

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:
TODO, Professor, DEI/ISEP

Members:

TODO, Professor, DEI/ISEP
TODO, Professor, DEI/ISEP
TODO, Professor, DEI/ISEP

*Experience is merely the name
men gave to their mistakes.*

OSCAR WILDE

Dedicatory

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, my girlfriend, that closely accompanied me in this journey.

Thank you

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 these 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 these platforms are tailor made for the devices that the company offers. This dissertation presents a platform focused on enabling others to create IoT-based services and three Proof of Concept (PoC) services built on top of this platform. 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, Stream Processing, 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

TODO

Contents

List of Figures	xvii
List of Tables	xxi
List of Source Code	xxiii
1 Introduction	1
1.1 Context	1
1.2 Problem	2
1.3 Objectives	2
1.4 Approach	3
1.5 Achieved Results	4
1.6 Document Structure	5
2 State of the Art	7
2.1 Internet of Things	7
2.1.1 Brief Description	7
2.1.2 Practical Applications	7
2.1.3 Enterprise Challenges	7
2.1.4 Renowned Solutions	7
2.2 Big Data	7
2.2.1 Brief Description	7
2.2.2 Challenges	7
2.3 Synopsis	7
3 Analysis	9
3.1 Business Analysis	9
3.1.1 Fleet Management	9
3.1.2 Smart Irrigation	9
3.1.3 Fire Outbreak Surveillance	9
3.2 Technical Analysis	9
3.2.1 Data Aggregation	9
3.2.2 Data Filtering	9
3.2.3 Data Storage	9
3.2.4 Data Transformation	9
3.2.5 Data Analysis	9
3.2.6 Data Presentation	9
3.2.7 Trigger Warning System	9
3.2.8 User Authentication/Authorization	9
3.3 Synopsis	9

4 Requirements Elicitation	11
4.1 Functional Requirements	12
4.1.1 Roles	13
4.1.2 Sensae Console	13
4.1.2.1 Manager	13
4.1.2.2 Costumer	14
4.1.3 External Services	14
4.1.3.1 Fleet Management	15
4.1.3.2 Smart Irrigation	15
4.1.3.3 Indoor Fire Detention	15
4.2 Non Functional Requirements	16
4.2.1 Functionality Requirements	16
4.2.2 Usability Requirements	16
4.2.3 Reliability Requirements	17
4.2.4 Performance Requirements	17
4.2.5 Supportability Requirements	17
4.2.6 Design Requirements	18
4.2.7 Implementation Requirements	18
4.2.8 Interface Requirements	18
4.2.9 Physical Requirements	19
4.3 Synopsis	19
5 Design	21
5.1 System Scopes	21
5.1.1 Configuration Scope	23
5.1.2 Data Flow Scope	23
5.1.3 External Services Scope	24
5.2 Architectural Design	24
5.2.1 C4 Level 1 - Context	25
5.2.1.1 Context Level - Logical View	25
5.2.1.2 Context Level - Physical View	26
5.2.1.3 Context Level - Synopsis	27
5.2.2 C4 Level 2 - Containers	27
5.2.2.1 Container Level - Logical View NEW	28
5.2.2.2 Container Level - Logical View - OLD	36
5.2.2.3 Container Level - Process View	41
5.2.2.4 Container Level - Implementation View	53
5.2.2.5 Container Level - Physical View	56
5.2.2.6 Container Level - Synopsis	57
5.2.3 C4 Level 3 - Components	57
5.2.3.1 Components Level - Logical View	58
5.2.3.2 Components Level - Process View	62
5.2.3.3 Components Level - Implementation View	66
5.2.3.4 Components Level - Synopsis	68
5.3 Architectural Alternatives Discussed	68
5.3.1 Backend Segregation	69
5.3.2 Frontend Segregation	71
5.3.3 User Authentication/Authorization	71
5.3.3.1 Internal Authentication Server	72

5.3.3.2	External Authentication Server	73
5.3.3.3	External Authentication Server with Internal Authorization Server	75
5.3.3.4	External Authentication Server with Internal OAuth2 Server	76
5.3.4	Data Flow Pipeline	76
5.3.5	Internal Communication	77
5.3.5.1	First Option	77
5.3.5.2	Second Option	78
5.3.5.3	Third Option	79
5.3.5.4	Fourth Option	80
5.3.5.5	Fifth Option	80
5.4	Domain	81
5.4.1	Taxonomy	81
5.4.2	Shared Model	82
5.4.2.1	Data Model	83
5.4.2.2	Message Envelope Model	88
5.4.2.3	Routing Model	88
5.4.3	Sensae Console Bounded Context	93
5.4.3.1	Data Processor	94
5.4.3.2	Data Decoder	96
5.4.3.3	Device Management	97
5.4.3.4	Identity Management	98
5.4.3.5	Rule Management	100
5.4.4	External Services Bounded Contexts	101
5.4.4.1	Notification Management	101
5.4.4.2	Smart Irrigation	102
5.4.4.3	Fleet Management	104
5.5	Synopsis	105
6	Implementation	107
6.1	Technical Decisions	107
6.1.1	Backend Technologies Usage throughout the Solution	108
6.1.1.1	Programming Language Used	108
6.1.1.2	General Backend Services	108
6.1.1.3	Data Flow Scope Backend Services	108
6.1.2	Frontend Technologies Usage through the Solution	109
6.1.2.1	Programming Language and Framework Used	109
6.1.2.2	Technologies used to create a Micro Frontend Architecture	110
6.1.2.3	Technologies used to build and manage the Frontend Services	110
6.1.2.4	Technologies used to provide map/location base services	110
6.1.3	Backend Services Expose a GraphQL API	110
6.1.4	Usage of RabbitMQ to support Internal Communication	111
6.1.5	Usage of Protocol Buffers in Internal Communication	113
6.1.6	Database Usage throughout the Solution	113
6.1.6.1	Relational Database Usage	114
6.1.6.2	Document-based Database Usage	114
6.1.6.3	Column-based Database Usage	115
6.1.6.4	Graph-based Database Usage	116

6.1.7	Rules Script Engine	116
6.1.8	Data Decoders Script Engine	117
6.1.9	Containerization of services via Docker	117
6.1.10	Orchestration of services via Docker Compose	118
6.1.11	Usage of Nginx as a web server and reverse proxy	119
6.1.12	Usage of Git as a version control system of the project	119
6.1.13	Usage of Github Issues to track issues, bugs and new features	120
6.1.14	Usage of Github Actions for CI/CD	122
6.1.15	Usage of Maven Repository to host Open-Source Code	125
6.2	Technical Description	125
6.2.1	Sensae Console UI	126
6.2.2	Sensae Console Custom Maps	127
6.2.3	Sensae Console Backend API	128
6.2.4	Sensae Console Data Ingestion Endpoint	129
6.2.5	Sensae Console Rule Engine	129
6.2.6	Sensae Console Data Decoders	132
6.2.7	Sensae Console Database Configuration	134
6.2.8	Sensae Console Containerization	136
6.2.9	Sensae Console Orchestration	138
6.2.10	Sensae Console Reverse Proxy Configuration	139
6.2.11	Sensae Console Configuration Files	141
6.2.12	External Services	142
6.2.12.1	Smart Irrigation Service	142
6.2.12.2	Fleet Management Service	143
6.2.12.3	Notification Management Service	143
6.2.13	Sensae Console Device Integration	143
6.3	Testing	144
6.3.1	Unit Tests	145
6.3.2	Integration Tests	147
6.3.3	Functional Tests	148
6.3.4	End-to-End Tests	152
6.3.5	Architectural Tests	153
6.4	Synopsis	155
7	Evaluation	157
7.1	Objectives	157
7.2	Approach	158
7.3	Experiences	158
7.3.1	Fleet Management Experience Scenario	160
7.3.2	Smart Irrigation Experience Scenario	160
7.3.3	Notification Management Experience Scenario	161
7.4	Discussion of the overall results	162
7.4.1	Data Ingestion Endpoint Performance	162
7.4.2	Data Processor versus Data Decoder Performance	163
7.4.3	Data Flow Caching Process Performance	164
7.4.4	External Services Scope Database Performance	165
7.4.5	System Bottlenecks	166
7.5	Synopsis	167

8 Conclusion	169
8.1 Achievements	169
8.2 Unfulfilled Results	170
8.3 Future Work	170
8.4 Synopsis	171
Bibliography	173
A Data Unit - Shared Model Schema	179
B Complete Container Level - Logical View	181
C Components Level - Logical View - Details	183
D Additional Sensae Console UI Pages	187
E IoT Core Package POM	193
F Performance Tests Specification	197
G Performance Tests Analysis	201
H Contexts - TODO	203

List of Figures

4.1	Design Requirements Diagram	18
5.1	System Scopes	22
5.2	Context Level - Logical View Diagram	26
5.3	Context Level - Physical View Diagram	27
5.4	Sensae Console - Container Level - Logical View Diagram	29
5.5	Data Processor - Container Level - Logical View Diagram	30
5.6	Data Decoder - Container Level - Logical View Diagram	31
5.7	Device Management - Container Level - Logical View Diagram	32
5.8	Rule Management - Container Level - Logical View Diagram	33
5.9	Identity Management - Container Level - Logical View Diagram	33
5.10	Fleet Management - Container Level - Logical View Diagram	34
5.11	Notification Management - Container Level - Logical View Diagram	35
5.12	Smart Irrigation - Container Level - Logical View Diagram	35
5.13	Container Level - Configuration Scope - Logical View Diagram	38
5.14	Container Level - Data Flow Scope - Logical View Diagram	39
5.15	Container Level - External Services Scope - Logical View Diagram	40
5.16	Container Level - System/Container Initialization - Process View Diagram	42
5.17	Container Level - System/Container Initialization - Part 2 - Process View Diagram	43
5.18	Container Level - Data Flow - Diagram	44
5.19	Container Level - Data Decoder Operation part 1 - Process View Diagram	45
5.20	Container Level - Data Decoder Operation Part 2 - Process View Diagram	46
5.21	Container Level - Consult Data Processors - Process View Diagram	47
5.22	Container Level - Edit Device Information - Process View Diagram	48
5.23	Container Level - User Authentication - Process View Diagram	49
5.24	Container Level - User Authorization - Process View Diagram	50
5.25	Container Level - Consult Device Live Location via Fleet Management - Process View Diagram	51
5.26	Container Level - Receive notification via Notification Management - Process View Diagram	52
5.27	Container Level - Valve Activation Process via Smart Irrigation - Process View Diagram	53
5.28	Container Level - Frontend Services - Implementation View Diagram	54
5.29	Container Level - Backend Services - Implementation View Diagram	55
5.30	Container Level - Database Services - Implementation View Diagram	56
5.31	Container Level - Physical View Diagram	57
5.32	Component Level - Data Decoder Frontend - Logical View Diagram	59
5.33	Component Level - Device Management Backend - Logical View Diagram	60
5.34	Component Level - Device Ownership Backend - Logical View Diagram	62

5.35 Component Level - Process Data Unit in Device Management Flow Backend - Process View Diagram	64
5.36 Component Level - Deploy Draft Rule Scenarios in Rule Management Backend - Process View Diagram	65
5.37 Component Level - Data Decoder Frontend - Implementation View Diagram	66
5.38 Component Level - Device Management Backend - Implementation View Diagram	67
5.39 Component Level - Device Ownership Backend - Implementation View Diagram	68
5.40 Monoliths and Microservices	70
5.41 User Authentication/Authorization - Internal Authentication Server Alternative - Sequence Diagram	72
5.42 User Authentication/Authorization - External Authorization Server Alternative - Sequence Diagram	74
5.43 User Authentication/Authorization - External Authentication Server with Internal Authorization Server Alternative - Sequence Diagram	75
5.44 Internal Communication - First Option - Logical View Diagram	78
5.45 Internal Communication - Second Option - Logical View Diagram	79
5.46 Internal Communication - Third Option - Logical View Diagram	79
5.47 Internal Communication - Fourth Option - Logical View Diagram	80
5.48 Internal Communication - Fifth Option - Logical View Diagram	81
5.49 Shared Model	83
5.50 Message Envelop Model	88
5.51 Routing Model	89
5.52 Data Processor Context Model	95
5.53 Data Decoder Context Model	96
5.54 Device Management Context Model	97
5.55 Identity Management Context Model	98
5.56 Domain Structure	99
5.57 High-Level View of a Information Flow Processing (IFP) System	100
5.58 Rule Management Context Model	100
5.59 Notification Management Context Model	101
5.60 Smart Irrigation Context Model - Irrigation Zone	102
5.61 Smart Irrigation Context Model - Device	103
5.62 Smart Irrigation Context Model - Reading	104
5.63 Fleet Management Context Model	105
6.1 Advanced Messaging Queue Protocol (AMQP) 0.9.1 Protocol Concepts	112
6.2 Comparison of VM and Container-based deployments	118
6.3 Branching Model	120
6.4 Future Branching Model	120
6.5 Github Issues	121
6.6 Github Issues Project Board	122
6.7 Sensae Console Home Page	126
6.8 Sensae Console Smart Irrigation Page	127
6.9 Sensae Console Device Management Page	127
6.10 Custom Map - Smart Irrigation	128
6.11 Helium Custom Integration Page	144
7.1 Scatter Chart - Notification Management Scenario - Experience C	163

7.2	Scatter Chart - Smart Irrigation Scenario - Experience B	164
7.3	Scatter Chart - Fleet Management Scenario - Experience D	165
7.4	Line Chart - Time Taken to Ingest, Store and Supply Measures	166
B.1	Container Level - Logical View Diagram	182
C.1	Component Level - Smart Irrigation Backend - Logical View Diagram . . .	183
C.2	Component Level - Notification Dispatcher Backend - Logical View Diagram	184
C.3	Component Level - Data Gateway - Logical View Diagram	184
C.4	Component Level - Device Commander - Logical View Diagram	185
C.5	Component Level - Data Store - Logical View Diagram	185
D.1	Identity Management Page	187
D.2	Rules Management Page	187
D.3	Data Processor Page	188
D.4	Data Processor Page	188
D.5	Notification Management Page	189
D.6	Notification Management Page - Configuration	189
D.7	Fleet Management Page	190
D.8	Smart Irrigation Page - Map	190
D.9	Smart Irrigation Page - Device History	191
H.1	Sensae Console - Context Level - Logical View Diagram	204
H.2	Sensae Console - Context Level V2 - Logical View Diagram	206
H.3	External Services - Context Level - Logical View Diagram	207
H.4	Data Decoder - Container Level - Logical View Diagram	207
H.5	Data Processor - Container Level - Logical View Diagram	208
H.6	Device Management - Container Level - Logical View Diagram	208
H.7	Identity Management - Container Level - Logical View Diagram	209
H.8	Rule Management - Container Level - Logical View Diagram	209
H.9	Fleet Management - Container Level - Logical View Diagram	210
H.10	Notification Management - Container Level - Logical View Diagram	210
H.11	Smart Irrigation Management - Container Level - Logical View Diagram . .	211

List of Tables

4.1	Summary of the main requirements of the requested business cases	12
5.1	Comparison of Operations in Data Flow and Configuration Scopes	24
5.2	Backend components responsibilities	61
5.3	Measure Data Types	85
5.4	Routing Options	91
6.1	Technologies Comparison - Angular vs React	109
6.2	Technologies Comparison - Reverse Proxy Web Server	119
7.1	Details about the experiences performed	159
7.2	Results for the Fleet Management Scenario (in seconds)	160
7.3	Results for the Smart Irrigation Scenario (in seconds)	161
7.4	Results for the Notification Management Scenario (in seconds)	161
7.5	Data Ingestion Endpoint response time results (in milliseconds)	163
7.6	Metrics collected (in seconds) - Notification Management Scenario - Experience C	164

List of Source Code

5.1	Inbound Information Example	95
6.1	Configuration File for <i>iot-core</i> Continuous Delivery	122
6.2	Configuration File for Sensae Console Continuous Integration	123
6.3	Sensae Console Test Suite Script	124
6.4	Smart Irrigation API Schema	128
6.5	Rule Scenario Example - Part 1	130
6.6	Rule Scenario Example - Part 2	131
6.7	Rule Scenario Example - Part 3	131
6.8	EM300-TH Data Decoder Example	132
6.9	Initialization Script Segment for Data Processor Database	134
6.10	Bootstrap function for Identity Management Database	135
6.11	Dockerfile for UI Aggregator Frontend	136
6.12	Dockerfile for Fleet Management Backend	136
6.13	Dockerfile for Device Commander	137
6.14	Docker Compose Configuration File for Production	138
6.15	Configuration File for Production Environment	139
6.16	Configuration File for Production Environment	141
6.17	Configuration Propagation Script	141
6.18	Unit Test Example in <i>iot-core</i> package	145
6.19	Unit Test - Data Decoder Backend Container	146
6.20	Unit Test - Device Management Frontend Model Library	147
6.21	Integration Test - Message Broker - Device Ownership Flow	147
6.22	Integration Test - Database - Notification Management Backend	148
6.23	Functional Test - Message Broker - Data Decoder Master Backend Setup	148
6.24	Functional Test - Foundation - Data Decoder Master Backend Setup	149
6.25	Functional Test - Database Interaction - Data Decoder Master Backend	150
6.26	Functional Test - Message Broker Interaction - Data Decoder Master Backend	151
6.27	Functional Test - Rest Client Interaction - Data Gateway	152
6.28	End-to-End Test - Custom Commands - UI Aggregator	152
6.29	End-to-End Test - Anonymous Authentication - UI Aggregator	153
6.30	End-to-End Test - Discover Available Domains - Identity Management	153
6.31	Architectural Test - Onion Architecture - Device Management Master Backend	154
6.32	Architectural Test - Simplified Onion Architecture - Data Processor Flow	154
A.1	Data Unit - Shared Model Schema	179
E.1	IoT Core Package POM	193
F.1	Smart Irrigation Performance Test Scenario Description	197
G.1	Analysis Script	201

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 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 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 customer-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 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:

- Agnoscitically 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 with a strong semantic meaning;
- 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' components and data depending on the user authentication and authorization level.

Finally, this project requires the creation of PoCs that answer specific business cases and utilize the platform developed. 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 IoT services. Some PoCs that answer the needs of various

business cases must be developed. These business cases are related to the capture and analysis of IoT data. Each business case is considered a concern and should be addressed in an independent PoC. The PoCs implemented must, in the end, rely on the platform to develop.

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 these 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 phase II. Even though these services are independent they all share 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 PoC's 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 must follow 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 it self 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, (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;
- Analysis: where concepts related to this project are referred;
- 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

2.1 Internet of Things

2.1.1 Brief Description

2.1.2 Practical Applications

2.1.3 Enterprise Challenges

2.1.4 Renowned Solutions

2.2 Big Data

2.2.1 Brief Description

2.2.2 Challenges

2.3 Synopsis

Chapter 3

Analysis

3.1 Business Analysis

- 3.1.1 Fleet Management**
- 3.1.2 Smart Irrigation**
- 3.1.3 Fire Outbreak Surveillance**

3.2 Technical Analysis

- 3.2.1 Data Aggregation**
- 3.2.2 Data Filtering**
- 3.2.3 Data Storage**
- 3.2.4 Data Transformation**
- 3.2.5 Data Analysis**
- 3.2.6 Data Presentation**
- 3.2.7 Trigger Warning System**
- 3.2.8 User Authentication/Authorization**

3.3 Synopsis

Chapter 4

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 4.1.

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

- **Hosting Options:** Should the solution be deployed in the organization's server farm or in the cloud;
- **Physical Resources:** Should the organization share an instance of the solution with others or not;
- **Data Access:** Does the organization wants to provide their data to the public;
- **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 4.1: Summary of the main requirements of the requested business cases

Org	Business Case	Hosting Option	Physical Resources	Data Access	Information Access and Visualization
A	Fleet Management	On-Site	Dedicated	Private	Sensae Console
	Smart Irrigation	On-Site	Dedicated	Private	Sensae Console
	Smart Parking	On-Site	Dedicated	Private	Sensae Console
	Indoor Fire Detention	On-Site	Dedicated	Private	SMS and Email
	Public Health Surveillance	On-Site	Dedicated	Public	Sensae Console
B	Fleet Management	Cloud	Dedicated	Private	Org B Platform
C	Smart Irrigation	Cloud	Shared	Private	Sensae Console
	Indoor Fire Detention	Cloud	Shared	Private	SMS and Email
D	Chicken Farm Monitoring	Cloud	Shared	Private	Sensae Console
	Smart Irrigation	Cloud	Shared	Private	Sensae Console

The requirements detailed in the following sections were founded on top of the requested business cases mentioned above. This 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.

4.1 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**.

4.1.1 Roles

The meetings that took place during this project lifespan lead to the definition of three main roles:

- **Manager**: a tenant with full control over the **Sensae Console** and all its data. He/She has also full control of all **External Services**;
- **Costumer**: a tenant 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 tenant 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 4.1.2.2.

Essentially, the difference between this roles boils down to what permissions each tenant has been assigned and the extent of data he can visualize. The Section 5.4.3.4 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.

4.1.2 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.

4.1.2.1 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 5.4.3.4.

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

- The Manager must be able to create, view, update and delete device payload decoders;
- The Manager must be able to create, view, update and delete device payload processors (or mappers);

- The Manager must be able to create, view, update and delete rules that trigger alerts;
- The Manager must be able to define, view, update and remove device specific information;
- The Manager must be able to define the permissions of any organization;
- The Manager must be able to assign new devices to a specific organization;
- The Manager must be able to assign new authenticated users to a specific organization;

As described in Sections 5.4.3.1 and 5.4.3.2, 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 5.4.3.5.

The device information mentioned in item four is detailed in Section 5.4.3.3.

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 4.1, 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 this 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.

4.1.2.2 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:

- A Costumer must be able to create and remove a department under his/her organization;
- A Costumer must be able to define the permissions for all tenants in a department under his/hers organization;
- A Costumer must be able to assign and move a tenant from/to a department under his/her organization;
- A Costumer must be able to move a sensor from one department to another department under his/hers organization;

4.1.3 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. Event 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.

4.1.3.1 Fleet Management

Within a simple Fleet Management business case the major utilities a Costumer can benefit from are: real-time tracking of his vehicles and visualizing past data regarding the whereabouts of his fleet. A more advanced Fleet Management would for example provide Key Performance Indicators (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:

- A Costumer must be able to track in real-time a vehicle location and motion status;
- A Costumer must be able to see the itineraries of a vehicle in defined time span;
- A Costumer must be able to see where, when and for how long a vehicle was parked;
- 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 5.4.4.3.

4.1.3.2 Smart Irrigation

Within a Smart Irrigation business case the major utilities a Costumer can benefit from are: real-time tracking of a garden/greenhouse conditions, archiving conditions for later use/consulting and 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:

- A Costumer must be able to manage his/her garden's information;
- A Costumer must be able to track a gardens' conditions in real-time;
- A Costumer must be able to see past conditions of a garden;
- 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 5.4.4.2.

4.1.3.3 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 but never implemented. As such, the only requirement related to this business case is:

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

4.2 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.

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

4.2.1 Functionality Requirements

Regarding the Functionality category, the following requirements were identified:

- User Authentication: Apart from the Anonymous Users, everyone else must be authenticated to use the system;
- User Authorization: Everyone only has access to what his/her permissions cover;
- Auditing and Traceability: All actions performed against the system must be recorded;
- Security in Communication: The use of secure protocols between clients and the system is mandatory, e.g.: https instead of http;
- Security in User-provided code: All user-provided code must run in sandbox's to prevent permission escalation, data theft and other related concerns;
- Data Filtering: The system shall be able to filter erroneous sensor data such as GPS coordinated of a vehicle appearing in the middle of the ocean;
- Real-time Alerts: The system must notify the interested clients in real time of any alarm triggered by custom rules;
- 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.

4.2.2 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 purposed.

4.2.3 Reliability Requirements

The Reliability category has the following requirements:

- The system must validate all user inputs, denying code injection according to OWASP 2021;
- The system must be able to recover from a failure state such as a crash in the system or any system component;
- 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.

4.2.4 Performance Requirements

Even though this work is in its early stages of development the Performance of the system is a priority. The requirements specified for this category are:

- 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);
- When an alarm is triggered, the system should dispatch the alarm within 10 seconds in 90% of the cases;
- Concurrent Utilization: The system must be able to be used by various users at the same time;
- 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.

4.2.5 Supportability Requirements

In the Supportability category the following requirements were identified:

- 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;
- 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;
- The system must provide simple methods to integrate external services that answer new business cases without the need to rebuilding it;
- The system must 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.

4.2.6 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 Customers, Organizations and **External Services**.

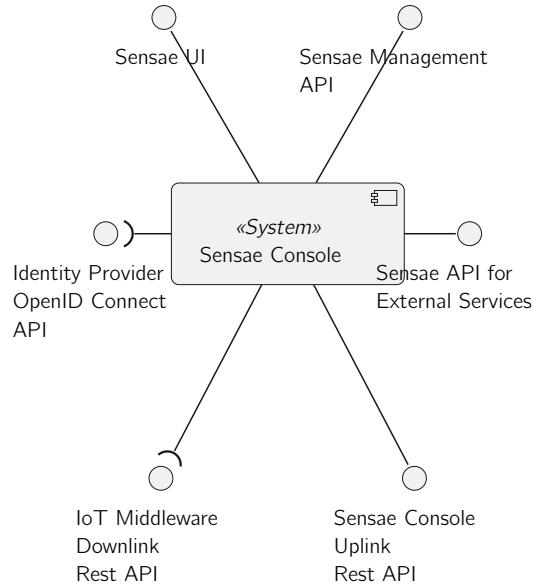


Figure 4.1: Design Requirements Diagram

4.2.7 Implementation Requirements

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

4.2.8 Interface Requirements

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

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

As for the **External Services**:

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

4.2.9 Physical Requirements

In the Physical category, the following requirements were identified:

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

4.3 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.

Altho 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 5

Design

This chapter's goal is to describe the overall system design to the reader. First, the system scopes will be introduced to guide the reader through the system design. After this, the system's architectural design will be presented and major decisions/alternatives are discussed. Then a more detailed vision of the system is given by presenting the domain models of each scope's contexts.

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 concept/concern/capability. Scopes are derived from the major system responsibilities of the solution as a whole, concept/concern/capability are derived from the major functionalities or business cases that the project has to answer.

5.1 System Scopes

The solution designed can be divided in three main scopes as disclosed in Figure 5.1.

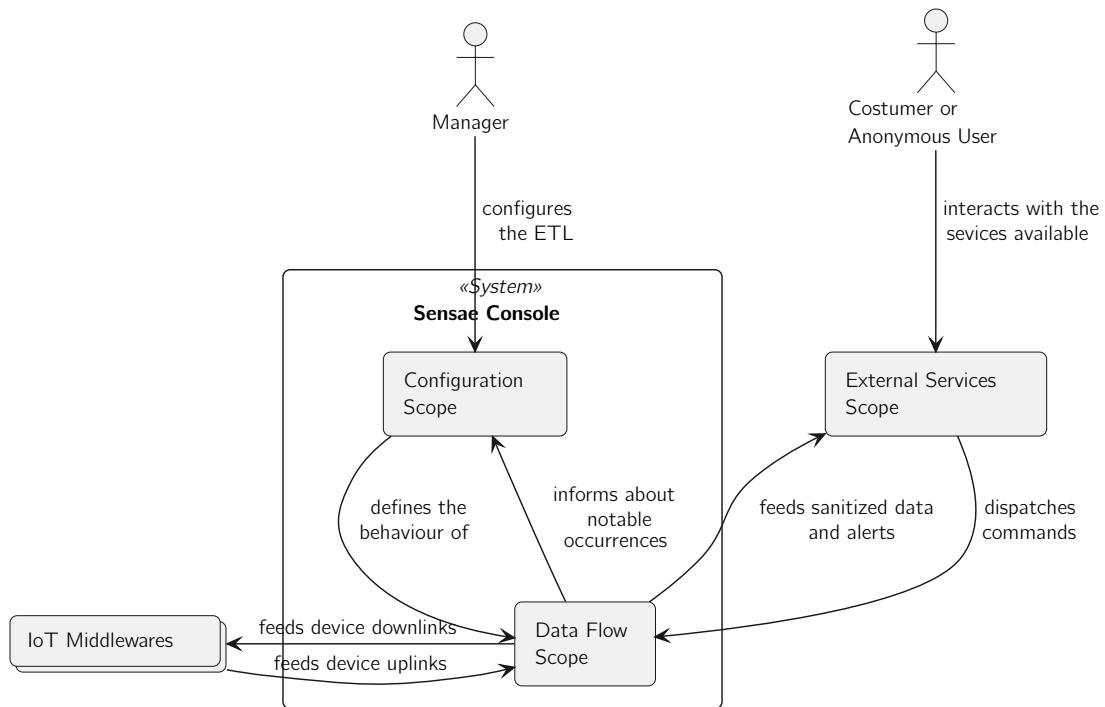


Figure 5.1: System Scopes

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 service.

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.

The **Configuration Scope** refers to the configuration and visualization of internal processes/contexts. Processes 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 this scope.

The **Data Flow Scope** acts as a pipeline where raw data, device uplinks, 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) alerts 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.

5.1.1 Configuration Scope

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

- **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.

This contexts can be directly linked to the functional requirements described in the Section related to the 4.1.2.1 role.

Each context allows an authorized user to manage its resources, e.g. the data processor context manages the creation, deletion and renovation of data mappers.

This operations require various verification's, alter the system internal state and are therefore prolonged operations.

5.1.2 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 contexts.

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

- Data Gateway: responsible for providing a bridge between the IoT middlewares and the **Sensae Console**;
- 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 defined state.

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

This scope was decoupled from the **Configuration Scope** even though they both work with the same contexts. The decision was taken based on the pretext that despite the similarities in context the operation/business responsibilities of this 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 5.1.

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.

Table 5.1: Comparison of Operations in Data Flow and Configuration Scopes

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

5.1.3 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 External Services section. The type of data each service handles is enforced by the concepts discussed in Sections 5.4.1 and 5.4.2.

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

5.2 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 2018b) and 4+1 will be followed.

The 4+1 View Model (By and Jiang 1995), 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 (Brown 2018b, Brown 2018a) advocates describing software through four levels of abstraction: (i) system, (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 likened to maps, e.g. the system view corresponds to the globe, the container corresponds to the map of each continent, the component view corresponds to the map of each of each country, and the code view to the map of roads and neighborhoods in each city.

Different levels allow you to 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 level deemed relevant for the design of the solution are presented.

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

5.2.1 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**, are demonstrated. Throughout this section the relevant C4 views of level 1 (context level) are presented.

5.2.1.1 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 4. This takes into account the interactions of **Sensae Console** and **External Services** with foreign systems and its interaction with the various actors of the system (Figure 5.2) as required by Section 4.2.6.

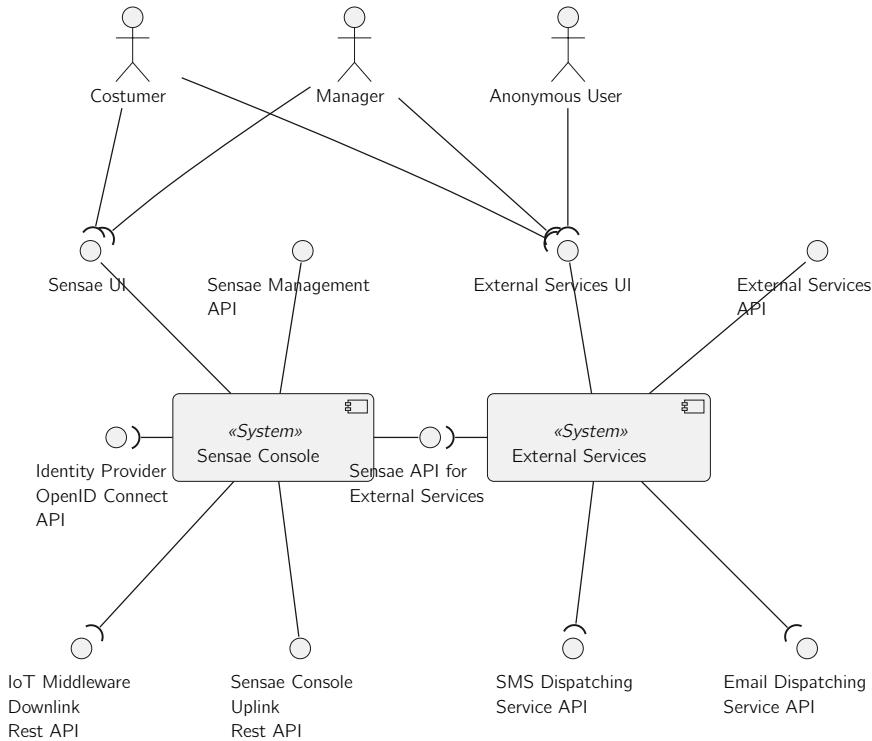


Figure 5.2: Context Level - Logical View Diagram

The **External Services** are represented as an independent system that consumes 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 5.4.2.

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

As mentioned before in Section 4.1.3.3 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 4.2.8, is further discussed in the Section 5.3.3.

5.2.1.2 Context Level - Physical View

Next is the physical view (Figure 5.3), intended to familiarize the reader with the environment where the solution runs.

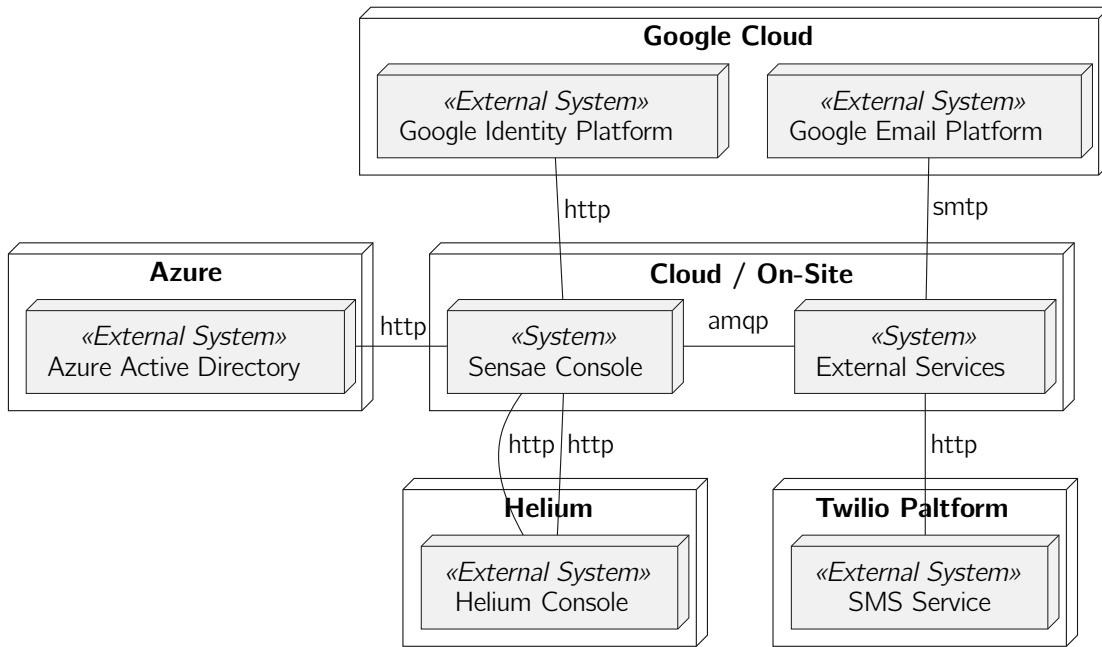


Figure 5.3: Context Level - Physical View Diagram

The Identity Providers currently in use by the solution are Google n.d. and Azure 2022 both of this platforms provide an OpenID Connect API.

The SMS dispatching service is the *Twilio Platform*. The Email dispatching service is the *Google Email Platform*. And finally, the only IoT Middleware in use currently is Helium 2022.

5.2.1.3 Context Level - Synopsis

The context level introduces the reader to the bigger picture of **Sensae Console**, but it contains little to no information about how the system functions internally, the Section 5.2.2 will dive into this subject.

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.

5.2.2 C4 Level 2 - Containers

The C4 level 2 introduces the reader to the various containers that compose the system. In this section all relevant views will be presented according to the alternative in use or idealized for the system. In the Section 5.3 other alternatives are discussed.

The description of this level of abstraction begins with a logical view of the containers that compose the system.

5.2.2.1 Container Level - Logical View NEW

In order to support the functional requirements identified (Section 4.1), 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 business 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 Database Management System (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 system the following diagram, Figure 5.4, presents a complete view of **Sensae Console** where each concept/concern/capability represents a group of containers that focus on the same concern. These groups are them explored in detail.

As seen in the diagram:

- Each concept/concern/capability 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 concept/concern/capability 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 concept/concern/capability 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 store data in a specific format, defined at startup;
- 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% as marked as incorrect.

Each concept/concern/capability 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 concept/concern/capability 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;

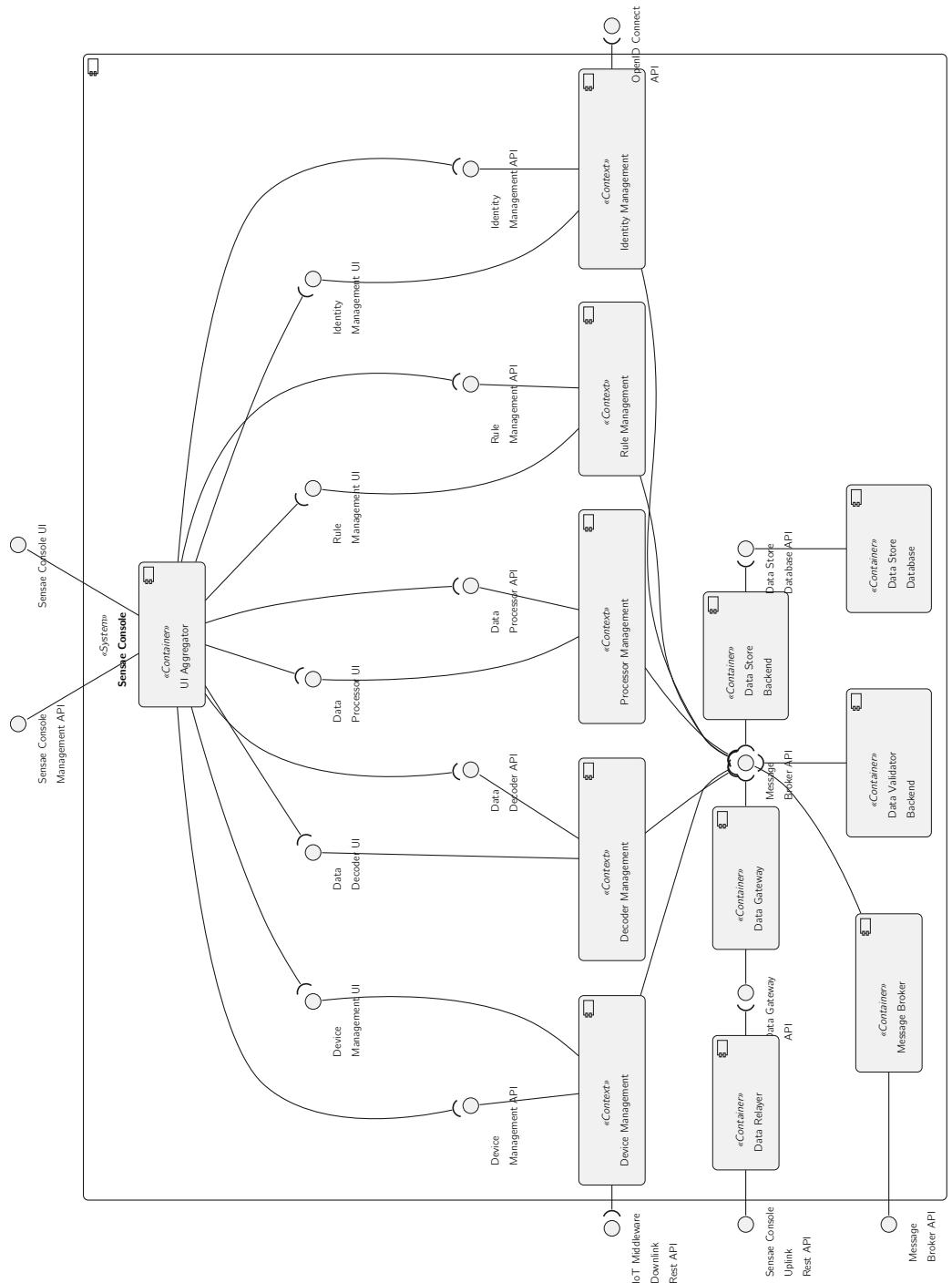


Figure 5.4: Sensae Console - Container Level - Logical View Diagram

- **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 concept/concern/capability:

- The frontend container corresponds to the **Presentation Layer** and exposes an UI;
- The backend container corresponds to the **Application Layer** and communication with the Data Flow container(s) exclusively through the **Message Broker**. The Backend publishes issues related to the concept/concern/capability 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 5.4.2.3 and is preformed in the Internal Topic;
- The backend container also exposes an API that is consumed by the frontend and optionally by properly authenticated external systems;
- The database container correspond to the **Data Layer**.

The Data Processor concept/concern/capability group is presented in Figure 5.5.

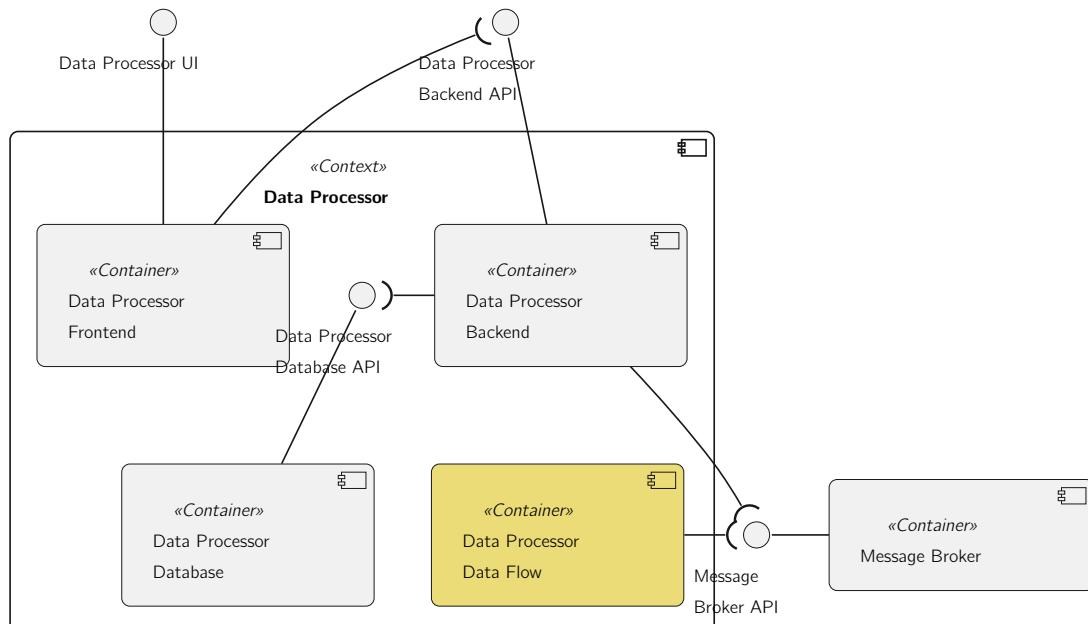


Figure 5.5: Data Processor - Container Level - Logical View Diagram

This concept/concern/capability 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 5.4.3.1. 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 concept/concern/capability group is presented in Figure 5.6.

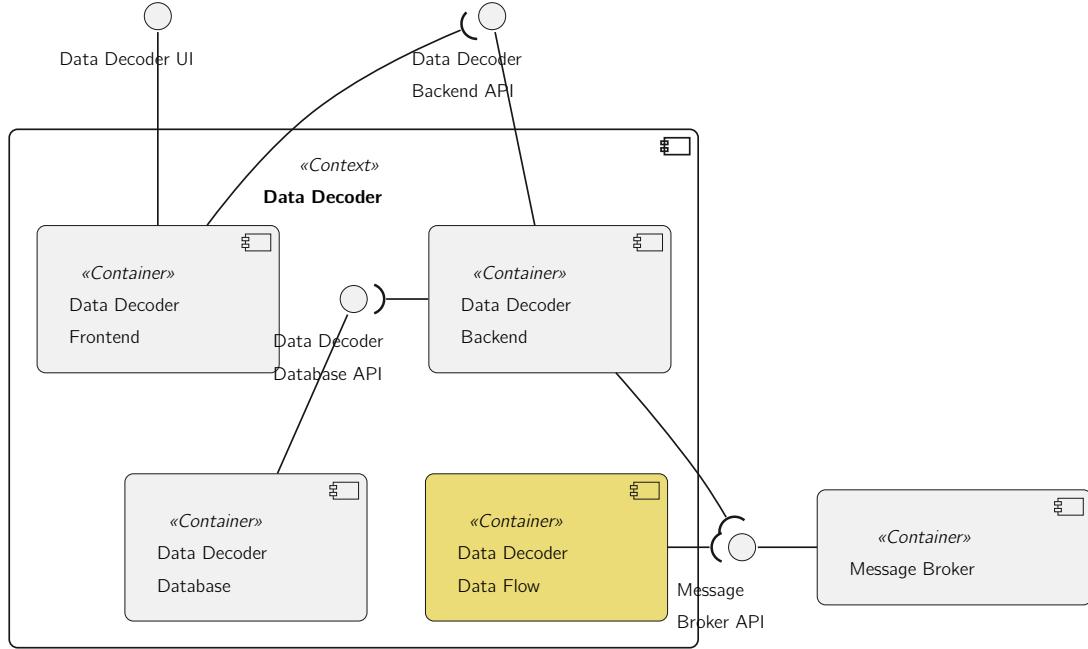


Figure 5.6: Data Decoder - Container Level - Logical View Diagram

This concept/concern/capability 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 5.4.3.2. 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 concept/concern/capability group is presented in Figure 5.7.

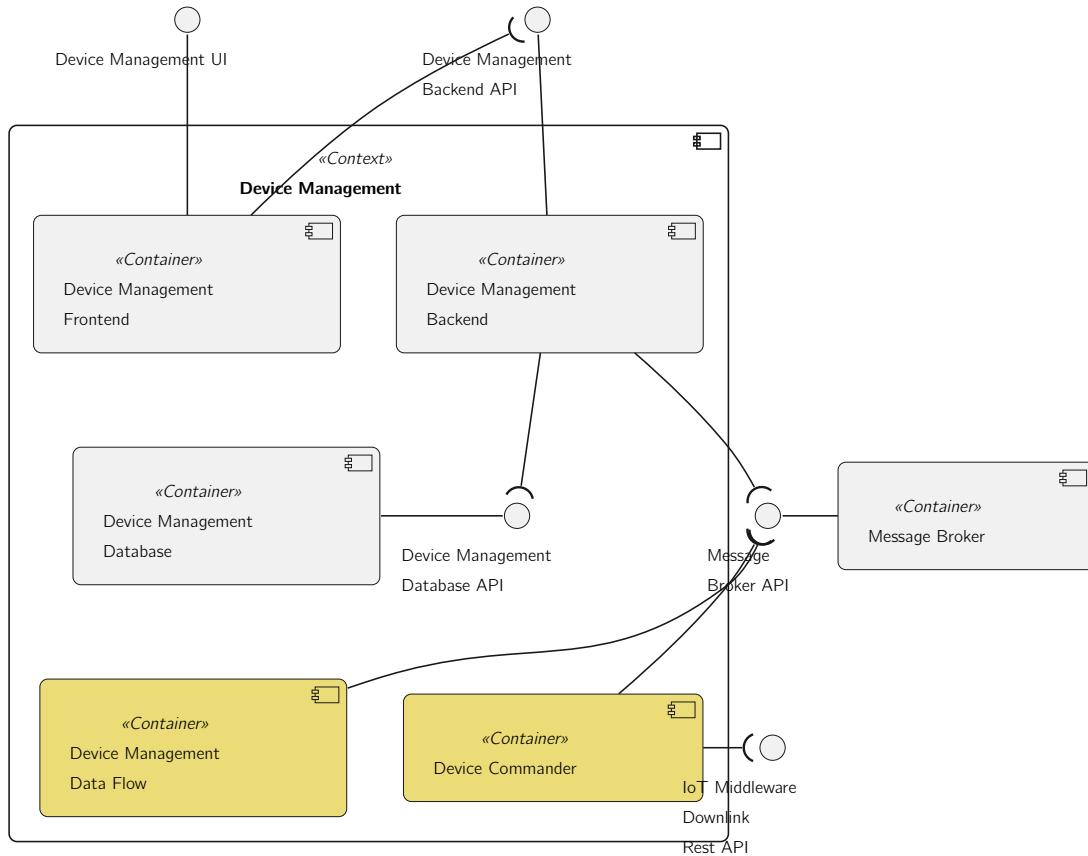


Figure 5.7: Device Management - Container Level - Logical View Diagram

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

The Device Management Data Flow enriching 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 concept/concern/capability is explored in Section 5.4.3.3. 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 Rule Management concept/concern/capability group is presented in Figure 5.8.

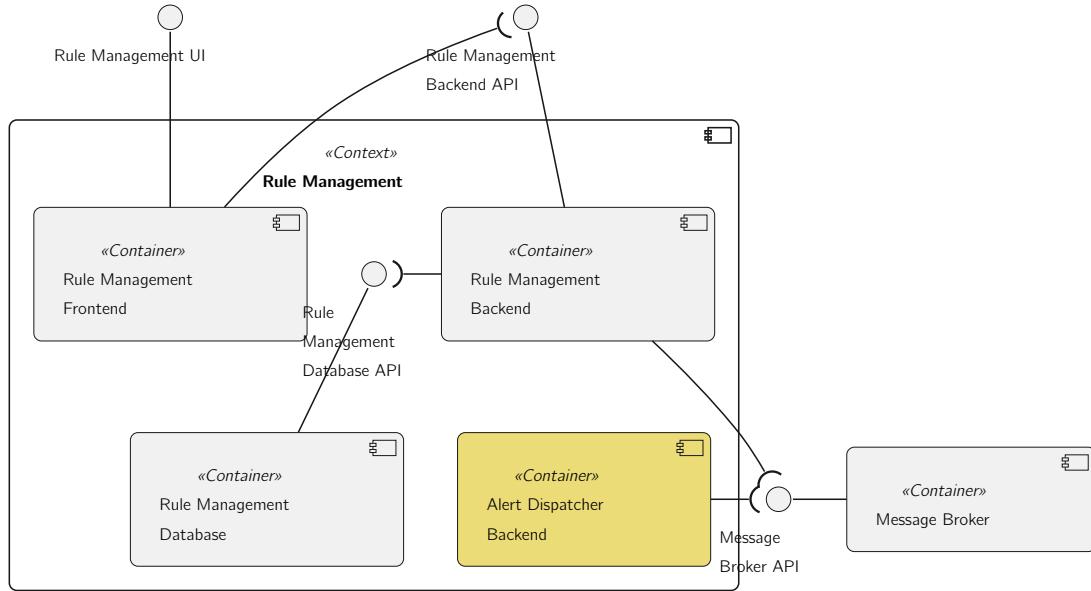


Figure 5.8: Rule Management - Container Level - Logical View Diagram

This concept/concern/capability 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 concept/concern/capability is explored in Section 5.4.3.5. This data flow container does not publish any metrics, its interactions are better described with the help of sequence diagrams available in Figures 5.16 and 5.36.

The Identity Management concept/concern/capability group is presented in Figure 5.9.

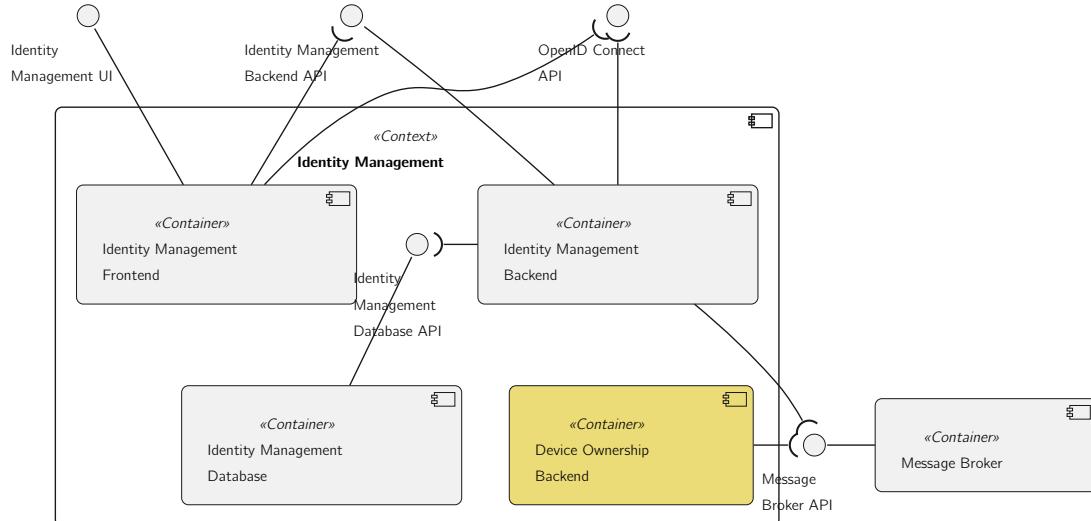


Figure 5.9: Identity Management - Container Level - Logical View Diagram

This concept/concern/capability 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 concept/concern/capability is explored in Section 5.4.3.4. This data flow container publishes metrics to the Message Broker regarding the time each organization information was used.

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 next diagrams present the three External Services that were developed.

Some of the similarities shared between the architecture of all services are:

- All contain a backend that exposes an API;
- All contain a frontend that exposes a UI;
- All contain at least a database that exposes an API consumed solely by the backend;
- Any communication with **Sensae Console** is performed by consuming the Message Broker's API;
- All follow the idea behind the separation of responsibilities seen in a three layer architecture;

The Fleet Management service is presented in Figure 5.10.

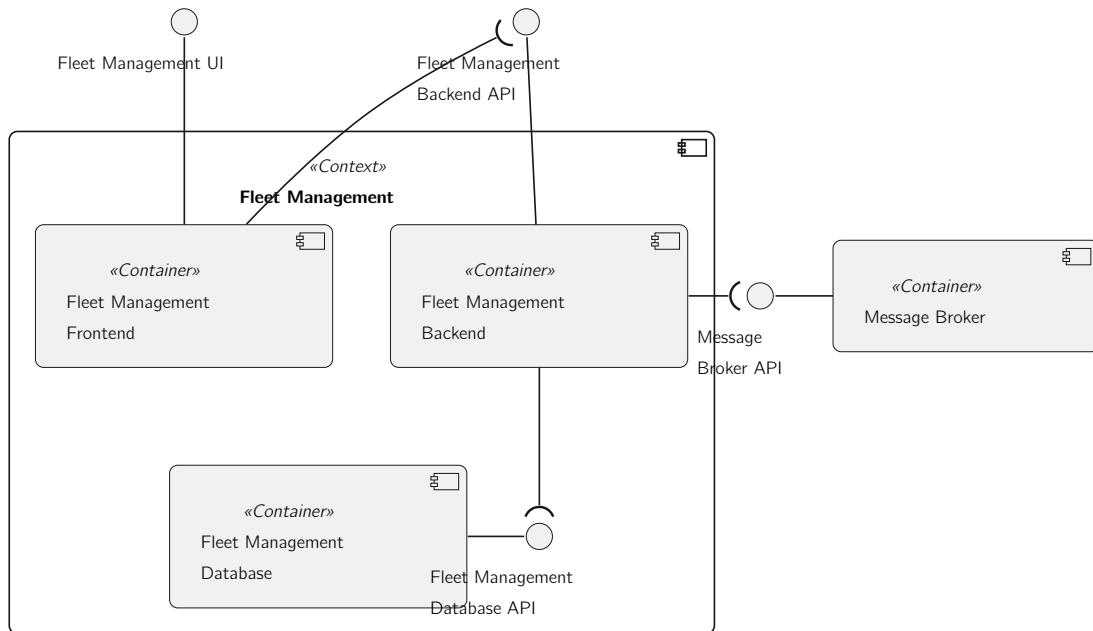


Figure 5.10: Fleet Management - Container Level - Logical View Diagram

This service is composed by a simple three layers architecture. The details related to this service are discussed in Section 5.4.4.3.

The Notification Management service is presented in Figure 5.11.

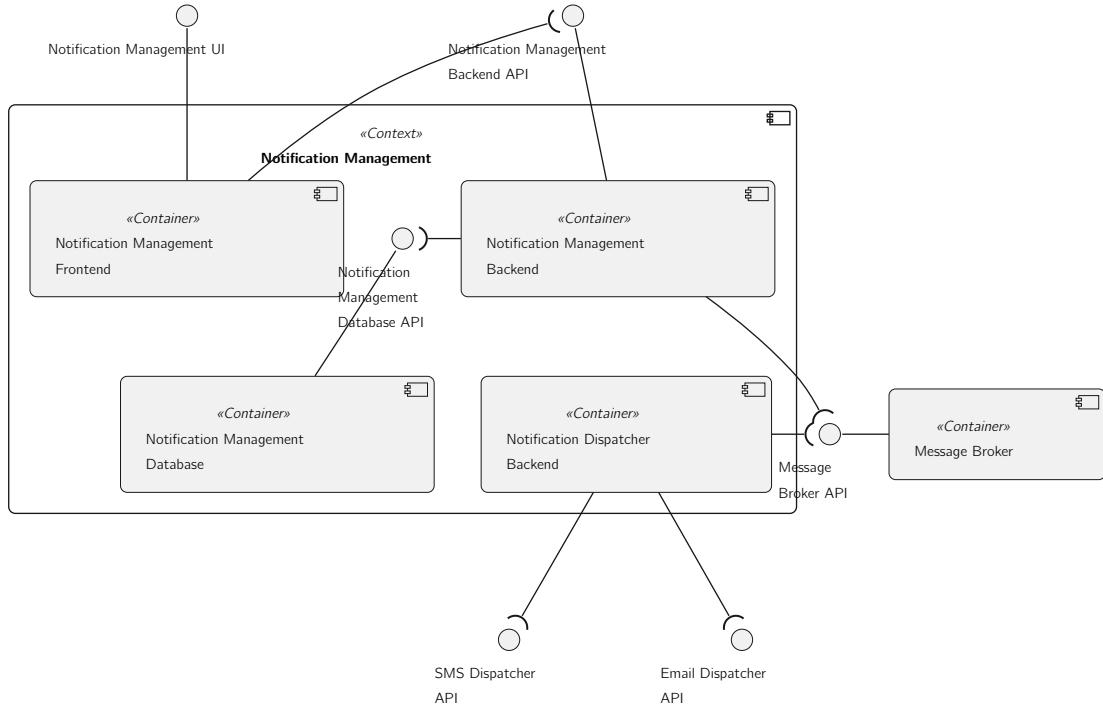


Figure 5.11: Notification Management - Container Level - Logical View Diagram

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 5.4.4.3.

Finally, the Smart Irrigation Management service is presented in Figure 5.12.

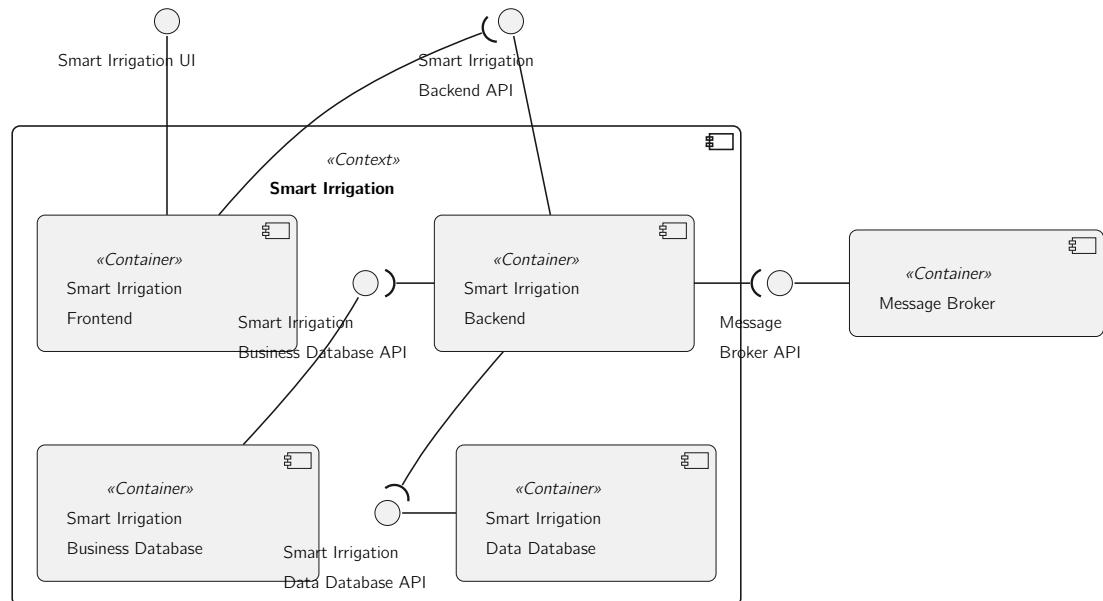


Figure 5.12: Smart Irrigation - Container Level - Logical View Diagram

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 5.4.4.2.

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

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

5.2.2.2 Container Level - Logical View - OLD

In order to support the functional requirements identified (Section 4.1), 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 business 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 system the following diagrams will be divided by scopes, mentioned in 5.1. In the Appendix B a complete logical view is provided.

The logical view of the **Configuration Scope** is represented in Figure 5.13. This scope is composed by the processes discussed in 5.1.1. Each process is composed by a three layer 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.

This scope was also divided into micro services - Newman 2021 - '*small, autonomous services that work together*'. Each bounded context/business process - (i) Data Processor, (ii) Data Decoder, (iii) Device Management, (iv) Identity Management, (v) Rule Management - is mapped to the three layer architecture mention before.

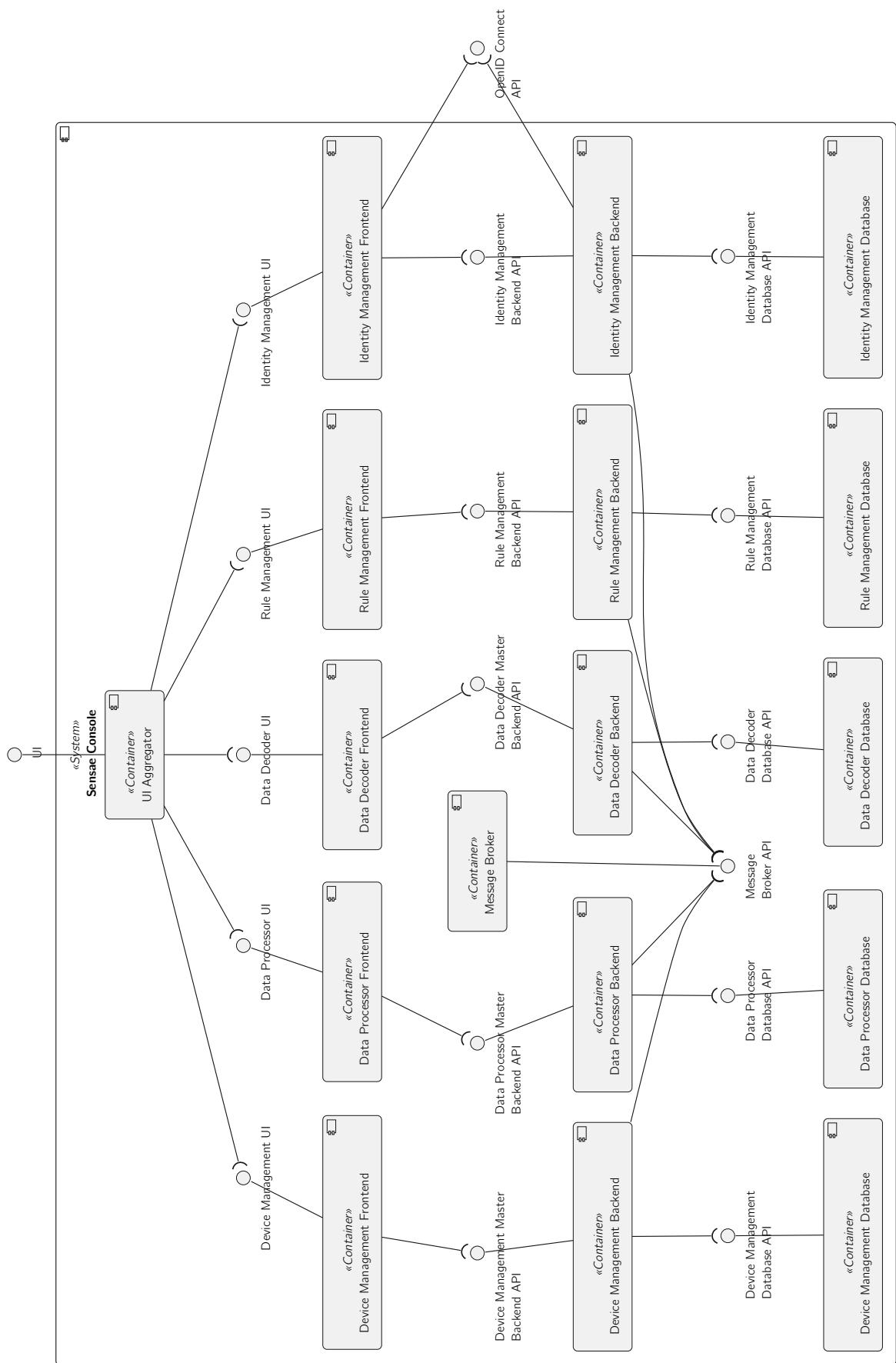


Figure 5.13: Container Level - Configuration Scope - Logical View Diagram

As a brief description:

- Frontend containers correspond to the **Presentation Layer** and are provided to the user through **UI Aggregator**;
- Backend containers correspond to the **Application Layer** and communication with each other through **Message Broker**;
- Database containers correspond to the **Data Layer**.

Next, the logical view of the **Data Flow** is represented in Figure 5.14. This scope is composed by the processes discussed in 5.1.2. In parallel with the **Configuration Scope** this scope is also divided into multiple micro services in order for them to better scale once needed. This scope is also built based on a *Reactive architecture* as described in Jonas Bonér and Thompson 2014 and Jansen 2020.

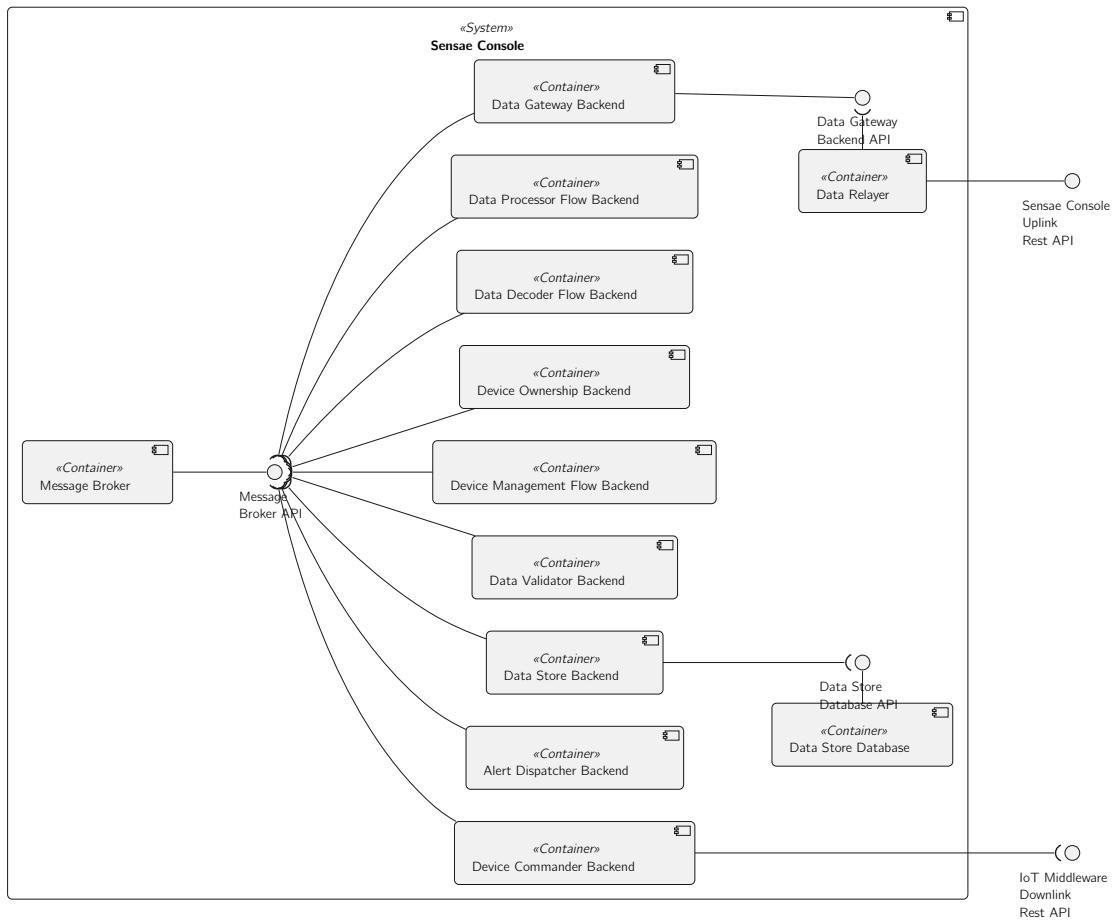


Figure 5.14: Container Level - Data Flow Scope - Logical View Diagram

Most containers presented here collect specific information from a single backend in the **Configuration Scope** through the *Internal Topic*:

- **Data Processor Flow Backend**: Collects information related to the **Data Processor** context - published by **Data Processor Backend** - and nothing else;
- **Data Decoder Flow Backend**: Collects information related to the **Data Decoder** context - published by **Data Decoder Backend** - and nothing else;

- **Device Ownership Backend:** Collects information related to the **Identity Management** context (more specifically device ownership) - published by **Identity Management Backend** - and nothing else;
- **Device Management Flow Backend:** Collects information related to the **Device Management** context - published by **Device Management Backend** - and nothing else;
- **Alert Dispatcher Backend:** Collects information related to the **Rule Management** context - published by **Rule Management Backend** - and nothing else;
- **Device Command Backend:** Collects information related to the **Device Management** context - published by **Device Management Backend** - and nothing else;
- The remaining containers don't subscribe to any type of information from the **Configuration Scope**.

Finally the **External Services Scope** is represented in Figure 5.15. This scope is composed by the processes discussed in 5.1.3.

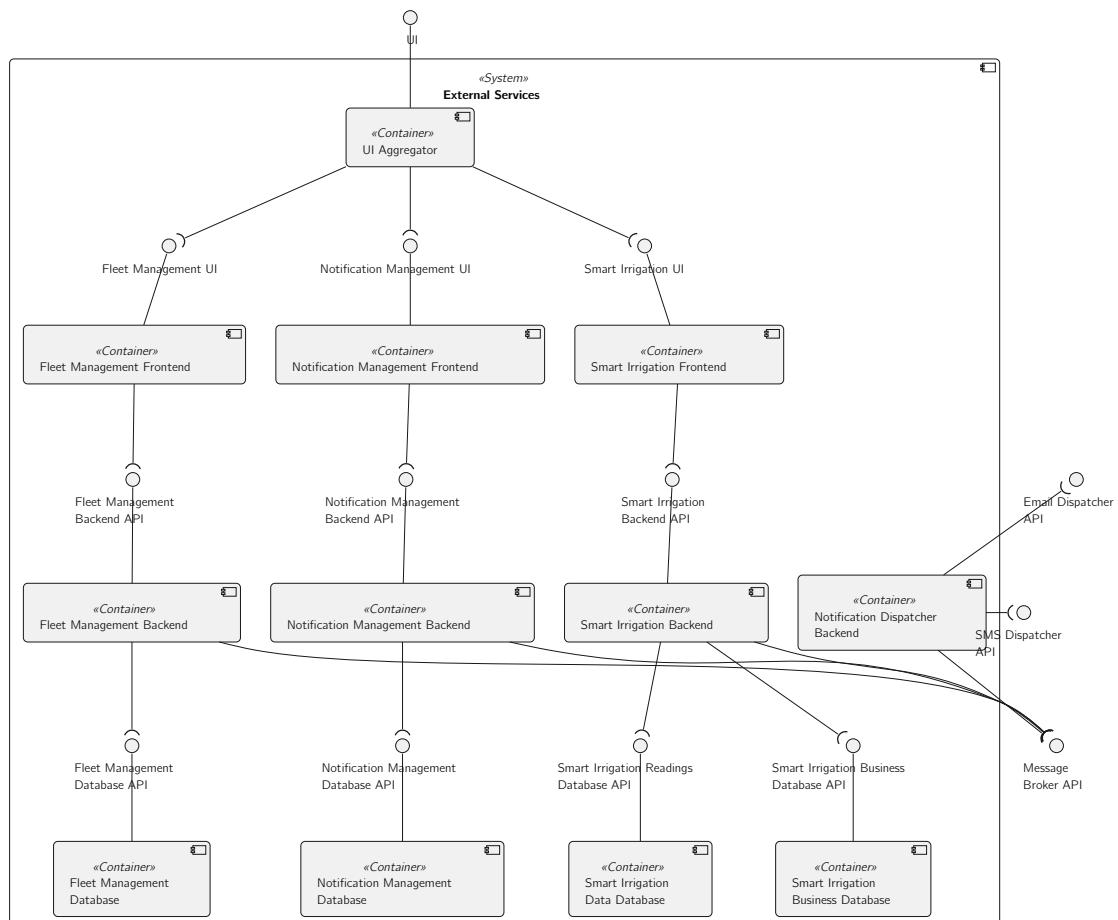


Figure 5.15: Container Level - External Services Scope - Logical View Diagram

Once again the ideas behind this scope architecture are the same discussed in the **Configuration Scope** apart from two particular points:

- **Smart Irrigation Data Database/Business Database:** As explained in the domain presented in 5.4.4.2, since there are two distinct types of information to store and manage it was decided to use different technologies for each type;
- **Notification Management/Dispatcher Backend:** It was also decided to split the delivery of notifications (by email and SMS) from their management.

Lastly, as we can see some containers are present in more than one scope. These containers, and their responsibilities are:

- **Message Broker:** Container responsible for routing messages/events sent by backend containers. This communication is explored in the section, Container Level - Process View;
- **UI Aggregator:** Container responsible for aggregating all frontends in a single UI.

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

5.2.2.3 Container Level - Process View

In this section several use cases (according to some functional requirements identified in Section 4.1) 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 5.4.2.3.

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 5.16, refers to the interval of time since a container is launched till it is ready to process requests or events.

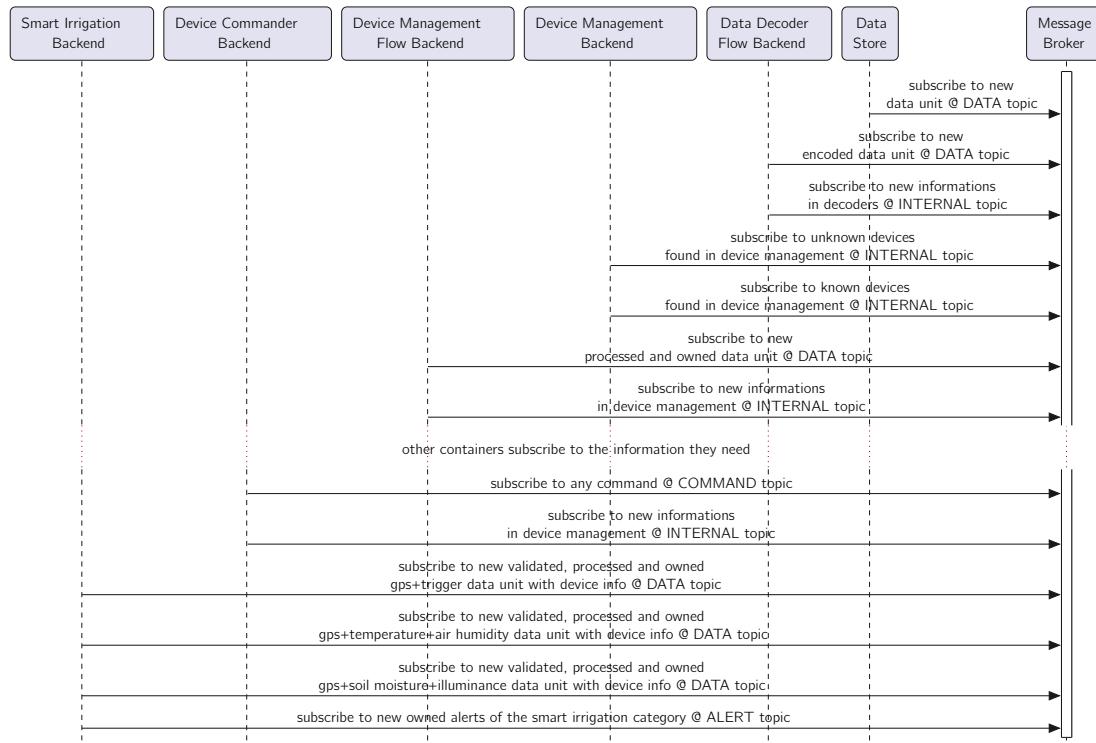


Figure 5.16: Container Level - System/Container Initialization - Process View Diagram

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 *ContextType* to function. So, after subscribing to the needed information, they notify the system that they have entered an *init state* for a specific context. 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 5.17, this only occurs in the Internal Topic.

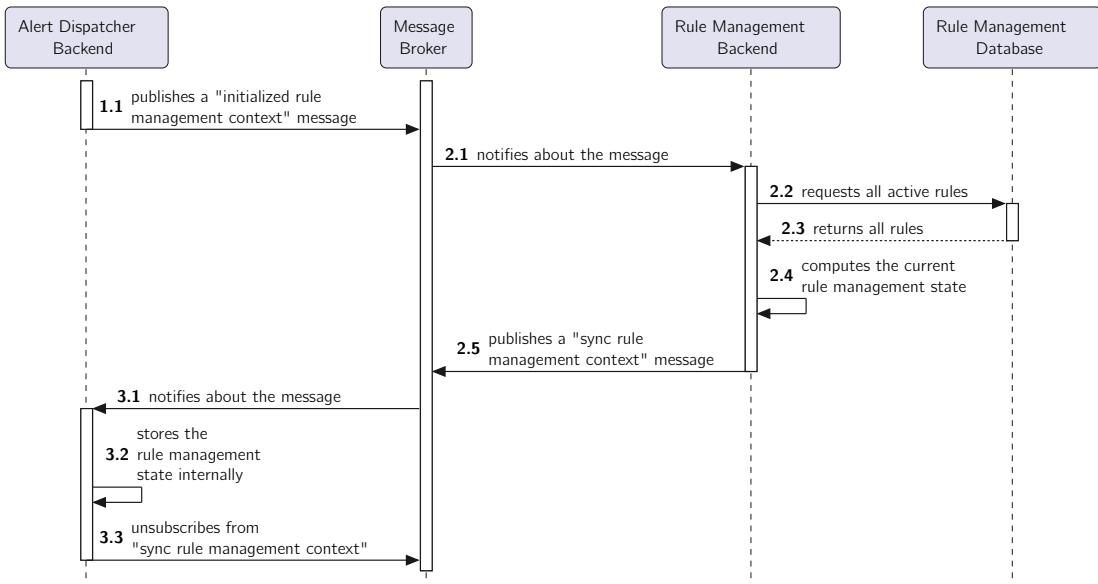


Figure 5.17: Container Level - System/Container Initialization - Part 2 - Process View Diagram

Apart from the Alert Dispatcher Backend, all containers in the **Data Flow Scope** benefit from a stateless process and can function with just a portion of a single *ContextType* state or no state at all.

To dive into this, some common data pipeline operations, related to the Data Flow Scope, are presented next. These 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 on Data Units, transforming, filtering and enriching this data.

The following diagram in Figure 5.18 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 messages.

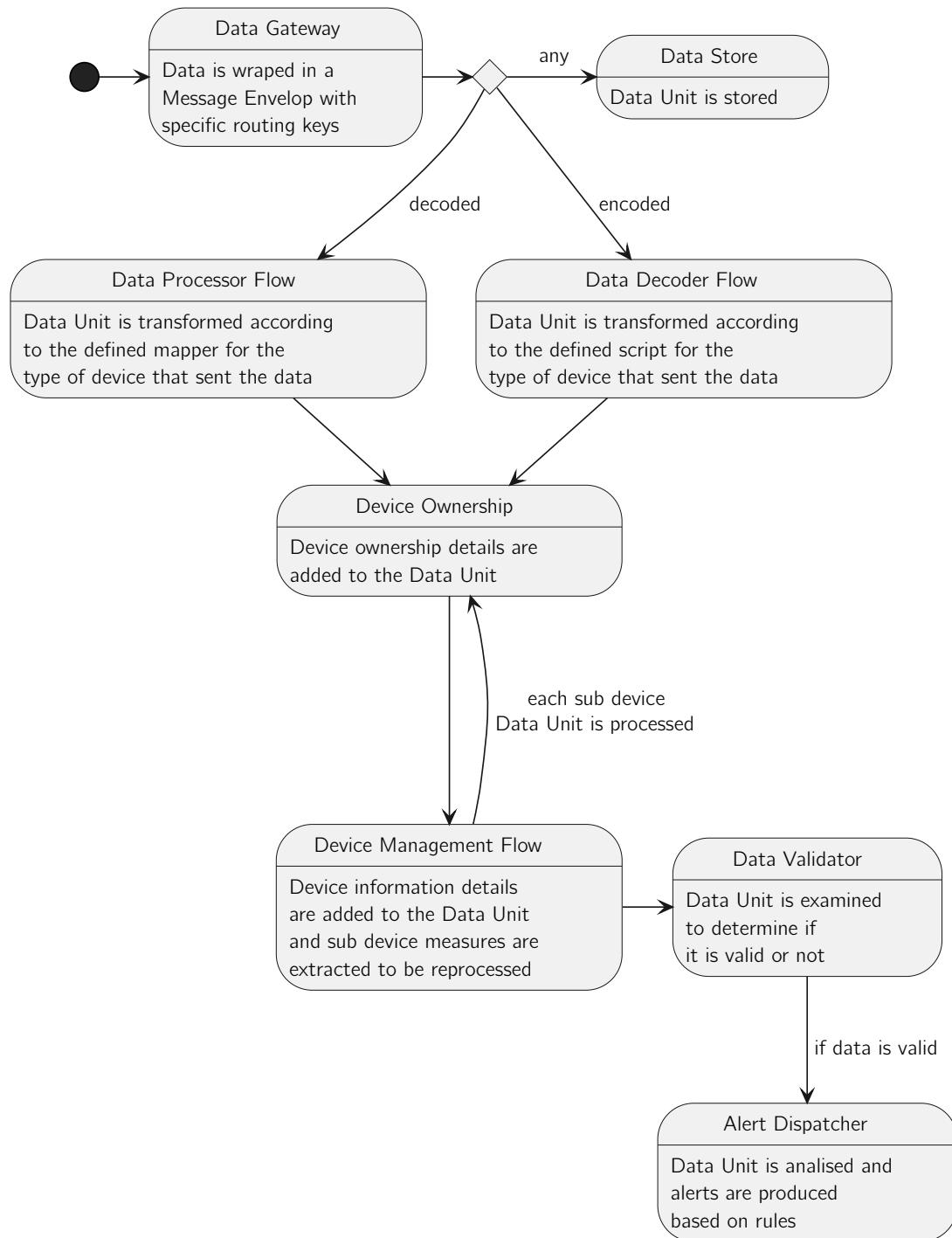


Figure 5.18: Container Level - Data Flow - Diagram

Most of these containers have just a portion of their context state and may be unable to preform the needed operation on some Data Units. The following diagrams, Figure 5.19 and Figure 5.20, addresses how state is managed in Data Decoder Flow Backend and most **Data Flow Scope** containers.

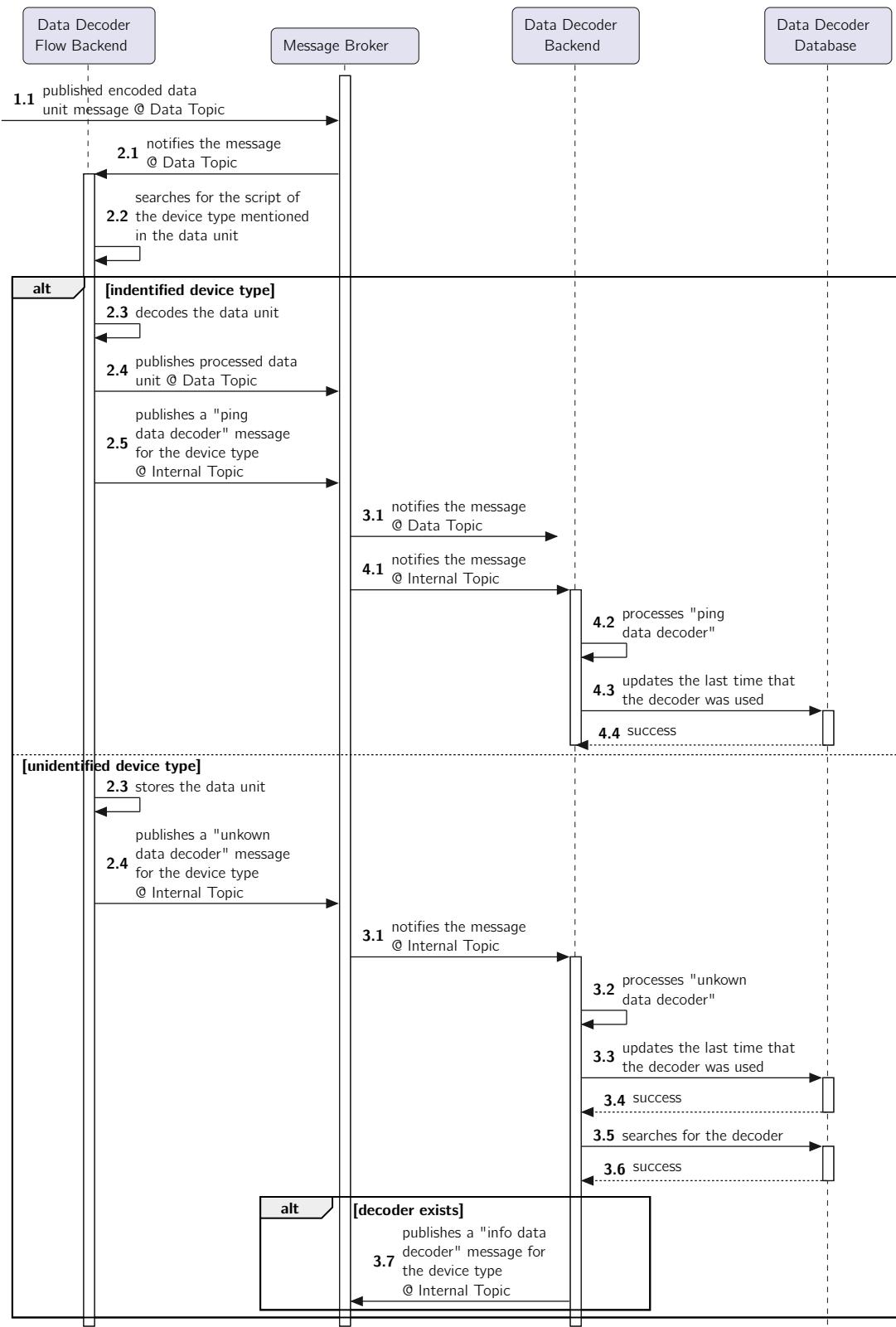


Figure 5.19: Container Level - Data Decoder Operation Part 1 - Process View Diagram

As we can see, the Data Decoder Flow Backend, upon receiving a Data Unit, can perform

two operations, 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 next diagram describes what happens when a message with a decoder is published (using the *OperationType* Info mentioned in Section 5.4.2.3).

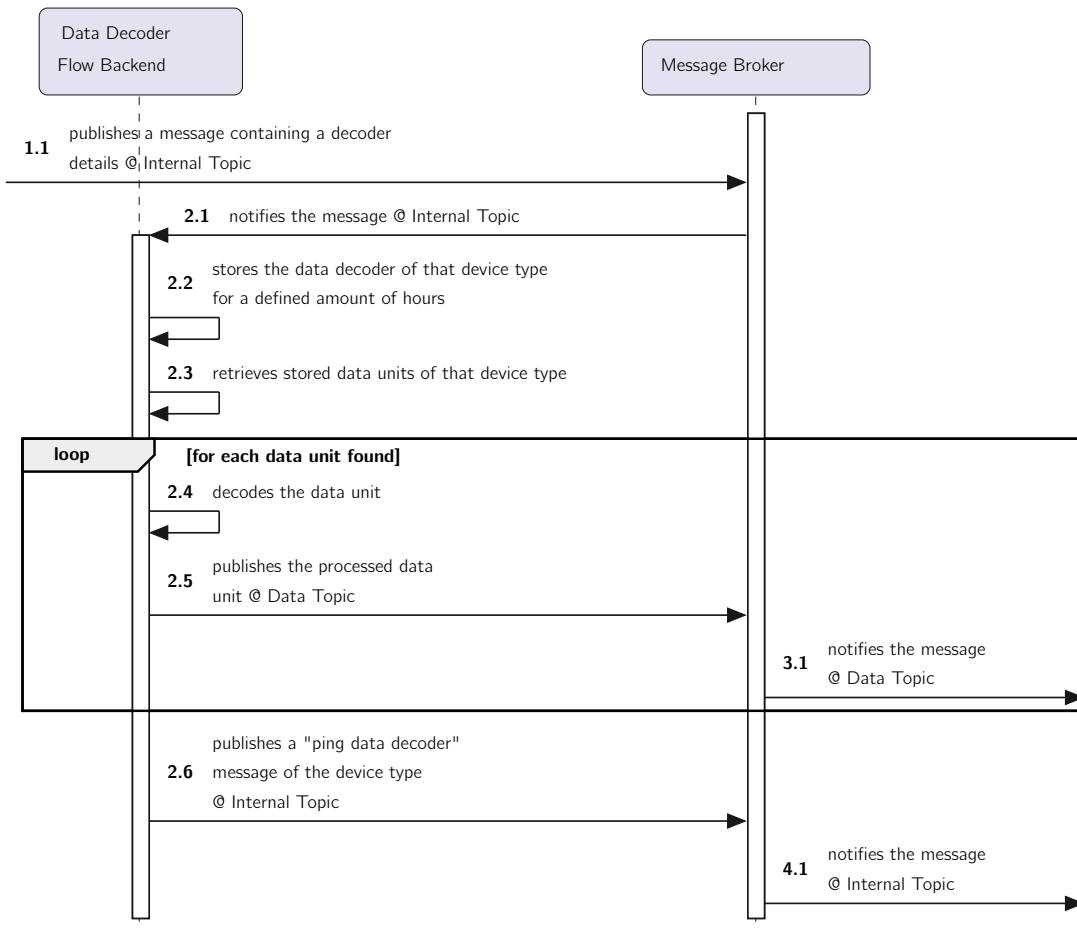


Figure 5.20: Container Level - Data Decoder Operation Part 2 - Process View Diagram

As we can see 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 contexts/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 5.21 and Figure 5.22 present some of the common operations that happen in that scope.

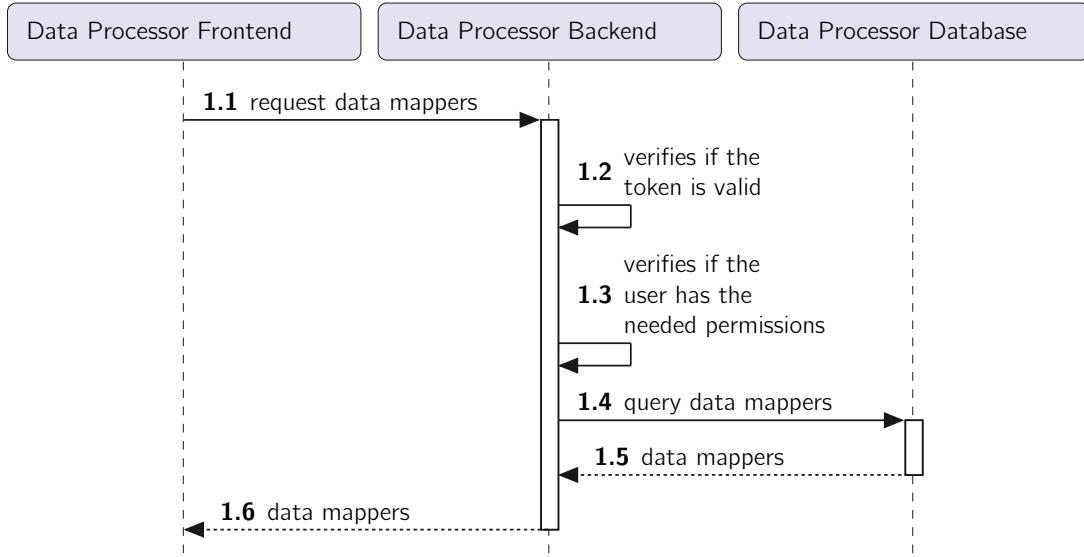


Figure 5.21: Container Level - Consult Data Processors - Process View Diagram

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 5.22 displays an example of this occurrence.

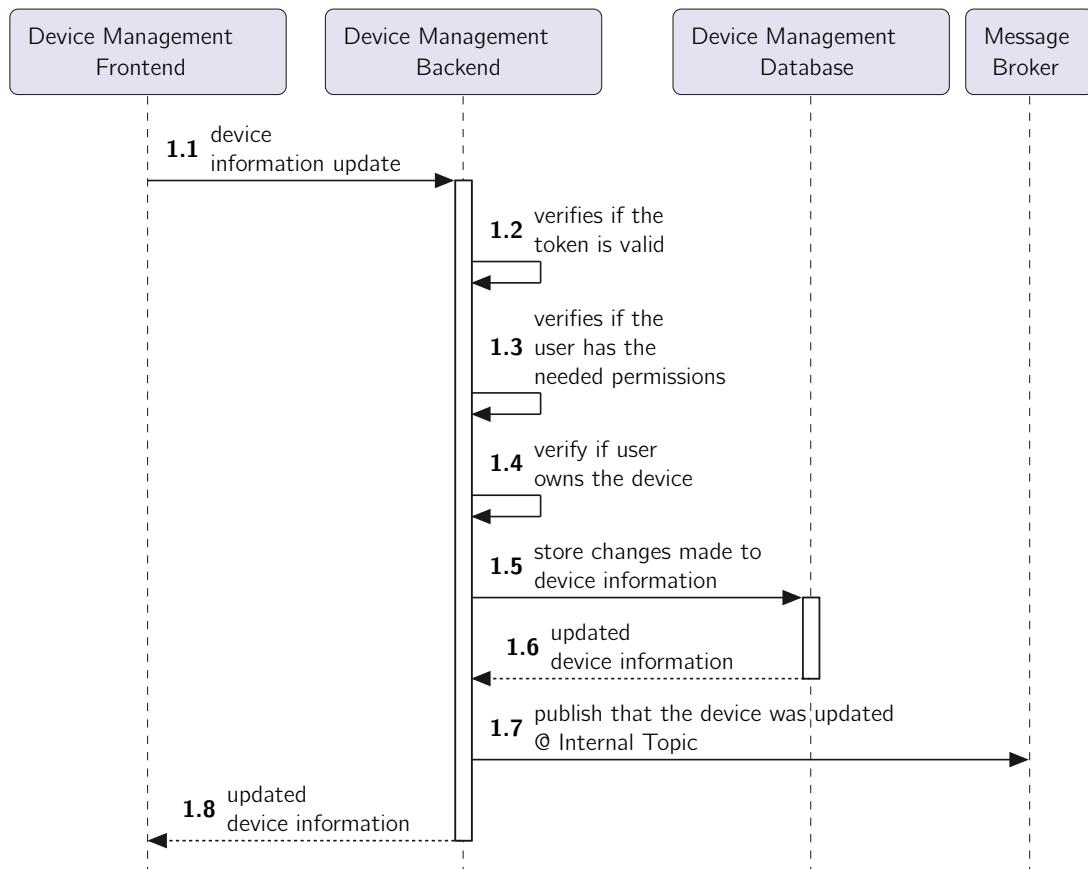


Figure 5.22: Container Level - Edit Device Information - Process View Diagram

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

As an example of this specific event, according to the Section 5.4.2.3, 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 5.23 and Figure 5.24 - authentication and authorization in the **Sensae Console** are addressed, other approaches are discussed in the User Authentication/Authorization Section.

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.

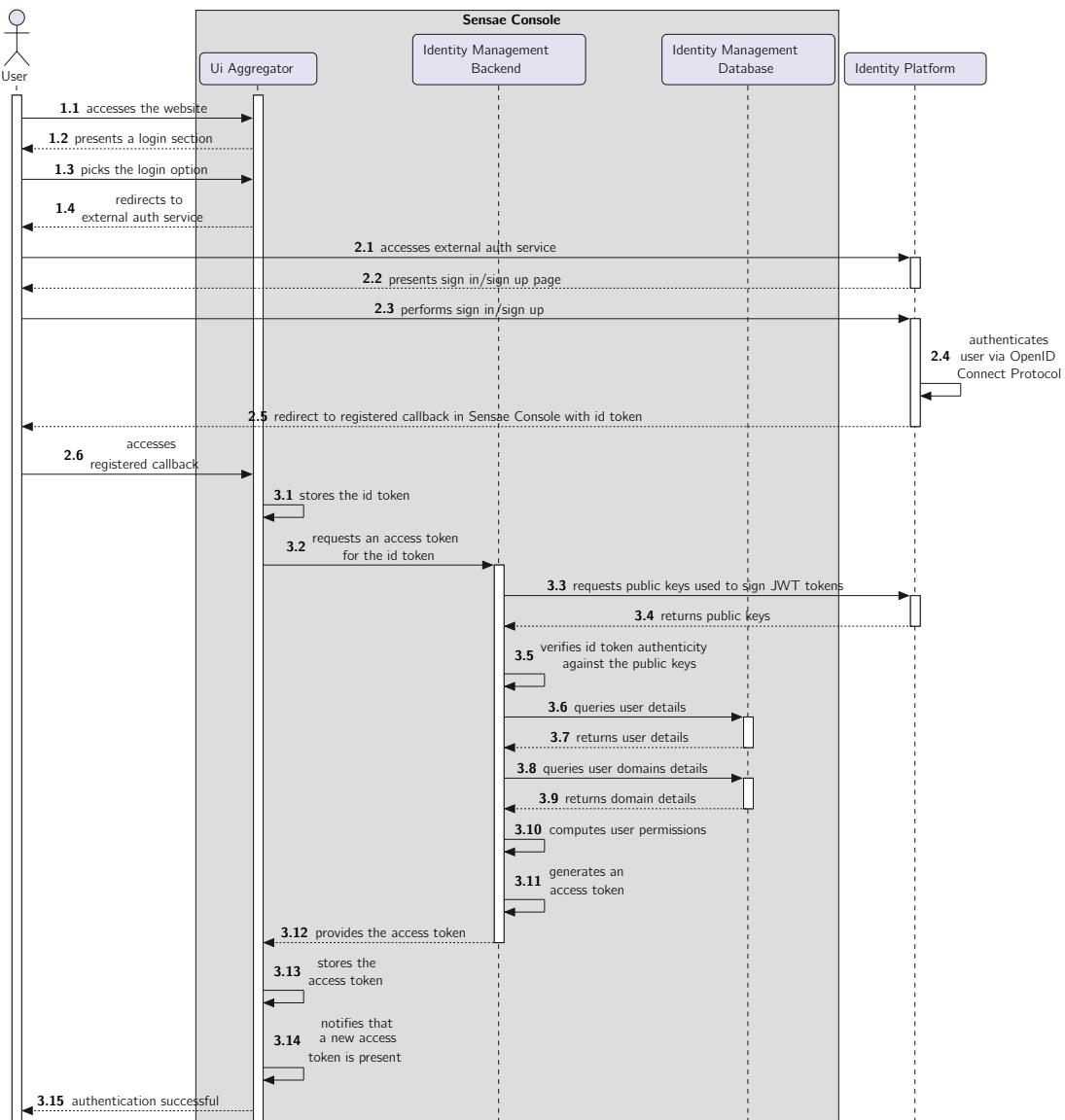


Figure 5.23: Container Level - User Authentication - Process View Diagram

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 5.24.

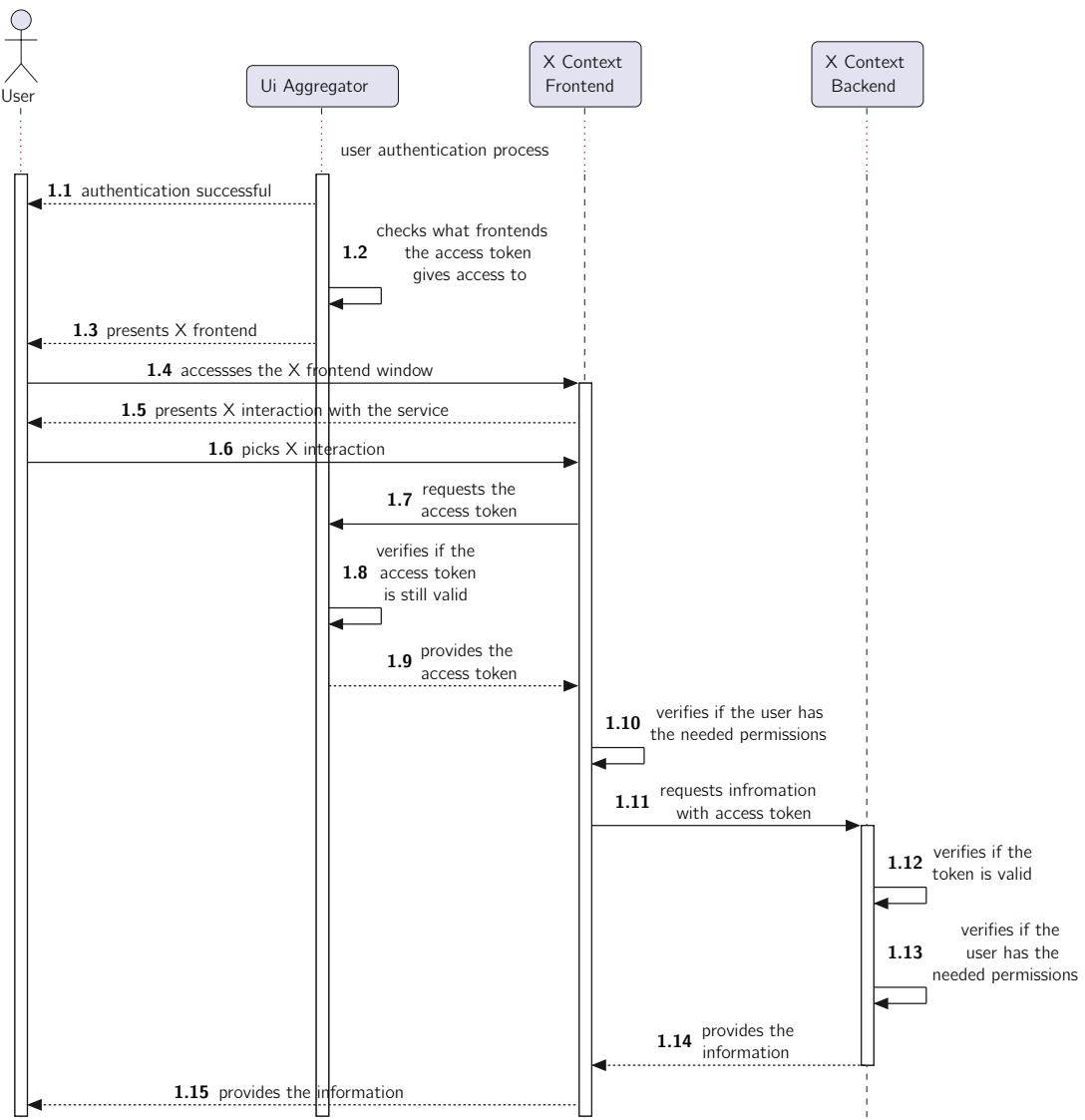


Figure 5.24: Container Level - User Authorization - Process View Diagram

In this diagram the expected behavior for any pair of frontend and backend containers in **Configuration Scope** and **External Services Scope** 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 the Section 5.3.3.

Finally some operations performed in the **External Services Scope** are presented starting with how a user can see the current location of a device via the Fleet Management Service (Figure 5.25). Authentication details will be omitted for brevity reasons.

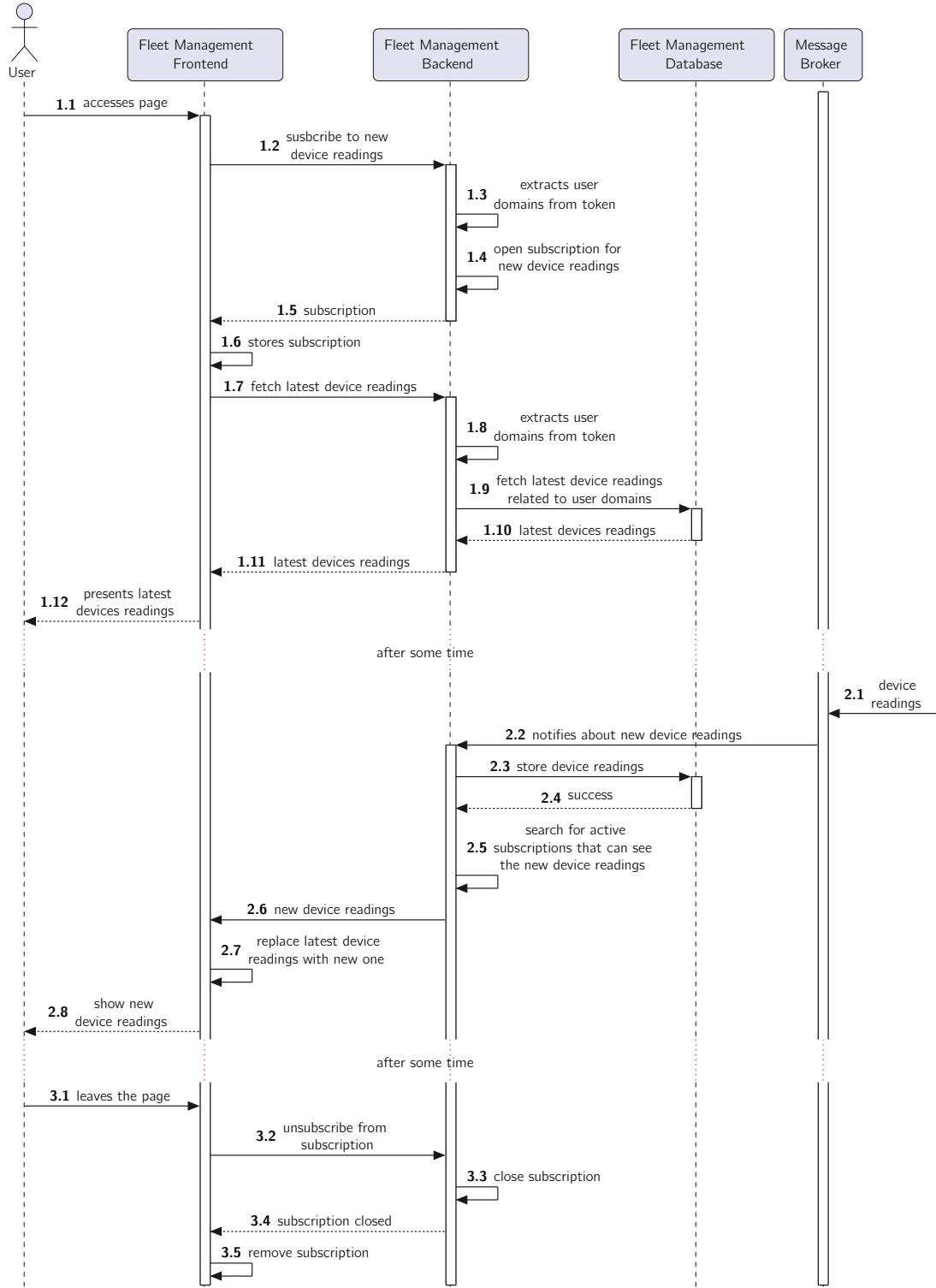


Figure 5.25: Container Level - Consult Device Live Location via Fleet Management - Process View Diagram

In order to provide live information to the user **External Services Scope** services rely 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 **2.6**. First 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**.

The next diagram in Figure 5.26 describes how a user receives notifications via several different delivery channels. For brevity reasons the subscription process is omitted.

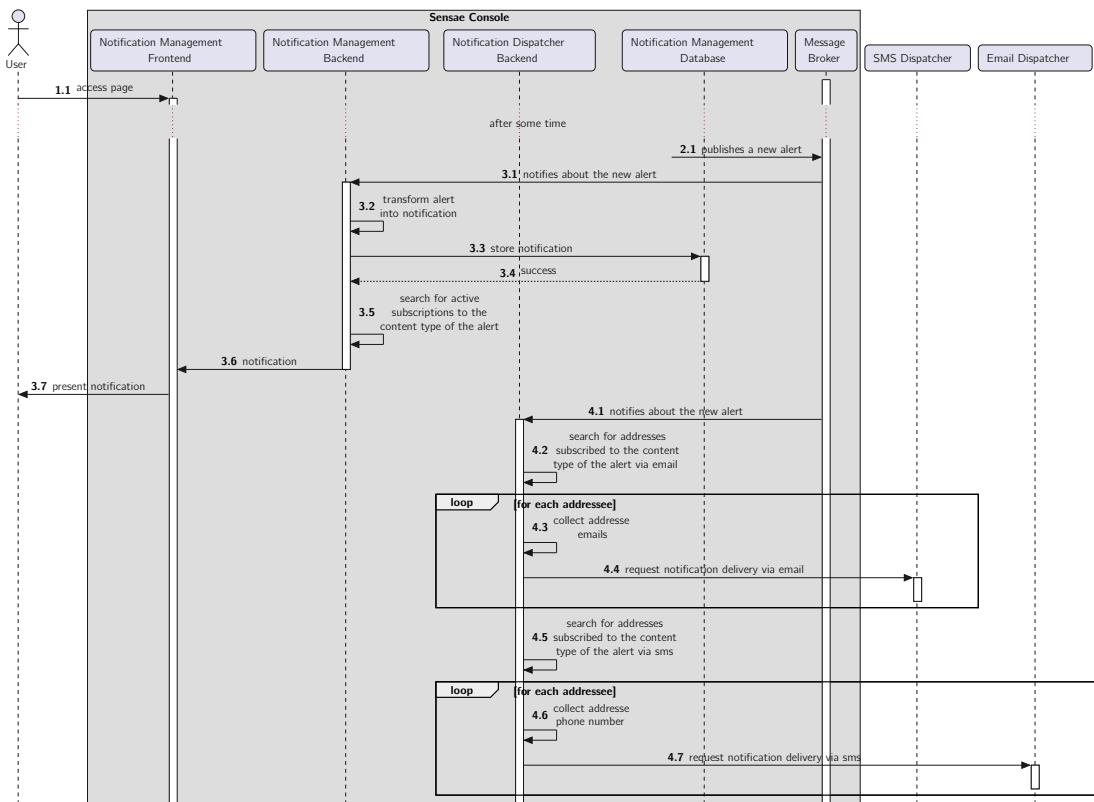


Figure 5.26: Container Level - Receive notification via Notification Management - Process View Diagram

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 through Device Ownership Backend to be enriched with the domains that own it and is then collected by, at least, Notification Management Backend and Notification Dispatcher Backend. Notification Management Backend delivers alerts in the form of UI notifications - step **3.5** and **3.6** - and stores this alert as a notification for later use - step **3.3**. Notification Dispatcher Backend delivers alerts in the form of Emails - step **4.4** - and SMS - step **4.7**.

Certain types of alerts are also collected by Smart Irrigation Backend to automatically control conditions inside an irrigation zone. In the next diagram, Figure 5.27, this process is presented.

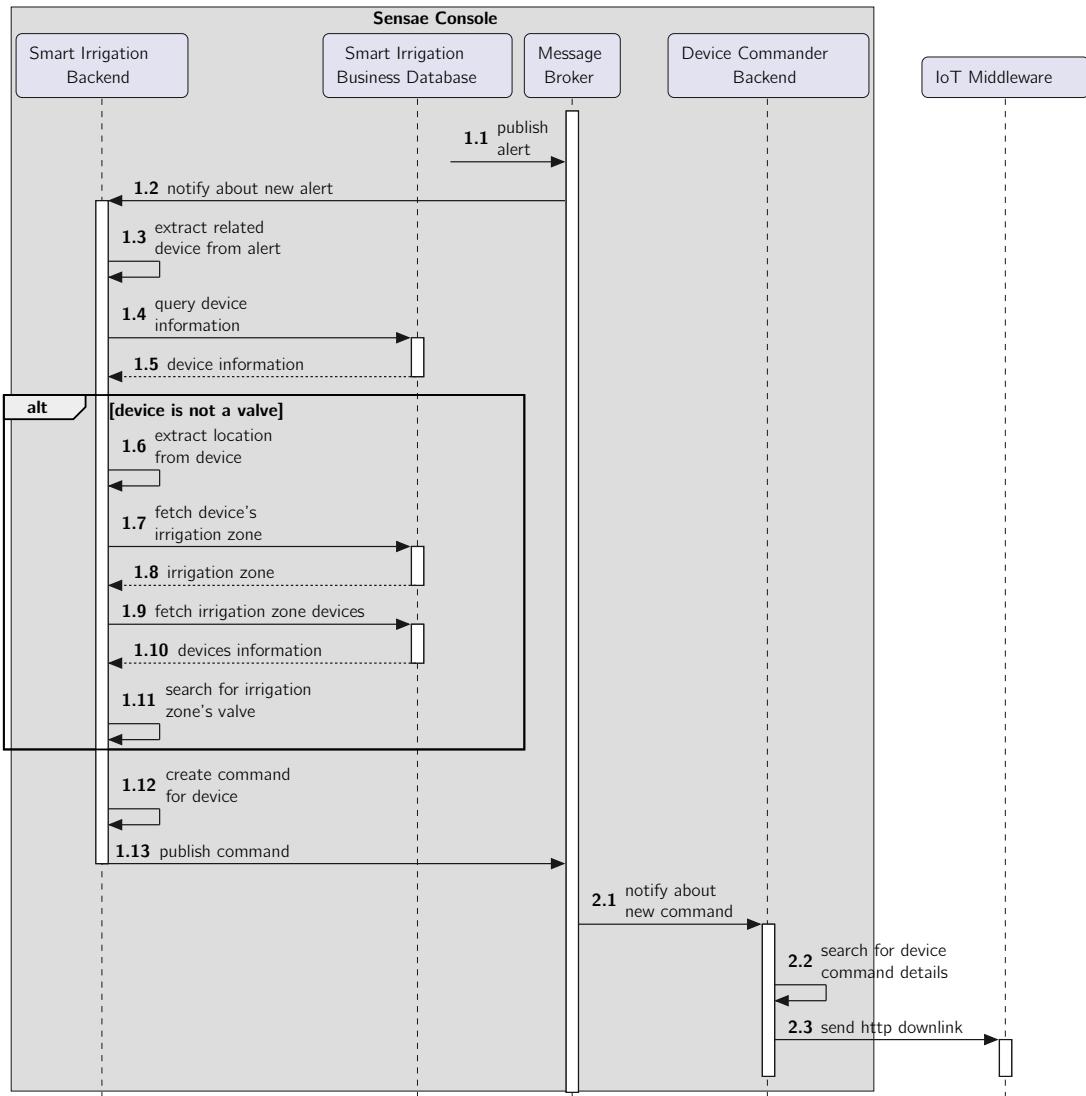


Figure 5.27: Container Level - Valve Activation Process via Smart Irrigation
- Process View Diagram

The alerts created in **Sensae Console** are captured by containers in the **External Services Scope** so that they can act based on the 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.

5.2.2.4 Container Level - Implementation View

Each container mentioned in the Section 5.2.2 is developed inside the same package, *sensae-console*. The following diagrams presents how containers are mapped to packages.

Frontend services are organized according to the diagram in Figure 5.28.

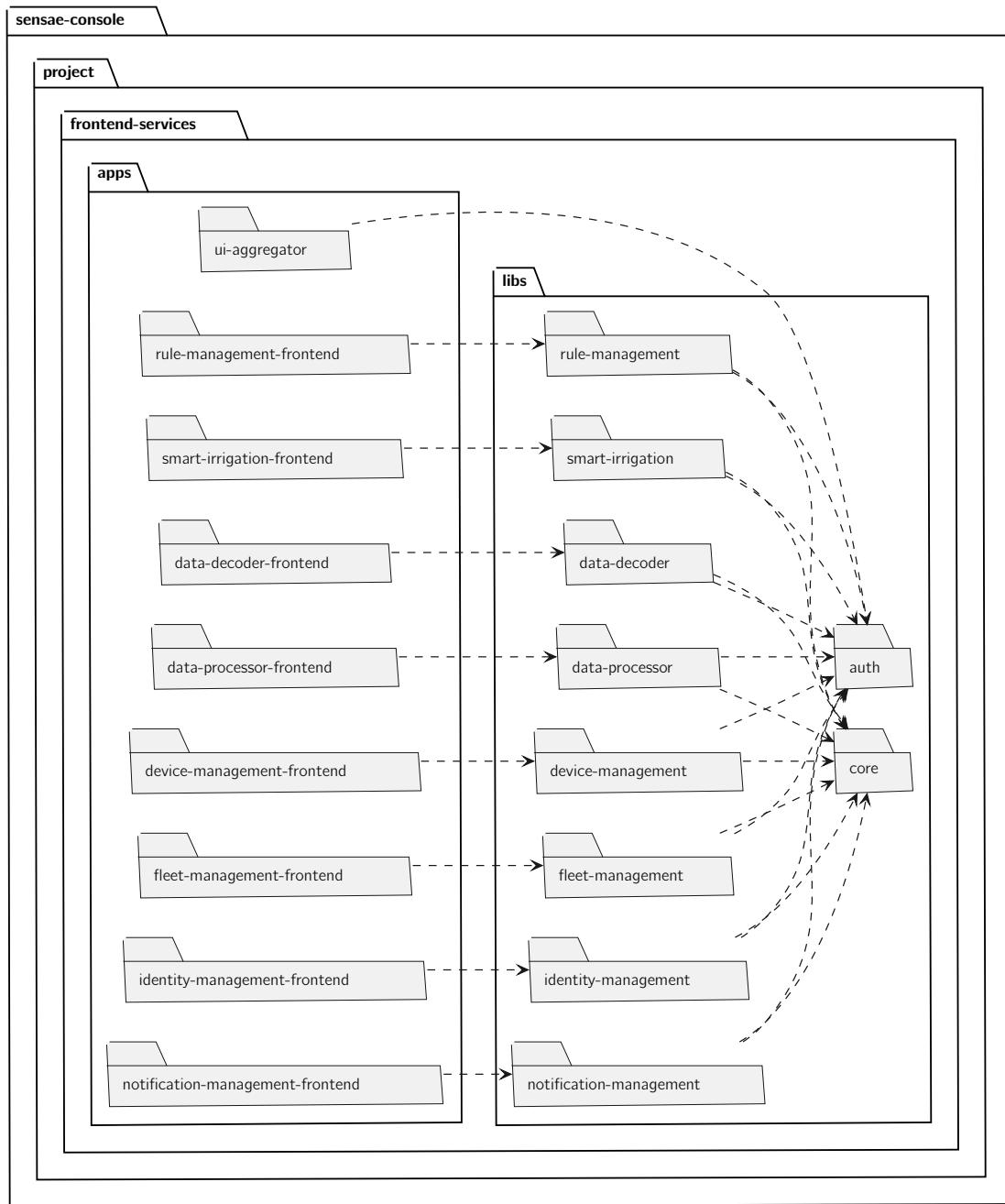


Figure 5.28: Container Level - Frontend Services - Implementation View Diagram

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.

Backend services are organized according to the diagram in Figure 5.29.

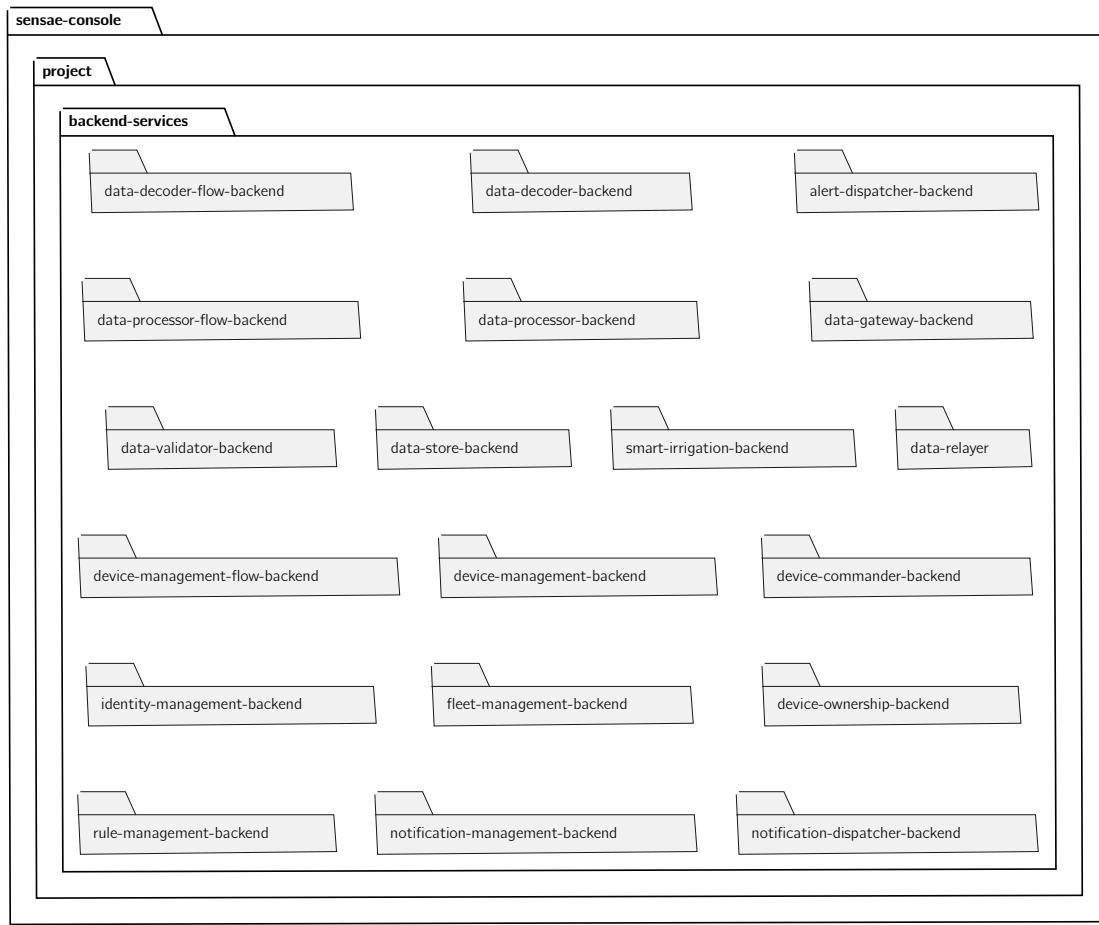


Figure 5.29: Container Level - Backend Services - Implementation View Diagram

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

Database services are organized according to the diagram in Figure 5.29.

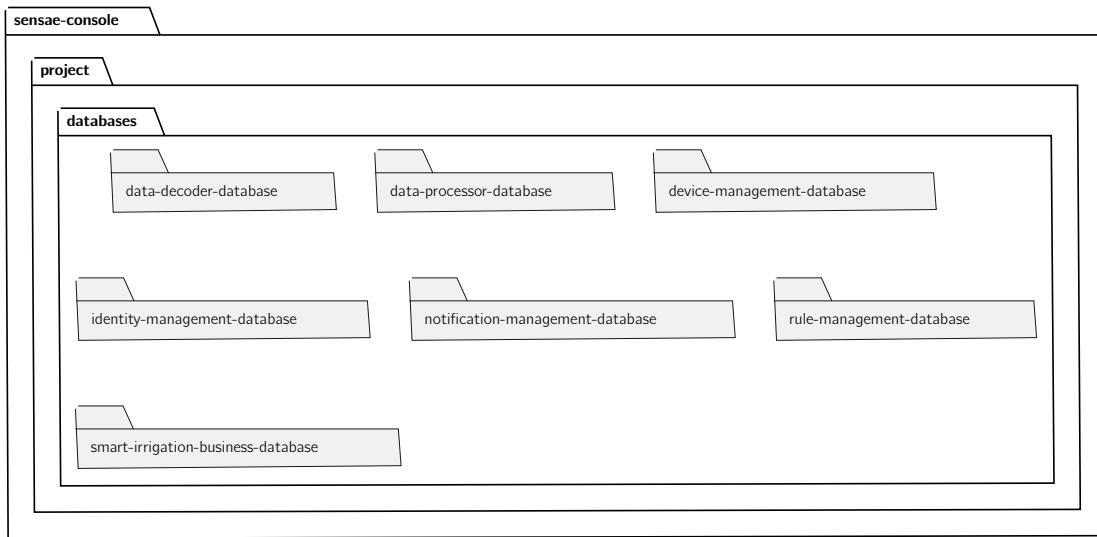


Figure 5.30: Container Level - Database Services - Implementation View Diagram

No database service has been developed, only configured. The Fleet Management Database and Smart Irrigation Data Database needed no configuration and as such aren't associated with any package. The Message Broker also has no package in the project since it didn't need any configuration and wasn't developed.

5.2.2.5 Container Level - Physical View

Next is the physical view (Figure 5.31), 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.

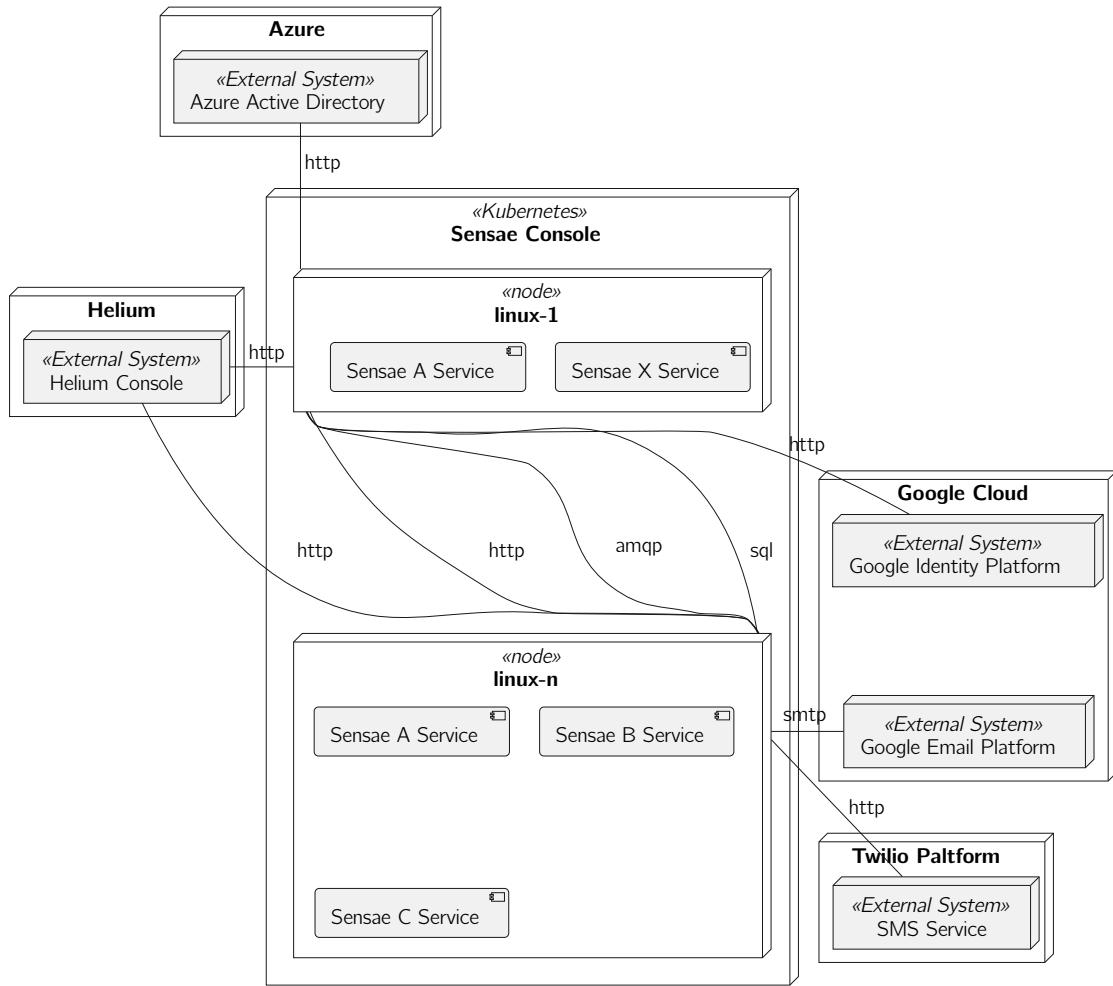


Figure 5.31: Container Level - Physical View Diagram

Even though the diagram above represents a *Kubernetes* cluster, 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.

Each Container represented in Section 5.2.2.2 is mapped to a container in this view. Following the *Database per service Pattern*, each logical database also corresponds to a physical database.

5.2.2.6 Container Level - Synopsis

The container level introduces the reader to the internals of **Sensae Console**. Each container is introduced and the interactions between them are explored. In the following section, Section 5.2.3, the developed containers are presented with a granularity of level 3 (in the C4 model).

5.2.3 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.

5.2.3.1 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 frontend containers;
- **Management Backend Architecture:** used on most service scope backend containers and all configuration scope backends;
- **Data Flow Architecture:** used on most containers related to the Data Flow scope.

Starting with the Frontend Architecture used, it was decided to maintain two distinct domains (Model and DTOS) in order to meet the Single Responsibility 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 and the Utils component has several methods commonly used to process backend requests. These two components are reused in all frontend containers.

As an example, the logical view of the Data Decoder Frontend is presented in Figure 5.32.

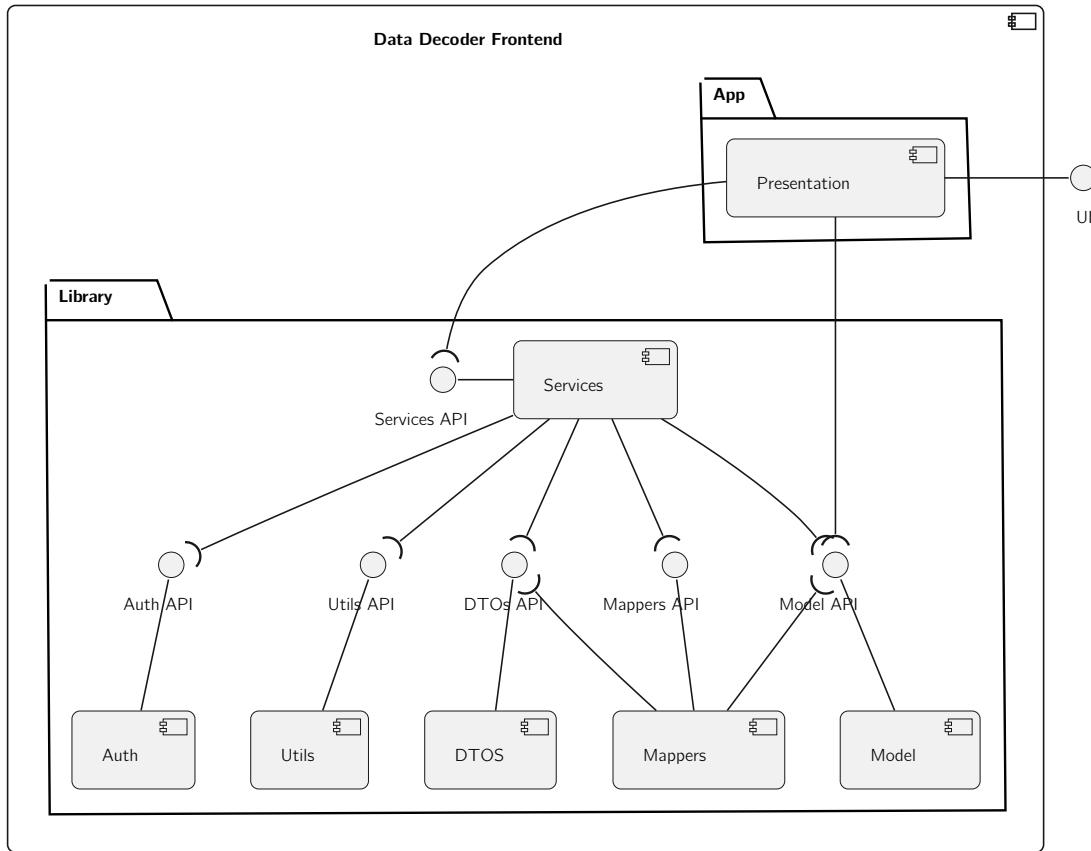


Figure 5.32: Component Level - Data Decoder Frontend - Logical View Diagram

This architecture is used on the containers: (i) Device Management Frontend, (ii) Data Decoder Frontend, (iii) Data Processor Frontend, (iv) Notification Management Frontend, (v) Identity Management Frontend, (vi) Rule Management Frontend, (vii) Fleet Management Frontend and (viii) Smart Irrigation Frontend. The UI Aggregator has a simpler architecture than 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 Management 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 5.33.

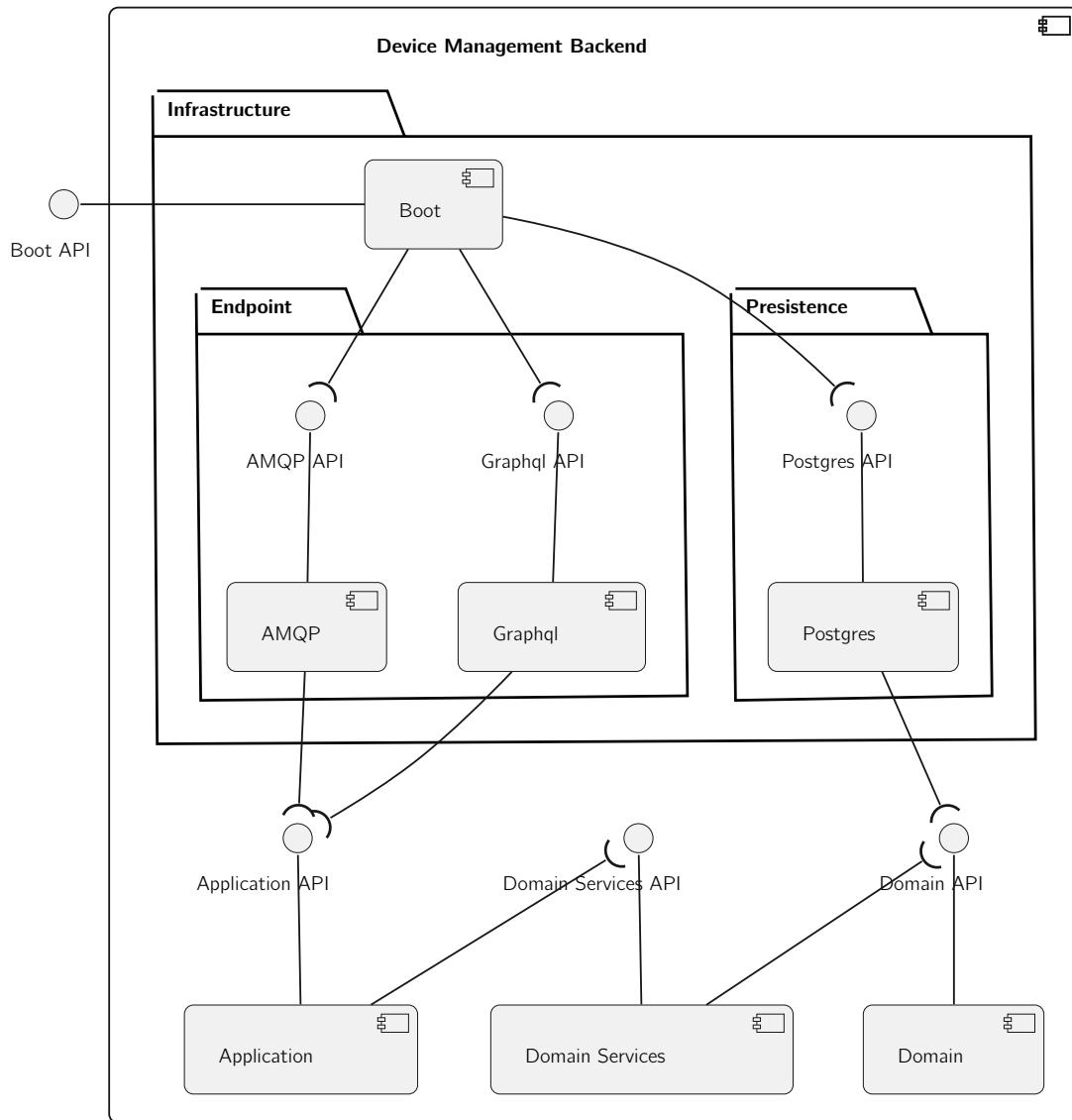


Figure 5.33: Component Level - Device Management Backend - Logical View Diagram

This architecture is used on the containers: (i) Device Management Backend, (ii) Data Decoder Backend, (iii) Data Processor Backend, (iv) Notification Management Backend, (v) Identity Management Backend, (vi) Rule Management Backend and (vii) Fleet Management Backend. The Smart Irrigation Backend has an additional component - QuestDB - inside the Persistence component with the same dependencies as the Postgres component. The Notification Dispatcher's architecture also differs from the architecture here presented. Further details of these differences can be consulted in Appendix C.

The following table, Table 5.2, discusses each component responsibilities.

Table 5.2: Backend components responsibilities

Component	Responsibilities
Infrastructure	<ul style="list-style-type: none"> - Enclose components that manage the Input/Output operations required by the container;
Boot	<ul style="list-style-type: none"> - Manage the start up of the container; - Construct the components' pieces according to the defined dependencies; - Manage the configuration of the container;
Endpoint	<ul style="list-style-type: none"> - Enclose components that are used by external containers to interact with the container;
AMQP	<ul style="list-style-type: none"> - Define how to consume and publish events in the Message Broker; - Delegate the handling of events received to specific Application processes;
GraphQL	<ul style="list-style-type: none"> - Define the interface to be consumed by the frontend; - Delegate external requests made to specific Application processes;
Persistence	<ul style="list-style-type: none"> - Enclose components that interface with containers responsible for persisting data;
Postgres	<ul style="list-style-type: none"> - Interact with a database to persist and query data;
Application	<ul style="list-style-type: none"> - 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	<ul style="list-style-type: none"> - Represent business processes; - Interact with the Domain; - Ensure the persistence of the data in question, requiring this responsibility to the Persistence;
Domain	<ul style="list-style-type: none"> - 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 this containers are much simpler.

As an example the logical view of the Device Ownership Backend is presented in Figure 5.34.

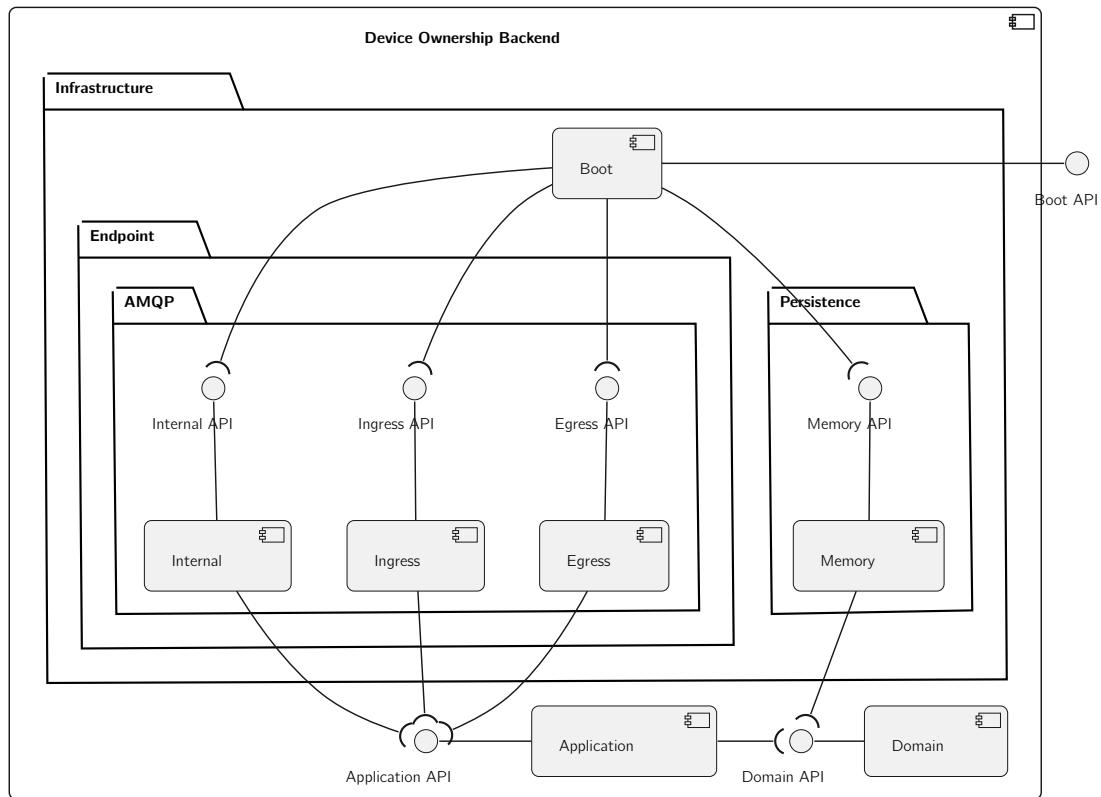


Figure 5.34: Component Level - Device Ownership Backend - Logical View Diagram

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 C.

5.2.3.2 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 5.35, details this process.

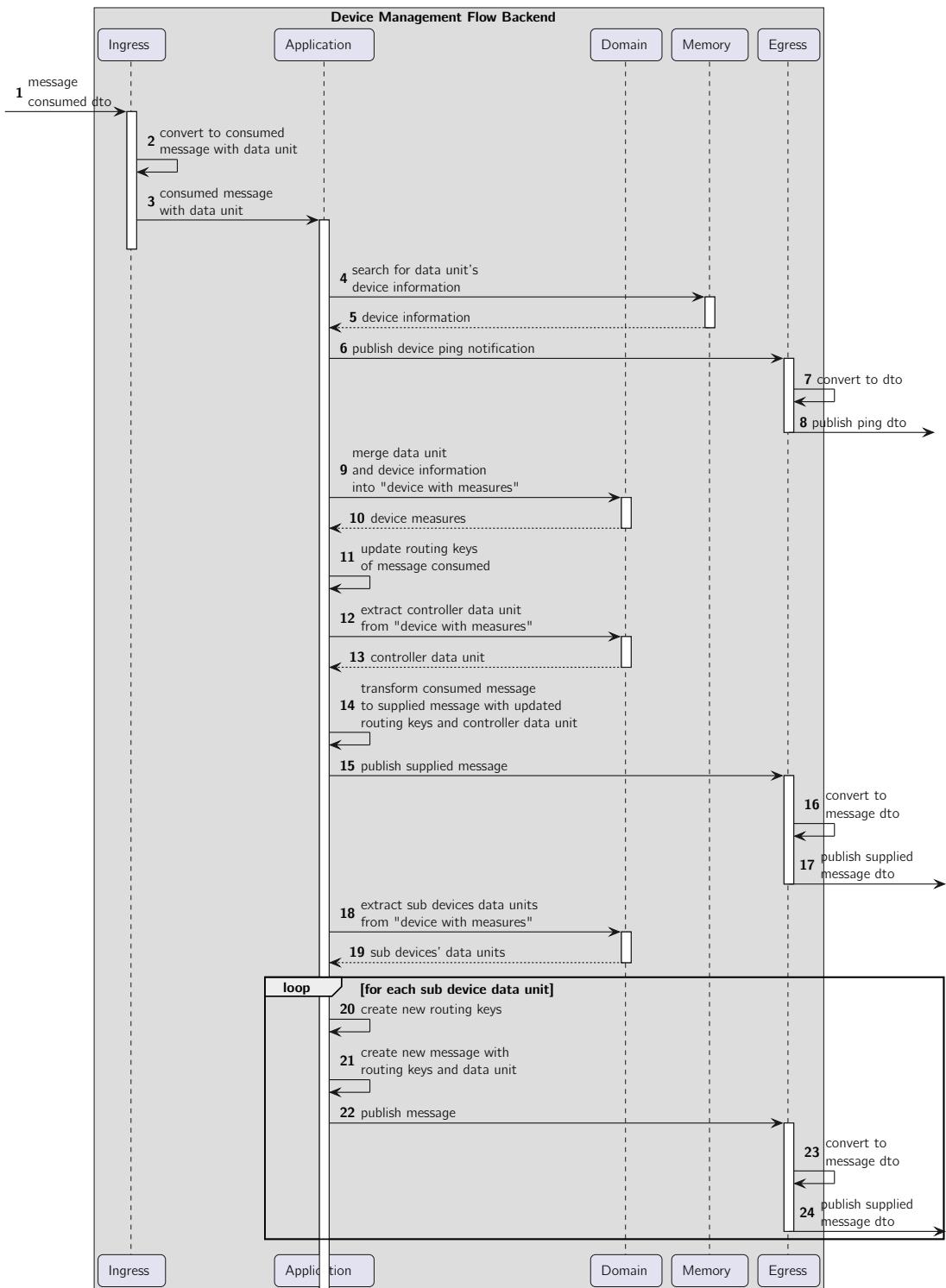


Figure 5.35: Component Level - Process Data Unit in Device Management Flow Backend - Process View Diagram

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 **16** and **23**. 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;
- For each sub device of the controller, a new data unit with that device measures is published in the system - steps **20** to **24**;

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 5.36, tackles this concern.

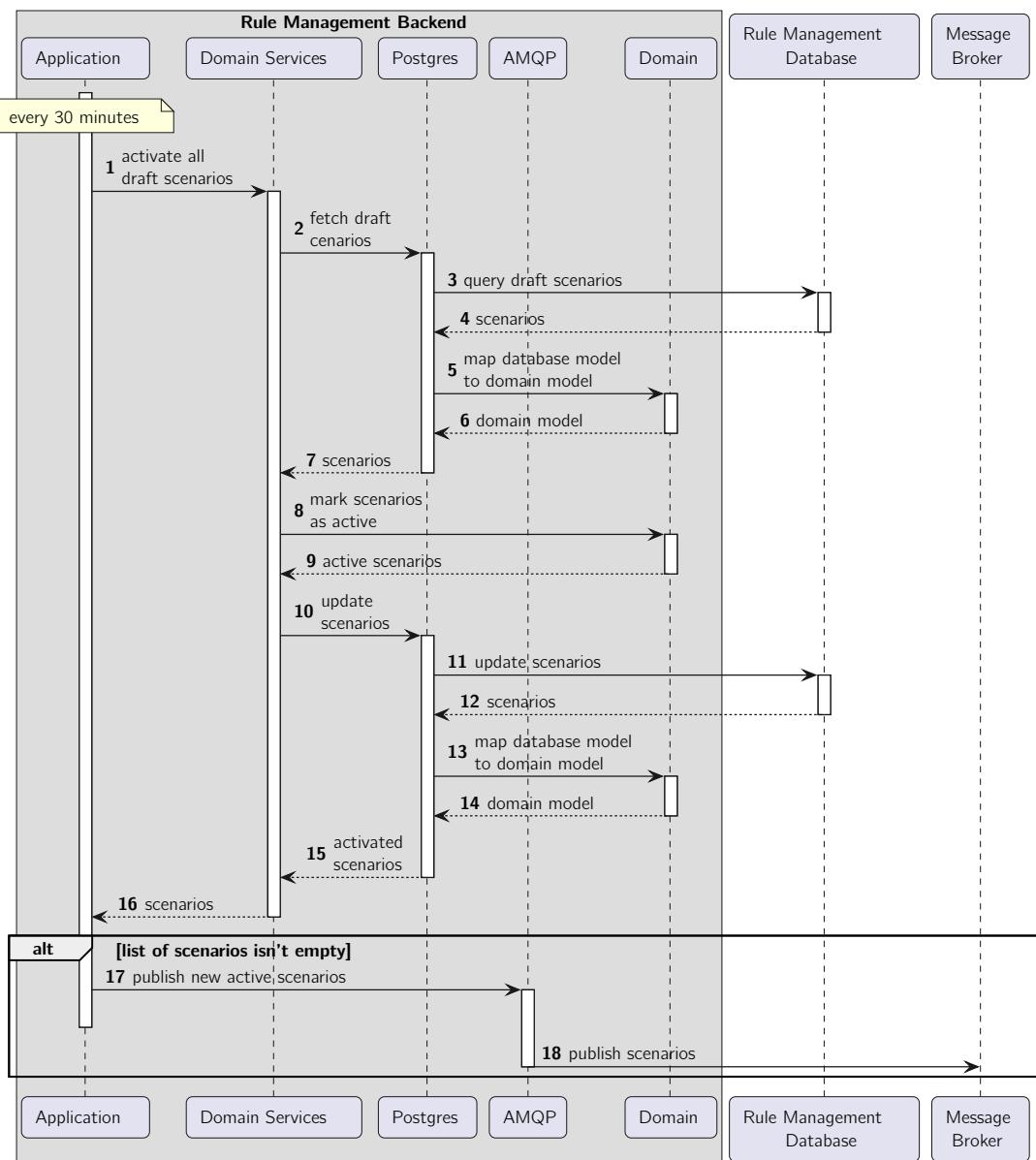


Figure 5.36: Component Level - Deploy Draft Rule Scenarios in Rule Management Backend - Process View Diagram

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**.

5.2.3.3 Components Level - Implementation View

The implementation view of each container can also be condensed in the same 3 distinct types presented in the Section Components Level - Logical View.

The next diagrams, Figure 5.37, Figure 5.38 and Figure 5.39 describe this view at the components level.

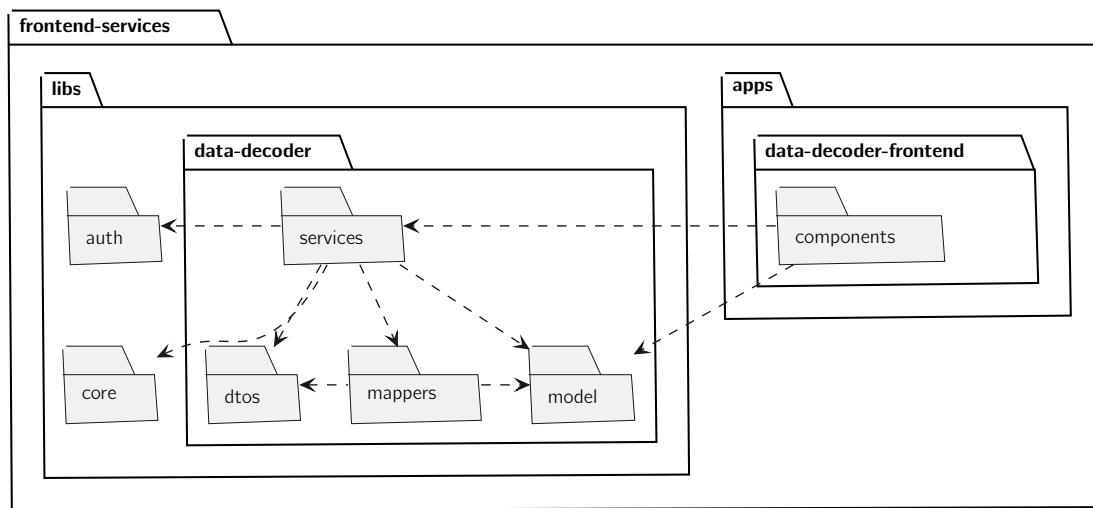


Figure 5.37: Component Level - Data Decoder Frontend - Implementation View Diagram

The packages presented correspond to the components described in the logical view (Figure 5.32). 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.

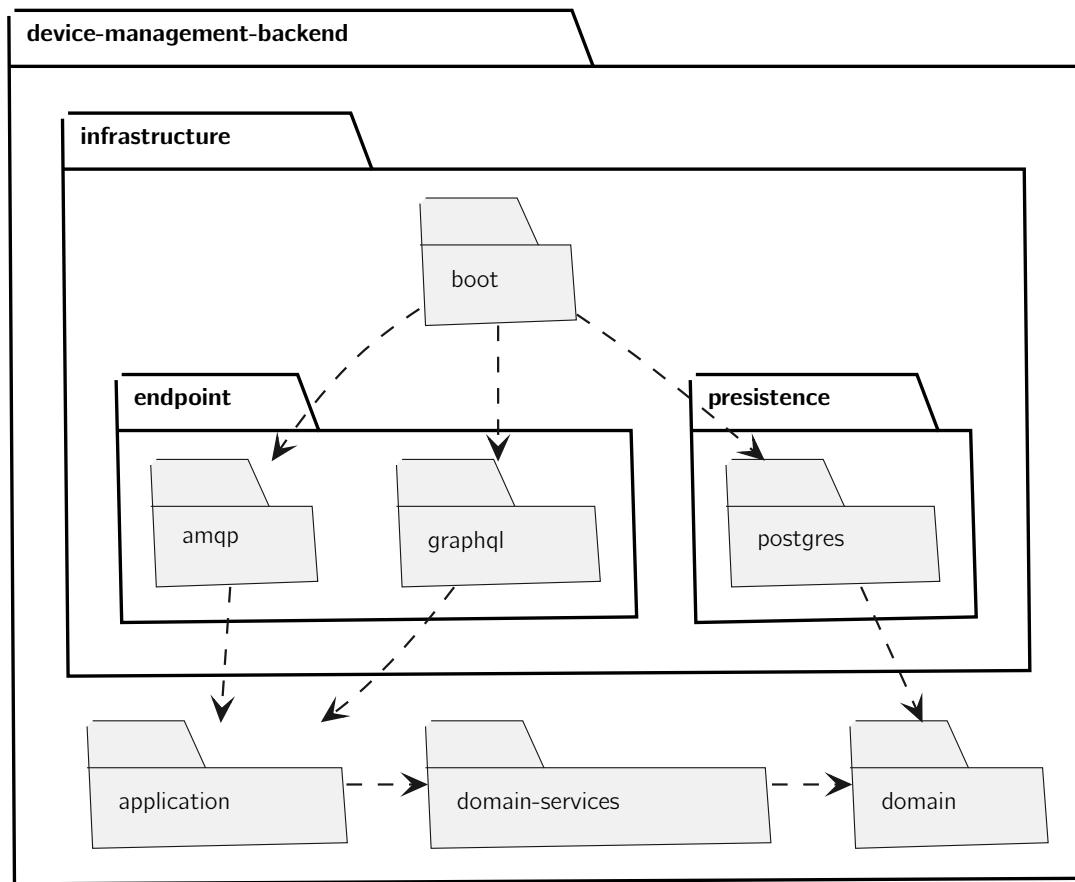


Figure 5.38: Component Level - Device Management Backend - Implementation View Diagram

The packages presented correspond to the components described in the logical view (Figure 5.33). The names given in both views differ only on the case used.

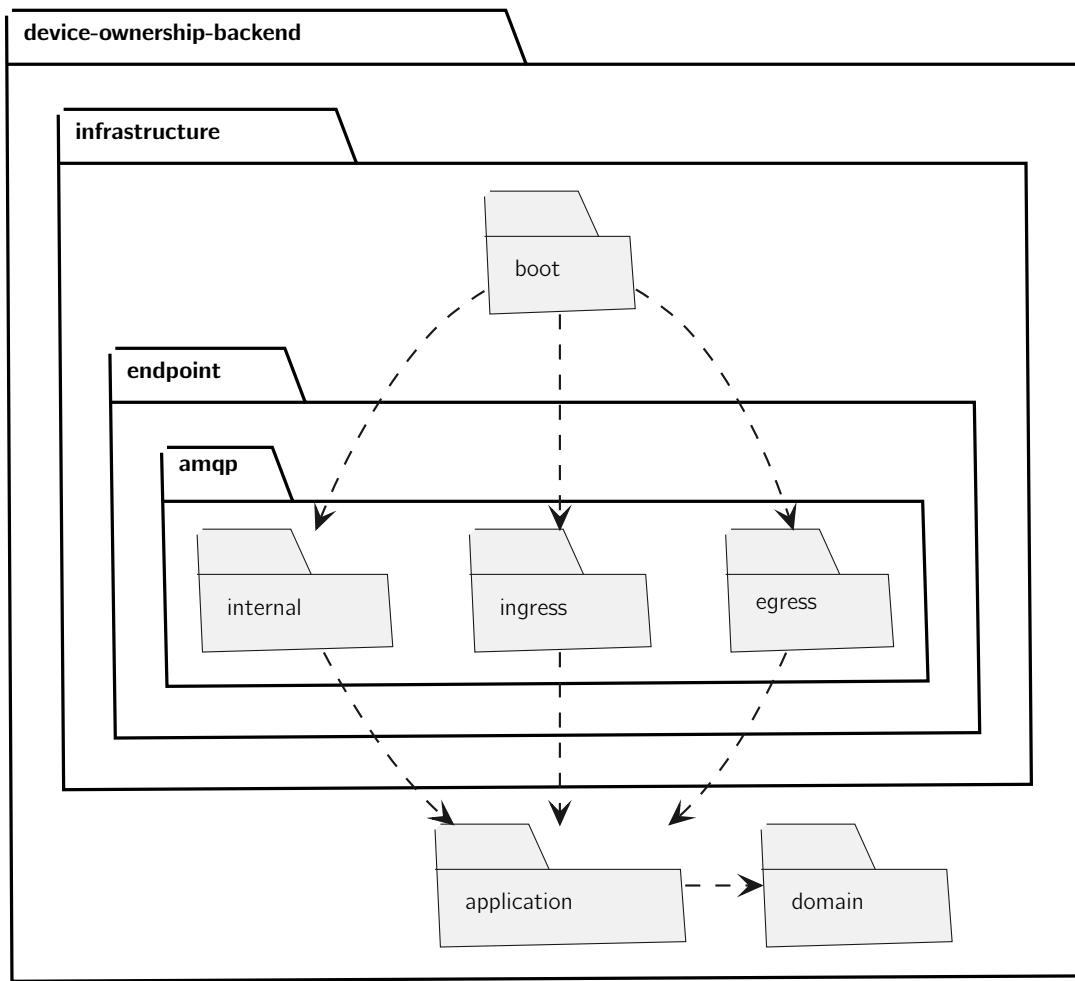


Figure 5.39: Component Level - Device Ownership Backend - Implementation View Diagram

The packages presented correspond to the components described in the logical view (Figure 5.34). The names given in both views differ only on the case used.

5.2.3.4 Components Level - Synopsis

This section presented the architecture used in the developed containers, how software is organized and how some internal process are executed inside this containers. In the following section alternatives to what was designed and developed are discussed.

5.3 Architectural Alternatives Discussed

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

5.3.1 Backend Segregation

There are three main architectural approaches to this topic: Monolithic Backend - Richardson 2021b -, Service Oriented Architecture (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 Disk 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 assigned to a set of customers. This option is crucial and always possible once the business grows and starts to assist various customers.

The following picture, Figure 5.40, summarizes how each architect scales, the SOA behaves similarly to the microservices architecture presented.

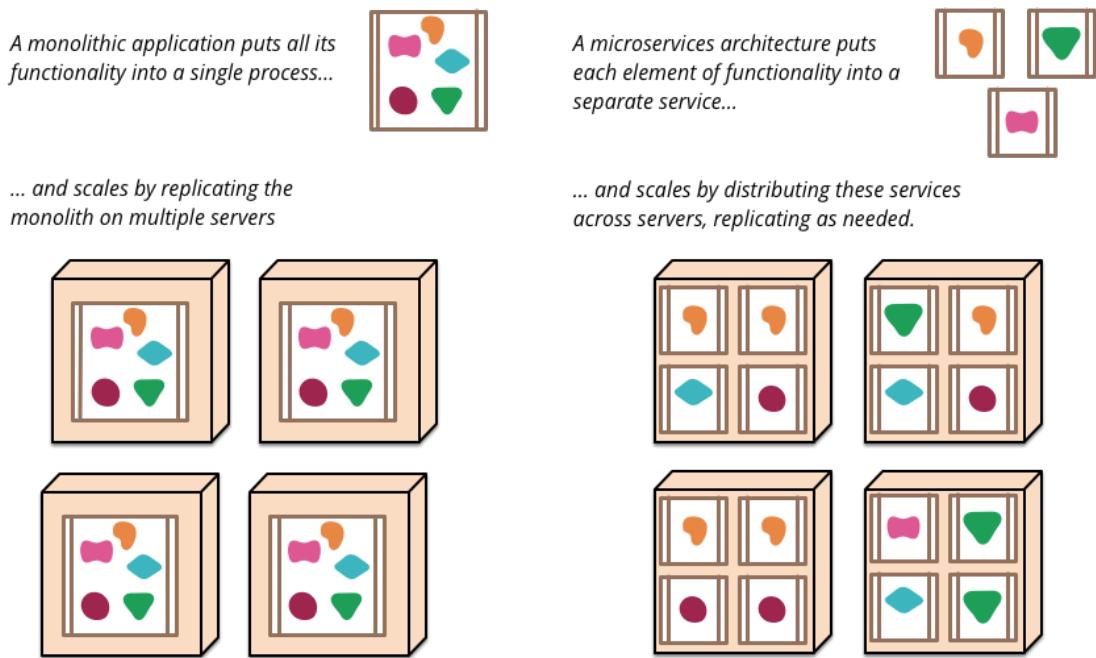


Figure 5.40: Monoliths and Microservices by Fowler and J. Lewis 2014

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 through 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 customers with diverse requirements regarding the deployment and development of the solution;
- Each customer 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 5.4.2) was created to ease the communication between services. This is something common in projects that follow SOA according to Cerny, Donahoo, and Pechanec 2017.

5.3.2 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 4.2 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 IoT 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.

5.3.3 User Authentication/Authorization

User Authentication/Authorization is an important aspect of the solution. During the requirements elicitation, mentioned in Section 4.1.1, 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.

5.3.3.1 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 5.41, presents the normal environment flow for this alternative.

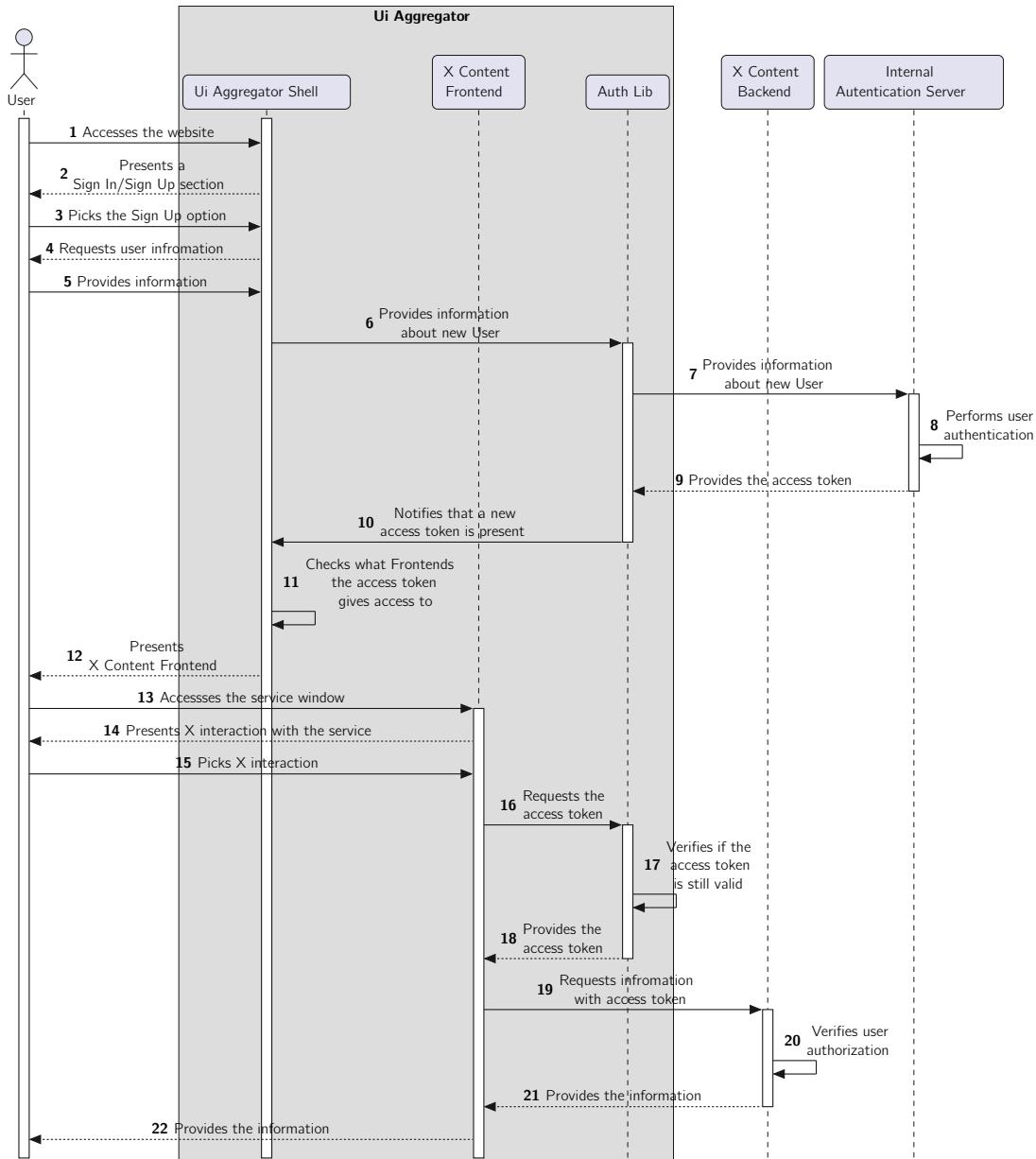


Figure 5.41: User Authentication/Authorization - Internal Authentication Server Alternative - Sequence Diagram

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.

5.3.3.2 External Authentication Server

By using an External Authentication Server there would be no need to store user credentials or permissions. These 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 5.42, presents the normal environment flow for this alternative.

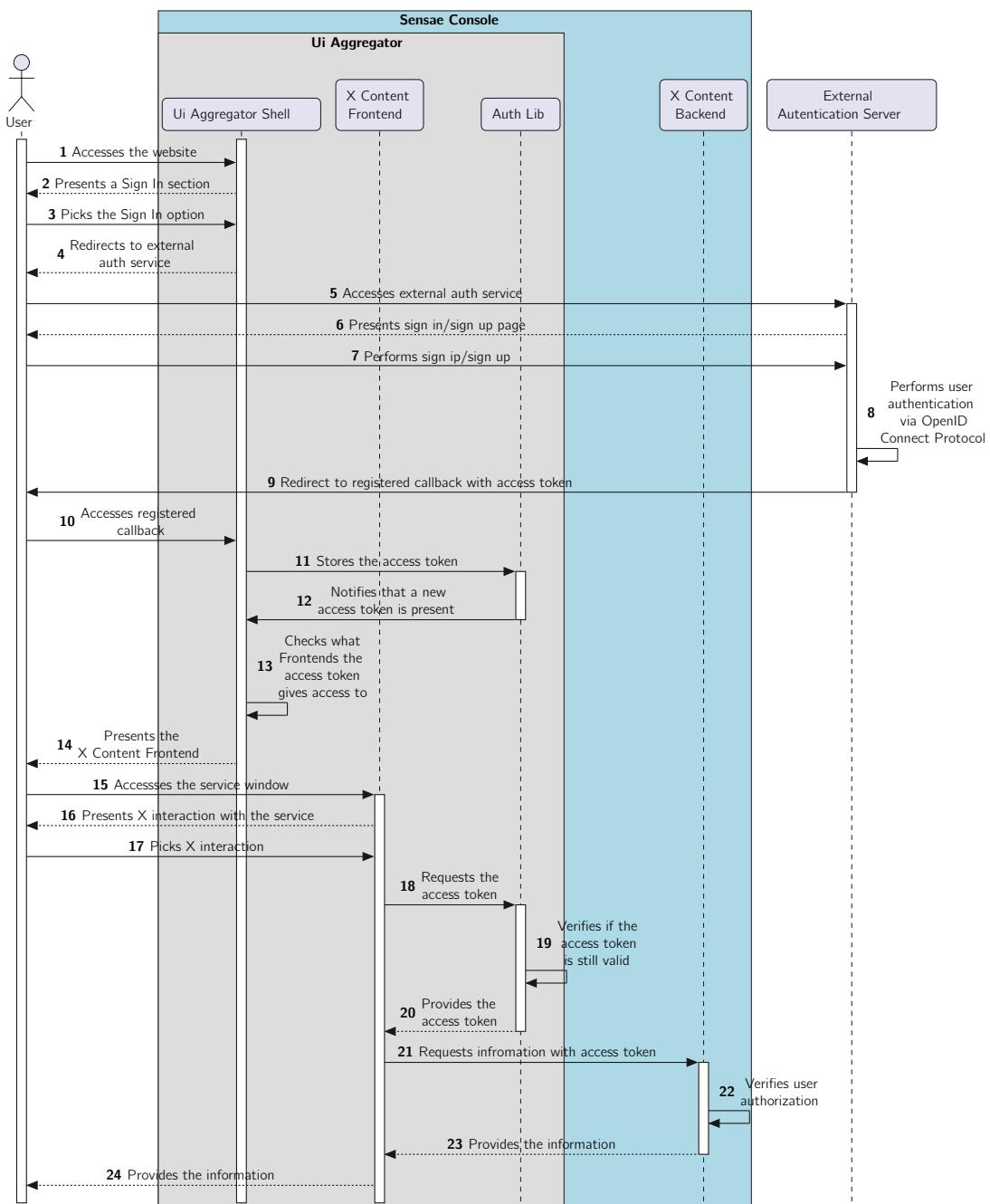


Figure 5.42: User Authentication/Authorization - External Authorization Server Alternative - Sequence Diagram

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 these 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.

5.3.3.3 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 an *Authorization Server*.

The following diagram, Figure 5.43, presents the normal environment flow for this alternative.

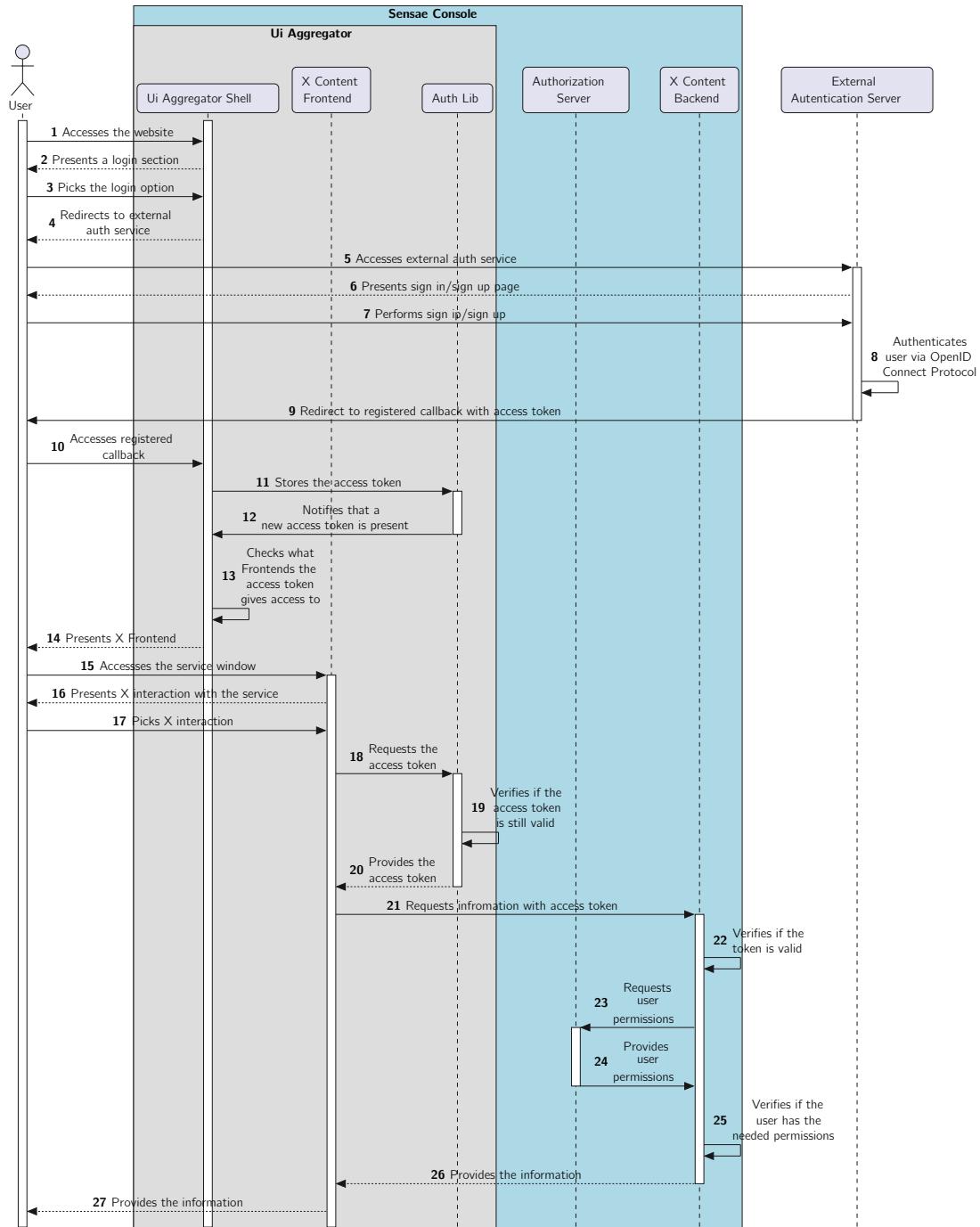


Figure 5.43: User Authentication/Authorization - External Authentication Server with Internal Authorization Server Alternative - Sequence Diagram

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 perform 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*.

5.3.3.4 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 5.23 and Figure 5.24 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.

5.3.4 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 5.18 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 modified 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.

5.3.5 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 Context;
2. Data Flow Containers request information to their context's Configuration Scope Container via synchronous calls;
3. Data Flow Containers are feed updates to their context 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.

5.3.5.1 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 5.44 describes how this option functions.

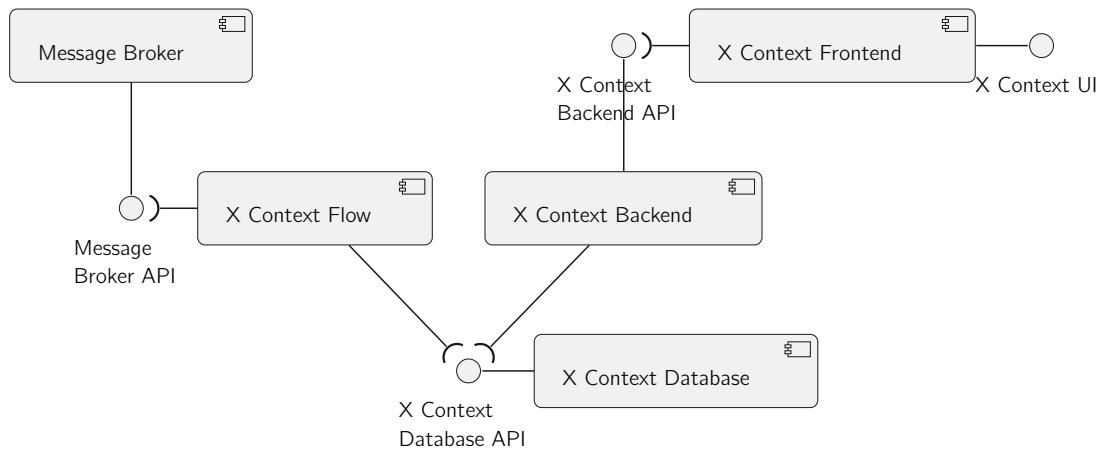


Figure 5.44: Internal Communication - First Option - Logical View Diagram

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 Context Flow* Container and the *X Context Database* Container. This approach has several drawbacks such as:

- The *X Context 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.

5.3.5.2 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 5.45 describes how this option functions.

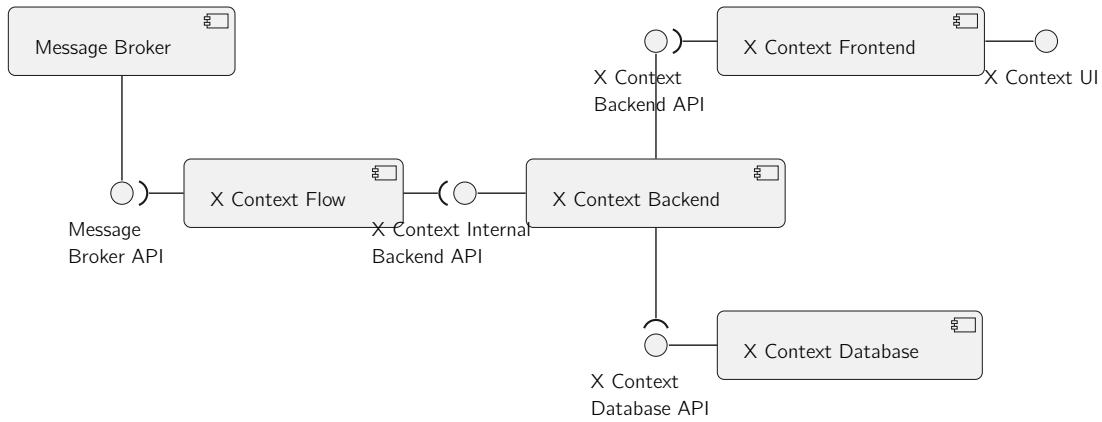


Figure 5.45: Internal Communication - Second Option - Logical View Diagram

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

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

5.3.5.3 Third Option

This option ensures that the Data Flow Containers are kept updated by allowing them to subscribe to changes made in their context's configuration. The logical view diagram in Figure 5.46 describes how this option functions.

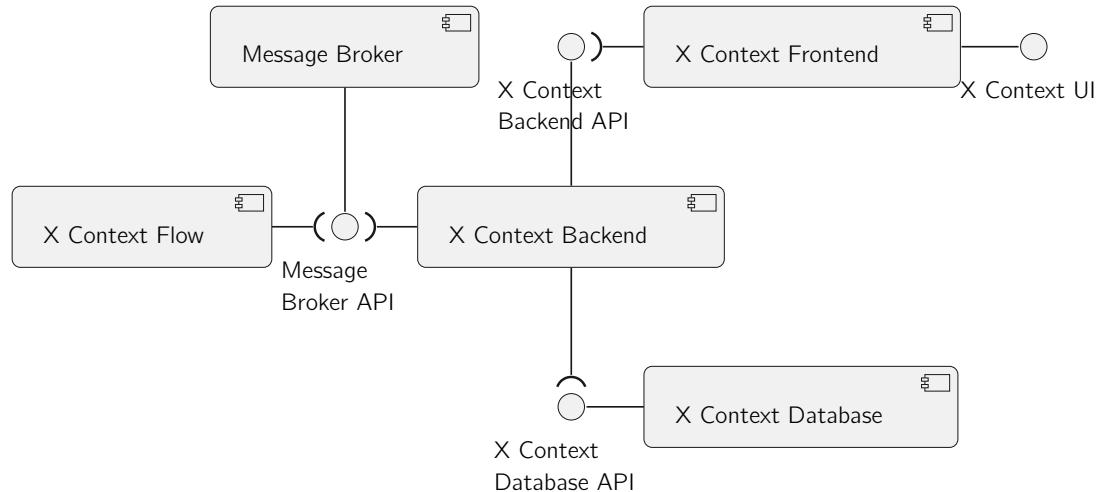


Figure 5.46: Internal Communication - Third Option - Logical View Diagram

The major improvement of this approach when compared with the options above is that, since *X Context Flow* Container subscribes to configuration updates, it can reliably keep a cache with just the needed information (and not the entire context configuration). This works since *X Context 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 Context 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 drawbacks.

5.3.5.4 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 5.47 describes how this option functions.

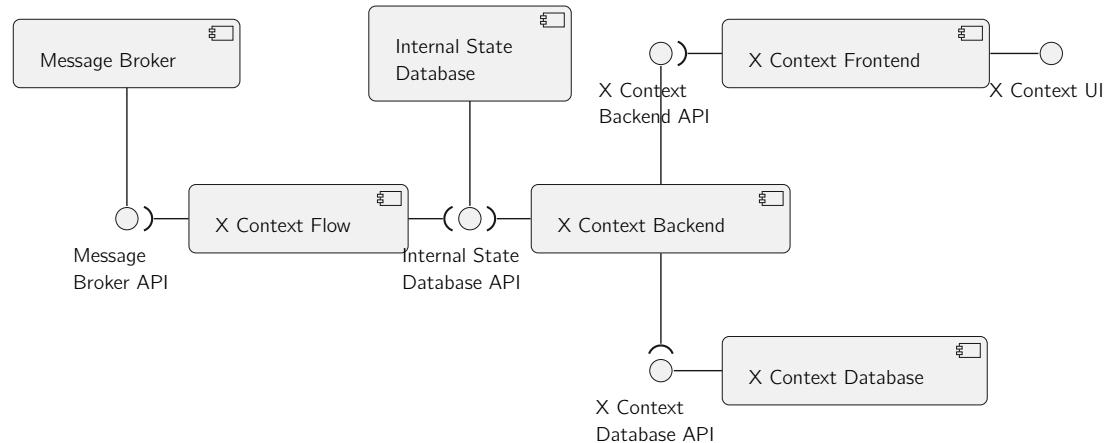


Figure 5.47: Internal Communication - Fourth Option - Logical View Diagram

This approach would remove the responsibility 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 though they can be mitigated by leveraging technologies that tackle distributed caching problems.

5.3.5.5 Fifth Option

This option ensures that the Data Flow Containers are kept updated by allowing them to subscribe to changes made in their context's configuration. This option diverges from the third option since the event store would persist all updates to contexts configurations. The logical view diagram in Figure 5.48 describes how this option functions.

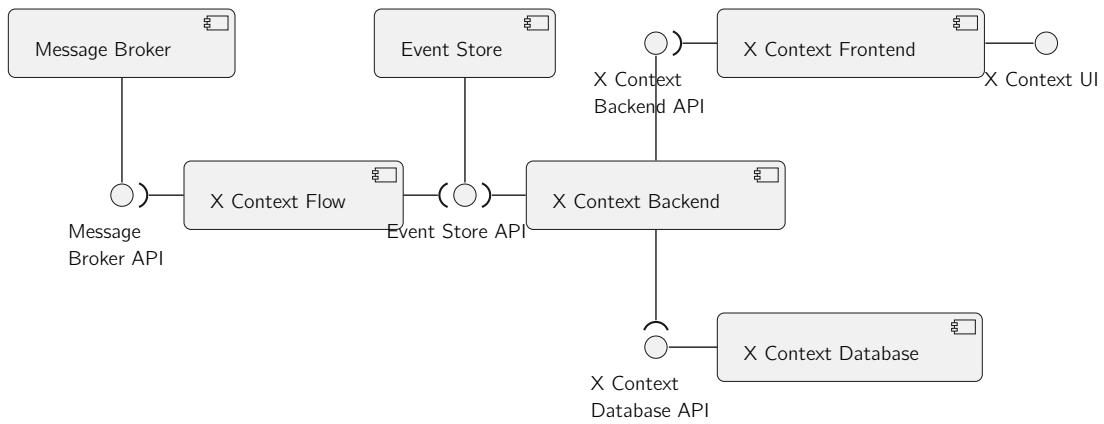


Figure 5.48: Internal Communication - Fifth Option - Logical View Diagram

The *X Context Flow* Container would use event sourcing to reach the current state of its context 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.

5.4 Domain

This solution's domain model will be discussed here. The idea behind this section is to introduce core business concepts to the reader and explain how they map to the contexts referred before. To represent this ideas the UML notation is used.

This section is split into four pieces: (i) concepts, (ii) shared model, (iii) sensae console bounded contexts and (iv) external services bounded contexts.

5.4.1 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 through **Uplinks** (commonly referred as a sensor). A device can also receive **Downlinks** and act base on what was received (commonly referred as an actuator);
- **Controller**: A controller is a **Device** that controls and aggregates data from various sub **Devices**;
- **Records/Metadata**: Records, or Metadata are labels associated to a **Device** that help an organization to classify and add some context to the owned **Devices**;
- **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 **Domains** and represents any of the roles discussed in Section 4.1;
- **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 **Topics** that flow in the system are:

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

This concepts are referenced across the document.

5.4.2 Shared Model

The shared 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 Assunção, 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.

5.4.2.1 Data Model

The data model represents the **Data Unit** that **Sensae Console** is currently capable of understanding. The following diagram, Figure 5.49, is a high level specification.

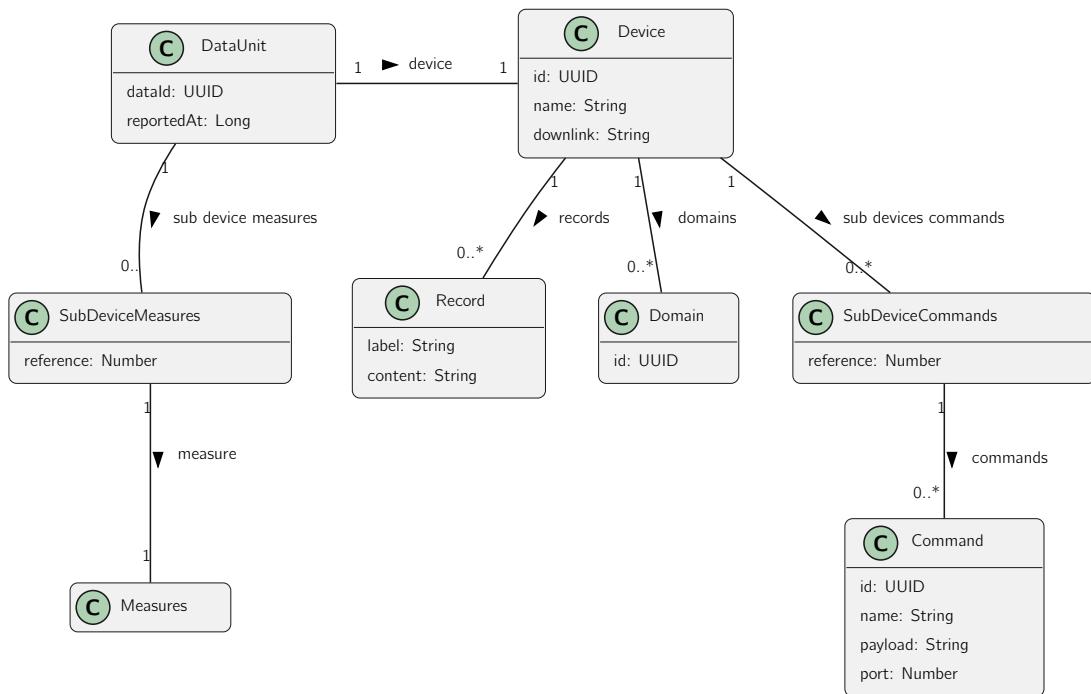


Figure 5.49: Shared Model

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* refers 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 refers to the device that sent the uplink;

- The *SubDeviceCommands* refers 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 refers 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 5.3. 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.

Table 5.3: Measure Data Types

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

Table 5.3 continued from previous page

Data Type	<i>Property</i>	<i>Sub Property</i>	Description	Unit
Illuminance	<i>illuminance</i>	<i>lux</i>	Illuminance level - luminous flux per unit area Value measured in	lux
CO2	<i>co2</i>	<i>ppm</i>	Atmospheric Carbon Dioxide concentration Value measured in	ppm
CO	<i>co</i>	<i>ppm</i>	Atmospheric Carbon Oxide concentration Value measured in	ppm
VOC	<i>voc</i>	<i>ppm</i>	Volatile Organic Compounds concentration measured by a device Value measured in	ppm
NH3	<i>nh3</i>	<i>ppm</i>	Atmospheric Ammonia concentration Value measured in	ppm
O3	<i>o3</i>	<i>ppm</i>	Atmospheric Ozone concentration measured by a device Value measured in	ppm
NO2	<i>no2</i>	<i>ppm</i>	Atmospheric Nitrogen dioxide concentration Value measured in	ppm
PM2.5	<i>pm2_5</i>	<i>microGramsPerCubicMeter</i>	Particulate Matter in the air (size up to 2.5 micrometers) Value measured in	$\mu\text{g}/\text{m}^3$
PM10	<i>pm10</i>	<i>microGramsPerCubicMeter</i>	Particulate Matter in the air (size up to 10 micrometers) Value measured in	$\mu\text{g}/\text{m}^3$
pH	<i>ph</i>	<i>value</i>	Scale used to specify how acidic or basic a water-based solution is Value between 0 and 14 measured in	pH

Table 5.3 continued from previous page

Data Type	Property	Sub Property	Description	Unit
Occupation	<i>occupation</i>	<i>percentage</i>	Occupation percentage measured inside a vessel Value measured in	%
Soil Conductivity	<i>soilConductivity</i>	<i>microSiemensPerCentimeter</i>	Substances ability to conduct an electrical current in the soil Value measured in	$\mu\text{S}/\text{cm}$
Air Humidity	<i>airHumidity</i>	<i>gramsPerCubicMeter</i> <i>relativePercentage</i>	Concentration of water vapour present in the air Value measured in Value measured in	g/m^3 %
GPS	<i>gps</i>	<i>latitude</i> <i>longitude</i> <i>altitude</i>	Point reference in the Geographic Coordinate System Value between -90 and 90 measured in Value between -180 and 180 measured in Value determined according to the mean sea level	degrees degrees meters
Battery	<i>battery</i>	<i>volts</i> <i>percentage</i> <i>maxVolts</i> <i>minVolts</i>	Battery of the device Value measured in Value measured in Minimum volts the battery needs for the device to work Maximum volts the battery can hold	volts % volts volts

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

5.4.2.2 Message Envelop Model

The message envelop model refers to how, coupled with the routing model in Section 5.4.2.3, information can reliably transverse the system.

The diagram present in Figure 5.50 details this model.

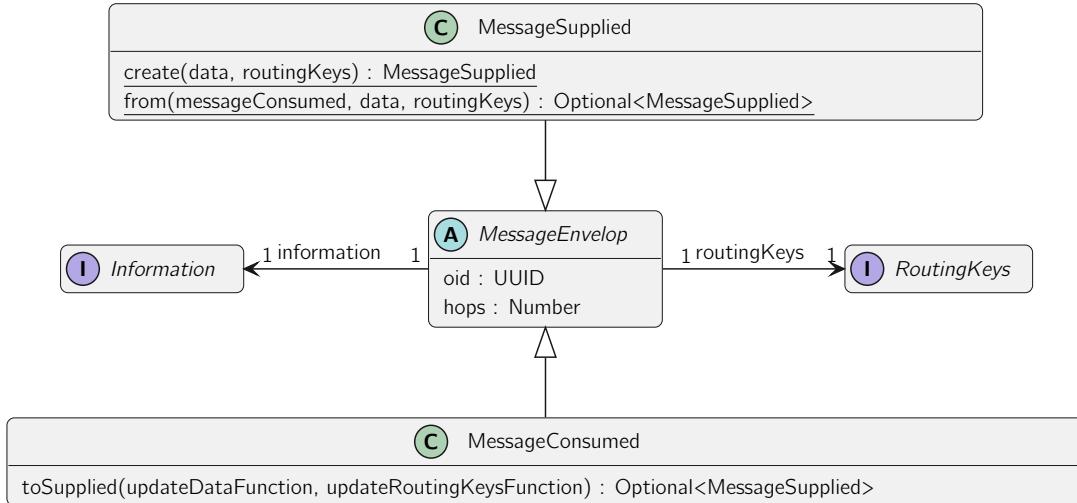


Figure 5.50: Message Envelop Model

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 5.4.2.3;

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.

5.4.2.3 Routing Model

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

The diagram present in Figure 5.51 details this model.

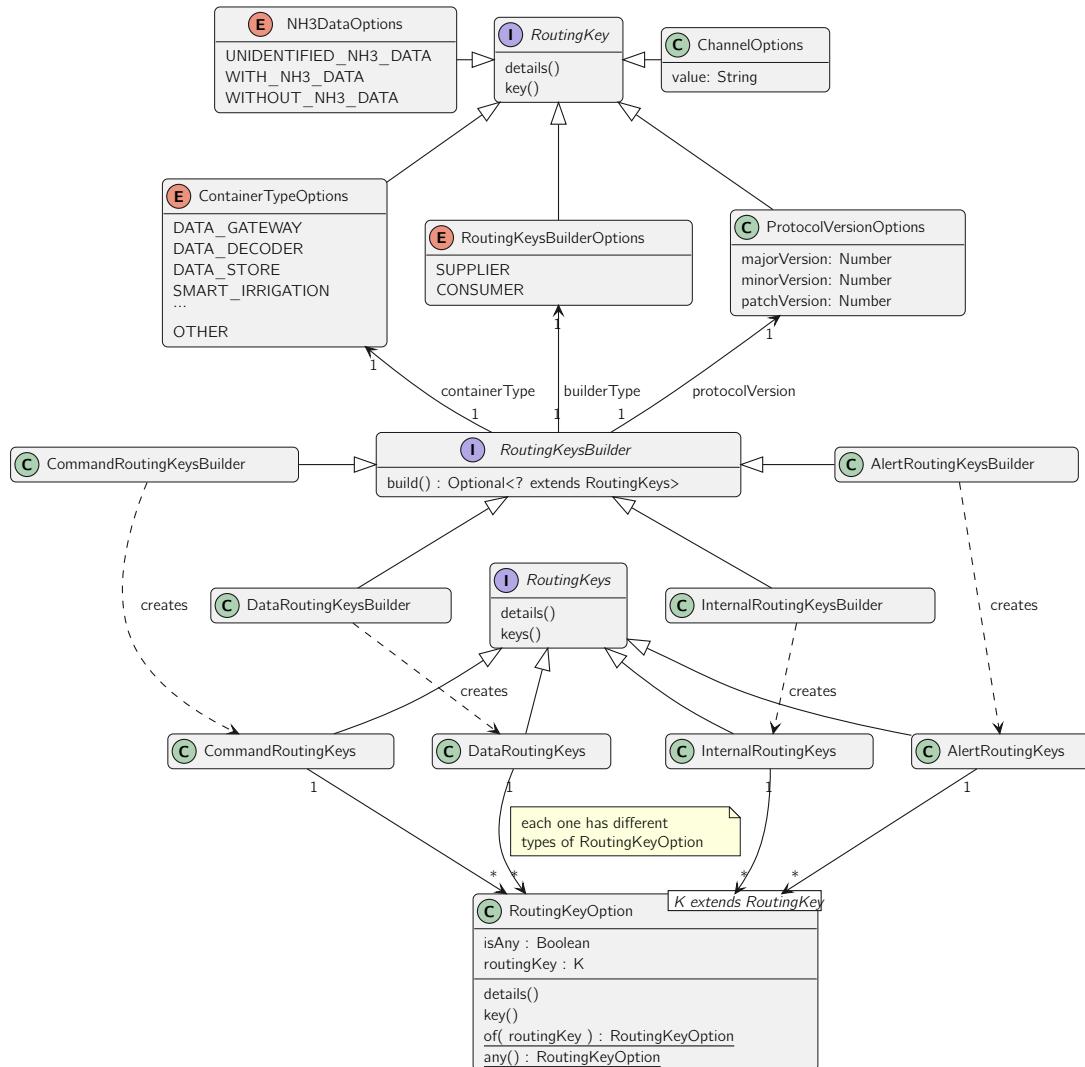


Figure 5.51: Routing Model

As a brief description:

- *RoutingKeys* is the concept referenced in Figure 5.50 and represents a collection of different *RoutingKeyOptions*;
- There are currently 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) and 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.
- The *RoutingKeysBuilder* implements the *Builder* pattern and its single responsibility is to validate and create *RoutingKeys*;
- 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;

- *NH3DataOptions* and *ChannelOptions* are two examples of *RoutingKey*, both used in the Data Topic.

In the Table 5.4 all currently used *RoutingKey* are presented.

Table 5.4: Routing Options

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

Table 5.4 continued from previous page

Topic	Routing Key	Description
<i>Occupation Data Options</i>		Does the data contains information about occupation levels ¹²
<i>pH Data Options</i>		Does the data contains information about ph level ¹²
<i>PM2.5 Data Options</i>		Does the data contains information about pm 2.5 concentration ¹²
<i>PM10 Data Options</i>		Does the data contains information about pm 10 concentration ¹²
<i>Soil Conductivity Data Options</i>		Does the data contains information about the soil conductivity ¹²
<i>Soil Moisture Data Options</i>		Does the data contains information about the soil moisture ¹²
<i>Temperature Data Options</i>		Does the data contains information about the temperature ¹²
<i>Trigger Data Options</i>		Does the data contains information about something that works as a switch ¹²
<i>Velocity Data Options</i>		Does the data contains information about the device velocity ¹²
<i>VOC Data Options</i>		Does the data contains information about VOC concentration ¹²
<i>Water Pressure Data Options</i>		Does the data contains information about water pressure ¹²
Command		Routing Keys that belong to the Command Topic
<i>Command Type Options</i>		Type of command, e.g. Open Valve
Alert		Routing Keys that belong to the Alert Topic
<i>Alert Category Options</i>		Category of the alert published, e.g. Fire Detention
<i>Alert Subcategory Options</i>		Category of the alert published, e.g. Humidity With High Rate Of Change
<i>Alert Severity Options</i>		Severity of the alert published, from Information level to Critical level

¹has three possible values: (i) UNDETERMINED, (ii) WITH, (iii) WITHOUT

²related to the explored Data Types

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, a Service in the **External Services Scope** 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 6.2.12.

5.4.3 Sensae Console Bounded Context

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 Context** referred 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 context 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 contexts will be briefly addressed in the following sections.

5.4.3.1 Data Processor

The **Data Processor** context refers to simple data mappers that translate inbound information to **Data Units**, discussed in Section 5.4.2.1.

The received information must be *decoded*, meaning that the inbound information simply has a different structure than **Data Unit**.

The diagram in Figure 5.52 displays the noteworthy concepts in this context.

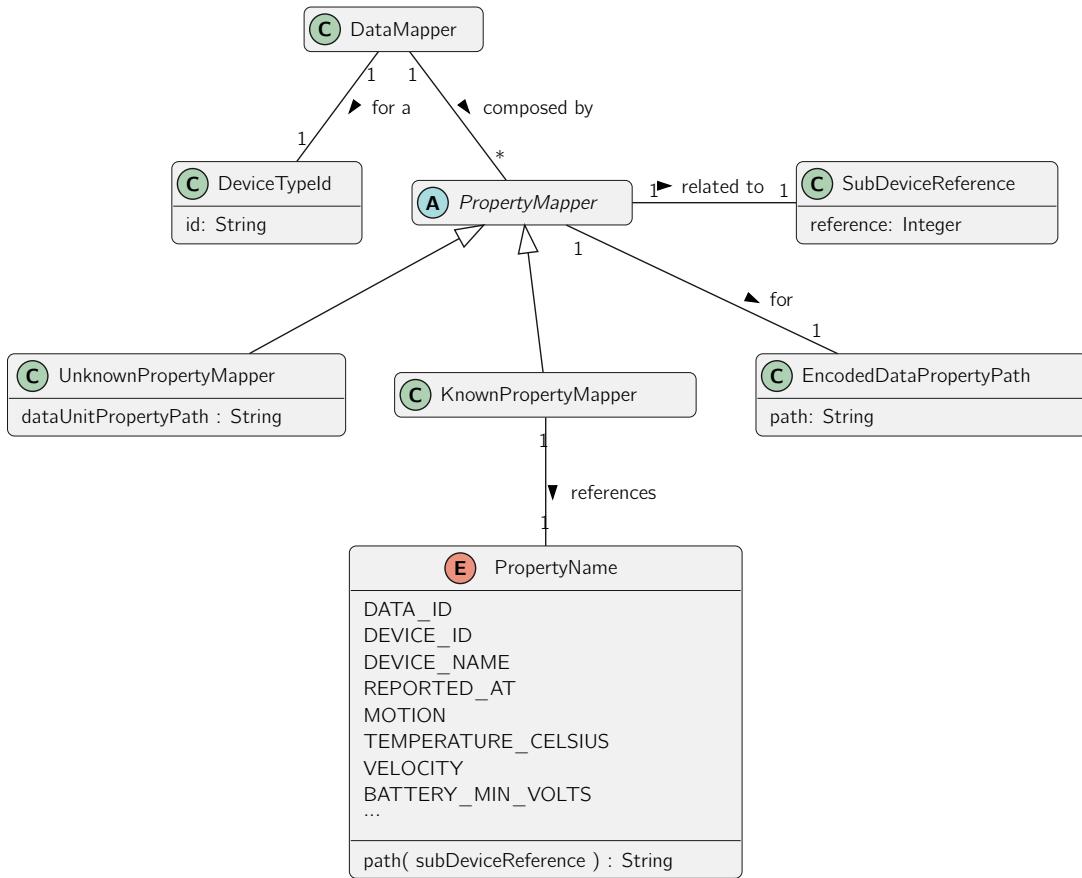


Figure 5.52: Data Processor Context Model

As a brief description:

- **DataMapper**, the root entity in this context 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 context the data mapper that has the **DeviceTypeid** equal to the message's *Device Type Options* routing key (mentioned in Table 5.4) is used to process that data unit;
- **SubDeviceReference** represents a number that will be used later to reference a sub device when dealing with **Controllers**. 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 5.1. To map the *temperature* value to the **TEMPERATURE_CELSIUS** property of a **Data Unit**, the **EncodedDataPropertyPath** would be *decoded.data[0].temperature*.

```

1 {
2   "uuid" : "de1a9d15-c018-4547-8453-87111cb4f81b",
3   "id" : "d81e6e69-1955-48a1-a1dd-4c812c15ebac",
4   "time" : 1657646955748,
  
```

```

5   "decoded": [
6     "data": [
7       {
8         "temperature": 18,
9       }
10    ]
11  }
12 }
```

Listing 5.1: 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.

5.4.3.2 Data Decoder

The **Data Decoder** context refers to a more complex data mapper that translates inbound information to **Data Units**, discussed in Section 5.4.2.1. It was created to deal with the limitations mentioned in Section 5.4.3.1.

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 5.53 displays the noteworthy concepts in this context.

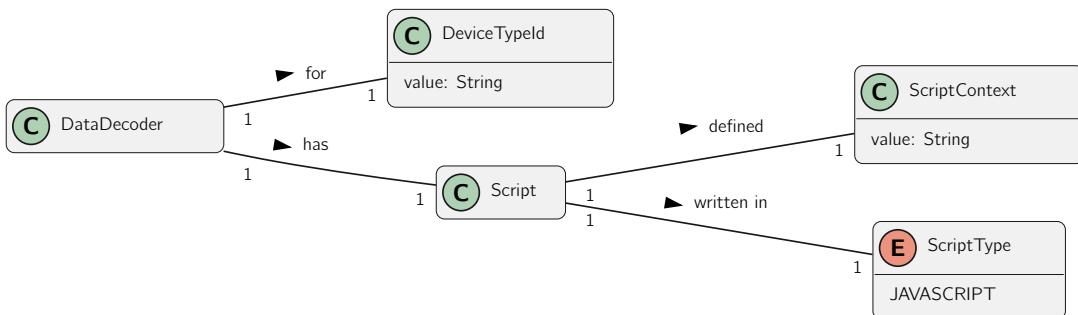


Figure 5.53: Data Decoder Context Model

As a brief description:

- **DataDecoder**, the root entity in this context 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.

5.4.3.3 Device Management

The **Device Management** context refers to the inventory of all registered **Devices** in the **Sensae Console**.

The diagram in Figure 5.54 displays the noteworthy concepts in this context.

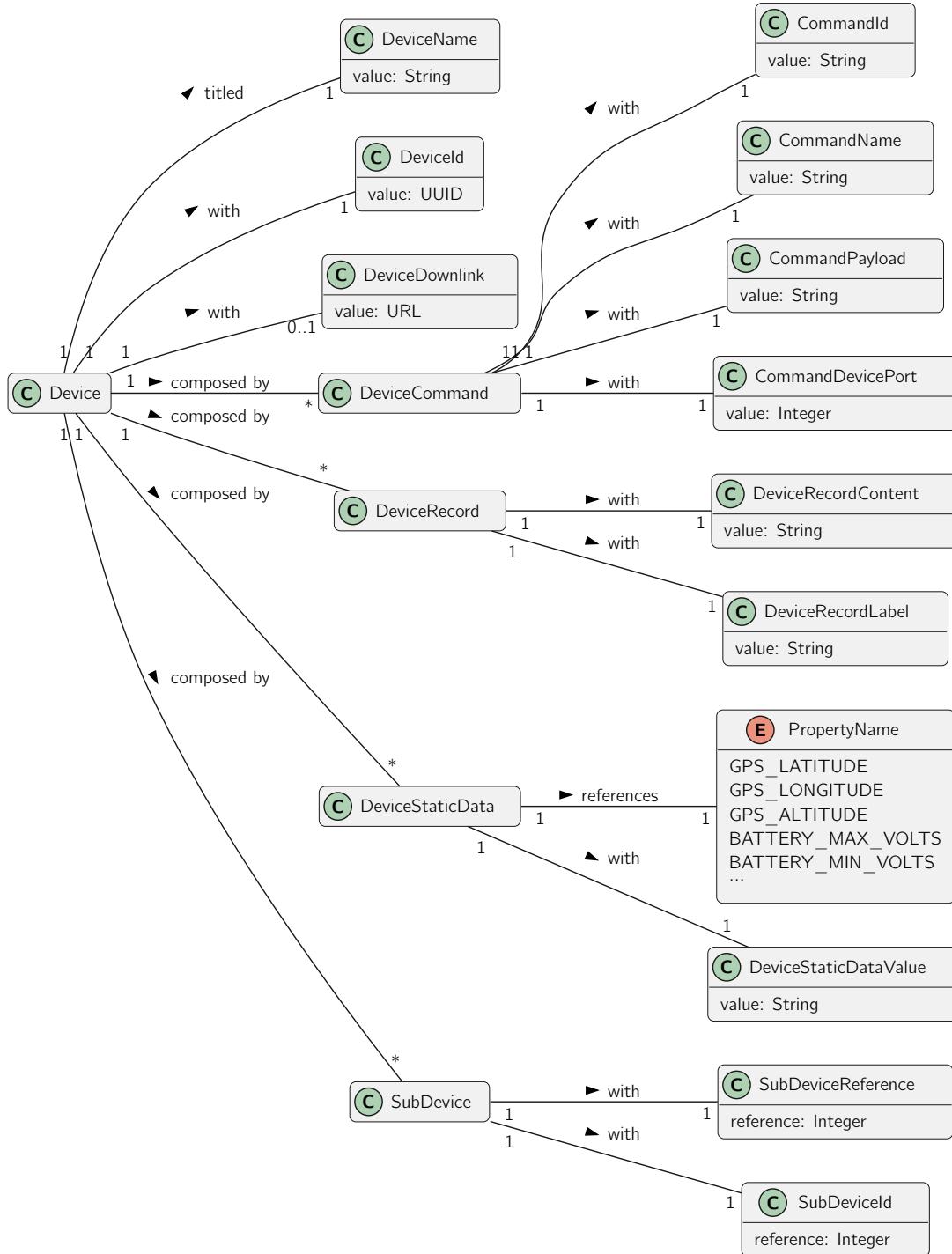


Figure 5.54: Device Management Context Model

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 5.49 help to split a **Controller's Data Unit** into various **Data Unit**, one for each referenced **SubDevice**.

5.4.3.4 Identity Management

The **Identity Management** is concerned with identifying **Tenants**, defining their permissions and what **Devices** they own. To simplify this, a forth concept is introduced: **Domain**.

The diagram in Figure 5.55 displays the noteworthy concepts in this context.

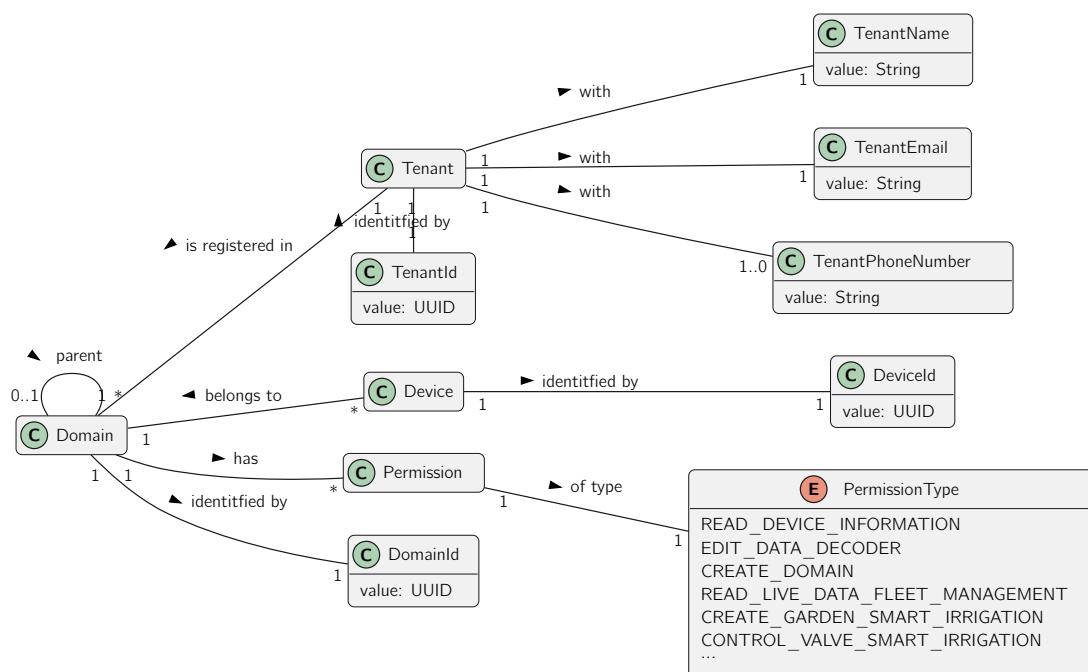


Figure 5.55: Identity Management Context Model

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 than 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 5.56.

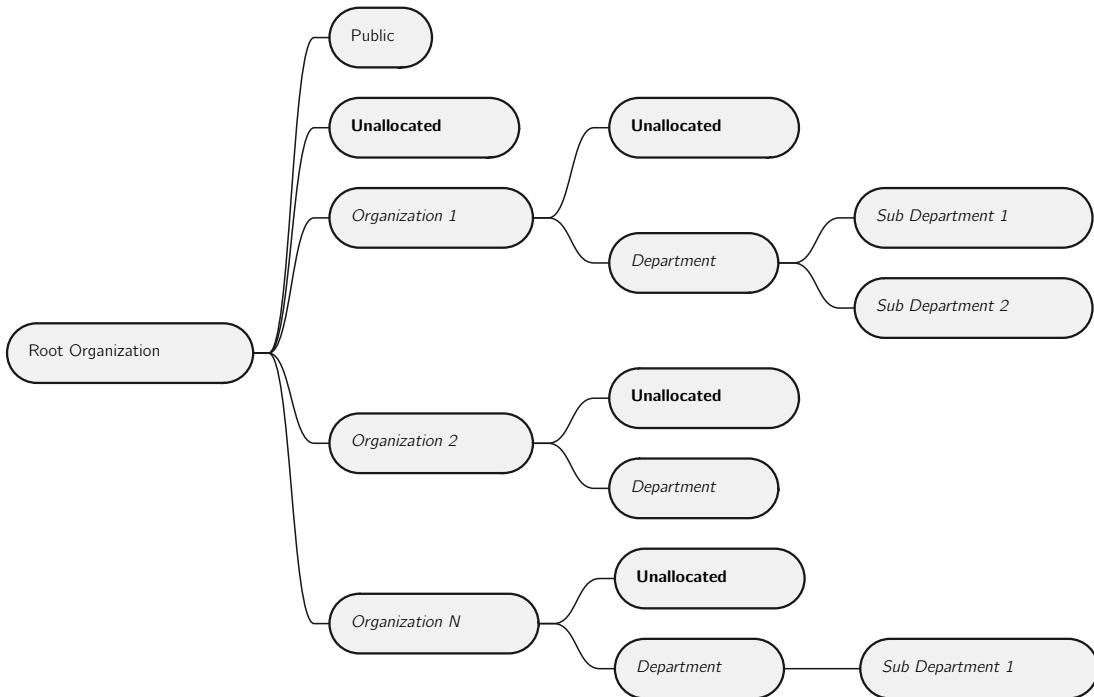


Figure 5.56: Domain Structure

Coupled with the figure above, there are other constraints:

- 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 context contains the *Root Organization* domain, the *Root Organization's Unallocated* domain and the *Public* domain.

Referring to the roles in Section 4.1.1, a Manager belongs to the *Root Organization*, any Costumer belongs to one or various *Organizations*, 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.

5.4.3.5 Rule Management

The **Rule Management** context refers to rule scenarios.

The purpose of this context is to provide a high-level language that can analyze a stream of **Data Units**, identify abnormal occurrences, and output **Alerts** base on them. This systems are usually categorized as Information Flow Processing (IFP) Systems, according to Cugola and Margara 2012.

The following diagram, Figure 5.57, represents this systems.

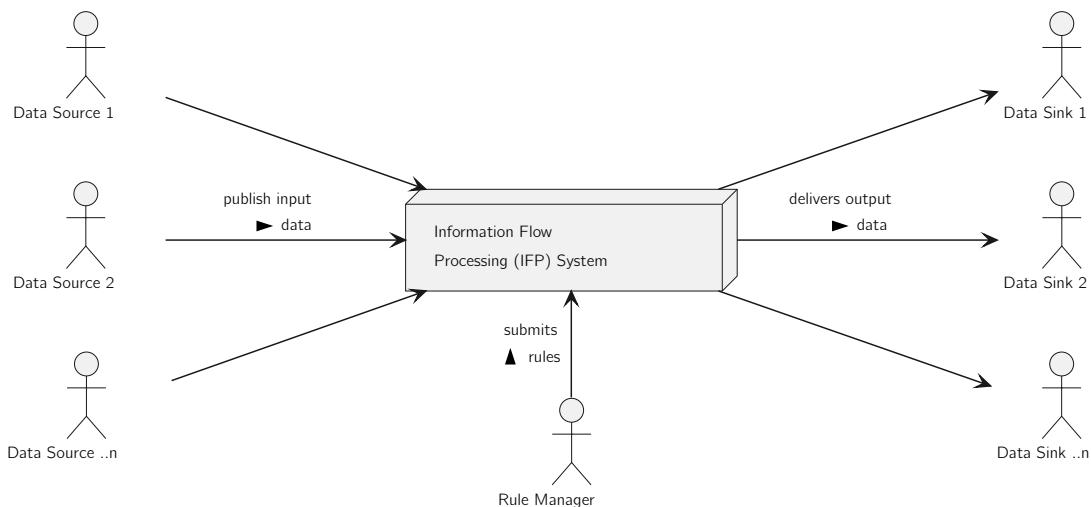


Figure 5.57: High-Level View of a IFP System

In this context, the input data are **Data Units** and the output data are the **Alerts**. This context is concerned about how *rules* are defined. The diagram in Figure 5.58 displays the noteworthy concepts.

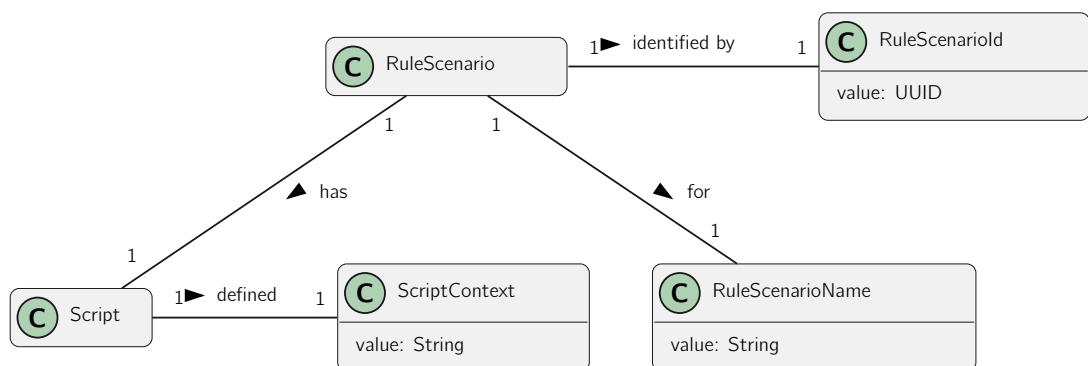


Figure 5.58: Rule Management Context Model

5.4.4 External Services Bounded Contexts

Once again, the term **Bounded Context** used here doesn't refer to the term coined by Evans 2014, but to a simple, concise and confined domain model.

Each Bounded Context is analogous to a specific external service that addresses a business case.

The developed **External Services**, and their bounded contexts are:

- Smart Irrigation;
- Fleet Management;
- Notification Management.

Each of these contexts will be briefly addressed in the following sections.

5.4.4.1 Notification Management

The **Notification Management** context refers to notifications and how/what types an addressee wants to receive. There are two main concepts in this context, a notification and an addressee.

The diagram in Figure 5.59 displays the noteworthy concepts in this context.

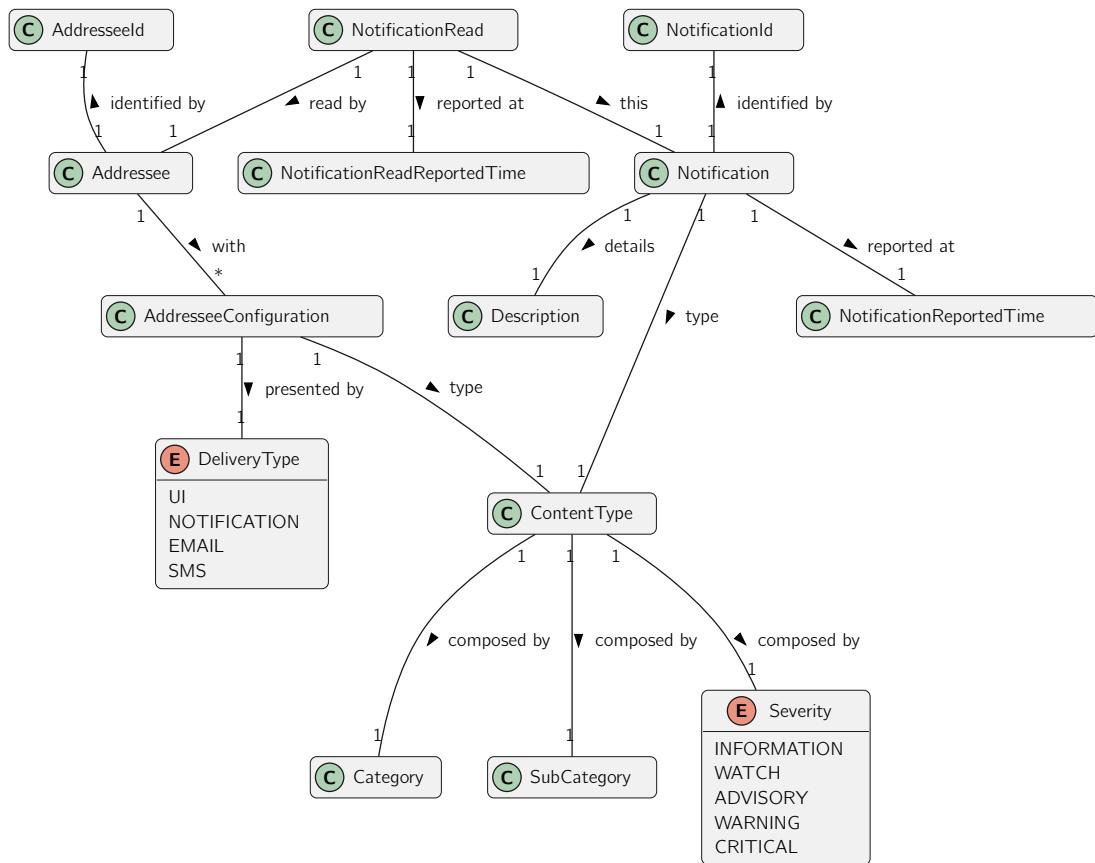


Figure 5.59: Notification Management Context Model

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 5.4 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**.

5.4.4.2 Smart Irrigation

The **Smart Irrigation** context 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 5.60 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.

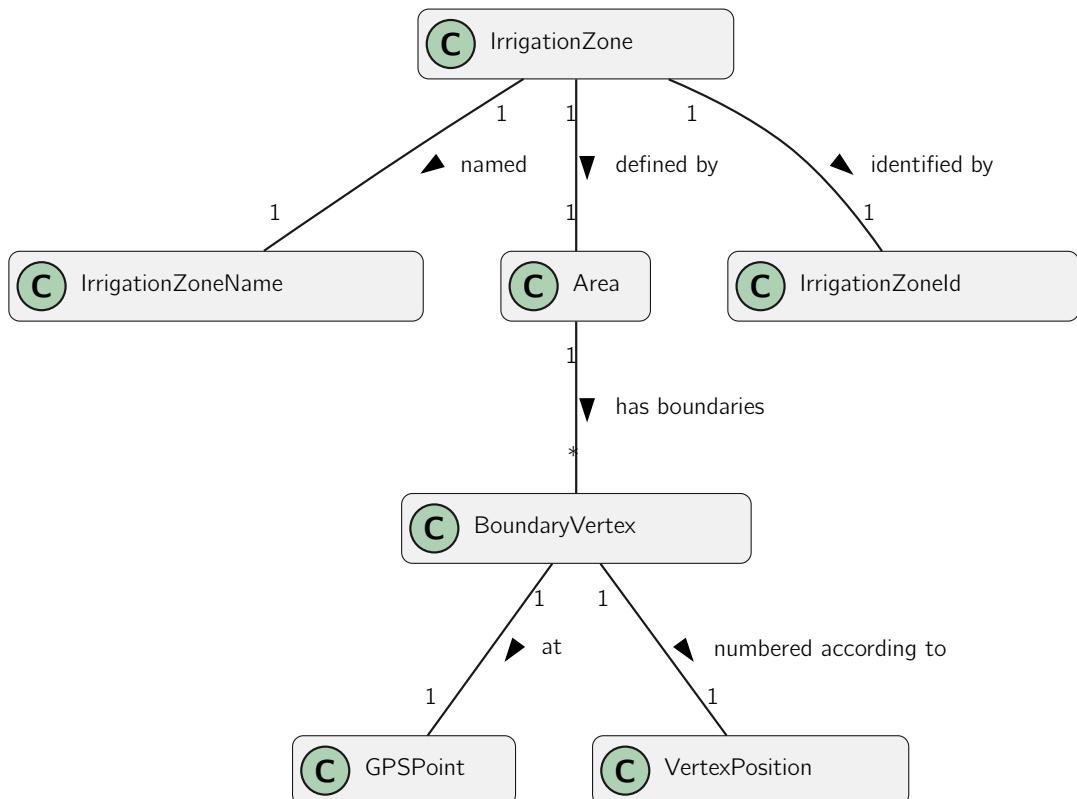


Figure 5.60: Smart Irrigation Context Model - Irrigation Zone

A sensor or valve belongs to an irrigation zone if it is inside the zone's **Area**.

As presented in the following diagram, Figure 5.61, a sensor/valve can be represented by a **Device**.

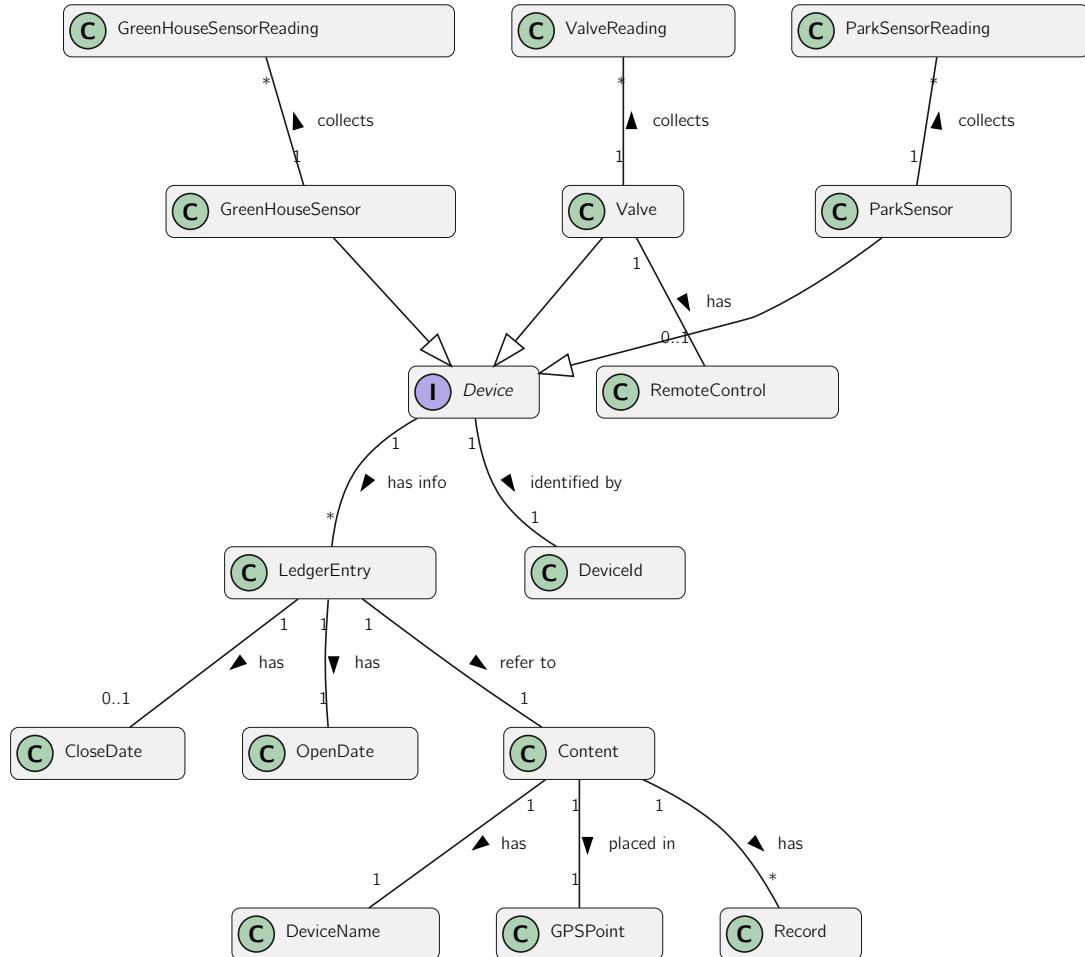


Figure 5.61: Smart Irrigation Context Model - Device

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 **Commands** (as defined in the Shared Model) are sent with the device's **Data Unit**: *OpenValve* and *CloseValve*;
- A **Device** is identified by its **Deviceld**;
- 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 these values don't change;
- There are three types of **Device**: (i) Green House Sensor, (ii) Park Sensor, (iii) Valve. Each of these types collect different measures discussed in Figure 5.62.

As mentioned above each type of device collects different readings. The following diagram, Figure 5.62, details these readings.

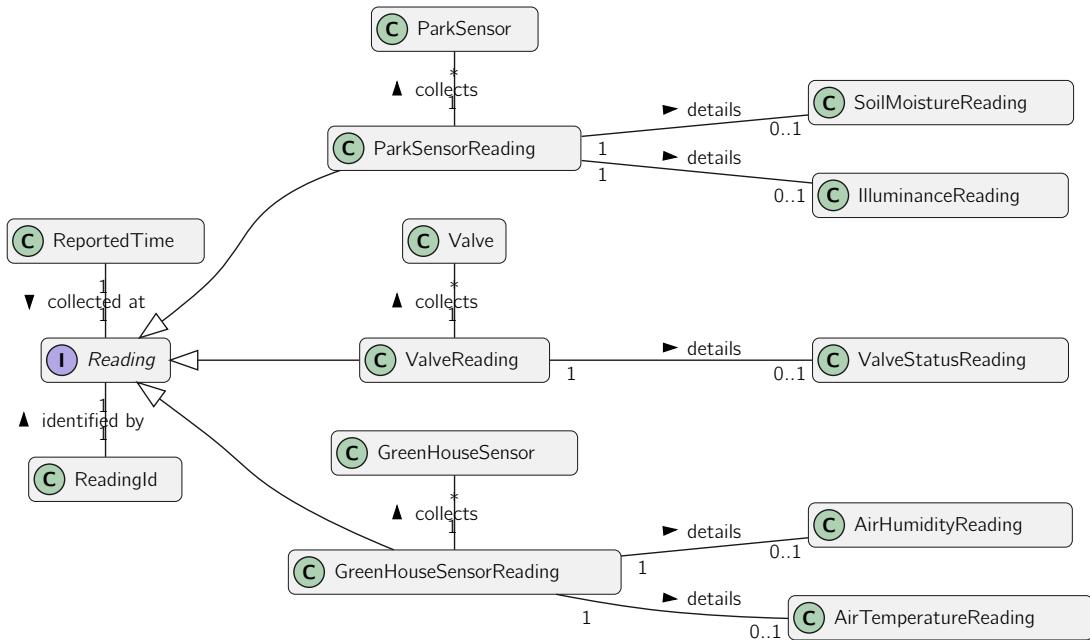


Figure 5.62: Smart Irrigation Context Model - Reading

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.

5.4.4.3 Fleet Management

The **Fleet Management** context simply refers to the past and current location of assets.

The diagram in Figure 5.63 displays the noteworthy concepts related to this context.

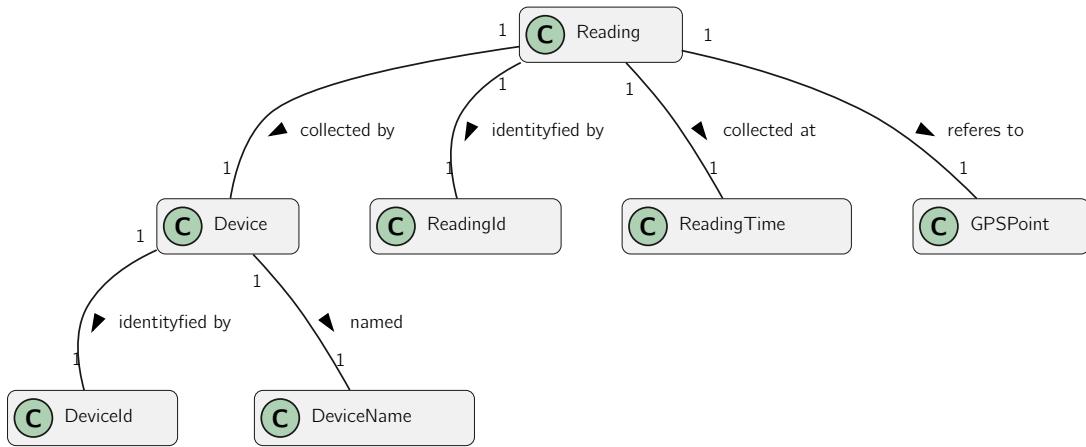


Figure 5.63: Fleet Management Context Model

This was the first External Service built as a PoC, it was intended to be straightforward. The model references Global Positioning System (GPS) readings and what device collected them.

5.5 Synopsis

This chapter presented the design of **Sensae Console**, 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 solution was implemented.

Chapter 6

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.

6.1 Technical Decisions

This section describes and justifies the decisions taken while developing **Sensae Console**. As a green field 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;
- Containerization of services via Docker;
- Orchestration of services via Docker Compose;
- Usage of Nginx as a web server and reverse proxy;
- Usage of Git as a version control system of the project;
- Usage of Github Issues to track issues, bugs and new features;
- Usage of Github Actions for CI/CD;
- Usage of Maven Repository to host Open-Source Code;

6.1.1 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.

6.1.1.1 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. 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*.

6.1.1.2 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].

6.1.1.3 Data Flow Scope Backend Services

As discussed in Section 5.1.2, the services that this section encompasses can be seen as more lightweight than the ones described above due to their associated requirements.

Since this 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*.

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*.

6.1.2 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.

6.1.2.1 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 6.1, describes the reason that lead the author to choose *Angular* over *React*.

Table 6.1: 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 6.1.

6.1.2.2 Technologies used to create a Micro Frontend Architecture

Module Federation was the tool used to seemly connect the various Frontend. 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.

6.1.2.3 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 before at Figure 5.37 resembles this two concepts.

6.1.2.4 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.

6.1.3 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 five 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.

6.1.4 Usage of RabbitMQ to support Internal Communication

As discussed in Section 5.3.4 and 5.3.5, 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.

As discussed in the article, *AMQP 0-9-1 Model Explained*, the Advanced Messaging Queue Protocol (AMQP) 0.9.1 protocol defines four main concepts: (i) publisher, (ii) exchange, (iii) queue, (iv) consumer. The following diagram, Figure 6.1 explains how this concepts interact.

"Hello, world" example routing

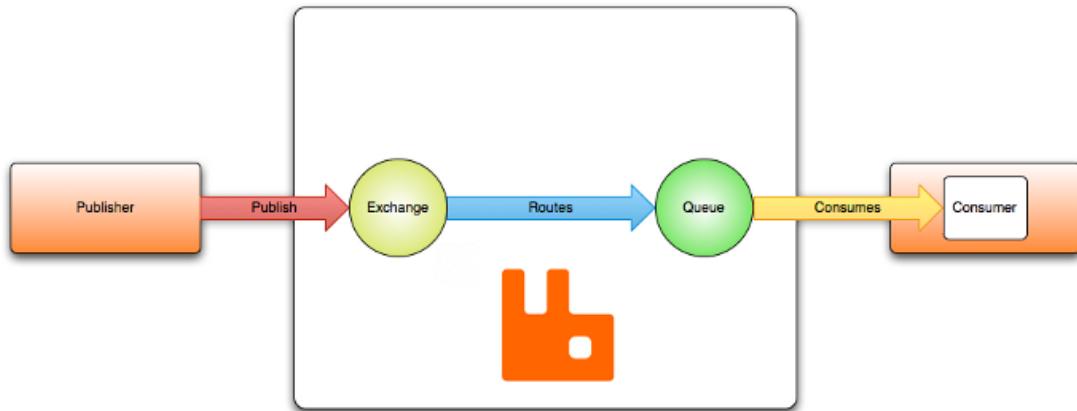


Figure 6.1: AMQP 0.9.1 Protocol Concepts by VMWare 2022a

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;

The exchange that better fits the defined requirements is the Topic Exchange.

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 *- this makes it complex to create routing keys with dynamic values. As an example, lets look at the *Channel* routing key defined in Table 5.4 of Section 5.4.2.3. 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 *channels* 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 described above by allowing Consumers to subscribe to multiple topics (equivalent to RabbitMQ's 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 5.4 of Section 5.4.2.3, 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 5.4 has three possible values: Encoded, Decoded and Processed, these values are respectively represented in the system as *e*, *d* and *p*.

6.1.5 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). These 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 customers 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 these 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 2022d].

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.

6.1.6 Database Usage throughout the Solution

This section refers to how information is stored across the system.

A 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 some of the data model types, according to 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 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.

6.1.6.1 Relational Database Usage

This data model has a wide variety of usage in the industry. Some of the technologies that follow this data model are: (i) *MySQL*, (ii) *PostgresSQL* and (iii) *MariaDB*.

It is intended for strictly structured data with well defined interrelations. This type of data can be found on most Bounded Contexts described in Section 5.4.3 such as Data Processor, Data Decoder, Device Management, Identity Management, Rule Management and the Irrigation Zone/Device concepts of the Smart Irrigation Context.

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 author had previous contact with all the cited DBMS, and 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.

6.1.6.2 Document-based Database Usage

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 type of requirements and data resembles the Data Store context described in Section 5.1.2 and Figures 5.14 and 5.18. This context, intended to mimic a Data Lake¹, stores any type of data for future use.

As such, this data model was adopted for the **Data Store Database** container.

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

The only technology considered, and therefore adopted, was *MongoDB* due to its vast community, excellent documentation 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.

6.1.6.3 Column-based Database Usage

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 Han et al. 2011, these databases “can maintain high-performance of data analysis and business intelligence processing”.

These features fit the requirements related to storing and reading vast amounts of device measures. As such, it was adopter for the **Fleet Management Database** and **Smart Irrigation Data Database** containers.

The author had no previous contact with this type of data model. 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 Structured Query Language (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 Java Database Connectivity (JDBC)), can be easily deployed using a single Docker Image, and also supports the *InfluxDB* Line Protocol (ILP).

The type of business this solution is tackling revolves around the capture and analysis of device readings, IoT. 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”.

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 JDBC. During the research of this two technologies no major downside was found for *QuestDB* when compared to *InfluxDB*.

6.1.6.4 Graph-based Database Usage

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

As stated in the bounded context’s section of Identity Management, the domains follow a hierarchical structure that can resemble a graph. This context 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.

6.1.7 Rules Script Engine

This section refers to the bounded context of **Rule Management**. As mentioned before, the purpose of this context 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 an open-source rule engine widely used in the industry [RedHat n.d.]. The features that stud out from other rule engines are:

- Supports for sliding windows of time;
- It is also a Complex Event Processing (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 6.2.5 details how one can write rule scenarios.

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

6.1.8 Data Decoders Script Engine

This section refers to the bounded context of **Data Decoder**. As mentioned before, this context purpose 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 Milesight³, SensationalSystems⁴ and Helium,⁵. This makes it easy and straightforward to integrate new decoders in **Sensae Console**.

The Section 6.2.6 details how one can write decoders.

6.1.9 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 Operating System (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 5.2.2 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 all the processes inherent to an OS as described by Bernstein 2014.

The Figure 6.2 compares a Virtual Machine (VM) and Container-based deployments.

³github.com/Milesight-IoT/SensorDecoders

⁴github.com/SensationalSystems

⁵github.com/helium/console-decoders

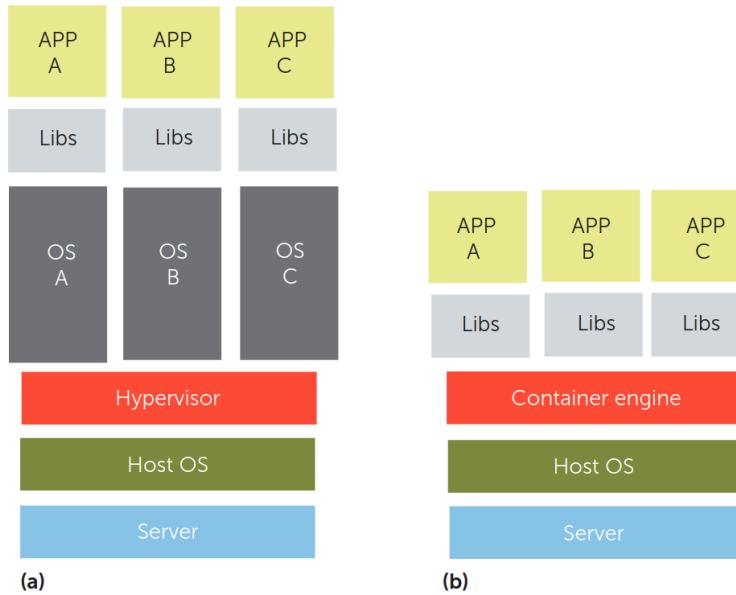


Figure 6.2: Comparison of VM (a) and Container-based (b) deployments by Bernstein 2014

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 6.2.8.

6.1.10 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 6.2.9.

6.1.11 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 6.2, describes this criteria.

Table 6.2: 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 6.2.10.

6.1.12 Usage of Git as a version control system of the project

Git is a Version Control System (VCS). What differentiates it from other systems such as *Mercurial* and *Bitkeeper* is its branching model. It is currently also the most widely used.

Github was the platform used to host the developed code. It offers private repositories with no additional costs. This platform also has other tools such as *Github Issues* and *Github Actions* that ease a developer's workflow.

A VCS is indispensable in software development, this system allows developers to store the history of changes made to the code in an organized manner and simplifies the management of the software by the development team. This system was chosen over others because the author was experienced with this software.

The development of the entire solution was made in two separated repositories, one for *iot-core* and another for **Sensae Console**.

The *iot-core* repository had a simple branching model consisting only of a master branch.

There was an extensive use of the branching feature in the repository of **Sensae Console**, following the model shown in Figure 6.3. The author settled for the following: a master branch that matches the deployed version, a development branch where the various features are introduced until a new version is published on the master branch, several branches dedicated to fixing bugs (hotfix) and another several branches that introduce new features and improvements (feature x).

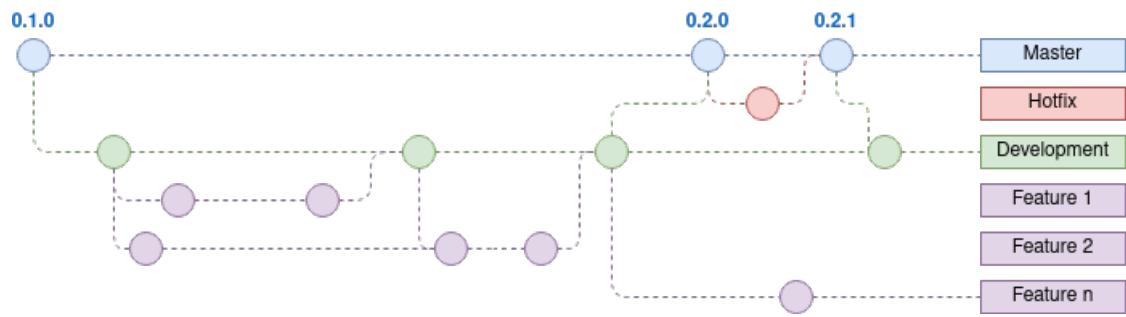


Figure 6.3: Branching Model

This model was adopted since the project was in an initial phase of development, in the future, a branching model with multiple releases, as detailed in Figure 6.4, is preferred. With this model one can release only the altered containers and not the entire system.

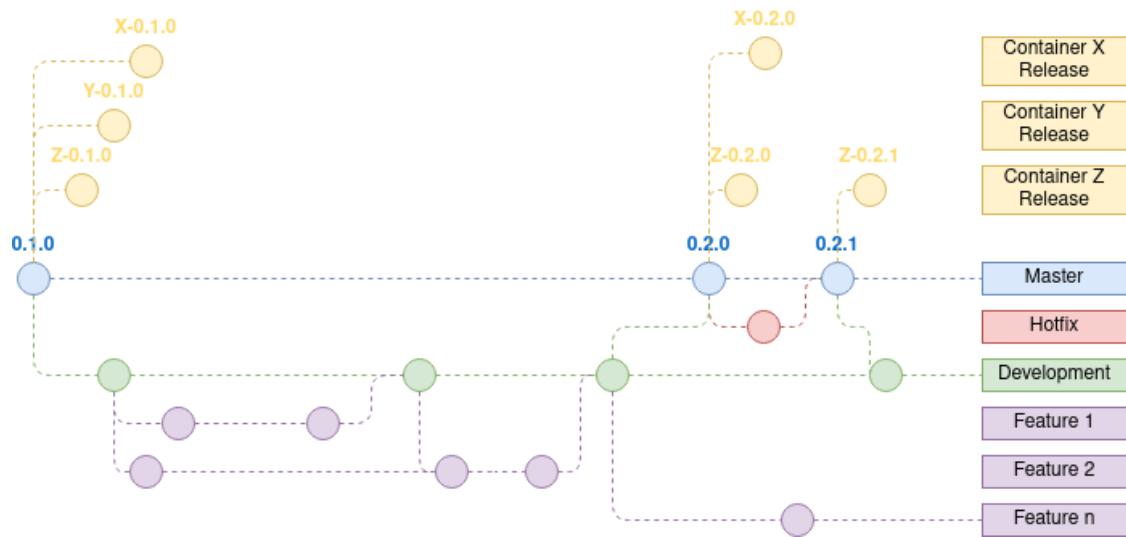


Figure 6.4: Future Branching Model

This is useful when using CI/CD pipelines to compile, package and deploy the various containers of the solution. If no changes have been made to X Container there is no need to redo all the work previously done with it.

The reason behind the monorepo approach for **Sensae Console** is that it allows frontend libraries to be shared without publishing somewhere. It is also much easier to keep track of the code in a monorepo since the solution is developed by a single developer.

6.1.13 Usage of Github Issues to track issues, bugs and new features

As described before, the code is hosted in *Github*. One of the services that this platform offers is *Github Issues*. This tool helps to track and document the development process alongside with the code.

This tool can be separated into two main views. A view is concerned about what issues, features and bugs are active in the project, Figure 6.5, and the other is concerned with the current state of each issue, feature and bug, Figure 6.6.

	Author	Label	Projects	Milestones	Assignee	Sort
10 Open ✓ 67 Closed						
<input type="checkbox"/> ● Upgrade all project dependencies ahead of 0.10.0 version release enhancement						
#140 opened on 28 Jun by FilipeMCruz ↗ 0.10.0						
<input type="checkbox"/> ● Update docs for upcoming version release 0.10.0 enhancement						
#139 opened on 28 Jun by FilipeMCruz ↗ 0.10.0						
<input type="checkbox"/> ● Downlinks are not working bug container: backend				⬆️ 1		
#86 opened on 10 May by FilipeMCruz ↗ 0.10.0						
<input type="checkbox"/> ● Ensure smaller screens, e.g. 11", are supported container: frontend enhancement				⬆️ 1		
#78 opened on 9 May by FilipeMCruz ↗ 1.XX						
<input type="checkbox"/> ● Microfrontends aren't creating their own apollo client and depend on ui-aggregator container: frontend enhancement						
service: fleet tool: decoder tool: device tool: iam tool: transformation						
#38 opened on 25 Mar by FilipeMCruz ↗ 1.XX						
<input type="checkbox"/> ● Metrics breakthrough						
#20 opened on 6 Feb by MeijeSibbel ↗ 8 tasks ↗ 1.XX						
<input type="checkbox"/> ● Fleet management UI enhancement service: fleet						
#19 opened on 6 Feb by MeijeSibbel ↗ 1.XX						
<input type="checkbox"/> ● Sensor Provisioning Tool Discussion						
#18 opened on 6 Feb by MeijeSibbel						
<input type="checkbox"/> ● Reports breakthrough						
#16 opened on 4 Feb by MeijeSibbel ↗ 1.XX						
<input type="checkbox"/> ● Sensors Discussion						
#14 opened on 22 Jan by MeijeSibbel						

Figure 6.5: Github Issues

Each issue has a list of tags that represent its scope and a defined milestone. With this tool, the team members can also discuss issues in depth.

The issues presented in this page are then tracked in the *project* page - Figure 6.6. The author decided to divided the issues into 4 criteria:

- **To Do:** Issues that have been discussed and are to be completed in the near future;
- **In Progress:** Issues that are currently under development and have an assigned feature branches;
- **Done:** Issues that have been completed and have been integrated in the *master* branch;
- **Future:** Issues that have been purposed but have no clear deadline.

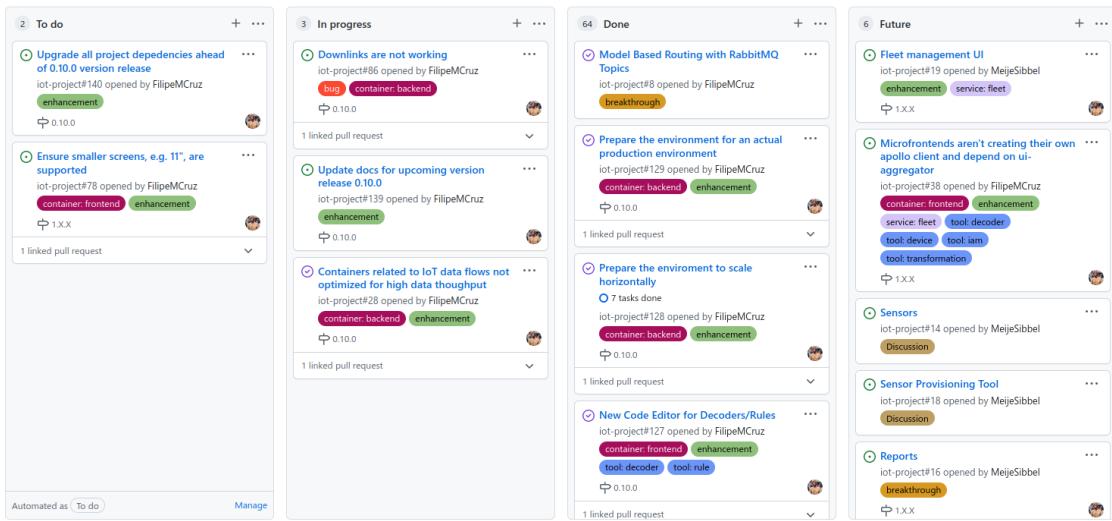


Figure 6.6: Github Issues Project Board

This view helps to define a simple project roadmap and track the overall state of issues, bugs and features in the project.

6.1.14 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 6.1.

```

1 name: IoT Core – Continuous Delivery to maven central
2 on:
3   push:
4     tags:
5       - '.*'
6       - '*'
7 jobs:
8   build:
9     runs-on: ubuntu-latest
10    steps:
11      - uses: actions/checkout@v2

```

```

12   - name: Set up Maven Central Repository
13     uses: actions/setup-java@v1
14     with:
15       java-version: 17
16       server-id: ossrh
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 6.1: 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 6.2, 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 }}"
24          - ${{ secrets.microsoft_audience }} "${{ secrets.google_audience }}"
25          - $"${{ secrets.admin_email }}"
```

Listing 6.2: 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, this tests will be discussed in the Testing Section.

The mentioned script has the following structure - Code Sample 6.3.

```

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 -l % 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 6.3: 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 6.2,
- 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 6.1.2.3 Section;
- Build and start an environment similar to the production one - lines **28** to **32**;
- Perform 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 6.1.2.3 Section;
- Shutdown the test environment.

6.1.15 Usage of Maven Repository to host Open-Source Code

As stated in the previous section *iot-core* is delivered to an artifact repository. Since the intent of this package is to be used by any one interested on integrating his/her tool with **Sensae Console**, the artifact repository has to be publicly available.

The Maven Central repository was the chosen one, since the *maven* and *gradle* tools use it, by default, to fetch dependencies.

According to the article *Why Do We Have Requirements?* by Sonatype 2022, to publish an artifact to maven central, a couple of additions have to be made in the *pom.xml* of the project namely: (i) Supply Javadoc and Sources, (ii) Provide Files Checksums, (iii) Sign Files with GPG/PGP, (iv) Sufficient Metadata, (v) Correct Coordinates, (vi) Project Name, Description and URL, (vii) License Information, (viii) Developer Information, (viii) SCM Information.

In the Appendix E Appendix the full *pom.xml* is presented.

6.2 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;
- Sensae Console Database Configuration;
- Sensae Console Containerization;
- Sensae Console Orchestration;
- Sensae Console Reverse Proxy Configuration;

- Sensae Console Configuration Files;
- External Services;
- Sensae Console Device Integration;

6.2.1 Sensae Console UI

In this subsection the UI is presented.

The Figure 6.7 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.

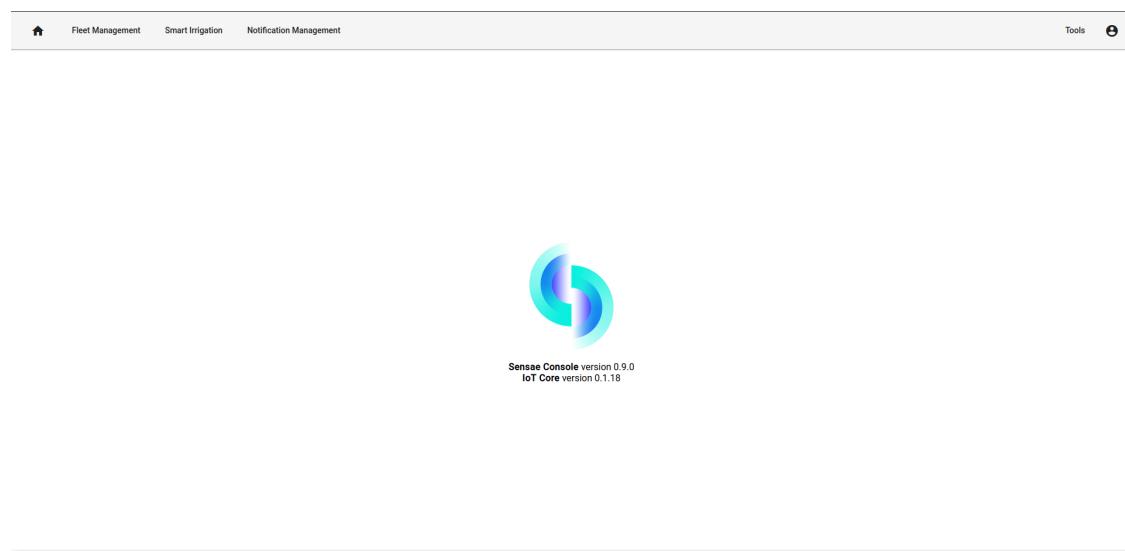


Figure 6.7: Sensae Console Home Page

From this page, if the user has sufficient permissions, one can access services pages, as an example the **Smart Irrigation Page** is displayed in Figure 6.8. This page presents a map where the user can see, search and create irrigation zones. Device measures are updated in real time via *Websockets*. The user can also see the irrigation zone details after clicking on it. From there it's possible to open/close valves and see the history of measures of each device.

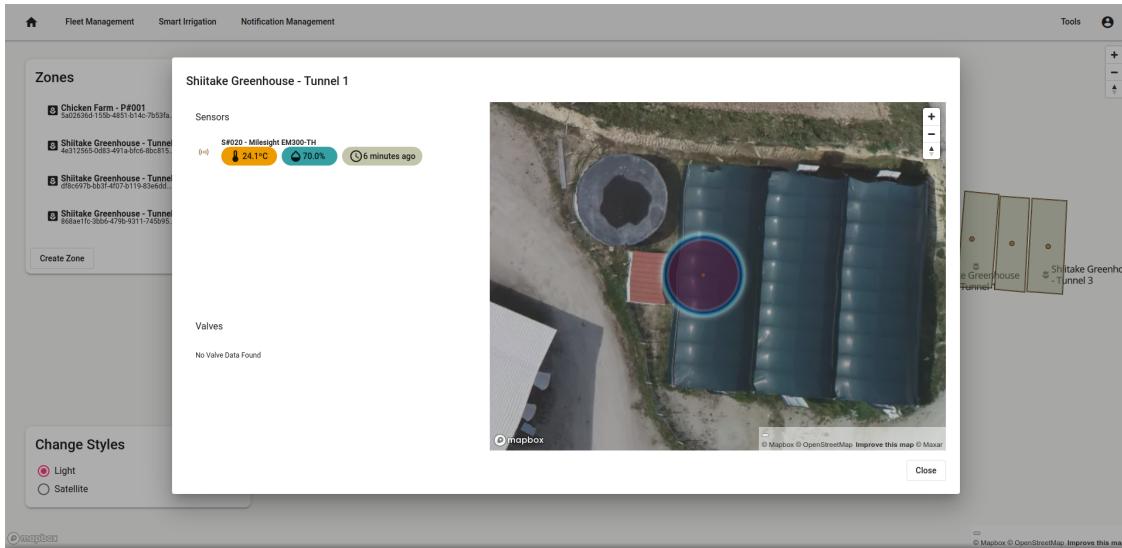


Figure 6.8: Sensae Console Smart Irrigation Page

From the home page the user can also access configuration pages, as an example the **Device Management Page** is displayed in Figure 6.9.

Figure 6.9: 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 Section Device Management of the *Bounded Contexts*.

Other relevant pages are presented in the Appendix D.

6.2.2 Sensae Console Custom Maps

This section describes how custom maps were built to fit the solution needs. Some customers 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 were 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 6.10 presents the new map with the customer 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 customer. This map is currently in use by the Smart Irrigation Service.



Figure 6.10: Custom Map - Smart Irrigation

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

6.2.3 Sensae Console Backend API

The **Sensae Console** API is served as a *GraphQL* API, one for each service/configuration context. This API is described with a schema.

As an example the Smart Irrigation API is presented in the Code Sample 6.4.

```

1 type Subscription {
2     data(filters: LiveDataFilter, Authorization: String) : SensorData
3 }
4
5 type Query {
6     history(filters: HistoryQueryFilters) : [SensorDataHistory]
7     fetchIrrigationZones : [IrrigationZone]
8     fetchLatestData(filters: LatestDataQueryFilters): [SensorData]
9 }
10
11 type Mutation {
12     createIrrigationZone(instructions: CreateIrrigationZoneCommand) :
13         IrrigationZone
14     updateIrrigationZone(instructions: UpdateIrrigationZoneCommand) :
15         IrrigationZone
  
```

```

14     deleteIrrigationZone(instructions: DeleteIrrigationZoneCommand) : 
15     IrrigationZone
16     switchValve(instructions: ValvesToSwitch): Boolean
}

```

Listing 6.4: 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 *textitdata* 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 4.1.3.2.

6.2.4 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 the Table 5.4. 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 5.14.

The *channel* parameter indicates the final service that it is destined to: *fleet* for Fleet Management Service or *irrigation* for Smart Irrigation Service. If another value is given the message is not routed to any 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 6.2.11.

6.2.5 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 Context, it provides a high-level language that can be used to detect anomalies in **Data Units** and turn them into **Alerts**.

Valid **Data Units** are captured by **Alert dispatcher Backend** and the 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 6.5, 6.6 and 6.7 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
3 import pt.sharespot.iot.core.data.model.data.DataUnitReadingsDTO;
4 import pt.sharespot.iot.core.data.model.DataUnitDTO;
5 import pt.sharespot.iot.core.data.model.device.records.
6     ↪ DeviceRecordEntryDTO;
7 import pt.sharespot.iot.core.model.properties.PropertyName;
8 import pt.sharespot.iot.core.alert.model.AlertBuilder;
9 import pt.sharespot.iot.core.alert.model.CorrelationDataBuilder;
10 import pt.sharespot.iot.core.alert.model.AlertLevel;
11 import java.util.List;
12 import java.util.UUID;
13
14 global pt.sharespot.iot.core.alert.model.AlertDispatcherService
15     ↪ dispatcher;
16
17 dialect "mvel"
18
19 declare StoveSensor
20     @role( event )
21     deviceld : UUID
22 end
23
24 declare StoveSensorData
25     @role( event )
26     deviceld : UUID
27     dataId : UUID
28     temperature : Float
29     humidity : Float
30 end

```

Listing 6.5: Rule Scenario Example - Part 1

As we can see, from line **3** to **9**, classes from *iot-core* are imported into the scenario. At line **13** the interface that defines how an alert can be sent is imported for later use. From line **17** to **28** 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
19     ↵ .relativePercentage);
20     insert(reading)
21 end

```

Listing 6.6: 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 "+
14        ↵ $s.deviceId + " detected low humidity/high temperature")
15      .setLevel(AlertLevel.ADVISORY)
16      .setContext(CorrelationDataBuilder.create())

```

```

16         .setDeviceIds($s.deviceId)
17         .setOther("Project #002")
18         .build())
19     .build());
20 end

```

Listing 6.7: 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 °C for more than 11 minutes.

Once it matches an Alert is dispatched using the referenced dispatcher in Code Sample 6.5. The Alert can be created using the builder pattern.

An Alert closely resembles a Notification from the Notification Management Bounded Context. It also has a category (line 13), a sub category (line 14), a severity level (line 16), a description (line 15) and a notification context (lines 17 to 20).

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 stored as a Notification, the *DeviceIds* associated to it are used to determine what domains will have access to the Notification. If no *DeviceIds* are associated only the root domain will have access to it.

6.2.6 Sensae Console Data Decoders

As mentioned in the Data Decoder Bounded Context Section, **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 6.8 presents the decoder for the device type EM500-TH⁶.

```

1 const decodePayload = (payload, port) =>
2   ({ "0": decoder(base64ToHex(payload)), port });
3
4 const base64ToHex = (() => {
5   const values = [] , output = [];
6
7   return function base64ToHex(txt) {
8     if (output.length <= 0) populateLookups();

```

⁶Milesight EM300-TH Decoder

```

9      const result = [];
10     let v1, v2, v3, v4;
11     for (let i = 0, len = txt.length; i < len; i += 4) {
12       v1 = values[txt.charCodeAt(i)];
13       v2 = values[txt.charCodeAt(i + 1)];
14       v3 = values[txt.charCodeAt(i + 2)];
15       v4 = values[txt.charCodeAt(i + 3)];
16       result.push(
17         parseInt(output[(v1 << 2) | (v2 >> 4)], 16),
18         parseInt(output[((v2 & 15) << 4) | (v3 >> 2)], 16),
19         parseInt(output[((v3 & 3) << 6) | v4], 16)
20       );
21     }
22     if (v4 === 64) result.splice(v3 === 64 ? -2 : -1);
23     return result;
24   };
25   function populateLookups() {
26     const keys =
27       "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/-";
28     ← mnopqrstuvwxyz0123456789+/-;
29     for (let i = 0; i < 256; i++) {
30       output.push(("0" + i.toString(16)).slice(-2));
31       values.push(0);
32     }
33     for (let i = 0; i < 65; i++) values[keys.charCodeAt(i)] = i;
34   }
35 })();
36
37 function decoder(bytes, port) {
38   let decoded = {}, temperature = {}, airHumidity = {}, battery = {};
39   for (let i = 0; i < bytes.length;) {
40     let channel_id = bytes[i++];
41     let channel_type = bytes[i++];
42     if (channel_id === 0x01 && channel_type === 0x75) {
43       decoded.battery = battery;
44       battery.percentage = bytes[i];
45       i += 1;
46     } else if (channel_id === 0x03 && channel_type === 0x67) {
47       decoded.temperature = temperature;
48       temperature.celsius = readInt16LE(bytes.slice(i, i+2))/10;
49       i += 2;
50     } else if (channel_id === 0x04 && channel_type === 0x68) {
51       decoded.airHumidity = airHumidity;
52       airHumidity.relativePercentage = bytes[i] / 2;
53       i += 1;
54     } else {
55       break;
56     }
57   }
58   return decoded;
59 }
60 const readUInt16LE = bytes => (bytes[1] << 8) + bytes[0] & 0xffff;
61
62 function readInt16LE(bytes) {
63   let ref = readUInt16LE(bytes);
64   return ref > 0xffff ? ref - 0x10000 : ref;
65 }
66 const convert = dataUnit => ({
  dataId: dataUnit.uuid,

```

```

67     reportedAt: dataUnit.reported_at ,
68     device: {
69       id: dataUnit.id ,
70       name: dataUnit.name ,
71       downlink: dataUnit.downlink_url ,
72     },
73     measures: decodePayload(dataUnit.payload , dataUnit.port) ,
74   };

```

Listing 6.8: 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 *O*. 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 decoder⁷ of this device.

6.2.7 Sensae Console Database Configuration

The solution designed relies on various databases, and as discussed in Section 6.1.6.1 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 6.9 exemplifies the content of this scripts.

```

1 create table if not exists public.transformation
2 (
3   persistence_id bigint generated by default as identity
4     primary key ,
5   device_type      varchar(255)
6     constraint unique_type_constraint
7     unique
8 );
9
10 create table if not exists public.property_transformation
11 (
12   persistence_id          bigint generated by default as
13   identity (maxvalue 2147483647)
14     primary key ,
15   value                   integer           not null ,
16   old_path                varchar(255) ,
17   transformation_persistence_id bigint
18     constraint ref_transformation_constraint
19     references public.transformation ,
20   sub_sensor_id           integer default 0 not null
21 );

```

⁷Milesight EM300-TH Decoder

Listing 6.9: Initialization Script Segment for Data Processor Database

This script defines two simple tables, *transformation* and *property_transformation*, following the concepts defined in Section 5.4.3.1.

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 6.10:

```

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,
14        unallocated_oid]);
15    INSERT INTO public.tenant (name, oid, phone_number, email,
16      domains)
17      VALUES ('Anonymous', gen_random_uuid(), '', '',
18        ARRAY[public_oid]);
19    INSERT INTO public.tenant (name, oid, phone_number, email,
20      domains)
21      VALUES ('Admin', gen_random_uuid(), '', '$SENSAE_ADMIN_EMAIL',
22        ARRAY[root_oid]);
23    RETURN root_oid;
24  END;
$root_oid$ LANGUAGE plpgsql;
25
26 select public.init_domains();
27
28 DROP FUNCTION public.init_domains;

```

Listing 6.10: 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 replaced by a valid email before building the database container with the full script. This variable configuration is discussed in the Section 6.2.11.

6.2.8 Sensae Console Containerization

The section describes how **Sensae Console** is containerized with docker. As explained in Section 6.1.9, the author choose to containerize the solution.

The following Code Samples describe how each container mentioned in Section 5.2.2.2 are packaged. To simplify, only three distinct samples will be presented.

The first sample, Listing 6.11, 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 6.11: 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 6.2.10 Section.

The second sample, Listing 6.12, refers to **Fleet Management Backend** and is similar to all backend containers in the Configuration or Services Scope.

```

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 6.12: 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 than 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 6.13, 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 6.13: 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**.

6.2.9 Sensae Console Orchestration

As described in Section 5.2.2.5, *Overview of Docker Compose* was the tool used to orchestrate the solution. 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/
40      ↳ postgresql/data/
41  data-processor-flow:
42    build: backend-services/data-processor-flow
43    image: sensae/data-processor-flow
44    env_file:
45      - ./secrets/prod/data-processor-flow.env
46    networks:
47      - sensae-network
48  networks:
49    sensae-network:
```

Listing 6.14: 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;

6.2.10 Sensae Console Reverse Proxy Configuration

This section reveals how *Nginx* is configured. As an example, the Listing 6.15, describes the **Smart Irrigation Frontend**.

```

1 server {
2
3     server_name localhost;
4
5     listen 443 ssl;
6
7     ssl_certificate /etc/nginx/ssl/nginx.crt;
8     ssl_certificate_key /etc/nginx/ssl/nginx.key;
9
10    root      /usr/share/nginx/html;
11
12    index     index.html index.htm;
13
14    include   /etc/letsencrypt/options-ssl-nginx.conf;
15
16    location ~ .*remoteEntry.js$ {
17        expires -1;
18        add_header 'Cache-Control' 'no-store, no-cache, must-revalidate,
19        proxy-revalidate, max-age=0';
20    }
21
22    location /smart-irrigation/graphql {
23        proxy_pass http://smart-irrigation-backend:8080/graphql;

```

```

23     proxy_set_header x-forwarded-prefix /smart-irrigation/graphql;
24     proxy_set_header Host $host;
25     proxy_set_header x-forwarded-host $host;
26     proxy_redirect off;
27     proxy_set_header x-forwarded-port 443;
28     proxy_set_header x-forwarded-proto https;
29 }
30
31 location /smart-irrigation/subscriptions {
32     proxy_pass http://smart-irrigation-backend:8080/subscriptions;
33     proxy_set_header x-forwarded-prefix /smart-irrigation/
34     subscriptions;
35     proxy_http_version 1.1;
36     proxy_set_header Upgrade $http_upgrade;
37     proxy_set_header Connection "Upgrade";
38     proxy_set_header Host $host;
39     proxy_read_timeout 6000;
40     proxy_send_timeout 6000;
41     proxy_redirect off;
42     proxy_set_header x-forwarded-port 443;
43     proxy_set_header x-forwarded-proto https;
44 }
45
46 location / {
47     try_files $uri $uri/ /index.html;
48 }
49
50 if ($scheme != "https") {
51     return 301 https://$host$request_uri;
52 } # managed by Certbot
53 }
```

Listing 6.15: Configuration File for Production Environment

The following conclusions can be observed:

- 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**.

6.2.11 Sensae Console Configuration Files

This section describes how a **Sensae Console** is 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 6.16, 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=
21 export SENSAE_EMAIL_SUBJECT=
22 export SENSAE_EMAIL_SENDER_PASSWORD=
23 export SENSAE_EMAIL_SMTP_HOST=
24 export SENSAE_EMAIL_SMTP_PORT=
25 export SENSAE_EMAIL_ACTIVATE=
26 export SENSAE_PROD_PUBLIC_DOMAIN=
27 export SENSAE_ADMIN_EMAIL=

```

Listing 6.16: 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 6.17 sheds a light on how the script propagates the configurations.

```

1#!/usr/bin/sh
2
3ROOT_DIR=$( git rev-parse --show-toplevel )
4
5cd "$ROOT_DIR"/project || exit
6
7./secrets/prod.conf
8
9SECRET_BACK=secrets/templates/prod/backend-services
10SECRET_FRONT=secrets/templates/prod/frontend-services
11SECRET_DB=secrets/templates/prod/databases

```

```

12 BACK_PREFIX=secrets/prod
13 FRONT_PREFIX=frontend-services/apps
14 FRONT_SUFFIX=src/environments/environment.prod.ts
15
16 envsubst < $SECRET_BACK/alert-dispatcher-backend.env > \
17     $BACK_PREFIX/alert-dispatcher-backend.env
18 # and all other backend services
19 envsubst < $SECRET_BACK/data-validator.env >
20     $BACK_PREFIX/data-validator.env
21
22
23 envsubst < $SECRET_FRONT/device-management-frontend.ts > \
24     $FRONT_PREFIX/device-management-frontend/$FRONT_SUFFIX
25 # and all other frontend services
26 envsubst < $SECRET_FRONT/ui-aggregator.ts > \
27     $FRONT_PREFIX/ui-aggregator/$FRONT_SUFFIX
28
29 envsubst < secrets/templates/prod/message-broker/message-broker.env > \
30     $BACK_PREFIX/message-broker.env
31
32 envsubst < $SECRET_DB/data-decoder-database.env > \
33     $BACK_PREFIX/data-decoder-database.env
34 # and all other databases
35 envsubst < $SECRET_DB/rule-management-database.env > \
36     $BACK_PREFIX/rule-management-database.env

```

Listing 6.17: Configuration Propagation Script

In the future, as more isolated deployments are made, a tool such as *Vault* should be integrated in the solution.

6.2.12 External Services

This section refers how services interact with the **Data Flow Scope**, this was briefly mentioned in Section 5.4.2.3.

In order to provide an easy to understand integration with the **Data Flow Scope**, 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 can capture are: (i) **Data Units** and (ii) **Alerts**. Each of this information are defined by their routing keys as described in Table 5.4.

A service can also publish **Commands**.

The following sub sections will detail each service information needs.

6.2.12.1 Smart Irrigation Service

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 Bounded Context, 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.

6.2.12.2 Fleet Management Service

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 GPS data sent to the 'fleet' channel.

6.2.12.3 Notification Management Service

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 5.15) that subscribe to the same information but handle it differently.

6.2.13 Sensae Console Device Integration

This section describes how devices can be connected to **Sensae Console**. As stated in Section 4.2, 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 6.2.4.

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 6.2.4 instructions.

The Figure 6.11 presents an example of the custom integration for the EM300-TH Device.

The screenshot displays the Helium Custom Integration Page for a device named "Sensae EM300-TH".

- INTEGRATION DETAILS:** Shows the device name, status (0/50), and an "Update" button.
- HTTP DETAILS:** Shows the type as "HTTP", ID as "8f9992e5-082a-48e9-99b8-44c66d51dcfc", and a "Delete Integration" button.
- Connection Details:**
 - Device Joins: 0
 - Piped Devices: 8
 - Downlink URL: https://console-vip.helium.com/api/v1/down/8f9992e5-082a-48e9-99b8-44c66d51dcfc/8vIkGcs2Gh7wu80ID5AH-QqgN7GBElsn/{optional_device_id}
 - Downlink Key: A masked key followed by "***** //Authorization secret".
- UPDATE YOUR CONNECTION DETAILS:**
 - Method: POST (selected)
 - Endpoint URL (Required): <https://location-tracking.sharespot.dev:8443/sensor-data/irrigation/encoded/em300th>
 - HTTP Headers (Optional usage for payload interpolation): Authorization: ***** //Authorization secret

Figure 6.11: Helium Custom Integration Page

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

6.3 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 2022.

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 met 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 pursued.

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 Design discussed in the Components Level - Logical View Section 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.

6.3.1 Unit Tests

This section focus on unit tests preformed throughout the solution.

The test presented in Listing 6.18 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*.

```

1 @Test
2 void ensureTransferWorksWithValidArrayPath() throws
3     JsonProcessingException {
4     var jsonNode = mapper.readTree("""
5         {
6             "path": [
7                 {
8                     "prop": "viva"
9                 }
10            ]
11        """
12    );
13    var objectNode = mapper.createObjectNode();

```

```

14     new KnownPropertyTransformation(
15         "path[0].prop", PropertyName.DEVICE_ID, 2)
16             .transfer(jsonNode, objectNode);
17
18     Assertions.assertEquals("viva",
19         objectNode.get("device").get("id").asText());
20 }

```

Listing 6.18: Unit Test Example in *iot-core* package

The test presented in Listing 6.19 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 6.19: Unit Test - Data Decoder Backend Container

The Listing 6.20 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 6.20: Unit Test - Device Management Frontend Model Library

6.3.2 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 automatically starting and shutting down the containers needed to perform this tests.

The code in Listing 6.21 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 6.21: 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 received in line 27.

The code in Listing 6.22 verifies that the database can be reached by the **Notification Management Backend**.

```

1 public class NotificationRepositoryImplTest extends IntegrationTest {
2
3     @Autowired
4     NotificationRepositoryImpl repository;
5
6     @Test
7     public void ensureDatabaseCanBeReached() {
8         var single = Domains.single(DomainId.of(UUID.randomUUID()));
9         var type = ContentType.of("a", "a", NotificationLevel.CRITICAL);
10        var query = NotificationBasicQuery.of(single, List.of(type));
11
12        Assertions.assertDoesNotThrow(() -> repository.find(query));
13    }
14 }
```

Listing 6.22: 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.

6.3.3 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 6.23 and 6.24 present the needed setup to run functional tests using *Test Containers* and *Spring Boot*.

```

1 public class DatabaseContainerTest extends
2     PostgreSQLContainer<DatabaseContainerTest> {
```

```

2
3     private static final String IMAGE_VERSION = "data-decoder-database";
4     private static DatabaseContainerTest container;
5
6     private DatabaseContainerTest() {
7         super(DockerImageName.parse(IMAGE_VERSION)
8             .asCompatibleSubstituteFor("postgres:14.5"));
9     }
10
11    public static DatabaseContainerTest getInstance() {
12        if (container == null) {
13            container = new DatabaseContainerTest()
14                .withUsername("user")
15                .withPassword("sa")
16                .withEnv("POSTGRESQL_USER", "user")
17                .withEnv("POSTGRESQL_PASSWORD", "sa")
18                .withExposedPorts(PostgreSQLContainer.POSTGRESQL_PORT);
19        }
20        return container;
21    }
22
23    @Override
24    public void stop() {
25        //do nothing, JVM handles shut down
26    }
27}

```

Listing 6.23: Functional Test - Message Broker - **Data Decoder Master 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 6.3. The same notion is applied for the Message Broker Container. These two containers are the ones that **Data Decoder Master Backend** depends on.

The Listing 6.24 presents the foundation of functional and integration tests.

```

1 @SpringBootTest
2 @Testcontainers
3 @ContextConfiguration(initializers =
4     ↳ {IntegrationTest.Initializer.class})
5 @ActiveProfiles(profiles = "test")
6 public abstract class IntegrationTest {
7     static class Initializer implements
8         ApplicationContextInitializer<ConfigurableApplicationContext> {
9         public void initialize(ConfigurableApplicationContext context) {
10            db.withDatabaseName("decoder");
11            TestPropertyValues.of(
12                "spring.datasource.url=" + db.getJdbcUrl(),
13                "spring.datasource.username=" + db.getUsername(),
14                "spring.datasource.password=" + db.getPassword(),
15                "spring.rabbitmq.host=" + mb.getHost(),
16                "spring.rabbitmq.port=" + mb.getAmqpPort(),
17                "spring.rabbitmq.username=" + mb.getAdminUsername(),
18                "spring.rabbitmq.password=" + mb.getAdminPassword())
19            ).applyTo(context.getEnvironment());
}

```

```

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(postgreSQLContainer);
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 6.24: Functional Test - Foundation - **Data Decoder Master 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 6.25, 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 6.25: Functional Test - Database Interaction - **Data Decoder Master**
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 6.26, 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            SensorTypeld.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 6.26: Functional Test - Message Broker Interaction - **Data Decoder Master Backend**

In this test the class to verify is the *DataDecoderHandlerService*. Once again this test extends the *IntegrationTest* class. Using *RabbitAdmin*, it's created a queue that subscribes to the expected type of routing keys in lines **18** to **29** and then bind 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 6.27.

```

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(MediaType.APPLICATION_JSON)
9             .contentType(MediaType.APPLICATION_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 6.27: Functional Test - Rest Client Interaction - **Data Gateway**

This test simply attempts to send an HTTP POST request to an invalid resource - line **11**.

6.3.4 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 6.28.

```

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 Cypress.Commands.add('goToIdentityPage', () => {
21   console.log('Custom command: go to Identity Page');
22   cy.get('#tools').click();
23   cy.contains('Identity Management').click();
24 });
25

```

Listing 6.28: 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 6.29 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 6.29: 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 6.30 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 6.30: 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 Bounded Context Section.

6.3.5 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 6.31 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 6.31: Architectural Test - Onion Architecture - **Device Management Master 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 6.32 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);
```

```

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 6.32: 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.

6.4 Synopsis

This chapter introduced the most important technical decisions taken during the solution's implementation. This 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 7

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 6.3. 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 4.2.

7.1 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 4.2.

Since the solution was designed to scale infinitely and handle high-levels of throughput, the performance of it in a multi-organization and shared infrastructure is undermined. The evaluation should instead focus on environments where resources are constrained.

Therefore, the performance of the solution will be tested against above-average usage conditions of small to medium organizations that require a dedicated infrastructure, 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.

7.2 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 500¹.

The evaluation encompasses 3 test scenarios, one for each developed service: (i) Fleet Management, (ii) Smart Irrigation, (iii) 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 F. The *K6* tool produces various metrics that are then analyzed using *R*, an example of the analysis scripts developed is presented in Appendix G.

For simplicity, the solution was deployed in a 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.

This 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 this realistic tests. Therefore, metrics such as the interval between two consecutive Uplinks of the same device, were severely narrowed.

7.3 Experiences

As described before, 3 scenarios that emulate a variable number of devices sending uplinks to **Sensae Console** were tested:

- **Scenario 1:** Fleet Management;
- **Scenario 2:** Smart Irrigation;

¹500 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

- **Scenario 3:** Notification Management;

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 in the **Service** Scope;
- 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 7.1 summarizes the experiences preformed.

Table 7.1: 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.

7.3.1 Fleet Management Experience Scenario

This scenario focus on the Fleet Management Service. The Table 7.2 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 7.2: 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

This 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 real number of devices is much higher than the one determined by this tests.

7.3.2 Smart Irrigation Experience Scenario

This scenario focus on the Smart Irrigation Service. The Table 7.3 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 7.3: 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

This 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 real number of devices is much higher than the one determine by this tests.

7.3.3 Notification Management Experience Scenario

This scenario focus on the Notification Management Service. The Table 7.4 presents the results related to the time is 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 7.4: 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.

7.4 Discussion of the overall results

As seen by the experiences preformed the system was capable of answering the defined requirements.

Apart from the results gathered it is 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 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 databases in the **External Services Scope** that store measures or notifications;
- The **Message Broker** that routes data units through the system;
- The WebSocket implementation used by GraphQL to supply measures and notifications in the Backends of the **External Services Scope**;
- The process of filling the **Data Flow Scope** Caches with the information managed by the various containers in the **Configuration Scope**.

7.4.1 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.

This experiences yielded the following results:

Table 7.5: Data Ingestion Endpoint response time results (in milliseconds)

Scenario	Average	Min	Median	Max	90% Percentile	95% Percentile
1	193.24	170.02	182.12	581.6	219.16	249.24
2	227.48	169.19	200.17	727.11	319.77	376.98
3	179.83	167.09	178.99	2632	186.36	189.27

This results present a stable performance even under a high number of requests, apart from the maximum value obtained in scenario 2, 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 dropping requests.

7.4.2 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 7.1 helps to debunk this belief.

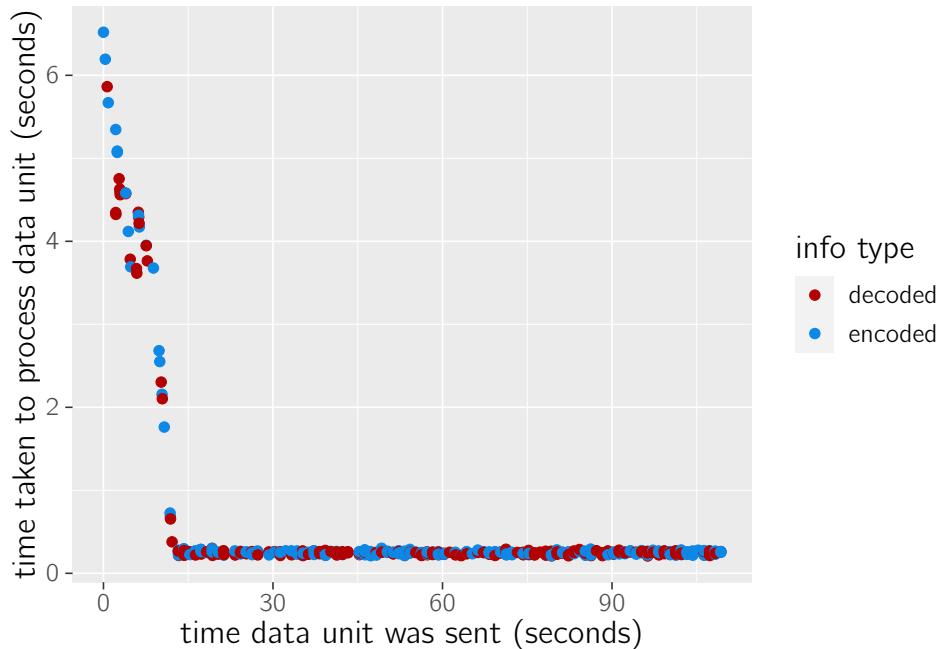


Figure 7.1: Scatter Chart - Notification Management Scenario - Experience C

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 Context and the ones with the 'encoded' *info type*, in blue, were sent to the Data Decoder Context.

In conjunction with the chart, the Table 7.6 presents some analysis preformed against the results of Experience C in the Notification Management Scenario.

Table 7.6: 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*.

7.4.3 Data Flow Caching Process Performance

The experiences preformed clearly display the process mentioned during the Design Chapter in Figure 5.20, about how the **Data Flow** state is maintained.

In the following charts, Chart 7.2 and 7.3, it's possible to envision the various caches in the **Data Flow** being filled during the first iteration.

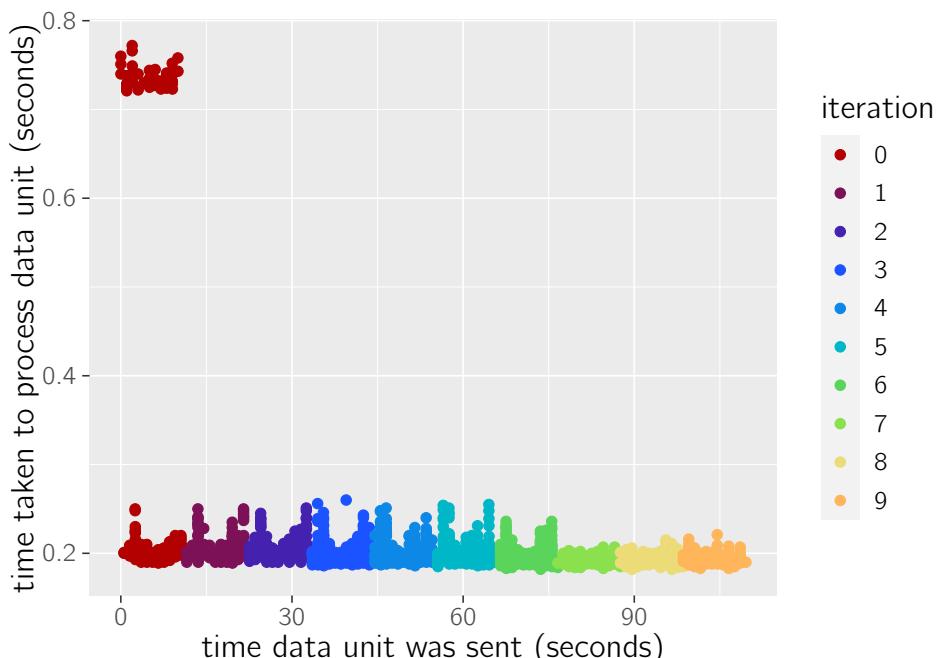


Figure 7.2: Scatter Chart - Smart Irrigation Scenario - Experience B

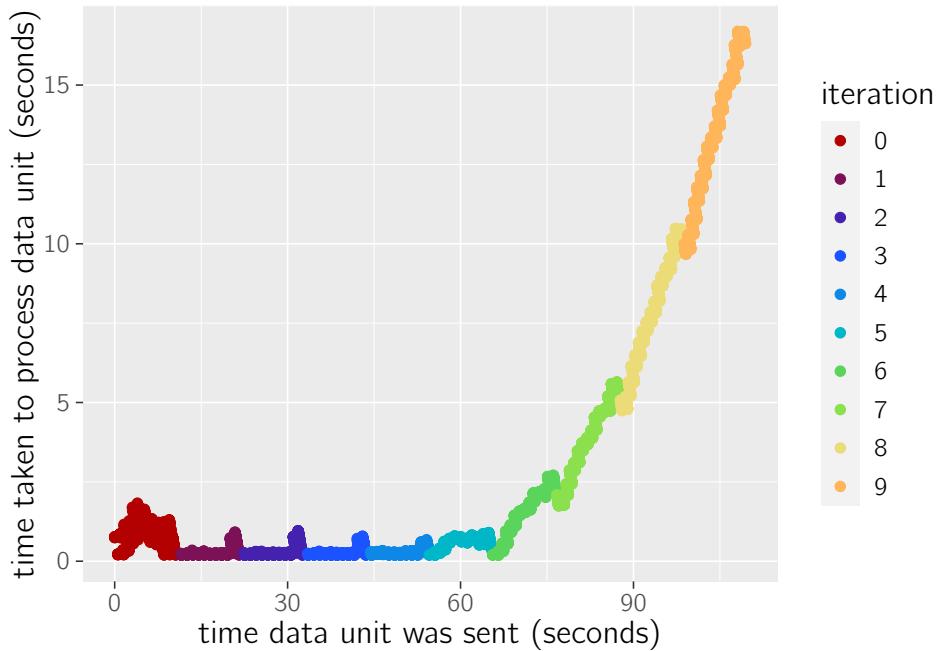


Figure 7.3: Scatter Chart - Fleet Management Scenario - Experience D

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 7.3 also shows that the system started to underperform around the 65 seconds mark. The **Data Flow** caches were already stable and therefore, under this experience, it is plausible to say that this process doesn't cause the performance degradation seen in higher throughput experiences.

7.4.4 External Services Scope Database Performance

Another important question is whether 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 6.1.6, the database used for this container is *PostgreSQL*. 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 7.4, 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).

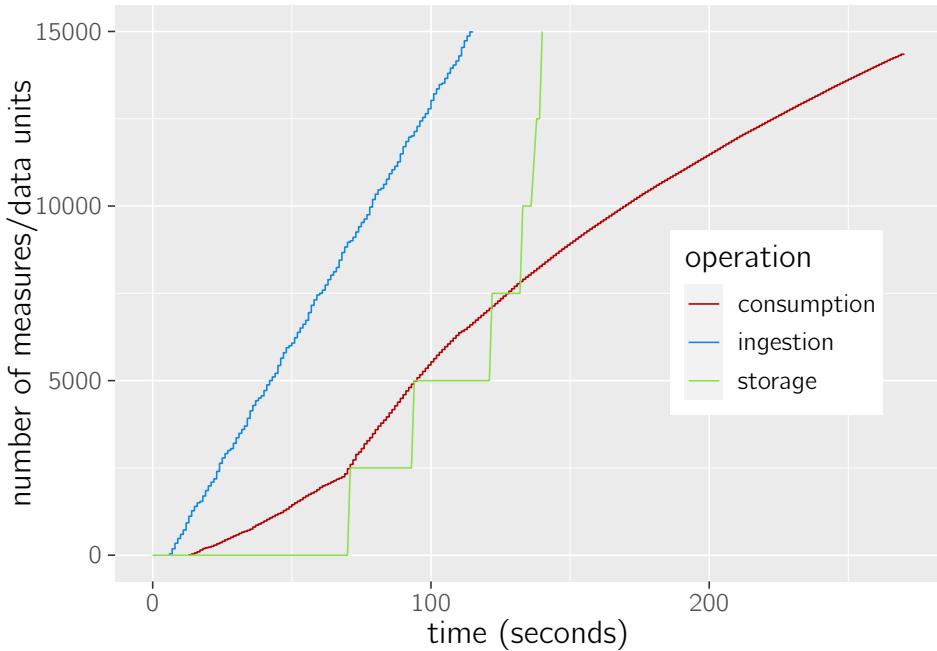


Figure 7.4: Line Chart - Time Taken to Ingest, Store and Supply Measures

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 Page² 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).

7.4.5 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 7.4.

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.

²link to *QuestDB* Commit Strategy Page

The experiences **H** of each scenario 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 new devices, decoders or processors that are not already cached.

7.5 Synopsis

The system performed successfully under the requirements defined. The conducted analysis helped to identify the most important bottlenecks to tackle in the future: (i) GraphQL Subscriptions, (ii) *PostgresSQL* Databases.

The performance tests helped the company to understand the platform limits and how far the production environments are from reaching this limits.

Based on this evaluation and the work described before, the following chapter describes the author opinion regarding the solution.

Chapter 8

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 meeting with costumers. The developed PoCs had a positive evaluation regarding their performance. Even though no 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 that most work needed to implement these PoCs could be integrated in the platform, **Sensae Console**.

8.1 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 virtual 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.

8.2 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 constraints 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 performed by analyzing questionnaires handed to employees of the organization A. Once again these 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.

8.3 Future Work

This project, and the solution it originated, still have a lot of ideas and features that are 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 projects measured the volume of wheat inside Silos. The sensors were installed in the silo's ceiling pointing downwards and 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 projects 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;
- 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;
- 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, the process of creating a public, monetizable platform to ease the creation of IoT Services is complex and drawn-out. Maintaining a service like this feels even more like an interminable task due to all the business cases surrounding IoT.

8.4 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, customers 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 knowledge regarding the IoT world and also the difficulties surrounding the creation of a business from the ground up.

Bibliography

- Amazon (2022). *Amazon Cognito*. url: <https://aws.amazon.com/cognito/>.
- Andy Clement Sébastien Deleuze, Filip Hanik (2022). *Spring Native*. url: <https://docs.spring.io/spring-native/docs/current/reference/htmlsingle/>.
- 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.
- Bitkeeper (2022). *Bitkeeper*. url: <https://www.bitkeeper.org/>.
- Brown, Simon (June 2018a). *The C4 Model for Software Architecture*. [Online; accessed 30. Jun. 2022]. url: <https://www.infoq.com/articles/C4-architecture-model/>.
- (2018b). *The C4 model for visualising software architecture*. [Online; accessed 30. Jun. 2022]. url: <https://c4model.com>.
- 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>.
- 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.
- Cisco (2017). *Cisco Survey Reveals Close to Three-Fourths of IoT Projects Are Failing*. Accessed: February 8, 2022.
- 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.
- 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>.
- 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.
- 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>.
- 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*.
- 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>.
- 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/>.
- Fowler, Martin and James Lewis (2014). *Microservices*. url: <https://www.martinfowler.com/articles/microservices.html>.
- Gartner (n.d.). *Customer Identity and Access Management (CIAM)*. url: <https://www.gartner.com/en/information-technology/glossary/customer-identity-access-management-ciamp>.
- 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."
- Git (2022). *Git*. url: <https://git-scm.com/>.
- Google (2022a). *Angular*. url: <https://angular.io/>.
- (2022b). *Google Cloud IoT*. Accessed: February 8, 2022.
- (2022c). *Google Maps*. url: <https://mapsplatform.google.com/maps-products/>.
- (2022d). *Protocol Buffers*. url: <https://developers.google.com/protocol-buffers/>.
- (n.d.). *Google Identity Platform*. url: <https://cloud.google.com/identity-platform/>.
- Grafana (2022). *K6*. url: <https://k6.io/>.

- Han, Jing et al. (2011). "Survey on NoSQL database". In: *2011 6th international conference on pervasive computing and applications*. IEEE, pp. 363–366.
- 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 (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>.
 - (2022). *How does software testing work?* url: <https://www.ibm.com/topics/software-testing>.
- Ilyushchenko, Vlad (2021). *How we achieved write speeds of 1.4 million rows per second*. Accessed: February 24, 2022.
- InfluxDB (2022a). *InfluxDB*. url: <https://www.influxdata.com/>.
- (2022b). *InfluxDB line protocol tutorial*. url: https://docs.influxdata.com/influxdb/v1.8/write_protocols/line_protocol_tutorial/.
- Jacobs, Mike and Craig Casey (2022). *What are Microservices?* url: <https://docs.microsoft.com/en-us/devops/deliver/what-are-microservices>.
- Jansen, Grace (Apr. 2020). *Getting started with Reactive Systems*. url: <https://developer.ibm.com/articles/reactive-systems-getting-started/>.
- 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>.
- Klishin, Michael (2022). *Fetching Individual Messages ("Pull API")*. url: <https://www.rabbitmq.com/consumers.html>.
- Lakshman, Avinash and Prashant Malik (2010). "Cassandra: a decentralized structured storage system". In: *ACM SIGOPS Operating Systems Review* 44.2, pp. 35–40.
- Lewis, William E. (2008). *Software Testing and Continuous Quality Improvement*.
- Lighttpd (2022). *Lighttpd*. url: <https://www.lighttpd.net/>.
- Mapbox (2022). *Mapbox GL JS*. url: <https://www.mapbox.com/mapbox-gljs>.
- MariaDB (2022). *MariaDB*. url: <https://mariadb.org/>.

- 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=OHYhAQAAIAAJ>.
- Mercurial (2022). *Mercurial*. url: <https://www.mercurial-scm.org/>.
- Microsoft (2022a). *Azure IoT*. Accessed: February 8, 2022.
- (2022b). *Github*. url: <https://www.github.com/>.
- (2022c). *TypeScript*. url: <https://www.typescriptlang.org/>.
- 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/>.
- MongoDB (2022). *MongoDB*. url: <https://www.mongodb.com/>.
- Mozilla (2022). *Javascript*. url: <https://developer.mozilla.org/en-US/docs/Web/JavaScript>.
- 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.
- Newman, S. (2021). *Building Microservices*. O'Reilly Media. isbn: 9781492033974. url: <https://books.google.pt/books?id=ZvM5EAAAQBAJ>.
- 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 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/>.
- 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/>.
- 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.

- Pittet, Sten (2022). *The different types of software testing*. url:
<https://www.atlassian.com/continuous-delivery/software-testing/types-of-software-testing>.
- 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>.
- questdb.io (2022). *QuestDB*. url: <https://questdb.io>.
- 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>.
- 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=Bd5mAQAAQAAJ>.
- 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). *Externalized configuration Pattern*. url:
<https://microservices.io/patterns/externalized-configuration.html>.
- 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.
- Slee, Mark, Aditya Agarwal, and Marc Kwiatkowski (2007). “Thrift”. In: *Facebook white paper 5.8*, p. 127.
- Sonatype (2022). *Why Do We Have Requirements?* url:
<https://central.sonatype.org/publish/requirements/>.
- Spring (2022). *Spring Native documentation*. url:
<https://docs.spring.io/spring-native/docs/current/reference/htmlsingle/>.

- 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.
- 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.
- 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>.
- Webpack (2022). *Module Federation*. url: <https://webpack.js.org/concepts/module-federation/>.
- Winslow, Robert (2021). *Time Series Benchmark Suite (TSBS)*. Accessed: February 24, 2022.

Appendix A

Data Unit - Shared Model Schema

This schema represents the Shared Model Schema of a processed Data Unit as of *iot-core* package version 0.1.20.

```
1 {  
2     "dataId": "[uuid]",  
3     "reportedAt": "[long]",  
4     "device": {  
5         "id": "[uuid]",  
6         "name": "[string]",  
7         "downlink": "[string]",  
8         "records": [{  
9             "label": "[string]",  
10            "content": "[string]"  
11        }],  
12        "domains": ["[uuid]"],  
13        "commands": {  
14            "[int)": [{  
15                "id": "[uuid]",  
16                "name": "[string]",  
17                "payload": "[base64 string]",  
18                "port": "[int]"  
19            }]  
20        }  
21    },  
22    "measures": {  
23        "[int)": {  
24            "airHumidity": {  
25                "gramsPerCubicMeter": "[float]",  
26                "relativePercentage": "[float]"  
27            },  
28            "airPressure": { "hectoPascal": "[float]" },  
29            "aqi": { "value": "[float]" },  
30            "battery": {  
31                "percentage": "[float]",  
32                "volts": "[float]",  
33                "maxVolts": "[float]",  
34                "minVolts": "[float]"  
35            },  
36            "co2": { "ppm": "[float]" },  
37            "co": { "ppm": "[float]" },  
38            "distance": {  
39        }  
40    }  
41}
```

```
39      "millimeters": "[float]",
40      "maxMillimeters": "[float]",
41      "minMillimeters": "[float]"
42    },
43    "gps": {
44      "latitude": "[double]",
45      "longitude": "[double]",
46      "altitude": "[float]"
47    },
48    "illuminance": { "lux": "[float]" },
49    "motion": { "value": "[ACTIVE, INACTIVE or UNKNOWN]" },
50    "nh3": { "ppm": "[float]" },
51    "no2": { "ppm": "[float]" },
52    "o3": { "ppm": "[float]" },
53    "occupation": { "percentage": "[float]" },
54    "ph": { "value": "[float]" },
55    "pm2_5": { "microGramsPerCubicMeter": "[float]" },
56    "pm10": { "microGramsPerCubicMeter": "[float]" },
57    "soilConductivity": {
58      "microSiemensPerCentimeter": "[float]"
59    },
60    "soilMoisture": { "relativePercentage": "[float]" },
61    "temperature": { "celsius": "[float]" },
62    "trigger": { "value": "[boolean]" },
63    "velocity": { "kilometerPerHour": "[float]" },
64    "voc": { "ppm": "[float]" },
65    "waterPressure": { "bar": "[float]" }
66  }
67}
68}
```

Listing A.1: Data Unit - Shared Model Schema

Appendix B

Complete Container Level - Logical View

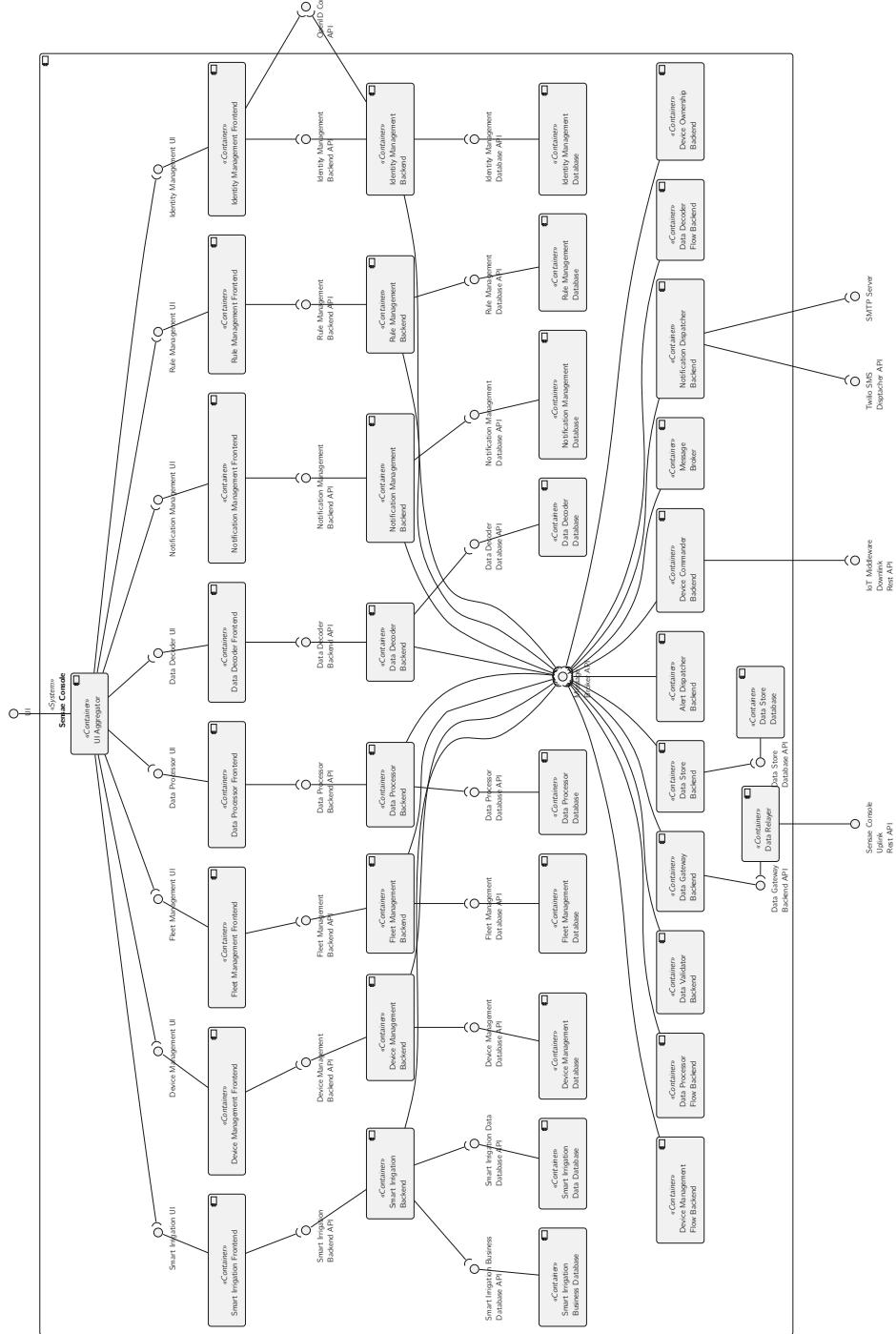


Figure B.1: Container Level - Logical View Diagram

Appendix C

Components Level - Logical View - Details

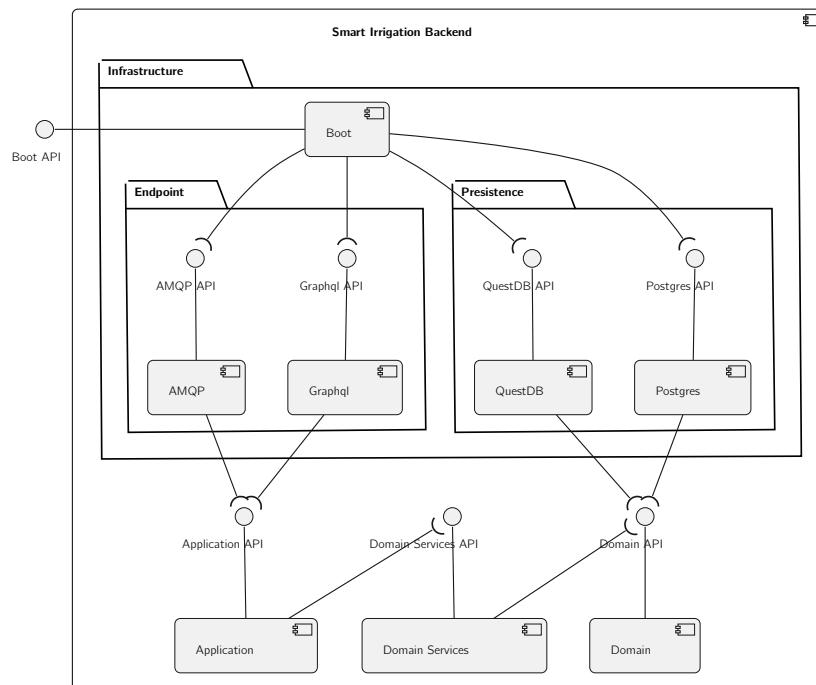


Figure C.1: Component Level - Smart Irrigation Backend - Logical View Diagram

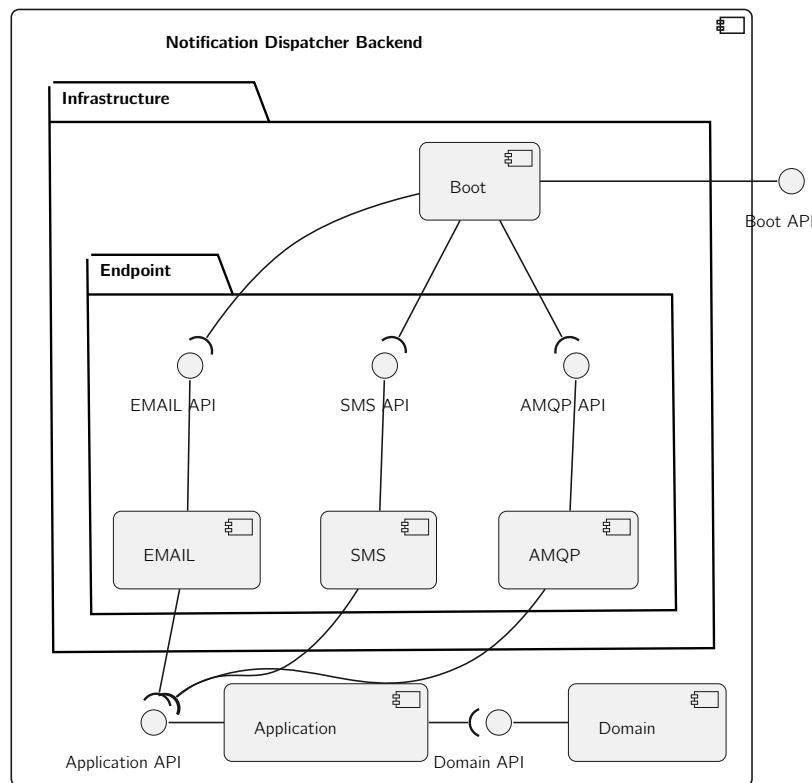


Figure C.2: Component Level - Notification Dispatcher Backend - Logical View Diagram

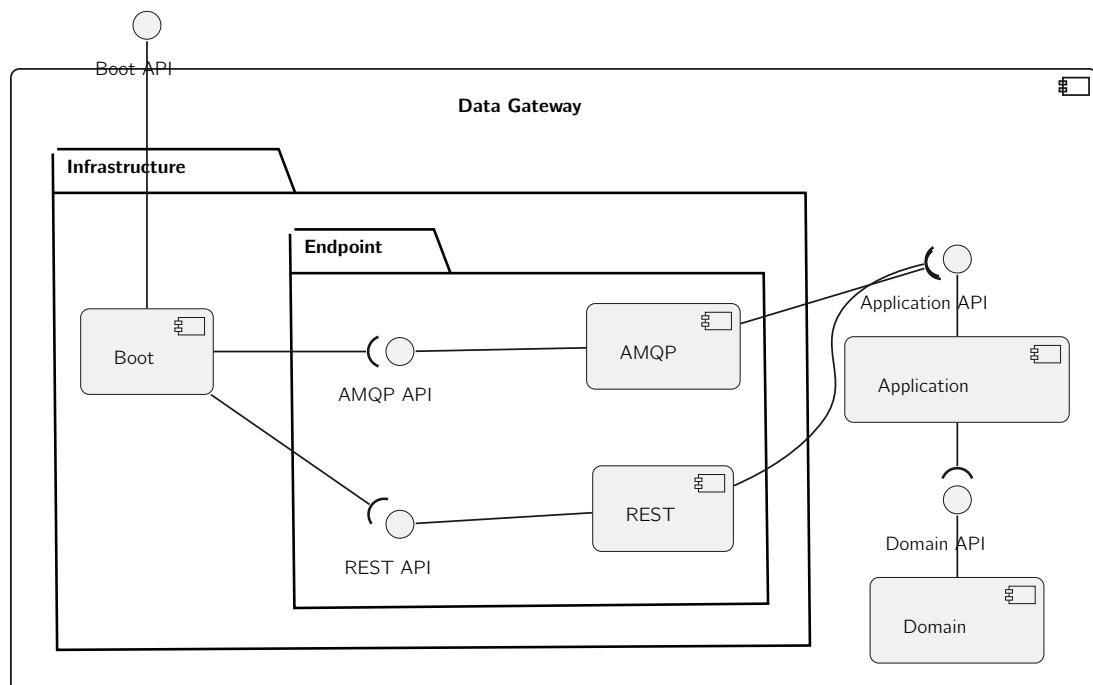


Figure C.3: Component Level - Data Gateway - Logical View Diagram

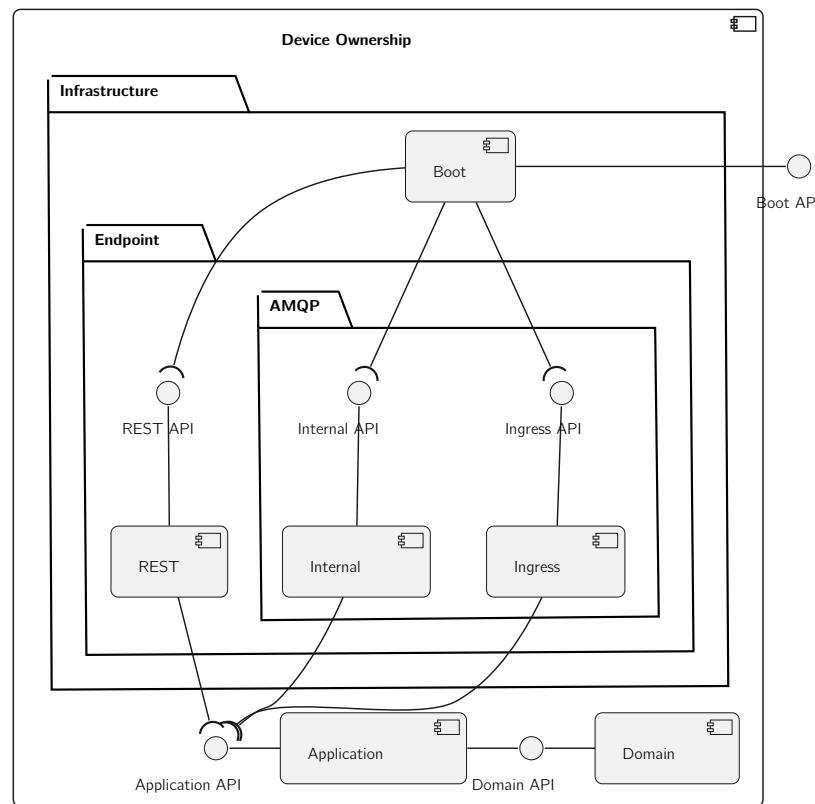


Figure C.4: Component Level - Device Commander - Logical View Diagram

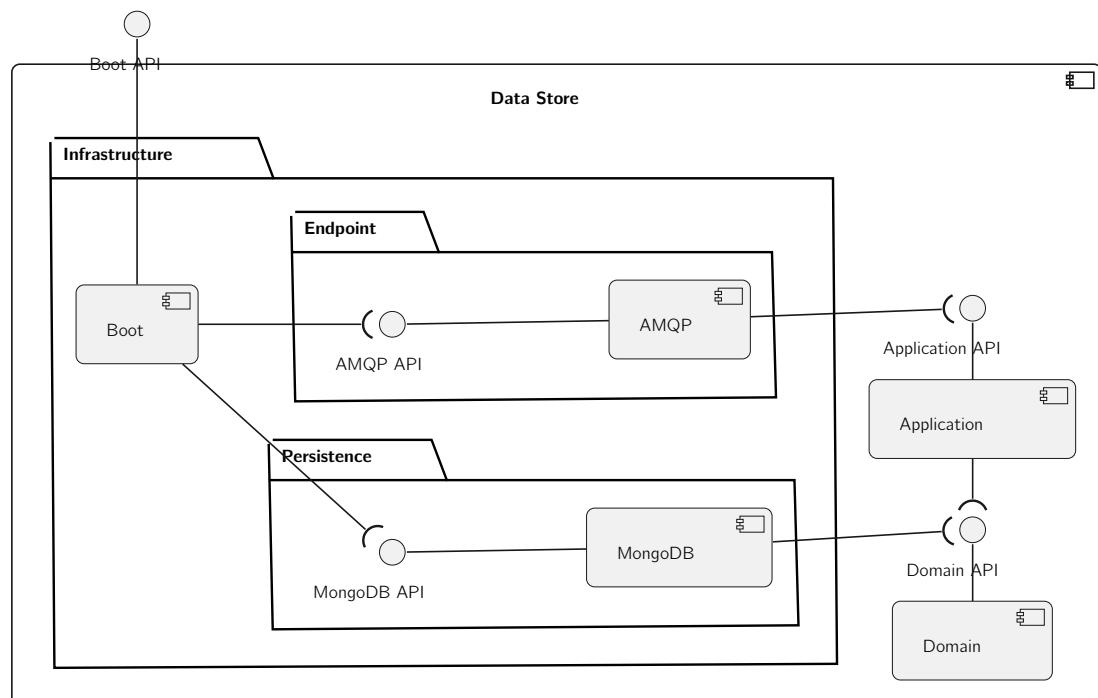


Figure C.5: Component Level - Data Store - Logical View Diagram

Appendix D

Additional Sensae Console UI Pages

This appendix presents other **Sensae Console** pages.

Figure D.1: Identity Management Page

Figure D.2: Rules Management Page

The screenshot displays a grid of six Data Transformation cards, each with a Device Type and various configuration options like 'Delete' and 'Update'. The cards include:

- Data Transformation (Device Type: controller)**: Device Downlink Url: downlink_url, Controller. Property Transformations: Device Downlink Url (downlink_url) Controller, Reported At (in millis) reported_at Controller.
- Data Transformation (Device Type: em300h)**: can only contain letters and numbers, e.g. lg92. Property Transformations: Data ID (uuid) Controller, Reported At (in millis) reported_at Controller.
- Data Transformation (Device Type: em300h)**: can only contain letters and numbers, e.g. lg92. Property Transformations: Data ID (uuid) Controller, Reported At (in millis) reported_at Controller.
- Data Transformation (Device Type: lg92)**: can only contain letters and numbers, e.g. lg92. Property Transformations: Device Downlink Url (downlink_url) Controller.
- Data Transformation (Device Type: park)**: can only contain letters and numbers, e.g. lg92. Property Transformations: Data ID (uuid) Controller, Reported At (in millis) reported_at Controller.
- Data Transformation (Device Type: stove)**: can only contain letters and numbers, e.g. lg92. Property Transformations: Device Downlink Url (downlink_url) Controller.

Figure D.3: Data Processor Page

The screenshot shows a Data Decoder card with the following details:

- Data Decoder (Device Type: em300h)**: can only contain letters and numbers, e.g. lg92.
- Script Content:**

```

38
39     function decoder(bytes, port) {
40         var decoded = {};
41         var temperature = {};
42         var airHumidity = {};
43         var battery = {};
44         decoded.temperature = temperature;
45         decoded.airHumidity = airHumidity;
46         decoded.battery = battery;
47
48         for (var i = 0; i < bytes.length; ) {
49             var channel_id = bytes[i+1];
50             var channel_type = bytes[i+2];
51
52             // BATTERY
53             if (channel_id === 0x01 && channel_type === 0x75) {
54                 battery.percentage = bytes[i];
55                 i += 1;
56             }
57             // TEMPERATURE
58             else if (channel_id === 0x03 && channel_type === 0x67) {
59                 temperature.celsius = readInt16LE(bytes.slice(i, i + 2)) / 10;
60                 i += 2;
61             }
62             // *F
63         }
    
```
- Message:** the script needs to have an entry point function called convert
- Last Update:** 1 minute ago

Figure D.4: Data Decoder Page

The screenshot shows a table of notifications with the following data:

Category	Sub Category	Severity	Reported	Read
Smart Irrigation	Dry Soil Detected	Yellow	1 month ago	X
Fire Detention	Temperature With High Rate Of Change	Orange	1 month ago	X
Fire Detention	Temperature With High Rate Of Change	Orange	1 month ago	X
Description: Project #001 - Device S#004 - Milesight EM300-TH Temperature changed from 31.6°C to 35.8°C.				
<input checked="" type="checkbox"/> Mark as read				
Fire Detention	Multiple Alarms Collected	Red	1 month ago	X
Fire Detention	Temperature With High Rate Of Change	Orange	1 month ago	X
Fire Detention	Temperature With High Rate Of Change	Orange	1 month ago	X
Smart Irrigation	Dry Soil Detected	Yellow	1 month ago	X

Figure D.5: Notification Management Page

The screenshot shows a configuration dialog box titled "Notification Delivery Configuration" with the following data:

Category	Sub Category	Severity	Show Old Notifications	Send UI Notification	Send Email	Send SMS	Delete
Fire Detention	CO2 With High Rate Of Change	Orange	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Fire Detention	Humidity With High Rate Of Change	Orange	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Fire Detention	Multiple Alarms Collected	Red	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>
Fire Detention	Temperature With High Rate Of Change	Orange	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Smart Irrigation	Dry Soil Detected	Yellow	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>

Below the table, there is a section for adding new categories and sub-categories:

- Category: Fire Detention
- Sub Category: New Sub Category
- Severity Level: Information
- Add entry button

At the bottom right of the dialog box are "Save Configuration" and "Exit" buttons.

Figure D.6: Notification Management Page - Configuration

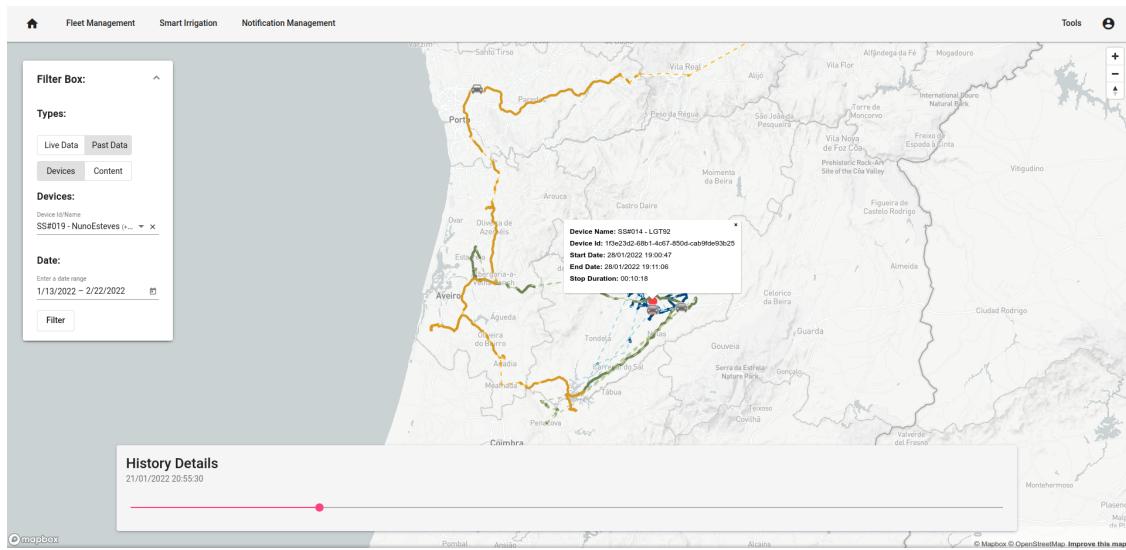


Figure D.7: Fleet Management Page

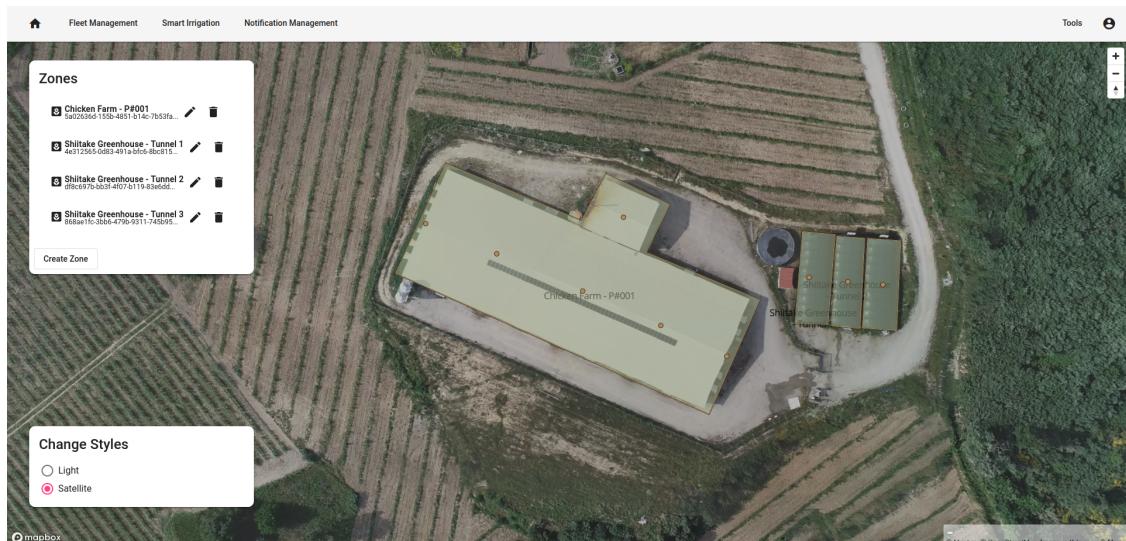


Figure D.8: Smart Irrigation Page - Map

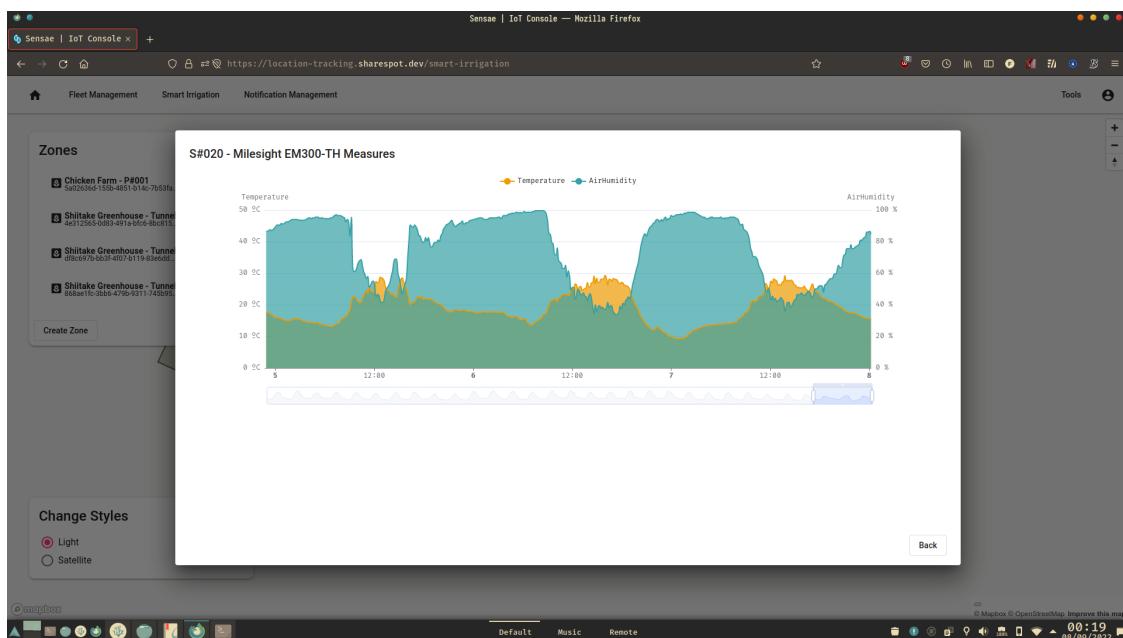


Figure D.9: Smart Irrigation Page - Device History

Appendix E

IoT Core Package POM

```

1<project>
2  <modelVersion>4.0.0</modelVersion>
3  <groupId>pt.sharespot</groupId>
4  <artifactId>iot-core</artifactId>
5  <version>0.1.20</version>
6  <name>${project.groupId}:${project.artifactId}</name>
7  <description>Core library to handle the structure and behavior
8   ↳ of messages inside Sensae Console</description>
9  <url>https://sharespot.pt</url>
10 <packaging>jar</packaging>
11 <properties>
12   <maven.compiler.source>17</maven.compiler.source>
13   <maven.compiler.target>17</maven.compiler.target>
14   <project.build.sourceEncoding>UTF-8</project.build.
15   ↳ sourceEncoding>
16 </properties>
17 <licenses>
18   <license>
19     <name>The MIT License (MIT)</name>
20     <url>https://mit-license.org/</url>
21   </license>
22 </licenses>
23 <developers>
24   <developer>
25     <id>filipeCruz</id>
26     <name>Filipe Cruz</name>
27     <email>filipe.cruz@sensae.pt</email>
28   </developer>
29 </developers>
30 <scm>
31   <connection>scm:git:git@github.com:ShareSpotPT/iot-core.git</
32   ↳ connection>
33   <developerConnection>scm:git:ssh://github.com:ShareSpotPT/iot
34   ↳ -core.git</developerConnection>
35   <url>https://github.com/ShareSpotPT/iot-core/tree/master</url>
36   </scm>
37 <dependencies><!-- omitted--></dependencies>
38 <build>
39   <finalName>${project.groupId}:${project.artifactId}</
40   ↳ finalName>
```

```
36   <plugins>
37     <plugin>
38       <groupId>org.apache.maven.plugins</groupId>
39       <artifactId>maven-compiler-plugin</artifactId>
40       <version>3.8.1</version>
41     </plugin>
42     <plugin>
43       <groupId>org.apache.maven.plugins</groupId>
44       <artifactId>maven-resources-plugin</artifactId>
45       <version>2.5</version>
46       <configuration>
47         <encoding>UTF-8</encoding>
48       </configuration>
49     </plugin>
50     <plugin>
51       <groupId>org.apache.maven.plugins</groupId>
52       <artifactId>maven-jar-plugin</artifactId>
53       <version>3.2.0</version>
54       <configuration>
55         <archive>
56           <manifest>
57             <addDefaultImplementationEntries>true</
58             <addDefaultImplementationEntries>
59             <addDefaultSpecificationEntries>true</
58             <addDefaultSpecificationEntries>
59               </manifest>
60             </archive>
61           </configuration>
62         </plugin>
63         <plugin>
64           <groupId>org.apache.maven.plugins</groupId>
65           <artifactId>maven-source-plugin</artifactId>
66           <version>3.2.1</version>
67           <executions>
68             <execution>
69               <id>attach-sources</id>
70               <goals>
71                 <goal>jar-no-fork</goal>
72               </goals>
73             </execution>
74           </executions>
75         </plugin>
76         <plugin>
77           <groupId>org.apache.maven.plugins</groupId>
78           <artifactId>maven-javadoc-plugin</artifactId>
79           <version>3.2.0</version>
80           <executions>
81             <execution>
82               <id>attach-javadocs</id>
83               <goals>
84                 <goal>jar</goal>
85               </goals>
86             </execution>
87           </executions>
```

```

88      <configuration>
89          <failOnErrors>false</failOnErrors>
90      </configuration>
91  </plugin>
92  <plugin>
93      <groupId>org.apache.maven.plugins</groupId>
94      <artifactId>maven-surefire-plugin</artifactId>
95      <version>2.22.2</version>
96  </plugin>
97 </plugins>
98 </build>
99 <profiles>
100    <profile><id>ci-cd</id>
101      <build>
102        <plugins>
103          <plugin>
104              <groupId>org.apache.maven.plugins</groupId>
105              <artifactId>maven-gpg-plugin</artifactId>
106              <version>3.0.1</version>
107              <executions>
108                <execution>
109                  <id>sign-artifacts</id>
110                  <phase>verify</phase>
111                  <goals>
112                      <goal>sign</goal>
113                  </goals>
114                  <configuration>
115                      <gpgArguments>
116                          <arg>--pinentry-mode</arg>
117                          <arg>loopback</arg>
118                      </gpgArguments>
119                  </configuration>
120                </execution>
121              </executions>
122          </plugin>
123        </plugins>
124      </build>
125    </profile>
126  </profiles>
127  <distributionManagement>
128    <snapshotRepository><id>ossrh</id>
129      <url>https://s01.oss.sonatype.org/content/repositories/
130      snapshots</url>
131    </snapshotRepository>
132    <repository><id>ossrh</id>
133      <url>https://s01.oss.sonatype.org/service/local/staging/
134      deploy/maven2</url>
135    </repository>
136  </distributionManagement>
137 </project>

```

Listing E.1: IoT Core Package POM

Appendix F

Performance Tests Specification

```

1 // imports
2
3 export const options = {
4   setupTimeout: "2m",
5   scenarios: {
6     subscribe: {
7       executor: "shared-iterations",
8       startTime: "0s",
9       vus: 1,
10      iterations: 1,
11      maxDuration: "3m",
12      exec: "subscribe",
13    },
14    ingestion: {
15      executor: "per-vu-iterations",
16      vus: 100,
17      iterations: 100,
18      startTime: "5s",
19      exec: "ingestion",
20      maxDuration: "3m",
21    },
22    consumption: {
23      executor: "shared-iterations",
24      startTime: "3m",
25      vus: 1,
26      iterations: 1,
27      maxDuration: "10s",
28      exec: "consumption",
29    },
30  },
31};
32
33 const timeLapseTrend = new Trend("time_lapse");
34
35 const sampleSize = new SharedArray("sampleSize", function () {
36   const sampleSize = [];
37   sampleSize.push(options.scenarios.ingestion.vus *
38     options.scenarios.ingestion.iterations
39   );
40   return sampleSize;
41});
42
43 const dataIds = new SharedArray("dataIds", function () {
44   const dataIds = [];
45   const numberOfWorkUnits = options.scenarios.ingestion.vus *
46     (options.scenarios.ingestion.iterations + 2);

```

```

47     for (let index = 0; index < numberOfDataUnits; index++)
48         dataIds.push(randomId());
49     return dataIds;
50 });
51
52 const data = new SharedArray("data", function () {
53     const data = [];
54     const total = options.scenarios.ingestion.vus + 2;
55     for (let index = 0; index < total; index++)
56         data.push(createDevice("em300th", index, true));
57     return data;
58 });
59
60 export function subscribe() {
61     const res = http.post(`http://${__ENV.SENSAE_INSTANCE_IP}:8086/
62     ↪ graphql`, anonymousLoginQuery, {
63         headers: { "Content-Type": "application/json" },
64     });
65
66     let received = [];
67     ws.connect(
68         `ws://${__ENV.SENSAE_INSTANCE_IP}:8801/subscriptions`,
69         {
70             headers: {
71                 "Sec-WebSocket-Protocol": "graphql-transport-ws",
72             },
73             (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.
92                             ↪ token,
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) * exec.vu.iterationInScenario];
117
118   sleep(device.interval);
119   const res = http.post(
120     'https://$_ENV.SENSAE_INSTANCE_IP}:8443/sensor-data/${device.
121     ↵ 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 F.1: Smart Irrigation Performance Test Scenario Description

Appendix G

Performance Tests Analysis

```

1 ## Import Libraries
2
3 prepare <- function(path) {
4   data <- read.csv(path)
5   data <- data[data$metric_name == 'time_lapse',]
6   data <- data[c('timestamp', 'metric_value', 'extra_tags')]
7   data$received <- data$timestamp - min(data$timestamp)
8   data$metric_value <- data$metric_value / 1000
9   data$sent_timestamp <- data$timestamp - data$metric_value
10  data$sent <- data$sent_timestamp - min(data$sent_timestamp)
11  data$iteration <- str_replace(data$extra_tags, 'iteration=' , '')
12  return(data)
13 }
14
15 create <- function(dataframe, xParam) {
16   ggplot(data=dataframe, mapping=aes(x=.data[[xParam]], y=metric_value,
17     ↪ col=iteration, label="")) +
18     geom_point(alpha = 1, stat = "unique") +
19     theme(legend.position = c(.9, .45)) +
20     xlab(paste("time data unit was", xParam, "(seconds)")) +
21     ylab("time taken to process data unit (seconds)")
22 }
23
24 outputTex <- function(pdot, path, xParam) {
25   tikz(file = path, width = 5, height = 3.3)
26   print(pdot)
27   dev.off()
28 }
29
30 process <- function(path) {
31   data <- prepare(paste(path, 'data.csv', sep = "/"))
32   pdot_sent <- create(data, "sent")
33   pdot_received <- create(data, "received")
34   outputTex(pdot_sent, paste(path, 'data_sent.tex', sep="/"))
35   outputTex(pdot_received, paste(path, 'data_received.tex', sep="/"))
36 }
37
38 processScenario <- function(path) {
39   scenario <- list.dirs(path)
40   scenario <- scenario[-1]
41   for (i in scenario) {
42     process(i)
43   }
44 }
45 processAll <- function() {

```

```
46 processScenario('scenario1')
47 processScenario('scenario2')
48 processScenario('scenario3')
49 }
50
51 setwd("/home/user/iot-project/project/k6/results")
52
53 processAll()
```

Listing G.1: Analysis Script

Appendix H

Contexts - TODO

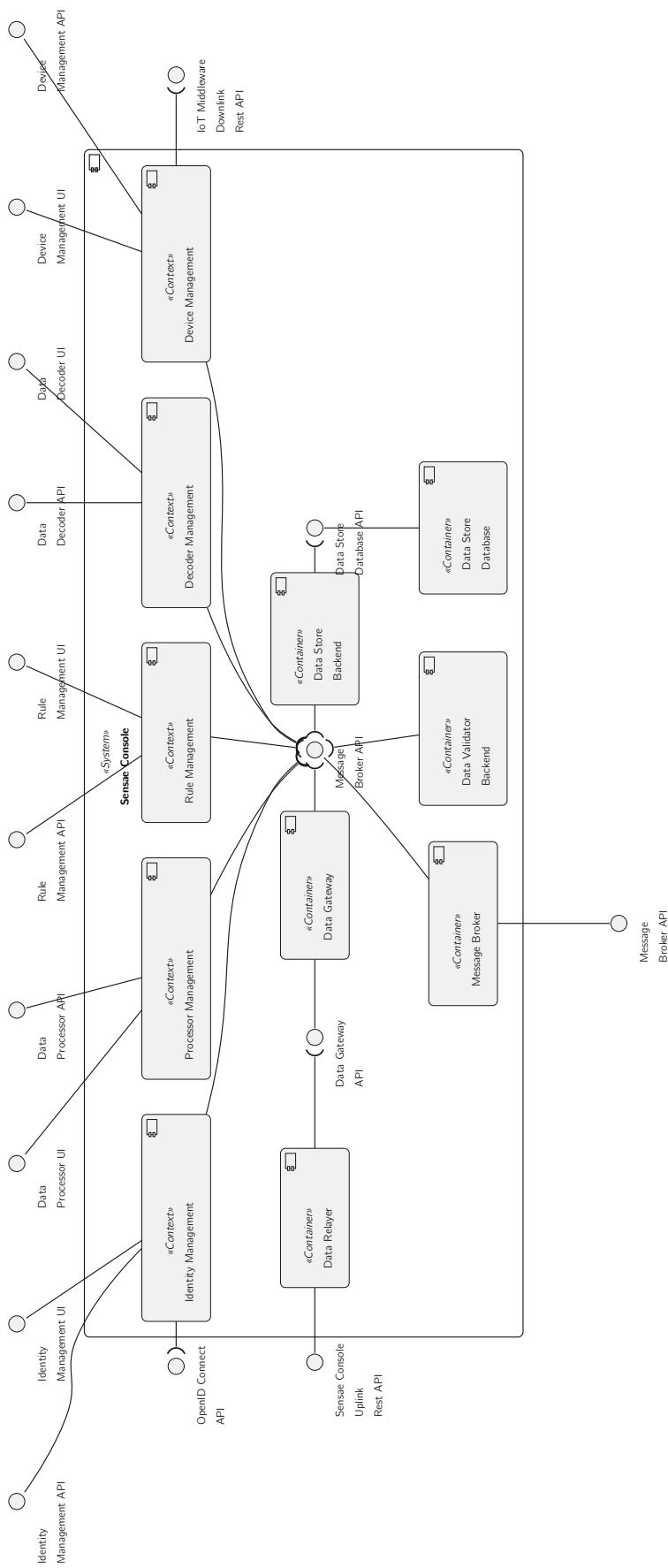


Figure H.1: Sensae Console - Context Level - Logical View Diagram

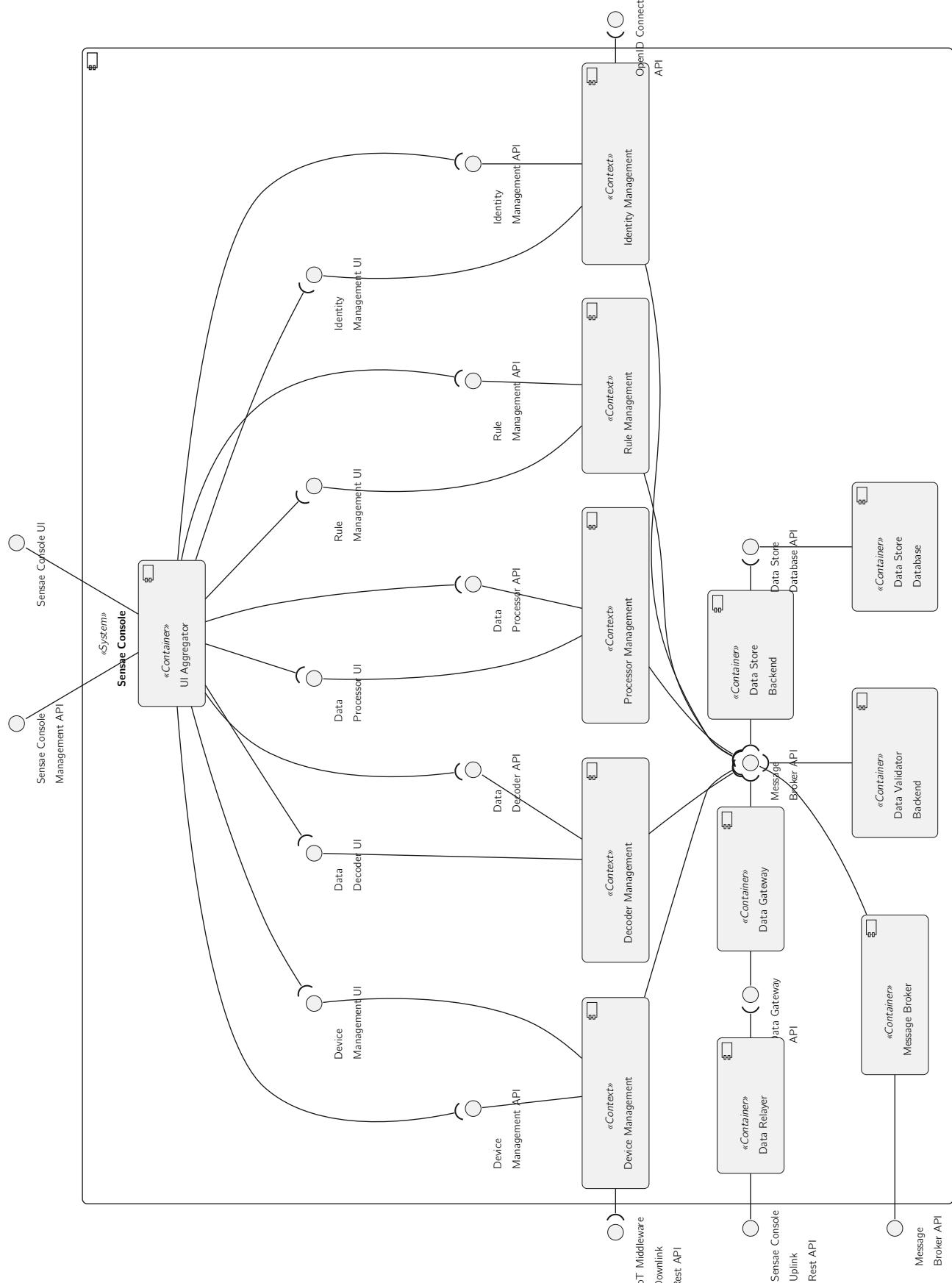


Figure H.2: Sensae Console - Context Level V2 - Logical View Diagram

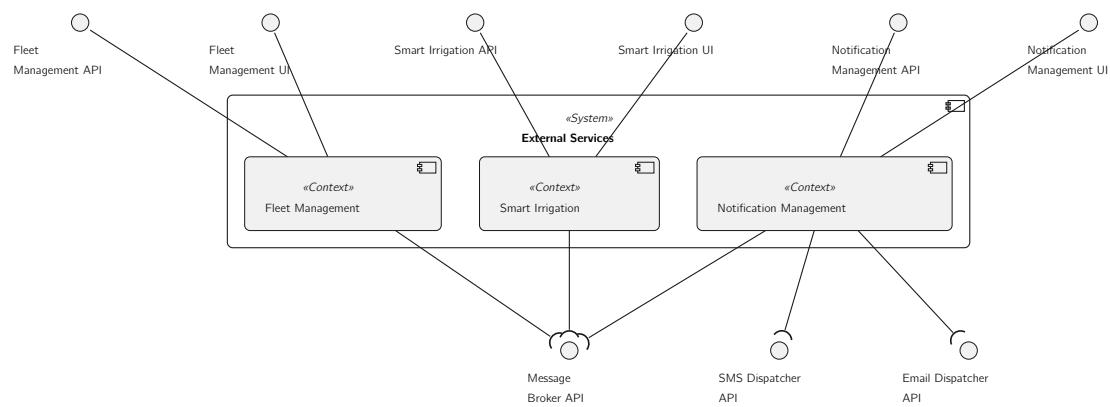


Figure H.3: External Services - Context Level - Logical View Diagram

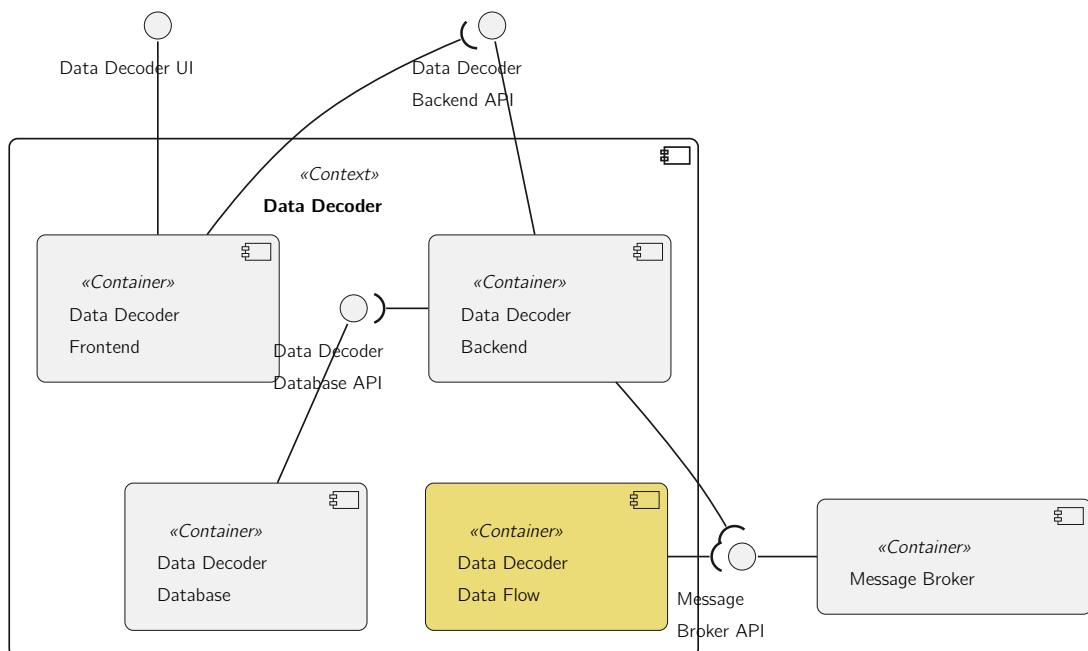


Figure H.4: Data Decoder - Container Level - Logical View Diagram

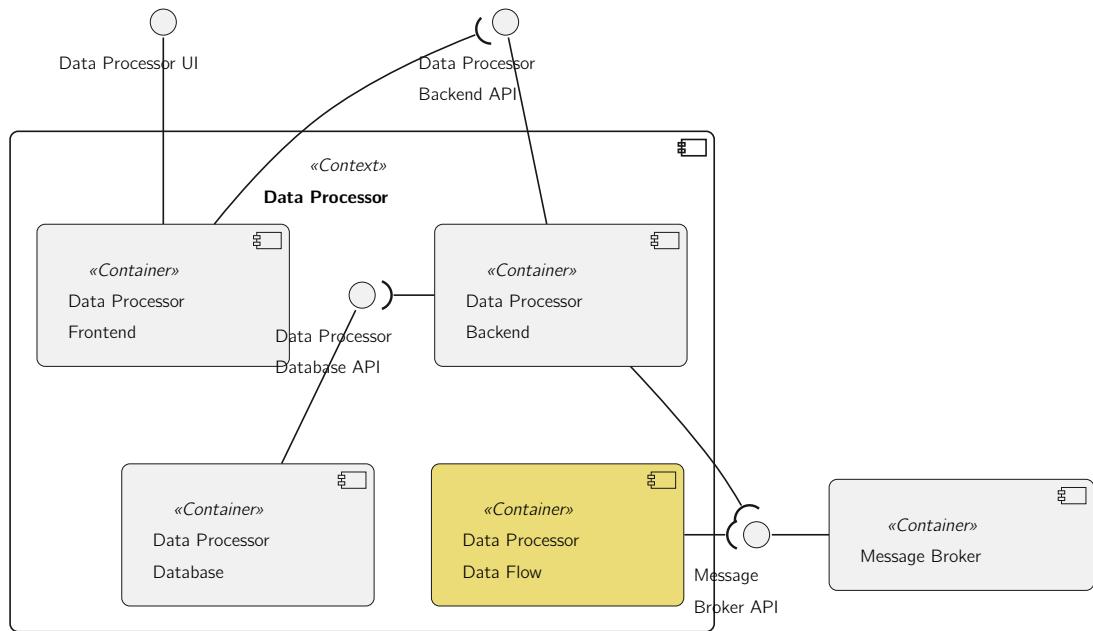


Figure H.5: Data Processor - Container Level - Logical View Diagram

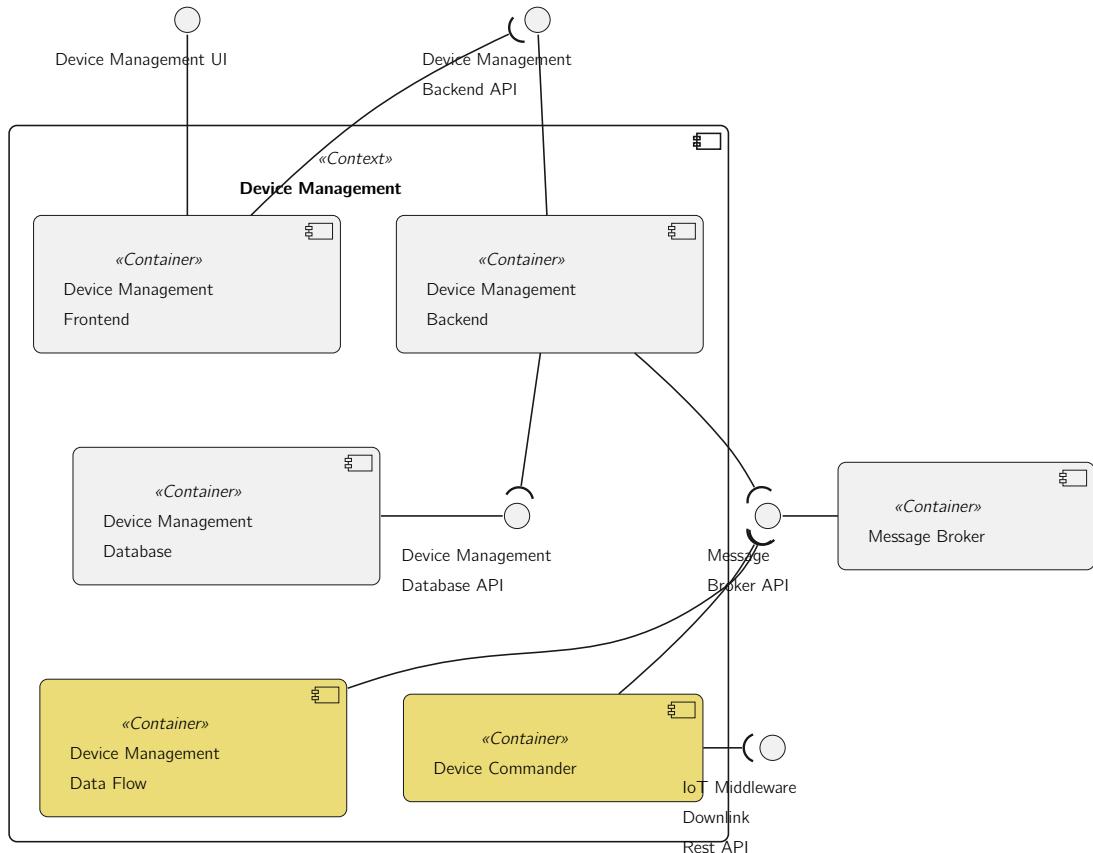


Figure H.6: Device Management - Container Level - Logical View Diagram

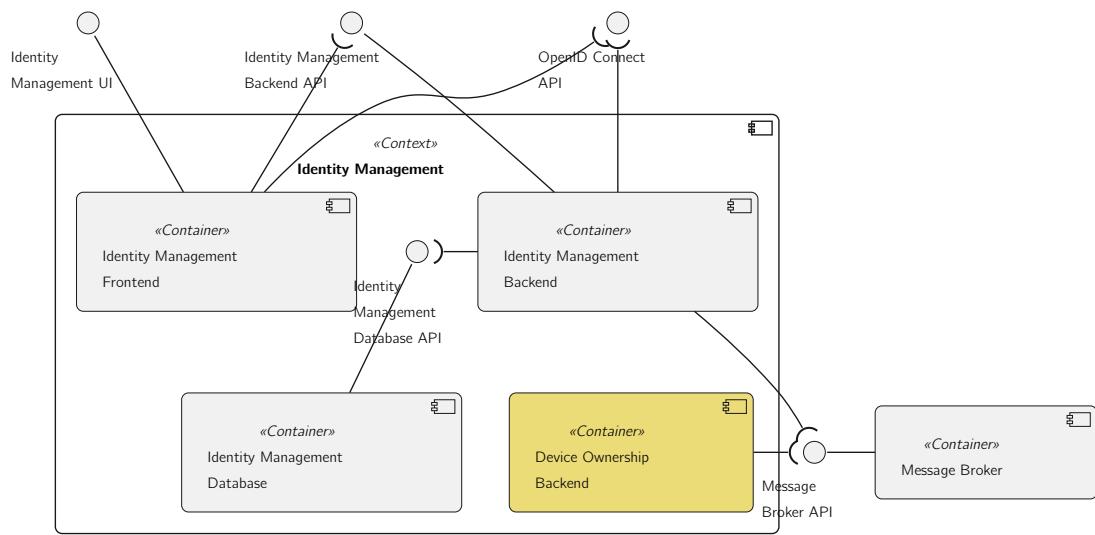


Figure H.7: Identity Management - Container Level - Logical View Diagram

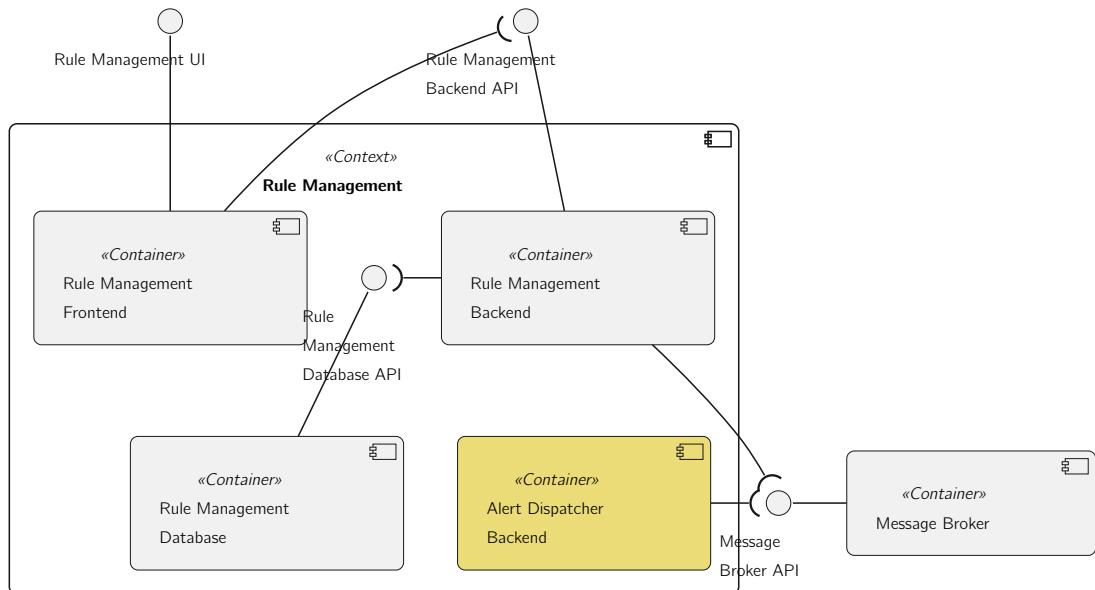


Figure H.8: Rule Management - Container Level - Logical View Diagram

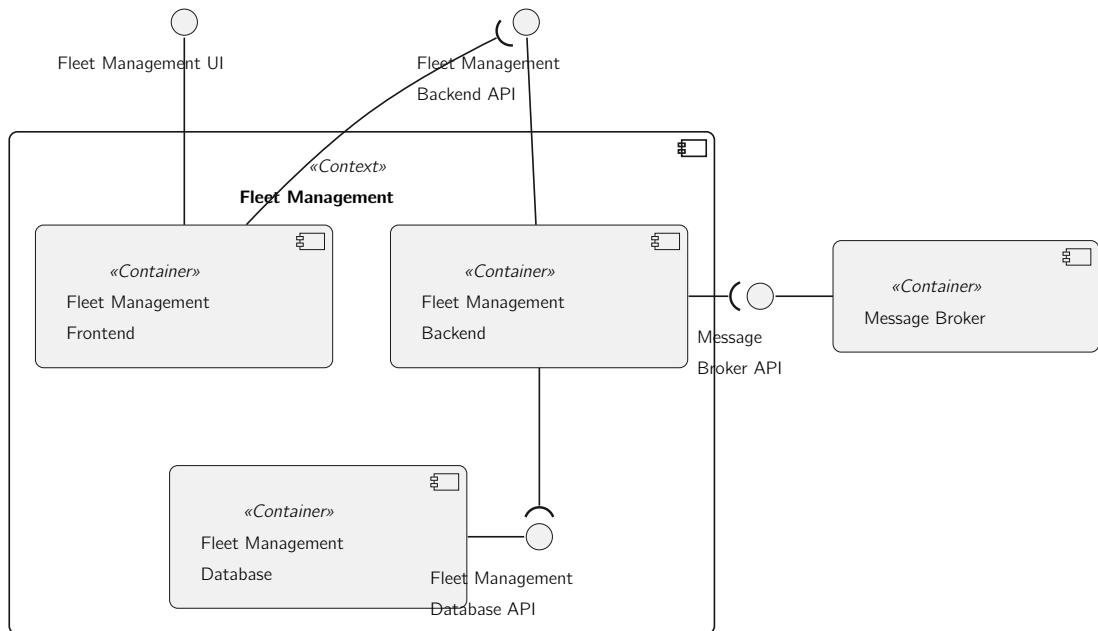


Figure H.9: Fleet Management - Container Level - Logical View Diagram

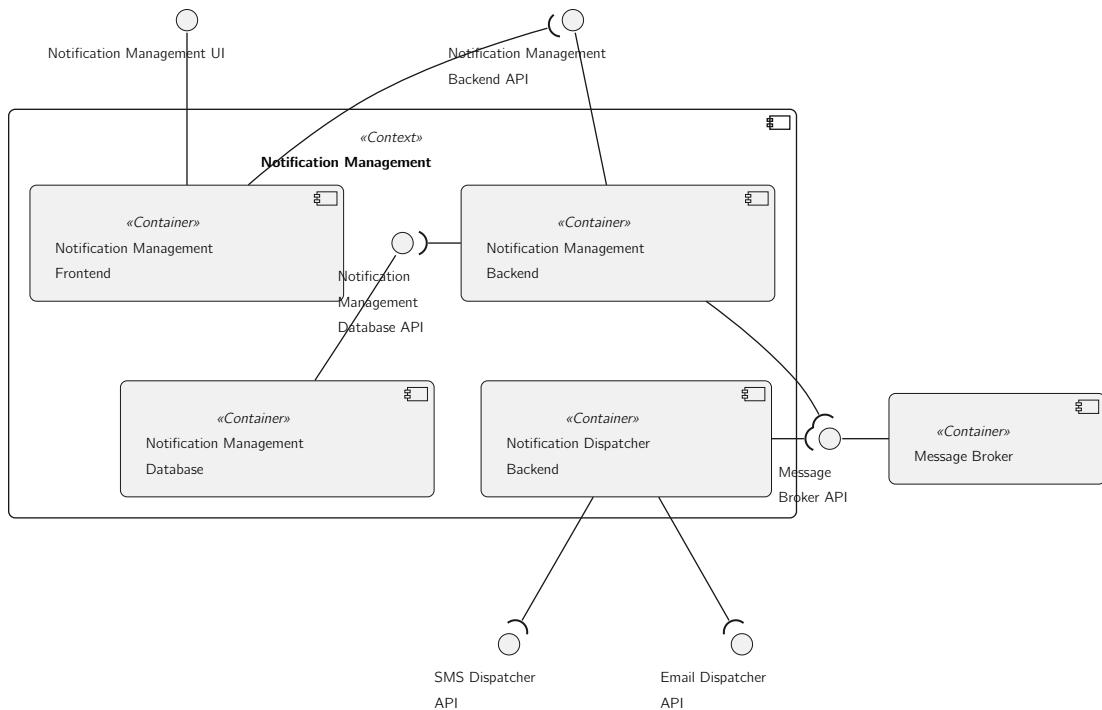


Figure H.10: Notification Management - Container Level - Logical View Diagram

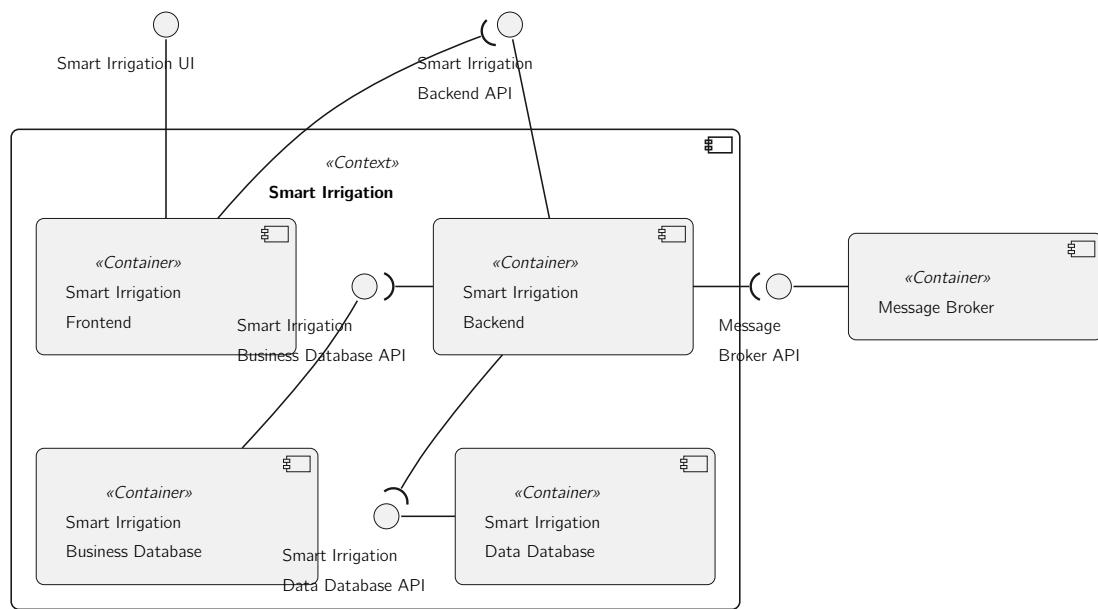


Figure H.11: Smart Irrigation Management - Container Level - Logical View Diagram