Instituto Superior de Engenharia do Porto

Smart IoT Service Builder Platform

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

Porto, September 8, 2022

Dedicatory

TODO

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 evaluation by many organizations. Some examples of intelligent connected sensors are: GPS asset tracking, parking spots, refrigerator thermostats, soil condition and many others. The limit of different objects that could become intelligent sensors is limited only by our imagination. But this devices are mostly useless without a platform to analyze, store and present the aggregated data.

Recently, several platforms have emerged to address this need and help companies/governments to increase efficiency, cut on operational costs and improve safety. Sadly, most of this platforms are tailor made for the devices that the company offers. This dissertation presents a platform and its development that assembles multiple services related to IoT into a single application. All the services provided by this platform attempt to be sensor-neutral and are to be exhibited under the same unified application.

Keywords: Internet of Things, Stream Processing, Big Data, Configurability, Real Time Systems

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 interacção entre milhões de dispositivos inteligentes através da Internet. Estes dispositivos e sensores conectados recolhem e disponibilizam dados para uso e avaliação por parte de muitas organizações. Alguns exemplos de sensores inteligentes e seus usos são: dispositivos GPS para rastreamento de activos, monitorização de vagas de estacionamento, termostatos em arcas frigoríficas, condição do solo e muitos outros. O número de diferentes objectos 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 que permite a criação e agregação de vários serviços relacionados com IoT num ambiente único. Todos os serviços fornecidos por esta plataforma procuram ser agnósticos em relação aos dispositivos inteligentes suportados.

Acknowledgement

TODO

Contents

List of Figures

List of Tables

List of Algorithms

List of Source Code

Chapter 1

Introduction

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

4.1 Functional Requirements

4.2 Non Functional Requirements

4.3 Synopsis

Chapter 5

Design

This section goal is to describe the overall system design to the reader. First the various system scopes will be introduced, followed by a section regarding the domain model. After this the system's architectural design will be presented and major decisions/alternatives discussed. At last, a synopsis of this chapter can be read.

5.1 System Scopes

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

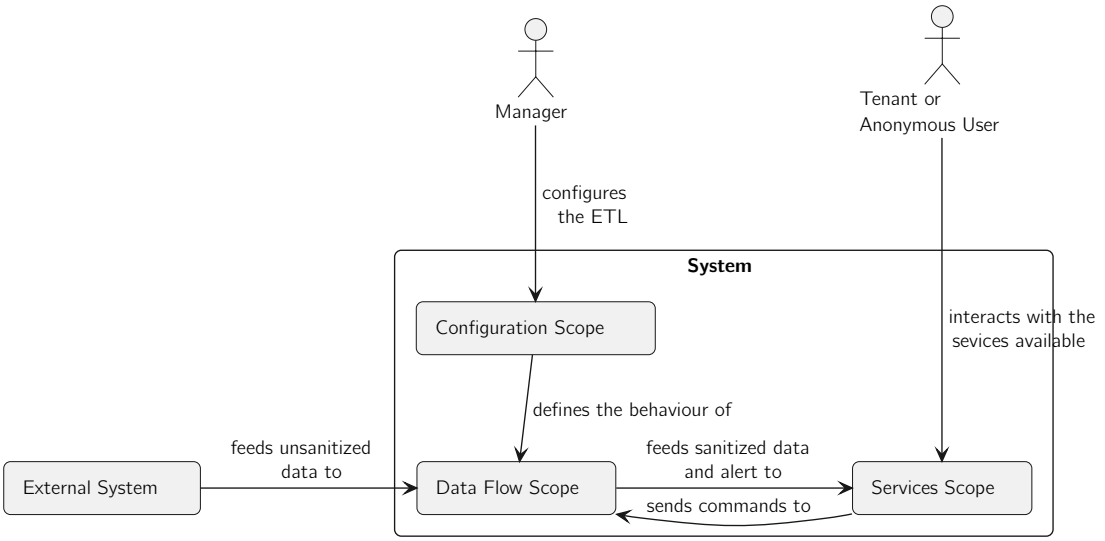

Figure 5.1: System Scopes

The **Configuration Scope** adheres to the configuration and visualization of internal processes/contexts. This processes, such as: (i) data decoders, (ii) data mappers (iii) device inventory, (iv) warning rules definition and (v) device ownership, are related to the **Data Flow Scope**. It is also possible to manage users' access and permissions in this scope.

The **Data Flow Scope** behaves according to what is defined in the **Configuration Scope** and acts as a pipeline where raw device data goes though various stages till it is sanitized and ready to be supplied to the **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) warnings dispatching.

The **Services Scope** is comprised of services that present and act according to the sanitized data that was supplied to them. This services applicability range from (i) smart irrigation, (ii) fleet management, (iii) fire detention, (iv) physical security access monitoring, (v) air quality monitoring and anything else deemed interesting.

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.

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, apart from the data validation and data store contexts (only present in this scope).

The data validation context preforms basic data filtering based on static rules, e.g. battery percentage reported has to be in between 0 and 100.

The data store context persists data captured in a defined and static state.

This scope applies changes to the device data that flows 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 processes 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.

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

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

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.

5.1.3 Service Scope

The **Service Scope** is responsible for presenting Internet of Things (IoT) business cases to end users. This scope is comprised of services that consume and publish data to **Data Flow Scope**. Currently, as a Minimum Value Product (MVP) the following business cases implemented 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 detailed in Sections 5.2.1 and 5.2.2.

5.1.4 Synopsis

This section introduces the system as three separated scopes each with different needs and responsibilities. Despite this they all have a common domain model. The Section 5.2 addresses this shared domain and each context peculiarity.

5.2 Domain

This system's domain model will be discussed here. The idea behind this section is to introduced core business concepts to the reader and explain how they map to the contexts present in the system. To represent this ideas the Unified Modeling Language (UML) notation is used.

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

5.2.1 Taxonomy

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

- **Device**: A device is a "Thing" that can collect data and submit it to **Sensae Console** via an external system though **Uplink**s. A device can, optionally, receive **Downlink**s;
- **Controller**: A controller is a **Device** that controls and aggregates data from various **Device**s;
- **Records/Metadata**: Records, or Metadata are labels associated to a **Device** that help an organization to classify and add some context to the owned **Device**s;

- **Downlink**: A downlink is a term commonly used in radio communications to denote the transmission from the network to the end user. In this case the network is the **Sensae Console** and the end user is a **Device**;
- **Uplink**: An uplink is the opposite of a **Downlink**, it's the transmission from a **Device** to the **Sensae Console**;
- **Data Unit**: A device data or measure is the collected data that is 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;
- **Device Command**: A device command is an abstraction on top of a **Downlink** intended to order a **Device** to execute a specific action. As an example, one could send a command to open or close a valve that is incorporated into a **Controller**;
- **Decoder**: A decoder is a function that translates a **Data Unit** into something that **Sensae Console** understands;
- **Domain**: A domain represents a department in a organization. An organization is composed of several domains structured in a tree like format;
- **Tenant**: A tenant is a user that belongs to one or more **Domain**s;
- **Alert/Warning**: A report about a detected condition based on the gather **Data Unit**;
- **Topic**: A Topic is a subcategory of the type of contents that are traded between the various containers in the system.

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

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

This concepts are referenced across the document.

5.2.2 Shared Model

The shared model is comprised of concepts that transverse the entire **Sensae Console** business model. Therefore, it is built as a separated project, *iot-core*, that can be imported in each micro service.

The intent behind this Shared Model is to alleviate one of the issues related to distributed systems - heterogeneity in data formats (Nadiminti, De Assunçao, and Buyya 2006) - and to provide a simple Software Development Kit (SDK) for third-parties to develop new services that interact with the **Sensae Console**. It can be seen as an explicit schema. According to **??**, "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."

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

Data Model

The data model represents the **Data Unit** that **Sensae Console** is currently capable of understanding. The following diagram, Figure 5.2, introduces a high level specification of it.

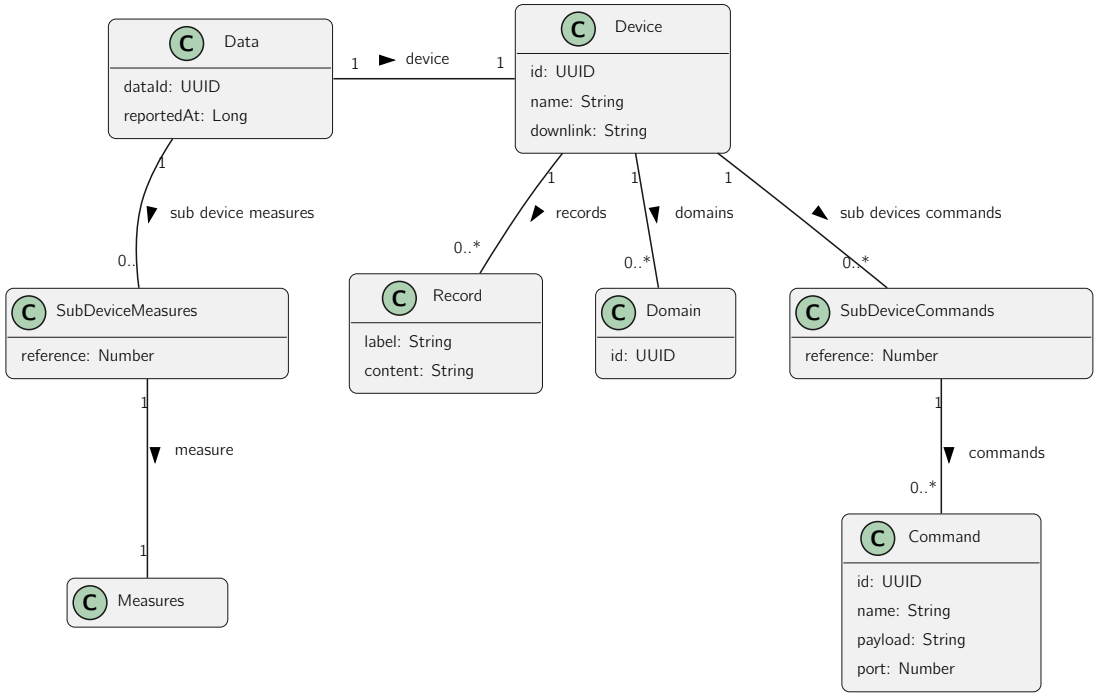

Figure 5.2: Shared Model

As a brief description:

- **Data Unit** is represented in the diagram as *Data* and is the entry point to the shared model;
- The *reportedAt* field represents the unix timestamp when the **Data Unit** was captured, in milliseconds;
- The *Device* concept represents the **Device** that sent the *Data*, and therefore the *Data*;
- The *Record* concept represents an entry of **Records/Metadata**;
- The *Domain* concept references the **Domain** that owns the *Device*;
- The *SubDeviceMeasures* concept introduces an approach to handle **Controller**s by mapping readings captured by a sub device to a *reference* that can later be resolved;
- The *SubDeviceCommands* concept introduces an approach to handle **Controller**s by mapping commands tailored for a sub device to a *reference* that can later be resolved;

- 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 5.2.

Data Type *Property*	*Sub Property*	**Description**	**Unit**
GPS		Point reference in the Geographic Coordinate System	
gps	*latitude*	Value between -90 and 90 measured in	degrees
	longitude	Value between -180 and 180 measured in	degrees
	altitude	Value determined according to the mean sea level	meters
Motion		Status related to the motion of a device	
motion	*value*	Value can be "ACTIVE", "INACTIVE" or "UNKNOWN"	n.a.
Velocity		How fast a device is moving	
velocity	*kilometerPerHour*	Value measured in	km/h
Temperature		Temperature measured by a device	
temperature	*celsius*	Value measured in	celsius
AQI		Air Quality Index according to the U.S. AQI	
aqi	*value*	Value measured in	AQI
Air Humidity		Concentration of water vapour present in the air	
airHumidity	*gramsPerCubicMeter*	Value measured in	g/m3
	relativePercentage	Value measured in	%
Air Pressure		Pressure within the atmosphere of Earth	
airPressure	*hectoPascal*	Value measured in	hPa
Battery		Battery of the device	
battery	*volts*	Value measured in	volts
	percentage	Value measured in	%
	maxVolts	Minimum volts the battery needs for the device to work	volts
	minVolts	Maximum volts the battery can hold	volts
Soil Moisture		Amount of water, including water vapor, in an unsaturated soil	
soilMoisture	*relativePercentage*	Value measured in	%
Illuminance		Illuminance level - luminous flux per unit area	
illuminance	*lux*	Value measured in	lux
Trigger		Type related to something with an on / off or open / close state	
trigger	*value*	Value true or false	boolean

Table 5.2 continued from previous page

Data Type *Property*	*Sub Property*	**Description**	**Unit**
CO2		Atmospheric Carbon Dioxide concentration	
co2	*ppm*	Value measured in	ppm
CO		Atmospheric Carbon Oxide concentration	
co	*ppm*	Value measured in	ppm
VOC		Volatile Organic Compounds concentration measured by a device	
voc	*ppm*	Value measured in	ppm
NH3		Atmospheric Ammonia concentration	
nh3	*ppm*	Value measured in	ppm
O3		Atmospheric Ozone concentration measured by a device	
o3	*ppm*	Value measured in	ppm
NO2		Atmospheric Nitrogen dioxide concentration	
no2	*ppm*	Value measured in	ppm
PM2.5		Particulate Matter in the air (size up to 2.5 micrometers)	
pm2_5	*microGramsPerCubicMeter*	Value measured in	μg/m3
PM10		Particulate Matter in the air (size up to 10 micrometers)	
pm10	*microGramsPerCubicMeter*	Value measured in	μg/m3
Water Pressure		Water Pressure measured in pipes by a device	
waterPressure	*bar*	Value measured in	bar
pH		Scale used to specify how acidic or basic a water-based solution is	
ph	*value*	Value between 0 and 14 measured in	pH
Occupation		Occupation percentage measured inside a vessel	
occupation	*percentage*	Value measured in	%
Soil Conductivity		Substances ability to conduct an electrical current in the soil	
soilConductivity	*microSiemensPerCentimeter*	Value measured in	μS/cm
Distance		Distance measured from the device to a surface	
	millimeters	Value measured in	mm
distance	*maxMillimeters*	Maximum distance the sensor can be to a given surface	mm
	minMillimeters	Minimum distance the sensor can be to a given surface	mm

Table 5.2 continued from previous page

Data Type *Property*	*Sub Property*	**Description**	**Unit**

Table 5.2: Measure Data Types

The current shared model schema can be found in Appendix A.

Message Envelop Model

The message envelop model refers to how, coupled with the routing model in Section 5.2.2, information can easily transverse the system. The message envelop is used when a message is expected to flow though the system and is therefore used in all **Topic**s but the **Internal** one.

The diagram present in Figure 5.3 details this model.

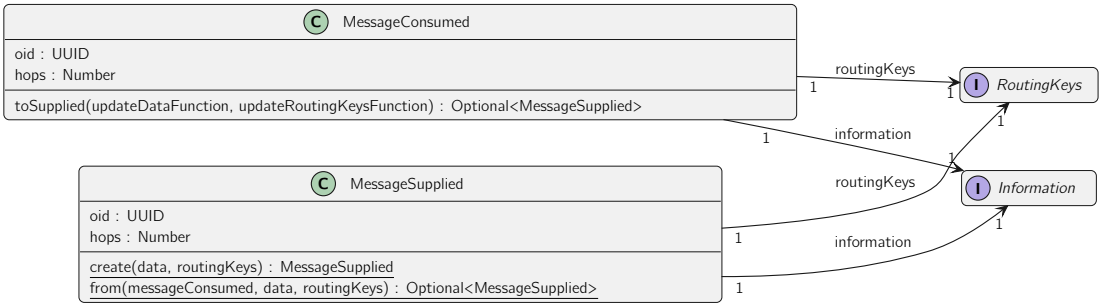

Figure 5.3: Message Envelop Model

As a brief description:

- A *MessageSupplied* is created in a container and supplied to start the flow of information in the system;
- A *MessageConsumed* is consumed by a container and can then be transformed into a *MessageSupplied* if needed;
- *Information* represents the content of the message;
- *RoutingKeys* represents the model referenced in Section 5.2.2;

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

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 *RoutingKey*s and publish information with *RoutingKey*s.

The diagram present in Figure 5.4 details this model.

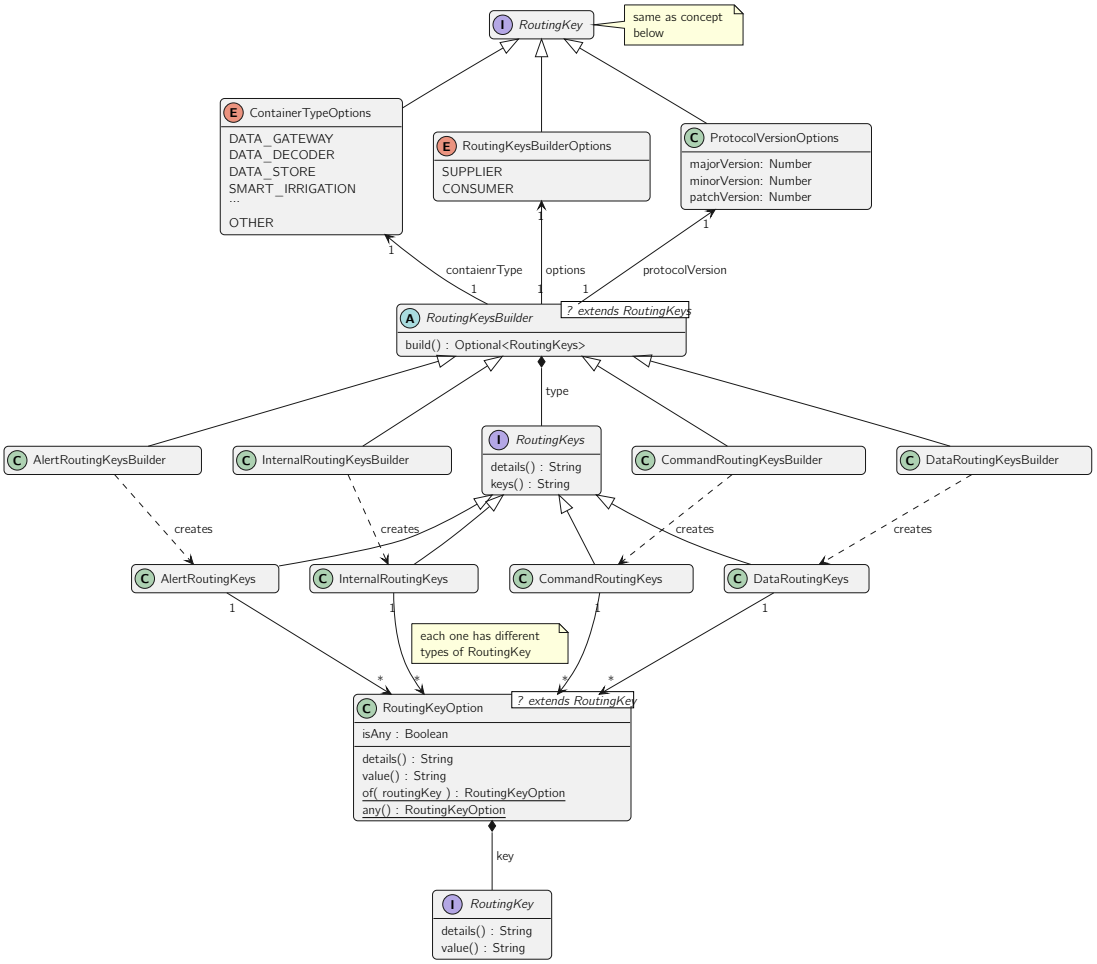

Figure 5.4: Routing Model

As a brief description:

- *RoutingKeys* is the concept referenced in Figure 5.3 and represents a collection of different *RoutingKeyOptions*;
- There are currently 4 types of *RoutingKeys*, one for each **Topic**;
- To ensure that the various containers in **Sensae Console** understand each other a *ProtocolVersionOptions* is provided. This concept follows the Semantic Versioning Specification 2.0 (Preston-Werner 2011) an is constructed 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 each *RoutingKey* must be specified, for containers that supply/publish messages.

In the Table 5.3 all currently used *RoutingKey* are presented.

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

Table 5.3 continued from previous page

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

Table 5.3: Routing Types

[1]has three possible values: (i) UNDETERMINED, (ii) WITH, (iii) WITHOUT
[2]related 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 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 **Service 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*.

5.2.3 Bounded Contexts

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 **Sensae Console** is composed by the following bounded contexts:

- In **Configuration/Data Flow Scopes**:

  - Data Processor;
  - Data Decoder;
  - Device Management;
  - Identity Management;
  - Rule Management.
- In **Service Scope**:
  - Smart Irrigation;
  - Fleet Management;
  - Notification Management;

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

Data Processor

The **Data Processor** context refers to simple data mappers that translate inbound information to **Data Unit**s, discussed in Section 5.2.2.

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

The diagram in Figure 5.5 displays the noteworthy concepts in this context.

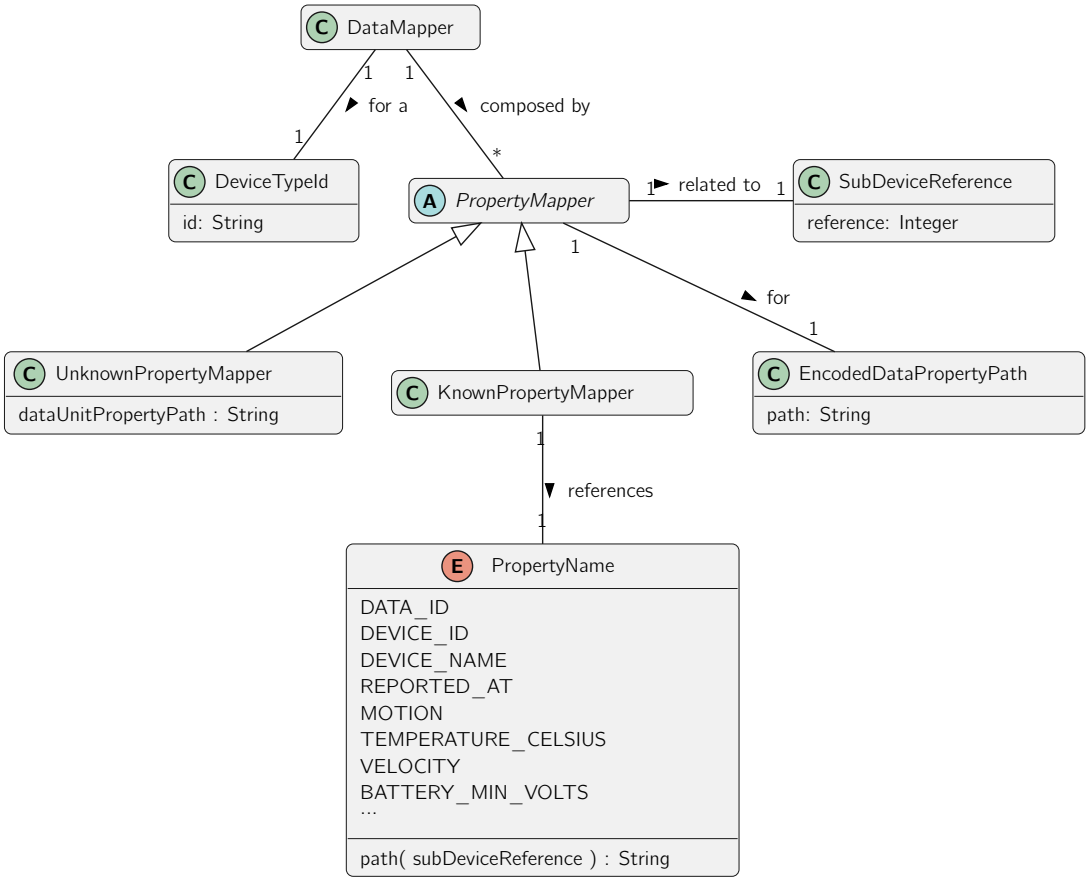

Figure 5.5: Data Processor Context Model

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** corresponds to the **Routing Key** *Device Type Options* mentioned in Table 5.3;
- **SubDeviceReference** represents a number that will be used later to reference a sub device when dealing with **Controller**s. For simple **Devices** the used and default value is *0*;
- **PropertyName** has much more properties that haven't been presented for brevity.

As an example, one could define an inbound information 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*.

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

```
        "data": [
            {
                "temperature": 18,
            }
        ]
    }
}
```

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.

Data Decoder

The **Data Decoder** context refers to a more complex data mapper that translates inbound information to **Data Unit**s, discussed in Section 5.2.2. It was created to deal with the limitations mentioned in Section 5.2.3.

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 5.6 displays the noteworthy concepts in this context.

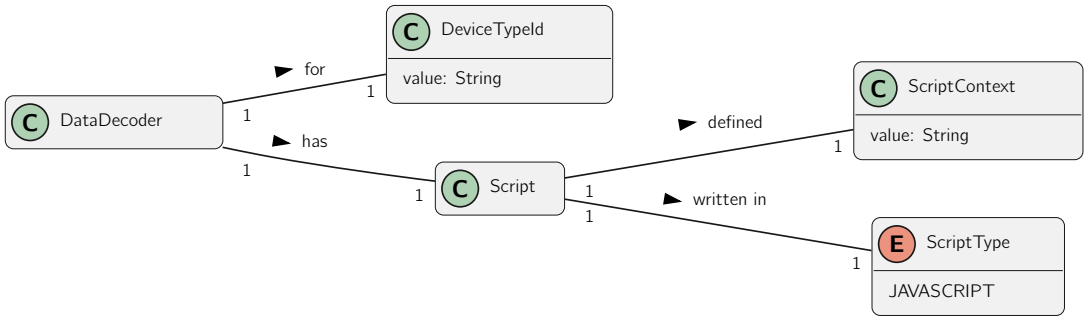

Figure 5.6: Data Decoder Context Model

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 programming language knowledge but is much more flexible than the **Data Processor** operation.

Device Management

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

The diagram in Figure 5.7 displays the noteworthy concepts in this context.

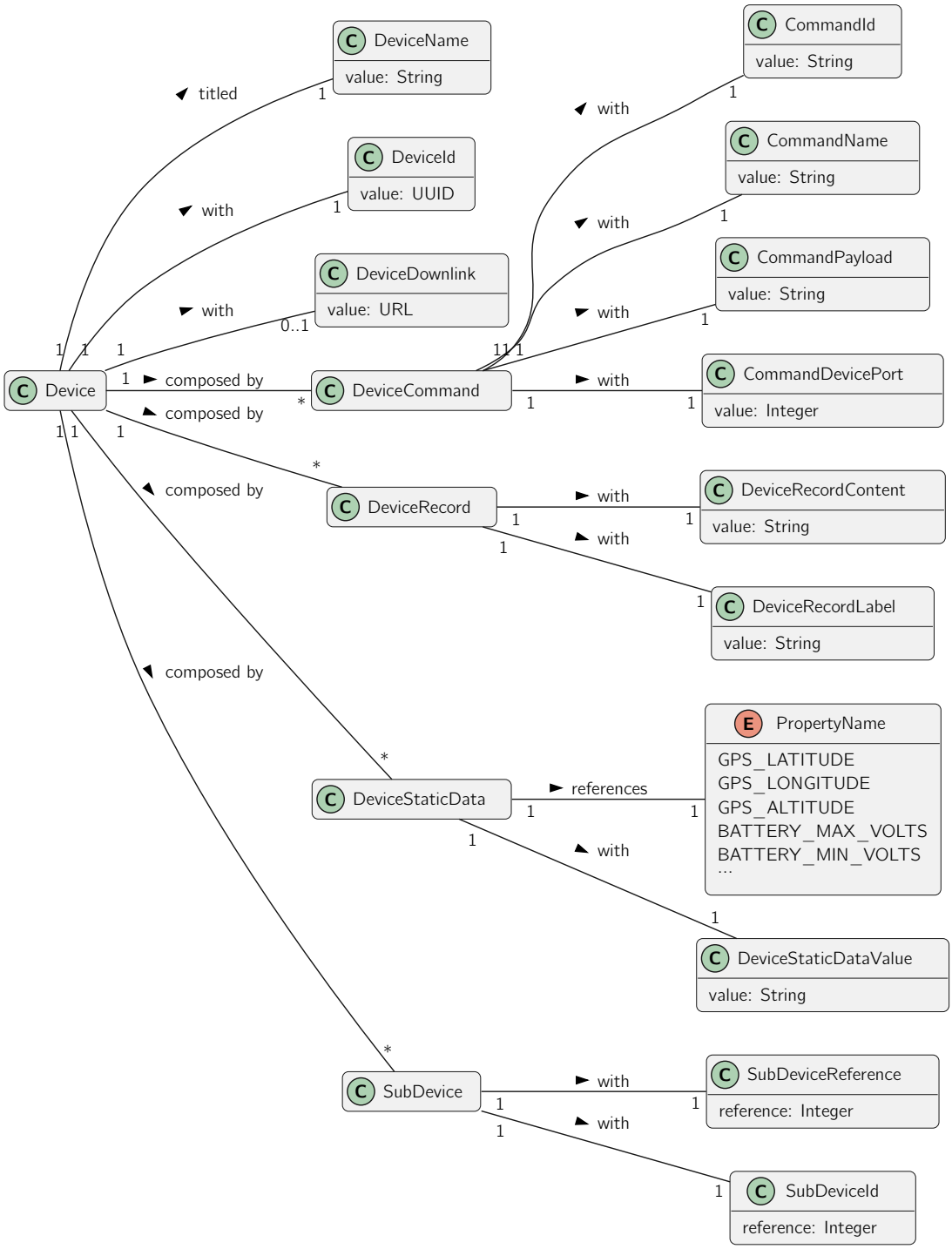

Figure 5.7: Device Management Context Model

As a brief description:

- A **Device** is uniquely identified by a **DeviceId**, has a **DeviceName** and may have a **DeviceDownlink**;
- 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 5.2 help to split a **Controller**'s **Data Unit** into various **Data Unit**, one for each referenced **SubDevice**.

Identity Management

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

The diagram in Figure 5.8 displays the noteworthy concepts in this context.

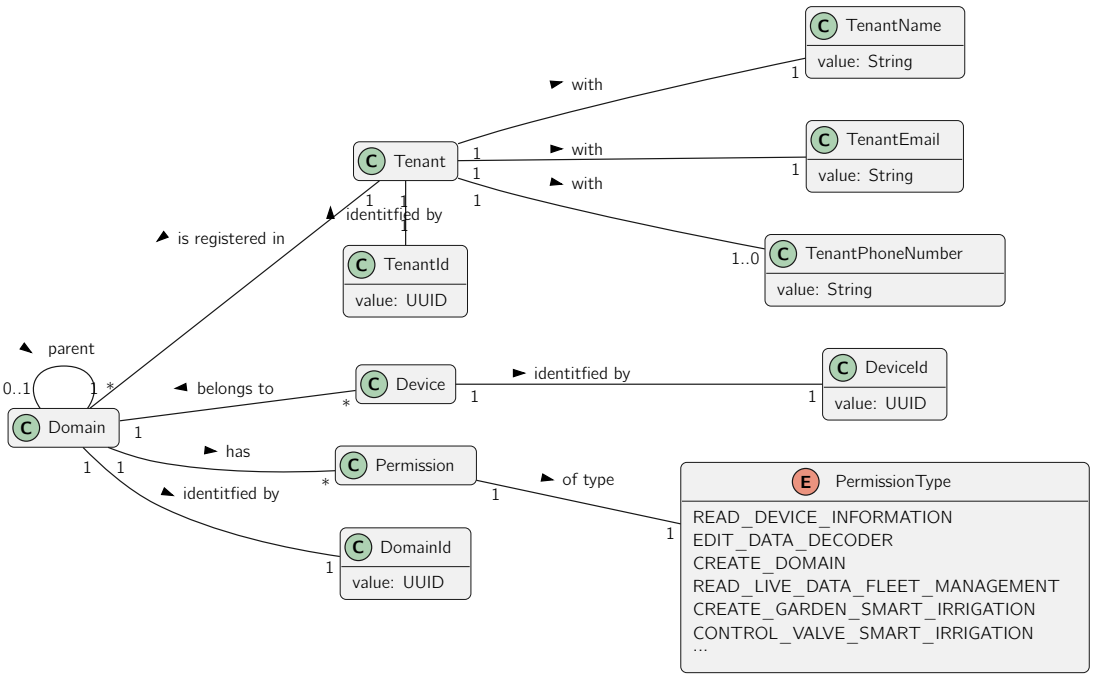

Figure 5.8: Identity Management Context Model

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 that haven't been presented for brevity.

A **Domain** represents a department in a hierarchical organization. An organization is composed by several domains in a tree like structure as presented in Figure 5.9.

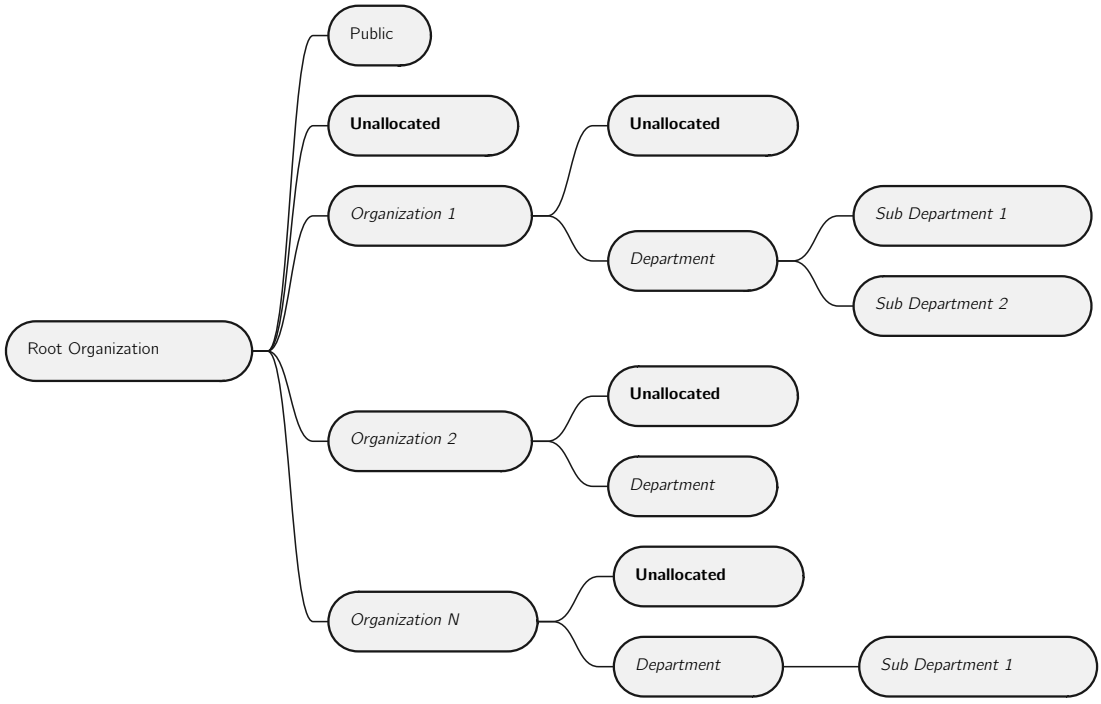

Figure 5.9: Domain Structure

Coupled with the figure above, there are other constrains:

- A domain owns all devices in it and in his subdomains;
- A domain can only inherit his parent domain permissions;
- A tenant has all the domain permissions that he is registered in;
- A tenant can only see the devices that the domains he 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 tenant account;

By default this context contains the *Root* domain, the *Root*'s *Unallocated* domain and the *Public* domain.

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 Unit**s and output **Alert**s base on them. This systems are usually categorized as Information Flow Processing (IFP) Systems, according to Cugola and Margara 2012.

The following diagram, Figure 5.10, represents this systems.

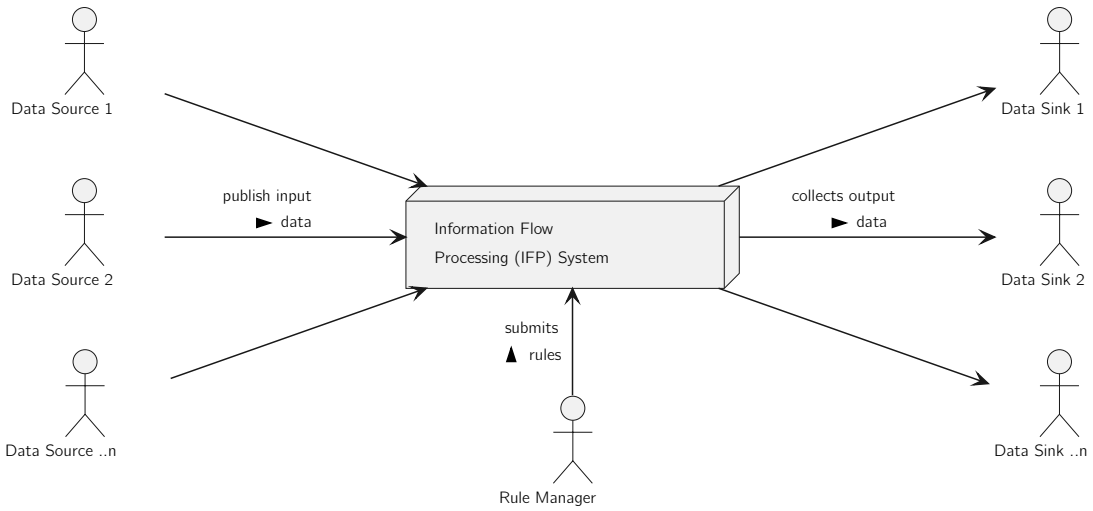

Figure 5.10: High-Level View of a IFP System

In this specific solution, the input data are **Data Unit**s and the output data are the **Alert**s. This context is concerned about how *rules* are defined, the diagram in Figure 5.11 displays the noteworthy concepts.

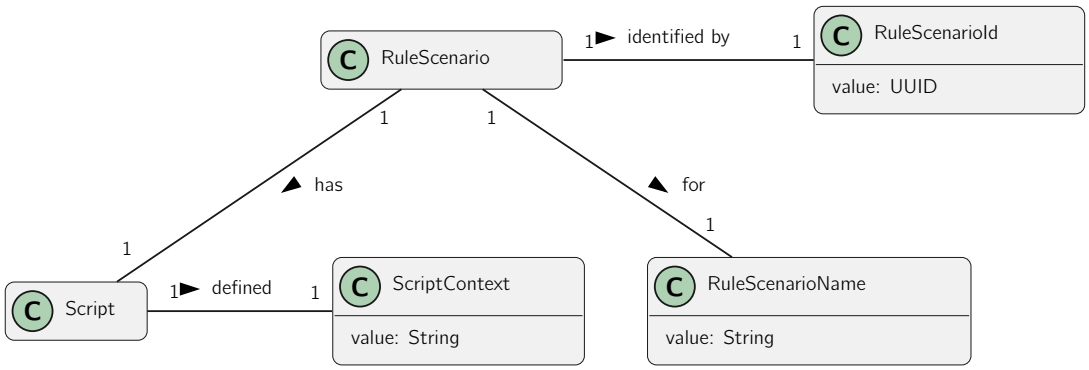

Figure 5.11: Rule Management Context Model

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 5.12 displays the noteworthy concepts in this context.

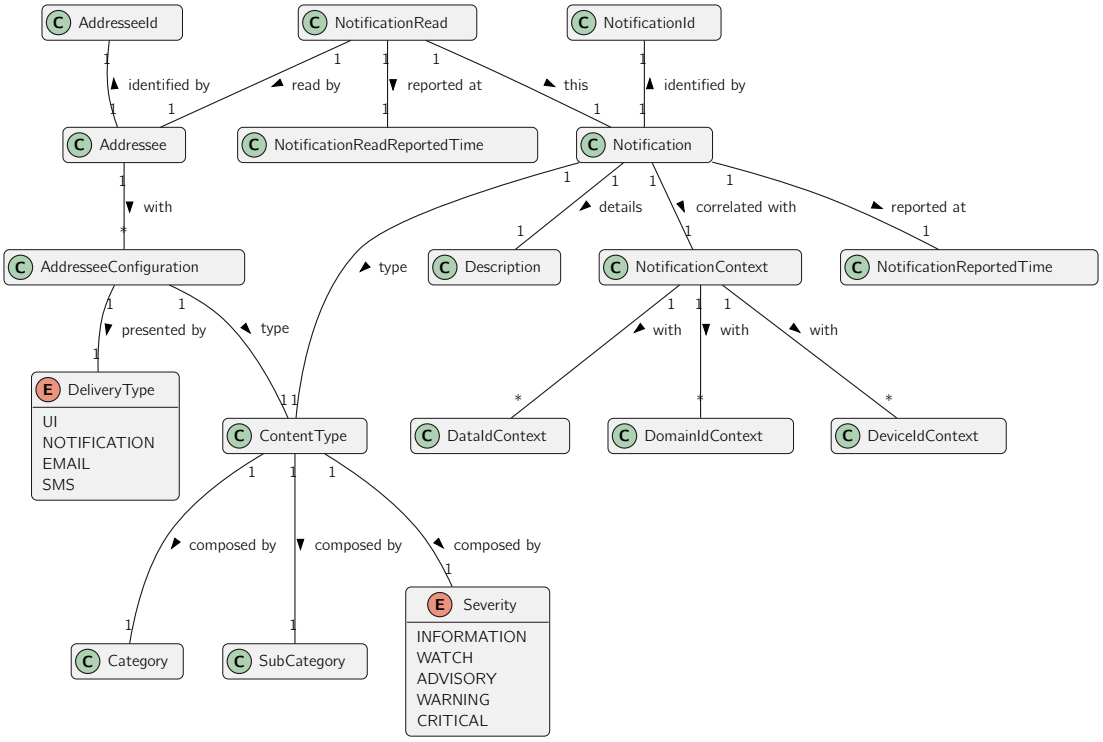

Figure 5.12: Notification Management Context Model

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 5.3 and defines the type of each **Notification**;
- A **NotificationContext** is data that can help to correlate the **Notification** with other contexts such as what devices - **DeviceIdContext** - were involved in the **Alert** trigger, or what domains - **DomainIdContext** - need to be notified, or what **Data Unit**s - **DataIdContext** - are related to the **Alert**;
- To enforce accountability in the system, the notion of who read a specific notification and when was added - **NotificationRead**.

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 5.13 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.

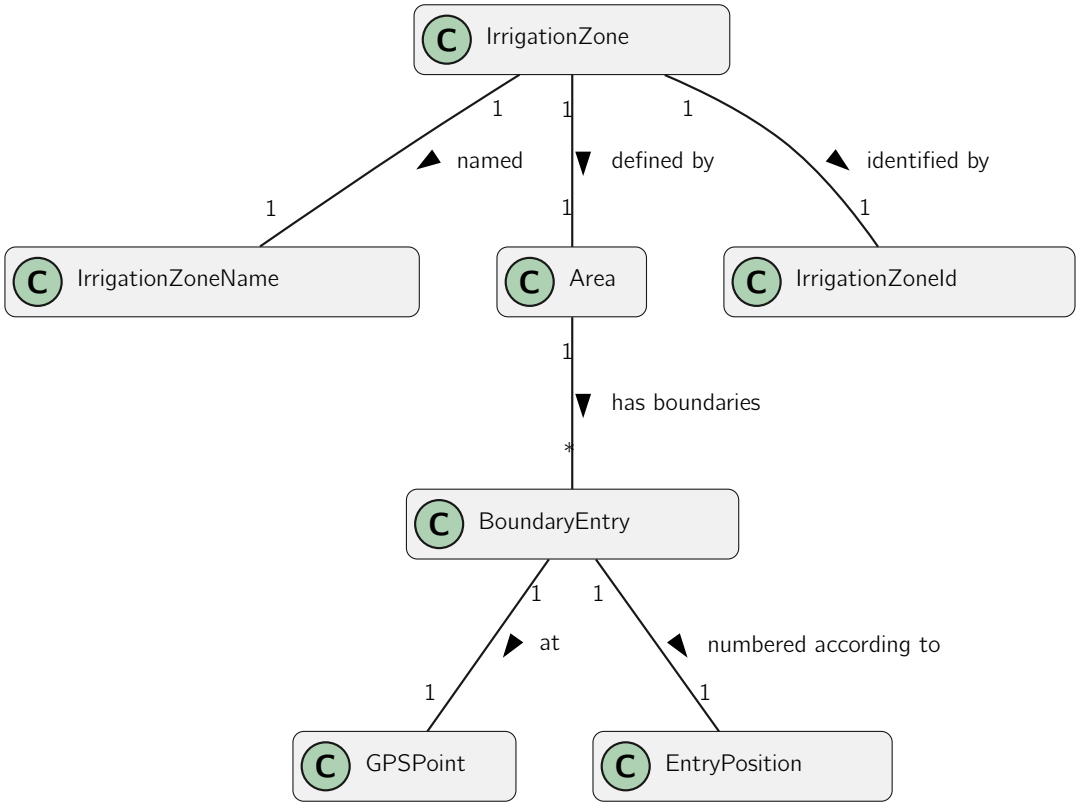

Figure 5.13: Smart Irrigation Context Model - Irrigation Zone

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

As presented in the following diagram, Figure 5.14, a sensor/valve can be represents by a **Device**.

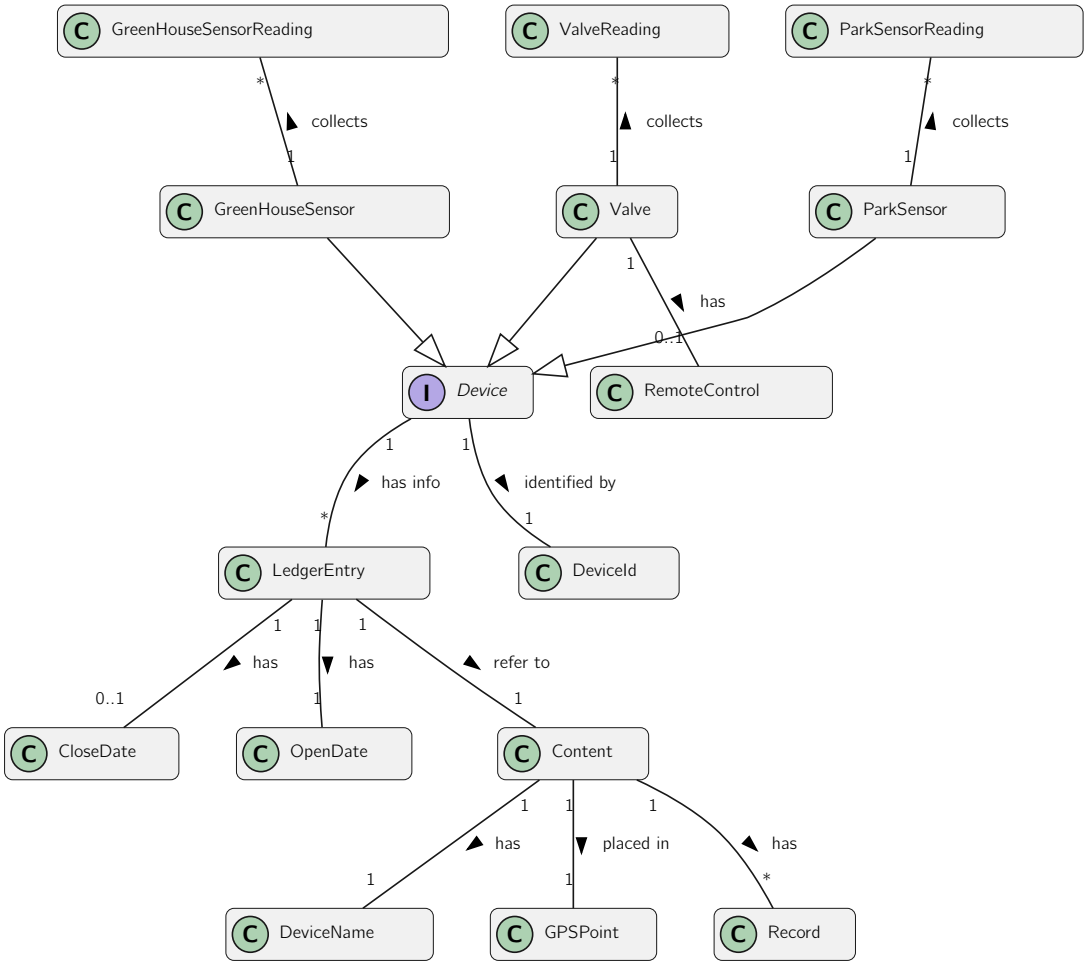

Figure 5.14: Smart Irrigation Context Model - Device

As a brief description:

- A **Valve** can be controlled remotely if two types of **Command**s are sent with the device **Data Unit**: *OpenValve* and *CloseValve*;
- A **Device** is identified by its **DeviceId**;
- Each **Device** stores an history of all its changes such as name, location or metadata in **Content**, the same **LedgerEntry** is used as long as this values don't change;
- There are three types of **Device**: (i) Green House Sensor, (ii) Park Sensor, (iii) Valve. Each of this types collect different measures discussed in Figure5.15.

As mentioned above each type of device collects different readings. The following diagram, Figure5.15, details this readings.

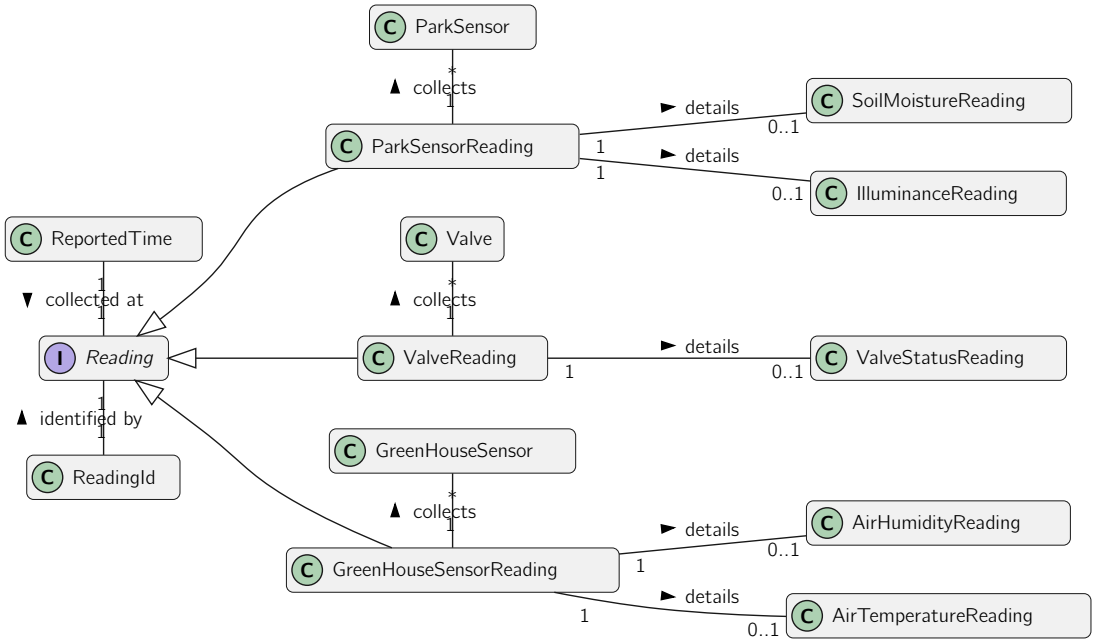

Figure 5.15: Smart Irrigation Context Model - Reading

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.

Fleet Management

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

The diagram in Figure 5.16 displays the noteworthy concepts related to this context.

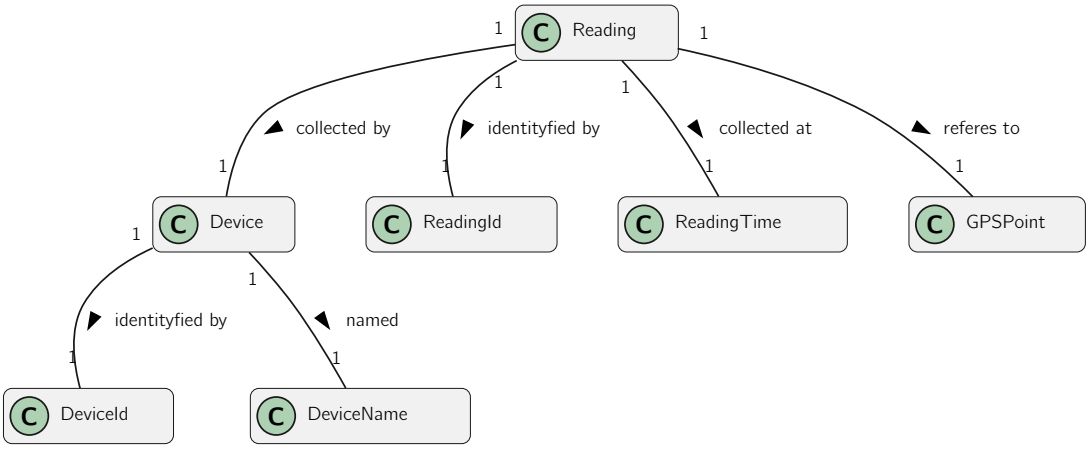

Figure 5.16: Fleet Management Context Model

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

5.2.4 Synopsis

In this section the various domains that **Sensae Console** incorporates are described. This domains share some concepts such as **Device** but it isn't clear how they interact with each other. In the next section - Architectural Design - it will be addressed how this domains are connected and cooperate.

5.3 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;
- **Development 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 presents 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 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.

5.3.1 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.

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 ***TODO***. This takes into account the interactions of the platform with external systems and its interaction with the various actors of the system (Figure 5.17).

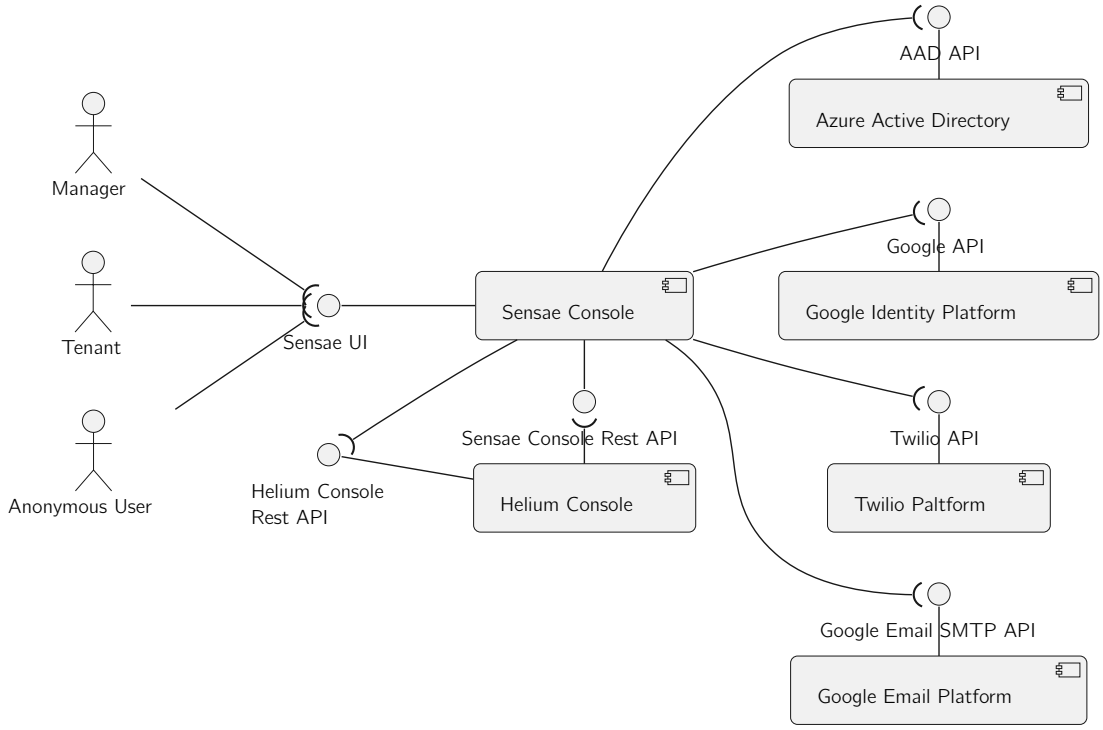

Figure 5.17: Context Level - Logical View Diagram

The external systems and its functions are as follows:

- **Helium Console**: Device data hub;
- **Azure Active Directory**: User authentication/identity;
- **Google Identity Platform**: User authentication/identity;
- **Twilio Platform**: SMS delivery;
- **Google Email Platform**: Email delivery.

The reason behind the use of external authentication/identity services is described in the Section 5.4.3.

Context Level - Development View

Next is the development view (Figure 5.18), intended to familiarize the reader with how the software is organized.

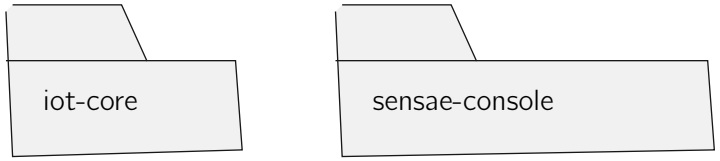

Figure 5.18: Context Level - Development View Diagram

The package *iot-core* contains the shared model discussed in the Section 5.2.2, and defines what type of information backend containers can subscribe to or publish (discussed in Section 5.2.1).

The package *sensae-console* contains software of the various containers needed to run the **Sensae Console**. As expected *iot-core* is a core dependency for the *sensae-console* backend containers.

Context Level - Physical View

Next is the physical view (Figure 5.19), intended to familiarize the reader with the environment where the solution runs.

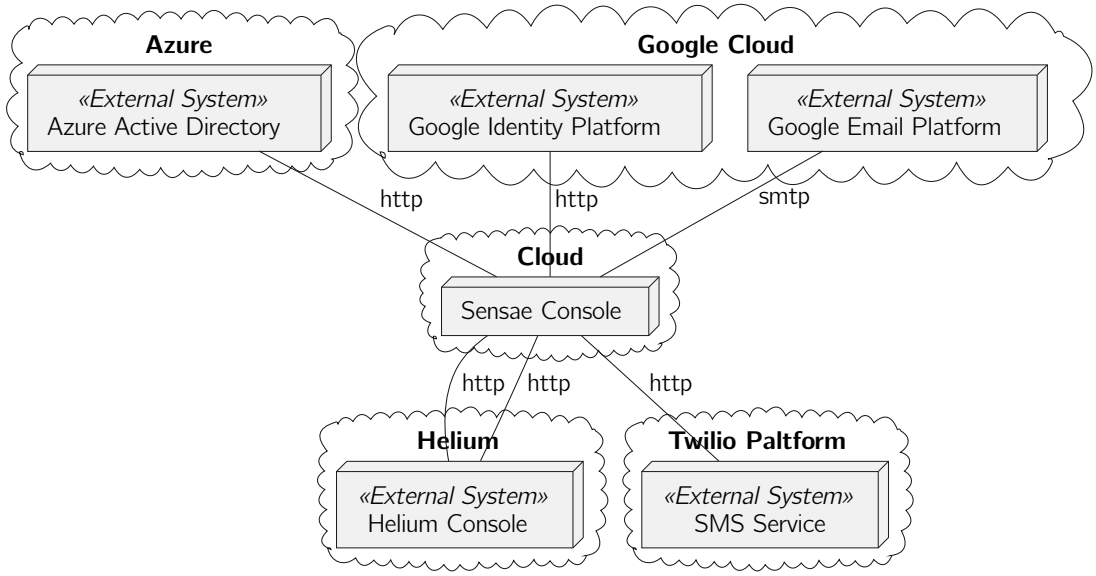

Figure 5.19: Context Level - Physical View Diagram

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 5.3.2 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.

5.3.2 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 5.4 other alternatives are discussed.

The Physical View will not be represented since the fundamental idea behind the idealized deployment of **Sensae Console** is already described in the Section Context Level - Physical View.

Container Level - Logical View

The description of this level of abstraction begins with a logical view of the containers that compose the system. Alternatives were also analyzed taking into account several requirements namely (i) configurability, (ii) maintainability, (iii) extensibility (iv) development cost and (v) scalability.

In order to support the functional requirements identified (*****TODO*****), 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 level of device data.

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 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 5.20. This scope is composed by the processes discussed in 5.1.1. Each process is composed by a three tier architecture, as per IBM 2020a:

- **Presentation Tier**: the user interface and communication tier of the application where the user interacts with the system;
- **Application Tier**: the business tier of the application where information from the **Presentation Tier** is processed and sent to the **Data Tier**;
- **Data Tier**: the infrastructure tier 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 tier architecture mention before.

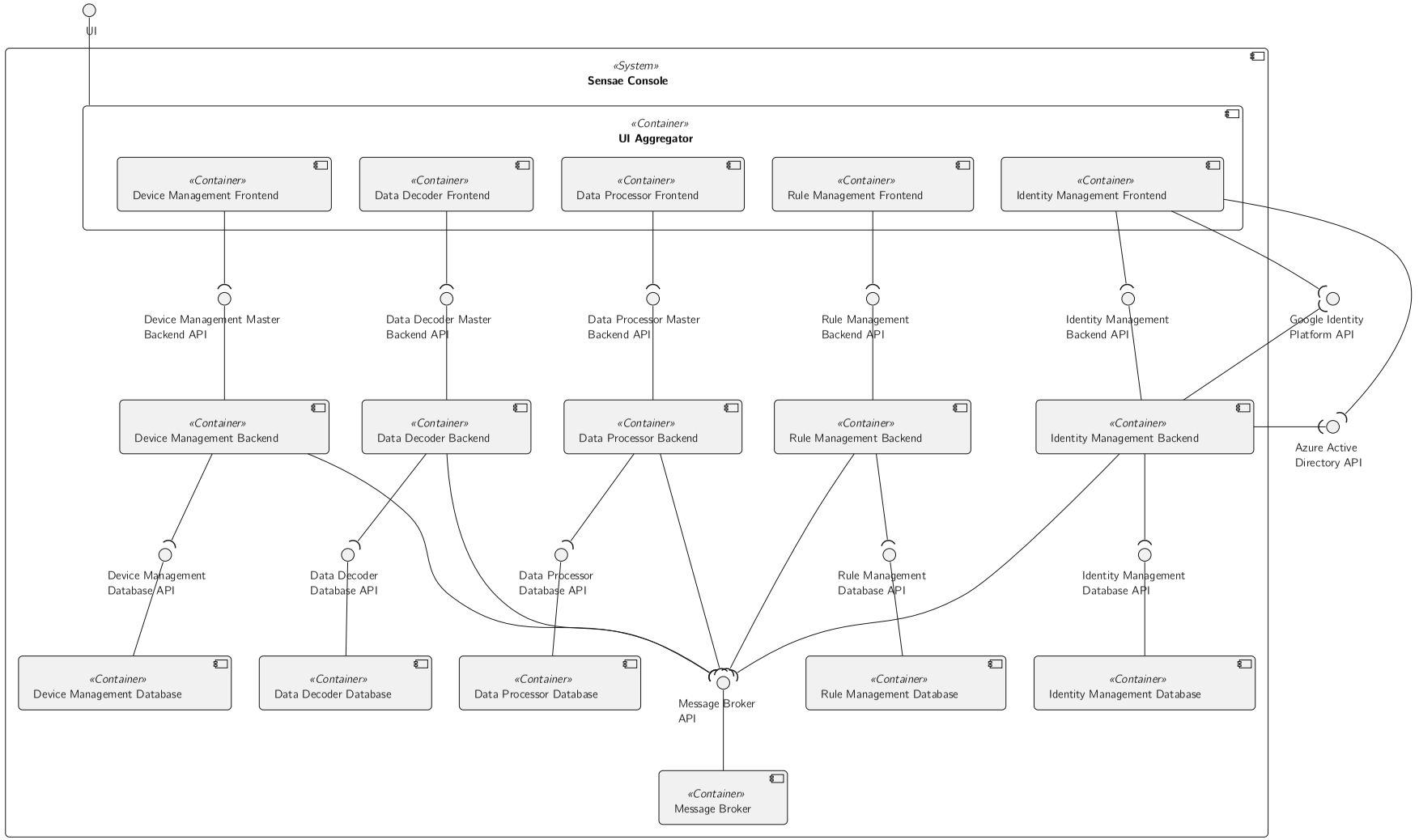

Figure 5.20: Container Level - Configuration Scope - Logical View Diagram

As a brief description:

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

Next, the logical view of the **Data Flow** is represented in Figure 5.21. 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.

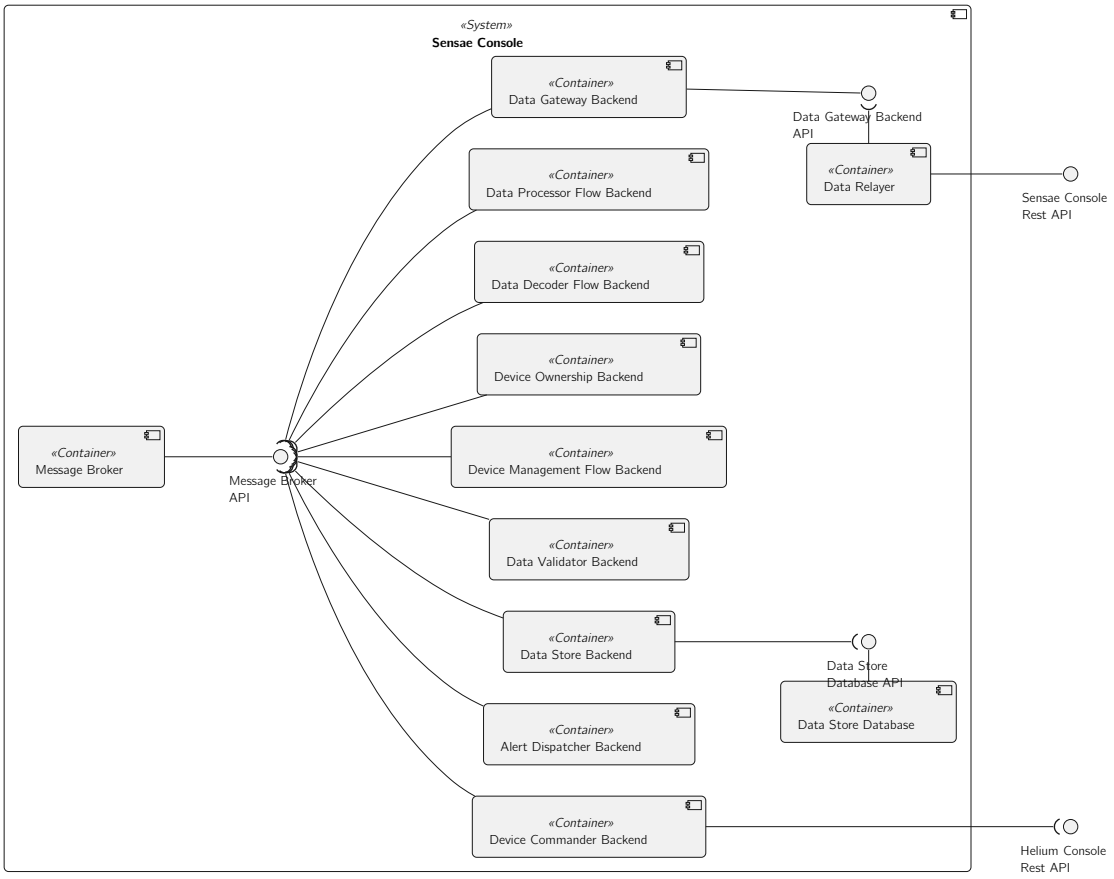

Figure 5.21: Container Level - Data Flow Scope - Logical View Diagram

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

- **Data Processor Flow Backend**: Collects information related to the **Data Processor** context - published by **Data Processor Backend** - and noting else;
- **Data Decoder Flow Backend**: Collects information related to the **Data Decoder** context - published by **Data Decoder Backend** - and noting else;

- **Device Ownership Backend**: Collects information related to the **Identity Management** context (more specifically device ownership) - published by **Identity Management Backend** - and noting else;
- **Device Management Flow Backend**: Collects information related to the **Device Management** context - published by **Device Management Backend** - and noting else;
- **Alert Dispatcher Backend**: Collects information related to the **Rule Management** context - published by **Rule Management Backend** - and noting else;
- **Device Command Backend**: Collects information related to the **Device Management** context - published by **Device Management Backend** - and noting else;
- The remaining containers don't subscribe to any type of information from the **Configuration Scope**.

Finally the **Service Scope** is represented in Figure 5.22. This scope is composed by the processes discussed in 5.1.3.

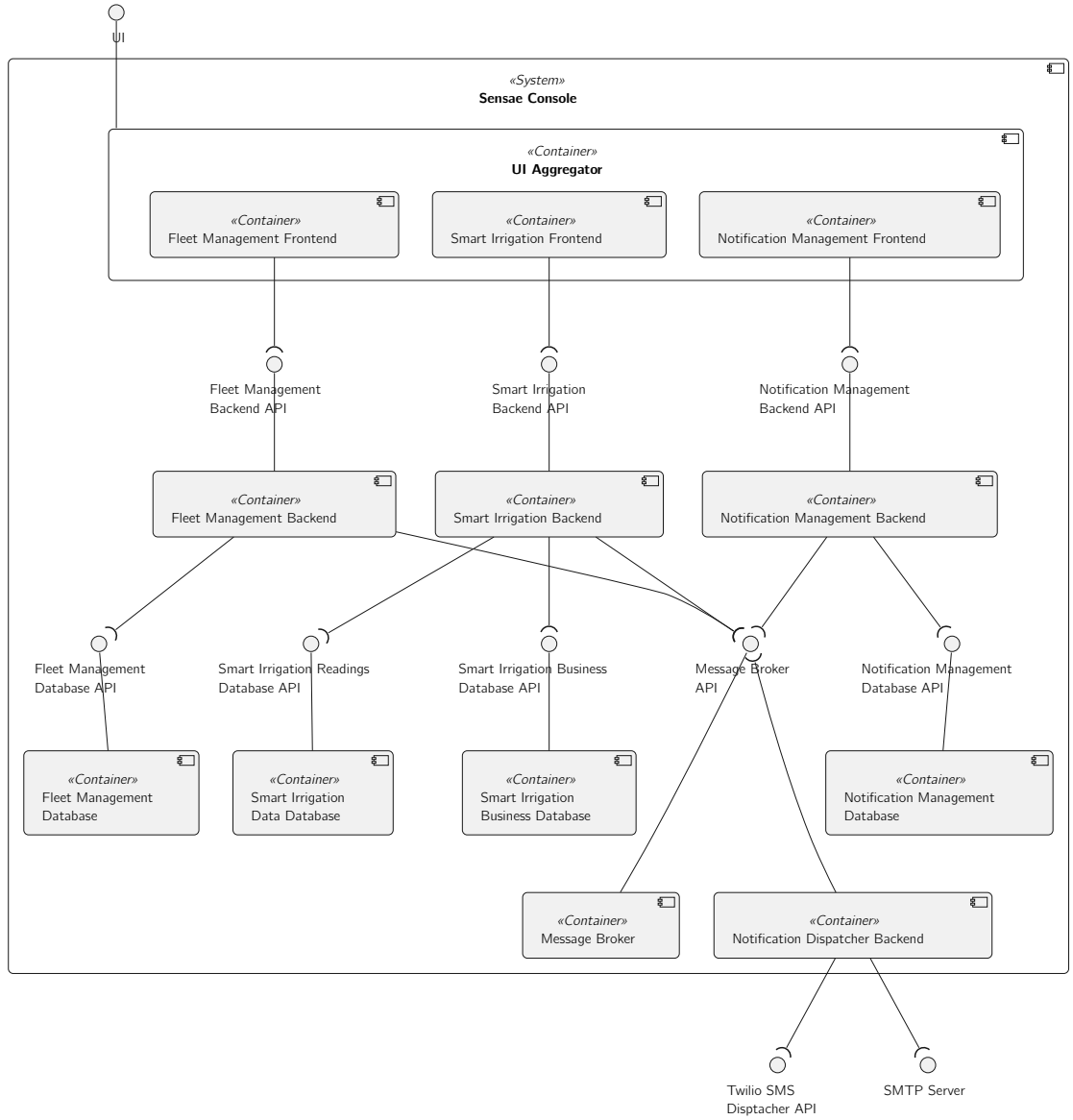

Figure 5.22: Container Level - Service Scope - Logical View Diagram

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 5.2.3, 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 the management of them.

Lastly, as we can seen some containers are present in more than one scope, this 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 User Interface (UI).

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

Container Level - Process View

In this section several use cases (according to ***TODO***) 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 5.2.2.

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

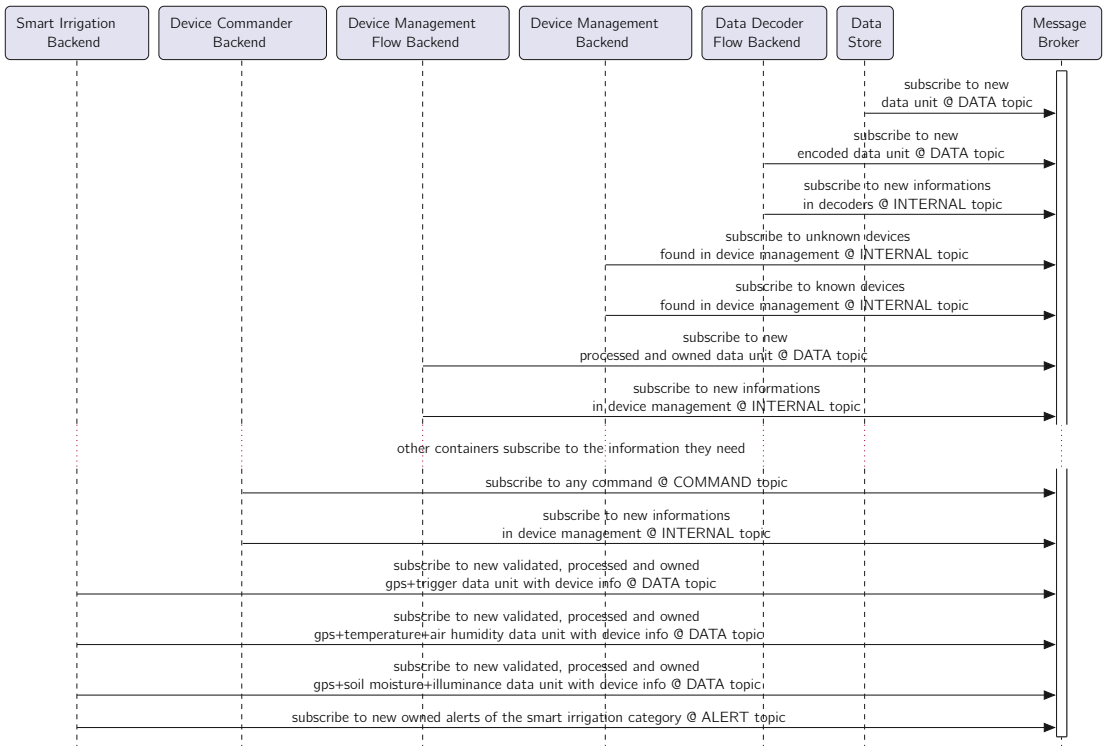

Figure 5.23: Container Level - System/Container Initialization - Process View Diagram

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 scenarios 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 5.24,note that this only occurs in the Internal Topic.

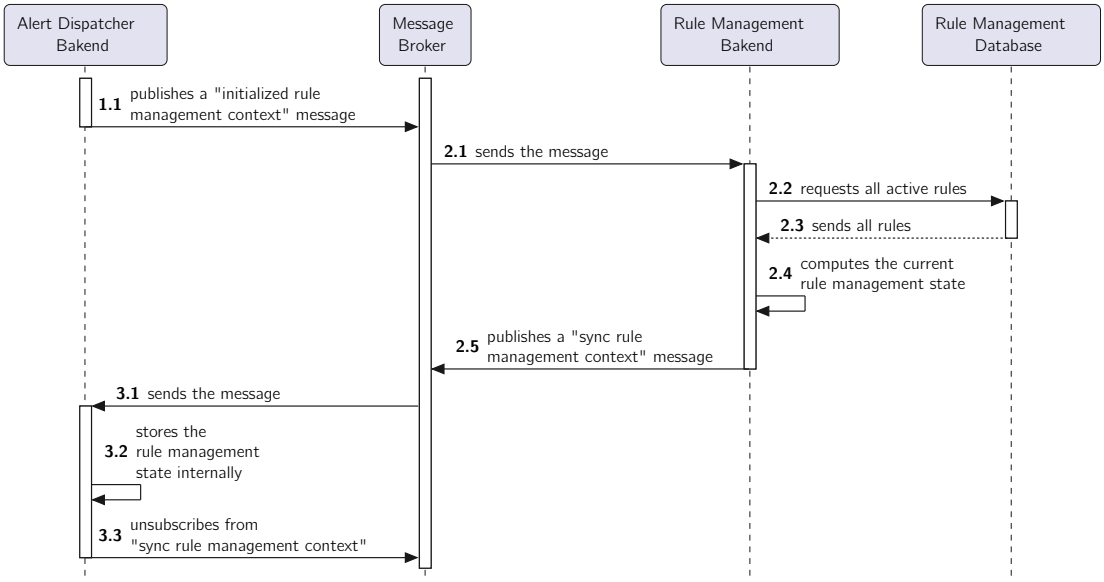

Figure 5.24: Container Level - System/Container Initialization - Part 2 - Process View Diagram

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

The following diagram in Figure 5.25 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 tho it is used by all containers but the Data Relayer to publish messages.

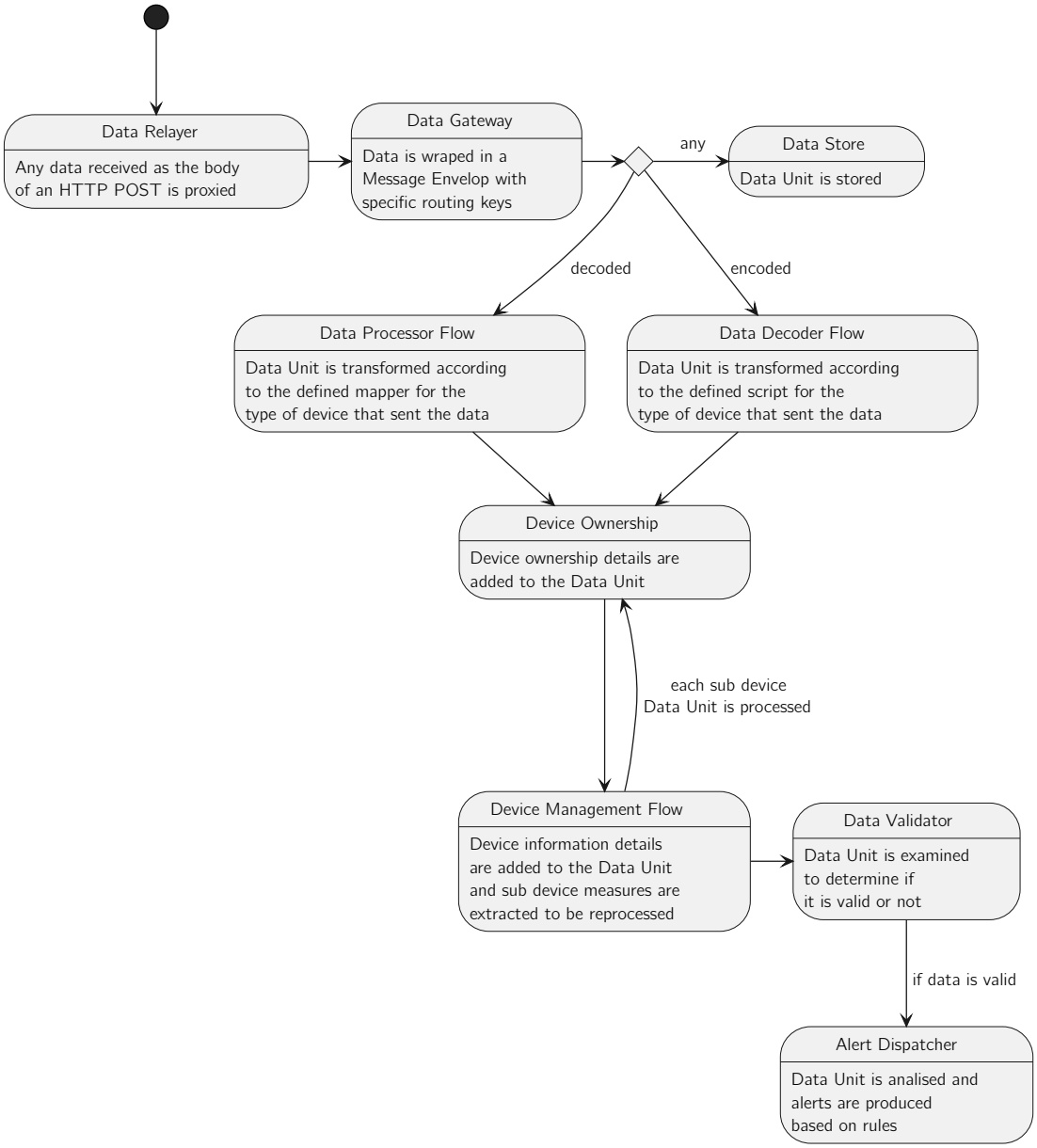

Figure 5.25: Container Level - Data Flow - Diagram

Most of this 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 5.26 and Figure 5.27, addresses how state is managed in Data Decoder Flow Backend and most **Data Flow Scope** containers.

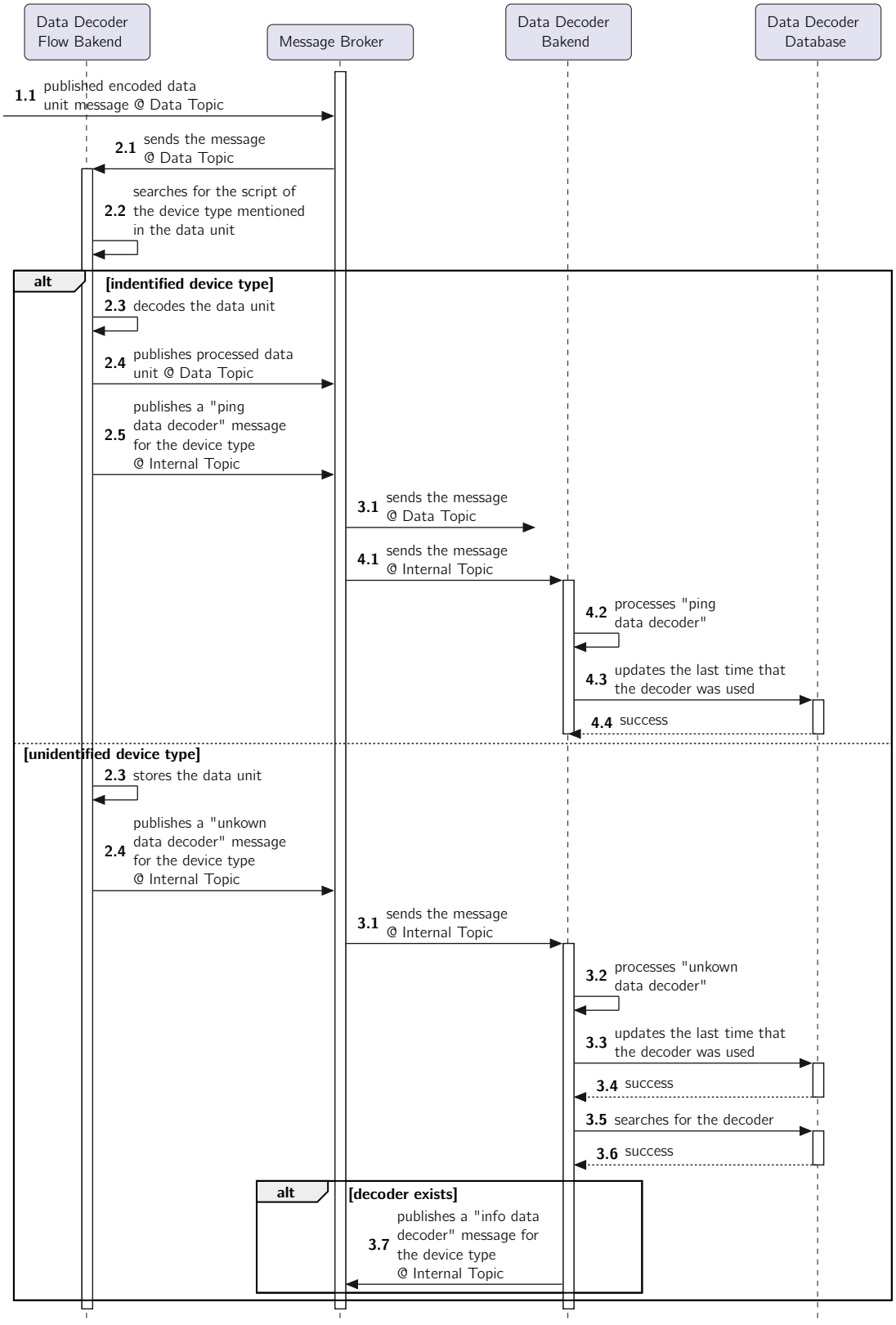

Figure 5.26: Container Level - Data Decoder Operation Part 1 - Process View Diagram

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

two operations depending on the script being available or not: 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 demonstrates what happens when a decoder is published via the *OperationType* Info.

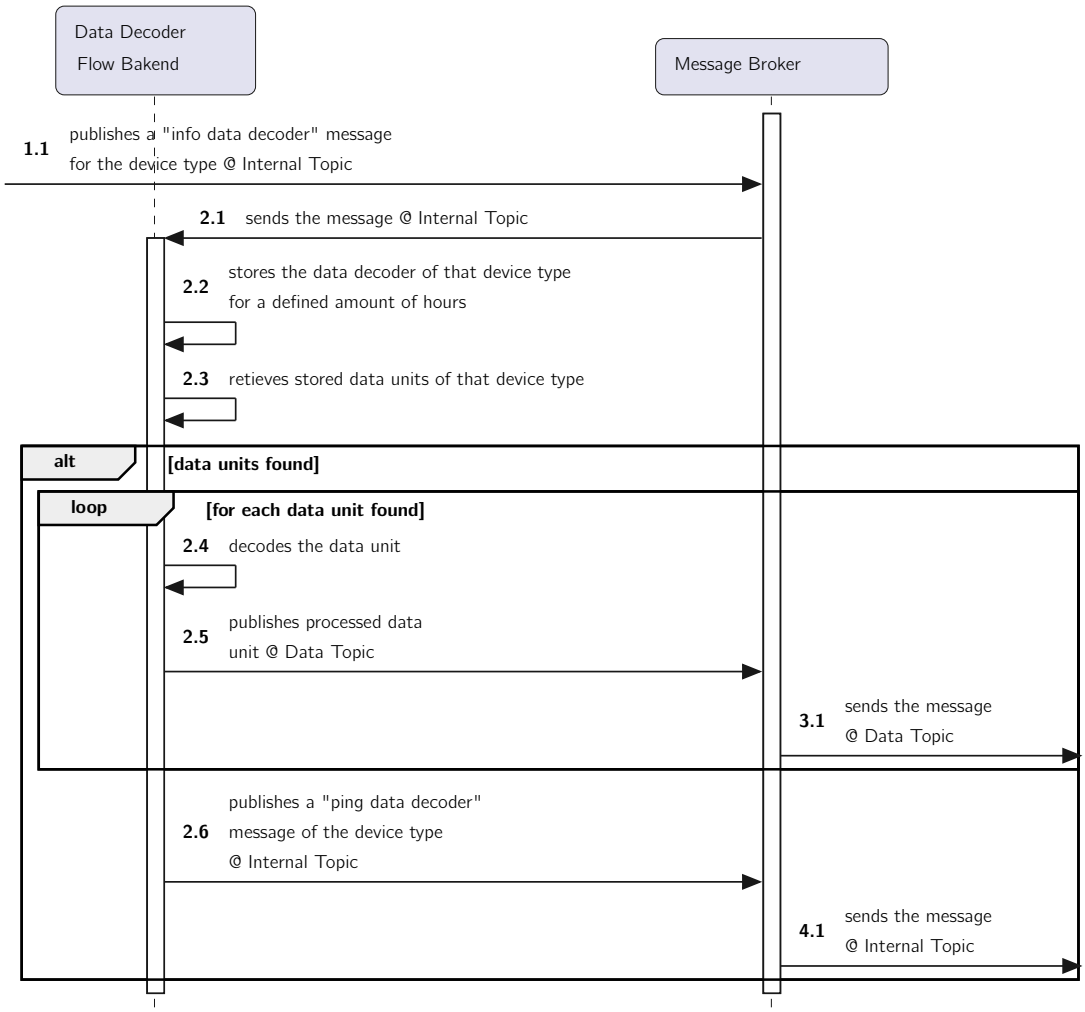

Figure 5.27: Container Level - Data Decoder Operation Part 2 - Process View Diagram

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** are configured according to what is defined in the **Configuration Scope**.

The next diagrams, in Figure 5.28 and Figure 5.29 present some of the common operations that happen in that scope.

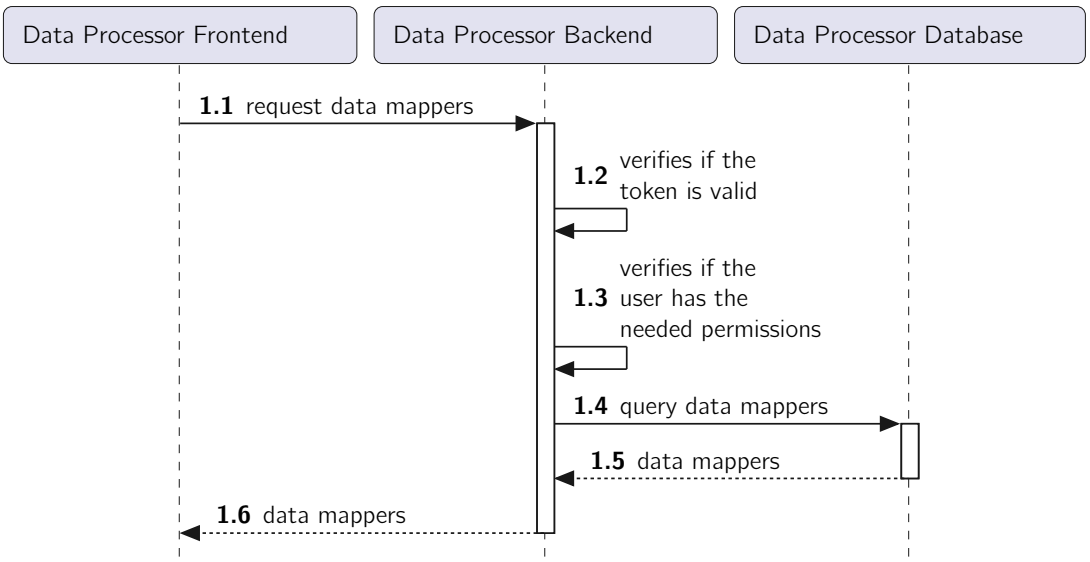

Figure 5.28: Container Level - Consult Data Processors - Process View Diagram

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 5.29 displays an example of this occurrence.

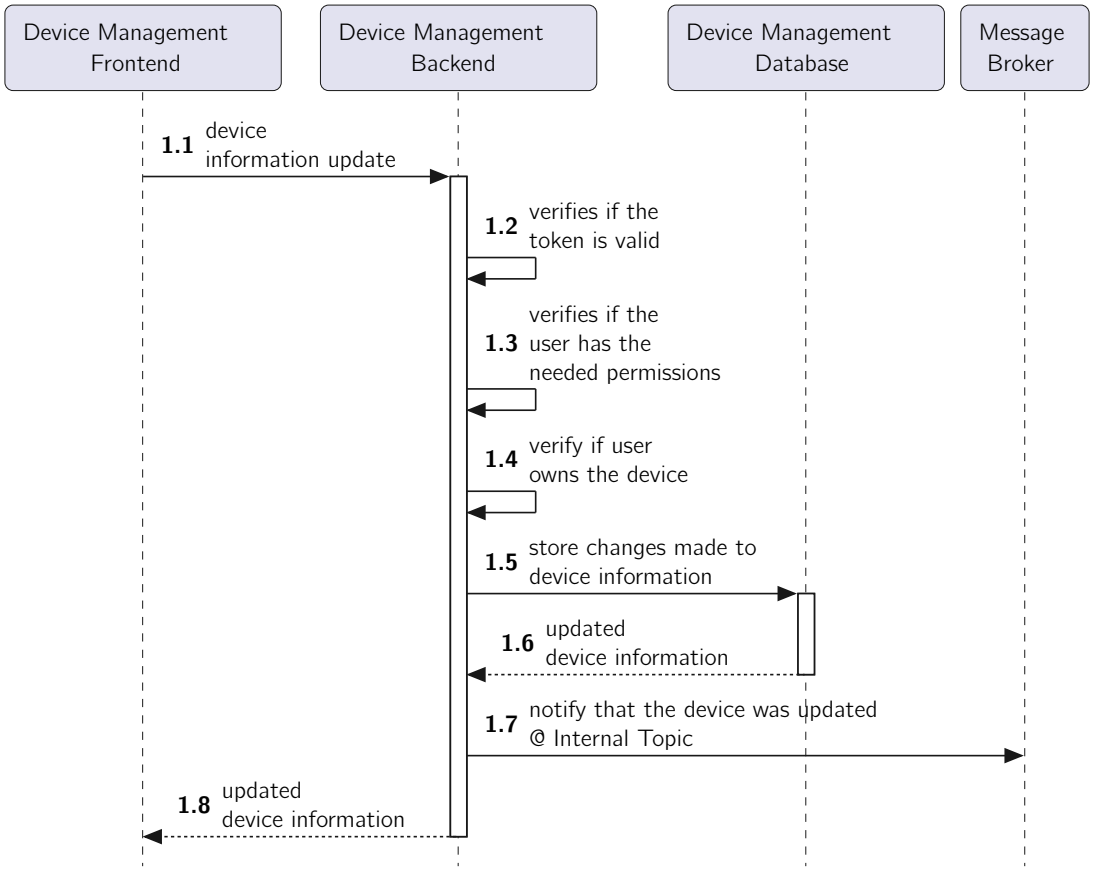

Figure 5.29: Container Level - Edit Device Information - Process View Diagram

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 this specific event, according to the Section 5.2.2, 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 5.30 and Figure 5.31 - authentication and authorization in the **Sensae Console** are addressed, other approaches are discussed in the User Authorization/Authentication 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.

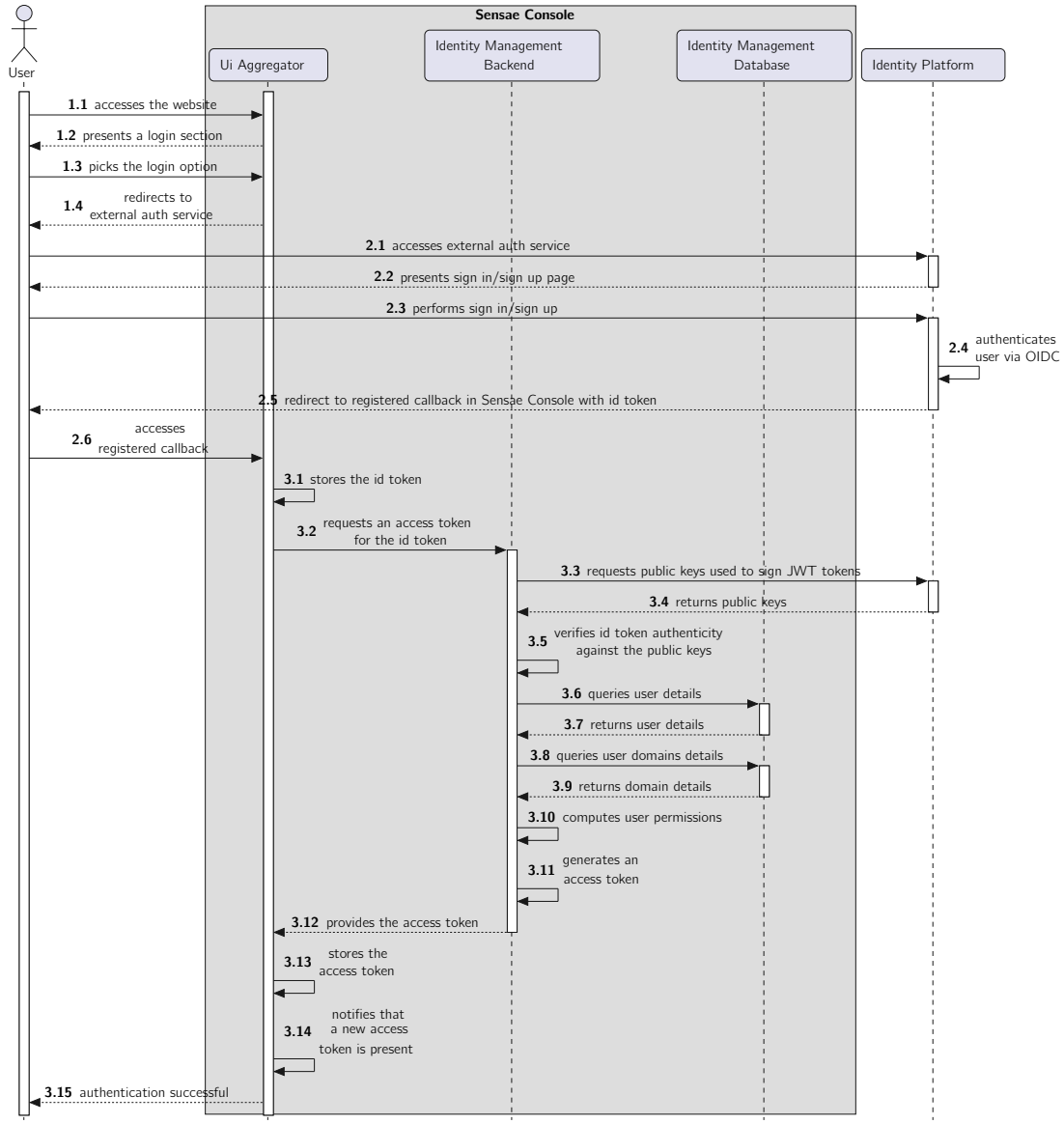

Figure 5.30: Container Level - User Authentication - Process View Diagram

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 5.31.

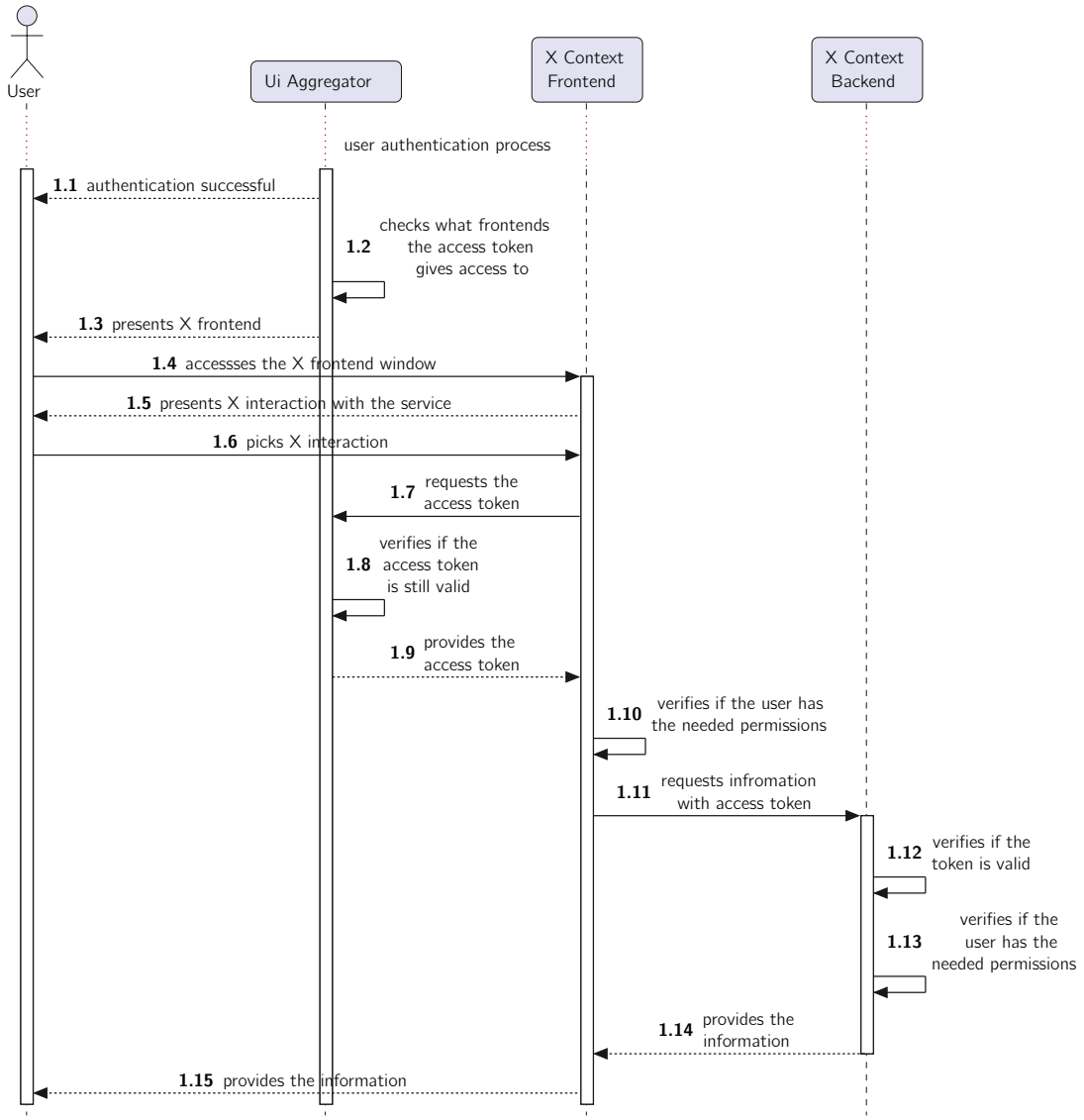

Figure 5.31: Container Level - User Authorization - Process View Diagram

In this diagram the expected behavior for any pair of frontend and backend containers in **Configuration Scope** and **Service 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 5.4.3.

Finally some operations performed in the **Service Scope** are presented starting with how a user can see the current location of a device via the Fleet Management Service (Figure 5.32). Authentication details will be omitted for brevity reasons.

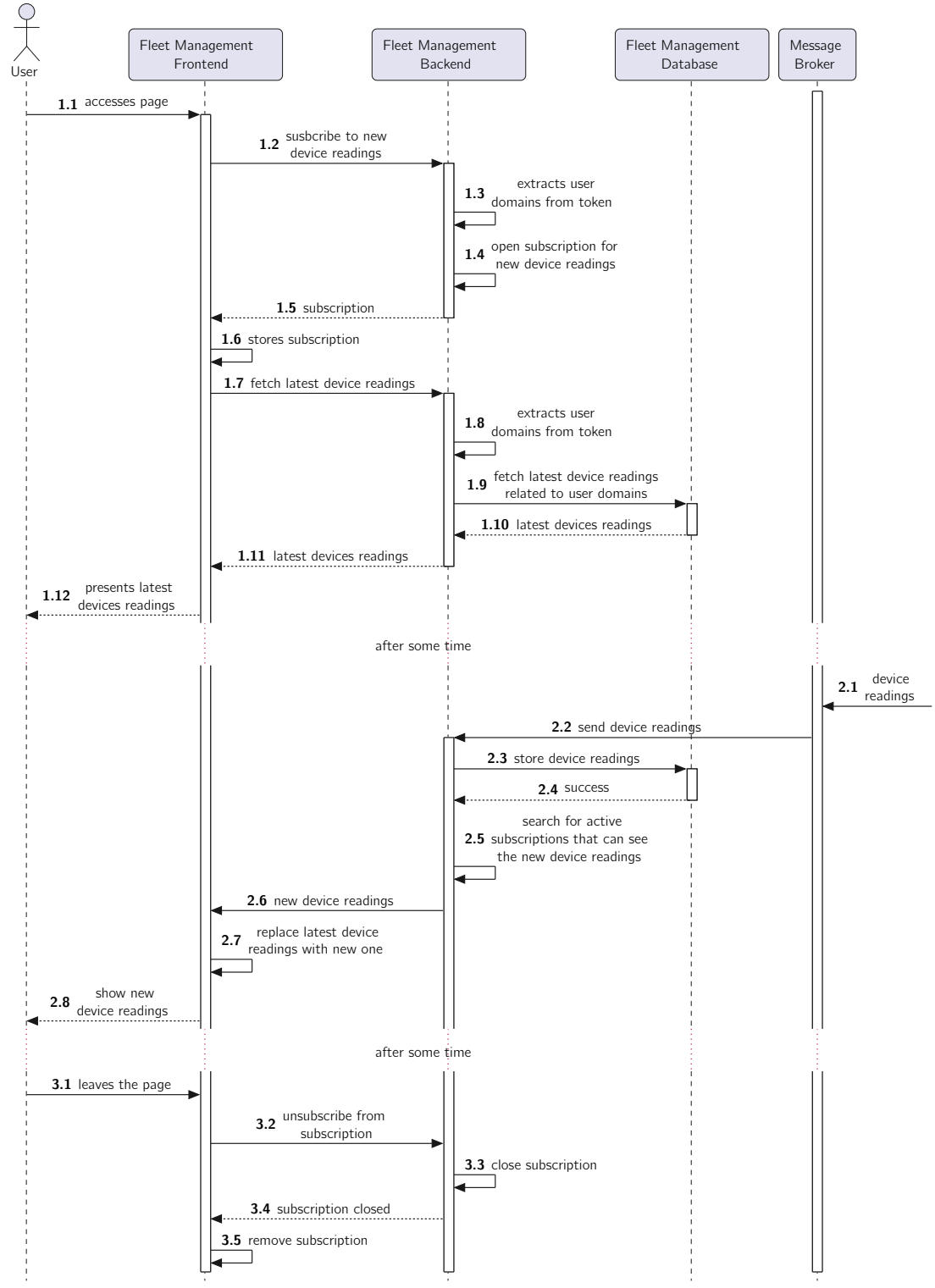

Figure 5.32: Container Level - Consult Device Live Location via Fleet Management - Process View Diagram

In order to provide live information to the user **Service 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**. Fist the frontend must subscribe to new information with a valid *access token* - steps **1.2** to **1.6** - then this channel is maintained till the user leaves the page. Once the user leaves the page the subscription is closed in the frontend and subsequently in the backend - steps **3.2** to **3.5**.

The next diagram in Figure 5.33 describes how a user receives notifications via several different delivery channels. For brevity reasons the subscription process is omitted.

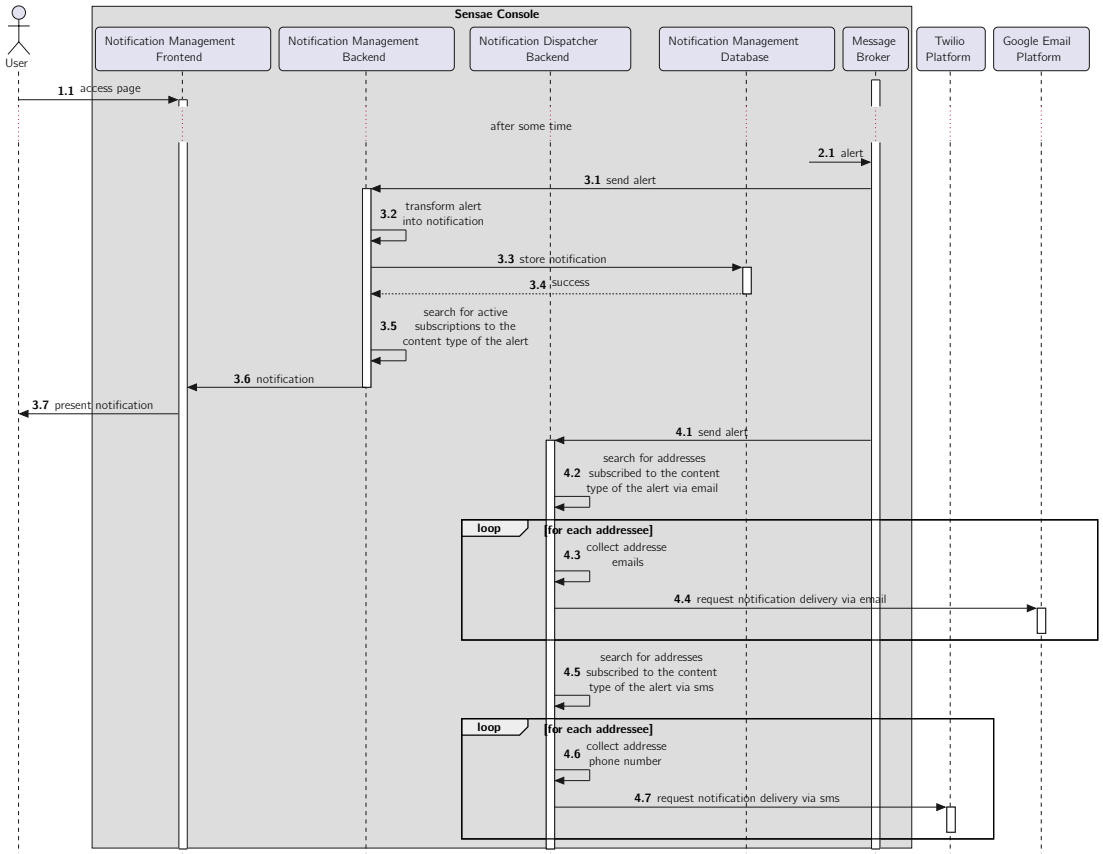

Figure 5.33: Container Level - Receive notification via Notification Management - Process View Diagram

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 though 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 deliveries 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 deliveries 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 5.34, this process is presented.

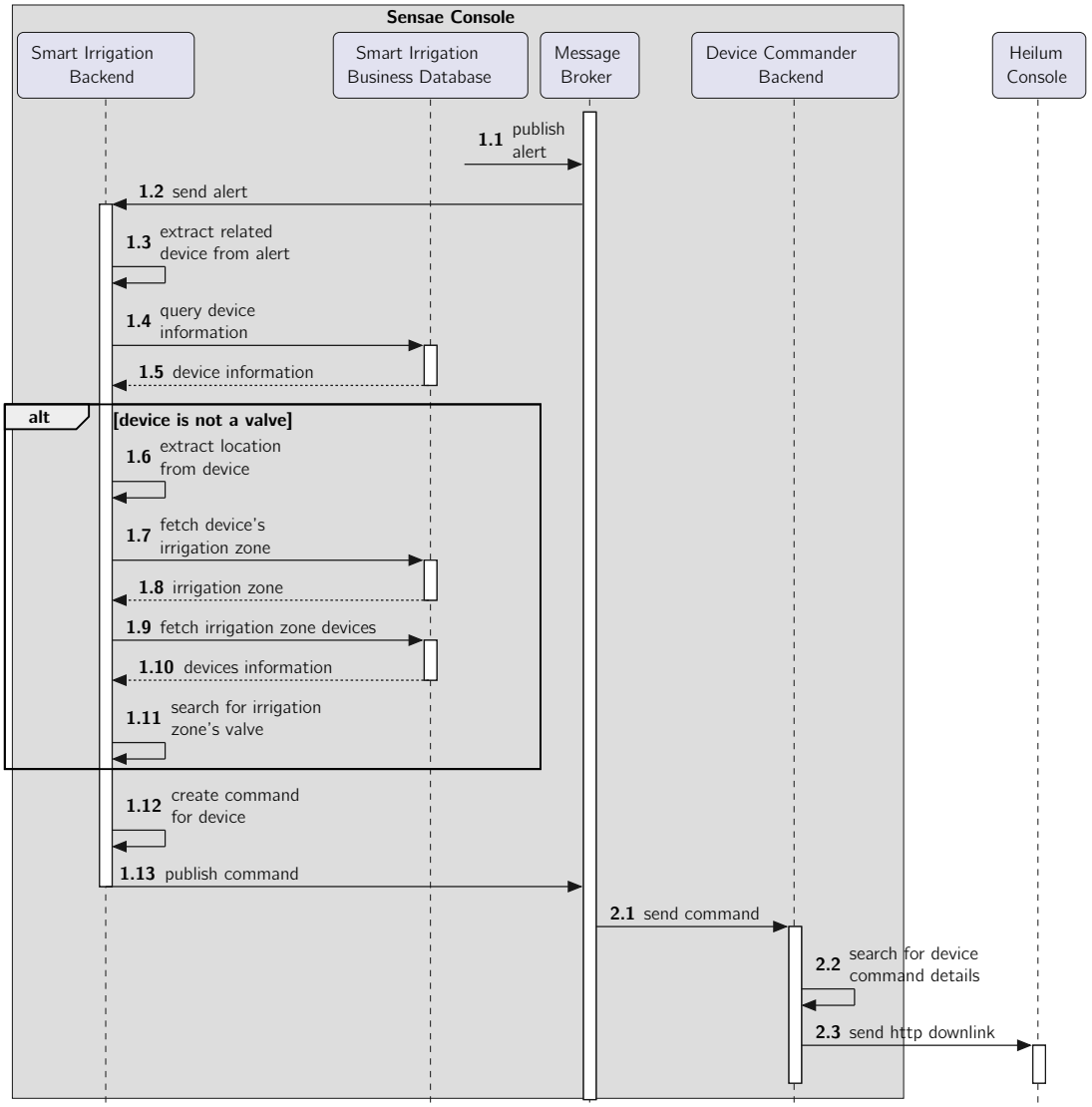

Figure 5.34: Container Level - Valve Activation Process via Smart Irrigation - Process View Diagram

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

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.

Container Level - Development View

Each container mentioned in the Section 5.3.2 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 5.35.

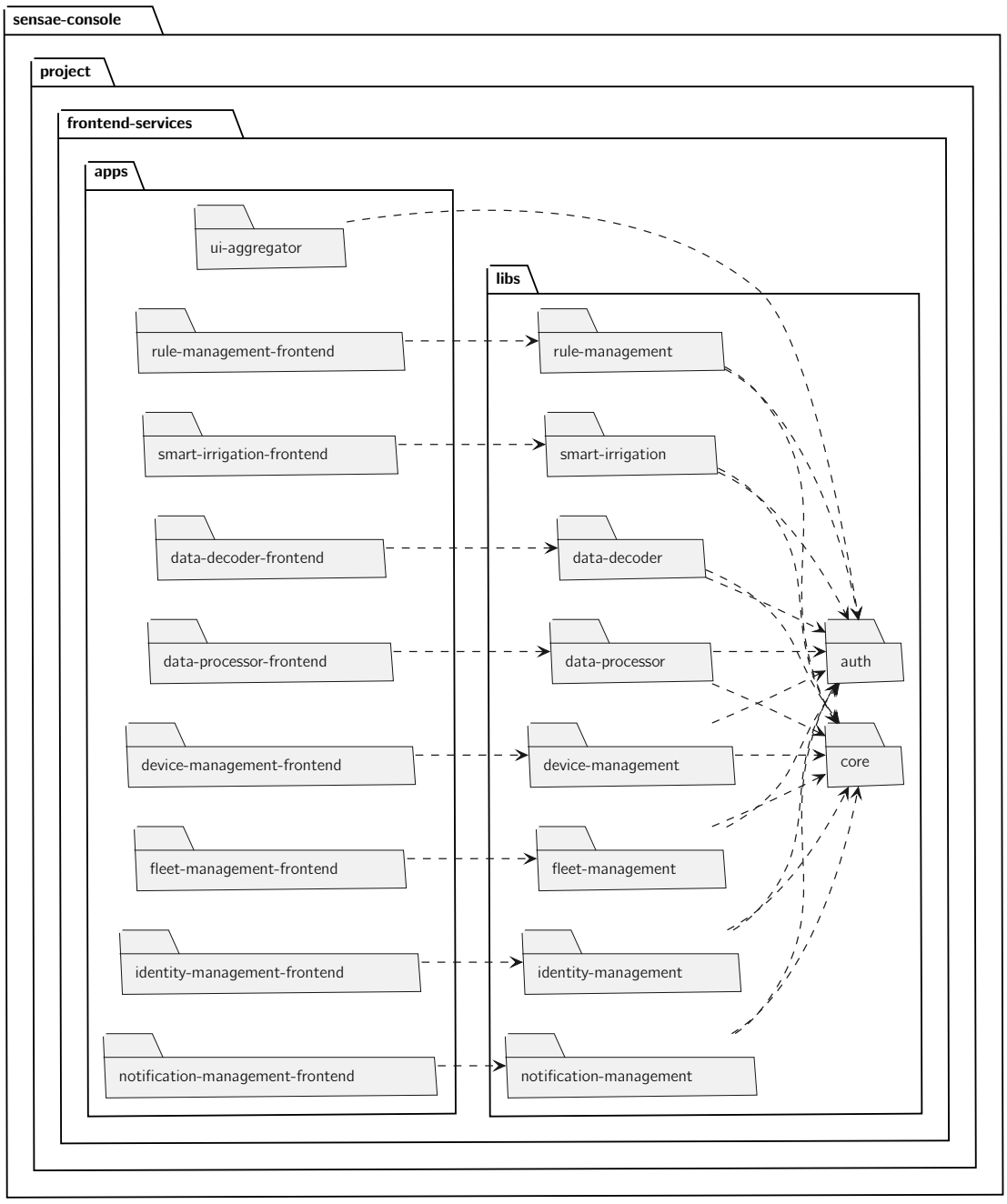

Figure 5.35: Container Level - Frontend Services - Development View Diagram

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 5.36.

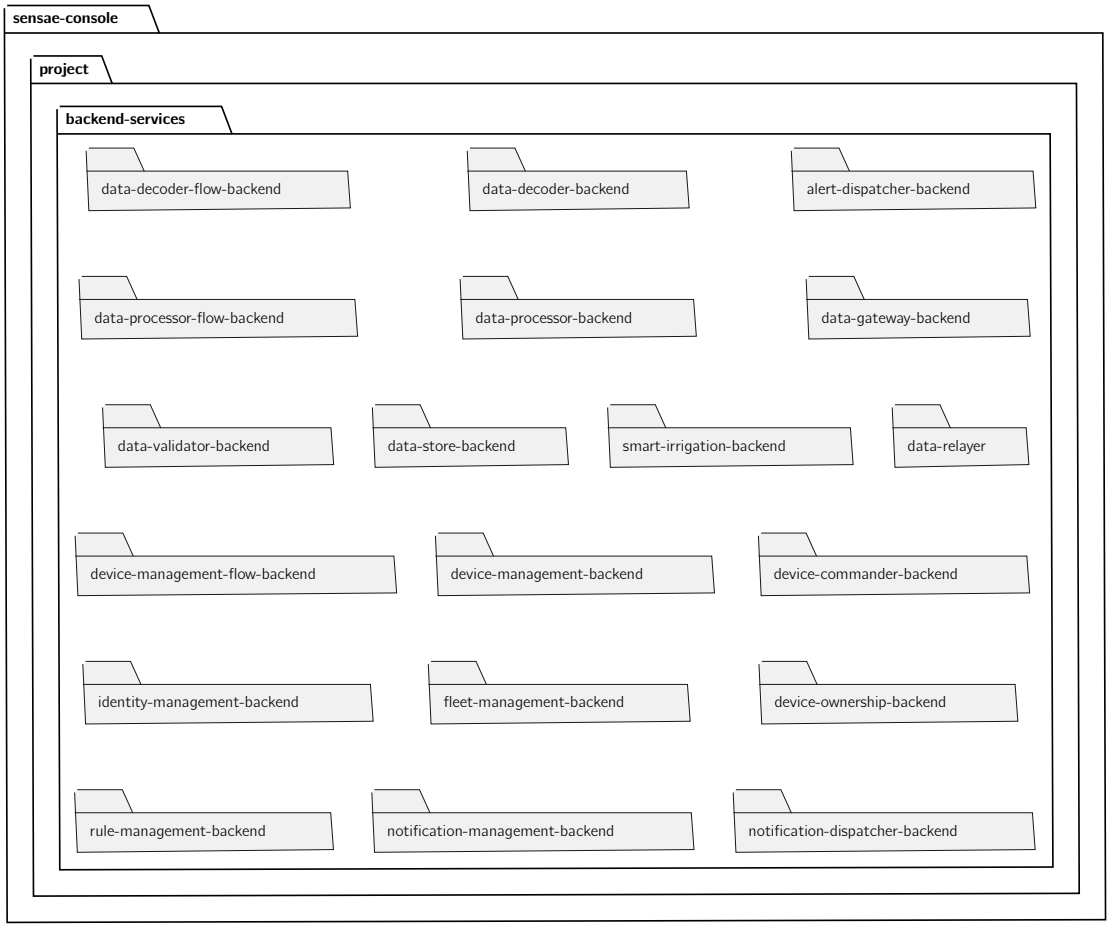

Figure 5.36: Container Level - Backend Services - Development View Diagram

Each backend service software lives inside its own package. All containers have been develop besides the *Data Relayer* that was only configured.

Database services are organized according to the diagram in Figure 5.36.

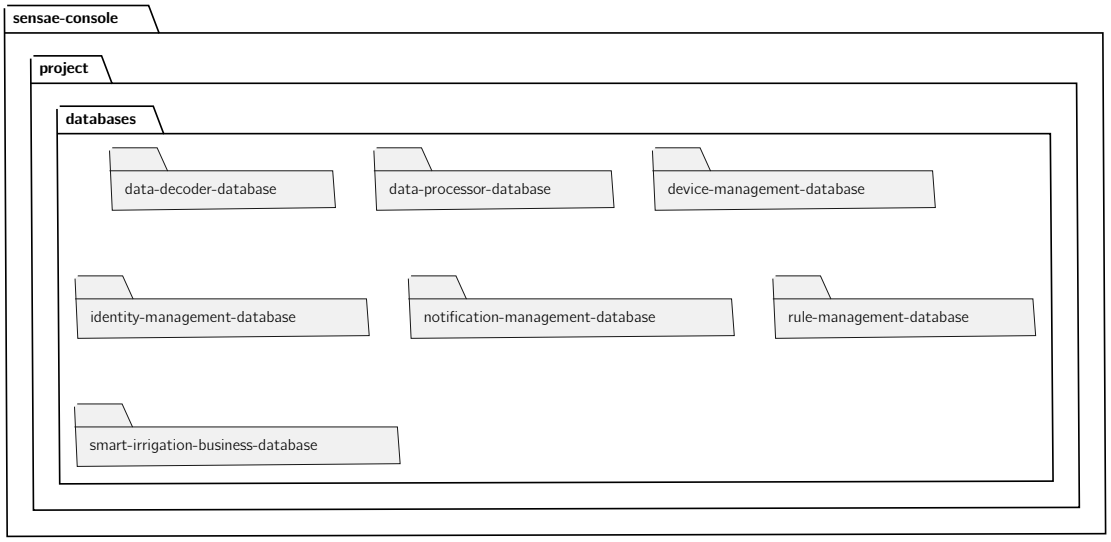

Figure 5.37: Container Level - Database Services - Development View Diagram

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.

Container Level - Physical View

Next is the physical view (Figure 5.38), 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.

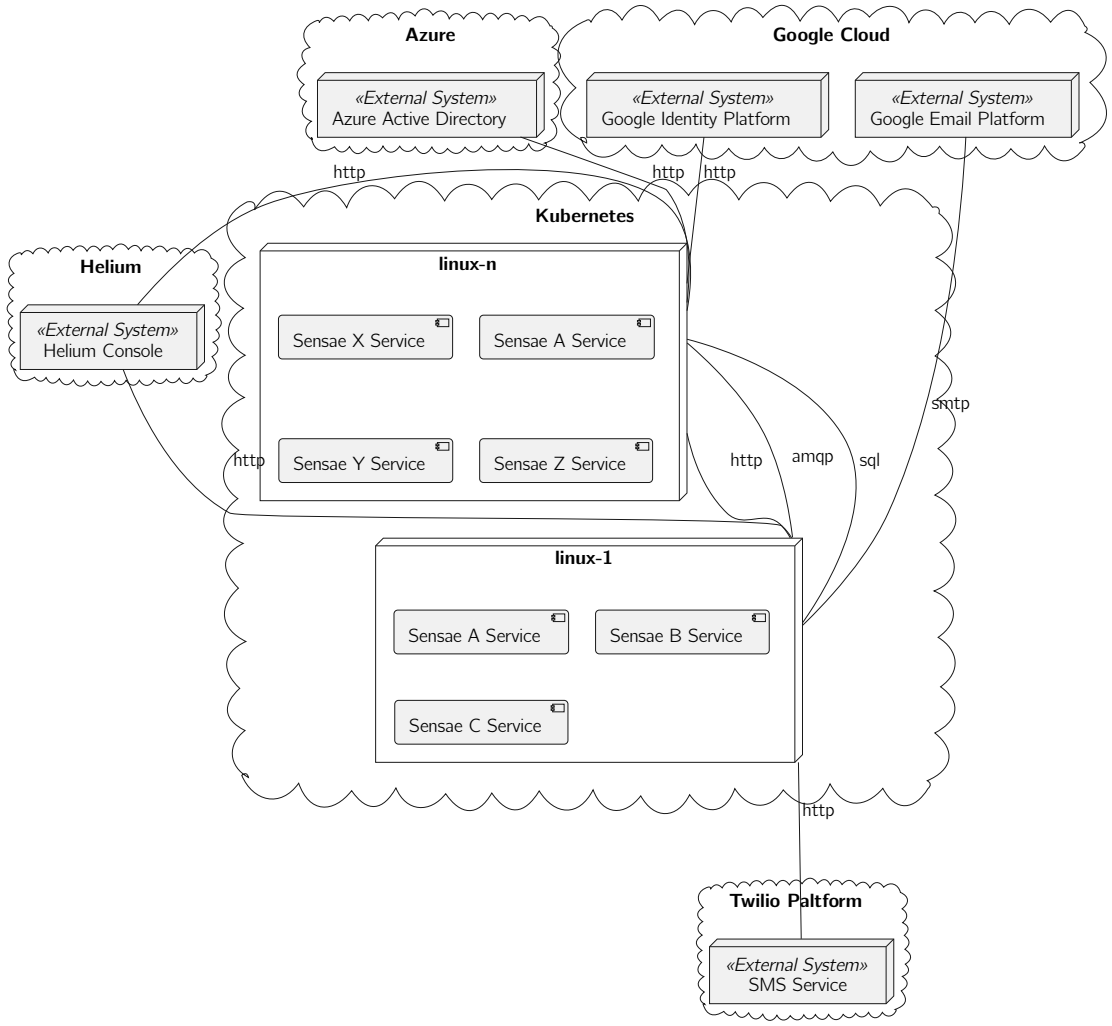

Figure 5.38: Container Level - Physical View Diagram

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.

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 5.3.3, the developed containers are presented with a granularity of level 3 (in the C4 model).

5.3.3 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.

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 Responsability Principle (SRP) (high cohesion) and to lower the coupling between the information displayed in the UI and the data sent/received by the container. This segmentation led to the addition of the Mapper component, which has the responsibility of converting the data (DTOS component) into information (Model component) and vice-versa. The Auth component indicates what backend resources the user has access to and the Utils component has several methods commonly used to process backend requests, this two components are reused in all frontend containers.

As an example the logical view of the Data Decoder Frontend is presented in Figure 5.39.

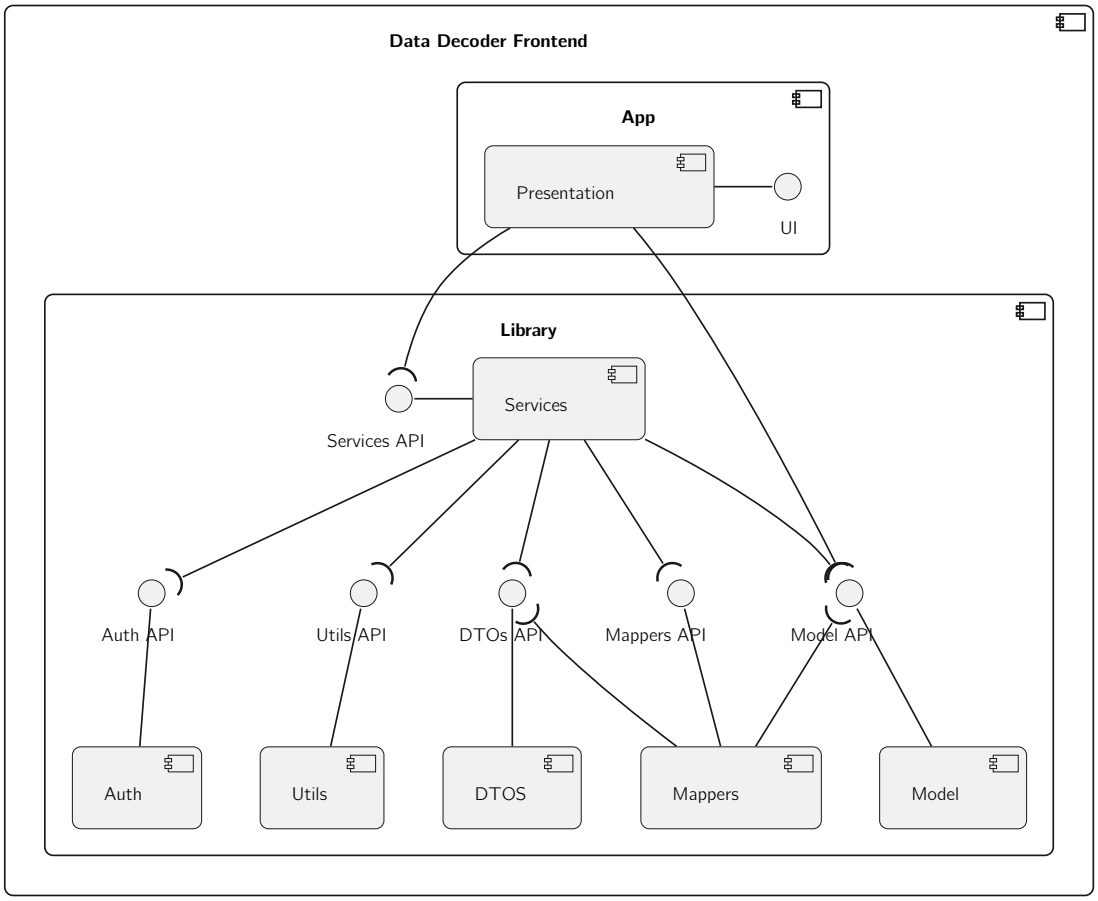

Figure 5.39: Component Level - Data Decoder Frontend - Logical View Diagram

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 them the other frontend containers, it is comprised by a Presentation component that depends on the Auth component to handle user authentication and authorization.

Next, the 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 5.40.

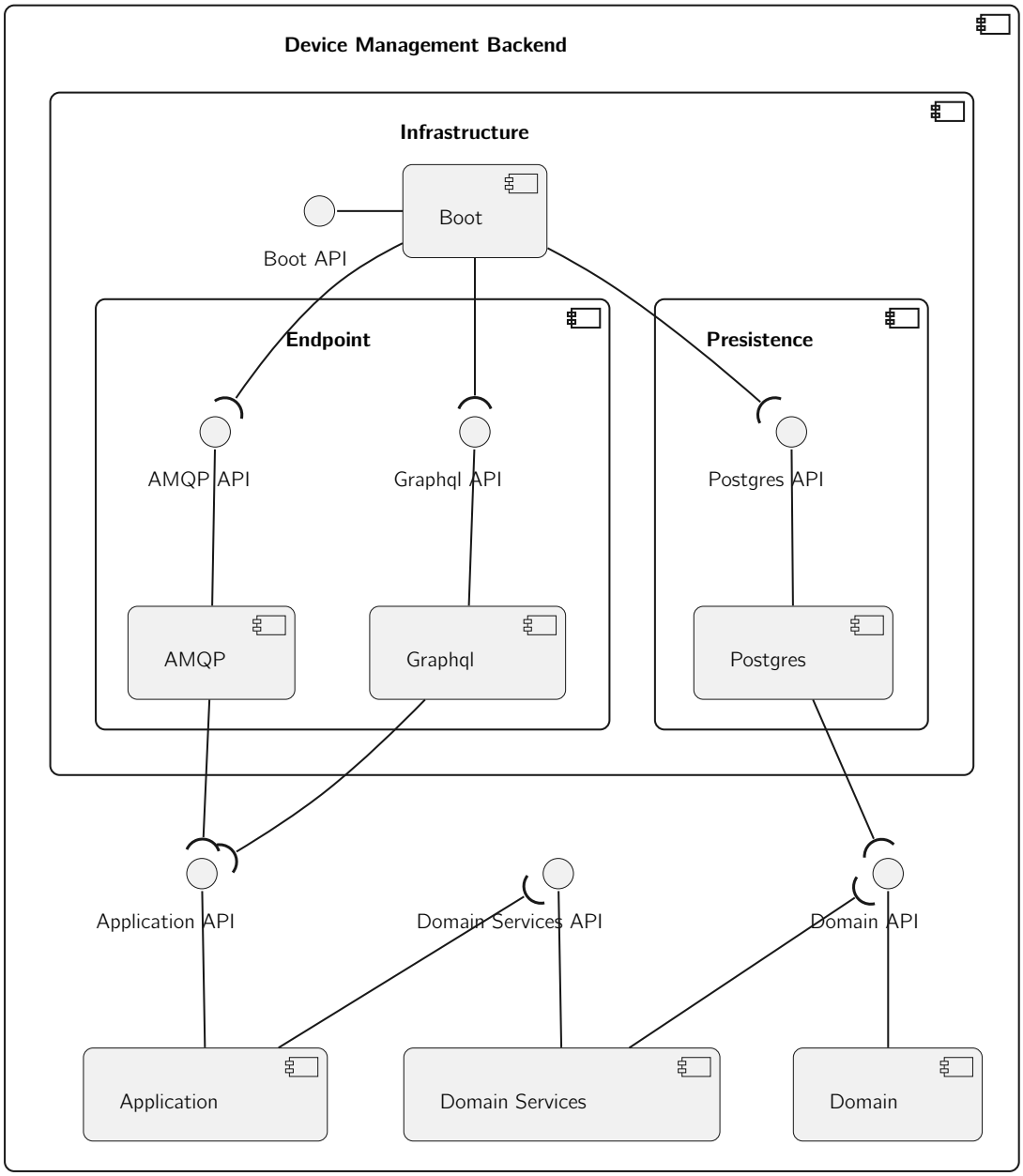

Figure 5.40: Component Level - Device Management Backend - Logical View Diagram

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 5.4, discusses each component responsibilities.

Component	Responsibilities
Infrastructure	- Enclose components that manage the Input/Output operations required by the container;
Boot	- Manage the start up of the container; - Construct the components' pieces according to the defined dependencies; - Manage the configuration of the container;
Endpoint	- Enclose components that are used by external containers to interact with the container;
AMQP	- Define how to consume and publish events in the Message Broker; - Delegate the handling of events received to specific Application processes;
GraphQl	- Define the interface to be consumed by the frontend; - Delegate external requests made to specific Application processes;
Persistence	- Enclose components that interface with containers responsible for persisting data;
Postgres	- Interact with a database to persist and query data;
Application	- Represent the application processes; - Ensure the propagation of events related to the process in question, requiring this responsibility to AMQP; - Ensure the execution of the process in question, requiring this responsibility to Domain Services; - Enforce user authorization;
Domain Services	- Represent business processes; - Interact with the Domain; - Ensure the persistence of the data in question, requiring this responsibility to the Persistence;
Domain	- Represent de business rules and concepts; - Manage the system information;

Table 5.4: Components responsibilities

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 5.41.

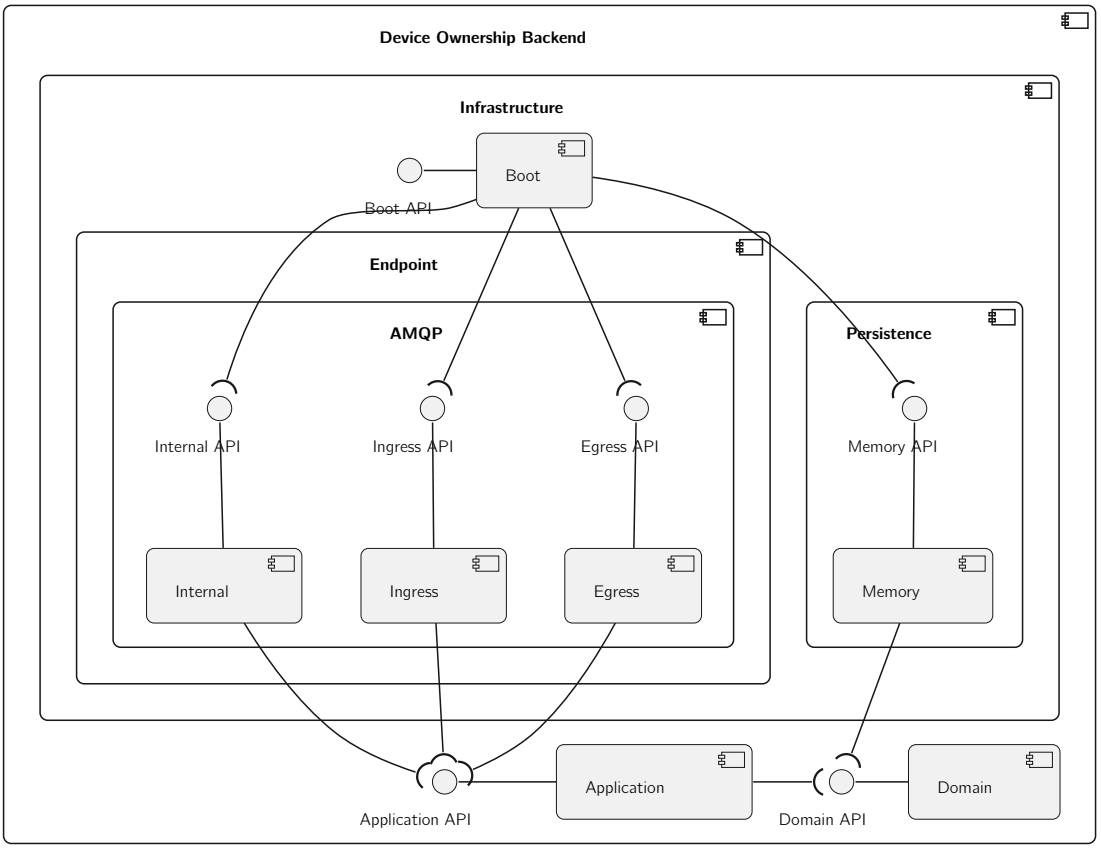

Figure 5.41: Component Level - Device Ownership Backend - Logical View Diagram

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.

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 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 5.42, details this process.

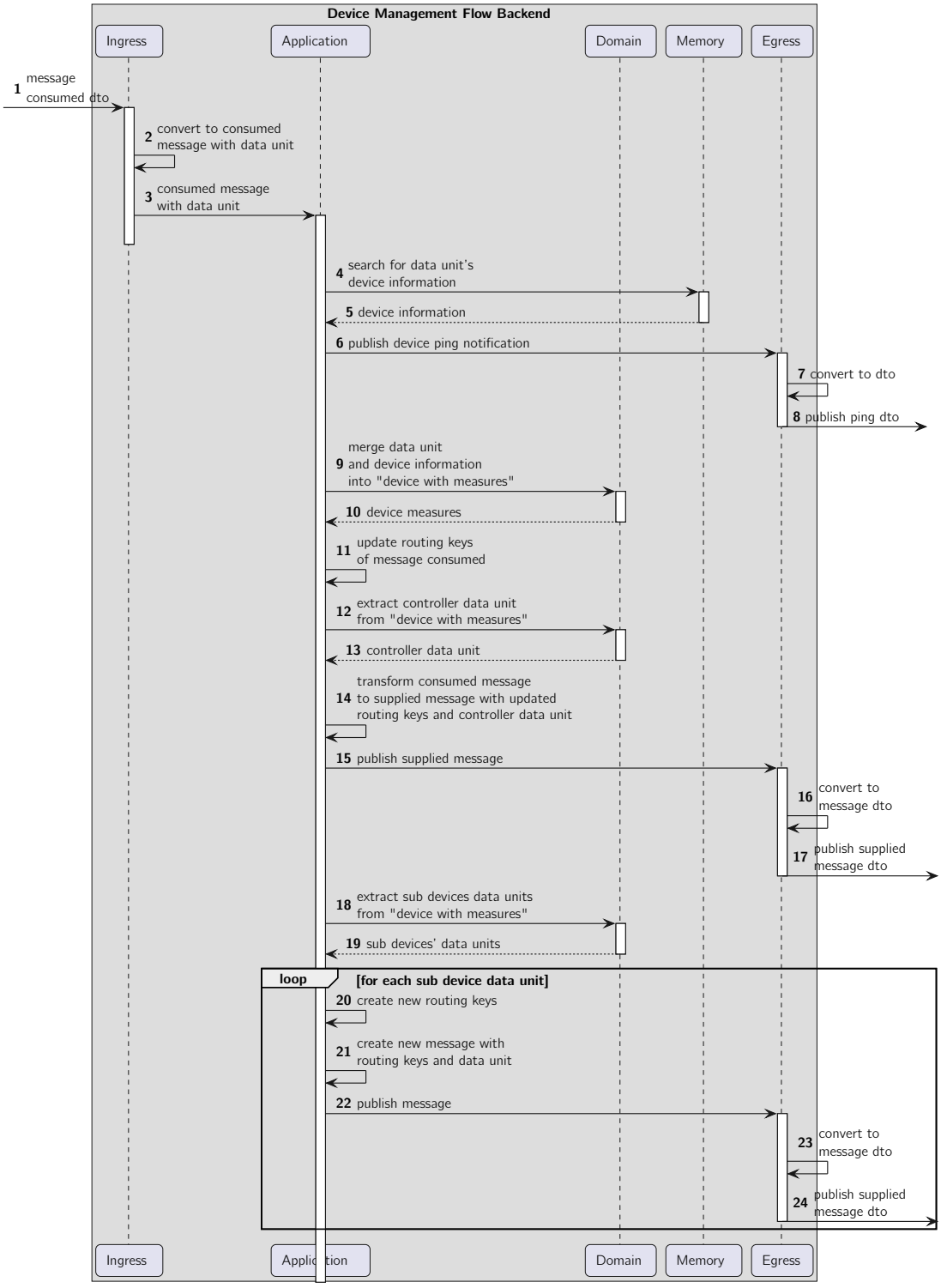

Figure 5.42: Component Level - Process Data Unit in Device Management Flow Backend - Process View Diagram

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;
- 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 clarified. 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 5.43, tackles this concern.

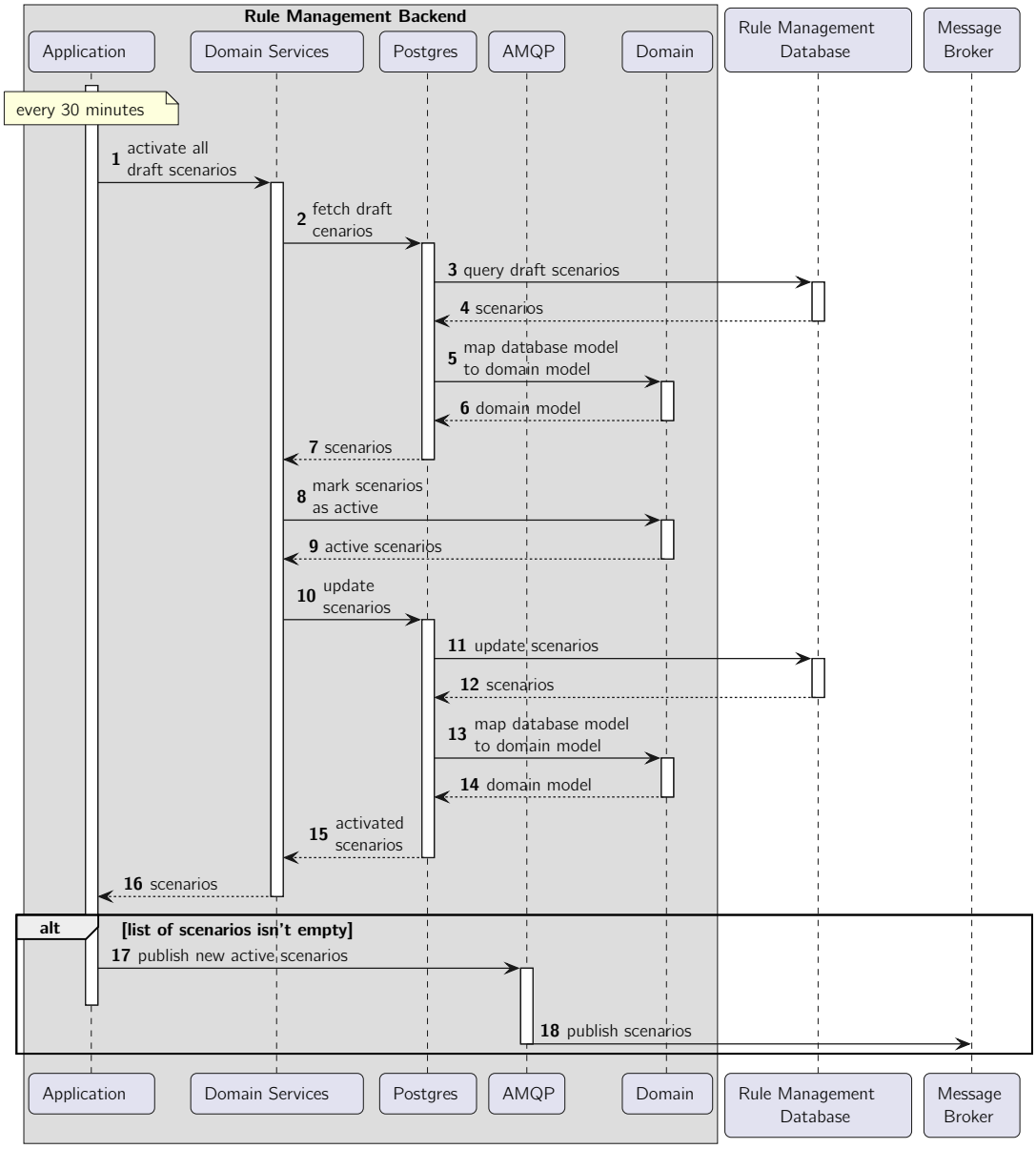

Figure 5.43: Component Level - Deploy Draft Rule Scenarios in Rule Management Backend - Process View Diagram

As seen in the diagram, to mitigate the number of lost alarms, new rule scenarios are published at best every 30 minutes - step **1** - and only if any change was made - step **17** and **18**.

Components Level - Development View

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

The next diagrams, Figure 5.44, Figure 5.45 and Figure 5.46 describe this view at the components level.

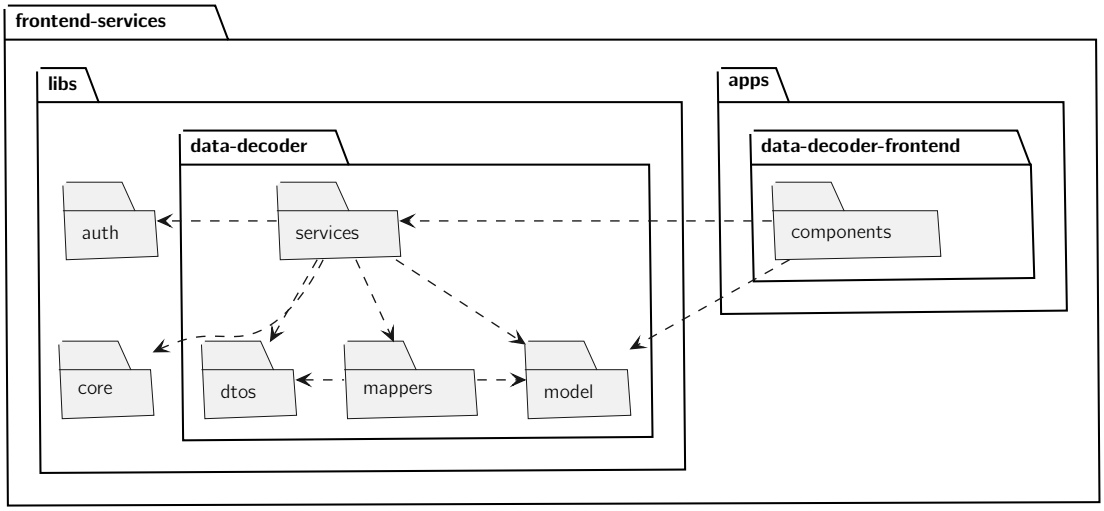

Figure 5.44: Component Level - Data Decoder Frontend - Development View Diagram

The packages presented correspond to the components described in the logical view (Figure 5.39). 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.

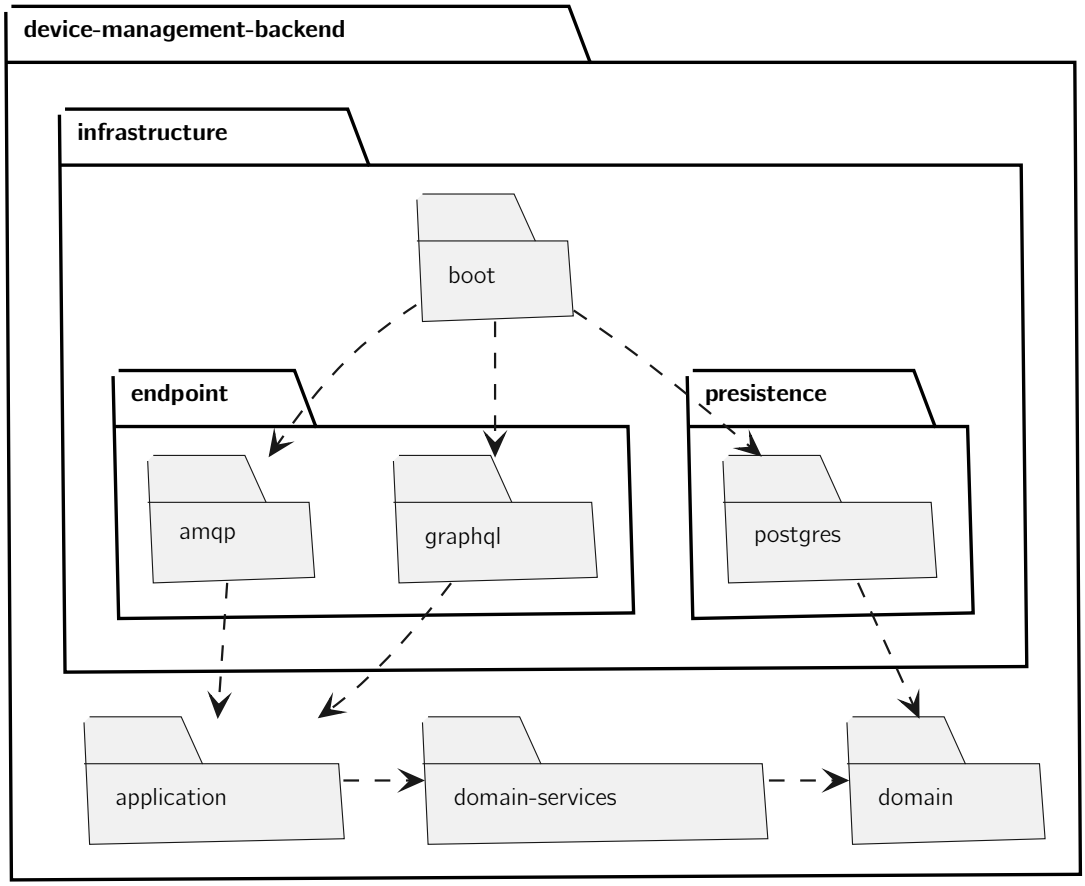

Figure 5.45: Component Level - Device Management Backend - Development View Diagram

The packages presented correspond to the components described in the logical view (Figure 5.40). The names given in both views differ only on the case used.

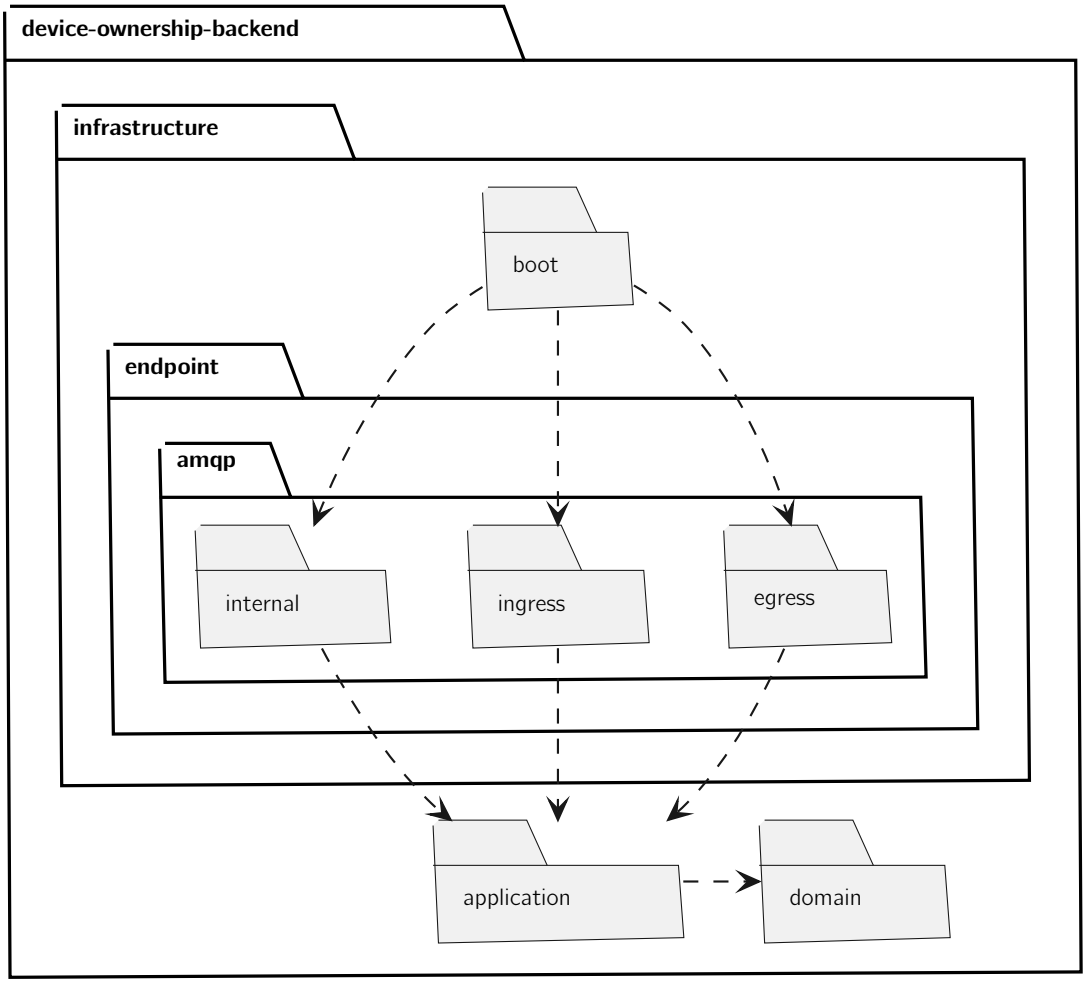

Figure 5.46: Component Level - Device Ownership Backend - Development View Diagram

The packages presented correspond to the components described in the logical view (Figure 5.41). The names given in both views differ only on the case used.

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.

5.4 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.

5.4.1 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 or not 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 IoT 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 a 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 distinct IoT services. On the order hand the SOA and Microservices architecture are preferred since they are open for extension - Jacobs and Casey 2022.

The last two concerns are related to the scalability of the solution. A Monolithic Backend can be scaled up by increasing the resources - RAM, CPU, GPU and Dick Capacity - of the physical server where the solution is deployed, this is commonly referred as Vertical Scaling. A SOA or Micro Service Backend Architecture can be scaled up by increasing the number of physical servers where the solution is deployed, this is commonly referred as Horizontal Scaling. One can also deploy various independent instances of the same solution and each instance would be assign to a set of costumers. This option is crucial and always possible once the business grows and starts to assist various customers.

The following picture, Figure 5.47, summarizes how each architect scales, the SOA behaves similarly to the microservices architecture presented.

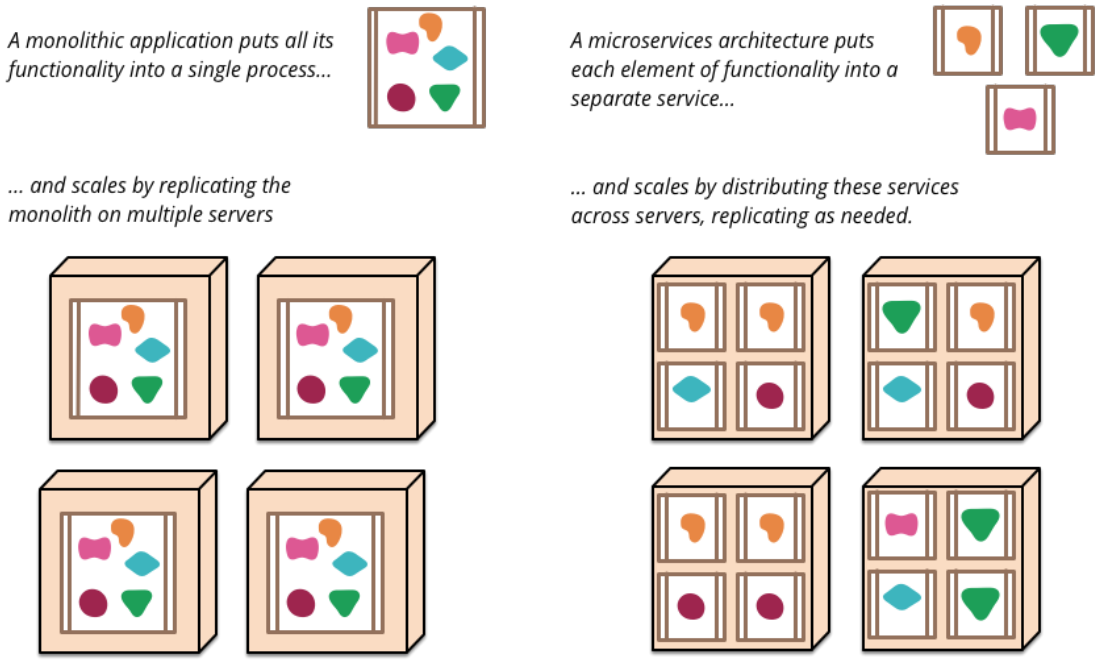

Figure 5.47: Monoliths and Microservices by Fowler and J. Lewis 2014

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

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

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

- There are well defined boundaries between the various business processes that the project needs to support;
- There is a need to scale the solution early on the road due to high volumes of IoT data to process and store;
- There are a high number of completely independent IoT services to develop;
- There are different types of costumers with diverse requirements regarding the deployment and development of the solution.

SOA was discarded since it focus on business functionality reuse instead of functional requirements segregation. With SOA each service is responsible only for one non functional requirement such as: auditing, security, logging, data storage, data presentation, business

process logic and others. All these services usually communicate via Enterprise Service Bus (ESB). A new functional requirement or business process requires every type of service to be modified. Microservices on the other hand are separated by functional requirements and each service is responsible for storing data, presenting data, logging and everything else deemed necessary. Microservices are more easily extended when/if needed compared with SOA since the focus is on loose coupling services and not highly reusable services.

5.4.2 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 requirements discussed in *****TODO***** enhance the need to develop a product that can be fully extensible and yet close for modifications, strictly following the Open/Close Principle (OCP). 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 inadvertently 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.

5.4.3 User Authorization/Authentication

User Authorization and Authentication is an important aspect of the solution. During the requirements elicitation, mentioned in *****TODO*****, it was clear that several different levels of access had to be given to Tenants, this levels of access 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 Authorization Server;
- External Authorization Server;
- External Authorization Server with Internal Permissions Server;
- External Authorization Server with Internal Oauth2 Server;

The fourth option was the approach taken.

Internal Authorization Server

By creating an Internal Authorization 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 5.48,presents the normal environment flow for this alternative.

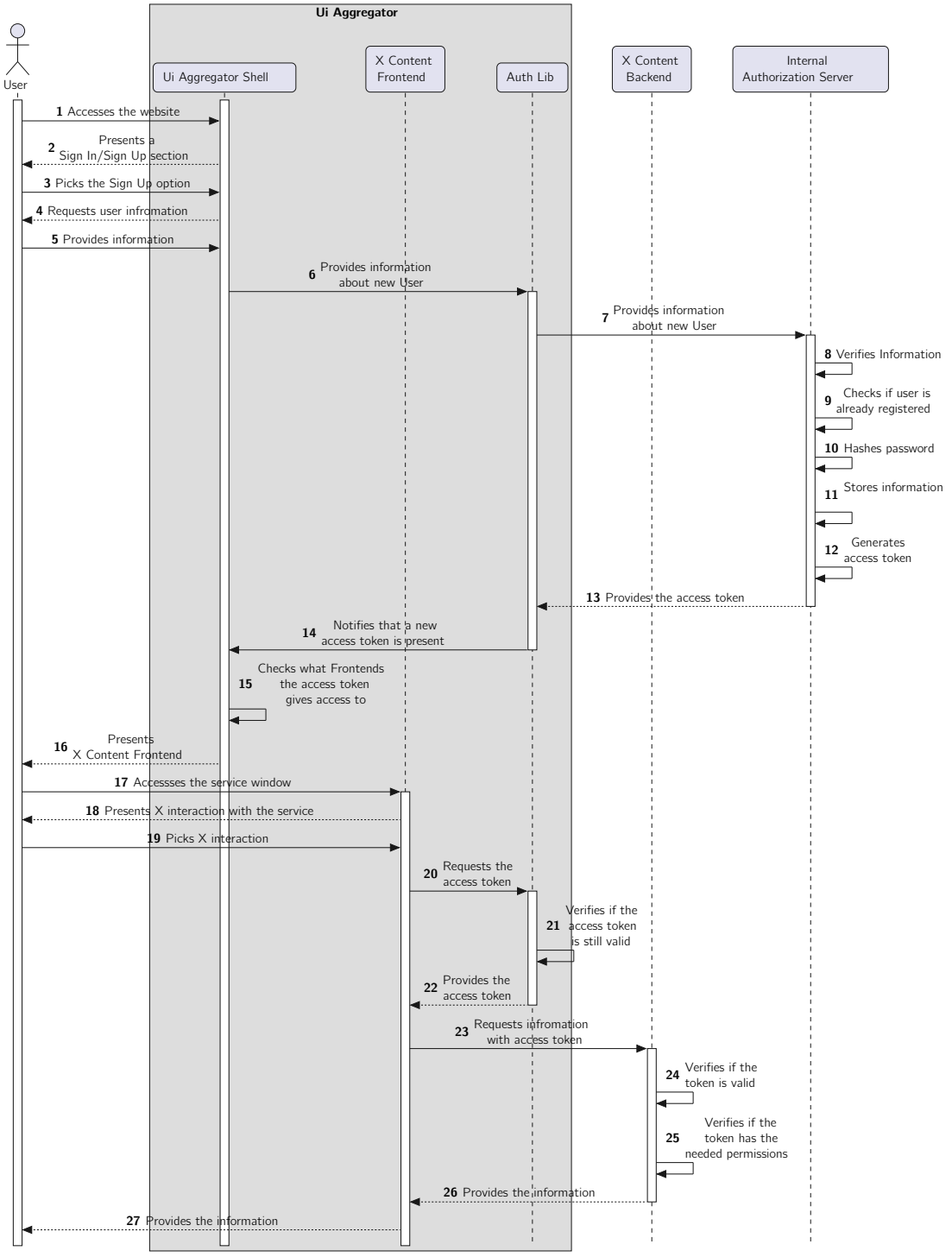

Figure 5.48: User Authorization/Authentication - Internal Authorization Server Alternative - Sequence Diagram

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.

External Authorization Server

By using an external Authorization Server there would be no need to store user credentials or permissions. This services are commonly identified as CIAM solutions.

The following diagram, Figure 5.49, presents the normal environment flow for this alternative.

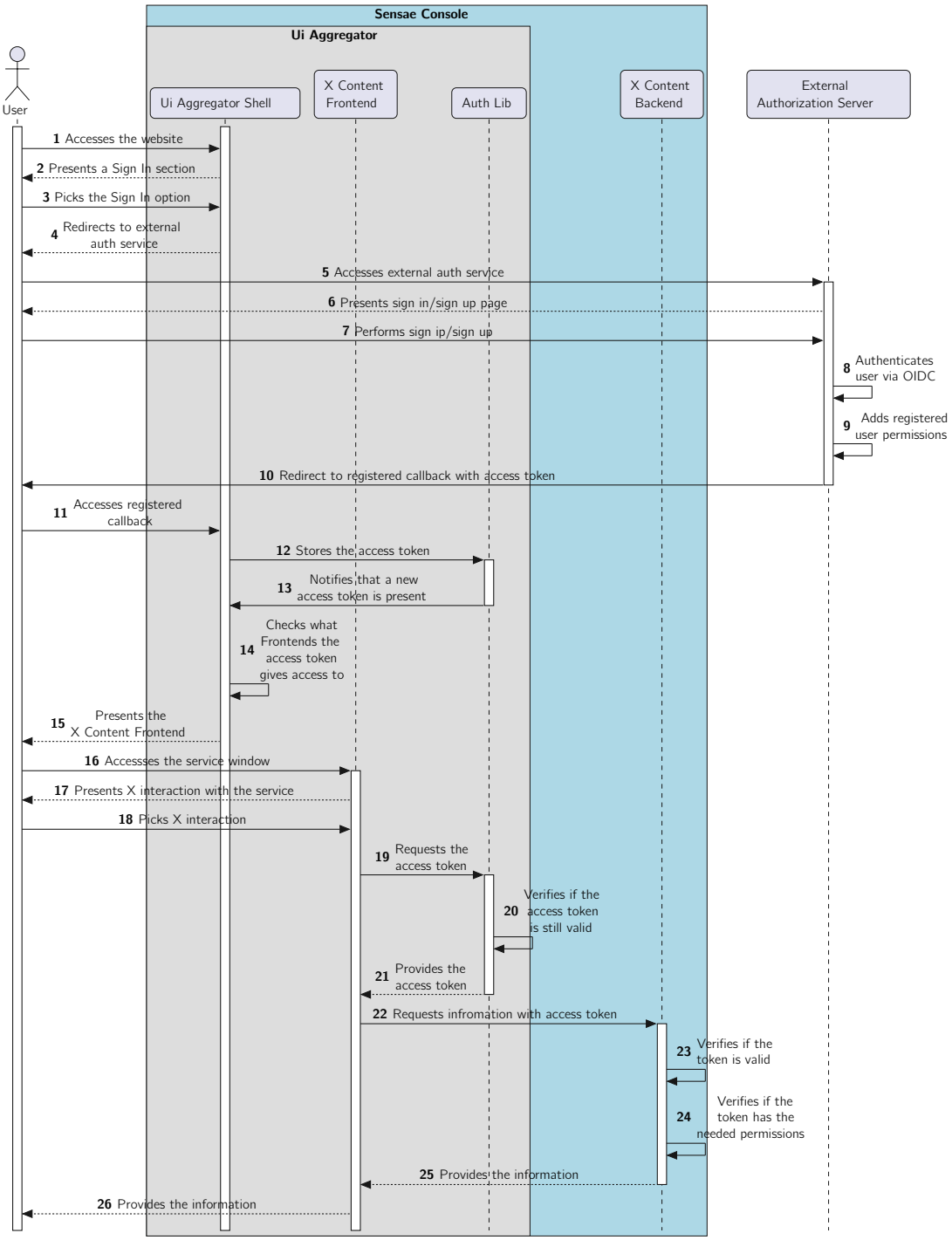

Figure 5.49: User Authorization/Authentication - External Authorization Server Alternative - Sequence Diagram

This approach would create a strong dependency to the CIAM solution used since all user permissions and credentials would have to be managed by the CIAM solution. Some of this services are: (i) *Auth0 Customer Identity*, (ii) *Google Identity Platform*, (iii) *Okta Customer Identity*, (iv) *Amazon Cognito* and (v) *Azure Active Directory (Azure AD)*.

The platform Auth0 was tested and is capable of registering user roles and permissions according to the 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.

External Authorization Server with Internal Permissions Server

By using an external Authorization Server there would be no need to store user credentials, the user permissions would then be managed internally via a *Permissions Server*.

The following diagram, Figure 5.50, presents the normal environment flow for this alternative.

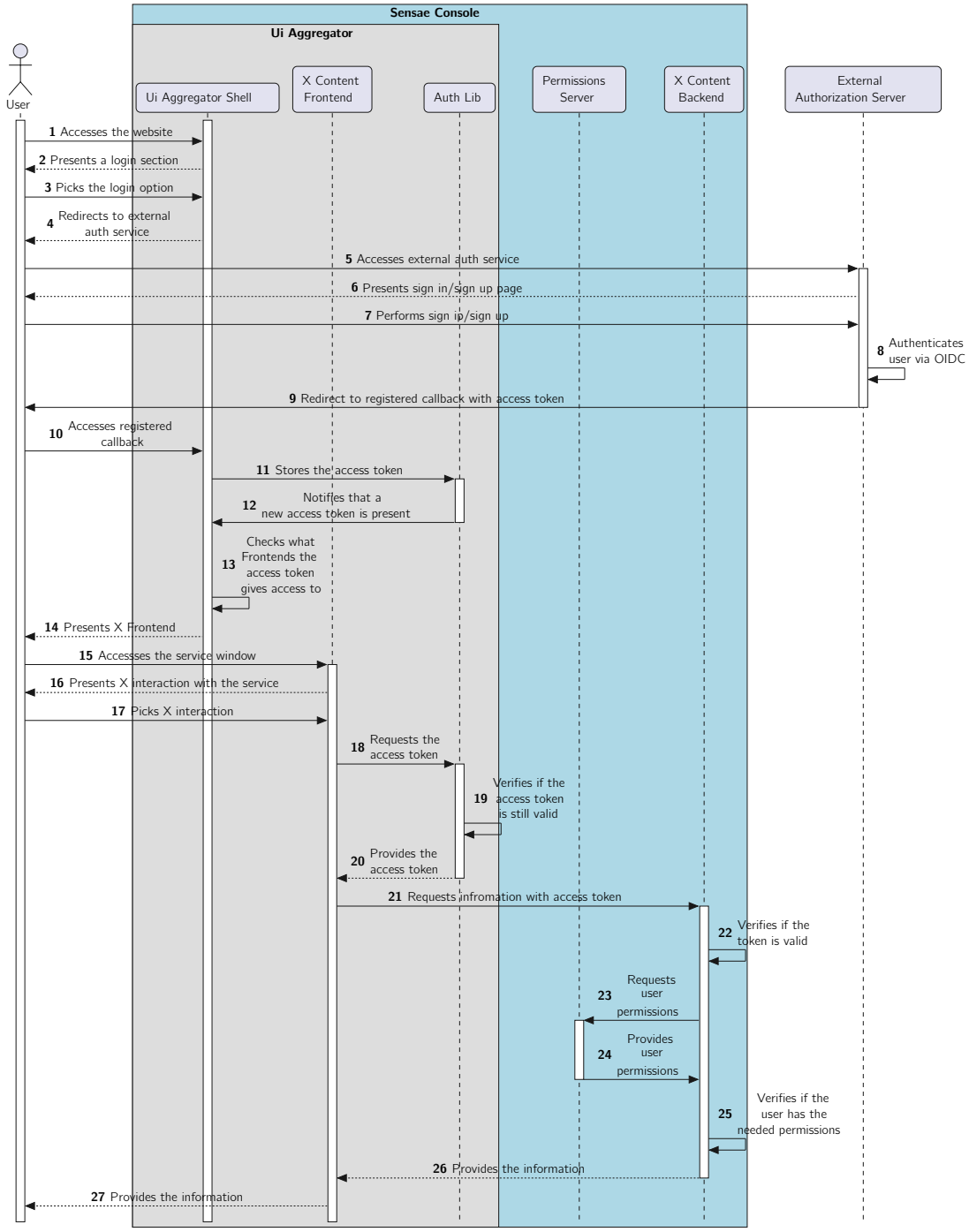

Figure 5.50: User Authorization/Authentication - External Authorization Server with Internal Permissions Server Alternative - Sequence Diagram

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 permissions would be managed internally. This approach would require any backend to query the *Permissions Server* for user permissions so that it could verify if the user was authorized to preform the requested action or not. This would therefore linger down the performance of

the system since each action would have to be verified according to the *Permissions Server* information.

External Authorization 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 5.30 and Figure 5.31 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.

5.4.4 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 5.25 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 simply, 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.

5.4.5 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:

- **First option**: Data Flow Containers directly access the Database related to their Context;
- **Second option**: Data Flow Containers request information to their context's Configuration Scope Container via synchronous calls;
- **Third option**: Data Flow Containers are feed updates to their context configurations via asynchronous calls and store this information;
- **Fourth option**: A shared, in memory, database is kept, Configuration Scope writes to it and Data Flow Scope queries information from it;
- **Fifth option**: 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.

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 5.51 describes how this option functions.

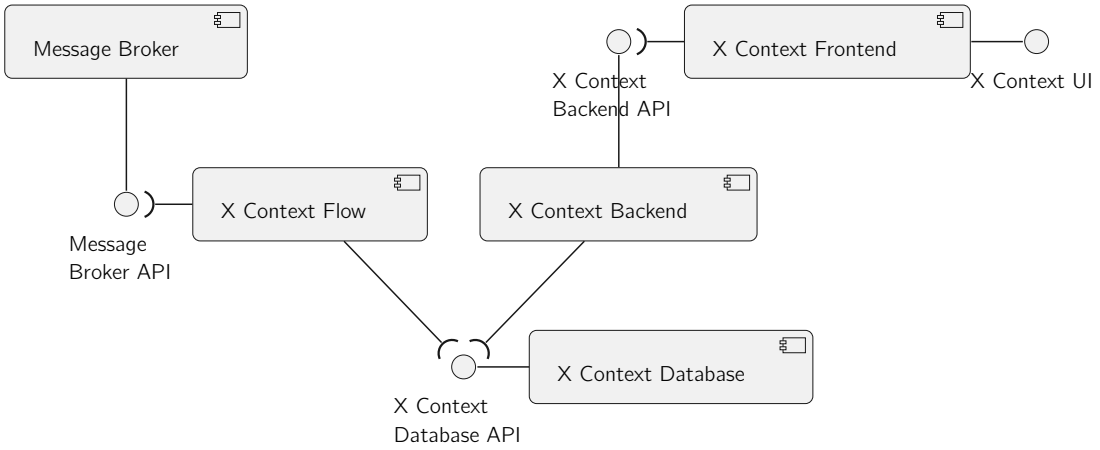

Figure 5.51: Internal Communication - First Option - Logical View Diagram

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.

Second Option

This option ensures that the Data Flow Containers are kept updated querying information from a Representational State Transfer (REST) Application Programming Interface (API) provided by the Configuration Containers. The logical view diagram in Figure 5.52 describes how this option functions.

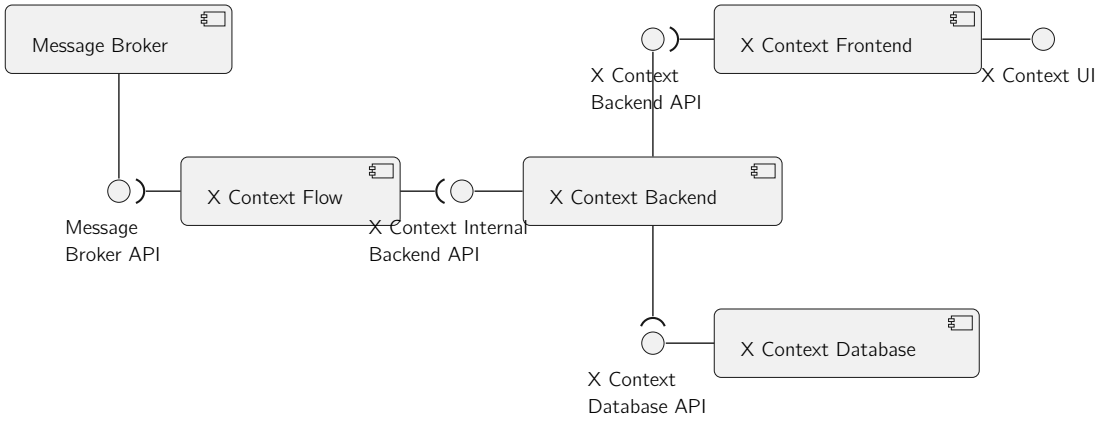

Figure 5.52: Internal Communication - Second Option - Logical View Diagram

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.

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 5.53 describes how this option functions.

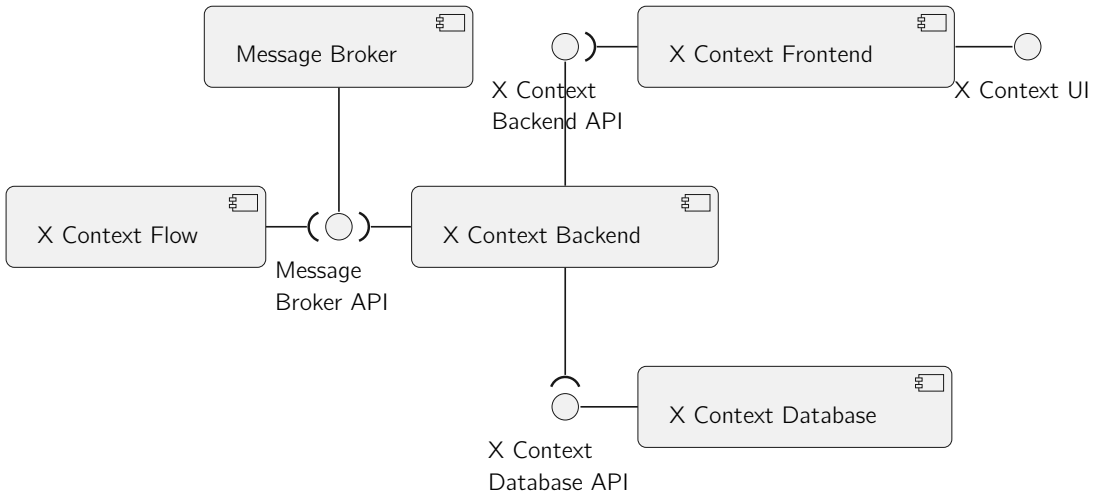

Figure 5.53: Internal Communication - Third Option - Logical View Diagram

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.

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 5.54 describes how this option functions.

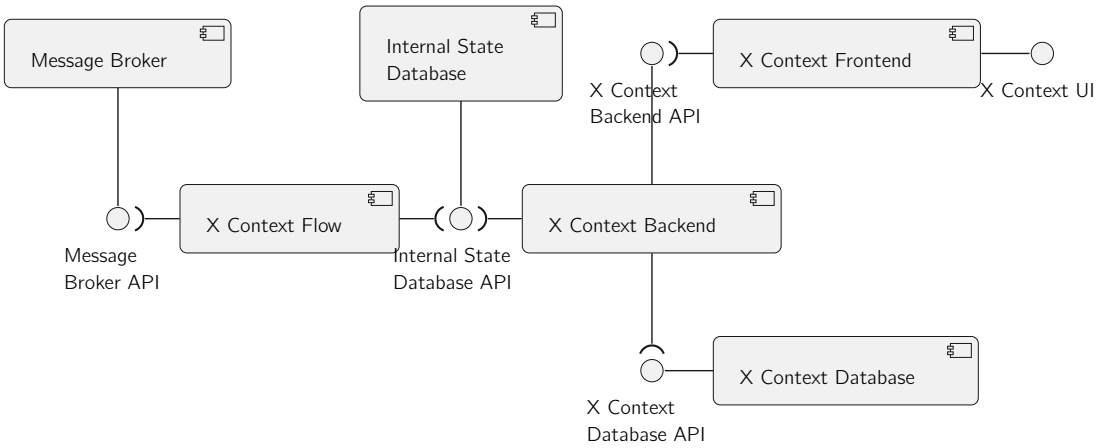

Figure 5.54: Internal Communication - Fourth Option - Logical View Diagram

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

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

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 5.55 describes how this option functions.

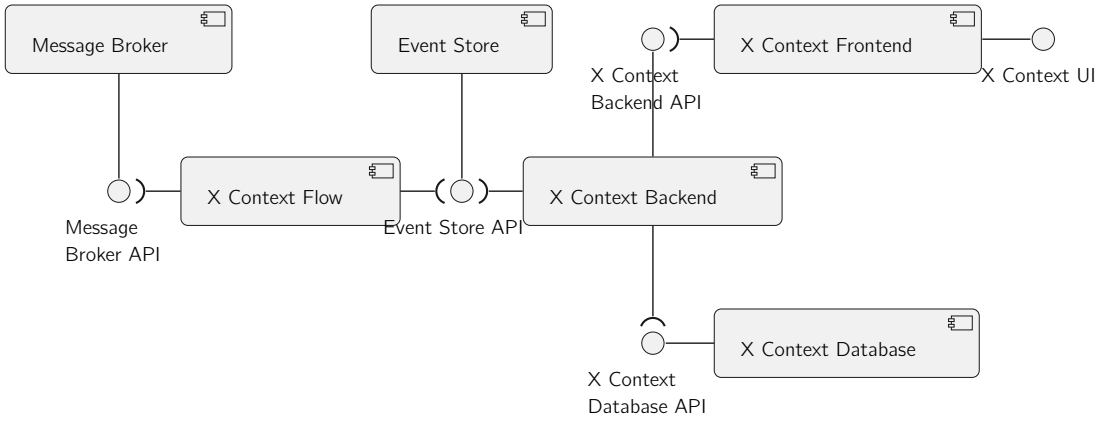

Figure 5.55: Internal Communication - Fifth Option - Logical View Diagram

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.

5.5 Synopsis

This chapter presented to the reader 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 thought the Solution;
- Expose a GraphQL API On Backend Services;
- 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.

Programming Language Used

As described in the development view of Section 5.3.1, a package named *iot-core*, an idealized SDK for **Sensae Console**, was developed to define the information that flows inside the system. Since this project is still in the early stages, the *iot-core* package was only 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 the first SDK in *Java* are:

- It's the programming language that the author is most familiarized with;
- 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*.

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.

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 thought 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.

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.

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

Table 6.1: Comparison of Angular with 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 the table above.

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.

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 5.44 resembles this two concepts.

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 with *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 Expose a GraphQL API On Backend Services

The API discussed in this section refers to the interfaces exposed to the outside world by backend containers of the Configuration and Service 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 5.4.5 and 5.4.4, the technology ultimately chosen was 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.

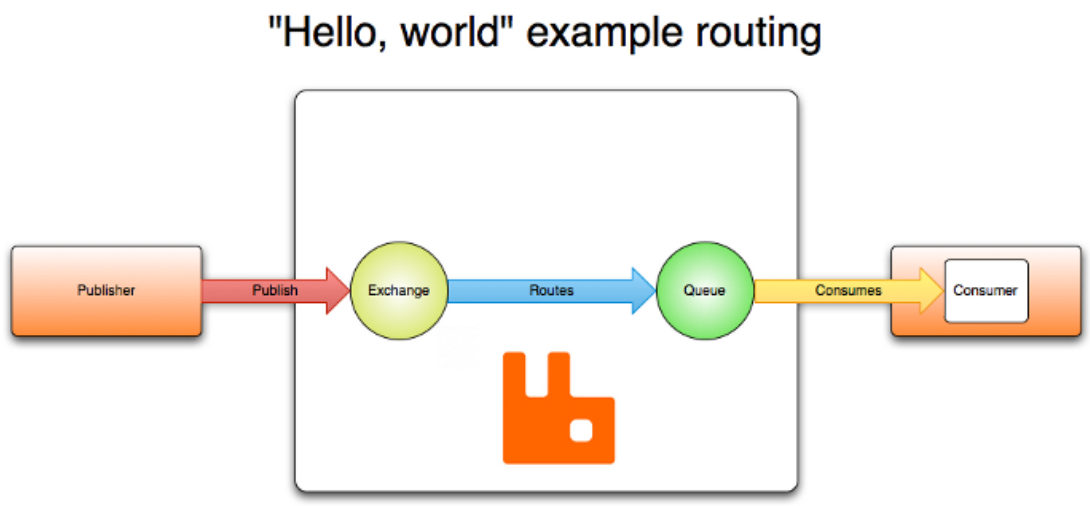

Figure 6.1: AMQP 0.9.1 Protocol Concepts by VMWare 2022a

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 5.3 of Section 5.2.2. This key defines the single destination of a data unit. For a data unit to have various dynamic service destinations there would be a need to either:

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

To tackle this issue, another Message Broker, such as *Pulsar*, with its own protocol, can be used in the future. This Message Broker answers the drawback describe above by allowing Consumers to subscribe to multiple topics (equivalent to RabbitMQ' routing keys) on the basis of a regular expression (regex), as stated in *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 5.3 of Section 5.2.2, 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 lead to the encoding of routing keys in a single character when possible.

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). This formats, when compared with *Protocol Buffers* or "Thrift", sacrifice size and de/serialization performance for human readability as stated in Sumaray and Makki 2012.

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

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

By leveraging this 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".

"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."

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 number of sites. 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.

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 5.2.3 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, 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.

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 (...) 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 5.21 and 5.25. This context, intended to mimic a Data Lake[1], stores any type of data for future use.

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

The 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

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

Elmasri et al. 2000, this features is useful once a single node isn't capable of withstanding all data collected while providing fast access to it.

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, this databases "can maintain high-performance of data analysis and business intelligence processing".

This 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*.

Graph-based Database Usage

Even tho this data model was ultimately not used, the author deemed relevant to mention 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. The features that stud out from other rule engines were:

- Supports for sliding windows of time;
- 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.

[2]taken 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 Milesight [3], Sensational-Systems, Helium, [4] and [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 using *Docker*.

As stated in *Docker overview*, 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 5.3.2 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 is run and not not all the processes inherent to an OS. Containers execute calls directly to the kernel running on the physical machine and can be seen, unlike virtual machines with their own kernel, as a normal process.

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*, “Compose is a tool for defining and running multi-container Docker applications”.

Since there is no need to automatically scale the solution it was decided to use a docker compose in production inserted of tools like Kubernetes.

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

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.

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

Table 6.2: Technologies Comparison - Reverse Proxy Web Server

The details about *Nginx* 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 of 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 6.2. 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*).

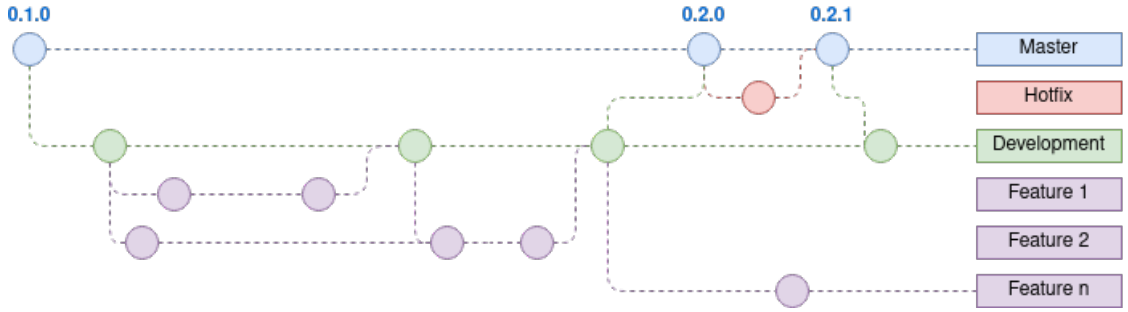

Figure 6.2: Branching Model

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 6.3, is preferred. With this model one can release only the altered containers and not the entire system.

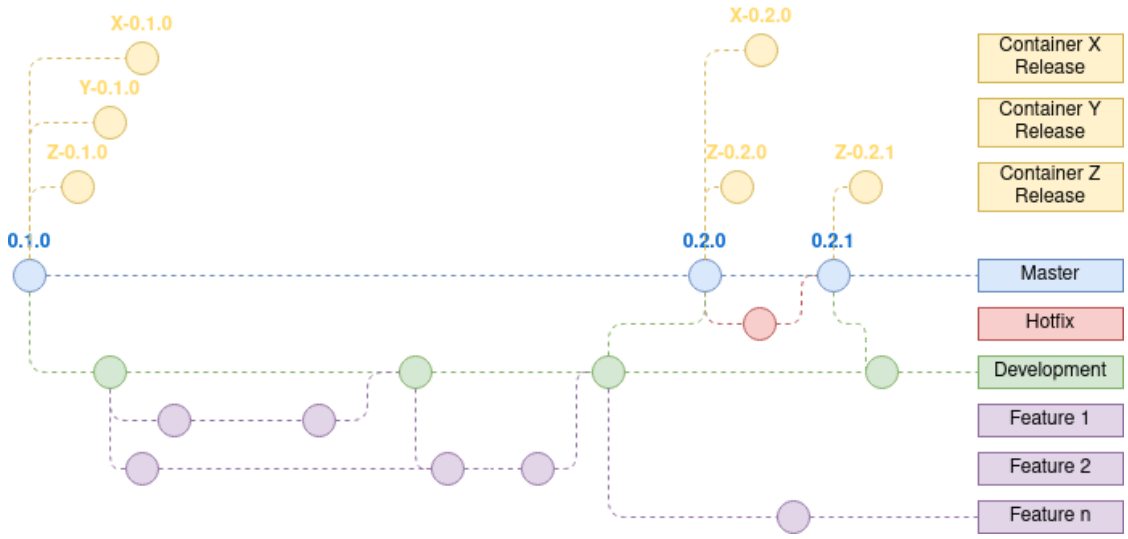

Figure 6.3: Future Branching Model

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 6.4, and the other is concerned with the current state of each issue, feature and bug, Figure 6.5.

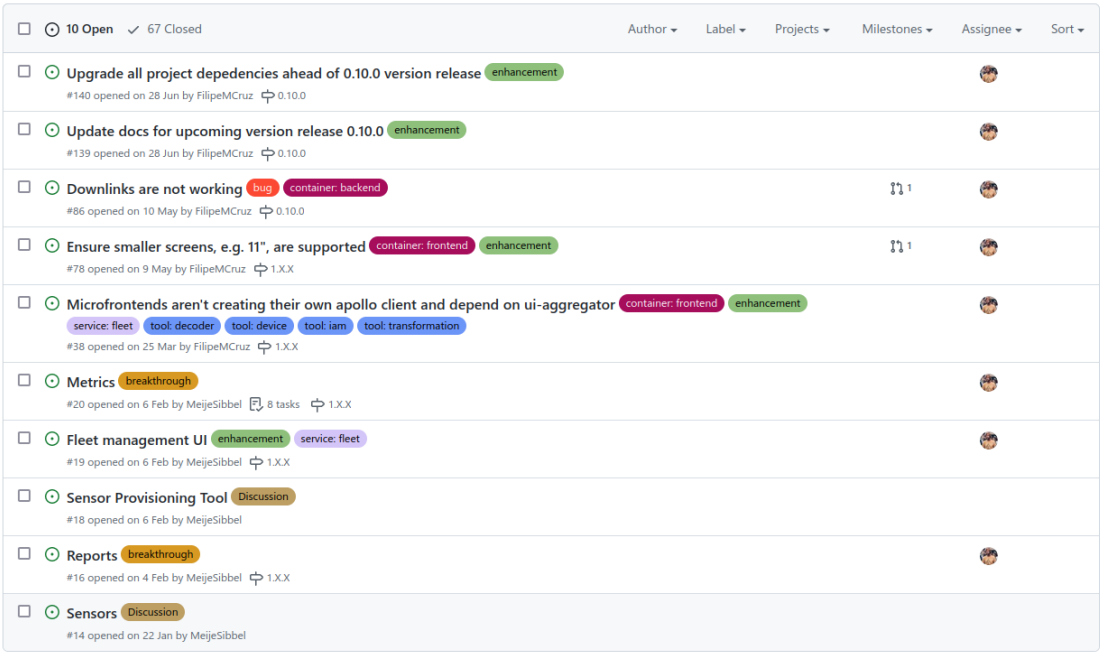

Figure 6.4: Github Issues

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 6.5. 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.

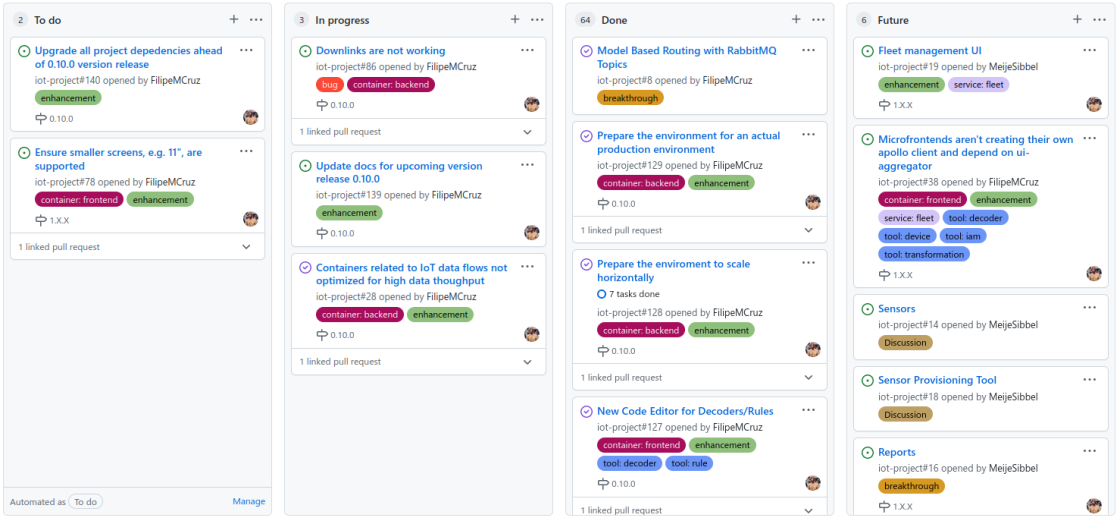

Figure 6.5: Github Issues Project Board

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.

```yaml
name: IoT Core - Continuous Delivery to maven central
on:
  push:
    tags:
      - '**'
      - '*'
jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - 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 }}" "
     ↪ ${{ secrets.microsoft_audience }}" "${{ secrets.google_audience
     ↪ }}" "${{ 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 -I % sh -c 'cd % && mvn test && \
18     echo % >> ../../reports/backend-test-pass.log || \
19     echo % >> ../../reports/backend-test-fail.log'
20
21 test ! -f ../reports/backend-test-fail.log
22
23 cd ../frontend-services || exit
24
25 npm install
26 npm run test-all
27
28 ./../scripts/build-images.sh
29
30 docker-compose -f ../docker-compose.test.yml up -d --build
31
32 sleep 60
33
34 npm run e2e-all
35
36 docker-compose -f ../docker-compose.test.yml down
```

Listing 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 Section;
- Build and start an environment similar to the production one - lines **28** to **32**;
- Preform end to end tests against the test environment - **34**. The script mentioned as *e2e-all* is: *nx run-many –all –target=e2e –parallel=1*. This script runs all end-to-end tests of the frontend apps using *Nx*, as mentioned in 6.1.2 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 Sonartype 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, (vii) Developer Information, (viii) SCM Information.

In the References***TODO*** Appendix the full *pom.xml* is presented.

6.2 Technical Description

This section guides the reader though **Sensae Console** 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;

- Sensae Console Services;
- Sensae Console Device Integration;

6.2.1 Sensae Console UI

In this subsection the UI is presented.

The Figure 6.6 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.

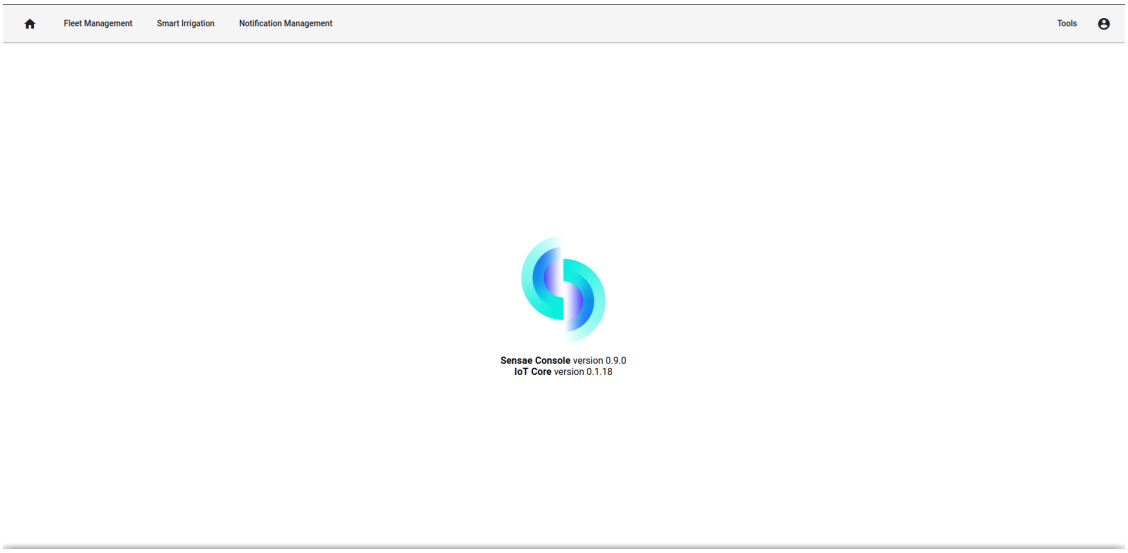

Figure 6.6: Sensae Console Home Page

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 6.7. 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 an see the history of measures of each device.

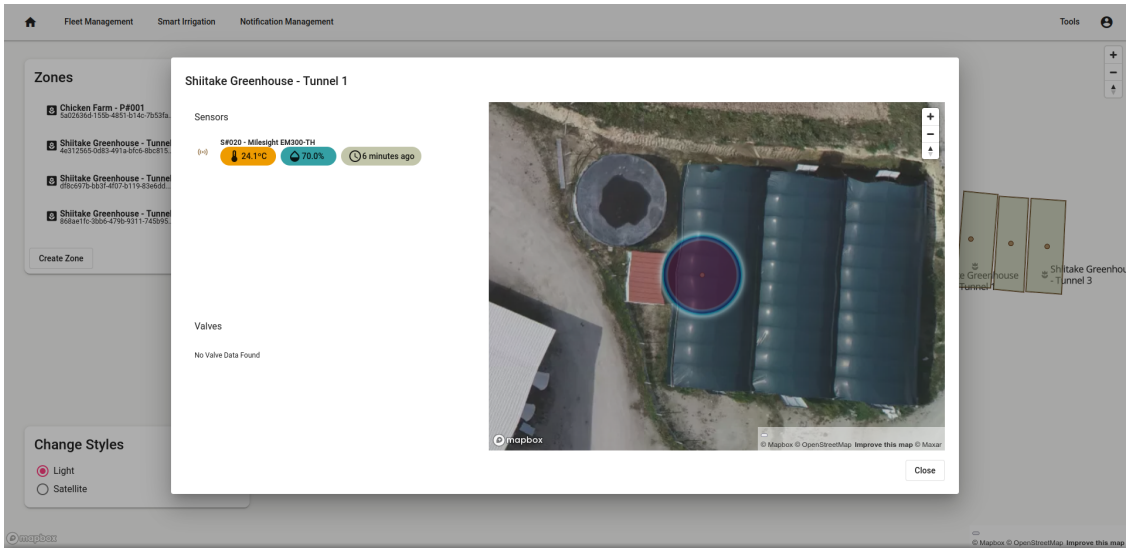

Figure 6.7: Sensae Console Smart Irrigation Page

From the home page the user can also access configuration pages, as an example the **Device Management Page** is displayed in Figure 6.8.

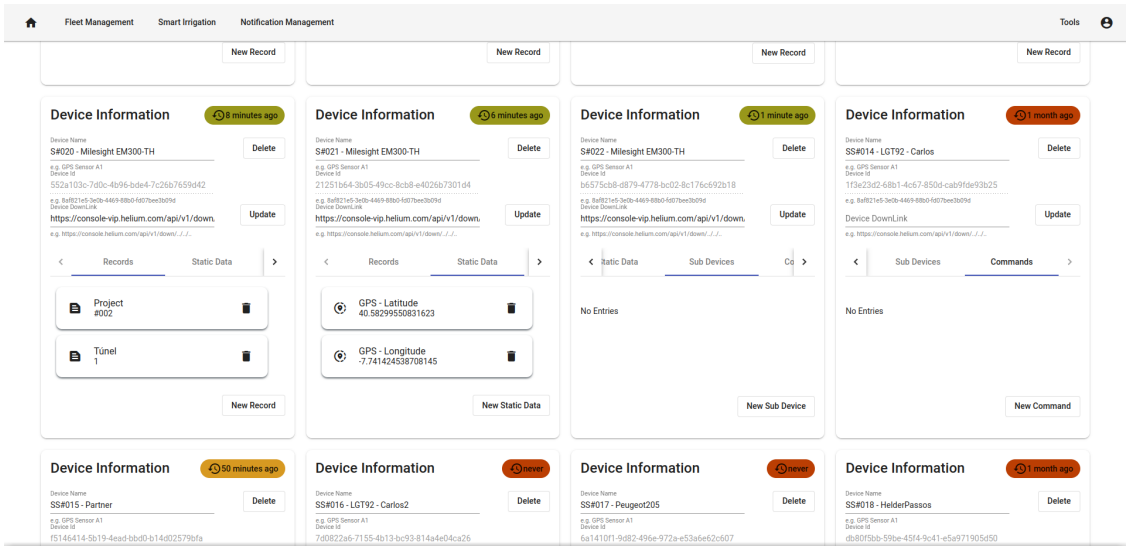

Figure 6.8: Sensae Console Device Management Page

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

Other relevant pages are presented in the Appendix D.

6.2.2 Sensae Console Custom Maps

This section describes how custom maps where built to fit the solution needs. Some costumers facilities were not present in the satellite view of *Google Maps* or *Mapbox GL JS*. A custom map, with the missing facilities, was built using satellite images taken with a

drone. The images where processed with *ArcGIS* and transformed in *.tiff* files that could be incorporated in the basic satellite layer of *Mapbox GL JS*.

The following image, Figure 6.9 presents the new map with the costumer facilities in a greener tone than the rest of the map. This map was used to display three greenhouses and a chicken farm that belong to a costumer. This map is currently in use by the Smart Irrigation Service.

Figure 6.9: 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.

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

type Query {
    history(filters: HistoryQueryFilters) : [SensorDataHistory]
    fetchIrrigationZones : [IrrigationZone]
    fetchLatestData(filters: LatestDataQueryFilters): [SensorData]
}

type Mutation {
    createIrrigationZone(instructions: CreateIrrigationZoneCommand) :
    IrrigationZone
    updateIrrigationZone(instructions: UpdateIrrigationZoneCommand) :
    IrrigationZone
```

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

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 *****TODO*****.

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 5.3. The *infoType* can have two values: ENCODED or DECODED. Depending on this value the message is routed to *Data Decoder Flow* or *Data Processor Flow* as described in Figure 5.21.

The *channel* parameter indicates the final service that it is destine 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 access 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 Unit**s and turn them into **Alert**s.

Valid **Data Unit**s 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.
       DeviceRecordEntryDTO;
6  import pt.sharespot.iot.core.data.model.properties.PropertyName;
7  import pt.sharespot.iot.core.alert.model.AlertBuilder;
8  import pt.sharespot.iot.core.alert.model.CorrelationDataBuilder;
9  import pt.sharespot.iot.core.alert.model.AlertLevel;
10 import java.util.List;
11 import java.util.UUID;
12
13 global pt.sharespot.iot.core.alert.model.AlertDispatcherService
       dispatcher;
14
15 dialect "mvel"
16
17 declare StoveSensor
18     @role( event )
19     deviceId : UUID
20 end
21
22 declare StoveSensorData
23     @role( event )
24     deviceId : UUID
25     dataId : UUID
26     temperature : Float
27     humidity : Float
28 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.
      relativePercentage);
19        insert(reading)
20 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 "+ $s
       .deviceId +" detected low humidity/high temperature")
14                             .setLevel(AlertLevel.ADVISORY)
15                             .setContext(CorrelationDataBuilder.create()
16                                 .setDeviceIds($s.deviceId)
17                                 .setOther("Project #002")
18                                 .build())
```

```
19                                    .build());
20 end
```

Listing 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 then 26 ºC 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 store as a Notification, the *DeviceId*s associated to it are used to determine what domains will have access to the Notification. If no *DeviceId*s are associated only the root domain will have access to it.

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-TH[6].

```
1  const decodePayload = (payload, port) =>
2      ({"0": decoder(base64ToHex(payload), port)});
3
4  const base64ToHex = (() => {
5      const values = [],
6      output = [];
7
8      return function base64ToHex(txt) {
9          if (output.length <= 0) populateLookups();
10         const result = [];
11         let v1, v2, v3, v4;
12         for (let i = 0, len = txt.length; i < len; i += 4) {
13             v1 = values[txt.charCodeAt(i)];
```

[6] https://www.milesight-iot.com/lorawan/sensor/em300-th

```
            v2 = values[txt.charCodeAt(i + 1)];
            v3 = values[txt.charCodeAt(i + 2)];
            v4 = values[txt.charCodeAt(i + 3)];
            result.push(
                parseInt(output[(v1 << 2) | (v2 >> 4)], 16),
                parseInt(output[((v2 & 15) << 4) | (v3 >> 2)], 16),
                parseInt(output[((v3 & 3) << 6) | v4], 16)
            );
        }
        if (v4 === 64) result.splice(v3 === 64 ? -2 : -1);
        return result;
    };
    function populateLookups() {
        const keys =
            "
    ↪ ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/="
    ↪ ;
        for (let i = 0; i < 256; i++) {
            output.push(("0" + i.toString(16)).slice(-2));
            values.push(0);
        }
        for (let i = 0; i < 65; i++) values[keys.charCodeAt(i)] = i;
    }
})();

function decoder(bytes, port) {
    let decoded = {}, temperature = {}, airHumidity = {}, battery = {};
    for (let i = 0; i < bytes.length;) {
        let channel_id = bytes[i++];
        let channel_type = bytes[i++];
        if (channel_id === 0x01 && channel_type === 0x75) {
            decoded.battery = battery;
            battery.percentage = bytes[i];
            i += 1;
        } else if (channel_id === 0x03 && channel_type === 0x67) {
            decoded.temperature = temperature;
            temperature.celsius = readInt16LE(bytes.slice(i,i+2))/10;
            i += 2;
        } else if (channel_id === 0x04 && channel_type === 0x68) {
            decoded.airHumidity = airHumidity;
            airHumidity.relativePercentage = bytes[i] / 2;
            i += 1;
        } else {
            break;
        }
    }
    return decoded;
}
const readUInt16LE = bytes => (bytes[1] << 8) + bytes[0] & 0xffff;

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

```
        name: dataUnit.name,
        downlink: dataUnit.downlink_url,
    },
    measures: decodePayload(dataUnit.payload, dataUnit.port),
});
```

Listing 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 *0*. The function *base64ToHex* is the function that reads a Base 64 string and transforms it into a Hex Array - to reduce bandwidth the device normally encodes and sends data as a base 64 string. The function *decoder*, *readInt16LE* and *readUInt16LE* were adapted from the TTN decoder[7] of this device.

6.2.7 Sensae Console Database Configuration

The solution designed relies on various databases, and as discussed in Section 6.1.6 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.

```
create table if not exists public.transformation
(
    persistence_id bigint generated by default as identity
        primary key,
    device_type    varchar(255)
        constraint unique_type_constrain
            unique
);

create table if not exists public.property_transformation
(
    persistence_id                bigint generated by default as
    identity (maxvalue 2147483647)
        primary key,
    value                         integer           not null,
    old_path                      varchar(255),
    transformation_persistence_id bigint
        constraint ref_transformation_constrain
            references public.transformation,
    sub_sensor_id                 integer default 0 not null
);
```

Listing 6.9: Initialization Script Segment for Data Processor Database

[7] https://github.com/Milesight-IoT/SensorDecoders/blob/master/EM300_Series/EM300-TH

This script defines two simple tables, *transformation* and *property_transformation*, following the concepts defined in Section 5.2.3.

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

Listing 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 replace 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 5.3.2 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 that the code written. The second stage is the one that runs the service. It only contains the Java Development Kit (JDK) and the compiled application.

The third sample, Listing 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 5.3.2, *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.

```yaml
services:
  data-processor-frontend:
    build:
      dockerfile: docker/data-processor-frontend/Dockerfile
      context: frontend-services
    image: data-processor-frontend
    volumes:
      - /etc/letsencrypt:/etc/letsencrypt/
      - /etc/nginx/ssl:/etc/nginx/ssl/
    networks:
      - sensae-network
    ports:
      - 443
    depends_on:
      - data-processor-master-backend
  data-processor-master-backend:
    build: backend-services/data-processor-master-backend
    image: data-processor-master-backend
    volumes:
      - ./secrets/keys:/etc/ssh/app
    environment:
      spring_profiles_active: prod
    env_file:
      - ./secrets/prod/data-processor-master-backend.env
    networks:
      - sensae-network
    ports:
      - 8080
  data-processor-database:
    build: databases/data-processor-database
    container_name: data-processor-database
    env_file:
      - ./secrets/prod/data-processor-database.env
    networks:
      - sensae-network
    ports:
      - 5482:5432
    volumes:
      - ./databases-data/prod/data-processor-database:/var/lib/
    ↪ postgresql/data/
  data-processor-flow:
    build: backend-services/data-processor-flow
    image: sensae/data-processor-flow
    env_file:
        - ./secrets/prod/data-processor-flow.env
    networks:
        - sensae-network
networks:
  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,
       proxy-revalidate, max-age=0';
19     }
20
21     location /smart-irrigation/graphql {
22         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/
    subscriptions;
34          proxy_http_version 1.1;
35          proxy_set_header Upgrade $http_upgrade;
36          proxy_set_header Connection "Upgrade";
37          proxy_set_header Host $host;
38          proxy_read_timeout 6000;
39          proxy_send_timeout 6000;
40          proxy_redirect off;
41          proxy_set_header x-forwarded-port 443;
42          proxy_set_header x-forwarded-proto https;
43      }
44
45      location / {
46          try_files $uri $uri/ /index.html;
47      }
48
49      if ($scheme != "https") {
50          return 301 https://$host$request_uri;
51      } # managed by Certbot
52  }
```

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:

```
export SENSAE_MAPBOX_ACCESS_TOKEN=
export SENSAE_MAPBOX_SIMPLE_STYLE=
export SENSAE_MAPBOX_SATELLITE_STYLE=
export SENSAE_BROKER_USERNAME=
export SENSAE_BROKER_PASSWORD=
export SENSAE_COMMON_DATABASE_PASSWORD=
export SENSAE_DATA_STORE_USER_PASSWORD=
export SENSAE_DATA_STORE_ROOT_PASSWORD=
export SENSAE_AUTH_PATH_PUB_KEY=
export SENSAE_AUTH_PATH_PRIV_KEY=
export SENSAE_AUTH_ISSUER=
export SENSAE_AUTH_AUDIENCE=
export SENSAE_DATA_AUTH_KEY=
export SENSAE_AUTH_EXTERNAL_MICROSOFT_AUDIENCE=
export SENSAE_AUTH_EXTERNAL_GOOGLE_AUDIENCE=
export SENSAE_SMS_TWILIO_ACCOUNT_SID=
export SENSAE_SMS_TWILIO_AUTH_TOKEN=
export SENSAE_SMS_SENDER_NUMBER=
export SENSAE_SMS_ACTIVATE=
export SENSAE_EMAIL_SENDER_ACCOUNT=
export SENSAE_EMAIL_SUBJECT=
export SENSAE_EMAIL_SENDER_PASSWORD=
export SENSAE_EMAIL_SMTP_HOST=
export SENSAE_EMAIL_SMTP_PORT=
export SENSAE_EMAIL_ACTIVATE=
export SENSAE_PROD_PUBLIC_DOMAIN=
export SENSAE_ADMIN_EMAIL=
```

Listing 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.

```
#!/usr/bin/sh

ROOT_DIR=$(git rev-parse --show-toplevel)

cd "$ROOT_DIR"/project || exit

. ./secrets/prod.conf

SECRET_BACK=secrets/templates/prod/backend-services
SECRET_FRONT=secrets/templates/prod/frontend-services
SECRET_DB=secrets/templates/prod/databases

BACK_PREFIX=secrets/prod
FRONT_PREFIX=frontend-services/apps
FRONT_SUFFIX=src/environments/environment.prod.ts

envsubst < $SECRET_BACK/alert-dispatcher-backend.env > \
     $BACK_PREFIX/alert-dispatcher-backend.env
# and all other backend services
envsubst < $SECRET_BACK/data-validator.env >
     $BACK_PREFIX/data-validator.env

envsubst < $SECRET_FRONT/device-management-frontend.ts > \
```

```
 $FRONT_PREFIX/device-management-frontend/$FRONT_SUFFIX
# and all other frontend services
envsubst < $SECRET_FRONT/ui-aggregator.ts > \
 $FRONT_PREFIX/ui-aggregator/$FRONT_SUFFIX

envsubst < secrets/templates/prod/message-broker/message-broker.env > \
 $BACK_PREFIX/message-broker.env

envsubst < $SECRET_DB/data-decoder-database.env > \
 $BACK_PREFIX/data-decoder-database.env
# and all other databases
envsubst < $SECRET_DB/rule-management-database.env > \
 $BACK_PREFIX/rule-management-database.env
```

Listing 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 Sensae Console Services

This section refers how services interact with the **Data Flow Scope**, this was briefly mentioned in Section 5.2.2.

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 Unit**s and (ii) **Alert**s. Each of this information are defined by their routing keys as described in Table 5.3.

A service can also publish **Command**s.

The following sub sections will detail each service information needs.

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.

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.

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'*.

6.2.13 Sensae Console Device Integration

This section describes how devices can be connected to **Sensae Console**. As stated in Section *****TODO*****, currently the service 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 6.10 presents an example of the custom integration for the EM300-TH Device.

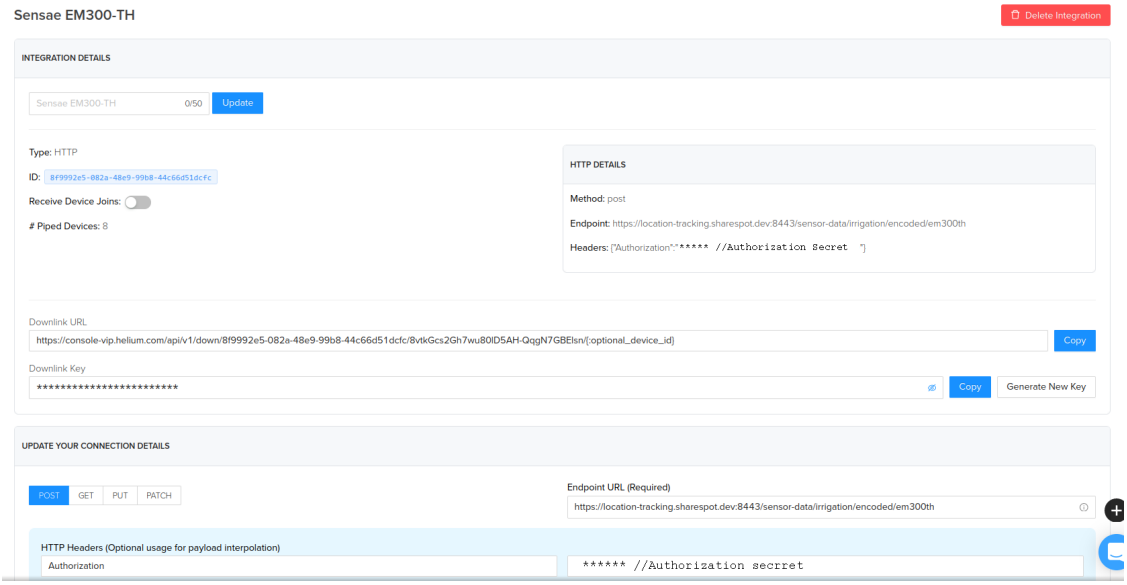

Figure 6.10: Helium Custom Integration Page

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 though *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 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 meet by the system;
- **End-to-End Testing**: Capture the need to verify that user interaction against common workflows works as expected in the system;
- **Acceptance Testing**: Capture the need to ensure that functional and non-functional requirements are accomplished;
- **Performance Testing**: Capture the need to verify how the environment behaves under heavy load. Their objective is to evaluate the stability, availability and reliability of the system;
- **Smoke Testing**: Capture the need to verify the overall state of the system before running heavier and extensive tests.

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

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

Architectural tests were added to the test suite to ensure that the 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 in various containers.

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*.

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

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

```
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 that the service needs 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 =
```

```
            new DeviceCommand('openValve', 'openValve', 'ldcn', 0, 70);
        const clone = deviceCommand.clone();
        expect(clone.id).toBe(deviceCommand.id);
        expect(clone.name).toBe(deviceCommand.name);
        expect(clone.ref).toBe(deviceCommand.ref);
        expect(clone.payload).toBe(deviceCommand.payload);
        expect(clone.port).toBe(deviceCommand.port);
    });
    it('should be invalid when it has no id', () => {
        const deviceCommand =
            new DeviceCommand('', 'openValve', 'ldcn', 0, 70);
        expect(deviceCommand.isValid()).toBeFalsy();
    });
});
```

Listing 6.20: Unit Test - Device Management Frontend Model Library

6.3.2 Integration Tests

This section, as an example, starts to focus on 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.

The code in Listing 6.21 verifies that the message broker can be reached by **Device Ownership Flow**.

```
@QuarkusTest
class DeviceInformationEmitterTest {

    @Inject
    DeviceInformationEmitter emitter;

    @Inject
    RoutingKeysProvider provider;

    @Inject
    @Any
    InMemoryConnector connector;

    @Test
    void testEmitterCanReachRabbitMQ() {
        var unknown = provider
            .getInternalTopicBuilder(RoutingKeysBuilderOptions.SUPPLIER)
            .withContainerType(ContainerTypeOptions.IDENTITY_MANAGEMENT)
            .withContextType(ContextTypeOptions.DEVICE_IDENTITY)
            .withOperationType(OperationTypeOptions.UNKNOWN)
            .build().orElseThrow();

        var deviceId = DeviceId.of(UUID.randomUUID());

        emitter.next(new DeviceTopicMessage(deviceId, unknown));

        var payload = connector.sink("egress-device-ownership")
                .received().get(0).getPayload();

        Assertions.assertNotNull(payload);
    }
```

```
32 }
```

Listing 6.21: Integration Test - Message Broker - **Device Ownership Flow**

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

The code in Listing 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
   ↪ 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. Theses 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 =
       ↪ {IntegrationTest.Initializer.class})
4  @ActiveProfiles(profiles = "test")
5  public abstract class IntegrationTest {
6      static class Initializer implements
7          ApplicationContextInitializer<ConfigurableApplicationContext> {
8          public void initialize(ConfigurableApplicationContext context) {
9              db.withDatabaseName("decoder");
10             TestPropertyValues.of(
11                 "spring.datasource.url=" + db.getJdbcUrl(),
12                 "spring.datasource.username=" + db.getUsername(),
13                 "spring.datasource.password=" + db.getPassword(),
14                 "spring.rabbitmq.host=" + mb.getHost(),
15                 "spring.rabbitmq.port=" + mb.getAmqpPort(),
16                 "spring.rabbitmq.username=" + mb.getAdminUsername(),
17                 "spring.rabbitmq.password=" + mb.getAdminPassword()
18             ).applyTo(context.getEnvironment());
19         }
20     }
21
22     @Container
23     public static PostgreSQLContainer<?> db =
24         DatabaseContainerTest.getInstance();
25
26     @Container
27     public static RabbitMQContainer mb =
28         MessageBrokerContainerTest.getInstance();
29
30     protected ResultSet performQuery(String sql) throws SQLException {
31         DataSource ds = getDataSource(postgresSQLContainer);
```

```
32          Statement statement = ds.getConnection().createStatement();
33          statement.execute(sql);
34          ResultSet resultSet = statement.getResultSet();
35
36          if (resultSet != null) resultSet.next();
37
38          return resultSet;
39      }
40  }
```

Listing 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              SensorTypeId.of("lgt92"), SensorTypeScript.of("asmc")));
37
38          var dto = (DataDecoderNotificationDTOImpl)
39              amqpTemplate.receiveAndConvert("info");
40
41          var type = DataDecoderNotificationTypeDTOImpl.UPDATE;
42
43          Assertions.assertEquals(type, dto.type);
44          Assertions.assertEquals("lgt92", dto.sensorType);
45          Assertions.assertEquals("asmc", dto.information.script);
46      }
47  }
```

Listing 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*, its 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(ContentType.JSON)
9              .contentType(ContentType.JSON)
10             .header("Authorization", "pass")
11             .post("/sensor-data/fleet/wrong/lgt92")
12             .then()
13             .statusCode(400)
14             .body("error", containsString(errorType));
15     }
16 }
```

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

Listing 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.

```
describe('ui-aggregator', () => {
    beforeEach(() => cy.visit('/'));
    it('should display welcome message for anonymous user', () => {
        cy.anonymous();
        cy.contains("Valid Credentials");
        cy.logout();
    });
});
```

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.

```
describe('ui-aggregator', () => {
    beforeEach(() => cy.visit('/'));
    it('should present various default domains', () => {
        cy.managerLogin();
        cy.goToIdentityPage();
        cy.contains("root");
        cy.get(".toggle").click();
        cy.contains("public");
        cy.contains("unallocated");
    });
});
```

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 / Service 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 / Service Scope** backend containers.

```
@AnalyzeClasses(packages = "pt.sensae.services")
public class ApplicationArchitectureTest {

    @ArchTest
    static final ArchRule architecture = Architectures
        .onionArchitecture()
        .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);
15
16      @ArchTest
17      static final ArchRule domainMustNotDependOnFrameworks =
18          ArchRuleDefinition.noClasses().that()
19              .resideInAnyPackage("..domain..")
20              .should().dependOnClassesThat()
21              .haveNameMatching("org.eclipse.")
22              .orShould().dependOnClassesThat()
23              .haveNameMatching("com.fasterxml.")
24              .orShould().dependOnClassesThat()
25              .haveNameMatching("com.google.")
26              .orShould().dependOnClassesThat()
27              .haveNameMatching("javax.")
28              .because("Domain should be free from Frameworks");
29  }
```

Listing 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

Chapter 8

Conclusion

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/.

Auth0 (2022). *Auth0 Customer Identity*. url: https://auth0.com/b2c-customer-identity-management.

Azure (2022). *Azure Active Directory (Azure AD)*. url: https://azure.microsoft.com/en-us/services/active-directory/.

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].

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.

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.

Docker (2022a). *Docker*. url: https://www.docker.com/.

– (2022b). *Docker overview*. url: https://docs.docker.com/get-started/overview/.

– (2022c). *Overview of Docker Compose*. url: https://docs.docker.com/compose/.

Drools (2022). *Drools*. url: https://www.drools.org/.

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.

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.
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 Maps*. url: https://mapsplatform.google.com/maps-products/.
– (2022c). *Protocol Buffers*. url: https://developers.google.com/protocol-buffers/.
– (n.d.). *Google Identity Platform*. url: https://cloud.google.com/identity-platform/.
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/.

Mercurial (2022). *Mercurial*. url: https://www.mercurial-scm.org/.

Microsoft (2022a). *Github*. url: https://www.github.com/.

– (2022b). *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/.

Mozzila (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.

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/.

Palermo, Jeffrey (2008). *The Onion Architecture*. url: https://jeffreypalermo.com/2008/07/the-onion-architecture-part-1/.

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.

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. 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.

– (2022). *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.

Slee, Mark, Aditya Agarwal, and Marc Kwiatkowski (2007). "Thrift". In: *Facebook white paper* 5.8, p. 127.

Sonartype (2022). *Why Do We Have Requirements?* url: https://central.sonatype.org/publish/requirements/.

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.

Urquhart, J. (2021). *Flow Architectures The Future of Streaming and Event-Driven Integration*. O'Reilly Media. isbn: 9781492075868.

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*.

```
{
  "dataId": "[uuid]",
  "reportedAt": "[long]",
  "device": {
    "id": "[uuid]",
    "name": "[string]",
    "downlink": "[string]",
    "records": [{
      "label": "[string]",
      "content": "[string]"
    }],
    "domains":  ["[uuid]"],
    "commands": {
      "[int]": [{
        "id": "[uuid]",
        "name": "[string]",
        "payload": "[base64 string]",
        "port": "[int]"
      }]
    }
  },
  "measures": {
    "[int]": {
      "airHumidity": {
        "gramsPerCubicMeter": "[float]",
        "relativePercentage": "[float]"
      },
      "airPressure": {  "hectoPascal": "[float]"  },
      "aqi": {  "value": "[float]"  },
      "battery": {
        "percentage": "[float]",
        "volts": "[float]",
        "maxVolts": "[float]",
        "minVolts": "[float]"
      },
      "co2": {  "ppm": "[float]"  },
      "co": {  "ppm": "[float]"  },
      "distance": {
```

```
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

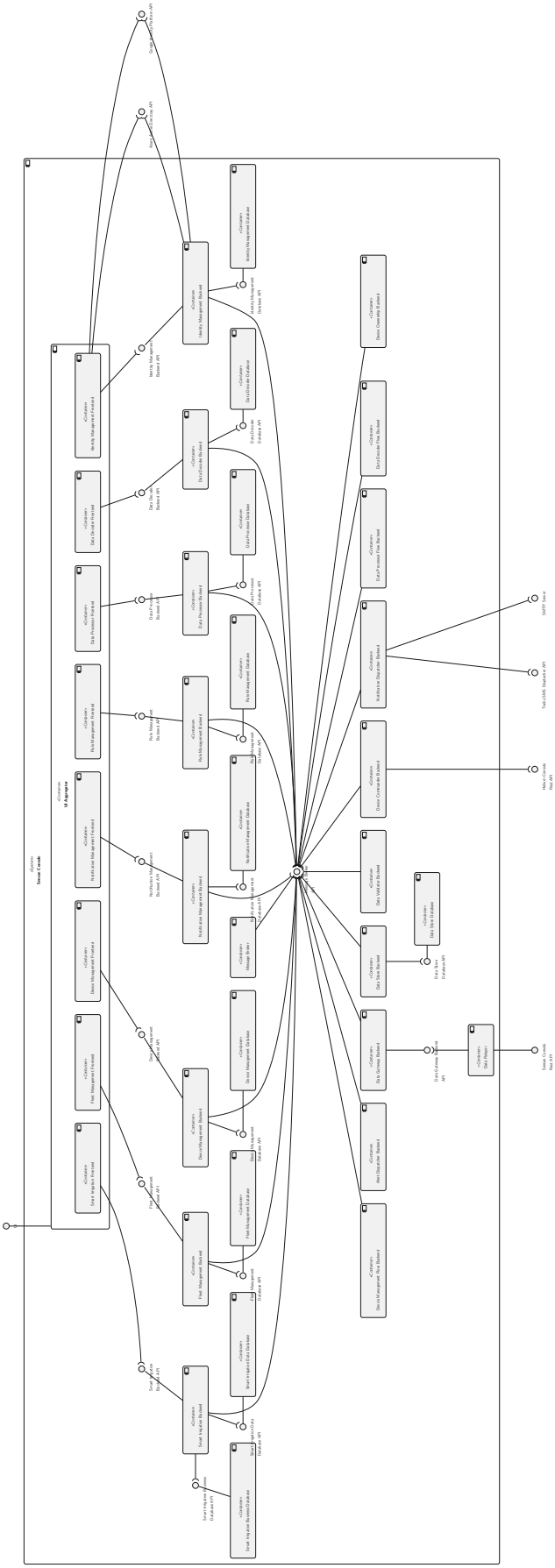

Figure B.1: Container Level - Logical View Diagram

Appendix C

Components Level - Logical View - Details

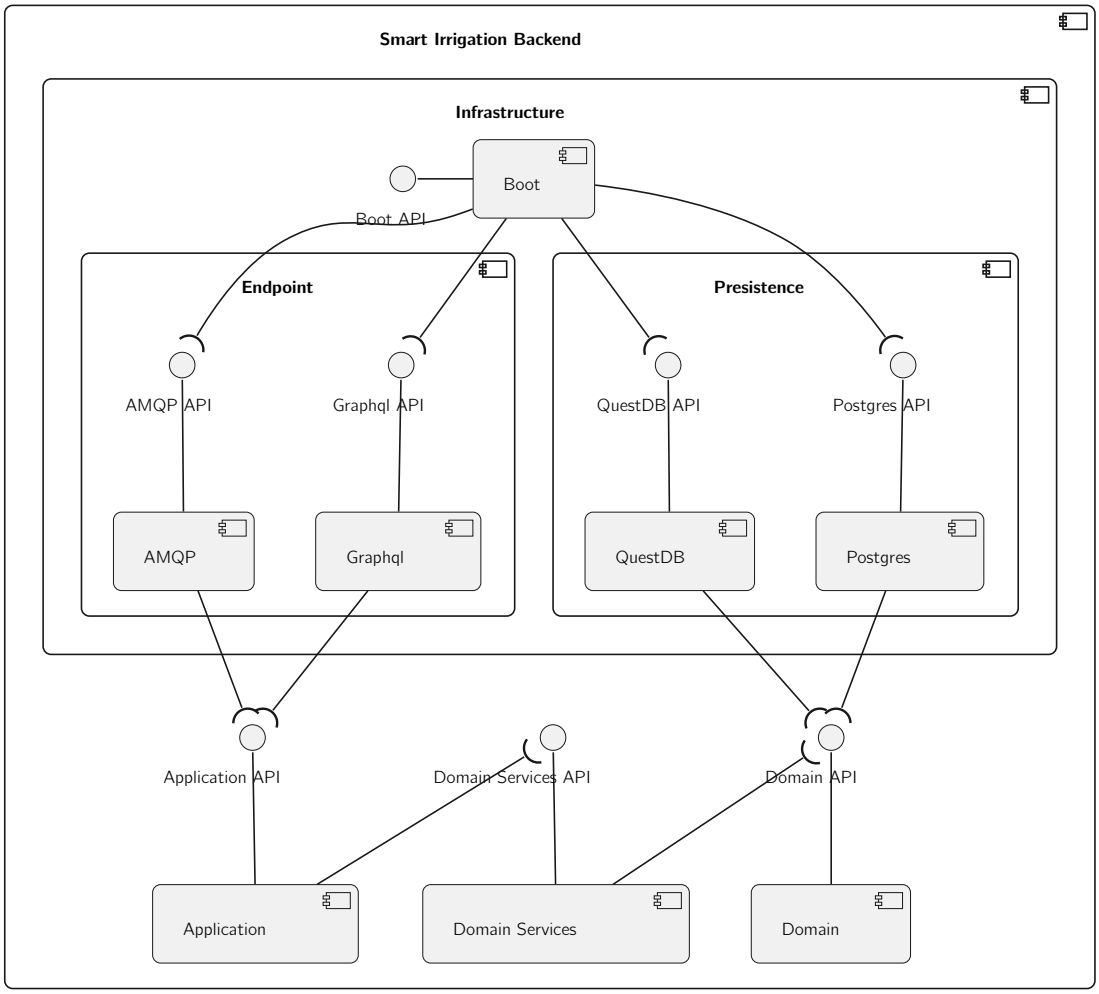

Figure C.1: Component Level - Smart Irrigation Backend - Logical View Diagram

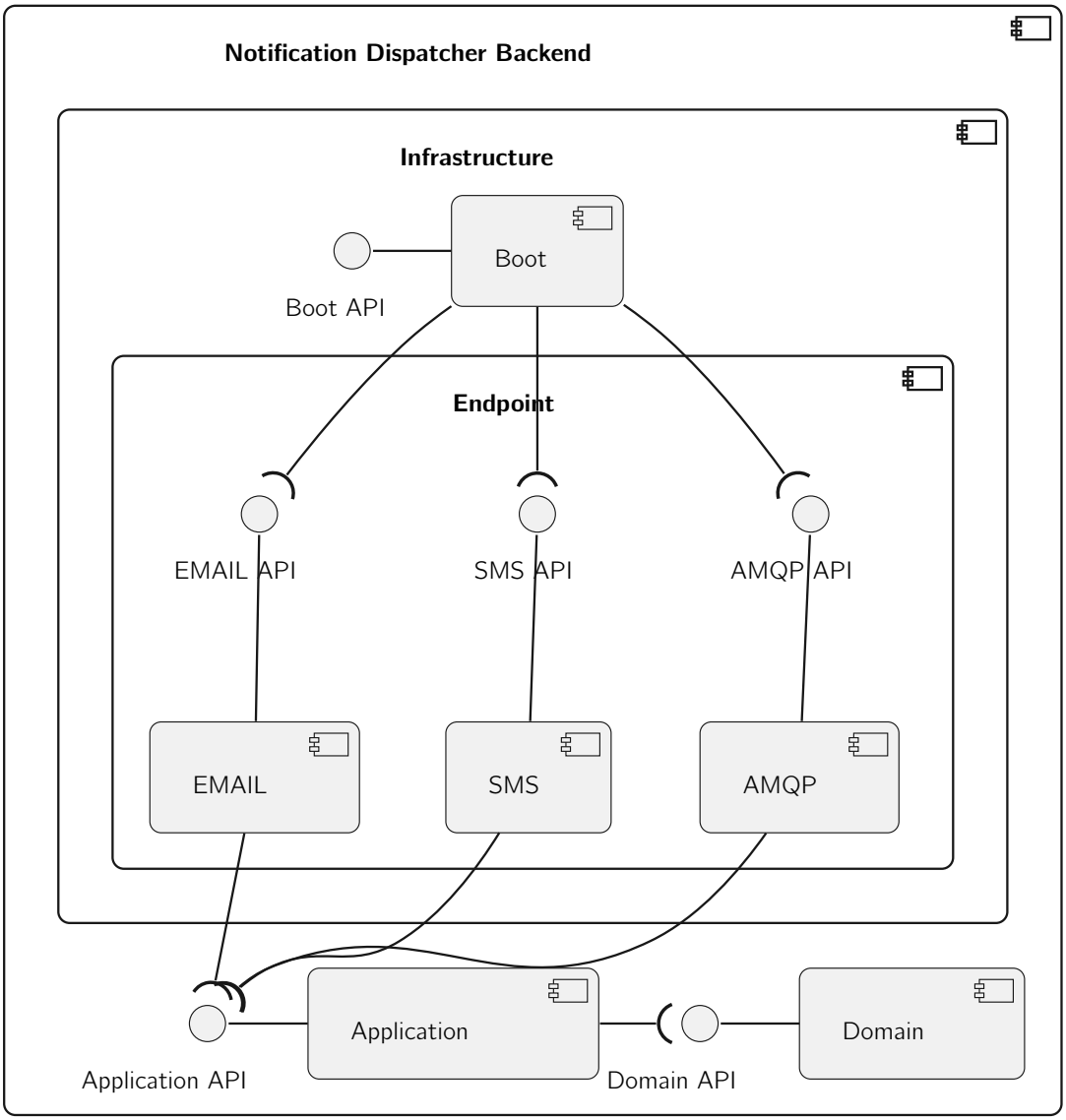

Figure C.2: Component Level - Notification Dispatcher Backend - Logical View Diagram

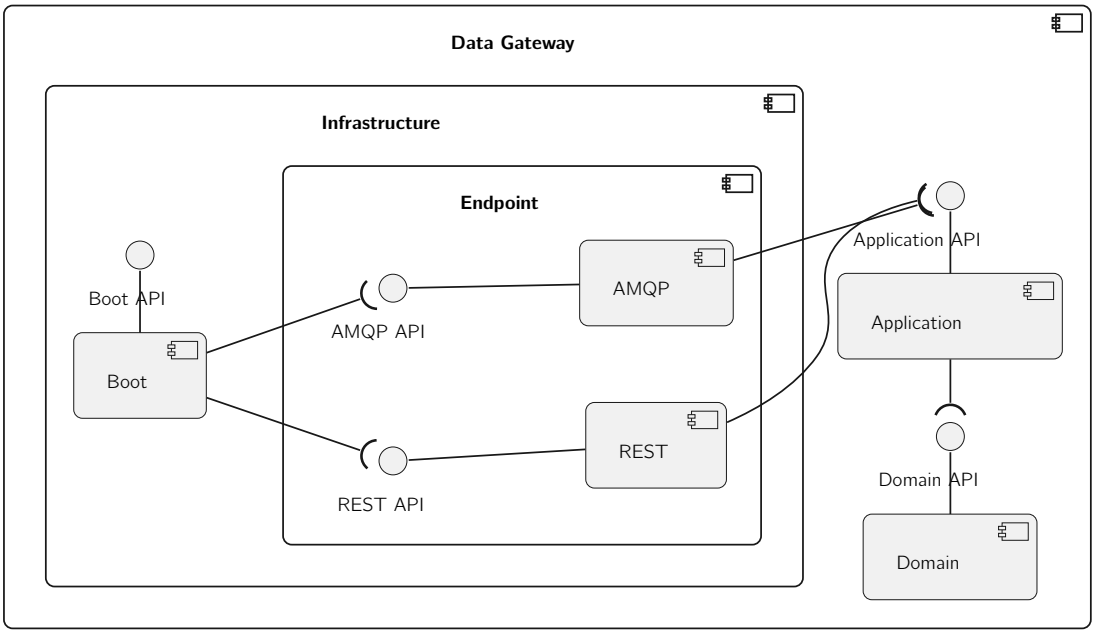

Figure C.3: Component Level - Data Gateway - Logical View Diagram

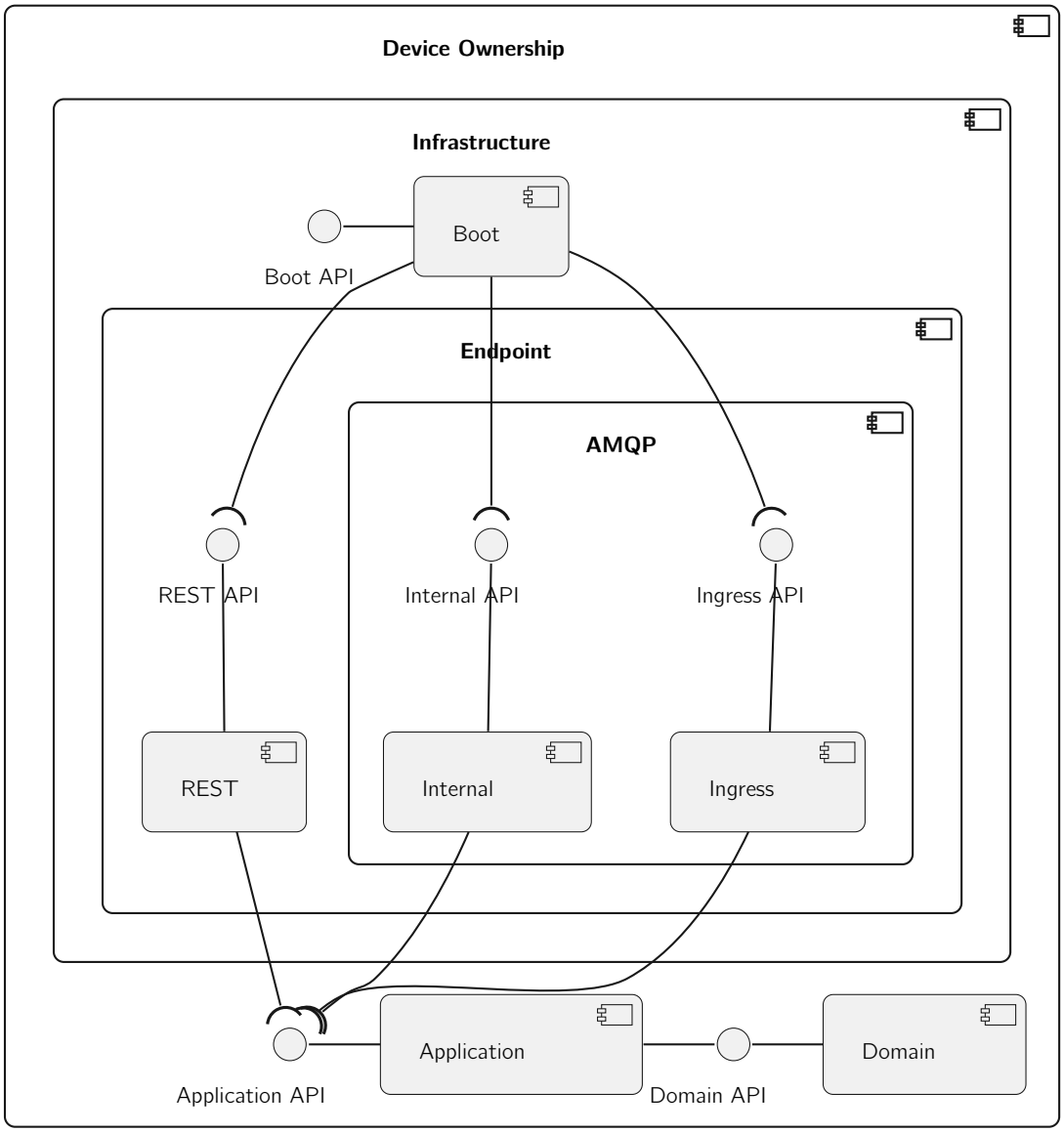

Figure C.4: Component Level - Device Commander - Logical View Diagram

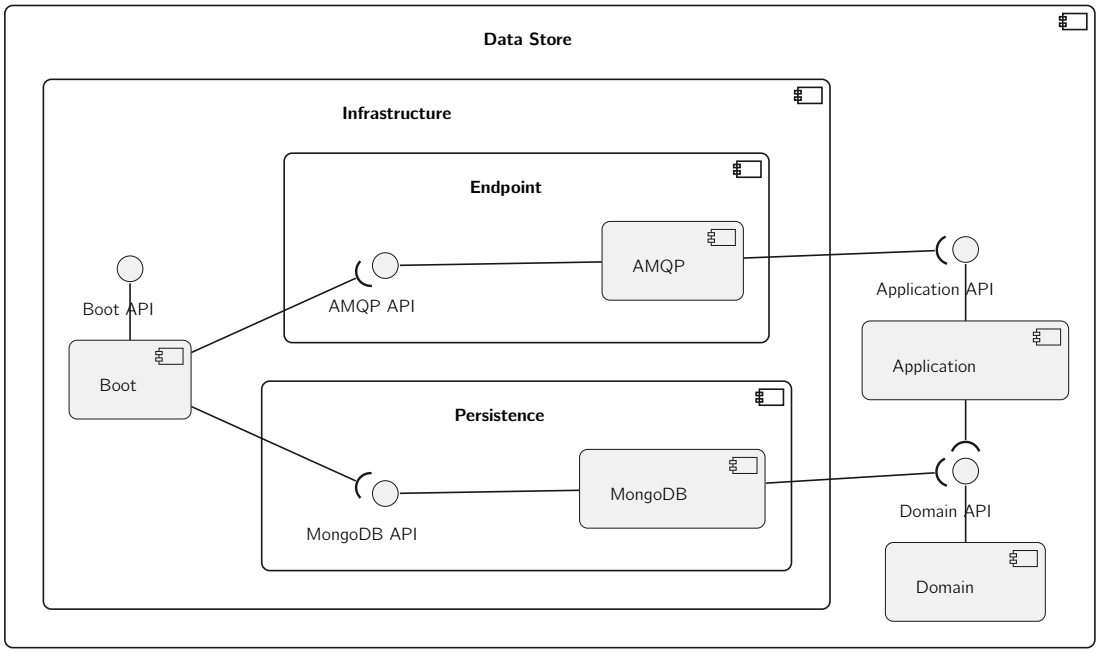

Figure C.5: Component Level - Data Store - Logical View Diagram

Appendix D

Additional Sensae Console UI Pages

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

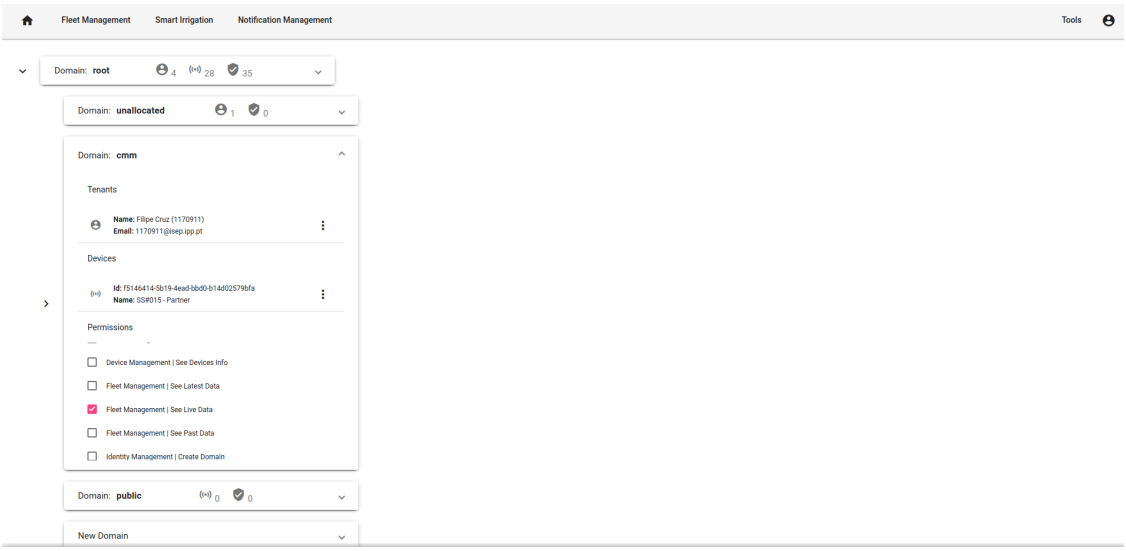

Figure D.1: Identity Management Page

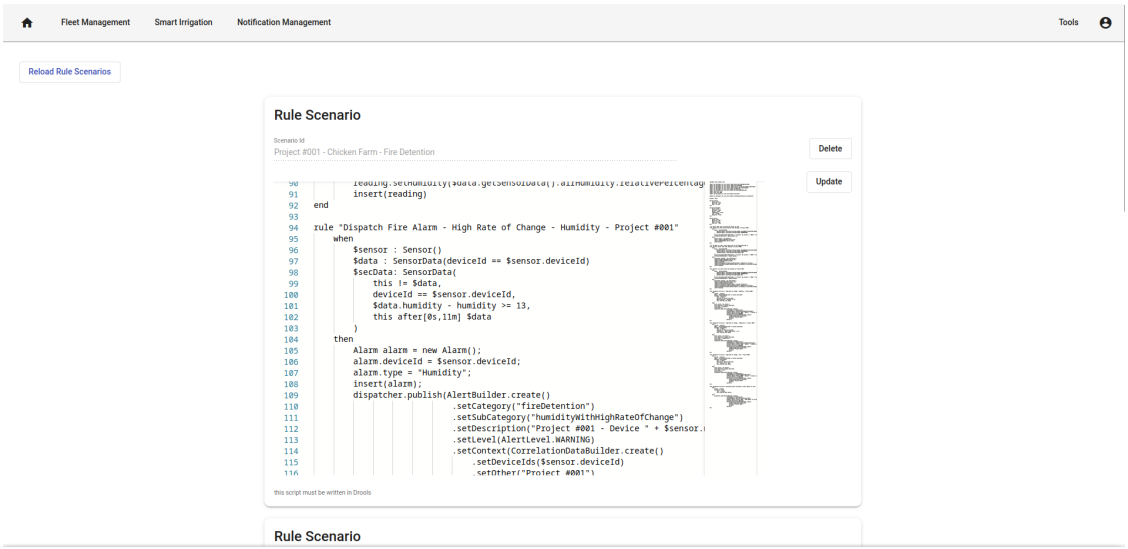

Figure D.2: Rules Management Page

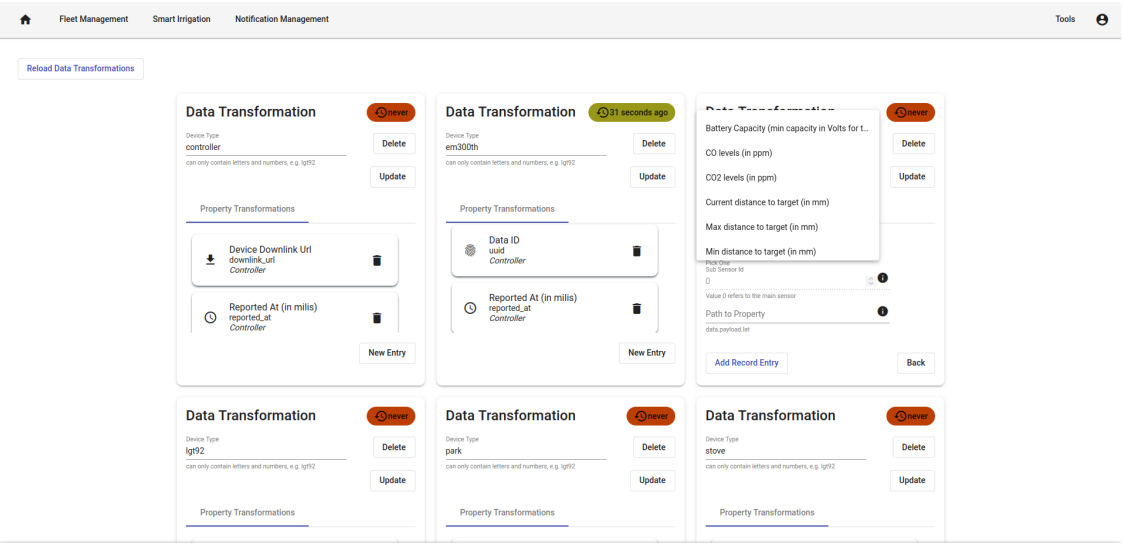

Figure D.3: Data Processor Page

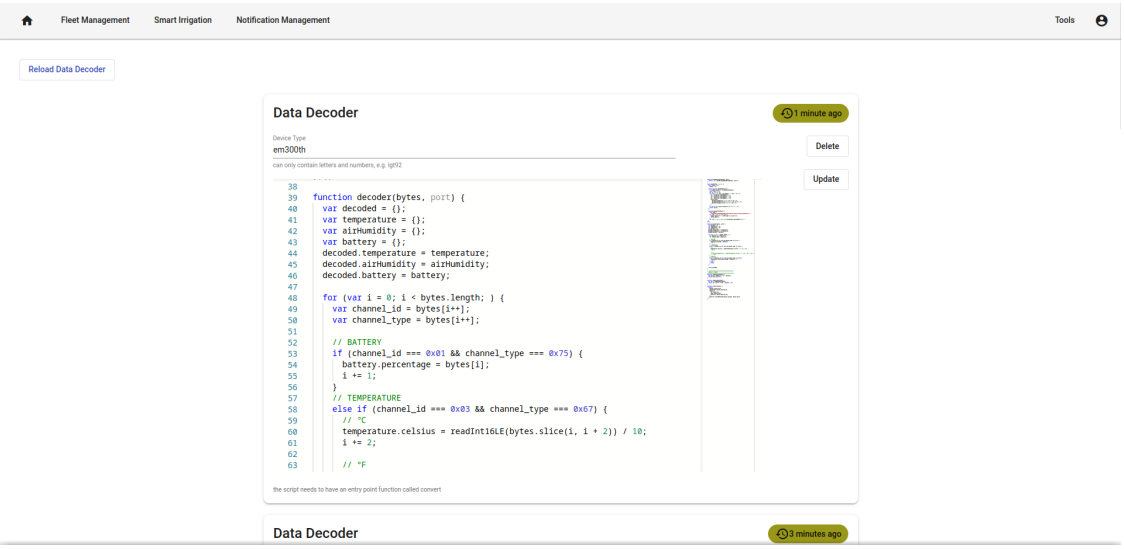

Figure D.4: Data Decoder Page

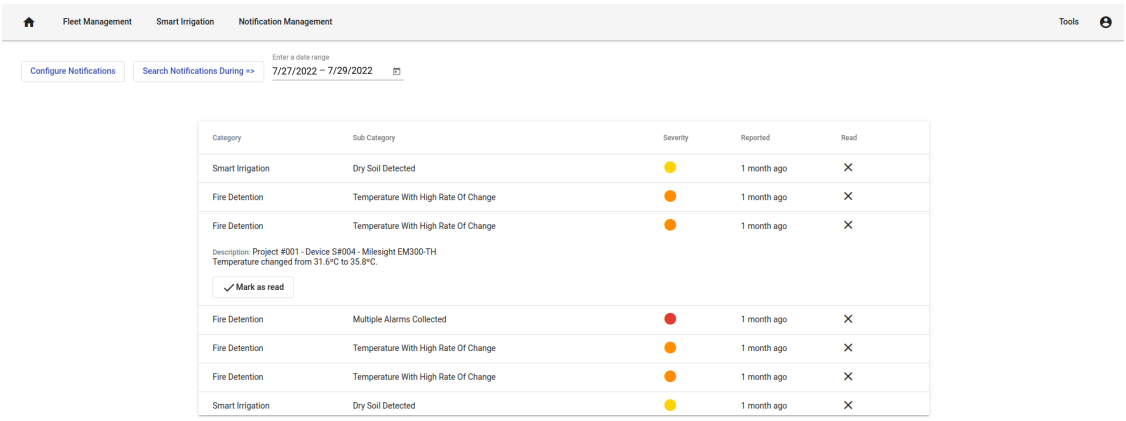

Figure D.5: Notification Management Page

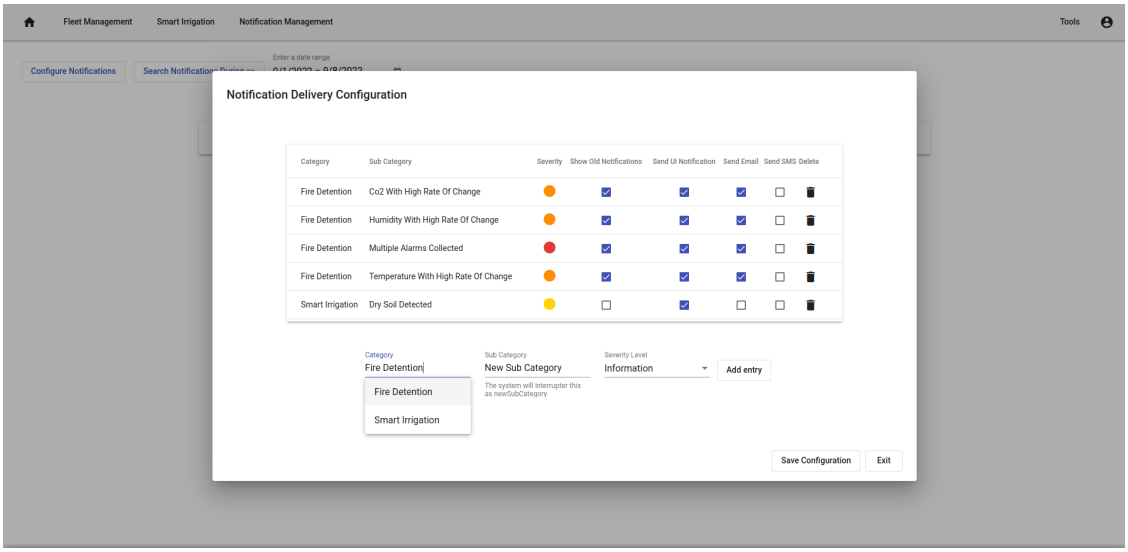

Figure D.6: Notification Management Page - Configuration

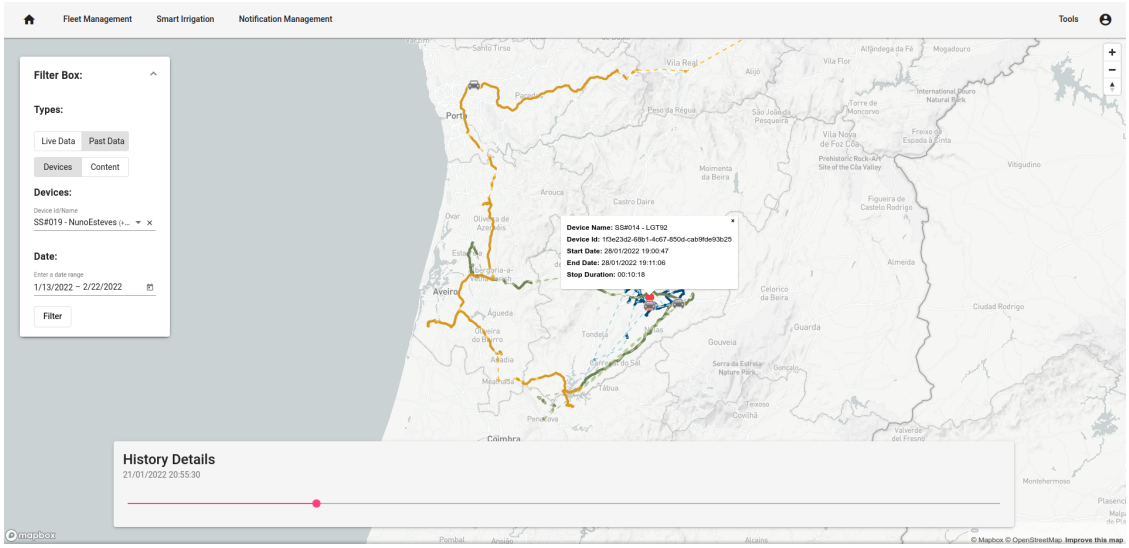

Figure D.7: Fleet Management Page

Figure D.8: Smart Irrigation Page - Map

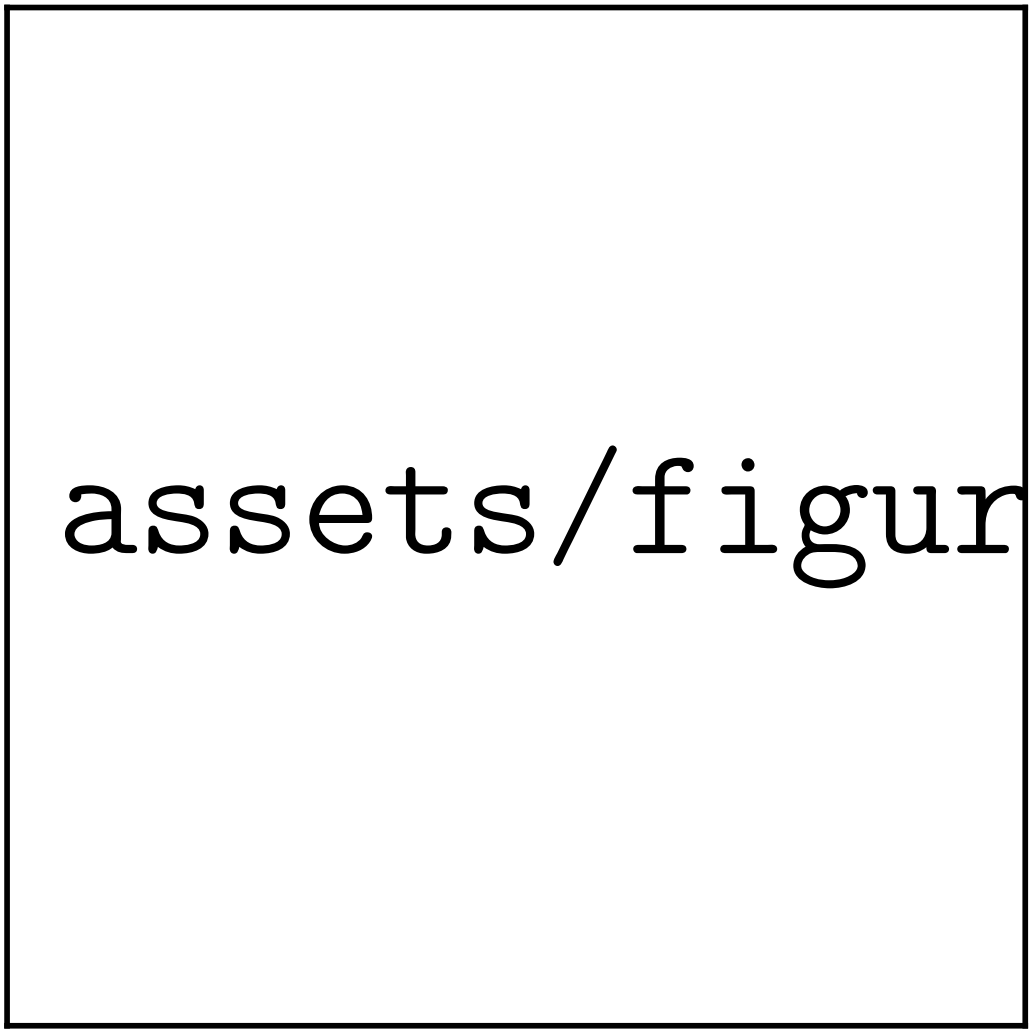

Figure D.9: Smart Irrigation Page - Device History