Instituto Superior de Engenharia do Porto

Sensae Console - Enabler Platform for IoT-based Services

Filipe Cruz

A dissertation submitted in partial fulfillment of the requirements for the degree of Master of Science, Specialisation Area of Software Engineering

Supervisor: Dr. Nuno Silva

Evaluation Committee:
President:
Luís Lino Ferreira, Professor, DEI/ISEP

Members:
André Restivo, Professor, DEI/FEUP

Porto, October 15, 2022

Experience is merely the name
men gave to their mistakes.

OSCAR WILDE

Abstract

Today there are more smart devices than people. The number of devices worldwide is forecast to almost triple from 8.74 billion in 2020 to more than 25.4 billion devices in 2030.

The Internet of Things (IoT) is the connection of millions of smart devices and sensors connected to the Internet. These connected devices and sensors collect and share data for use and analysis by many organizations. Some examples of intelligent connected sensors are: GPS asset tracking, parking spots, refrigerator thermostats, soil condition and many others. The limit of different objects that could become intelligent sensors is limited only by our imagination. But this devices are mostly useless without a platform to analyze, store and present the aggregated data.

Recently, several platforms have emerged to address this need and help companies/governments to increase efficiency, cut on operational costs and improve safety. Sadly, most of this platforms are tailor made for the devices that the company offers. This dissertation presents a platform focused on enabling other to create IoT-based services and three Proof of Concept (PoC) services built on top of it. This platform attempts to be device-neutral, IoT middleware-neutral and provide flexible upstream integration and hosting options while providing a simple and concise data streaming Application Programming Interface (API).

Keywords: Internet of Things, System design, Interoperability, Configurability, Real Time Analysis

Resumo

Atualmente, existem mais sensores inteligentes do que pessoas. O número de sensores em todo o mundo deve quase triplicar de 8,74 bilhões em 2020 para mais de 25,4 bilhões em 2030.

O conceito de IoT está relacionado com a interação entre milhões de dispositivos inteligentes através da Internet. Estes dispositivos e sensores conectados recolhem e disponibilizam dados para uso e análise por parte de muitas organizações. Alguns exemplos de sensores inteligentes e seus usos são: dispositivos GPS para rastreamento de ativos, monitorização de vagas de estacionamento, termostatos em arcas frigoríficas, condição do solo e muitos outros. O número de diferentes objetos que podem vir-se a tornar sensores inteligentes é limitado apenas pela nossa imaginação. Mas estes dispositivos são praticamente inúteis sem uma plataforma para analisar, armazenar e apresentar os dados por eles agregados.

Recentemente, várias plataformas surgiram para responder a essa necessidade e ajudar empresas/governos a aumentar a sua eficiência, reduzir custos operacionais e melhorar a segurança dos espaços e negócios. Infelizmente, a maioria dessas plataformas é feita à medida para os dispositivos que a empresa em questão oferece. Esta tese apresenta uma plataforma focada em que propiciar a criação de serviços relacionados com IoT e três provas de conceito apoiadas pela plataforma em questão. Esta plataforma procura ser agnóstica em relação aos dispositivos inteligentes e middleware de IoT usados por terceiros, oferece variadas e flexíveis opções de integração e hosting como também uma API de streaming simples e concisa.

Acknowledgement

The completion of this undertaking could not have been possible without the never ending support of those close to me.

First of all, i would like to thank Professor Dr. Nuno Silva for the involvement and support manifested during this project and my scholarship. I would like to express a deep appreciation and indebtedness to all my close relatives and friends that helped me turn this dream into a reality. Lastly, i would like to thank Margarida for making this journey bearable.

Thank you

Contents

List of Figures

List of Tables

List of Source Code

Chapter 1

Introduction

This chapter provides a short introduction to this dissertation. It describes this work's context, the problem it addresses, the objectives to be fulfilled, the approach taken and finally the results achieved. The chapter closes with the document's structure.

1.1 Context

The Internet of Things (IoT) is a fast-growing technological concept, which aims to integrate various physical and virtual objects into a global network to enable interaction and communication between those objects (Atzori, Iera, and Morabito 2010). According to Nieti, oli, et al. 2020 the main goal of IoT technologies is to simplify processes in different fields, to ensure a better efficiency of systems (technologies or specific processes) and finally to improve life quality. Currently many large-scale enterprises use custom-made IoT technologies to aid their decision making. For example:

- Ericsson has created a platform, Ericsson Maritime ICT, designed to collect and present data regarding cargo ships. Sensors capture information regarding the speed and location of the ship as well as the temperature and condition of the reefer containers. This information is updated in real time and presented to the various parties in the supply chain (Ericson 2020);
- John Deere has created the JDLink platform, designed to give farmers live information about their fleet's location as well as diagnostic and usage data for each machine. Sensors that measure soil and crop conditions in real time help farmers to decide the best time to start harvesting (Deere 2020);
- Verizon has created a platform, Verizon Connect, designed to help reduce fuel consumption, monitor vehicle diagnostics & vehicle maintenance needs, prevent unauthorized out of area use and much more. Sensors installed by Verizon in cars, trucks and machines give insights in real-time about the fleet (Verizon 2022).

Like these, many other large companies are building platforms to aid decision making based on sensor data harvesting. In a pursuit for sustainability, companies are looking to IoT as an approach to increase efficiency and decrease waste. According to Bibri 2018 the IoT and related big data applications can play a key role in catalyzing and improving the process of environmentally sustainable development.

Some of the benefits that IoT, and these platforms, bring to companies are: more operational efficiency, increased security conditions, and cost reduction (T-Mobile 2021).

1.2 Problem

Despite the promised benefits, the initial investment this technology requires to be employed is very high for small and medium companies. As such, its adoption is often postponed or discarded.

In addition to the high costs, these platforms are often associated with a company and its products or businesses, for example, according to S. Chen et al. 2014 in China most IoT applications are domain-specific or application-specific solutions. Another study by Noura, Atiquzzaman, and Gaedke 2019 determine that vendor lock-in is a real concern in IoT, quoting: "each solution provides its own IoT infrastructure, devices, APIs, and data formats leading to interoperability issues". This is often a problem. As an example, for small farmers it is economically unthinkable to change machines and fleet just to be able to benefit from these services.

A service that acts upon IoT data is composed of many pieces and processes, such as (i) managing device network connectivity and ownership, (ii) capturing data via sensors, (iii) routing data through the network, (iv) aggregating and storing data, (v) transforming data into concise information, (vi) analyzing the information captured, (vii) triggering alarms based on this analysis, (viii) providing the gathered information visually or programmatically. It's a complex and constantly evolving system.

In order to deal with these needs there are platforms on the market that facilitate the creation of these services by taking care of device connection and management, such as *AWS IoT Core*, *Azure IoT*, *Google Cloud IoT* and others. Their main purpose is to act as a middleware between costumer-facing application and physical *things* deployed somewhere, such as sensors, actuators or hybrid devices. Each service provides a set of additional functionalities such as data visualization, transformation, storage and analysis.

However these platforms don't provide pre-made specialized solutions to aid the decision making process of end customers and small businesses, such as fleet management, smart irrigation, tracking of deliveries and goods, indoor fire detection, and others. This is often a problem to companies that have little to no background in IoT and in software development. As an example, for a small transportation company it's unthinkable to resort to this middleware services in order to create a fleet management system and perceive the benefits IoT can provide.

Due to this obstacles the adoption of IoT technologies by small companies and individuals is lingered. According to Cisco 2017, 60% of IoT projects stall at the Proof of Concept (PoC) stage.

1.3 Objectives

This work idealizes the creation of a platform responsible for further facilitating the creation of IoT based services. It must focus on:

- Agnostically interacting with different IoT middlewares (receiving sensor measures and dispatching commands to actuators through these platforms);
- Homogenizing and sanitizing the device information, commands available and measures received in a single concise form and semantic;

- Providing various means to interact with the platform and the information handled by it, depending on the costumer needs, such as: (i) full-fledged access via User Interface (UI), (ii) high-level Application Programming Interface (API) focused on its core functionalities, (iii) low-level and generic API to consume device measures and alerts.

To answer these high-level objectives, the platform should encompass essential functionalities such as:

- Data Aggregation: responsible for providing a simple entry-point to the system for any IoT middleware;
- Data Filtering: responsible for discarding erroneous device measures;
- Data Retention: responsible for storing the device measures received;
- Data Transformation: responsible for processing unsanitized data and extracting relevant information from it;
- Data Presentation: responsible for presenting information to the user in real-time;
- Trigger Warning System: responsible for dispatching alerts based on rules applied to the data in motion;
- User Authentication/Authorization: responsible for allowing/denying access to the various platform's components and data depending on the user authentication and authorization level.

Finally, this project envisages the creation of PoCs that answer specific business cases and utilize the developed platform. These PoCs can follow distinct approaches for user interactions: from a full-fledged UI, a simple and business case focused API, or a basic service that dispatches emails/SMS based on alerts captured.

Some of the business cases to address, and their main requirements, are:

- Fleet Management: real-time fleet location feed, fleet location history, calculation of distance traveled by the fleet;
- Smart Irrigation: storage and presentation of environmental conditions captured by sensors and automatic activation of the irrigation system via commands sent to actuators;
- Indoor Fire Outbreak Surveillance: real-time room conditions, alarm trigger system based on abnormal conditions;
- Smart Parking: real-time information regarding free and occupied parking slots.

As such, this project's tangible objectives can be tracked and measured according to two conceptual axis. An axis is related to the platform and its core functionalities (that any service, specific to a business case, relies on) and requirements (being agnostic to IoT middlewares, defining a semantically sound and homogeneous data model, offering different user-faced means of interaction). The other axis is related to the PoCs focused on specific business cases.

1.4 Approach

This work is a greenfield project with the intent of designing and implementing a platform that simplifies the creation of applications based on IoT captured and analysed data and interactions with actuators. Some PoCs that answer the needs of various business cases must be developed. Each business case is considered a concern and should be addressed in an independent PoC. In the end of the project, the envisaged platform will support the intended PoCs.

The pursued approach envisions the project divided in four phases:

- Phase I: Design and implement PoCs that support each business case;
- Phase II: Identify commonality and variability between all designed prototypes;
- Phase III: Design and implement a platform that simplifies the development of this PoCs by aggregating common needs and concepts;
- Phase IV: Refactor the PoCs so that they rely on the platform's functionalities.

During the first phase it is extremely important that the design and implementation of each PoC takes into account the goals of phase II. Even though these services are independent they all shared core responsibilities, functionalities and procedures that can be reused.

During the second phase, the various PoCs will be evaluated so that most common components can be moved to the platform and later reused by them.

During the third phase, a platform that comprises shared functionalities of all PoCs' business cases must be designed and implemented. This platform must offer an agnostic, homogeneous and concise access to sensors and actuators regardless of the IoT platform used to connect to them.

In the final phase, the developed PoCs must be integrated with the API provided by the platform.

The project management will adopt the Scrum methodology described by Schwaber 1997, with monthly sprints that end in presentations of the software to the company and weekly meetings focusing on reviewing the progress, discussing issues that rose and future ideas to add to the backlog.

1.5 Achieved Results

This work gave birth to a platform, **Sensae Console**, capable of handling the desired requirements and functionalities. Some of its main features are: (i) real-time data monitoring, (ii) powerful data classification and categorization, (iii) custom data manipulation via scripting, (iv) virtual device registry and ownership, (v) integrated rule engine to dispatch alerts, (vi) strict user authentication and authorization, (vii) rich set of GraphQL API for management, (viii) designed to scale, (ix) designed to incorporate third-party services as plugins, (x) flexible hosting options: multi-tenant or dedicated.

This platform proved itself capable of integrating with the researched IoT middlewares while offering to consumers a semantically rich and homogeneous data model. The concepts tackled by this data model were materialized in a Software Development Kit

(SDK), *iot-core*, that was created to ease the development of services integrated with this platform via the low-level, generic and event-based API.

Three PoCs were designed and implemented during this project time span: (i) fleet management, (ii) smart irrigation and (iii) notification management.

The platform and PoCs were later evaluated according to the performance requirements envisioned for them as a dedicated hosted solution.

1.6 Document Structure

This document is divided into 7 more chapters that explore this work.

- State of the Art: where literature related to this work is explored;
- Requirements Elicitation: where this project's requirements are listed;
- Design: where the architectural design of the solution is presented;
- Implementation: where the implementation of the solution is addressed;
- Evaluation: where the evaluation of the solution is presented and results discussed;
- Conclusion: where a final overview of the project is presented, wrapping up the achievements and future work of this project and solution.

Chapter 2

State of the Art

This chapter introduces a modest introduction to the IoT landscape, focusing first on technologies, solutions and the architectural context surrounding them and later the various business areas where IoT is used. The intent of this chapter is to introduce the reader to the subjects related to this work.

The core objective of this project is the development of an IoT platform. Therefore this chapter will not focus on the physical side of the business (installation, deployment, onboarding and management of devices and IoT gateways).

2.1 IoT Architectural Context

This section focus on the landscape of services, solutions, tools, terms and technologies that are related to IoT. It starts by dividing them in three distinct categories:

- Infrastructure: concepts that don't answer any specific business case but are common or even a requirement for most IoT related projects;
- Platforms: tools that support and ease the creation of IoT projects but are not a necessity;
- Solutions: solutions that answer specific business cases.

After these categories are explored, some Reference Architectures are presented.

2.1.1 Infrastructure

Here some technologies and services that are almost a requirement to build IoT Solutions are presented. This section is divided in the following themes: Mediator, IoT Middleware, Rule Engine and Data Storage.

These specific themes are mentioned since they are relevant for the project.

2.1.1.1 Mediator

This section refers to technologies that enable the system-wide use of the publish/subscribe pattern by mediating messages or events between entities of the system in a asynchronous manner.

According to Dias, Restivo, and Ferreira 2022, "broker-mediator architectures are highly used [in IoT Solutions]" and, the publish/subscribe "communication pattern is also common in many IoT and Web applications" (Lazidis, Tsakos, and Petrakis 2022).

The publish/subscribe pattern can be summarized by the following description: "Subscribers have the ability to express their interest in an event, or a pattern of events, and are subsequently notified of any event, generated by a publisher, which matches their registered interest" (Eugster et al. 2003).

In this section two open-source mediators, *RabbitMQ* and *Apache Kafka*, will be introduced.

According to Lazidis, Tsakos, and Petrakis 2022, RabbitMQ offers better latency than Kafka but has a significant lower throughput of messages per second.

2.1.1.1.1 RabbitMQ

RabbitMQ is a message broker with support for various protocols (some via extensions) such as: (i) Advanced Messaging Queue Protocol (AMQP) 0.9.1, (ii) STOMP, (iii) Message Queuing Telemetry Transport (MQTT), (iv) AMQP 1.0, (v) HTTP and WebSockets and (vi) RabbitMQ Streams. This section will focus on AMQP 0.9.1 and MQTT.

As discussed in the article, *AMQP 0-9-1 Model Explained*, the AMQP 0.9.1 protocol defines four main concepts: (i) publisher, (ii) exchange, (iii) queue, (iv) consumer. The following diagram, Figure 2.1 explains how this concepts interact.

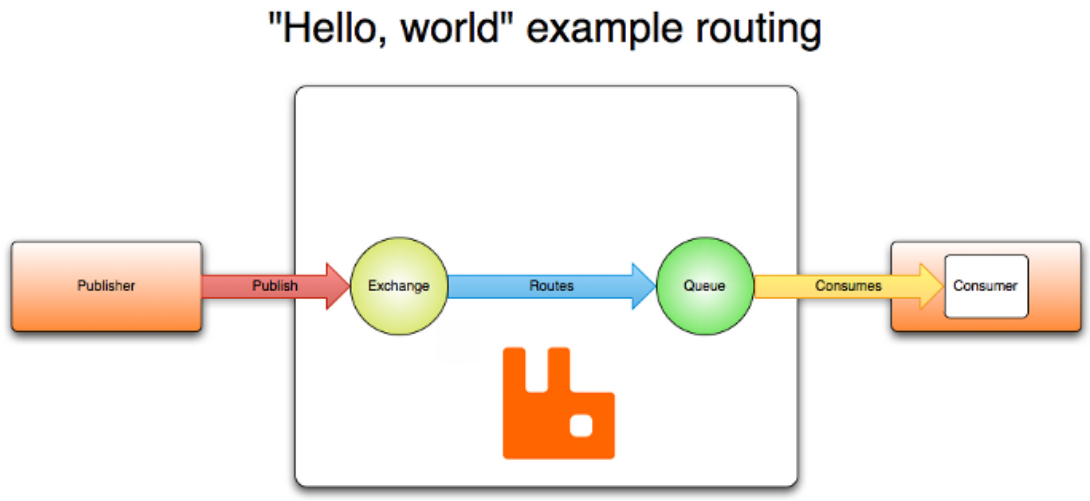


Figure 2.1: AMQP 0.9.1 Protocol Concepts by VMWare 2022a

As discussed in *AMQP 0-9-1 Model Explained*, there are four types of exchanges:

- Direct Exchange: ideal for the unicast routing of messages;
- Fanout Exchange: ideal for the broadcast routing of messages;
- Topic Exchange: ideal for the multicast routing of messages, queues subscribe to specific routing keys;
- Header Exchange: ideal for more flexible unicast routing of messages, queues subscribe to specific message headers.

This is the "core" protocol supported by the system.

The MQTT protocol is a "binary protocol emphasizing lightweight publish / subscribe messaging, targeted towards clients in constrained devices" (VMWare 2022b). According to MQTT 2022, this is the standard protocol for IoT Messaging.

Craggs 2022 mentions that "MQTT has just one routing mechanism - topic subscriptions", when compared with AMQP. It can be seen as simpler version of AMQP tailored for IoT low powered devices. It has a lower message overhead, and instead of using *'.'* as the level separator it uses a *'/'*, for wildcards it uses *'+'* instead of *'*'*.

Both protocols have various robust client libraries written for various programing languages (VMWare 2022b).

According to Dobbelaere and Esmaili 2017 the unique features of RabbitMQ are: (i) Standardized Protocol, (ii) Multi-protocol, (iii) Distributed Topology Modes, (iv) Comprehensive Management and Monitoring Tools, (v) Multi-tenancy and Isolation, (vi) Consumer Tracking, (vii) Disk-less Use, (viii) Publisher Flow Control, (ix) Queue Size Limits and (x) Message time to live.

2.1.1.1.2 Kafka

Apache Kafka is a distributed event streaming platform that can be seen as an append-only-log. Its main concepts are: (i) events, (ii) producers, (iii) consumers and (iv) topics.

Events are sent by producers to specific topics that are partitioned by several Kafka brokers. Events are persisted, something that the AMQP protocol doesn't do, and consumers can pull events from their subscribed topics. Since events are stored, a consumer can deliberately rewind back to an old offset and re-consume data (Kafka 2022). This enables consumers to easily follow the *Event sourcing Pattern* 2022b and reconstruct an entity's current state by replaying the events.

"The Kafka ecosystem offers libraries and tools that provide additional functionality on top of Kafka" (Dobbelaere and Esmaili 2017), such as Kafka Connect and Kafka Streams.

According to Confluent 2022a, "Kafka Connect works as a centralized data hub for simple data integration between databases, key-value stores, search indexes, and file systems".

Kafka Streams "is a client library for building applications and microservices, where the input and output data are stored in an Apache Kafka cluster. It combines the simplicity of writing and deploying standard Java and Scala applications on the client side with the benefits of Kafka's server-side cluster technology" (Confluent 2022b).

According to Dobbelaere and Esmaili 2017 the unique features of Kafka are: (i) Long term message storage, (ii) message replay, (iii) kafka connect and (iv) log compaction.

2.1.1.2 IoT Middleware

The term IoT Middleware in this work referes to services/solutions that offer or focus on three main features:

- Device Management: device onboarding, maintenance, updates and monitoring;
- Data Transmission: device data collection and provision in real-time;
- Device Control: offer an API to send commands to devices.

These solutions handle the lower-layers communication protocols such as: (i) RFID, (ii) Bluetooth/BLE, (iii) LoRa and LoRaWAN, (iv) SigFox, (v) ZigBee, (vi) Thread, (vii) EnOcean, (viii) ANT, (ix) GPRS/2G/3G/4G/5G cellular, (x) Wi-Fi (Dias, Restivo, and Ferreira 2022).

The article "Designing and constructing internet-of-Things systems: An overview of the ecosystem" 2022 argues that due to a lack of standards a designer can't objectively choose from one or another protocol when developing IoT Systems. These solutions have the benefit of abstracting these lower-layers communication details and provide APIs that provision all data gathered with high-layer communication protocols like (i) HTTP, (ii) MQTT, (iii) AMQP, (iv) CoAP, (v) XMPP, (vi) LwM2M, (vii) LLAP, (viii) UPnP (Dias, Restivo, and Ferreira 2022).

The services that will be briefly mentioned are:

- The Things Stack;
- Azure IoT Hub;
- Helium Console.

This project focus mostly on Helium Console since it controls the lower-layers communication with the company's installed devices.

2.1.1.2.1 The Things Stack

The Things Industries 2021a is behind the creation of a platform that focus on interacting with devices via LoRa and LoRaWAN.

Apart from the common features referred above, their cloud platform provides integrations with AWS IoT, Azure IoT Hub and other platforms such as Cumulocity, ThingsBoard, and LoRa Cloud. It also provides agnostic APIs based on MQTT, webhooks and PubSubs to connect with other systems and publish the gathered device measures.

The company maintains an open-source version of their platform. According to *The Things Stack Github Page* 2021b, this platform supports: almost all the LoRaWAN protocols specifications, device classes, regional parameters and onboarding options; device payload conversion for well-known formats or via custom Javascript functions; user and entity management; and GRPC and HTTP management APIs.

2.1.1.2.2 Azure IoT Hub

Azure IoT Hub 2022b is a platform provided by Microsoft "that acts as a central message hub for communication between an IoT application and its attached devices" (Microsoft 2022b). A helper service, Azure IoT Hub Device Provisioning Service, handles the registration and provisioning of devices.

According to Microsoft 2022b data can also be routed to different services for further processing, it is possible to route data to: azure storage containers, event hubs, service bus queues and services bus topics. According to Microsoft 2022c, it is also possible to subscribe to an AMQP endpoint to receive the device measures sent to IoT Hub. Azure IoT Hub also integrates with Azure Event Grid that allows one to register an HTTP endpoint for Azure Event Grid to dispatch events to, in this case, device measures.

According to *Magic Quadrant for Industrial IoT Platforms* 2022, Azure is a leading IoT Platform.

2.1.1.2.3 Helium Console

Helium Console 2022 is a new IoT Platform. This platform's business model is fairly different from the ones mentioned before. Besides the platform itself, the company also maintains an open-source blockchain designed to power IoT devices with wireless connectivity. Helium technology enables communication between devices and the internet by promoting the expansion of a new network operated by common people.

According to Helium 2018a, " Powering the Helium network is a blockchain with a native protocol token incentivizing a two-sided marketplace between coverage providers and coverage consumers". Coverage providers install in their homes an IoT gateway, Hotspot, that provides wide-area LoRaWAN coverage to the area surrounding it. Coverage consumers, e.g. IoT devices, connect to this LoRaWAN network and send the gathered measures to the Hotspot. The Hotspot then routes the measures to the cloud in exchange for a transport fee, paid by the owner of the IoT device. The fee corresponds to a token that can be traded for dollars, bitcoin or any other available commodity on online exchanges.

In theory, this creates a self sustained economy that can provide to everyone a cheap and open communication layer for IoT devices that is backed by open standards, and, provide a consistent income to those who want to support the network. The *Helium Explorer Page*[1] documents the number and location of each Hotspot, current there are almost a million of IoT gateways installed throughout the world. Europe, USA and China are the regions with better network coverage.

This network, as an example, gives local farmers in remote locations the possibility to install a smart irrigation system without paying excruciating fees to an Internet Service Provider (ISP) for satellite or 4G coverage.

The company's business idea is to utilize this low-cost network to offer IoT-based services to small-to-medium size organizations. Therefore, the solution to develop in this project will, at least in the near future, focus on the IoT Middleware solution provided by Helium.

Helium Console enables devices to connect to pre-configured, cloud-based applications or send data directly over HTTP or MQTT via Integrations. It currently supports two core integrations, HTTP and MQTT, and multiple community integrations such as: (i) Helium Cargo, (ii) myDevices Cayenne, (iii) AWS IoT Core, (iv) Azure IoT Hub, (v) Azure IoT Central, (vi) Adafruit IO, (vii) Akenza, (viii) Datacake, (ix) Google Sheets, (x) Microshare, (xi) TagoIO, (xii) Ubidots (Helium 2018b).

2.1.1.2.4 Summary

This section mentioned three IoT middlewares, each with their distinctive characteristics:

- The Things Stack: an open-source solution;
- Azure IoT Hub: a widely used cloud service;
- Helium: the solution that this project is required to adopt first.

[1] link to Helium Explorer Page

Next, various Data Storage solutions will be discussed.

2.1.1.3 Data Storage

This section refers to how information is stored and managed across systems.

A Database Management System (DBMS) is a general-purpose software system that facilitates the processes of defining, constructing, manipulating, and sharing data - *Fundamentals of Database Systems*. DBMSs can be categorized according to several criteria, such as the data model, number of users or distribution. This section focus on the data model.

These are the most common data models, as per Elmasri et al. 2000:

- The **relational data model** represents a database as a collection of tables, where each table can be stored as a separate file;
- The **document-based data model** is based on JSON (Java Script Object Notation) and stores the data as documents, which somewhat resemble complex objects;
- The **column-based data model** stores the columns of rows clustered on disk pages for fast access and allow multiple versions of the data;
- The **graph-based data model** stores objects as graph nodes and relationships among objects as directed graph edges;
- The **key-value data model** associates a unique key with each value (which can be a record or object) and provides very fast access to a value given its key.

The following sections briefly discuss these data models.

2.1.1.3.1 Relational Data Model

This data model has a wide variety of usage in the industry. It relies on a "schema to define exactly and unambiguously all the relationships of interest to the users" (Zaniolo and Meklanoff 1981). Therefore, it excels to represent concrete concepts within a business domain. According to Jatana et al. 2012 almost all relational databases use Structured Query Language (SQL) to access and modify the data stored in the database. "SQL consists of a set of facilities for defining, accessing and otherwise managing data" (Date 1989).

The reliability of a database model is checked with the help of ACID properties: **A**tomicity, **C**onsistency, **I**solation, **D**urability.

Some of the technologies that follow this data model are: (i) *MySQL*, (ii) *PostgresSQL* and (iii) *Oracle database*. These are all ACID compliant and follow the SQL standard.

This data model is intended for strictly structured data with well defined interrelations.

2.1.1.3.2 Document-based Data Model

This data model rose from the increasing need to store and analyze unstructured data as stated by Miloslavskaya and Tolstoy 2016. Citing Elmasri et al. 2000, a "major difference between document-based systems versus object and object-relational systems (relational database systems) is that there is no requirement to specify a schema".

This data model, and all other mentioned below have some key differences when compared with the Relational Data Model. Databases with this data model usually don't adhere to the SQL standard since there is no define notion of the data structure. And, they follow the BASE acronym: **B**asically **A**vailable, **S**oft State and **E**ventually Consistent, instead of the ACID properties.

Some of the technologies that follow this data model are: *MongoDB* and *Firestore*.

MongoDB is open source and has a "flexible document data model along with support for ad-hoc queries, secondary indexing, and real-time aggregations to provide powerful ways to access and analyze data" (MongoDB 2022).

Contrary to *MongoDB*, *Firestore* is a close source service developed by Google to offer a simple to use database service that can be accessed via a SDK for Android and IOS.

2.1.1.3.3 Column-based Data Model

This data model is used in applications that require large amounts of data storage, and is commonly named *data warehouses*. According to Dehdouh et al. 2015, a data warehouse is "designed according to a dimensional modelling which has for objective to observe facts through measures, also called indicators, according to the dimensions that represent the analysis axes". Citing J. Han et al. 2011, these databases "can maintain high-performance of data analysis and business intelligence processing".

Some of the technologies related to this concept are: (i) *HBase*, (ii) *CassandraDB*, (iii) *InfluxDB*, (iv) *QuestDB*.

According to George 2011 HBase is a "distributed, persistent, strictly consistent storage system with near-optimal write and excellent read performance". This database uses Hadoop Distributed File System (HDFS) as its file system, and so, it is built on top of Hadoop. HBase does not support a structured query language like SQL, "even though it's comprised of a set of standard tables with rows and columns, much like a traditional database" (IBM 2020c).

CassandraDB is a distributed storage system for managing very large amounts of structured data spread out across many commodity servers, while providing highly available service with no single point of failure (Lakshman and Malik 2010). It was developed internally by Facebook and then later open-sourced to the Apache Foundation. It doesn't support SQL.

According to Naqvi, Yfantidou, and Zimányi 2017, InfluxDB is an "open-source schemaless Time Series Database (TSDB) with optional closed-sourced components developed by InfluxData. It is written in Go programming language and it is optimized to handle time series data." It provides an SQL-like query language and also defines a new protocol for fast data ingestion (InfluxDB 2022b).

QuestDB is a relational column-oriented database designed for time series and event data and entitles it self as the "fastest open source time series database" (questdb.io 2022). According to benchmarks (Ilyushchenko 2021) preformed using the Time Series Benchmark Suite (TSBS), Winslow 2021, QuestDB ranks as the fastest option in the market. It has out-of-the-box support for SQL Postgres wire protocol, (thus integrating with *PostgresSQL* client libraries), can be easily deployed using a single Docker Image, and also supports the InfluxDB Line Protocol (ILP).

2.1.1.3.4 Graph-based Data Model

According to Angles and Gutierrez 2008, the databases that follow this data model can be characterized by the following principles:

- "Data and/or the schema are represented by graphs, or by data structures generalizing the notion of graph (hypergraphs or hypernodes)";
- "Data manipulation is expressed by graph transformations, or by operations whose main primitives are on graph features like paths, neighborhoods, subgraphs, graph patterns, connectivity, and graph statistics";
- "Integrity constraints enforce data consistency. These constraints can be grouped in schema-instance consistency, identity and referential integrity, and functional and inclusion dependencies".

These databases flourish when the importance of the information relies on the relations more or equal than on the entities, e.g. biology, Web mining, semantic Web, social networking and recommendation engines (Angles 2012; Miller 2013).

Some of the technologies related to this concept are: (i) *Neo4j*, (ii) *Dgraph*.

2.1.1.3.5 Key-Value Data Model

Databases that follow this data model have their data represented as simple key and value relations. Citing Pokorny 2011, "A key uniquely identifies a value (typically string, but also a pointer, where the value is stored) and this value can be structured or completely unstructured (typically BLOB). The approach key-value reminds simple abstractions as file systems or hash tables (DHT), which enables efficient lookups".

Some of the technologies related to this concept are: (i) *DynamoDB*, (ii) *Redis*.

According to DeCandia et al. 2007 " Dynamo has a simple key/value interface, is highly available with a clearly defined consistency window, is efficient in its resource usage, and has a simple scale out scheme to address growth in data set size or request rates".

When compared with *DynamoDB*, *Redis* has a must more compelling feature, its license. *Redis* is an "open source, in-memory data store used by millions of developers as a database, cache, streaming engine, and message broker" (Redis 2022).

2.1.1.3.6 Summary

This section tackled most types of databases available. It became clear that the Column-based Data Model is tailored for IoT, due to the large amounts of data it can ingest in short amount of time. The focus on time series analysis by some solutions is also deemed important since sensor data is always interconnected to the time it was collected.

In the next section, the concept of Rule Engines, and their benefits for IoT, will be introduced.

2.1.1.4 Rule Engine

Chisholm 2004 defines a rule engine as a "program that uses rules to reach conclusions from facts (premises)". These systems help to externalize business logic from the application code. In this area they are intended to provide the means to react based on the

measures gathered. Rule Engines have a wide use in IoT Systems according to Luo et al. 2021; Milenkovic 2020; Pierleoni et al. 2020.

According to Waylay 2020, most of the IoT platforms on the market today have a rules engine based on forward chaining algorithms, such as Redhat Drools, Cumulocity, Eclipse Smart Home (discontinued), AWS rule engine, Thingsboard and others.

Rule Engines are often also categorized as Information Flow Processing (IFP) Systems, as per Cugola and Margara 2012. The following diagram, Figure 2.2, represents them.

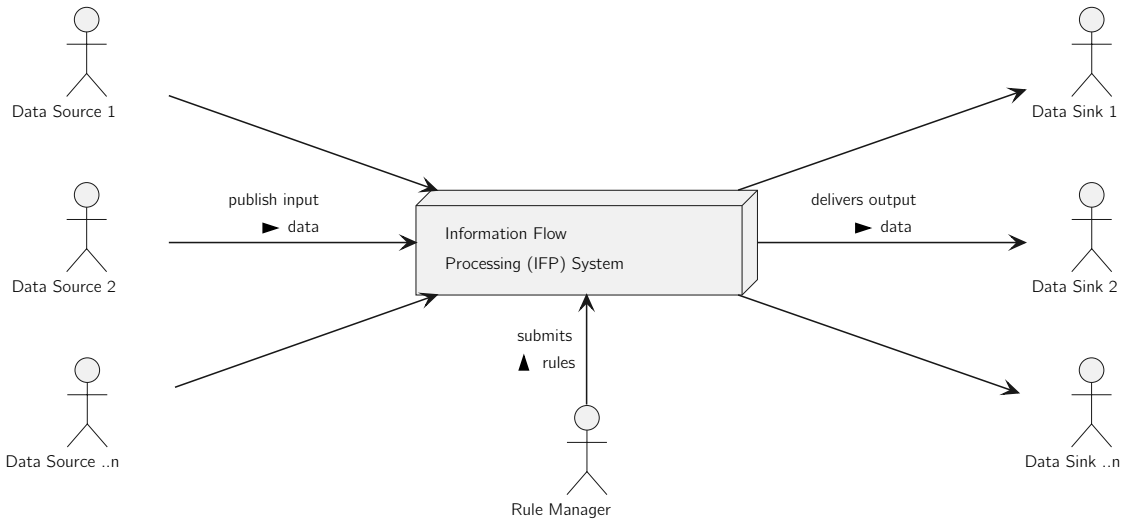


Figure 2.2: High-Level View of a IFP System

Some Rule Engine can support a Flow-based programing, a term coined by J. P. Morrison 1994. Flow-based programing refers to the use of a visualization technique that allows users without coding experience to understand, create and alter programs by manipulating graphical blocks that represent program components or functionalities (J. Morrison 2010). A fine example of this is *Node-RED* 2022b. This tool is widely used in IoT (M. Silva et al. 2020).

As for the other tool mentioned:

Cumulocity is an IoT platform that allows users to define business operations for immediate processing of incoming data from devices or other data sources. As per Cumulocity 2022, the operation logic is implemented in Apama's Event Processing Language (EPL).

Drools is a "business-rule management system with a forward-chaining and backward-chaining inference-based rules engine, allowing fast and reliable evaluation of business rules and complex event processing". The operation logic is implemented in the Drools Rule Language.

AWS Rule Engine service is coupled with *AWS IoT Core*, the operation logic is implemented according to a json-schema where one can define what triggers an action, via an SQL statement processed against an MQTT topic, and define a set of actions, e.g. send a message to *Apache Kafka*, store data in *DynamoDB*, send a POST request to and endpoint, and others (Services 2022).

Thingsboard offers a rule engine based on *Node-RED*, according to their documentation in *Rule Engine Overview* 2022a.

Stream Processing rule engines such as *Apache Storm* 2022, *Apache Spark* 2022 and *Apache Flink* 2022 can also fit in this category but were not investigated.

2.1.2 Platforms

The platforms described here provide a Mashup-based development environment. According to Dias, Restivo, and Ferreira 2022, "mashup tools are solutions that allow developers to construct application and systems in a component-based fashion (e.g. Widgets) or Web service composition (e.g. REST APIs)".

ThingsBoard, DataCake and TagoIO are some examples of these platforms.

2.1.2.1 ThingsBoard

ThingsBoard is "an open-source IoT platform for data collection, processing, visualization, and device management" (ThingsBoard 2022b). Some of its features are: (i) devices, assets and customers provisioning with the possibility to define relations between them, (ii) device and assets data collection and visualization, (iii) rule engine to trigger alarms, (iv) device control via remote procedure calls, (v) enables the design dynamic and responsive dashboards and present device or asset telemetry and insights to customers, (vi) use-case specific features using customizable rule chains and (vii) device data can be pushed to other systems.

This platform provides several pre-made solutions like: (i) Smart Energy, (ii) Smart Farming, (iii) Fleet Tracking, (iv) Smart Metering, (v) Environment Monitoring, (vi) Smart Office, (vii) Water Metering, and (viii) Smart Retail.

2.1.2.2 DataCake

DataCake is a "multi-purpose, low-code IoT platform that requires no programming skills and minimal time to create custom IoT applications that can be brought into a white label IoT solution at the push of a button" (Cake 2021).

This platform provides several pre-made solutions like: (i) Welding Fume Monitoring, (ii) Urban Air Quality, (iii) Industrial Gas Supply, (iv) Air Quality Monitoring, (v) Climate Monitoring, (vi) Cryogenics Monitoring, (vii) CO2 Monitoring, (viii) Water Level and Flood Monitoring and (ix) Industrial IoT.

2.1.2.3 TagoIO

TagoIO "offers the tools for a business to manage devices, store data, run analytics, and integrate services. It combines everything with an easy-to-use application and user management system" (TagoIO 2022). Its features are: (i) possibility to connect to any device, (ii) enables the design of custom dashboards, (iii) easy data management, (iv) can run custom analysis written in Javascript, (v) custom defined dispatch of SMS, notifications and e-mails and (vi) possibility to tailor the UI to the company's desire (e.g. logo and color).

Contrary to the other mentioned platforms, the author thinks that the pre-made solutions are much more lackluster and device-centered instead of business-case centered and therefore weren't mentioned.

2.1.2.3.1 Summary

This section indicated some of the platforms built to ease the creation of business case focused applications that rely on IoT data via a Mashup-based development. These platforms enable someone with close to no programing knowledge to design dashboards that present data in various manners, however "these approaches commonly have limitations and unaddressed challenges, such as not capturing the full software development life-cycle, having large-scale limitations and suffering from leaky abstractions" (Dias, Restivo, and Ferreira 2022).

The intent behind this section is to provide some context to what the market as to offer for those who want to build IoT applications with ease.

Next, some solutions focused on specific business cases will be presented.

2.1.3 Solutions

This section focus on several solutions devoted to specific business cases. The objective of this section is to simply introduce the major features of the solutions, how they work and what is the business model behind their company, when applicable.

The business cases addressed here are the ones this project attempts to answer.

2.1.3.1 Indoor Fire Detection

This section mentions some Indoor Fire Detention systems discussed in academia. The benefits of this business case are: (i) increased security against fire outbreaks, (ii) reduced fire detection times and (iii) automated alert dispatch to relevant authorities.

Listyorini and Rahim 2018 introduces a prototype that relies on IoT and fuzzy logic to detect fires. In the experiment, a device that captures temperature and the light source wavelength, closely monitors a specific place (maximum distance of 60cm from the fire) where the fire simulation takes place. The device sends the measures collected to the cloud to be analyzed by an algorithm that relies on fuzzy logic to reduce false positives. The prototype was able to detect "real fires" and ignore small fire sources such as cigarettes, lighters, and matches.

Saeed et al. 2018 proposes the use of a wireless sensor network to support fire detention sensor in smart homes. The experiment used smoke sensors, gas detectors and heat sensors to detect changes in the home. It considered that each division would have 3 sensors installed. In order to detect a fire, two measures would have to surpass the thresholds defined for their division. The proposal was able to detect all fires.

Wu, L. Chen, and Hao 2021 proposes a multi-sensor data fusion method based on a back propagation neural network. The study used temperature, smoke and CO measures to detect a fire. The study indicates that the method used can reduce the fire detection time by 32%.

Mohammed et al. 2021 proposes the use of computer vision technology for fire-detection. A Closed-circuit television (CCT) camera monitors the house and streams the data collected to a server. The server then processes the images received, three algorithms were explored: color detection, moving object detection and flicker detection.

As presented here, this business case is being tackled from various angles with no clear and definitive answer to the problem. Qureshi et al. 2015 argues that conventional methods are not as robust and dynamic as video-based image processing methods.

2.1.3.2 Smart Irrigation

This section mentions some Smart Irrigation Solutions discussed by Obaideen et al. 2022.

According to Elijah et al. 2018 a Smart Irrigation System offers the following benefits: (i) Community Farming, (ii) Safety Control and Fraud Prevention, (iii) Competitive Advantages, (iv) Wealth Creation and Distributions, (v) Cost Reduction and Wastage, (vi) Operational Efficiency, (vii) Awareness and (ix) Asset Management.

WaterBit provides a "secured wireless connectivity to its autonomous irrigation solution, allowing management and control of local irrigation" (Obaideen et al. 2022). WaterBit gathers soil moisture levels measured every few minutes and presents theses measures via a mobile-friendly application where farmers can control the irrigation system.

Ipswich city council "depicted that using an automated soil-moisture monitoring system as a driver of irrigation leads to significantly conserve water along with saving costs" (Obaideen et al. 2022). The system's method to control the irrigation system was considered highly efficient when compared with a rainfall-based allocation method.

Maejo University in Thailand built a system to control the environment where mushrooms were cultivated. The system measured light, air temperature, air humidity and air flow to determine when to activate the irrigation system (Obaideen et al. 2022). The system was powered by solar energy.

AgirSens is a dynamic irrigation scheduling system based on IoT for efficient water management of irrigated crop fields. According to Roy et al. 2021 "AgriSens has a farmer-friendly user interface, which provides field information to the farmers in a multimodal manner - visual display, cell phone, and Web portal".

2.1.3.3 Fleet Management

This section focus on some Fleet Management and Asset Tracking solutions. No relevant papers were found regarding the use of IoT in Fleet Management or Asset Tracking solutions, therefore the solutions here presented were gathered by the author based on the quality of information available regarding their company's business model, iot devices used and features offered.

Verizon Connect is a Fleet Management solution that provides it's own sensors/devices, platform and application under a subscription (Verizon 2022). A team from Verizon installs the devices in the fleet according to the specification and access to the platform is given. This is an Hassle-free solution but the costs associated with it can be high. Since Verizon is a telecommunications conglomerate the protocols used to transport measures from devices to their platform is GPRS/2G/3G/4G/5G cellular.

SmartDrive is a Fleet Management solution that also provides it's own hardware, SR4, platform and application under a subscription. The SR4 is composed by a controller (solid-state storage, multiple communication protocols, advanced driver assistance system functions, computer-vision aided processing requirements), cameras, sensor bar (GPS, accelerometers, microphone and driver-support LEDs), keypad (for manual event recording and privacy mode initiation) and Wireless Key Fob (initiate a manual recording event, both inside and outside the vehicle) (SmartDrive 2018a). With all the data gathered, their platform, Transportation Intelligence Platform, is capable of "delivering breakthrough driving performance insights and analytic intelligence", and "helping fleets improve fleet safety and efficiency, and manage an entirely new set of business challenges arising in the future" (SmartDrive 2018b).

LocaTrack is a platform for Asset Tracking built by Safecube (Safecube 2021), this service provides the following Key Features:

- Geolocation: Distance, traveled road and availability;
- Asset use rate analysis: better visibility on the use of assets;
- Optimization: detailed analysis to optimize the use of assets;
- Real-time alert: real-time alerts on movements, geofencing, sleeping assets and others;
- Prediction: Maintenance, Servicing and Asset location;
- Condition monitoring: IoT trackers can detect changes in temperature, humidity or shocks and others.

According to Rogerson 2021, LocaTrack uses SigFox for lower-level communication and Azure IoT Hub to collect the location of tracked assets.

Finally, TrackPac is a service for asset tracking (TracPac 2022). This service offers an application that presents real-time location data, battery data and geofence alerts of the managed assets. By using LoRaWAN (provided by Helium, the same network used in this project) for lower-level communication it promises a service ten times cheaper than the ones that require 4G. This service works with LoRaWAN class A trackers and does not enforce the use of any proprietary, close-source devices even though it recommends the use of Browan Tabs Object Locator, Digital Matter Oyster3 or Abeeway Micro.

2.1.3.4 Summary

This section referenced some solutions found in academia and the market. Their descriptions alluded to how narrow-scoped IoT solutions tend to be, commonly requiring specific devices and cloud platforms to operate.

Next, some reference architectures for IoT will be introduced. These, among many things, intend to address the interoperability problems described above by introducing a common interpretation and vocabulary to the IoT landscape.

2.1.4 Reference Architectures

As the IoT domain covers such a wide spectrum of application fields with very little in common, the development cycles, technologies and architectures used can be completely

different. The vast array of choices given to those involved in a greenfield project of this area, coupled with the lack of standards with a broad usage (Dias, Restivo, and Ferreira 2022; Weyrich and Ebert 2015), can linger the design, development and interoperability of IoT systems.

Reference Architectures for the IoT aim to help developers tackle some of these problems (Weyrich and Ebert 2015).

According to Muller 2008, a Reference Architecture "captures the essence of the architecture of a collection of systems. The purpose of a Reference Architecture is to provide guidance for the development of architectures for new versions of the system or extended systems and product families".

This section sheds a light on some of the Reference Architectures of this field, what their focus is and the relevancy they have for this project.

The Reference Architectures discussed are: (i) IoT-A, (ii) SAT-IoT, (iii) IIRA, (iv) WSO2 IRA, (v) IEEE P2413, (vi) RAMI 4.0, (vii) Azure IRA, (viii) Arrowhead.

This section was based on the papers written by Lynn et al. 2020 and Dias, Restivo, and Ferreira 2022.

Intel System Architecture Specifications (Intel SAS) will not be discussed since no relevant information was found about it. The IoT-A Section also mentions the IoT Architectural Reference Model (ARM).

2.1.4.1 IoT-A

The IoT Architecture goals are to create "an architectural reference model for the interoperability of Internet-of-Things systems, outlining principles and guidelines for the technical design of its protocols, interfaces, and algorithms" that shall "lead to corresponding mechanism for its efficient integration into the service layer of the Future Internet". (European Lighthouse Integrated Project 2013a). This project's final version is dated back to July 2013.

It defines a collection of Unified Requirements that support and validate concrete architectures created according to the IoT ARM. Some of these requirements were also applied to this project.

According to Kro, Pokri, and Carrez 2014, this project motto is "to offer IoT architects a common technical grounding in order to optimize interoperability. In that case, IoT applications would not be any longer built as stand-alone silo applications, but as inter-operable vertical applications still having a common "horizontal" grounding - the ARM (compliant components, protocol suites, etc.)".

This project originated the IoT ARM that can be divided into three interconnected parts (Kro, Pokri, and Carrez 2014):

- The IoT Reference Model;
- The IoT Reference Architecture;
- A set of Guidance (also called best practice).

The reference model defines several models that help to describe certain aspects of the IoT architecture, some of this models are described by Haller et al. 2013. It defines five core concepts:

- Augmented Entity: a combination of a Physical Entity (real world object) and its Virtual Entity (digital representation);
- User: Those who interact with the system, human beings, devices, services and others;
- Device: Hardware to monitor or interact with real world objects;
- Resource: Computational element that gives access to information or control over a real-world object;
- Service: Entities that expose resources via a common interface, making then available for consumption by other services.

Each concept is them explored in detail. These concepts are them used to create the reference architecture.

According to *Introduction to the Architectural Reference Model for the Internet of Things* 2013b, “the Reference Architecture can be visualized as the "Matrix" that eventually gives birth ideally to all concrete architectures”. “Guidance in form of best practices can be associated to a reference architecture in order to derive use-case-specific architectures from the reference architecture”.

The IoT Reference Architecture provides a functional view, presented in the Figure 2.3.

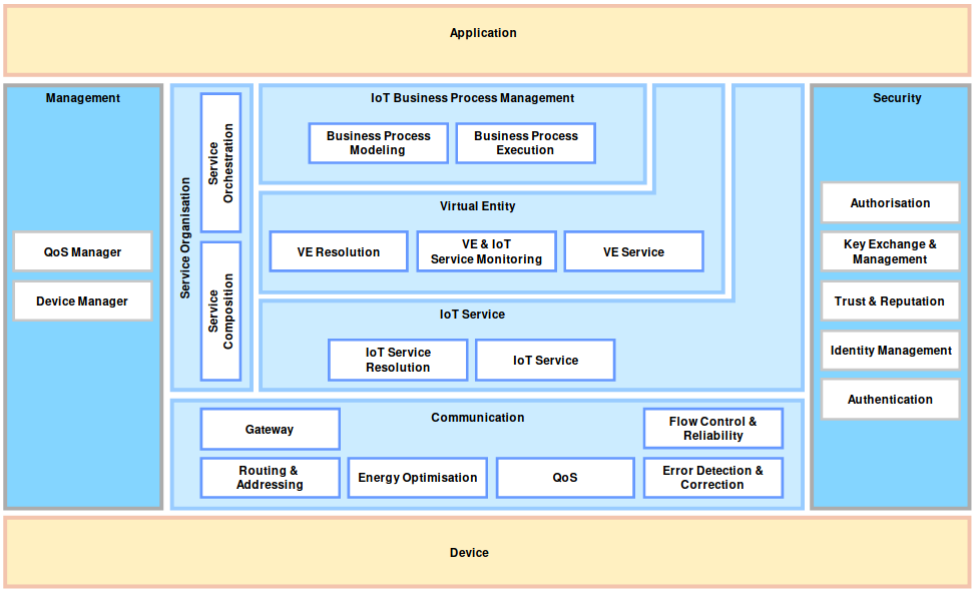


Figure 2.3: ARM Functional View, European Lighthouse Integrated Project 2013c

This functional view divides the architecture of a system in various functional components with well defined concerns.

2.1.4.2 SAT-IoT

López Peña and Muñoz Fernández 2019 present an architectural model definition that lead to the development of "a new advanced IoT platform referred as SAT-IoT". This model attempts to integrate concepts such as: "the paradigm of edge/cloud computing transparency, the IoT computing topology management, and the automation and integration of IoT visualization systems". This project's final version is dated back to 2019, it apeares that the envisioned platform was not implemented since no other reference to it was found.

The diagram in Figure 2.4 defines the concepts, services and relations of this model.

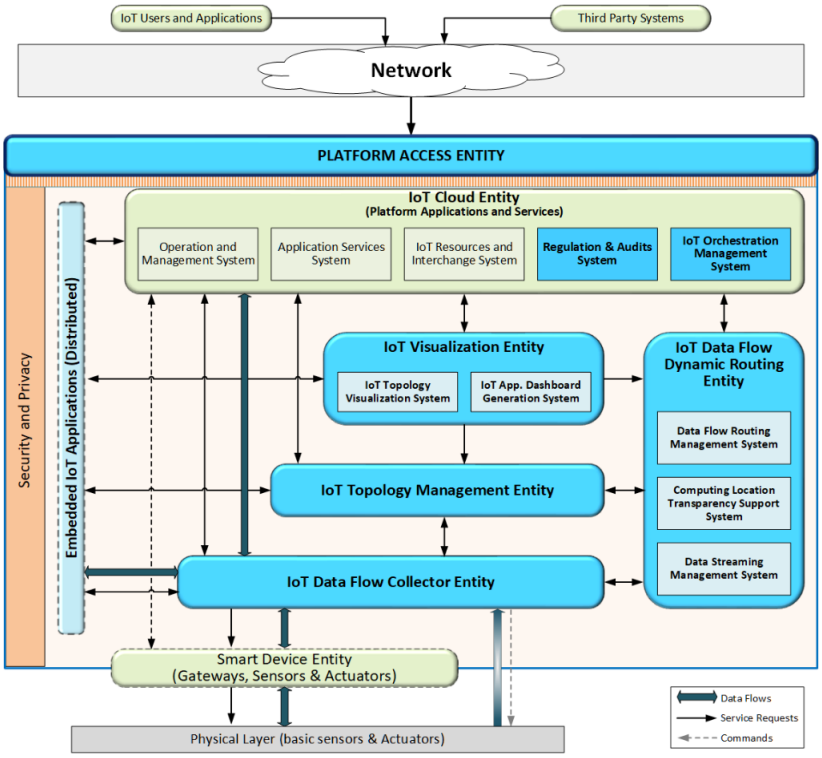


Figure 2.4: SAT-IoT Architectural Model, López Peña and Muñoz Fernández 2019

This model focus on a distributed system with a tree-like structure, where data can be dynamically processed in different node levels (edge, mid or cloud) in order to optimize response latency, bandwidth consumption, storage and other metrics. Components such as "IoT Topology Management Entity", "IoT Data Flow Dynamic Routing Entity" and "IoT Visualization Entity" focus on optimally distributing the workload across all nodes of the system.

The architecture also enables one to host "Embedded IoT Applications" that have full access to the system internals, leading to strongly integrated applications.

2.1.4.3 IIRA

The Industrial Internet Reference Architecture (IIRA) "addresses the need for a common architecture framework to develop interoperable IIoT systems for diverse applications

across a broad spectrum of industrial verticals in the public and private sectors to achieve the true promise of IIoT" (Industry IoT Consortium 2019). This project's final version is dated back to June 2019.

It decomposes a typical Industrial IoT system in five distinct functional domains:

- **Control Domain**: this domain focus on functions that are performed by industrial control and automation systems. It is deployed in proximity to the physical systems and therefore geographically distributed;
- **Operations Domain**: this domain focus on the management and operation of the control domain. It should be able to configure, register, track and control assets. It is also responsible for providing real-time prognostics, monitoring and diagnostics of the managed assets;
- **Information Domain**: this domain is responsible for managing and processing data, it should transform, persist, and model or analyze data to acquire high-level intelligence about the overall system;
- **Application Domain**: this domain is responsible for applying business focused rules and logic to the gathered information;
- **Business Domain**: this domain is responsible for implementing business processes, such as Enterprise Resource Planning, Costumer Relationship Management, Manufacturing Execution System, Billing and Payment, Work Planning and Scheduling Systems.

These domains interact according to Figure 2.5.

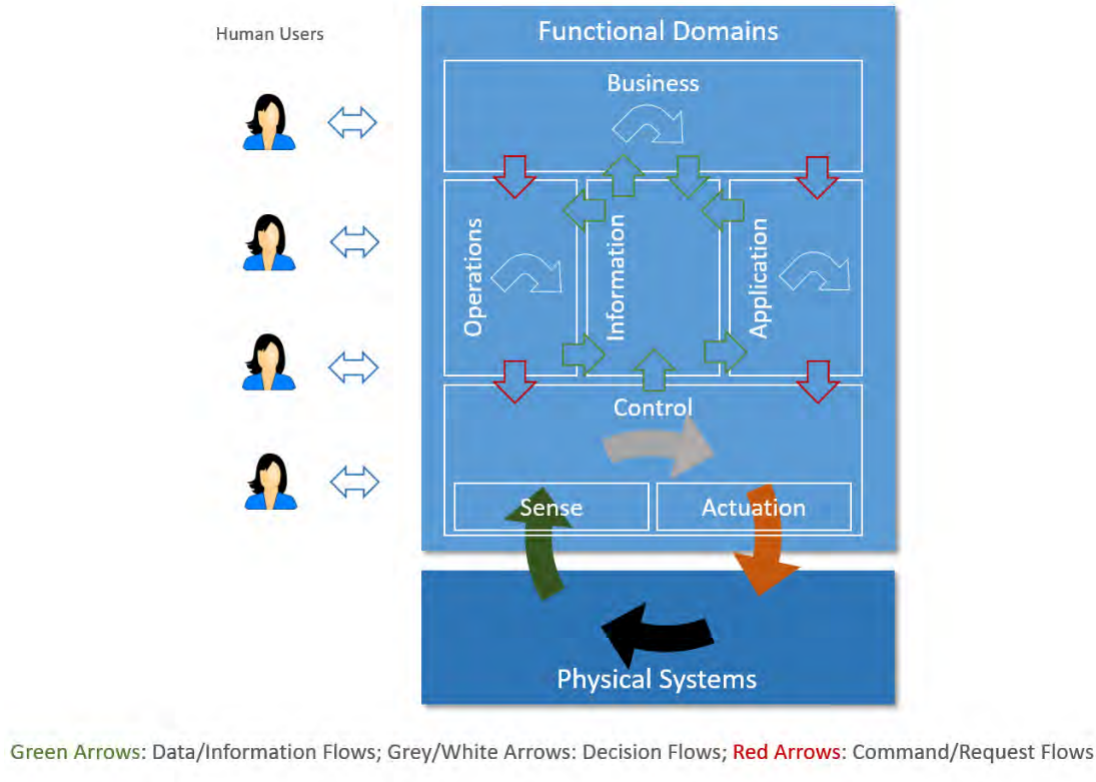


Figure 2.5: IIRA Functional Domains, Industry IoT Consortium 2019

As information flows from the control domain to the business domain it is enriched, cleaned, filtered and combined leading to a broader and richer notion of the complete environment. New information can be derived, and new intelligence may emerge from this broader information.

When applied to the common three tier architecture for IoT systems, these domains are organized according to Figure 2.6.

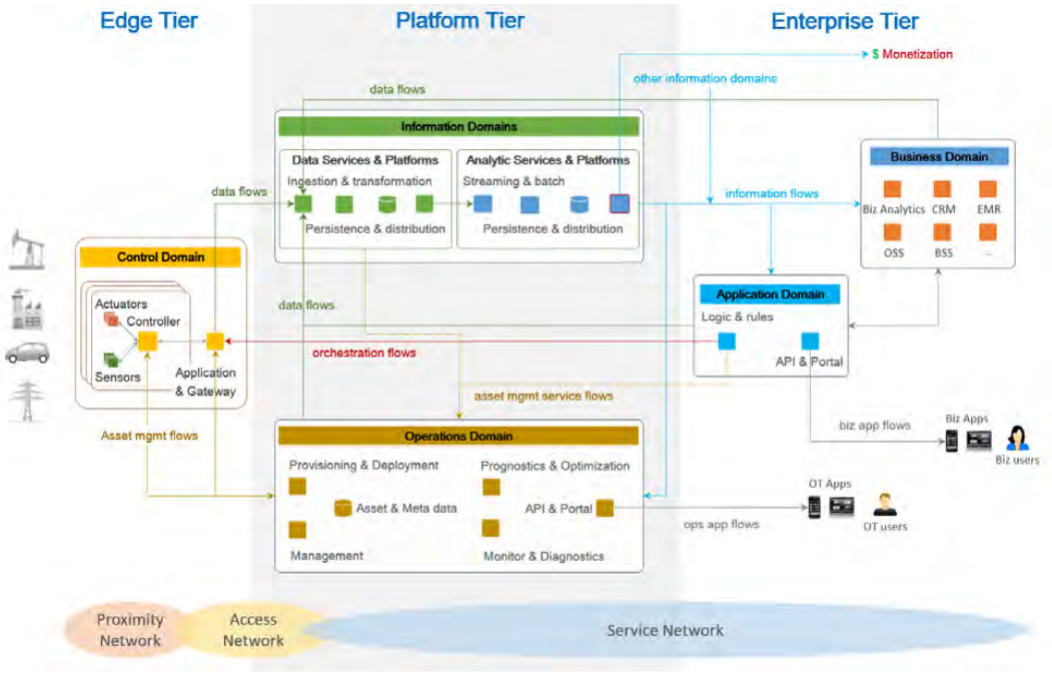


Figure 2.6: Mapping between a three tier architecture and the IIRA function domains, Industry IoT Consortium 2019

2.1.4.4 WSO2 IRA

The WSO2 reference architecture aims to "provide an architecture that supports integration between systems and devices" (WSO2 2015). This project's final version is dated back to October 2015.

It groups the IoT related requirements in the following key categories: (i) Connectivity and communications, (ii) Device management, (iii) Data collection, analysis, and actuation, (iv) Scalability, (v) Security, (vi) high-availability, (vii) Predictive analysis and (viii) Integration.

This categories gave birth to the following reference architecture, Figure 2.7.

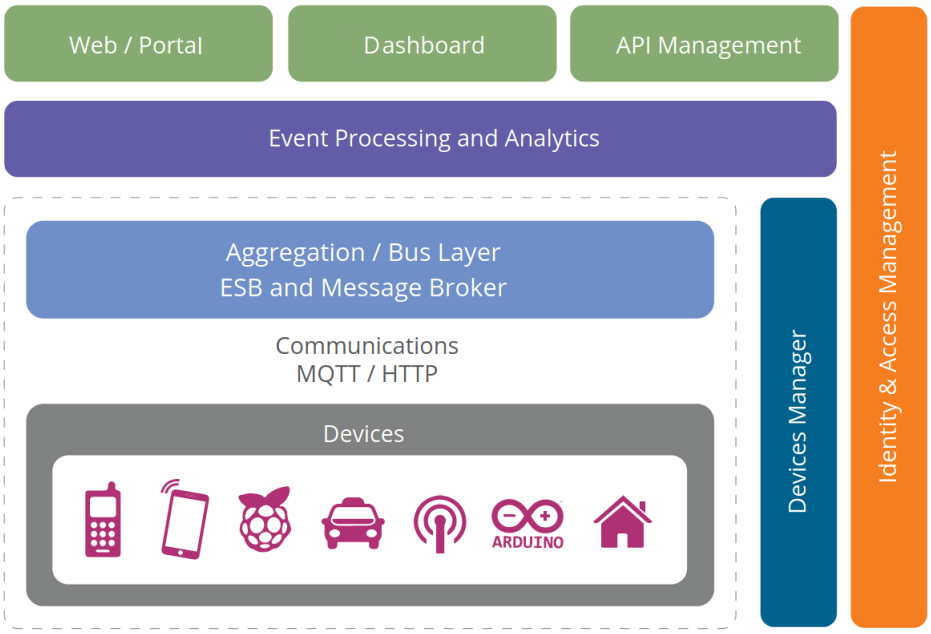


Figure 2.7: Reference Architecture for IoT, WSO2 2015

This reference architecture envisions two cross-cutting and five horizontal layers:

- **Device Layer** (in grey): related to the physical IoT devices;
- **Communications Layer** (without any representative color): related to the connectivity of devices;
- **Aggregation/bus Layer** (in light blue): related to the aggregation and supply of data upstream, bridging between the protocols used in upstream layers and downstream layers;
- **Event processing and Analytics Layer** (in purple): related to data processing, analysis and storage;
- **Client/external Communications Layer** (in green): related to web-based frontends and portals that interact with the Event processing and Analytics layer, dashboards that offer views into analytics and event processing, and APIs for machine to machine communication;
- **Device Management Layer** (in dark blue): related to the management, onboarding and remote control of devices;
- **Identity Access Management** (in orange): related to the authentication and authorization of users and systems that interact with the system.

In the Event processing and analytics Layer it's recommended the use of "a highly scalable, column-based data storage for storing events", "map-reduce for long-running batch-oriented processing of data", "complex event processing for fast in-memory processing and real-time reaction and autonomic actions based on the data an activity of devices and other systems" and "traditional application processing platforms" (custom-made applications for data processing).

2.1.4.5 IEEE P2413

The IEEE Standard for an Architectural Framework for the IoT "defines an architecture framework description for IoT". The architecture framework defined in the standard "will promote cross-domain interaction, aid system interoperability and functional compatibility, and further fuel the growth of the IoT market" (IEEE 2020). This project's final version is dated back to June 2020.

Its architecture framework covers the definition of basic architectural building blocks and their ability to be integrated into multi-tiered systems. It describes different detailed viewpoints of the framework (IEEE 2020):

- **Conceptual Viewpoint**: concerned with defining a common vocabulary and semantics regarding a IoT System to ease the communication across teams and encourage the reuse of concepts;
- **Compatibility Viewpoint**: concerned with the compatibility between systems and devices to lower the cost of integration. This viewpoint urges for the creation of new standards and compliance with those standards. It defines six levels of compatibility focused specially on physical devices: (i) incompatible, (ii) coexistent, (iii) inter connectable, (iv) inter workable, (v) interoperable and (vi) exchangeable;
- **Lifecycle Viewpoint**: concerned with a system's assurance, performance, maintainability and evolvability across its lifecycle: design, development, production, support, upgrade and retirement;
- **Communication Viewpoint**: concerned with how devices can exchange information with each other and IoT systems in a accurate, precise and effective manner;
- **Information Viewpoint**: concerned with how information is semantically defined, structured, stored, shared, manipulated, managed, and distributed across the IoT system. This viewpoint should focus on documenting system-level information, e.g., information exchanged between the various subsystems;
- **Function Viewpoint**: concerned with how devices can function according to their intended purpose or characteristic action, such as actuation, sensing, analysis, or control of entities of interest;
- **Thread model Viewpoint**: concerned with identifying potential threats that could exploit vulnerabilities in the device, network or subsystems that encapsulate the IoT system;
- **Security and safety monitoring Viewpoint**: concerned with monitoring the events occurring in an IoT system and analyzing them for signs of possible incidents, which are violations or imminent threats of violation of security, safety, or acceptable use policies, or standard security practices;
- **Access control Viewpoint**: concerned with permitted activities of legitimate users, mediating every attempt by a user to access a resource in the IoT system. It is composed by three security functions: identification, authentication and authorization;
- **Privacy and trust Viewpoint**: concerned with the privacy of individuals or groups and trust in systems or organizations. In a complex IoT system, arbitrary device data can be grouped and analyzed to determine the users activities;

- **Collaboration Viewpoint**: concerned with the collaboration of systems that belong to different application domains;
- **Computing resource Viewpoint**: concerned with the computing resources needed to support the IoT system as a whole, such as gateways, data centers, Programmable Logic Controller (PLC)s and Distributed Control System (DCS) controllers, microcontrollers embedded in sensors and actuators or others.

The idealized IoT System should be examined according to these viewpoints in order to better define its architecture.

IEEE 2020 then procedes to define the Standard for a Reference Architecture for Smart Cities in P2413.1, the major focus of this project's business cases.

One of the architectures proposed in the standard and derived from the architecture framework is presented in Figure 2.8.

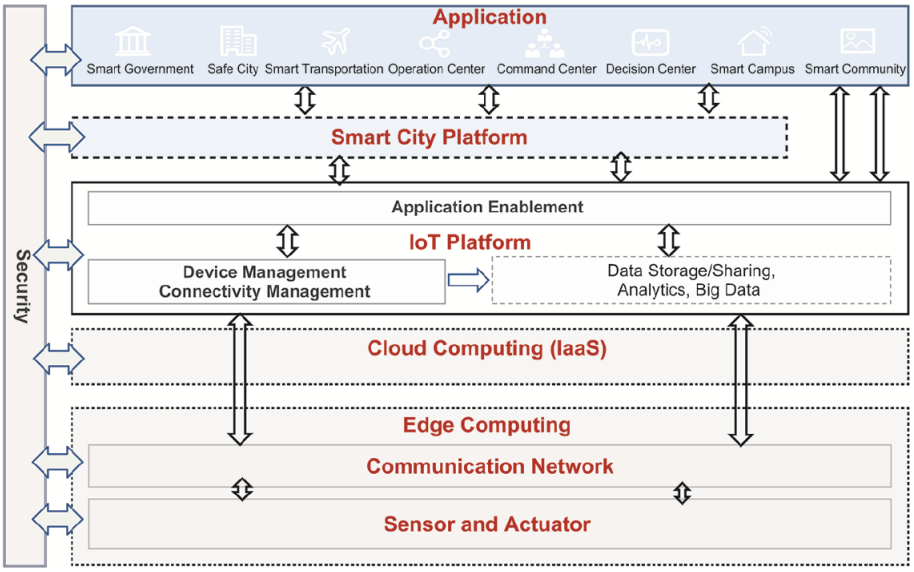


Figure 2.8: Example of an IoT System Architecture for Smart Cities, IEEE 2020

This architecture, derived from the common three tier architecture for IoT Systems, proposes a new tier entitled Smart City Platform, in it "Northbound APIs support diverse vertical applications development and southbound APIs connect to different IoT Platforms" (IEEE 2020).

2.1.4.6 RAMI 4.0

The Reference Architectural Model Industry 4.0 "ensures that all participants involved share a common perspective and develop a common understanding" and is represented by a "three-dimensional map showing the most important aspects of Industrie 4.0" (Hankel and Rexroth 2015). It represents a service-oriented architecture according to the manufactures association that defined it. This project's final version is dated back to August 2018 and has clear focus on the IoT business area related to the Industry, e.g. smart factories.

The three-dimensional map is depicted in Figure 2.9.

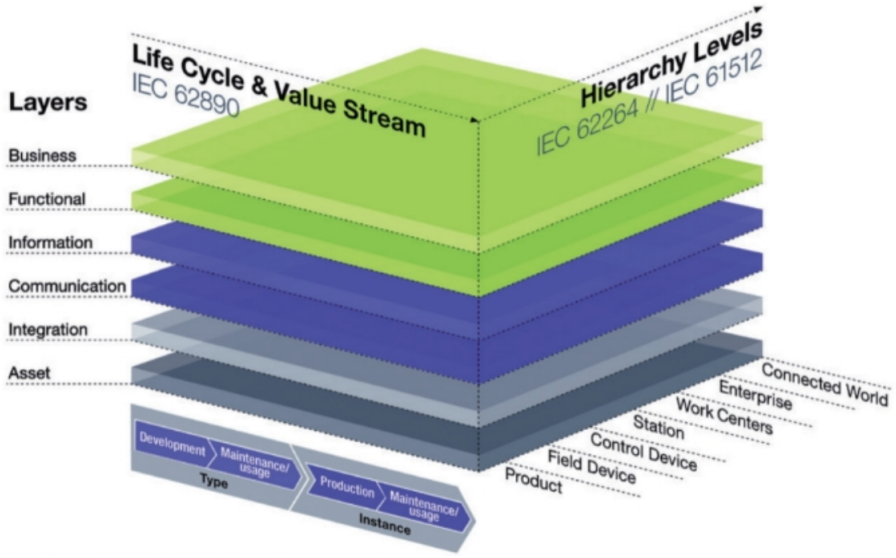


Figure 2.9: RAMI 4.0 Three-dimensional map, German Electrical and Electronic Manufacturers' Association 2017a

According to Kaviraju 2021 it is comprised of six architecture layers stretched across the hierarchy and life cycle axes:

- **Business Layer**: concerned with Organization and Business processes;
- **Functional Layer**: concerned with the Functions of assets;
- **Information Layer**: concerned with the processing of the necessary data;
- **Communication Layer**: concerned with how to gain access to the information needed;
- **Integration Layer**: concerned with the transition from things to the digital world;
- **Asset Layer**: concerned with the physical things in the real world.

This reference architecture mentions an administration shell that sits in between the asset (machine, sensor, unit or plant) and the network. This administration shell is the interface connecting the IoT platform to the asset, storing all data and information about the asset and standardizing the network's communication. According to German Electrical and Electronic Manufacturers' Association 2017b, "each physical thing has its own administration shell" and "several assets can form a thematic unit with a common administration shell". This administration shell allows for distributed data analysis and control over assets.

According to Lin et al. 2017, this reference architecture is aligned with IIRA. The following picture, Figure 2.10 describes how RAMI 4.0 concepts can be represented according to IIRA.

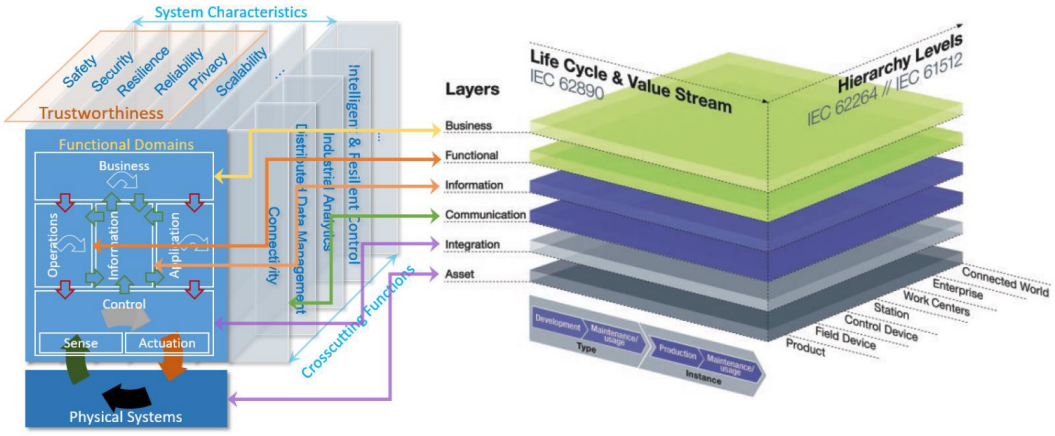


Figure 2.10: IRRA and RAMI 4.0 Functional Mapping, Lin et al. 2017

The need to map the concepts of both reference architectures derives from the fact that these two are the most actively references used in the industry (Dias, Restivo, and Ferreira 2022). Therefore, the document by Lin et al. 2017 provides guidelines on how to achieve better interoperability between systems built according to different reference architecture.

2.1.4.7 Azure IRA

The *Microsoft Azure IoT Reference Architecture* 2018 proposes the architecture envisioned in Figure 2.11, this architecture relies heavily on the Azure platform services. According to Microsoft 2018, the recommended architecture is "cloud native, microservice, and serverless-based". This project's final version is dated back to April 2021.

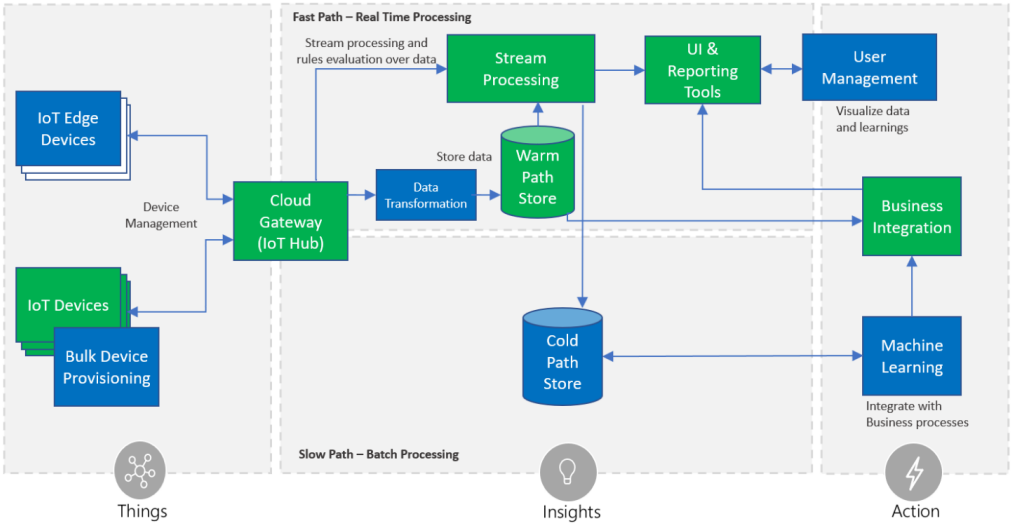


Figure 2.11: Azure IoT Reference Architecture, Microsoft 2018

Microsoft 2018 recommends that "subsystems should be built as discrete services that are independently deployable, and able to scale independently", to "enable greater scale, more flexibility in updating individual subsystems, and provide the flexibility to choose appropriate technology on a per-subsystem basis".

This reference architecture is based on the lambda architecture. According to Kiran et al. 2015, the lambda architecture "combines both batch and stream processing capabilities for online processing and handling of massive data volumes in a uniform manner, reducing costs in the process." It is comprised of three layers (or patches), a Batch or Slow layer for extensive and prolonged analysis, a Speed or Fast layer for real-time evident information, and a Serving layer responsible for providing the results gathered by the other two layers.

As we can see in Figure 2.11 the Insights section is divided into two patches, the Fast Patch for real-time processing, and the Slow Patch for batch processing. Results are then provided in the Action section.

2.1.4.8 Arrowhead

According to Varga et al. 2017, "the objective of the Arrowhead Framework is to efficiently support the development, deployment and operation of interconnected, cooperative systems. It is based on the Service Oriented Architecture (SOA) philosophy". It has a big focus on Interoperability between systems and services already in production.

"The Arrowhead project targets five business domains; Production (process and manufacturing), Smart Buildings and infrastructures, Electro mobility, Energy production and Virtual Markets of Energy" (Blomstedt et al. 2014).

The Arrowhead challenge is to enable interoperability between these systems, therefore, it starts by defining how one should document his/her solutions. Three hierarchical levels of solutions are proposed: (i) service, (ii) system, (iii) system of systems (Figure 2.12).

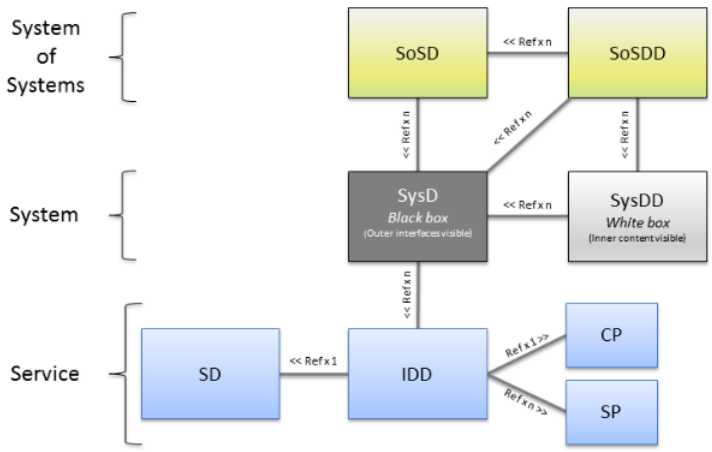


Figure 2.12: Arrowhead Framework Core and Application Services, Blomstedt et al. 2014

The lower level, service, can be something that, for example, indicates the current measured humidity level by sensor X (pull-typed service) or that opens/closes valve Y (push-typed service). A system is composed by several services. A System of Systems is composed by several systems that work in harmony in a Arrowhead local cloud. A Arrowhead local cloud is composed by at least three mandatory core systems: (i) ServiceRegistry system, (ii) Authorization system and (iii) Orchestration system presented in Figure 2.13.

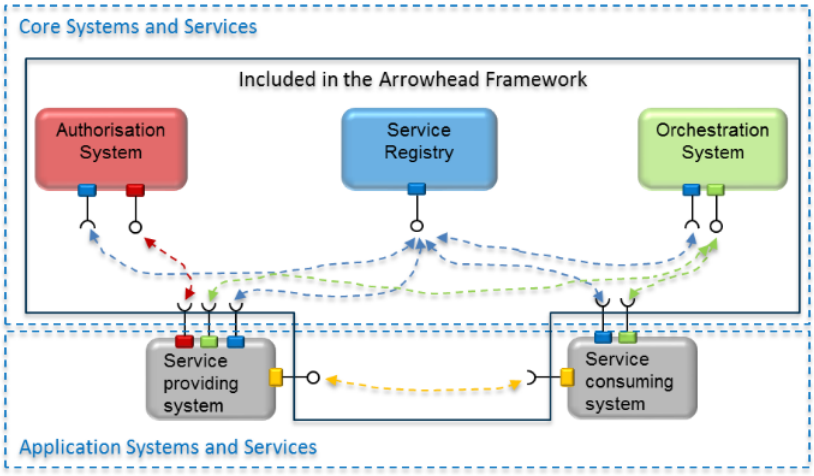


Figure 2.13: Arrowhead Framework Core and Application Services, Blomstedt et al. 2014

These core systems handle essential features for a local cloud such as: service discovery, service registration, service advertisement, authentication of consuming services, data exchange between systems and service coordination according to Marcu et al. 2020. A local cloud can expose its systems to other local clouds and consume other local clouds systems. Since each service and system is well documented and arrowhead compliant it's possible to ensure interoperability between local clouds.

2.1.4.9 Overall Perspective

To close this section, some of the author sentiment and ideas surrounding these reference architecture models, and how they may shape this project's solution, are presented:

- IoT-A: The Unified Requirements detailed by this initiative gave the author an idea of the basic features for an IoT system and enriched the requirements proposed by the company;
- SAT-IoT: The extendability notion behind the "Embedded IoT Application" component was interesting to author from a business point of view. This idea would give customers the possibility to integrate custom-made solutions in the platform;
- IIRA: The author argues that the clear division between the Application, Information and Business domains provide a common abstraction that can be applied to any IoT System to ease the cognitive burden taken to understand it;
- WSO2 IRA: The author finds the clear division between the responsibilities of gathering measures, processing and analyzing them, and providing them in a business focused manner very aligned with this project's goals;
- IEEE P2413: The concept of "Smart City Platform", and how it interacts with various systems is highly related with the overall requirements of this project;
- RAMI 4.0: This reference architecture provided no relevant insight for the author, mostly due to the discrepancy between the IoT domains each project was devoted to (Smart Cities vs Industry 4.0);

- Azure IRA: This reference architecture presented the author with a conceivable architecture for the future of the current solution. It introduced the notion of a Batch layer to better handle complex and prolonged analysis and provide Key Performance Indicators (KPI) reports;
- Arrowhead: This reference architecture appeared to be out-of-scope for the project since it focus on a more fine-grained access and control of devices (without the need for a IoT Midlleware that handle device management and propagation of measures).

2.1.5 Synopsis

This section introduced the reader to the technological landscape of IoT. The next section discusses some of its business areas or domains.

2.2 Business Areas

Even though there's no concise structure, it is obvious that the IoT technologies can be used in a broad range of areas/sectors. As per Nieti, Djilali, et al. 2019, the most valuable areas are: Smart Cities, Industrial IoT, Connected Health and Smart Homes. The general market division of IoT technologies is presented in Figure 2.14.

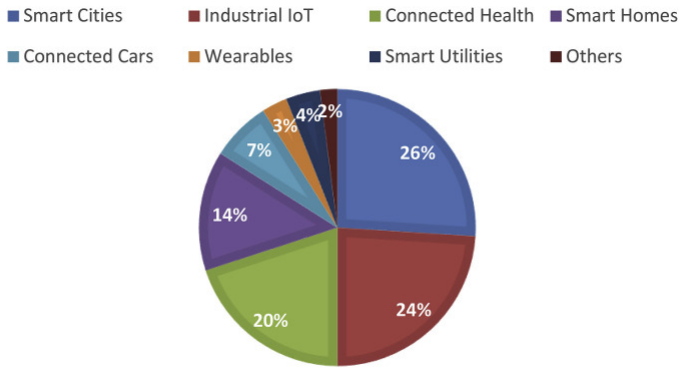


Figure 2.14: General market structure of IoT technologies, Nieti, Djilali, et al. 2019

From another point of view, and according to Gazis et al. 2015, the sectors IoT is related to are: Energy, Smart City, Transportation, Smart Home, Environment, Supply Chain, and Health Care.

According to S. Chen et al. 2014 these are the main application fields for IoT in China: industry, smart agriculture, smart logistics, intelligent transportation, smart grids, smart environmental protection, smart safety, smart medical and smart home.

Even though this work focus mostly on Smart Cities other areas are also be described. Each of this areas incorporate several interconnected use cases that are briefly described in the following segments.

2.2.1 Smart Cities

The Smart Cities sector includes numerous use cases related to public safety, the environment, mobility, energy, infrastructure and many other municipal concerns. According to Wegner 2020 this are the use cases being prioritized.

- Connected Public Transport: real-time monitoring of public transportation vehicles' locations, stops and itineraries, and the possibility to be notified when a public transportation vehicle is arriving at a stop;
- Traffic Monitoring and Management: real-time monitoring and management of traffic flows in a efficient manner;
- Water level / Flood Monitoring: real-time monitoring of level of water in public water basins such as rivers, channels, or even lakes and seas to warn and predict fast water level shifts;
- Video Surveillance & Analytics: real-time monitoring using CCT cameras and analytics to detect specific situations, e.g. accidents, crimes, potential threats, or recognize specific features (face recognition, demographics, etc.);
- Connected Streetlights: real-time monitoring and management of streetlights' health status and energy consumption to decrease costs and become more sustainable;
- Weather Monitoring: real-time monitoring of weather conditions such as temperature, humidity, rainfall, wind speed and direction to predict the weather and future natural disasters;
- Air Quality / Pollution Monitoring: real-time monitoring of air quality to warn the community about hazardous conditions;
- Smart Metering - Water: remote real-time monitoring of water usage in homes to address the world's water demand and scarcity issues and faster localize sewage leaks;
- Fire / Smoke Detention: real-time monitoring of possible indoor fires and CO2 levels to prevent injuries, fatalities and building degradation;
- Water Quality Monitoring: real-time monitoring of water conditions such as pH levels, percentage of salts and other elements that can threaten the public health.

Apart from these use cases, others are arising, such as smart parking (Goap et al. 2018), smart irrigation (Khanna and Anand 2016) and waste management (Shyam, Manvi, and Bharti 2017).

- Smart parking provides a simple method to the community of knowing the available parking spots, which, alone, lowers the carbon footprint and traffic congestions in cities.
- Smart irrigation tackles the need to save water by irrigating the soil only when needed and not when it is already moist, it's raining or it is expected to rain in the following hours.
- Waste management can eliminate the cost of unnecessary waste collections and therefore reduce the carbon footprint. Data gathered can then help to identifying cost-effective itineraries to collect waste and eventually lower overall transportation and staff costs.

All this use cases refine the efficiency of the municipal workforce and help the town council to reduce costs and improve the environment sustainability in the long term.

2.2.2 Industry

According to Gilchrist 2016, “the Industrial IoT provides a way to get better visibility and insight into the company’s operations and assets", therefore this leads to “operational efficiency gains and accelerated productivity, which results in reduced unplanned downtime and optimized efficiency, and thereby profits”. It is comprised of several use cases (Tracy 2017) such as:

- Predictive Maintenance: real-time monitoring of equipment conditions and applied data analytics can help a company to significantly decrease operational expenditures. “Other potential advantages include increased equipment lifetime, increased plant safety and fewer accidents with negative environmental impact” (Tracy 2017);
- Smart metering: real-time monitoring of energy, water or natural gas consumption of a building can reduce operating expenses by managing manual operations remotely, reduce energy theft and improve forecasting and streamline power-consumption (SierraWireless 2017);
- Asset tracking: real-time monitoring of resources helps "to easily locate and monitor key assets, along the supply chain (e.g. raw materials, final products and containers) to optimize logistics, maintain inventory levels, prevent quality issues and detect theft" (Tracy 2017).
- Connected vehicles: computer-enhanced vehicles that automate many normal driving tasks can lower crash rates, and help decreasing the number of vehicles a company needs to function.
- Fleet management: real-time monitoring of vehicles location and conditions can help “improving efficiency and productivity while reducing overall transportation and staff costs” (Tracy 2017).

As we can see from the list above, the Industrial IoT sector is focused on business efficiency and staff safety, which, as a side effect, brings environmental benefits.

2.2.3 Healthcare

According to Firouzi et al. 2018 new opportunities are now arising as a result of fast-paced expansion in the areas of the IoT and Big Data for healthcare industries. People across the globe have begun to adopt wearable biosensors, whose data is feed into the new emerging individualized health applications. This sector incorporates numerous use cases (Kumar and Chatterjee 2020) such as:

- Remote Healthcare Monitoring: real-time monitoring of a patient conditions such as pulse rate and heartbeat can prevent unwanted deaths;
- Drug management: medicine monitoring and reminder system can help the elderly to take medicine on time;
- Employee health management: real-time monitoring of employee’s state can predict burnouts and increase a workforce productivity;

The benefits these use cases provide are a more convenient lifestyle, improvement of one life's quality, reduction in costs and increased survival rates of patients (Kumar and Chatterjee 2020).

2.2.4 Smart Homes

Visions of smart homes have long caught the attention of researchers and considerable effort has been put toward enabling home automation. However, these technologies have not been widely adopted despite being available for over three decades (Brush et al. 2011). Based on Alaa et al. 2017 most home automation services offer the following use cases:

- Smart Lighting: remote and automated control of lights inside a house can help to decrease energy wasted;
- Smart Air Conditioning: remote and automated control of air conditioners can keep the house comfortable while minimizing the energy wasted;
- Remote health monitoring: when dealing with the elderly, complex smart systems can anticipate their needs without direct human intervention;
- Device Automation: smart systems can turn the lights off when no one is home, open the door when an identified person arrives and must more, improving the overall comfort of the residents.

A smart home delivers various benefits such as reducing energy waste, comfort, allowing remote control of the house, monitoring of elderly patients and easy communication with health institutions (Alaa et al. 2017).

2.2.5 Open Challenges

Even though it seems IoT is the obvious next step for the industry, healthcare, everyone's home, public spaces/services and everything else there are some obstacles to overcome.

One of the big challenges ahead of everyone is related with antiquated ideas, tools and processes still in use today. Each of the use cases above mentioned require a big shift in how a company works since it demands a modernization of the organization infrastructure. Tapscott and Williams 2006, explained that "In an age where mass collaboration can reshape an industry overnight, the old hierarchical ways of organizing work and innovation do not afford the level of agility, creativity, and connectivity that companies require to remain competitive in today's environment".

According to Gazis et al. 2015 this are the most important challenges regarding IoT applications:

- Technological Interoperability: achieving a seamless interaction between devices and people with devices (according to Al-Qaseemi et al. 2016 there's a lack of standardization in IoT devices and technologies);
- Semantic Interoperability: guaranty that the devices interpret the shared information correctly and act accordingly (improvements have to be made regarding distributed ontologies, semantic web, or semantic device discovery);
- Security and Privacy: improving data integrity, unique device identification, encryption and implement proper data/device ownership for legal/liability issues;

- Smart Things: ultra low power circuits and devices capable of tolerating harsh environments have to be developed;
- Resilience and Reliability: in industrial environments or in emergency use cases temporary outages cannot be accepted.

According to the author this challenges substantially lingered the growth of IoT, an area that was expected to have a much bigger impact in day-to-day life of everyone. According to Dave Evans 2011 there would be 50 billion of devices connected to the Internet by 2020 but Statista 2021 reported only 8.74 billion of connected devices.

Noura, Atiquzzaman, and Gaedke 2019 introduced more issues in IoT related to interoperability from different perspectives:

- Device interoperability: concerned with the exchange of information between heterogeneous devices and the ability to integrate new devices into any IoT platform;
- Network interoperability: concerned with information addressing, routing, security, resource optimization, Quality of Service (QoS) and mobility support;
- Syntactical interoperability: concerned with the format and structure of the information exchanged between heterogeneous systems;
- Semantic interoperability: concerned with the meaning behind the information exchanged, heterogeneous devices can, for example, work with diverse unit measurements;
- Platform interoperability: concerned with heterogeneous platforms that use diverse programming languages, Operating System (OS) and software architectures (also mentioned by Dias, Restivo, and Ferreira 2022; Koo and Kim 2022; Ray 2016; B. N. Silva, Khan, and K. Han 2018).

For IoT Technologies to deliver on the promises made by companies like Cisco or Gartner, these barriers must be surpassed.

2.3 Synopsis

This chapter presented the big theme surrounding this work: IoT. Major business areas and relevant solutions/technologies for this work were introduced.

In the following chapter, Requirements Elicitation, some of the business cases and challenges discussed here will be tackled.

Chapter 3

Requirements Elicitation

In this chapter the functional and non-functional requirements will be presented.

"A software requirement is a capability needed by the user to solve a problem or to achieve an objective. In other words, requirement is a software capability that must be met or possessed by a system or system component to satisfy a contract, standard, specification, or other formally imposed documentation. Ultimately, what we want to achieve is to develop quality software that meets customers' real needs on time and within budget." (Paradigm 2020).

The project high-level goal was well defined since the start:

Develop an IoT Platform with focus on extensibility to decrease the delivery time of new business cases and allow others to implement their business on top of the platform.

The definitive business cases to develop changed various times during the project lifespan due to intricate contract promises with third parties that never ended up seeing the light of day. The business cases, ordered by the first time they were requested, and grouped by organization, can be summarized in Table 3.1.

These business cases can be vaguely characterized according to the following organization's needs:

- **Deployment Environment**: Should the solution be deployed on-premise or in the cloud;
- **Multi-Tenancy**: Should the organization share an instance of the solution with others or not (according Gillis 2020);
- **Data Shareability**: Does the organization wants to provide their data to the public (according to Yoon et al. 2017);
- **Information Access and Visualization**: Where and how to present and serve information. Present information visually in the costumer organization platform, directly in this solution or via other means such as a simple API, SMS, or email.

Table 3.1: Summary of the main requirements of the requested business cases

Org	Business Case	Deployment Environment	Multi-Tenancy	Data Shareability	Information Access and Visualization
A	Fleet Management	On-Premise	Single-Tenant	Private	Sensae Console
	Smart Irrigation	On-Premise	Single-Tenant	Private	Sensae Console
	Smart Parking	On-Premise	Single-Tenant	Private	Sensae Console
	Indoor Fire Detention	On-Premise	Single-Tenant	Private	SMS and Email
	Public Health Surveillance	On-Premise	Single-Tenant	Public	Sensae Console
B	Fleet Management	Cloud	Single-Tenant	Private	Org B Platform
C	Smart Irrigation	Cloud	Multi-Tenant	Private	Sensae Console
	Indoor Fire Detention	Cloud	Multi-Tenant	Private	SMS and Email
	Chicken Farm Monitoring	Cloud	Multi-Tenant	Private	Sensae Console
D	Smart Irrigation	Cloud	Multi-Tenant	Private	Sensae Console

The requirements detailed in the following sections were founded on top of the requested business cases mentioned above. These requirements were constantly tailored according to the latest talks with the third parties involved. Even though many requested business cases weren't implemented, they guided the author to the design and development of the final solution, **Sensae Console** and PoCs.

At the time of writing, the PoCs developed answer three business cases: (i) Fleet Management, (ii) Smart Irrigation and (iii) Indoor Fire Detention. The other business cases were either abandoned or requested too close to the writing of this dissertation and therefore will not be detailed.

3.1 Functional Requirements

Functional Requirements define the user-faced functionalities/operations that the solution to develop must support in the future.

According to Van Lamsweerde 2009, "Functional requirements define the functional effects that the software-to-be is required to have on its environment. The effects characterized by such requirements result from operations to be automated by the software. Functional requirements may also refer to environmental conditions under which operations should be applied."

The following sections describe the requirements associated with each role inside **Sensae Console**, the solution that this project aims to deliver, and the PoCs developed, refereed as **External Services**.

3.1.1 Roles

The meetings that took place during this project's time span lead to the definition of three main roles:

- **Manager**: a role with full control over the **Sensae Console** and all its data. He/She has also full control of all **External Services**;
- **Costumer**: a role with restricted control over **Sensae Console**, controlling only the devices, employees and departments registered under his/her own organization. He/She has access to the requested **External Services**;
- **Anonymous User**: a role with no account in the system. He/She has access to the publicly available **External Services** and data feed from *'public'* devices in the system.

Apart from the basic costumer requirements inside **Sensae Console**, each **External Service** has specific use cases that will be detailed in the section 3.1.2.2.

Essentially, the difference between these roles boils down to what permissions each has and the extent of data each one can visualize. The Section G.4 details how this is handled by the solution.

The following sections will be divided in:

- Sensae Console: presenting the functional requirements associated with each role;
- External Services: presenting the functional requirements associated with each business case supported.

3.1.2 Sensae Console

The idea behind Sensae Console functional requirements boils down to the core functionalities it should provide so that creating and maintaining external services is simplified.

The Anonymous User role is disregarded here since his/her goal is to simply benefit from curated and publicly available information provided by the external services.

3.1.2.1 Manager

The purpose of the Manager is to supervise an instance of **Sensae Console** and its costumers. This role is an extension of the Costumer role and can do and see everything a Costumer can. A Manager is assign to an instance of **Sensae Console** at creation time and belongs to the highest domain, the *Root Organization* as described at Section G.4.

The following list documents the functional requirements related to this actor regarding the **Sensae Console** administration:

1. The Manager must be able to create, view, update and delete device payload decoders;
2. The Manager must be able to create, view, update and delete device payload processors (or mappers);
3. The Manager must be able to create, view, update and delete rules that trigger alerts;
4. The Manager must be able to define, view, update and remove device specific information;

5. The Manager must be able to define the permissions of any organization;
6. The Manager must be able to assign new devices to a specific organization;
7. The Manager must be able to assign new authenticated users to a specific organization.

As described in Sections G.1 and G.2, the decoders and processors referenced in the first and second items are meant to translate unsanitized device data. This is highly required since "the nonexistence of interoperability standards is one of IoT's most pressing issues, (...) designing a system using the latest available standard proposal does not ensure its adoption or that the standard will be deprecated before the system reaches the market" - Dias, Restivo, and Ferreira 2022.

The rules referenced in the third item can be used to program how the system answers to certain abnormal occurrences, more context is given in Section G.5.

The device information mentioned in item four is detailed in Section G.3.

Even though the first four groups of operations belong to the Manager role, they can be assigned to normal Costumers on special occasions. As an example, the Organization A and B referenced in Table 3.1, had employees capable of fully managing the solution and wanted an instance of **Sensae Console** exclusively for them. This meant that, when given access to these operations, there was a lower risk for them to misconfigure the platform due to a lack of knowledge and no risk to interfere with other Organizations' data pipeline, since they were the only ones in that instance.

3.1.2.2 Costumer

The purpose of a Costumer is to manage his/her own organizations. The following list documents the universal functional requirements related to this role:

1. A Costumer must be able to create and remove a department under his/her organization;
2. A Costumer must be able to define the permissions for all other Costumers in a department under his/her organization;
3. A Costumer must be able to assign and move another Costumer from/to a department under his/her organization;
4. A Costumer must be able to move a sensor from one department to another department under his/hers organization.

3.1.3 External Services

This section describes the functional requirements associated with each external service needs from the point of view of a costumer.

The Anonymous User role was created to answer organization A concerns regarding the Public Health Surveillance business case. The external service should be available for the public to consult the current and past Air Quality Index (AQI) levels measured in the city without needing to create an account. Even though this business case was abandoned, the Anonymous User role was integrated in the solution.

Each supported external service has specific use cases defined below.

3.1.3.1 Fleet Management

Within a simple Fleet Management business case the major utilities a Costumer can benefit from are: (i) real-time tracking of his vehicles and (ii) visualizing past data regarding the whereabouts of his fleet. A more advanced Fleet Management would for example provide KPI reports about the fleet or alerts when a vehicle would enter or leave a geofence. This advanced topics were mentioned by organization A close to the day when they withdrawn the contract and therefore were never implemented.

The following list documents the key functional requirements of this business case as prescribed by the third parties:

1. A Costumer must be able to track in real-time a vehicle location and motion status;
2. A Costumer must be able to see the itineraries of a vehicle in defined time span;
3. A Costumer must be able to see where, when and for how long a vehicle was parked;
4. A Costumer must be able to see the traveled distance of a vehicle, in a defined time span.

This business case' concepts are discussed with more detail in Section H.1.

3.1.3.2 Indoor Fire Detention

An Indoor Fire Detention system usual main objective is to trigger an alarm when precarious conditions are meet. As a first milestone, both companies, A and C, requested a simple alarm system with no other features. Features such as data retention, data visualization and continuous camera vigilance were later requested. As such, the only requirement related to this business case is:

1. A Costumer must be able to receive alerts regarding critical conditions that may indicate a fire outbreak, either via SMS or email.

3.1.3.3 Smart Irrigation

Within a Smart Irrigation business case the major utilities a Costumer can benefit from are: (i) real-time tracking of a garden/greenhouse conditions, (ii) archiving conditions for later use/consulting and (iii) activate/deactivate the irrigation system remotely.

The following list documents the key functional requirements related to this business case as prescribed by the third parties:

1. A Costumer must be able to manage his/her garden's information;
2. A Costumer must be able to track a gardens' conditions in real-time;
3. A Costumer must be able to see past conditions of a garden;
4. A Costumer must be able to activate and deactivate the irrigation system remotely.

The concepts surrounding this business case are discussed with more detail in Section H.3.

3.2 Non Functional Requirements

Non-functional requirements define constraints on software development, maintenance, and allocation. According to Van Lamsweerde 2009, Non-functional requirements define constraints on the way the software-to-be should satisfy its functional requirements or on the way it should be developed.

This analysis used the FURPS+ model (Eeles 2005), which distributes the non-functional requirements into the following categories: functionality, usability, reliability, performance, supportability, design requirements, implementation requirements, interface requirements and physical requirements. Some of the requirements here presented were extrapolated from the ones mentioned by European Lighthouse Integrated Project 2013a and therefore reference their *UNI ID*.

Each category's requirements are presented in the following sections.

3.2.1 Functionality Requirements

Regarding the Functionality category, the following requirements were identified:

1. User Authentication: Apart from the Anonymous Users, everyone else must be authenticated to use the system;
2. User Authorization: Everyone only has access to what his/her permissions cover, the system shall provide different access permissions to information (UNI.067);
3. Data Exposure Control: Users have control how their data is exposed to (Single-Tenant with) other users (UNI.002);
4. Communication: The system shall support event-based, periodic, and/or autonomous communication (UNI.005);
5. Autonomicity: The system shall enable autonomous goal-driven (task-driven) collaboration between devices or services (UNI.010);
6. Data parameterization: The system shall provide a resolution infrastructure for naming, addressing and assignment of virtual entities and services (UNI.030);
7. Data Storage: The system shall provide historical information about the IoT device (physical entity) (UNI.041);
8. Data Interoperability: the system shall provide interoperable naming and addressing (UNI.048);
9. Auditing and Traceability: All actions performed against the system must be recorded/logged;
10. Security in Communication: The use of secure protocols between clients and the system is mandatory, e.g.: https instead of http;
11. Security in User-provided code: All user-provided code must run in sandbox's to prevent permission escalation, data theft and other related concerns;
12. Data Analysis: The system shall support the integration with a Complex Event Processing (CEP) component (UNI.232);

13. Data Filtering: The system shall be able to filter erroneous sensor data (e.g. GPS location coordinates of a land vehicle appearing in the middle of the ocean;
14. Real-time Alerts: The system must notify the interested clients in real time of any alarm triggered by custom rules;
15. Real-Time Information: Any change to the system must be notified to the client in real-time without resorting to techniques like automatic/manual polling. This includes new sensor data, changes to virtual devices, alarms/rules definitions, decoders and anything else deemed important.

3.2.2 Usability Requirements

Since this project is a greenfield and is still in the early stages of conception, the Usability category is not a major concern. No requirements were proposed.

3.2.3 Reliability Requirements

The Reliability category has the following requirements:

1. The system must validate all user inputs, denying code injection according to OWASP 2021;
2. The system must be able to recover from a failure state such as a crash in the system or any system component;
3. The system shall provide availability through resilience (UNI.064);
4. The system must identify or protect itself against compatibility errors due to versions mismatches between the system and third-party scripts or components, e.g. a valid rule in the system version 1 may not be compatible with the system version 2; if that is the case the system should inform the Costumer and not use the rule.

3.2.4 Performance Requirements

Even though this work is in its early stages of development, the performance of the system is a priority. For single-tenant instances, the requirements specified for this category are:

1. When a new and valid device data is received, the system should make this information available to any user within 2 seconds in 90% of the cases. The time for the information to be presented should never exceed 5 seconds unless the network connection is broken (in which case the user should be notified);
2. When an alarm is triggered, the system should dispatch the alarm within 10 seconds in 90% of the cases;
3. Concurrent Utilization: The system must be able to be used by various users at the same time;
4. High Data Ingestion: The system must be able to successfully process, evaluate and store device data with a throughput of at least 5000 data units per minute.

3.2.5 Supportability Requirements

In the Supportability category the following requirements were identified:

1. The system must be highly configurable so that support for any type of device, specially payload decoding, can be added without the need for restarting/rebuilding it;
2. The system must be agnostic to cloud computing platforms and be independent of any service provided by cloud computing platforms. This ensures that it can be deployed on-site or on a single cloud computing platform;
3. The system must provide simple methods to integrate external services that answer new business cases without the need to rebuilding it;
4. The system shall be extensible for future technologies (UNI.093);
5. The system must attempt to be agnostic to IoT middleware platforms, being able to exchange data with most of them without the need to restarting/rebuilding it. At least *Helium Console* has to be supported.

3.2.6 Design Requirements

The Design Requirements identified are related to how **Sensae Console** must interact with External Systems, namely IoT middlewares and Identity Providers. This requirements also describe what API should be served to Costumers, Organizations and **External Services**.

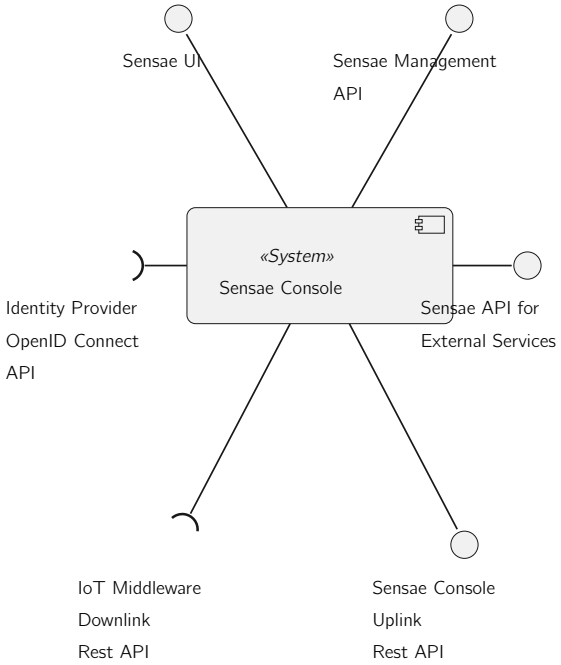


Figure 3.1: Design Requirements Diagram

Most Identity Providers adhere to the OpenID Connect Standard, therefore it's possible to develop a single solution that can exchange information with various Identity Providers in a agnostic manner. The *'IoT Middleware Downlink Rest API'* may require a different implementation for each IoT Middleware since there is no Standard that these platforms can follow, according to Koo and Kim 2022.

3.2.7 Implementation Requirements

In the Implementation category, the system's shall provide a Single Page Application (SPA) to end users.

3.2.8 Interface Requirements

In the Interface category, the following requirements for **Sensae Console** were identified:

1. The system shall require user authentication via OpenID Connect Protocol offered by any Identity Provider;
2. The system shall support the dispatch of downlinks to devices using, at least, the *Helium Console*;
3. The system shall support the ingestion of uplinks from devices using, at least, the *Helium Console*.

As for the **External Services**:

1. The Indoor Fire Retention related External Service shall support the dispatch of emails using Simple Mail Transfer Protocol (SMTP);
2. The Indoor Fire Retention related External Service shall support the dispatch of messages using Short Message Service (SMS).

3.2.9 Physical Requirements

In the Physical category, the following requirements were identified:

1. The system must be publicly available under a single Fully Qualified Domain Name (FQDN);
2. The system shall be deployed in machines running a Linux kernel;
3. The various system components shall be containerized using docker;
4. The various system components shall be orchestrated using docker-compose or kubernetes.

3.3 Synopsis

This chapter mentioned the functional requirements of the project defined during its lifespan. This requirements addressed the needs of the various shareholders, divided in three major roles: (i) manager, (ii) costumer and (iii) anonymous user. While the focus of the project lays in supporting common functionalities of IoT related services within **Sensae Console**, this chapter also mentioned the **External Services** requested by third-parties, and their specific requirements.

Although more vague, the non-functional requirements of the project were also presented using the FURPS+ model.

These requirements lead to the solution's design, presented in the next chapter.

Chapter 4

Design

This chapter's goal is to describe the overall system design to the reader.

The contents here presented corresponds to the final phase of the project, referenced as Phase IV in Section 1.4. The conclusions and necessities gathered in each phase of the project lead to this outcome.

First, the system scopes will be introduced to present the reader a high-level picture of the system. After this, the system's architectural design will be presented and major decisions/alternatives are discussed. Then, a more detailed vision of the canonical data model (related to the *Sensae API for External Services*)is mentioned.

According to Dias, Restivo, and Ferreira 2022, IoT solutions, on a high-level, are commonly composed by three tiers:

- Cloud Tier: Servers, Applications and Data Centers;
- Fog Tier: Routes and Gateways;
- Edge Tier: Embedded Systems, sensors and actuators (things).

This chapter focus only on the Cloud Tier, the other tiers are out of scope since the author had no relevant involvement in their development.

To ease the interpretation of the solution's architectural design, it was divided according to two subjects, scopes and concerns. Scopes are derived from the major system responsibilities of the solution as a whole, concerns are derived from the major functionalities or business cases that the project has to answer.

4.1 System Scopes

The solution designed can be divided in three main scopes as disclosed in Figure 4.1.

The **Sensae Console** is composed by two scopes, **Configuration Scope** and **Data Flow Scope**. These scopes are static and always available in any installation. They answer core/common functionalities of any IoT-based platform. **Sensae Console** is similar to the "Smart City Platform" in the proposed architecture for Smart Cities (IEEE P2413) or the "Event Processing and Analytics Layer" of WSO2 IRA.

The **External Services Scope** is where actual business cases concerns are tackled. This scope is dynamic, meaning that an installation can have different types of external services depending on the costumer needs. The requested PoCs belong to this scope. This scope is

analogous to the "Embedded IoT Applications" in SAT-IoT or the "Application Layer" in the proposed architecture for Smart Cities (IEEE P2413).

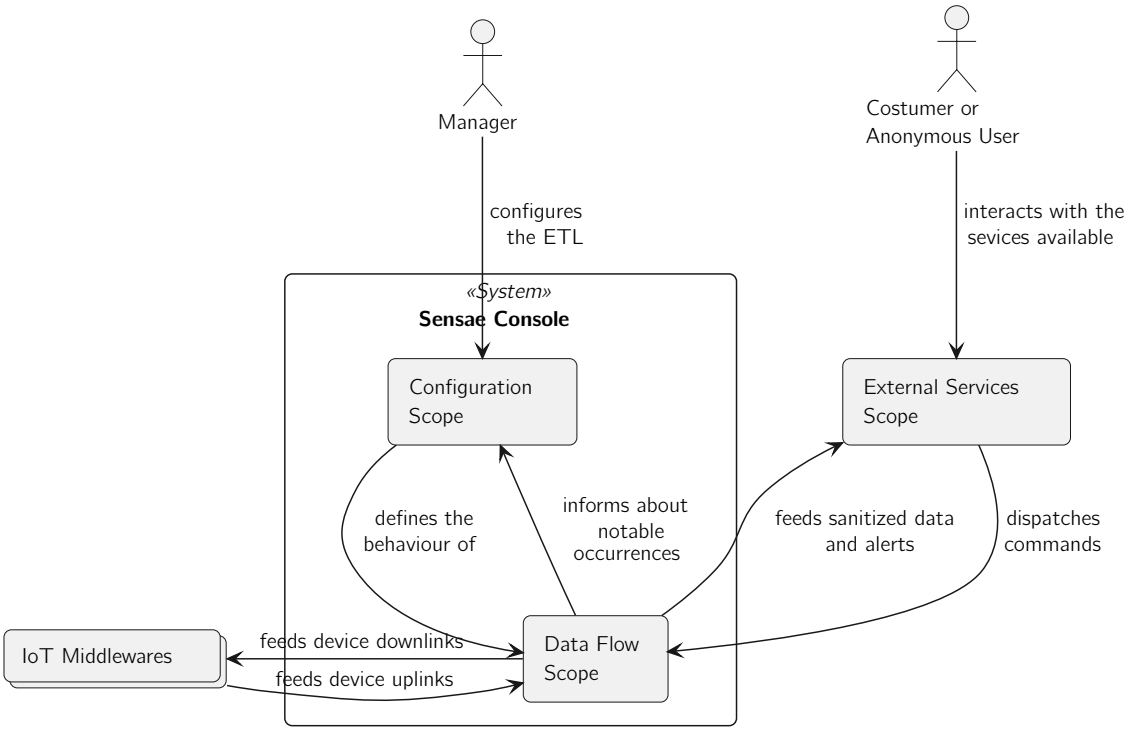


Figure 4.1: System Scopes

The **Configuration Scope** refers to the configuration and visualization of internal processes/concerns, such as: (i) data decoders, (ii) data mappers (iii) device inventory, (iv) warning rule scenarios definition and (v) device ownership - related to the **Data Flow Scope**. It is also possible to manage users' access and permissions in the **Configuration Scope**.

The **Data Flow Scope** acts as a pipeline where raw data - device uplink - goes through various stages till it is sanitized and ready to be supplied to the **External Services Scope**. The **Data Flow Scope** is where internal processes occur, such as: (i) data transformation, (ii) data enrichment, (iii) data validation, (iv) data ownership clarification and (v) alert dispatching. It behaves according to what is defined in the **Configuration Scope**.

The **External Services Scope** is comprised of services that present and act according to the sanitized data and alerts that were supplied to them. These services applicability range from (i) smart irrigation, (ii) fleet management, (iii) fire detection, (iv) physical security access monitoring, (v) air quality monitoring and anything else deemed interesting. The services currently developed are smart irrigation, fleet management and notification management. These will be addressed throughout this and the Implementation chapter.

4.1.1 Configuration Scope

The **Configuration Scope** is responsible for managing the following concerns:

- **Data Processor**: manages simple data mappers;

- **Data Decoder**: manages scripts to transform data;
- **Device Management**: manages device information such as name, metadata, static data and other notions;
- **Identity Management**: manages device ownership and users permissions;
- **Rule Management**: manages scripts that consume device data and produce alerts.

These concerns can be directly linked to the functional requirements described in the Section related to the 3.1.2.1 role.

Each concern can be managed by an authorized user, e.g. the data processor concern focus on the creation, deletion and renovation of data mappers.

These operations require various verifications, alter the system internal state and are therefore prolonged.

4.1.2 Data Flow Scope

The **Data Flow Scope** is responsible for processing incoming data according to what is defined in the **Configuration Scope**. Both scopes share the same concerns.

This scope also contains four independent units, that aren't controlled by the **Configuration Scope**:

- Data Relayer: responsible for providing a bridge between the IoT middlewares and the **Sensae Console**;
- Data Gateway: responsible for starting the flow of data in this scope by publishing device uplinks in it;
- Data Validator: responsible for filtering device measures based on static rules, e.g. battery percentage reported has to be in between 0 and 100.
- Data Store: responsible for persisting data captured in a previously defined state.

This scope applies changes to the device measures that flow through the system. These changes are stateless and don't change the overall state of the internal system.

This scope was decoupled from the **Configuration Scope** even though they both work with the same concerns. The decision was taken based on the pretext that despite the similarities in concerns the operation/business responsibilities of these two scopes were conflicting.

The **Configuration Scope** requires scarce but heavy computations that alter the internal system state, while the **Data Flow Scope** requires plentiful but light computations that don't alter the internal system state as summarized in the Table 4.1.

Table 4.1: Comparison of Operations in Data Flow and Configuration Scopes

Comparison of Operations	**Configuration Scope**	**Data Flow Scope**
Alter internal system state	yes	no
Alter device measures	no	yes
Required computation power/time	high	low
Frequency of usage	low	high

Due to this discrepancy it's expected for each scope to have different requirements regarding horizontal scaling. With the addition of more devices to the platform, and subsequently higher ingress volume, **Data Flow Scope** will need to scale. Since the **Configuration Scope** is intended mostly for the manager of the platform, a small user pool, the need to scale is smaller.

4.1.3 External Services Scope

The **External Services Scope** is responsible for presenting IoT business cases to end users. This scope is detached from the **Sensae Console** due to its dynamic nature. The services that belong to this scope are analogous to plugins.

The scope is comprised of services that consume data and publish commands to **Data Flow Scope**. Currently, as a Minimum Value Product (MVP) the implemented business cases are:

- **Fleet Management**: basic service to monitor a fleet of cars regarding their location;
- **Smart Irrigation**: service to automate and monitor the irrigation of zones based on sensor readings;
- **Notification Management**: service to view and manage the delivery of triggered alerts.

Each service is bounded to what type of data receives and sends back to the **Data Flow Scope** as later detailed in the Solutions - External Services Section. The type of data each service handles is enforces by the concepts discussed in Sections 4.4.1 and 4.4.2.

Just like plugins, services in this scope are validated and attached to the final deployment by the entity that manages that specific instance. When working in a multi-tenant instance, custom external services can't be properly verified and therefore their usage is denied.

4.2 Architectural Design

In order to describe the system in detail at the architectural level, an approach based on the combination of two models, C4 (Brown 2018) and 4+1 (By and Jiang 1995) will be followed.

The 4+1 View Model, proposes the description of the system through complementary views, thus allowing to separately analyze the requirements of various software stakeholders, such as users, system administrators, project managers, architects, and programmers.

The five views are thus defined as follows:

- **Logical view**: relative to the aspects of the software aimed at responding to business challenges;
- **Process view**: relative to the process flow or interactions within the system;
- **Implementation View**: relative to the organization of the software in its development environment;
- **Physical view**: relative to the mapping of the various components of the software in hardware, i.e. where the software is executed;

- **Scenario view**: related to the association of business processes with actors capable of triggering them.

The C4 Model advocates for the description of software through four levels of abstraction: (i) context, (ii) container, (iii) component, (iv) code. Each level adopts a finer granularity than the level that precedes it, thus giving access to more details of a smaller portion of the system. These levels can be linked to maps, e.g. the context view corresponds to the globe, the container corresponds to the map of each continent, the component view corresponds to the map of each country, and the code view to the map of roads and neighborhoods in each city.

Different levels tell different stories to different audiences.

The levels are defined as follows:

- **Level 1**: Description (context) of the system as a whole;
- **Level 2**: Description of system containers;
- **Level 3**: Description of components of the containers;
- **Level 4**: Description of the code or smaller parts of the components.

These two models can be said to expand along distinct axes, with the C4 Model presenting the system with different levels of detail and the 4+1 View Model presenting the system from different perspectives. By combining the two models it becomes possible to represent the system from several perspectives, each with various levels of detail. To visually model/represent the ideas designed and alternatives considered, the Unified Modeling Language (UML) was used.

In the following sections only combinations of perspectives and levels deemed relevant for the design of the solution are presented.

The C4 level 4, code, will not be exhibited.

4.2.1 C4 Level 1 - Context

The context level aims at introducing the system as a whole. The external systems and users that communicate/interact with the system, **Sensae Console**, and solutions, **External Services** are demonstrated. Throughout this section the relevant C4 views of level 1 (context level) are presented.

4.2.1.1 Context Level - Logical View

The logical view of the system is introduced here, complete but not detailed, in order to answer the use cases and requirements discussed in Chapter 3. This takes into account the interactions of **Sensae Console** and **External Services** with foreign systems and their interactions with the various actors of the system (Figure 4.2) as required by Section 3.2.6.

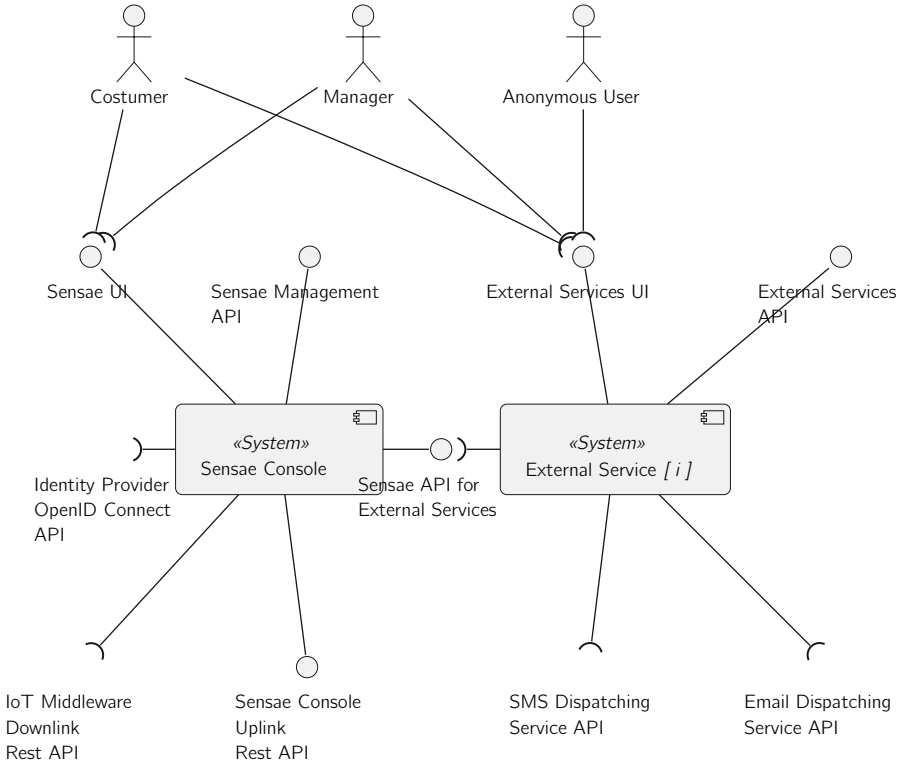


Figure 4.2: Solution - Context Level - Logical View Diagram

The **External Services** are represented as an independent collection of systems that consume the **Sensae Console** API. This API is responsible for streaming information such as device measures, device commands, alerts and internal state asynchronously. These concept's semantics and structure are enforced by a library, *iot-core*, also developed and discussed in Section 4.4.2.

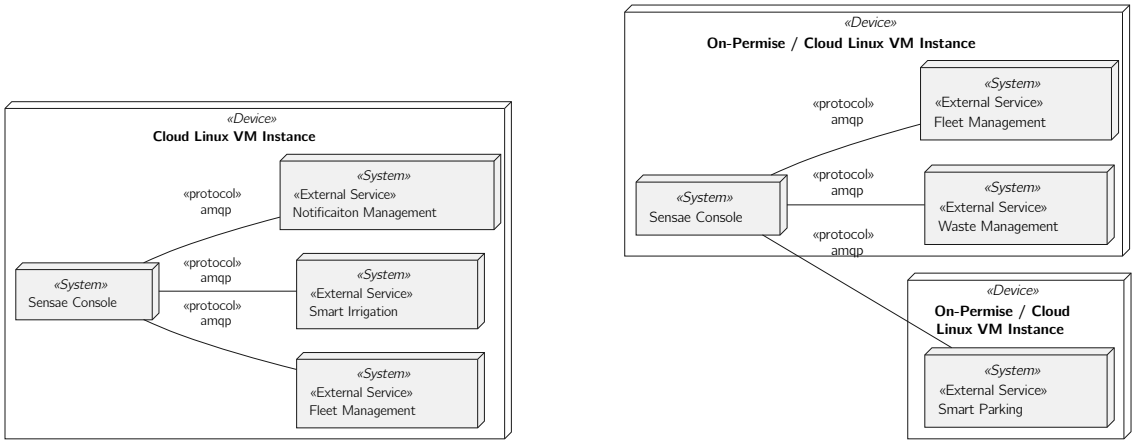
All systems provide an API for automated management/control and a UI for ease of use and data visualization.

As mentioned before in Section 3.1.3.2 there is a need to integrate the final product with an Email and SMS dispatch service.

The reason that lead to the use of external authentication/identity services, as required in Section 3.2.8, is further discussed in Appendix F.

4.2.1.2 Context Level - Physical View

Next is the physical view (Figure 4.3), intended to familiarize the reader with the environment where the solution runs.



(a) Multi-Tenant Instance

(b) Single-Tenant Instance

Figure 4.3: Solution - Context Level - Physical View Diagrams

There are two major options when deploying a new **Sensae Console**: (i) cloud and (ii) on-premise deployment. Each deployment can be a:

- Multi-Tenant Instance (Figure 4.3a): This deployment serves various costumers and therefore, all external services are developed and validated by the company to avoid interacting with services that may abuse the system in nefarious ways. Currently, for this type of instance, the Sensae Console and the External Services run in a single instance. The external services correspond to the PoCs developed.
- Single-Tenant Instance (Figure 4.3b): This deployment serves a single costumer, therefore he/she can connect custom external services that aren't validated or developed by the company. The two custom external services serve as an example of the freedom given to the costumer to interact with the system.

The connections to external systems and interactions with users were hidden for brevity reasons. The reason for these distinct deployment options derive from the discussion in Chapter 3.

4.2.1.3 Context Level - Synopsis

The context level introduces the reader to the bigger picture of the whole solution, but it contains little to no information about how the system functions internally.

The process view was not represented since at this level the interactions between the system, actors and external systems, are too abstract to be relevant for the reader. The implementation view was also not represented since the **Sensae Console** and **External Services** were developed as a single project.

The Sections Platform - Sensae Console and Solutions - External Services will dive into the internal of the **Sensae Console** and the solutions developed.

4.2.2 Platform - Sensae Console

This section will explore the internals of **Sensae Console** from an architectural point of view. It discusses the C4 container level. The C4 component level is discussed in Appendix C.

The C4 level 2 introduces the reader to the various containers that compose the platform. In this section all relevant views will be presented according to the alternative in use or idealized for the system. In the Section 4.3 other alternatives are discussed.

The description of this level of abstraction begins with a logical view.

4.2.2.1 Container Level - Logical View

In order to support the functional requirements identified (Section 3.1), and knowing that **Sensae Console** will serve multiple users with different levels of access to the managed information, the various business concepts were segregated from the user interaction. The configuration management also had to be separated from the data pipeline, knowing that **Sensae Console** will process a high volume of device measures.

Considering the need to persist and provide the information collected, the system integrates databases, which are not developed, but only configured and operated - using a DBMS.

The system also uses one (or more) message brokers, IBM 2020b, that will be configured but not developed.

In order to ease the analysis of the platform, the following diagram (Figure 4.4) presents a complete view of **Sensae Console** where each concern represents a group of containers. These groups are then explored in detail.

As seen in the diagram:

- Each concern exposes a UI and an API, these are aggregated in the **UI Aggregator** container that then exposes everything as a single UI and API for management;
- The Device Management concern consumes the IoT Middleware API since it is responsible for sending downlinks to devices;
- The Message Broker exposes an API, this is the API that the **External Services** consume to access the information that flow in **Sensae Console**;
- The Identity Management concern consumes the Identity Provider's OpenID Connect API to handle User Authentication;
- The Message Broker is responsible for routing messages through the system and ensuring that the various containers communicate;
- The Data Store Backend and Data Store Database are responsible for storing data in a specific format, defined at startup via configuration;
- The Data Relayer and Data Gateway are responsible for exposing an API for data ingestion and publish the ingested data in the system through the Message Broker;
- The Data Validator applies simple filters to incoming data, for example, measures that report a soil moisture of 120% are marked as incorrect.

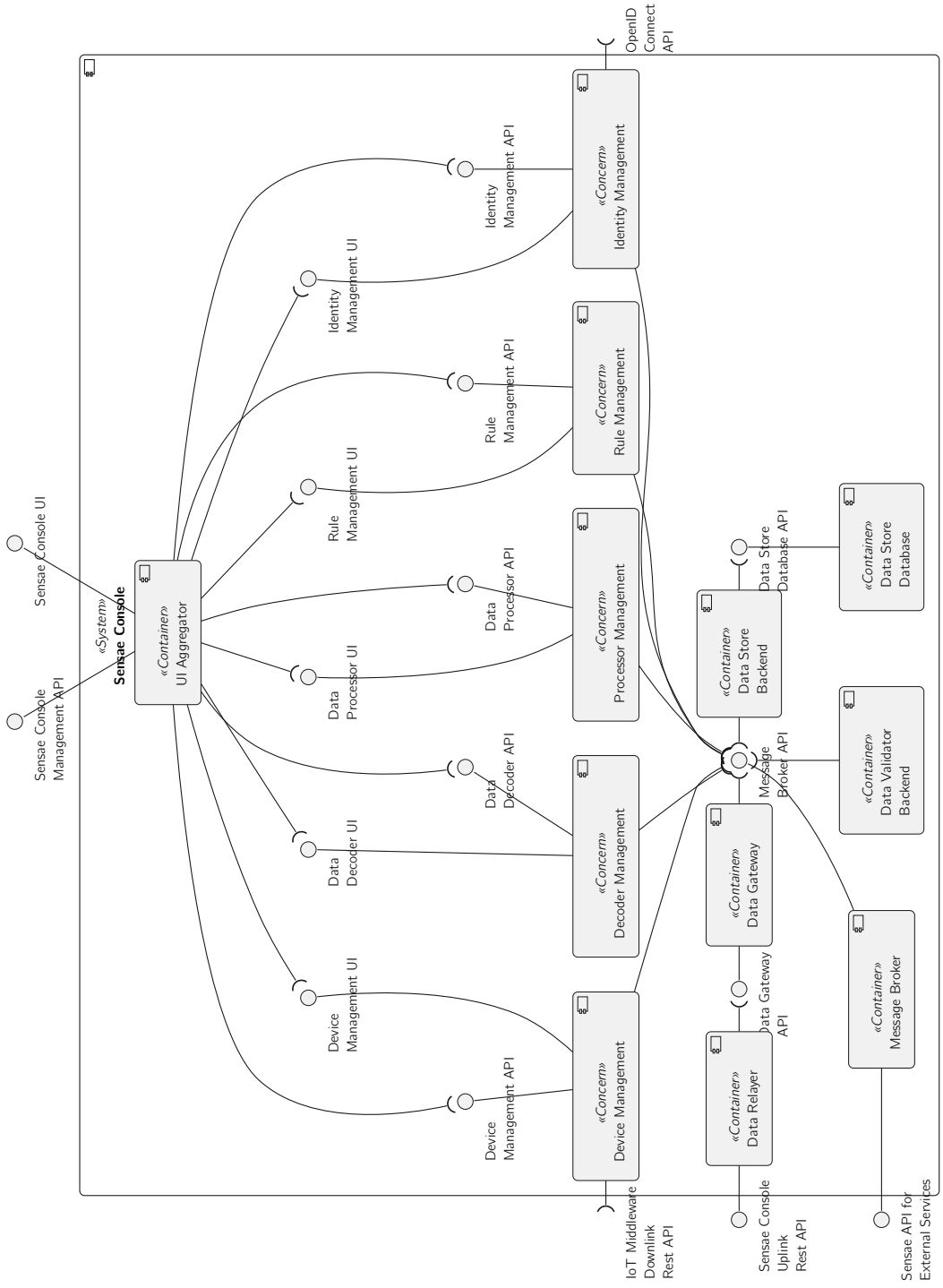


Figure 4.4: Sensae Console - Container Level - Logical View Diagram

Each concern is composed by containers that belong to the **Configuration** and **Data Flow** Scopes (represented in yellow in the following diagrams). The Configuration Scope of each concern is composed by a three layers architecture, as per IBM 2020a:

- **Presentation Layer**: the user interface and communication layer of the application where the user interacts with the system;
- **Application Layer**: the business layer of the application where information from the **Presentation Layer** is processed and sent to the **Data Layer**;
- **Data Layer**: the infrastructure layer of the application where data is stored and requested as needed.

The Data Flow Scope is usually composed by a single container that only consumes the Message Broker API.

As a brief description of some of the similar characteristics of all concerns:

- The frontend container corresponds to the **Presentation Layer** and exposes an UI;
- The backend container corresponds to the **Application Layer** and communicates with the Data Flow container(s) exclusively through the **Message Broker**. The Backend publishes issues related to the concern's configuration that the Data Flow Container consumes. The Data Flow container publishes metrics related to what resources are being used that are then consumed by the Backend;
- The communication exchanged between Backend and Data Flow containers is parameterized according to the Section 4.4.2.3 and is preformed in the Internal Topic;
- The backend container exposes an API that is consumed by the frontend and optionally by properly authenticated external systems;
- The database container corresponds to the **Data Layer**.

The Data Processor concern group is presented in Figure 4.5.

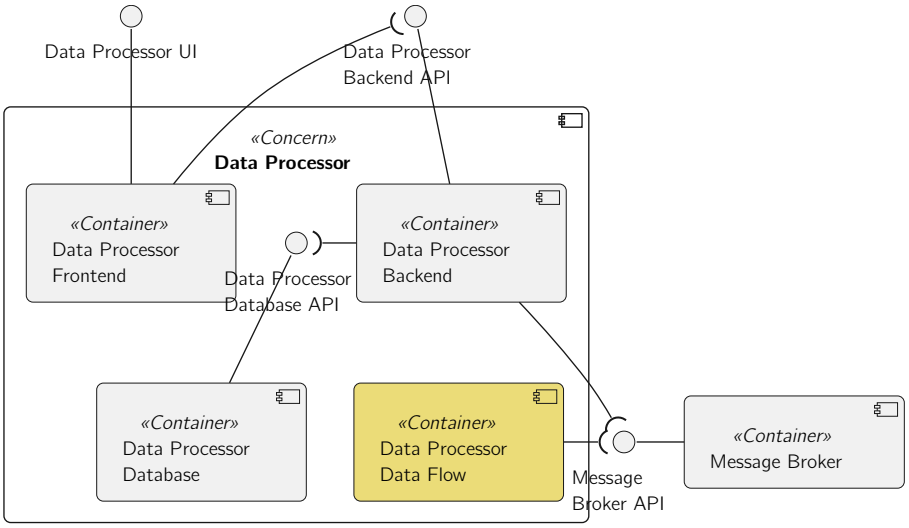


Figure 4.5: Data Processor - Container Level - Logical View Diagram

This concern is responsible for transforming the data received in a format and semantic that can be understood by the system, it is explored in detail in Section G.1. The Data Processor Data Flow publishes metrics to the Message Broker regarding the time each Data Processor was used so that the Backend can then report this usages.

The Data Decoder concern group is presented in Figure 4.6.

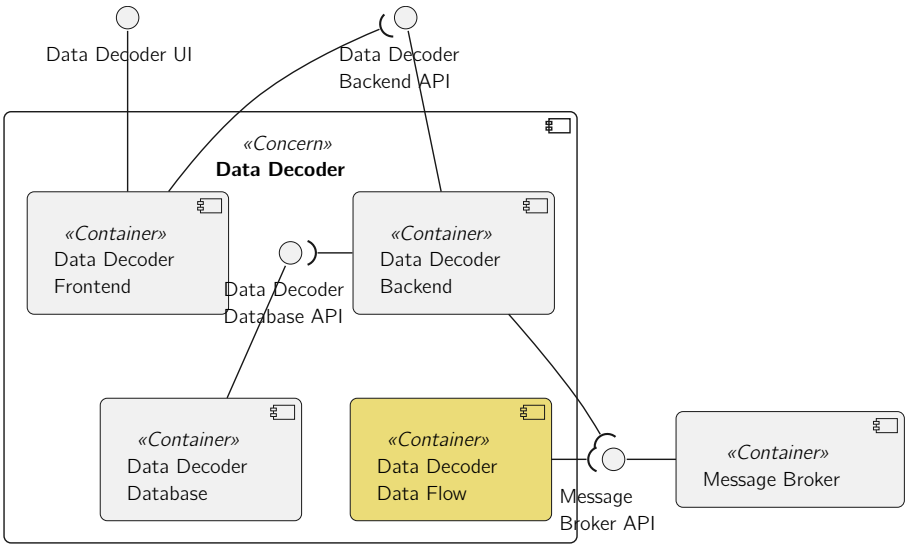


Figure 4.6: Data Decoder - Container Level - Logical View Diagram

This concern is also responsible for transforming the data received in a format and semantic that can be understood by the system. In contrast with the Data Processor, it provides a more flexible but complex way of manipulating data, it is explored in detail in Section G.2. The Data Decoder Data Flow publishes metrics to the Message Broker regarding the time each Data Decoder was used so that the Backend can then report this usages.

The Device Management concern group is presented in Figure 4.7.

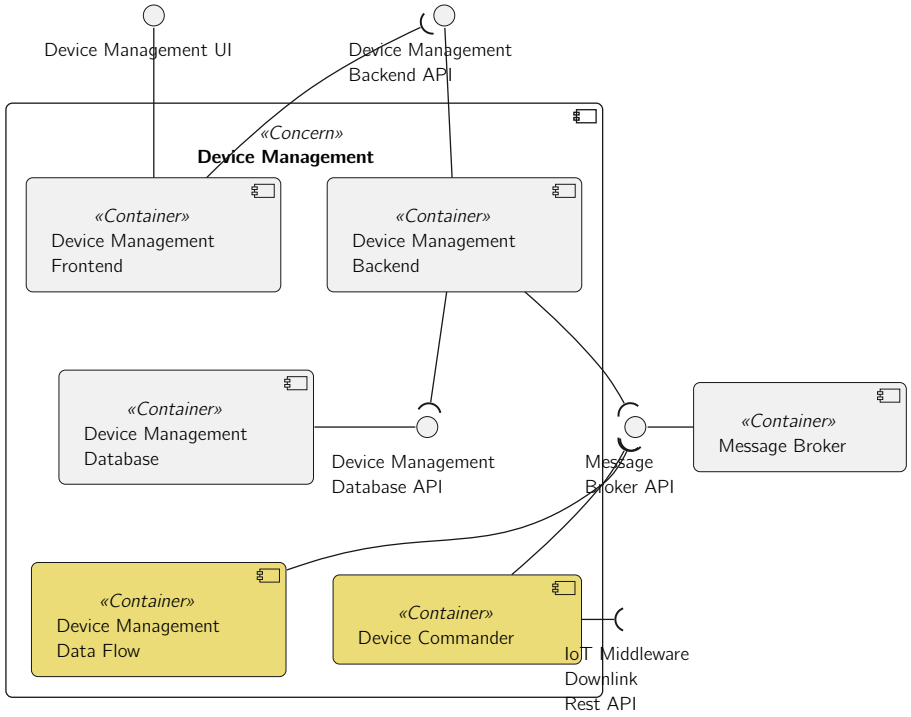


Figure 4.7: Device Management - Container Level - Logical View Diagram

This concern is responsible for maintaining a registry of the devices in use by the platform.

The Device Management Data Flow enriches the measures collected with more information regarding the device that sent them. The Device Commander consumes an IoT Middleware REST API to dispatch downlinks to devices. This downlinks contain commands that control the behavior of the implied actuator. This concern is explored in Section G.3. The Data Flow containers publishes metrics to the Message Broker regarding the time each device was used so that the Backend can then report this usages.

The Identity Management concern group is presented in Figure 4.8.

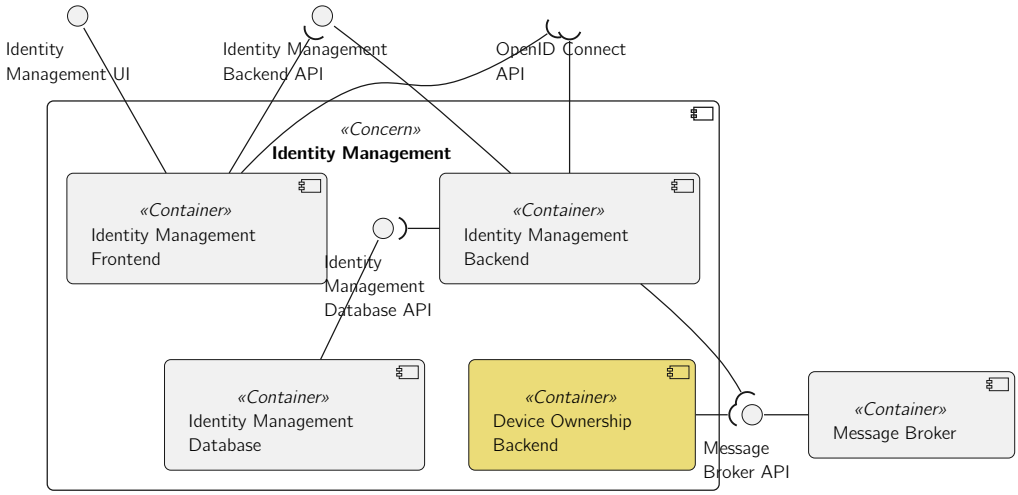


Figure 4.8: Identity Management - Container Level - Logical View Diagram

This concern is responsible for managing devices ownership, user identity and organization's details. The backend and frontend containers communicate with an identity provider via OpenID Connect to verify the user identity. The Device Ownership Backend enriches the data measures and alerts with information regarding the organizations that own the device responsible for sending the measures or that lead to the dispatch of an alert. This concern is explored in Section G.4. This data flow container publishes metrics to the Message Broker regarding the time each organization information was used.

The Rule Management concern group is presented in Figure 4.9.

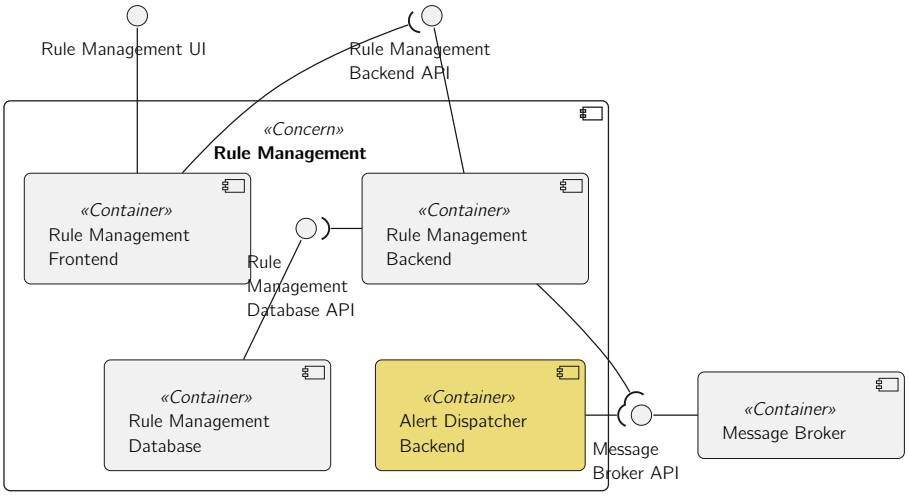


Figure 4.9: Rule Management - Container Level - Logical View Diagram

This concern is responsible for managing rule scenarios that produce alerts based on the captured device measures.

The Alert Dispatcher is responsible for publishing alerts based on the rule scenarios published by the Rule Management Backend. The Rule Management Backend ensures that the rules submitted are valid. This concern is explored in Section G.5. This data flow container does not publishes any metrics, its interactions are better described with the help of sequence diagrams available in Figures 4.10 and C.5.

As the diagrams above presented, all communication between backend containers of both scopes is guaranteed by the Message Broker. This Message Broker exposes its API so that External Services can consume all information and act according to it. The Section 4.2.3 explores the solutions developed.

In the following section the internal communication of the system is clarified.

4.2.2.2 Container Level - Process View

In this section, several use cases (according to some functional requirements identified in Section 3.1) are presented through sequence diagrams, in order to introduce the reader to the interactions that occur between the various containers of the **Sensae Console**.

The routing keys used for communication between backend containers can be extrapolated from the model described in the Section 4.4.2.3.

This section is composed by five sets of important functionalities to discuss at this level of abstraction: (i) system/container initialization (ii) data pipeline operation, (iii) data pipeline configuration, (iv) user authentication/authorization, (v) service usage.

The system/container initialization, presented in Figure 4.10, refers to the interval of time since a container is launched till it is ready to process requests or events.

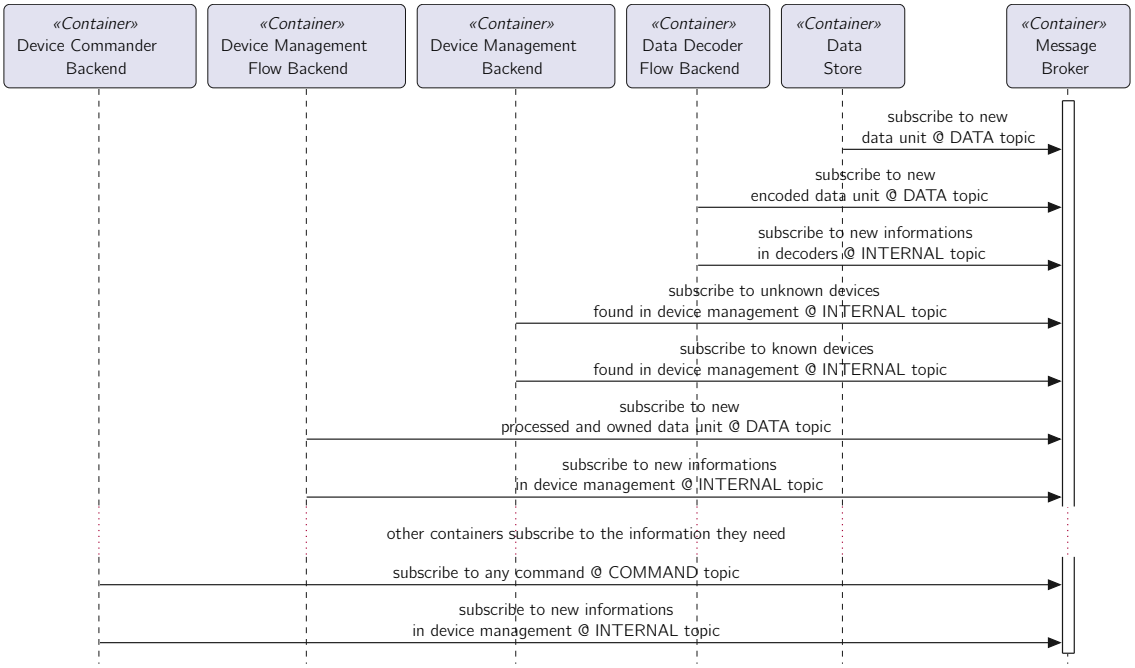


Figure 4.10: System/Container Initialization - Part 1 - Container Level - Process View Diagram

Not all containers are displayed in this diagram for brevity reasons. The system relies heavily in the Pub/Sub (Reselman 2021) pattern to communicate internally via a message broker. In this scenario the first step in a container life cycle is to subscribe to the information that it needs as presented in the diagram above.

Certain containers need the entire state related to their concern to function. So, after subscribing to the needed information, they notify the system that they have entered an *init state* for a specific concern. This triggers the creation of new events to help that container to reach a *ready state*. An example of this interaction is presented in the following diagram, Figure 4.11.

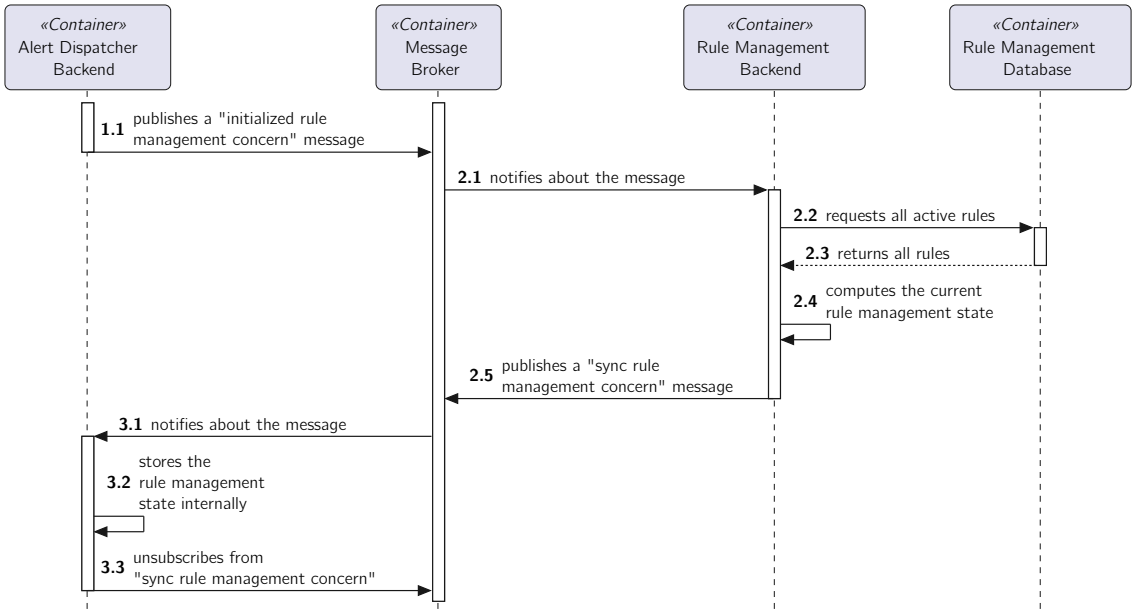


Figure 4.11: System/Container Initialization - Part 2 - Container Level - Process View Diagram

Apart from the Alert Dispatcher Backend, all containers in the **Data Flow Scope** function with just a portion of a single concern state or no state at all.

To dive into this, some common data pipeline operations, related to the Data Flow Scope, are presented next. This operations are intended to behave in a *reactive* manner (Jonas Bonér and Thompson 2014) and are therefore non-blocking. The idea behind the Data Flow Scope is analog to a data pipeline. This scope operates mostly with Data Units, transforming, filtering and enriching this data.

The following diagram in Figure 4.12 presents a high level view of the flow that a Data Unit takes through the system in the Data topic. This diagram does not account for what happens to invalid Data Units and the interactions with the message broker are hidden for brevity reasons even though it is used by all containers to publish and receive messages.

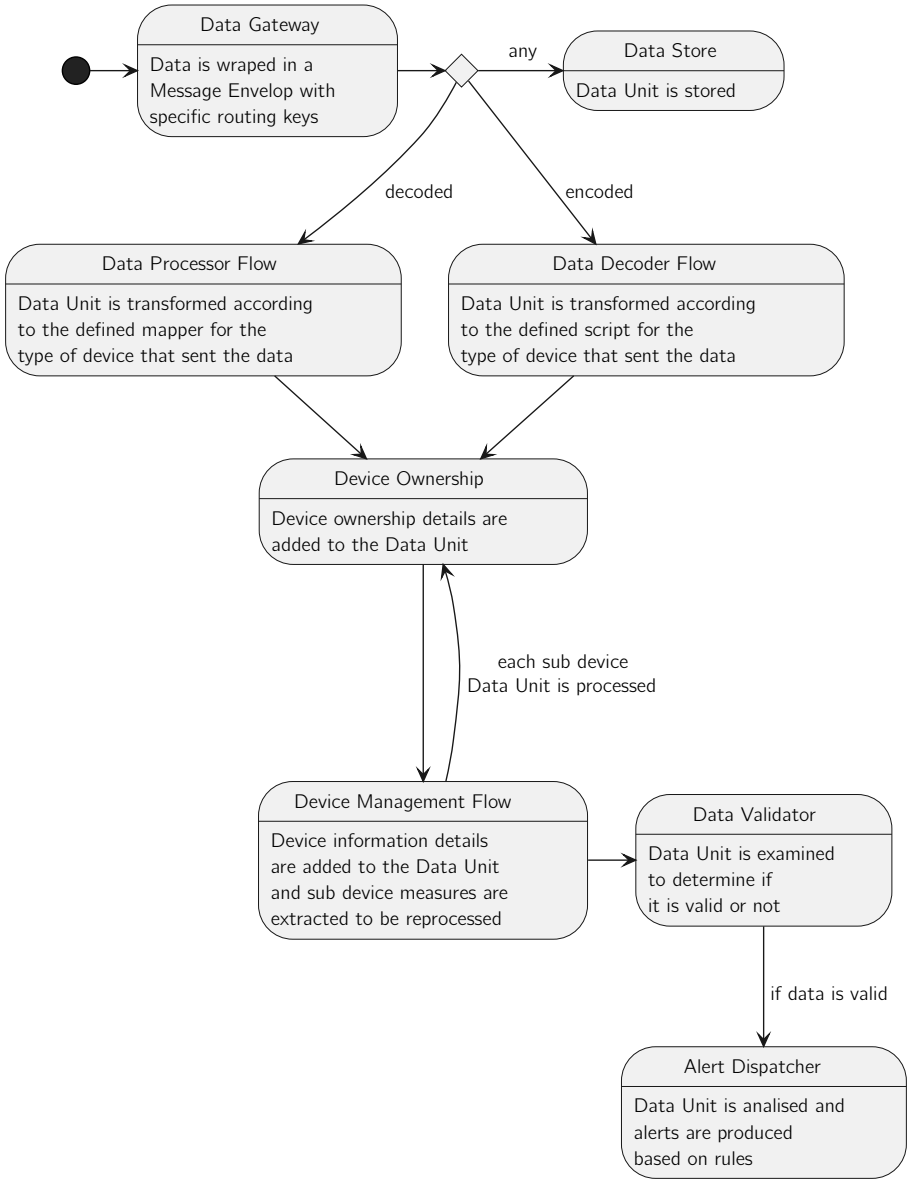


Figure 4.12: Data Flow - Container Level - Diagram

Most of these containers have just a portion of their concern's state and may be unable to preform the needed operation on some Data Units. The following diagrams, Figure 4.13 and Figure 4.14, addresses how state is managed in Data Decoder Flow Backend and most **Data Flow Scope** containers.

As we can see, in Figure 4.13, the Data Decoder Flow Backend, upon receiving a Data Unit, can preform two operations, depending on whether or not the script is available: decode the Data Unit and notify that the script was used or store the Data Unit and notify that a script for an unknown device type is needed.

The diagram in Figure 4.14 describes what happens when a message with a decoder is published (using the *OperationType* Info mentioned in Section 4.4.2.3).

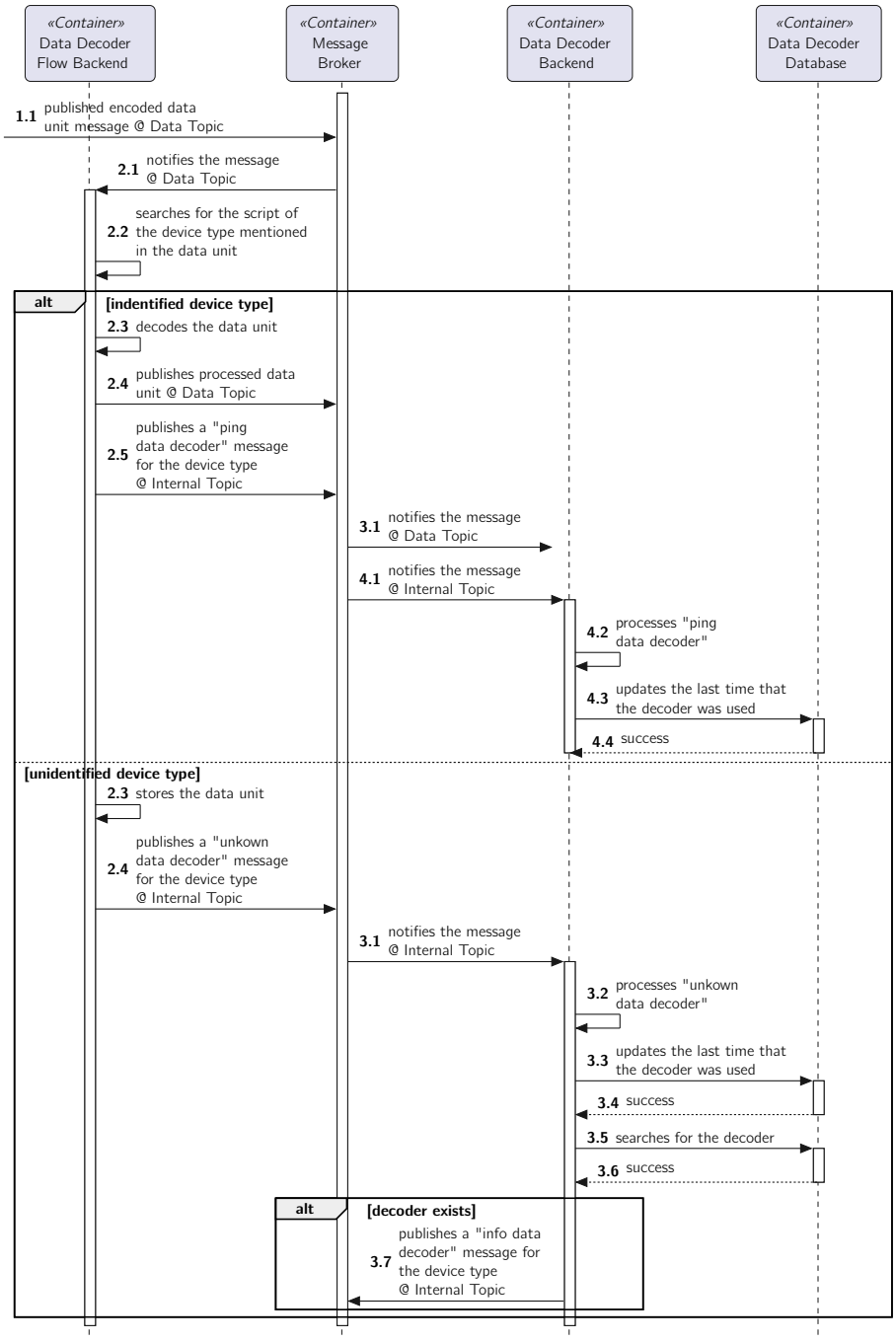


Figure 4.13: Data Decoder Operation - Part 1 - Container Level - Process View Diagram

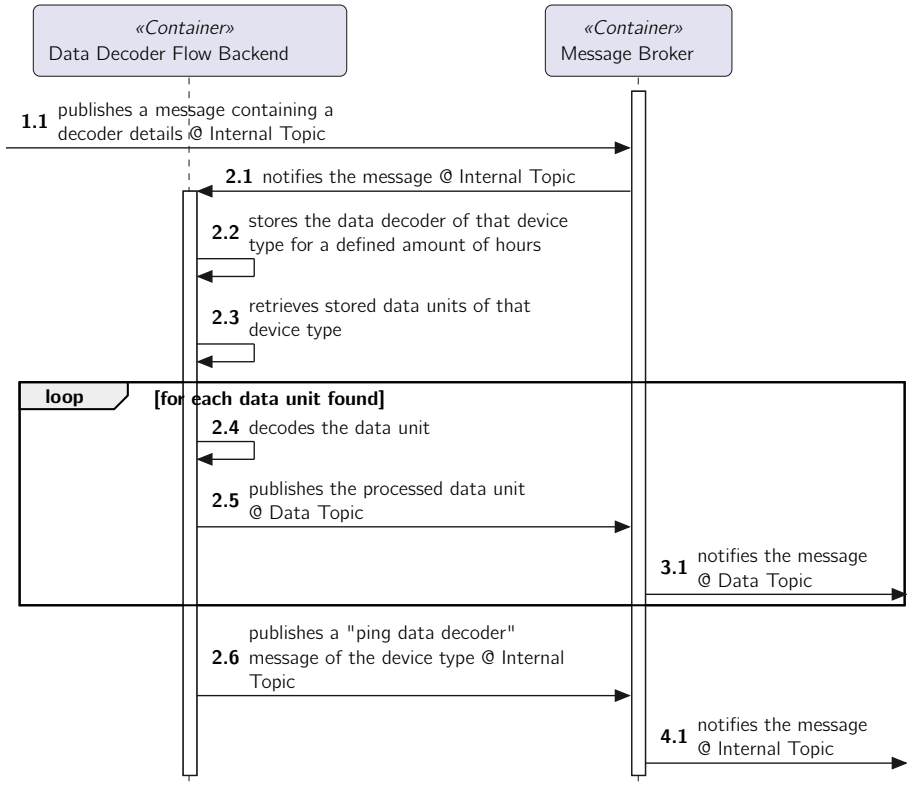


Figure 4.14: Data Decoder Operation - Part 2 - Container Level - Process View Diagram

As we can see, Figure 4.14, Data Decoder Flow Backend, upon receiving an info regarding a data decoder, searches for unhandled Data Units and processes them. To minimize the memory in use, a data decoder has to be continually used in order for it to remain in cache. As seen in step **2.2**, if X hours pass since the last time a decoder was used it is evicted from the container internal state.

The operations described here for the Data Decoder Flow Backend are replicated in the following concerns/containers:

- **Data Processor Context**: Data Processor Flow Backend;
- **Device Management Context**: Device Management Flow Backend and Device Commander Backend;
- **Identity Management Context**: Device Ownership.

As described before, containers that belong to the **Data Flow Scope** operate according to what the **Configuration Scope** defined.

The next diagrams, in Figure 4.15 and Figure 4.16 present some of the common operations that happen in that scope.

The diagram presented above represents a simple consult of data mappers, as we can see, only the Data Processor Context in the Configuration Scope is invoked. When a change to the state is made in any Context of the Configuration Scope, events are published. The next diagram, Figure 4.16 displays an example of this occurrence.

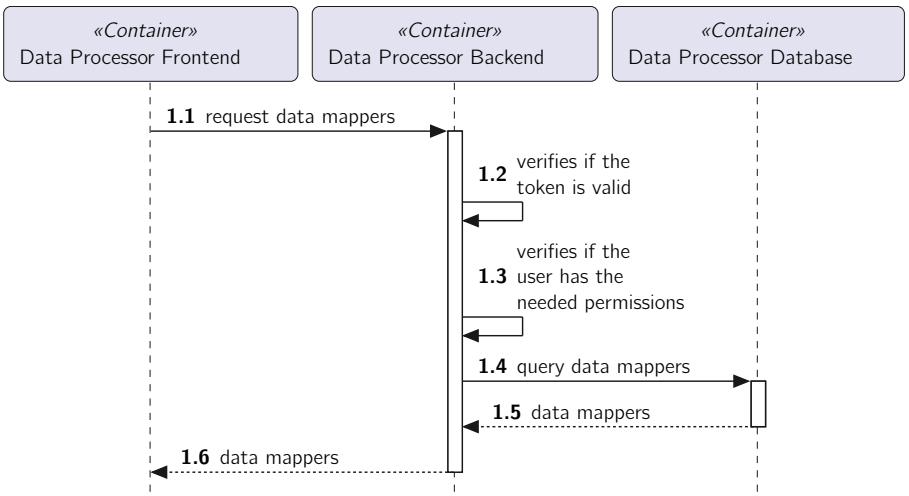


Figure 4.15: Consult Data Processors - Container Level - Process View Diagram

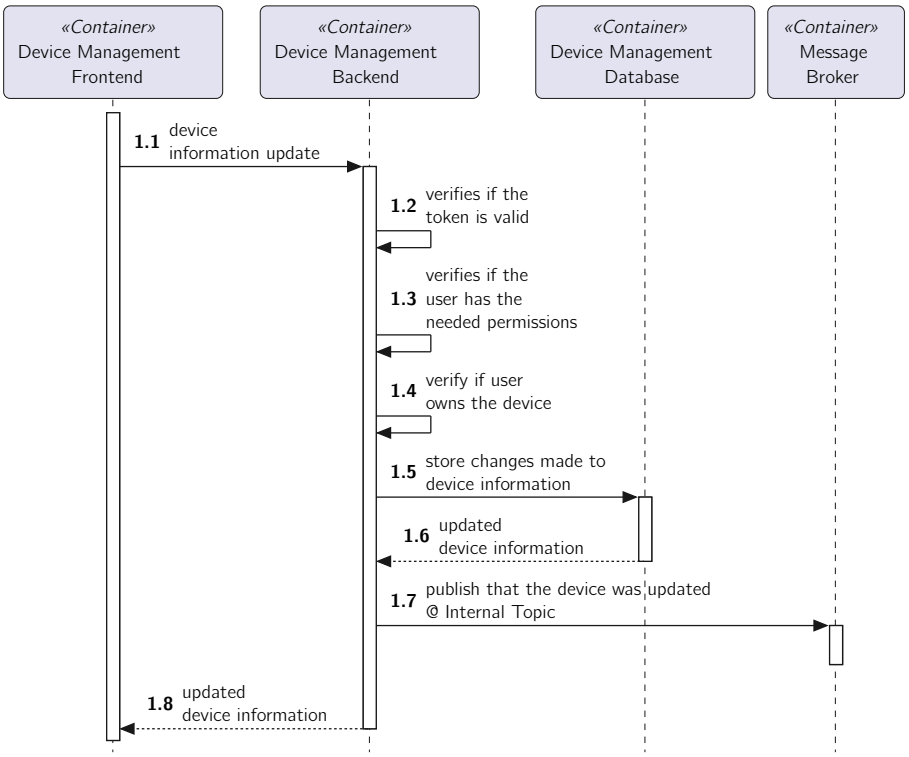


Figure 4.16: Edit Device Information - Container Level - Process View Diagram

In this use case a device information is changed. Since this operation changes the internal state of the device management concern, an event is published in the Internal Topic.

According to the Section 4.4.2.3, this specific event uses the following *Routing Keys*:

- **Protocol Version**: the version of *iot-core* currently in use by Device Management Backend;
- **Container Type**: Device Management Backend;
- **Topic Type**: Internal;
- **Operation Type**: Info;
- **Context Type**: Device Management.

There are three containers that subscribe to this specific type of event:

- **Device Management Flow Backend**: so that the Data Units of the device changed are enriched with the latest information;
- **Device Command Backend**: so that commands for this device are treated according to the latest information;
- **Identity Management Backend**: so that information related to the device changed is presented according to the latest update. This container maintains local copies of all devices names to present to the user without needing to request Device Management for that information every time.

The step **1.3** in the last two diagrams references user permissions but there is no mention of how this permissions are associated to the user. In the next diagrams - Figure 4.17 and Figure 4.18 - authentication and authorization in the **Sensae Console** are addressed, other approaches are discussed in Appendix F.

The system verifies the identity of a user based on the authentication performed by an external Customer Identity and Access Management (CIAM) solution using OpenID Connect 1.0, OpenID 2014, an identity layer on top of the OAuth 2.0 protocol. According to D. Hardt 2012 OAuth2.0 "enables a third-party application to obtain limited access to an HTTP service". In this situation the Frontend of **Sensae Console** is the third-party application and the HTTP service is any of the **Sensae Console** backend services.

This diagram illustrates how a user can authenticate against **Sensae Console**. The user identity and credentials validation are assured by an external identity platform such as *Google Identity Platform* or *Azure Active Directory (Azure AD)*. Once an *id token* is provided to **Sensae Console** it can use it to verify the user identity against the local registry. To ensure that the *id token* is valid, Identity Management Backend checks if it was signed by the platform that supposedly issued it (step **3.3** and **3.5**). After validating the *id token* it searches for the needed information to create an *access token* and then provides it. The *access token* can then be used for a limited time to access any protected HTTP resource of **Sensae Console** as demonstrated in Figure 4.18.

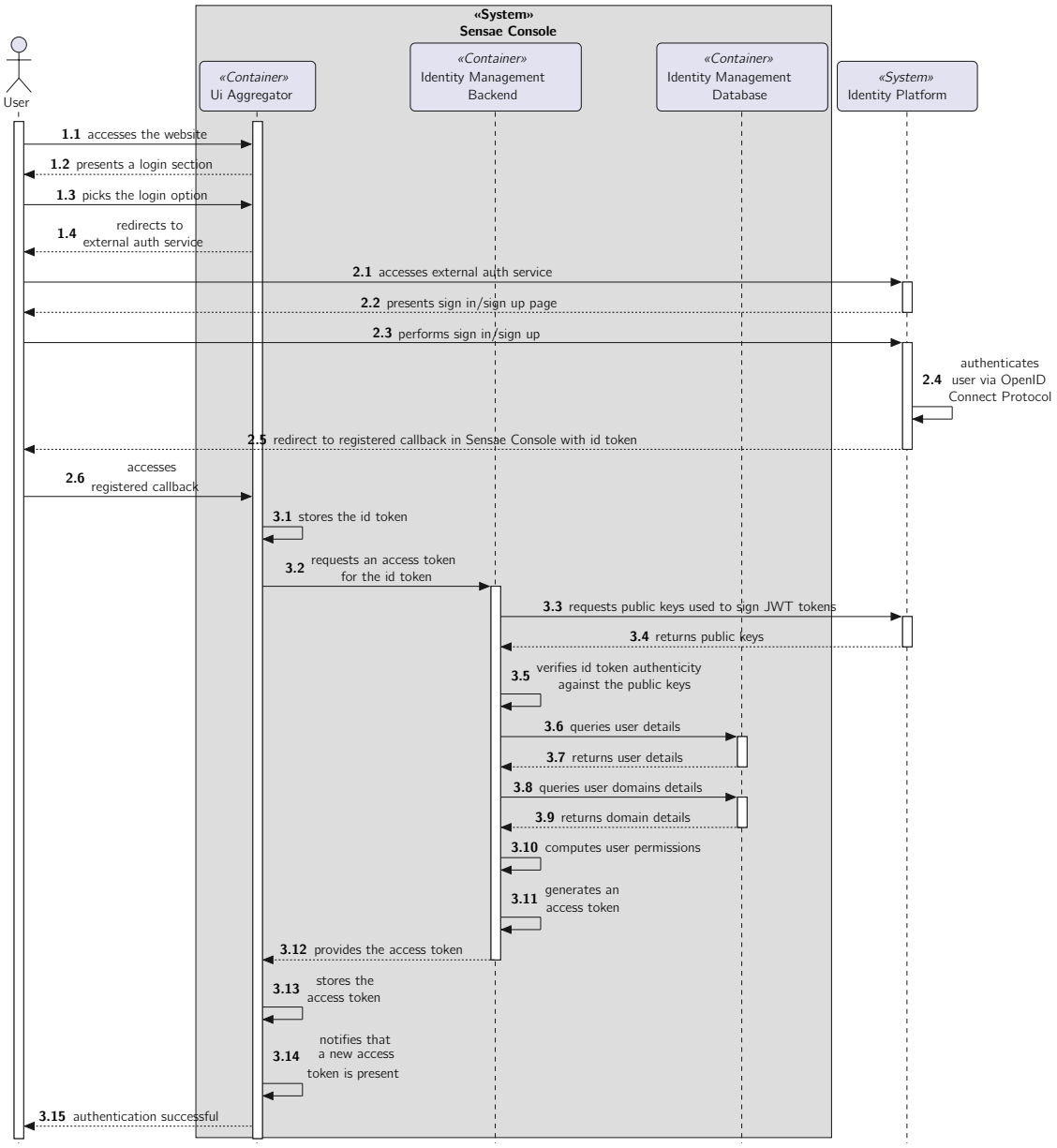


Figure 4.17: User Authentication - Container Level - Process View Diagram

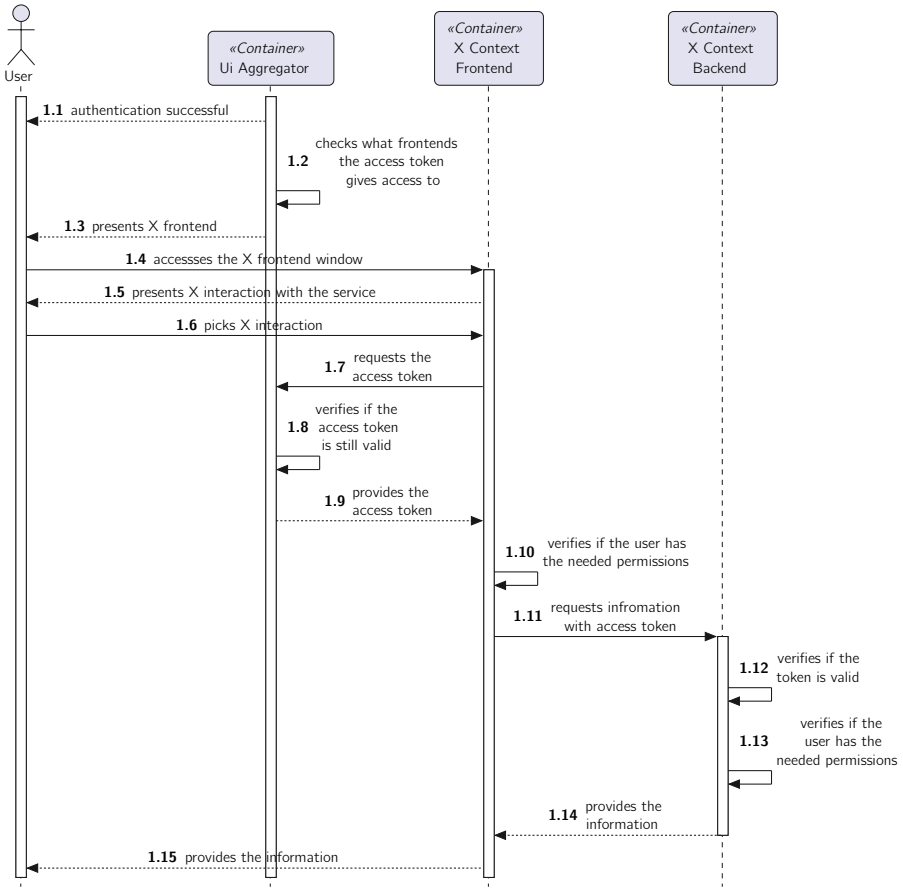


Figure 4.18: User Authorization - Container Level - Process View Diagram

In this diagram the expected behavior for any pair of frontend and backend containers in **Configuration Scope** (and **External Services**, when served from the **UI Aggregator**) is presented. Each frontend displays only the actions and information that the user permissions allow. The user permissions are once again verified in the backend to secure the system against malicious accesses. Other alternatives related to authentication and authorization are presented in Appendix F.

4.2.2.3 Container Level - Implementation View

Each container mentioned in the Section 4.2.2.1 is developed inside the same package, *sensae-console*. The following diagrams presents how containers are mapped to packages.

The following diagrams are divided into:

- Backend Containers: Figure 4.19;
- Database Containers: Figure 4.20;
- Frontend Containers: Figure 4.21;

Backend services are organized according to the diagram in Figure 4.19.

Each backend service container is mapped to its own individual package. The Data Relayer Container was the only one configured, all others were developed.

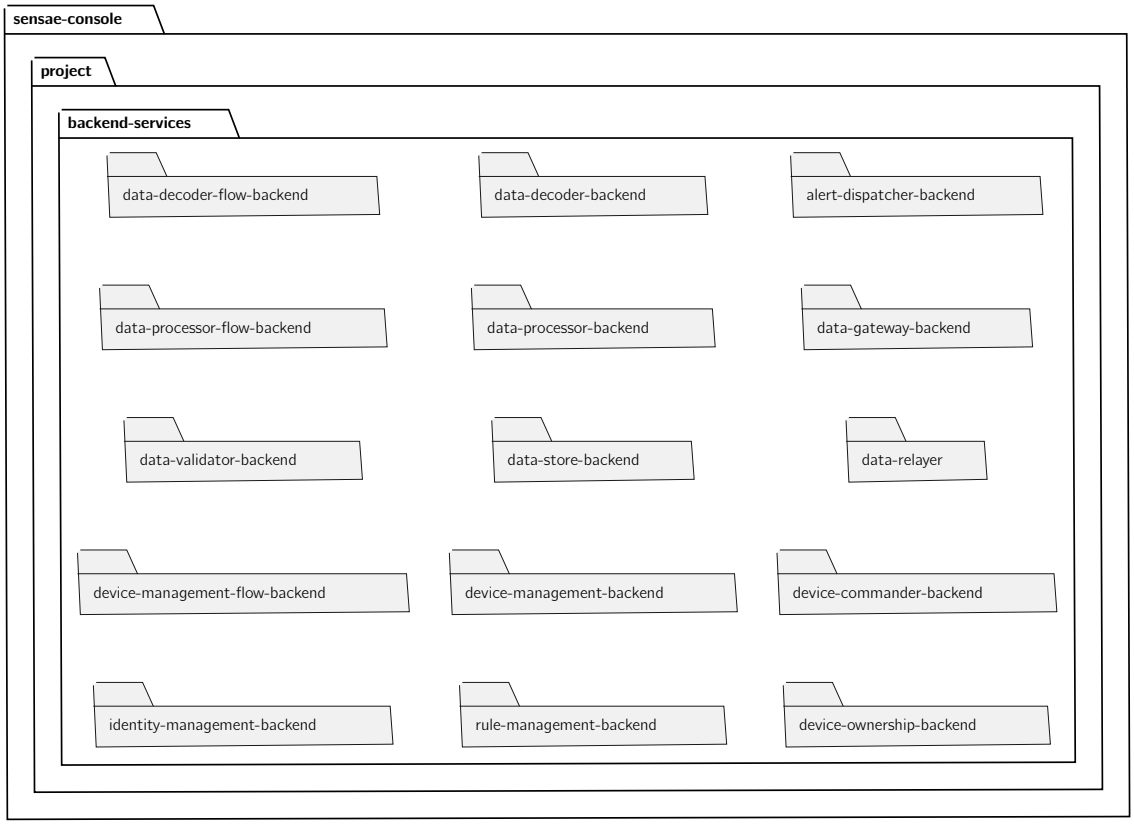


Figure 4.19: Backend Services - Container Level - Implementation View Diagram

Database services are organized according to the diagram in Figure 4.20.

No database service has been developed, only configured. The Message Broker also has no package in the project since it didn't need any configuration and wasn't developed.

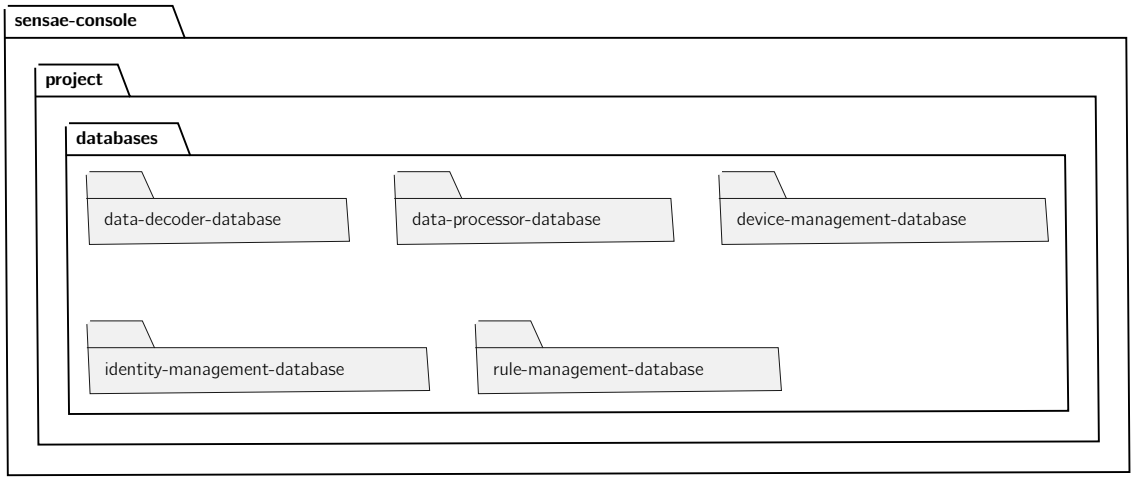


Figure 4.20: Database Services - Container Level - Implementation View Diagram

Frontend services are organized according to the diagram in Figure 4.21.

Each frontend service is divided between the *apps* package and *libs* package. Each *app* depends on the corresponding *lib*. Every *lib* depend on the *core* and *auth* packages. The UI Aggregator depends only on the *auth* package.

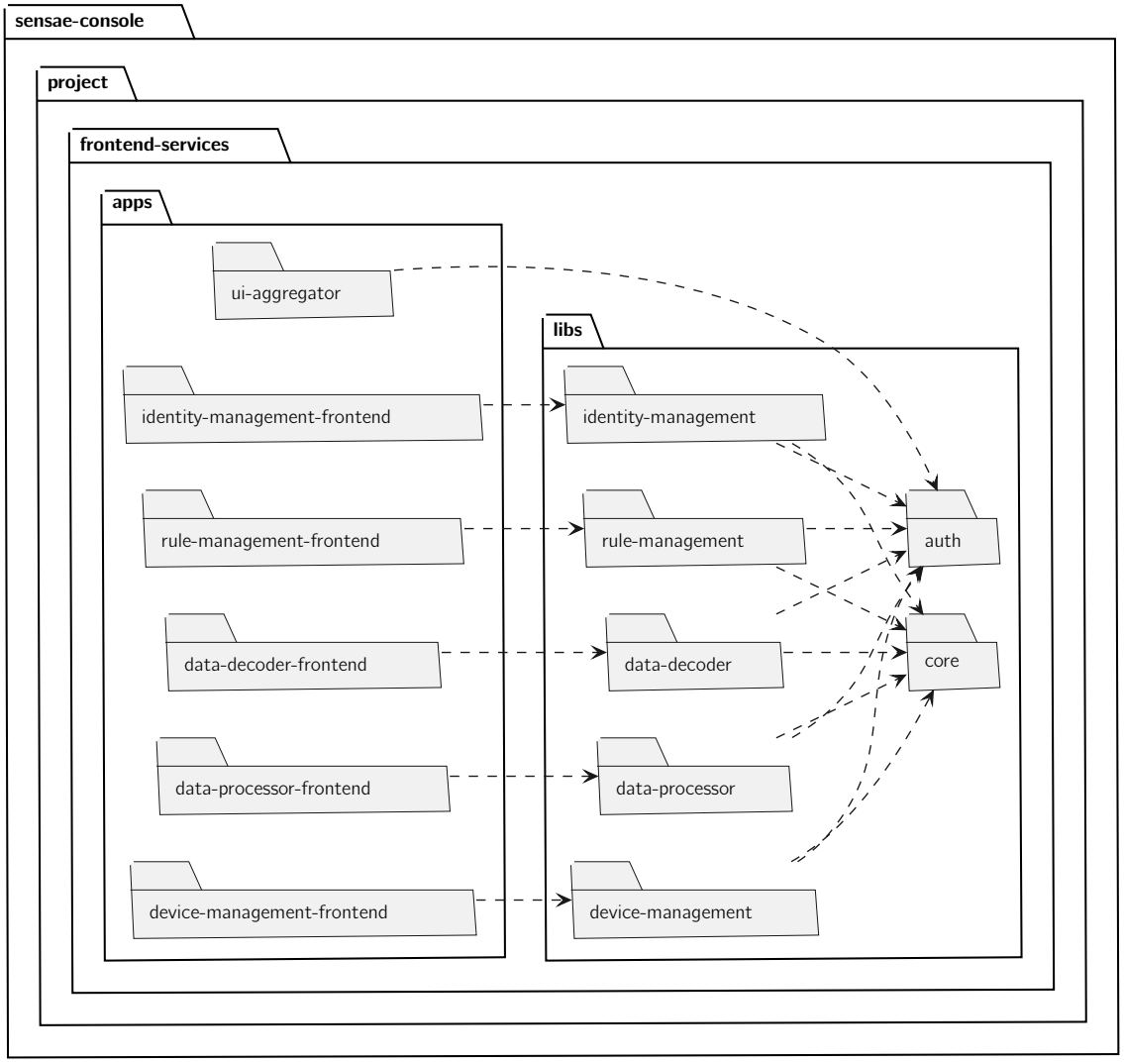


Figure 4.21: Frontend Services - Container Level - Implementation View Diagram

4.2.2.4 Container Level - Physical View

Next is the physical view, intended to familiarize the reader with the idealized production environment. Each container that composes the system is containerized via *Docker* so that orchestration software like *Docker Compose*, *Docker Swarm*, *Kubernetes* and *OpenShift* can be used to ease the operation phase.

The production environment is orchestrated using *Docker Compose* running in a single node/server. This decision was taken after acknowledging that currently there is no need to scale the solution, a single node has been capable of handling all throughput. The Appendix K details how this was implemented.

Each Container represented in Section 4.2.2.1 is mapped to a container in this view. Following the *Database per service Pattern*, each logical database also corresponds to a physical database.

As an example, the physical view of the Rule Management Concern is presented in Figure 4.22. The complete Sensae Console solution is not presented for brevity reasons.

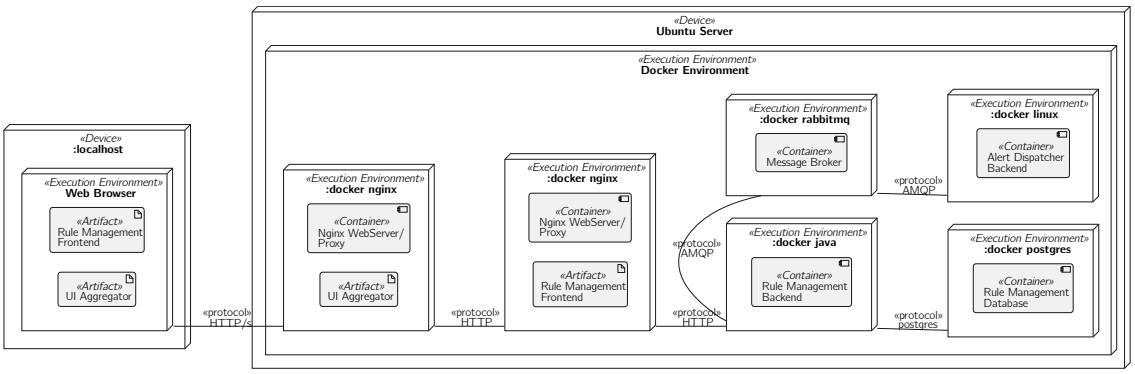


Figure 4.22: Rule Management - Container Level - Physical View Diagram

The two *Device*s presented correspond to a user's machine (localhost) and the server where Sensae Console is deployed (Ubuntu Server). The **UI Aggregator** and **Rule Management Frontend** are served by their corresponding Nginx WebServer, and all user-centered communications with **Sensae Console** are secured and conducted by the UI Aggregator Nginx WebServer.

In the future, if the need arises, **Sensae Console** should be orchestrated with *Kubernetes* or *OpenShift*. This would allow the solution to auto-scale, multiplying the containers under excessive load.

4.2.2.5 Platform - Synopsis

This section presented the architecture used in the **Sensae Console** platform, how software is organized and how some internal process are executed inside this platform. The PoCs developed are discussed in the following section.

4.2.3 Solutions - External Services

This section will explore the details of each External Service developed as a PoC from an architectural point of view. Some of the similarities shared between the architecture of all services are:

- All include a backend that exposes an API;
- All include a frontend that exposes a UI;
- All include at least a database that exposes an API consumed solely by the service's backend;
- Any communication with **Sensae Console** is preformed by consuming the Message Broker's API;

- All follow the idea behind the separation of responsibilities seen in a three layer architecture.

Even though it isn't required, the **UI Aggregator** can be configured to consume the UI and API belonging to each External Service. By doing so, the complete solution, UI and API can be presented under a single FQDN. This view can be seen in Appendix B.

For brevity reasons the C4 level 3 of the solutions will not be discussed, the architecture of most containers follows what was discussed in Appendix C Section. The frontend containers behave exactly as the ones designed for the platform and most backend containers follow the same ideas behind the Configuration Backend Architecture for containers in the platform. The architectures that diverge a bit can be consulted in Appendix E.

4.2.3.1 Fleet Management

The logical view of the Fleet Management service is presented in Figure 4.23.

This service is composed by a simple three layers architecture. The details related to this service are discussed in Section H.1.

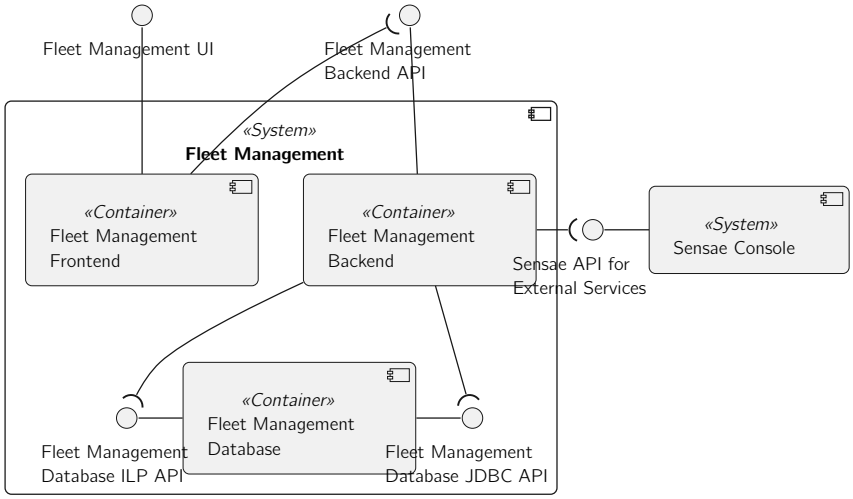


Figure 4.23: Fleet Management - Container Level - Logical View Diagram

Next, to better understand the internal processes of this service, Figure 4.24 presents how a user can see the current location of a device. Authentication details are omitted for brevity reasons.

In order to provide live information to the user, the Fleet Management service (and all other **External Services**) relies on *WebSockets*. A bidirectional channel is created between the frontend and backend so that data can be sent directly from the backend to the frontend as we can see in the step **2.5**. Fist the frontend must subscribe to new information with a valid *access token* - steps **1.2** to **1.6** - then this channel is maintained till the user leaves the page. Once the user leaves the page the subscription is closed in the frontend and subsequently in the backend - steps **3.2** to **3.5**.

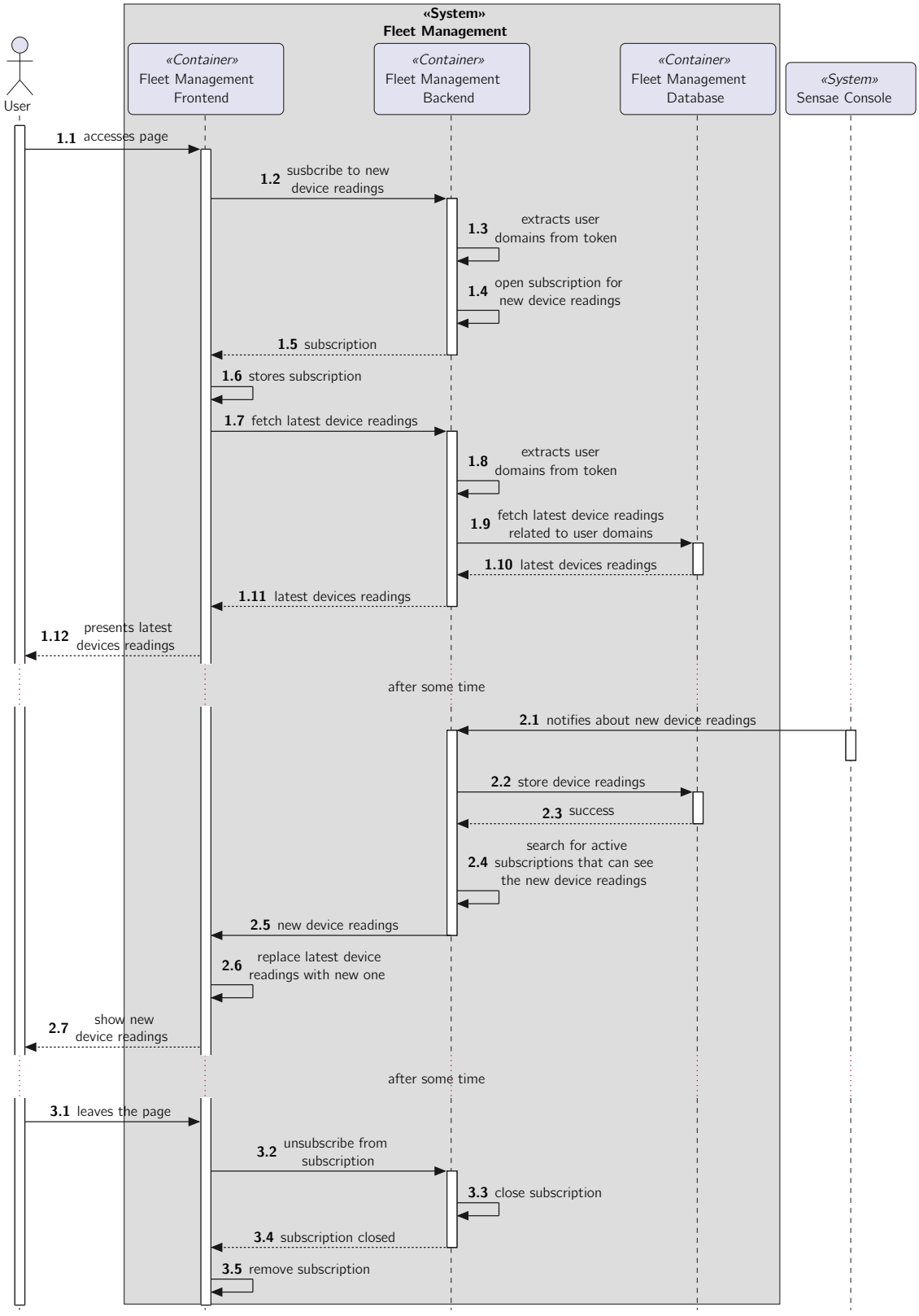


Figure 4.24: Consult Device Live Location - Container Level - Process View Diagram

4.2.3.2 Notification Management

The logical view of Notification Management service is presented in Figure 4.25.

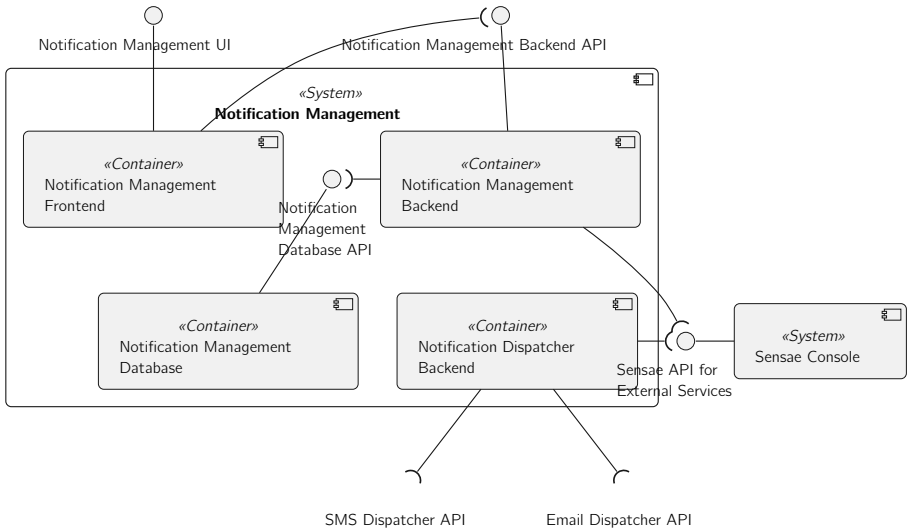


Figure 4.25: Notification Management - Container Level - Logical View Diagram

This service is composed by a simple three layers architecture and has a separated container responsible for dispatching SMS and emails, the Notification Dispatcher Backend container. Information regarding the type of alerts each user is interested in are exchanged between the Backend and Dispatcher containers through the Message Broker. The details related to this service are discussed in Section H.2.

The next diagram in Figure 4.26 describes how a user receives notifications via several different delivery channels. For brevity reasons the subscription process is omitted.

As a brief description this diagram describes what happens when an alert is dispatched inside **Sensae Console**. An alert is created in Alert Dispatcher Backend, flows though Device Ownership Backend to be enriched with the domains that own it and is then collected by, at least, the Notification Management Service. Notification Management Backend deliveries alerts in the form of UI notifications - step **2.5** and **2.6** - and stores this alert as a notification for later use - step **2.3**. Notification Dispatcher Backend deliveries alerts in the form of Emails - step **3.4** - and SMS - step **3.7**.

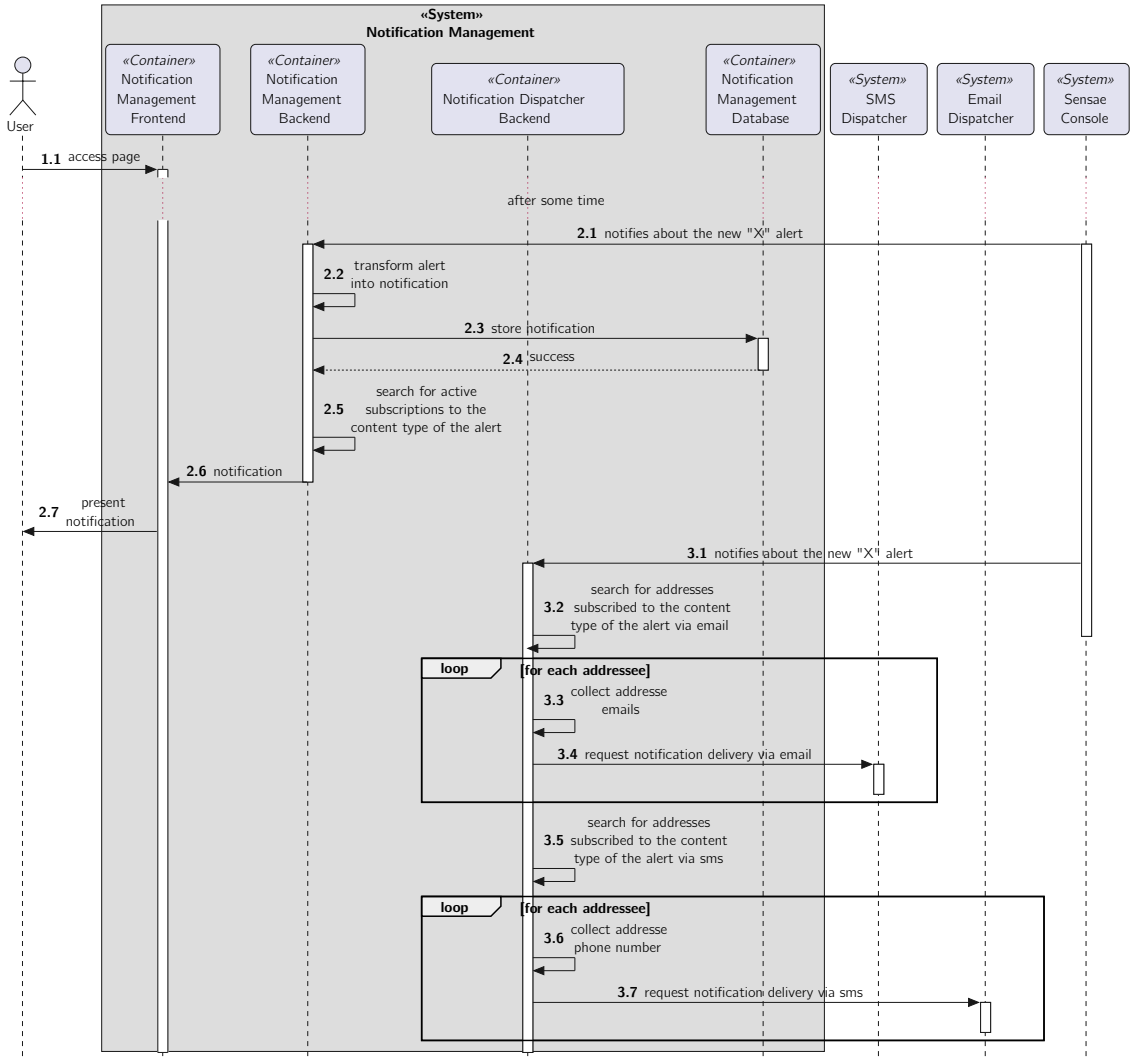


Figure 4.26: Receive Notification - Container Level - Process View Diagram

4.2.3.3 Smart Irrigation

The logical view of the Smart Irrigation service is presented in Figure 4.27.

This service is composed by a three layers architecture, the **Data Layer** is divided in two databases, one responsible for storing device measures and another responsible for storing information regarding the Irrigation Zones and device information. The details related to this service are discussed in Section H.3.

Certain types of alerts are also collected by Smart Irrigation Backend to automatically control conditions inside an irrigation zone. In the next diagram, Figure 4.28, this process is presented.

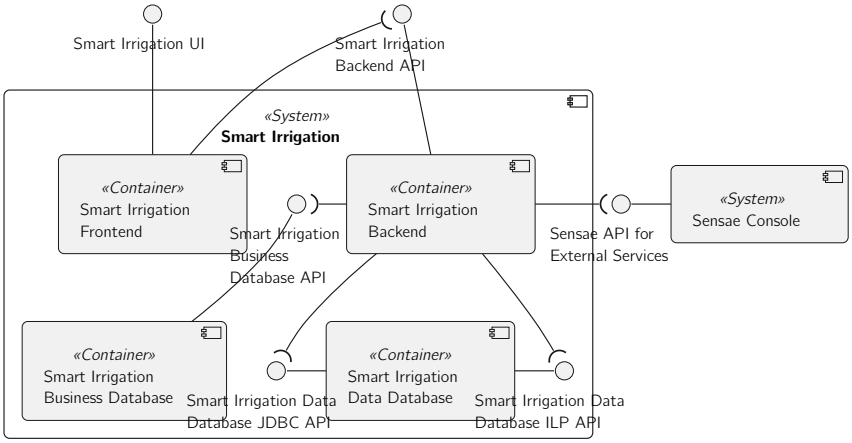


Figure 4.27: Smart Irrigation - Container Level - Logical View Diagram

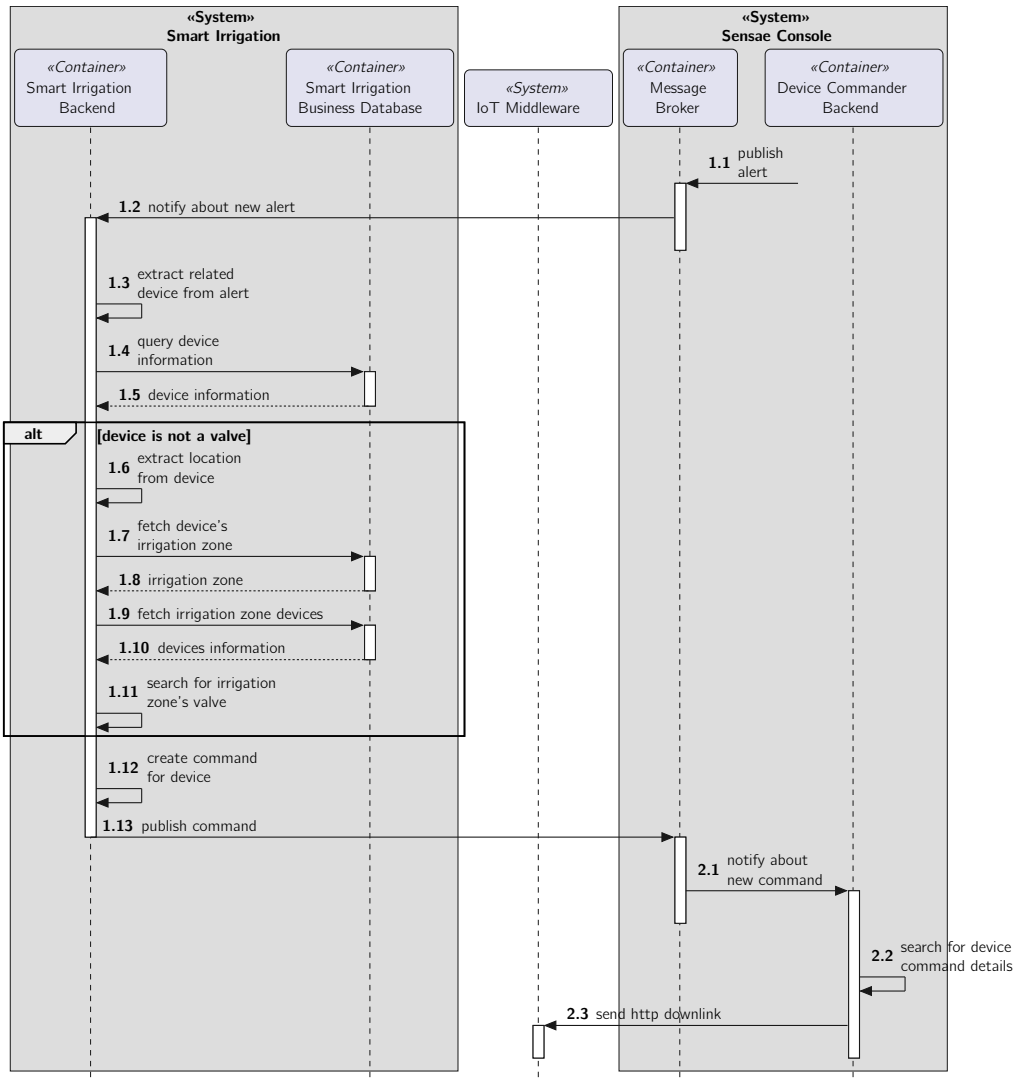


Figure 4.28: Valve Activation - Container Level - Process View Diagram

The alerts created in **Sensae Console** are captured by **External Services**'s containers so that they can immediately act based on those alerts.

The Smart Irrigation Backend subscribes to three types of *Sub Category* alerts, all with the same *Category* - *Smart Irrigation*:

- **Damped Environment**: a valve needs to be closed;
- **Dry Environment**: a valve needs to be open;
- **Valve Open For Lengthy Period**: a valve needs to be close.

4.2.4 Synopsis

This section discussed the architecture used in the platform and developed PoCs. It presented how some internal processes are handled by the system as a whole. In the following section, alternatives to what was designed and developed across the system are discussed.

4.3 Architectural Alternatives Discussed

This section tackles important alternatives that were proposed and discussed during the design and development of the system but were discarded in detriment for the approaches presented in the Architectural Design.

More alternatives are discussed in the appendixes, namely User Authentication, in Appendix F.

4.3.1 Backend Segregation

There are three main architectural approaches to this topic: Monolithic Backend - Richardson 2021b -, SOA - IBM 2021c - or Microservices - Fowler and J. Lewis 2014. The first question regarding what to choose is whether to split the system in multiple units of work: Monolith vs the other two approaches.

If the decision is to split the system, then an important question must be asked: how should one split the system? The system architecture depends on the answer given: a SOA emphasizes the reuse of the system functionalities, IBM 2021c, while Micro Services emphasis the decoupling of the various system components - Richardson 2021a - and can therefore introduce some functionality duplication as opposed to SOA - Powell 2021.

But, to pick one of this architectures, the most important question to ask is: Why do i need architecture X? To answer this, a set of the concerns deemed more important, with regards to this solution requirements, are discussed:

- Time To Market: a MVP should be available and ready to use as soon as possible;
- Extensibility of the solution: it should be easy to extend the solution with new External Services;
- Operation Cost: the solution has to be efficient to lower the infrastructure costs, tied to the system performance;

- System performance: the solution has to be capable of processing high volumes of data efficiently, tied to the system performance.

The first concern, Time to Market, weights heavily in favor of the Monolith approach when developing a MVP, Harris n.d. This approach is simpler to develop, deploy and has less cognitive overhead when compared to the other two approaches.

Regarding the extensibility of the solution, a Monolith is inherently rigid and hard to extend as the business evolves. This problem is inflated by the fact that the business model envisioned relies heavily on the creation of several External Services. On the order hand the SOA and Microservices architecture are preferred since, due to their inherently decoupled nature, they are easier to extend using the interfaces they expose - Jacobs and Casey 2022.

The last two concerns are related to the scalability of the solution. A Monolithic Backend can only be scaled up by increasing the resources - RAM, CPU, GPU and Dick Capacity - of the physical server where the solution is deployed, this is commonly referred as Vertical Scaling. A SOA or Micro Service Backend Architecture, apart from the Vertical Scaling option, can also be scaled up by increasing the number of physical servers where the solution is deployed, this is commonly referred as Horizontal Scaling.

Another option, that can be used with any architecture, is to deploy various independent instances of the same solution. Each instance would be assign to a set of costumers. This option is crucial and always possible once the business grows and starts to assist various customers.

The following picture, Figure 4.29, summarizes how each architect scales, the SOA behaves similarly to the microservices architecture presented.

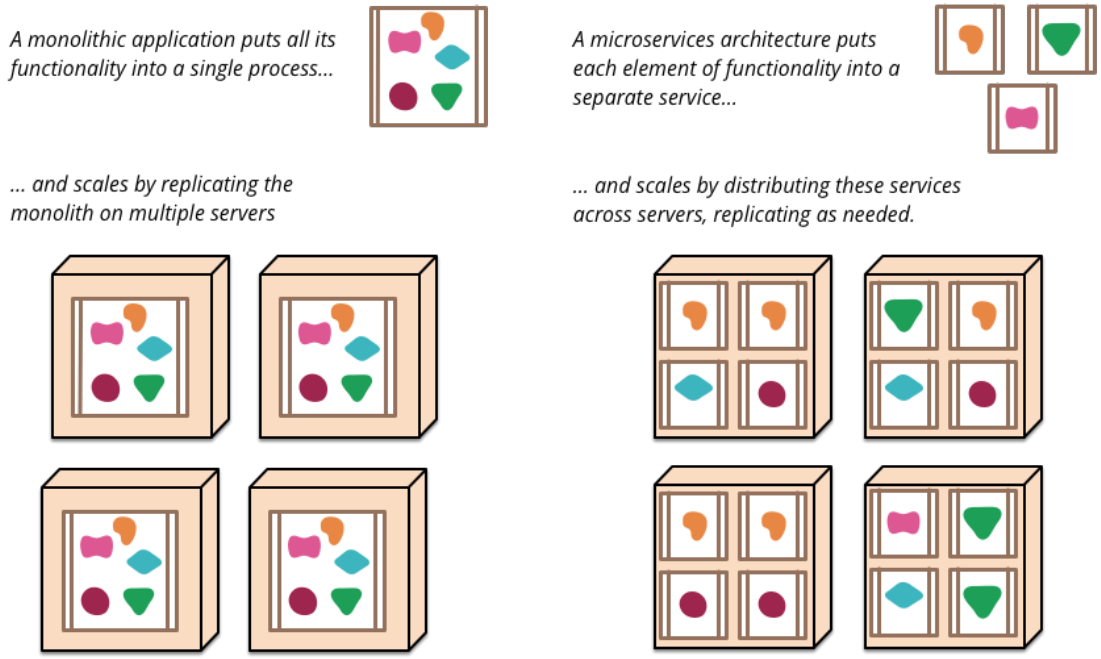


Figure 4.29: Monoliths and Microservices by Fowler and J. Lewis 2014

The final decision was to follow an architecture based on Microservices, even tho this decision had several oversights:

- Development Team size: microservices are commonly adopted by big companies where each team of developers is responsible for a subset of microservices. This lowers the friction between teams when developing and deploying the solution and is seen as a big reason to move to a microservice architecture. For this solution, a single developer is responsible for everything;
- Time to Market: microservices need to interact with each other though the network. This added demand takes time to design and develop when compared to a monolith solution where communication is done via code;
- A solution shouldn't start with a microservice architecture: a solution should migrate to microservices when it becomes too complex and hard to maintain, IBM 2021a.

The decision made was based on the following assumptions, perceptions and findings:

- There are well defined boundaries between the various business processes that the project needs to support;
- There is a perception that the solution will need to scale early on the road due to high volumes of IoT data to process and store;
- There are a high number of completely independent external services to develop and deploy;
- There are different types of costumers with diverse requirements regarding the deployment and development of the solution;
- Each costumer is interested in their specific business case or cases and therefore requires different combinations of external services.

SOA was discarded since: "Although the concept of a share-as-much-as-possible architecture solves issues associated with the duplication of business functionality, it also tends to lead to tightly coupled components and increases the overall risk associated with change". (Richards 2015). Microservices are more easily extended when/if needed compared with SOA since the focus is on loose coupling services and not highly reusable services.

Despite this, the solution adopted some architecture decisions that are usually associated with SOA, as an example a canonical data model (Section 4.4.2) was created to ease the communication between services. This is something common in projects that follow SOA according to Cerny, Donahoo, and Pechanec 2017.

4.3.2 Frontend Segregation

This section tackles the need for segregating the frontend into various independent frontends - Microfrontends, Geers 2017 - or to develop a single Frontend to answer the identified requirements.

The non-functional requirements discussed in Section 3.2 enhance the need to develop a product that can be fully extensible and yet close for modifications, following the idea behind the Open/Close Principle (OCP) (introduced by Martin 2003). This need arises so that costumer entities can easily create new external services without the need to alter any close source code that is produced internally.

The Microfrontends Architecture when applied to this project has the same oversights, assumptions and perceptions that lead to the decision taken in the Backend Segregation Section. As such, the decision was to drop the design and development of a single frontend in favor of a Microfrontends Architecture.

Ultimately this decision, coupled with the Backend Segregation decision made, enforces a business model that follows OCP and simplifies the adoption of this solution by third parties.

4.3.3 Data Flow Pipeline

This section debates how the various Data Flow Containers should communicate with each other.

Synchronous communication, such as HTTP requests, was promptly discarded since there is no need for each Container to acknowledge the outcome of the Data Unit that it sent and this type of communication would linger the performance of the Data Flow Scope by creating chained requests, an anti pattern when using a Microservice Architecture (Nish Anil and Veloso 2022b).

According to Nish Anil and Veloso 2022a, there are two kinds of asynchronous messaging communication: single receiver message-based communication, and multiple receivers message-based communication. It is common to use both of this types in the same solution depending on the requirements. This type of communication is usually composed by the following participants:

- Broker: responsible for establishing a communication channel between Receivers and Publishers;
- Publishers: responsible for sending messages;
- Receivers: responsible for consuming messages.

Looking at the Figure 4.12 it appears that a simple *single receiver message-based communication* would be sufficient but this approach isn't as flexible as other options. By following a *multiple receivers message-based communication*, additional receivers can be added in the future without the need to modify the sender service. As an example, the Data Store container can be configured to consume any type of Data Unit without changing the containers that produce them.

The final issue to discuss is whether Receivers should pull messages from the Broker (via pulling) or the Broker should push messages to Receivers. This topic is discussed in *Kafka Design: The Consumer*, mentioned as Push vs Pull. Pushing messages to Receivers can overwhelm a receiver when its rate of consumption falls below the rate of production. The Pull approach offers Receivers the option to consume messages at the rate that they are capable of but can be wasteful in systems where messages are not abundant (Klishin 2022). The operations preformed in each Data Flow container are meant to be fast and simple, and as such, overwhelming a receiver was not taken into consideration. The Push approach was preferred since it theoretically enables faster reactions to new message compared to the Push approach.

As such, it was decided that the Data Flow Pipeline would work based on the publish/subscribe pattern on top of asynchronous messaging communication. Messages would be published to a broker and then routed to consumers.

4.3.4 Internal Communication

This section tackles how the Data Flow Scope should be kept up to date on the configurations made in the Configuration Scope. Five alternatives have been discussed:

1. Data Flow Containers directly access the Database related to their concern;
2. Data Flow Containers request information to their concern's Configuration Scope Container via synchronous calls;
3. Data Flow Containers are feed updates to their concern configurations via asynchronous calls and store this information;
4. A shared, in memory, database is kept, Configuration Scope writes to it and Data Flow Scope queries information from it;
5. An append-only log is used to store configuration logs, the Configuration Scope writes to it and the Data Flow Scope can always read from it.

The third option was the approach taken.

4.3.4.1 First Option

This option ensures that the Data Flow Containers are kept updated by giving them direct access to the source of truth, the database. The logical view diagram in Figure 4.30 describes how this option functions.

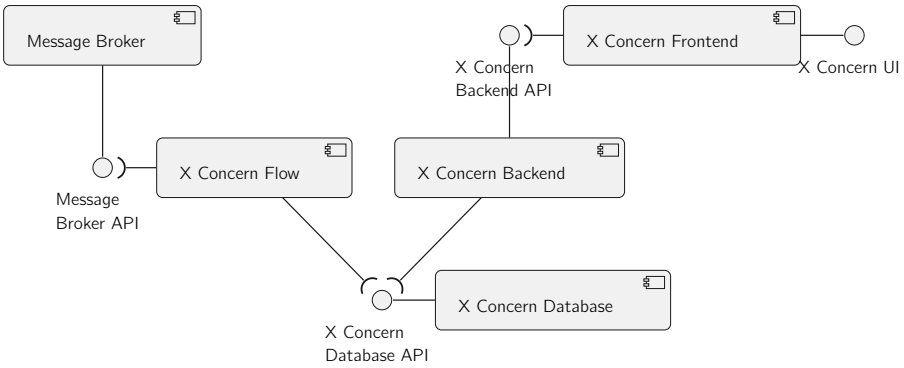


Figure 4.30: Internal Communication - First Option - Logical View Diagram

This approach ensures that the Message Broker is only used to transport Data Units, Alerts and Commands, alleviating it from an heavy responsibility. That responsibility is assigned to the *X Concern Flow* Container and the *X Concern Database* Container. This approach has several drawbacks such as:

- The *X Concern Flow* Container has full access to superfluous configuration details related to that context configuration;
- The same database access has to be developed and maintained in two separated containers;
- All database accesses are blocking calls by nature that would slow down the process;

- Data Flow containers can't reliably cache information collected since there is no way to know when the corresponding information was updated. Meaning that every time a new message arrives the database has to be queried.

Due to this drawbacks this option was eventually dropped.

4.3.4.2 Second Option

This option ensures that the Data Flow Containers are kept updated querying information from a Representational State Transfer (REST) API provided by the Configuration Containers. The logical view diagram in Figure 4.31 describes how this option functions.

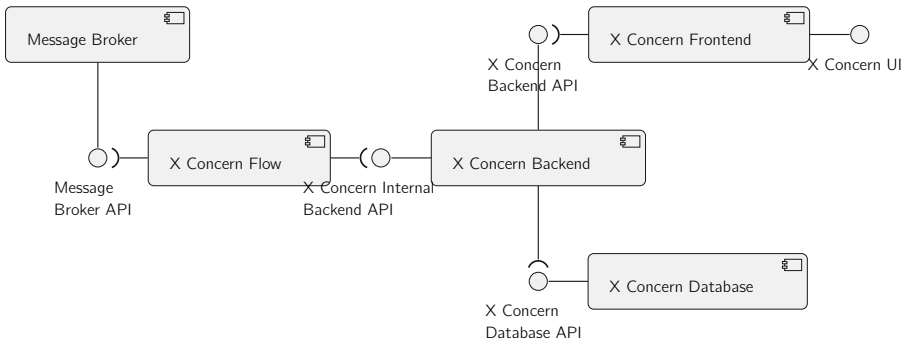


Figure 4.31: Internal Communication - Second Option - Logical View Diagram

This approach doesn't suffer from all drawbacks stated for the first option but still requires a blocking call to the *X Concern Backend* Container every time a new message arrives to the *X Concern Flow* Container.

It's an improvement of the first option but still has some serious drawbacks and therefore it was also abandoned.

4.3.4.3 Third Option

This option ensures that the Data Flow Containers are kept updated by allowing them to subscribe to changes made in their concern's configuration. The logical view diagram in Figure 4.32 describes how this option functions.

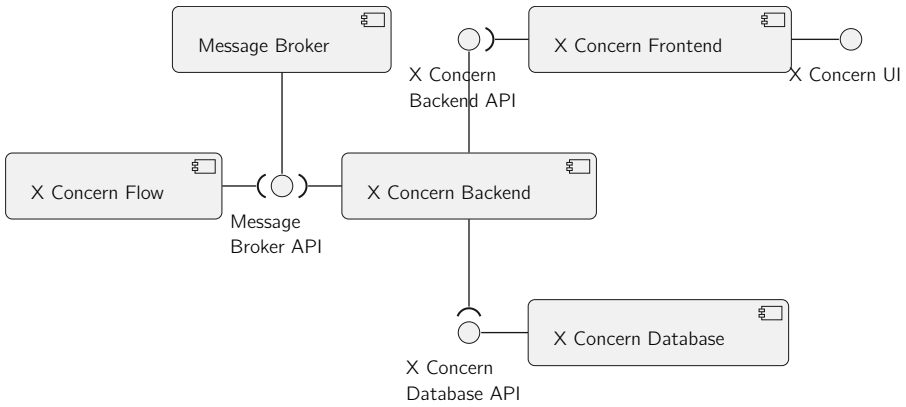


Figure 4.32: Internal Communication - Third Option - Logical View Diagram

The major improvement of this approach when compared with the options above is that, since *X Concern Flow* Container subscribes to configuration updates, it can reliably keep a cache with just the needed information (and not the entire concern configuration). This works since *X Concern Flow* Containers can discard updates related to information that they currently don't use. Once the container needs that information, it can send an event requesting what it needs and that information arrives later as a normal update to the configuration. All *X Concern Flow* external interactions also rely on asynchronous communication, ensuring a more robust performance.

The main drawback to this option is that the *Message Broker* becomes responsible for yet another communication topic inside the environment.

Despite this drawback this is the option currently in use. The following options purpose alternatives to tackle this drawback.

4.3.4.4 Fourth Option

This option ensures that the Data Flow Containers are kept updated by allowing them to query information from an *Internal State Database*. This approach differs from the first option since the *Internal State Database* is supposed to be a fast in memory database with only the needed information for Data Flow Containers to process Data Units, Alerts and Commands. The logical view diagram in Figure 4.33 describes how this option functions.

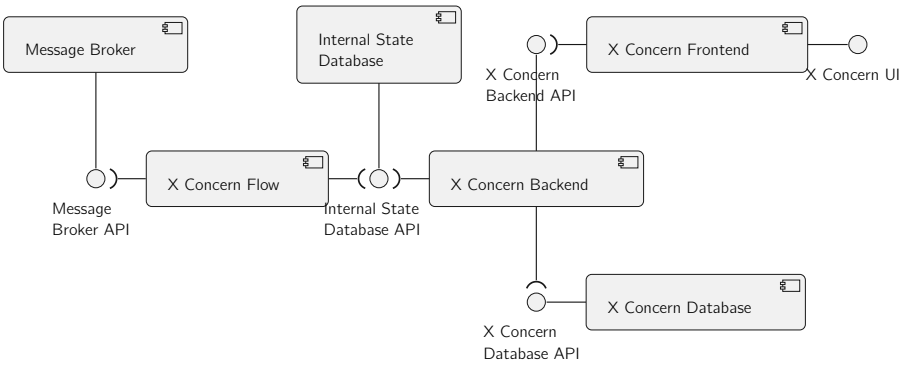


Figure 4.33: Internal Communication - Fourth Option - Logical View Diagram

This approach would remove the responsibly from the *Message Broker* to maintaining the internal state updated in the Data Flow Scope. The *Internal State Database* would in turn store information that *X Content Flow* could query.

The main drawbacks of this approach are the same stated in the second option, even tho they can be mitigated by leveraging technologies that tackle distributed caching problems.

4.3.4.5 Fifth Option

This option ensures that the Data Flow Containers are kept updated by allowing them to subscribe to changes made in their concern's configuration. This option diverges from the third option since the event store would persist all updates to concerns configurations. The logical view diagram in Figure 4.34 describes how this option functions.

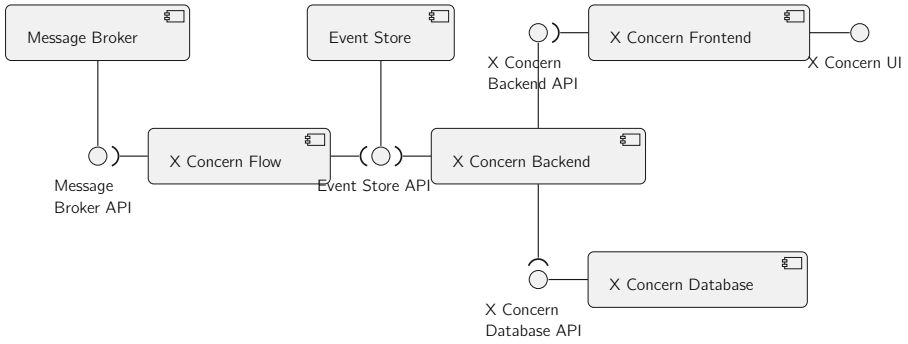


Figure 4.34: Internal Communication - Fifth Option - Logical View Diagram

The *X Concern Flow* Container would use event sourcing to reach the current state of its concern configuration on start up and then cache this state internally. New events would then be sent automatically via subscription to keep the state up-to-date.

The main drawback of this approach is that the container can't keep just the needed portion of configurations without recreating the entire state though event sourcing.

4.4 Canonical Model

The idea behind this section is to introduce core communication concepts of **Sensae Console** to the reader. To represent this ideas the UML notation is used.

This section is split into two pieces: (i) taxonomy and (ii) model. The various domains inside **Sensae Console** and **External Services** are discussed in Appendix G and H, respectively.

4.4.1 Taxonomy

In order for the reader to better understand how the system operates, some concepts need to be better classified and explained:

- **Device**: A device is a "Thing" that can collect data and submit it to **Sensae Console** via an external system though **Uplink**s (commonly refereed as a sensor). A device can also receive **Downlink**s and act base on what was received (commonly refereed as an actuator);
- **Controller**: A controller is a **Device** that controls and aggregates data from various sub **Device**s;
- **Records/Metadata**: Records, or Metadata are labels associated to a **Device** that help an organization to classify and add some information to the owned **Device**s;
- **Downlink**: A downlink is a term commonly used in radio communications to denote the transmission from the network to the end user. In this case the network is the **Sensae Console** and the end user is a **Device**;
- **Uplink**: An uplink is the opposite of a **Downlink**, it's the transmission from a **Device** to the **Sensae Console**;

- **Data Unit**: A data unit represents the collected measures that are atomically submitted via an **Uplink** to the **Sensae Console**. This data should be, at least, enriched with an unique identifier of the **Uplink** and **Device** that sent it. The data unit can contain measures captured by various devices, in that case the device is identified as a Controller;
- **Device Command**: A device command is an abstraction on top of a **Downlink**, intended to instruct a **Device** to execute a specific action. This devices are commonly identified as actuators. As an example, one could send a command to open or close a valve that is incorporated into a **Device**;
- **Decoder**: A decoder is a function that translates a **Data Unit** into something that **Sensae Console** understands;
- **Domain**: A domain represents a department in a organization. An organization is composed of several domains structured in a tree like format;
- **Tenant**: A tenant is a user that belongs to one or more **Domain**s and represents any of the roles discussed in Section 3.1;
- **Alert**: A report about a detected condition based on the gather **Data Unit**;
- **Topic**: A Topic is a subcategory of the type of contents that are traded between the various containers in the system.

Currently the **Topic**s that flow in the system are:

- **Data**: This topic references the **Data Unit** concept and is intended to be processed by the **Data Flow Scope** and consumed by the **External Services**;
- **Command**: This topic references the **Device Command** concept and is intended to be used mainly by the **External Services**;
- **Alert**: This topic references the **Alert** concept and is intended to be consumed mainly by the **External Services**;
- **Internal**: This topic references the internal state maintained in the **Configuration Scope** and **Data Flow Scope**.

This concepts are referenced across the document.

4.4.2 Model

The canonical model is comprised of concepts that transverse the entire **Sensae Console** business model, and by extension any **External Service**. Therefore, it is built as a library, *iot-core*, that can be used by entities that rely on the exchange of information with/inside **Sensae Console**. It can be seen as a domain that focus on defining the protocol of exchange of information between the various entities of the system.

The intent behind this Shared Model is to alleviate one of the issues related to distributed systems - heterogeneity in data formats (Nadiminti, De Assunçao, and Buyya 2006) - and to provide a simple SDK for third-parties to develop new external services that interact with **Sensae Console**. It can be seen as an explicit schema. According to Bellemare 2020, “any implementation of event-based communication between a producer and consumer that lacks an explicit predefined schema will inevitably end up relying on an implicit

schema. Implicit data contracts are brittle and susceptible to uncontrolled change, which can cause much undue hardship to downstream consumers."

According to Byars 2021, "while we have historically drawn up our project plans and costs around the boxes the digital products we are introducing the lines are the hidden and often primary driver of organizational tech debt. They are the reason that things just take longer now than they used to." The *'lines'* in this solution are a first class citizen and, instead of just linking the system together, they act as the pillars that shape the entire ecosystem.

It is comprised of three big components: (i) data model, (ii) message envelop model, and (iii) routing model.

4.4.2.1 Data Model

The data model represents the **Data Unit** that **Sensae Console** is currently capable of understanding. The following diagram, Figure 4.35, is a high level specification.

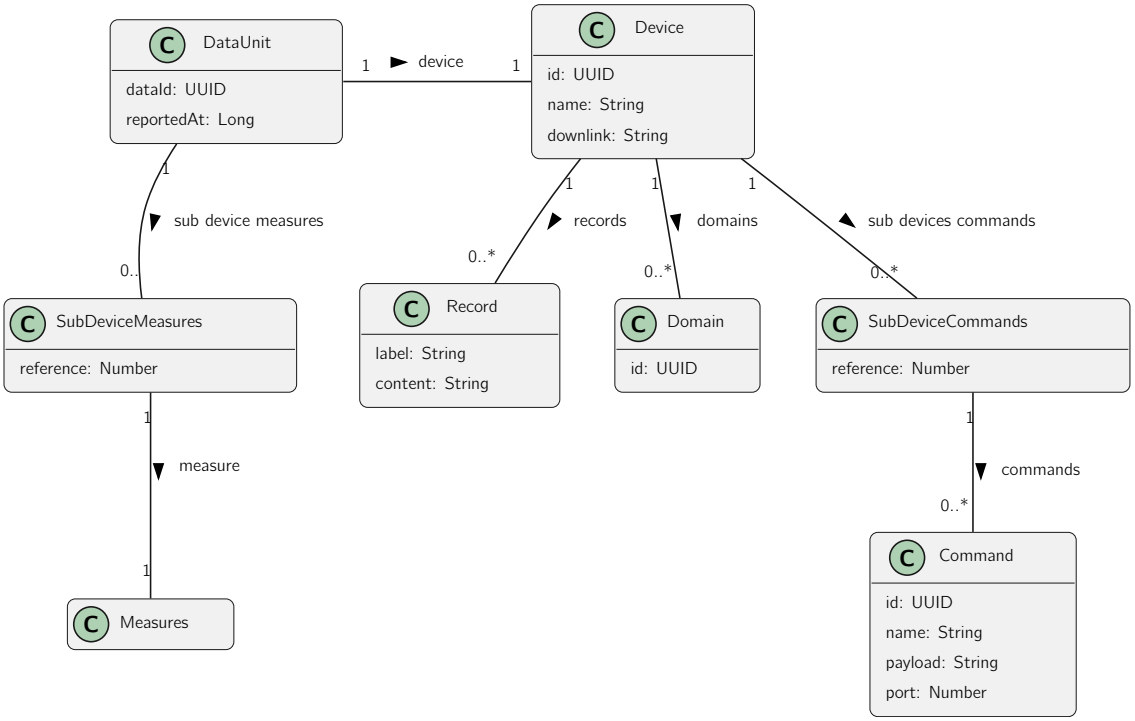


Figure 4.35: Canonical Model - Data Unit

As a brief description:

- **Data Unit** is the entry point to the shared model;
- The *reportedAt* attribute represents an absolute timestamp of when the **Data Unit** was captured, in milliseconds;
- The *Device* concept represents the **Device** that sent the *Data Unit*;
- The *Record* concept represents an entry of **Records/Metadata**;
- The *Domain* concept references the **Domain** that owns the *Device*;

- The *SubDeviceMeasures* referes to the collected measures. When the device is a Controller there's a need to map each sub device's measures with it and not with the Controller that sent the uplink. The *reference* attribute indicates what sub device collected the measure, the reference *zero* referes to the device that sent the uplink;
- The *SubDeviceCommands* referes to the available commands to control the device. When the device is a Controller there's a need to map each sub device commands with it and not with the Controller that sent the uplink. The *reference* attribute indicates the sub device that is controlled by the commands mentioned, the reference *zero* referes to the device that sent the uplink;
- The *Measures* concept contains various common data types related to IoT.

As explained, *Measures* contains various data types. Currently the supported types are presented in the Table 4.2. The team involved in this project decided what data types were needed to support based on the requested PoCs and the purchased sensors. In the future more data types are expected to be included in the model.

The full json-like model schema can be found in Appendix A.

Table 4.2: Measure Data Types

Data Type *Property*	*Sub Property*	**Description**	**Unit**
Trigger		Type related to something with an on / off or open / close state	
trigger	*value*	Value can be true or false	boolean
Motion		Status related to the motion of a device	
motion	*value*	Value can be "ACTIVE", "INACTIVE" or "UNKNOWN"	n.a.
Velocity		How fast a device is moving	
velocity	*kilometerPerHour*	Value measured in	km/h
Temperature		Temperature measured by a device	
temperature	*celsius*	Value measured in	celsius
AQI		Air Quality Index according to the U.S. AQI	
aqi	*value*	Value measured in	AQI
Air Pressure		Pressure within the atmosphere of Earth	
airPressure	*hectoPascal*	Value measured in	hPa
Distance		Distance measured from the device to a surface	
	millimeters	Value measured in	mm
distance	*maxMillimeters*	Maximum distance the sensor can be to a given surface	mm
	minMillimeters	Minimum distance the sensor can be to a given surface	mm
Soil Moisture		Amount of water, including water vapor, in an unsaturated soil	
soilMoisture	*relativePercentage*	Value measured in	%
Water Pressure		Water Pressure measured in pipes by a device	
waterPressure	*bar*	Value measured in	bar

Table 4.2 continued from previous page

Data Type *Property*	*Sub Property*	**Description**	**Unit**
Illuminance		Illuminance level - luminous flux per unit area	
illuminance	*lux*	Value measured in	lux
CO2		Atmospheric Carbon Dioxide concentration	
co2	*ppm*	Value measured in	ppm
CO		Atmospheric Carbon Oxide concentration	
co	*ppm*	Value measured in	ppm
VOC		Volatile Organic Compounds concentration measured by a device	
voc	*ppm*	Value measured in	ppm
NH3		Atmospheric Ammonia concentration	
nh3	*ppm*	Value measured in	ppm
O3		Atmospheric Ozone concentration measured by a device	
o3	*ppm*	Value measured in	ppm
NO2		Atmospheric Nitrogen dioxide concentration	
no2	*ppm*	Value measured in	ppm
PM2.5		Particulate Matter in the air (size up to 2.5 micrometers)	
pm2_5	*microGramsPerCubicMeter*	Value measured in	μg/m3
PM10		Particulate Matter in the air (size up to 10 micrometers)	
pm10	*microGramsPerCubicMeter*	Value measured in	μg/m3
pH		Scale used to specify how acidic or basic a water-based solution is	
ph	*value*	Value between 0 and 14 measured in	pH

Table 4.2 continued from previous page

Data Type *Property*	*Sub Property*	**Description**	**Unit**
Occupation		Occupation percentage measured inside a vessel	
occupation	*percentage*	Value measured in	%
Soil Conductivity		Substances ability to conduct an electrical current in the soil	
soilConductivity	*microSiemensPerCentimeter*	Value measured in	μS/cm
Air Humidity		Concentration of water vapour present in the air	
airHumidity	*gramsPerCubicMeter*	Value measured in	g/m3
	relativePercentage	Value measured in	%
GPS		Point reference in the Geographic Coordinate System	
gps	*latitude*	Value between -90 and 90 measured in	degrees
	longitude	Value between -180 and 180 measured in	degrees
	altitude	Value determined according to the mean sea level	meters
Battery		Battery of the device	
battery	*volts*	Value measured in	volts
	percentage	Value measured in	%
	maxVolts	Minimum volts the battery needs for the device to work	volts
	minVolts	Maximum volts the battery can hold	volts

4.4.2.2 Message Envelop Model

The message envelop model refers to how, coupled with the routing model in Section 4.4.2.3, information can reliably transverse the system.

The diagram present in Figure 4.36 details this model.

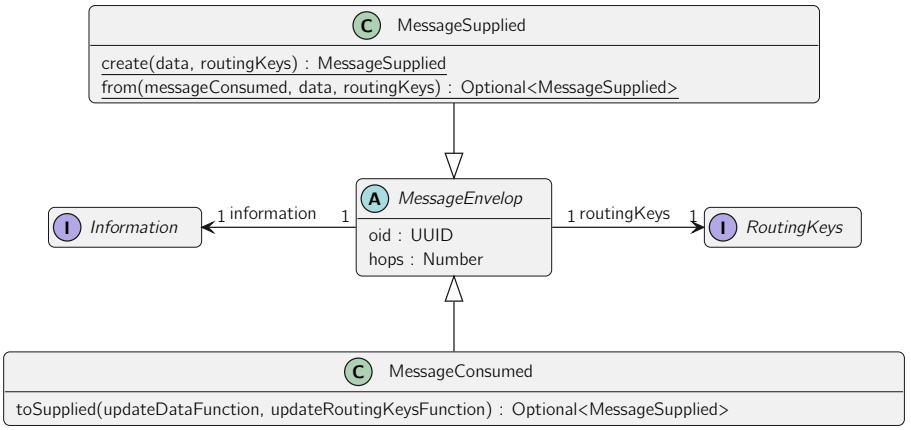


Figure 4.36: Canonical Model - Message Envelop Model

As a brief description:

- A *MessageSupplied* is created in a issuer system and supplied to start the flow of information in the system;
- A *MessageConsumed* is consumed by a consumer system and can then be transformed into a *MessageSupplied* to be supplied;
- *Information* represents the content of the message;
- *RoutingKeys* represents the model referenced in Section 4.4.2.3;

This concept is mainly used to ensure that information flowing in the system is not reprocessed, by verifying the unique id - *oid*, and is eliminated if it enters a routing loop by verifying that the *hops* have not reached a maximum value.

4.4.2.3 Routing Model

The routing model refers to how information can be routed through the system based on various parameters. The current idea is based on the *pub/sub* pattern, as discussed by Urquhart 2021. Containers subscribe to information in a **Topic** with specific *RoutingKey*s and publish information with *RoutingKey*s.

The diagram presented in Figure 4.37 details this model.

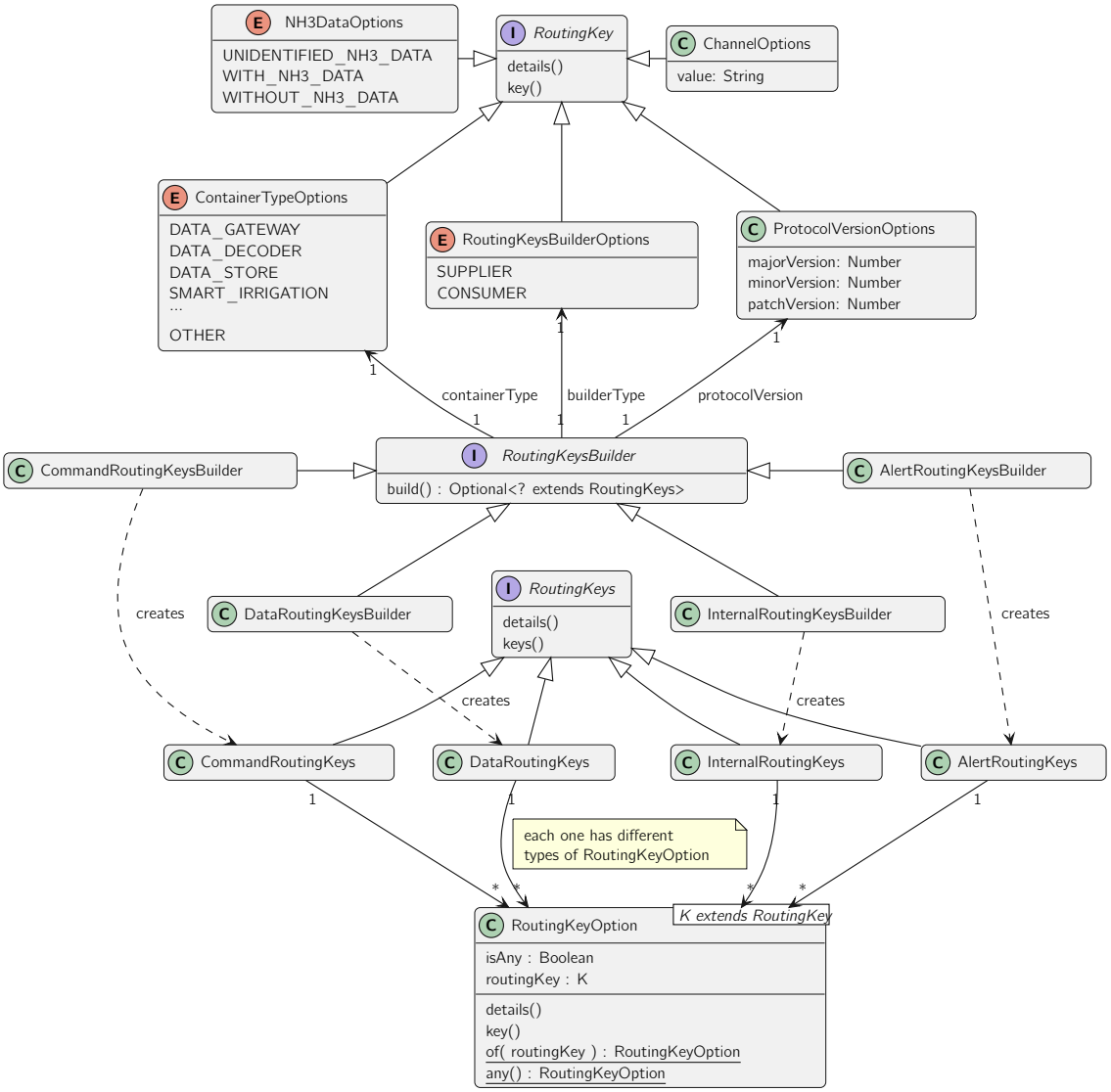


Figure 4.37: Canonical Model - Routing

As a brief description:

- *RoutingKeys* is the concept referenced in Figure 4.36 and represents a collection of different *RoutingKeyOptions*;
- There are 4 types of *RoutingKeys*, one for each **Topic** (according to Taxonomy);
- To ensure that the various containers in **Sensae Console** understand each other, a *ProtocolVersionOptions* is provided. This concept follows the Semantic Versioning Specification 2.0 (Preston-Werner 2011) an is assembled according to the version of *iot-core* imported by the container;
- There are multiple *RoutingKey* types not displayed in the diagram for brevity;
- A *RoutingKeyOption* can have the value *any*, if the *RoutingKeysBuilderOptions* has the value *CONSUMER*. This provides a 'relaxed' mode, for containers that consume/subscribe to messages and a 'strict' mode, where all *RoutingKey* must be specified, for containers that supply/publish messages;

- The *RoutingKeysBuilder* implements the *Builder* pattern and its single responsibility is to validate and create *RoutingKeys*;
- *NH3DataOptions* and *ChannelOptions* are two examples of *RoutingKey*, both used in the Data Topic.

Table 4.3 presents all currently used *RoutingKey*s.

The routing key *OperationType* from the **Internal** topic can have the following values:

- **Sync**: message contains the current state of the related *ContextType*, used to populate a container's state;
- **Info**: message contains information about an entry of the related *ContextType*, e.g. entry X in context Y was removed;
- **Unknown**: message contains entry of the related *ContextType* that the container that published the message can't identify;
- **Init**: message to notify that a container has initiated and needs the current state of the related *ContextType* to be ready;
- **Ping**: message to notify that an entry of the related *ContextType* was used, e.g. entry X in context Y was just used.

The *ContextType*, used to identity what piece of the state is referenced, can currently have the following values: (i) *Data Processor*, (ii) *Data Decoder*, (iii) *Device Information*, (iv) *Device Identity*, (v) *Tenant Identity*, (vi) *Addressee Configuration* and (vii) *Rule Management*.

Routing keys help to strengthen the boundaries that a container is expected to have. As an example, an external service related to Waste Management would subscribe to the *Data Topic* with the following *Routing Keys*:

- *Info Type Options*: PROCESSED;
- *Channel Options*: 'wasteManagement';
- *Data Legitimacy Options*: CORRECT;
- *GPS Data Options*: WITH;
- *Occupation Data Options*: WITH;
- *Records Options*: WITH;
- *Ownership Options*: WITH.

And would, for example, subscribe to the *Alert Topic* with the following *Routing Keys*:

- *Alert Category Options*: 'wasteManagement';
- *Alert SubCategory Options*: 'garbageFull';
- *Ownership Options*: WITH.

As expected, the structure and semantics of the information subscribed to are known upfront with the help of the package *iot-core*. The services developed and their pre-defined boundaries regarding data types consumed are detailed in Section 5.2.7.

Table 4.3: Routing Options

Topic *Routing Key*	**Description**
Common	Routing Keys that belong to every Topic
Protocol Version Options	Version of the used *iot-core* package
Container Type Options	Type of the Container that published the message
Ownership Options	Does the message contains the **Domain**s that own it[1]
Topic Type Options	Topic used to publish the message
Internal	Routing Keys that belong to the Internal Topic
Operation Type Options	Intent of the message, e.g. unknown context found
Context Type Options	Type of content in the message, e.g. device information
Data	Routing Keys that belong to the Data Topic
Info Type Options	How data is shaped: (i) ENCODED, (ii) DECODED and (iii) PROCESSED
Device Type Options	Type of device, e.g. LGT-92 or EM300-TH
Channel Options	Name of channel where data flows, e.g. *smartIrrigation* or *default*
Data Legitimacy Options	Is the data legitimate: (i) UNKNOWN, (ii) CORRECT, (iii) INCORRECT and (iv) UNDETERMINED
Records Options	Does the data contains **Records/Metadata**[1]
Air Humidity Data Options	Does the data contains information about Air Humidity[12]
Air Pressure Data Options	Does the data contains information about Air Pressure[12]
Air Quality Data Options	Does the data contains information about Air Quality[12]
Battery Data Options	Does the data contains information about the device Battery[12]
CO2 Data Options	Does the data contains information about CO2 levels[12]
CO Data Options	Does the data contains information about CO levels[12]
Distance Data Options	Does the data contains information about distances to a surface[12]
GPS Data Options	Does the data contains information about the device GPS coordinates[12]
Illuminance Data Options	Does the data contains information about illuminance in the environment[12]
Motion Data Options	Does the data contains information about the device motion[12]
NH3 Data Options	Does the data contains information about NH3 levels[12]
NO2 Data Options	Does the data contains information about NO2 levels[12]
O3 Data Options	Does the data contains information about O3 levels[12]

Table 4.3 continued from previous page

Topic *Routing Key*	**Description**
Occupation Data Options	Does the data contains information about occupation levels[1][2]
pH Data Options	Does the data contains information about ph level[1][2]
PM2.5 Data Options	Does the data contains information about pm 2.5 concentration[1][2]
PM10 Data Options	Does the data contains information about pm 10 concentration[1][2]
Soil Conductivity Data Options	Does the data contains information about the soil conductivity[1][2]
Soil Moisture Data Options	Does the data contains information about the soil moisture[1][2]
Temperature Data Options	Does the data contains information about the temperature[1][2]
Trigger Data Options	Does the data contains information about something that works as a switch[1][2]
Velocity Data Options	Does the data contains information about the device velocity[1][2]
VOC Data Options	Does the data contains information about VOC concentration[1][2]
Water Pressure Data Options	Does the data contains information about water pressure[1][2]
Command	Routing Keys that belong to the Command Topic
Command Type Options	Type of command, e.g. Open Valve
Alert	Routing Keys that belong to the Alert Topic
Alert Category Options	Category of the alert published, e.g. Fire Detention
Alert Subcategory Options	Category of the alert published, e.g. Humidity With High Rate Of Change
Alert Severity Options	Severity of the alert published, from *Information* level to *Critical* level

[1]has three possible values: (i) UNDETERMINED, (ii) WITH, (iii) WITHOUT
[2]related to the explored Data Types

4.5 Synopsis

This chapter presented the design of the platform, **Sensae Console**, and the solutions, **External Services**. Topics such as the domain, the architectural design and alternatives have been discussed here. To complement the description of the system, the next chapter introduces how, following the design proposed, this whole solution was implemented.

Chapter 5

Implementation

This chapter addresses the implementation of the design detailed before. First, the technical decisions will be presented, followed by a technical view of the software developed. The next section explains how the software was tested by displaying some code examples. Finally, a brief synopsis closes this chapter.

5.1 Technical Decisions

This section describes and justifies the decisions taken while developing **Sensae Console**. As a greenfield project, **Sensae Console** lacks constraints imposed by prior work. As such, all decisions have been taken during the thesis time span.

The following list unveils the most relevant technical decisions for **Sensae Console**:

- Backend Technologies Usage throughout the Solution;
- Frontend Technologies Usage through the Solution;
- Backend Services Expose a GraphQL API;
- Usage of RabbitMQ to support Internal Communication;
- Usage of Protocol Buffers in Internal Communication
- Database Usage throughout the Solution;
- Rules Script Engine;
- Data Decoders Script Engine;
- Usage of Github Actions for CI/CD;

The Containerization and Orchestration of services via Docker and Docker Compose, along the usage of Nginx to serve the UI and API, are discussed in Appendix K.

5.1.1 Backend Technologies Usage throughout the Solution

The backend development is divided into three main areas:

- *iot-core* package;
- Data Flow Scope backend containers;
- Service and Configuration Scope backend containers (named General Backend Services).

In the following sub sections a brief description and justification of the technologies used is presented.

5.1.1.1 Programming Language Used

A package named *iot-core*, an idealized SDK for **Sensae Console**, was developed to define the information that flows inside the system and is served to External Services. The *iot-core* package was developed in *Java*.

In the future, more programming languages may be supported though new SDKs. The *Rust* programming language is the next candidate due to its low memory footprint, fast startup times and expressive syntax.

The reasons that lead to the development of it in *Java* are:

- It's the programming language that the author is most familiarized with;
- It is widely used in industry for backend service development;
- Vast and robust support for virtually any technology used for backend development: database access, synchronous and asynchronous communication protocols, streaming platforms, embedded caches, rule engines and script engines.

The development of *iot-core* in *Java* lead to the development of all backend services also in *Java*.

5.1.1.2 General Backend Services

The services that this section encompasses can be seen as more robust and heavy due to their associated requirements.

As such, the framework used to develop them was *Spring Boot*, due to its vast documentation and big community. This framework comes with several modules that help to easily create stand-alone, production-grade applications. The author also had previously worked with this framework.

The main drawbacks of this framework are the slow start up time and high memory consumption, since these are not ideal for the microservices/cloud world (Spring 2022).

5.1.1.3 Data Flow Scope Backend Services

As discussed in Section 4.1.2, the services that this section encompasses can be seen as more lightweight than the ones described above due to their associated requirements.

Since these containers process inbound device data, they have a bigger need to automatically scale. Since they need to react faster to throughput changes, their start up times must be small.

As such, the framework used to develop them was *Quarkus*. This framework has first-class support for *GraalVM* and a reactive execution model that allow for higher concurrency, smaller memory footprint, and improved deployment density (redhat 2021) .

According to Oracle 2022b, GraalVM is a "high-performance JDK designed to accelerate the execution of applications written in Java and other JVM languages while also providing runtimes for JavaScript, Python, and a number of other popular languages. GraalVM

offers two ways to run Java applications: on the HotSpot JVM with Graal just-in-time (JIT) compiler or as an ahead-of-time (AOT) compiled native executable. GraalVM's polyglot capabilities make it possible to mix multiple programming languages in a single application while eliminating foreign language call costs."

This features, coupled with the fact that the *Quarkus* architecture follows the *The Reactive Manifesto*, are appealing when compared with *Spring Boot* that only has experimental support for *GraalVM*, via *Spring Native*.

5.1.2 Frontend Technologies Usage through the Solution

Even though a micro frontend architecture empowers the selection of different technologies depending on the requirements of the solution and team affinity with the stack, the Frontend Containers were developed using the same technological stack. At the time of writing there was only one developer involved, this diminished the cognitive load needed to work on the solution while still allowing future collaborators to use different frontend frameworks.

5.1.2.1 Programming Language and Framework Used

The author had previous contact with the following frameworks: (i) *Angular*, (ii) *React*, and therefore no other tool was discussed when choosing the one to use in the solution.

The programming language used was *Typescript* since it is a strongly typed language and therefore leads to more robust and predictable code. Static typing helps to avoid various bugs that arise when using *Javascript*. Before transpiling *Typescript* code to *Javascript*, it is analyzed to detect bugs related to type errors.

As for the framework/library used, the following table, Table 5.1, describes the reason that lead the author to choose Angular over React.

Table 5.1: Technologies Comparison - Angular vs React

Framework/Library	**Angular**	**React**
Separation of User Interface and Business Logic	enforced	flexible
Language Requirements	typescript	javascript or typescript
Familiarity with the tool	high	medium
UI Component Libraries with wide community support	material	ant design, material ui, react bootstrap, semantic ui react

Both tools have a wide support from the community and excellent documentation. For the author, Angular outclasses React in this project since it enforces the use of good design principles via the first and second entry described in Table 5.1.

5.1.2.2 Technologies used to create a Micro Frontend Architecture

Module Federation was the tool used to seemly connect the various frontends. No other tool was considered or researched since *Angular* already relies on *Webpack 5* to bundle the

application and therefore it's effortless to use this tool. *Module Federation* allows programs to reference other programs parts that are not known at compile time. In addition, the micro frontends can share libraries with each other, so that the individual bundles do not contain any duplicates.

5.1.2.3 Technologies used to build and manage the Frontend Services

This section describes how the various frontends are built and share common pieces of code. Angular comes with a tool to build and manage project but it was deemed too minimal for this project. Instead, the tool used was *Nx*. *Nx* describes it self as a "Smart, Fast and Extensible Build System", the "Next generation build system with first class monorepo support and powerful integrations".

This tool provides features needed to manage multiple frontends in a single repository, without dealing with libraries versions mismatch.

This tool has two main concepts that are widely used in the solution's frontend: apps and libraries. Apps focus on the UI and libraries on everything else, such as the domain or the interactions with backend services. The diagram presented at Figure C.6 resembles these two concepts.

5.1.2.4 Technologies used to provide map/location base services

This section briefly describes the library used to render and work with maps.

The two options in regards to this requirement were: (i) *Google Maps* and (ii) *Mapbox GL JS*.

The author picked *Mapbox GL JS* due to better documentation, a more stable API, and a much suitable pricing plan for small businesses, when compared to *Google Maps*.

This library can render custom maps and is bundled with powerful data visualization tools with a simple to use API, two features deemed important for the solution.

5.1.3 Backend Services Expose a GraphQL API

The API discussed in this section refers to the interfaces exposed to the outside world by backend containers of the Configuration and External Services Scopes and isn't related to the internal communication or device data ingestion interface exposed by the **Data Relayer** Container.

The two approaches considered were: (i) *Rest API* and (ii) *GraphQL*.

According to Facebook 2022b, "GraphQL provides a complete and understandable description of the data in your API, gives clients the power to ask for exactly what they need and nothing more, makes it easier to evolve APIs over time, and enables powerful developer tools."

According to IBM 2021b, "REST APIs provide a flexible, lightweight way to integrate applications, and have emerged as the most common method for connecting components in microservices architectures."

These two approaches have vast differences but they both try to answer the same question: How should one expose internal data to the outside world?

Eizinger 2017, compares these two approaches under seven criteria: (i) operation reusability, (ii) discoverability, (iii) component responsibility, (iv) simplicity, (v) performance, (vi) interaction visibility and (vii) customizability.

GraphQL was the chosen approach mainly due to better operation reusability: "The flexibility in the definition of the exactly returned data allows clients to tailor it for their specific needs, thereby achieving highly reusable data retrieval operations." and interaction visibility: "With GraphQL featuring a declarative language, intermediaries capable of understanding the GraphQL grammar can at least partly reason about the communication between a client and a GraphQL server."

Eizinger 2017, when discussing the complexity of each approaches also highlights that "GraphQL makes fetching data in various ways really simple for the client."

The idea behind the highly decoupled architecture of this solution derives from the need to provide knowledgeable customers with the tools to easily design and incorporate their solutions in **Sensae Console**. The usage of *GraphQL* further complements this idea by providing an API that is simple to understand and consume.

5.1.4 Usage of RabbitMQ to support Internal Communication

As discussed in Sections 4.3.3 and 4.3.4, the technology needed for this solution had to act as a message broker. It should enable the system to follow a push approach and ease the complex routing model envisioned.

The technology chosen for internal communication was *RabbitMQ*. This message broker was chosen in detriment of others since the author had previously worked with the technology and, according to Dobbelaere and Esmaili 2017 "RabbitMQ would be a good choice for realtime processing, based on the complex filtering the broker could provide".

Dobbelaere and Esmaili 2017 also mentions that RabbitMQ would be best-suited to be the Underlying Layer for IoT Applications Platform.

Under the AMQP 0.9.1 protocol, the exchange that better fits the defined requirements is the Topic Exchange, due to the possibility to subscribe to specific messages depending on their routing keys (as mentioned in Section 2.1.1.1.1).

When working with this protocol and type of exchange, some drawbacks were found:

When dealing with Topic Exchanges a Consumer can only subscribe to one specific routing key or all at once - via the *'*'* keyword - this makes it complex to create routing keys with dynamic values. As an example, lets look at the *Channel* routing key defined in Table 4.3 of Section 4.4.2.3. This key defines the single destination of a data unit. For a data unit to have various dynamic service destinations there would be a need to either:

- Ensure that every single service subscribes to all relevant combinations of *channel*s possible, deemed impractical;
- Duplicate data units, where each copy would be assigned a different channel, deemed inefficient.

To tackle this issue, another Message Broker, such as *Pulsar*, with its own protocol, can be used in the future. This Message Broker answers the drawback describe above by allowing Consumers to subscribe to multiple topics (equivalent to RabbitMQ' routing keys) on the basis of a regular expression (regex) (*Pulsar - Multi-topic subscriptions*).

The other drawback found is that, according to the *Advanced Message Queuing Protocol Specification, Version 0-9-1* the routing keys have a max size of 255 bytes. As described in Table 4.3 of Section 4.4.2.3, the system currently supports various keys and more keys are expected to be added in the future, meaning that this cap may one day be reached. This limitation was tackled by mapping each routing key to a single character when possible. As an example, the routing key *Info Type Options* in Table 4.3 has three possible values: Encoded, Decoded and Processed, theses values are respectively represented in the system as *e*, *d* and *p*.

5.1.5 Usage of Protocol Buffers in Internal Communication

This section refers to how messages that flow in the system (via Message Broker) are serialized and deserialized. The common formats used to send structured data across systems are JavaScript Object Notation (JSON) and Extensible Markup Language (XML). This formats sacrifice size and de/serialization performance for human readability as stated by Sumaray and Makki 2012.

As mentioned before, **Sensae Console** aims to provide a good developer experience for external costumers that want to expand the solution according to their needs. Due to this, the final decision weighted heavily on formats that were self-documented, e.g. defined by a strict *data schema*, such as *Protocol Buffers* and "Thrift".

These two technologies, *Protocol Buffers* and "Thrift", have similar goals and approaches to the problem they try to solve. They both rely on code generation based on a schema of the data structure. The tools related to this formats officially support various languages such as *Java*, *C++*, *C#*, *Python*, *Go* and others.

By leveraging these features, creating a basic SDK in a new programming language is trivial since serialization, deserialization and data structure is already taken care by the code generation tool.

Protocol Buffers are a "language-neutral, platform-neutral, extensible mechanism for serializing structured data" (Google 2022f).

Thrift's "primary goal is to enable efficient and reliable communication across programming languages by abstracting the portions of each language that tend to require the most customization into a common library that is implemented in each language" (Slee, Agarwal, and Kwiatkowski 2007).

Ultimately *Protocol Buffers* were chosen due to better documentation and community support.

5.1.6 Database Usage throughout the Solution

This section refers to how information is stored across the system and follows the terms mentioned in Section 2.1.1.3.

The requirements gathered unveil the need to use three different database data models throughout the system: (i) relational, (ii) document-based and (iii) column-based data models. The following sections answer why these data models were needed and what technologies were chosen for each of them. A final section unveils an optional solution that was considered but ultimately not pursued.

Appendix L presents details related to how these databases are configured.

5.1.6.1 Relational Database Usage

This type of data, strict and with well-defined relations, can be found on most Bounded Concerns described in Appendixes G (related to Section 4.2.2) and H (related to Section 4.2.3).

As such, this data model was adopted for the **Device Management Database**, **Data Decoder Database**, **Data Processor Database**, **Rule Management Database**, **Identity Management Database**, **Smart Irrigation Business Database** and **Notification Management Database** containers. The decision was based on the discussion in Section 2.1.1.3.1.

The decision to use *PostgresSQL* was taken based on the fact that, contrary to the other options, *PostgresSQL* supports a vast number of Data Types such as JSON, Arrays, Universally unique identifier (UUID), and Ranges. *PostgresSQL*'s data model is an extension of the relation data model, named object-relational data model - Elmasri et al. 2000. This data model supports various concepts such as objects, classes and inheritance and therefore can lead to entity models more expressive and close to the business ideas.

5.1.6.2 Document-based Database Usage

The type of features of this data model resembles the requirements and data stored by the Data Store container described in Section 4.1.2 and Figure 4.12. This container, intended to mimic a Data Lake[1], stores any type of data for future use.

As such, this data model was adopted for the **Data Store Database** container. The decision was based on the discussion in Section 2.1.1.3.2.

The only technology considered, and therefore adopted, was *MongoDB* due to its vast community, excellent documentation, open-source license and large number of libraries that ease the database management operations. *MongoDB* also supports replication and sharding. According to Elmasri et al. 2000, these features are useful once a single node isn't capable of withstanding all data collected while providing fast access to it.

5.1.6.3 Column-based Database Usage

The features mentioned in Section 2.1.1.3.3 fit the requirements related to storing and reading vast amounts of device measures. As such, this database type was adopted for the **Fleet Management Database** and **Smart Irrigation Data Database** containers.

The type of business this solution tackles revolves around the capture and analysis of device readings. So the notion of time has to be treated as a first class citizen. The measurements that constitute a time series are ordered on a timeline, which reveals information about underlying patterns.

As stated by Naqvi, Yfantidou, and Zimányi 2017, TSDB "can be used to efficiently store sensors and devices' data" since, "such technologies are generating large amount of data which is usually time-stamped".

[1] Massively scalable storage repository that holds a vast amount of raw data in its native format (ńas isż) until it is needed, by Miloslavskaya and Tolstoy 2016

With this requirements in hand, a column-based data model isn't enough. The technology adopted should also natively support time series to ease data analysis. As such, the *HBase* and *CassandraDB* options were discarded.

Between the two missing options, the author picked *QuestDB* due to better support for SQL though Java Database Connectivity (JDBC). During the research of this two technologies no major downside was found for *QuestDB* when compared to *InfluxDB*.

The author had no previous contact with this type of data model.

5.1.6.4 Graph-based Database Usage

Even thought this data model was ultimately not used, the author deemed relevant to analyze it.

As stated in the bounded concern's section of Identity Management (Appendix G), the domains follow a hierarchical structure that can resemble a graph. This concern in particular would benefit from a graph-based database, but this option was not pursued since the author had no previous contact with this family of technologies. Instead *PostgresSQL* was used.

PostgresSQL can represent logical hierarchical structures and concepts using the array data type as the *path* from the root domain to the current domain.

Queries that revolve around graph concepts such as: select parent node, select child nodes, move nodes to a new parent and others, can be preformed efficiently using array operators such as **&&**, **||** and **@>**[2].

5.1.7 Rules Script Engine

This section refers to the bounded concern's section of Rule Management (Appendix G). As mentioned before, the purpose of this concern is to provide a high-level language that can analyze a stream of Data Units and output alerts base on them. The technology adopted was *Drools*.

Drools is a rule engine widely used in the industry (RedHat n.d.). The features that stud out from other rule engines mentioned in 2.1.1.4 were:

- Open-source license;
- Support for sliding windows of time;
- It is also a CEP System;
- Integrates with the *iot-core* package since it is also written in *Java*;
- Can be used as a standalone application or an embedded component of another application;
- Has an expressive, yet complex, syntax to write rules;
- Can dynamically load rules at runtime.

The Section 5.2.5 details how one can write rule scenarios.

[2]taken from PostgresSQL Documentation: *Array Functions and Operators & Array Functions and Operators*

5.1.8 Data Decoders Script Engine

This section refers to the bounded concern of **Data Decoder**. As mentioned in Section G.2, the purpose of this concern is to translate inbound Data Units into a format and semantics that the system can understand. The technology adopted was *Javascript*.

Javascript is a high level language with an enormous community and is widely used in the industry. Another big reason behind this decision is that a lot of companies producing IoT devices provide open-source decoders written in *Javascript*, such as Milesight [3], SensationalSystems [4] and Helium, [5]. This makes it easy and straightforward to integrate new decoders in **Sensae Console**.

The Section 5.2.6 details how one can write decoders.

5.1.9 Usage of Github Actions for CI/CD

Since the code is hosted in *Github*, it was decided to leverage the CI/CD features of the platform. *Github Actions* purpose is to automate software workflows via CI/CD.

According to RedHat 2022, the term CI/CD represents a method to delivering applications to clients by introducing automation into the development states. It is divided into three concepts:

- **Continuous Integration**: new versions of the project are regularly submitted, tested and merged into the current project;
- **Continuous Delivery**: new versions of the project are automatically archived in a repository where they can then be deployed to a production environment;
- **Continuous Deployment**: new versions of the project are automatically deployed to a production environment.

The *iot-core* package is archived in a repository so that it can then be integrated in the backend containers of **Sensae Console**, and possibly in other projects. To do so, the team uses *Github Actions*. This tool's behavior is defined in a YAML file, presented in the Code Sample 5.1.

```yaml
name: IoT Core - Continuous Delivery to maven central
on:
  push:
    tags:
      - '**'
      - '*'
jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v2
      - name: Set up Maven Central Repository
        uses: actions/setup-java@v1
        with:
          java-version: 17
          server-id: ossrh
```

[3] github.com/Milesight-IoT/SensorDecoders
[4] github.com/SensationalSystems
[5] github.com/helium/console-decoders

```
17            server-username: MAVEN_USERNAME
18            server-password: MAVEN_PASSWORD
19            gpg-private-key: ${{ secrets.MAVEN_GPG_PRIVATE_KEY }}
20            gpg-passphrase: MAVEN_GPG_PASSPHRASE
21        - name: Deploy with Maven
22          run: mvn -B clean deploy -Pci-cd
23          env:
24            MAVEN_USERNAME: ${{ secrets.OSSRH_USERNAME }}
25            MAVEN_PASSWORD: ${{ secrets.OSSRH_TOKEN }}
26            MAVEN_GPG_PASSPHRASE: ${{ secrets.MAVEN_GPG_PASSPHRASE }}
```

Listing 5.1: Configuration File for *iot-core* Continuous Delivery

As we can see in lines **2** to **6**, this action is triggered every time a new git tag is pushed to the repository. This action then proceeds to download and setup java and maven - lines **12** to **20**. Finally it runs a maven command to deploy the new version to the artifact repository - lines **21** to **26**.

The **Sensae Console** has an action to deal with Continuous Integration - Code Sample 5.2, where changes made to the software are tested.

```
1  name: Sensae Console - Continuous Integration - Test changes
2  on:
3    push:
4      branches:
5        - master
6        - dev
7  jobs:
8    test:
9      runs-on: ubuntu-latest
10     steps:
11       - uses: actions/checkout@v3
12       - name: Set up JDK 17
13         uses: actions/setup-java@v3
14         with:
15           java-version: "17"
16           distribution: "adopt"
17       - name: Set up Node 16
18         uses: actions/setup-node@v3
19         with:
20           node-version: 16
21       - name: Test Suite
22         run: |
23           ./project/scripts/run-tests.sh "${{ secrets.mapbox_token }}" "
     ↪ ${{ secrets.microsoft_audience }}" "${{ secrets.google_audience
     ↪ }}" "${{ secrets.admin_email }}"
```

Listing 5.2: Configuration File for Sensae Console Continuous Integration

As we can see in lines **2** to **6**, this action is triggered every time a new commit is push to the *dev* and *master* branches. This action then proceeds to download and setup java and maven - lines **10** to **16**, and then node and npm - lines **17** to **20**. Finally it runs a script that tests the solution - line **23**. The script requires the displayed secrets to run some tests that will be discussed in the Testing Section.

The mentioned script has the following structure - Code Sample 5.3.

```
1  #!/bin/bash
```

```
2  set -eo pipefail
3
4  ROOT_DIR=$(git rev-parse --show-toplevel)
5
6  cd "$ROOT_DIR"/project || exit
7
8  ./scripts/generate-test-config.sh "$@"
9
10 docker-compose -f docker-compose.build.yml build
11
12 rm --f -- reports/backend-test-pass.log
13 rm --f -- reports/backend-test-fail.log
14
15 cd backend-services || exit
16
17 ls -l data-relayer | xargs -I % sh -c 'cd % && mvn test && \
18     echo % >> ../../reports/backend-test-pass.log || \
19     echo % >> ../../reports/backend-test-fail.log'
20
21 test ! -f ../reports/backend-test-fail.log
22
23 cd ../frontend-services || exit
24
25 npm install
26 npm run test-all
27
28 ./../scripts/build-images.sh
29
30 docker-compose -f ../docker-compose.test.yml up -d --build
31
32 sleep 60
33
34 npm run e2e-all
35
36 docker-compose -f ../docker-compose.test.yml down
```

Listing 5.3: Sensae Console Test Suite Script

This script first intent is to defined a basic environment where tests can be run. The flag *set -eo pipefail* ensures that if any command fails the script will terminate and exit with an error. It runs the following steps:

- Generate configurations - line **8** - to run every test according to the secrets provided by the github action presented at Listing 5.2,
- Build the database containers - line **10**. The file *docker-compose.build.yml* references all the solution's databases that need a custom build due to their predefined schema;
- Run the command *mvn test* for all backend containers and store the results of each container's test in a file - lines **17** to **19**;
- Checks if any container didn't pass the tests - line **15**;
- Run tests related to the frontend at lines **23** to **26**. The script mentioned as *test-all* is: *nx run-many –all –target=test*. This script runs all unit tests of both frontend libraries and apps using *Nx*, as mentioned in 5.1.2.3 Section;
- Build and start an environment similar to the production one - lines **28** to **32**;

- Preform end to end tests against the test environment - **34**. The script mentioned as *e2e-all* is: *nx run-many –all –target=e2e –parallel=1*. This script runs all end-to-end tests of the frontend apps using *Nx*, as mentioned in 5.1.2.3 Section;
- Shutdown the test environment.

5.2 Technical Description

This section guides the reader through **Sensae Console** and **External Services** with a technical description of the various elements that are exposed to the final costumers and platform managers.

It describes the following topics:

- Sensae Console UI;
- Sensae Console Custom Maps;
- Sensae Console Backend API;
- Sensae Console Data Ingestion Endpoint;
- Sensae Console Rule Engine;
- Sensae Console Data Decoders;
- Solutions - External Services;
- Sensae Console Device Integration.

5.2.1 Sensae Console UI

In this subsection the UI is presented.

The Figure 5.1 represents the main layout for any user. It is comprised of a toolbar with a section for **Service Pages**, another for **Configuration Pages** and a final one for authentication purposes.

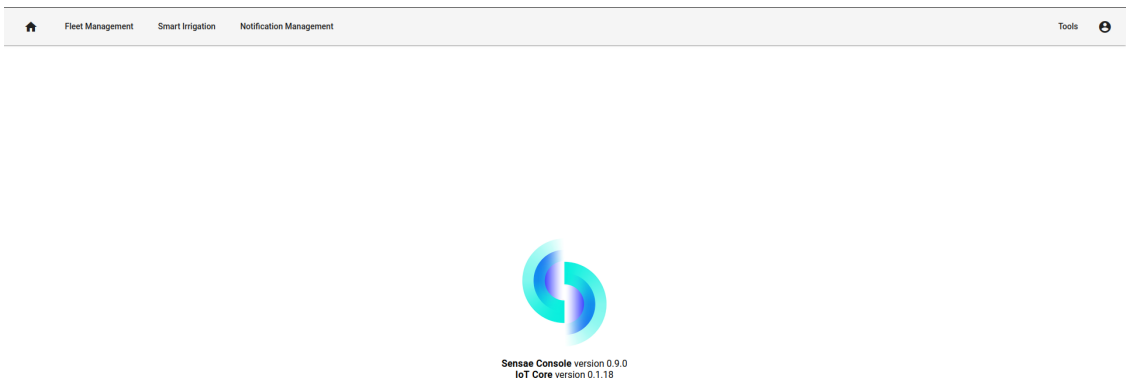


Figure 5.1: Sensae Console Home Page

From this page, if the user has sufficient permissions, he/she can access configuration pages, as an example the **Device Management Page** is displayed in Figure 5.2.

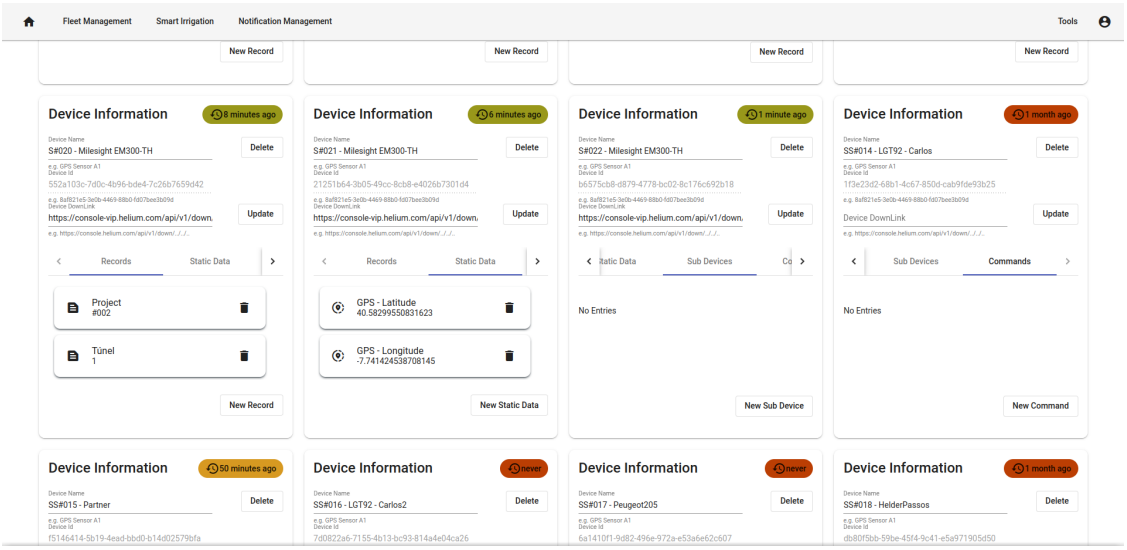


Figure 5.2: Sensae Console Device Management Page

In this page the user can see when was the last time the device interacted with the platform, create and delete devices and edit the details of each device according to the model presented in SectionDevice Management of the *Bounded Contexts*.

From the home page, if the UI Aggregator was configured to fetch external services, one can access those services' pages too, as an example the **Smart Irrigation Page** is displayed in Figure 5.3. This page presents a map where the user can see, search and create irrigation zones. Device measures are updated in real time via *Websocket*s. The user can also see the irrigation zone details after clicking on it. From there it's possible to open/close valves an see the history of measures of each device.

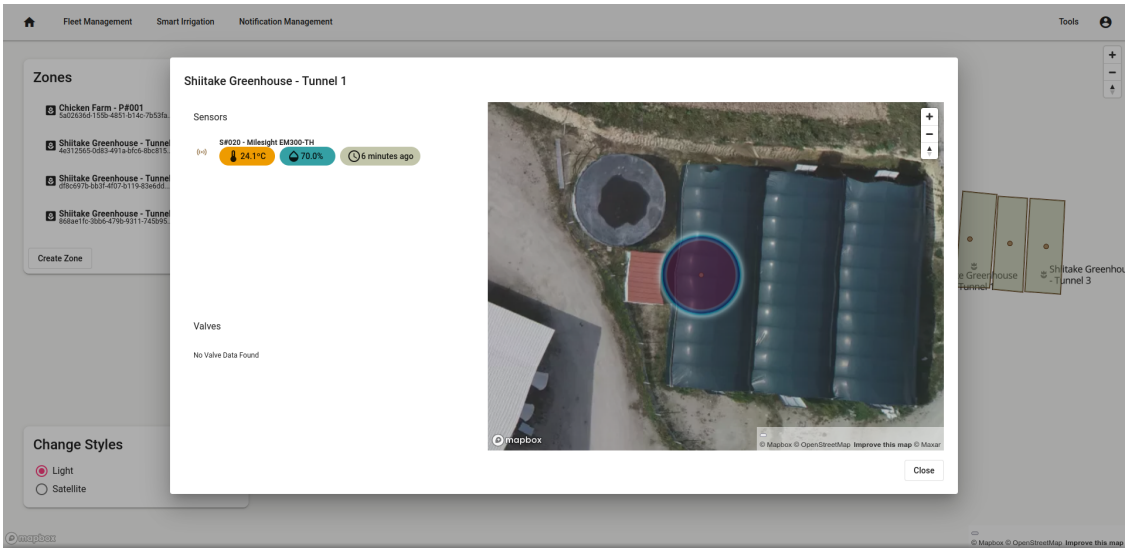


Figure 5.3: External Services - Smart Irrigation Page

Other relevant pages are presented in the Appendix I, for Sensae Console, and Appendix J, for the Solutions developed.

5.2.2 Sensae Console Custom Maps

This section describes how custom maps where built to fit the solution needs. Some costumers facilities were not present in the satellite view of *Google Maps* or *Mapbox GL JS*. A custom map, with the missing facilities, was built using satellite images taken with a drone. The images where processed with *ArcGIS* and transformed in *.tiff* files that could be incorporated in the basic satellite layer of *Mapbox GL JS*.

The following image, Figure 5.4 presents the new map with the costumer facilities in a greener tone than the rest of the map. This map was used to display three greenhouses and a chicken farm that belong to a costumer. This map is currently in use by the Smart Irrigation Service.

Figure 5.4: External Services - Smart Irrigation Page - Custom Map

The road trajectory mismatch present in the map could be reduced by taking pictures from more angles. *ArcGIS* would create a better model with a wider pool of information.

5.2.3 Sensae Console Backend API

The **Sensae Console** and **External Services** APIs are served as a *GraphQL* API, one for each configuration/service concern. These APIs are described with a schema.

As an example the Smart Irrigation API is presented in the Code Sample 5.4.

```
type Subscription {
    data(filters: LiveDataFilter, Authorization: String) : SensorData
}

type Query {
    history(filters: HistoryQueryFilters) : [SensorDataHistory]
```

```
7       fetchIrrigationZones : [IrrigationZone]
8       fetchLatestData(filters: LatestDataQueryFilters): [SensorData]
9   }
10
11  type Mutation {
12      createIrrigationZone(instructions: CreateIrrigationZoneCommand) :
        IrrigationZone
13      updateIrrigationZone(instructions: UpdateIrrigationZoneCommand) :
        IrrigationZone
14      deleteIrrigationZone(instructions: DeleteIrrigationZoneCommand) :
        IrrigationZone
15      switchValve(instructions: ValvesToSwitch): Boolean
16  }
```

Listing 5.4: Smart Irrigation API Schema

From the observation of the code sample one can see that:

- The *data* function serves new *SensorData* in real-time according to the filters provided in the *filters* parameter;
- The *data* function uses *Websocket* to operate as a full duplex communication channel. This spec, contrary to the HTTP spec does not account for HTTP Headers, as such the JSON Web Token (JWT) that provides the user authentication details has to be sent as a normal parameter and not as an Authorization HTTP Header;
- There are three query type functions. One to fetch the history regarding Irrigation Zones or Devices over a time span. One to fetch the Irrigation Zones. And the last one to fetch the latest data of each device;
- There are four mutations, each corresponding to the use cases referenced in Section 3.1.3.3.

5.2.4 Sensae Console Data Ingestion Endpoint

The Data Ingestion Endpoint refers to how device data is sent to **Sensae Console**.

The endpoint corresponds to an HTTP POST verb with the following Uniform Resource Locator (URL) schema:

`https://<ip>:<port>/sensor-data/{channel}/{infoType}/{deviceType}`

The endpoint collects the request body and then forwards it with the appropriate routing keys.

The routing keys are created according to Table 4.3. The *infoType* can have two values: ENCODED or DECODED. Depending on this value the message is routed to *Data Decoder Flow* or *Data Processor Flow* as described in Figure 4.12.

The *channel* parameter indicates the final service that it is destine to: *fleet* for Fleet Management Service or *irrigation* for Smart Irrigation Service.

Finally, to ensure that the requests to this endpoint are trustworthy, a secret has to be sent in the Authorization HTTP Header. This secret is defined as a configuration of the **Sensae Console**, discussed in Section K.7.

5.2.5 Sensae Console Rule Engine

The rule engine can be accessed from the **Rule Management Page** of the UI and, as stated in Rule Management Bounded Concern (Appendix G), it provides a high-level language that can be used to detect anomalies in **Data Unit**s and turn them into **Alert**s.

Valid **Data Unit**s are captured by **Alert dispatcher Backend** and then inserted in the Rule Engine.

As stated in Rules Script Engine, the rule engine used was *Drools*. To write rules for **Sensae Console** one must follow several guidelines.

A *Drools* rule is composed by conditions, actions and facts.

Facts are inserted in the rule engine. If a fact or group of facts match a condition (*when* section), an action is triggered (*then* section).

The rule engine, is tailored to managers or developers and not for final clients since it can be hard to create meaningfully rules without side effects.

To clarify the guidelines the following Code Samples 5.5, 5.6 and 5.7 are presented.

The first Code Sample presents the beginning of the rule scenario, where imports and new Facts are created.

```
1  package rules.project.two;
2  //imports (hidden for brevity)
3  global pt.sharespot.iot.core.alert.model.AlertDispatcherService
       ↪ dispatcher;
4
5  declare StoveSensor
6      @role( event )
7      deviceId : UUID
8  end
9  declare StoveSensorData
10     @role( event )
11     deviceId : UUID
12     dataId : UUID
13     temperature : Float
14     humidity : Float
15 end
```

Listing 5.5: Rule Scenario Example - Part 1

As we can see, at line **3** the interface that defines how an alert can be sent is imported for later use. From line **5** to **15** two facts are declared, this can later be used as simple *Java* POJOs. A fact defined with the *event* role means that it occurred at a specific time (upon creation) and can be used for CEP.

The following code sample presents a simple rule to store *StoveSensorData* facts in the working memory of *Drools*.

```
1  rule "Collect stove sensor data that belongs to Project #002"
2      when
3          $d : DataUnitDTO(
4               getSensorData()
5               .hasProperty(PropertyName.AIR_HUMIDITY_RELATIVE_PERCENTAGE),
6               getSensorData()
7               .hasProperty(PropertyName.TEMPERATURE)
```

```
8           )
9           exists DeviceRecordEntryDTO(
10              label == "Project" && content == "#002"
11          ) from $d.device.records
12          not(StoveSensorData(dataId == $d.dataId))
13      then
14          StoveSensorData reading = new StoveSensorData();
15          reading.setDeviceId($d.device.id);
16          reading.setDataId($d.dataId);
17          reading.setTemperature($d.getSensorData().temperature.celsius);
18          reading.setHumidity($d.getSensorData().airHumidity
    ↪ .relativePercentage);
19          insert(reading)
20  end
```

Listing 5.6: Rule Scenario Example - Part 2

As we can see this rule is composed by two sections, the *when* and *then* sections. In the *when* the following conditions are defined:

- The captured DataUnitDTO has AIR HUMIDITY RELATIVE PERCENTAGE and TEMPERATURE measures - lines **3** to **8**;
- The capture DataUnitDTO has a record with a "Project" label and "#002" content - lines **9** to **11**;
- The DataUnitDTO is not a duplicate fact in the working memory - line **12**.

Once this conditions are meet a *StoveSensorData* is created with all the needed information and then inserted into the working memory - lines **14** to **19**.

The following code sample presents a simple rule to dispatch an **Alert** after some conditions are meet.

```
1   rule "Dispatch Stove Alarm - Dry Soil Scenario - Project #002"
2       when
3           $s : StoveSensorData(temperature > 26, humidity < 50)
4           not(StoveSensorData(this != $s,
5                   temperature < 26,
6                   humidity > 50,
7                   this after[0s,11m] $s)
8           )
9       then
10          dispatcher.publish(AlertBuilder.create()
11                              .setCategory("irrigation")
12                              .setSubCategory("drySoilDetected")
13                              .setDescription("Project #002 - Device "+
    ↪ $s.deviceId +" detected low humidity/high temperature")
14                              .setLevel(AlertLevel.ADVISORY)
15                              .setContext(CorrelationDataBuilder.create()
16                                  .setDeviceIds($s.deviceId)
17                                  .setOther("Project #002")
18                                  .build())
19                              .build());
20  end
```

Listing 5.7: Rule Scenario Example - Part 3

As we can see this rule matches when the same device reports measures of air humidity higher than 50% and temperature lower than 26 Celsius degrees for more than 11 minutes.

Once the rule is matched, an Alert is dispatched using the referenced dispatcher in Code Sample 5.5. The Alert can be created using the builder pattern.

An Alert closely resembles a Notification from the Notification Management Concern (Appendix H). It also has a category (line **13**), a sub category (line **14**), a severity level (line **16**), and a description (line **15**).

For an **Alert** to be sent at least the category and sub category parameters have to be set. By default the **INFORMATION** severity level is used.

In order for services to act upon a received **Alert**, it has to be associated with a *DeviceId* (this association helps services like **Smart Irrigation** to know what Valve must be turned on or off), a *DataId* or *Other*.

An **Alert** is later transformed and store as a Notification, the *DeviceId*s associated to it are used to determine what domains will have access to the Notification. If no *DeviceId*s are associated only the root domain will have access to it.

5.2.6 Sensae Console Data Decoders

As mentioned in the Data Decoder Concern Section (Appendix G), **Data decoder**'s purpose is to provide a flexible option to transform inbound data units into something that the system understands.

This happens when a **Data Unit** has a routing key with the ENCODED info type.

There are certain guidelines to follow in order to create a decoder:

- Has to be written in vanilla *javascript*;
- Has to have an *entry* function with the following signature *function convert(dataUnit)*;
- Can't import any node function, npm package or reference other scripts.

As an example, the Code Sample 5.8 presents the decoder for the device type EM500-TH[6].

```
1  const decodePayload = (payload, port) =>
2      ({"0": decoder(base64ToHex(payload), port)});
3
4  const base64ToHex = (() => {
5      //hidden for bevity
6  })();
7
8  function decoder(bytes, port) {
9      let decoded = {}, temperature = {}, airHumidity = {}, battery = {};
10     for (let i = 0; i < bytes.length;) {
11         let channel_id = bytes[i++];
12         let channel_type = bytes[i++];
13         if (channel_id === 0x01 && channel_type === 0x75) {
14             decoded.battery = battery;
15             battery.percentage = bytes[i];
16             i += 1;
17         } else if (channel_id === 0x03 && channel_type === 0x67) {
```

[6]Milesight EM300-TH Decoder

```
            decoded.temperature = temperature;
            temperature.celsius = readInt16LE(bytes.slice(i,i+2))/10;
            i += 2;
        } else if (channel_id === 0x04 && channel_type === 0x68) {
            decoded.airHumidity = airHumidity;
            airHumidity.relativePercentage = bytes[i] / 2;
            i += 1;
        } else {
            break;
        }
    }
    return decoded;
}
const readUInt16LE = bytes => (bytes[1] << 8) + bytes[0] & 0xffff;

function readInt16LE(bytes) {
    let ref = readUInt16LE(bytes);
    return ref > 0x7fff ? ref - 0x10000 : ref;
}
const convert = dataUnit => ({
    dataId: dataUnit.uuid,
    reportedAt: dataUnit.reported_at,
    device: {
        id: dataUnit.id,
        name: dataUnit.name,
        downlink: dataUnit.downlink_url,
    },
    measures: decodePayload(dataUnit.payload, dataUnit.port),
});
```

Listing 5.8: EM300-TH Data Decoder Example

As we can see, this code sample decodes an EM300-TH **Data Unit**. The function *convert* is the one mentioned in the guidelines, it assigns values such as *id*, *name*, *reported_at*, *downlink_url*, *uuid* to its correct place and calls the function *decodePayload* to gather the device measures. The *decodePayload* stores every measure in the *controller* key - value *0*. The function *base64ToHex* is the function that reads a Base 64 string and transforms it into a Hex Array - to reduce bandwidth the device normally encodes and sends data as a base 64 string. The function *decoder*, *readInt16LE* and *readUInt16LE* were adapted from the TTN decoder[7] of this device.

5.2.7 Solutions - External Services

This section discusses how external services interact with the Sensae API, this was briefly mentioned in Section 4.4.2.3.

In order to provide an easy to understand integration with the platform, the routing keys concept was introduced. The idea, from the point of view of someone developing a service, is to start by defining what type of information that service should capture.

The two types of information a service usually needs are: (i) **Data Unit**s and (ii) **Alert**s. Each of this information are defined by their routing keys as described in Table 4.3.

A service can also publish **Command**s to interact with actuators.

[7] Milesight EM300-TH Decoder

The following sub sections will detail each service information needs.

5.2.7.1 Fleet Management Service

The focus of this service was to simply collect the location of the costumers' fleet since the company was given no permission to access the vehicles' On-board diagnostics (OBD) System. Access to the OBD System would provide a deeper knowledge regarding the vehicles' conditions.

Therefore, this service captures information of a single type (and doesn't publish any Command):

Data Topic: *'processed'*, *'correct'*, with *'defined ownership'* and *'device information'* data unit with *'gps'* readings in the channel *'fleet'*.

At a high-level view, this service only requires Global Positioning System (GPS) data sent to the *'fleet'* channel.

5.2.7.2 Notification Management Service

The focus of this service was to simply showcase and dispatch the alerts sent by **Sensae Console**. A simple rule scenario for Indoor Fire Detention was then created based on tests performed on a costumer's facilities (Appendix O). The rule scenario was based on the following metrics: CO2 Particle per Million (PPM), Air humidity and Air Temperature.

This service captures information of a single type (and doesn't publish any Command):

Alert Topic: alerts with *'defined ownership'*.

At a high-level view, this service requires all alerts that already have a *'defined ownership'*.

It is divided in two backend containers (as described in Figure 4.25) that subscribe to the same information but handle it differently.

5.2.7.3 Smart Irrigation Service

The focus of this service was on the two most common types of agriculture mentioned in literature, outdoor and greenhouse, according to Garca et al. 2020.

The Greenhouse agriculture type collects two measures: (i) Air Temperature and (ii) Air Humidity, the Outdoor agriculture also collects two measures: (i) Luminosity and (ii) Soil Moisture. Both rely on a irrigation system that can be activated and deactivated remotely, like a simple switch, to regulate the monitured environment.

The measures to capture were chosen according to costumers' sugestions and Garca et al. 2020 that states that the Soil moisture, Air Temperature, Air Humidity and Luminosity parameters are the most common monitured parameters in papers that propose an irrigation system.

This service captures information of the given types:

- **Data Topic**: *'processed'*, *'correct'*, with *'defined ownership'* and *'device information'* data unit with *'gps'* and *'trigger'* readings in the channel *'irrigation'* (for valves);

- **Data Topic**: *'processed'*, *'correct'*, with *'defined ownership'* and *'device information'* data unit with *'gps'*, *'temperature'* and *'air humidity'* readings in the channel *'irrigation'* (for green house sensors);
- **Data Topic**: *'processed'*, *'correct'*, with *'defined ownership'* and *'device information'* data unit with *'gps'*, *'illuminance'* and *'soil moisture'* readings in the channel *'irrigation'* (for park sensors);
- **Alert Topic**: alerts with the category *'smartIrrigation'* and sub category *'drySoil'* (to open all valves in a garden);
- **Alert Topic**: alerts with *'defined ownership'*, the category *'smartIrrigation'* and sub category *'moistSoil'* (to close all valves in a garden);
- **Alert Topic**: alerts with *'defined ownership'*, the category *'smartIrrigation'* and sub category *'valveOpenForLengthyPeriod'* (to close that specific valve).

It then publishes Commands to close or open valves. The service can only issue a command if the Data Unit sent by the valve refers two commands, one to open and another to close the valve. This commands, usually defined in the **Device Management Page**, and mentioned in the Device Management Concern (Appendix G), need to have the *CommandId* value as *'openValve'* or *'closeValve'*.

At a high-level view, this service requires data from *Park* sensors, *Green Houses* and *Valves* that flow in the *'irrigation'* channel. It captures Alerts to decide when to open or close Valves by sending specific Commands.

5.2.8 Sensae Console Device Integration

This section describes how devices can be connected to **Sensae Console**. As stated in Section 3.2, the service that must be used to communicate with devices is *Helium Console*. This solution works with other platforms, such as Azure IoT Hub, since it provides an agnostic data ingestion endpoint as stated in Section 5.2.4.

Virtually any device can be integrated, via *Helium Console*, with **Sensae Console**. To do so, one needs to register new devices in *Helium Console*, for example via Over the Air Authentication (OTAA). Then create a Custom HTTP Integration, following the Section 5.2.4 instructions.

The Figure 5.5 presents an example of the custom integration for the EM300-TH Device.

Finally, in the *Helium Console* flows page, connect the registered device to the custom integration.

This method will require the user to register the endpoint with the *encoded* type and write a **Data Decoder** in **Sensae Console** to translate the payload sent by the device through *Helium Console*.

If the user intends to use the **Data Processor**, he/she needs to:

- Register the endpoint with the *decoded* type;
- Define a **Data Processor** in **Sensae Console** to map the payload sent by *Helium Console*;
- Write the decoder in *Helium Console* - in the *Function* page;

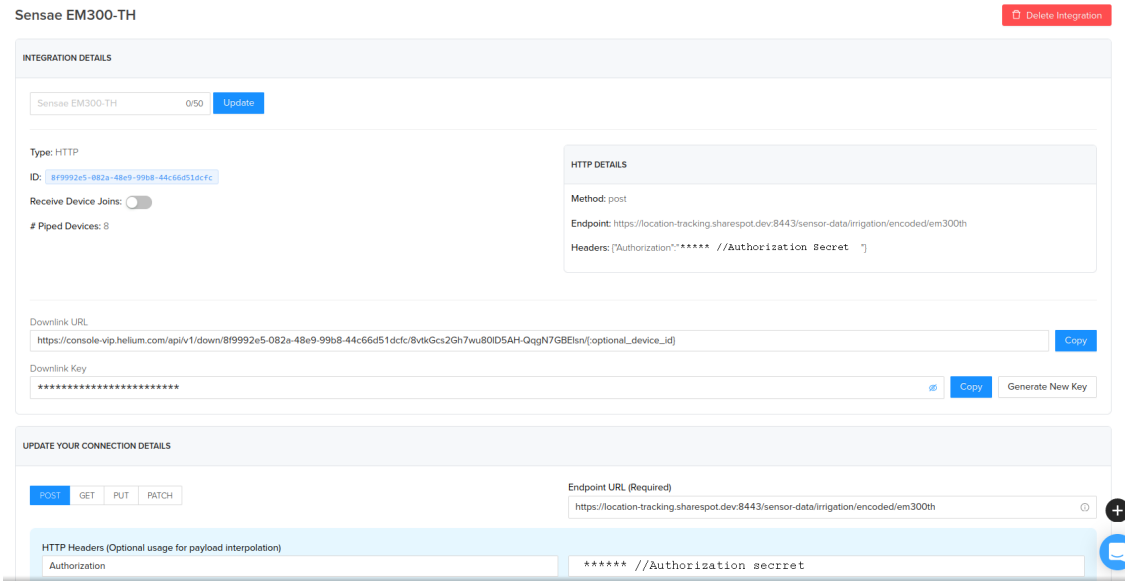


Figure 5.5: Helium - Custom Integration Page

- Link the device to the *Function* in *Helium Console*;
- Link the *Function* to the custom integration;

5.3 Testing

According to W. E. Lewis 2008: "Software testing is the activity of running a series of dynamic executions of software programs after the software source code has been developed." Tests have a fundamental role in the development of software, they validate the work done, prevent production bugs, regressions and improve code quality, according to Hughes 2017 and IBM 2022a.

According to Pittet 2022 there are seven categories of tests:

- **Unit Testing**: Capture the need to verify and validate the individual behavior of small pieces of the solution.
- **Integration Testing**: Capture the need to verify that different modules/components of the system work collectively as expected;
- **Functional Testing**: Capture the need to verify that business requirements are meet by the system;
- **End-to-End Testing**: Capture the need to verify that user interaction against common workflows works as expected in the system;
- **Acceptance Testing**: Capture the need to ensure that functional and non-functional requirements are accomplished;
- **Performance Testing**: Capture the need to verify how the environment behaves under heavy load. Their objective is to evaluate the stability, availability and reliability of the system;

- **Smoke Testing**: Capture the need to verify the overall state of the system before running heavier and extensive tests.

These categories complement each other to ensure the correct behavior of the system. Nevertheless, the Smoke and Acceptance Testing categories were not pursuit.

The smoke tests were replaced by common unit tests. The acceptance tests weren't required since, at the time of writing, the project had no clear and concise functional requirements that the platform could be tested against.

Architectural tests were added to the test suite to ensure that the C4 component level design discussed in AppendixC would always be respected.

The performance tests will be discussed in depth in the Evaluation Chapter.

In the following sections examples for each test category will be presented.

5.3.1 Unit Tests

This section focus on unit tests preformed throughout the solution.

The test presented in Listing 5.9 verifies that a value referenced via the path *'path[0].prop'* can be found and transferred to the path defined in the mentioned Property: *DEVICE_ID*. It uses the *JUnit5 Testing Framework*.

```
@Test
void ensureTransferWorksWithValidArrayPath() throws
    ↪ JsonProcessingException {
    var jsonNode = mapper.readTree("""
            {
                "path": [
                    {
                        "prop": "viva"
                    }
                ]
            }
            """);
    var objectNode = mapper.createObjectNode();

    new KnownPropertyTransformation(
        "path[0].prop", PropertyName.DEVICE_ID, 2)
            .transfer(jsonNode, objectNode);

    Assertions.assertEquals("viva",
     objectNode.get("device").get("id").asText());
}
```

Listing 5.9: Unit Test Example in *iot-core* package

The test presented in Listing 5.10 verifies that a user with the appropriate permissions can fetch a decoder and the last time it was used. This test relies on database access to fetch decoders and access to an RSA file to verify the authenticity of the user's access token. Since this is a unit test and its responsibility is not to verify the solution integration, it mocks the classes that access the mentioned resources using the *Mockito Testing Framework*.

This test only verifies the isolated behavior of the service *DataDecoderCollectorService* - line **14**. Other classes - lines **5**, **8** and **11** - needed by the service, are mocked and then injected in it with the annotation *@InjectMocks*.

```
1  @Mock
2  DataDecoderCollector collector;
3
4  @Mock
5  DataDecoderMapper mapper;
6
7  @Mock
8  TokenExtractor tokenExt;
9
10 @Mock
11 LastTimeSeenDecoderRepository repository;
12
13 @InjectMocks
14 DataDecoderCollectorService service;
15
16 @Test
17 void ensureServiceWorksWhenUserHasPermissionsAndDecoderWasNeverUsed() {
18     var decoder = CommonObjectsFactory.dataDecoder();
19
20     Mockito.when(tokenExt.extract(Mockito.any(AccessTokenDTO.class)))
21             .thenReturn(CommonObjectsFactory.validTenantInfo());
22     Mockito.when(collector.collect()).thenReturn(Stream.of(decoder));
23
24     var list = service.collectAll(new FakeAccessTokenDTO()).toList();
25
26     Mockito.verify(tokenExt, Mockito.times(1))
27         .extract(Mockito.any(AccessTokenDTO.class));
28     Mockito.verify(collector, Mockito.times(1)).collect();
29     Mockito.verify(mapper, Mockito.times(1)).domainToDto(decoder, 0L);
30
31     Assertions.assertEquals(list.size(), 1);
32 }
```

Listing 5.10: Unit Test - Data Decoder Backend Container

The Listing 5.11 presents some tests that verify the behavior of *DeviceCommand*. This test relies in the *Jest Testing Framework*.

```
1  describe('Device Command Unit Test', () => {
2      it('should deep clone every single parameter', () => {
3          const deviceCommand =
4              new DeviceCommand('openValve', 'openValve', 'ldcn', 0, 70);
5          const clone = deviceCommand.clone();
6          expect(clone.id).toBe(deviceCommand.id);
7          expect(clone.name).toBe(deviceCommand.name);
8          expect(clone.ref).toBe(deviceCommand.ref);
9          expect(clone.payload).toBe(deviceCommand.payload);
10         expect(clone.port).toBe(deviceCommand.port);
11     });
12     it('should be invalid when it has no id', () => {
13         const deviceCommand =
14             new DeviceCommand('', 'openValve', 'ldcn', 0, 70);
15         expect(deviceCommand.isValid()).toBeFalsy();
16     });
17 });
```

Listing 5.11: Unit Test - Device Management Frontend Model Library

5.3.2 Integration Tests

This section, as an example, describes the integration tests preformed in the **Device Ownership Flow** Container and then moves on to **Notification Management Backend**.

The tool used to ease the formulation of integration tests was *Test Containers*. This tool uses docker to fabricate the needed environment where integration tests can run. It is responsible for automaticaly starting and shuting down the containers needed to performed this tests.

The code in Listing 5.12 verifies that the message broker can be reached by **Device Ownership Flow**.

```
1  @QuarkusTest
2  class DeviceInformationEmitterTest {
3
4      @Inject
5      DeviceInformationEmitter emitter;
6
7      @Inject
8      RoutingKeysProvider provider;
9
10     @Inject
11     @Any
12     InMemoryConnector connector;
13
14     @Test
15     void testEmitterCanReachRabbitMQ() {
16         var unknown = provider
17             .getInternalTopicBuilder(RoutingKeysBuilderOptions.SUPPLIER)
18             .withContainerType(ContainerTypeOptions.IDENTITY_MANAGEMENT)
19             .withContextType(ContextTypeOptions.DEVICE_IDENTITY)
20             .withOperationType(OperationTypeOptions.UNKNOWN)
21             .build().orElseThrow();
22
23         var deviceId = DeviceId.of(UUID.randomUUID());
24
25         emitter.next(new DeviceTopicMessage(deviceId, unknown));
26
27         var payload = connector.sink("egress-device-ownership")
28                 .received().get(0).getPayload();
29
30         Assertions.assertNotNull(payload);
31     }
32 }
```

Listing 5.12: Integration Test - Message Broker - Device Ownership Flow

In this example a *RabbitMQ* instance, the only system that this container depends on, is started by the *Test Containers* library before running the tests and shuted down once they end.

The class tested is *DeviceInformationEmitter*, line **5**. As we can see, a message is sent in line **25** and, as expected it is recieved in line **27**.

The code in Listing 5.13 verifies that the database can be reached by the **Notification Management Backend**.

```
public class NotificationRepositoryImplTest extends IntegrationTest {

    @Autowired
    NotificationRepositoryImpl repository;

    @Test
    public void ensureDatabaseCanBeReached() {
        var single = Domains.single(DomainId.of(UUID.randomUUID()));
        var type = ContentType.of("a", "a", NotificationLevel.CRITICAL);
        var query = NotificationBasicQuery.of(single, List.of(type));

        Assertions.assertDoesNotThrow(() -> repository.find(query));
    }
}
```

Listing 5.13: Integration Test - Database - Notification Management Backend

This test verifies that the *NotificationRepositoryImpl* can reach the database by ensuring that no exception is thrown when executing a query to it. This class extends *IntegrationTest*, the behavior of it is similar to the *IntegrationTest* class discussed in the next section.

5.3.3 Functional Tests

This section, as an example, starts to focus on functional tests performed in the **Data Decoder Master Backend**. Other service and configuration scope backend containers rely on similar tests.

The tool used to ease the formulation of functional tests was, once again, *Test Containers*. Contrary to *Quarkus*, *Spring Boot* doesn't provide a ready to use environment according to the application needs, for that reason, the following Listings 5.14 and 5.15 present the needed setup to run functional tests using *Test Containers* and *Spring Boot*.

```
public class DatabaseContainerTest extends
    ↪ PostgreSQLContainer<DatabaseContainerTest> {

    private static final String IMAGE_VERSION = "data-decoder-database";
    private static DatabaseContainerTest container;

    private DatabaseContainerTest() {
        super(DockerImageName.parse(IMAGE_VERSION)
            .asCompatibleSubstituteFor("postgres:14.5"));
    }

    public static DatabaseContainerTest getInstance() {
        if (container == null) {
            container = new DatabaseContainerTest()
                .withUsername("user")
                .withPassword("sa")
                .withEnv("POSTGRESQL_USER", "user")
                .withEnv("POSTGRESQL_PASSWORD", "sa")
                .withExposedPorts(PostgreSQLContainer.POSTGRESQL_PORT);
        }
        return container;
```

```
21      }
22
23      @Override
24      public void stop() {
25          //do nothing, JVM handles shut down
26      }
27  }
```

Listing 5.14: Functional Test - Message Broker - Data Decoder Backend Setup

The *DatabaseContainerTest* follows the Singleton Pattern to ensure that all tests use the same instance. In line **8** we can see that the base image is *PostgresSQL*, but the image actually used is *data-decoder-database*. This image is *PostgresSQL* with the data decoder schema and built in line **11** of the script referenced in Listing 5.3. The same notion is applied for the Message Broker Container. Theses two containers are the ones that **Data Decoder Master Backend** depends on.

The Listing 5.15 presents the foundation of functional and integration tests.

```
1  @SpringBootTest
2  @Testcontainers
3  @ContextConfiguration(initializers =
       ↪ {IntegrationTest.Initializer.class})
4  @ActiveProfiles(profiles = "test")
5  public abstract class IntegrationTest {
6      static class Initializer implements
7          ApplicationContextInitializer<ConfigurableApplicationContext> {
8          public void initialize(ConfigurableApplicationContext context) {
9              db.withDatabaseName("decoder");
10             TestPropertyValues.of(
11                 "spring.datasource.url=" + db.getJdbcUrl(),
12                 "spring.datasource.username=" + db.getUsername(),
13                 "spring.datasource.password=" + db.getPassword(),
14                 "spring.rabbitmq.host=" + mb.getHost(),
15                 "spring.rabbitmq.port=" + mb.getAmqpPort(),
16                 "spring.rabbitmq.username=" + mb.getAdminUsername(),
17                 "spring.rabbitmq.password=" + mb.getAdminPassword()
18             ).applyTo(context.getEnvironment());
19         }
20     }
21
22     @Container
23     public static PostgreSQLContainer<?> db =
24         DatabaseContainerTest.getInstance();
25
26     @Container
27     public static RabbitMQContainer mb =
28         MessageBrokerContainerTest.getInstance();
29
30     protected ResultSet performQuery(String sql) throws SQLException {
31         DataSource ds = getDataSource(postgresSQLContainer);
32         Statement statement = ds.getConnection().createStatement();
33         statement.execute(sql);
34         ResultSet resultSet = statement.getResultSet();
35
36         if (resultSet != null) resultSet.next();
37
38         return resultSet;
```

```
39      }
40  }
```

Listing 5.15: Functional Test - Foundation - Data Decoder Backend Setup

The application environment properties are loaded in line **10** to **18** according to the containers used. The *@SpringBootTest* annotation indicates that the full application has to be started, the *@TestContainers* and *@Container* annotations indicate that docker containers are to be used, and the *@ActiveProfiles* annotation changes the profile in use so that specific beans are not loaded.

The following sample, Listing 5.16, presents a functional test related to the database.

```
1  public class DataDecodersRepositoryImplTest extends IntegrationTest {
2
3      @Autowired
4      DataDecodersRepositoryImpl repository;
5
6      @AfterEach
7      public void cleanUp() throws SQLException {
8          performQuery("TRUNCATE decoder");
9      }
10
11     @Test
12     public void ensureSavedDecoderCanBeFound() throws SQLException {
13         var query = "INSERT INTO decoder(device_type, script) "
14             + "VALUES ('lgt92', 'ascma')";
15         performQuery(query).close();
16
17         var found = repository.findById(SensorTypeId.of("lgt92"))
18             .orElseThrow();
19
20         Assertions.assertEquals("lgt92", found.id().value());
21         Assertions.assertEquals("ascma", found.script().value());
22     }
23 }
```

Listing 5.16: Functional Test - Database Interaction - Data Decoder Backend

As we can see this test extends the foundation described before. In line **4** the service to be tested, *DataDecodersRepositoryImpl*, is loaded. In the test presented a new Data Decoder is stored directly in the database and then the repository service attempts to fetch it. A database clean up is preformed after each test as described in lines **6** to **9**.

The test presented in Listing 5.17, verifies the correct interaction with the message broker container.

```
1  public class DataDecoderInfoEmitterTest extends IntegrationTest {
2
3      @Autowired
4      DataDecoderHandlerService publisher;
5
6      @Autowired
7      RabbitAdmin rabbitAdmin;
8
9      @Autowired
10     RabbitTemplate amqpTemplate;
11
```

```
12      @Autowired
13      RoutingKeysProvider provider;
14
15      @BeforeEach
16      public void init() {
17          if (rabbitAdmin.getQueueInfo("info") == null) {
18              var supplierBuilder = RoutingKeysBuilderOptions.SUPPLIER;
19              var keys = provider
20                  .getInternalTopicBuilder(supplierBuilder)
21                  .withContextType(ContextTypeOptions.DATA_DECODER)
22                  .withContainerType(ContainerTypeOptions.DATA_DECODER)
23                  .withOperationType(OperationTypeOptions.INFO)
24                  .build().orElseThrow();
25              var queue = QueueBuilder.durable("info").build();
26              rabbitAdmin.declareQueue(queue);
27              rabbitAdmin.declareBinding(BindingBuilder.bind(queue)
28                  .to(new TopicExchange(IoTCoreTopic.INTERNAL_EXCHANGE))
29                  .with(keys.toString()));
30          }
31      }
32
33      @Test
34      public void ensureNewDecoderIsSentAsExpected() {
35          publisher.publishUpdate(new DataDecoder(
36              SensorTypeId.of("lgt92"), SensorTypeScript.of("asmc")));
37
38          var dto = (DataDecoderNotificationDTOImpl)
39              amqpTemplate.receiveAndConvert("info");
40
41          var type = DataDecoderNotificationTypeDTOImpl.UPDATE;
42
43          Assertions.assertEquals(type, dto.type);
44          Assertions.assertEquals("lgt92", dto.sensorType);
45          Assertions.assertEquals("asmc", dto.information.script);
46      }
47  }
```

Listing 5.17: Functional Test - Message Broker Interaction - Data Decoder Backend

In this test the class to verify is the *DataDecoderHandlerService*. Once again this test extends the *IntegrationTest* class. Using *RabbitAdmin*, its created a queue that subscribes to the expected type of routing keys in lines **18** to **29** and then binded to the expected topic - *INTERNAL_TOPIC*. An update is published in line **35** using the *DataDecoderHandlerService* and then captured with *RabbitTemplate* in line **38**.

The **Data Gateway** Container was tested against different post requests to its data retention endpoint, an example of this tests is described in Listing 5.18.

```
1   @QuarkusTest
2   class DataControllerTest {
3
4       @Test
5       public void testInfoTypeDetection() {
6           var errorType = "Info Type must be of value encoded or decoded";
7           given().when()
8               .accept(ContentType.JSON)
9               .contentType(ContentType.JSON)
10              .header("Authorization", "pass")
```

```
11              .post("/sensor-data/fleet/wrong/lgt92")
12              .then()
13              .statusCode(400)
14              .body("error", containsString(errorType));
15      }
16  }
```

Listing 5.18: Functional Test - Rest Client Interaction - Data Gateway

This test simply attempts to send an HTTP POST request to an invalid resource - line **11**.

5.3.4 End-to-End Tests

This section presents some of the end-to-end tests of **Sensae Console**. This tests evaluate how the system responds to various user actions.

All end-to-end tests rely on *Cypress*, an end-to-end testing framework. To improve tests readability new cypress commands were created. The methods *anonymous*, *logout* and *goToIdentityPage* are some examples of this commands - Listing 5.19.

```
1  declare namespace Cypress {
2      interface Chainable<Subject> {
3          anonymous(): void;
4          logout(): void;
5          goToIdentityPage(): void;
6      }
7  }
8
9  Cypress.Commands.add('anonymous', () => {
10     console.log('Custom command: Anonymous Login');
11     cy.contains('Login').click();
12     cy.contains('Anonymous').click();
13 });
14
15 Cypress.Commands.add('logout', () => {
16     console.log('Custom command: Logout');
17     cy.get('#account').click();
18     cy.contains('Logout').click();
19 });
20
21 Cypress.Commands.add('goToIdentityPage', () => {
22     console.log('Custom command: go to Identity Page');
23     cy.get('#tools').click();
24     cy.contains('Identity Management').click();
25 });
```

Listing 5.19: End-to-End Test - Custom Commands - UI Aggregator

As an example, the *anonymous* command searches for something with the text *Login*, and clicks on it.

The Listing 5.20 presents a test that ensures anyone can enter the system as an anonymous user.

```
1  describe('ui-aggregator', () => {
2      beforeEach(() => cy.visit('/'));
3      it('should display welcome message for anonymous user', () => {
4          cy.anonymous();
```

```
5           cy.contains("Valid Credentials");
6           cy.logout();
7       });
8   });
```

Listing 5.20: End-to-End Test - Anonymous Authentication - UI Aggregator

The test verifies that a successful login notification is received in line **5**. Both of the commands previously described are used in this test.

The Listing 5.21 presents a test that walks though the **Identity Management Page** verifying that an authenticated manager can see every available domain.

```
1  describe('ui-aggregator', () => {
2      beforeEach(() => cy.visit('/'));
3      it('should present various default domains', () => {
4          cy.managerLogin();
5          cy.goToIdentityPage();
6          cy.contains("root");
7          cy.get(".toggle").click();
8          cy.contains("public");
9          cy.contains("unallocated");
10     });
11 });
```

Listing 5.21: End-to-End Test - Discover Available Domains - Identity Management

This test verifies that a user in the root domain can see all default domains in the **Identity Management Page**, as described in Identity Management Concern (Appendix G).

5.3.5 Architectural Tests

This section presents some of the architectural tests of **Sensae Console**'s Containers. This tests are only performed in the backend containers. As an example it will be displayed one test for the **Configuration / External Services Scope** and another for the **Data Flow Scope**.

The tool used was ArchUnit, according to Richards and Ford 2020, it "provides a variety of predefined governance rules codified as unit tests and allows architects to write specific tests that address modularity"".

The Listing 5.22 presents an example of the tests made for **Configuration / External Services Scope** backend containers.

```
1  @AnalyzeClasses(packages = "pt.sensae.services")
2  public class ApplicationArchitectureTest {
3
4      @ArchTest
5      static final ArchRule architecture = Architectures
6          .onionArchitecture()
7          .domainModels("..domain..")
8          .domainServices("..domainservices..")
9          .applicationServices("..application..")
10         .adapter("amqp connector", "..amqp..")
11         .adapter("in memory persistence", "..memory..")
12         .adapter("postgres persistence", "..postgres..")
13         .adapter("graphql endpoint", "..graphql..")
```

```
14          .ignoreDependency(resideInAPackage("..boot.."), alwaysTrue());
15
16      @ArchTest
17      static final ArchRule domainMustNotDependOnFrameworks =
18          ArchRuleDefinition.noClasses().that()
19              .resideInAnyPackage("..domain..")
20              .should()
21              .dependOnClassesThat()
22              .haveNameMatching("org.springframework.")
23              .orShould()
24              .dependOnClassesThat()
25              .haveNameMatching("javax.persistence.")
26              .because("Domain should be free from dependencies");
27  }
```

Listing 5.22: Architectural Test - Onion Architecture - Device Management Backend

The test *architecture* at lines **4** to **14** ensures that the onion architecture is followed. The test *domainMustNotDependOnFrameworks* at lines **16** to **26** ensures that the domain and domain services components are free of dependencies.

The Listing 5.23 presents an example of the tests made for **Data Flow Scope**.

```
1   @AnalyzeClasses(packages = "pt.sensae.services")
2   public class ArchitecturalTest {
3
4       @ArchTest
5       static final ArchRule architecture = Architectures
6           .onionArchitecture()
7           .domainModels("..domain..")
8           .applicationServices("..application..")
9           .adapter("amqp internal topic connector", "..internal..")
10          .adapter("amqp ingress data topic connector", "..ingress..")
11          .adapter("amqp egress data topic connector", "..egress..")
12          .adapter("in memory persistence", "..memory..")
13          .ignoreDependency(resideInAPackage("..boot.."), alwaysTrue())
14          .allowEmptyShould(true);
15
16      @ArchTest
17      static final ArchRule domainMustNotDependOnFrameworks =
18          ArchRuleDefinition.noClasses().that()
19              .resideInAnyPackage("..domain..")
20              .should().dependOnClassesThat()
21              .haveNameMatching("org.eclipse.")
22              .orShould().dependOnClassesThat()
23              .haveNameMatching("com.fasterxml.")
24              .orShould().dependOnClassesThat()
25              .haveNameMatching("com.google.")
26              .orShould().dependOnClassesThat()
27              .haveNameMatching("javax.")
28              .because("Domain should be free from Frameworks");
29  }
```

Listing 5.23: Architectural Test - Simplified Onion Architecture - Data Processor Flow

The test *architecture* at lines **4** to **12** ensures that, such as the previous test, the onion architecture is followed. The difference between the two is that this one allows empty

components - line **12**, since the **Data Flow Scope** containers have no domain services. The test *domainMustNotDependOnFrameworks* at lines **14** to **26** ensures that the domain component are free of dependencies.

5.4 Synopsis

This chapter introduced the most important technical decisions taken during the solution's implementation. These decisions were followed with a technical description of **Sensae Console** tailored for those who manage and develop the platform. Lastly some of the tests that ensure the proper operation of the solution were presented.

In the next chapter, Evaluation, the performance of the platform will be extensively discussed.

Chapter 6

Evaluation

This chapter intent is to describe the evaluations preformed against the solution. For that the following sections will tackle:

- Objectives and execution environment of this evaluation;
- Approach applied to evaluate the software developed;
- Drafted experiences and results collected;
- Analysis of the results collected;
- Observations taken from the analysis conducted.

The expected behavior of the system according to functional requirements can be attested with deterministic tests presented in Section 5.3. On the other hand, some non-functional requirements, such as performance and usability requirements, can't be deterministically attested with simple tests.

Since the company and this project's solution were both in the early stages of conception no strick usability requirements were defined. The experiences here documented focus on the performance of the solution according to the points defined in Section 3.2.

6.1 Objectives

The objective of this evaluation is to determine the throughput limits of the entire solution (**Sensae Console** and **External Services**) regarding data ingestion, within the requirements detailed in Section 3.2.

Since the solution was designed to scale infinitely and handle high-levels of throughput, the performance of it in a multi-tenant instance is undermined. The evaluation should instead focus on environments where resources are more constrained.

Therefore, the performance of the solution will be tested against above-average usage conditions of small to medium organizations that require a single-tenant instance, either on-site or in the cloud.

This type of organizations encompass entities such as:

- Public Institutes: town councils, public transportation organizations, waste management departments and others;
- Private owned business: chicken farms, greenhouse farms, goods transportation agencies, industrial warehouses, agriculture cooperatives and others.

The objective is to determine the platform limits of data ingestion, processing, storage and real-time supply within the desired requirements.

This evaluation also helps to understand what components are the first to degrade the performance of the system.

6.2 Approach

The approach taken to evaluate the solution was to send increasingly higher volumes of HTTP requests to the Sensae Console Data Ingestion Endpoint in order to determine the platform limits of healthy operation.

Given the type of organizations that require this deployment mode it is expected that the number of devices installed doesn't go beyond 500[1].

The evaluation encompasses 4 test scenarios, one for the platform and one for each external service: (i) Sensae Console, (ii) Fleet Management, (iii) Smart Irrigation, (iv) Notification Management.

The performance tests use the *K6* tool. This tool allows one to design performance tests entirely in *Javascript*. An example of the scripts developed is presented in Appendix M. The *K6* tool produces various metrics that are then analyzed using *R*, an example of the analysis scripts developed is presented in Appendix N.

For simplicity, the solution was deployed in a Virtual Machine (VM) Instance of the Google Cloud Platform, type *'e2-standard-4'*, with the following specs:

- Memory: 16 GB;
- Number of vCPU Cores: 4;
- Disk Type: Balanced Persistence Disk.

As of September, 2022, the cost associated with this VM rounds the 100€per month.

These tests were executed through the author' machine which may undermine the communication with the VM Instance, e.g. number of HTTP requests and stability of the Websocket connection.

The approach taken isn't an attempt to mimic normal usage patterns but simply to envision the platform throughput limits. Even though an approach closer to the reality would present results easier to interpreter, it would take to much time to design, implement and run these realistic tests. Therefore, metrics such as the interval between two consecutive Uplinks of the same device, were severely narrowed.

6.3 Experiences

As described before, 4 scenarios that emulate a variable number of devices sending uplinks to **Sensae Console** were tested:

1. Sensae Console;

[1] 500 devices in use by a single entity is expected to be a huge amount of devices for the realistic business needs of small/medium organizations

2. Fleet Management;
3. Notification Management;
4. Smart Irrigation;

Each scenario examines the time it takes to notify a user about a measure or notification since the system collected the correspondent data unit.

The tests preformed against the system ensured that no Device Information, Device Ownership, Data Decoders or Data Processors were cached in the **Data Flow** Scope to create even harsher conditions.

In order to focus on the raw performance of the system, the following conjectures were applied:

- A single type of Data Decoder is used to decode Data Units;
- A single type of Data Processor is used to process Data Units;
- The Data Units sent will evenly require a Data Decoder or a Data Processor;
- The Data Decoder and Data Processor operations are identical, meaning that, given the same input, both must provide the same output;
- A single *Anonymous user* is notified about new measures or notifications;
- Each scenario focus on a single business case, or none at all (first scenario);
- All devices belong to the *Public* Domain (this eases the process of authentication);
- No erroneous data will be sent, e.g. data units with unknown data decoders/processors/devices, incorrect measures or invalid structure;
- The default configuration regarding database connection pools, cache size, cache eviction policies and others is used;
- Ten iterations of requests are sent in each experience, each iteration sends one Data Unit per Device;

The Table 6.1 summarizes the experiences preformed.

Table 6.1: Details about the experiences performed

Experience	Number of Devices	Interval between device uplinks	Average number of uplinks per second	Total number of Uplinks
A	100	10 seconds	9	1000
B	200	10 seconds	18	2000
C	500	10 seconds	45	5000
D	1000	10 seconds	90	10000
E	100	3 seconds	25	1000
F	200	3 seconds	50	2000
G	500	3 seconds	125	5000
H	1000	3 seconds	250	10000

The results of each scenario will be presented in the next sections. The results will be displayed in a table with the following metrics: (i) average, (ii) minimum, (iii) median, (iv) maximum, (v) 90th Percentile and (vi) 95 Percentile.

6.3.1 Sensae Console Experience Scenario

This scenario focus on the platform developed. It mimics an hosting option that separates the External Services from the Sensae Console. The Table 6.2 presents the results related to the time it takes for a Data Unit to be processed and supplied to a potential external service.

Table 6.2: Results for the Sensae Console Scenario (in seconds)

Experience	Average	Min	Median	Max	90% Percentile	95% Percentile
A	0.199	0.181	0.188	0.859	0.193	0.197
B	0.200	0.179	0.189	0.854	0.197	0.199
C	0.206	0.183	0.194	0.865	0.206	0.213
D	0.285	0.181	0.197	2.298	0.425	0.895
E	0.201	0.181	0.189	0.847	0.198	0.207
F	0.204	0.181	0.191	0.864	0.206	0.214
G	0.285	0.183	0.248	0.958	0.398	0.454
H	20.91	0.201	22.08	28.96	28.21	28.47

These results show that, a system focused only on processing and delivering Data Units to External Services, can successfully handle 125 Data Units per second while answering the defined requirements.

The experience **H** had the measures supplied with a delay of 20 seconds on average, these results are far from the optimal response time defined in the requirements.

6.3.2 Fleet Management Experience Scenario

This scenario focus on the Fleet Management Service. The Table 6.3 presents the results related to the time it takes for a Data Unit to be processed and supplied as a measure to the user.

Table 6.3: Results for the Fleet Management Scenario (in seconds)

Experience	Average	Min	Median	Max	90% Percentile	95% Percentile
A	0.206	0.182	0.193	0.765	0.205	0.215
B	0.207	0.182	0.193	0.770	0.211	0.220
C	0.225	0.184	0.209	0.921	0.241	0.257
D	2.871	0.189	0.674	16.69	10.14	13.31
E	0.213	0.185	0.200	0.780	0.214	0.220
F	0.214	0.180	0.200	0.789	0.224	0.235
G	0.921	0.183	0.754	2.992	2.015	2.326
H	38.43	0.218	35.39	83.37	71.52	77.08

These results show that, a system focused on the Fleet Management business case, can successfully handle 125 Data Units per second while answering the defined requirements.

The experience **H** had the measures supplied with a delay of 40 seconds on average, this results are far from the optimal response time defined in the requirements.

Currently, the devices used in production for this business case send, at best, measures every minute, this means that the maximum number of devices concurrently communicating with the solution is much higher than the one determined by these tests.

6.3.3 Notification Management Experience Scenario

This scenario focus on the Notification Management Service. The Table 6.4 presents the results related to the time it takes for a Data Unit to be processed and trigger a new alert that is supplied to the user as a notification.

In this experience it was simulated that an average of 10% of the data units would produce an alert.

Table 6.4: Results for the Notification Management Scenario (in seconds)

Experience	Average	Min	Median	Max	90% Percentile	95% Percentile
A	0.268	0.208	0.226	1.697	0.265	0.308
B	0.263	0.196	0.223	1.735	0.245	0.266
C	0.434	0.199	0.235	4.029	0.287	2.364
D	1.678	0.195	0.241	11.95	6.885	9.728
E	0.334	0.207	0.230	3.947	0.323	0.372
F	0.856	0.200	0.241	5.629	2.379	4.875
G	10.02	1.188	11.09	19.08	14.01	15.03
H	27.51	13.48	27.77	35.50	29.68	31.14

The results captured in this experiences infer that this system can comfortably withstand a throughput of around 500 devices (each sending measures every 10 seconds) without undermining its overall behavior. This means that the platform can only process a maximum of 50 notification per second.

6.3.4 Smart Irrigation Experience Scenario

This scenario focus on the Smart Irrigation Service. The Table 6.5 presents the results related to the time it takes for a Data Unit to be processed, stored and supplied as a measure to the user.

Table 6.5: Results for the Smart Irrigation Scenario (in seconds)

Experience	Average	Min	Median	Max	90% Percentile	95% Percentile
A	0.204	0.181	0.190	0.761	0.202	0.210
B	0.211	0.182	0.199	0.772	0.220	0.233
C	0.389	0.183	0.299	1.290	0.760	0.978
D	20.72	0.205	21.85	44.16	36.98	41.02
E	0.221	0.182	0.205	0.767	0.242	0.254
F	4.358	0.187	2.925	15.69	10.46	13.10
G	12.95	0.255	13.71	20.00	14.60	15.34
H	70.06	0.260	70.98	122.3	109.8	115.4

These results show that, a system focused on the Smart Irrigation business case, can successfully handle 45 Data Units per second while answering the defined requirements. The experiences **D**, **F**, **G**, **H** shed a light on the possible limits for this Service.

The results captured in this experiences infer that this system can comfortably withstand a throughput of around 500 devices, each sending their measures every 10 seconds without undermining its overall behavior.

Currently, the devices used in production for this business case send measures every 10 minutes, this means that the maximum number of devices concurrently communicating with the solution is much higher than the one determined by these tests.

6.4 Discussion of the overall results

As seen by the experiences preformed the system was capable of answering the defined requirements in Section 3.2.4.

Apart from the results gathered it's important to mention the following findings:

- The system answered all HTTP requests within an average of 0.2 seconds;
- There was no visible performance discrepancy between the use of Data Decoders versus Data Processors;

- The bootstrapping of **Data Flow** Caches is noticeable in most experiences, specially the lighter ones. The first iteration usually takes the longer to be processed when the system is not overwhelmed;
- The system was always capable of storing all the measures and alerts;
- The VM's RAM didn't surpassed 8GB of usage in any test but the CPU reported usages of 100% under severe load (experiences **D**, **G**, **H**).

The experiences performed helped to determine the platform throughput limits but didn't indicated what components were underperforming and degrading the results.

In the following sections some components are individually evaluated so that the bottlenecks of the system can be found.

Looking at the architecture the five logical bottlenecks are:

- The Sensae Console Data Ingestion Endpoint that collects requests (in the **Data Relayer** Container);
- The process of filling the **Data Flow Scope** Caches with the information managed by the various containers in the **Configuration Scope**;
- The **Message Broker** that routes data units through the system;
- The databases in the **External Services** that store measures or notifications;
- The Websocket implementation used by GraphQL to supply measures and notifications in the Backends of the **External Services**.

6.4.1 Data Ingestion Endpoint Performance

In order to evaluate this piece of the system first the **H** experiences of each scenario are presented since they represent the higher throughput of all experiences preformed.

These experiences yielded the following results:

Table 6.6: Data Ingestion Endpoint response time results (in milliseconds)

Scenario	Average	Min	Median	Max	90% Percentile	95% Percentile
1	186.76	169.22	176.54	463.7	197.02	213.38
2	193.24	170.02	182.12	581.6	219.16	249.24
3	227.48	169.19	200.17	727.1	319.77	376.98
4	179.83	167.09	178.99	2632	186.36	189.27

These results present a stable performance even under a high number of requests, apart from the maximum value obtained in scenario 4, a clear outlier, all requests were answered within a second.

In order to find the amount of requests that would leave the system unresponsive more tests were performed, this tests only focused on the Data Ingestion Endpoint responses. The system was capable of ingesting around 600 requests per second before starting to drop requests.

6.4.2 Data Processor versus Data Decoder Performance

The experiences preformed also helped to understand if the Data Decoder underperformed when compared to the Data Processor. This was the expected result since the Data Decoder uses an embedded *Javascript* engine to process data units, and the Data Processor relies only on *Java* to process messages.

As an example, the following chart presented in Figure 6.1 helps to debunk this belief.

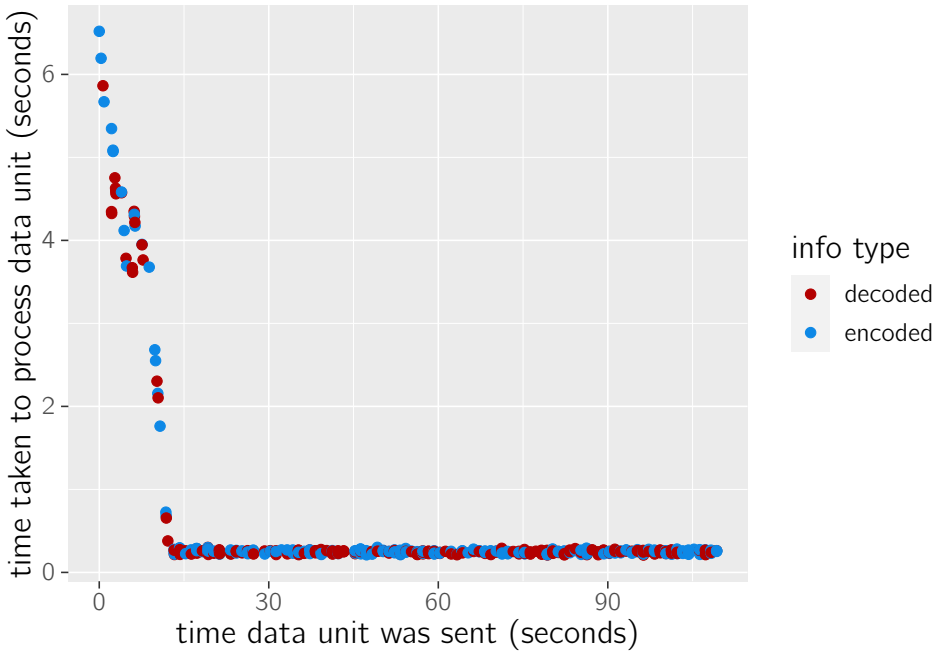


Figure 6.1: Notification Management Scenario - Experience C - Scatter Chart

The X axis represents time in seconds since the first request with a data unit was sent, the Y axis represents the time it took for the client to receive the corresponding notification. Each dot represents a data unit.

The data units with the 'decoded' *info type*, in red, were sent to the Data Processor Concern and the ones with the 'encoded' *info type*, in blue, were sent to the Data Decoder Concern.

In conjunction with the chart, the Table 6.7 presents some analysis preformed against the results of Experience C in the Notification Management Scenario.

Table 6.7: Metrics collected (in seconds) - Notification Management Scenario - Experience C

Info Type	Average	Min	Median	Max	90% Percentile	95% Percentile
decoded	0.558	0.209	0.252	5.863	0.286	3.923
encoded	0.528	0.212	0.253	6.519	0.288	3.479

As we can see the time taken to process or decode a data unit is very similar, this is possible due to *GraalVM*.

6.4.3 Data Flow Caching Process Performance

The experiences preformed clearly display the process mentioned during the Design Chapter in Figure 4.14, about how the **Data Flow** state is maintained.

In the following charts, Figures 6.2 and 6.3, it's possible to envision the various caches in the **Data Flow** Scope being filled during the first iteration.

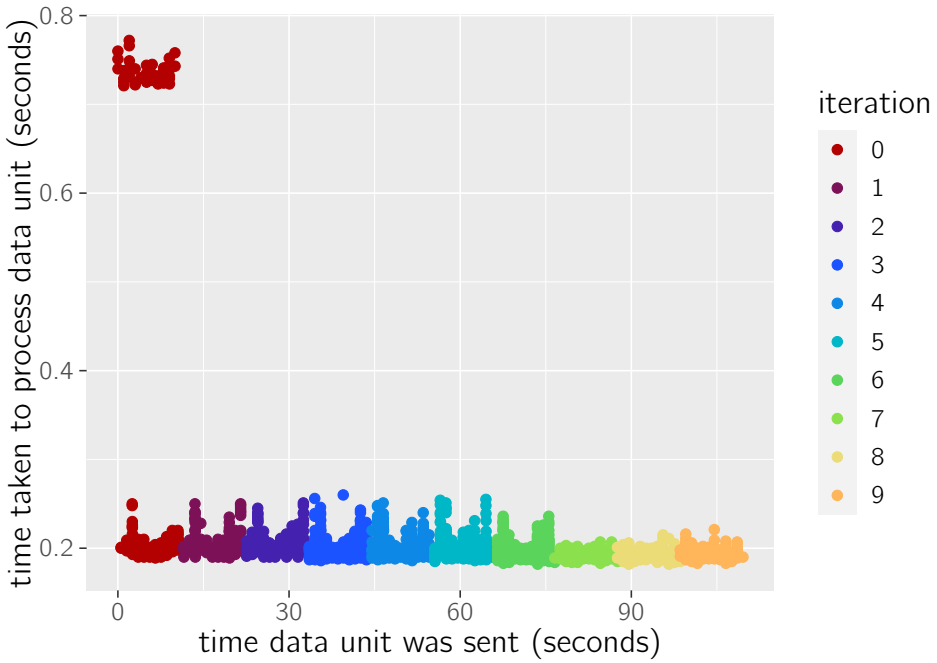


Figure 6.2: Smart Irrigation Scenario - Experience B - Scatter Chart

In both charts the X axis represents time in seconds since the first request with a data unit was sent, the Y axis represents the time it took for the client to receive the corresponding measure. Each dot represents a data unit. Each color represents the test iteration responsible for sending the data unit.

The chart in Figure 6.3 also shows that the system started to underperform around the 65 seconds mark. The **Data Flow** caches were already stable and therefore, under these experiences, it is plausible to say that this process doesn't cause the performance degradation seen in higher throughput experiences.

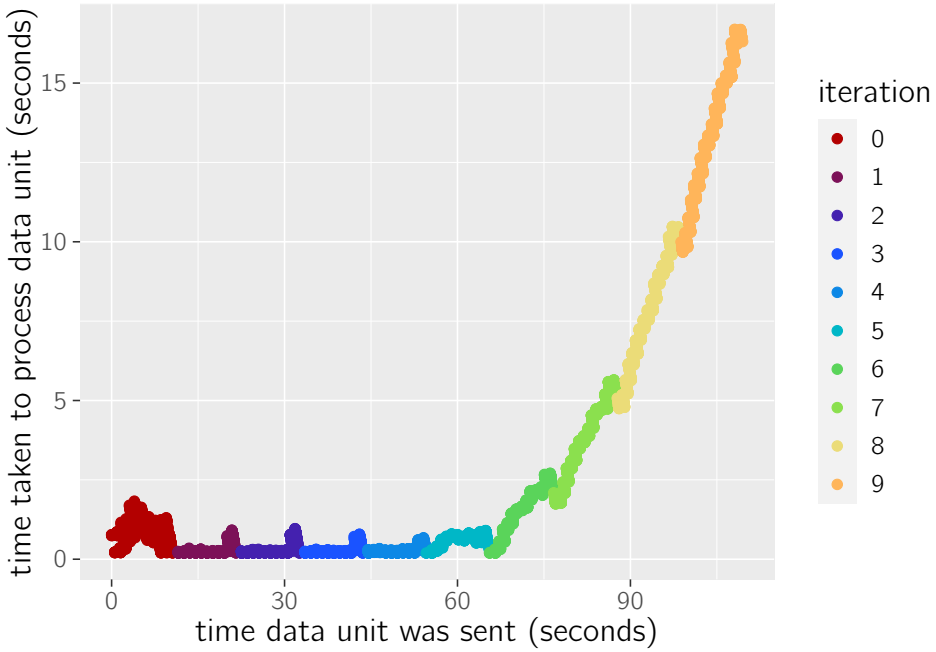


Figure 6.3: Fleet Management Scenario - Experience D - Scatter Chart

6.4.4 External Services Database Performance

Another important question is wether the performance degradation recorded is due to database access or not.

The experiences performed were able to determine that this was in fact the case with the **Notification Management Database**, and, to an extent the **Smart Irrigation Business Database**. As explained in Section 5.1.6, the database used for those containers is *PostgresSQL*. This database, contrary to the one used in **Fleet Management Data Database** and **Smart Irrigation Data Database**, is not focused on high-throughput ingestion.

The following chart, Figure 6.4, shows the discrepancy between storing and serving GPS locations with the **Fleet Management Backend**. The experiment was performed by mimicking 1500 devices, each sending 10 data units with an interval of around 10 seconds.

This chart represents the number of data units/measures ingested, stored and supplied (Y axis) over time (X axis).

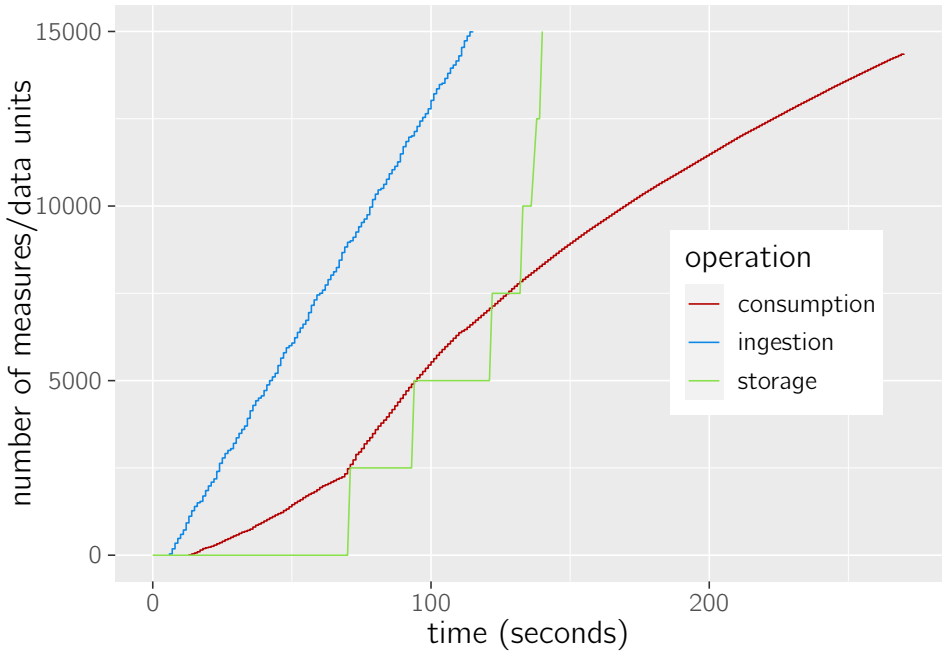


Figure 6.4: Time Taken to Ingest, Store and Supply Measures - Line Chart

This Chart displays three distinct lines:

- Consumption (in red): Measures received by a websocket client connected to the **Fleet Management Backend**;
- Ingestion (in blue): Data Units ingested by the **Data Relayer**;
- Storage (in green): Measures stored in the **Fleet Management Data Database**.

The data is stored in *QuestDB* via ILP and therefore is only committed after a while. The time taken for data to be committed is derived from various parameters and conditions as explained in the Commit Strategy Page[2] of *QuestDB*.

The chart shows that data is stored long before it is consumed by the websocket client. It also shows that the time between storage and ingestion is relatively small, and once the **Data Flow** caches stabilize this gap starts to decrease. Implying that the **Message Broker** does not fall behind the ingestion throughput enforced by the test (136 request per second).

6.4.5 System Bottlenecks

This section briefly discusses the bottlenecks discovered during the performance tests and analysis preformed.

The components that degraded the test results the most were the GraphQL Subscriptions as envisioned in Figure 6.4.

The next bottleneck of the solution appears to be the *PostgresSQL* Databases, this assessment is based on the result's discrepancy between the three Scenarios.

[2]link to QuestDB Commit Strategy Page

The experiences **H** of each scenario, and specially the one related to the platform, foresee that the **Message Broker** is the next logical bottleneck of the system.

The Sensae Console Data Ingestion Endpoint only becomes a bottleneck with a ridiculous amount of devices for a small/medium organization.

If the **Data Flow** cache sizes are configured correctly, the process of filling them will hardly become a bottleneck, specially since it will be very rare to receive a high number of data units from devices, decoders or processors that are not already cached.

6.5 Synopsis

This evaluation determined that, for the requirements defined in Section 3.2.4, a type *'e2-standard-4'* VM instance in Google Cloud Platform is sufficient. The conducted analysis also helped to identify the most important bottlenecks to tackle in the future: (i) GraphQL Subscriptions, (ii) *PostgresSQL* Databases, (iii) Message Broker.

The performance tests helped the company to understand the platform limits and how far the production environments are from reaching those limits.

Based on this evaluation, the work described before and the knowledge gathered during the project, the following chapter describes the author opinion regarding the solution.

Chapter 7

Conclusion

This chapter discusses the Achievements, Unfulfilled Results, and Future Work of this project. In the end it's presented an overview of the influence this work had on the development of the solution and the author perception of the IoT landscape.

This work had two main objectives:

- Create a platform to ease the development of IoT solutions;
- Create PoCs that tackled business cases related to IoT;

During this project's time span it was clear that the initial objectives were much more challenging and ambitious than envisioned given the time and resources available. The constant changes made to the requirements regarding the business cases lead to a lot of wasted time and resources. Nevertheless, the author focused on three business cases. They were addressed according to the requirements discussed during meetings with costumers. The developed PoCs had a positive evaluation regarding their performance. Even though no factual survey was made, the costumers had favorable opinions regarding the Notification Management Service for Indoor Fire Detention and the Smart Irrigation Service for Greenhouse Humidity Control. It was also clear in Phase II, that most of the work needed to implement these PoCs could be integrated in the platform, **Sensae Console**.

7.1 Achievements

The developed PoCs allowed, in the first phase of the project, to determine the various processes that most services needed to function. This assessment helped to define the most important functionalities **Sensae Console** had to provide.

After developing the platform and integrating the PoCs in it, its possible to infer that **Sensae Console** tackles the most crucial requirements and concerns in this area. The platform eases the integration with multiple IoT Middlewares while providing ways to homogenize the data sent by virtually any device. The model envisioned to represent devices and their measures is far from being mature and complete but the author thinks the development of a separated, open-source library to handle it, paves the way for constant improvements. The library also facilitates the integration of new custom services with the platform. The rule engine, even though complex, also proved to be an important feature due to its flexibility. With it, and the notification management service, several business cases that don't require a UI can be promptly addressed.

By decoupling the solution's architecture according to the various functionalities and responsibilities discussed it's possible to easily support the hosting requirements of most costumers. One can choose between integrating one or various frontends directly in their platform, create new frontends that consume the provided API or use the complete UI provided by the platform. The UI Aggregator can also be configured to consume and serve custom made services with UI or just an API.

Even though this project is still in its early phases, the work done here paves the way for a platform that is easy to maintain, improve and extend.

For these reasons, the author believes that the pivotal requirements of this project were successfully fulfilled.

7.2 Unfulfilled Results

This project's initial requirements envisioned the creation of PoCs for smart parking and public health condition monitoring for organization *A*. Neither of these two were tackled due to time constrains and the service contracts being cancelled. The same organization that required these two solutions also required the generation of reports with several KPI for their fleet management solution. The creation of these reports was once again postponed and not included in the final list of requirements for this project.

The initial idea behind this project's proposal envisioned that the evaluation of the solution would be preformed by analyzing questionnaires handed to employees of the organization *A*. Once again this objectives were not fulfilled due to the termination of contracts with the organization.

The requirements mentioned above were removed in April after it became clear that organization *A* was not interested in pursuing further agreements.

In retrospective, the initial proposal was ambitious and nearly impossible to fulfill given the time span of the project and the size of the team.

With all this in mind, the author thinks that this project's requirements were partially addressed, nonetheless, the final solution proved itself to answer the most important requirements of the initial proposal.

7.3 Future Work

This project, and the solution it originated, still have a lot of ideas and features not supported. Apart from all the business cases that were not addressed, and the much needed improvements for those that were developed, it is clear that the **Sensae Console** needs to support the following features:

- Post-Processing of device measures: One of the company's project measured the volume of wheat inside Silos. The sensors were installed in the silo's ceiling pointing downwards to measured the distance between themselves and the surface of the wheat. This distance had to be translated to the occupied wheat volume in the silo. Since each silo had different sizes and shapes, there was a need to calculate the required volume depending on the device that sent it. The current solution doesn't easily support this;

- Image and Video support as device measures: One of the company's project filmed the interior of a chicken farm. The sensor was, in this case, a simple camera. The intent behind this project was to stream, in real time, the site, and if an alarm warning about an indoor fire was received the owner could verify it by accessing the live stream;
- UI Custom composition: One of the company's requirements was for the platform to support the creation of UI tailored for each costumer's needs by dragging and dropping resizable elements such as maps, charts, panels with latest/average device measures and buttons to interact with actuators (Mashup-based development);
- Query-able Data Lake with device measures: One of the company's ideas was to provide a simple endpoint to query the latest information regarding any device measures;
- Customizable monthly reports: One of the company's costumers requested the creation and delivery of reports with various monthly KPI, such as: fleet's distance traveled per day, fleet's active/inactive hours per day, frequent stop locations;
- Observability: The author argues that there's a need to monitor the internal state and conditions of the platform in real-time so that problems can be found and resolved faster;
- Automatic Scalability: Currently most costumers request a shared and remote hosting option managed by the company. This means that the number of devices and, consequently, the generated network traffic and data targeting the platform's cloud instance will increase. In the following months the platform should be orchestrated by a tool such as Kubernetes to automatically scale the solution as needed;
- Big Data analytics: Some of the most advanced features this platform could provide would be automatic analytics to help decision making and driving business decisions for costumers. This topic is beyond the knowledge of the author but is something increasingly important in the today's competitive world where every company is trying to squeeze the most value from available assets;
- IoT-A requirements compliance: The requirements gathered by European Lighthouse Integrated Project 2013a focus on may important features that this solution doesn't answer, such as UNI.027, UNI.047 UNI.239, UNI.094 and UNI.023 (European Lighthouse Integrated Project 2013a);
- *Sensor Measurement Lists (SenML)* Standard compliance: The open-source library created should be able to translate between the model envisioned here and SenML;
- Monetizing Policies: The revenue model needs to be discussed so that this solution can be monetized. Normally this platforms measure metrics such as MB of data stored, network bandwidth volume, number of devices registered and others to calculate the monthly bills of each costumer. To do so, one must first register and monetize the metrics related to each costumer and then incorporate a payment system in the platform.

As seen by this list, creating a public, monetizable platform to ease the creation of IoT Services is a complex and drawn-out process. Maintaining a service like this feels even more like an interminable task due to all the business cases surrounding IoT.

7.4 Synopsis

In summary the solution can be seen as a first and very important step to create an IoT platform but it isn't ready to be sold as a service to third-parties. It is advised to keep developing the solution and services surrounding IoT related business cases for at least another year while offering customers early access to the platform. Continuous costumers evaluations would help to guide the solution to the desired outcomes.

Even though it is easy to envision the continuous development of this solution for the forthcoming years, without a solid product, costumers will start to cease their contracts. Without revenue streams it's expected that this solution will be abandoned and the company dissolved. In retrospective, the best approach for the problem in hands was not to build a platform from the ground up but to rely on open-source solutions or paid services.

Nonetheless the author benefited immensely with the development of this project. The author gained a lot of experience and knowledge regarding the IoT world and also the difficulties surrounding the creation of a business from the ground up.

Bibliography

Alaa, Musaab et al. (Sept. 2017). "A Review of Smart Home Applications based on Internet of Things". In: *Journal of Network and Computer Applications* 97. doi: 10.1016/j.jnca.2017.08.017.

Amazon (2022a). *Amazon Cognito*. url: https://aws.amazon.com/cognito/.

– (2022b). *DynamoDB*. url: %5Curl%7Bhttps://aws.amazon.com/dynamodb/%7D.

Andy Clement Sébastien Deleuze, Filip Hanik (2022). *Spring Native*. url: https://docs.spring.io/spring-native/docs/current/reference/htmlsingle/.

Angles, Renzo (2012). "A comparison of current graph database models". In: *2012 IEEE 28th International Conference on Data Engineering Workshops*. IEEE, pp. 171–177.

Angles, Renzo and Claudio Gutierrez (2008). "Survey of graph database models". In: *ACM Computing Surveys (CSUR)* 40.1, pp. 1–39.

Apache (2022). *Apache HTTP Server Project*. url: https://httpd.apache.org/.

Atzori, Luigi, Antonio Iera, and Giacomo Morabito (2010). "The Internet of Things: A survey". In: *Comput. Networks* 54, pp. 2787–2805.

Auth0 (2022). *Auth0 Customer Identity*. url: https://auth0.com/b2c-customer-identity-management.

AWS (2022). *AWS IoT Core*. Accessed: February 8, 2022.

Azure (2022). *Azure Active Directory (Azure AD)*. url: https://azure.microsoft.com/en-us/services/active-directory/.

Bellemare, A. (2020). *Building Event-Driven Microservices*. O'Reilly Media. Chap. 3, pp. 39–45. isbn: 9781492057840.

Bernstein, David (2014). "Containers and cloud: From lxc to docker to kubernetes". In: *IEEE cloud computing* 1.3, pp. 81–84.

Bibri, Simon Elias (2018). "The IoT for smart sustainable cities of the future: An analytical framework for sensor-based big data applications for environmental sustainability". In: *Sustainable Cities and Society* 38, pp. 230–253. issn: 2210-6707. doi: \url{https://doi.org/10.1016/j.scs.2017.12.034}. url: %5Curl%7Bhttps://www.sciencedirect.com/science/article/pii/S2210670717313677%7D.

Blomstedt, Fredrik et al. (2014). "The arrowhead approach for SOA application development and documentation". In: *IECON 2014-40th Annual Conference of the IEEE Industrial Electronics Society*. IEEE, pp. 2631–2637.

Brown, Simon (2018). *The C4 model for visualising software architecture*. [Online; accessed 30. Jun. 2022]. url: https://c4model.com.

Brush, A.J. Bernheim et al. (2011). "Home Automation in the Wild: Challenges and Opportunities". In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI '11. Vancouver, BC, Canada: Association for Computing Machinery, pp. 2115–2124. isbn: 9781450302289. doi: 10.1145/1978942.1979249. url: https://doi.org/10.1145/1978942.1979249.

By, Slides and Jack ZhenMing Jiang (Nov. 1995). "Architectural Blueprints–The "4+ 1" View Model of Software Architecture". In: [Online; accessed 30. Jun. 2022].

Byars, Brandon (Dec. 2021). *You Can't Buy Integration*. url: https://martinfowler.com/articles/cant-buy-integration.html.

Cake, Data (2021). *Making IoT accessible to Everyone*. Accessed: February 8, 2022.

Cerny, Tomas, Michael J Donahoo, and Jiri Pechanec (2017). "Disambiguation and comparison of soa, microservices and self-contained systems". In: *Proceedings of the International Conference on research in adaptive and convergent systems*, pp. 228–235.

Chen, Shanzhi et al. (2014). "A Vision of IoT: Applications, Challenges, and Opportunities With China Perspective". In: *IEEE Internet of Things Journal* 1.4, pp. 349–359. doi: 10.1109/JIOT.2014.2337336.

Chisholm, Malcolm (2004). "1 - What are Business Rules and Business Rules Engines?" In: *How to Build a Business Rules Engine*. Ed. by Malcolm Chisholm. The Morgan Kaufmann Series in Data Management Systems. San Francisco: Morgan Kaufmann, pp. 1–7. isbn: 978-1-55860-918-1. doi: https://doi.org/10.1016/B978-155860918-1/50002-8. url: https://www.sciencedirect.com/science/article/pii/B9781558609181500028.

Cisco (2017). *Cisco Survey Reveals Close to Three-Fourths of IoT Projects Are Failing*. Accessed: February 8, 2022.

Confluent (2022a). *Kafka Connect*. url: https://docs.confluent.io/platform/current/connect/index.html.

– (2022b). *Kafka Streams*. url: https://docs.confluent.io/platform/current/streams/index.html.

Craggs, Ian (2022). *MQTT vs AMQP for IoT*. url: https://www.hivemq.com/blog/mqtt-vs-amqp-for-iot/.

Cugola, Gianpaolo and Alessandro Margara (2012). "Processing flows of information: From data stream to complex event processing". In: *ACM Computing Surveys (CSUR)* 44.3, pp. 1–62.

Cumulocity (2022). *Real-time processing*. url: %5Curl%7Bhttps://cumulocity.com/guides/concepts/realtime/%7D.

Cypress (2022). *Cypress*. url: https://www.cypress.io/.

D. Hardt, Ed. (2012). *The OAuth 2.0 Authorization Framework*. url: https://datatracker.ietf.org/doc/html/rfc6749.

Date, Christopher John (1989). *A Guide to the SQL Standard*. Addison-Wesley Longman Publishing Co., Inc.

Dave Evans, Cisco (2011). *The Internet of Things: How the Next Evolution of the Internet Is Changing Everything*. Accessed: February 8, 2022.

DeCandia, Giuseppe et al. (2007). "Dynamo: Amazon's highly available key-value store". In: *ACM SIGOPS operating systems review* 41.6, pp. 205–220.

Deere, John (2020). *What is JDLink?* Accessed: February 8, 2022.

Dehdouh, Khaled et al. (2015). "Using the column oriented NoSQL model for implementing big data warehouses". In: *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)*. The Steering Committee of The World Congress in Computer Science, Computer , p. 469.

Dgraph (2022). *Dgraph*. url: %5Curl%7Bhttps://dgraph.io/%7D.

Dias, João Pedro, André Restivo, and Hugo Sereno Ferreira (2022). "Designing and constructing internet-of-Things systems: An overview of the ecosystem". In: *Internet of Things* 19, p. 100529. issn: 2542-6605. doi: https://doi.org/10.1016/j.iot.2022.100529. url: https://www.sciencedirect.com/science/article/pii/S2542660522000312.

Dobbelaere, Philippe and Kyumars Sheykh Esmaili (2017). "Kafka versus RabbitMQ: A Comparative Study of Two Industry Reference Publish/Subscribe Implementations: Industry Paper". In: *Proceedings of the 11th ACM International Conference on Distributed and Event-Based Systems*. DEBS '17. Barcelona, Spain: Association for Computing Machinery, pp. 227–238. isbn: 9781450350655. doi: 10.1145/3093742.3093908. url: https://doi.org/10.1145/3093742.3093908.

Docker (2022a). *Docker overview*. url: https://docs.docker.com/get-started/overview/.

– (2022b). *Overview of Docker Compose*. url: https://docs.docker.com/compose/.

Drools (2022). *Drools*. url: https://www.drools.org/.

Eeles, Peter (2005). "Capturing architectural requirements". In: *IBM Rational developer works*.

Eizinger, Thomas (2017). "API design in distributed systems: a comparison between GraphQL and REST". In: *University of Applied Sciences Technikum Wien-Degree Program Software Engineering*.

Elijah, Olakunle et al. (2018). "An Overview of Internet of Things (IoT) and Data Analytics in Agriculture: Benefits and Challenges". In: *IEEE Internet of Things Journal* 5.5, pp. 3758–3773. doi: 10.1109/JIOT.2018.2844296.

Elmasri, R et al. (2000). *Fundamentals of Database Systems*. Springer.

Ericson (2020). *Maritime Mesh Networks set to transform connectivity at sea*. Accessed: February 8, 2022.

Esri (2022). *ArcGIS*. url: https://www.arcgis.com/index.html.

Eugster, Patrick Th. et al. (June 2003). "The Many Faces of Publish/Subscribe". In: *ACM Comput. Surv.* 35.2, pp. 114–131. issn: 0360-0300. doi: 10.1145/857076.857078. url: https://doi.org/10.1145/857076.857078.

European Lighthouse Integrated Project (2013a). *Internet of Things - Architecture*. Accessed: April 14, 2022.

– (2013b). *Introduction to the Architectural Reference Model for the Internet of Things*. Accessed: April 20, 2022.

– (2013c). *The Internet of Things - Architecture*. Accessed: April 25, 2022.

Evans, E. (2014). *Domain-Driven Design Reference: Definitions and Pattern Summaries*. Dog Ear Publishing. isbn: 9781457501197.

Facebook (2022a). *CassandraDB*. url: https://cassandra.apache.org/.

– (2022b). *GraphQL*. url: https://graphql.org/.

– (2022c). *Jest Testing Framework*. url: https://jestjs.io/.

– (2022d). *React*. url: https://reactjs.org/.

Firouzi, Farshad et al. (2018). "Internet-of-Things and big data for smarter healthcare: From device to architecture, applications and analytics". In: *Future Generation Computer Systems* 78, pp. 583–586. issn: 0167-739X. doi: https://doi.org/10.1016/j.future.2017.09.016. url: https://www.sciencedirect.com/science/article/pii/S0167739X17319726.

flink.apache.org (2022). *Apache Flink*. Accessed: February 22, 2022.

Fowler, Martin and James Lewis (2014). *Microservices*. url: https://www.martinfowler.com/articles/microservices.html.

Garca, Laura et al. (2020). "IoT-based smart irrigation systems: An overview on the recent trends on sensors and IoT systems for irrigation in precision agriculture". In: *Sensors* 20.4, p. 1042.

Gartner (2022). *Magic Quadrant for Industrial IoT Platforms*. url: %5Curl%7Bhttps://www.gartner.com/doc/reprints?id=1-27IESWUW&ct=210922&st=sb%7D.

Gartner (n.d.). *Customer Identity and Access Management (CIAM)*. url: https://www.gartner.com/en/information-technology/glossary/customer-identity-access-management-ciam.

Gazis, Vangelis et al. (2015). "Short Paper: IoT: Challenges, projects, architectures". In: *2015 18th International Conference on Intelligence in Next Generation Networks*, pp. 145–147. doi: 10.1109/ICIN.2015.7073822.

Geers, Michael (2017). *Microfrontends*. url: https://micro-frontends.org/.

George, Lars (2011). *HBase: the definitive guide: random access to your planet-size data*. " O'Reilly Media, Inc."

German Electrical and Electronic Manufacturers' Association (2017a). *RAMI4.0 - a reference framework for digitalisation*. Accessed: June 26, 2022.

– (2017b). *Reference Architectural Model Industrie 4.0 - An introduction*. Accessed: June 26, 2022.

Gilchrist, Alasdair (2016). *Industry 4.0: The Industrial Internet of Things*. 1st. USA: Apress. isbn: 1484220463.

Gillis, Alexander S. (2020). *Definition: multi-tenancy*. url: %5Curl%7Bhttps://www.techtarget.com/whatis/definition/multi-tenancy%7D.

Goap, Amarendra et al. (2018). "An IoT based smart irrigation management system using Machine learning and open source technologies". In: *Computers and Electronics in Agriculture* 155, pp. 41–49. issn: 0168-1699. doi: https://doi.org/10.1016/j.compag.2018.09.040. url: https://www.sciencedirect.com/science/article/pii/S0168169918306987.

Google (2022a). *Angular*. url: https://angular.io/.

– (2022b). *Firestore*. url: %5Curl%7Bhttps://firebase.google.com/docs/firestore/%7D.

– (2022c). *Google Cloud IoT*. Accessed: February 8, 2022.

– (2022d). *Google Identity Platform*. url: https://cloud.google.com/identity-platform/.

– (2022e). *Google Maps*. url: https://mapsplatform.google.com/maps-products/.

– (2022f). *Protocol Buffers*. url: https://developers.google.com/protocol-buffers/.

Grafana (2022). *K6*. url: https://k6.io/.

Haller, Stephan et al. (2013). "A Domain Model for the Internet of Things". In: *2013 IEEE International Conference on Green Computing and Communications and IEEE Internet of Things and IEEE Cyber, Physical and Social Computing*, pp. 411–417. doi: 10.1109/GreenCom-iThings-CPSCom.2013.87.

Han, Jing et al. (2011). "Survey on NoSQL database". In: *2011 6th international conference on pervasive computing and applications*. IEEE, pp. 363–366.

Hankel, Martin and Bosch Rexroth (2015). "The reference architectural model industrie 4.0 (rami 4.0)". In: *ZVEI* 2.2, pp. 4–9.

Hardt, Red (2022). *Test Containers*. url: https://www.testcontainers.org/.

Harris, Chandler (n.d.). *Microservices vs. monolithic architecture*. url: https://www.atlassian.com/microservices/microservices-architecture/microservices-vs-monolith.

HashiCorp (2022). *Vault*. url: https://www.vaultproject.io/.

Hat, Red (2022). *Quarkus*. url: https://quarkus.io/.

HBase (2022). *HBase*. url: https://hbase.apache.org/.

Helium (2018a). *Helium Whitepaper*. url: %5Curl%7Bhttp://whitepaper.helium.com/%7D.

– (2018b). *Integrations*. url: %5Curl%7Bhttps://docs.helium.com/use-the-network/console/integrations%7D.
– (2022). *Helium Console*. url: https://www.helium.com/console.
Hughes, Karl (2017). *Why Testing Is Important for Distributed Software*. url: https://www.linuxfoundation.org/blog/testing-important-distributed-software/.
IBM (Oct. 2020a). *Three-Tier Architecture*. url: https://www.ibm.com/cloud/learn/three-tier-architecture.
– (Jan. 2020b). *What are Message Brokers?* url: https://www.ibm.com/cloud/learn/message-brokers.
– (2020c). *What is HBase?* Accessed: February 22, 2022.
– (Mar. 2021a). *Microservices*. url: https://www.ibm.com/cloud/learn/microservices#toc-anti-patte-uScI1WAE.
– (Apr. 2021b). *Rest API*. url: https://www.ibm.com/cloud/learn/rest-apis.
– (Apr. 2021c). *SOA (Service-Oriented Architecture)*. url: https://www.ibm.com/cloud/learn/soa.
– (2022a). *How does software testing work?* url: https://www.ibm.com/topics/software-testing.
– (2022b). *Node-RED*. url: %5Curl%7Bhttps://nodered.org/%7D.
IEEE (2020). "IEEE Standard for an Architectural Framework for the IoT". In: *IEEE Std 2413-2019*, pp. 1–269. doi: 10.1109/IEEESTD.2020.9032420.
Ilyushchenko, Vlad (2021). *How we achieved write speeds of 1.4 million rows per second*. Accessed: February 24, 2022.
Industries, The Things (2021a). *The Things Industries*. Accessed: February 8, 2022.
– (2021b). *The Things Stack Github Page*. Accessed: February 8, 2022.
Industry IoT Consortium (2019). *The Industrial Internet of Things Volume G1: Reference Architecture*. Accessed: May 20, 2022.
InfluxDB (2022a). *InfluxDB*. url: https://www.influxdata.com/.
– (2022b). *InfluxDB line protocol tutorial*. url: %5Curl%7Bhttps://docs.influxdata.com/influxdb/v1.8/write%5C_protocols/line%5C_protocol%5C_tutorial/%7D.
Jacobs, Mike and Craig Casey (2022). *What are Microservices?* url: https://docs.microsoft.com/en-us/devops/deliver/what-are-microservices.
Jatana, Nishtha et al. (2012). "A survey and comparison of relational and non-relational database". In: *International Journal of Engineering Research & Technology* 1.6, pp. 1–5.
Jennings, C. et al. (Aug. 2018). *Sensor Measurement Lists (SenML)*. RFC 8428. RFC Editor.
Jonas Bonér Dave Farley, Roland Kuhn and Martin Thompson (Sept. 2014). *The Reactive Manifesto*. url: https://www.reactivemanifesto.org/pdf/the-reactive-manifesto-2.0.pdf.
JUnit5 (2022). *JUnit5 Testing Framework*. url: https://junit.org/junit5/.
Kafka (2022). *Kafka Design: The Consumer*. url: https://kafka.apache.org/documentation/#theconsumer.
kafka.apache.org (2022). *Apache Kafka*. Accessed: February 22, 2022.
Kaviraju, R D (2021). *RAMI 4.0: Explained with example*. Accessed: June 26, 2022.
Khanna, Abhirup and Rishi Anand (2016). "IoT based smart parking system". In: *2016 International Conference on Internet of Things and Applications (IOTA)*, pp. 266–270. doi: 10.1109/IOTA.2016.7562735.
Kiran, Mariam et al. (2015). "Lambda architecture for cost-effective batch and speed big data processing". In: *2015 IEEE International Conference on Big Data (Big Data)*. IEEE, pp. 2785–2792.

Klishin, Michael (2022). *Fetching Individual Messages ("Pull API")*. url: https://www.rabbitmq.com/consumers.html.

Koo, Jahoon and Young-Gab Kim (2022). "Resource identifier interoperability among heterogeneous IoT platforms". In: *Journal of King Saud University - Computer and Information Sciences* 34.7, pp. 4191–4208. issn: 1319-1578. doi: https://doi.org/10.1016/j.jksuci.2022.05.003. url: https://www.sciencedirect.com/science/article/pii/S1319157822001525.

Kro, Srdjan, Boris Pokri, and Francois Carrez (2014). "Designing IoT architecture (s): A European perspective". In: *2014 IEEE world forum on internet of things (WF-IoT)*. IEEE, pp. 79–84.

Kumar, Abhishek and Jyotir Chatterjee (Mar. 2020). *Internet of Things Use Cases for the Healthcare Industry*. isbn: 978-3-030-37525-6.

Lakshman, Avinash and Prashant Malik (2010). "Cassandra: a decentralized structured storage system". In: *ACM SIGOPS Operating Systems Review* 44.2, pp. 35–40.

Lazidis, Apostolos, Konstantinos Tsakos, and Euripides G.M. Petrakis (2022). "Publish-Subscribe approaches for the IoT and the cloud: Functional and performance evaluation of open-source systems". In: *Internet of Things* 19, p. 100538. issn: 2542-6605. doi: https://doi.org/10.1016/j.iot.2022.100538. url: https://www.sciencedirect.com/science/article/pii/S2542660522000403.

Lewis, William E. (2008). *Software Testing and Continuous Quality Improvement*.

Lighttpd (2022). *Lighttpd*. url: https://www.lighttpd.net/.

Lin, Shi-Wan et al. (2017). *Architecture Alignment and Interoperability*. Accessed: June 26, 2022.

Listyorini, Tri and Robbi Rahim (2018). "A prototype fire detection implemented using the Internet of Things and fuzzy logic". In: *World Trans. Eng. Technol. Educ* 16.1, pp. 42–46.

López Peña, Miguel Angel and Isabel Muñoz Fernández (2019). "SAT-IoT: An Architectural Model for a High-Performance Fog/Edge/Cloud IoT Platform". In: *2019 IEEE 5th World Forum on Internet of Things (WF-IoT)*, pp. 633–638. doi: 10.1109/WF-IoT.2019.8767282.

Luo, Xi et al. (2021). "A scalable rule engine system for trigger-action application in large-scale IoT environment". In: *Computer Communications* 177, pp. 220–229. issn: 0140-3664. doi: https://doi.org/10.1016/j.comcom.2021.06.016. url: https://www.sciencedirect.com/science/article/pii/S0140366421002401.

Lynn, Theo et al. (2020). "The Internet of Things: Definitions, Key Concepts, and Reference Architectures". In: *The Cloud-to-Thing Continuum: Opportunities and Challenges in Cloud, Fog and Edge Computing*. Ed. by Theo Lynn et al. Cham: Springer International Publishing, pp. 1–22. isbn: 978-3-030-41110-7. doi: 10.1007/978-3-030-41110-7_1. url: https://doi.org/10.1007/978-3-030-41110-7_1.

Mapbox (2022). *Mapbox GL JS*. url: https://www.mapbox.com/mapbox-gljs.

Marcu, Ioana et al. (2020). "Arrowhead technology for digitalization and automation solution: Smart cities and smart agriculture". In: *Sensors* 20.5, p. 1464.

Martin, R.C. (2003). *Agile Software Development: Principles, Patterns, and Practices*. Alan Apt series. Pearson Education. isbn: 9780135974445. url: https://books.google.pt/books?id=0HYhAQAAIAAJ.

Microsoft (2018). *Microsoft Azure IoT Reference Architecture*. Accessed: June 26, 2022.

– (2022a). *Azure IoT*. Accessed: February 8, 2022.

– (2022b). *Azure IoT Hub*. Accessed: February 25, 2022.

– (2022c). *Read device-to-cloud messages from the built-in endpoint*. Accessed: April 3, 2022.
– (2022d). *Typescript*. url: https://www.typescriptlang.org/.
Milenkovic, Milan (2020). "Chapter 8: IoT Platforms". In: *Internet of Things: Concepts and System Design*. Cham: Springer International Publishing, pp. 247–265. isbn: 978-3-030-41346-0. doi: 10.1007/978-3-030-41346-0_8. url: https://doi.org/10.1007/978-3-030-41346-0%5C_8.
Miller, Justin J (2013). "Graph database applications and concepts with Neo4j". In: *Proceedings of the southern association for information systems conference, Atlanta, GA, USA*. Vol. 2324. 36.
Miloslavskaya, Natalia and Alexander Tolstoy (2016). "Big data, fast data and data lake concepts". In: *Procedia Computer Science* 88, pp. 300–305.
Mockito (2022). *Mockito Testing Framework*. url: https://site.mockito.org/.
Mohammed, Abrar Ahmed et al. (2021). "Computer Vision Based Autonomous Fire Detection and IoT Based Fire Response System". In: *Proceedings of International Conference on Communication and Computational Technologies*. Springer, pp. 551–560.
MongoDB (2022). *MongoDB*. url: https://www.mongodb.com/.
Morrison, J Paul (1994). "Flow-based programming". In: *Proc. 1st International Workshop on Software Engineering for Parallel and Distributed Systems*, pp. 25–29.
Morrison, J.P. (2010). *Flow-based Programming: A New Approach to Application Development*. J.P. Morrison Enterprises. isbn: 9781451542325. url: https://books.google.pt/books?id=R06TSQAACAAJ.
Mozzila (2022). *Javascript*. url: https://developer.mozilla.org/en-US/docs/Web/JavaScript.
MQTT (2022). *MQTT: The Standard for IoT Messaging*. url: https://mqtt.org/.
Muller, Gerrit (2008). "A reference architecture primer". In: *Eindhoven Univ. of Techn., Eindhoven, White paper*.
MySQL (2022). *MySQL*. url: https://www.mysql.com/.
Nadiminti, Krishna, Marcos Dias De Assunçao, and Rajkumar Buyya (2006). "Distributed systems and recent innovations: Challenges and benefits". In: *InfoNet Magazine* 16.3, pp. 1–5.
Naqvi, Syeda Noor Zehra, Sofia Yfantidou, and Esteban Zimányi (2017). "Time series databases and influxdb". In: *Studienarbeit, Université Libre de Bruxelles* 12.
Neo4j (2022). *Neo4j*. url: %5Curl%7Bhttps://neo4j.com/%7D.
Nginx (2022). *Nginx*. url: https://nginx.org/en/.
Nish Anil, Tarun Jain and Miguel Veloso (2022a). *Asynchronous message-based communication*. url: https://docs.microsoft.com/en-us/dotnet/architecture/microservices/architect-microservice-container-applications/asynchronous-message-based-communication.
– (2022b). *Communication in a microservice architecture*. url: https://docs.microsoft.com/en-us/dotnet/architecture/microservices/architect-microservice-container-applications/communication-in-microservice-architecture.
Nieti, Sandro, Nedjib Djilali, et al. (2019). "Smart technologies for promotion of energy efficiency, utilization of sustainable resources and waste management". In: *Journal of cleaner production* 231, pp. 565–591.
Nieti, Sandro, Petar oli, et al. (2020). "Internet of Things (IoT): Opportunities, issues and challenges towards a smart and sustainable future". In: *Journal of Cleaner Production* 274, p. 122877. issn: 0959-6526. doi:

https://doi.org/10.1016/j.jclepro.2020.122877. url: https://www.sciencedirect.com/science/article/pii/S095965262032922X.

Noura, Mahda, Mohammed Atiquzzaman, and Martin Gaedke (2019). "Interoperability in internet of things: Taxonomies and open challenges". In: *Mobile networks and applications* 24.3, pp. 796–809.

Nx (2022). *Nx*. url: https://nx.dev/.

Obaideen, Khaled et al. (2022). "An overview of smart irrigation systems using IoT". In: *Energy Nexus* 7, p. 100124. issn: 2772-4271. doi: https://doi.org/10.1016/j.nexus.2022.100124. url: https://www.sciencedirect.com/science/article/pii/S2772427122000791.

Okta (2022). *Okta Customer Identity*. url: https://www.okta.com/solutions/secure-ciam/.

OpenID (2014). *OpenID Connect*. url: https://openid.net/connect/.

Oracle (2022a). *GraalVM*. url: https://www.graalvm.org/.

– (2022b). *Introduction to GraalVM*. url: https://www.graalvm.org/22.2/docs/introduction/.

– (2022c). *Oracle database*. url: %5Curl%7Bhttps://www.oracle.com/database/%7D.

OWASP (2021). *TOP 10*. Accessed: February 14, 2022.

Palermo, Jeffrey (2008). *The Onion Architecture*. url: https://jeffreypalermo.com/2008/07/the-onion-architecture-part-1/.

Paradigm, Visual (2020). *Requirement Analysis Techniques*. Accessed: February 14, 2022.

Pierleoni, Paola et al. (2020). "Amazon, Google and Microsoft Solutions for IoT: Architectures and a Performance Comparison". In: *IEEE Access* 8, pp. 5455–5470. doi: 10.1109/ACCESS.2019.2961511.

Pittet, Sten (2022). *The different types of software testing*. url: https://www.atlassian.com/continuous-delivery/software-testing/types-of-software-testing.

Pokorny, Jaroslav (2011). "NoSQL databases: a step to database scalability in web environment". In: *Proceedings of the 13th International Conference on Information Integration and Web-based Applications and Services*, pp. 278–283.

PostgresSQL (2022a). *Array Functions and Operators*. url: https://www.postgresql.org/docs/current/arrays.html.

– (2022b). *Array Functions and Operators*. url: https://www.postgresql.org/docs/current/functions-array.html.

– (2022c). *PostgresSQL*. url: https://www.postgresql.org/.

Powell, Ron (Oct. 2021). *SOA vs microservices: going beyond the monolith*. url: https://circleci.com/blog/soa-vs-microservices/.

Preston-Werner, Tom (June 2011). *Semantic Versioning 2.0.0*. [Online; accessed 30. Jun. 2022]. url: https://semver.org/.

Pulsar, Apache (2022a). *Pulsar*. url: https://pulsar.apache.org/docs/2.6.0/pulsar-2.0.

– (2022b). *Pulsar - Multi-topic subscriptions*. url: https://pulsar.apache.org/docs/2.6.0/concepts-messaging#multi-topic-subscriptions.

Al-Qaseemi, Sarah A et al. (2016). "IoT architecture challenges and issues: Lack of standardization". In: *2016 Future technologies conference (FTC)*. IEEE, pp. 731–738.

questdb.io (2022). *QuestDB*. url: https://questdb.io.

Qureshi, Waqar et al. (Apr. 2015). "QuickBlaze: Early Fire Detection Using a Combined Video Processing Approach". In: *Fire Technology* 52. doi: 10.1007/s10694-015-0489-7.

Ray, Partha Pratim (2016). "A survey of IoT cloud platforms". In: *Future Computing and Informatics Journal* 1.1, pp. 35–46. issn: 2314-7288. doi: https://doi.org/10.1016/j.fcij.2017.02.001. url: https://www.sciencedirect.com/science/article/pii/S2314728816300149.

RedHat (2022). *What is CI/CD?* url: https://www.redhat.com/en/topics/devops/what-is-ci-cd.

– (n.d.). *Drools Testimonials and Case Studies*. url: https://www.drools.org/learn/testimonialsAndCaseStudies.html.

redhat (2021). *Quarkus Reactive Architecture*. url: https://quarkus.io/guides/quarkus-reactive-architecture.

Redis (2022). *Redis*. url: %5Curl%7Bhttps://redis.io/%7D.

Reselman, Bob (Mar. 2021). *The pros and cons of the Pub-Sub architecture pattern*. url: https://www.redhat.com/architect/pub-sub-pros-and-cons.

Richards, M. (2015). *Microservices Vs. Service-oriented Architecture*. O'Reilly Media. isbn: 9781491975657. url: https://books.google.pt/books?id=Bd5mAQAACAAJ.

Richards, M. and N. Ford (2020). *Fundamentals of Software Architecture: An Engineering Approach*. O'Reilly Media. Chap. 6, pp. 86–87. isbn: 9781492043409.

Richardson, Chris (2021a). *Pattern: Microservice Architecture*. url: https://microservices.io/patterns/microservices.html.

– (2021b). *Pattern: Monolithic Architecture*. url: https://microservices.io/patterns/monolithic.html.

– (2022a). *Database per service Pattern*. url: https://microservices.io/patterns/data/database-per-service.html.

– (2022b). *Event sourcing Pattern*. url: https://microservices.io/patterns/data/event-sourcing.html.

– (2022c). *Externalized configuration Pattern*. url: https://microservices.io/patterns/externalized-configuration.html.

Rogerson, Steve (2021). *Safecube uses Microsoft Azure for asset tracking*. Accessed: February 8, 2022.

Roy, Sanku Kumar et al. (2021). "AgriSens: IoT-Based Dynamic Irrigation Scheduling System for Water Management of Irrigated Crops". In: *IEEE Internet of Things Journal* 8.6, pp. 5023–5030. doi: 10.1109/JIOT.2020.3036126.

Saeed, Faisal et al. (2018). "IoT-based intelligent modeling of smart home environment for fire prevention and safety". In: *Journal of Sensor and Actuator Networks* 7.1, p. 11.

Safecube (2021). *Locatrack Solution: Locate your assets, Manage your fleet*. Accessed: February 8, 2022.

Sanjay Aiyagari, Matthew Arrott (2008). *Advanced Message Queuing Protocol Specification, Version 0-9-1*. url: https://www.rabbitmq.com/resources/specs/amqp0-9-1.pdf.

Schwaber, Ken (1997). "Scrum development process". In: *Business object design and implementation*. Springer, pp. 117–134.

Services, Amazon Web (2022). *Rules for AWS IoT*. url: %5Curl%7Bhttps://docs.aws.amazon.com/iot/latest/developerguide/iot-rules.html%7D.

Shyam, Gopal Kirshna, Sunilkumar S. Manvi, and Priyanka Bharti (2017). "Smart waste management using Internet-of-Things (IoT)". In: *2017 2nd International Conference on*

Computing and Communications Technologies (ICCCT), pp. 199–203. doi: 10.1109/ICCCT2.2017.7972276.

SierraWireless (2017). *Build Efficient and Reliable IoT Smart Meters for Utilities*. Accessed: February 8, 2022.

Silva, Bhagya Nathali, Murad Khan, and Kijun Han (2018). "Towards sustainable smart cities: A review of trends, architectures, components, and open challenges in smart cities". In: *Sustainable Cities and Society* 38, pp. 697–713. issn: 2210-6707. doi: https://doi.org/10.1016/j.scs.2018.01.053. url: https://www.sciencedirect.com/science/article/pii/S2210670717311125.

Silva, Margarida et al. (2020). "Visually-Defined Real-Time Orchestration of IoT Systems". In: *MobiQuitous 2020 - 17th EAI International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services*. MobiQuitous '20. Darmstadt, Germany: Association for Computing Machinery, pp. 225–235. isbn: 9781450388405. doi: 10.1145/3448891.3448938. url: https://doi.org/10.1145/3448891.3448938.

Slee, Mark, Aditya Agarwal, and Marc Kwiatkowski (2007). "Thrift". In: *Facebook white paper* 5.8, p. 127.

SmartDrive (2018a). *SR4: Fuelling The Future of Intelligent Transportation*. Accessed: February 8, 2022.

– (2018b). *Transportation Intelligence Platform*. Accessed: February 8, 2022.

spark.apache.org (2022). *Apache Spark*. Accessed: February 22, 2022.

Spring (2022). *Spring Native documentation*. url: https://docs.spring.io/spring-native/docs/current/reference/htmlsingle/.

Statista, Arne von See (2021). *Number of Internet of Things (IoT) connected devices worldwide from 2019 to 2030*. Accessed: February 6, 2022.

storm.apache.org (2022). *Apache Storm*. Accessed: February 22, 2022.

Sumaray, Audie and S Kami Makki (2012). "A comparison of data serialization formats for optimal efficiency on a mobile platform". In: *Proceedings of the 6th international conference on ubiquitous information management and communication*, pp. 1–6.

T-Mobile (2021). *IoT: How It's Making Small Businesses More Efficient*. Accessed: February 8, 2022.

TagoIO (2022). *TagoIO*. url: https://tago.io/.

Tapscott, Don and Anthony D. Williams (Dec. 2006). *Wikinomics: How Mass Collaboration Changes Everything*. New York, NY: Portfolio. isbn: 978-1591841388.

ThingsBoard (2022a). *Rule Engine Overview*. url: %5Curl%7Bhttps://thingsboard.io/docs/user-guide/rule-engine-2-0/overview/%7D.

– (2022b). *ThingsBoard*. url: https://thingsboard.io/.

TracPac (2022). *TrackPac: Track everything that matters*. Accessed: July 13, 2022.

Tracy, Phillip (2017). *The top 5 industrial IoT use cases*. Accessed: February 8, 2022.

Urquhart, J. (2021). *Flow Architectures The Future of Streaming and Event-Driven Integration*. O'Reilly Media. isbn: 9781492075868.

Van Lamsweerde, Axel (2009). *Requirements engineering: From system goals to UML models to software*. Vol. 10. Chichester, UK: John Wiley & Sons.

Varga, Pal et al. (2017). "Making system of systems interoperable–The core components of the arrowhead framework". In: *Journal of Network and Computer Applications* 81, pp. 85–95.

Verizon (2022). *Fleet management software*. Accessed: February 8, 2022.

VMWare (2022a). *AMQP 0-9-1 Model Explained*. url: https://www.rabbitmq.com/tutorials/amqp-concepts.html.

– (2022b). *RabbitMQ*. url: https://www.rabbitmq.com/.

– (2022c). *Spring Boot*. url: https://spring.io/projects/spring-boot.
Waylay (2020). *A guide to: Rules Engines*. url: %5Curl%7Bhttps://query.prod.cms.rt.microsoft.com/cms/api/am/binary/RE36M30%7D.
Webpack (2022). *Module Federation*. url: https://webpack.js.org/concepts/module-federation/.
Wegner, Philipp (2020). *The top 10 Smart City use cases that are being prioritized now*. Accessed: February 8, 2022.
Weyrich, Michael and Christof Ebert (2015). "Reference architectures for the internet of things". In: *IEEE Software* 33.1, pp. 112–116.
Winslow, Robert (2021). *Time Series Benchmark Suite (TSBS)*. Accessed: February 24, 2022.
WSO2 (2015). *A Reference Architecture for the Internet of Things*. Accessed: May 20, 2022.
Wu, Lesong, Lan Chen, and Xiaoran Hao (2021). "Multi-sensor data fusion algorithm for indoor fire early warning based on BP neural network". In: *Information* 12.2, p. 59.
Yoon, Ayoung et al. (2017). "In between data sharing and reuse: Shareability, availability and reusability in diverse contexts". In: *Proceedings of the Association for Information Science and Technology* 54.1, pp. 606–609. doi: https://doi.org/10.1002/pra2.2017.14505401085. eprint: https://asistdl.onlinelibrary.wiley.com/doi/pdf/10.1002/pra2.2017.14505401085. url: https://asistdl.onlinelibrary.wiley.com/doi/abs/10.1002/pra2.2017.14505401085.
Zaniolo, Carlo and Miachel A Meklanoff (1981). "On the design of relational database schemata". In: *ACM Transactions on Database Systems (TODS)* 6.1, pp. 1–47.

Appendix A

Data Unit - Shared Model Schema

This schema represents the Shared Model Schema of a Data Unit as of *iot-core* package version *0.1.20*.

Due to the lack of standards, this model and the represented measures are possibly incomplete. As the IoT landscape evolves and defines stricter and well-known standards for sensor measurements and device parameters representation, the *iot-core* package must adhere to them and be able to translate between the supported representations.

The version that is currently in development, *0.2.0*, is planed to support Jennings et al. 2018. The reason behind the delayed support for it, steams from the lack of open-source *Java* libraries related to *SenML*, the unfamiliarity of the author with the standard till early April 2022, and the fast paced development of the solution required to meet costumers' needs.

```
{
  "dataId": "[uuid]",
  "reportedAt": "[long]",
  "device": {
    "id": "[uuid]",
    "name": "[string]",
    "downlink": "[string]",
    "records": [{
      "label": "[string]",
      "content": "[string]"
    }],
    "domains":  ["[uuid]"],
    "commands": {
      "[int]": [{
        "id": "[uuid]",
        "name": "[string]",
        "payload": "[base64 string]",
        "port": "[int]"
      }]
    }
  },
  "measures": {
    "[int]": {
      "airHumidity": {
        "gramsPerCubicMeter": "[float]",
        "relativePercentage": "[float]"
      },
```

```
      "airPressure": {  "hectoPascal": "[float]"  },
      "aqi": {  "value": "[float]"  },
      "battery": {
        "percentage": "[float]",
        "volts": "[float]",
        "maxVolts": "[float]",
        "minVolts": "[float]"
      },
      "co2": {  "ppm": "[float]"  },
      "co": {  "ppm": "[float]"  },
      "distance": {
        "millimeters": "[float]",
        "maxMillimeters": "[float]",
        "minMillimeters": "[float]"
      },
      "gps": {
        "latitude": "[double]",
        "longitude": "[double]",
        "altitude": "[float]"
      },
      "illuminance": {  "lux": "[float]"  },
      "motion": {  "value": "[ACTIVE, INACTIVE or UNKNOWN]"  },
      "nh3": {  "ppm": "[float]"  },
      "no2": {  "ppm": "[float]"  },
      "o3": {  "ppm": "[float]"  },
      "occupation": {  "percentage": "[float]"  },
      "ph": {  "value": "[float]"  },
      "pm2_5": {  "microGramsPerCubicMeter": "[float]"  },
      "pm10": {  "microGramsPerCubicMeter": "[float]"  },
      "soilConductivity": {
        "microSiemensPerCentimeter": "[float]"
      },
      "soilMoisture": {  "relativePercentage": "[float]"  },
      "temperature": {  "celsius": "[float]"  },
      "trigger": {  "value": "[boolean]"  },
      "velocity": {  "kilometerPerHour": "[float]"  },
      "voc": {  "ppm": "[float]"  },
      "waterPressure": {  "bar": "[float]"  }
    }
  }
}
```

Listing A.1: Data Unit - Shared Model Schema

Appendix B

Container Level - Logical View

This logical view represents a system when all external services are included in the platform, **Sensae Console**. It corresponds to the system used currently in production.

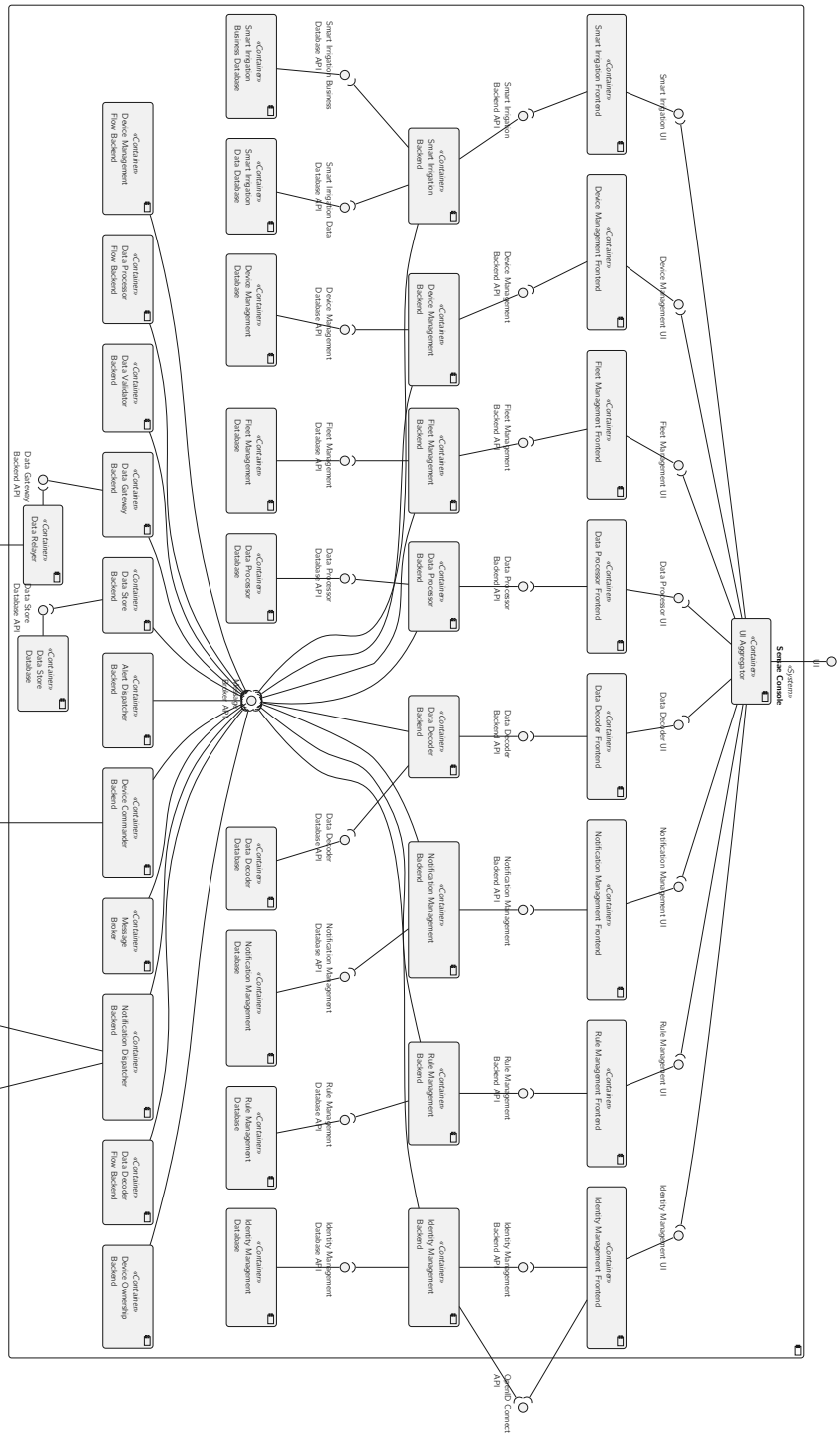


Figure B.1: Complete Solution - Container Level - Logical View Diagram

Appendix C

C4 Level 3 - Components

The component level describes the internals of a specific container. A container is made up of a number of components, each with well-defined responsibilities. In the following diagrams the dependencies between the various components will also be presented.

Most developed containers share the same architecture and will therefore be addressed as groups of containers.

The physical view will not be presented since all relevant details have been addressed above.

C.1 Components Level - Logical View

The architectures used in the various developed containers can be condensate into 3 types with minor variations:

- **Frontend Architecture**: used on all Configuration scope frontend containers;
- **Configuration Backend Architecture**: used on all Configuration scope backend containers;
- **Data Flow Architecture**: used on most Data Flow scope containers.

Starting with the Frontend Architecture used, it was decided to maintain two distinct domains (Model and DTOS) in order to meet the Single Responsability Principle (SRP) (high cohesion) and to lower the coupling between the information displayed in the UI and the data sent/received by the container. This segmentation led to the addition of the Mapper component, which has the responsibility of converting the data (DTOS component) into information (Model component) and vice-versa. The Auth component indicates what backend resources the user has access to, by decoding the *access token*, and the Utils component has several methods commonly used to process backend requests. These two components are reused in all frontend containers, including the ones related to the External Services.

As an example, the logical view of the Data Decoder Frontend is presented in Figure C.1.

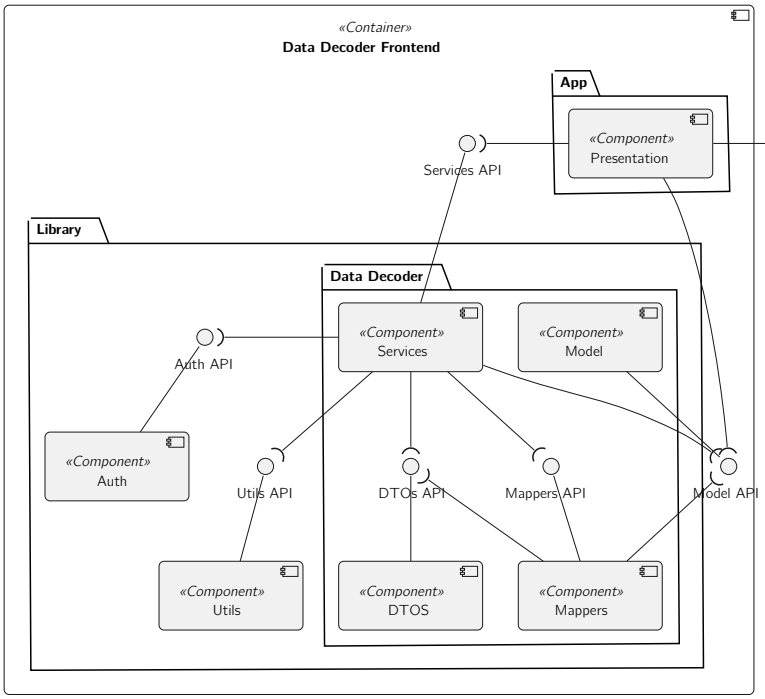


Figure C.1: Data Decoder Frontend - Component Level - Logical View Diagram

This architecture is used on the containers: (i) Device Management Frontend, (ii) Data Decoder Frontend, (iii) Data Processor Frontend, (iv) Rule Management Frontend. The UI Aggregator has a simpler architecture them the other frontend containers, it is comprised by a Presentation component that depends on the Auth component to handle user authentication and authorization.

Next, the Configuration Backend Architecture is discussed. It is based on the Onion Architecture, an architecture pattern that "emphasizes separation of concerns throughout the system" and "leads to more maintainable applications" (Palermo 2008).

As an example the logical view of the Device Management Backend is presented in Figure C.2.

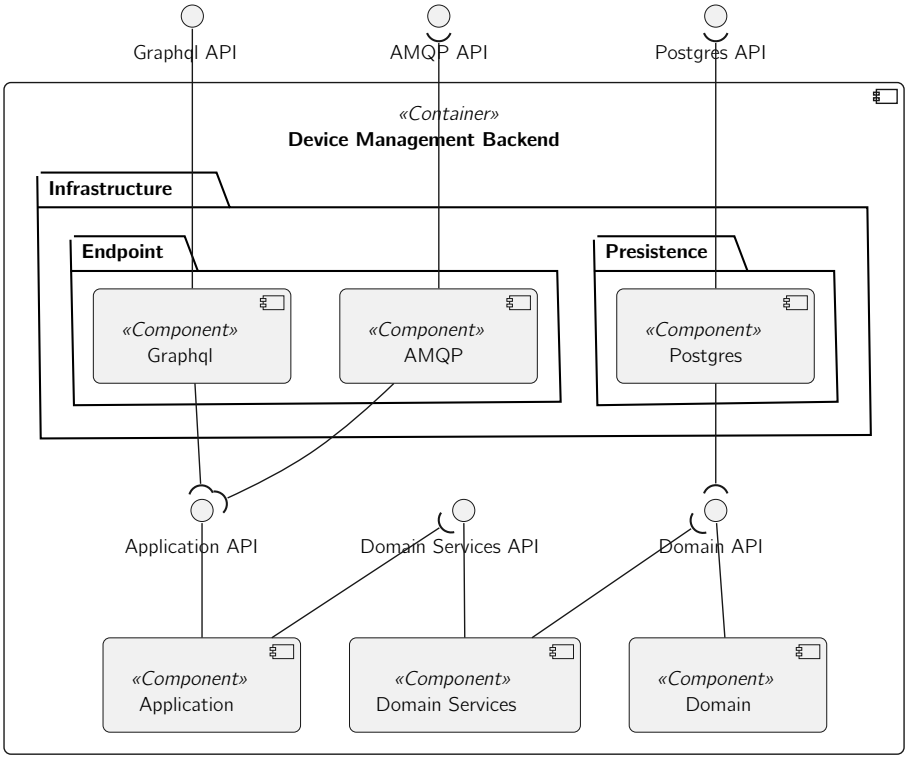


Figure C.2: Device Management Backend - Component Level - Logical View Diagram

This architecture is used on the containers: (i) Device Management Backend, (ii) Data Decoder Backend, (iii) Data Processor Backend, (iv) Rule Management Backend, (v) Identity Management Backend.

The following table, Table C.1, discusses each component responsibilities.

Table C.1: Configuration Backend components responsibilities

Component	Responsibilities
Infrastructure	- Enclose components that manage the Input/Output operations required by the container.
Endpoint	- Enclose components that are used by external containers to interact with the container.
AMQP	- Define how to consume and publish events in the Message Broker; - Delegate the handling of events received to specific Application processes.
GraphQl	- Define the interface to be consumed by the frontend and external Systems; - Delegate external requests made to specific Application processes.
Persistence	- Enclose components that interface with containers responsible for persisting data.
Postgres	- Interact with a database to persist and query data.
Application	- Represent the application processes; - Ensure the propagation of events related to the process in question, requiring this responsibility to AMQP; - Ensure the execution of the process in question, requiring this responsibility to Domain Services; - Enforce user authorization.
Domain Services	- Represent business processes; - Interact with the Domain; - Ensure the persistence of the data in question, requiring this responsibility to the Persistence.
Domain	- Represent de business rules and concepts; - Manage the system information.

Finally the architecture used in containers related to the Data Flow Scope is presented. It is based on a simplified version of the Onion Architecture since the intrinsic processes of these containers are much simpler.

As an example the logical view of the Device Ownership Backend is presented in Figure C.3.

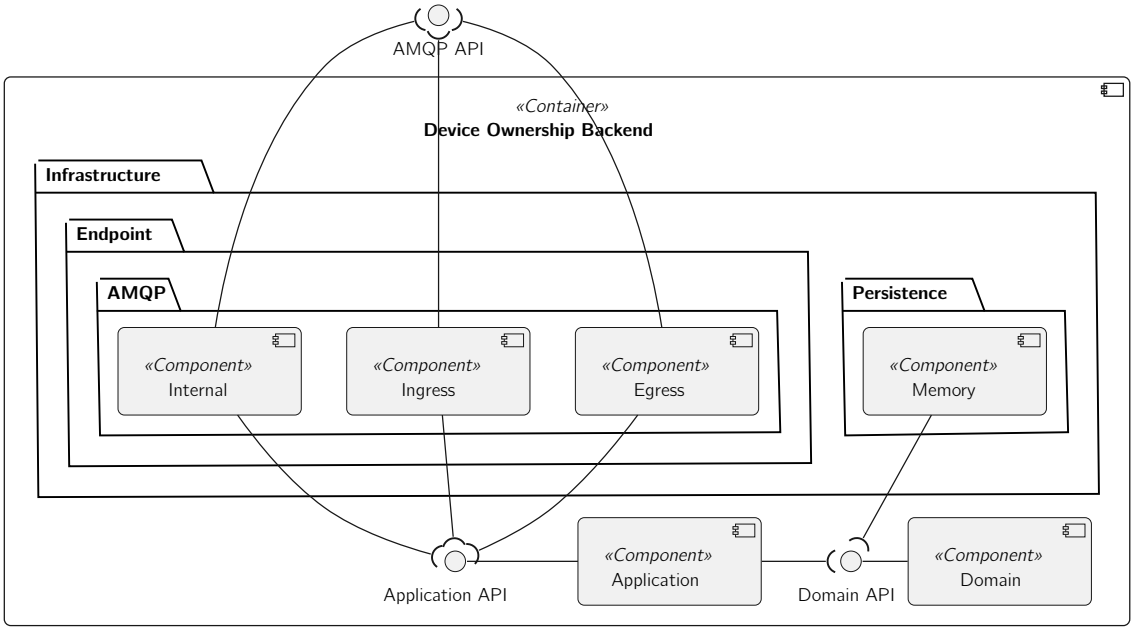


Figure C.3: Device Ownership Backend - Component Level - Logical View Diagram

This architecture is used on the containers: (i) Device Management Flow Backend, (ii) Data Decoder Flow Backend, (iii) Data Processor Flow Backend, (iv) Device Ownership Backend. The responsibilities of the components inside AMQP are:

- Internal: responsible for communicating with the system via internal topic;
- Ingress: responsible for consuming events/messages coming from data, alert or command topics;
- Egress: responsible for publishing events/messages to the data or alert topics.

The Memory component is responsible for caching unhandled data units and other information relevant for each context. This component is not present in Data Validator Backend and Alert Dispatcher Backend since they don't need to store context information to function.

The Data Gateway, Device Commander and Data Store backend containers have architectures that derive from this one and can be consulted in Appendix D.

C.2 Components Level - Process View

In this section some internal process deemed relevant are presented through sequence diagrams in order to familiarize the reader with the interactions that occur between components inside a container.

The internal processes that will be evaluated are:

- Process Data Unit in Device Management Flow Backend;
- Deploy Draft Rule Scenarios in Rule Management Backend.

This processes have been chosen in order to introduce the reader to specific operations not yet explored in this chapter.

The first process to explore is meant to clarify how a Data Unit sent by a Controller (devices that collect and report measures of various sensors) is processed inside the Device Management Flow Backend. As explained in the Device Management Section, Data Units sent by a Controller are partitioned into various Data Units. The following diagram, Figure C.4, details this process.

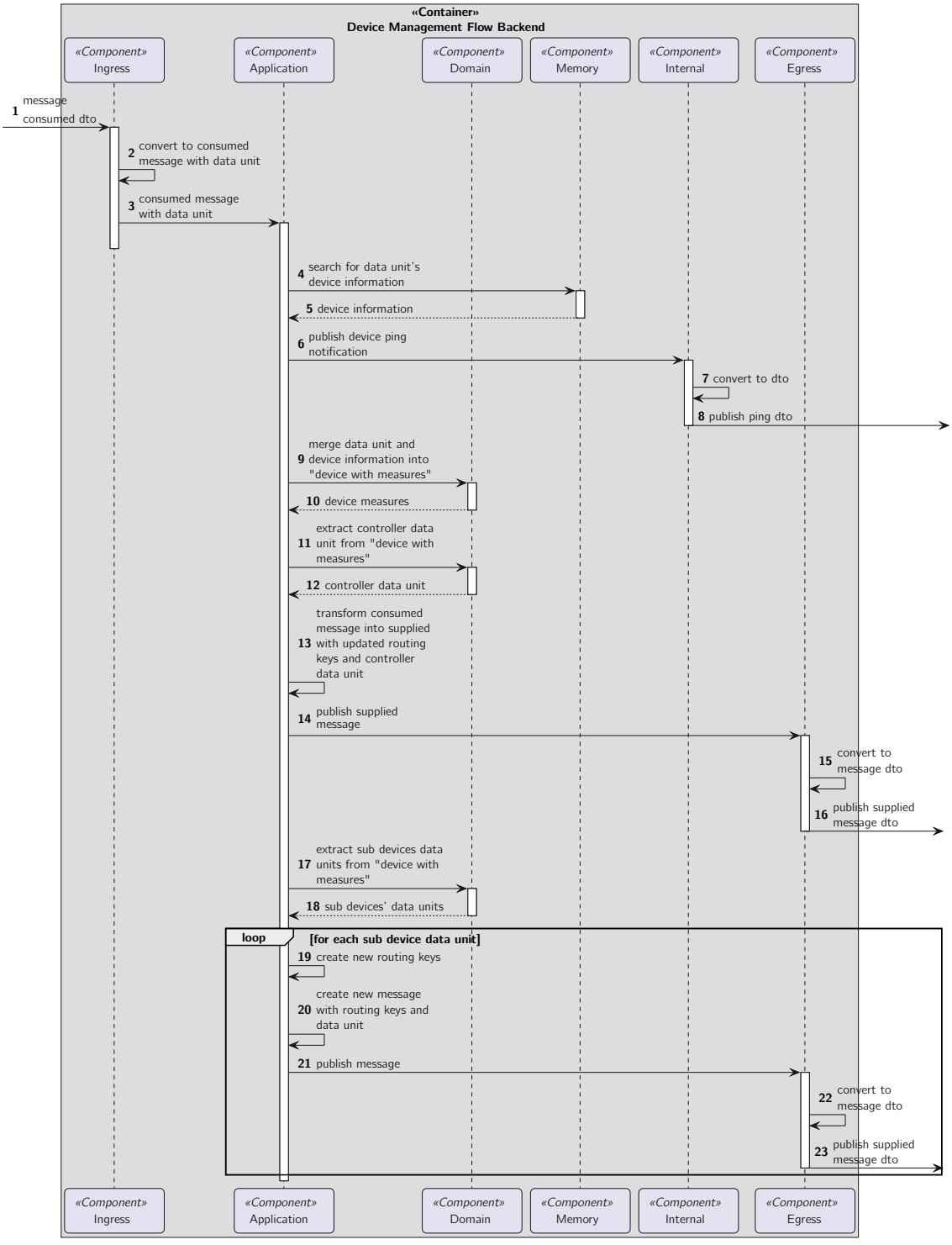


Figure C.4: Process Data Unit in Device Management Flow Backend - Component Level - Process View Diagram

As presented in the diagram:

- As soon as the message dto arrives, it is mapped to the *iot-core* data unit model - step **2** - this model is used inside every Data Flow container. Before publishing the

data unit it is mapped to the dto once again - step **15** and **22**. This conversion happens with any other event published and consumed in the system;

- If the device information is found, a *ping* notification for that device is sent - steps **6** to **8**, otherwise an *unknown* notification would be sent and the container would store the data unit in the Memory component and process it when possible;
- For each sub device of the controller, a new data unit with that device measures is published in the system - steps **19** to **23**;

Next, the process of deploying draft rule scenarios is described. Draft scenarios exist since adding, removing or changing a rule scenario in Alert Dispatcher Backend requires the entire data set to be removed. This procedure can lead to alerts not being dispatched. The next diagram, Figure C.5, tackles this concern.

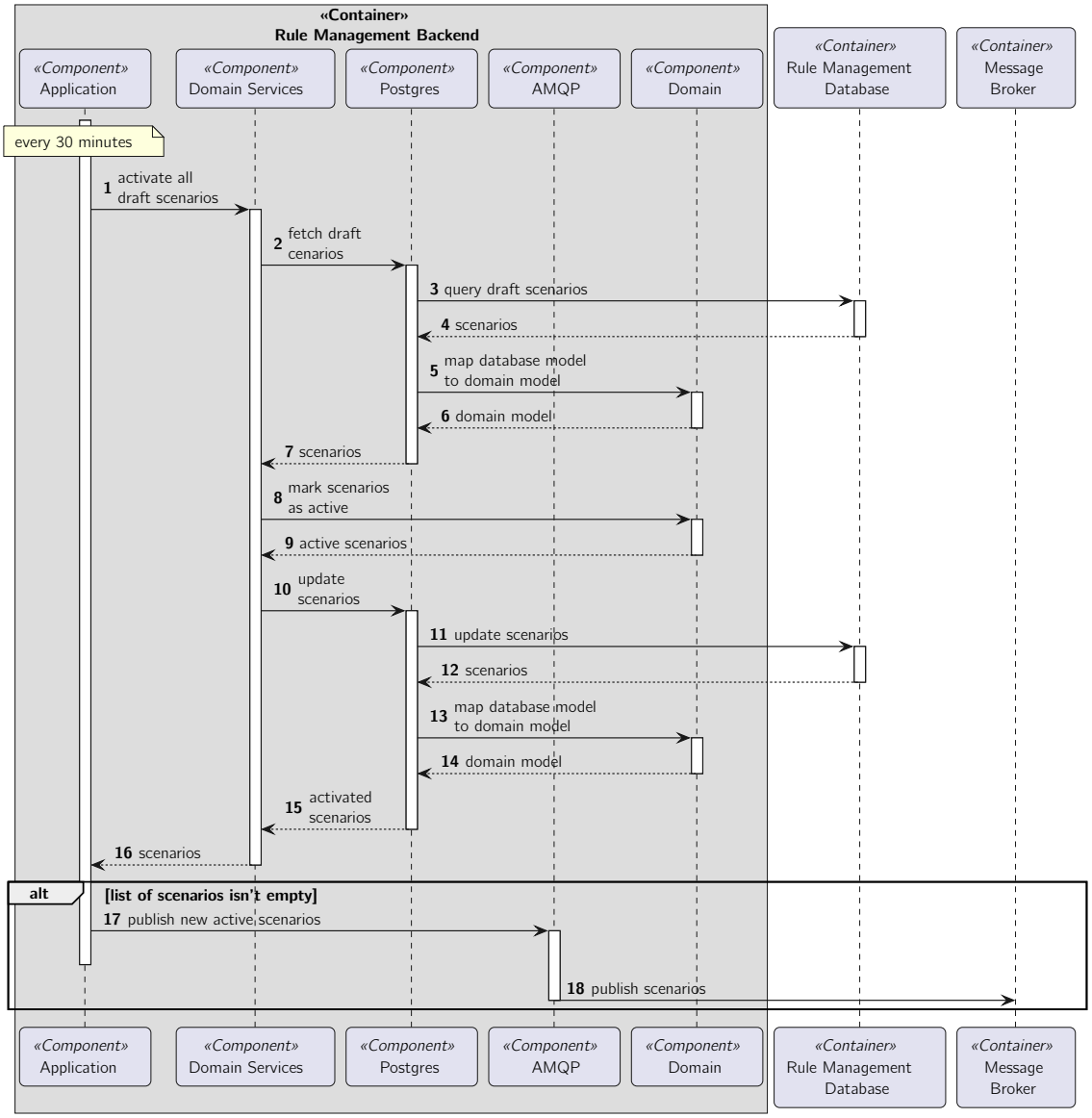


Figure C.5: Deploy Draft Rule Scenarios - Component Level - Process View Diagram

As seen in the diagram, to mitigate the number of lost alerts, new rule scenarios are published at best every 30 minutes - step **1** - and only if any change was made - step **17** and **18**.

C.3 Components Level - Implementation View

The implementation view of each container can also be condensate in the same 3 distinct types presented in the Section Components Level - Logical View.

The next diagrams, Figure C.6, Figure C.7 and Figure C.8 describe this view at the components level.

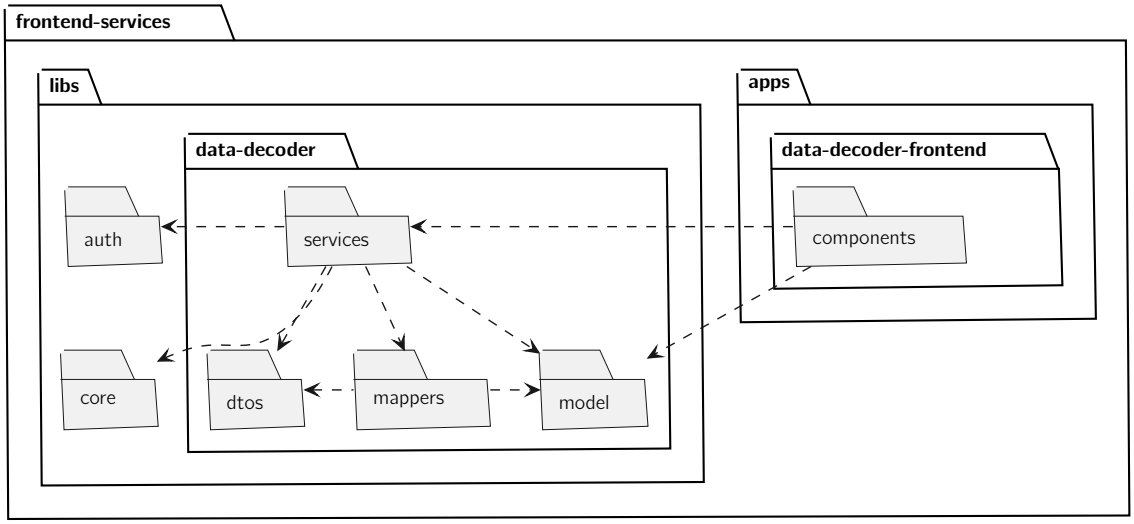


Figure C.6: Data Decoder Frontend - Component Level - Implementation View Diagram

The packages presented correspond to the components described in the logical view (Figure C.1). Since the names given in both views are different, the following list maps the logical view into the implementation view:

- *components* package corresponds to the *Presentation* component;
- *auth* package corresponds to the *Auth* component;
- *core* package corresponds to the *Utils* component;
- *dtos* package corresponds to the *DTOS* component;
- *mappers* package corresponds to the *Mappers* component;
- *model* package corresponds to the *Model* component;
- *services* package corresponds to the *Services* component.

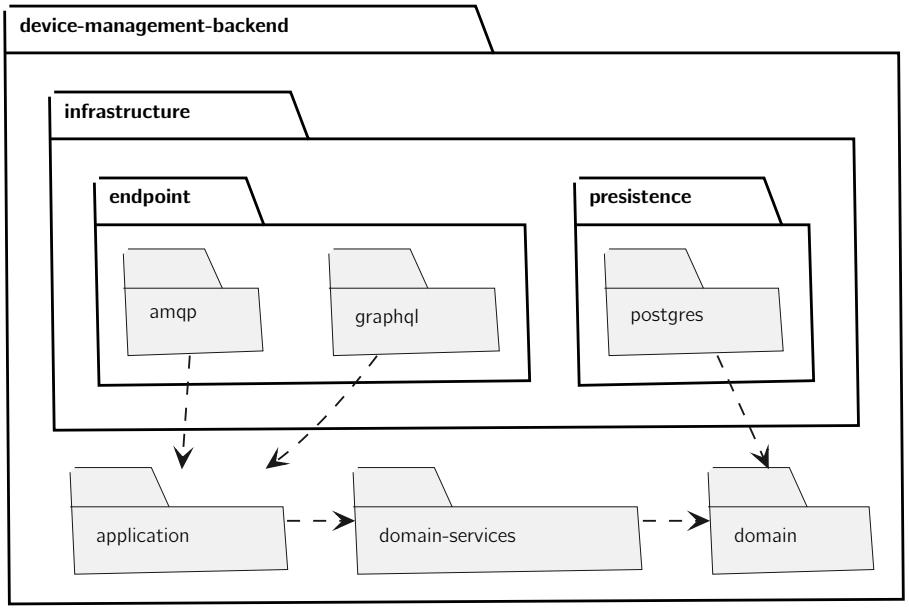


Figure C.7: Device Management Backend - Component Level - Implementation View Diagram

The packages presented correspond to the components described in the logical view (Figure C.2). The names given in both views differ only on the case used.

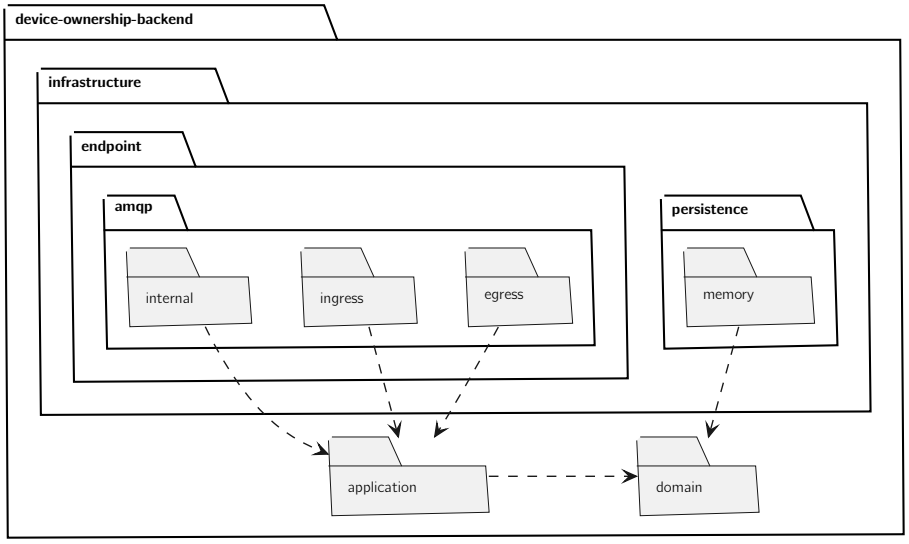


Figure C.8: Device Ownership Backend - Component Level - Implementation View Diagram

The packages presented correspond to the components described in the logical view (Figure C.3). The names given in both views differ only on the case used.

Appendix D

Sensae Console - Components Level - Logical View

This Appendix presents the logical view, component level, of the Data Flow scope containers that had minor differences when compared with the other containers.

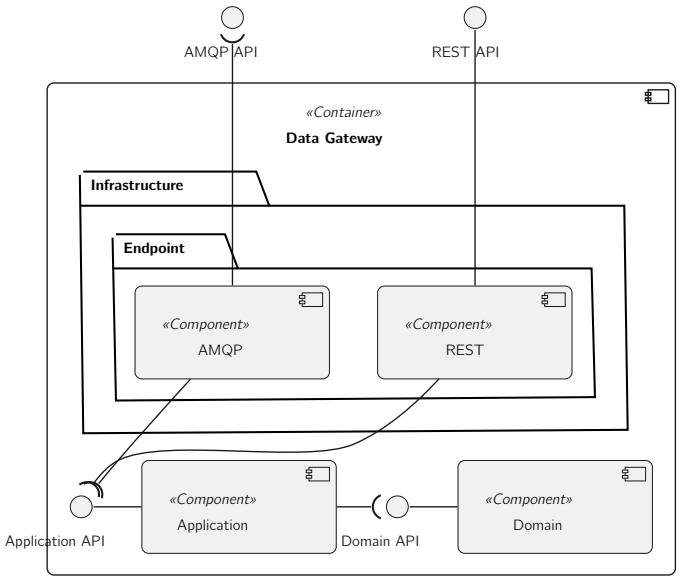


Figure D.1: Data Gateway - Component Level - Logical View Diagram

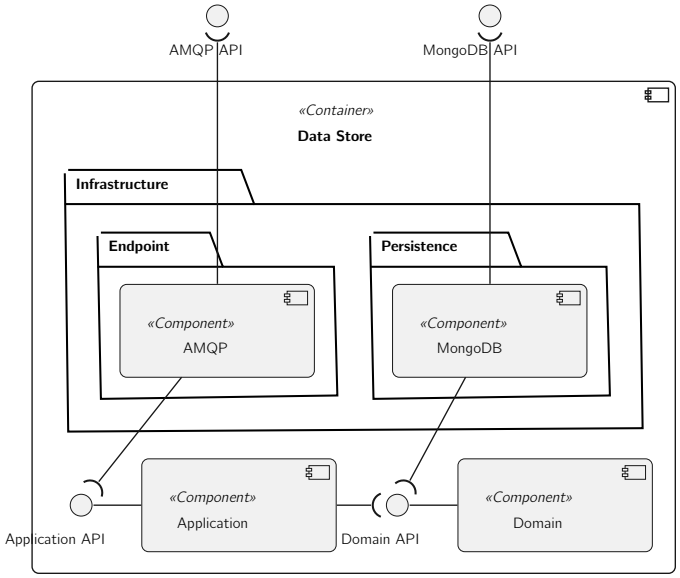


Figure D.2: Data Store - Component Level - Logical View Diagram

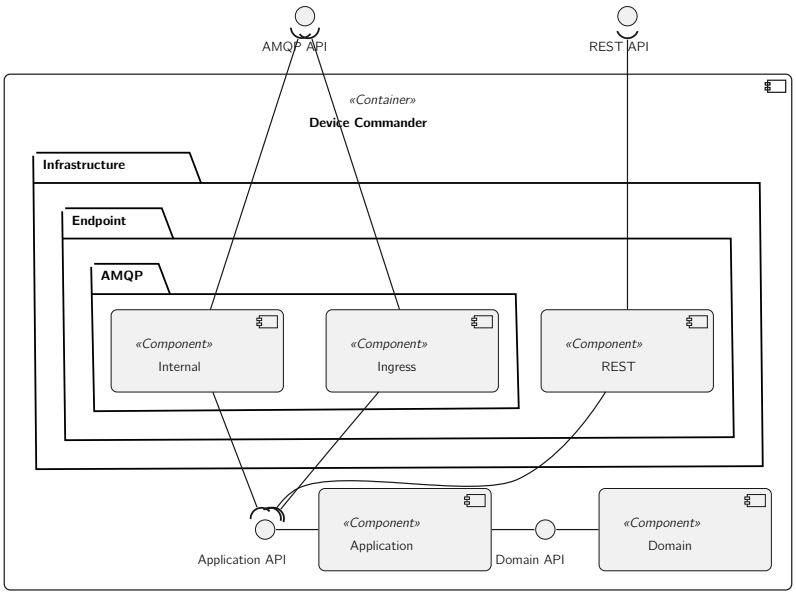


Figure D.3: Device Commander - Component Level - Logical View Diagram

Appendix E

External Services - Components Level - Logical View

This Appendix presents the logical view, component level, of all backend containers related to the PoCs developed.

The *AMQP API* is the one represented as *Sensae API for External Services* in most logical diagrams.

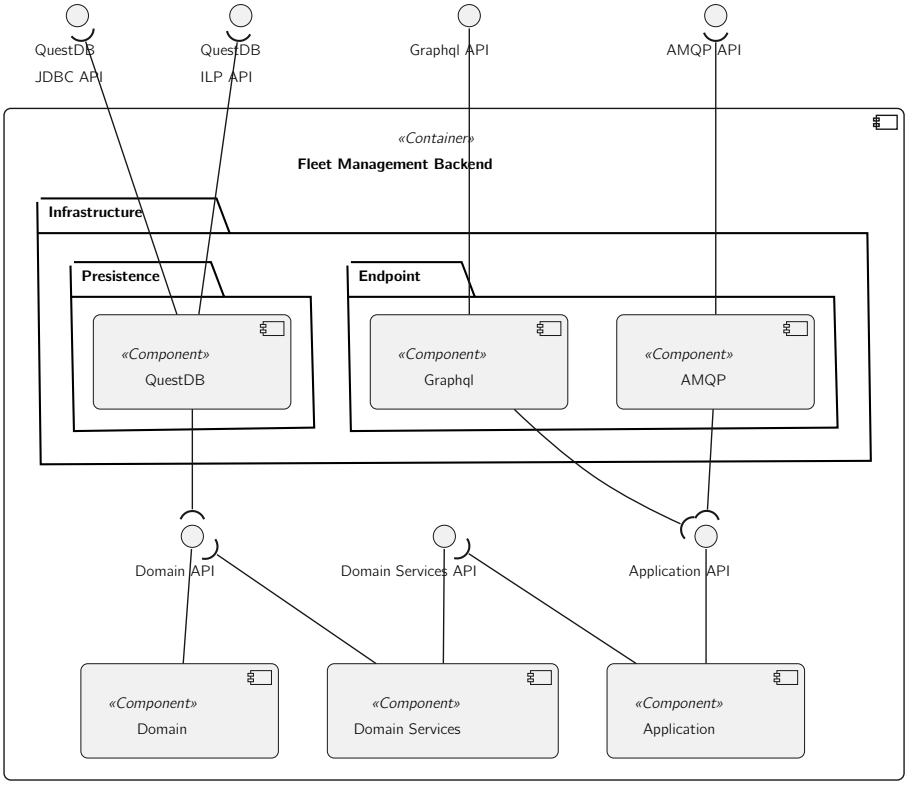


Figure E.1: Fleet Management Backend - Component Level - Logical View Diagram

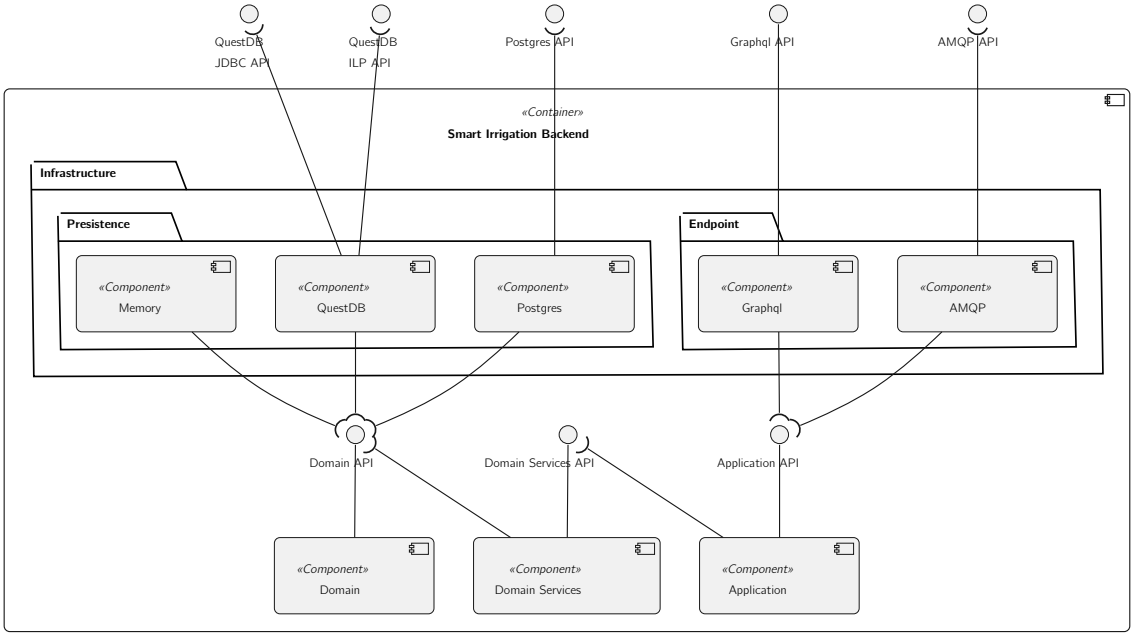


Figure E.2: Smart Irrigation Backend - Component Level - Logical View Diagram

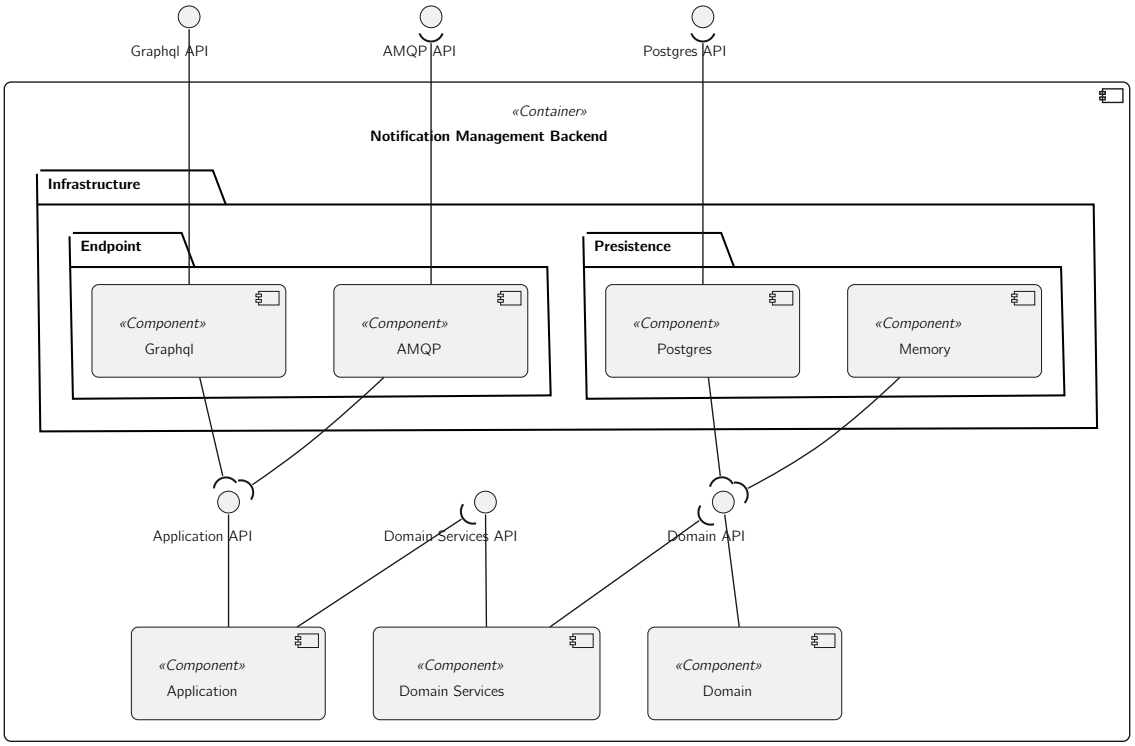


Figure E.3: Notification Management Backend - Component Level - Logical View Diagram

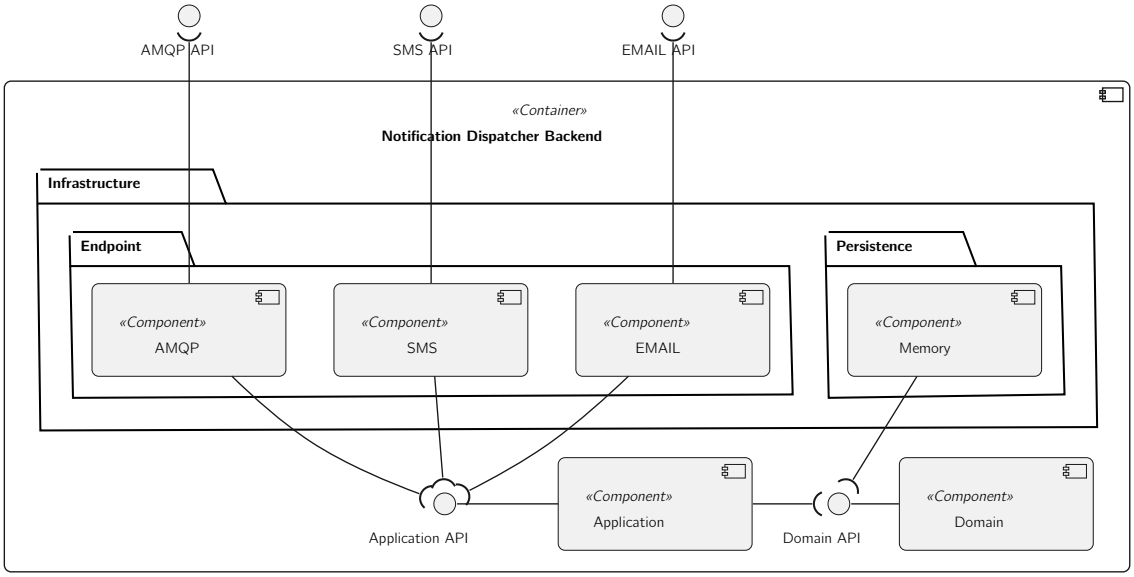


Figure E.4: Notification Dispatcher Backend - Component Level - Logical View Diagram

Appendix F

User Authentication/Authorization

User Authentication/Authorization is an important aspect of the solution. During the requirements elicitation, mentioned in Section 3.1.1, it was clear that several different levels of permissions had to be given to Tenants. These levels of permissions also had to be managed by someone. As such, users had to be authenticated in the system and all accesses had to be authorized.

Four approaches were considered:

- Internal Authentication Server;
- External Authentication Server;
- External Authentication Server with Internal Authorization Server;
- External Authentication Server with Internal Oauth2 Server.

The fourth option was the approach taken.

F.1 Internal Authentication Server

By creating an Internal Authentication Server we could have a normal, private and controlled user authentication/authorization flow in the environment. Both user credentials and permissions would be managed internally.

The following diagram, Figure F.1,presents the normal environment flow for this alternative.

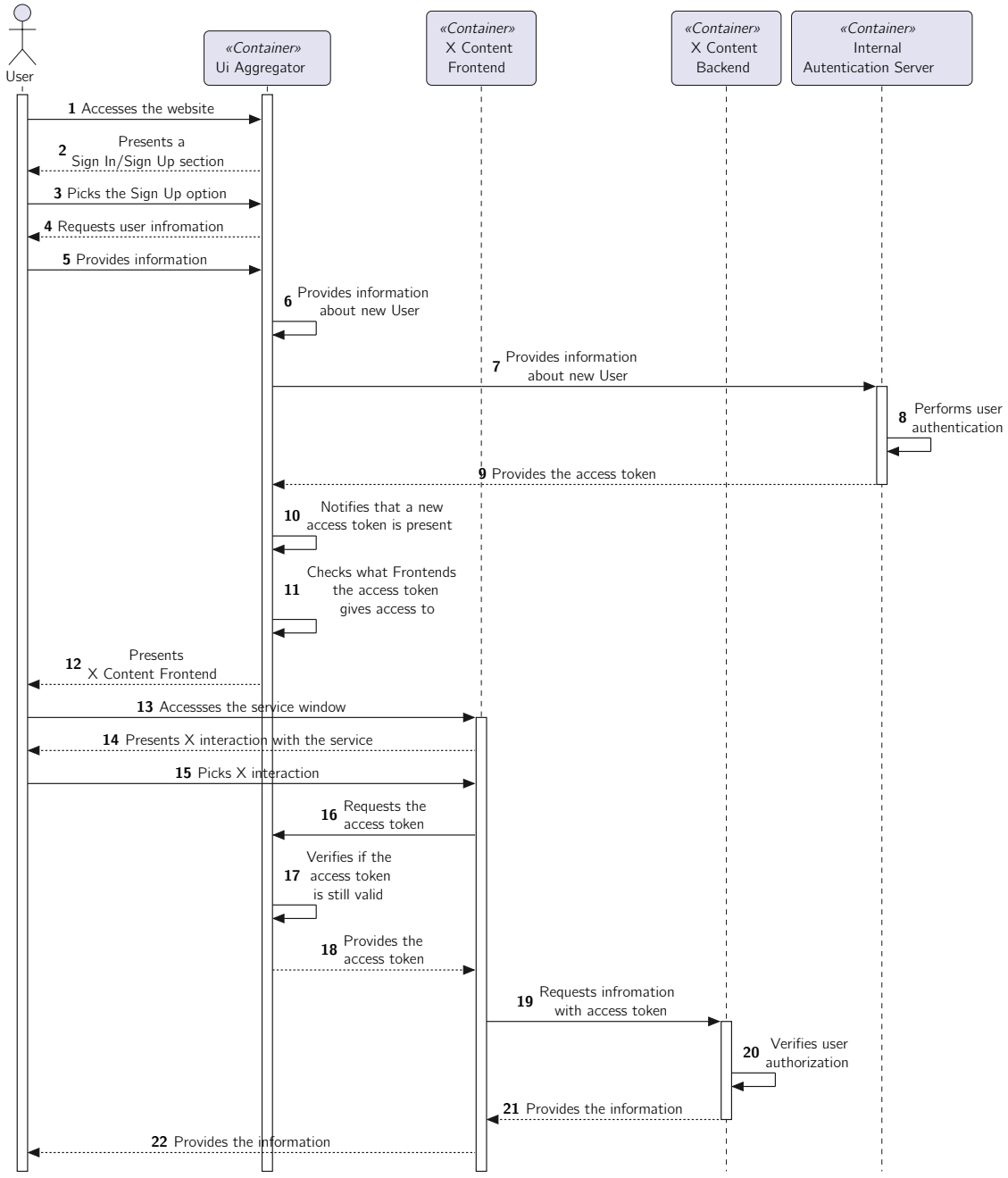


Figure F.1: User Authentication/Authorization - Internal Authentication Server Alternative - Sequence Diagram

This alternative introduces the need to internally secure user credentials and other sensitive information from data breaches. It would also require each user to register in sensae with a new account credentials. For this reasons this alternative was discarded.

F.2 External Authentication Server

By using an External Authentication Server there would be no need to store user credentials or permissions. This services are commonly identified as CIAM solutions. According to Gartner n.d. these solutions include features such as “self-service for

registration, password and consent management, profile generation and management, authentication and authorization into applications, identity repositories, reporting and analytics, APIs and SDKs for mobile applications, and social identity registration and login".

The following diagram, Figure F.2, presents the normal environment flow for this alternative.

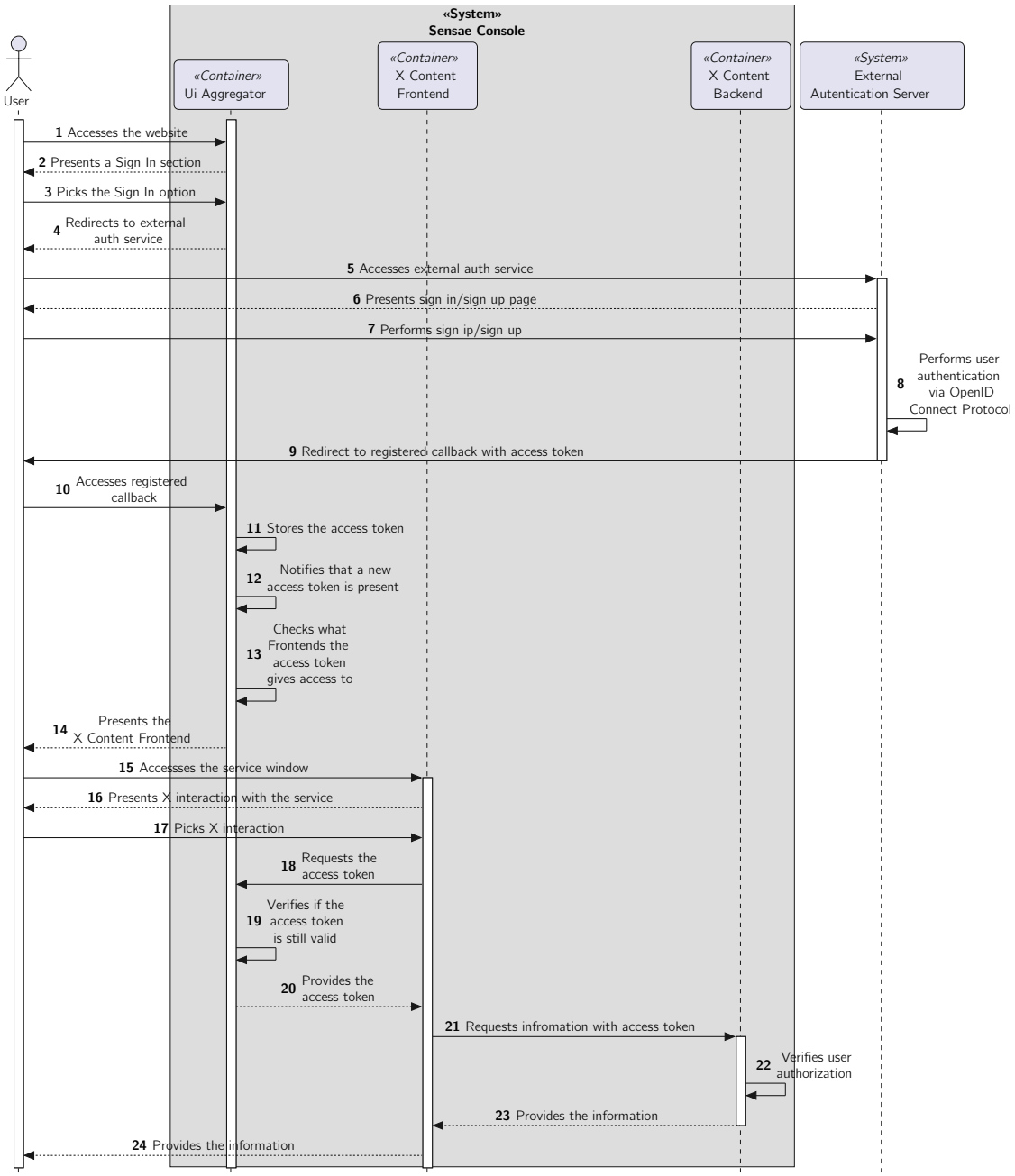


Figure F.2: User Authentication/Authorization - External Authorization Server Alternative - Sequence Diagram

This approach would create a strong dependency to the CIAM solution used since all user credentials and authorization level would have to be managed by the CIAM solution. Some

of this services are: (i) *Auth0 Customer Identity*, (ii) *Google Identity Platform*, (iii) *Okta Customer Identity*, (iv) *Amazon Cognito* and (v) *Azure Active Directory (Azure AD)*.

The platform Auth0 was tested and is capable of answering all of this project's requirements.

As stated before, the dependency created would force the environment to always be coupled to the chosen CIAM solution. For this reason this alternative was discarded.

F.3 External Authentication Server with Internal Authorization Server

By using an External Authentication Server there would be no need to store user credentials, the user authorization aspects would then be managed internally via and *Authorization Server*.

The following diagram, Figure F.3, presents the normal environment flow for this alternative.

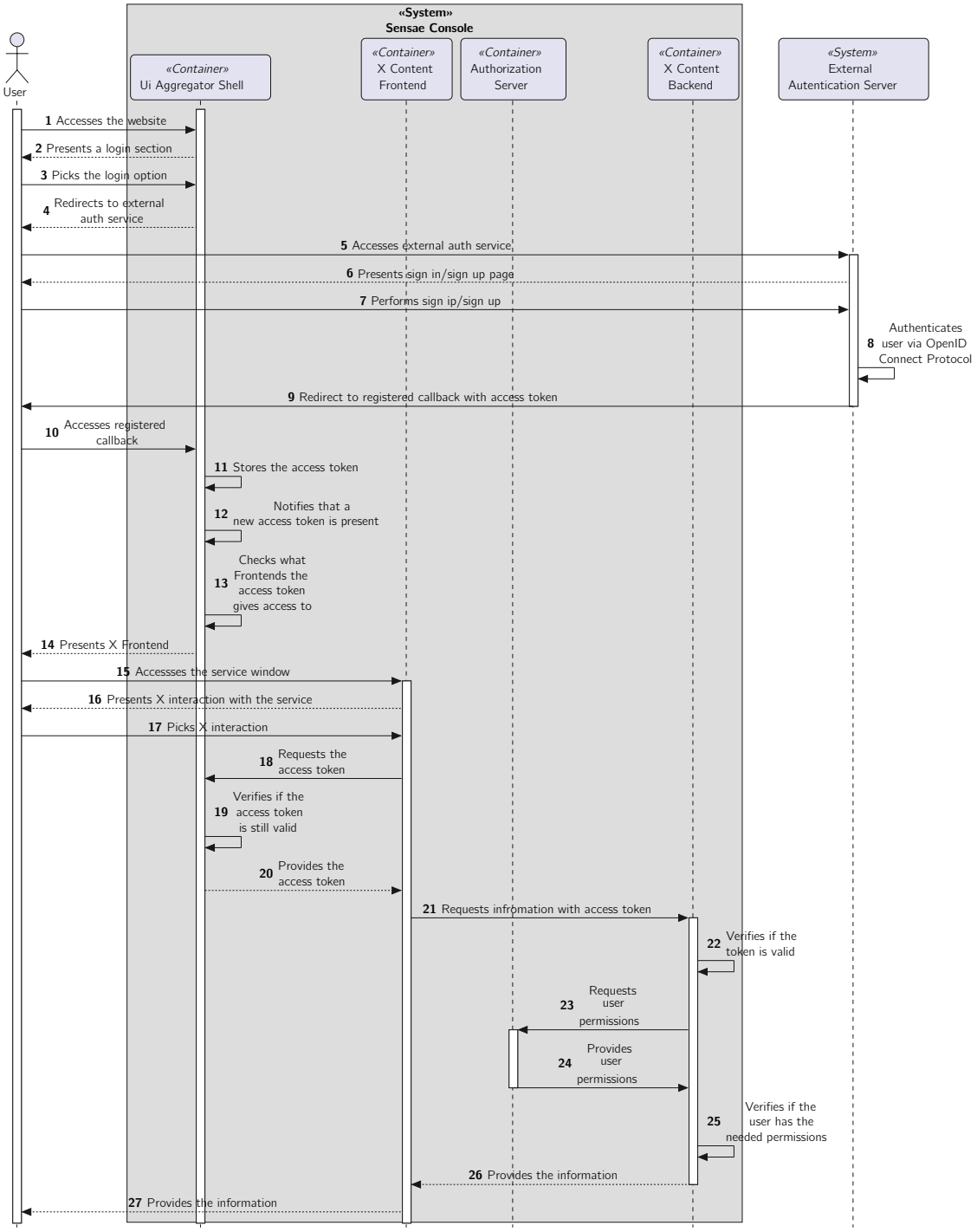


Figure F.3: User Authentication/Authorization - External Authentication Server with Internal Authorization Server Alternative - Sequence Diagram

This approach would create a dependency to the CIAM solution used and presented in the second alternative.

This dependency is less severe compared with the second alternative since all authorization aspects would be managed internally. This approach would require any backend to query the *Authorization Server* for user permissions so that it could verify if the user was authorized to preform the requested action or not. This would therefore linger down the

performance of the system since each action would have to be verified in a single container: the *Authorization Server*.

F.4 External Authentication Server with Internal Oauth2 Server

By using an external Authorization Server there would be no need to store user credentials. An internal Oauth2 Server would remove the direct dependency to the *Permissions Server* presented in the third alternative.

This alternative is introduced in Figure 4.17 and Figure 4.18 where the Internal Oauth2 Server is the Identity Management Backend.

This approach would create a dependency to the CIAM solution used and presented in the second alternative. This dependency is less severe compared with the second alternative since all user permissions would be managed internally. This approach would require the system to create and refresh *access tokens* based on the *id token* received by the external CIAM solution. Contrary to the third alternative it would not create excessive pressure in a specific container.

This approach also allows the system to easily integrate with more than one CIAM solution while managing user permissions in a single place. The CIAM solutions that **Sensae Console** is integrated with are:

- Google Identity Platform: for common individuals that want to use the system, since almost everyone has a google account;
- Azure Active Directory: for companies and organizations since most use Office 365 services internally.

Due to the reasons presented above, this was the adopted approach.

Appendix G

Sensae Console Domains

The **Bounded Context** concept, defined by Evans 2014, refers to an unified model - with well-defined boundaries and internally consistent - that is, a single piece in a larger system composed by various bounded contexts.

The concept **Bounded Concern** refereed in this section draws inspiration from the one coined by Evans 2014, without the notion of Aggregates, Value Objects, Aggregate Root and other Domain Driven Design (DDD) concepts. It is here to simply characterize the various models of the system that, when isolated, can be more clearly interpreted and understood by the reader.

For **Sensae Console**, each bounded concern can be pictured as a core business process of the system, it is composed by the following:

- Data Processor;
- Data Decoder;
- Device Management;
- Identity Management;
- Rule Management.

Each of this concerns will be briefly addressed in the following sections.

G.1 Data Processor

The **Data Processor** concern refers to simple data mappers that translate inbound information to **Data Unit**s, discussed in Section 4.4.2.1.

The received information must be *decoded*, meaning that the inbound information simply has a different structure than **Data Unit**.

The diagram in Figure G.1 displays the noteworthy concepts in this concern.

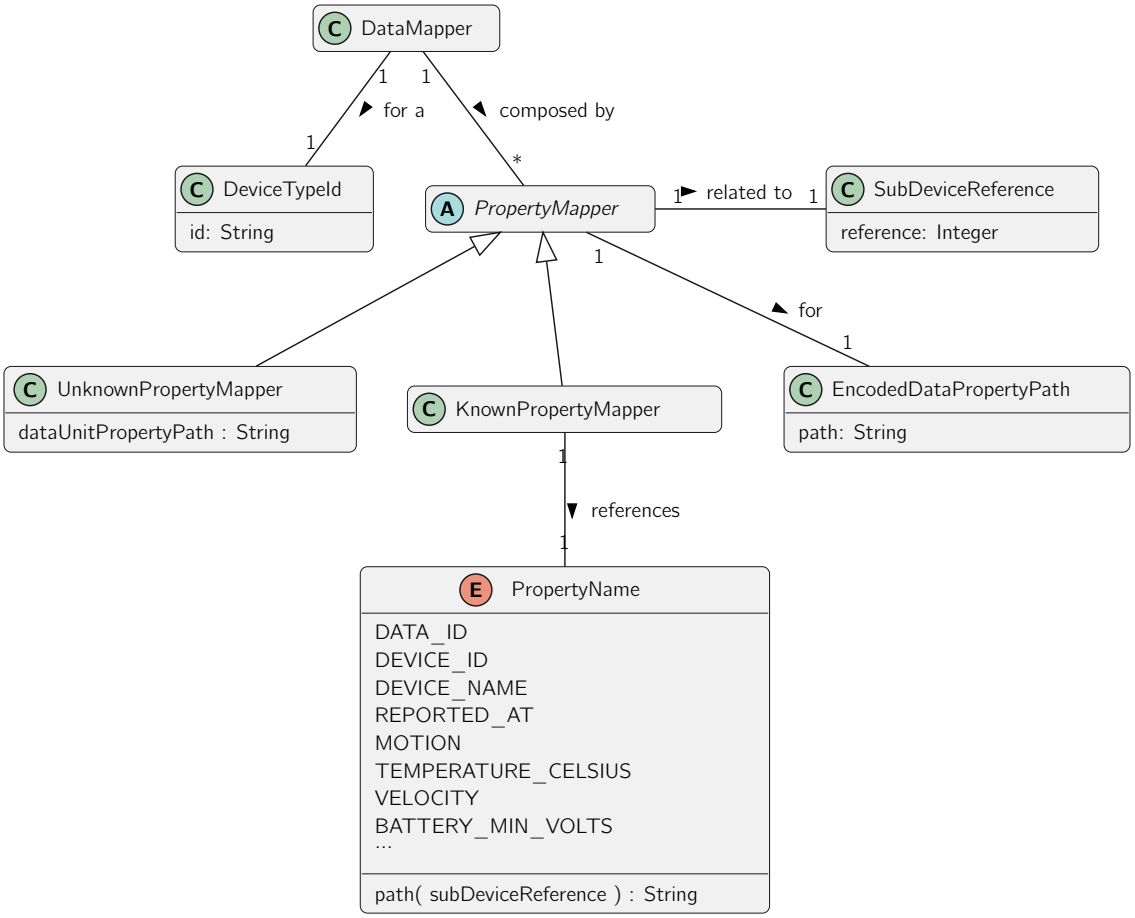


Figure G.1: Data Processor Concern Model

As a brief description:

- **DataMapper**, the root entity in this concern is identified by a **DeviceTypeId** and has various instructions to map properties from the inbound information to a **Data Unit** properties;
- **DeviceTypeId** identifies the type of device that can be processed by this data mapper. When a data unit's message is supplied to this concern the data mapper that has the **DeviceTypeId** equal to the message's *Device Type Options* routing key (mentioned in Table 4.3) is used to process that data unit;
- **SubDeviceReference** represents a number that will be used later to reference a sub device when dealing with **Controller**s. For simple **Devices** the used and default value is *0*;
- **PropertyName** has much more properties that haven't been presented for brevity.

As an example, consider the inbound information represented as a JSON document with the structure in the example G.1. To map the *temperature* value to the **TEMPERATURE_CELSIUS** property of a **Data Unit**, the **EncodedDataPropertyPath** would be *decoded.data[0].temperature*.

```
{
    "uuid": "de1a9d15-c018-4547-8453-87111cb4f81b",
```

```
    "id": "d81e6e69-1955-48a1-a1dd-4c812c15ebac",
    "time": 1657646955748,
    "decoded": {
        "data": [
            {
                "temperature": 18,
            }
        ]
    }
}
```

Listing G.1: Inbound Information Example

This process is simple since it expects the inbound information to be predisposed, but when working with IoT Devices, to optimize the bandwidth used, it is common to send information encoded. The following section presents an alternative to this process.

G.2 Data Decoder

The **Data Decoder** concern refers to a more complex data mapper that translates inbound information to **Data Unit**s, discussed in Section 4.4.2.1. It was created to deal with the limitations mentioned in Section G.1.

The received information is usually *encoded*, meaning that the inbound information is received as it was sent by the **Device**, commonly as a *Base64* encoded string, that needs to be processed so that information can be extracted.

The diagram in Figure G.2 displays the noteworthy concepts in this concern.

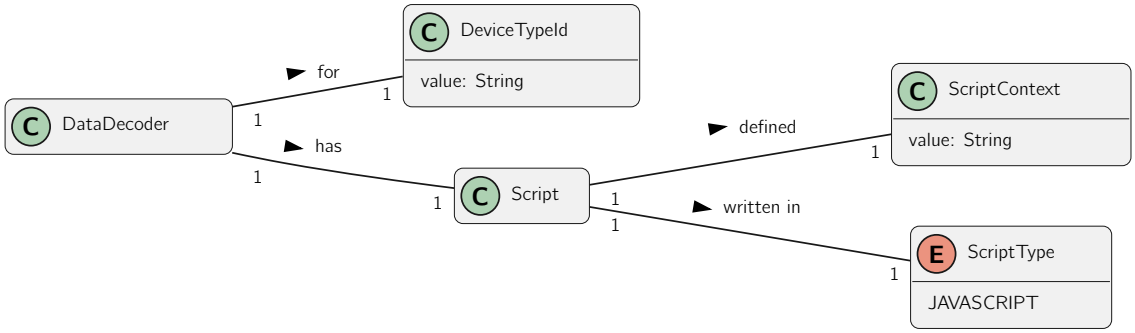


Figure G.2: Data Decoder Concern Model

As a brief description:

- **DataDecoder**, the root entity in this concern is identified by a **DeviceTypeId** and has a **Script**;
- Currently, a **Script** can only be written in *JavaScript* but in the future more languages like *Python* or *Groovy* can be added;
- The **ScriptContent** contains the code that will run for each inbound information that matches the **DeviceTypeId**.

This process requires some knowledge of the *Javascript* language but it's much more flexible than the **Data Processor** operation.

G.3 Device Management

The **Device Management** concern refers to the inventory of all registered **Device**s in the **Sensae Console**.

The diagram in Figure G.3 displays the noteworthy concepts in this concern.

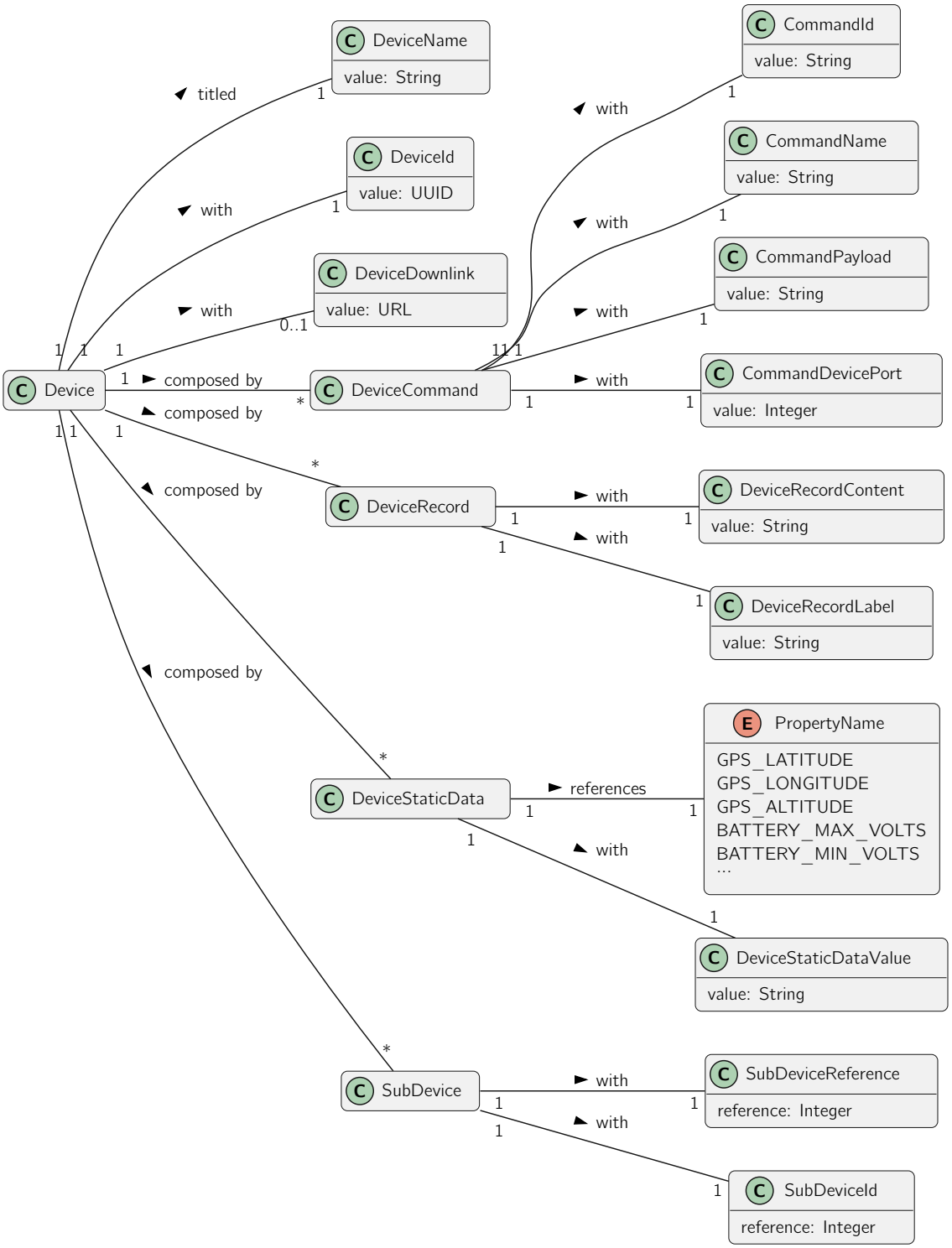


Figure G.3: Device Management Concern Model

As a brief description:

- A **Device** is uniquely identified by a **DeviceId** and a **DeviceName**. It may have a **DeviceDownlink**, an URL used to send device commands to;
- A **DeviceCommand** defines how to send a **Downlink** with a specific action;
- A **DeviceStaticData** helps to define data such as the device location;
- A **DeviceRecord** enriches the device information with anything deemed important. This can also help to group devices by projects, type of utility and others;
- A **SubDevice** references another **Device** by its **DeviceId**. This, coupled with the concepts **SubDeviceMeasures** and **SubDeviceCommands** presented in Figure 4.35 help to split a **Controller**'s **Data Unit** into various **Data Unit**, one for each referenced **SubDevice**.

G.4 Identity Management

The **Identity Management** is concerned with identifying **Tenant**s, defining their permissions and what **Device**s they own. To simplify this, a forth concept is introduced: **Domain**.

The diagram in Figure G.4 displays the noteworthy concepts in this concern.

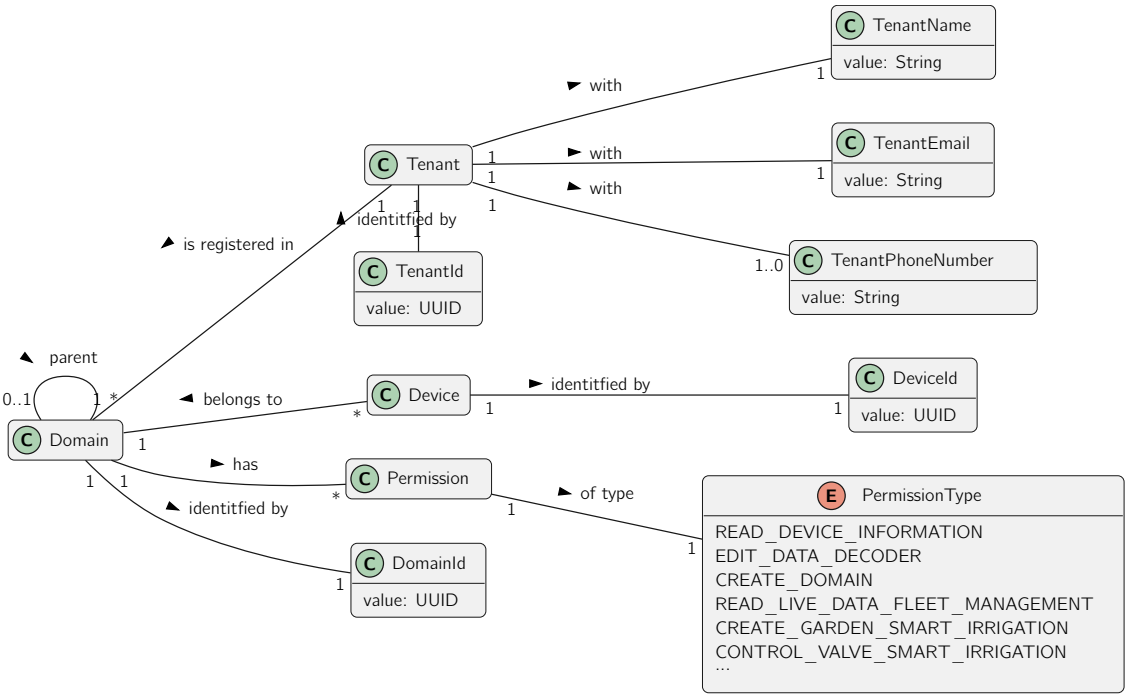


Figure G.4: Identity Management Concern Model

As a brief description:

- A **Domain** is uniquely identified by a **DomainId** and can have a parent **Domain**;
- There's a root **Domain**, the only one that doesn't have a parent and has all the available permissions;

- A **Tenant** has a **TenantName** and **TenantEmail**, unique **TenantId** and can have a **TenantPhoneNumber**;
- A special **Tenant**, Anonymous, exists by default to give access to users without an account in the platform;
- A **Device** is uniquely identified by a **DeviceId**;
- The **PermissionType** has much more types that haven't been presented for brevity.

A **Domain** represents a department in a hierarchical organization. An organization is composed by several domains in a tree like structure as presented in Figure G.5.

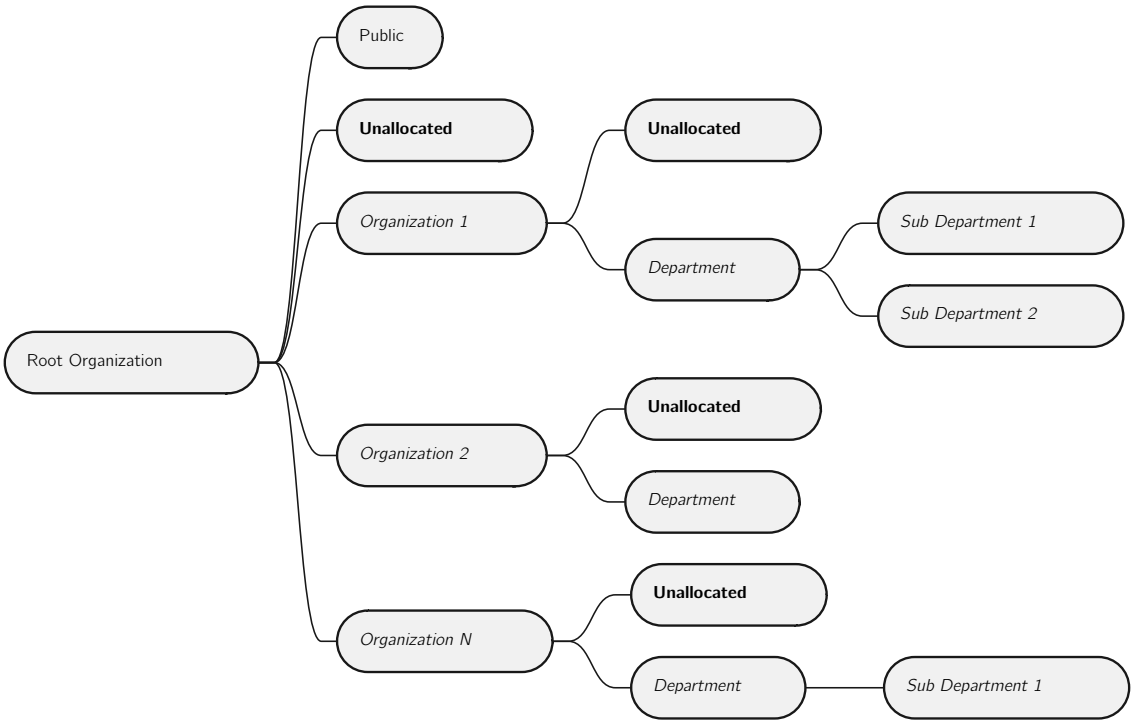


Figure G.5: Domain Structure

Coupled with the figure above, there are other constrains:

- A domain owns all devices in it and in his subdomains;
- A domain can only inherit his parent domain permissions;
- A tenant has all the domain permissions that he/she is registered in;
- A tenant can only see the devices that the domains he/she is registered in has access to;
- All *Unallocated* domains have no permissions or devices and contain only tenants that are waiting to be assigned to a department or organization;
- The creation of an *Organization* (level 2 domain), triggers the creation of its *Unallocated* domain;
- The *Public* domain can be accessed by any tenant, including those who are not authenticated in the system - with the Anonymous User account.

By default this concern contains the *Root Organization* domain, the *Root Organization*'s *Unallocated* domain and the *Public* domain.

Referring to the roles in Section 3.1.1, a Manager belongs to the *Root Organization*, any Costumer belongs to one or various *Organization*s, and the Anonymous user belongs to the *Public* domain. Ultimately, what defines a user role is the domain he/she belongs to. Even if an *Organization* ends up having all available permissions it will not be able to control or access other *Organization*'s device data or employees information.

G.5 Rule Management

The **Rule Management** concern refers to rule scenarios.

The purpose of this concern is to provide a high-level language that can analyze a stream of **Data Unit**s, identify abnormal occurrences, and output **Alert**s base on them.

In this concern, and according to Figure 2.2 in Section 2.1.1.4, the input data are **Data Unit**s and the output data are the **Alert**s. This concern is involved on how *rules* are defined. The diagram in Figure G.6 displays the noteworthy concepts.

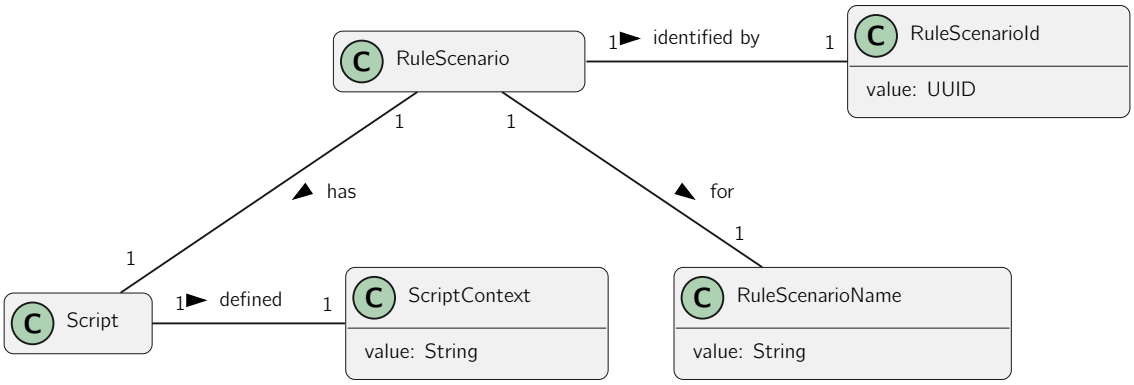


Figure G.6: Rule Management Concern Model

Appendix H

External Services Domains

The developed PoCs are:

- Smart Irrigation;
- Fleet Management;
- Notification Management.

Each of this services' models will be briefly addressed in the following sections. The shared model presented above defined the structure and semantic of incoming data. Each service then uses the shared model how they envision, in their specific practical and pragmatic fashion.

H.1 Fleet Management

The **Fleet Management** model simply refers to the past and current location of assets.

The diagram in Figure H.1 displays the noteworthy concepts related to this service.

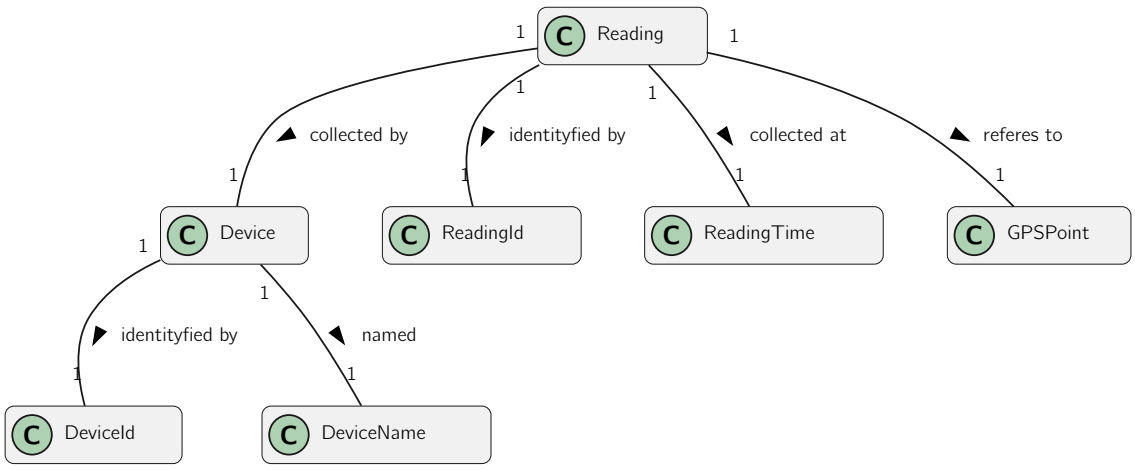


Figure H.1: Fleet Management Model

This was the first External Service built as a PoC, it was intended to be straightforward. The model references GPS readings and what device collected them.

H.2 Notification Management

The **Notification Management** model refer to notifications and how/what types an addressee wants to receive. There are two main concepts in this service, a notification and an addressee.

The diagram in Figure H.2 displays the noteworthy concepts related to this service.

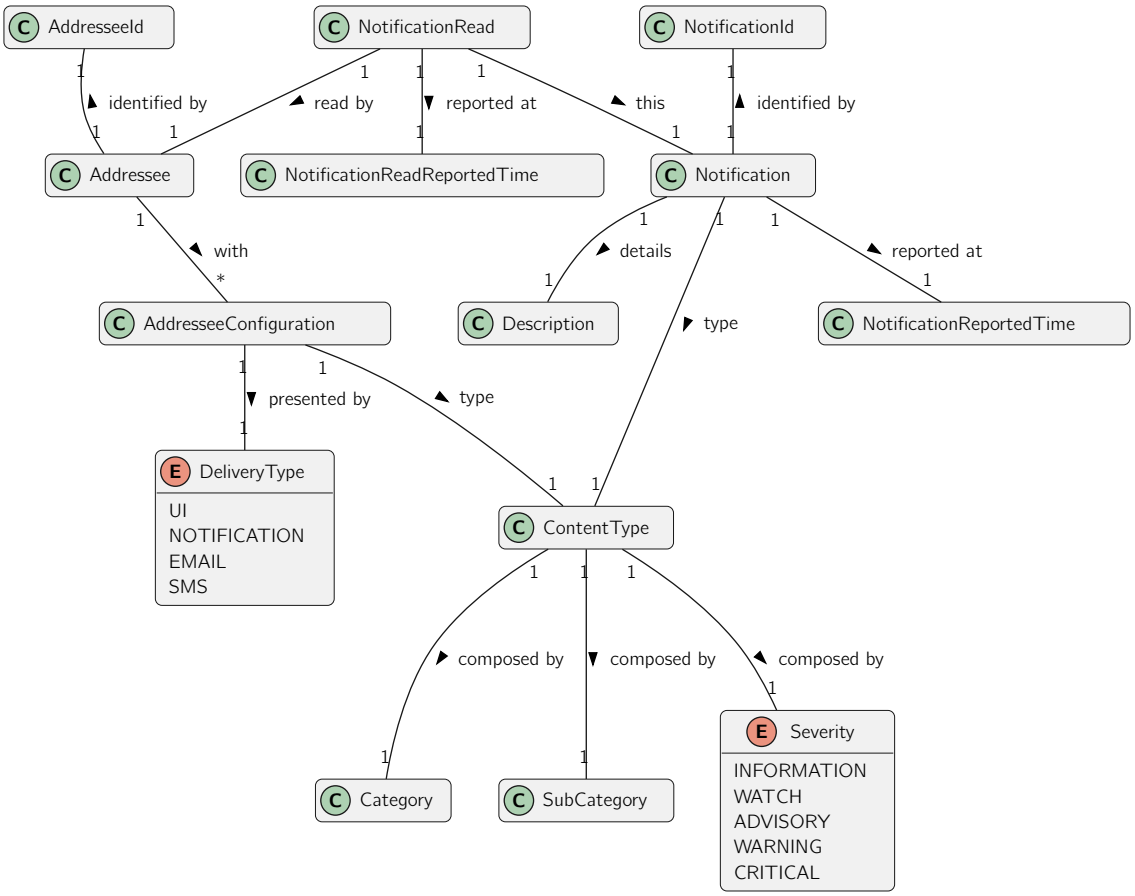


Figure H.2: Notification Management Model

As a brief description:

- A **Notification** is a sanitized **Alert** that was captured with the intent to be presented or delivered to addressees, its identified by an **NotificationId**;
- An **Addressee** is someone that receives notifications based on his configurations and is identified by an **AddresseeId**;
- An **AddresseeConfiguration** defines for each type of notification - **ContentType** - what will be the delivery method - **DeliveryType**;
- A **DeliveryType** can be of four types: (i) present in SPA - **UI**, (ii) publish notification in SPA - **NOTIFICATION**, (iii) send an email - **EMAIL**, (iv) send an SMS - **SMS**;
- A **ContentType** is derived from the **Alert** Routing Keys mentioned in the Table 4.3 and defines the type of each **Notification**;

- To enforce accountability in the system, the notion of who read a specific notification and when was added - **NotificationRead**.

H.3 Smart Irrigation

The **Smart Irrigation** model refers to irrigation zones, sensors that read environmental conditions in this zones, valves and the associated readings. This concepts are divided in three diagrams presented below.

The diagram in Figure H.3 displays the noteworthy concepts related to irrigation zones.

An irrigation zone is an area intended to function as an isolated environment that may or may not have valves or sensors.

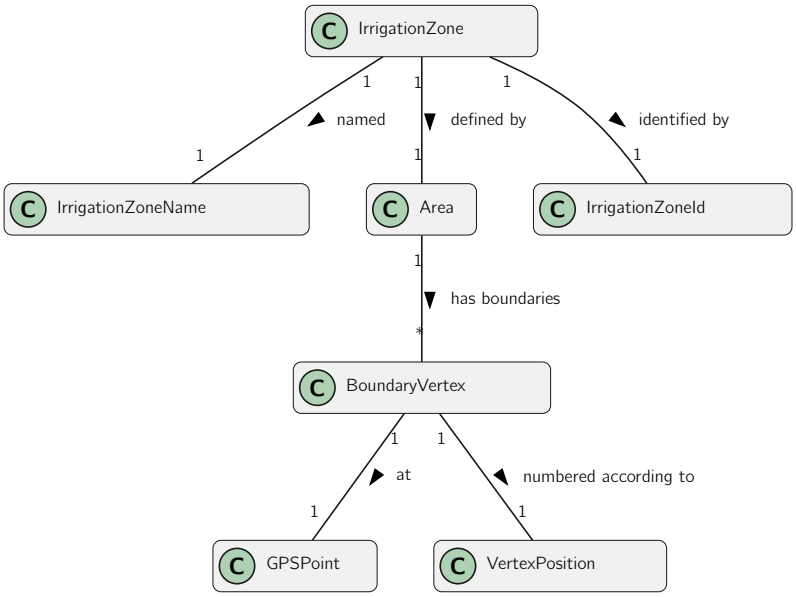


Figure H.3: Smart Irrigation Model - Irrigation Zone

A sensor or valve belongs to an irrigation zone if it is inside the zone's **Area**.

As presented in the following diagram, Figure H.4, a sensor/valve can be represents by a **Device**.

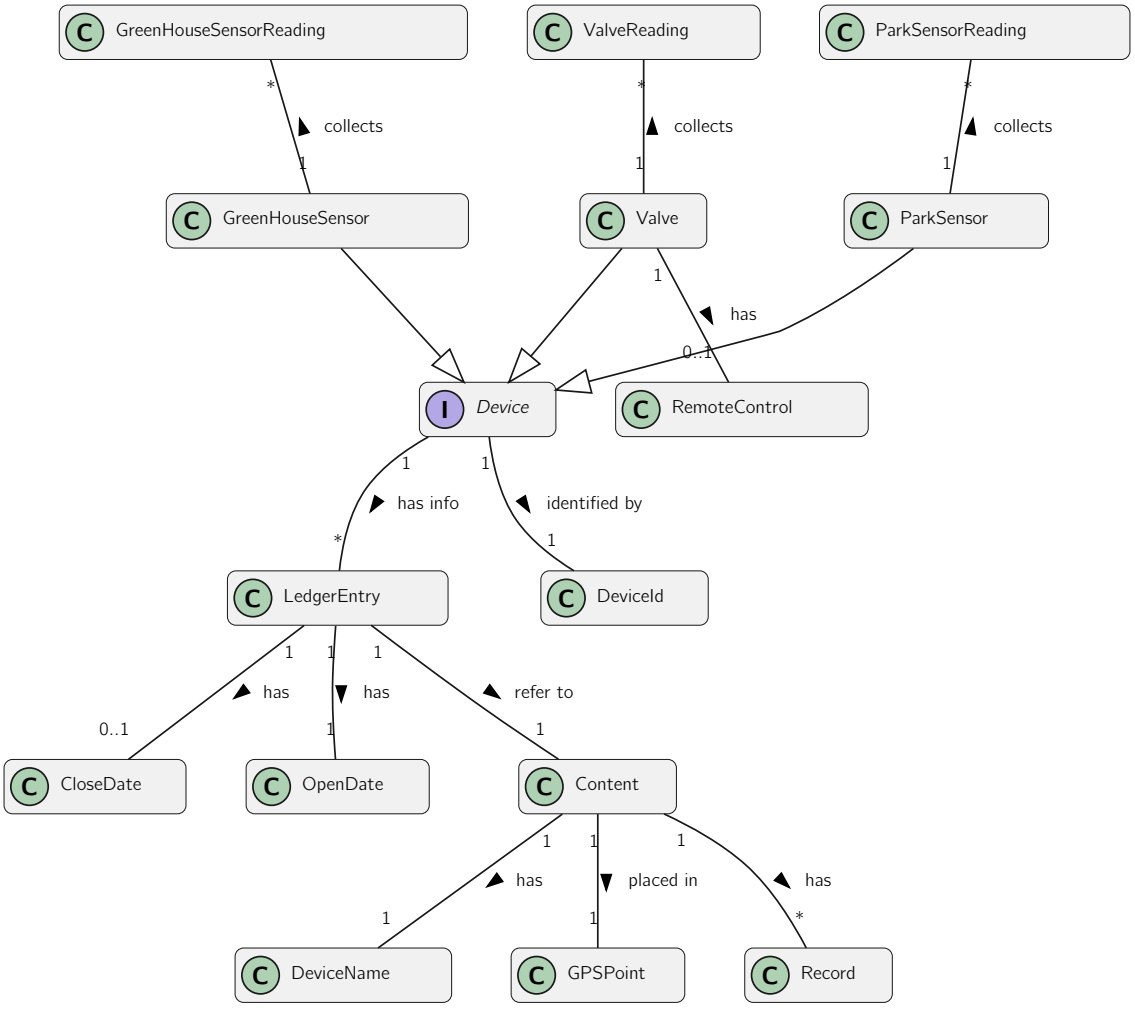


Figure H.4: Smart Irrigation Model - Device

As a brief description:

- The **RemoteControl** defines if a **Valve** can be controlled remotely. A valve can be controlled remotely only if two specific types of **Command**s (as defined in the Model) are sent with the device's **Data Unit**: *OpenValve* and *CloseValve*;
- A **Device** is identified by its **DeviceId**;
- Each **Device** stores an history of all its changes such as name, location or metadata in **Content**, the same **LedgerEntry** is used as long as this values don't change;
- There are three types of **Device**: (i) Green House Sensor, (ii) Park Sensor, (iii) Valve. Each of this types collect different measures discussed in FigureH.5.

As mentioned above each type of device collects different readings. The following diagram, FigureH.5, details this readings.

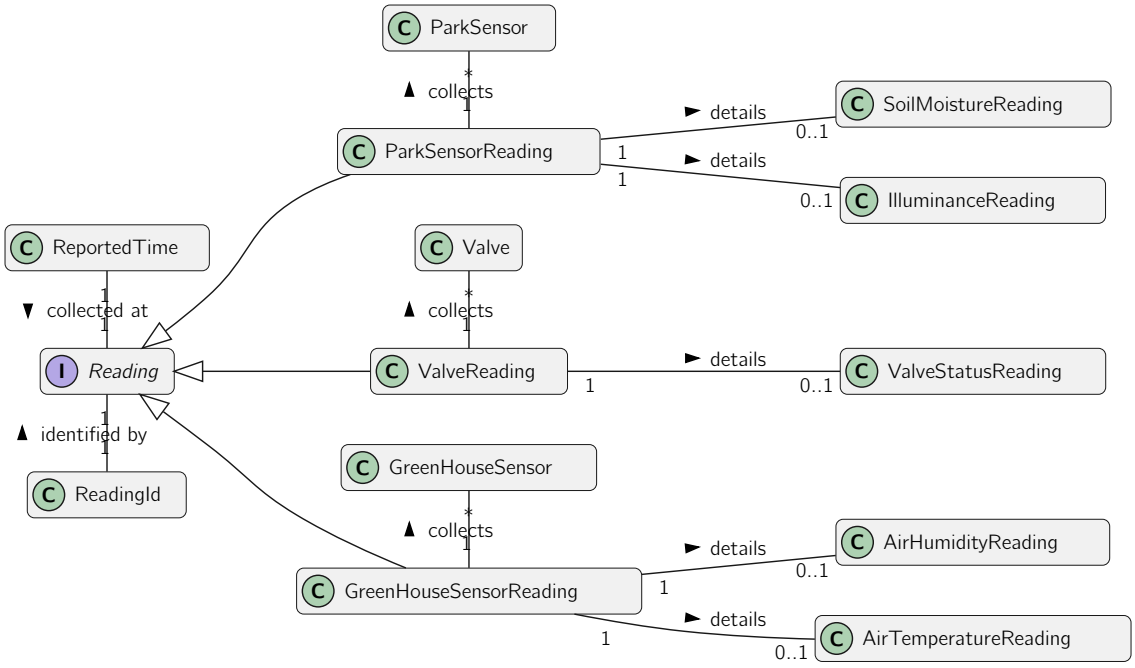


Figure H.5: Smart Irrigation Model - Reading

As a brief description:

- A **Reading** is always identified by its **ReadingId** and is associated to the instant that it was captured by the **Device** - **ReportedTime**;
- A **ParkSensorReading** measures soil moisture and illuminance;
- A **Valve** indicates if it is open or closed;
- A **GreenHouseSensor** measures air humidity and air temperature.

The concepts in this last diagram are different from the concepts in the other two diagram since readings data is suppose to be immutable and ample as opposed to devices and irrigation zones where information should be mutable but with a negligible size compared with readings.

Appendix I

Sensae Console - Additional UI Pages

This appendix presents other **Sensae Console** pages.

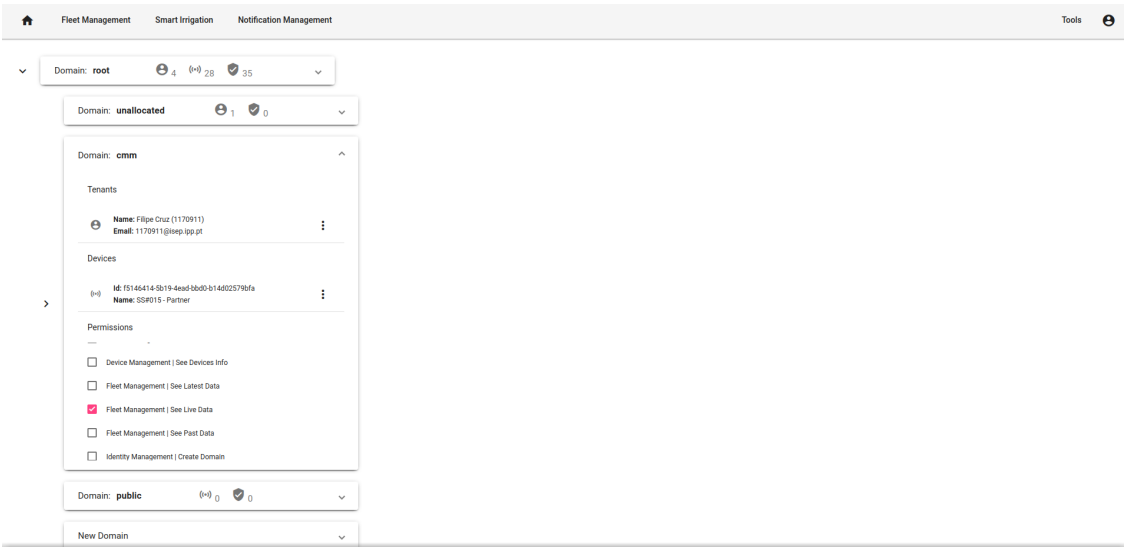


Figure I.1: Identity Management Page

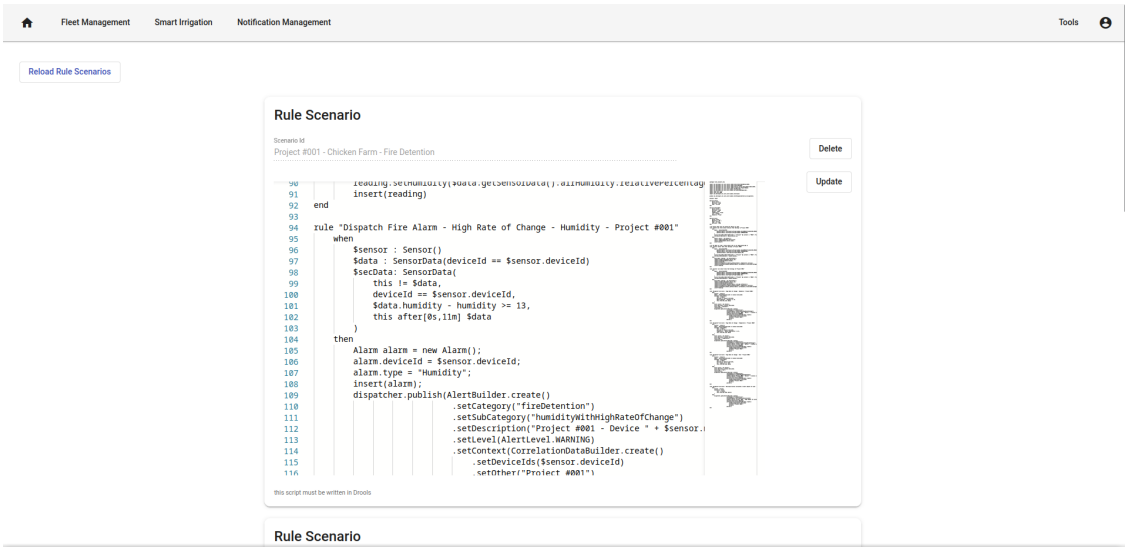


Figure I.2: Rules Management Page

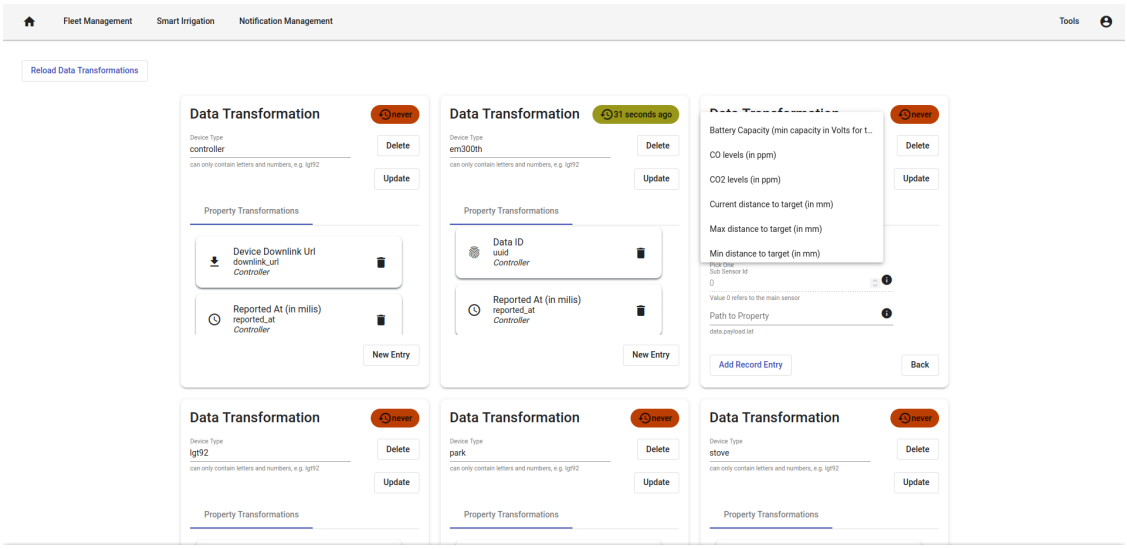


Figure I.3: Data Processor Page

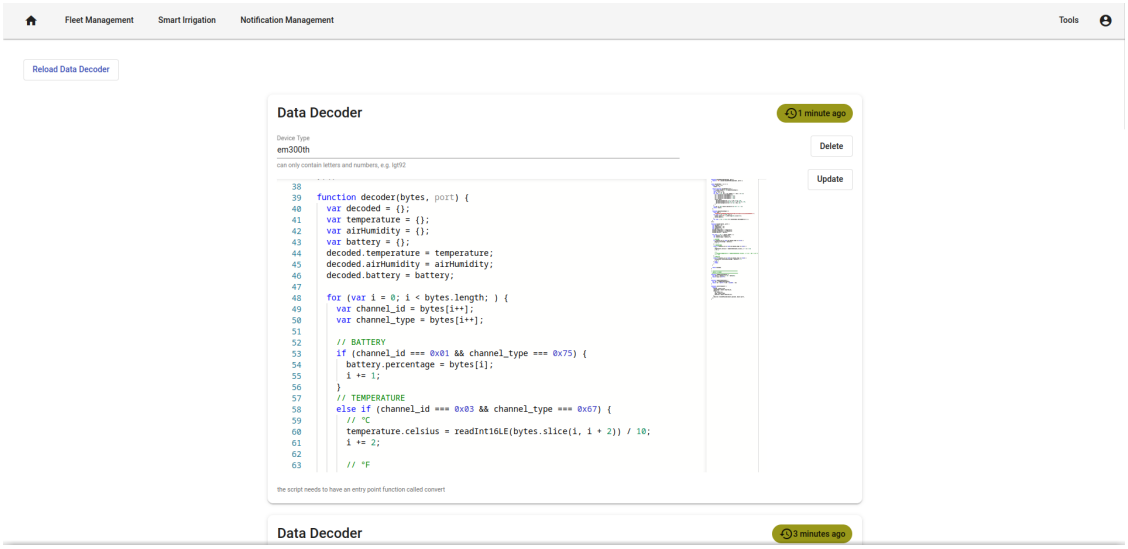


Figure I.4: Data Decoder Page

Appendix J

External Services - Additional UI Pages

This appendix presents more pages related to the Solutions developed.

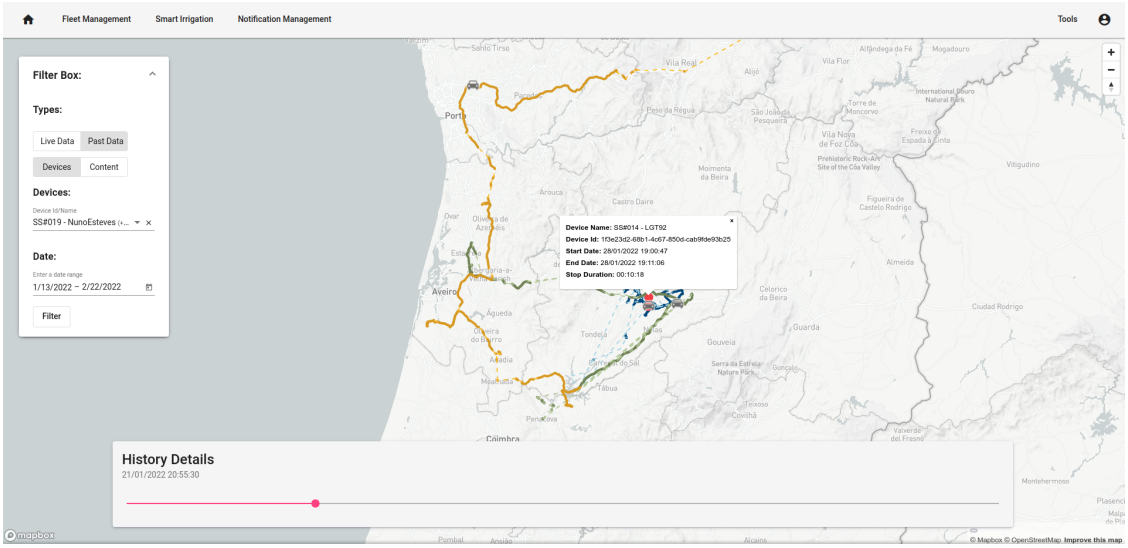


Figure J.1: Fleet Management Page

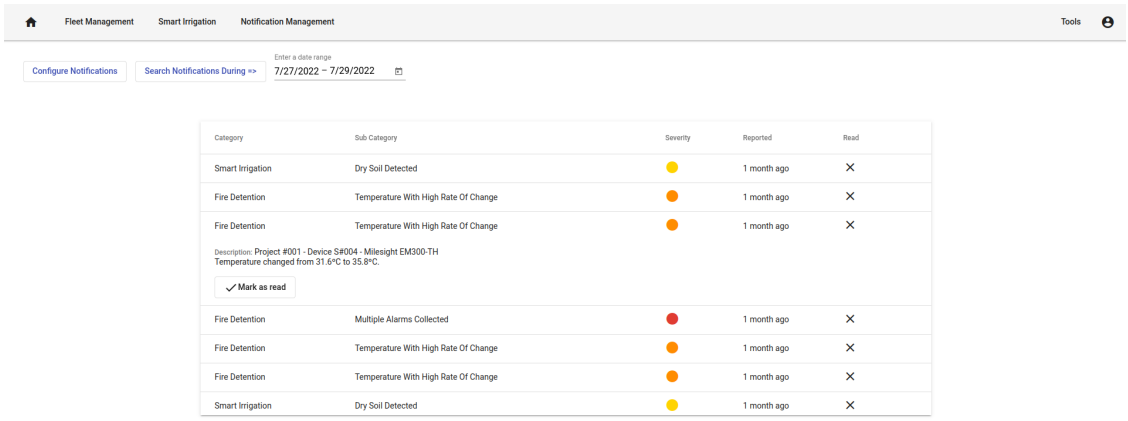


Figure J.2: Notification Management Page

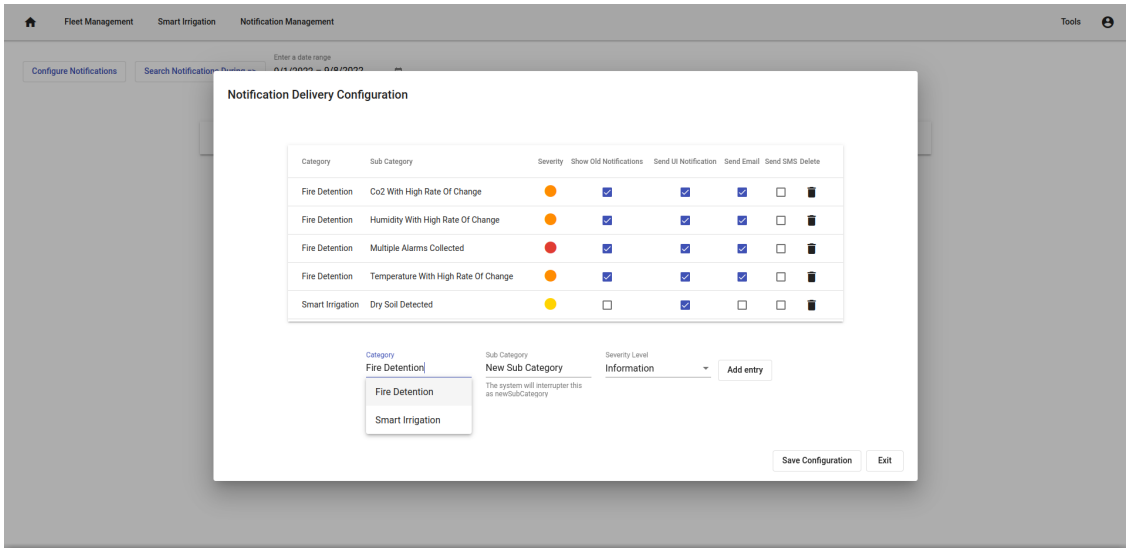


Figure J.3: Notification Management Page - Configuration

Figure J.4: Smart Irrigation Page - Map

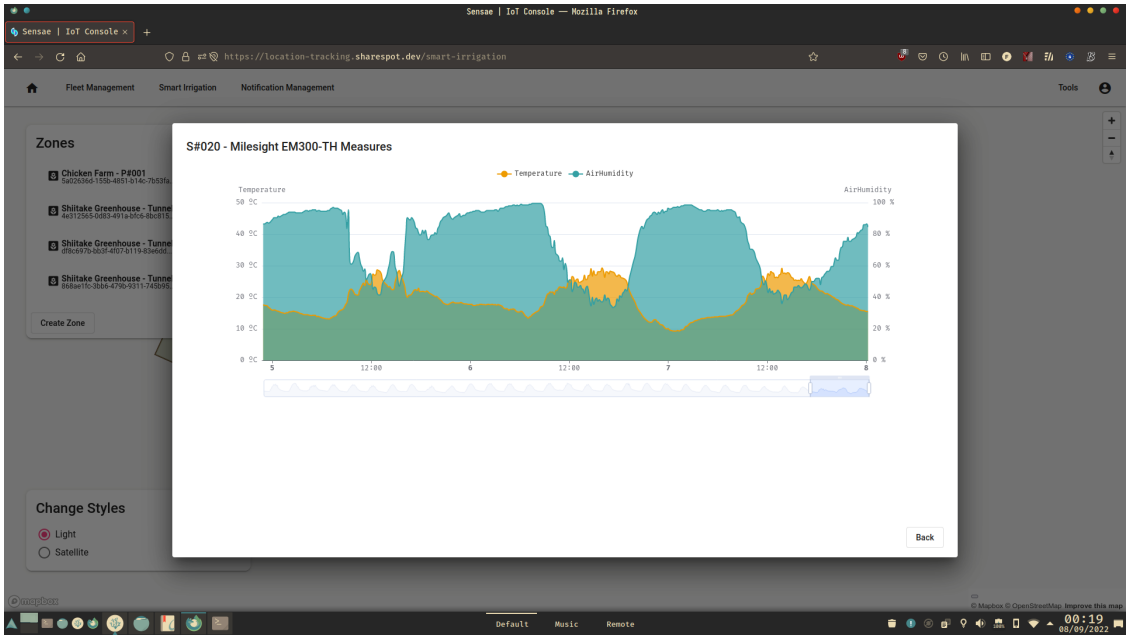


Figure J.5: Smart Irrigation Page - Device History

Appendix K

Production Deployment Details

This appendix details how **Sensae Console** and the **External Services** are currently managed in production.

K.1 Containerization of services via Docker

This section describes how the final product is packaged into containers.

As stated in *Docker overview* 2022a, Docker acts as an intermediary layer between the application to be deployed and the operating system where it will be deployed, ensuring that the developed software has the same behavior regardless of the system. The dependencies of the solution do not have to be present in the system, it is only necessary to install the Docker tool in the OS.

This tool thus makes it possible to lower the coupling between the OS and the software to be deployed.

With regards for this solution, each container defined in Section 4.2 is mapped into a docker container. A container is often compared to a virtual machine running on a hypervisor or OS, but it has a much lower resource consumption, since only the application runs and not not all the processes inherent to an OS as described by Bernstein 2014.

The Figure K.1 compares a VM and Container-based deployments.

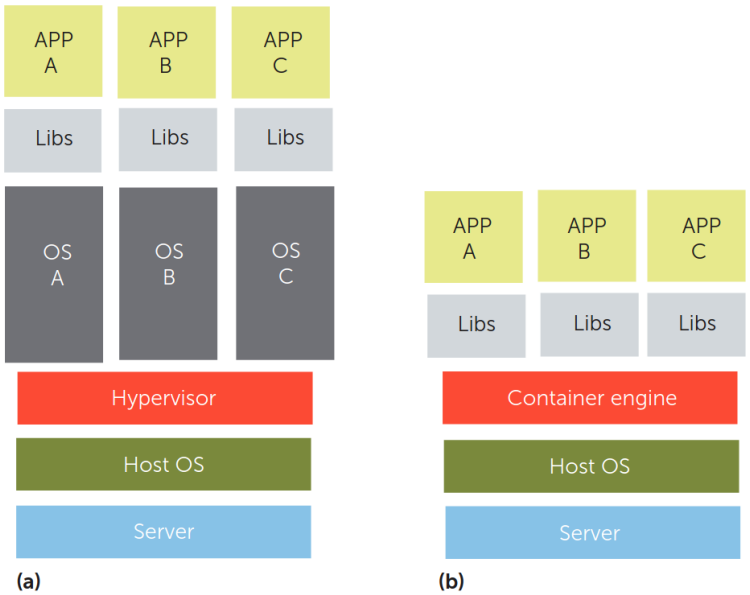


Figure K.1: Comparison of VM (a) and Container-based (b) deployments by Bernstein 2014

The system is thus represented as a collection of containers that communicate with each other and the outside through standard protocols such as HTTP or AMQP.

The production environment can thus be quickly replicated on another machine in case of a failure disaster or a overwhelming number of interaction with the server.

Details about service containerization can be found in Section K.4.

K.2 Orchestration of services via Docker Compose

This section describes how the final product is orchestrated using Docker Compose.

As stated in the article *Overview of Docker Compose* 2022b, "Compose is a tool for defining and running multi-container Docker applications".

Currently a single node is capable of handling the traffic generated by all the managed devices and costumers. Due to this, it was decided to use a docker compose in production inserted of tools like Kubernetes (that can ease the process of autoscaling individual containers).

The solution's orchestration is defined in a *YAML* file and then started with a single command. To improve security, only the needed container ports are exposed. To ensure data integrity throughout service disruptions, persistence data is mapped to folder in the OS. To ensure an easy management of the environment, configurations are kept in the OS and fetched by each container once they start.

The details about the solution orchestration can be found in Section K.5.

K.3 Usage of Nginx as a web server and reverse proxy

To serve the frontend pages and redirect requests made to backend containers, the following technologies were analyzed:

- *Nginx*;
- *Apache HTTP Server Project*;
- *Lighttpd*.

All of them support the necessary requirements, but some factors lead the author to pick Nginx over the others, the following table, Table K.1, describes this criteria.

Table K.1: Technologies Comparison - Reverse Proxy Web Server

Criteria/Technology	Nginx	Apache HTTP Server	Lighttpd
Resource Consumption	low	high	medium
Community Size	high	very high	medium
Familiarity with the tool	high	low	low

The details about *Nginx* adoption and configuration can be found in Section K.6.

K.4 Sensae Console Containerization

The section describes how **Sensae Console** is containerized with docker. As explained in Section K.1, the author choose to containerize the solution.

The following Code Samples describe how each container mentioned during the Design Chapter are packaged. To simplify, only three distinct samples will be presented.

The first sample, Listing K.1, refers to UI Aggregator and is similar to all other frontend containers.

```
1  FROM node:18-alpine AS build
2  WORKDIR /workspace
3  COPY package.json ./
4  COPY . .
5  RUN npm install
6  RUN npm run nx build ui-aggregator --omit=dev
7
8  FROM nginx:1.23.1
9  COPY apps/ui-aggregator/nginx/nginx.conf /etc/nginx/conf.d/default.conf
10 COPY --from=build /workspace/dist/apps/ui-aggregator /usr/share/nginx/
       html
```

Listing K.1: Dockerfile for UI Aggregator Frontend

This Dockerfile contains two stages to reduce the size of the final image. The first stage, lines **1** to **6**, builds the project. The second one, containing only *Nginx* and the code that was previously built, is used to serve the UI Aggregator Frontend and route requests. The *Nginx* configuration file at line **9** is discussed in the K.6 Section.

The second sample, Listing K.2, refers to **Fleet Management Backend** and is similar to all backend containers in the Configuration Scope or External Services.

```
1  FROM maven:3.8.5-openjdk-18 AS build
2  WORKDIR /app
3  # copy all pom.xml to pull only external dependencies
4  COPY application/pom.xml application/pom.xml
5  COPY domain/pom.xml domain/pom.xml
6  COPY infrastructure/boot/pom.xml infrastructure/boot/pom.xml
7  COPY infrastructure/endpoint/pom.xml infrastructure/endpoint/pom.xml
8  COPY infrastructure/persistence/pom.xml infrastructure/persistence/pom.
       xml
9  COPY infrastructure/persistence/questdb/pom.xml infrastructure/
       persistence/questdb/pom.xml
10 COPY infrastructure/endpoint/graphql/pom.xml infrastructure/endpoint/
       graphql/pom.xml
11 COPY infrastructure/endpoint/amqp/pom.xml infrastructure/endpoint/amqp/
       pom.xml
12 COPY infrastructure/pom.xml infrastructure/pom.xml
13 COPY pom.xml pom.xml
14 # build all external dependencies
15 RUN mvn -B -e -C org.apache.maven.plugins:maven-dependency-plugin:3.1.2:
       go-offline -DexcludeArtifactIds=fleet-management-backend,application,
       domain,infrastructure,endpoint,graphql,boot,amqp,questdb
16
17 COPY . .
18 RUN mvn clean package
19
20 FROM openjdk:17
21 WORKDIR /app
22 COPY --from=build /app/infrastructure/boot/target/fleet-management-
       backend.war /app
23 CMD ["java", "-jar", "fleet-management-backend.war"]
```

Listing K.2: Dockerfile for Fleet Management Backend

This sample also presents a multi-stage Dockerfile. The first stage, line **1** to **18** builds the project with Maven. All *pom.xml* files and dependencies are added first to reduce build time during development, since these change less that the code written. The second stage is the one that runs the service. It only contains the Java Development Kit (JDK) and the compiled application.

The third sample, Listing K.3, refers to **Device Commander** and is similar to all backend containers in the Data Flow Scope.

```
1  FROM quay.io/quarkus/ubi-quarkus-native-image:22.1-java17 AS build
2  COPY --chown=quarkus:quarkus mvnw /code/mvnw
3  COPY --chown=quarkus:quarkus .mvn /code/.mvn
4  COPY --chown=quarkus:quarkus pom.xml /code/
5  USER quarkus
6  WORKDIR /code
7  RUN ./mvnw -B org.apache.maven.plugins:maven-dependency-plugin:3.1.2:go-
       offline
8  COPY src /code/src
9  RUN ./mvnw package -Pnative
10
11 FROM quay.io/quarkus/quarkus-micro-image:1.0
12 WORKDIR /work/
13 COPY --from=build /code/target/runner /work/application
```

```
14
15 # set up permissions for user `1001`
16 RUN chmod 775 /work /work/application \
17     && chown -R 1001 /work \
18     && chmod -R "g+rwX" /work \
19     && chown -R 1001:root /work
20
21 EXPOSE 8080
22 USER 1001
23
24 CMD ["./application", "-Dquarkus.http.host=0.0.0.0"]
```

Listing K.3: Dockerfile for Device Commander

This sample, once again, is also a multi-stage Dockerfile. It was adapted from the one generated by *Quarkus* when setting up the application. In the first stage the application is built with a *GraalVM* native-image - lines **1** to **9**. This allows the image to run without Java Virtual Machine (JVM). The second stage runs the service after setting user permissions, so that the process doesn't run as root, at lines **17** to **20**.

K.5 Sensae Console Orchestration

As described in Section 4.2.2.4, *Overview of Docker Compose* was the tool used to orchestrate the whole solution, the **Sensae Console** and External Services. This tool consumes a configuration file to know what containers, and their configurations, are needed. The complete configuration file for production is vast, a summarized version will be presented containing only the **Data Processor** Context' related containers.

```
1  services:
2    data-processor-frontend:
3      build:
4        dockerfile: docker/data-processor-frontend/Dockerfile
5        context: frontend-services
6      image: data-processor-frontend
7      volumes:
8        - /etc/letsencrypt:/etc/letsencrypt/
9        - /etc/nginx/ssl:/etc/nginx/ssl/
10     networks:
11       - sensae-network
12     ports:
13       - 443
14     depends_on:
15       - data-processor-master-backend
16   data-processor-master-backend:
17     build: backend-services/data-processor-master-backend
18     image: data-processor-master-backend
19     volumes:
20       - ./secrets/keys:/etc/ssh/app
21     environment:
22       spring_profiles_active: prod
23     env_file:
24       - ./secrets/prod/data-processor-master-backend.env
25     networks:
26       - sensae-network
27     ports:
28       - 8080
29   data-processor-database:
```

```
30      build: databases/data-processor-database
31      container_name: data-processor-database
32      env_file:
33        - ./secrets/prod/data-processor-database.env
34      networks:
35        - sensae-network
36      ports:
37        - 5482:5432
38      volumes:
39        - ./databases-data/prod/data-processor-database:/var/lib/
      ↪ postgresql/data/
40    data-processor-flow:
41      build: backend-services/data-processor-flow
42      image: sensae/data-processor-flow
43      env_file:
44          - ./secrets/prod/data-processor-flow.env
45      networks:
46          - sensae-network
47  networks:
48    sensae-network:
```

Listing K.4: Docker Compose Configuration File for Production

The following conclusions can be observed:

- This context, similar to other contexts, is composed by four containers, a Frontend - *data-processor-frontend*, a Configuration Backend - *data-processor-master-backend*, a Database - *data-processor-database*, and a Data Flow Backend - *data-processor-flow*;
- All services communicate in the same network - *sensae-network*;
- All services have instructions on how to build them;
- Various configuration files are loaded, e.g. in lines **19** to **20** and **28** to **31**, this files content will be discussed in the Sensae Console Configuration Files Section;
- The Frontend has two volumes mapped, one loads the *letsencrypt* configuration file for *Nginx* and the other loads the SSL certificate - lines **7** to **9**.
- The Configuration Backend needs to validate the authentication tokens received, for that, it has access to the public key that pairs the private key used to created then in **Identity Management Backend** - line **19** - **20**;
- The database exposes a port to the host so that it can be managed remotely - lines **36** to **37**;
- The database maps its data to a directory in the host, so that data is persisted between server restarts - lines **38** to **39**;
- The Data Flow container doesn't need to expose any port since it only exchanges information with the message broker;

K.6 Sensae Console Reverse Proxy Configuration

This section reveals how *Nginx* is configured for all frontend containers in the solution. As an example, the Listing K.5, describes the **Smart Irrigation Frontend**.

```
server {

    server_name localhost;

    listen 443 ssl;

    ssl_certificate /etc/nginx/ssl/nginx.crt;
    ssl_certificate_key /etc/nginx/ssl/nginx.key;

    root        /usr/share/nginx/html;

    index       index.html index.htm;

    include /etc/letsencrypt/options-ssl-nginx.conf;

    location ~ .*remoteEntry.js$ {
        expires -1;
        add_header 'Cache-Control' 'no-store, no-cache, must-revalidate,
    proxy-revalidate, max-age=0';
    }

    location /smart-irrigation/graphql {
        proxy_pass http://smart-irrigation-backend:8080/graphql;
        proxy_set_header x-forwarded-prefix /smart-irrigation/graphql;
        proxy_set_header Host $host;
        proxy_set_header x-forwarded-host $host;
        proxy_redirect off;
        proxy_set_header x-forwarded-port 443;
        proxy_set_header x-forwarded-proto https;
    }

    location /smart-irrigation/subscriptions {
        proxy_pass http://smart-irrigation-backend:8080/subscriptions;
        proxy_set_header x-forwarded-prefix /smart-irrigation/
    subscriptions;
        proxy_http_version 1.1;
        proxy_set_header Upgrade $http_upgrade;
        proxy_set_header Connection "Upgrade";
        proxy_set_header Host $host;
        proxy_read_timeout 6000;
        proxy_send_timeout 6000;
        proxy_redirect off;
        proxy_set_header x-forwarded-port 443;
        proxy_set_header x-forwarded-proto https;
    }

    location / {
        try_files $uri $uri/ /index.html;
    }

    if ($scheme != "https") {
        return 301 https://$host$request_uri;
    } # managed by Certbot
}
```

Listing K.5: Configuration File for Production Environment

The following conclusions can be inferred:

- It only exposes the HTTPS port - line **4** and lines **49** to **51**;
- It loads the SSL certificates mapped in the *Overview of Docker Compose* file - lines **7** and **8**;
- It uses the *letsencrypt* configuration - line **14**;
- The *remoteEntry* file, responsible for providing the entry point to the service in a Micro Frontend environment, is never cached in the client browser since it points to the current compiled version of the service. If this file is cached, the updated version of a micro frontend, can only be accessed by the client browser once the local cache is cleaned up - lines **16** to **19**;
- The *GraphQL* endpoint is defined as a reverse proxy endpoint. Requests made to */smart-irrigation/graphql* are routed to *http://smart-irrigation-backend:8080/graphql*. It doesn't use a secure connection, HTTPS, since this communication already happens inside the docker network where man in the middle attacks are disregarded - lines **21** to **29**;
- The *GraphQL* subscription endpoint is also defined, this type of connection, *Websocket*, requires the use of HTTP version 1.1 and the two Headers presented at lines **34** to **36**;
- All other requests are handled in lines **45** to **47**.

K.7 Sensae Console Configuration Files

This section describes how a **Sensae Console** and External Services are configured. One of the problems that arise from a microservice architecture is how to maintain all configurations for each container developed and configured. Following the *Externalized configuration Pattern*, all configurations are defined via configuration files that support three environments: *dev*, *test* and *prod*.

This configurations are defined, for each environment, in a single file. This file, Listing K.6, has the following structure:

```
1  export SENSAE_MAPBOX_ACCESS_TOKEN=
2  export SENSAE_MAPBOX_SIMPLE_STYLE=
3  export SENSAE_MAPBOX_SATELLITE_STYLE=
4  export SENSAE_BROKER_USERNAME=
5  export SENSAE_BROKER_PASSWORD=
6  export SENSAE_COMMON_DATABASE_PASSWORD=
7  export SENSAE_DATA_STORE_USER_PASSWORD=
8  export SENSAE_DATA_STORE_ROOT_PASSWORD=
9  export SENSAE_AUTH_PATH_PUB_KEY=
10 export SENSAE_AUTH_PATH_PRIV_KEY=
11 export SENSAE_AUTH_ISSUER=
12 export SENSAE_AUTH_AUDIENCE=
13 export SENSAE_DATA_AUTH_KEY=
14 export SENSAE_AUTH_EXTERNAL_MICROSOFT_AUDIENCE=
15 export SENSAE_AUTH_EXTERNAL_GOOGLE_AUDIENCE=
16 export SENSAE_SMS_TWILIO_ACCOUNT_SID=
17 export SENSAE_SMS_TWILIO_AUTH_TOKEN=
18 export SENSAE_SMS_SENDER_NUMBER=
19 export SENSAE_SMS_ACTIVATE=
20 export SENSAE_EMAIL_SENDER_ACCOUNT=
```

```
export SENSAE_EMAIL_SUBJECT=
export SENSAE_EMAIL_SENDER_PASSWORD=
export SENSAE_EMAIL_SMTP_HOST=
export SENSAE_EMAIL_SMTP_PORT=
export SENSAE_EMAIL_ACTIVATE=
export SENSAE_PROD_PUBLIC_DOMAIN=
export SENSAE_ADMIN_EMAIL=
```

Listing K.6: Configuration File for Production Environment

This file variables are then passed on to each container's environment configuration file with the help of a script. The Code Sample K.7 sheds a light on how the script propagates the configurations.

```
#!/usr/bin/sh

ROOT_DIR=$(git rev-parse --show-toplevel)

cd "$ROOT_DIR"/project || exit

. ./secrets/prod.conf

SECRET_BACK=secrets/templates/prod/backend-services
SECRET_FRONT=secrets/templates/prod/frontend-services
SECRET_DB=secrets/templates/prod/databases

BACK_PREFIX=secrets/prod
FRONT_PREFIX=frontend-services/apps
FRONT_SUFFIX=src/environments/environment.prod.ts

envsubst < $SECRET_BACK/alert-dispatcher-backend.env > \
     $BACK_PREFIX/alert-dispatcher-backend.env
# and all other backend services
envsubst < $SECRET_BACK/data-validator.env >
     $BACK_PREFIX/data-validator.env

envsubst < $SECRET_FRONT/device-management-frontend.ts > \
 $FRONT_PREFIX/device-management-frontend/$FRONT_SUFFIX
# and all other frontend services
envsubst < $SECRET_FRONT/ui-aggregator.ts > \
 $FRONT_PREFIX/ui-aggregator/$FRONT_SUFFIX

envsubst < secrets/templates/prod/message-broker/message-broker.env > \
 $BACK_PREFIX/message-broker.env

envsubst < $SECRET_DB/data-decoder-database.env > \
 $BACK_PREFIX/data-decoder-database.env
# and all other databases
envsubst < $SECRET_DB/rule-management-database.env > \
 $BACK_PREFIX/rule-management-database.env
```

Listing K.7: Configuration Propagation Script

In the future, as more isolated deployments are made, a tool such as *Vault* should be integrated in the solution.

Appendix L

Sensae Console Database Configuration

The solution designed relies on various databases, and as discussed in Section 5.1.6.1 some are relational databases. *PostgresSQL* and most databases of this data-model type require a database schema. For this solution the schema of each database is defined in a *sql* file that is executed at the start of the database, only if no data is found.

Further database schema migrations are preformed using custom SQL scripts when needed. In the future, once more instance of **Sensae Console** are deployed, the use of liquidbase or flyway is preferred.

The following Code Sample L.1 exemplifies the content of this scripts.

```
create table if not exists public.transformation
(
    persistence_id bigint generated by default as identity
        primary key,
    device_type    varchar(255)
        constraint unique_type_constrain
            unique
);

create table if not exists public.property_transformation
(
    persistence_id                bigint generated by default as
    identity (maxvalue 2147483647)
        primary key,
    value                         integer           not null,
    old_path                      varchar(255),
    transformation_persistence_id bigint
        constraint ref_transformation_constrain
            references public.transformation,
    sub_sensor_id                 integer default 0 not null
);
```

Listing L.1: Initialization Script Segment for Data Processor Database

This script defines two simple tables, *transformation* and *property_transformation*, following the concepts defined in Section G.1.

Apart from the schema, the **Identity Management Database** also requires the following bootstrap data, as implied in Identity Management Bounded Context Section:

- Root domain;

- Public domain;
- Unallocated Root domain;
- Anonymous Tenant account;
- Admin Tenant account;

This data is inserted using the following function, Code Sample L.2:

```
1  CREATE FUNCTION public.init_domains ()
2  RETURNS varchar(255) AS $root_oid$
3      DECLARE
4          root_oid varchar(255) := gen_random_uuid();
5          public_oid varchar(255) := gen_random_uuid();
6          unallocated_oid varchar(255) := gen_random_uuid();
7      BEGIN
8          INSERT INTO public.domain (name, oid, path)
9          VALUES ('root', root_oid, ARRAY[root_oid]);
10         INSERT INTO public.domain (name, oid, path)
11         VALUES ('public', public_oid, ARRAY[root_oid, public_oid]);
12         INSERT INTO public.domain (name, oid, path)
13         VALUES ('unallocated', unallocated_oid, ARRAY[root_oid,
       unallocated_oid]);
14         INSERT INTO public.tenant (name, oid, phone_number, email,
       domains)
15         VALUES ('Anonymous', gen_random_uuid(), '', '', ARRAY[public_oid
       ]);
16         INSERT INTO public.tenant (name, oid, phone_number, email,
       domains)
17         VALUES ('Admin', gen_random_uuid(), '', '$SENSAE_ADMIN_EMAIL',
       ARRAY[root_oid]);
18         RETURN root_oid;
19     END;
20 $root_oid$ LANGUAGE plpgsql;
21
22 select public.init_domains();
23
24 DROP FUNCTION public.init_domains;
```

Listing L.2: Bootstrap function for Identity Management Database

This function starts by declaring three UUID - lines **4** to **6** - that will later be used to populate the domain's *path* and the tenant's *domains* - lines **7** to **17**. In the end the function is executed and then removed to ensure that it isn't executed again.

In line **17**, the variable **$SENSAE_ADMIN_EMAIL** is replace by a valid email before building the database container with the full script. This variable configuration is discussed in the Section K.7.

Appendix M

Performance Tests Specification

```
//imports

export const options = {
   setupTimeout: "2m",
   scenarios: {
      subscribe: {
      executor: "shared-iterations",
      startTime: "0s",
      vus: 1,
      iterations: 1,
      maxDuration: "3m",
      exec: "subscribe",
   },
      ingestion: {
      executor: "per-vu-iterations",
      vus: 100,
      iterations: 100,
      startTime: "5s",
      exec: "ingestion",
      maxDuration: "3m",
   },
      consumption: {
      executor: "shared-iterations",
      startTime: "3m",
      vus: 1,
      iterations: 1,
      maxDuration: "10s",
      exec: "consumption",
      },
   },
};

const timeLapseTrend = new Trend("time_lapse");

const sampleSize = new SharedArray("sampleSize", function () {
   const sampleSize = [];
   sampleSize.push(options.scenarios.ingestion.vus *
      options.scenarios.ingestion.iterations
   );
   return sampleSize;
});

const dataIds = new SharedArray("dataIds", function () {
   const dataIds = [];
   const numberOfDataUnits = options.scenarios.ingestion.vus *
      (options.scenarios.ingestion.iterations + 2);
   for (let index = 0; index < numberOfDataUnits; index++)
      dataIds.push(randomId());
   return dataIds;
});

const data = new SharedArray("data", function () {
   const data = [];
   const total = options.scenarios.ingestion.vus + 2;
   for (let index = 0; index < total; index++)
      data.push(createDevice("em300th", index, true));
   return data;
});

export function subscribe() {
   const res = http.post(`http://${__ENV.SENSAE_INSTANCE_IP}:8086/graphql`, anonymousLoginQuery, {
      headers: { "Content-Type": "application/json" },
   });

   let received = [];
   ws.connect(
      `ws://${__ENV.SENSAE_INSTANCE_IP}:8801/subscriptions`,
      {
         headers: {
            "Sec-WebSocket-Protocol": "graphql-transport-ws",
         },
      },
      (socket) => {
```

```
74            socket.on("message", (msg) => {
75               const message = JSON.parse(msg);
76               if (message.type == "next") {
77                  timeLapseTrend.add(new Date().getTime() -
78                     message.payload.data.data.reportedAt
79                  );
80                  received.push(message.payload.data.data.dataId);
81                  if (received.length === sampleSize[0])
82                     closeSocket(socket, received);
83               }
84            });
85            socket.on("open", () => {
86               socket.send(initSubscription());
87               socket.send(
88                  createSubscription(subscribeToLiveDataQuery, {
89                     filters: createLiveDataFilters(data),
90                     Authorization:
91                        "Bearer " + JSON.parse(res.body).data.anonymous.token,
92                     })
93               );
94            });
95            socket.setTimeout(() =>
96               closeSocket(socket, received), 300000);
97         }
98      );
99   }
100
101  export function closeSocket(socket, received) {
102     check(received, { "data units were received":
103        (rec) => rec.length === sampleSize[0],
104     });
105     received.forEach((dataId) => {
106        check(dataId, {
107           "data units was sent": (id) => dataIds.includes(id),
108        });
109     });
110     socket.close();
111  }
112
113  export function ingestion() {
114     const vu = exec.vu.idInTest - 1; //vus start at 1, arrays at 0;
115     const device = data[vu];
116     const id = dataIds[vu + (data.length - 2) *
117        exec.vu.iterationInScenario];
118
119     sleep(device.interval);
120     const res = http.post(
121        `https://${__ENV.SENSAE_INSTANCE_IP}:8443/sensor-data/${device.channel}/${device.data_type}/${
         ↪ device.device_type}`,
122        randomBody(id, device),
123        {
124           headers: {
125              Authorization: `${__ENV.SENSAE_DATA_AUTH_KEY}`,
126              "Content-Type": "application/json",
127           },
128        }
129     );
130     check(res, { "status was 202": (r) => r.status === 202 });
131  }
132
133  export function consumption() {
134     var numberEntries = countSmartIrrigationMeasuresEntries();
135     check(numberEntries, {
136        "data units were all stored": (res) => res === sampleSize[0],
137     });
138  }
139
140  export function setup() {
141     initSmartIrrigationDatabase();
142     data.forEach(insertDevice);
143     data.forEach(moveDeviceToPublicDomain);
144     givePermissionsToPublicDomain();
145     createEM300THProcessor();
146     createEM300THDecoder();
147  }
148
149  export function teardown() {
150     clearDevices();
151     clearProcessors();
152     clearDecoders();
153     clearDomainsDevicesTenants();
154     resetIdentity();
155     clearIrrigationData();
156  }
```

Listing M.1: Smart Irrigation Performance Test Scenario Description

Appendix N

Performance Tests Analysis

```
## Import Libraries

prepare <- function(path) {
  data <- read.csv(path)
  data <- data[data$metric_name == 'time_lapse',]
  data <- data[c('timestamp', 'metric_value', 'extra_tags')]
  data$received <- data$timestamp - min(data$timestamp)
  data$metric_value <- data$metric_value / 1000
  data$sent_timestamp <- data$timestamp - data$metric_value
  data$sent <- data$sent_timestamp - min(data$sent_timestamp)
  data$iteration <- str_replace(data$extra_tags,'iteration=',')
  return(data)
}

create <- function(dataframe, xParam) {
  ggplot(data=dataframe, mapping=aes(x=.data[[xParam]], y=metric_value, col=iteration, label="")) +
    geom_point(alpha = 1, stat = "unique") +
    theme(legend.position = c(.9, .45)) +
    xlab(paste("time data unit was", xParam, "(seconds)")) +
    ylab("time taken to process data unit (seconds)")
}

outputTex <- function(pdot, path, xParam) {
  tikz(file = path, width = 5, height = 3.3)
  print(pdot)
  dev.off()
}

process <- function(path) {
  data <- prepare(paste(path,'data.csv', sep = "/"))
  pdot_sent <- create(data, "sent")
  pdot_received <- create(data, "received")
  outputTex(pdot_sent, paste(path, 'data_sent.tex', sep="/"))
  outputTex(pdot_received, paste(path, 'data_received.tex', sep="/"))
}

processScenario <- function(path) {
  scenario <- list.dirs(path)
  scenario <- scenario[-1]
  for (i in scenario) {
    process(i)
  }
}

processAll <- function() {
  processScenario('scenario1')
  processScenario('scenario2')
  processScenario('scenario3')
}

setwd("/home/user/iot-project/project/k6/results")

processAll()
```

Listing N.1: Analysis Script

Appendix O

Fire Detection Simulation Report

This Appendix presents the report related to the first fire detection simulation made inside a costumer's Chicken Farm. This report lead to the creation of a rule scenario to alert the costumer of possible fire outbreaks in the farm.

Fire Detection Simulation (#1)- Sensae

Following is the variable analysis of the fire detection simulation which took place at 16:00 on 17/05/2022 for a duration of 20 min. The fire was contained in a relatively small surface area of <1m^2. The burning material composition was limited to pine wood. The experiment took place between sensors #2 and #3 which are mounted at a height of 11 m.

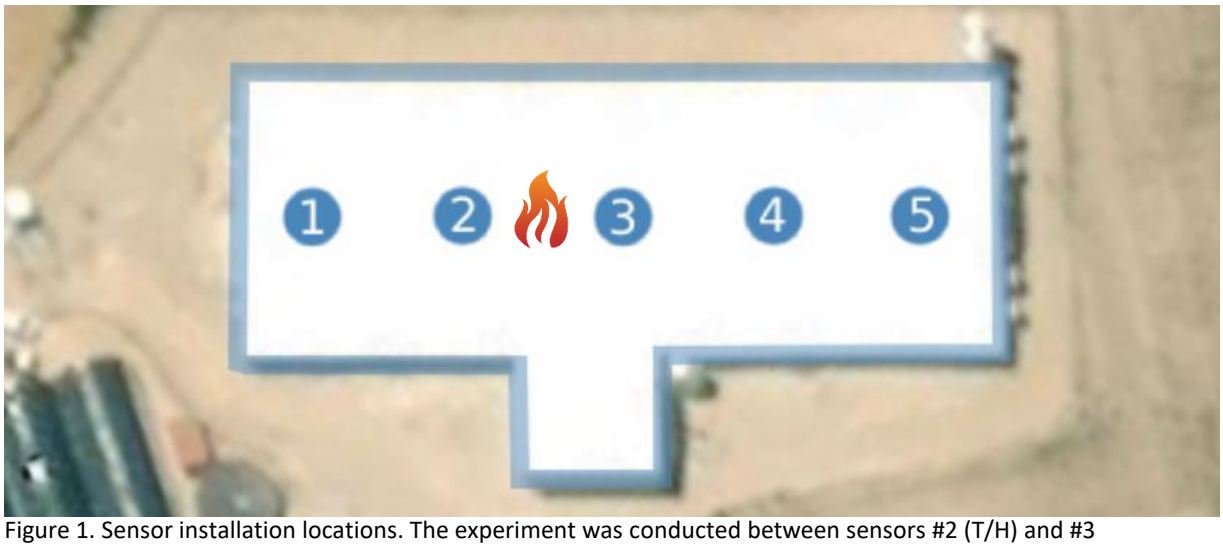


Figure 1. Sensor installation locations. The experiment was conducted between sensors #2 (T/H) and #3 (CO2/T/H/Pressure).

Results

Relative Humidity

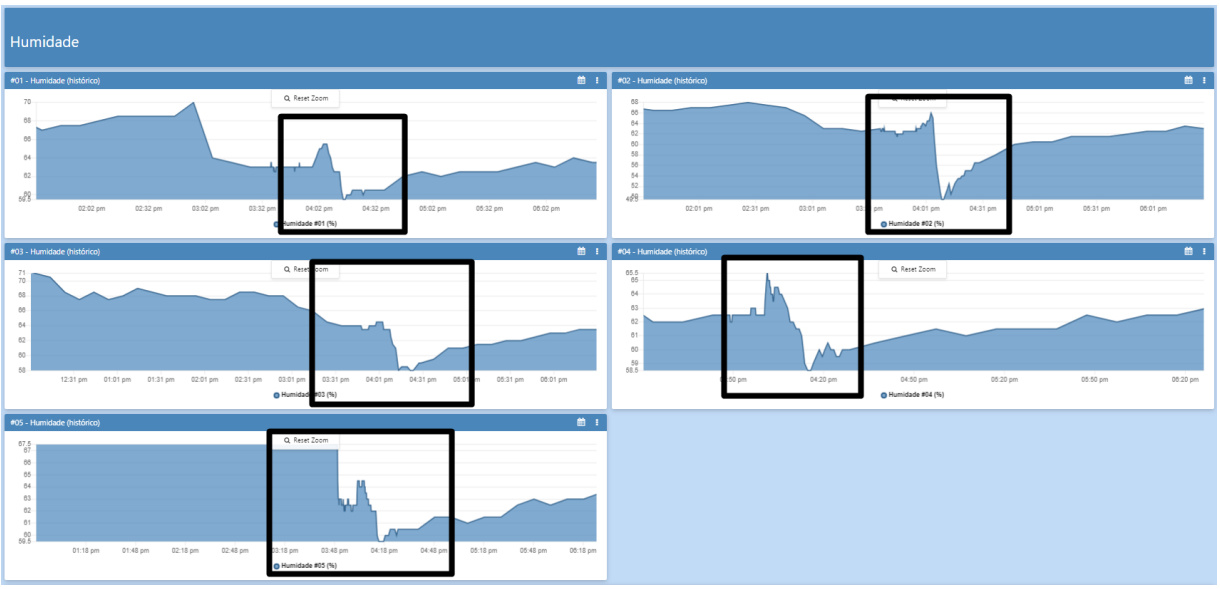


Figure 2. Relative Humidity variation during the experiment. A slight increase before followed by a drop after the experiment can be observed across all sensors.

#1 Before: (64 %); After: (60 %); (4:00 – 4:14PM); (dt: 14 min); (d%: 4%)
#2 Before: (64 %); After: (50 %); (4:00 – 4:09PM); (dt: 9 min); (d%: 14%)
#3 Before: (65 %); After: (56 %); (4:03 – 4:14PM); (dt: 11 min); (d%: 9%)
#4 Before: (65 %); After: (59 %); (4:00 - 4:12PM); (dt: 12 min); (d%: 6%)
#5 Before: (63 %); After: (60 %); (4:00 – 4:14PM); (dt: 14 min); (d%: 3%)

Temperature:

Figure 3. Temperature variation during the experiment. A slight temperature increase can be observed during the experiment across all sensors.

#1 Before: (23.1 °C); After: (24.3 °C); (4:00 – 4:14PM); (dt: 14 min); (dT: 1.2°C)
#2 Before: (23.1 °C); After: (28.8 °C); (4:00 – 4:08PM); (dt: 8 min); (dT: 5.7°C)
#3 Before: (22.9 °C); After: (25.8 °C); (4:03 – 4:14PM); (dt: 11 min); (dT: 2.9°C)
#4 Before: (23.0 °C); After: (24.0 °C); (4:00 - 4:12PM); (dt: 12 min); (dT: 1.0°C)
#5 Before: (23.0 °C); After: (24.0°C); (4:00 – 4:10PM); (dt: 10 min); (dT: 1.0°C)

CO2 / Pressure

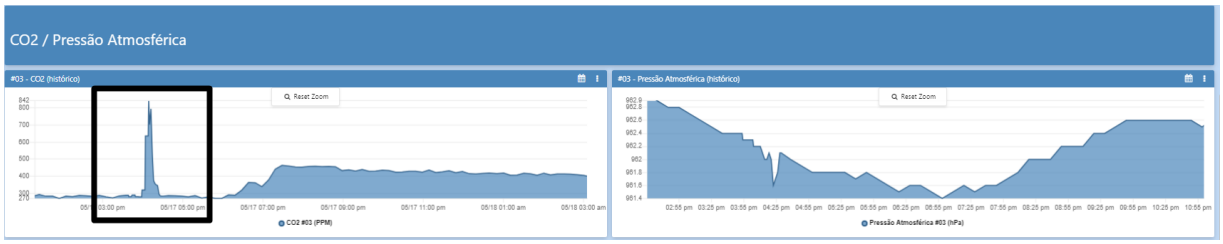


Figure 4. CO2 variation during the experiment. A large increase in CO2 concentration can be observed in #3.

#3 Before: (319 PPM); After: (842 PPM); (4:00 – 4:06PM); (dt: 6 min); (dPPM:523)

There is no observable difference in atmospheric pressure.

We can observe that there is a large variability in values obtained of which the magnitude is reversely correlated to the distance of the burning epicenter. Sensors closer to the burning epicenter display larger numerical outliers.

Temperature

It can be observed that within 8-10 min after the start of the experiment all sensors display an increase between 1 – 5.7°C as a result of the burning fire. Sensor #2 which is a T/H sensor and located downwind to the burning fire also detected the larger increase in temperature of 5.7 °C or **0.71°C/min.** This value is beyond what is expected during normal weather variations.

Humidity

It can be observed that within 8-10 min after the start of the experiment all sensors display a 3 - 14% drop in humidity with the larger drop occurring from sensor #2 at 14% 9 min after start of the experiment. This equates to a drop of **1.55%/min**. This value is beyond what is expected during normal weather variations.

CO_2

It can be observed that within 6 min after the start of the experiment the CO_2 sensor displays a increase in concentration from 319 PPM to 842 PPM after which it steadily decreases to pre-experimental values. This equates to a increase of **87 PPM/min**. This value is beyond what is expected during normal weather variations.

Discussion

The fire simulation in this experiment was of relatively small scale compared to the 3000m^2 chicken farm the experiment was carried out in. This is however a very valuable experiment to calibrate the detection algorithms to detect the fire as early as possible.

From the results we can observe a change in both Temperature, Humidity and CO_2 as a consequence of the fire. Furthermore their rate of change (ROC) is beyond what is expected during normal conditions. From the measured variables, the CO_2 concentration displays the largest relative increase and rate of change (87 PPM/min) followed by the humidity (1.55%/min) and temperature (0.71°C/min).

To optimize the detection window we suggest to not only monitor for the magnitude of the measured variables, however, we also advise to take into account the 'Rate of Change (ROC)' (slope of curve) in the detection algorithm. This way, alerts can be set for variables that change rapidly. This way outlier events like fires can be detected sooner. We furthermore suggest to carefully correlate the rate of change between monitored variables to increase the robustness of the detection algorithm.

Sound Level and Fire Detection

When animal life is present during a fire, the increase of smoke, CO_2 and other volatile elements makes it hard to breath. We hypothesize that this will significantly increase the desire to flee and thus greatly increase their sound output. We therefore suggest to include sound level in the fire detection algorithm. Beyond fire detection monitoring sound level of animal activity provides further important insight into the animal state of mind.

Points of consideration

1. The experiment was executed during stable/ideal conditions. The effect on the obtained data of having biological life active in the experiment area cannot be neglected.
2. Very little ventilation was active during the experiment. During normal operation the additional ventilation provides a significant oxygen supply which can significantly increase the burning rate and thus impact the rate of change of the monitored variables.

Suggestions for further experiments

After implementing the current optimizations we suggest the following improvements:

1. Decrease the detection time of the CO_2 Sensor.
2. Install at least 1 additional CO_2 sensor to increase the detection area.
3. Take into consideration the sound level.
4. Explore the possibility of adding a water resistant photoelectric sensor, will this work?
5. Experiment with TIR sensors, how does biological life impact the measured values?