schema or data unintentionally on application restart.
- ***embedded***: Only initialize the database if an embedded database is used (like H2). This is often the default setting in development setups.

User Interface (UI) Implementation

User experience (UX) principles were incorporated to optimize navigation and minimize cognitive load in the user interface. UI components were built using React and Bootstrap

Authentication and Landing Page

As explained earlier, authentication is token-based, and users are directed to a landing page after successfully logging in.

Navigation

NavBar component is used to visually represent navigation within the application. An "Admin" dropdown menu provides additional options for administrators in the NavBar. Users with “user” and “view” roles do not see this dropdown, which ensures that they aren't presented with administrative functions. This menu provides access to admin-specific views for managing users and components. Admins and users can access the command view from the home page. Notifications and registers are also accessible from the home page.

In the NavSide, a list of station names is retrieved and displayed as buttons in a functional component. When a button is clicked, the component updates which station is considered “selected” and may trigger a dynamically named callback function from its props. CSS class styling is also used to determine the component's visual appearance.

All User Views

There is a schematic representation of the entire plant layout on the landing page, giving users a high-level understanding of the plant's structure.

A dropdown menu allows users to select specific stations. Each station-specific view displays:

- The station's schema.
- Data from its components.

Users can gain an overview of key metrics across the plant, and they can navigate to specific components using the station dropdown menu. In addition, they can access a detailed view of a component by clicking on its value. This view includes an interactive graph displaying historical data for that component.

Notifications Panel

On this panel, users can manage personal settings, such as:

- Changing their password.
- Modifying email notification preferences.
- Enabling or disabling email notifications for alerts.

Registers Panel

This panel is crucial for monitoring the plant's status and performance. In addition to displaying the registers with the latest component status obtained from the plant, it provides a variety of filtering, sorting, exporting, and visualizing functionality.

Alerts Panel

This panel orchestrates the management and display of alert messages within a user interface, as well as providing functionality for filtering, sorting, and exporting them.

It catches the alert received through websockets and saves it in an indexedDB - a low level API for client-side storage. There is a cleanup operation, where alerts that are older than one month are deleted during each cleanup operation. In this way, the application is able to manage and discard stale data on a regular basis while minimizing the burden on resources by scheduling the cleanup operation.

Administrator Views

User Dashboard

Users can be managed via dedicated views for administrators. From here, administrators can:

- View existing user accounts.
- Create new user accounts.
- Delete user accounts.
- Manage user roles.
- Configure email settings for users.

Component Dashboard

Station Management and Component Management enable administrators to oversee the assigned components and stations of the plant. Here, they can:

- Assign or remove components from stations.
- Delete components
- Adjust component parameters, including min/max values and canAct properties.

Command Dashboard

Administrators can control pumps and solenoid valves using the "Commands dashboard”.

Functionality Implementation

User Registration and Login

Registration and login functionality are implemented using Spring Security for authentication and JWT (JSON Web Tokens) for session management. Passwords are hashed before storage.

In order to ensure that authorized individuals can access and interact with the application securely, user registration and login are fundamental components. Towards this goal, we used Spring Security, a Java application security framework that provides comprehensive security features.

The data users provide when they register on our platform, including their usernames, e-mail addresses, and passwords, is securely stored in the SQLite database. Passwords are hashed before storage for enhanced security, which makes accessing user credentials extremely difficult.

JSON Web Tokens (JWT)

Our application uses JSON Web Tokens (JWTs) for session management and authorization. In addition to carrying information about the user, JWTs are digitally signed to ensure their integrity. JWTs are compact and self-contained tokens that provide information about the user.

Security configurations for the web-based application are meticulously defined across three key Java classes.

The **CustomAuthenticationFilter** class, which extends `UsernamePasswordAuthenticationFilter`, interfaces with the `AuthenticationManager` to authenticate a user based on provided username and password. Upon successful authentication, it employs the `com.auth0.jwt` library to generate a HMAC256 encoded JWT that includes user details, roles, and a set expiration time. This token is then returned in a JSON format as part of the HTTP response.

In tandem, the **`CustomAuthorizationFilter`** class, derived from `OncePerRequestFilter`, oversees the authorization of users attempting to access API endpoints. This filter checks for a "Bearer" token in the request headers, decodes the JWT if present, retrieves associated user roles, and sets them within the Spring Security context.

Finally, the **`SecurityConfig`** class integrates these filters into the application's security flow. It employs a `DelegatingPasswordEncoder` for password encryption and establishes a stateless session management policy. HTTP request routes are mapped to specific user roles ensuring granular access control. Through the fusion of these classes, a robust JWT-based authentication and authorization mechanism is established, ensuring secure user access.

Secure Password Storage

Secure password storage practices were implemented to protect users data:

Password Hashing

The user passwords are hashed before being stored in the database. Hashing transforms the password into an irreversible string of characters, making it extremely difficult for attackers to retrieve the original password.

Roles and Permissions

User roles and permissions are included in JWTs. This allows us to control access to various parts of the application, ensuring that only the functionalities that are authorized to use are accessible. There are three roles that users can be assigned to users: Admin, User, View. The rights access for each role are as follows:

User Roles	User dashboard	Component dashboard	Command dashboard	Notifications panel	Registers panel	Alerts panel
Admin	🟩	🟩	🟩	🟩	🟩	🟩
User			🟩	🟩	🟩	🟩
View				🟩	🟩	🟩

Table 1: User roles permissions

In *chapter 6, Results and Discussions*, the different dashboards and panels are explained in detail.

Route-based Navigation

It is imperative that modern web applications guide users seamlessly according to their roles and authentication status, ensuring both security and optimal user experiences. Essentially, route-based navigation employs specialized components, each designed to manage and dictate access based on user roles or authentication states.

By implementing user rights navigation, we ensure that users can only access the functionalities they are authorized to use, enhancing security and user experience.

PublicRoute

The PublicRoute component is designed to manage unauthenticated users' access to certain routes. Users who are already authenticated are automatically redirected to the home page. As a result, users who are already logged in do not need to re-login repeatedly. However, users who are not authenticated are allowed to access the intended route.

PrivateRoute

PrivateRoute is responsible for handling routes that require authentication. When a user tries to access such a route, the system checks if they are authenticated. In this way, unauthorized access to sensitive parts of the application is prevented by redirecting them to the login page.

AdminRoute

The AdminRoute component caters specifically to administrators. It checks if the user is both authenticated and has the role of 'ADMIN'. If the user doesn't meet these criteria, they are redirected to the login page. This route is dedicated to administrators and grants them access to exclusive features and views tailored to their role.

Using this role-based navigation methodology, users are presented with an optimized set of options relevant to their roles, which enhances security as well as optimizes the user experience. By doing so, it ensures that the application is interacting with each user in accordance with their permissions and responsibilities.

UserRoute

Users and administrators are served by the UserRoute component. It checks if the user is both authenticated and has the role of 'ADMIN' or 'USER'. If these criteria are not met, the user will be redirected to the login page. Using this route, they gain access to exclusive features and views that are tailored to their roles.

CRUD Operations

In web application development, CRUD stands for Create, Read, Update, and Delete, which are fundamental operations for managing data in a web application. CRUD operations correspond to the following actions:

- **Create:** Adding new data records or entities.
- **Read:** Retrieving and viewing data.
- **Update:** Modifying existing data.
- **Delete:** Removing data.

CRUD operations serve as the core functionality of our web application. These operations manage components, stations, registers, and alerts. CRUD is critical to our project for the following reasons:

1. **Data Management:** Users can manage the application's data through CRUD operations, such as creating new components, reading station details, updating component names, or deleting alerts.
2. **User Interaction:** A CRUD operation enables users to interact with the system, allowing them to input, retrieve, and modify data, which is essential for the application's functionality and usability.
3. **Data Integrity:** In order to maintain data integrity, CRUD operations provide controlled ways to modify data, which minimizes the risk of accidental or unauthorized data modifications.
4. **Consistency:** As part of CRUD operations, data management is standardized to ensure consistency.

Let's look at some examples from our API implementation in the api.js file:

fetchComponentsWithLatestRegisterByStationId(id): This function corresponds to the "Read" operation. It retrieves components with their latest registers by station ID.

fetchAllRegisters( ): This function is an example of the "Read" operation, fetching all registers.

changeRole(username, newRole): This function represents the "Update" operation, allowing the modification of a user's role.

insertComponent( ): Here, we have an example of the "Create" operation, which inserts a new component into the system.

deleteComponent(componentId): This function demonstrates the "Delete" operation, enabling the removal of a component.

With these CRUD operations, our web application enables users to manage data efficiently, ensure data integrity, and deliver a seamless user experience.

In our project, CRUD operations enable users to interact with the application's features and ensure data reliability and consistency.

Real-time Messaging

With WebSocket technology, a communication protocol enabling real-time, bidirectional data exchange, this module empowers users to send and receive messages, which serve as a means of delivering alert notifications and commands as well as to receive updated component statuses from the plant. It uses

Key Components

1. **Email Service for Alerts:** In the event of an alert triggering in our system, the Email Service is responsible for sending email notifications to users. As a result, users receive timely notifications via email containing essential information about the alert, such as the component's name and the associated message.
2. **Message Formatter:** The Message Formatter has been implemented to ensure consistency and readability in alert messages. It takes the information about the alerted component and the associated message and creates a standardized message based on the information. The formatter ensures that alerts are presented in a clear and uniform manner by including information such as the component's name, the timestamp, and the message itself.
3. **WebSocket Controller**: The WebSocket Controller manages the real-time messaging functionality. It listens for messages sent by the API / plant and, when received, broadcasts them to all connected WebSocket clients. In addition, it also sends commands to the server side, where the plant is listening. This approach allows the web application to send alerts to the user by browser notifications, send commands from the user interface to the server side where the plant is listening and also receive the objects coming from the plant in order to create the registers with the latest component status.
4. **WebSocket Configuration:** A WebSocket endpoint and a message broker have been configured in the application to enable WebSocket support. A WebSocket Configuration class identifies the endpoint of WebSocket connections and specifies which destinations should be enabled for message broadcasting. Using this configuration, WebSocket communication is seamless and efficient.

The WebSocketConfiguration class provides a structured approach to implementing WebSocket communication within Spring applications, focusing primarily on **STOMP** messaging. With STOMP (Simple Text Oriented Messaging Protocol), WebSockets are enhanced by a simple, lightweight messaging protocol, allowing message exchanges between client and server. This protocol is neatly integrated with a robust framework to manage subscriptions and message broadcasts.The configuration class designates "/ws/" as a WebSocket endpoint through which clients can establish WebSocket connections to servers.

The Message Broker ensures that messages destined for application-level handling are routed appropriately by setting **"/ws"** as the application destination prefix via .setApplicationDestinationPrefixes("/ws").

With .enableSimpleBroker**("/message", "/info")**, it activates a memory-based message broker for managing subscriber subscriptions and broadcasting messages across specified destination prefixes to subscribers.

In order to streamline the organization of messages within defined channels and facilitate topic-based subscriptions from clients, the selected prefixes ("/message", "/info") implicitly define the overarching topics or destinations that the broker will manage.

The prefix "/message" is used for alert notifications. The user is listening to the API for an alert message.

The prefix “/info” is used for plant communication. The user sends commands to that direction where the plant is listening. Moreover, the plant sends component status and the API listens and generates a register based on the component id and value it receives. With the temporized last register values fetchers present in the command and register components, the application automatically updates.

The used format is JSON:

• **Plant to server:**

```
[
        {
                "id" : [COMPONENT ID],
                "value" : [COMPONENT VALUE]
        }
        , (...)
]
```

• **Server to plant:**

```
{
        "id" :[COMPONENT ID],
        "command" : [COMMAND VALUE]
}
```

Email Notification System

The EmailService class, which adheres to the singleton design pattern, ensures that only one instance of the email service is instantiated and utilized throughout the application lifecycle, maintaining centralized control and utilization. In this context, adopting a singleton pattern is motivated by the desire to centralize email dispatching, ensuring that all email-related operations are funneled through a single, unified object, possibly enhancing manageability and control. SendEmailForTheAlert encapsulates the functionality of writing and dispatching an email, if the user's email address is not null.

Using JavaMailSender, it crafts an email containing various parameters, such as the sender's email, the recipient's email, the subject, and the body.

As a message composer, the MessageFormatorImpl class uses its pivotal createStandardMessage method to create standardized email messages by combining static text with dynamic data - component names and current timestamps, along with a custom message.

Moreover, email dispatching isn't just a function of the email service and formatter but is also substantially influenced by the parameterization done via application properties, such as those specified for email (e.g., spring.mail.host, spring.mail.username, etc.). As a result of these parameters, email dispatching is not only secured by credential usage but also adheres to SMTP configurations that enhance reliability and security of email communication. The email server (smtp.gmail.com), authentication credentials, and various SMTP configurations are used to ensure email dispatching is secure and reliable.

Here are some email settings of the backend application.properties:

```
spring.mail.host=smtp.gmail.com

spring.mail.port=587

spring.mail.username=biometallumapp@gmail.com

spring.mail.password=****

spring.mail.properties.mail.smtp.auth=true

spring.mail.properties.mail.smtp.starttls.enable=true
```

Here's what each property means:

- **spring.mail.host** and **spring.mail.port:** Specify the hostname and port the SMTP server. For instance, Gmail's SMTP server.
- **spring.mail.username** and **spring.mail.password:** Email address and password of the account that will be used to send notifications.
- **spring.mail.properties.mail.smtp.starttls.enable=true:** It will use SMTP authentication when sending emails through the configured SMTP server.
- **spring.mail.properties.mail.smtp.starttls.enable:** Enable TLS for secure email transmission.

Scalability and Future Enhancements

The microservices architecture of the web application allows this application to be horizontally scaled to handle increasing user loads. Moreover, the application has a scalable management system, adeptly enabling dynamic interactions and management of various entities and operational settings. Different components associated with different stations can be dynamically interacted with and modified using this system. In addition to altering settings (such as min and max values), this platform allows for tailoring operational thresholds and triggering alerts in accordance with differing requirements without modifying the code itself. Thus, it lays down a comprehensive, user-adaptable framework that can efficiently cater to an array of component management needs.

A notable characteristic of this system is its ability to seamlessly adapt and manage various component types, such as pumps and solenoid valves, ensuring that it can seamlessly integrate and manage new components. By auto-detection and displaying new components alongside existing ones, the system creates an environment for dynamic, real-time management.

However, there is a limitation regarding the visual representation of new components within existing imagery, which could pose challenges in terms of user experience and operational monitoring. Despite this, the system enables interactive and real-time modification of component settings, demonstrating an attentive approach to user engagement and operability.

A modular development approach might also be beneficial, considering the scalability and maintainability of the system in the future. A system can be more maintainable and scalable if it is broken down into smaller, more manageable components and functionalities. While the system stands out for its flexibility and interactive user interface, future developments could enhance its utility and user experience by improving state management, code modularity, and visual representation.

Deployment

Hosting Considerations

Connecting to the Backend

In the front-end application, it is specified the URL for connecting to the backend API. It must be correctly configured to point to the server where the application will be hosted:

const API_URL = 'http://*biometallum.epsem.upc.edu*/api';

Nginx.config

As explained in the plant existing software chapter, Nginx is used as a reverse proxy. Here is the basic scenario:

- React Front end is served as static files by NGINX.
- Spring boot backend is the API server to which NGINX proxies API requests.

Location proxy should be updated to the localhost where the application will be running:

- With this setup, when users navigate to the site, they are directly served the static files generated by the React app.

  ```
  location /  {
              proxy_pass http://localhost:...
  }
  ```

- With this setup, any request that starts with `/api/` is forwarded to the Spring Boot application.

  ```
  location /api/  {
              proxy_pass http://localhost:...
  }
  ```

By segregating API routes from frontend routes, it's possible to neatly separate the handling of static files and API calls.

Different Needs: Frontend (React) and backend (Spring Boot) have different requirements:

- **Frontend**: Only needs to serve static files (HTML, CSS, JS) and does not need to access the backend server unless it calls an API.
- **Backend**: Manages data processing, API requests, etc., and is not concerned with serving static frontend files.

API Calls: The API calls in the React app hits the /api/ endpoint to proxy to the Spring Boot application.

Tomcat Server

To host the Spring Boot web application, it is needed to install a Tomcat server on the hosting server.

6. Results and discussion

The images of this chapter illustrate the user interface as it is seen by the user across the app's functionalities. In previous sections, we have discussed the fact that the web app boasts a responsive design, meaning its appearance and layout will change depending on the device on which it is viewed, whether it is a mobile device, tablet, or computer screen. The set of images presented correspond to different viewports, providing a better understanding of the app's varied display options. The captions for each image will indicate the size of the display pixels.

Sign In

The sign in view allows users with the right credentials to log into the application.

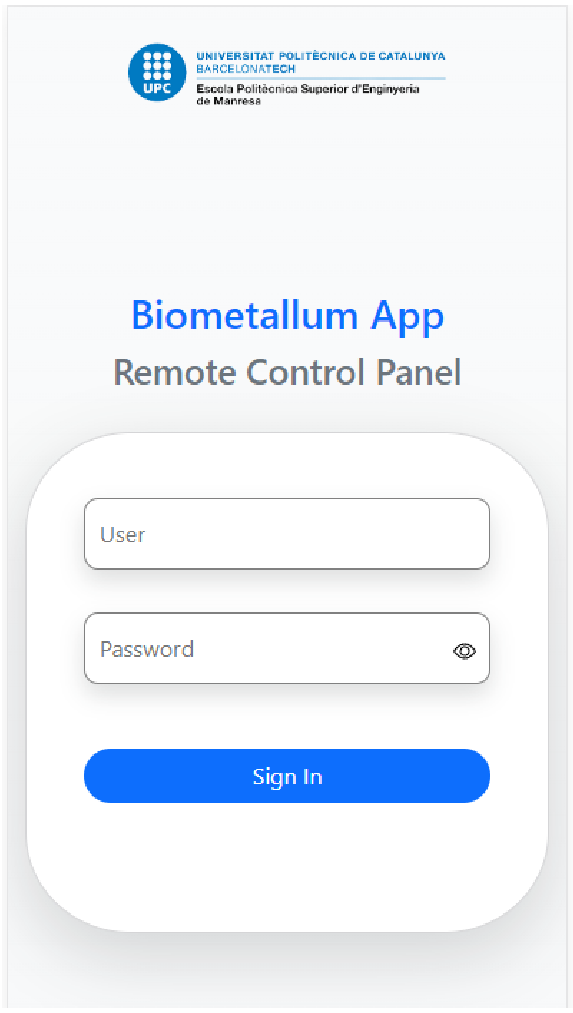

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Escola Politècnica Superior d'Enginyeria de Manresa
Biometallum App
Remote Control Panel
User
Password
Sign In

Figure 4*: Sign In window (mobile viewport)*

Data is traditionally passed from parent component to child component by means of properties, or "props.". By using the Context system, data can be distributed throughout the component tree without having to pass props manually at each level of the nested component tree.

This web application is using AuthContext, which serves as a centralized repository to convey authentication status, user roles, and associated methods to any part of the application that requires them.

In response to user commands, the login asynchronously communicates with the backend. Upon a successful response, this function not only updates the local storage with crucial data such as the access_token and roles, but it also sends the user's username and password via a GET request, eagerly awaiting confirmation. Additionally, it updates the authState to reflect the user's current authentication status, ensuring the entire application is in sync.

Once the user has successfully logged in, the logout function is activated by clicking a button located in the navbar. This function clears user roles and data from the application's memory and authentication state. By doing so, it ensures that once a user logs out, their data will not be lingering around.

Main Page

The "Plant overview" component of the home page is displayed once the user successfully logs in.

A navigation component is available on the home page to facilitate user app navigation. “NavBar” and “NavSide” components are both present on the home page. The navbar component allows the user to return to the previous window, display a dropdown menu, or log off.

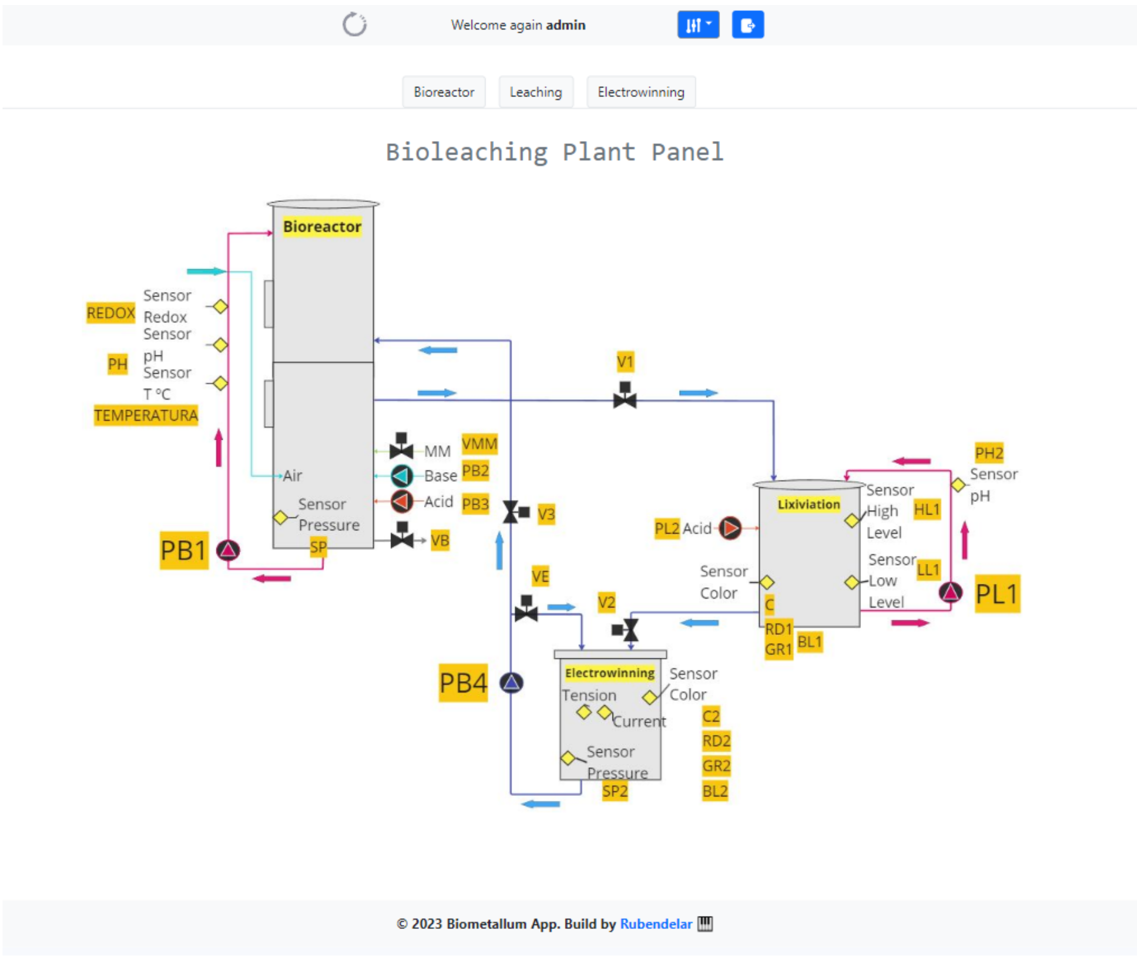
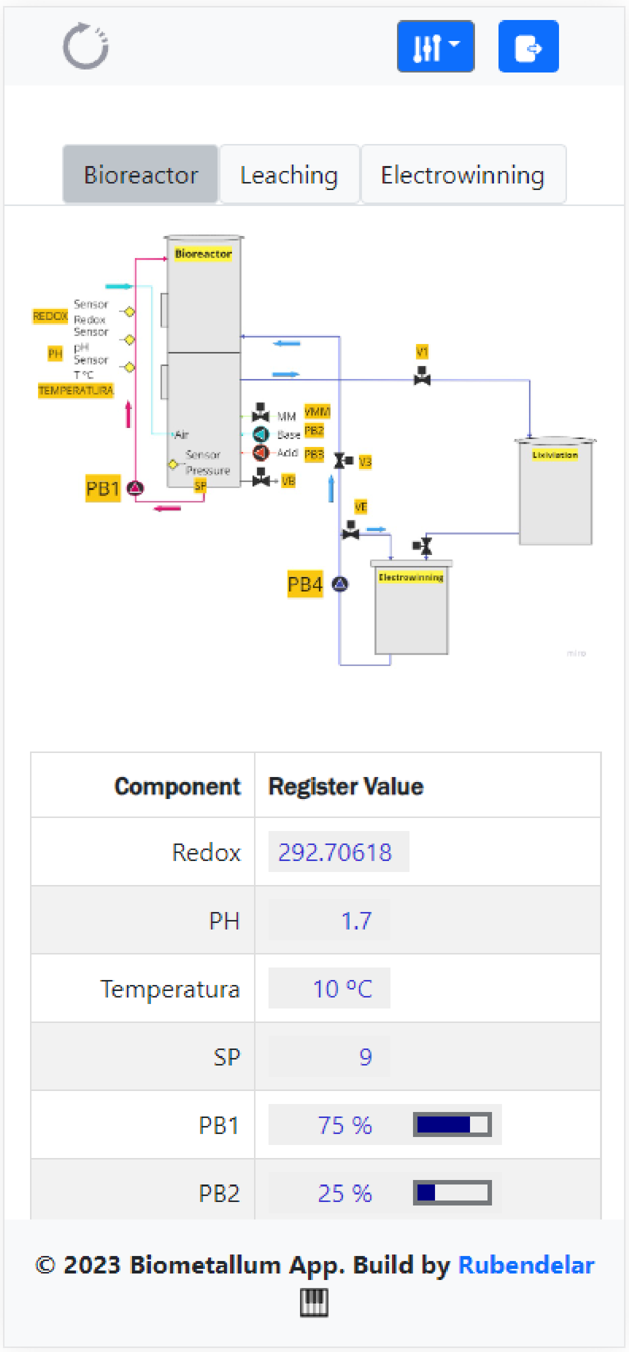


Figure 5: Plant overview, home window (desktop viewport)

On the other hand, `NavSide` component is only present in the home view and it shows the name of the active station and has a dropdown menu that allows a station selection. Depending on the active station, different `Station` components are rendered, each with a unique `stationId` and corresponding image.

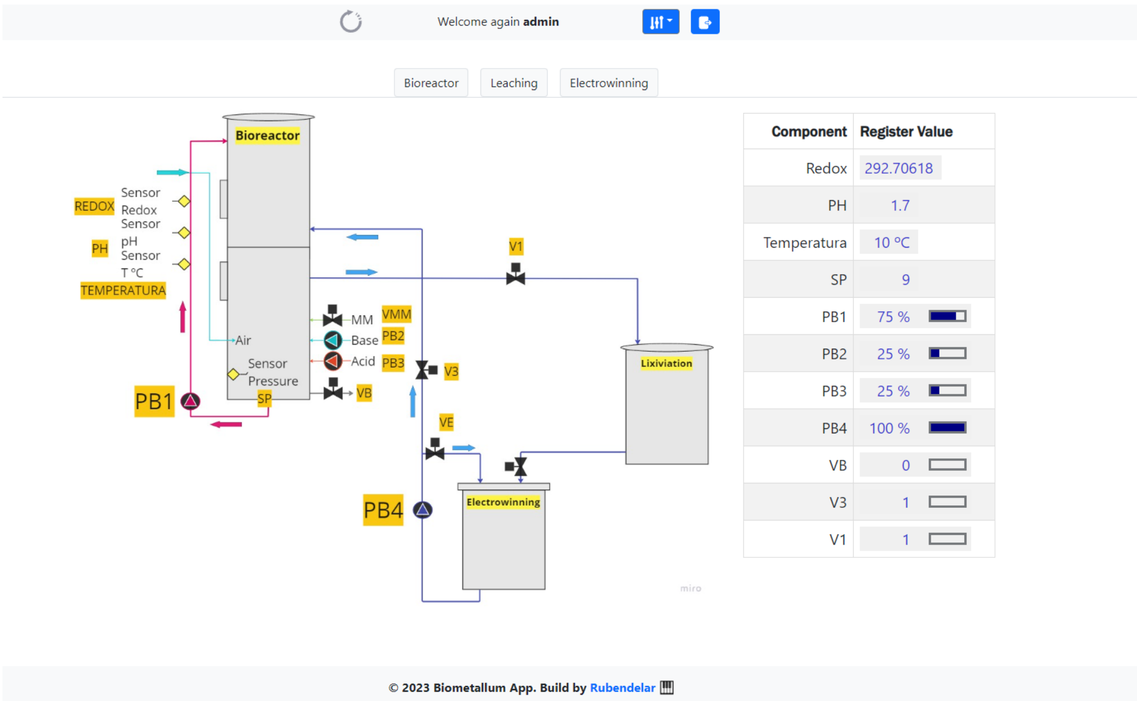
Clicking on any of these stations buttons in the dropdown will involve the station component and reveal a detailed image of the chosen station alongside a table containing the latest records of components assigned to that station. Component data is fetched using the `fetchComponentsWithLatestRegisterByStationId` API call. This populates the `components` state with the components' latest register values.



Component	Register Value
Redox	292.70618
PH	1.7
Temperatura	10 ºC
SP	9
PB1	75 %
PB2	25 %

© 2023 Biometallum App. Build by Rubendelar

Figure 6: Station selection and Bioreactor view, home window (mobile viewport)



Welcome again **admin**

Component	Register Value
Redox	292.70618
PH	1.7
Temperatura	10 ºC
SP	9
PB1	75 %
PB2	25 %
PB3	25 %
PB4	100 %
VB	0
V3	1
V1	1

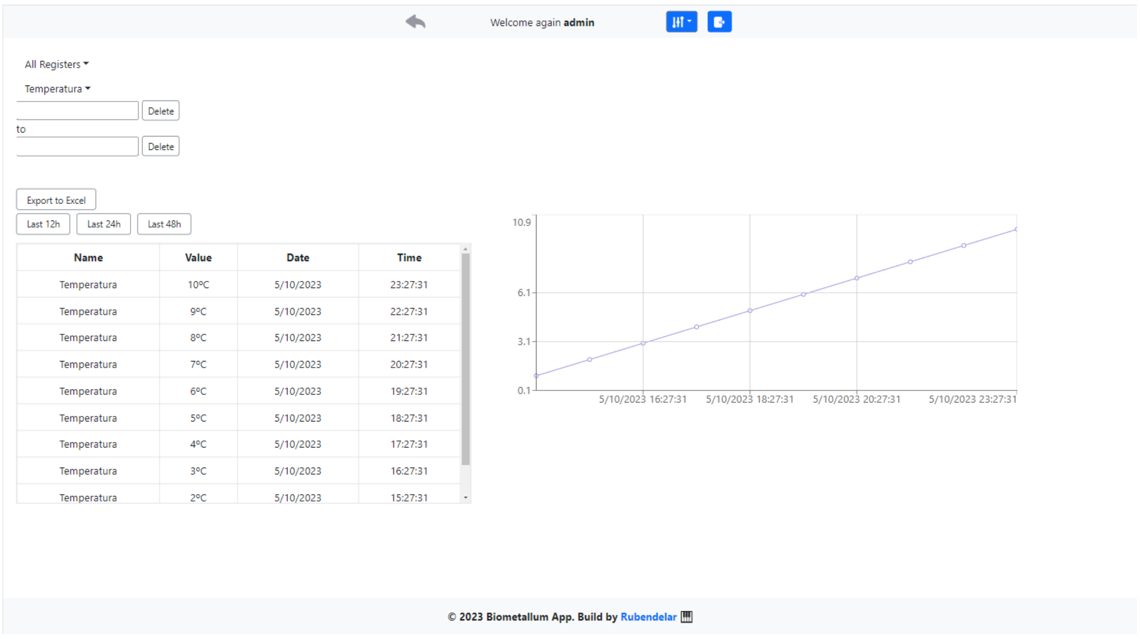
© 2023 Biometallum App. Build by Rubendelar

Figure 7: Bioreactor view, home window (desktop viewport)

Components for the station are listed here. If a component can act (`component.canAct`), a visual pump icon is displayed, with a gradient effect based on the component's latest register value.

For each component, there's a value display. Clicking on any record will redirect the user to the registers view, where a table displaying the total records of that component will appear, along with a graph. The `navigate` function from `react-router-dom` is used for this purpose.

Registers Panel



Name	Value	Date	Time
Temperatura	10ºC	5/10/2023	23:27:31
Temperatura	9ºC	5/10/2023	22:27:31
Temperatura	8ºC	5/10/2023	21:27:31
Temperatura	7ºC	5/10/2023	20:27:31
Temperatura	6ºC	5/10/2023	19:27:31
Temperatura	5ºC	5/10/2023	18:27:31
Temperatura	4ºC	5/10/2023	17:27:31
Temperatura	3ºC	5/10/2023	16:27:31
Temperatura	2ºC	5/10/2023	15:27:31

© 2023 Biometallum App. Build by Rubendelar

Figure 8: Temperature registers, registers window (desktop viewport)

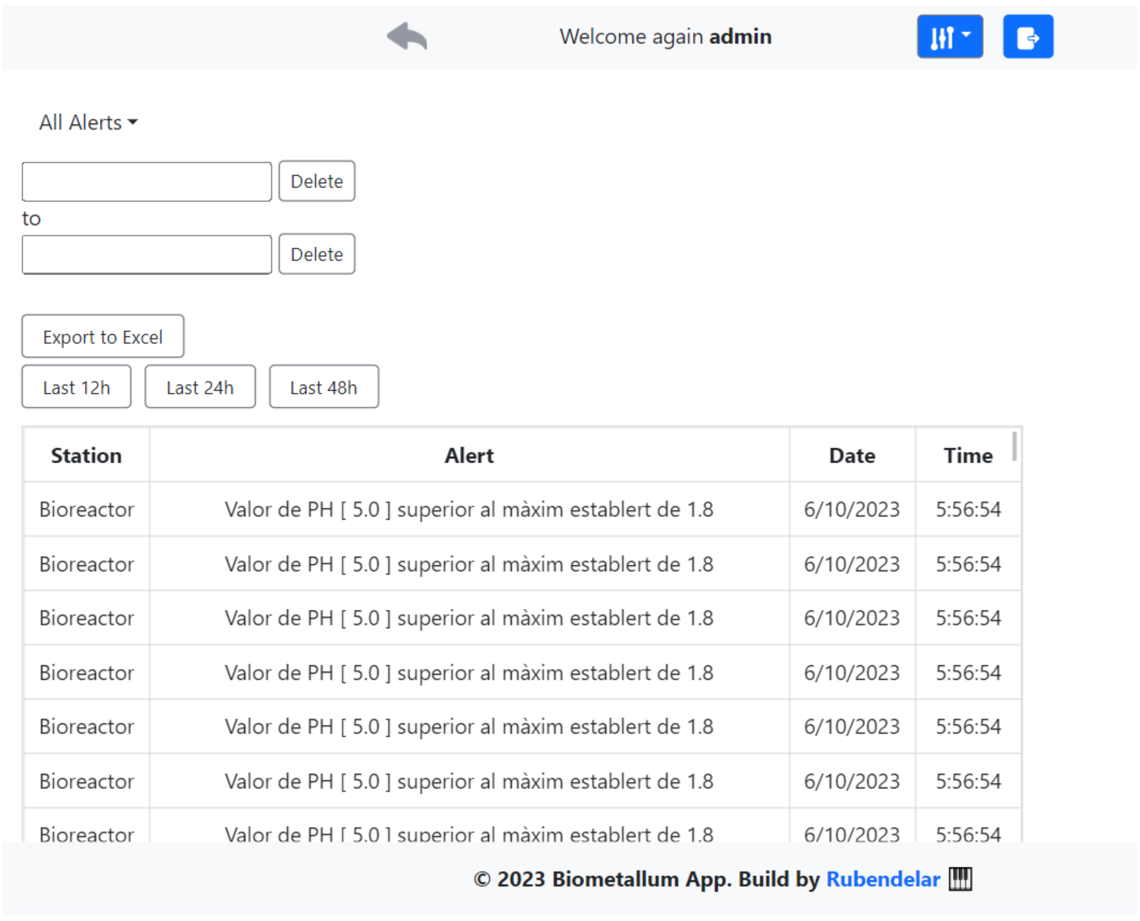
By clicking on the register value in the station dropdown menu, the registers panel is accessed. This access displays a graphic of the selected component. Alternatively, the register button is accessible from the home dropdown menu.

The graphic is displayed due to the RegistersGraphics component, which utilizes the **recharts library** to visually plot register data. Data is either fetched for all stations or for a specific station depending on the station chosen by the user. This dynamism ensures efficient data retrieval, minimizing unnecessary data transfer. A user can also filter the data according to a component's name or date range. With **react-datepicker,** users can fine-tune the date range for the data they wish to view. This ensures that only the data relevant to their query is displayed.

A user can view and filter records by station or component, view graphs, and export data to Excel using the Registers panel. The function generates an Excel-friendly format from filteredRegisters and initiates the download by using the xlsx library. In addition to visualizing, users can export the presented data.

Registers Panel integrates data fetching, filtering, visualization, and exportation, while maintaining a responsive and user-friendly interface.

Alerts Panel



Station	Alert	Date	Time
Bioreactor	Valor de PH [5.0] superior al màxim establert de 1.8	6/10/2023	5:56:54
Bioreactor	Valor de PH [5.0] superior al màxim establert de 1.8	6/10/2023	5:56:54
Bioreactor	Valor de PH [5.0] superior al màxim establert de 1.8	6/10/2023	5:56:54
Bioreactor	Valor de PH [5.0] superior al màxim establert de 1.8	6/10/2023	5:56:54
Bioreactor	Valor de PH [5.0] superior al màxim establert de 1.8	6/10/2023	5:56:54
Bioreactor	Valor de PH [5.0] superior al màxim establert de 1.8	6/10/2023	5:56:54
Bioreactor	Valor de PH [5.0] superior al màxim establert de 1.8	6/10/2023	5:56:54

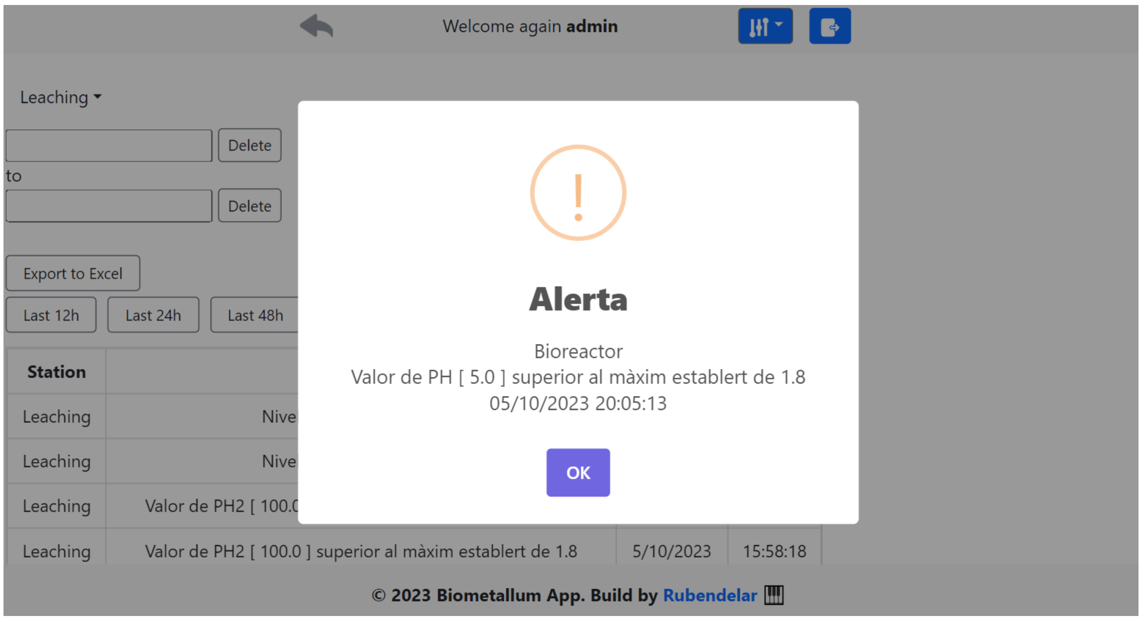
© 2023 Biometallum App. Build by Rubendelar

Alerts component allows to display and handle alert messages interactively. It offers a combination of asynchronous data retrieval, structured data processing, user-driven filtering, and exporting the alert data directly into an Excel file, leveraging the xlsx library.

A variety of controls are available in the user interface, such as a Dropdown for station-based filtering, DatePickers for setting date and time bounds for visible alerts, and buttons providing quick access to predefined time-ranges, such as the last 12, 24, or 48 hours. In addition, the alerts are sorted by date and time, once they have been parsed and filtered, and presented in a table that neatly displays the aforementioned attributes (station, alert content, date, and time).

It engages multiple useEffect hooks to retrieve alerts from indexedDB storage, add any new unique alerts to the context, and make them globally accessible across the component tree after initialization. Additionally, it performs a regular cleanup of alerts, ensuring a regulated and non-redundant data flow through intervals of approximately one hour, as well as obtaining station data which can be utilized to filter alerts.

Alerts Notifications



Alerta

Bioreactor
Valor de PH [5.0] superior al màxim establert de 1.8
05/10/2023 20:05:13

Station			
Leaching	Nive		
Leaching	Nive		
Leaching	Valor de PH2 [100.0		
Leaching	Valor de PH2 [100.0] superior al màxim establert de 1.8	5/10/2023	15:58:18

© 2023 Biometallum App. Build by Rubendelar

Figure 9: Alert notification, (desktop viewport)

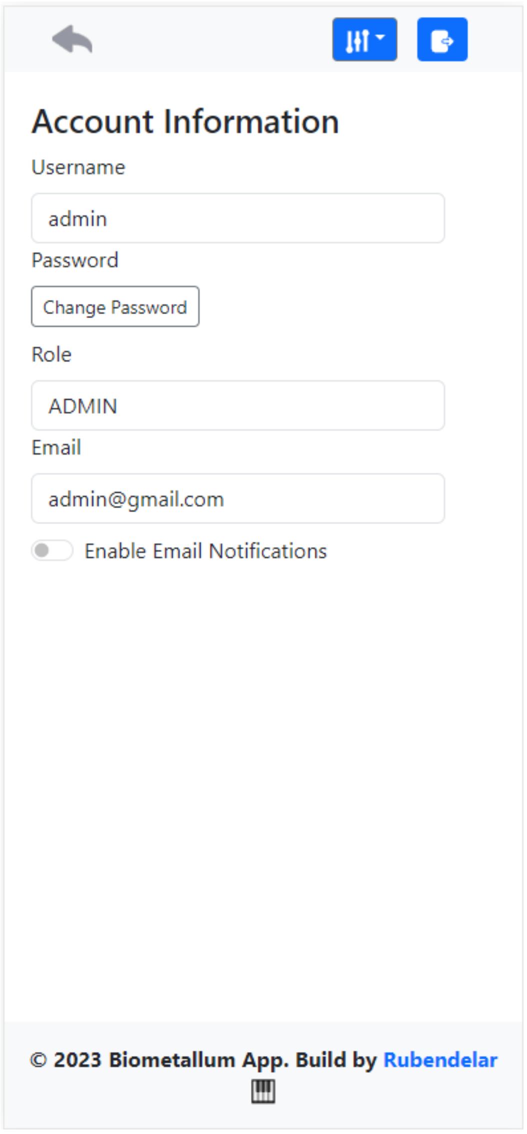
In the case of previously established components reaching the previously set limit value, alerts can be sent to the user using websocket communication between the user interface and the backend API.

ScheduleRegisterAlert component incorporates scheduled tasks, especially with the checkAlertWithTheLastRegister() method, to check for alert conditions across various the generated registers every 1 minute. The component navigates through various alert check methods, each adapted to specific alert scenarios, such as pH levels, high/low activity levels, and prolonged high activity scenarios in the Bioreactor and Leaching stations. Each alert type has its own method, which results in a structured and organized approach to handling different alerts. PHAlertCheck(User user), HL1AlertCheck(User user), and other methods scrutinize the respective parameters to determine if they violate predefined safety or operational thresholds.

In addition to detecting alert conditions, the component ensures that these alerts are communicated to the user in a timely and effective manner. WebSockets are used by messageController to dispatch alert messages to the user interface, enabling real-time notifications without requiring user action or refresh. In addition, the component integrates email notifications via emailServiceInterface, enhancing the notification channels and keeping users informed even when they're not actively using the interface. The alert throttling mechanism uses a map (lastAlertTime) to keep track of when the last alert for a particular component was issued. The interval time has been setted to 12 hours to avoid notification fatigue and ensure that users aren't inundated with notifications for the same issue repeatedly.

Notifications panel

Additionally, the home panel's dropdown menu allows users to access the notifications panel, which is designed to provide users with account information, notification preferences, and password management features.


Account Information

Username

admin

Password

Change Password

Role

ADMIN

Email

admin@gmail.com

Enable Email Notifications

© 2023 Biometallum App. Build by Rubendelar

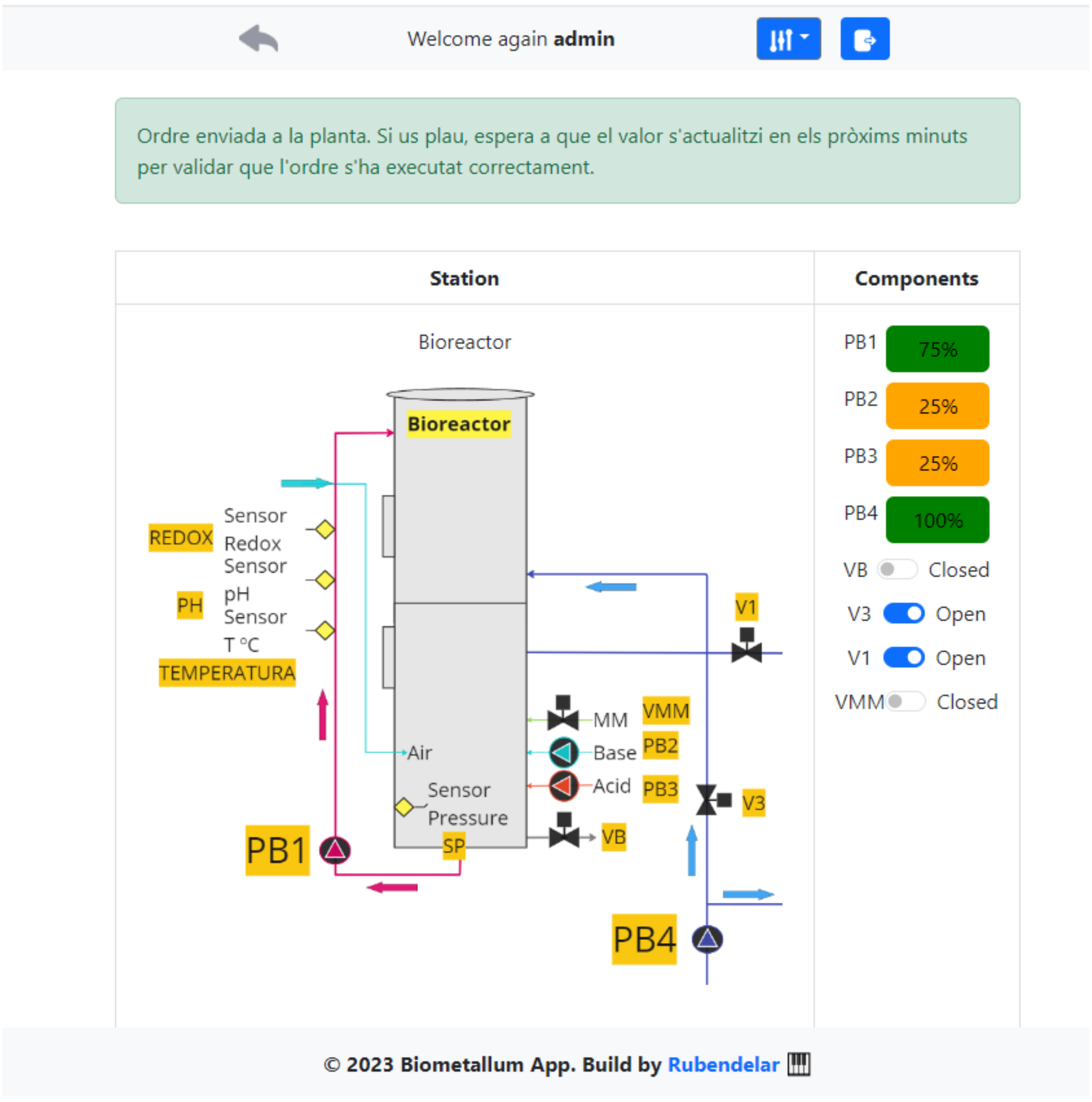
Figure 10: Notifications window (mobile viewport)

The useEffect hook allows the component to retrieve the user's data upon mounting. It retrieves the user's email and notification preference and updates the local state accordingly. Through useAuthContext, the component extracts user information from a shared context, including roles and usernames. To manage backend interactions, several functions (changePassword, receiveNotification, getUser) are imported. GetUser obtains the user's preference regarding enabling email notifications, which is used as the default value in the switch selector.

Command Dashboard

Users with admin and user rights can also access the Command Dashboard via the dropdown menu located in the Navbar. Users with view rights have no access to it and cannot access it.

Users can activate or deactivate pumps and solenoid valves in the Commands Dashboard.



Welcome again **admin**

Ordre enviada a la planta. Si us plau, espera a que el valor s'actualitzi en els pròxims minuts per validar que l'ordre s'ha executat correctament.

Station	Components
Bioreactor	PB1 75% PB2 25% PB3 25% PB4 100% VB Closed V3 Open V1 Open VMM Closed

© 2023 Biometallum App. Build by Rubendelar

Figure 11: Command window (desktop viewport)

A useEffect hook retrieves station and component data on initial load, including whether certain components are actionable (canAct).

Consequently, all the components which appear in the user interface are pumps and solenoid valves. If the user sends a command with the associated id of the component, it will be an actionable component.

A WebSocket connection allows users to send commands to the plant. In the case of pumps, it updates the local state, inputValues, whenever the input field changes. Since solenoid valves only have open and close positions, it toggles the status of the component (from 0 to 1 or vice versa) and sends the command via websocket.

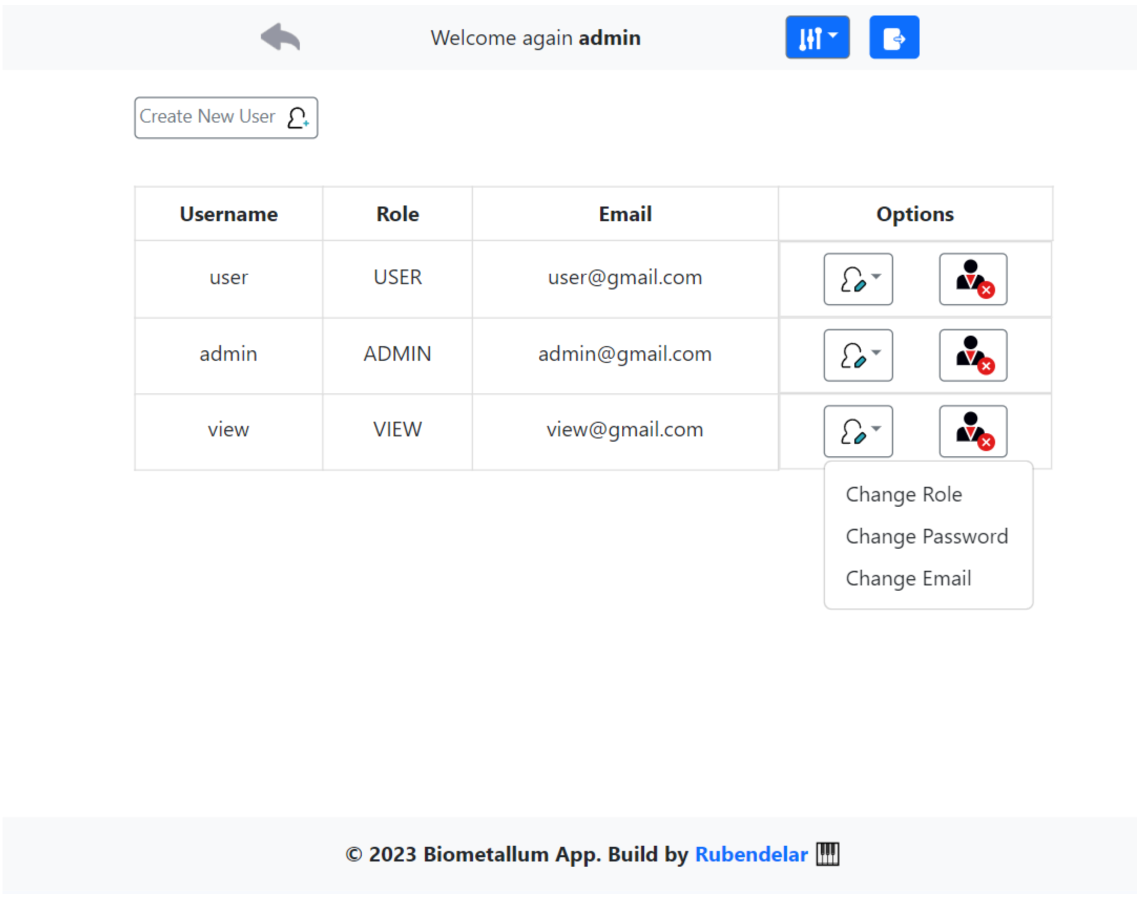
However, the status button does not change to the command value if the command is successfully sent via websocket. Rather than that, it displays a message warning that the plant may not have listened to the new register even if the command was successfully sent. The plant will appear on screen when it sends the new register. To doi that, an interval of 30 seconds is set up to automatically fetch register values of certain components at a specific interval, which is crucial for real-time communication.

Admin views

Administrators will see a dropdown menu in the navbar with links that are only accessible to them. A link to the user management page allows the admin to create or delete users, modify roles, passwords, and emails. Another link takes the admin to the component control page.

User Dashboard

Users can be created or deleted. In addition, roles, passwords, and emails can be modified in the user management screen.


Welcome again **admin**

Create New User

Username	Role	Email	Options
user	USER	user@gmail.com	
admin	ADMIN	admin@gmail.com	
view	VIEW	view@gmail.com	

© 2023 Biometallum App. Build by Rubendelar

Figure 12: User window (desktop viewport)

Administrators can create user accounts and edit their information on the UserPanel. using a variety of methods from the API. Passwords are always encoded using the passwordEncoder bean to ensure their security.

Component Dashboard

Another administrator-only dashboard is the component control screen, which displays the components assigned to each station in a table. New components can be created, assigned to the chosen station, deleted, or modified. For instance, an administrator can change the name of the PH component to PH_Bioreactor.

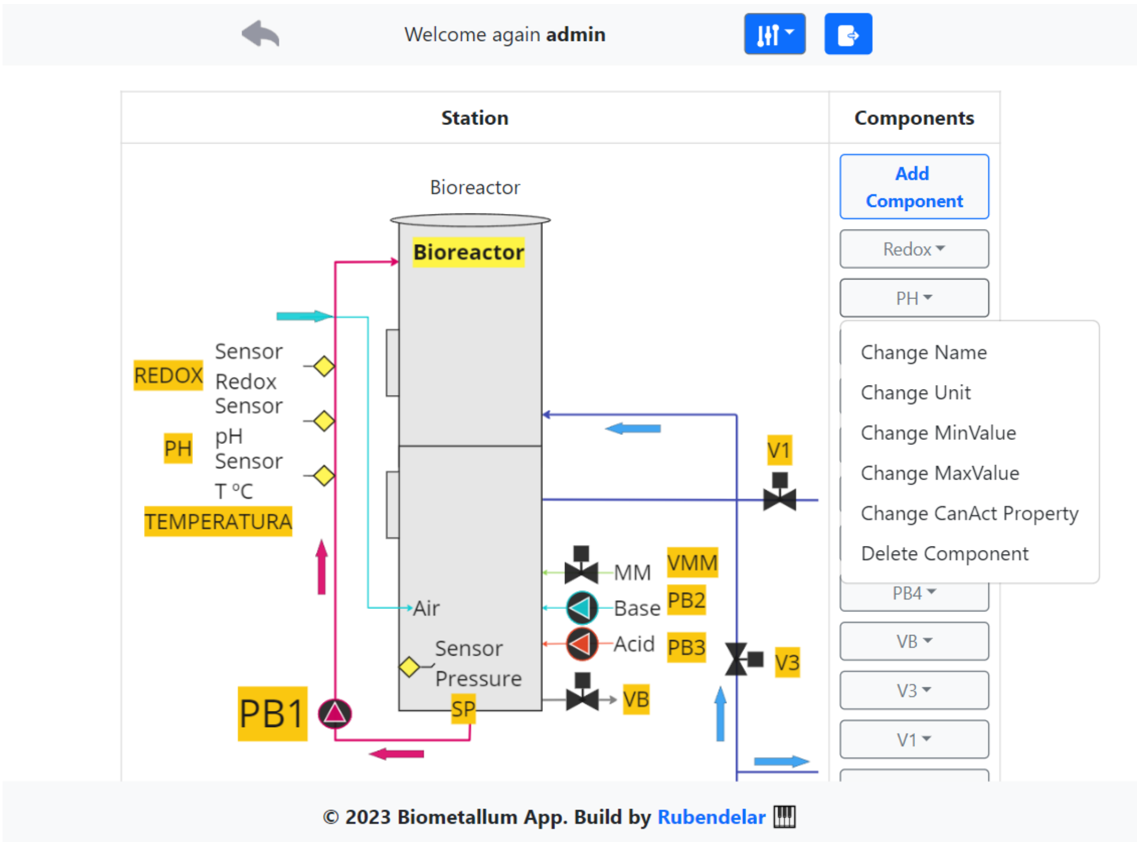


Figure 13: Component window (mobile viewport)

The app's scalability relies heavily on this screen. Future station components can be added directly from the app if needed. Once a component is created, its ID is auto-generated and can be given to the sensor technician who will map the component id with the sensor, making all new sensor data visible and allowing activation and deactivation of canAct components. When introducing a new pump or solenoid valve, the "canAct " attribute should be set to true after the new component has been created.

As of now, only the display images are not scalable. The schematic image can only be updated by someone familiar with the app code.

7. Conclusion

As a result of this project, a Web application for monitoring a bioleaching plant has been developed successfully. To meet the project's objectives and specifications, a robust application was developed that provides detailed insights into the plant's status and development. To achieve these goals, well-considered decisions were made that ensured efficient and secure communication.

The user interface design stands as a testament to our commitment to delivering real-time, user-friendly information. It offers extensive functionality for interaction and data retrieval from the plant's database, all while respecting each user's specific permissions.

The collected information is easily accessible and is presented intuitively with visual aids that enhance the user experience.

These tools created throughout the project will continue to play a pivotal role in the ongoing development phases of the plant, assisting in its monitoring and data recording.

8. References

1. Rittinghouse J W, Ransome J F. Cloud Computing: Implementation,Management, and Security. Boca Raton: CRC Press, 2016.

2. Zhang Y F, Zhang G, Wang J Q, et al. Real-time information capturing and integration framework of the internet of manufactur-ing things. International Journal of Computer Integrated Manufac-turing, 2015, 28(8): 811–822.

3. Farsi, M.; Daneshkhah, A.; Hosseinian-Far, A.; Jahankhani, H. (Eds.)Digital Twin Technologies and Smart Cities; Springer:Berlin/Heidelberg, Germany, 2020.

4. Majeed, M.A.A.; Rupasinghe, T.D. Internet of things (IoT) embedded future supply chains for Industry 4.0: An assessment froman ERP-based fashion apparel and footwear industry.Int. J. Supply Chain. Manag.2017,6, 25–40.

5. Oliff, H.; Liu, Y. Towards Industry 4.0 Utilizing Data-Mining Techniques: A Case Study on Quality Improvement.Procedia CIRP2017,63, 167–172.

6. Ovsthus, A.A.K.S.K.; Kristensen, L.M. An Industrial Perspective on Wireless Sensor Networks—A Survey of Requirements,Protocols, and Challenges.IEEE Commun. Surv. Tutor.2014,16, 1391–1412.

7. Muneeba, N.; Javed, A.R.; Tariq, M.A.; Asim, M.; Baker, T. Feature engineering and deep learning-based intrusion detectionframework for securing edge IoT.J. Super Comput.2022, 1–15.

8. S. H. Ahmed, G. Kim, and D. Kim, “Cyber physical system:Architecture, applications and research challenges,” inProc. IFIPWireless Days (WD), Valencia, Spain, Nov. 2013, pp. 1–5.

9. F.-J. Wu, Y.-F. Kao, and Y.-C. Tseng, “Review: From wireless sensornetworks towards cyber physical systems,”Pervasive Mobile Comput.,vol. 7, no. 4, pp. 397–413, Aug. 2011.

10. A. A. Cardenas, S. Amin, and S. Sastry, “Secure control: Towardssurvivable cyber-physical systems,” inProc. 28th Int. Conf. Distrib.Comput. Syst. Workshops, Beijing, China, Jun. 2008, pp. 495–500.

11. R. G. Helps and S. J. Pack, “Cyber-physical system concepts forIT students,” inProc. 14th Annu. ACM SIGITE Conf. Inf. Technol.Educ. (SIGITE), Orlando, FL, USA, Oct. 2013, pp. 7–12.

12. J. Lin, W. Yu, X. Yang, G. Xu, and W. Zhao, “On false data injec-tion attacks against distributed energy routing in smart grid,” inProc.IEEE/ACM 3rd Int. Conf. Cyber-Phys. Syst. (ICCPS), Beijing, China,Apr. 2012, pp. 183–192.

13. J. Linet al., “A novel dynamic en-route decision real-time routeguidance scheme in intelligent transportation systems,” inProc. IEEE35th Int. Conf. Distrib. Comput. Syst. (ICDCS), Columbus, OH, USA,Jun. 2015, pp. 61–72.

14. A. A. Cardenas, S. Amin, and S. Sastry, “Secure control: Towardssurvivable cyber-physical systems,” inProc. 28th Int. Conf. Distrib.Comput. Syst. Workshops, Beijing, China, Jun. 2008, pp. 495–500.

15. X. Yanget al., “A novel en-route filtering scheme against false datainjection attacks in cyber-physical networked systems,”IEEE Trans.Comput., vol. 64, no. 1, pp. 4–18, Jan. 2015.

16. Devesh, M.; Kant, A.K.; Suchit, Y.R.; Tanuja, P.; Kumar, S.N. Fruition of CPS and IoT in Context of Industry 4.0. InIntelligentCommunication, Control and Devices; Springer: Singapore, 2020; pp. 367–375.

17. Saniuk, S.; Grabowska, S.; Gajdzik, B. Social expectations and market changes in the context of developing the Industry 4.0concept.J. Sustain.2020,12, 1362.

18. Deepa, N.; Pham, Q.V.; Nguyen, D.C.; Bhattacharya, S.; Prabadevi, B.; Gadekallu, T.R.; Pathirana, P.N. A survey on blockchain forbig data: Approaches, opportunities, and future directions.Future Gener. Comput. Syst.2022,131, 209–226.

19. Number of Internet of Things (IoT) Connected Devices Worldwide from 2019 to 2030. 2021. Available online: https://www.statista.com/statistics/1183457/iot-connected-devices-worldwide/ (accessed on 20 July 2023).

20. Jovanovi´c, B. Key IoT Statistics. 2021. Available online: Https://dataprot.net/statistics/iot-statistics/ (accessed on 10 July 2023).

21. AI Multiple.30 Internet of Things—IoT Stats from Reputable Sources in 2021. Available online: https://research.aimultiple.com/iot-stats/ (accessed on 04 July 2023).

22. O’Dea, S. Wide-Area and Short-Range IoT Devices Installed Base Worldwide 2014–2026.2021.Available online:Https://www.statista.com/statistics/1016276/wide-area-and-short-range-iot-device-installed-base-worldwide/ (accessed on 20 July 2023).

23. DIS. ISO. 9241-210: 2010. Ergonomics of human system interac-tion-Part 210: Human-centred design for interactive systems.International Standardization Organization (ISO), 2009.

24. Tseng M M, Jiao R J, Wang C. Design for mass personalization.CIRP Annals-Manufacturing Technology, 2010, 59(1): 175–178.

25. Schmidt R, Möhring M, Härting R C, et al. Industry 4.0—Potentialsfor creating smart products: Empirical research results. In:International Conference on Business Information Systems. 2015,16–27.

26. Janak L, Hadas Z. Machine tool health and usage monitoringsystem: An intitial analyses. MM Science Journal, 2015, 2015(4):794–798.

27. Qiu X, Luo H, Xu G Y, et al. Physical assets and service sharing for IoT-enabled supply hub in industrial park (SHIP). InternationalJournal of Production Economics, 2015, 159: 4–15.

28. Wang M L, Qu T, Zhong R Y, et al. A radio frequencyidentification-enabled real-time manufacturing execution systemfor one-of-a-kind production manufacturing: A case study in mouldindustry. International Journal of Computer Integrated Manufactur-ing, 2012, 25(1): 20–34.

29. Rajalingam S, Malathi V. HEM algorithm based smart controller forhome power management system. Energy and Building, 2016, 131:184–192.

30. Javed A, Larijani H, Ahmadinia A, et al. Smart random neuralnetwork controller for HVAC using cloud computing technology.IEEE Transactions on Industrial Informatics, 2016, (99): 1–11.

31. Zhong R Y, Huang G Q, Lan S, et al. A two-level advancedproduction planning and scheduling model for RFID-enabledubiquitous manufacturing. Advanced Engineering Informatics,2015, 29(4): 799–812.

32. Wang X V, Xu X W. A collaborative product data exchangeenvironment based on STEP. International Journal of ComputerIntegrated Manufacturing, 2015, 28(1): 75–86.

33. Zhong R Y, Li Z, Pang A L Y, et al. RFID-enabled real-timeadvanced planning and scheduling shell for production decision-making. International Journal of Computer Integrated Manufactur-ing, 2013, 26(7): 649–662.

34. Zhong R Y, Newman S T, Huang G Q, et al. Big data for supplychain management in the service and manufacturing sectors:Challenges, opportunities, and future perspectives. Computers & Industrial Engineering, 2016, 101: 572–591.

35. Zhang L, Luo Y, Tao F, et al. Cloud manufacturing: A newmanufacturing paradigm. Enterprise Information Systems, 2014, 8(2): 167–187.

36. Zhong R Y, Huang G Q, Lan S L, et al. A big data approach forlogistics trajectory discovery from RFID-enabled production data.International Journal of Production Economics, 2015, 165: 260–272.

Part II.

APPENDIX

Appendix

A. Source Code

The file Biometallum_APP.zip contains the source code files. The Developer Manual C describes some of these files.

FRONTEND

```
 index.js
\components
    api.js
    App.css
    App.js

  \mainComponents
    \alerts
        Alerts.css
        Alerts.js

    \dashboard
        CommandPanel.css
        CommandPanel.js
        ComponentPanel.js
        Panel.css
        UserPanel.js

      \images
          add_component.svg
          add_user.svg
          Bioreactor.png
          delete_user.svg
          edit_user.svg
          Electrowinning.png
        Leaching.png
\footer
      Footer.js

\navs
      NavBar.css
      NavBar.js
      NavSide.css
      NavSide.js

   \icons
        back_arrow.svg
        control_icon.svg
        list_icon.svg
        logout_icon.svg
        reload.svg

\notifications
      Notifications.css
      Notifications.js

\egisters
      Registers.css
      Registers.js
      RegistersGraphics.js

\Stations
      DisplayComponent.css
      PlantOverview.js
      Station.js
```

```
    \images
        bioreactor_squema.jpg
        electrowinning_squema.jpg
        leaching_squema.jpg
        logo.jpg
        logovector.svg
        logo_epsem.png
        plant_squema.jpg
        plant_squema_obsolete.jpg
        ruler.svg

\router
    AdminRoute.js
    PrivateRoute.js
    PublicRoute.js
    UserRoute.js

\views
  \Admin
      CommandDashboard.js
      ComponentDashboard.js
      UserDashboard.js

  \AlertsPanel
      AlertsPanel.css
      AlertsPanel.js

  \Home
      Home.css
      Home.js
    \NotificationsPanel
        NotificationsPanel.css
        NotificationsPanel.js

    \RegistersPanel
        RegistersPanel.css
        RegistersPanel.js

    \UserLogs
        SignIn.css
        SignIn.js

\config
  \router
      paths.js

\contexts
    alertContext.js
    authContext.js

\utils
    indexedDB.js
```

BACKEND

```
\server
    ServerApplication.java

  \config
      PlantWebSocketController.java
      WebSocketConfiguration.java
      WebSocketResponse.java

  \controllers
    \impl
        AlertController.java
        ComponentController.java
        MessageController.java
        RegisterController.java
        RoleController.java
        StationController.java
        UserController.java

    \interfaces
        AlertControllerInterface.java
        ComponentControllerInterface.java
        MessageControllerInterface.java
        RegisterControllerInterface.java
        RoleControllerInterface.java
        StationControllerInterface.java
        UserControllerInterface.java

  \dtos
      ComponentThingDTO.java
```

PlantCommandDTO.java

PlantDataDTO.java

RegisterDTO.java

RoleToUserDTO.java

UserDTO.java

\filters

CustomAuthenticationFilter.java

CustomAuthorizationFilter.java

\models

Alert.java

ComponentThing.java

Register.java

Role.java

Station.java

User.java

\repositories

AlertRepository.java

ComponentRepository.java

RegisterRepository.java

RoleRepository.java

StationRepository.java

UserRepository.java

\security

CorsConfiguration.java

SecurityConfig.java

\services

```
  \impl
      AlertService.java
      ComponentService.java
      RegisterService.java
      StationService.java
      UserService.java

  \interfaces
      AlertServiceInterface.java
      ComponentServiceInterface.java
      RegisterServiceInterface.java
      StationServiceInterface.java
      UserServiceInterface.java

\ utils
  email
    \impl
        EmailService.java

    \interfaces
        EmailServiceInterface.java

  message_formatter
      MessageFormatterImpl.java
      MessageFormatterInterface.java

  \schedule
    \impl
        ScheduleRegisterAlert.java
    \interfaces
        ScheduleRegisterAlertInterface.java
```

B. Plant Components

Bioreactor Components:

- **Redox**
- **PH**
- **Temperatura**
- **SP**
- **PB1**
- **PB2**
- **PB3**
- **PB4**
- **VB**
- **V3**
- **V1**
- **VMM**

Redox

○ id:	4
○ name:	Redox
○ description:	Sensor Nivell Redox - Bioreactor
○ unit:	null
○ canAct:	false
○ valueType:	1
○ maxValue:	null
○ minValue:	null

- **PH**
  - id: 3
  - name: pH
  - description: Sensor Nivell pH - Bioreactor
  - unit: null
  - canAct: false
  - valueType: 1
  - maxValue: 1.8
  - minValue: 1.6

- **Temperatura**
  - id: 6
  - name: Temperatura
  - description: Sensor Nivell Temperatura - Bioreactor
  - unit: ºC
  - canAct: false
  - valueType: 1
  - maxValue: null
  - minValue: null

- **SP**

id:	200
name:	SP
description:	Sensor Pressió - Bioreactor
unit:	null
canAct:	false
valueType:	3
maxValue:	null
minValue:	null

- **PB1**

id:	9
name:	PB1
description:	Bomba de Recirculació - Bioreactor
unit:	%
canAct:	true
valueType:	2
maxValue:	100
minValue:	0

- **PB2**

id:	10
name:	PB2
description:	Bomba de Subministrament - pH Base Bioreactor
unit:	%
canAct:	true
valueType:	2
maxValue:	100
minValue:	0

- **PB3**

id:	11
name:	PB3
description:	Bomba de Subministrament - pH Acid Bioreactor
unit:	%
canAct:	true
valueType:	2
maxValue:	100
minValue:	0

- **PB4**

id:	117
name:	PB4
description:	Bomba de Circulació - Sortida Electroobtenció / Entrada Bioreactor
unit:	%
canAct:	true
valueType:	2
maxValue:	100
minValue:	0

- **VB**

id:	112
name:	VB
description:	Electrovàlvula - Rebuig Bioreactor
unit:	null
canAct:	true
valueType:	2
maxValue:	1
minValue:	0

- **V3**
    - id: 116
    - name: V3
    - description: Electrovàlvula - Sortida Electroobtenció / Entrada Bioreactor
    - unit: null
    - canAct: true
    - valueType: 2
    - maxValue: 1
    - minValue: 0

- **V1**
    - id: 113
    - name: V1
    - description: Electrovàlvula - Sortida Bioreactor / Entrada Lixiviacio
    - unit: %null
    - canAct: true
    - valueType: 2
    - maxValue: 1
    - minValue: 0

- **VMM**

id:	111
name:	VMM
description:	Electrovàlvula - Medi M. Bioreactor
unit:	null
canAct:	true
valueType:	2
maxValue:	1
minValue:	0

Leaching Components:

- **PH2**
- **C**
- **RD1**
- **GR1**
- **BL1**
- **HL1**
- **LL1**
- **PL1**
- **PL2**
- **V1**
- **V2**

- **PH2**

id:	12
name:	PH2
description:	Sensor Nivell pH - Lixiviació
unit:	null
canAct:	No
valueType:	1
maxValue:	1.8
minValue:	1.6

- **C**

id:	13
name:	C
description:	Número compost de tots els colors - Lixiviació
unit:	null
canAct:	No
valueType:	1
maxValue:	null
minValue:	null

- **RD1**
  - id: 36
  - name: RD1
  - description: Color Vermell - Lixiviació
  - unit: null
  - canAct: No
  - valueType: 1
  - maxValue: null
  - minValue: null

- **GR1**
  - id: 37
  - name: GR1
  - description: Color Verd - Lixiviació
  - unit: null
  - canAct: No
  - valueType: 1
  - maxValue: null
  - minValue: null

- **BL1**

id:	38
name:	BL1
description:	Color Blau - Lixiviació
unit:	null
canAct:	No
valueType:	1
maxValue:	null
minValue:	null

- **HL1**

id:	14
name:	HL
description:	Sensor de Nivell Alt - Lixiviació
unit:	null
canAct:	No
valueType:	3
maxValue:	null
minValue:	null

- **LL1**

id:	15
name:	LL
description:	Sensor de Nivell Baix - Lixiviació
unit:	null
canAct:	No
valueType:	3
maxValue:	null
minValue:	null

- **PL1**

id:	16
name:	PL1
description:	Bomba de Recirculació - Lixiviació
unit:	%
canAct:	Sí
valueType:	2
maxValue:	100
minValue:	0

- **PL2**

id:	17
name:	PL2
description:	Bomba de Subministrament - pH Acid Lixiviació
unit:	%
canAct:	Sí
valueType:	2
maxValue:	100
minValue:	0

- **V1**

id:	113
name:	V1
description:	Electrovàlvula - Sortida Bioreactor / Entrada Lixiviacio
unit:	null
canAct:	Sí
valueType:	2
maxValue:	1
minValue:	0

- **V2**
    - id: 114
    - name: V2
    - description: Electrovàlvula - Sortida Lixiviació / Entrada Electroobtenció
    - unit: null
    - canAct: Sí
    - valueType: 2
    - maxValue: 1
    - minValue: 0

Electrowinning Components:

- **PB4**
- **VE**
- **V2**
- **SP2**
- **Tension**
- **Current**
- **C2**
- **RD2**
- **GR**
- **BL2**

- **PB4**
  - id: 117
  - name: PB4
  - description: Bomba de Circulació - Sortida Electroobtenció / Entrada Bioreactor
  - unit: %
  - canAct: true
  - valueType: 2
  - maxValue: 100
  - minValue: 0

- **VE**
  - id: 115
  - name: VE
  - description: Electrovàlvula de Recirculació - Electroobtenció
  - unit: null
  - canAct: true
  - valueType: 2
  - maxValue: 1
  - minValue: 0

- **V2**
    - id: 114
    - name: V2
    - description: Electrovàlvula - Sortida Lixiviació / Entrada Electroobtenció
    - unit: null
    - canAct: true
    - valueType: 2
    - maxValue: 1
    - minValue: 0

- **SP2**
    - id: 201
    - name: SP2
    - description: Sensor Pressió - Electroobtenció
    - unit: null
    - canAct: false
    - valueType: 3
    - maxValue: null
    - minValue: null

- **Tension**

id:	22
name:	Tension
description:	Sensor de Tensió - Electroobtenció
unit:	null
canAct:	false
valueType:	1
maxValue:	null
minValue:	null

- **Current**

id:	21
name:	Current
description:	Sensor de Corrent - Electroobtenció
unit:	null
canAct:	false
valueType:	1
maxValue:	null
minValue:	null

- **C2**

○ id:	23
○ name:	C2
○ description:	Número compost de tots els colors - Electroobtenció
○ unit:	null
○ canAct:	false
○ valueType:	1
○ maxValue:	null
○ minValue:	null

- **RD2**

○ id:	39
○ name:	RD2
○ description:	Color Vermell - Electroobtenció
○ unit:	null
○ canAct:	false
○ valueType:	1
○ maxValue:	null
○ minValue:	null

- **GR2**

id:	40
name:	GR2
description:	Color Verd - Electroobtenció
unit:	null
canAct:	false
valueType:	1
maxValue:	null
minValue:	null

- **BL2**

id:	41
name:	BL2
description:	Color Blau - Electroobtenció
unit:	null
canAct:	false
valueType:	1
maxValue:	null
minValue:	null

Database ER

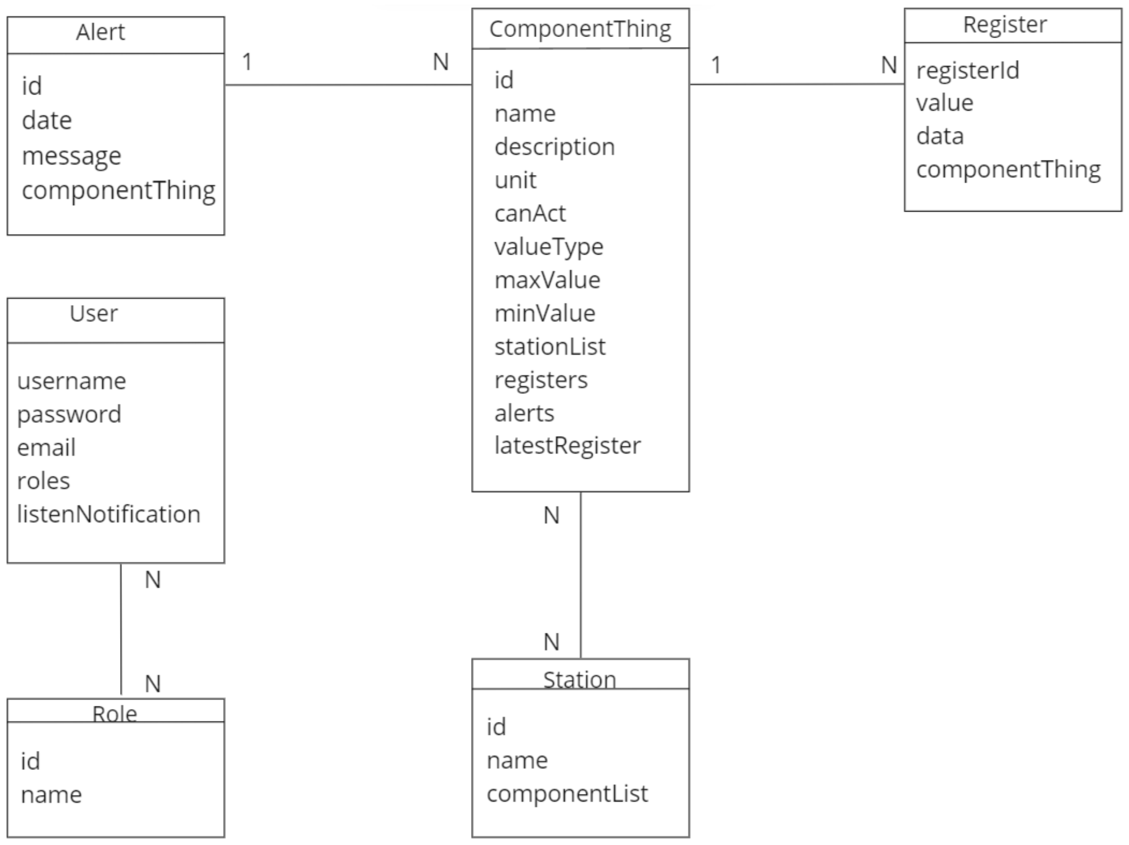


Figure 14 B:. Database ER schema. Own Source.

C. Developer’s Manual

The manual contains descriptions of the different modules of the project as well as instructions for maintaining them.

Front End React

The important directories for front end development are listed.

Biometallum_APP\FrontEnd\src\components\views\UserLogs\SignIn.js

SignIn uses React's useState hook to manage its local state. To control user input, toggle password visibility, display messages, and set alert types, states for username, password, showPassword, message, and alertVariant are managed, respectively.

In response to form submission, controlled by the handleSubmit function, the component attempts to authenticate the user. Authentication is accomplished by using the login function, extracted from the authentication context of the application, with the user's username and password entered. The setMessage function displays an error message if the login attempt fails, and it is automatically cleared after a predefined period of time.

1. **User Input**: Collects **username** and **password** via text inputs.
2. **Password Visibility Toggle**: Uses a clickable icon to toggle password visibility.
3. **Alert Notification**: Displays alert messages for authentication issues (e.g., invalid credentials) using React Bootstrap’s **Alert** component.
4. **Form Submission**: On form submission (**handleSubmit**):
    - Prevents default form submission behavior.
    - Attempts user authentication using **login**.
    - Sets an error message for failed authentication, which disappears after 6 seconds.
    - Displays appropriate messages and alters UI states as needed.

Biometallum_APP\FrontEnd\src\components\views\Home\Home.js

The home view utilizes a websocket connection to handle and manage real-time alerts, ensuring that users are notified instantly when important messages or alerts are received. Through the **websocketConnexion** function and the **useEffect hook**, these functions establish a websocket connection and set up a message handler to receive **incoming alerts** upon component mounting. In addition to being displayed in the user interface through addAlert, the received alerts are also stored in indexedDB through storeAlert, ensuring that they are accessible even when offline.

```
useEffect(() => {
  websocketConnexion((message) => {
    const alertMessage = message.body;
    addAlert(alertMessage);
    const alertData = {
        timestamp: Date.now(),
        message: alertMessage
    };
    storeAlert(alertData);
  });
}, [addAlert]);
```

Upon component mounting, a **WebSocket** connection is established using websocketConnection. When a message is received over the WebSocket, it extracts the message body, sends it to be displayed as an alert using **addAlert**, and stores the alert data (presumably for future reference or display) using storeAlert which is not defined in the provided code.

This component manages the state and persistence of the "**active**" **station** (e.g., Bioreactor, Leaching, Electrowinning) using the useState hook and localStorage. The activeStation state determines which station is currently selected and displayed to the user. The handleStationClick function enables users to select a station, and this choice is subsequently stored to localStorage, thereby preserving their selection regardless of whether they navigate away from the page or refresh it, improving the user experience by maintaining a consistent state across sessions.

As the activeStation state changes, the Home component renders different Station components conditionally. The station component is rendered with the appropriate stationId and image properties according to whether the activeStation is set to 'Bioreactor', 'Leaching', or 'Electrowinning'. Through this dynamic display, users can interactively select and view details related to different stations without having to navigate to different pages or routes, ensuring a streamlined and intuitive user experience.

Biometallum_APP\FrontEnd\src\components\mainComponents\registers\Registers.js

A number of useEffect hooks manage the **fetching**, **filtering**, and **updating** of register data, including **re-fetching** data at regular intervals and dynamically updating available filters:

```
useEffect(() => {
  // Fetching data...
  const intervalId = setInterval(() => {
    // Re-fetching data...
  }, 60000);


  return () => {
    clearInterval(intervalId);
  };
}, [stationFilter, componentFilter]);
```

When the component is unmounted, the cleanup function clears the interval, preventing possible memory leaks.

Data retrieval is orchestrated via API calls (such as fetchAllRegisters and fetchRegistersByStationId), and the results are stored in component state (rawRegisters). Data is **re-fetched every minute** by setting an interval, ensuring data remains up-to-date without the need for manual updating.

Using the **exportToExcel** function, users can export the filtered data to Excel, maintaining a desirable format and providing offline access and usability of the captured data. In addition to improving usability and facilitating rapid access to common views, several quick-access buttons allow users to specify predefined date ranges for data filtering. As a result of the selected component filters, either a detailed or summarized view is displayed based on the graphical display (via **RegistersGraphics**) and tabular data.

Biometallum_APP\FrontEnd\src\config\router\paths.js

The project route paths become a single source of truth when they are defined as constants (HOME, LOGIN, etc.). The result is that if it is needed to update a path, the app only has to do so in **one place** rather than finding and replacing it throughout the entire codebase, which reduces the risk of error. Here are all the project paths:

```
export const HOME = '/';
export const LOGIN = '/login'
export const LOGOUT = '/logout'
export const USERSDASHBOARD = '/users'
export const COMPONENTDASHBOARD = '/components'
export const COMMANDDASHBOARD = '/commands'
export const REGISTERS = '/registers'
export const NOTIFICATIONS = '/notifications'
export const ALERTS = '/alerts'
```

In order to define the navigation of the application, paths and components are used within a **router**:

```
<AuthProvider>
   <AlertsProvider>
   <BrowserRouter>
     <Routes>


     <Route path={LOGIN} element = {<PublicRoute />}>
       <Route path={LOGIN} element={<SignIn/>} />
     </Route>

     <Route path={HOME} element = {<PrivateRoute />}>
       <Route path={HOME} element={<Home/>} />
     </Route>

     <Route path={USERSDASHBOARD} element = {<AdminRoute />}>
       <Route path={USERSDASHBOARD} element={<UserDashboard />} />
     </Route>

     <Route path={COMPONENTDASHBOARD} element={<AdminRoute />}>
       <Route path={COMPONENTDASHBOARD} element={<ComponentDashboard/>} />
     </Route>

     <Route path={COMMANDDASHBOARD} element={<UserRoute/>}>
       <Route path={COMMANDDASHBOARD} element={<CommandDashboard/>} />
     </Route>

     <Route path={REGISTERS} element={<PrivateRoute/>}>
       <Route path={REGISTERS} element={<RegistersPanel/>} />
     </Route>

     <Route path={NOTIFICATIONS} element={<PrivateRoute/>}>
       <Route path={NOTIFICATIONS} element={<NotificationsPanel/>} />
     </Route>

     <Route path={ALERTS} element={<PrivateRoute/>}>
       <Route path={ALERTS} element={<AlertsPanel/>} />
```

```
        </Route>

        </Routes>
      </BrowserRouter>
      </AlertsProvider>
    </AuthProvider>
```

Biometallum_APP\FrontEnd\src\components\router\AdminRoute.js

The AdminRoute component is a protective route that validates whether a user is authenticated and holds the **'ADMIN'** role before allowing access to a particular navigational outlet. By using the **useAuthContext** custom hook, it is able to retrieve the user's authentication status and roles.

```
function AdminRoute() {
  const { isAuthenticated, roles } = useAuthContext();

  if (!isAuthenticated || roles !== 'ADMIN') {
    return <Navigate to={LOGIN} replace />
  }

  return <Outlet/>
}
```

It ensures that the subsequent rendered child routes or components within the Outlet are shielded and accessible only to authorized administrators if the user is not authenticated or does not possess the 'ADMIN' role.

Biometallum_APP\FrontEnd\src\components\router\UserRoute.js

In the UserRoute component, route access is managed using React and useAuthContext. Users who have authenticated and have a user role of either **'ADMIN'** or **'USER'** can access the route corresponding to their role. Otherwise, they are directed to the login page, ensuring limited access to the specified routes.

```
function UserRoute() {
  const { isAuthenticated, roles } = useAuthContext();

  if (!isAuthenticated || (roles !== 'ADMIN' && roles !== 'USER')) {
    return <Navigate to={LOGIN} replace />
  }

  return <Outlet/>
}
}
```

Biometallum_APP\FrontEnd\src\components\router\PrivateRoute.js

By using the useAuthContext hook, the PrivateRoute component ensures access is only granted to authenticated users.

Biometallum_APP\FrontEnd\src\components\router\PublicRoute.js

PublicRoute in React uses the useAuthContext hook to check user authentication and conditionally redirects authenticated users to the home page, preventing them from accessing public routes.

Biometallum_APP\FrontEnd\src\contexts\alertContext.js

It is a Context for managing alerts within an application.

- The **AlertsContext** is created using the createContext() method, which provides access to alert data and functions throughout the application.
- An **AlertsProvide**r provides alert data and functions to child components at a higher level in the component tree. It maintains the state of alerts using useState([]), starting with an empty array.
    - **addAlert(newAlert)**: The function addAlert takes a newAlert as an argument, and updates the alerts state by adding the new alert to the beginning of the previous alerts array, ensuring that the most recent alert is always displayed at the top of the array. It uses a functional update pattern to ensure that it always uses the most up-to-date state.
    - **Returning a Provide**r: The AlertsProvider returns an AlertsContext.Provider that provides the alerts array and addAlert function to any nested child components.
- **useAlertsContext**: A custom hook useAlertsContext is defined for consuming the context conveniently in function components. It utilizes useContext to access the AlertsContext and will throw an error if it is used outside of an AlertsProvider, ensuring proper usage of the context.

This structure allows any child component under AlertsProvider to access the current array of alerts and add a new alert.

Biometallum_APP\FrontEnd\src\contexts\authContext.js

The AuthContext is used to manage authentication and authorization. The AuthProvider component manages the state regarding user authentication, roles, and usernames, and provides login and logout functionality.

- **Authentication**: Asynchronous login functions retrieve data from a login API, set an **access_token** and user **roles** in **local storage**, and update authState to reflect the user role and authentication. When login fails, it logs errors and sets isAuthenticated to false.

- **Logout**: The logout function removes all user data and roles from local storage and sets isAuthenticated to false in the authState variable.

With the useAuthContext custom hook, child components can access authentication and user information, as well as login and logout methods, as long as it is used within an AuthProvider. This implementation is fundamental for restricting/allowing access to various parts of an application based on user authentication and roles.

Biometallum_APP\FrontEnd\src\utils\indexedDB.js

Alerts are managed using **IndexedDB**, a low-level API for storing structured data on the client side. This will allow users to check the alert registers even if they are online.

Here's a brief overview of the functionality within the code:

1. **Opening/Creating a Database:**
    - **dbName** and **storeName** are constants defining the name of the database and object store respectively.
    - **dbOpenPromise**: A promise is utilized to manage the opening/creation of an IndexedDB database and its readiness for transactions. Event handlers manage success (**onsuccess**), error (**onerror**), and upgrade (**onupgradeneeded**) scenarios.
2. **Storing Alerts:**
    - **storeAlert(alert)**: An asynchronous function that stores an alert object in the IndexedDB once the database is ready. It performs a "readwrite" transaction on the object store and attempts to add the alert, logging whether the operation is successful or encounters an error.
3. **Fetching Alerts:**
    - **fetchAlerts(callback)**: Utilizes the opened database to retrieve all stored alerts, utilizing a "readonly" transaction. The alerts are fetched via **store.getAll()**, and if successful, they are passed to the provided callback function for further handling.
4. **Cleaning Up Old Alerts:**
    - **cleanupOldAlerts()**: This function scans through stored alerts and deletes any older than one month (calculated as a timestamp in milliseconds). A cursor is utilized to traverse available records, deleting where applicable and logging the process and/or any errors.

Back End Spring Boot

biometallum\server\config\WebSocketConfiguration.java

Java class that configures a WebSocket with **STOMP** (Streaming Text Oriented Messaging Protocol) protocol support in a Spring application. There are two possible routes:

/message:

- **API:**
  - The API sends a data object (alert message) through the **/message** endpoint.
- **Front End**:
  - The Front End listens to the API messages on **/message** prefixed destinations.

/info:

- **Plant DevicesI**:
  - Plant devices send an array of objects containing register values and component ids through the **/info** endpoint.
- **API**:
  - The API listens to messages on **/info** prefixed destinations, acting upon received data from the plant devices (registers generation).
- **Front End:**
  - Sends an object to plant devices containing component id and a command using the **/info** endpoint.

As a practical implementation for testing, the following code demonstrates how to establish a WebSocket connection for real-time data communication using a STOMP-like protocol. By adhering to STOMP's structure, communications can be standardized, reliable, and organized, resulting in smooth data flow between the server and the client, particularly in a system that transmits critical or live data, such as a plant information system.

The script is used to send test messages to ensure that the server is able to understand and process the JSON format and structure of plant incoming objects, which will ultimately be used to generate an register.

```
import websocket
import json

def on_message(ws, message):
   print("Received: {}".format(message))

def on_error(ws, error):
   print("Error: {}".format(str(error)))

def on_close(ws, close_status_code, close_msg):
   print("Closed")

def on_open(ws):
   connect_frame = "CONNECT\naccept-version:1.1,1.0\n\n\x00"
   ws.send(connect_frame)

   subscribe_frame = "SUBSCRIBE\nid:1\ndestination:/ws/info\n\n\x00"
   ws.send(subscribe_frame)

   data_to_send = [
      {"id": 3, "value": 5},
      {"id": 12, "value": 1.5},
        {"id": 17, "value": 100}
    ]

    send_frame = "SEND\ndestination:/ws/info\n\n" + json.dumps(data_to_send) +
"\n\x00"
    ws.send(send_frame)

ws_url = "ws://localhost:8080/ws/"
websocket.enableTrace(True)
ws = websocket.WebSocketApp(ws_url,
                            on_message=on_message,
                            on_error=on_error,
                            on_close=on_close,
                            on_open=on_open)

ws.run_forever()
```

biometallum\server\config\PlantWebSocketController.java

Handles JSON messages related to "Plant" information. The WebSocket protocol provides full-duplex communication channels over a single TCP connection. Below is a breakdown of the code snippets provided:

Main Message Handling:

handlePlantMessage(String message)

- **Purpose**: To receive, deserialize, and handle JSON messages.
- **Operations**:
    - Deserializes received JSON string messages.
    - Handles messages which can be an array of JSON objects or a single JSON object.
    - For arrays, it iterates through each object and processes them.
- **Error Handling**: Logs and prints stack traces of JSON processing errors.

JSON Object Processing:

processJsonObject(JsonNode jsonObject)

- **Purpose**: To determine the type of JSON object and process it accordingly.
- **Operations**:
    - Checks if JSON objects have certain properties ("command" or "value") and directs them to appropriate processing functions.
    - Handles arrays and single objects differently to construct a list of data.

Plant Data Processing:

processPlantData(List<PlantDataDTO> incomingDataList)

- **Purpose**: To process a list of plant data objects and handle them by creating new registers.
- **Operations**:
  - Iterates through a list of **PlantDataDTO** objects, validates data, and manages it according to business logic.
  - Retrieves **ComponentThing** objects, validates, creates new **Register** objects, and saves them using the registers repository.

Plant Command Processing:

processPlantCommand(PlantCommandDTO command)

- **Purpose**: To process received commands and send responses back through WebSocket.
- **Operations**:
  - Logs the received command.
  - Logic related to command processing is presumed to be implemented.
  - Sends back a JSON response through WebSocket on the "/message/response" endpoint.

biometallum\server\filters\CustomAuthenticationFilter.java

Custom authentication filter named **CustomAuthenticationFilter**. This filter extends the **UsernamePasswordAuthenticationFilter** class provided by the Spring Security framework and is tailored to handle user authentication within the application.

-attemptAuthentication

- Retrieves **username** and **password** from the HTTP request.
- Generates an **UsernamePasswordAuthenticationToken** using credentials.
- Calls the **authenticate** method of **authenticationManager** to perform the actual authentication.

-successfulAuthentication

- Generates a JWT (JSON Web Token) with HMAC256 encoding, containing user details and roles.
- Sets the expiration time for the token.
- Creates a response map containing the token and user roles.
- Specifies the response type as JSON and allows cross-origin requests from a specific url.
- Writes the token and roles into the response output stream.

biometallum\server\filters\CustomAuthorizationFilter.java

CustomAuthorizationFilter, is a custom implementation of a filter within the Spring framework, designed to handle authorization logic within a Spring Boot application. This filter extends **OncePerRequestFilter**, meaning that this filter will be executed once per request.

- **Authenticate Requests**: Validate JWT (JSON Web Token) tokens sent in HTTP request headers for secured endpoints.
- **Allow Certain Requests**: Bypass the filtering for specific endpoints like **/api/login**. For other endpoints, it verifies the JWT token. If token verification is successful, retrieves user details and sets authentication in the security context.

biometallum\server\models\ComponentThing.java

1. **Basic Attributes**:
    - **id**: Unique identifier for each **ComponentThing**.
    - **name**: A string name of the component.
    - **description**: A string that holds a description.
    - **unit**: A string for unit measurement, with a custom column definition allowing NULLs.
    - **canAct**: A boolean that perhaps indicates whether the component can perform an action.
    - **valueType**: A Long presumably representing a type of value.
    - **maxValue** and **minValue**: Float values setting upper and lower limits.
2. **Relationships**:
    - **stationList**: A many-to-many relationship with **Station**.
    - **registers**: A one-to-many relationship with **Register**.
    - **alerts**: A one-to-many relationship with **Alert**, with cascade type ALL.
3. **Non-persisted Attribute**:
    - **latestRegister**: Not stored in the DB, only used in application logic.

biometallum\server\utils\email\impl\EmailService.java

The **sendEmailForTheAlert** method it's designed to send an email alert based on a **ComponentThing** and a message to a **User**.

1. **Email Sender and Helper**: Utilizes **MimeMessage** and **MimeMessageHelper** to construct and send an email.
2. **Conditional Sending**: Emails are sent conditionally if the **User** object has a non-null email address.
3. **Email Contents**: The email’s subject and body are dynamically generated based on the **ComponentThing** and **messageText** parameters.

biometallum\server\utils\schedule\impl\ScheduleRegisterAlert.java

The class uses the **@Scheduled** annotation to periodically perform these checks. Below are explanations and breakdowns of the different alerts and how they are executed:

checkAlertWithTheLastestRegister Method:

- **Functionality**: Iterates through all users and performs several alert checks, as defined by the methods called within the loop (**phAlertCheck**, **hl1AlertCheck**, **ll1AlertCheck**, **spAlertCheck**, and **sp2AlertCheck**).
- **Scheduling**: Utilizing **@Scheduled(fixedRate = 1000 * 60)**, this method will be triggered every minute (60,000 milliseconds).

canSendAlert Method:

- **Functionality**: Determines whether it's permissible to send an alert for a given component, based on the time of the last alert sent for that component.
- **Mechanism**: Maintains a **Map<Long, LocalDateTime> lastAlertTime** that keeps track of the last time an alert was sent for each component.

phAlertCheck Method:

- **Alert Type**: pH level checks for two components (Bioreactor and Leaching).
- **Functionality**: Validates if the pH level recorded in the latest register entry is outside the defined minimum or maximum limits.
- **Criteria**: If a pH reading is above **maxValue** or below **minValue**, and no alert has been sent for this component in the last 12 hours, an alert is sent.

hl1AlertCheck Method:

- **Alert Type**: High-level alert (probably related to a high liquid level or similar) in the Leaching station.
- **Functionality**: Validates if the latest reading is above a predetermined level (value is still to be defined).
- **Criteria**: If the reading is higher than the predetermined level and it's permissible to send an alert, an alert is dispatched.

ll1AlertCheck Method:

- **Alert Type**: Low-level alert in the Leaching station.
- **Functionality**: Checks if the latest register entry has a value below a certain threshold (value is still to be defined).
- **Criteria**: If the reading is below the predetermined level and it's permissible to send an alert, an alert is dispatched.

spAlertCheck and sp2AlertCheck Methods:

- **Alert Type**: Pressure sensor high-level alert for prolonged periods in the Bioreactor (spAlertCheck) and Electrowinning (sp2AlertCheck) stations.
- **Functionality**: Validates if the pressure reading is above a level (value is still to be defined) and has been in this state for more than 10 minutes.
- **Criteria**: If the above conditions are true and no alert has been sent for this component in the last 12 hours, an alert is sent.

Execution Flow:

1. **Scheduled Execution**: Every minute, **checkAlertWithTheLastestRegister** is invoked.
2. **User Iteration**: For each user, different alert check methods are invoked.
3. **Alert Checks**: Each alert check method will:
    - Retrieve the relevant component(s).
    - Validate if it's permissible to send an alert (using **canSendAlert**).
    - Fetch the latest register entries and assess whether alert criteria are met.
4. **Sending Alerts**: When alert criteria are met, a message may be sent in one of two ways:
    - Direct messaging via the **MessageController**.
    - Email notification if the user is configured to receive them.
5. **Alert Time Update**: After an alert is sent, the time is logged to prevent another alert within 12 hours of the previous alert.