Development of a Device-Oriented Open Source Cloud Platform

Ignacio Gomis Payá

CONTENTS

LIST OF FIGURES

I. Introduction

A. Context and Justification of the Project

The project aims to address the increasing demands in monitoring and telemetry, mainly by using IoT (Internet Of Things) devices, providing an open-source platform for efficient organization and management of large-scale deployments. The platform answers the next main needs:

1) Current Demands in Monitoring and Telemetry: With the proliferation of IoT devices and cloud technologies, there is a growing need for robust monitoring and telemetry solutions to handle the vast amount of data generated.

2) Need for Organization and Management of Deployments: Large-scale deployments require effective organization and management to ensure seamless operation and maintenance.

B. Project Objectives

The primary objectives include the development of an open-source platform and its deployment on major cloud providers such as AWS [1], Google Cloud [2], and Azure [3].

1) Development of an Open-Source Platform: Creating a platform that is accessible, customizable, and open to community contributions for continuous improvement.

2) Deployment on Cloud Providers (AWS, Google Cloud, Azure): Ensuring compatibility and deployment flexibility by targeting major cloud providers.

C. Scope and Limitations

1) Key Platform Functionalities: The platform will support device provisioning from various sources, historical data storage, and device visualization and analysis.

2) Restrictions and Out-of-Scope Areas: Certain functionalities, such as detailed device analytics, may be considered for future improvements due to project scope constraints.

II. Documentation Process

A. Literature Review

1) Current State of IoT Platforms: Nowadays, several IoT platforms have gained prominence for their features, scalability, and integration capabilities. Below are listed some of the more relevant ones:

- **Amazon Web Services (AWS) IoT** [4]
  AWS IoT provides a comprehensive set of services to connect, manage, and secure IoT devices. It includes device management, message brokering, rule-based actions, and integration with other AWS services.
- **Microsoft Azure IoT Suite** [5]
  Azure IoT Suite offers a range of services for device connectivity, management, and data analysis. It integrates with Azure IoT Hub, Azure Stream Analytics, and Azure Machine Learning for advanced analytics and AI capabilities.
- **ThingSpeak** [6]
  ThingSpeak is an open IoT platform from MathWorks that allows users to collect, analyze, and visualize IoT data in real-time. It's often used with MATLAB for advanced analytics and modeling.
- **ThingsBoard** [7]
  ThingsBoard is an open-source IoT platform designed to facilitate the efficient management, visualization, and analysis of connected devices and their data. With a focus on scalability and flexibility, ThingsBoard offers a comprehensive set of features for IoT solution development.
- **Particle IoT Platform** [8]
  Particle offers a comprehensive IoT platform that includes hardware, connectivity, and cloud services. It's known for its ease of use and supports prototyping and scaling IoT solutions.

2) Background and related technologies: This section encompass a broad technological landscape that involves various components and layers. Below, some of the key elements are described in the context of IoT platforms: **Internet of Things (IoT):**

- **Background**
  The IoT is a network of interconnected physical devices, vehicles, appliances, and other objects embedded with sensors, software, and connectivity. These devices collect and exchange data to enable intelligent decision-making and automation.
- **Related Technologies**
  IoT devices may use various communication protocols such as MQTT [9], CoAP [10], and HTTP [11] to transmit data. Sensors, actuators, and microcontrollers are essential hardware components in IoT devices.

Cloud Services:

- **Background**
  Cloud computing provides scalable and on-demand access to a pool of computing resources over the internet. Cloud services are integral to IoT platforms, offering storage, processing power, and other infrastructure services.
- **Related Technologies**
  - **Infrastructure as a Service (IaaS):** Provides virtualized computing resources.
  - **Platform as a Service (PaaS):** Offers a platform for developers to build, deploy, and manage applications without dealing with the underlying infrastructure.
  - **Software as a Service (SaaS):** Delivers software applications over the internet without the need for local installation.

Communication Protocols:

- **Background**
  Communication protocols are essential for IoT devices to exchange data securely and efficiently. They define how devices communicate with each other and with cloud platforms.
- **Related Technologies**
  - **MQTT (Message Queuing Telemetry Transport):** A lightweight, publish-subscribe messaging protocol widely used in IoT.
  - **CoAP (Constrained Application Protocol):** Designed for resource-constrained devices and networks.
  - **HTTP/HTTPS (Hypertext Transfer Protocol):** Commonly used for web-based communication.

Open-Source Frameworks:

- **Background**
  Open-source frameworks provide a foundation for building and deploying IoT applications. They promote collaboration, flexibility, and customization.
- **Related Technologies**
  - **FIWARE [12]:** A comprehensive open-source framework for building smart solutions in various domains, including IoT, Smart Cities, and Industry 4.0.
  - **Eclipse IoT [13]:** An open-source community that provides a set of projects for IoT development.

- **Node-RED [14]:** A flow-based development tool for visual programming of IoT applications.
- **IoTivity [15]:** Open-source framework for device-to-device communication and interoperability.
- **OpenStack [16]:** A cloud computing platform that provides IaaS for building and managing public and private clouds.
- **Sentilo [17]:** Sentilo is an open-source platform developed for smart city and IoT applications. It facilitates real-time data acquisition, storage, and management of sensor and actuator data. The platform promotes interoperability and enables developers to create applications for monitoring and controlling various aspects of urban environments.

B. Theoretical Framework

1) Key Concepts for Platform Development: In the context of developing an IoT platform, several key concepts play a crucial role in ensuring the platform's effectiveness and functionality. Understanding these concepts provides a solid foundation for designing and implementing a robust system.

- **Device Provisioning:** Device provisioning is the process of onboarding and configuring IoT devices to make them operational within the platform. This involves assigning unique identifiers and defining communication protocols. Efficient device provisioning ensures a streamlined integration of new devices into the IoT ecosystem.
- **Data Model Design:** Developing a robust and flexible data model is critical for ensuring the long-term flexibility and scalability of the platform. A well-designed data model accommodates evolving requirements, new types of devices, and changing data structures. It forms the foundation for data interoperability and future expansions.
- **Data Storage:** Data storage is another fundamental aspect of IoT platforms, involving the secure and scalable storage of information generated by IoT devices. Historical data storage allows for retrospective analysis, trend identification, and informed decision-making. The choice of a suitable database system influences the platform's overall performance.
- **User Interface:** The design of the user interface plays a key role in the overall success of the platform. An intuitive and user-friendly interface facilitates efficient interaction, reducing the learning curve for users. It ensures that users can easily navigate through functionalities such as entity management, data visualization/exploration, and system configuration.

2) Technologies and tools: The selection of technologies and tools for IoT platform development is a critical decision that significantly impacts the platform's efficiency, scalability, and flexibility. The following technologies and specific tools should be chosen based on their suitability for various aspects of the IoT ecosystem:

- **Context Broker:** The broker serves as the central hub for real-time data management in the platform. Its capability have to handle context information efficiently for managing the dynamic and diverse data generated by IoT devices.
- **Context database:** This database system is employed for storing current state and time-aggregated data of entities within the platform. A document-oriented structure is well-suited for handling unstructured data common in IoT applications, providing flexibility and scalability.
- **Pub-sub messaging protocol:** A lightweight, publish-subscribe messaging protocol for efficient communication between IoT devices and the platform should fits for the platform. A low overhead and support for intermittent connectivity for resource-constrained IoT devices is a requirement.
- **UI framework:** A framework for developing the user interface of the platform is crucial for the user experience. This framework should provides the creation of interactive, web-based visualizations, providing a user-friendly experience for monitoring and managing IoT devices.
- **Persistence database:** A suited database should be employed for persisting historical data from the broker. A distributed database architecture ensures reliable storage and retrieval of time-series data, essential for analyzing trends and patterns over time.
- **Containerization:** It allows to package the various components of the platform, facilitating seamless deployment and scalability. Containers encapsulate each service, ensuring consistent performance across different environments.
- **RESTful API connectivity:** RESTful APIs provide a standardized and scalable means of integrating external services into the IoT platform.
- **Web Scraping:** Web scraping techniques approach allows for the extraction of relevant information from websites, enhancing the platform's ability to incorporate diverse data sources.

By leveraging these technologies and tools, the IoT platform is equipped with a robust foundation for device management, data processing, and user interaction. The chosen stack should be aligned with industry best practices and ensures a comprehensive and efficient solution for large-scale IoT deployments.

III. Challenges for IoT platforms

A. Problem Identification and Open-Source Platform Paradigm

- Challenges in Large-Scale Deployment Management: Managing a large number of deployed devices efficiently poses several challenges crucial for IoT platform success. Key challenges include scalability, handling increasing data volume and velocity, device heterogeneity, security and privacy concerns, reliability and fault tolerance, monitoring and diagnostics, over-the-air updates, and cost management.
- Open-Source Platform Paradigm: The decision to develop an open-source IoT platform addresses limitations of proprietary solutions and specific user needs. Reasons include independence from proprietary solutions, long-term sustainability, community collaboration, customization and flexibility, avoiding vendor lock-in, transparency and security, global collaboration and support, and adherence to standards.

B. IoT Platform Requirements

- Functional Requirements for the IoT Platform: The IoT platform must ensure effectiveness and user satisfaction in key functional areas such as device provisioning, secure and scalable data storage, and robust visualization capabilities.
- Non-Functional Requirements:
  - Scalability:
    - Horizontal Scalability: Support handling a growing number of IoT devices through multiple instances.
    - Load Balancing: Implement mechanisms for even distribution of incoming requests.
    - Database Sharding: Distribute MongoDB data across multiple nodes for enhanced performance.
    - MQTT Broker Scalability: Ensure scalability of a cloud-based MQTT broker or deploy a scalable solution.
  - Reliability:
    - High Availability: Ensure critical components have redundant instances and failover mechanisms.
    - Health Checks: Regularly perform health checks to proactively identify potential issues.
    - Fault Tolerance: Implement mechanisms to handle individual component failures without system-wide disruptions.
    - Data Integrity: Ensure historical data integrity in the CrateDB database during data processes.
  - Deployment Flexibility:
    - Cloud Provider Compatibility: Design the platform to be cloud-agnostic for deployment on various providers.
    - On-Premise Deployment: Support on-premise deployment for organizations with specific security or compliance requirements.
    - Containerization Compatibility: Ensure containerized services for deployment across different environments.
    - Configuration Management: Implement tools for easy deployment adaptability in various environments.
    - Addressing these non-functional requirements ensures scalability, reliability, deployment flexibility, and security in both local server and cloud environments for the IoT platform.

C. Architecture Design

UNE 178104:2017 [18] depicts an standard for the development of smart city platforms. The Figure 1, extracted from the standard document, outlines the platform as a layer model.

According to that standard, the approach for the developed IoT platform is summarized in the figure 2.

This architecture aims to address the issues and requirements identified in the preceding sections:

- Capturing systems: This layer is made up of all the sensors/actuators managed by the IoT platform and other sources as external IT systems.
- Acquisition layer/interconnection: The objective is to offer mechanisms for capturing data from the capture systems, enabling interconnection with other systems which only use data. Consequently, it abstract information from the capture system using a standard semantic approach.
- Knowledge layer: It supports the processing of data by receiving data from the acquisition/interconnection layer. The layer also provides functions which enable scalable movement of data (with a set of unified APIs to interconnect elements) and generate new datasets or modify/complete existing ones.
- Interoperability layer: This layer facilitates the provision of services in the IoT platform.
- Service layer: It contains the services provided by the IoT platform to the users.
- Support layer: This is a transversal layer which provides support to the whole platform layers, such as management, security...

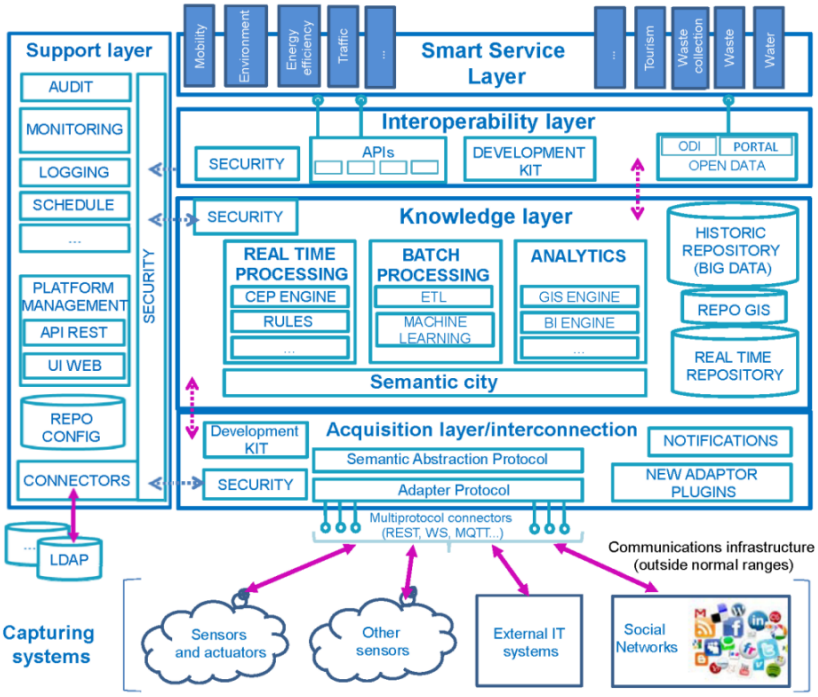


Fig. 1: UNE 178104:2017 Layer Model

IV. Proposed Solution

The previous section defines the needs and the approach for the IoT platform. This section depicts the specific proposed solution adopted for the different components of the platform.

FIWARE is the chosen core framework for the IoT platform ecosystem, as it addresses the identified needs for the IoT platform through an open-source component catalog, community support, and ongoing development.

The next subsections follow the structure of the Layer Model according to the UNE 178104:2017, the figure 3 depicts the proposed architecture considering the used components.

A. Main components

1) Capturing systems: For the developed IoT platform, 3 different sources has been considered and integrated on it:

- **IoT Device**: A combination of an ESP32 board [19] and a DHT22 [20] module.
  The ESP32 with DHT22 module is a combination of hardware components commonly used in Internet of Things (IoT) projects for monitoring and controlling temperature and humidity: ESP32
  - The ESP32 is a versatile and powerful microcontroller developed by Espressif Systems. It is widely used in IoT applications due to its built-in Wi-Fi and Bluetooth capabilities, as well as its low power consumption.
  - The ESP32 serves as the brain of the IoT device, handling data processing, communication with other devices or the cloud, and executing programmed tasks.

  DHT22 Module
  - The DHT22 is a digital temperature and humidity sensor module.
  - It consists of a humidity sensor, a temperature sensor, and a microcontroller that converts the analog sensor readings into digital signals.

  The necessary code has been developed to implement on the ESP32 for sending temperature and humidity readings via MQTT protocol every 60 seconds.
  The ESP32 requires WiFi's SSID and password of the network to connect, as well as user credentials and password if required by the MQTT server where it publishes messages.
  The message and topic follow the UltraLight protocol [21] to ensure interpretation according to the semantics used in the platform.
  Initially, a verification process of the DHT22 modules was conducted. For this purpose, all 3 modules were placed together, and readings of their temperature and humidity parameters were registered. The position of the sensors during the verification process is depicted in Figure 4a, while Figure 4b illustrates the recorded values.

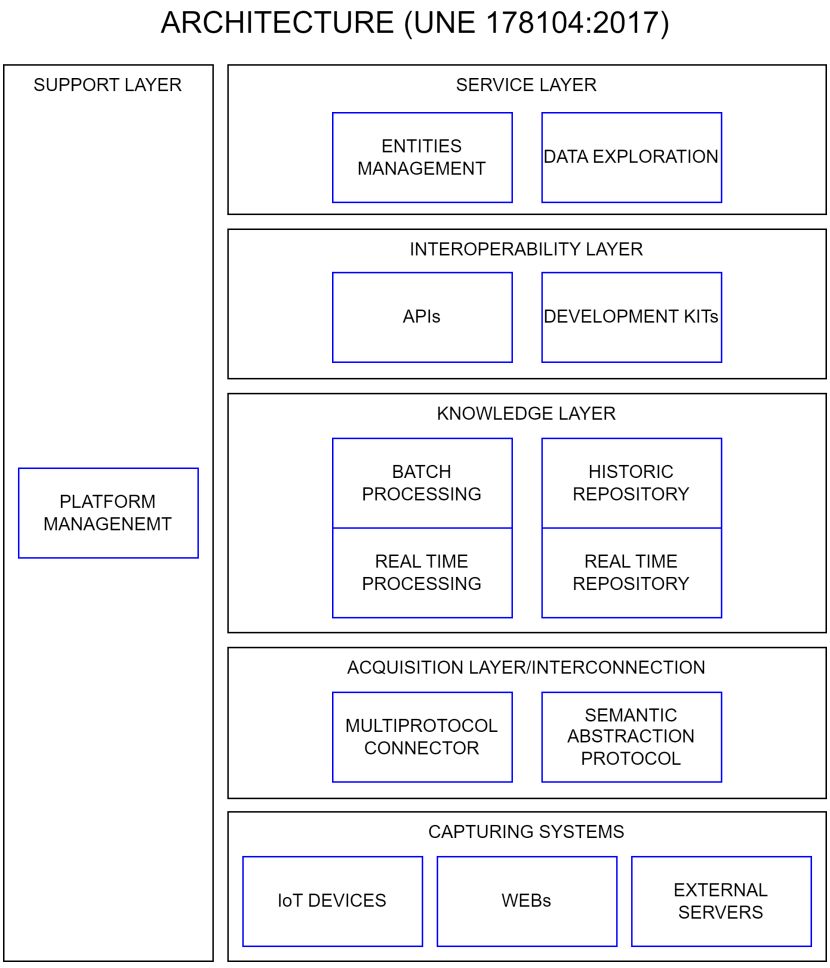


Fig. 2: Architecture according to UNE 178104:2017 Layer Model

- **Web scrapping**: This technique is used to fetch weather data from a station that publishes its records on the web. For web scraping, Python Beautiful Soup library [22] has been employed to extract data published by the weather station. Once extracted, the data is adapted to the semantics of the platform and sent to the broker using HTTP protocol.

- **API**: This last source employs an existing API [23] to fetch data from AIRZONE cloud [24], where the thermostats from the same provider upload the data.
  To connect to the API, a token should be used. The first time credentials are sent (user and pass for the AIRZONE cloud) and the server answer with two tokens, one is valid for one day and the second one is used to renew the first one. Therefore, the credentials need to be sent to the server only during the initial setup and not subsequently. The fetched data from the API is sent to broker using HTTP protocol and adapting the data to the semantics of the platform.

2) Acquisition layer/interconnection: An IoT Agent [25] is component that lets a group of devices send their data to and be managed from a Context Broker using their own native protocols. In the FIWARE catalogue [26] there are several available:

- IoT Agent for JSON
- IoT Agent for LWM2M
- IoT Agent for Ultralight
- IoT Agent for LoRaWAN
- IoT Agent for Sigfox
- IoT Agent for ISOXML
- IoT Agent for OPC-UA

There also another component in the catalogue, IoT Agent library, that allows to develop your own IoT Agent just in case that available ones do not fulfill the required needs. Considering the sources explained previously, only the IoT Devices needs the use of an IoT Agent (web scrapping and API directly send the data to the context broker). The Figure 5a shows the generic Northbound traffic (measurements) flow from an IoT Device to the context broker.

The ESP32 sents the data using MQTT transport protocol, so IoT Agent for JSON and IoT for Ultralight are possible

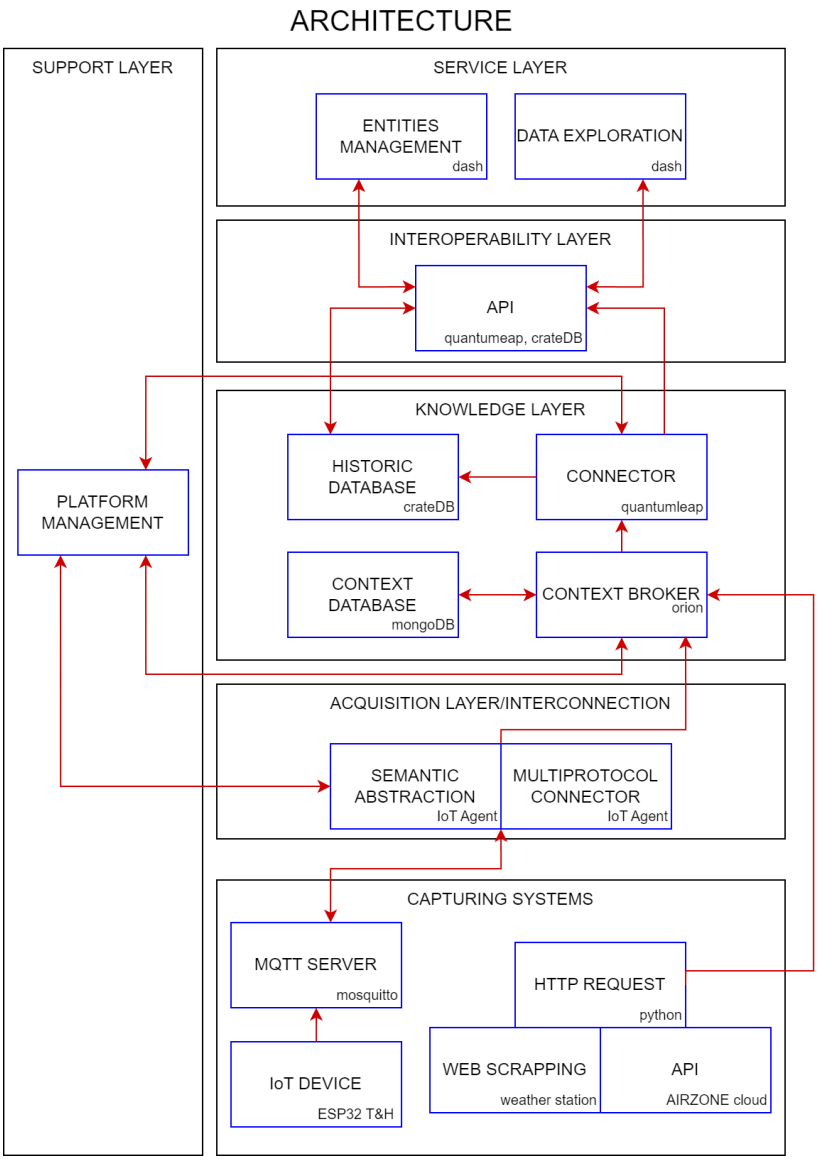


Fig. 3: Proposed architecture

candidates, but considering that IoT devices should send small quantity of data, the IoT Agent for Ultralight 2.0 is chosen.

Ultralight 2.0 is a lightweight text based protocol aimed to constrained devices and communications where the bandwidth and device memory may be limited resources.

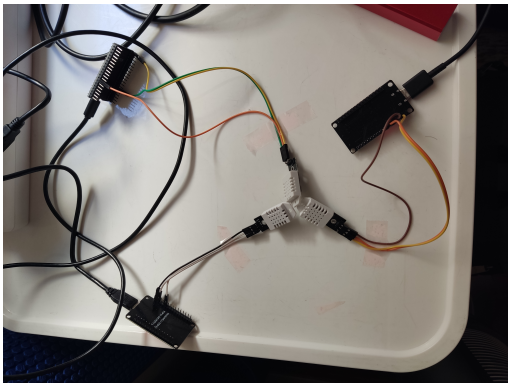
This IoT Agent works as bridge between Ultralight and the NGSIv2 interface [27] of a context broker, supporting MQTT as transport protocol for the device side. MQTT utilizes a publish-subscribe model, pushing event-driven messages to clients through a central broker. Clients publish messages with associated topics, serving as routing information. Subscribing clients receive messages based on matching topics, enabling scalable and decoupled communication without direct client interactions. The Figure 5b shows an example of a setup.

The chosen option for the MQTT broker is Mosquitto [28], an open-source MQTT broker. Its lightweight and efficient design makes it suitable for resource-constrained environments, common in IoT applications. Cross-platform support allows deployment on Linux, Windows, and macOS, enhancing its versatility.

With scalability as a key feature, Mosquitto handles a large number of clients and messages, ensuring seamless communication. Known for its reliability and stability, Mosquitto undergoes rigorous testing and maintenance for robust performance in production environments.

Security features, including SSL/TLS encryption and authentication mechanisms, make Mosquitto ideal for securing communication channels in IoT applications. Message retention allows clients to retrieve messages sent during temporary disconnections.

Configurability is a strength, enabling administrators to tailor settings to specific application requirements. The active

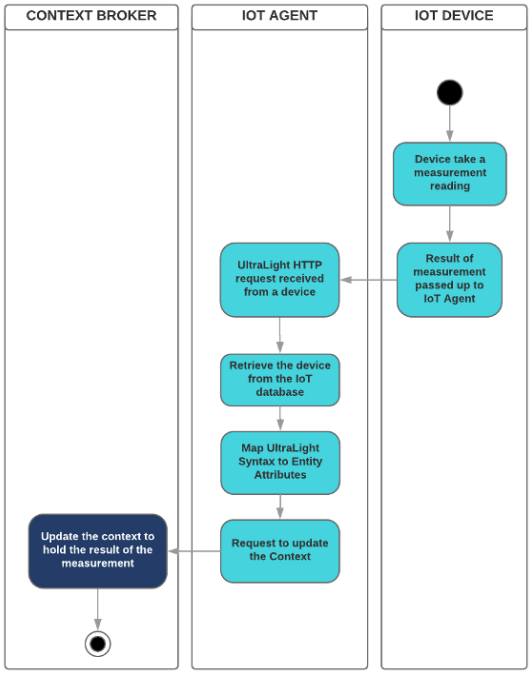


(a) DHT22 calibration - Position of Sensors

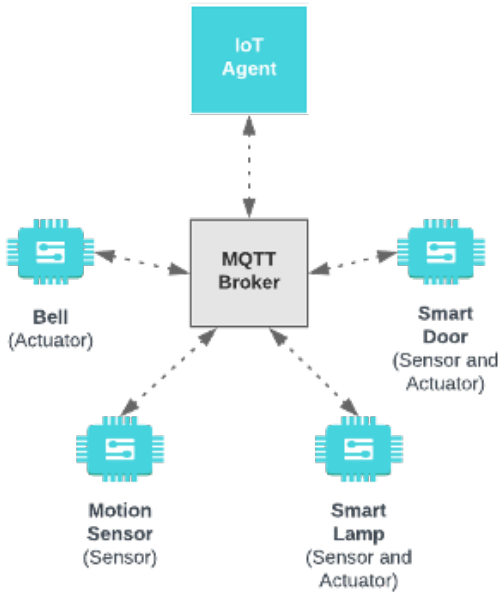
```
30p-a842e35625c8/attrs t|18.20#h|60.50
30p-e86beac48688/attrs t|18.10#h|56.00
38p-08b61f8b11e0/attrs t|17.90#h|60.30
30p-a842e35625c8/attrs t|18.30#h|61.10
30p-e86beac48688/attrs t|18.10#h|56.60
38p-08b61f8b11e0/attrs t|18.00#h|61.00
30p-a842e35625c8/attrs t|18.30#h|61.90
30p-e86beac48688/attrs t|18.20#h|57.80
38p-08b61f8b11e0/attrs t|18.00#h|61.90
30p-a842e35625c8/attrs t|18.30#h|62.20
30p-e86beac48688/attrs t|18.10#h|57.30
38p-08b61f8b11e0/attrs t|18.00#h|59.70
30p-a842e35625c8/attrs t|18.20#h|60.70
30p-e86beac48688/attrs t|18.10#h|56.20
38p-08b61f8b11e0/attrs t|18.00#h|63.00
30p-a842e35625c8/attrs t|18.20#h|63.90
30p-e86beac48688/attrs t|18.10#h|58.80
38p-08b61f8b11e0/attrs t|17.90#h|61.60
30p-a842e35625c8/attrs t|18.20#h|63.00
30p-e86beac48688/attrs t|18.10#h|58.10
38p-08b61f8b11e0/attrs t|17.90#h|62.90
30p-a842e35625c8/attrs t|18.20#h|64.00
30p-e86beac48688/attrs t|18.10#h|59.20
38p-08b61f8b11e0/attrs t|17.90#h|63.40
30p-a842e35625c8/attrs t|18.20#h|64.50
30p-e86beac48688/attrs t|18.00#h|59.80
38p-08b61f8b11e0/attrs t|17.90#h|63.30
30p-a842e35625c8/attrs t|18.20#h|63.90
30p-e86beac48688/attrs t|18.00#h|59.50
```

(b) DHT22 calibration - Recorded Values

Fig. 4: DHT22 Calibration Process



(a) IoT Agent Northbound flow



(b) MQTT setup

Fig. 5: IoT Agent / MQTT

community support around Mosquitto provides users with resources for learning, troubleshooting, and optimization.

In summary, Mosquitto's open-source nature, lightweight design, cross-platform compatibility, scalability, reliability, security features, message retention, configurability, and community support make it a well-suited and popular choice for a variety of IoT and messaging applications.

For the ESP32 equipped with temperature and humidity module, an example of a message to publish is "/888888/30p-a842e35625c8/attrs t|19.80#h|67.00" where "888888" is the apikey to identify a group of devices, "30p-a842e35625c8" is the device ID for the IoT Agent, "attrs" is a key to be identified as a measurement by the IoT Agent and "t|19.80#h|67.00" contains the key-value pairs for temperature and humidity separated by the hashtag symbol.

3) Knowledge layer: This layer is the core of the IoT platform. It mainly handles:

- Access to all information both past records and in real time.
- Movement of data received from the acquisition layer, between the different functions of the knowledge layer for storage, processing and recovery, and towards the interoperability layer. Data in this layer has already been extracted from the source devices and should be processed following standard data models. Support for real time processing of data received from the acquisition layer through modules with CEP engines, rule engines...
- Batch handling support for data received from ETL and Machine Learning.

a) CONTEXT BROKER: FIWARE includes 4 different context brokers: 1) Orion [29], 2) Orion-LD [30], 3) Scorpio [31] and 4) Stellio[32]. These context brokers rely on the publish-subscribe pattern and the model data aligned with the NGSI standards (NGSI-v2 or NGSI-LD [33]) published by the European Telecommunications Standards Institute (ETSI).

The answer of this stackoverflow thread explains the differences between the available brokers in FIWARE:

The FIWARE Catalogue currently contains four context brokers:

- *Orion - NGSI-v2 only*
- *Orion-LD - mixed NGSI-LD and NGSI-v2 support*
- *Scorpio - NGSI-LD only*
- *Stellio - NGSI-LD only*

Orion-LD is a fork from the original Orion repository and aims to be merged back at some point. It is the only context broker which can service both NGSI-v2 and NGSI-LD.
Scorpio and Stellio are pure NGSI-LD brokers which don't require the compromises of having to serve both syntaxes. Scorpio positions itself as the heavyweight broker, with a strong interest in federations. Stellio is somewhere in the middle between Scorpio and Orion.

According to the Stars and Forks of the repositories and with the answers of this stackoverflow thread, Scorpio and Stellio are the more inmature options. So these two were discarded.

Considering what the users answered in the stackoverflow thread:

NGSI-v2 standard perfectly covers whichever use case you may need
Currently, we're also using NGSIv2 and, so far, we haven't encountered any issues with it

Finally, **Orion context broker** was chosen as the broker of the IoT platform and it is depicted below:

Orion is a C++ implementation of the NGSIv2 REST API binding developed as a part of the FIWARE platform.
Orion Context Broker allows you to manage the entire lifecycle of context information including updates, queries, registrations and subscriptions. It is an NGSIv2 server implementation to manage context information and its availability. Using the Orion Context Broker, you are able to create context elements and manage them through updates and queries. In addition, you can subscribe to context information so when some condition occurs (e.g. the context elements have changed) you receive a notification.

The Orion NGSI API is fully compatible with the original NGSIv2 specification although some minor differences are described in an annex at the end of its documentation.

The FIWARE NGSI (Next Generation Service Interface) API defines:

- a data model for context information, based on a simple information model using the notion of context entities
- a context data interface for exchanging information by means of query, subscription, and update operations

The main elements in the NGSI data model are context entities, attributes and metadata, as shown in the Figure 6.

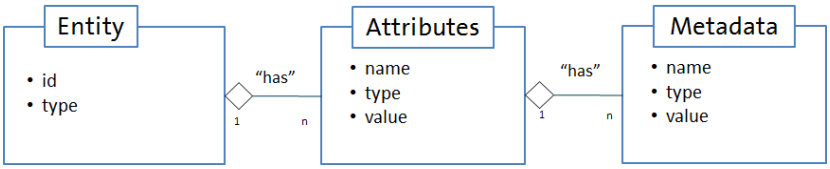


Fig. 6: NGSI data model

Context Entities

Context entities, or simply entities, are the center of gravity in Orion NGSIv2 based information model. An entity represents a thing, i.e., any physical or logical object (e.g., a sensor, a person, a room, an issue in a ticketing system, etc.). Each entity has an entity id.

Furthermore, the type system of Orion NGSIv2 based information model enables entities to have an entity type. Entity types are semantic types; they are intended to describe the type of thing represented by the entity. For example, a context entity with id sensor-365 could have the type temperatureSensor.

Orion supports hierarchical scopes, so entities can be assigned to a scope at creation time. Then, query and subscription can be also scoped to locate entities in the corresponding scopes.

Each entity, for a service path, is uniquely identified by the combination of its id and type.

Context Attributes

Context attributes are properties of context entities. For example, the current speed of a car could be modeled as attribute current_speed of entity car-104.

In the NGSI data model, attributes have an attribute name, an attribute type, an attribute value and metadata. The attribute name describes what kind of property the attribute value represents of the entity, for example current_speed. The attribute type

represents the NGSI value type of the attribute value. Note that FIWARE NGSI has its own type system for attribute values, so NGSI value types are not the same as JSON types. The attribute value finally contains the actual data.

Context Metadata

Context metadata is used in FIWARE NGSI in several places, one of them being an optional part of the attribute value as described above. Similar to attributes, each piece of metadata has: * a metadata name, describing the role of the metadata in the place where it occurs; for example, the metadata name accuracy indicates that the metadata value describes how accurate a given attribute value is * a metadata type, describing the NGSI value type of the metadata value * a metadata value containing the actual metadata

Note that in NGSI it is not foreseen that metadata may contain nested metadata.

b) CONTEXT DATABASE: This database system is employed for storing current state and time-aggregated data of entities within the platform. A document-oriented structure is well-suited for handling unstructured data common in IoT applications, providing flexibility and scalability.

MongoDB [34] is required to run either in the same host where Orion Context Broker is to be installed or in a different host accessible through the network. The recommended MongoDB version is 6.0 (according to the date when this document is being writen).

Orion Context Broker uses different collections to organize its context database. The main collections are Entities and Subscriptions. Below is a summary of both collections:

Entities Collection

The entities collection stores information about NGSI entities. Each document in the collection corresponds to an entity. These are the most important fields:

- `_id`: stores the EntityID, including the ID itself and type. The JSON document for this field includes:
  - `id`: entity NGSI ID
  - `type`: entity NGSI type
  - `servicePath`: related to the service path functionality.
- `attrs` is a keymap of different attributes created for that entity. Each element in the map has the following information:
  - `type`: the attribute type
  - `value`: the attribute value
  - `md` (optional): custom metadata, a keymap of metadata objects.
  - `mdNames`: an array of strings, names of metadata of the attribute.
  - `creDate`: timestamp for attribute creation.
  - `modDate`: timestamp for the last attribute update.
- `attrNames`: an array of strings, names of the attributes of the entity.
- `creDate`: timestamp for entity creation date.
- `modDate`: timestamp for the last entity update. It matches `creDate` if the entity has not been modified after creation.
- `location` (optional): geographic location of the entity, composed of the following fields:
  - `attrName`: the attribute name identifying the geographic location in the `attrs` array.
  - `coords`: a GeoJSON representing the location of the entity.

c) CONNECTOR AND HISTORIC DATABASE: A suited database should be employed for persisting historical data from the broker. A distributed database architecture ensures reliable storage and retrieval of time-series data, essential for analyzing trends and patterns over time.

FIWARE catalogue has a suitable service to persist the historic of the context data, QuantumLeap [35]. It's a REST service for storing, querying and retrieving NGSI v2 spatial-temporal data. It converts the context data into tabular format and stores it in a time-series database, associating each database record with a time index and, if present in the NGSI data, a location on Earth. The Figure 7 shows the connection between QuantumLeap and Orion Context Broker, considering also different possible services associated.

QuantumLeap supports both CrateDB [36] and Timescale [37] as back end databases. Considering CrateDB is the default option, it is easy to scale thanks to its shared-nothing architecture and well suited for containerisation, CrateDB is the selected option as data persistence layer.

4) Interoperability layer: The interoperability layer facilitates the provision of services, facilitating them the access and query of the information by publishing of APIs that can be used from the smart service layer. For the IoT platform two APIs are considered:

- QuantumLeap API: Link to QuantumLeap API documentation
- CrateDB API: Link to CrateDB python client implementing DB API

5) Service layer: The IoT platform has two main services: Entities management and Data exploration. This services are associated with the development of user interfaces which allow the users to interact with the platform in a friendly manner.

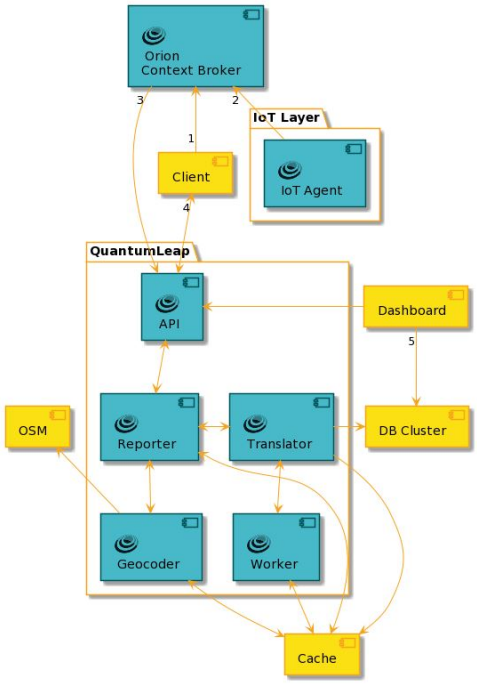


Fig. 7: QuantumLeap connections

These interfaces are developed using Dash [38] which is a low-code framework for building data apps in Python. It is built on top of Plotly.js, React and Flask, allowing to create dynamic elements including dropdowns, sliders and graphs directly using Python language.

Below are presented and described the main functionalities of the different pages developed for the user interface of the IoT platform (the figures are located in the Screenshots section VIII:

- **USER CREDENTIALS:** The app is accessed by entering credentials (user-password pair showed in Figure 11). This credentials are checked with an user credential store.
- **HOME:** Figure 12 is the main page when the app validates the entered credentials by the user. It presents a Plotly map displaying the entities in the ORION broker based on their locations. Entities with no location are not presented. It is possible to zoom in and zoom out dynamically. If some entities are located very close, they are grouped until the zoom allows to differentiate correctly.
- **FIWARE ENTITIES**
  - **ENTITIES LIST:** Displays a table with all entities in the ORION broker as shown in Figure 13a. Clicking on an entity redirects to a page showing its current context data (Figure 13b). It is possible to Delete the entity by using the corresponding button, DELETE ENTITY. A text box shows if the deletion was successful or not.
  - **CREATE ENTITY:** This page, showed in Figure 14, enables users to create a new entity, specifying minimum attributes like id, type, and location. Additional attributes (text or numeric) can be added using the ”ADD ATTRIBUTE” button. The ”CREATE ENTITY” button finalizes the entity creation in the ORION broker. Two text boxes show if the creation of the entity and the subscription to the historic database were successful or not.
  - **UPDATE ENTITY:** Allows users to select an existing entity and update its attributes (id and type cannot be updated). The ENABLE/DISABLE MODIFICATION allows the user to modify or not the attributes to be updated. The user has a button SEND MODIFICATION to confirm and send the update. Ther is a box which show if the update were successful or not. The figure 15 shows that page.
- **MQTT ENTITIES**
  - **SERVICE LIST:** Displays a table with all services defined in the IoT Agent as shown in Figure 16a. Clicking on an service redirects to a page showing its current context data (Figure 16b). It is possible to Delete the service by using the corresponding button, DELETE ENTITY. A text box shows if the deletion was successful or not.
  - **MQTT DEVICES LIST:** Displays a table with all MQTT devices defined in the IoT Agent as shown in Figure 17a. Clicking on an entity redirects to a page showing its current context data (Figure 17b). It is possible to Delete the device by using the corresponding button, DELETE ENTITY. A text box shows if the deletion was successful or not.
  - **CREATE T&H ESP32 DEVICE:** This page, showed in Figure 18, enables users to create a MQTT device for a ESP32 with temperature and humidity module specifying the required attributes:

    * `api_key`: Identifies the associated service for the device.
    * `device_id`: Identifies the device uniquely.
    * `WiFi SSID`: The name of the Wi-Fi network the device will connect to.
    * `WiFi PASS`: The password for the Wi-Fi network.
    * `MQTT Host`: The hostname or IP address of the MQTT broker.
    * `MQTT Port`: The port number on which the MQTT broker is running.
    * `MQTT User`: The username for MQTT authentication.
    * `MQTT Pass`: The password for MQTT authentication.
    * `entity_name`: Name of the FIWARE entity associated with the device.
    * `entity_type`: Type of the FIWARE entity associated with the device. This attribute is static and the user can not modify it.
    * `temperature`: Attribute representing temperature readings, represented with a "t" in the MQTT message. This attribute is static and the user can not modify it.
    * `humidity`: Attribute representing humidity readings, represented with a "h" in the MQTT message. This attribute is static and the user can not modify it.
    * `interval`: Time interval for sending MQTT messages or readings.

    The page also serves the functionality of download the user the ARDUINO CODE associated with the defined attributes to upload to the ESP32 board. A text box shows if the creation was successful or not.
  - **CREATE SERVICE:** This page, showed in Figure 19, enables users to create a MQTT service specifying the required attributes:
    * `service_name`: Tag the service.
    * `api_key`: Internal ID for the IoT Agent to associated linked devices to that service
    * `type`: Default type for the associated devices.
    * `attribute`: Users have the ability to create as many default attributes as they desire using the "ADD ATTRIBUTE" button. For each attribute, the user must specify the full name of the attribute, which will be defined for the FIWARE entity, and the short name of the attribute to be used in the MQTT message transmission. The user have to press the CREATE SERVICE button and a text box shows if the creation was successful or not.
  - **CREATE MQTT DEVICE:** This page, showed in Figure 20, enables users to create a generic MQTT device specifying the required attributes:
    * `api_key`: Identifies the associated service for the device.
    * `device_id`: Identifies the device uniquely.
    * `entity_name`: Name of the FIWARE entity associated with the device.
    * `entity_type`: Type of the FIWARE entity associated with the device. This attribute is static and the user can not modify it.
    * `attribute`: Users have the ability to create as many attributes as they desire using the "ADD ATTRIBUTE" button. For each attribute, the user must specify the full name of the attribute, which will be defined for the FIWARE entity, and the short name of the attribute to be used in the MQTT message transmission. The user have to press the CREATE MQTT DEVICE button and a text box shows if the creation was successful or not.
- ANALYTICS
  - **DATA:** This page, showed in Figure 21, enables users to select different devices and numeric attributes from an specific FIWARE device type to display the time-series in a table from the historic database. The user also have to select a period and a frequency of sampling (RAW, 5min, 15min, 1hour, 1 day) before pressing the UPDATE button to display the table. The datatable is paginated with a length of 100 registers and sorted by default by the time_index attribut but the user can change it by clicking in the desired attribute header to change the sorting method.
  - **GRAPH:** Figure 22 shows the graph page which has the functionality of creating dynamic chart that allows users to query the specific value of each represented point by placing the mouse pointer over the point of interest. Additionally, the chart enables specific zooming on a defined period and value range. The user also have to select a period and a frequency of sampling (5min, 15min, 1hour, 1 day) before pressing the UPDATE button to display the table

6) Support layer: This transversal layer is based on different functionalities of the used components across the developed IoT platform layers.

Authentication is present in the login of the dash app but also to access to the crateDB database. The MQTT broker has also authentication method blocking devices that has no credentials.

Orion context broker and its linked mongoDB serves as a centralized repository of the entities (FIWARE and MQTT devices) that it is finally managed using the devoloped user interface.

B. Dataset generation

During the development of the IoT platform, a provisional solution were deployed to register the available data from the thermostats and the identified weather station web. A Raspberry Pi [39], a small single-board computer with low consumption, is used to deploy the provisional solution.

1) AIRZONE thermostats: AIRZONE is a provider of automated systems to control mainly the temperature of each room within a centralized HVAC system. The available system is composed by centralized air heat pump that supplies hot/cold air to 4 different rooms. Each room has a wired thermostat and a motorized grille for the supply air side. The heat pump, the thermostats and the motorized grilles are connected to a controller board which allows to control the availability of the system in each room and the desired temperature.

The system also includes a WebServer that uses the WiFi to connecto to AIRZONE cloud server, allowing remote control of the system through an app or Web. The AIRZONE cloud server also has a Web API interface with documentation of the available RESTful requests.

Therefore, at the beginning of the project, a script was prepared using Python to request the thermostat temperature of each room in an interval of 1 minute. These temperature registers are stored in a sqlite database in the Raspberry Pi.

2) Weather Station: AEMET has an open API [40] to request weather data but it only has daily registers. A research on the internet was done to find if there is any open source to fetch weather data from a local weather station. Finally, a weather station was found publishing its data in a web [41] and refreshing the data with a frequency even lower than 15 minutes. This weather station belongs to an academic research group, GOFIMA (Grupo de Oceanografía Física de la universidad de MÁlaga). Its location is also close to the location of the thermostats.

Thus, a script was prepared using Beautiful Soup, a Python library to parse html documents to fetch the weather data published on the web every 1 minute. These weather data registers are stored in a sqlite database [42] in the Raspberry Pi.

C. Containerization and deployment

To ensure scalability and ease of deployment in different infrastructures, the main components are containerized for an easy deployement.

1) Containers: Next, the different used containers are presented and explained:

- ORION CONTEXT BROKER: The broker is part of the FIWARE catalogue, it is possible to build an image from the public repository in GitHub but the official Orion Context Broker image is used. It can be found in quay.io [43] or dockerhub [44] repositories.
- MONGODB: The Orion Context Broker is linked to a context database and only mongoDB is a possibility. The official mongo image from dockerhub repository is used.
- QUANTUMLEAP: This component is part of the FIWARE catalogue, it is possible to build an image from the public repository in GitHub but the official QuantumLeap image is used. It can be found in quay.io or dockerhub repositories.
- CRATEDB: The QuantumLeap component is linked to a historic database and only crateDB is the selected database. The official crateDB image from dockerhub repository is used.
- IOT AGENT ULTRALIGHT: This component is also from FIWARE catalogue and the image can be built using the GitHub repository. Official image, hosted in quay.io and dockerhub repositories, is used in the platform.
- MOSQUITTO: The MQTT used in the platform is mosquitto. It is possible to find the official image in dockerhub repository but for the proposed solution the mosquitto broker is deployed in the Raspberry and the broker is directly installed from raspbian repositories without depolying a service using docker.
- DASH APP: The dash app is containerized for an easy deployment. The source files are also in a GitHub repository.
- AIRZONE API AND WEATHER STATION WEBSCRAPPING SCRIPT: The AIRZONE and WEATHER STATION script are also containerized and the image is hosted in dockerhub repository.
- INITIAL DEPLOYMENT: In order to facilitate an easy and fast deployment, a container is built which executes the creation of the different sources, subscriptions and entites to start with the IoT platform considering the different sources.

2) Deployment: The previous sections listed all the single images but for the deployment it is still needed to setup the orchestration between the different services and the infrastructure where it can be deployed:

- Docker Compose: A docker compose file is developed to orchestrate all the needed services. This docker compose file has a .env file where all the environment variables are listed (docker compose access to this .env using `$(<env_variable>)` as access structure). This .env contains could contain private information such as secrets, private configurations, therefore, it is not recommended to share in a public repository. The whole docker compose can be review in source code document: 1 listed in the appendices section. Each service has these common features: service name, labels, image repository, hostname, container name, exposed and redirected ports, network, persistence volumes and healthcheck. Below are described some important aspects for each service:
  - **MongoDB:** This mongoDB service is the database for Orion Context Broker. A healthcheck is setup in order because when Orion Contex Broker starts, the mongoDB has to be ready. It has the required persistence volumes.

  - **Orion:** The Orion Context Broker as the main componente of the platform also has a healthcheck when starting the service. It depends on the mongoDB service to start.
  - **CrateDB:** This crateDB service is the historic database of the platform and the dash service and QuantumLeap service depends on it to start. A healthcheck is setup, veryfing the availability of the service. It has the required persistence volume.
  - **QuantumLeap:** This connector service to persist the historic data dependes on crate-db service to start listening.
  - **IoT-Agent UltraLight:** This service relies on the availability of the mongoDB database. It needs several environmental variables to map the context broker, mongoDB and MQTT host.
  - **Dash:** This is the user interface, it depends on the healthy of crateDB service. It also has all the required environmental variables to connect with Orion Context Broker, crateDB, IoT Agent and confidential information as secrets and tokens.

  The docker compose also includes the definition of the default network and the persistence volumes used by the databases:
  - networks
    - Default:
      - Subnet: 172.18.1.0/24
  - volumes
    - mongo-db_db
    - mongo-db_config
    - crate-db_db

- Infrastructure: Previous sections depicted the containerization philosophy adopted for the development of the services in the IoT platform. The users of the platform can decide to run the service in local servers but also to deploy them using public cloud, considering using Virtual Machine or even Container Services.
  This subsection describes an example of the deployment of the IoT platform in Azure, a public cloud infrastructure, using Virtual Machine service.
  System requirements
  The required resources for the deployment of the IoT platform depends on various factors, including the volume of data generated, the frequency of data updates, the number of concurrent users, and the specific services running on the VM. For this example, where the platform is not going to deal with a high number of concurrent users and devices, the selected virtual machine is a Standard B2s with this configuration:
  - Nº of cores: 2
  - RAM: 4 GB
  - Disk: 30 GB (SSD)
  - Operating system: Linux (Debian 11)
  - Network: A network security group is defined, defining 3 inbound port rules:
    - Port 22: Allow any custom to access this port to connect with the machine through SSH.
    - Port 1250: Allow any custom to access this port to interact with the user interface developed with dash for the platform.
    - Port 4200: Allow any custom to access this port to use the crateDB user interface.
  -

  Notice that for this example the mosquitto broker is installed directly in a Raspberry Pi and not in the Virtual Machine hosted in Azure. But this is a light service that has no impact in the requirements of the VM.
  System metrics
  Figure 8 shows the percentage of CPU used by the VM during last 24h. An average of only 8.3% of CPU usage demonstrates that the VM is enough to handle the developed IoT platform for this example.
  Related to the RAM memory, the Figure 9 shows that the VM only use an average of 1.1GB and it only has a peak of 3.1GB when the VM was rebooted.

Fig. 8: Azure VM CPU average usage

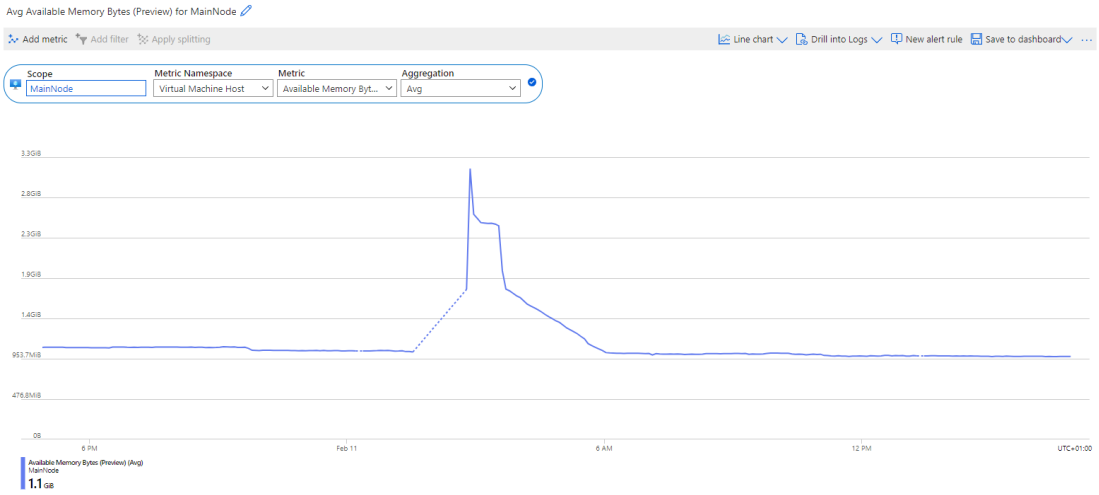


Fig. 9: Azure VM RAM average usage

V. Conclusions

Figure 10 shows a generic implementation architecture of the developed IoT platform.

The comprehensive development of the IoT platform, structured around a multi-layered architecture aligned with the standard UNE 178104:2017 already presented in Figure 3, successfully demonstrates that it is a robust, scalable, and flexible solution for IoT device management and data processing. Each layer of the architecture played a pivotal role in achieving a harmonious integration of devices, data flow, analysis, and user interaction.

A. Key features

1) Capturing Systems: The deployment of various sources, including IoT devices, web scraping, and API integrations, has underscored the platform's versatility in data acquisition. This diversity in data sources enriches the platform's dataset, providing a comprehensive view of the monitored environment.

2) Acquisition Layer/Interconnection: The implementation of the IoT Agent for Ultralight 2.0, coupled with the Mosquitto MQTT broker, facilitated seamless data transmission and protocol translation. This layer proved crucial in bridging the gap between diverse device protocols and the standardized NGSI model, enhancing data homogeneity and interoperability.

3) Knowledge Layer: The Orion Context Broker and MongoDB duo emerged as the backbone of the platform, orchestrating the context management with efficacy. Their ability to handle real-time with precision and speed has been instrumental in ensuring data integrity and accessibility. The QuantumLeap connector ensures the persistence of the historic data in the crateDB database.

4) Interoperability Layer: The QuantumLeap API and CrateDB's inclusion significantly bolstered the platform's data analytics and visualization capabilities. This layer's contribution to providing actionable insights through historical data analysis cannot be overstated, driving informed decision-making.

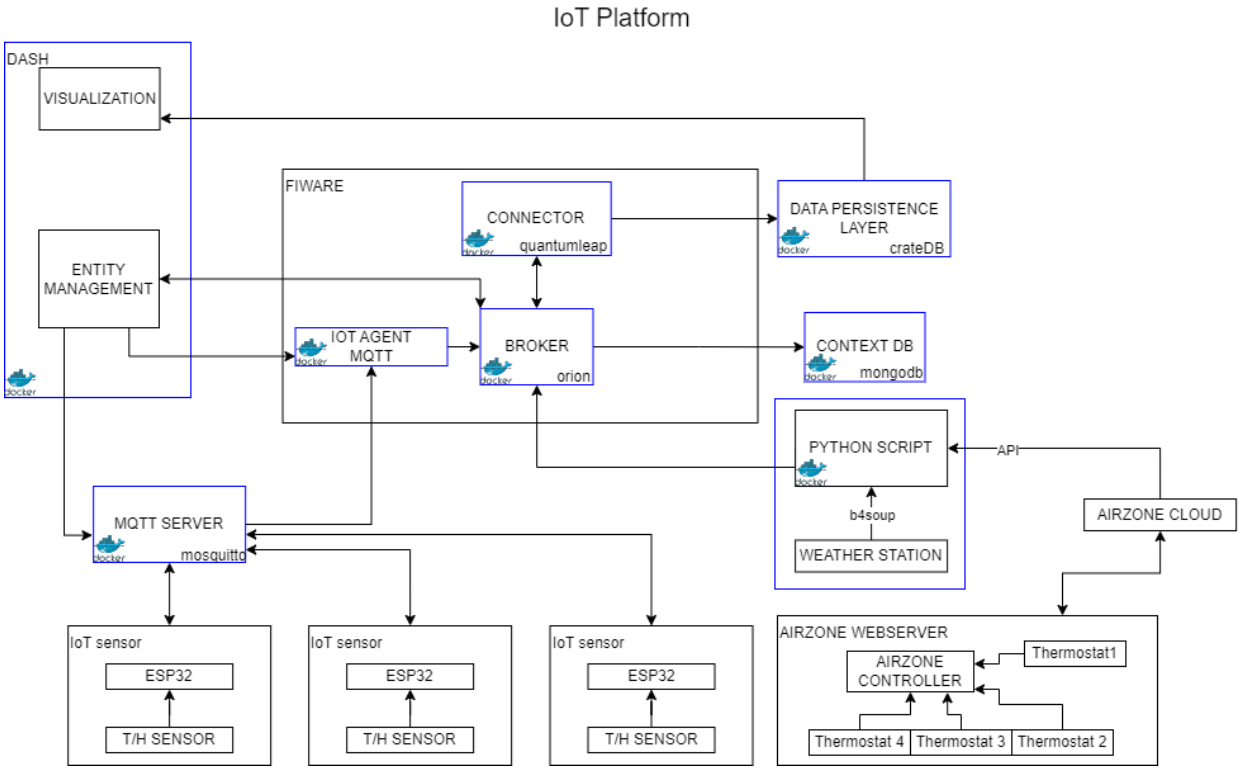


Fig. 10: Generic IoT platform architecture

5) Service Layer: The developed Dash-based interface has greatly enhanced the platform's usability. By offering intuitive navigation and dynamic data visualization, making complex data sets accessible and understandable to users, regardless of their technical background.

6) Support Layer: The cross-layer support functionalities, including security, authentication, and system monitoring, have established a solid foundation for platform reliability and user trust. These mechanisms ensure the platform's resilience against cyber threats and operational anomalies.

B. Proposals for Future Improvements

Looking ahead, platforplatform's capabilities by integrating advanced machine learning algorithms for predictive analytics and enhancing device compatibility. Our commitment to adhering to open standards and fostering community collaboration will continue to guide our evolution, ensuring the platform remains at the forefront of IoT innovation.

The platform has the foundations for the expansion of its features and offered services. Below is a list of some of them that have already been identified during the development of the current version and could be considered for future platform updates:

- Capturing systems:
  - Integration of diverse data sources: Improvements in the Capturing Systems layer include the integration of new information sources. This involves extending compatibility to IoT devices with different communication protocols and establishing connections with open data platforms and public APIs, such as AEMET for weather data and ESIOS for querying electrical market information. This expanded data integration will enrich the platform's capabilities and provide users with a more comprehensive and diverse set of information for analysis.
- Acquisition layer/interconnection:
  - Increased IoT agents: Enhancements in the Acquisition Layer involve the introduction of additional IoT agents to efficiently gather data from the new sources. These agents will play a crucial role in adapting the incoming data to the semantic framework utilized by the platform's broker. This expansion ensures a seamless integration of diverse data types, maintaining consistency in the platform's semantic representation and enhancing its overall data processing capabilities.
- Service layer:
  - Machine learning: With the platform already effectively managing data from the knowledge layer, the next step to consider would be the implementation of new services such as anomaly detection in time-series.

  - Improved data exploration: Currently, the data exploration service allows the selection of multiple devices of the same entity type. However, it would be interesting for the user, starting from the selection of an attribute, to be able to choose all the devices they want to visually check the values. Additionally, the utility of comparing data series of the same length but starting at different periods is also being considered.
- Support layer:
  - Robust authentication procedures: Enhancements in the support layer involve the implementation of strong authentication procedures such as OAuth and LDAP. This will bolster the platform's security measures, ensuring a more secure and reliable user authentication process.

VI. Source Code

Listing 1: docker-compose.yml

```yaml
version: "3.8"
services:
  orion:
    labels:
      org.fiware: 'deployment'
    image: quay.io/fiware/orion:${ORION_VERSION}
    hostname: ${ORION_HOST}
    container_name: fiware-orion
    depends_on:
      - mongo-db
    networks:
      - default
    expose:
      - "${ORION_PORT}"
    ports:
      - "${ORION_EXT_PORT}:${ORION_PORT}"
    command: -dbhost mongo-db -logLevel DEBUG
    healthcheck:
      test: curl --fail -s http://${ORION_HOST}:${ORION_PORT}/version || exit 1
      interval: 5s

  iot-agent:
    labels:
      org.fiware: 'deployment'
    image: quay.io/fiware/iotagent-ul:${ULTRALIGHT_VERSION}
    hostname: ${IOTA_HOST}
    container_name: fiware-iot-agent
    depends_on:
      - mongo-db
    networks:
      - default
    expose:
      - "${IOTA_NORTH_PORT}"
    ports:
      - "${IOTA_EXT_NORTH_PORT}:${IOTA_NORTH_PORT}" # localhost:4041
    environment:
      - IOTA_CB_HOST=${ORION_HOST} # name of the context broker to update context
      - IOTA_CB_PORT=${ORION_PORT} # port the context broker listens on to update context
      - IOTA_NORTH_PORT=${IOTA_NORTH_PORT}
      - IOTA_REGISTRY_TYPE=mongodb #Whether to hold IoT device info in memory or in a database
      - IOTA_LOG_LEVEL=DEBUG # The log level of the IoT Agent
      - IOTA_TIMESTAMP=true # Supply timestamp information with each measurement
      - IOTA_CB_NGSI_VERSION=v2 # use NGSIv2 when sending updates for active attributes
      - IOTA_AUTOCAST=true # Ensure Ultralight number values are read as numbers not strings
      - IOTA_MONGO_HOST=${MONGO_DB_HOST} # The host name of MongoDB
      - IOTA_MONGO_PORT=${MONGO_DB_PORT} # The port mongoDB is listening on
      - IOTA_MONGO_DB=iotagentul # The name of the database used in mongoDB
      - IOTA_MQTT_HOST=${MQTT_SERVER_HOST} # The host name of the MQTT Broker
      - IOTA_MQTT_PORT=${MQTT_PORT} # The port the MQTT Broker is listening on to receive topics
      - IOTA_MQTT_USERNAME=${MQTT_USERNAME}
      - IOTA_MQTT_PASSWORD=${MQTT_PASSWORD}
      - IOTA_DEFAULT_RESOURCE= # Default is blank. I'm using MQTT so I don't need a resource
      - IOTA_PROVIDER_URL=http://${IOTA_HOST}:${IOTA_NORTH_PORT}
      - IOTA_DEFAULT_TRANSPORT=MQTT
    healthcheck:
      interval: 5s

  # Database
  mongo-db:
    labels:
      org.fiware: 'deployment'
    image: mongo:${MONGO_DB_VERSION}
    hostname: ${MONGO_DB_HOST}
    container_name: fiware-db-mongo
    expose:
      - "${MONGO_DB_PORT}"
    ports:
      - "${MONGO_DB_EXT_PORT}:${MONGO_DB_PORT}" # localhost:27017
    networks:
      - default
    volumes:
      - mongo-db_db:/data/db
      - mongo-db_config:/data/configdb
```

```
74      healthcheck:
75        test: |
76          host=`hostname --ip-address || echo '127.0.0.1'`;
77          mongo --quiet $host/test --eval 'quit(db.runCommand({ ping: 1 }).ok ? 0 : 2)' && echo 0 || echo 1
78        interval: 5s
79
80    quantumleap:
81      labels:
82        org.fiware: 'deployment'
83      image: orchestracities/quantumleap:${QUANTUMLEAP_VERSION}
84      hostname: ${QUANTUMLEAP_HOST}
85      container_name: fiware-quantumleap
86      ports:
87          - "${QUANTUMLEAP_EXT_PORT}:${QUANTUMLEAP_PORT}"
88      depends_on:
89          - crate-db
90      environment:
91          - CRATE_HOST=${CRATE_HOST}
92          - LOGLEVEL=DEBUG
93      networks:
94        default:
95
96    crate-db:
97      labels:
98        org.fiware: 'deployment'
99      image: crate:${CRATE_VERSION}
100     hostname: ${CRATE_HOST}
101     container_name: fiware-cratedb
102     ports:
103         # Admin UI
104         - "${CRATE_EXT_PORT}:${CRATE_PORT}"
105         # Transport protocol
106         - "${CRATE_EXT_TRANSPORT_PORT}:${CRATE_TRANSPORT_PORT}"
107     command:
108         crate -Cdiscovery.type=single-node
109     environment:
110         - CRATE_HEAP_SIZE=2g
111     networks:
112       default:
113     volumes:
114         - crate-db_db:/data
115     healthcheck:
116       test: ["CMD", "curl", "--fail", "-s", "http://${CRATE_HOST}:${CRATE_EXT_PORT}/"]
117       interval: 2s
118       timeout: 1s
119       retries: 10
120
121   # Dash app
122   dash:
123     labels:
124       org.fiware: 'deployment'
125     image: igomisp/iot-gui:0.2.1
126     hostname: ${GUI_HOST}
127     container_name: fiware-dash
128     depends_on:
129       crate-db:
130         condition: service_healthy
131     environment:
132         - ORION_HOST=${ORION_HOST}
133         - CRATE_HOST=${CRATE_HOST}
134         - QUANTUMLEAP_HOST=${QUANTUMLEAP_HOST}
135         - QUANTUMLEAP_EXT_PORT=${QUANTUMLEAP_EXT_PORT}
136         - IOTA_HOST=${IOTA_HOST}
137         - IOTA_EXT_NORTH_PORT=${IOTA_EXT_NORTH_PORT}
138         - DASH_PORT=${DASH_PORT}
139         - MAPBOX_TOKEN=${MAPBOX_TOKEN}
140         - DASH_USER=${DASH_USER}
141         - DASH_PASS=${DASH_PASS}
142     expose:
143       - "${DASH_PORT}"
144     ports:
145       - "${DASH_EXT_PORT}:${DASH_PORT}"
146     networks:
147       default:
148
149 networks:
150   default:
```

```
    labels:
      org.fiware: 'deployment'
    ipam:
      config:
        - subnet: 172.18.1.0/24

volumes:
  mongo-db_db: ~
  mongo-db_config: ~
  crate-db_db: ~
```

VII. Bibliography

References

[1] Amazon web services (aws). Accessed on February 11, 2024. [Online]. Available: https://aws.amazon.com/
[2] Google cloud platform (gcp). Accessed on February 11, 2024. [Online]. Available: https://cloud.google.com/gcp/
[3] Microsoft azure. Accessed on February 11, 2024. [Online]. Available: https://azure.microsoft.com/es-es/
[4] Amazon web services (aws) iot. Accessed on February 11, 2024. [Online]. Available: https://aws.amazon.com/iot/
[5] Azure iot hub. Accessed on February 11, 2024. [Online]. Available: https://azure.microsoft.com/en-us/products/iot-hub/
[6] Thingspeak. Accessed on February 11, 2024. [Online]. Available: https://thingspeak.com/
[7] Thingsboard. Accessed on February 11, 2024. [Online]. Available: https://thingsboard.io/
[8] Particle. Accessed on February 11, 2024. [Online]. Available: https://www.particle.io/
[9] Mqtt protocol. Accessed on February 11, 2024. [Online]. Available: https://mqtt.org/
[10] Coap protocol. Accessed on February 11, 2024. [Online]. Available: https://datatracker.ietf.org/doc/html/rfc7252
[11] Http protocol. Accessed on February 11, 2024. [Online]. Available: https://datatracker.ietf.org/doc/html/rfc7540
[12] Fiware. Accessed on February 11, 2024. [Online]. Available: https://www.fiware.org/
[13] Eclipse iot. Accessed on February 11, 2024. [Online]. Available: https://iot.eclipse.org/
[14] Node-red. Accessed on February 11, 2024. [Online]. Available: https://nodered.org/
[15] Iotivity. Accessed on February 11, 2024. [Online]. Available: https://iotivity.org/
[16] Openstack. Accessed on February 11, 2024. [Online]. Available: https://www.openstack.org/
[17] Sentilo. Accessed on February 11, 2024. [Online]. Available: https://www.sentilo.io/
[18] Une 178104:2017. Accessed on February 11, 2024. [Online]. Available: https://www.une.org/encuentra-tu-norma/busca-tu-norma/norma?c=N0059471
[19] Esp32. Accessed on February 11, 2024. [Online]. Available: https://www.espressif.com/en/products/socs/esp32
[20] Dht22 datasheet. Accessed on February 11, 2024. [Online]. Available: https://www.mouser.com/datasheet/2/737/dht-932870.pdf
[21] Ultralight 2.0 protocol. Accessed on February 11, 2024. [Online]. Available: https://fiware-iotagent-ul.readthedocs.io/en/latest/
[22] Beautiful soup library. Accessed on February 11, 2024. [Online]. Available: https://www.crummy.com/software/BeautifulSoup/bs4/doc/
[23] Airzonewebapi. Accessed on February 11, 2024. [Online]. Available: https://developers.airzonecloud.com/docs/web-api/#servers
[24] Airzone cloud. Accessed on February 11, 2024. [Online]. Available: https://www.airzone.es/airzone-cloud/
[25] Fiware iot agent. Accessed on February 11, 2024. [Online]. Available: https://github.com/FIWARE/tutorials.IoT-Agent
[26] Fiware catalogue. Accessed on February 11, 2024. [Online]. Available: https://www.fiware.org/catalogue/
[27] Ngsiv2 specification. Accessed on February 11, 2024. [Online]. Available: https://fiware.github.io/specifications/ngsiv2/stable/
[28] Mosquitto. Accessed on February 11, 2024. [Online]. Available: https://mosquitto.org/
[29] Fiware orion. Accessed on February 11, 2024. [Online]. Available: https://fiware-orion.readthedocs.io/en/master/
[30] Fiware orion-ld. Accessed on February 11, 2024. [Online]. Available: https://github.com/FIWARE/context.Orion-LD
[31] Fiware scorpio. Accessed on February 11, 2024. [Online]. Available: https://scorpio.readthedocs.io/en/latest/
[32] Fiware stellio. Accessed on February 11, 2024. [Online]. Available: https://stellio.readthedocs.io/en/latest/
[33] Ngsi-ld specification. Accessed on February 11, 2024. [Online]. Available: https://www.etsi.org/deliver/etsi_gs/CIM/001_099/006/01.02.01_60/gs_CIM006v010201p.pdf
[34] Mongodb. Accessed on February 11, 2024. [Online]. Available: https://www.mongodb.com/
[35] Fiware quantumleap. Accessed on February 11, 2024. [Online]. Available: https://quantumleap.readthedocs.io/en/latest/
[36] Cratedb. Accessed on February 11, 2024. [Online]. Available: https://cratedb.com/
[37] Timescale. Accessed on February 11, 2024. [Online]. Available: https://www.timescale.com/
[38] Dash. Accessed on February 11, 2024. [Online]. Available: https://dash.plotly.com/
[39] Raspberry pi. Accessed on February 11, 2024. [Online]. Available: https://www.raspberrypi.com/
[40] Aemet open data. Accessed on February 11, 2024. [Online]. Available: https://opendata.aemet.es/centrodedescargas/inicio
[41] Weather station. Accessed on February 11, 2024. [Online]. Available: http://150.214.57.165/meteo/Current_Vantage.htm
[42] Sqlite. Accessed on February 11, 2024. [Online]. Available: https://www.sqlite.org/index.html
[43] Quay repository. Accessed on February 11, 2024. [Online]. Available: https://quay.io/
[44] Docker hub. Accessed on February 11, 2024. [Online]. Available: https://hub.docker.com/

VIII. Screenshots

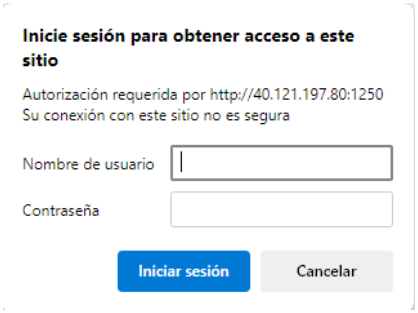


Fig. 11: App credentials

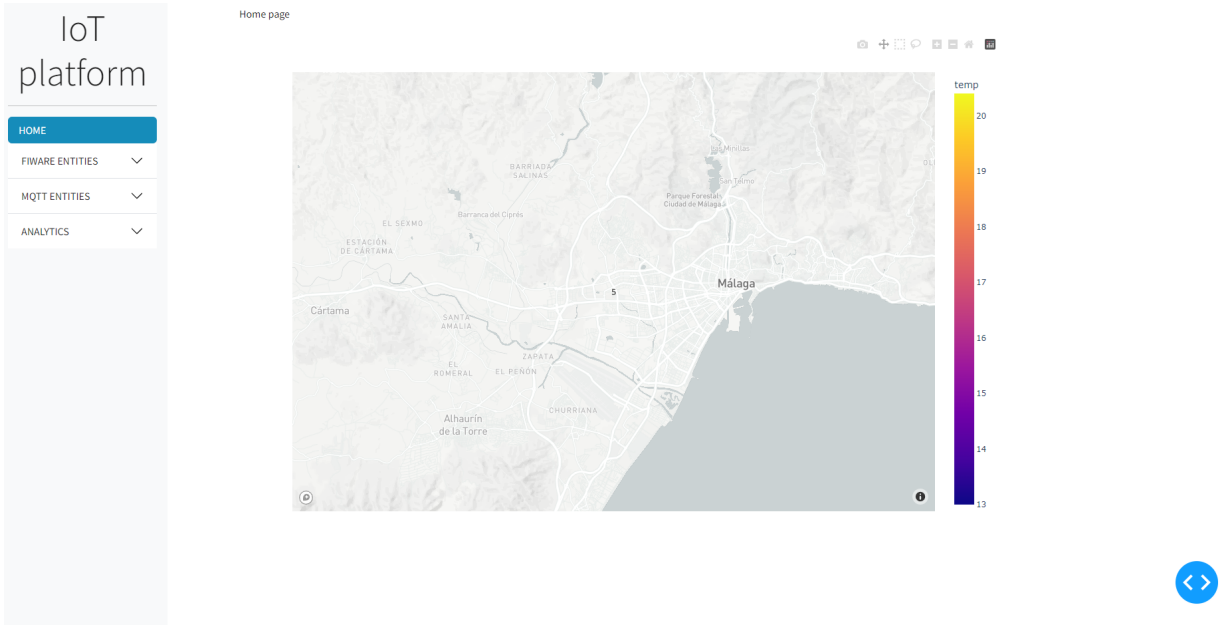
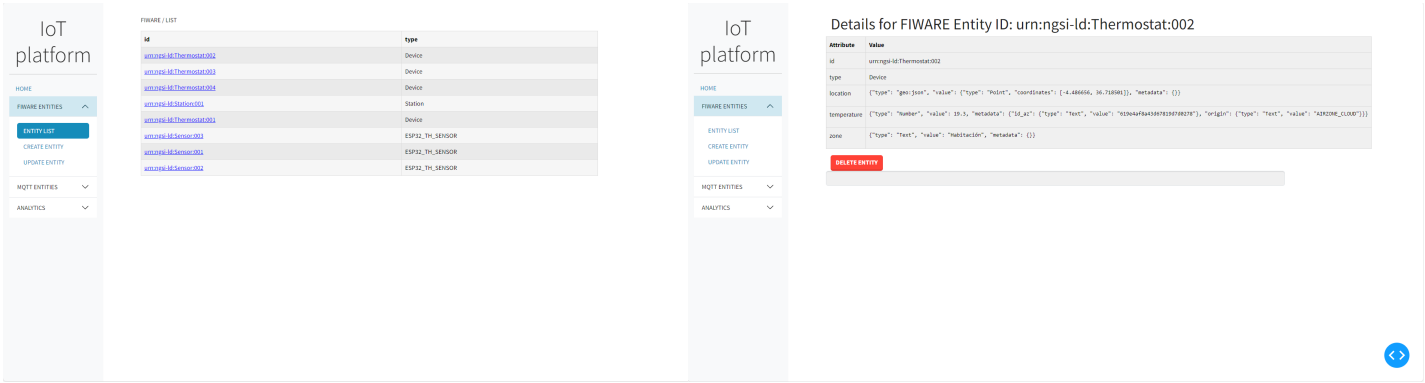


Fig. 12: Home page



(a) FIWARE Entity List page

(b) FIWARE Entity details page

Fig. 13: FIWARE ENTITIES LIST&DETAILS

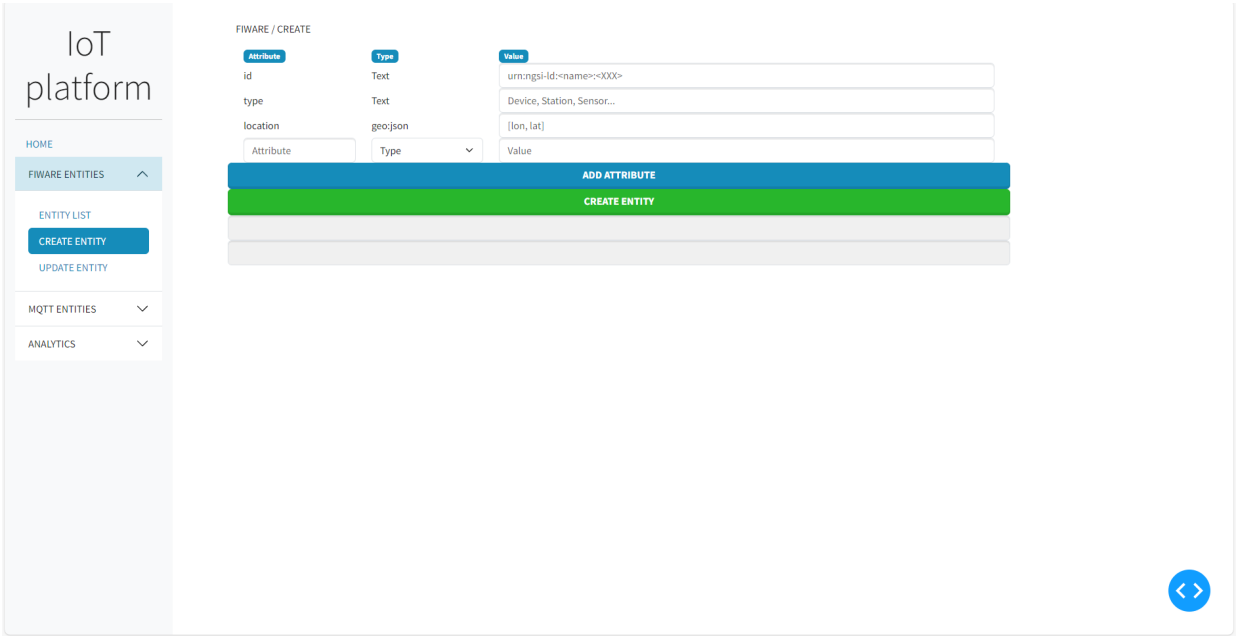


Fig. 14: FIWARE Create Entity

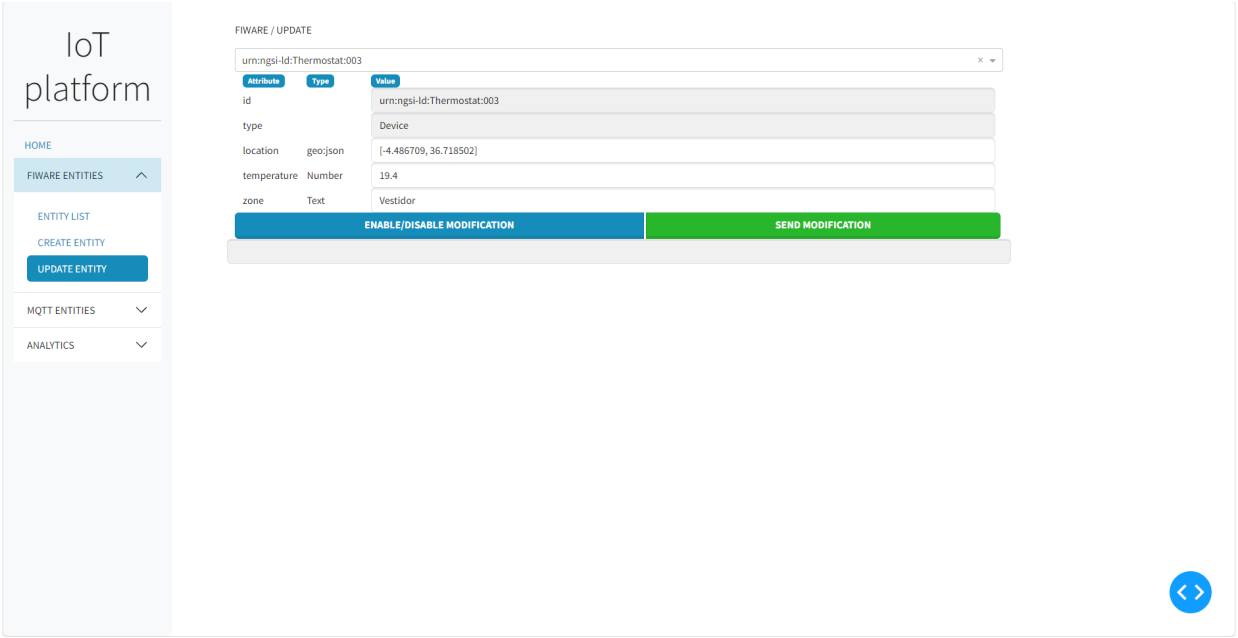
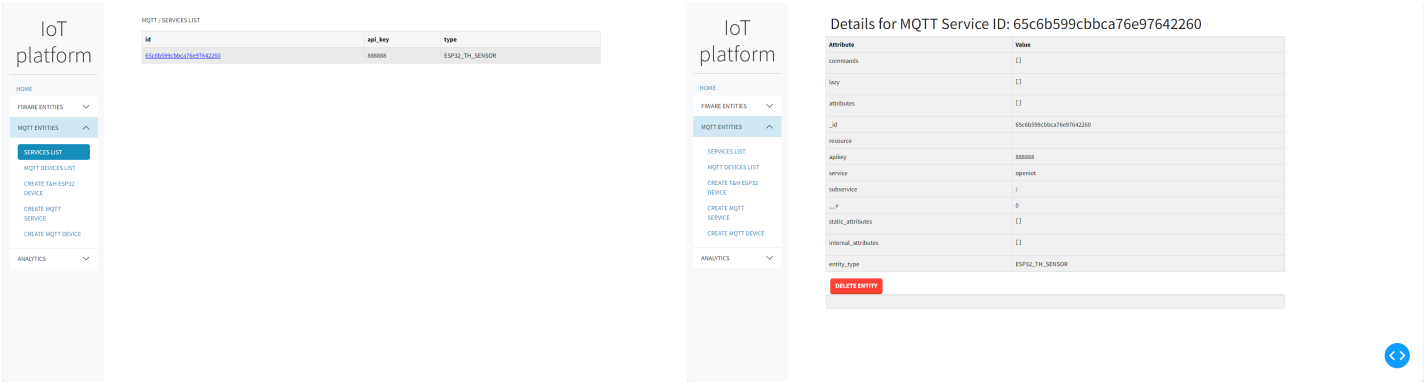
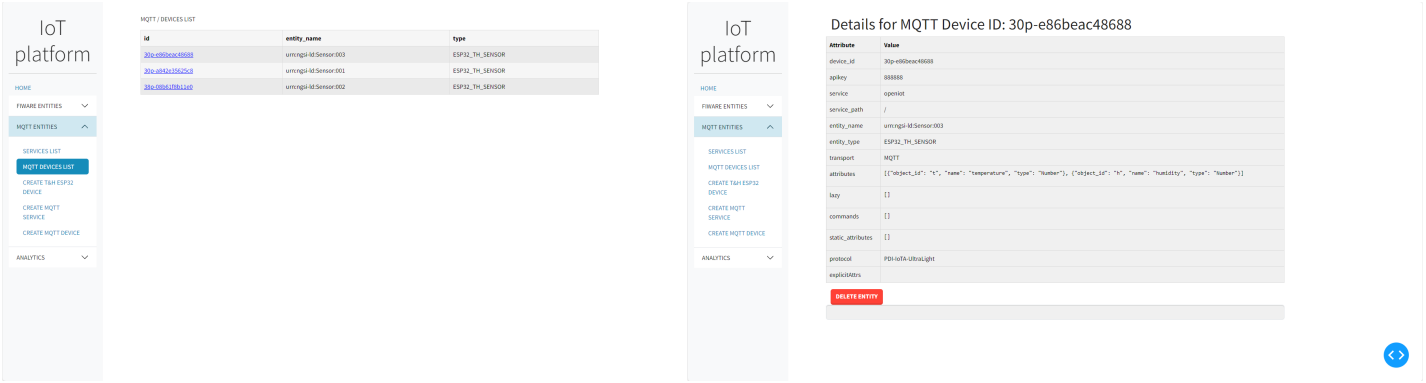


Fig. 15: FIWARE Update Entity



(a) MQTT Service List page (b) MQTT Service details page

Fig. 16: MQTT Services list&Details



(a) MQTT Devices List page (b) MQTT Device details page

Fig. 17: MQTT Devices list&Details

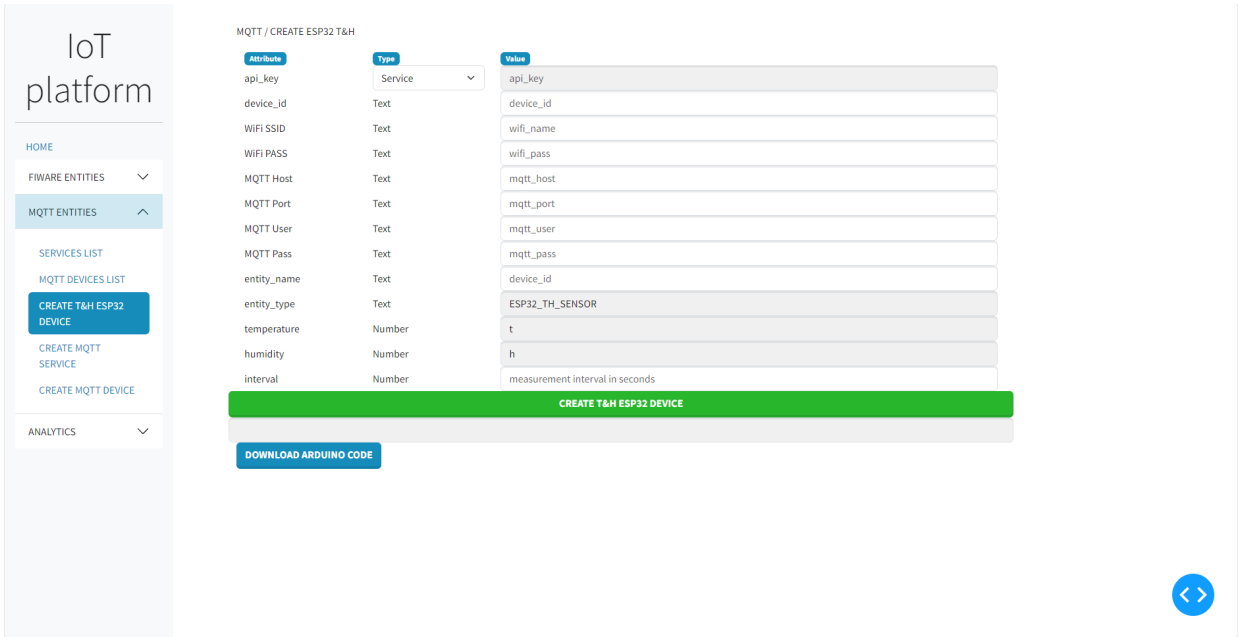


Fig. 18: MQTT Create ESP32 T&H Device

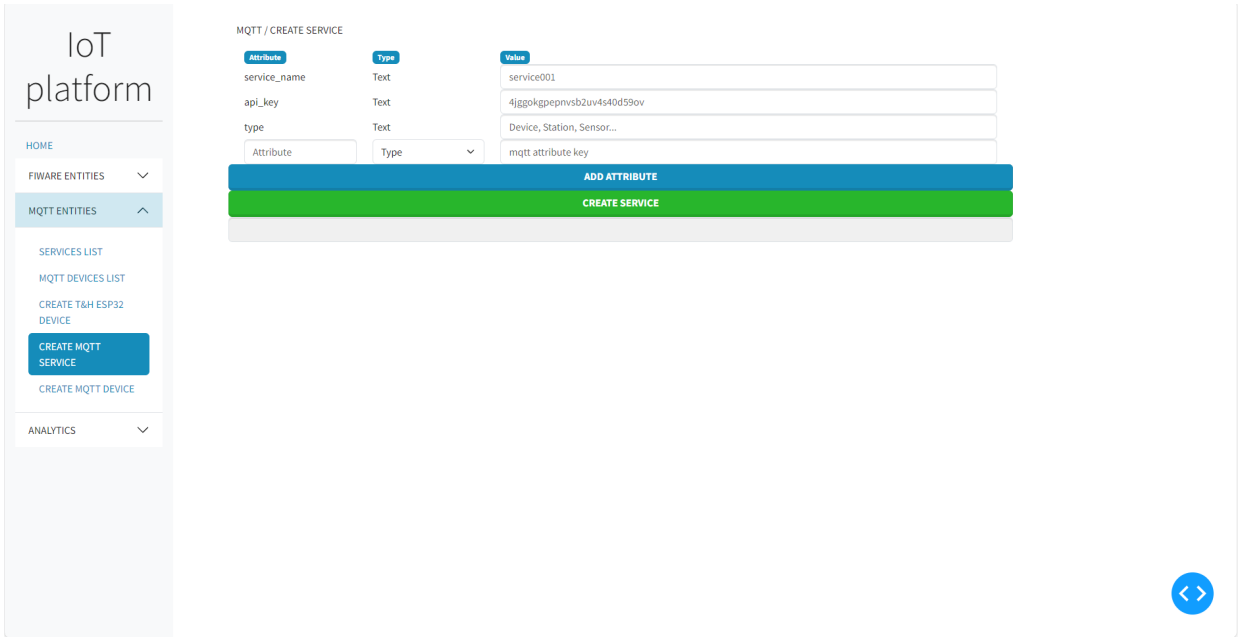


Fig. 19: MQTT Create Service

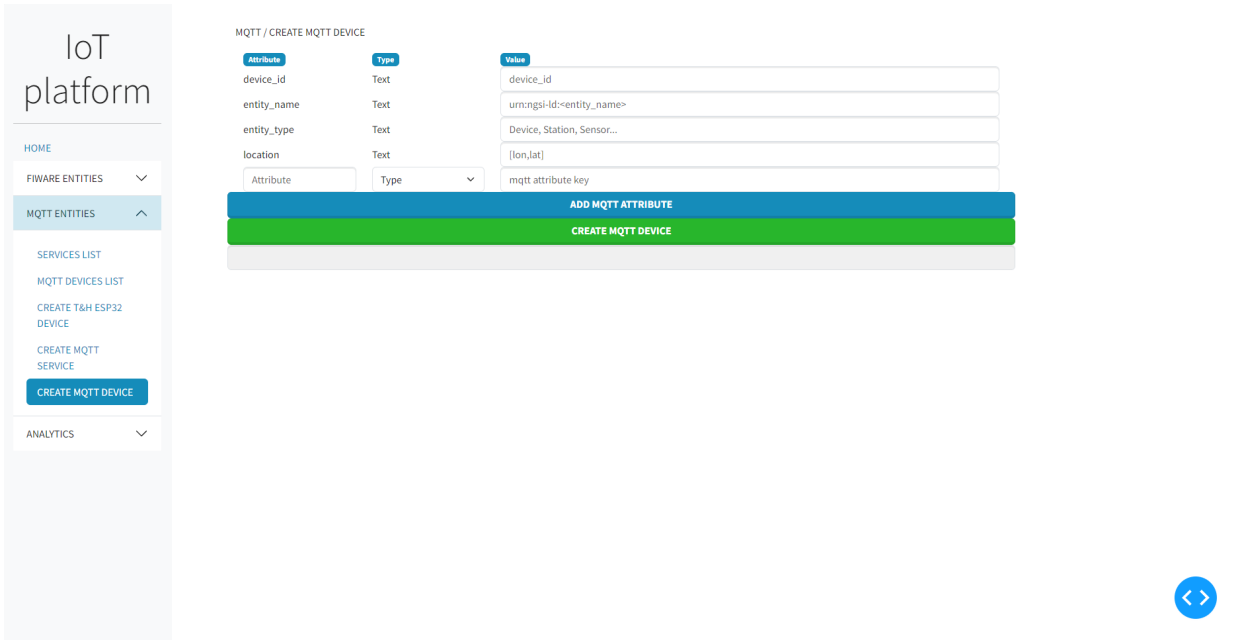


Fig. 20: MQTT Create MQTT device

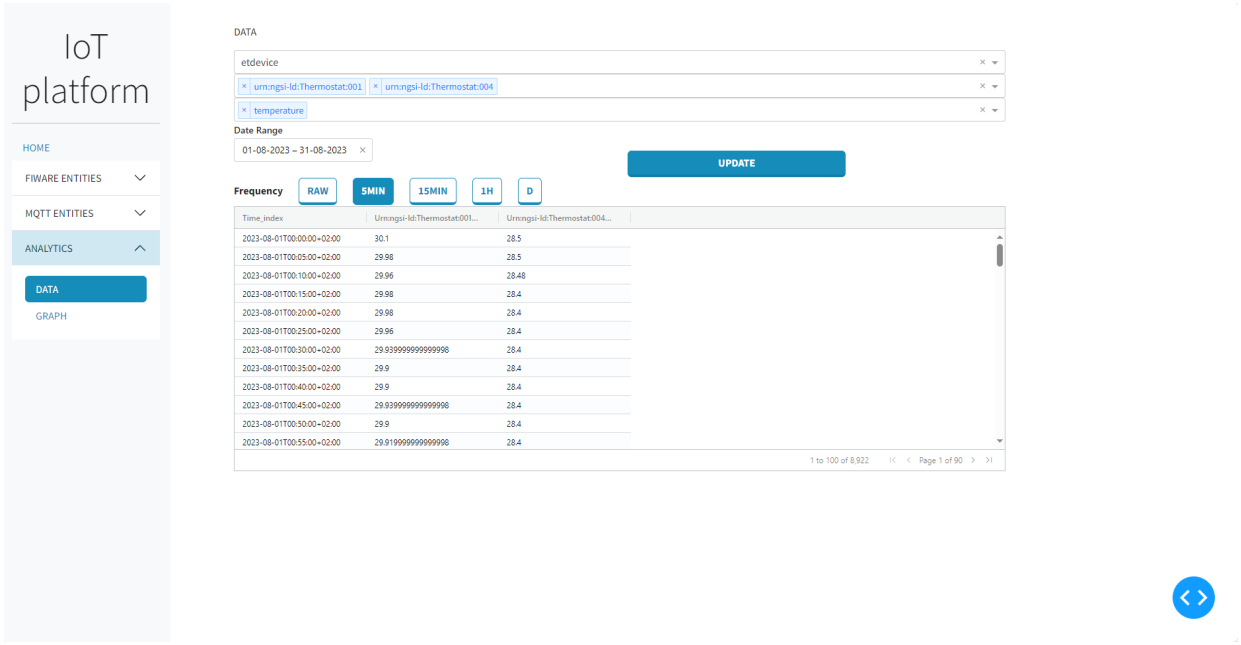


Fig. 21: Analytics datatable page

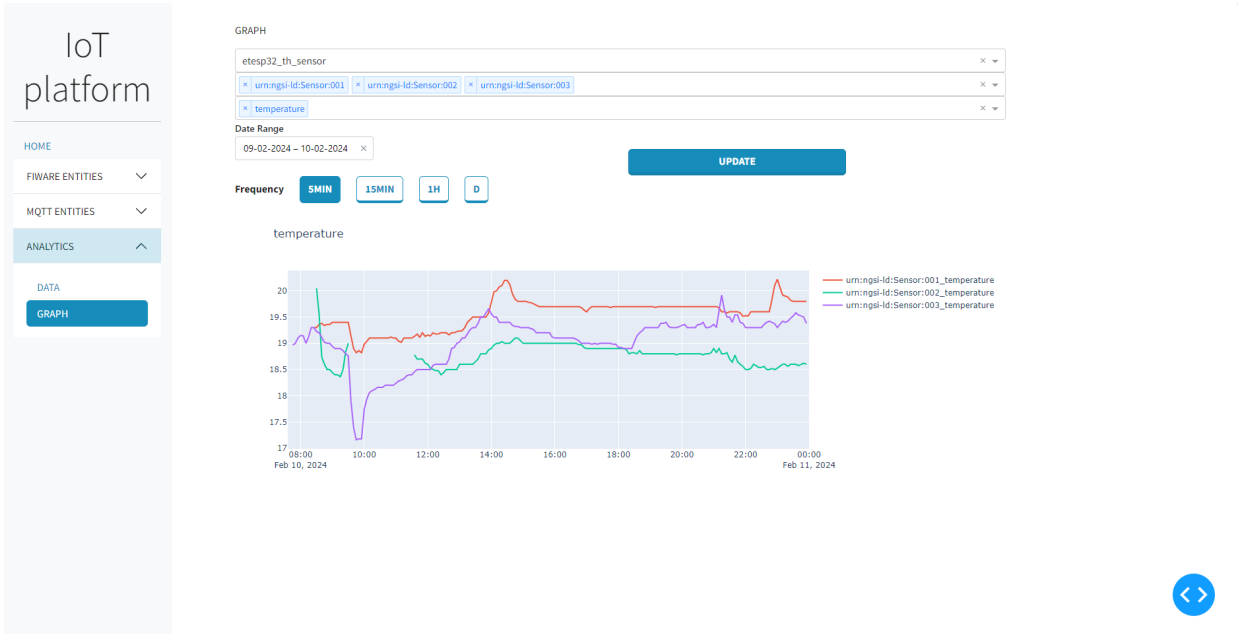


Fig. 22: Analytics graph page