POLITECNICO DI TORINO

Master degree course
in Computer Engineering

Master Thesis

Open-Source Microservices IoT Infrastructure for Monitoring LoRa-based Industrial Sensor Networks

Supervisors
Prof. Alessio CARULLO
Prof. Simone CORBELLINI
Co-supervisor
Eng. Marco SENTO

Candidate
Christian DELLISANTI

Accademic year 2024-2025

Summary

In recent years, with the advent of the Internet of Things (IoT) and the transition from Industry 4.0 toward Industry 5.0, the amount of data generated by connected devices has grown exponentially. Sensors, industrial machinery and smart devices are continuously producing information that needs to be collected, processed and interpreted in real time. This growth of data poses new challenges in terms of scalability, interoperability and efficient information management. In the enterprise environment, especially in production lines and industrial automation the need to collect data from sensors is critical to monitor and optimize as similar to calibrate the machinery. However, many proprietary solutions create closed ecosystems, making integration between devices from different manufacturers difficult. For this reason, it is of significant importance the development of a standardized, open-source infrastructure that enables the management of data from heterogeneous sensors while ensuring scalability, reliability and ease of integration. A critical issue in today's enterprises is the lack of effective solutions for monitoring sensor status and calibration. Although many industries have implemented data acquisition systems for production control, these systems often do not provide detailed insight into the proper operation and reliability of the sensors themselves. In many cases, calibration is performed manually or with instruments that are not interconnected to the rest of the enterprise digital infrastructure, increasing the risk of errors and inefficiencies. In other situations the absence of continuous monitoring can lead to undetected failures and wasted resources. To solve these issues, it becomes essential to develop a platform capable of not only collecting data from sensors, but also monitoring their status and managing calibration processes in an automated manner. In this context, the use of a microservices architecture based on Kafka and Spring Boot represents an innovative and effective approach. Kafka enables reliable and scalable management of data flows, while Spring Boot facilitates the development and maintenance of modular and independent services. Beside different IoT communications technologies, the usage of the LoRa (Long Range) transceiver for sensor communication enables wide area coverage with low

power consumption, making it ideal for industrial environments. The objective of this work is therefore to design and develop a distributed and scalable system for managing a network of IoT sensors, with a focus on its application in monitoring the relevant industrial processes and the remote calibration of the nodes of the network. The platform aims to provide an open-source solution that can be adopted and customized by companies with different needs, reducing integration costs and improving the efficiency of production processes. To achieve the goal of efficient management, the system is composed of two main parts: the Last Mile Network (wireless sensor network) and the Core Network (microservices for data processing). The sensors, organized into Nodes, communicate primarly via LoRa with the Gateway, and The Core Server, based on a microservices architecture, uses Kafka to ensure reliable and scalable communications between asynchronous components, optimizing production efficiency and reducing integration costs. In conclusion, this work aims to develop a scalable and open-source system for managing IoT sensor networks in industrial environments, ensuring efficient monitoring and automated calibration through a reliable microservices infrastructure.

Acknowledgements

Contents

List of Figures

Listings

A computer is like air conditioning - it becomes useless when you open Windows
[Linus Torvalds]

Chapter 1

Introduction

Context and Motivation

The ongoing evolution of industrial systems, marked by the advent of Industry 4.0 and the emerging paradigm of Industry 5.0, is reshaping the manufacturing landscape. Companies are increasingly embracing digitalization, automation, and interconnected smart devices to build resilient, sustainable, and human-centric production ecosystems. Within this transformation, the management and certification of sensor networks become critical components for ensuring process reliability and quality control.

In this thesis, we focus on the design and implementation of an infrastructure dedicated to managing a network of sensors. The objective is to develop a comprehensive system capable not only of collecting and visualizing sensor data but also of guaranteeing its reliability through a certification process grounded in recognized standards.

Sensors are organized into groups, referred to as nodes, which represent the fundamental logical units of the network. Each node is visualized on a map, enabling centralized and interactive management of the distributed network. Users can access measurements recorded by each node and verify their calibration status by analyzing the associated calibration certificate. The calibration certificate adopted in this project complies with a modern, open standard developed by the

Physikalisch-Technische Bundesanstalt (PTB), Germany's national metrology institute. This standard promotes interoperability and automation in the management of metrological information.

Metrology and Calibration Process

Calibration is performed using a specialized node, called the reference node, characterized by higher sensitivity and precision compared to other nodes. This node serves as the benchmark for verifying the accuracy of measurements collected across the network, thereby enabling the certification of data quality. A significant challenge addressed in this thesis concerns the current industrial practices for sensor calibration management. Presently, calibration certificates are often produced manually and stored in paper format. There is no universally adopted standard for their structure, and certificates are generally not designed for digital processing. This traditional approach results in operational inefficiencies. The absence of digital management forces companies to manually monitor the validity period of each certificate. Every certificate has an expiration date beyond which the sensor cannot be considered metrologically reliable. Without an automated system, some sensors risk being used beyond their calibration validity, potentially compromising the integrity of the entire production process. Introducing a software infrastructure capable of centralized and automated calibration certificate management would constitute a significant advancement. Such a system can continuously monitor the calibration status of all sensors in the enterprise, issuing timely alerts when certificates approach expiration or when potential metrological drift is detected. This approach not only enhances operational efficiency but could also encourage the adoption of a common standard for managing digital calibration certificates. The digitalization of calibration processes thus aligns perfectly with the broader transition from Industry 4.0 to Industry 5.0, wherein automation and human centrality coexist within smarter, more resilient, and sustainable production ecosystems.

System Architecture and Approach

To meet the requirements of this transition, a comprehensive system is developed based on a distributed infrastructure and a web application. Central to this architecture are devices called gateways, designed to connect and communicate directly with the sensors, which are organized into logical units called nodes. Communication between nodes and gateways utilizes LoRa (Long Range), a wireless protocol that supports long-distance data transmission with extremely low energy consumption. This makes LoRa especially suitable for scenarios where sensors are dispersed across wide areas and continuous power supply is impractical. Data collected from nodes are forwarded by gateways to a central server hosting a web application accessible to companies. Through this interface, operators can visualize real-time measurements from various sensors and continuously monitor the calibration certificate status. The entire system aims to provide an effective, scalable, and automated solution for sensor metrology management, improving transparency and control within organizations and actively contributing to industrial digitalization.

Architectural Evolution and Justification

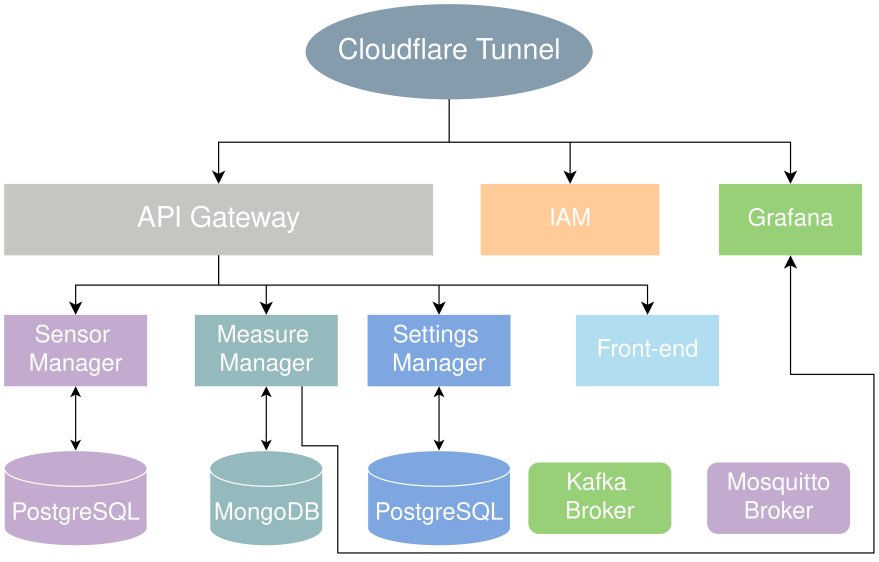

Figure 1.1: Server Architecture

Server architectures have evolved significantly over time, from monolithic designs to more modular and scalable approaches. Monolithic architectures consist of a single, indivisible application block with tightly coupled components. While simpler to develop initially, this approach suffers from limitations in scalability, maintainability, and deployment agility—any modification requires redeploying the entire system. Conversely, microservices architectures decompose the system into independent modules, each responsible for a specific functionality. This division fosters flexibility by enabling autonomous design, testing, and deployment of each microservice, facilitating issue isolation and parallel development by multiple teams. A further advantage is the suitability for adopting Test Driven Development (TDD), where tests are written before code implementation, enhancing software reliability and goal-oriented design. In this thesis, the server is organized into three primary macro-components: front-end, back-end, and gateway. The back-end is further segmented into multiple microservices, including:

- IAM (Identity and Access Management): managing identities and permissions.
- Gateway API: manage the API of the back-end.
- Sensor Manager (SM): organizing and controlling sensors.
- Measure Manager (MM): processing and storing measurements.
- Settings Manager : processing commands to nodes.

Each microservice follows the Clean Code Pattern, also known as Onion Architecture, which structures code into concentric layers.[1] The outermost layer consists of controllers handling external communication, the intermediate layer comprises application services, and the innermost core contains domain entities representing the business logic. This design enables modularity, extensibility, maintainability, clear separation of concerns, and well-defined logical flows.

[1]for more about Clean Code Pattern see Robert C. Martin (2012)

Modern Web Application Development

All microservices are containerized using Docker, a platform that enables the development, deployment, and execution of applications within lightweight, portable, and self-sufficient environments known as containers. A container is a standardized unit of software that packages code and all its dependencies—such as libraries, configurations, and system tools—so that the application runs reliably and consistently across different computing environments. Docker containers provide isolated execution environments, ensuring that each microservice runs independently and does not interfere with others, even if they require different software versions or configurations. This isolation enhances reproducibility, scalability, and maintainability of the system.

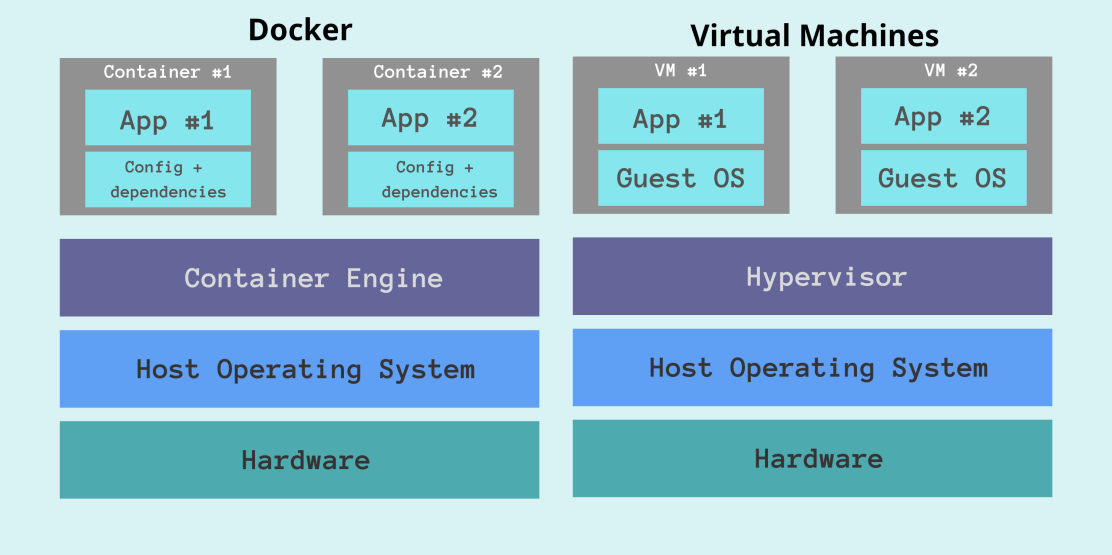

Figure 1.2: Docker vs VM infrastructure

Containers and virtual machines (VMs) are similar because thery are both technologies designed to provide isolated environments for running applications, yet they differ significantly in their underlying architecture and resource consumption. Virtual machines run full guest operating systems on top of a hypervisor, which emulates hardware resources and is managed by the host system. This allows for the simultaneous execution of multiple VMs with entirely different operating systems on a single physical machine. Containers, on the other hand, share the host operating system's kernel and package only the application and its dependencies, resulting in significantly lower overhead, faster startup times, and more efficient resource utilization. However, this shared-kernel model requires

that containers and host systems use the same base operating system (e.g., Linux). Despite these differences, containers and VMs share common goals and functionalities: both provide application isolation, enhance security, improve portability, and support automation in deployment and orchestration. They enable consistent and reproducible environments across development, testing, and production stages, and are widely used in modern distributed systems and cloud-native architectures. In essence, while containers are more lightweight and optimized for microservice-based applications, VMs remain advantageous in scenarios that require full OS-level separation and greater flexibility.[2]

To orchestrate the deployment of these services, Docker Compose is employed. It is a declarative, YAML-based tool that simplifies the management of multi-container applications by defining service configurations, dependencies, volumes, and exposed ports in a single file. This setup enables efficient and consistent initialization of the entire infrastructure on the server with minimal manual intervention.

[2]Karl Matthias Sean Kane (2023)

```yaml

services:
  sensor-manager:
    image: 'docker.io/christiand9699/sensor-manager:latest'
    restart: unless-stopped
    depends_on:
      - postgres
    expose:
      - '8080'
  measure-manager:
    image: 'docker.io/christiand9699/measure:latest'
    restart: unless-stopped
    depends_on:
      mongodb:
        condition: service_healthy
    expose:
      - '8080'
  frontend:
    image: 'docker.io/christiand9699/frontend-react:latest'
    restart: unless-stopped
    ports:
      - "5173:80"
    volumes:
      - ./default.conf:/etc/nginx/conf.d/default.conf
  gateway-iam:
    image: 'docker.io/christiand9699/gateway-iam:latest'
    depends_on:
      keycloak:
        condition: service_healthy
    restart: unless-stopped
    ports:
      - '8080:8080'
  settings:
    image: 'docker.io/christiand9699/settings:latest'
    restart: unless-stopped
    expose:
      - 8080
```

Listing 1.1: Example of Docker Compose File

Communication between Microservices

To enable efficient and scalable communication between microservices, Apache Kafka is employed as a message broker facilitating publish-subscribe event streams. Kafka is a distributed streaming platform designed for high-throughput, low-latency, and fault-tolerant handling of real-time data feeds. It acts as an intermediary that decouples producers (which generate data) from consumers (which process data), enabling asynchronous and scalable message exchange between services.

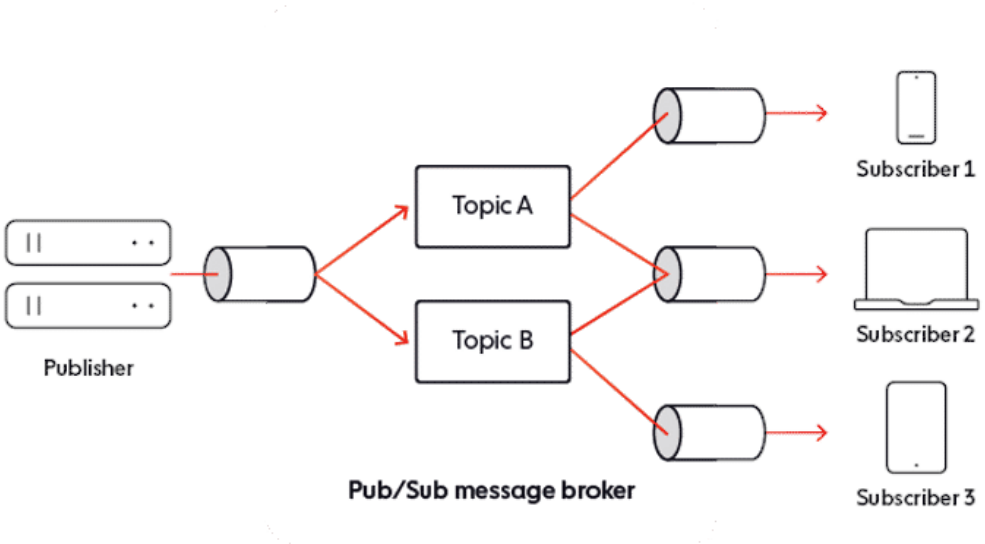

Figure 1.3: Kafka Pub-Sub Architecture

Each event published to Kafka represents a specific occurrence—such as a new sensor measurement or a change in sensor state. Events are organized into topics, which are durable and partitioned logs of messages. Kafka ensures resilience and message persistence by replicating data across multiple nodes in a cluster, making it robust to failures.

Moreover, Kafka supports complex workflows through patterns such as the Saga pattern, which is used to coordinate distributed transactions across independent microservices. This pattern helps maintain system-wide consistency by allowing

each local transaction to be executed independently, with compensating actions triggered in the event of failures or rollbacks, rather than relying on traditional two-phase commits.

Spring Boot

To develop each container in the back-end architecture, the Spring Boot framework was employed. Spring Boot is a powerful framework built on top of the Spring ecosystem that facilitates the rapid development of stand-alone, production-grade applications. It inherently supports and promotes several fundamental programming paradigms, among which Inversion of Control (IoC), Dependency Injection (DI), and Aspect-Oriented Programming (AOP) are of central importance.[3] Inversion of Control (IoC) is a design principle whereby the control over the creation and lifecycle of objects is transferred from the application itself to a dedicated container or framework. Instead of having application components instantiate and manage their dependencies, developers define how objects should be constructed and wired, and the framework orchestrates this process. This paradigm enables a decoupled architecture, increasing modularity, testability, and ease of maintenance. Dependency Injection (DI) is a specific and widely adopted form of IoC. It refers to the process by which an object's dependencies are provided to it by an external entity—typically the IoC container—rather than the object creating them internally. In Spring Boot, DI is implemented through annotations such as @Autowired, @Component, @Service, and @Configuration, allowing the framework to resolve and inject dependencies automatically at runtime. This promotes a clear separation of concerns and facilitates the reuse of components across the system. Aspect-Oriented Programming (AOP) complements IoC and DI by providing a mechanism to modularize cross-cutting concerns—such as logging, security, transaction management, or error handling—that tend to be scattered and tangled across multiple components. AOP achieves this by allowing developers to define "aspects" that encapsulate behaviors affecting multiple classes, which are then applied declaratively or programmatically without polluting the core business logic. In Spring, AOP is implemented via proxies and annotations such as @Aspect and @Around, enabling clean separation of concerns and improving code maintainability and readability.

[3]Ryan Breidenbach Craig Walls (2005)

Through the combined use of these paradigms, Spring Boot enables the design of a highly modular, loosely coupled, and scalable back-end architecture. Each service or container can be independently developed, tested, and deployed—characteristics that are particularly advantageous in a microservices-oriented system, where cohesion and autonomy of components are critical for success.

Open Philosophy

An additional fundamental aspect of this thesis is its adherence to an open philosophy. The system is designed with openness and interoperability in mind, leveraging open standards such as the PTB calibration certificate format and open-source technologies. This approach fosters transparency, encourages collaboration, and ensures that the infrastructure can evolve and integrate with other systems without vendor lock-in or proprietary limitations. It reflects a commitment to sustainable, future-proof solutions aligned with the evolving needs of industrial digitalization. To further embody this open philosophy, an organization named Measurestream has been created on GitHub, hosting all repositories related to this work. Every component is released as open-source under the GNU General Public License v3 (GPL-3.0), reinforcing the commitment to free software principles. The GPL-3.0 license ensures that anyone is free to use, study, modify, and distribute the software, provided that any derivative works are also distributed under the same license. This guarantees that the software and any improvements remain open and accessible to the community, protecting user freedoms and promoting collaborative development over time.

Chapter 2

Back-end

2.1 Sensor Manager

The sensor manager is the element responsible for managing all the sensors.[1] These sensors can be created, read, updated and deleted: all CRUD methods are provided. The sensors are represented by nodes, defined as entities composed by the union of multiple Control Units(CU) and Measurement Units(MU).
The definition of CU and MU is now offered for concision.

- The CU is the physical part of the sensor in charge of retrieving data from the MU and it sends the data to the Gateway – which is the hardware part that can make a "bridge" between sensors and server. The CU, in the present solutions, communicates through a LoRa(Long Range) network and employs a cryptography protocol to communicate with the server.
- On the other hand, the MU is the physical part of the sensor responsible for measuring the physical quantity it was designed for. It is essential that the MU is certified and calibrated.

Within the server, these two entities are designated as the Measurement Unit and the Control Unit, collectively forming a node. In the context of MeasureStream, a node can contain multiple CUs and MUs. The presence of different

[1] About the division in microservices see 13

MUs is attributed to the potential for diverse sensors and the objective to collect data from varying physical quantities. Conversely, the possibility of a node being composed of multiple CUs has been permitted, which may facilitate multiple connections with disparate technology networks (e.g., LoRa, Bluetooth).

The Sensor Manager(SM) is responsible for the management of all entities and operations within the system. The SM is organised in a microservice structure,[2] following the onion pattern or the Clean Code Pattern [3].
Consequently, it is divided into:

1. Controllers
2. Services
3. Repositories

The incoming and outgoing data are DTOs (Data Transfer Object), and it mirrors the entities that are the business core of the service. The controller is responsible for both the reception and transmission of DTOs, while the Service maps DTOs into entities and vice versa, whereas repositories retrieve entities from a DB.
It is imperative to note that entities never leave the application for security reasons – and also permit agile development.

REST controllers

There are five different REST controllers, which respond to:

1. /API/controlunits
2. /API/measurementunits
3. /API/nodes
4. /API/user
5. /API/dcc

All of them offer standard CRUD requests and also implement different services to manage the requests for each entity. These are: node, CU, MU, User and DCC. Each service has been designed to manage a specific entity: every service is an interface. The implementation of the services varies. Each entity contains different information and different properties:

[2]Further insights can be found on page 15

[3]Robert C. Martin (2012)

1. **Node:**
   - `id: Long` – Unique identifier of the node.
   - `name: String` – Name of the node (mandatory field).
   - `standard: Boolean` – Indicates whether the node is a standard or not.
   - `controlUnits: MutableSet<ControlUnit>` – Set of control units associated with the node.
   - `measurementUnits: MutableSet<MeasurementUnit>` – Set of measurement units associated with the node.
   - `location: Point` – Location of the node (geographical coordinates).
2. **ControlUnit:**
   - `id: Long` – Unique identifier of the control unit.
   - `networkId: Long` – Network identifier, must be unique.
   - `name: String` – Name of the control unit (mandatory field).
   - `remainingBattery: Double` – Percentage of remaining battery, with a maximum value of 100.
   - `rssi: Double` – Indicates the RSSI value, which must be negative or zero.
   - `node: Node?` – Node associated with the control unit.
3. **MeasurementUnit:**
   - `id: Long` – Unique identifier of the measurement unit.
   - `type: String` – Type of the measurement unit (mandatory field).
   - `measuresUnit: String` – Unit of measurement (mandatory field).
   - `networkId: Long` – Unique network identifier.
   - `dcc: DCC?` – DCC associated with the measurement unit.
   - `node: Node?` – Node associated with the measurement unit.

4. **User**

    - `userId: String` Unique identifier of the user. It is generated by the IDP[4] and it is retrived by the Identity Token.[5]
    - `name: String`
    - `surname: String`
    - `email: String`
    - `role: String`
    - `nodes: MutableSet<Node>` set of nodes owned by user.
    - `mus: MutableSet<MeasurementUnit>` set of MUs owned by user.
    - `cus: MutableSet<ControlUnit>` set of Cus owned by user.

5. **DCC**

    - `id: Long` Unique identifier of the DCC.
    - `expiration: LocalDate` indicates the expiration of the calibration document
    - `filename: String`
    - `pdf: ByteArray`
    - `mu: MeasurementUnit?`

The entities in question are interconnected, and they all bear specific relationships. For instance, a CU may be associated with a node, though this is not mandatory. This is due to the necessity of designing the system in such a manner that any replacing of a CU in some node can be performed without the node being deleted. Furthermore, an MU may be associated with a node for the same reason as for a CU. In addition, a node may contain multiple CUs and MUs. The Node, MU, and CU are associated with a user to ensure secure access for each individual user of the application. Lastly, each DCC, which contains the calibration information of a MU, is associated with a single MU via a one-to-one relationship.

[4]Identity Provider

[5]aggiungere citazione nota

Database

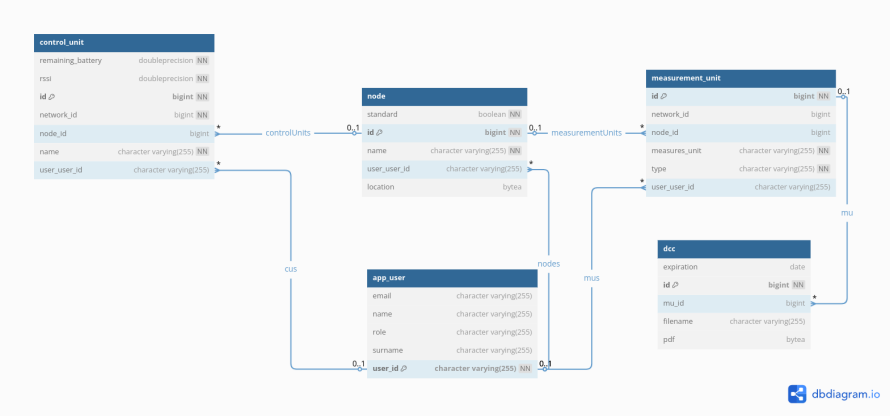

Figure 2.1: ER Diagram of Sensor Manager's Entities

The functionalities of SM are based on the usage of a DBMS (Database Management System). The relational database PostgresSQL was selected for this purpose, as it is ACID-compliant[6], i.e. it respects certain properties:

1. Atomicity ensures that all transactions are executed as a single, non-interrupted process. In the event of a failure in the DBMS or a transaction, the assurance of non-interruption is guaranteed, as no intermediate states are stored.

2. Consistency denotes the property by means of which data remain in a stable state. Consequently, in the event of an error, the consistency property ensures a rollback to the previous stable state.

3. Isolation is defined as the property that ensures that only one process at a time can modify the same data

4. Durability ensures that data can persist.

Moreover, PostgreSQL is characterised as an open-source software solution. It facilitates replication and partitioning, thereby enhancing the scalability of the database management system(DBMS) and the associated applications. In order to retrieve data, the software utilises SQL. The connection between the entities within the SM application and the underlying DBMS is managed through Spring Data JPA, which leverages Object-Relational Mapping(ORM). ORM is a programming

[6]Francis Botto (1999), p. 18

technique that automatically maps Java objects(entities) into relational database tables, allowing developers to interact with the database by using object-oriented paradigms. By this process, each entity class is mapped into a corresponding table, and each instance of the class represents a row in that table. Spring Data JPA abstracts much of the boilerplate code and ensures that all entity relationships – such as one-to-many or many-to-many – are correctly mapped and enforced according to the database schema.

Security

SM operates as a resource server within the OAuth2 security framework. It is secured by means of Spring Security and enforces access control by requiring a valid bearer token for all incoming requests. The token must be issued by a trusted authorization server, thereby ensuring that only authenticated and authorised clients may access protected endpoints. In the event of an unauthenticated or invalid token being supplied, the server responds with an HTTP status code, such as *401 Unauthorized*. SM is packaged as a Docker container to ensure portability and consistency across environments.[7] This container is uploaded to Docker Hub – making it easily accessible for deployment.[8] It is included in a Docker Compose Configuration, which orchestrates the launch of all microservices in the system, thus enabling seamless integration, network configuration, and environment variable management across services.

Messages

The SM publishes events to different Kafka topics to notify other components of the application about modifications in the sensor infrastructure, such as the creation or deletion of a Node, MU, or CU. These Kafka topics are named "cu-creation", "mus", and "node-event", respectively. Each topic receives JSON objects that follow a consistent schema composed of two fields: eventType and value. The eventType is a string that indicates the type of event, such as "CREATE" or "DELETE", while the value field contains the data related to the affected element. Specifically, it holds a NodeDTO for node-related events, a MeasurementUnitDTO for MU events, and a corresponding DTO for CU events.

[7] For background information, consult page 15.

[8] see `https://hub.docker.com/u/christiand9699`

2.2 Measure Manager

The Measure Manager(MM) manages all the measures retrived from the sensors. MM is organized according to a microservices structure which uses the Onion Pattern. MM exposes different URLs to only read data; however, it doesn't offer any URL to update or to create any kind of measures. The only way to insert measures in MM is by a Kafka connection between MM and the Gateway.[9]

Kafka is an open-source event streaming platform developed by the Apache Software Foundation.[10] It was designed for high-throughput, fault-tolerant, scalable handling of real-time data feeds. Kafka enables systems to publish, subscribe to, store, and process streams of records in a distributed and durable way. In particular, Kafka is a particular message broker that handles events. The term 'event' denotes an occurrence that has already transpired. Within the domain of information technology, 'event' is utilised to construct pipelines or to trigger sequences of actions among disparate entities that function asynchronously. The concept of event is predicated on the notion of occurrence and is in contrast with the idea of command – which is used to describe an action to be performed. The ideas of event and command can be used to orchestrate asynchronous actors that must cooperate, but the event concept needs an easier implementation. The concept of event fits perfectly within the requirements of collecting data from the sensors; for this reason I have chosen a message broker to connect Gateway and Server.

A further rationale for the selection of Kafka is its approach to the pubblication of events. Within a Kafka communication, three actors are to be identified: the publisher, the topic and the consumer. The producer is responsible for the transcription of the events while the topic serves as the repository for all events put in chronological order. In turn, the consumer has the capacity to retrieve specific events – or the entirety of events – within a given topic, at a time of their choosing. This mode of communication is particularly efficacious for asynchronous communication, as it does not necessitate the synchronisation of the producer and consumer. It can be regarded as a form of lazy communication in that the consumer has the capacity to access data at a time of their choosing – in contrast with other message broker like RabbitMQ, which it is not lazy.

[9]The Gatweay is the physical part that collects data from nodes and sends it to server, it must not be confused with Cloud Gateway microservice.

[10]Apache Software Foundation (2025)

REST controllers

MM serves different GET endpoints to retrive data and they are:

1. */API/measures/* retrives a list of all measures;
2. */API/measures/P* gets a list of all measures in pagination mode;
3. */API/measures/nodeId* responds with all the measures of a specific nodeId and a specfic unit of the sensor's measure[11] – the nodeId and the unit are specified with the usage of request parameters, which are the nodeId and the variable measureUnit. For instance, in order to get all Celsius measures from node 1 the correct endpoint is:

   /API/measures/nodeId/?nodeId=1&measureUnit=Celsius
4. */API/measures/measureUnitOfNode* accepts the request parameter nodeId and provides information regarding the unit of measures employed in the measures.
5. */API/measures/download* this endpoint allows clients to retrieve sensor measurement data in JSON format as a downloadable file. It accepts four query parameters: **start** and **end**, which are optional timestamps used to filter the data by a specific time interval, and **nodeId** and **measureUnit**, which are required to identify the source node and the type of measurement to be retrieved. When invoked, the endpoint queries the relevant measurements based on the provided filters, serializes the data into JSON, and returns it as a byte array within an HTTP response. The response includes headers that prompt the browser or client to download the file with the name measures.json, and the content type is set to application/json. This mechanism is useful for exporting data related to a specific node and measurement type, optionally within a defined time range, in a format suitable for further analysis or archiving.

The initial three endpoints also accept the start and end request parameters. These are always registered in UTC time and they possess the capability to filter data within a specified period.

[11]The term "unit of measure" must not be confused with Measurement Unit(abbreviated as MU): the first stands for the designated magnitude of a quantity, such as Celsius, Kelvin and the like; the latter denotes the physical part of a node which detects the measure. For more on MU cfr. 21

As previously outlined, MM adheres to the Clean Code Paradigm and, in response to the aforementioned parameters, it returns MeasureDTO. The core business of the microservice is Measure, a Kotlin object linked to a table or a collection in some DB[12], and it follows this schema:

Measure

- **id**: Unique identifier of the measurement, automatically generated by Database.
- **value**: Numerical value of the recorded measurement.
- **measureUnit**: Unit associated with the value (e.g., `°C`, `ppm`, `%RH`).
- **time**: Timestamp indicating when the measurement was taken. It must be in the past or present and it is in UTC.
- **nodeId**: Numerical identifier of the sensor node that generated the measurement. It must be zero or positive.

Database

It is evident that measures are stored within a database; conversely, the mapping between the database and the entities is provided by Spring Data JPA, thereby facilitating ORM(Object Relationship Mapping) – as was also previously evident in the SM. Whilst the most prominent types of NoSQL databases are Document, Key-Value, Wide-Column and Graph, I opted for MongoDB, a NoSQL database.[13][14] MongoDB is an open-source database based on documents with the fundamental premise of substituting the concept of a row with a more flexible model, the document. This document-oriented approach facilitates the representation of complex hierarchical relationships with a single record by allowing both embedded documents and arrays. MongoDB is also schema-free, meaning that a document's keys are not predefined or fixed in any way. This feature eliminates all requirements for extensive data migrations, as the system can adapt to changes without the need for significant alterations to its structure. New or missing keys can be addressed

[12] Additional information about Clean Code Pattern can be found on page 13.

[13] The term 'NoSQL' is an acronym for 'Not Only SQL', indicating that these databases do not store information in a relational manner – as is typical of SQL databases – instead it utilises different organisational methods for data storage

[14] For the official documentation on MongoDB, see MongoDB, Inc. (2025)

at the application level, avoiding the need to impose a uniform data structure on all data. This affords developers a considerable degree of flexibility in their approach to working with evolving data models.[15]. Many documents are stored in a collection, which is analogous to a table in a relational environment. In the application under consideration, a collection has been created, named 'measures', for the storage of various measures. This collection is saved within a database named 'SENSORS'. The decision to employ MongoDB as the main mechanism for storing measures was guided by understanding that it offers superior scalability in comparison to other databases, and its flexibility makes it particularly well-suited to future development requirements.

Grafana

To enabled the visualisation of the charts of measures, the software Grafana has been employed. A multi-platform, open-source analytics, Grafana is an interactive visualisation web application capable of generating charts, graphs and alerts for the web when connected to suitable supported data sources. Although there is a Cloud version, this is not entirely free of charge; however, there is a self-hosted version that comes in two variants: Grafana Self-Hosted OSS[16] and Grafana Self-Hosted Enterprise, the former of which is free. The self-hosted OSS version was selected due to it being free of charge and its ability to generate an unlimited number of visualisations and charts. The deployment of Grafana within a Docker container facilitates connectivity to data sources – including MongoDB – without the necessity for additional coding.

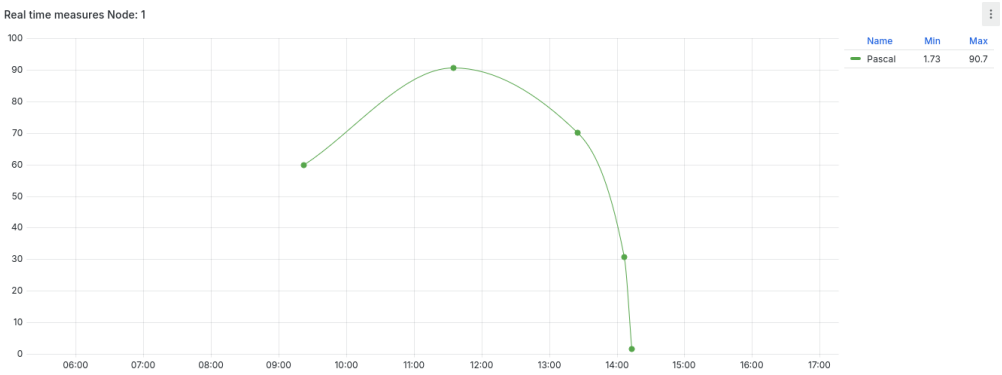

Figure 2.2: Grafana dashboard example

[15]Michael Dirolf Kristina Chodorow (2010), pp. 1–2

[16]Open Source Software

Grafana has many functions, from the creation of dashboards for the monitoring of sensors to the generation of time series charts and the incorporation of non-time series charts. The system's flexibility ensures seamless integration into a wide range of front-end applications. The process of adding a data source to Grafana necessitates the installation of a dedicated plugin. Despite there being a dedicated plugin that facilitates direct connectivity to MongoDB which allows for a seamless integration, it should be noted that such plugin is exclusively available within the enterprise licence of Grafana – which is notably expensive. Nevertheless, a complementary plugin exists that offers direct connectivity to the comparable but distinct document-based database InfluxDB named InfluxDB plugin.[17]

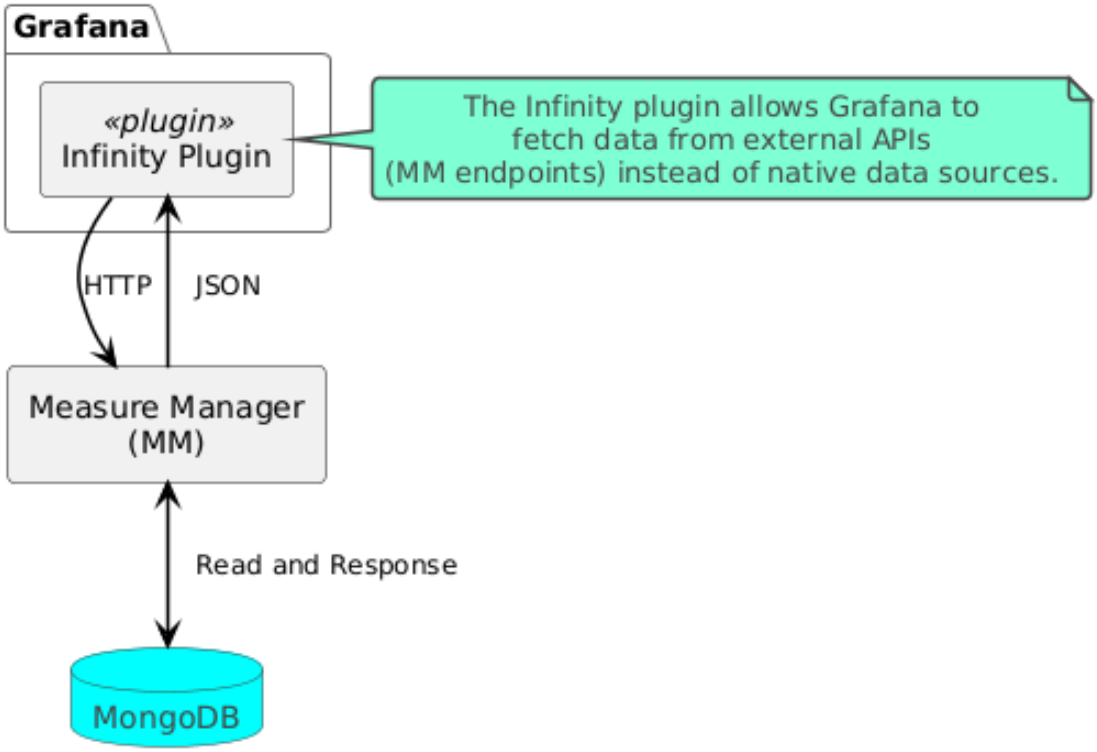

Figure 2.3: Data flow from Grafana to MongoDB through the MM using the Infinity plugin

[17]Obviously, an alternative approach would involve the utilization of InfluxDB instead of MongoDB.

Therefore, MongoDB has been connected to Grafana passing through the microservice MM by means of the plugin Infinity. The implementation of this process is somewhat challenging, as Infinity facilitates connection between Grafana and a general data source, thereby offering a structured method for data provision in many formats, such as JSON, XML or CSV.[18]. A safe connection was first established between the IAM(Identity Access Manager) – in this case Keycloak – and Infinity. In order to do so, a new client ID and the secret were added to the IAM and inserted into the OAuth2 Authentication part of the Infinity plugin configuration. Lastly, Infinity was connected to the /API/measures/nodeId endpoint. Two variables were used in the Grafana dashboard: nodeId and measureUnit. These were respectively employed to specify the node and the unit of measurement, whose information was retrieved using the UQL parser within the plugin.

Messages

The MM consumes events from two Kafka topics. The first one, named "measures", contains all the measurements sent by the sensors and represents the only entry point for inserting and creating measurement data within the application. The second topic, "node-event", carries information sent by the SM regarding nodes and the users who own them. This data is essential to ensure that any operation related to a node is correctly associated with a specific user ID, maintaining the integrity of user-linked operations in the system.

[18]For this purpose, inspiration was drawn from a guide Akshita Dixit (2024). Yet this guide only provides information on connecting to an unprotected Infinity plugin

2.3 API Gateway and IAM

Whilst designing a microservices architecture, I took the decision to introduce an API Gateway as the central point for managing and routing requests to the various back-end services. This component plays a pivotal role, acting as an intermediary between the client – whether a web front-end or a mobile app – and the back-end microservices. It functions as the sole entry point into the system, directing requests into their appropriate microservice. The API Gateway performs a mapping between URL endpoints and microservices.

The implementation of the API Gateway was achieved through the utilization of a Spring Cloud Gateway MVC dependency, a contemporary solution offered by Spring for the management of API traffic. Within this framework, a series of routes has been defined, which in turn map incoming client requests to their appropriate IPs of Docker containers that house the requisite services. In essence, each request undergoes analysis and is then sent dynamically to the designated microservice, ensuring seamless and transparent service delivery to the client. Implementing these functionalities entailed the creation of a Docker network spanning 172.20.0.0/24, as well as allocating IP addresses to each container. Spring Cloud Gateway, therefore, acts as the single access point for the back-end, meaning any request from a client must first go through it. This approach simplifies centralized control, security, and shared policy enforcement across all services.

The following example illustrates the configuration of a route to SM. It should be noted that the routes are added inside the application.yaml file. The URI indicates the destination of the endpoint whilst the TokenRelay is responsible for the re-transmission of the authentication token to the destination microservice.[19]

[19]The authentication token will be explained in short down below.

```yaml
spring:
  application:
    name: iam_module
  main:
    web-application-type: servlet

  cloud:
    gateway:
      mvc:
        http-client:
          type: AUTODETECT
        routes:
          - id: sensor-manager-nodes
            uri: http://172.20.0.10:8080
            predicates:
              - Path=/API/nodes/**
            filters:
              - TokenRelay
```

In order to enhance security and manage access control, the integration of Spring Cloud Security dependency was also implemented, with the result that an Identity and Access Management system(IAM) was able to be set up. The IAM is responsible for identifying, authenticating, and authorizing users interacting with the services, ensuring secure and controlled access to resources.

In the capacity of IAM identity provider, the decision was taken to utilize Keycloak, an open-source platform for identity and access management. Keycloak supports standard protocols such as OpenID Connect(OIDC) and SAML 2.0, making it highly compatible with contemporary applications. Make note that OIDC is based on the OAuth 2.0 protocol, which securely and efficiently manages user authentication – which is why the OIDC protocol has been implemented within Keycloak.[20]

[20]For more about OAuth 2.0 see IETF RFC 6749 and 6750

OIDC simplifies the way identification method of users based on the authentication performed by an Authorization Server and to obtain user profile information in an interoperable and REST-like manner. Thanks to OIDC I thus implemented secure sign-in flows for web, mobile, and JavaScript clients, so to receive verifiable identity information about users. The protocol is also extensible, supporting optional features such as encrypted identity data, OpenID Provider discovery and session logout, since OIDC offers a secure and trustworthy method to know the identity of the user that is intereacting with the app. Most importantly, OIDC eliminates the need for applications to manage passwords, reducing the risk of data breaches caused by stolen or mismanaged credentials.[21]

Authentication

During the Authentication process some actors take part to the action. These actors are:

- IAM, the entity that has implemented the OpenID Connect and OAuth 2.0 protocols; sometimes it also known as IDP(Identity Provider), security token service, OpenId Provider(OP) or Authorization Server.
- OAuth 2 Client, an application that outsources its user authentication function to an OP. It is also known as Relying Party(RP).
- User, the authenticating person.
- User Agent; it could be a mobile application or a browser, as it is in our case.

[21]For all information about OIDC, please consult OpenID Foundation (2025)

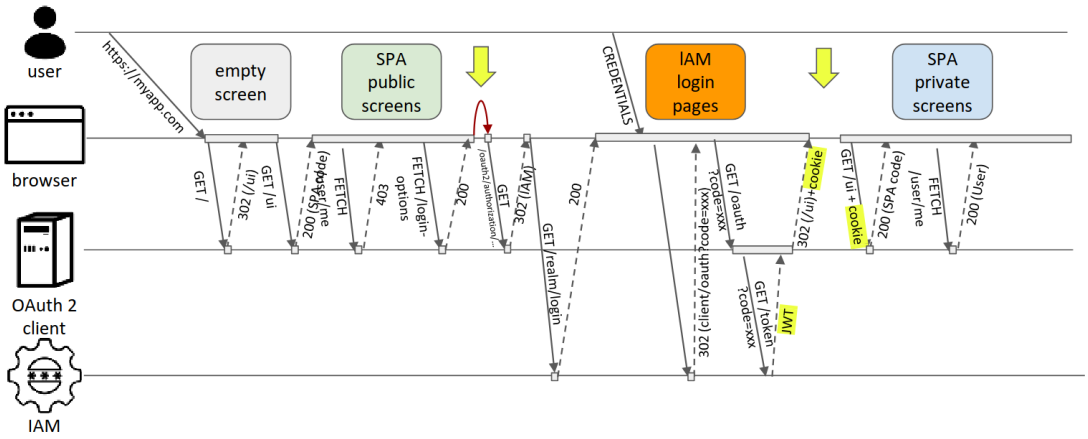

Figure 2.4: OpenID Fundation Authentication

The Authentication protocol observes the following steps:

1. The User clicks on sign-in ;
2. Then, the same is redirected to the login page of the IAM;
3. Afterwards, the User type username and password on the website;
4. After that, the IAM assigns an authentication code to the browser session;
5. Subsequently, the browser is redirected to the OAuth2 Client with the authentication code;
6. The Oauth2 Client exchanges the authentication code for the Identity Token;
7. Then, the Identity Token is associated to a cookie;
8. The cookie is sent to the browser;
9. Lastly, the browser is redirect with the cookie to a user private screen;

The outcome of the authentication process is the Identity Token, which contains an identifier of the user and other information. An ID token is encoded as a JSON Web Token(JWT), a standard format that allows your application to easily inspect its content and make sure it comes from the expected issuer and that no one else has changed it.[22] The JWT is composed of three fields: the header, the payload and the signature. Each field is separated by a dot and it is not encrypted but it is encoded in base64.[23] The following is an example of a JWT structure:

[Base64(HEADER)].[Base64(PAYLOAD)].[Base64(SIGNATURE)]

The header is composed of the algorithm employed to sign the JWT. The algorithm I used is HMAC with SHA-256 – in brief, HS256. The following is the pseudo-code representing the signature of the JWT:

HS256(base64UrlEncode(header) + "." + base64UrlEncode(payload), KEY)

Here, KEY is the IAM private key made out of 256 bit. The signature's authenticity is validated by IAM, which recalculates the signature. When the two signatures are found to be identical, the verification is succesfull.

It is best now to delve deeper into what HMAC actually is, and how is functions. HMAC is a function used by the message sender to produce a value formed by condensing the secret key and the message input. Such formed value is called MAC(message authentication code). The HMAC function used is based on the usage of a hash function:[24]

$$\mathrm{HMAC}_{\mathrm{SHA256}}(K, m) = H\left((K_0 \oplus \mathrm{opad}) \parallel H\left((K_0 \oplus \mathrm{ipad}) \parallel m\right)\right)$$

In this formula:

- H is the generic hash function, in our case it is SHA-256. The hash function splits the message in blocks of size B;

[22] For the definition of the Identity Token see Auth0 (n.d.)

[23] This encoding does not provide any kind of additional security and the content can be read with a simple base64 conversion.

[24] A specific hash function is not provided in the definition of the HMAC algorithm because any hash function can be used. For the definition of HMAC, cfr. National Institute of Standards and Technology (2008)

- K is the original secret key;
- B is the block size in bytes;
- K_0 is the key after either padding has been added or after undergoing truncation, whose length is in B-sized bytes ($B = 64$ bytes for SHA-256);
- m is the message to authenticate;
- opad is the byte 0x5C repeated B times;
- ipad is the byte 0x36 repeated B times;
- $\oplus$ denotes bitwise XOR;
- $\|$ denotes concatenation.

Step	**Description of HMAC Algorithm**
1	If the length of $K = B$: set $K_0 = K$. Proceed to Step 4.
2	If the length of $K > B$: compute $H(K)$ to obtain an L-byte string, then pad it with $(B - L)$ zero bytes to form K_0 (i.e., $K_0 = H(K)\|00\ldots00$). Proceed to Step 4.
3	If the length of $K < B$: pad K with zeros to form a B-byte string K_0 (e.g., if K is 20 bytes and $B = 64$, pad with 44 bytes of 0x00).
4	XOR K_0 with `ipad` (0x36 repeated B times): $K_0 \oplus$ ipad.
5	Append the message `text` to the result of Step 4: $(K_0 \oplus$ipad$)\|$text.
6	Apply the hash function H to the result of Step 5: $H((K_0 \oplus$ ipad$)\|$text).
7	XOR K_0 with `opad` (0x5C repeated B times): $K_0 \oplus$ opad.
8	Append the result of Step 6 to the result of Step 7: $(K_0 \oplus$ opad$)\|H((K_0 \oplus$ ipad$)\|$text).
9	Apply the hash function H to the result of Step 8: $H((K_0 \oplus$ opad$)\|H((K_0 \oplus$ ipad$)\|$text$))$.
10	Take the leftmost t bytes of the result from Step 9 as the final MAC.

The payload of the Identity Token is the core of the logged user information, which contains:

- iss, from the word issuer, a case-sensitive string or URI that uniquely identifies the party that issued the JWT. Its interpretation is application specific (there is no central authority managing issuers).
- sub, from the word subject, a case-sensitive string or URI that uniquely identifies the party that this JWT carries information about. In other words, the claims contained in this JWT are statements about this party. The JWT spec specifies that this claim must be unique in the context of the issuer or, in cases where that is not possible, globally unique. Handling of this claim is application specific. This field is used by the SM to create an User entity in the DB.
- aud, from the word audience. This is either a single case-sensitive string or URI or an array of such values that uniquely identifies the intended recipients of this JWT. When this claim is present, the party reading the data in this JWT must confront its name with the aud claim or disregard the data contained in the JWT.
- exp, from the word expiration (time), a number representing a specific date and time in the format "seconds since epoch", as defined by POSIX6. This claim sets the exact moment from which this JWT is considered invalid. Some implementations may allow for a certain skew between clocks – by considering this JWT to be valid for a few minutes after the expiration date.
- iat, from issued at (time). Such number represents the specific date and time (in the same format as exp) at which this JWT was issued.

Although the Identity Token constitutes the heart of the authentication protocol, this is not stored in the browser; rather, the OAuth2 Client associates a cookie with each Identity Token and transmits said cookie to the browser, which then stores it. Upon acquiring a specific website's cookie, the browser incorporates it into the header of all the subsequent requests directed to that website. In this way, the user session is maintained. Information of various types may be contained within a cookie, including: JSESSIONID; OAuthSTATE; XSRF-TOKEN, etc...

There is a link between the JSESSIONID and the Identity Token in the OAuth 2 client. If an individual were to steal the JSESSIONID, they could impersonate the user, which is why the JSESSIONID is a large number used to prevent a

brute force attack, generated using a random number generator that cannot be predicted. The OAuthSTATE and XSRF-TOKEN are used to prevent the Cross-Site Request Forgery(CSRF) attack, an attack that forces the end user to execute unwanted actions onto a web application they're currently authenticated in. With the aid of social engineering – such as sending a malicious link through email or chat – an attacker may deceive web application users into performing actions on the attacker's behalf without them knowing it. If the targeted user has a regular account, a successful CSRF attack can trigger unintended state-changing operations, such as transferring money or updating account settings. However, if the victim holds administrative privileges, such an attack could lead to a complete compromise of the web application.[25] The CSRF attack can be prevented through the implementation of OAuthState and XSRF-TOKEN. These tokens are generated at the beginning of the user session and then stored within the cookie. It is imperative that these tokens be incorporated into all operations that result in a state-change on the back-end. Afterwards, the back-end performs a verification process in which it asserts whether the tokens added in the header of the operation correspond to the tokens stored in the cookie. However, in the event that an individual were to surreptitiously appropriate the cookie, they would possess the capability to impersonate the logged-in user. Consequently, the vulnerability to this specific attack persists.[26]

Regarding the authorization – a similar but different operation from authentication – the IAM can also send an Access Token that is very simliar to the Identity Token and commonly provided in a JWT format. Nonetheless, it also has the scope field where is found the list of actions the user has the authorization to perform.[27]

[25]See KirstenS et al. (n.d.)

[26]For this kind of vulnerability, cfr. OWASP Foundation (n.d.)

[27]For a discussion on Access tokens, I recommend Sebastian E. Peyrott (2024)

In order to support OIDC, the authentication and authorization code flow, it was implemented Keycloak – an open source IAM– by creating a Keycloak realm named MeasureStreams, which represents the application's security domain. Within this realm, it was configured a client with support for the Authorization Code Flow, and then copied the corresponding client-id and client-secret into my Spring Cloud Gateway configuration to establish the connection between the IAM and the Gateway.

Finally, it was defined several user roles in both Keycloak and Spring Cloud Gateway to differentiate access levels and permissions based on user type. Specifically, it was designed three types of end users:

- APP_Admin, who have full access to all features, including advanced statistics and server performance analytics;
- Customers, who can only view measurement data from their own LoRa nodes installed within their facilities, as well as the status of their DCCs;

This architecture results in a system that is scalable, secure, and flexible, with centralized and consistent management of API routing, authentication, and authorization.

DCC

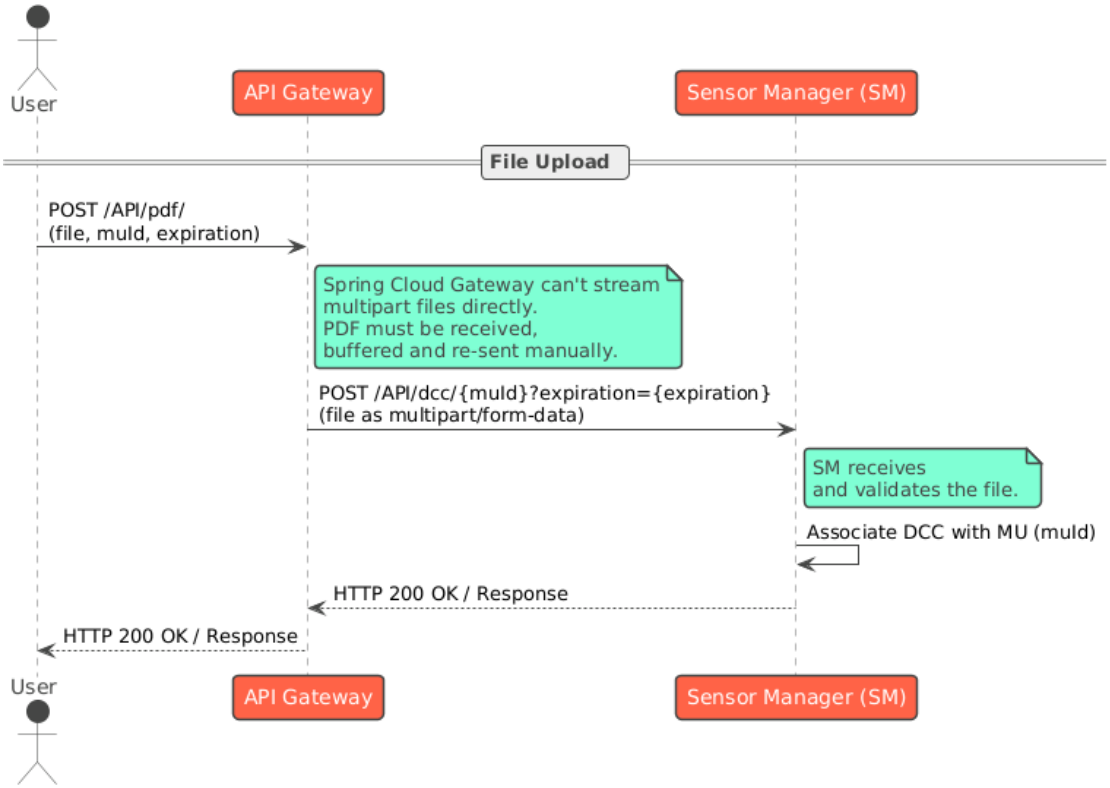

Figure 2.5: Data flow of DCC

The API Gateway is responsible not only for ensuring security and managing APIs within the application but also plays a vital role in supporting the Sensor Manager (SM). In particular, the API Gateway manages the uploading of PDFs—which represent the Digital Calibration Certificates (DCC) in this context—and subsequently forwards them to the SM, which associates each certificate with the corresponding Measurement Unit (MU). This design is necessary because, in Spring Cloud Gateway, it is not possible to directly forward a multipart file upload (such as a PDF) to a downstream API without first storing or buffering the file. This limitation arises from the gateway's reactive, non-blocking architecture, which streams requests and responses. Without buffering, the multipart data stream cannot be reliably forwarded downstream.

To address this, an HTTP POST method at /API/pdf/, accepting the following parameters:

- authorizedClient: An OAuth2 authorized client instance, automatically injected to support secure token relay.
- file: The PDF file to be uploaded, received as a multipart HTTP request parameter.
- muId: The identifier of the Measurement Unit to which the uploaded certificate should be linked.
- expiration: The expiration date of the certificate.

Internally, the method constructs a URL targeting the DCC microservice, embedding the muId and expiration as query parameters. It prepares HTTP headers specifying the content type as multipart/form-data and attaches the OAuth2 bearer token to ensure authenticated communication. The file is wrapped into a suitable resource (MultipartInputStreamFileResource) to be sent as part of the request body. A new HTTP entity is created encapsulating the body and headers, and a POST request is issued to the DCC microservice using a RestTemplate. Finally, the method returns the HTTP response status and body received from the microservice, thereby enabling the API Gateway to act as a secure proxy for file uploads, delegating storage and association responsibilities to the Sensor Manager via the DCC service.

2.4 Settings Manager

The Settings Manager constitutes the core component responsible for managing connectivity between the server and the underlying hardware gateways. Its primary role is to dispatch commands to the hardware infrastructure, particularly to Control Units (CUs) and Measurement Units (MUs). The architecture of the Settings Manager adheres to the principles of Clean Code, mirroring the design of other components such as the Sensor Manager (SM) and the Measure Manager (MM). Accordingly, it is structured into three main layers: controllers, services, and repositories. Communication with hardware gateways is established through MQTT messaging, utilizing a Mosquitto broker instance deployed within the system.

The Settings Manager is responsible for handling the following core entities:

- **CuSetting:** represents the configuration parameters for a Control Unit, such as bandwidth, coding rate, spreading factor, and update interval. Each CU configuration is associated with a User, and optionally with a Gateway, establishing a many-to-one relationship.
- **MuSetting:** encapsulates the settings associated with a Measurement Unit, primarily characterized by its sampling frequency. Each MU is associated with a User, and optionally linked to a CuSetting, reflecting its logical connection to a control unit.
- **Gateway:** models the physical device responsible for interfacing the CUs with the serverand maintains a one-to-many relationship with the associated CuSetting entities. This enables centralized management of all CU configurations connected to a specific hardware gateway.

Additionally, a User entity is maintained for security and authorization purposes, although it is not directly involved in configuration management.

Rest Controllers

The Setting Manager serves different endpoints to retrive and update data and they are:

1. GET */API/command/start/{muid}* initiates the acquisition process on the MU identified by the specified `{muid}` .[28]

2. GET */API/command/stop/{muid}* stops the acquisition process on the MU identified by the specified `{muid}`.

3. GET POST PUT */API/cu-setting/* these HTTP methods are used to manage CuSetting entities. The POST method allows the creation of a new CuSetting, the PUT method enables updating an existing configuration, and the GET method retrieves all available CuSetting instances from the system.

4. GET */API/cu-setting/{cusettingid}* retrives a specific CuSetting identified by the specified `{cusettingid}`.[29]

5. DELETE */API/cu-setting/{cusettingid}* deletes a specific CuSetting.

6. GET */API/cu-setting/{cusettingid}/isalive* this endpoint is used to verify the availability of a specific CU identified by `cusettingid`. If the CU is active, the response contains the `networkId` of the associated Gateway; otherwise, it returns `null`.

7. POST */API/cu-setting/arealive* is used to verify the availability of multiple CUs. The request body must contain a list of CU identifiers. For each CU, the response returns the `networkId` of the associated Gateway if the unit is reachable; otherwise, it returns `null`.

8. GET POST PUT */API/mu-setting/*These HTTP methods are used to manage MuSetting entities. The POST method allows the creation of a new MuSetting, the PUT method enables updating an existing configuration, and the GET method retrieves all available MuSetting instances from the system.

[28]The term "muid" refers to the unique network identifier networkId assigned to a MU. It is of type Long, as defined in the MU entity structure cfr. Sensor Manager section for further details

[29]The term "cusettingid" refers to the unique network identifier networkId assigned to a CU. It is of type Long, as defined in the MU entity structure cfr. Sensor Manager section for further details

9. GET */API/mu-setting/{musettingid}* retrives a specific MuSetting identified by the specified `{musettingid}`[30]

10. DELETE */API/mu-setting/{musettingid}* it is used to delete a specified MuSetting.

Database

In order to persist and manage all relevant information, a PostgreSQL database named "SETTINGS" is used as the underlying data storage system. Although the data maintained by the Settings Manager are conceptually related to entities managed by the Sensor Manager (SM), a deliberate architectural decision was made to separate them. This separation arises from the responsibility of the Settings Manager to communicate directly with hardware gateways. As such, it was deemed appropriate to decouple the two functionalities—sensor event management and hardware configuration.

However, for correct operation, the Settings Manager must remain informed about lifecycle events occurring in the Sensor Manager. This is because a CuSetting is logically associated with a specific instance of a Control Unit (CU), and similarly, a MuSetting is linked to a Measurement Unit (MU). Therefore, the lifecycle of these settings must be consistent with the lifecycle of their corresponding hardware units.

Two primary architectural solutions were considered to achieve this synchronization. The first involved using a shared database between the Sensor Manager and the Settings Manager, enabling the latter to validate the existence of each CU or MU before performing operations on their respective settings. However, this approach was ultimately rejected due to the strong coupling it introduces, violating the principle of separation of concerns and reducing the flexibility and scalability of the system.

Instead, the chosen solution was to implement event-based communication using Kafka topics. This approach allows the Sensor Manager to publish events describing state changes or lifecycle operations, which are then consumed by the

[30]The term "musettingid" refers to the unique network identifier networkId assigned to a MU. It is of type Long, as defined in the MU entity structure cfr. Sensor Manager section for further details

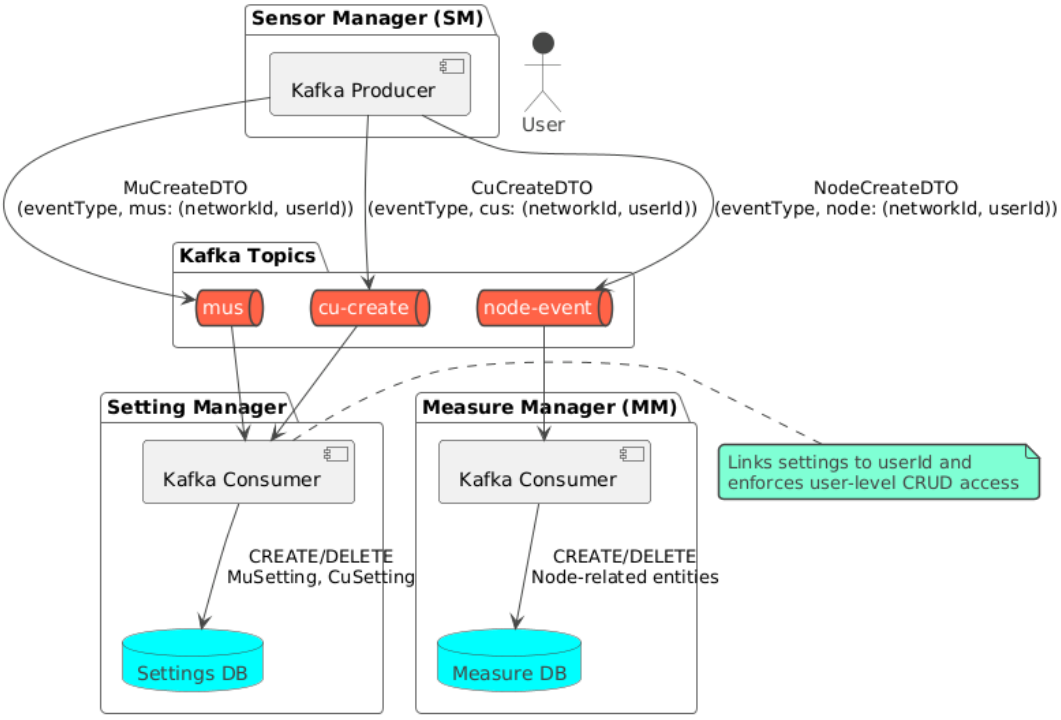

Figure 2.6: UML schema of Kafka internal architecture

Settings Manager. Through this mechanism, the two services remain fully decoupled at the database level, allowing independent deployment—even on separate physical machines—while still maintaining logical consistency across the system. In particular there are 3 kafka topics:

- mus
- cu-create
- node-event

A single producer, referred to as the Setting Manager (SM), is responsible for publishing events to these topics. Each event is composed of two fields: an eventType, which is a string indicating the operation type and can be either "CREATE" or "DELETE", and a value, which varies depending on the topic. In particular, the mus topic carries a value of type MuCreateDTO, which includes both a networkId and a userId. The node-event topic transmits a NodeCreateDTO, while the cu-create topic uses a CuCreateDTO, whose structure is analogous to that of the MuCreateDTO.

There are two categories of consumers deployed within the Setting Manager and within the Link component. These consumers are responsible for linking the lifecycle of the MU and CU entities managed by the SM with their respective counterparts, MuSetting and CuSetting, in the Setting Manager. When a consumer receives an event with eventType equal to "CREATE", it persists a corresponding entity in the database. Conversely, when the eventType is "DELETE", the consumer deletes the associated entity from the database.

In addition to enabling lifecycle synchronization across services, the Kafka topics also support user-specific access control by associating entities in the Setting Manager with defined users. This mechanism ensures that CRUD operations are restricted to the users to whom the entities belong, thereby reinforcing access control and data integrity.

Security

Access control within the Setting Manager is enforced using Spring Security. Each incoming request is authenticated via a JWT (JSON Web Token), which contains claims such as the userId and application-specific roles. The service validates access by comparing the userId extracted from the token with the userId associated with the target resource. These ownership associations—linking users to specific CuSetting and MuSetting entities—are established dynamically by the Kafka consumers, as described earlier. Specifically, upon receiving "CREATE" events from the relevant Kafka topics, the consumers persist the configuration entities along with the corresponding userId, which is extracted from the event payload. If the userId from the JWT matches the one stored with the resource, the requested operation is authorized. Alternatively, if the token includes the Realm Role App_Admin, indicating administrative privileges, the service allows all operations regardless of ownership. This hybrid approach combines event-driven ownership registration with role-based access control, ensuring that configuration changes can only be performed by authorized users or administrators.

Messages

To enable the transmission of commands to the gateway and to configure parameters on the CU and MU, a service based on the MQTT protocol has been implemented. For the underlying communication infrastructure, the open-source MQTT broker Mosquitto was adopted, providing a lightweight and reliable messaging solution suitable for constrained environments.

MQTT (Message Queuing Telemetry Transport) is a lightweight, publish/subscribe messaging protocol specifically designed for resource-constrained devices and low-bandwidth, high-latency, or unreliable networks. Its decoupled architecture—based on the concepts of publishers, subscribers, and a central broker—makes it particularly suitable for scenarios involving distributed sensing and control, such as in IoT and sensor networks.

To establish the MQTT-based communication infrastructure, the Eclipse Mosquitto broker was adopted. Mosquitto is an open-source MQTT message broker that implements version 5.0 of the MQTT protocol. It is well-regarded for its small footprint, ease of deployment, and support for standard authentication and authorization mechanisms. In this architecture, Mosquitto acts as the central messaging hub, receiving messages from publishers (e.g., sensor devices or service controllers) and distributing them to subscribers (e.g., configuration services or gateways) based on topic-based filtering.

This MQTT-based communication layer enables efficient and scalable interaction among services, facilitating both the dissemination of configuration commands and the coordination of distributed components in the networked sensing system. It is important to highlight that MQTT and Kafka serve different purposes and offer complementary characteristics. Kafka is a distributed event streaming platform designed for high-throughput, persistent, and fault-tolerant data pipelines. It is particularly suited for handling large volumes of data across microservices and for guaranteeing message durability and ordering. Conversely, MQTT is optimized for real-time, low-latency message delivery in constrained environments, with limited support for persistent storage and historical replay. In this architecture, Kafka is employed to manage the lifecycle synchronization and event distribution between services, while MQTT is used to handle real-time control and configuration tasks, ensuring a clear separation between event logging and command dissemination functionalities.

MQTT configuration

The MQTT communication infrastructure was organized into six distinct topics, categorized into two groups: downlink and uplink topics.

The downlink topics comprise the following:

- downlink/gateway
- downlink/cu
- downlink/mu

These topics contain all messages transmitted by the server and directed towards the corresponding recipients. Specifically, messages published on the downlink/-gateway topic are intended for the gateway, whereas messages published on downlink/cu and downlink/mu are directed to the Control Units (CU) and Measurement Units (MU), respectively.

The uplink topics include:

- uplink/gateway
- uplink/cu
- uplink/mu

These topics are designated for messages received from the gateway. Messages originating from a CU device are published to the uplink/cu topic, and similarly, messages originating from an MU device are published to the uplink/mu topic. The uplink/gateway topic carries messages sent by the gateway itself.

This topic organization enables a clear and structured bidirectional communication flow between the server and network devices, facilitating effective message routing and processing based on the message origin and destination.

Each gateway periodically transmits a message—once every minute—on the uplink/gateway MQTT topic. The message is represented by the following data structure:

```
data class GatewayDTO(val id: Long, val cus: Set<Long>)
```

Listing 2.1: Structure of the GatewayDTO used for receive the online gateways and the corresponding CUs

This message includes the identifier of the gateway (id) and the set of identifiers corresponding to the Control Units (CUs) that the gateway is currently able to reach. By collecting and analyzing these periodic updates, the system is able to dynamically determine the communication paths required to reach a specific CU or, indirectly, a Measurement Unit (MU) associated with it.

In contrast to the periodic messages published on the uplink/gateway topic, messages on other topics are structured according to a unified command format, represented by the following data structure:

```
data class CommandDTO(
    val commandId: Long,
    val gateway: Long,
    val cu: Long,
    val mu: Long,
    val type: String,
    val cuSettingDTO: CuSettingDTO?,
    val muSettingDTO: MuSettingDTO?
)
```

Listing 2.2: Structure of the CommandDTO used for sending configuration commands

Transmission Flow Commands to Downlink

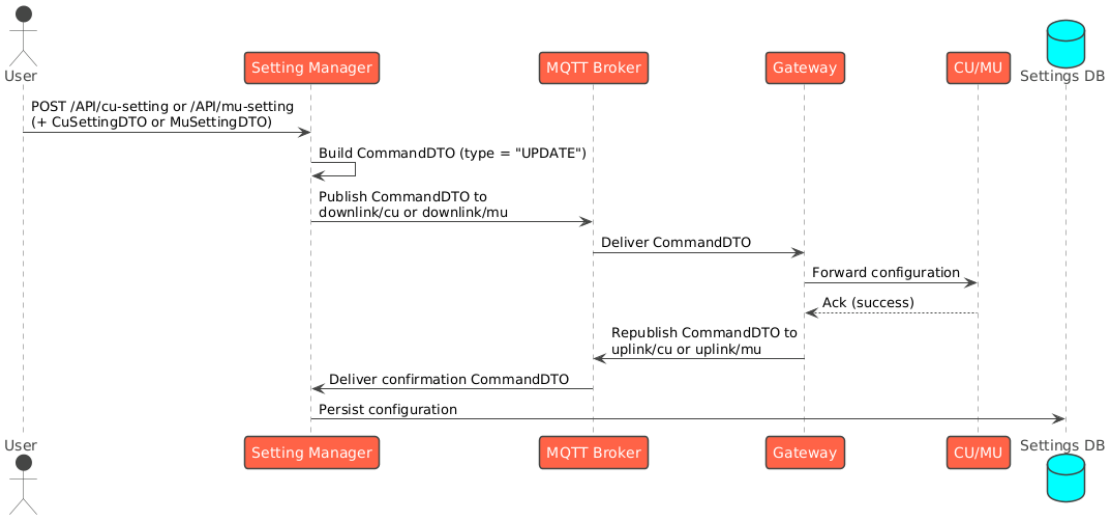

Figure 2.7: UML schema of an example of update delivered by MQTT

When a user initiates a configuration request—either for a Control Unit (CU) or a Measurement Unit (MU)—by sending a POST request to the corresponding REST endpoint, the Setting Manager does not immediately persist the configuration in the database. Instead, it constructs a CommandDTO, populating the cuSettingDTO or muSettingDTO fields as appropriate, and sets the type field to "UPDATE". This command is then published on the MQTT topics downlink/cu or downlink/mu, depending on the target unit.

Upon receiving the command, the gateway forwards the specified configuration to the designated CU or MU. If the update operation succeeds, the gateway republishes the same CommandDTO message to the corresponding uplink topic (i.e., uplink/cu or uplink/mu). Once the Setting Manager receives this confirmation message through the uplink channel, it finalizes the process by persisting the received configuration in the database.

Chapter 3

Front-end

3.1 SPA: Single Page Application

The back-end constitutes the core of the application; however, as previously delineated in the introduction, the development and architecture of MeasureStream are layered. On top of these layers is the front-end. The front end is the part of a website that is designed to an User Interface(UI) to the user. There are two primary methods for developing a UI. The first method is the traditional approach, which involves a multi-page application. This approach has the potential to compromise the user experience by introducing a constant delay upon each modification to the user interface. This is due to the loading of a new page, regardless of the extent of the change.This methodology for developing front-end applications stands in opposition to the notion of a Single-Page Application (SPA). A Single-Page Application(SPA) is a web application that loads a single web document and then updates its body content using JavaScript APIs, such as Fetch, whenever different content needs to be displayed.[1]

Speed is a primary benefit of SPAs when compared to traditional web pages. The reason for this is that SPAs are fully loaded at the outset. Thereafter, only the content of the page is updated, not the entirety of the application. Furthermore, all logic is handled on the client-side. This means that the server does not need to render a new page for each request. The aforementioned process has been

[1]MDN Web Docs (2025)

demonstrated to reduce the number of requests and responses sent to the server, thereby improving performance. Furthermore, SPAs have the capacity to cache data (e.g., via service workers or client-side storage), thus enabling the application to load more expeditiously upon a user's subsequent visit or navigation to a specific section.

The MeasuresStream user interface (UI) has been developed using the single-page application (SPA) approach, as previously outlined. The most effective approach to developing a SPA is to utilize a JavaScript framework. For this reason, the React framework has been selected. React is not the sole option for developing a SPA. Other frameworks, such as Angular, Svelte, and Vue, are equally valuable. However, the decision to utilize React was influenced by its widespread use and its capacity for rapid development and response.

3.2 React and Typescript

React is a widely-used JavaScript library designed for building UIs. It was developed by Facebook to solve the challenges posed by large, data-intensive websites. First released in 2013, React has since become a go-to tool for creating dynamic web applications. In React, UI is built from small units like buttons, text, and images. React lets you combine them into reusable, nestable react components. From web sites to phone apps, everything on the screen can be broken down into components. in detail, a React component is a JavaScript function that you can sprinkle with markup. Components can be as small as a button, or as large as an entire page. React components use a syntax extension called JSX to represent that markup. JSX looks a lot like HTML, but it is a bit stricter and can display dynamic information.[2]

It must be noted that a SPA does not request a new page each time there is a change in the UI. The following question must therefore be posed: how is this possible? In order to answer this question, the concept of the Document Object Model (DOM) must first be introduced. The DOM is an interface that browsers provide for web pages to interact with HTML or XML documents. It's essentially a tree structure that represents the content of a webpage, where each part of the page (elements like paragraphs, headings, images, and links) is a node in the tree. When a browser loads a webpage, it parses the HTML, turning it into a

[2]React Team (2025)

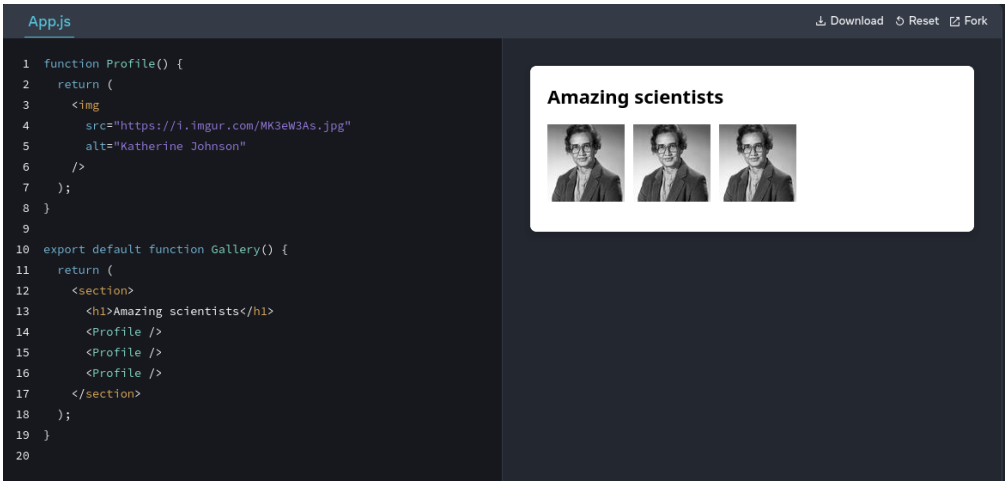

Figure 3.1: Example of React component

hierarchical structure (the DOM). Each HTML element is a node in the tree. The DOM is crucial because it enables developers to programmatically interact with a webpage. Without the DOM, there would be no way to modify the content or structure of a webpage after it has been loaded. It is possible for a developer to utilize solely JavaScript for the purpose of modifying the DOM. However, the employment of React has been demonstrated to expedite the development process. React permits developers to modify the DOM without the need for manual access, furthermore React uses a virtual DOM to efficiently manage updates to the real DOM, it collects all the updates to the virtual DOM and than it updates the DOM accordingly. Instead of directly modifying the real DOM every time a change is made, React updates the virtual DOM first then React compares the current state of the virtual DOM to a previous version using an algorithm called Reconciliation and react updates only the necessary parts of the real DOM by applying the changes that it finds between the virtual DOM and the real DOM. This is more efficient because React minimizes the number of direct interactions with the real DOM, which can be slow and expensive.[3]

JavaScript is not the only language available to develop SPA using React, also TypeScript(TS) is a good choice, and for some project it is a better choice than JavaScript. TS is a high-level open-source programming language that adds static typing with optional type annotations to JavaScript. It is designed for the development of large applications. JavaScript is a versatile programming language, it

[3]MDN Web Docs (2025)

does not detect errors at compile time, it attempts to manage data, even if it is erroneous. This quality is advantageous when developing conceptual applications or when errors are infrequent, indeed JS does not provide a mechanism to verify whether the code is accessing properties that are not present or accessing variables that have not been declared.

```
1    const obj = { width: 10, height: 15 };
2    // Why is this NaN? Spelling is hard! height -> heigth
3    const area = obj.width * obj.heigth;
```

Misspelling errors like this above are very dangerous, in a large JS project, because they are silent and the code still works, the app continues to work, and they are difficult to identify. To improve upon JS, TS was created. TS introduces a static type checker, which analyzes the code at compile time to detect errors. For this reason, TS provides more robust and resilient code compared to JS. For this reason TS was used to develop the SPA.

In a SPA, it's important to define how to implement and manage the back button and page navigation, because even though there is technically only one page, users still expect a multi-page experience. A common solution to this challenge is to use a library that handles both navigation and back button functionality, such as React Router DOM. React Router DOM is a standard library for routing in React applications. It enables developers to define multiple routes, each associated with a specific component, and it updates the browser's URL without reloading the page. This way, navigation feels seamless and natural to the user. It also manages the browser history, allowing features like the back and forward buttons to work correctly. With tools like BrowserRouter, Routes, and Link, React Router DOM simplifies the process of building complex, multi-view applications while maintaining the fast and smooth experience typical of SPAs.

```
<Router basename={"/ui"}>
    <MyNavbar   me={me} />
    <Container fluid>
        <Routes>
            <Route path="/" element={ me.name?   <Nodes nodes={ nodes}/> : <LandingPageENG/> } />
            <Route path="/add" element={<AddNode/> } />
            <Route path="/nodes/:nodeId" element={  <NodeInfoPage  nodes = { nodes} />  } />
            {/* Add more routes here as needed */}
        </Routes>
    </Container>
</Router>
```

Figure 3.2: Example of React Router Routes

When there is information that needs to be shared across multiple components in a React application, integrating React Context can be very helpful. It provides a convenient way to share data across components without having to pass props explicitly through each level of the component tree. React Context is particularly useful for sharing information about the logged-in user, allowing you to provide different visualizations or content based on whether a user is logged in or not. To manage this user-related information, it has been introduced the MeInterface, which contains the following properties, saved inside the application context:

- name: The name of the logged-in user.
- loginUrl: The URL used for logging in the user.
- principal: The main user object, which can be any type or null if no user is logged in.
- xsrfToken: The XSRF token used for securing requests.
- logoutUrl: The URL used for logging out the user.

This interface helps to structure the data related to the logged-in user, ensuring that the information is easily accessible throughout the application.

3.3 UI Implementation

Now it will be provided some example of the implementation of the UI. When a user visits the Base URL of MeasureStream ("/"), it is redirected to "/ui". If the user is not logged in, it will be shown a general landing page that explains the core functionalities of MeasureStream.[4] For the design of this landing page, it has been utilized Bootstrap, a popular CSS framework that simplifies web development by providing pre-designed components, a responsive grid system, and utilities to help create visually appealing and consistent layouts. Bootstrap ensures the page adapts smoothly to different screen sizes, making it responsive across all devices.

On the landing page, it has been implemented several information cards that provide quick insights into the various features and functionalities available in

[4]for more about how redirection are implemented see the Chapter 1 section API Gateway and IAM

MeasureStream. Additionally, it has been included a hamburger button for better navigation on mobile devices. The hamburger menu contains a Login button, which, when clicked, redirects the user to the Keycloak login page for authentication.

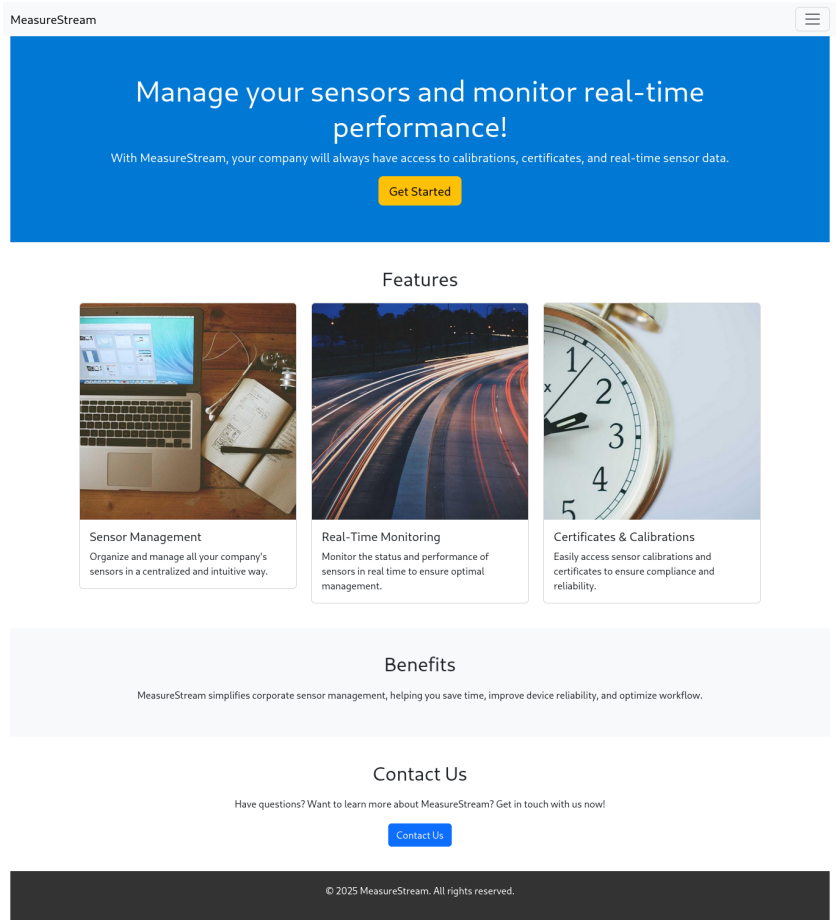

Figure 3.3: MeasureStream landing page

After the user clicks the login button, it will be redirected to the Keycloak login page, where it will authenticate himself. it has only been created a single user in Keycloak for testing purposes. [5]

[5]for more about the login process see the Chapter 1 section API Gateway and IAM

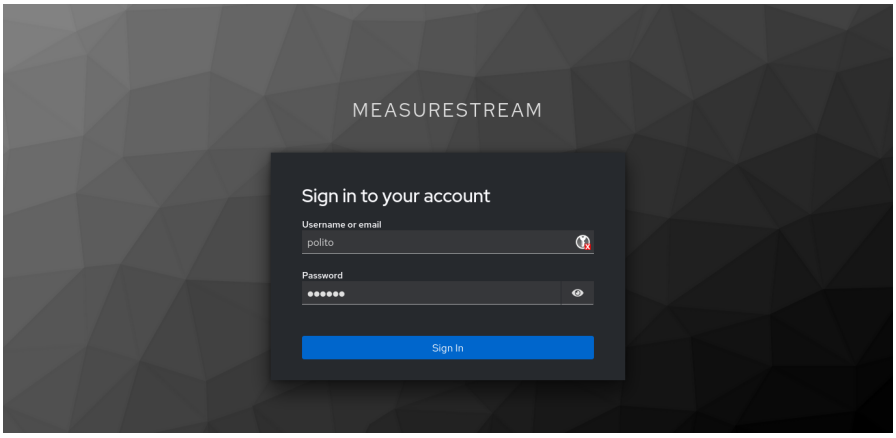

Figure 3.4: Keycloak Login page

After a successful login, the user will be redirected to the main page for logged-in users. On this page, it can view a map powered by the React Leaflet library, which allows for seamless integration of interactive maps into React applications. The map displays blue markers to indicate the positions of all nodes, while a red marker highlights the position of a selected node. When the user clicks on a marker, it turns red, and additional information about the node is shown. On the left side, there is a list of all nodes. By clicking the info icon next to a node, the user can navigate to a page that provides detailed information about the selected node. The React Leaflet library makes the map interactive and responsive, enhancing the user experience with smooth navigation and marker interactions. Additionally, even when the user is logged in, the hamburger menu includes a logout button. By clicking this button, the user can log out and will be redirected to the landing page, ensuring a smooth session management process.

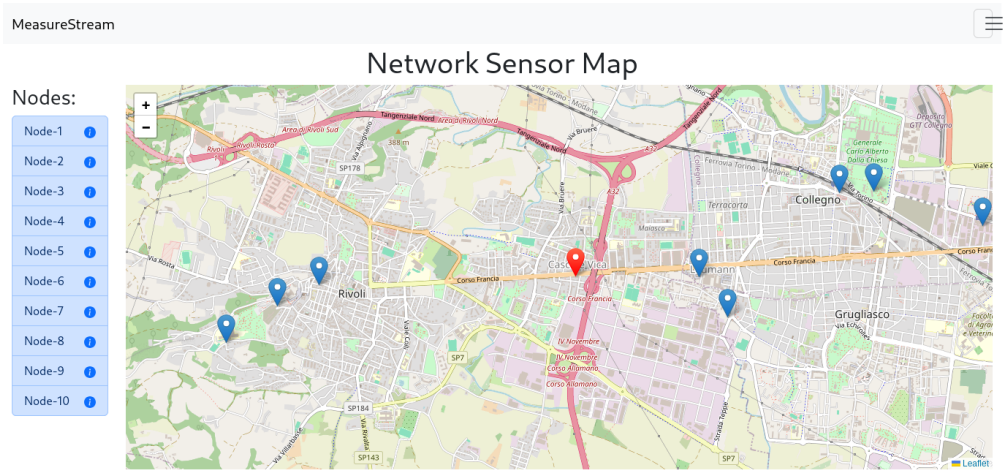

Figure 3.5: MeasureStream logged-in page

The node information page presents comprehensive details about the node, including its constituent Control Units (CUs) and Measurement Units (MUs). It also provides real-time data on the operational status of the CUs, such as remaining battery life and the Received Signal Strength Indicator (RSSI), which reflects the signal power. Additionally, this page enables management of all Device Configuration Containers (DCCs) associated with the MUs present in the node, allowing users to download, upload, or delete these configurations. The status of each component is visually indicated by a green light if the component is online and available, or a red circle if it is offline. Furthermore, expanding the accordion sections for each CU or MU permits the execution of start and stop commands, as well as the adjustment of specific settings managed via the Settings Manager[6].

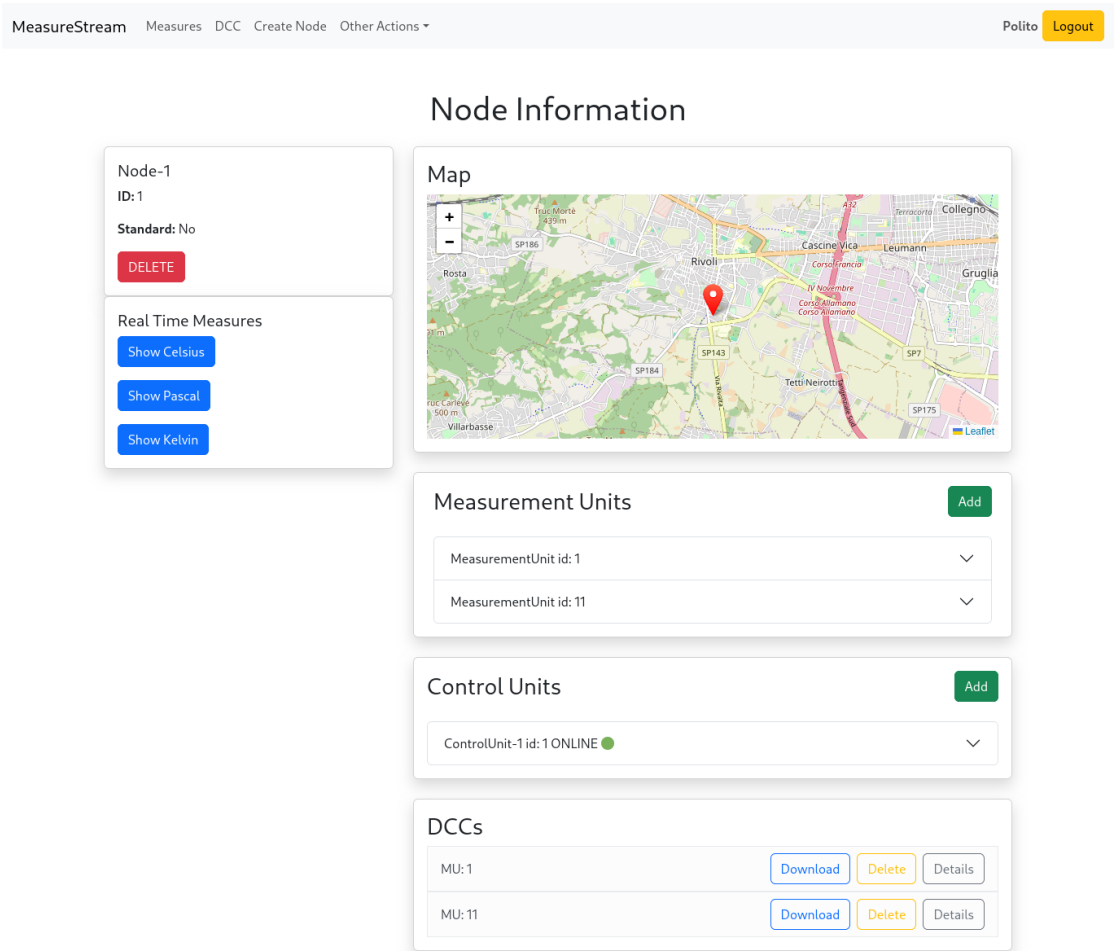

Figure 3.6: Node information page

[6]For comprehensive details on the settings, see Section 44.

At the top of the page, there's a small card that contains a map showing the current location of the node. On the left side, buttons to delete the node or modify its fields are present. Additionally, there is a list of buttons that allow users to view charts of sensor data, powered by Grafana. These charts display real-time values of the sensors, providing valuable insights into the node's performance.

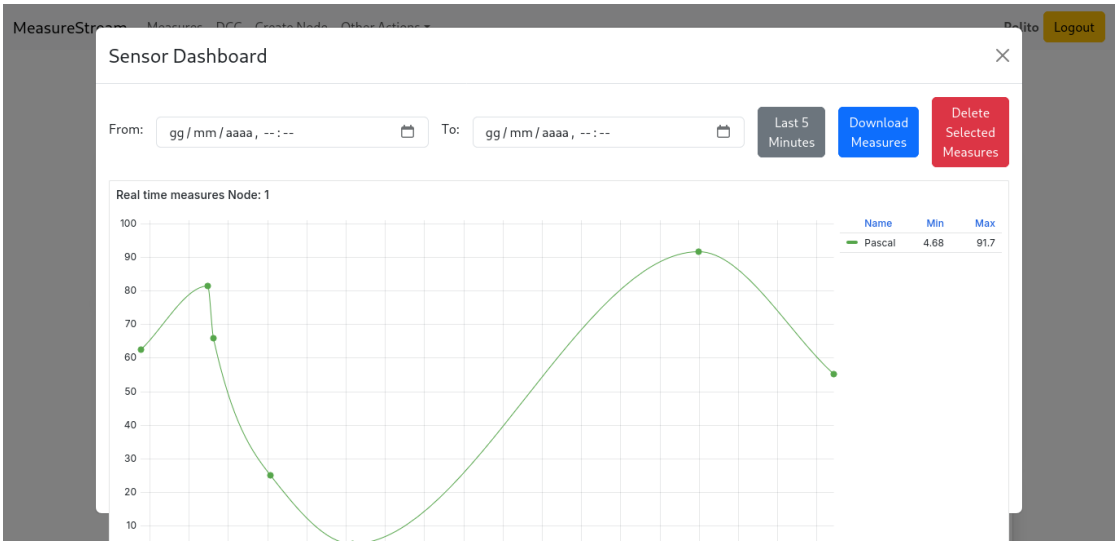

Figure 3.7: Graphic showing real-time measurements.

Grafana has been integrated into the interface using an embedded iframe, a standard HTML element that allows the display of external web content within a web page. In this case, the iframe loads specific Grafana dashboards directly into the application's UI, enabling users to view real-time sensor charts without leaving the platform. To facilitate this, the front-end communicates with Grafana directly, bypassing the API Gateway. This approach was chosen for its simplicity, as it avoids the need to proxy or reconfigure Grafana behind the gateway. However, it also means that not all application traffic is routed through the API Gateway, which may reduce the consistency of centralized monitoring and access control.

```
grafana:
    image: grafana/grafana-enterprise
    restart: always
    volumes:
      - grafana_data:/var/lib/grafana
      - /var/lib/grafana/plugins
    ports:
      - 3000:3000
    environment:
      - GF_SERVER_ROOT_URL=https://grafana.christiandellisanti.uk
      - GF_SERVER_DOMAIN=grafana.christiandellisanti.uk
      - GF_SECURITY_ADMIN_USER=${GF_SECURITY_ADMIN_USER}
      - GF_SECURITY_ADMIN_PASSWORD=${GF_SECURITY_ADMIN_PASSWORD}
      - GF_INSTALL_PLUGINS=yesoreyeram-infinity-datasource
      - GF_SECURITY_ALLOW_EMBEDDING=true
      - GF_AUTH_GENERIC_OAUTH_ENABLED=true
      - GF_AUTH_GENERIC_OAUTH_NAME=Keycloak
      - GF_AUTH_GENERIC_OAUTH_CLIENT_ID=grafana
      - GF_AUTH_GENERIC_OAUTH_CLIENT_SECRET=${
          GF_AUTH_GENERIC_OAUTH_CLIENT_SECRET}
      - GF_AUTH_GENERIC_OAUTH_SCOPES=openid profile email
      - GF_AUTH_GENERIC_OAUTH_AUTH_URL=https://auth.
          christiandellisanti.uk/realms/measurestream/protocol/
          openid-connect/auth
      - GF_AUTH_GENERIC_OAUTH_TOKEN_URL=https://auth.
          christiandellisanti.uk/realms/measurestream/protocol/
          openid-connect/token
      - GF_AUTH_GENERIC_OAUTH_API_URL=https://auth.
          christiandellisanti.uk/realms/measurestream/protocol/
          openid-connect/userinfo
      - GF_AUTH_GENERIC_OAUTH_LOGOUT_URL=https://auth.
          christiandellisanti.uk/realms/measurestream/protocol/
          openid-connect/logout?redirect_uri=https://grafana.
          christiandellisanti.uk
      - GF_AUTH_OAUTH_AUTO_LOGIN=true
      - GF_AUTH_GENERIC_OAUTH_USE_PKCE=true
      - GF_AUTH_GENERIC_OAUTH_USERNAME_PATH=preferred_username #
          or 'sub'
      - GF_AUTH_GENERIC_OAUTH_EMAIL_PATH=email
      - GF_AUTH_GENERIC_OAUTH_NAME_PATH=name
    networks:
      app_network:
        ipv4_address: 172.20.0.30
```

Listing 3.1: Docker Compose configuration for Grafana integration

To enable seamless integration of Grafana dashboards into the front-end interface, Grafana is deployed as a Docker service using the official grafana-enterprise image. The container exposes port 3000 to allow external access and mounts persistent volumes for storing Grafana data, custom plugins, and configuration files. Authentication and user management are delegated to Keycloak, which serves as the OAuth2 identity provider. The OAuth2 integration is enabled and configured by specifying the necessary client credentials and endpoints. The environment variables define the client ID and secret, the URLs for authorization, token exchange, user information retrieval, and logout redirection. This setup enables single sign-on capabilities and allows the platform to leverage identity federation across multiple services. Finally, GF_SECURITY_ALLOW_EMBEDDING=true is essential to permit Grafana dashboards to be embedded within iframes – without this, modern browsers would block such embedding due to security policies. This setup simplifies the user experience, allowing the UI to load Grafana content directly without routing through the API Gateway or requiring user authentication.

```
# Stage 1: build React app
FROM node:20-alpine AS builder
WORKDIR /app
COPY package.json ./
COPY package-lock.json ./
RUN npm ci
COPY . .
RUN npm run build

# Stage 2: serve with nginx
FROM nginx:alpine
COPY --from=builder /app/dist /usr/share/nginx/html/ui
EXPOSE 80
```

Listing 3.2: Dockerfile for building and serving the React front-end

Finally, regarding application deployment, the front-end is fully containerized using Docker. The image is built through a GitHub Action, which automates the build process. The Dockerfile defines two main stages: in the first stage, the React application is built using a lightweight base image (such as Alpine Linux), installing dependencies via package.json and package-lock.json, and compiling the project with npm run build. In the second stage, the generated static files are served using Nginx, ensuring fast and efficient deployment. Thanks to this architecture, the user interface can be easily deployed across different environments, offering a high degree of scalability and maintainability.

Chapter 4

Hardware

4.1 Hardware Description

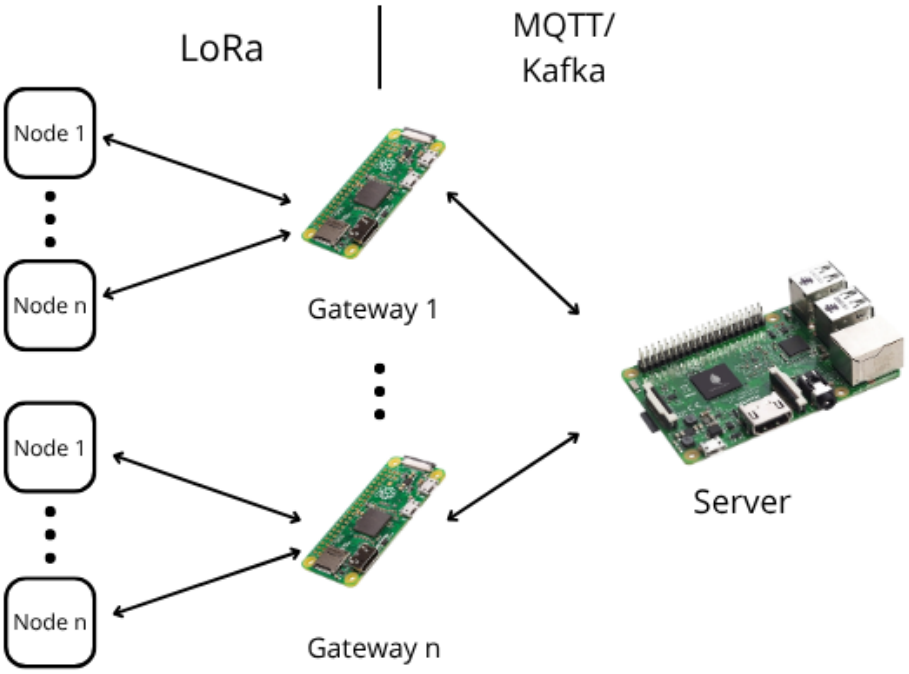

Figure 4.1: MeasureStream Architecture

The hardware system architecture is composed by three component: the nodes, the gateway(GW) and the server. Each play key roles in the network of sensors, ensuring the communication, processing and distribution of data between nodes and the software platform. The system follows a star architecture, where multiple nodes are connected to a single gateway, and the gateways are, in turn, connected to a central server.

The gateway serves as the link between the sensor nodes deployed in the field and the central server. Its main function is to collect the measurements gathered by the nodes and transmit them over the network to the server, as well as to forward any control messages to the nodes. For the gateway, it has been chosen the Raspberry Pi Zero W due to its ability to run a lightweight operating system, which is essential for deploying Kafka and MQTT, as well as enabling the communication with the central server. The Raspberry Pi Zero W features a 1GHz single-core ARM11 CPU, 512MB of RAM, and built-in 2.4GHz 802.11n Wi-Fi, providing sufficient resources and wireless connectivity for basic data processing and message brokering. For communication with the sensor nodes, the gateway uses LoRa technology, which integrates seamlessly with the Raspberry Pi Zero W through its GPIO interface. Additionally, its low power consumption—typically around 100 mA at 5V—makes it an ideal choice for field deployments where energy efficiency and compact form factor are critical.

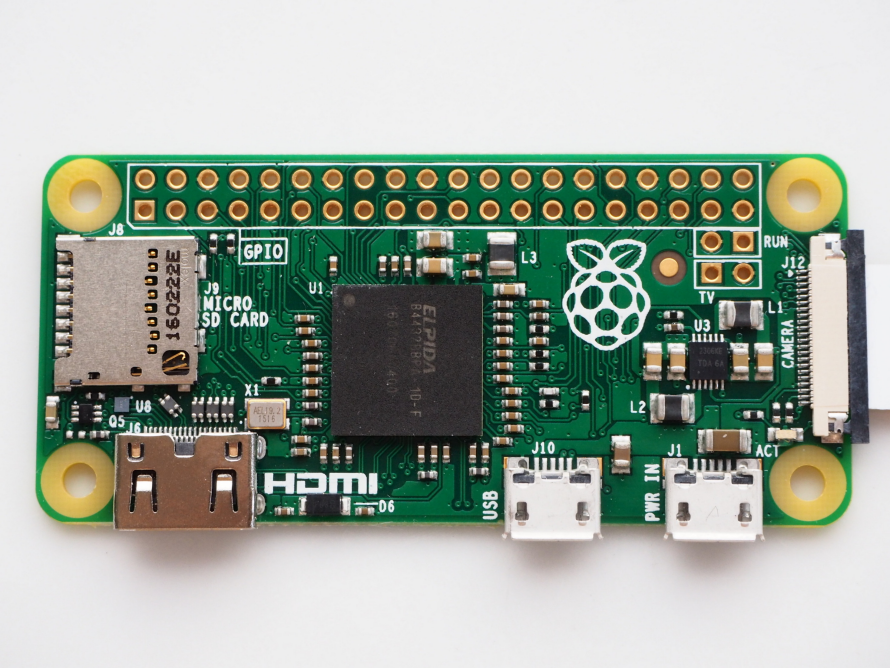

Figure 4.2: Raspberry Pi Zero

The Raspberry Pi Zero W runs Raspberry Pi OS Bookworm Lite, a minimal version of the operating system without a graphical user interface. This choice was made to reduce resource usage and improve system performance, as a desktop environment is unnecessary for our use case. The headless setup allows the device to operate efficiently in the field, dedicating its limited hardware resources entirely to essential tasks such as running Kafka, managing LoRa communication, and handling network operations. Moreover, the lightweight OS simplifies maintenance and enhances system stability in long-term deployments.

For testing purposes, the Raspberry Pi Zero W was connected to the Politecnico di Torino's Wi-Fi network. Although the initial setup presented some challenges due to network restrictions and configuration complexity, we successfully established internet connectivity, which was essential for running Kafka and enabling remote access via SSH. The connection was configured using the NetworkManager CLI tool with the following command:

```
nmcli connection add type wifi ifname wlan0 con-name Polito
   ssid Polito \
wifi-sec.key-mgmt wpa-eap 802-1x.eap peap \
802-1x.identity "username@studenti.polito.it" \
802-1x.password "password" \
802-1x.system-ca-certs no \
802-1x.phase2-auth mschapv2
```

Listing 4.1: Command used to connect the Raspberry Pi Zero W to the Polito Wi-Fi network using WPA-Enterprise (802.1X) authentication

This allowed secure access to the university network using WPA-Enterprise (802.1X) authentication. Additionally, a dedicated user named gateway was created with the password gateway to simplify device access and management during development.

For the central server, it has been selected a Raspberry Pi 4 Model B with 8GB of RAM and a 128GB SD card, providing ample resources for running containerized services and managing sensor data. The Raspberry Pi 4 features a quad-core ARM Cortex-A72 processor clocked at 1.5GHz, dual-band 802.11ac Wi-Fi, Bluetooth 5.0, and Gigabit Ethernet, making it a highly capable and compact solution for edge computing applications. The device is connected to the network via Ethernet and configured with a public IP address, although only port 22 has been opened to allow secure remote management via SSH. As with the gateway, we installed Raspberry

Pi OS Bookworm Lite without a graphical user interface to optimize performance and reduce unnecessary overhead. A dedicated user named measurestream was created with the password measurestream. To enhance security, SSH access was configured to use RSA key-based authentication, and password-based login was subsequently disabled. Finally, Docker was installed on the system to enable the deployment and orchestration of various services, such as Kafka, within isolated containers. To ensure proper thermal management and physical protection, a custom 3D-printed case for the Raspberry Pi was realized and a heatsink was added in order to improve heat dissipation during continuous operation.

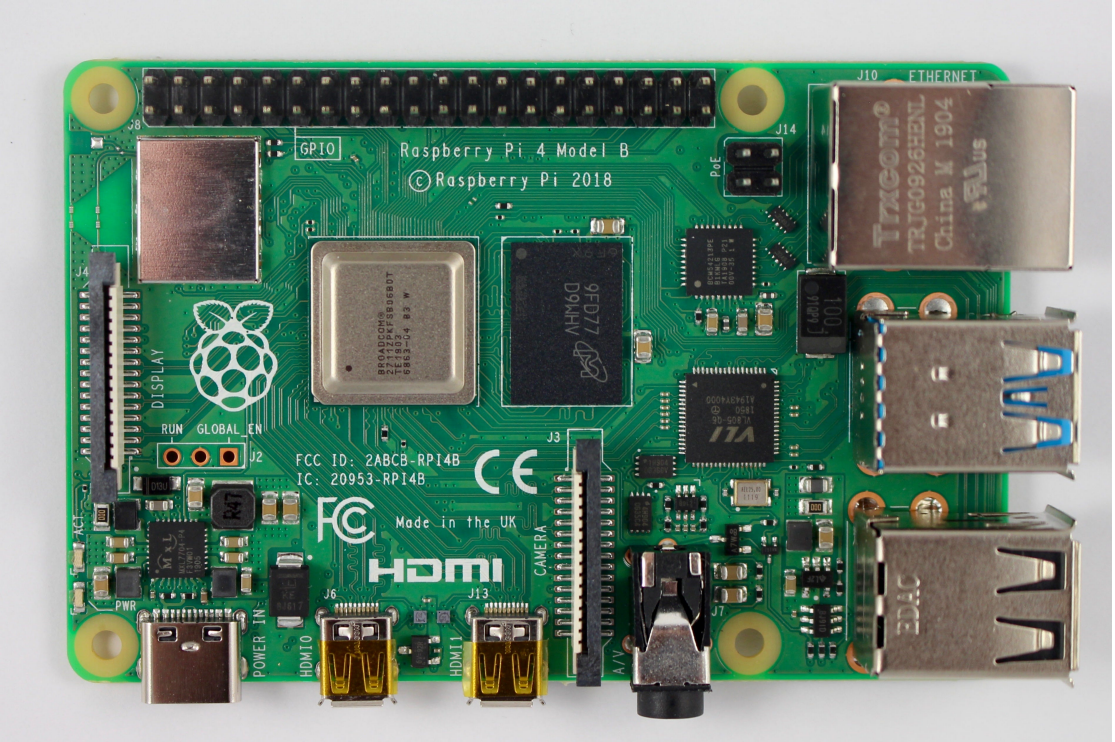

Figure 4.3: Raspberry Pi 4 Model B

4.2 Development

As part of the deployment pipeline for the Spring Boot application on a Raspberry Pi 4, a continuous integration and continuous deployment (CI/CD) strategy was adopted using GitHub Actions. CI/CD is a modern practice within the broader DevOps paradigm, which promotes a set of cultural philosophies, tools, and practices aimed at unifying software development (Dev) and IT operations (Ops). The primary objective of DevOps is to shorten the software development lifecycle while ensuring high-quality and reliable delivery of software. Within this framework, CI/CD automates the process of building, testing, and deploying code, enabling more frequent and consistent releases.

In this project, GitHub Actions was configured to automatically execute a workflow upon each push to the main branch, ensuring that every committed change is promptly integrated and made available for deployment. The workflow begins by checking out the source code and setting up the Java 17 environment using the Temurin distribution. It then compiles the Spring Boot application using Gradle's bootJar task. A critical component of the pipeline is the use of Docker Buildx, which facilitates the cross-compilation of Docker images for multiple target architectures. This step is essential because the development and CI environments typically run on an x86_64 architecture, while the Raspberry Pi 4 operates on ARM64. To address this architectural difference, the workflow builds a multi-arch Docker image for both linux/amd64 and linux/arm64 and pushes it to Docker Hub under the tag latest.

On the Raspberry Pi, deployment is handled through Docker Compose, which is configured to pull the latest version of the image from Docker Hub. This setup ensures that the most recent build is always deployed on the device without requiring manual updates or interventions. Overall, this automated workflow exemplifies an effective DevOps-oriented CI/CD pipeline, providing a reliable, architecture-aware, and fully automated approach to building and deploying the application across heterogeneous environments.

To ensure proper version tracking and release management, the workflow also integrates versioning based on the Semantic Versioning (SemVer) standardṠemantic Versioning is a widely adopted versioning convention that uses a three-part version number format: MAJOR.MINOR.PATCH. According to the SemVer specification:

- The MAJOR version is incremented when incompatible API changes are introduced.
- The MINOR version is incremented when new, backward-compatible functionality is added.
- The PATCH version is incremented when backward-compatible bug fixes are made.

This versioning scheme was incorporated into the CI pipeline using Git tags, allowing each Docker image to be tagged with its corresponding version number (e.g., v1.2.0) in addition to the latest tag. This practice not only improves traceability and rollback capabilities but also supports better coordination in multi-environment deployments and collaborative development scenarios. Finally, the development phase itself is conducted by connecting directly to the Raspberry Pi via SSH (Secure Shell). For secure and password-less access, SSH is configured to use public key authentication. In this method, the developer generates a pair of cryptographic keys on their local machine: a private key, which remains secret and secure on the client, and a public key, which is copied to the Raspberry Pi and stored in the /.ssh/authorized_keys file of the target user. When an SSH connection is initiated, the Raspberry Pi uses the stored public key to verify a cryptographic challenge signed by the client's private key. If the verification is successful, access is granted without requiring a password. This approach enhances security by eliminating the need to transmit passwords over the network and also facilitates automation and scripting by allowing secure, non-interactive logins.

4.3 Communication

The gateway serves as a crucial communication bridge between the physical sensor network and the central server, utilizing two main communication protocols to fulfill its functions. First, it handles the ingestion of sensor measurements collected via LoRa, a long-range, low-power wireless protocol designed for Internet of Things (IoT) applications. All data received through LoRa are published by the gateway into a Kafka topic named "measures", using the JSON format and adhering to the schema defined by the MeasureDTO data class, which includes fields such as the measurement value, unit of measurement, timestamp, and the node identifier. This asynchronous and scalable architecture enables the decoupling of data producers and consumers and facilitates reliable high-throughput ingestion into the back-end system. In addition to data transmission, the gateway is responsible for bidirectional communication concerning sensor configuration and control. This is achieved through MQTT protocol managed via Mosquitto, an open-source MQTT broker. The gateway subscribes to and publishes messages on a series of predefined topics.All MQTT messages follow a well-defined schema, particularly the CommandDTO data class, which includes information such as the command ID, the IDs of the target gateway, CU, or MU, the operation type (e.g., "CREATE", "UPDATE", "DELETE"), and optional configuration payloads (CuSettingDTO and MuSettingDTO).

Both Mosquitto, which is based on the MQTT protocol, and Kafka, which uses its own custom binary protocol over TCP, require open ports for communication. By default, Mosquitto operates on port 1883, while Kafka brokers communicate over port 9092. In their standard configurations, both services transmit data over unencrypted channels, which exposes the system to a range of cybersecurity threats, including Man-in-the-Middle (MITM) attacks, spoofing, and distributed denial-of-service (DDoS) attacks. To address these vulnerabilities, two primary approaches can be considered:

- Enabling secure communication through the configuration of TLS/SSL encryption for both Mosquitto and Kafka.
- Establishing a private network using a VPN (Virtual Private Network), which ensures the confidentiality and integrity of messages by encapsulating traffic within a secure tunnel.

While the first solution—configuring encryption directly on each service—is a valid and commonly recommended practice, it presents significant deployment challenges in certain environments. Specifically, within the infrastructure of the Politecnico di Torino, strict firewall policies and the absence of static public IP addresses complicate the setup of TLS-secured endpoints and the necessary port forwarding. As a result, the adopted solution was to configure a VPN connection between all participating devices. Among the various available VPN solutions, the focus was placed on free and open-source technologies. In particular, WireGuard and Tailscale were evaluated due to their performance, ease of configuration, and strong security guarantees. These tools provided a robust and practical foundation for securing inter-device communication within the system. WireGuard is a modern and high-performance Virtual Private Network (VPN) protocol that leverages state-of-the-art cryptographic primitives, such as Curve25519 for key exchange and ChaCha20 for symmetric encryption. It operates directly over the UDP protocol, which enables it to deliver superior performance and lower latency compared to traditional VPN protocols like IPsec and OpenVPN. To establish a WireGuard-based VPN, at least one node within the network must act as a listening peer, often referred to as the server. This peer must have a publicly reachable UDP port—typically port 51820—to accept inbound connections from other peers. However, this requirement presents a limitation in specific network environments. In particular, within the infrastructure of the Politecnico di Torino, opening external UDP ports is not feasible due to institutional firewall restrictions. A potential workaround involves renting a Virtual Private Server (VPS) and configuring it as the WireGuard listening peer. While technically effective, this approach incurs additional monthly costs and is therefore not considered ideal for the development phase of a prototype. To address this limitation, the proposed system adopts Tailscale, a VPN service built on top of WireGuard. Tailscale provides a simplified deployment model by managing peer discovery, NAT traversal, and key exchange, without requiring manual configuration of public IPs or port forwarding. Furthermore, Tailscale offers a free license tier for networks comprising fewer than 100 devices, making it a highly practical and cost-effective solution for prototyping. Nonetheless, for production deployments, where scalability, control, and long-term reliability are crucial, a migration to a self-managed WireGuard infrastructure is recommended.

Internet Exposure Architecture

To expose the application to the public internet, several solutions were evaluated. However, considering time constraints and the firewall restrictions imposed by the University's network, the most suitable approach was the implementation of a Cloudflare Tunnel. A domain name — christiandellisanti.uk — was registered through Cloudflare, and various subdomains were configured to map to specific services running on a Raspberry Pi 4B server. These mappings are as follows:

- www.christiandellisanti.uk maps to the main web application running on localhost:8080
- auth.christiandellisanti.uk maps to the Keycloak authentication server.
- grafana.christiandellisanti.uk maps to the Grafana monitoring dashboard.
- portainer.christiandellisanti.uk maps to the Portainer container management interface.

This configuration allows the services to be accessed externally without requiring port forwarding or changes to the University's firewall. A Cloudflare Tunnel is a secure method of exposing services hosted on private networks to the internet. Instead of requiring open inbound ports, the tunnel establishes an encrypted outbound connection from the internal server to Cloudflare's global network. Cloudflare then acts as a reverse proxy, securely routing incoming requests to the internal services. This solution offers three key advantages:

- Firewall and NAT Bypass: No inbound connections are required, making it ideal in restricted environments such as academic institutions.
- Enhanced Security: Traffic is encrypted and protected by Cloudflare's security infrastructure, including DDoS protection and automatic HTTPS.
- No Public IP Required: The server can operate behind dynamic IPs or NAT without the need for a static IP or dynamic DNS.

In summary, the use of a Cloudflare Tunnel provided a robust, secure, and firewall-friendly method for exposing internal services to the internet, significantly simplifying deployment and access management.

Chapter 5

Conclusion

This work presents the design and implementation of a distributed and scalable infrastructure for the management of LoRa sensor networks, specifically targeting industrial contexts where sensor monitoring and calibration are essential. The proposed system integrates a microservices architecture built with Spring Boot and Kafka, enabling asynchronous communication between independent services and ensuring high modularity, scalability, and resilience. To achieve a fully functional and maintainable system, the back-end services were containerized using Docker and orchestrated via Docker Compose. Sensor data, including calibration certificates, are securely stored and accessed through dedicated APIs, while user authentication and authorization are handled through an Identity and Access Management (IAM) system based on Keycloak and OpenID Connect protocols.

On the front-end side, a single-page application (SPA) was developed using React and TypeScript to offer a responsive, modern, and interactive interface for visualizing measurements and managing device status. Data visualization is further enhanced through the integration of Grafana dashboards.

To automate the software lifecycle, a CI/CD pipeline was implemented using GitHub Actions. This pipeline ensures that each code update pushed to the main branch is automatically built, tested, and packaged as a multi-architecture Docker image using Docker Buildx. The image is published to Docker Hub and pulled by the Raspberry Pi 4 device, which serves as the deployment target. This setup guarantees a seamless and architecture-aware delivery process, supporting both amd64 and ARM64 platforms.

Development and debugging were performed via SSH access to the Raspberry Pi. SSH authentication was configured using public key cryptography, where a public key is stored on the device and a corresponding private key is securely held by the developer. During authentication, the client proves possession of the private key by responding to a cryptographic challenge, ensuring a secure, password-less connection.

Furthermore, the project adopts Semantic Versioning (SemVer) for tagging releases, following the standard MAJOR.MINOR.PATCH format. This versioning scheme provides a clear and structured way to communicate changes, ensuring compatibility and traceability across builds.

In addition to the architectural and technical contributions, this work offers tangible benefits for companies that rely on sensor-based monitoring systems. By introducing a digital and automated management system for sensor calibration certificates, organizations can significantly reduce operational inefficiencies, eliminate manual tracking processes, and ensure ongoing compliance with metrological standards. The centralized dashboard and alerting mechanisms improve the visibility over sensor health and status, while the integration of open-source technologies enables adaptability and cost-effectiveness.

Thanks to its modularity, the proposed system can be easily extended or integrated into existing industrial infrastructures, regardless of the manufacturer or technology of the sensors involved. This opens the way for broader adoption of standardized sensor networks and supports the transition to Industry 5.0, where human-centric and intelligent production systems require high levels of data reliability and process automation.

Overall, this thesis demonstrates a comprehensive and modern approach to the development of an open, flexible, and secure IoT infrastructure aligned with the principles of Industry 5.0 and DevOps best practices, providing companies with a powerful tool to enhance sensor network management, improve decision-making, and optimize industrial performance.

Appendix: Automatic Backup with Duplicati

Duplicati is a free, open-source backup software that enables secure, encrypted, and incremental backups to a variety of remote storage providers, including cloud services like Dropbox, Google Drive, Amazon S3, and many others. It is designed for both personal and server environments, and supports compression, strong AES-256 encryption, and scheduling.

In this project, Duplicati has been configured to automatically back up the PostgreSQL database used by Keycloak, which serves as the identity and access management solution of the system. The backup is uploaded to Dropbox to ensure offsite availability and recovery in case of failure. The backup job is scheduled to run daily at 13:00, ensuring that a recent copy of the Keycloak database is always available. This configuration leverages Duplicati's built-in scheduler and command-line or web-based interface to define and monitor backup tasks. Maintaining automated backups of critical components—such as the Keycloak database—is fundamental for any reliable and secure system. In the event of:

1. Data corruption
2. Hardware or disk failures
3. Security incidents (e.g., ransomware attacks)

having a recent backup allows for a fast recovery, minimizing downtime, data loss, and service disruption. Given that Keycloak manages authentication, user roles, and permissions, a loss of this database would compromise the security model of the entire system. Automating backups ensures that administrative oversight or forgetfulness doesn't jeopardize recovery capabilities.

Appendix: Container Management with Portainer

Portainer is an open-source container management tool that provides a user-friendly web interface for managing Docker environments. It abstracts the complexity of container orchestration and allows developers and system administrators to interact with their containers, images, volumes, networks, and even stacks through an intuitive graphical dashboard.

In this project, Portainer has been used to manage the entire Docker-based microservices infrastructure, which includes services such as Keycloak, PostgreSQL, Duplicati, and others. It simplifies operations such as:

- Starting, stopping, or restarting containers
- Inspecting logs and container metrics
- Monitoring resource usage (CPU, memory, etc.)
- Updating container configurations
- Managing Docker volumes and networks

The use of Portainer is particularly beneficial in development and testing environments where rapid deployment, debugging, and resource management are required. It also helps ensure the reliability and maintainability of the system by making the state of the container ecosystem transparent and manageable even for those who are less familiar with Docker's command-line interface. By enabling visual control over the deployed services, Portainer enhances the operational workflow, reduces the chance of configuration errors, and provides a centralized place to monitor the health of the entire containerized infrastructure.

Bibliography

[1] Francis Botto. *Dictionary of Multimedia and Internet Applications*. Wiley, 1999.

[2] Ryan Breidenbach Craig Walls. *Spring in Action*. illustrated edition. In Action series. Manning, 2005. ISBN: 9781932394351; 1932394354.

[3] Michael Dirolf Kristina Chodorow. *MongoDB: The Definitive Guide*. O'REILLY, 2010. ISBN: 1449381561; 9781449381561.

[4] National Institute of Standards and Technology. *The Keyed-Hash Message Authentication Code (HMAC)*. Federal Information Processing Standards Publication 198-1. Supersedes FIPS 198 (2002). Will be superseded by NIST SP 800-224. U.S. Department of Commerce, National Institute of Standards and Technology, July 2008. URL: `https://doi.org/10.6028/NIST.FIPS.198-1`.

[5] Sebastian E. Peyrott. *The JWT Handbook*. Auth0 Inc., 2024. ISBN: 1449381561; 9781449381561.

[6] Karl Matthias Sean Kane. *Docker: Up & Running: Shipping Reliable Containers in Production (Early Release)*. 3rd ed. O'Reilly Media, 2023. ISBN: 9781098131821; 1098131827.

Sitography

[1] Apache Software Foundation. *Apache Kafka Documentation*. Accessed: 2025-04-09. 2025. URL: https://kafka.apache.org/documentation/#zk2kraft-summary.

[2] Auth0. *ID Token vs Access Token: What's the Difference?* Accessed: 2025-04-13. n.d. URL: https://auth0.com/blog/id-token-access-token-what-is-the-difference/.

[3] Akshita Dixit. *Mimicking MongoDB Plugin in Grafana for Free – A Workaround*. Published in CodeX. Accessed on April 10, 2025. Medium. Aug. 4, 2024. URL: https://medium.com/codex/mimicking-mongodb-plugin-in-grafana-for-free-a-workaround-5da3639306af.

[4] KirstenS et al. *Cross Site Request Forgery (CSRF)*. Accessed: 2025-04-23. OWASP Foundation. n.d. URL: https://owasp.org/www-community/attacks/csrf.

[5] Robert C. Martin. *The Clean Architecture*. Accessed: 2025-04-02. 2012. URL: https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html.

[6] MDN Web Docs. *Introduction to the Document Object Model (DOM)*. Accessed: 2025-04-25. 2025. URL: https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model/Introduction.

[7] MDN Web Docs. *SPA (Single-page application)*. Accessed: 2025-04-25. 2025. URL: https://developer.mozilla.org/en-US/docs/Glossary/SPA.

[8] MongoDB, Inc. *NoSQL Explained*. Accessed on April 10, 2025. 2025. URL: https://www.mongodb.com/en/resources/basics/databases/nosql-explained.

[9] OpenID Foundation. *How OpenID Connect Works*. Accessed: 2025-04-13. 2025. URL: https://openid.net/developers/how-connect-works/.

[10] OWASP Foundation. *Cross-Site Request Forgery Prevention Cheat Sheet*. Accessed: 2025-04-23. n.d. URL: https://cheatsheetseries.owasp.org/cheatsheets/Cross-Site_Request_Forgery_Prevention_Cheat_Sheet.html.

[11] React Team. *Describing the UI*. Accessed: 2025-04-25. 2025. URL: https://react.dev/learn/describing-the-ui.