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, August 8, 2022

Dedicatory

TODO

Abstract

Today there are more smart devices than people. According to **statista-number-devices** 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 analyse, 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. De acordo com **statista-number-devices**, 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

List of Symbols

a	distance	m
P	power	W ($\mathrm{J\,s^{-1}}$)
ω	angular frequency	rad

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 reference architectures used for this project will be presented. Then the various system scopes will be introduced, followed by 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 Reference Architecture

TODO - 3 tier architecture - onion architecture pattern - reactive architecture - microservice architecture - data pipeline

5.2 System Scopes

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

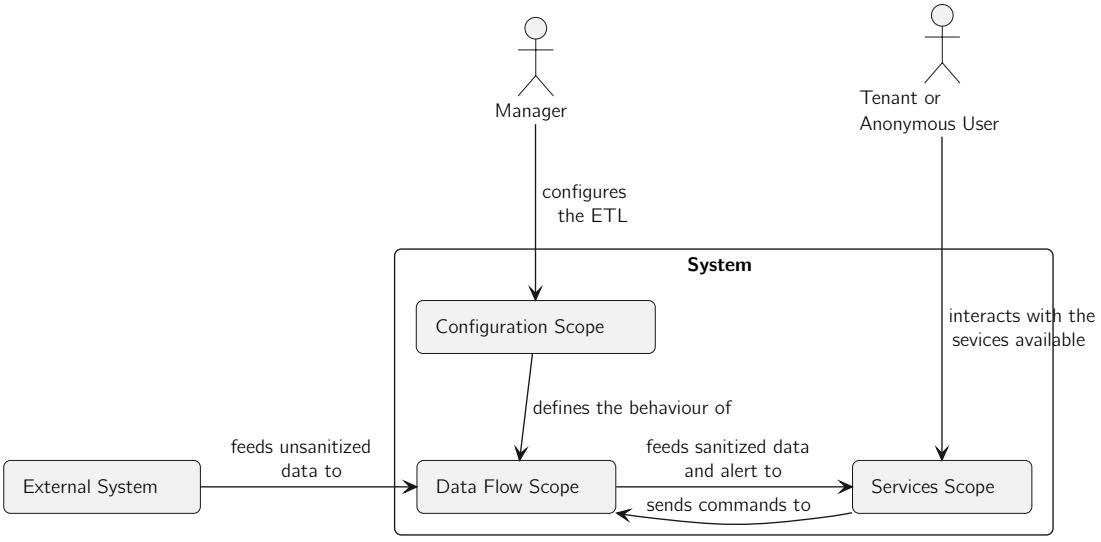

Figure 5.1: System Scopes

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.

5.2.1 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 verifications, alter the system internal state and are therefore prolonged operations.

5.2.2 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.

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

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

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.

5.2.3 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 Minimal 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 5.3.1 and 5.3.2.

5.2.4 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 5.3 addresses this shared domain and each context peculiarity.

5.3 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.

5.3.1 Concepts

In order for the reader to better understand how the system functions some concepts need to be better 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.

5.3.2 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 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.

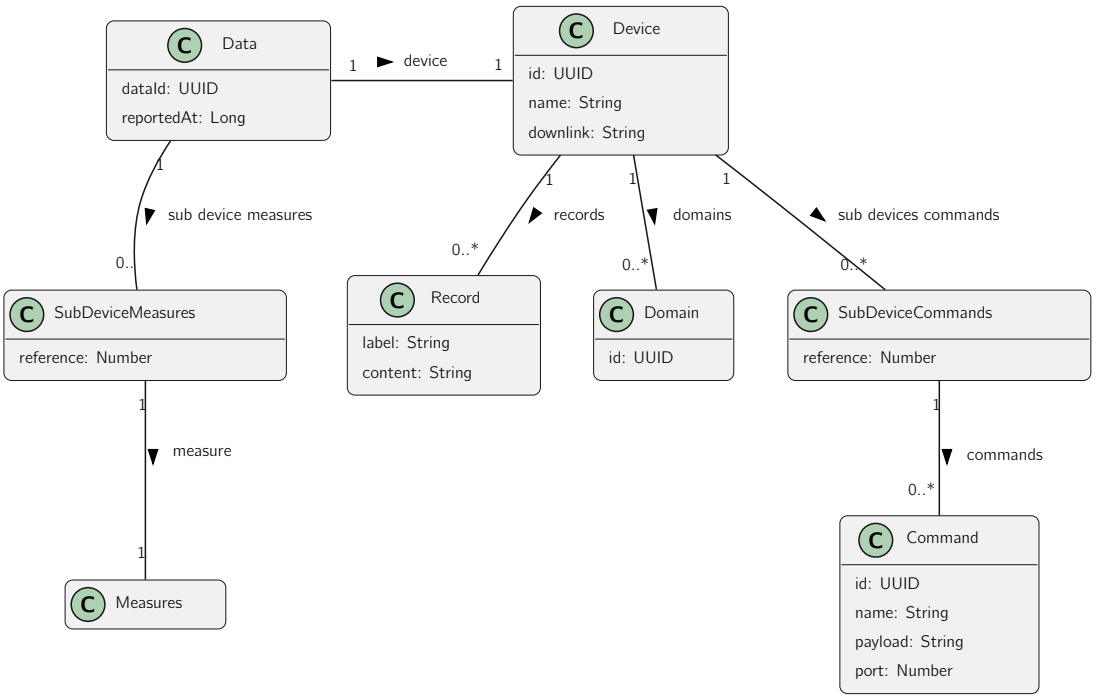

Figure 5.2: Shared Model

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 ***TODO***.

Message Envelop Model

The message envelop model refers to how, coupled with the routing model in Section 5.3.2, 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.

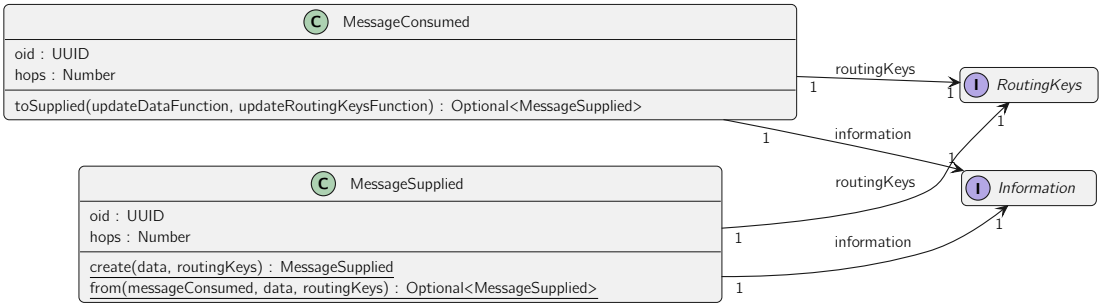

Figure 5.3: Message Envelop Model

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 5.3.2;

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 (**TODO**), 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.

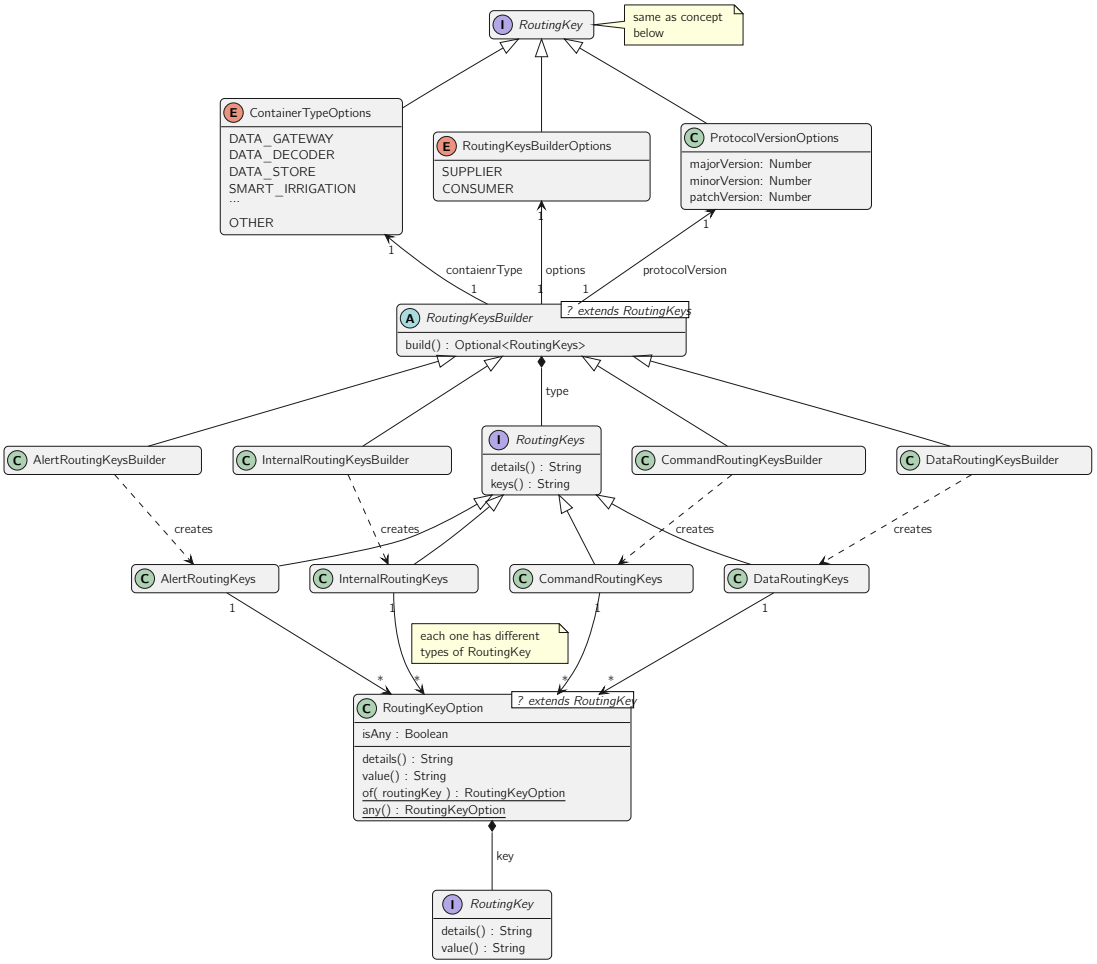

Figure 5.4: Routing Model

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*.

5.3.3 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 5.3.2.

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.

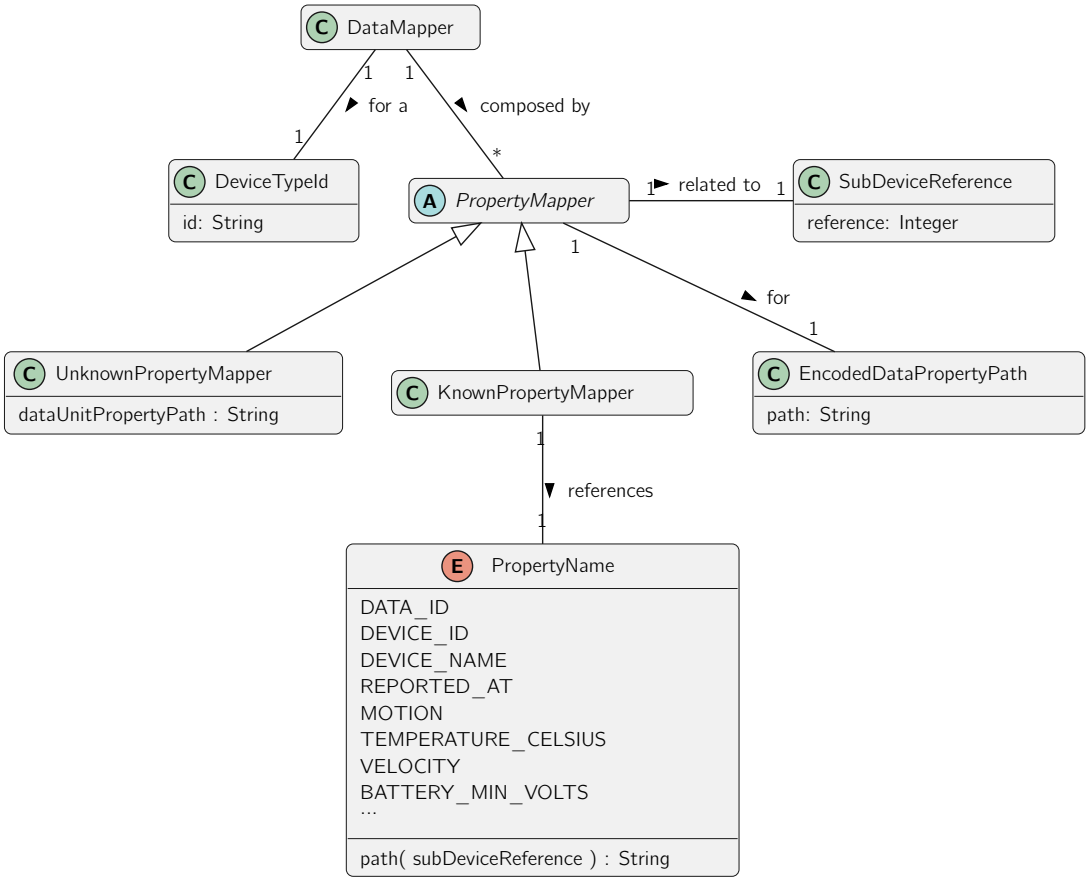

Figure 5.5: Data Processor Context Model

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 5.3.2. It was created to deal with the limitations mentioned in Section 5.3.3.

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.

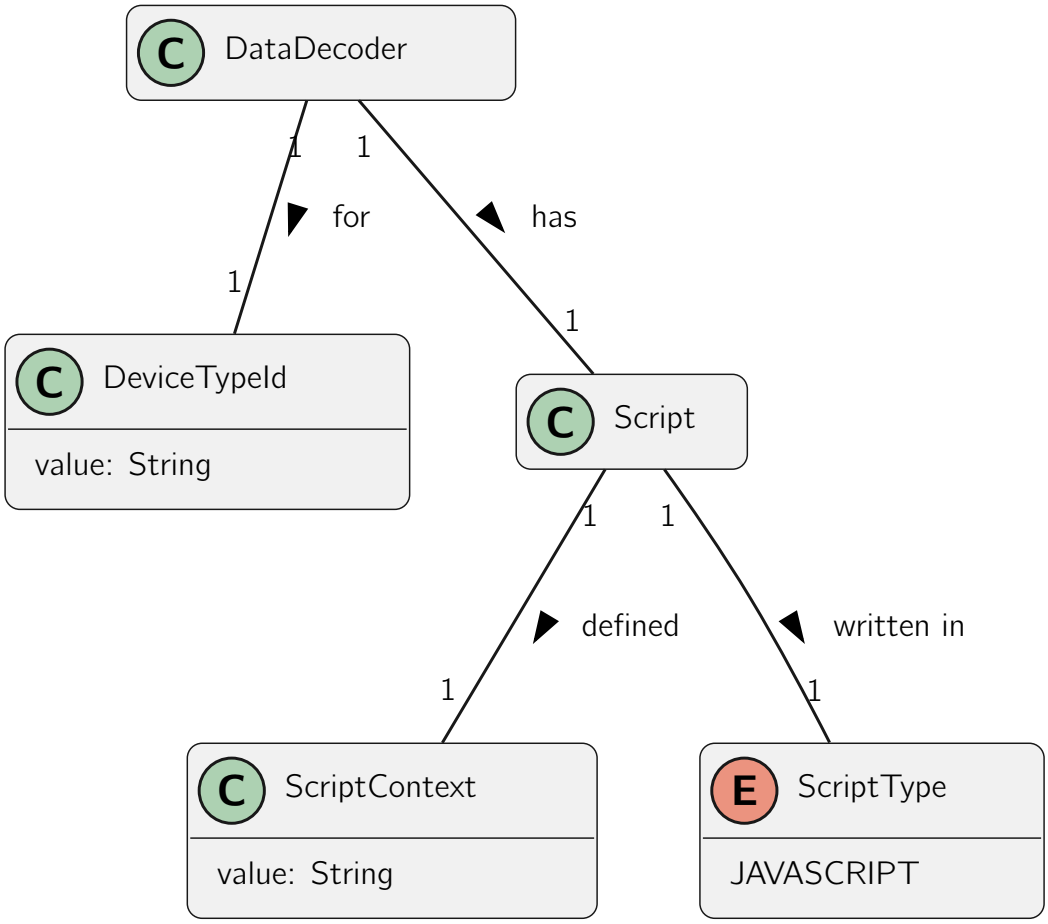

Figure 5.6: Data Decoder Context Model

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 programing 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.

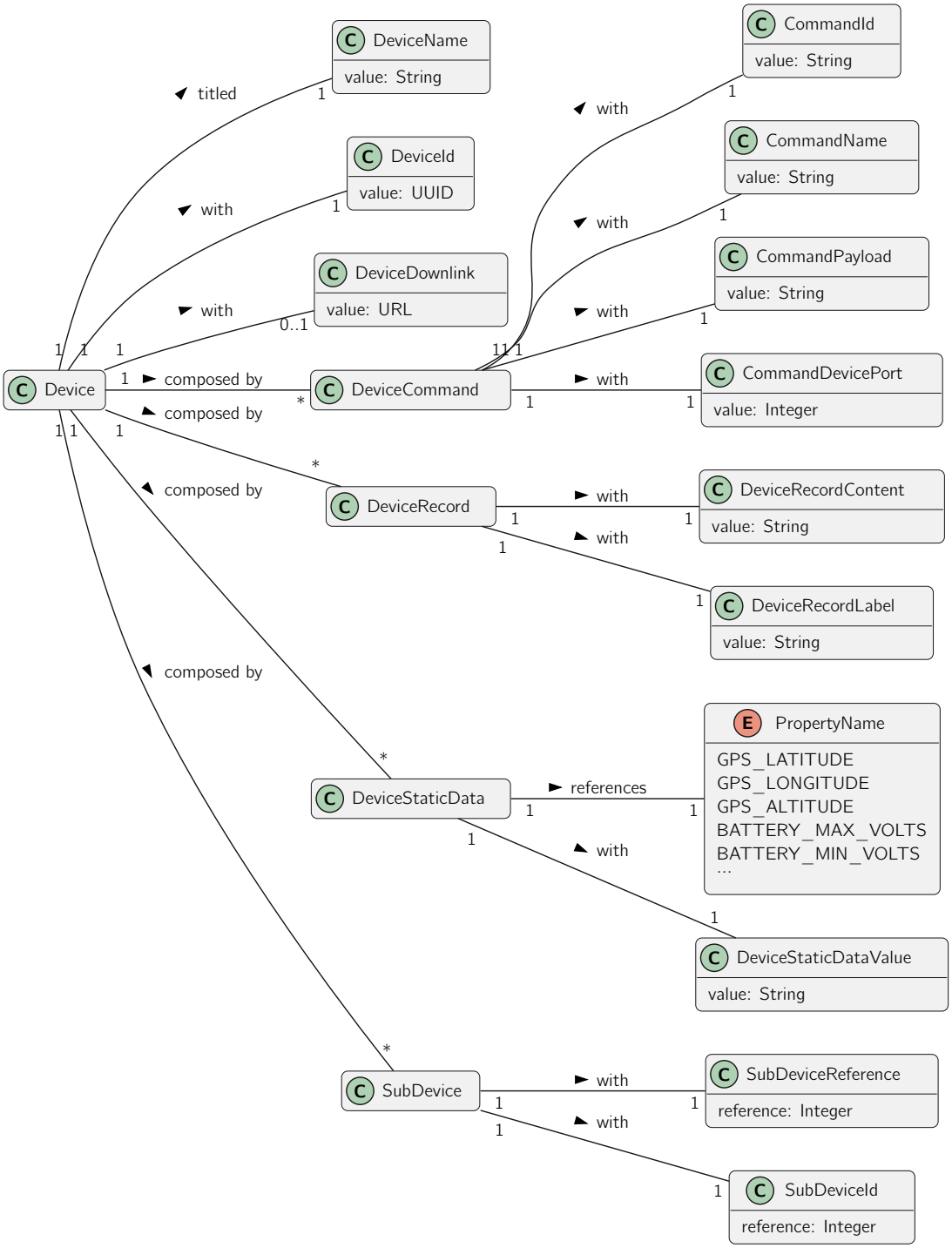

Figure 5.7: Device Management Context Model

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.

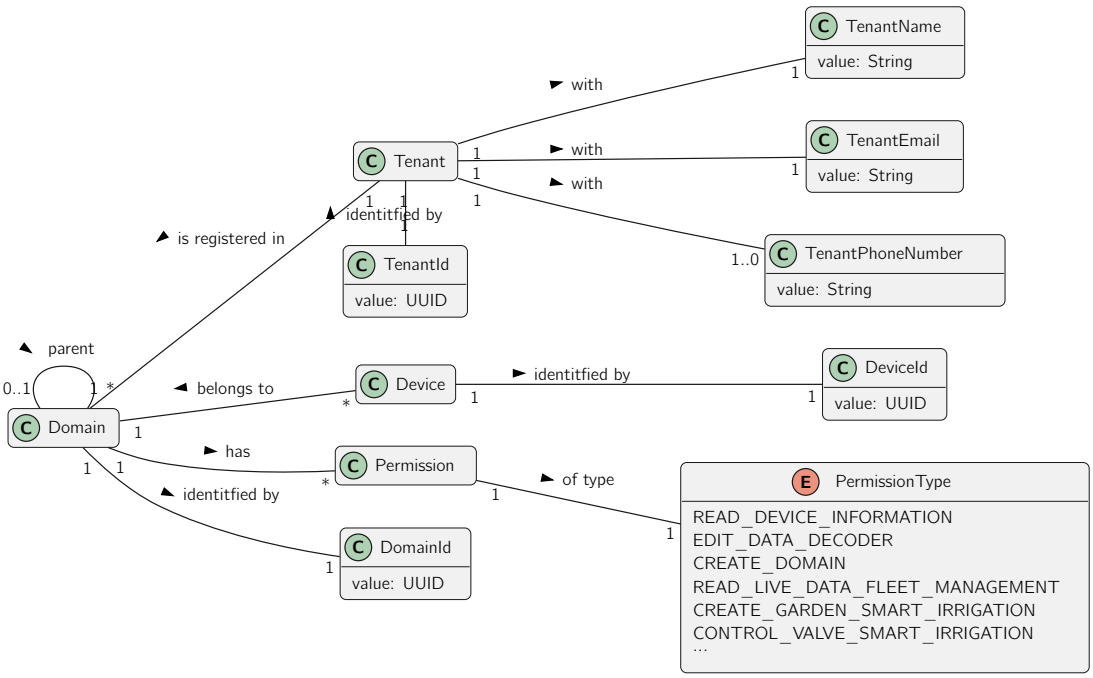

Figure 5.8: Identity Management Context Model

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 doesn't have a parent and has all available permissions;
- A **Tenant** has a **TenantName** and **TenantEmail**, unique **TenantId** and can have a **TenantPhoneNumber**;
- 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.

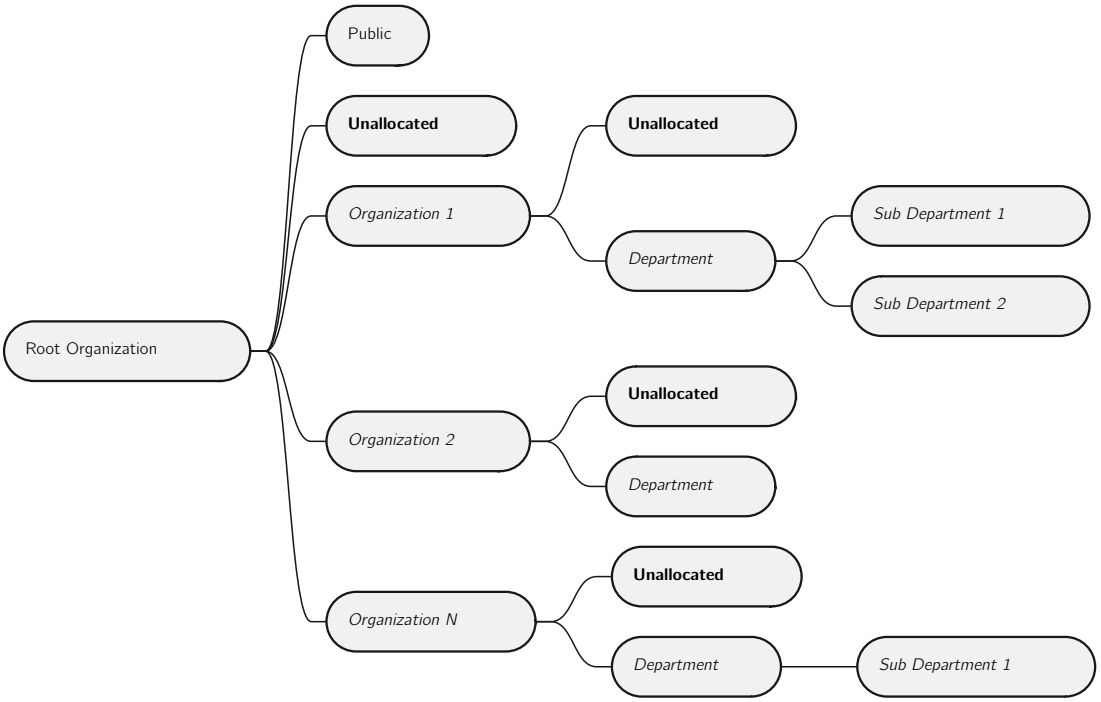

Figure 5.9: Domain Structure

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 *Public* domain can be accessed by any tenant, including those who are not authenticated in the system;

Rule Management

The **Rule Management** context refers to rule scenarios that produce **Alert**s based on incoming **Data Unit**s.

The diagram in Figure 5.10 displays the noteworthy concepts in this context.

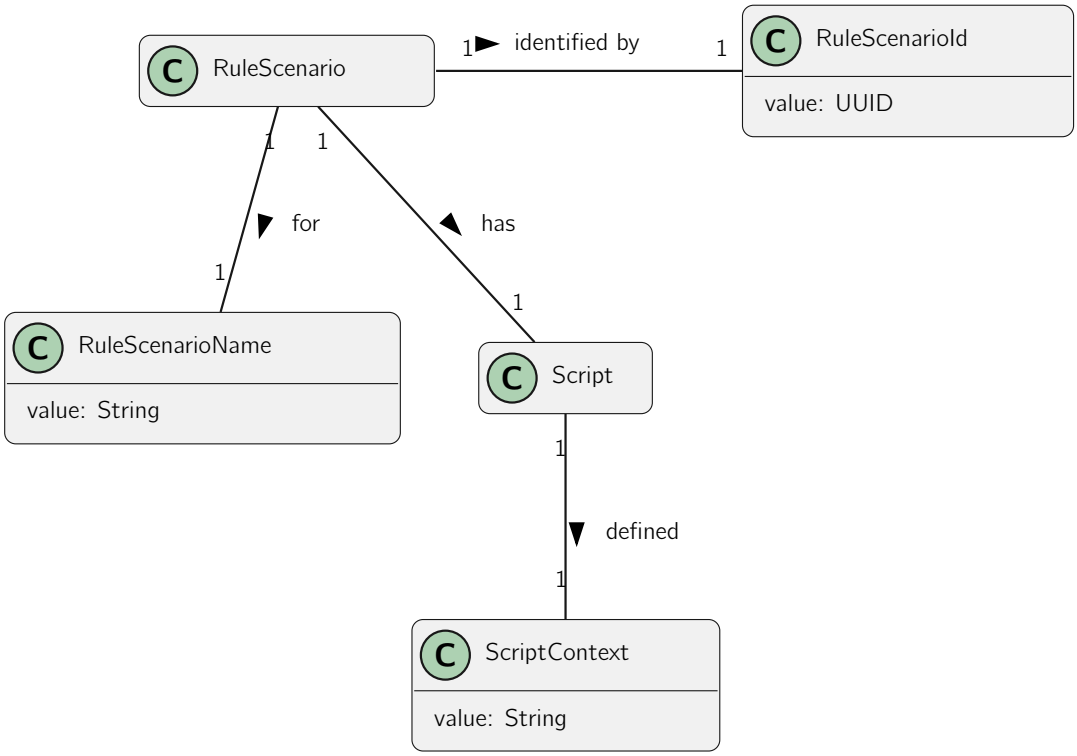

Figure 5.10: Rule Management Context Model

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 5.11 displays the noteworthy concepts in this context.

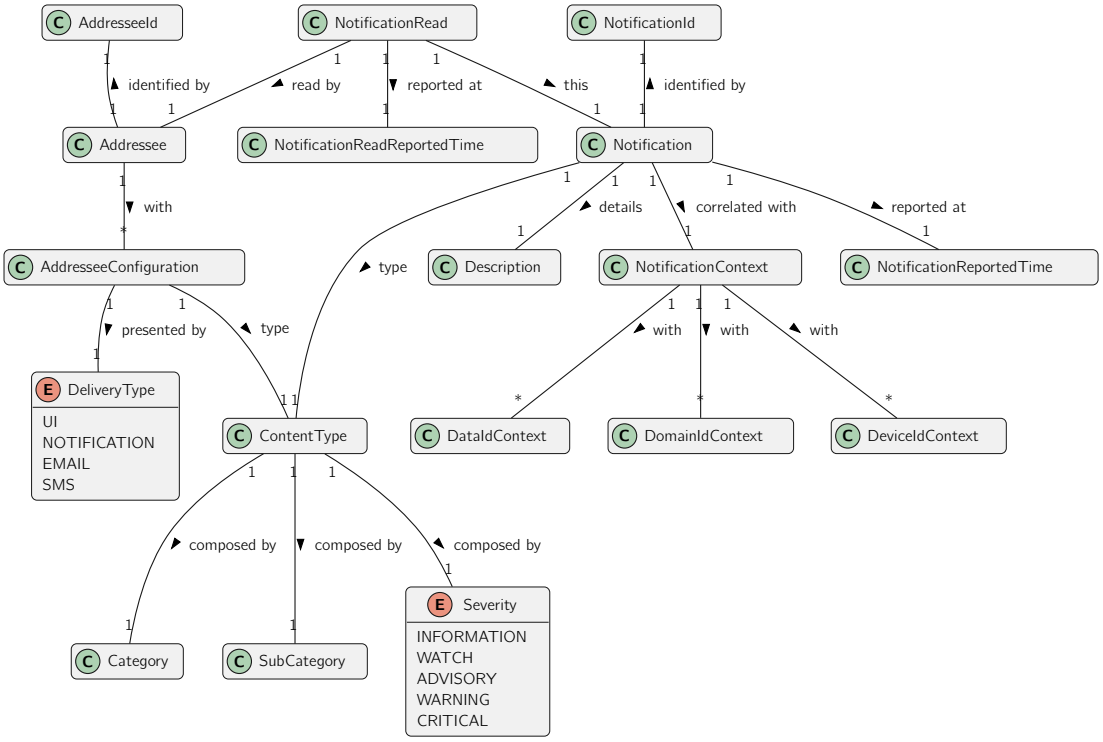

Figure 5.11: Notification Management Context Model

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 5.12 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.

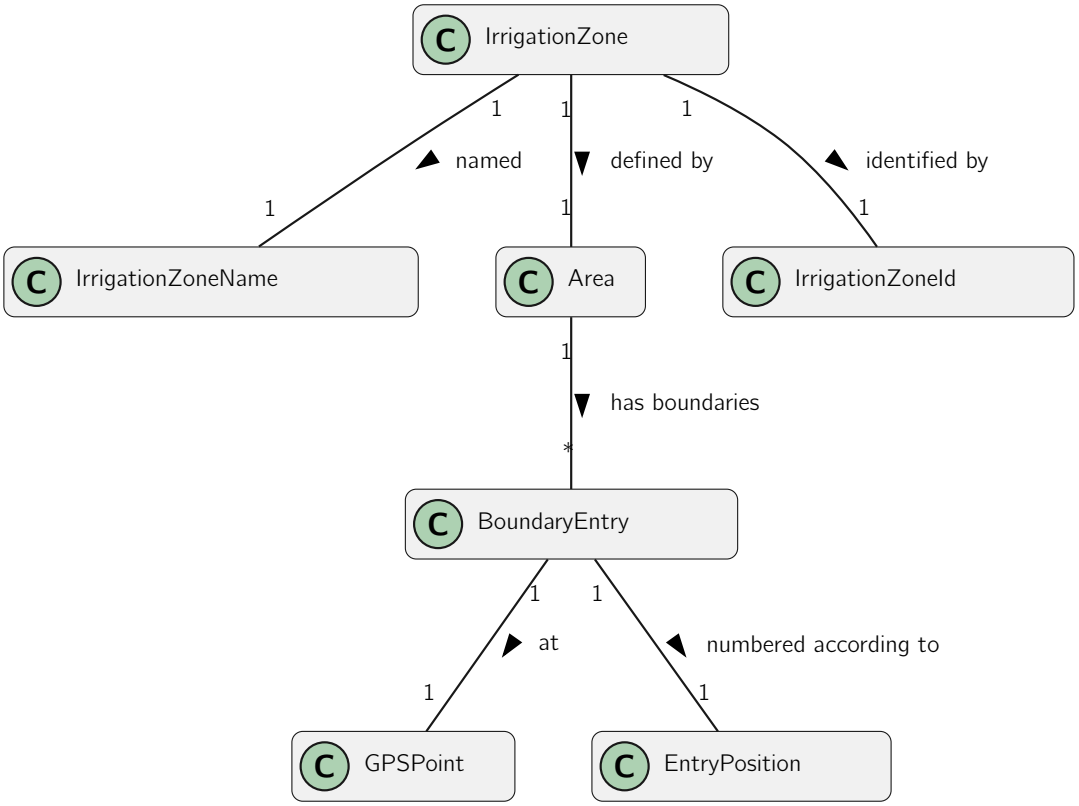

Figure 5.12: Smart Irrigation Context Model - Irrigation Zone

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

As presented in the following diagram, Figure 5.13, a sensor/valve can be represents by a **Device**.

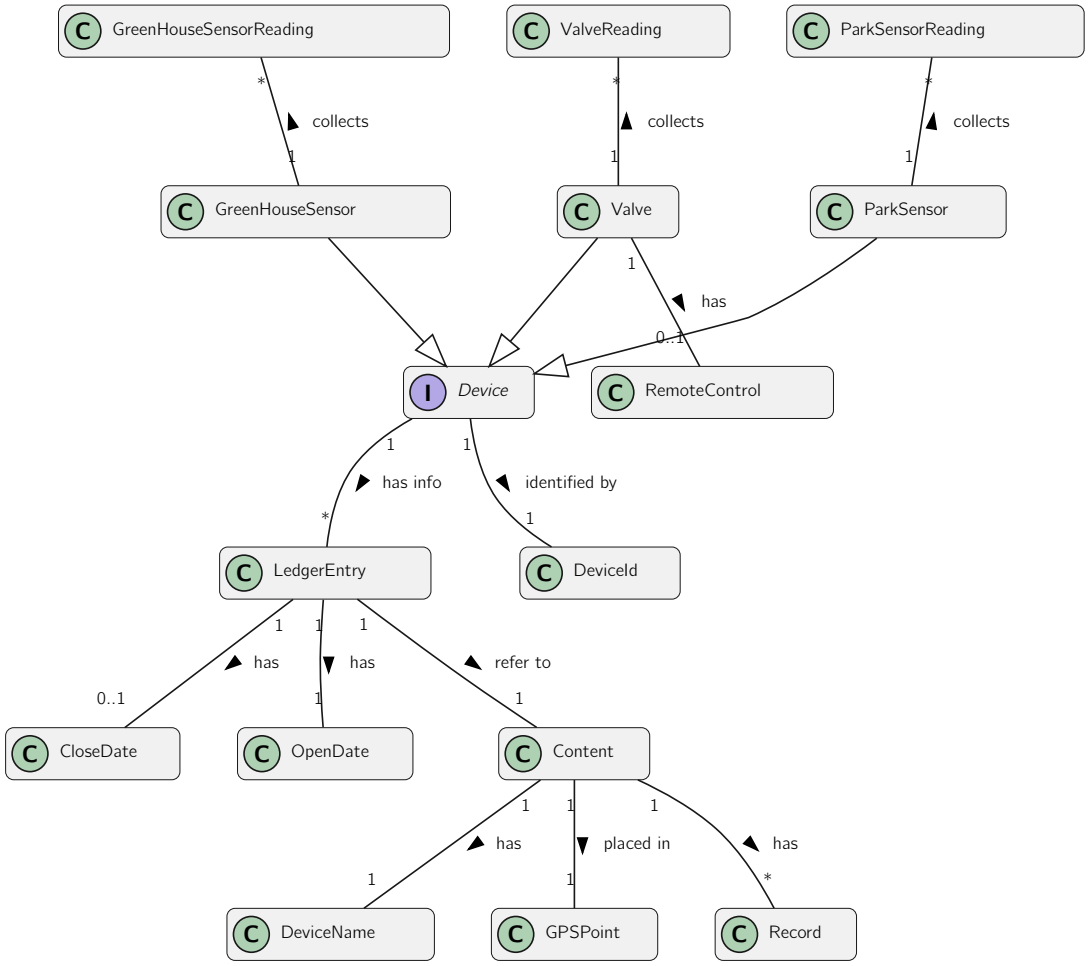

Figure 5.13: Smart Irrigation Context Model - Device

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 Figure5.14.

As mentioned above each type of device collects different readings. The following diagram, Figure5.14, details this readings.

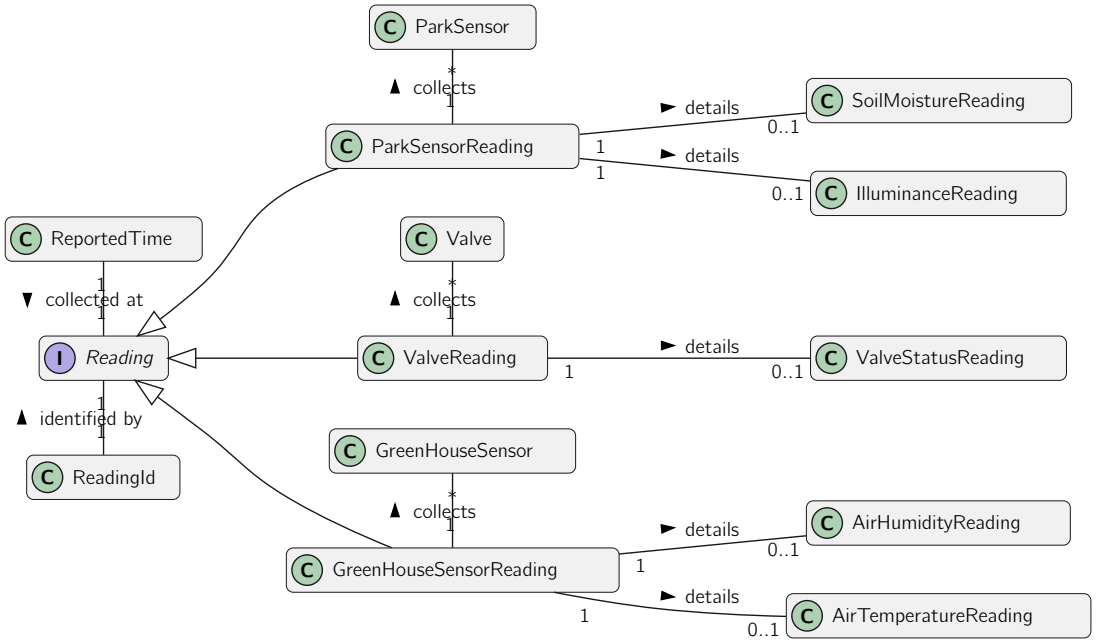

Figure 5.14: Smart Irrigation Context Model - Reading

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 5.15 displays the noteworthy concepts related to this context.

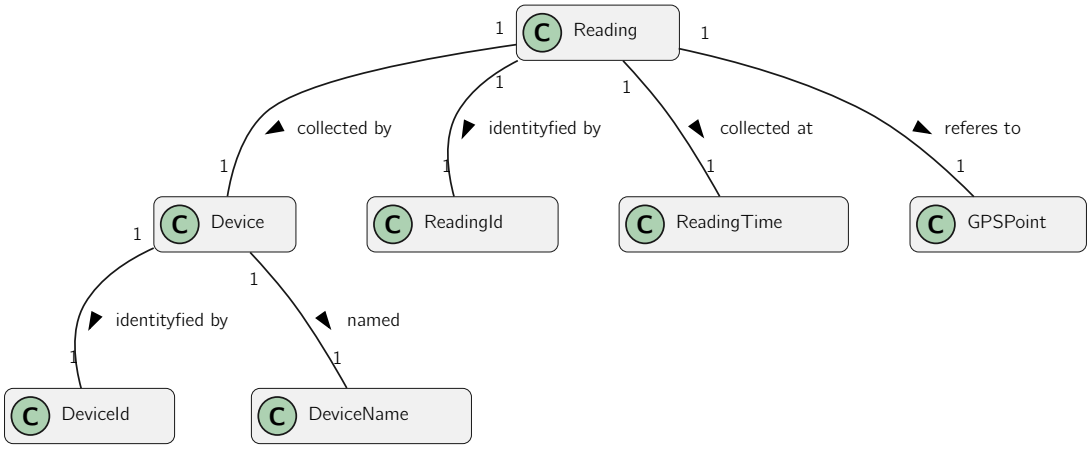

Figure 5.15: Fleet Management Context Model

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.

5.3.4 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.

5.4 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.

5.4.1 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 5.16).

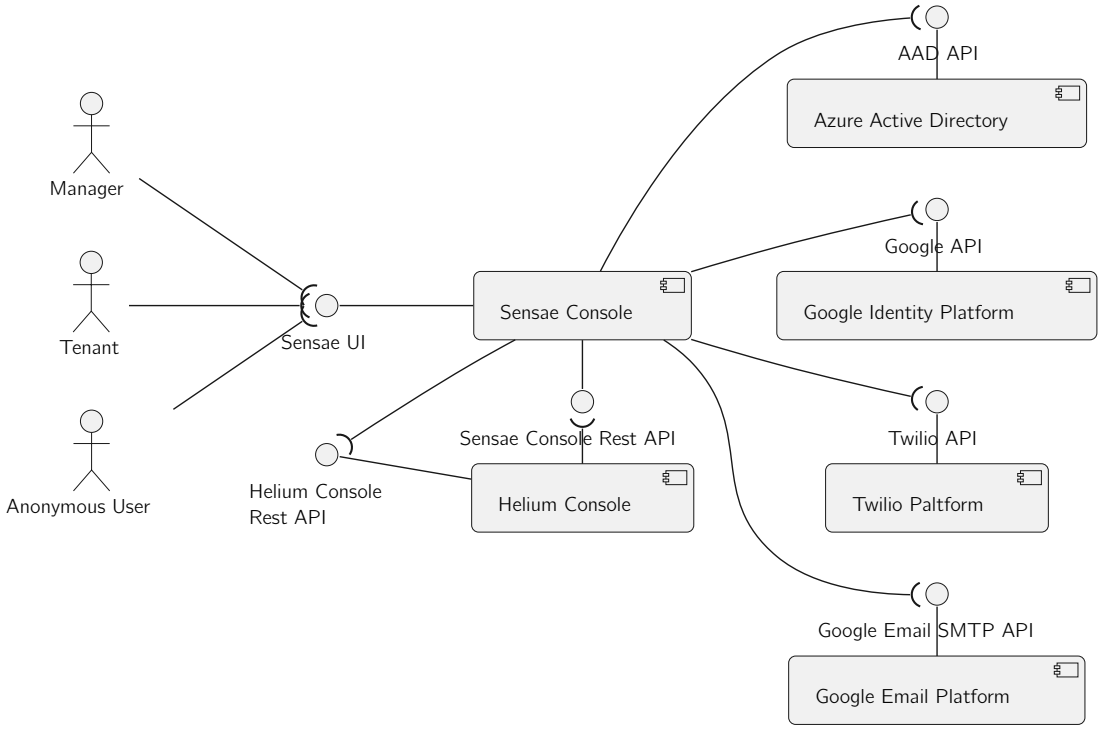

Figure 5.16: Context Level - Logical View Diagram

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 5.5.3.

Context Level - Development View

Next is the development view (Figure 5.17), intended to familiarize the reader with how the software is organized.

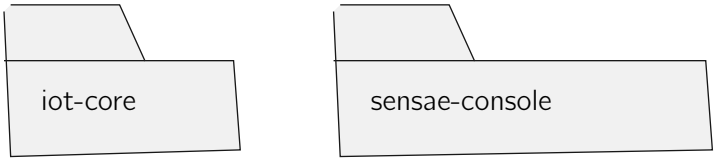

Figure 5.17: Context Level - Development View Diagram

The package *iot-core* contains the shared model discussed in the Section 5.3.2, and functions to define what type of information a backend containers wants to subscribe or publish (discussed in Section 5.3.1).

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 5.18), intended to familiarize the reader with the environment where the solution runs.

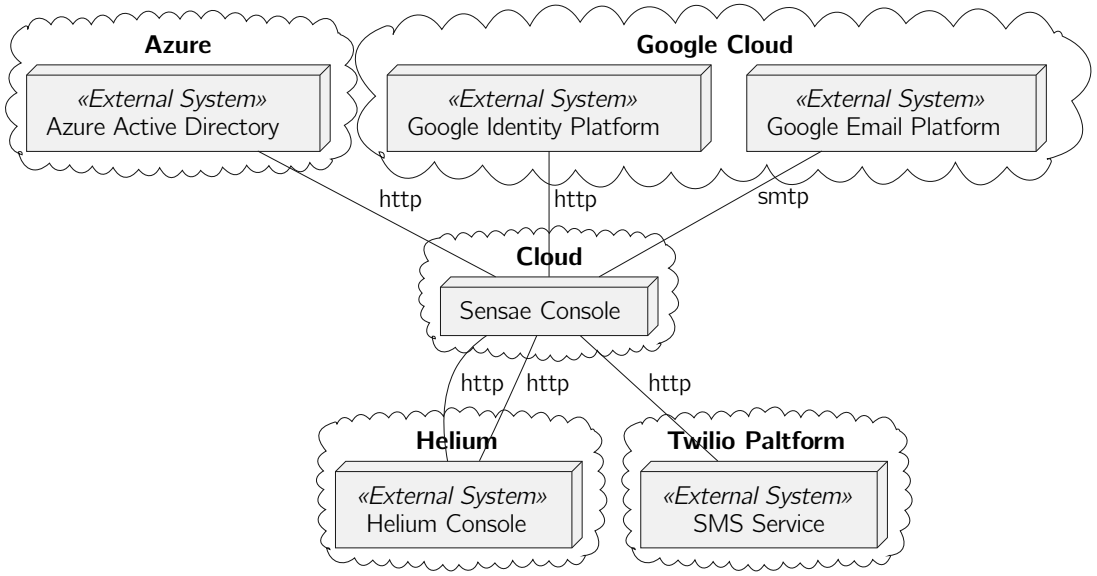

Figure 5.18: Context Level - Physical View Diagram

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 5.4.2 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.

5.4.2 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 5.5 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 2020a, 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 5.2. In the Appendix TODO a complete logical view is provided.

The logical view of the **Configuration Scope** is represented in Figure 5.19. This scope is composed by the processes discussed in 5.2.1. Each process is composed by a three tier architecture, as per IBM 2020b:

- **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.

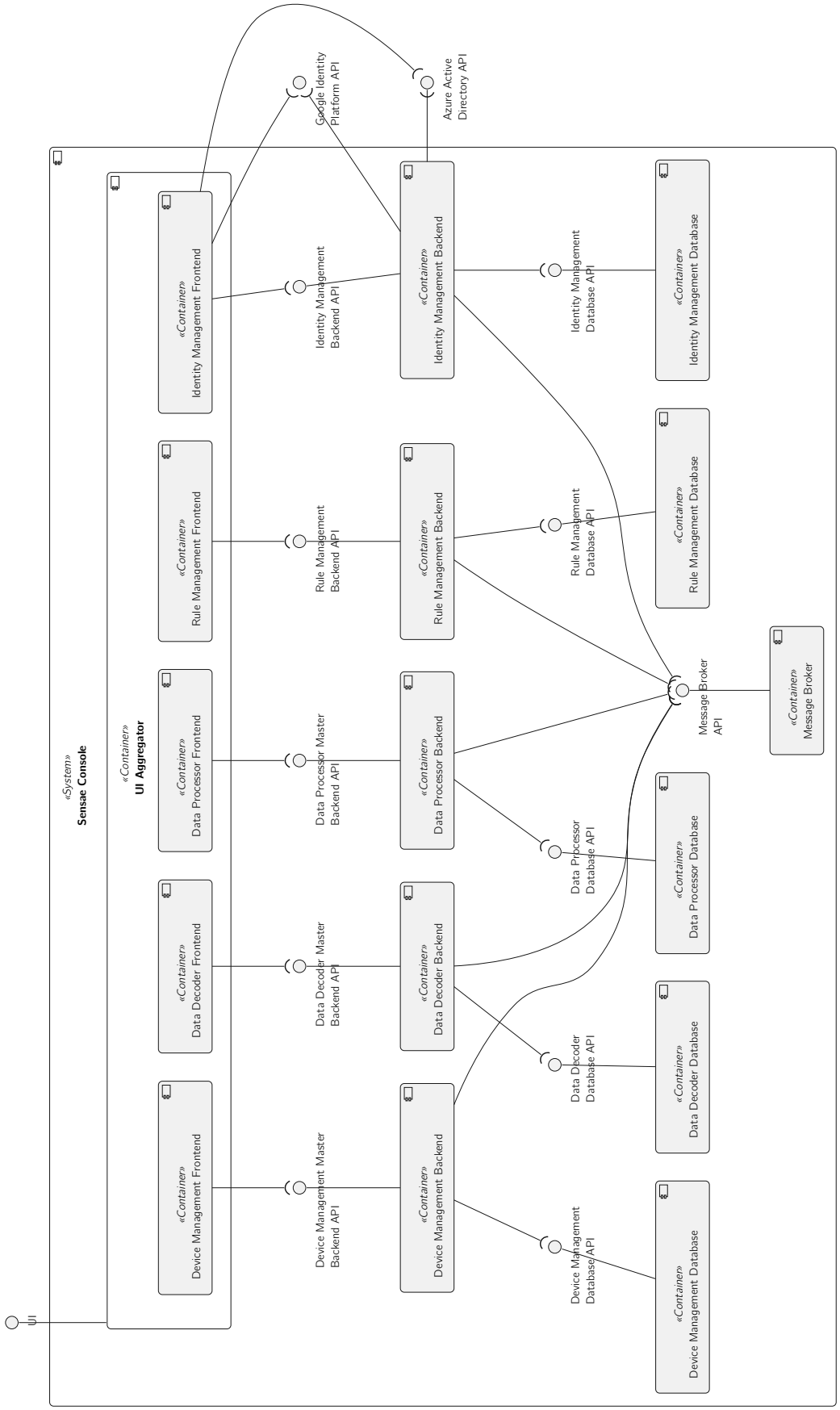

Figure 5.19: Container Level - Configuration Scope - Logical View Diagram

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 5.20. This scope is composed by the processes discussed in 5.2.2. 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.

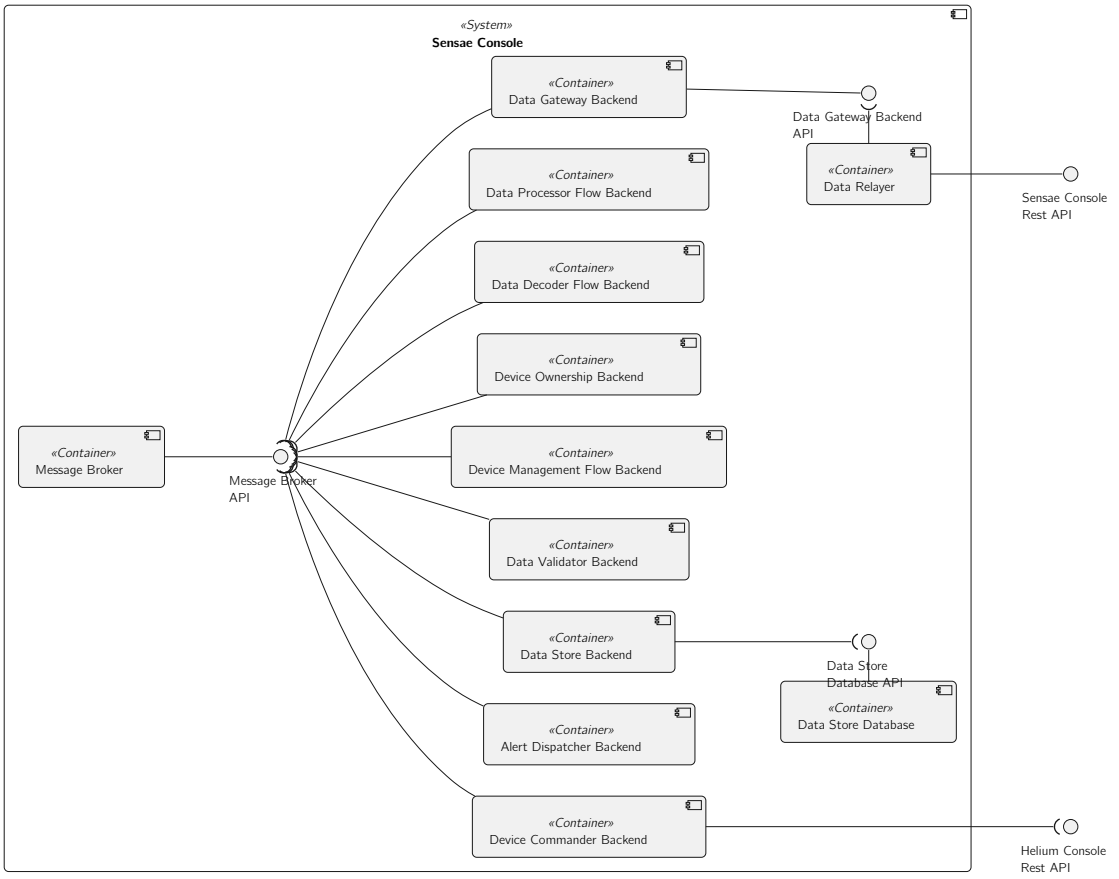

Figure 5.20: Container Level - Data Flow Scope - Logical View Diagram

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 5.21. This scope is composed by the processes discussed in 5.2.3.

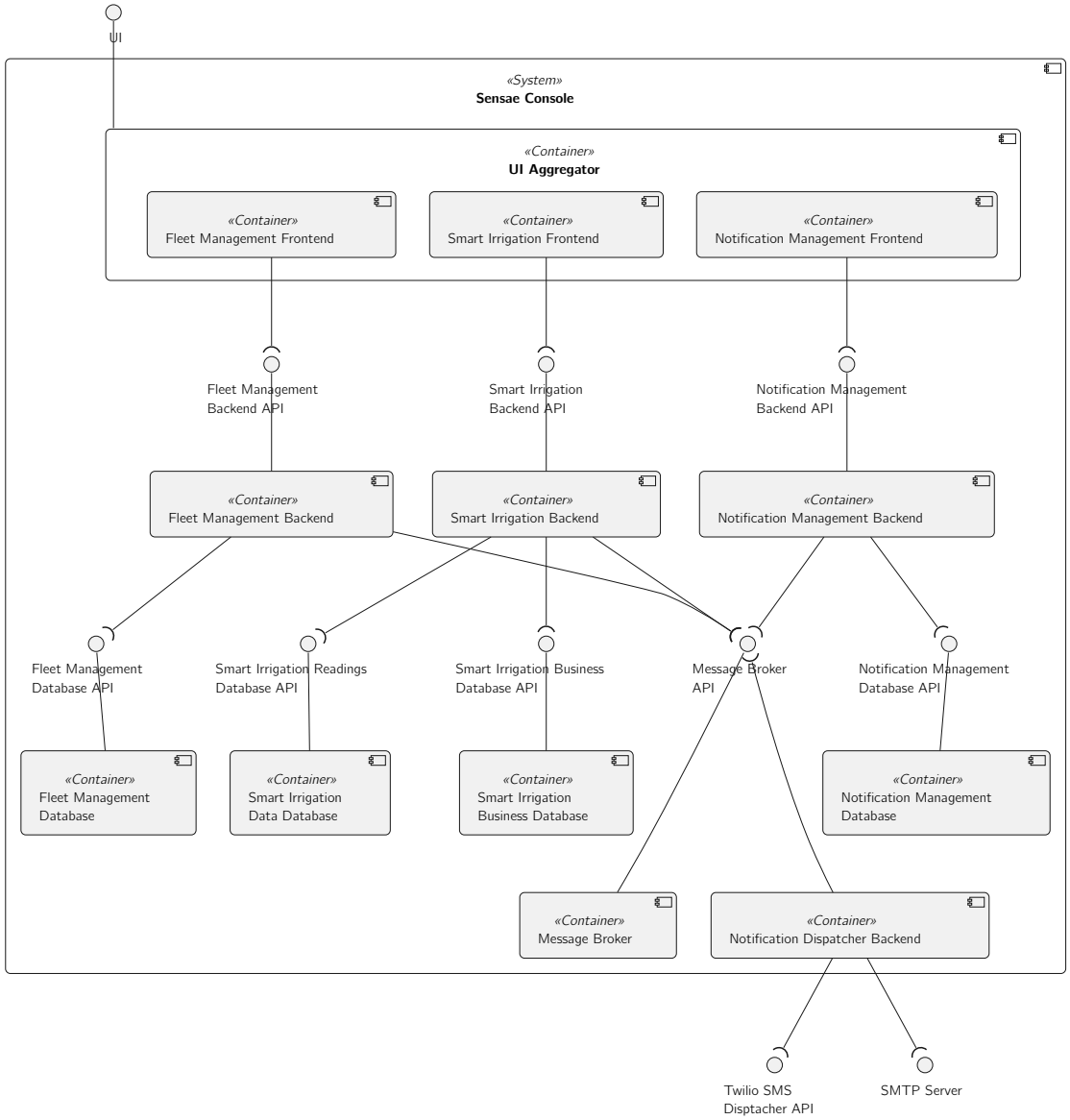

Figure 5.21: Container Level - Service Scope - Logical View Diagram

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 5.3.3, 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 5.3.2.

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

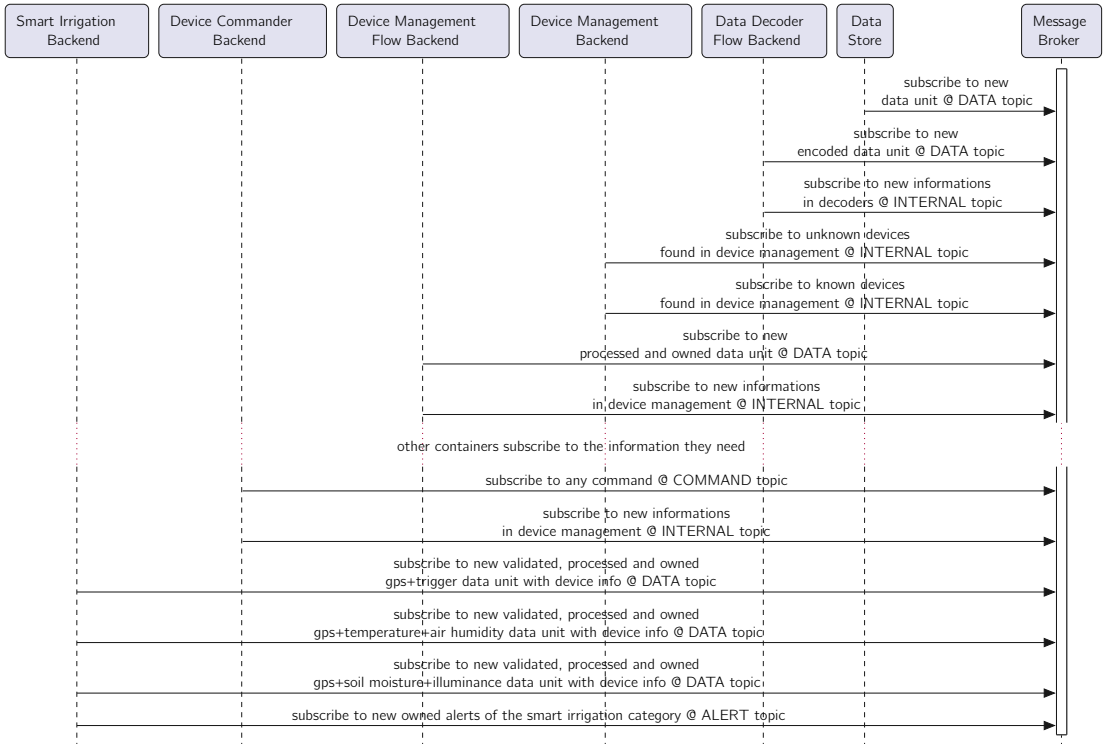

Figure 5.22: Container Level - System/Container Initialization - Process View Diagram

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 lifecycle 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 5.23,note that this only occurs in the Internal Topic.

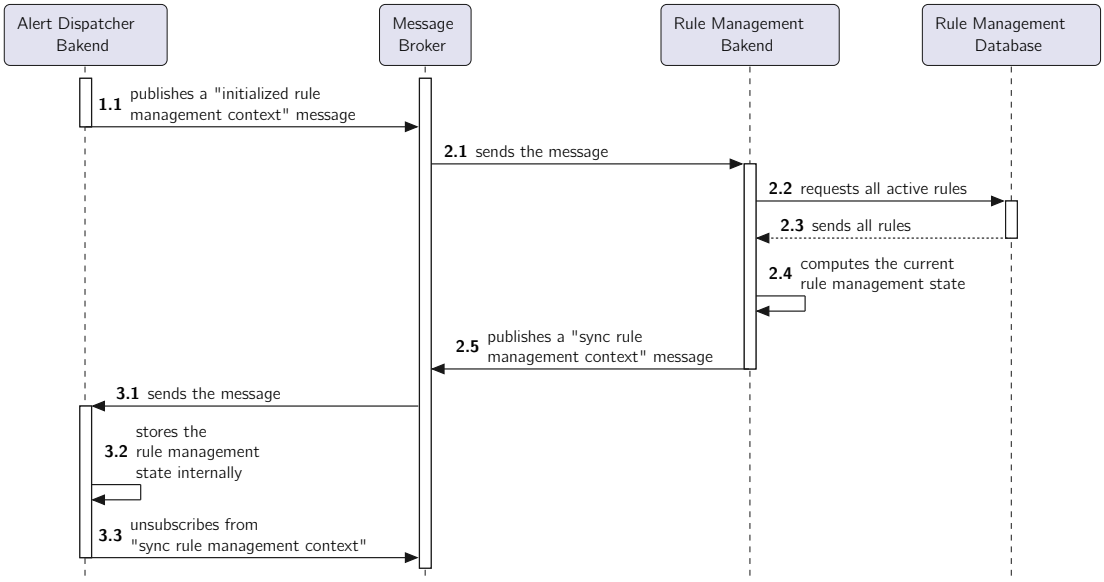

Figure 5.23: Container Level - System/Container Initialization - Part 2 - Process View Diagram

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 5.24 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.

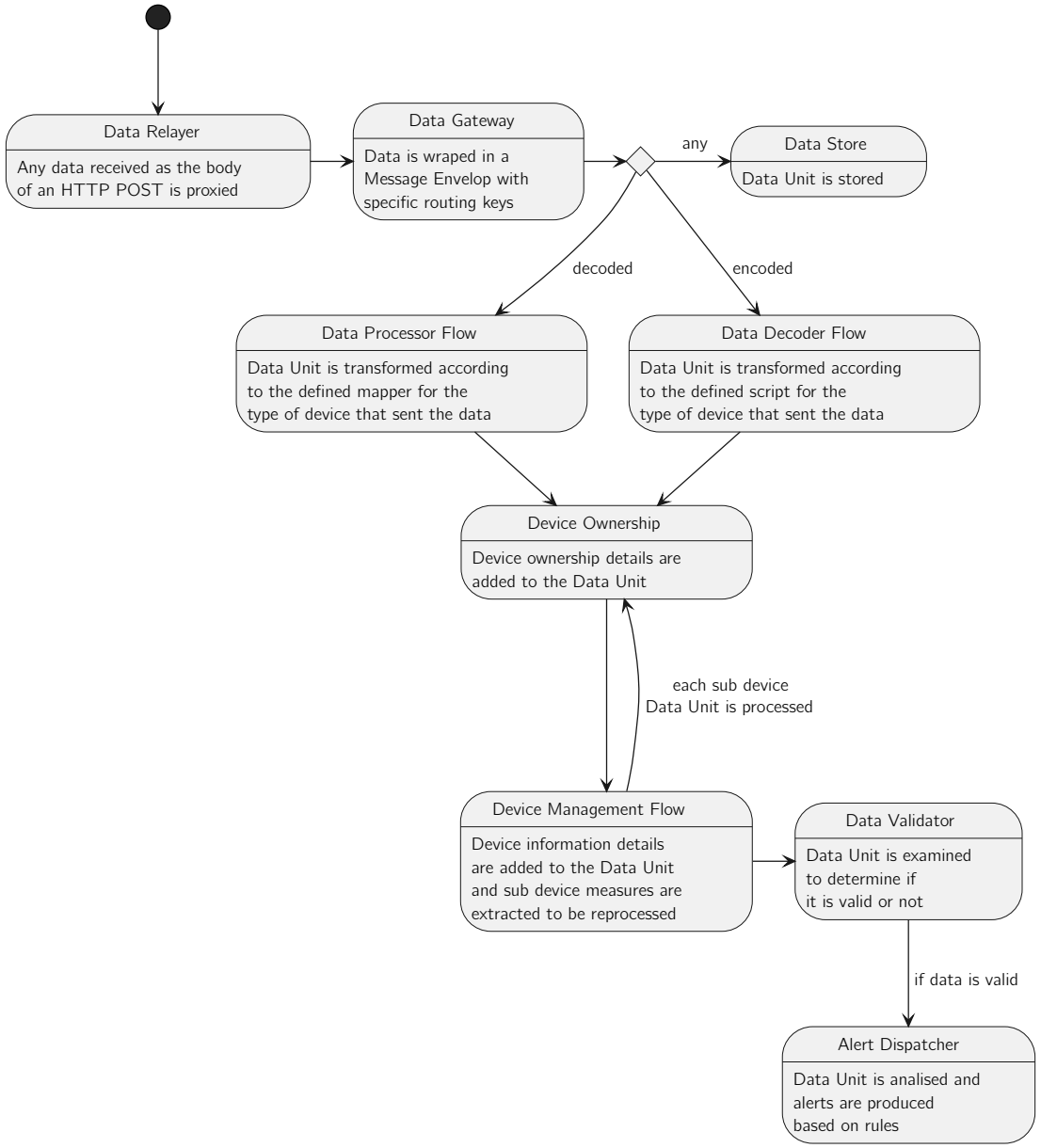

Figure 5.24: Container Level - Data Flow - Diagram

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 5.25 and Figure 5.26, addresses how state is managed in Data Decoder Flow Backend and most **Data Flow Scope** containers.

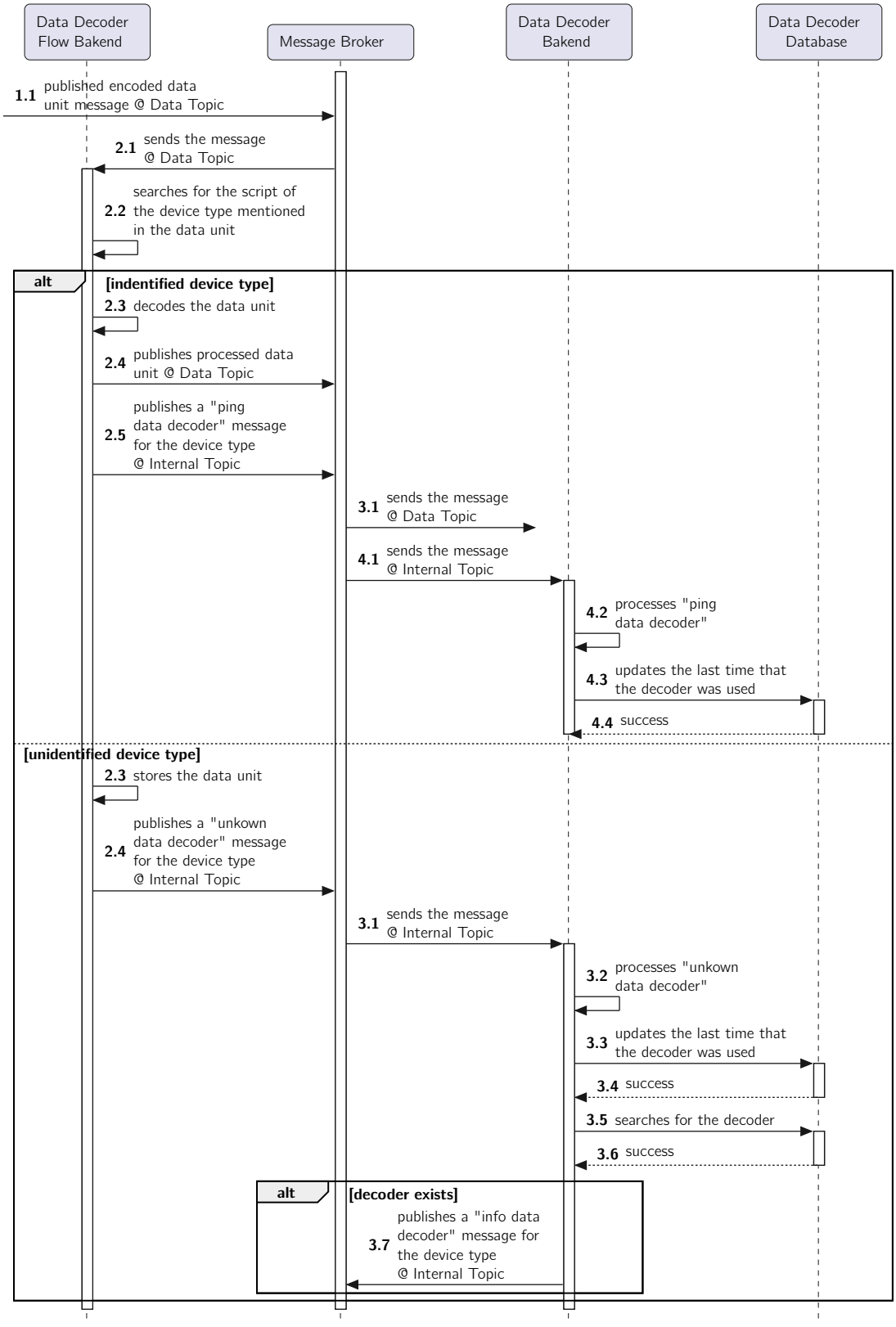

Figure 5.25: Container Level - Data Decoder Operation Part 1 - Process View Diagram

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.

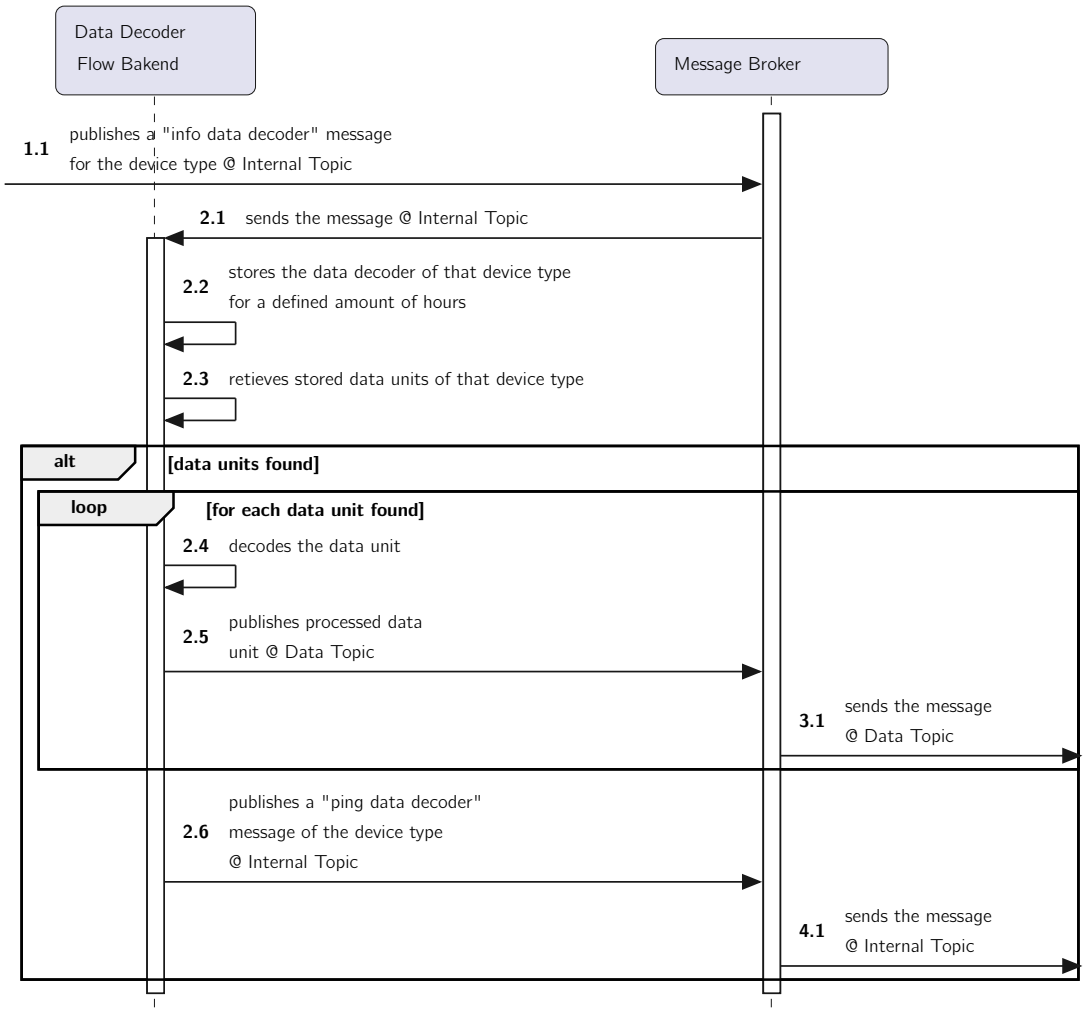

Figure 5.26: Container Level - Data Decoder Operation Part 2 - Process View Diagram

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 5.27 and Figure 5.28 present some of the common operations that happen in that scope.

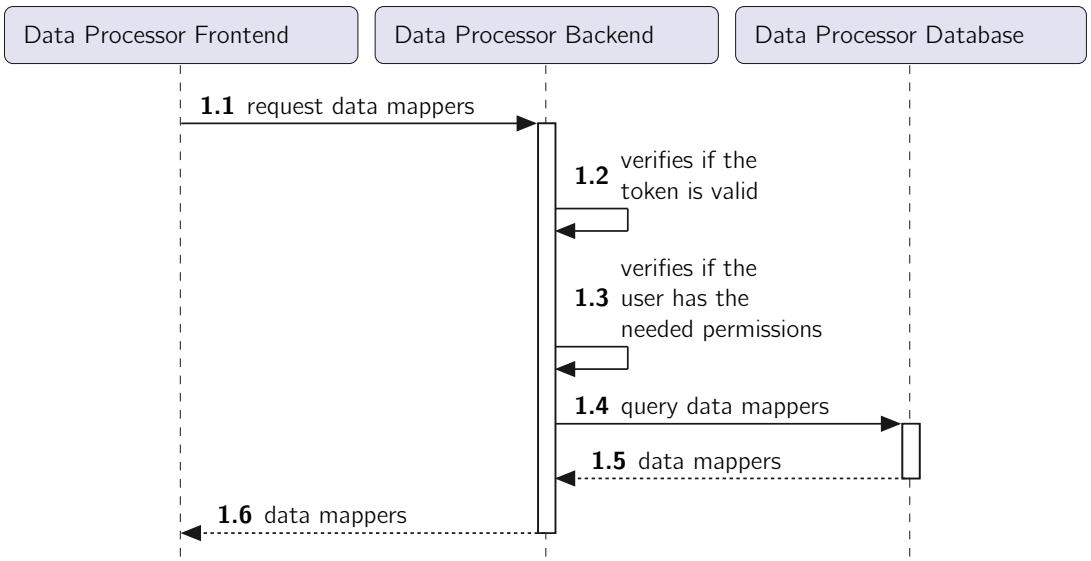

Figure 5.27: Container Level - Consult Data Processors - Process View Diagram

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 5.28 displays an example of this occurrence.

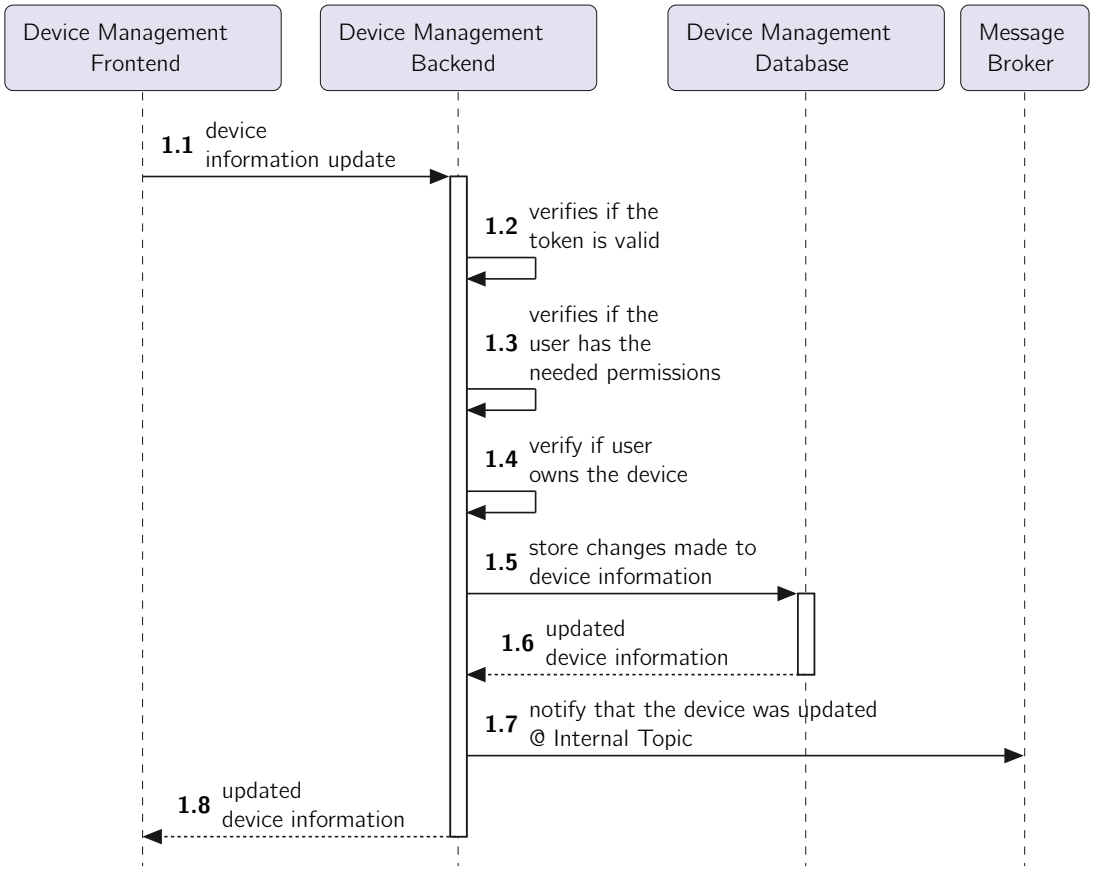

Figure 5.28: Container Level - Edit Device Information - Process View Diagram

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 5.3.2, 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 5.29 and Figure 5.30 - 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.

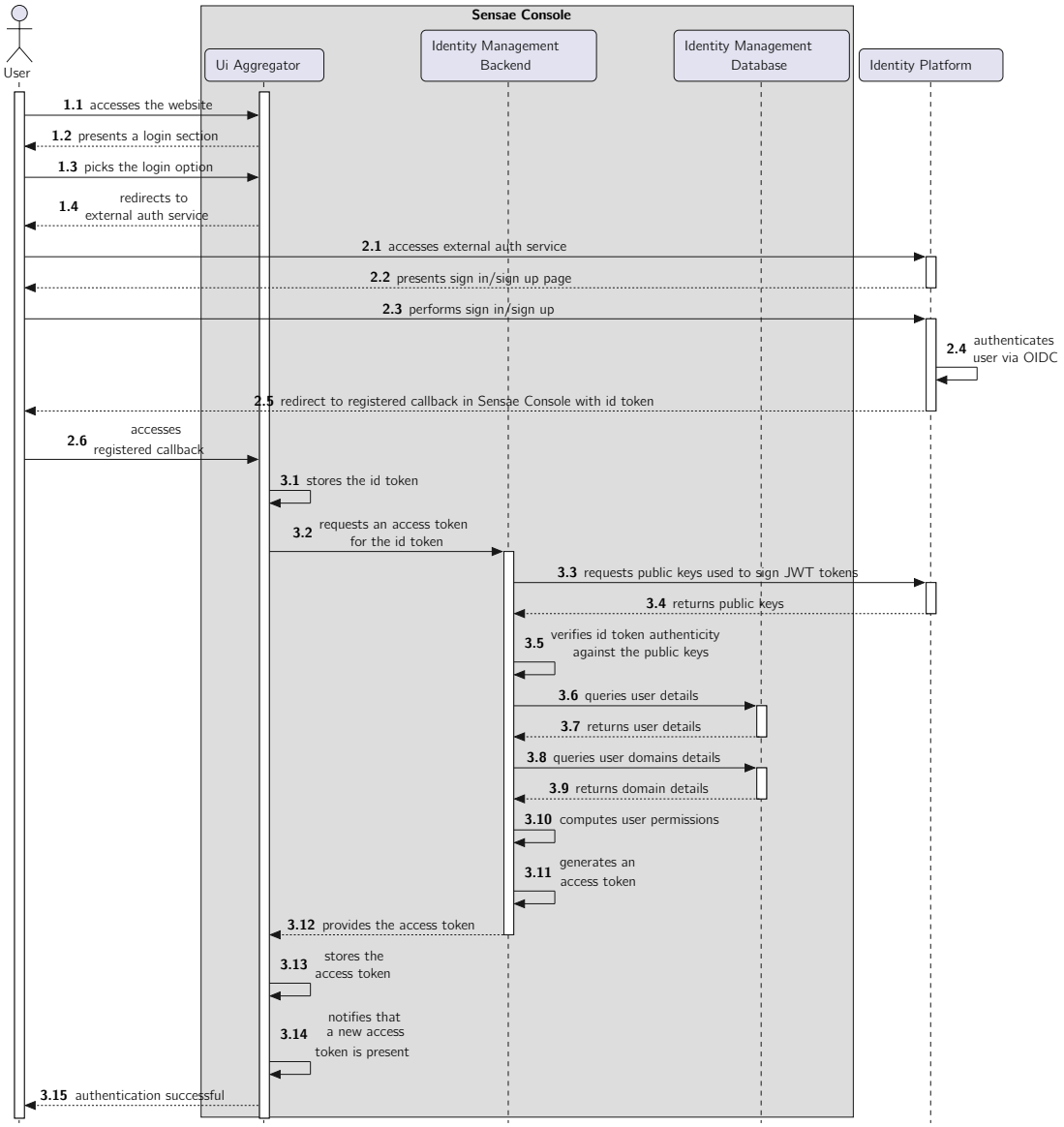

Figure 5.29: Container Level - User Authentication - Process View Diagram

This diagram ilustrastes 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 5.30.

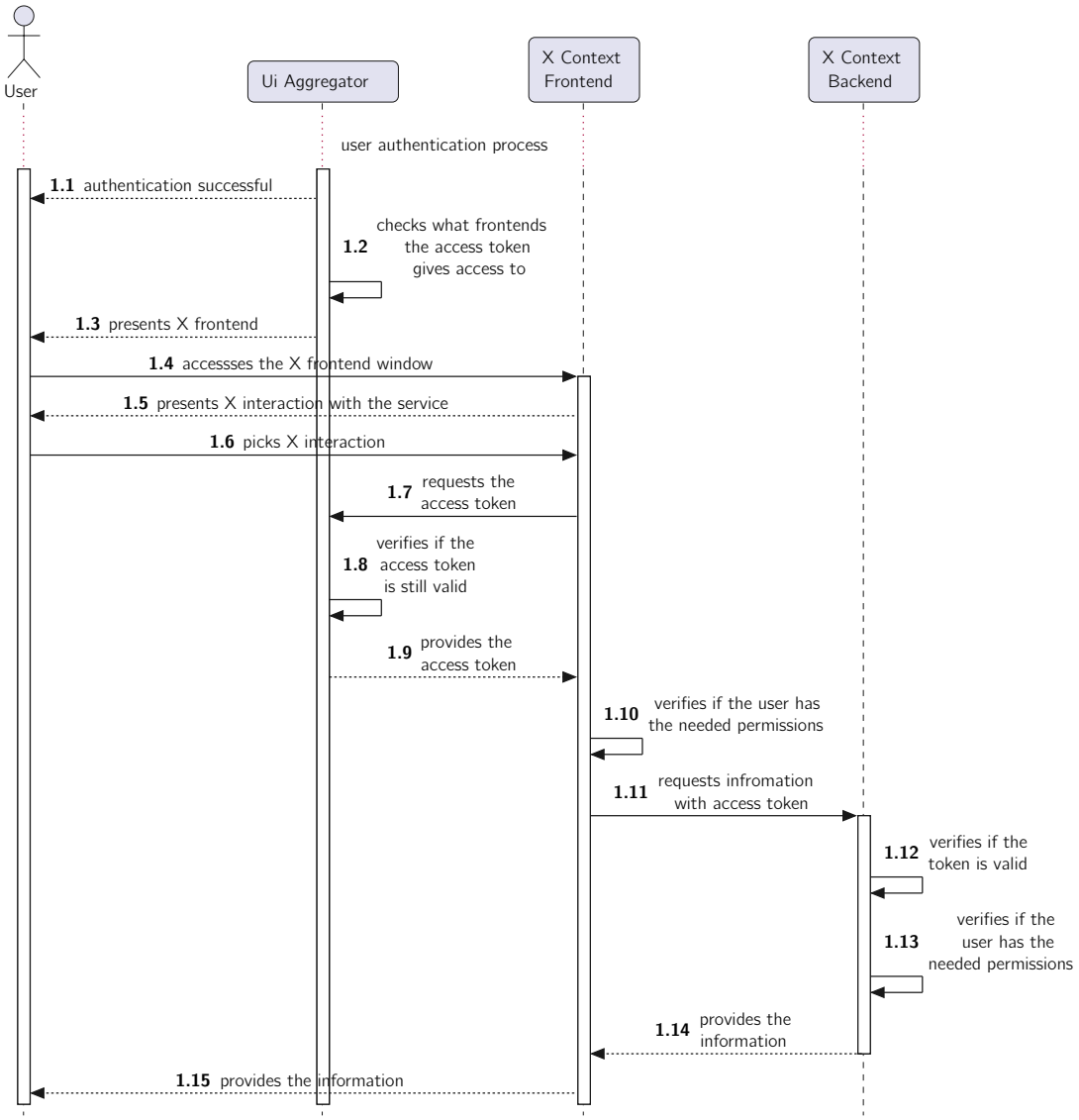

Figure 5.30: Container Level - User Authorization - Process View Diagram

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 5.5.3.

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 5.31). Authentication details will be omitted for brevity reasons.

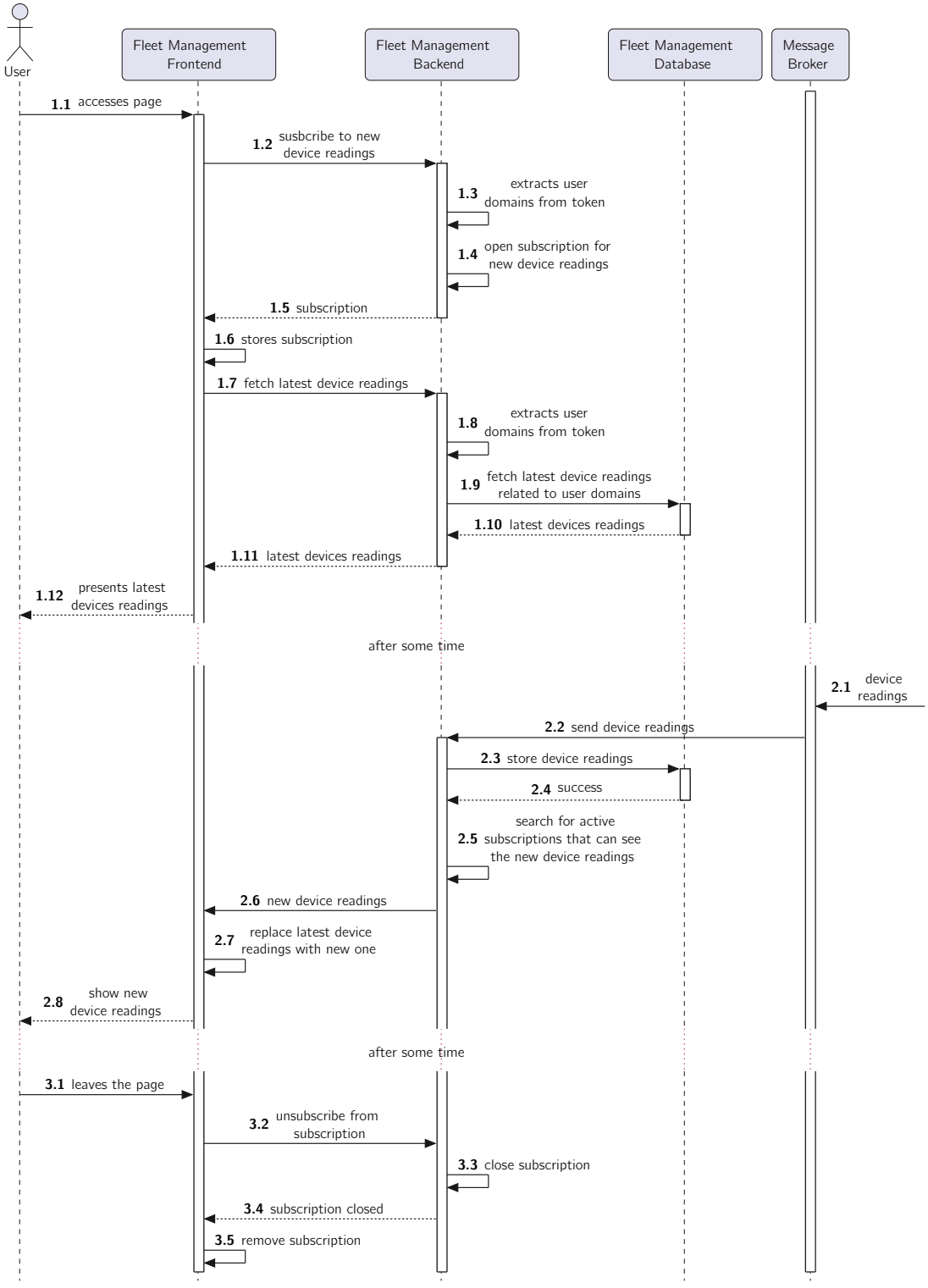

Figure 5.31: Container Level - Consult Device Live Location via Fleet Management - Process View Diagram

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 5.32 describes how a user receives notifications via several different delivery channels. For brevity reasons the subscription process is omitted.

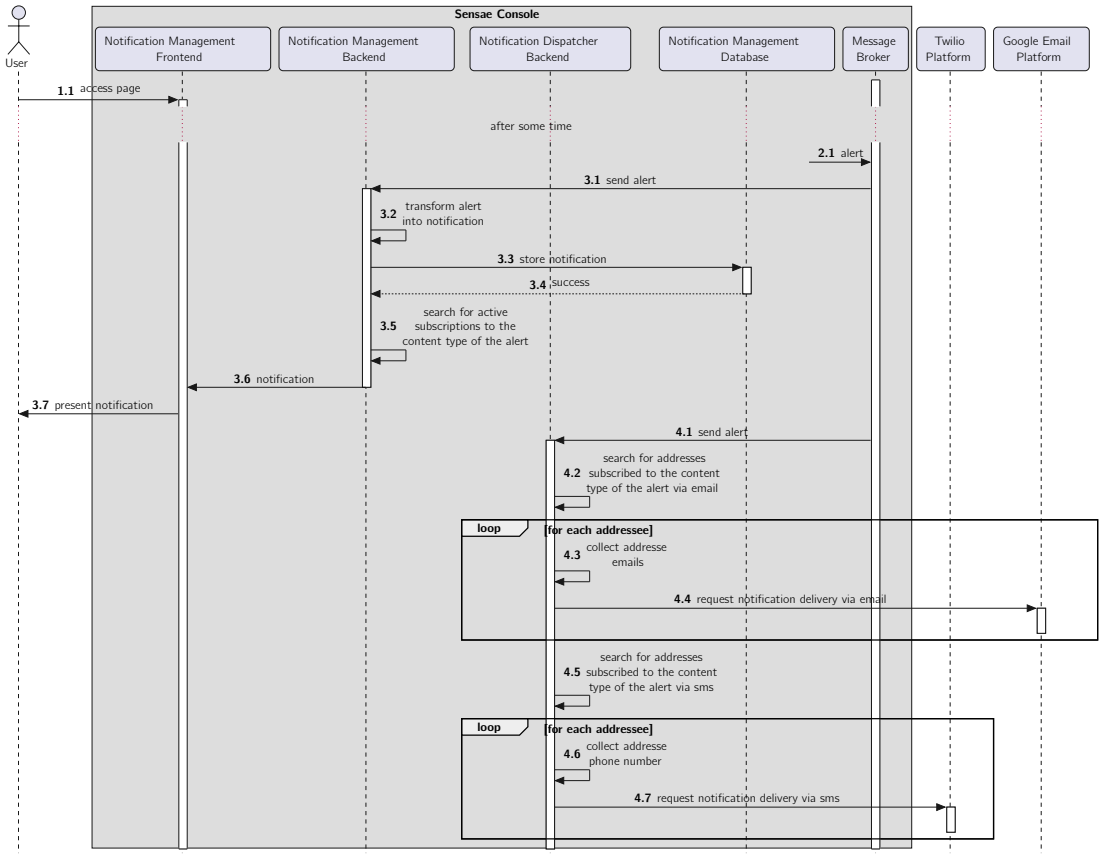

Figure 5.32: Container Level - Receive notification via Notification Management - Process View Diagram

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 5.33, this process is presented.

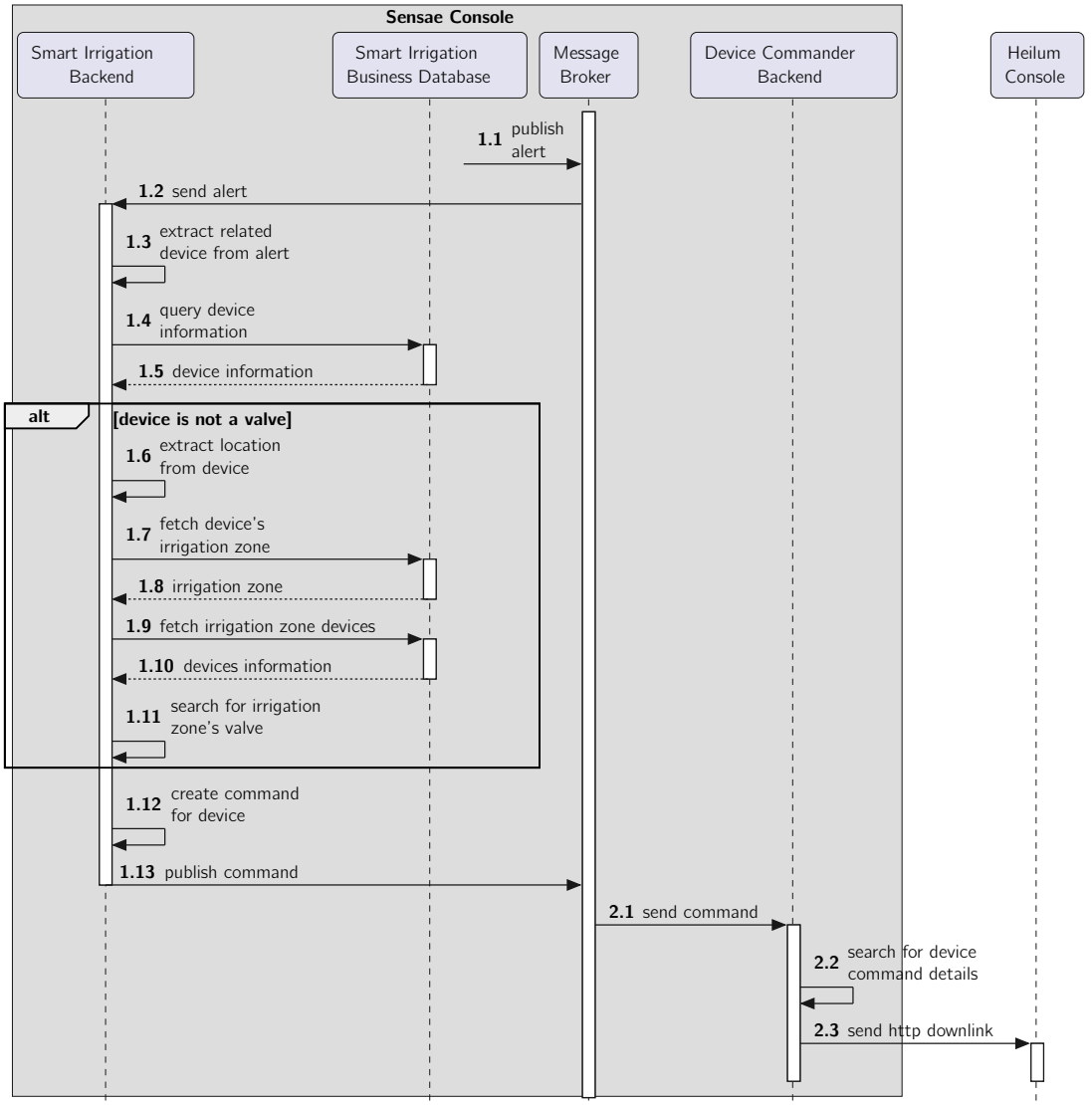

Figure 5.33: Container Level - Valve Activation Process via Smart Irrigation - Process View Diagram

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 5.4.2 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 5.34.

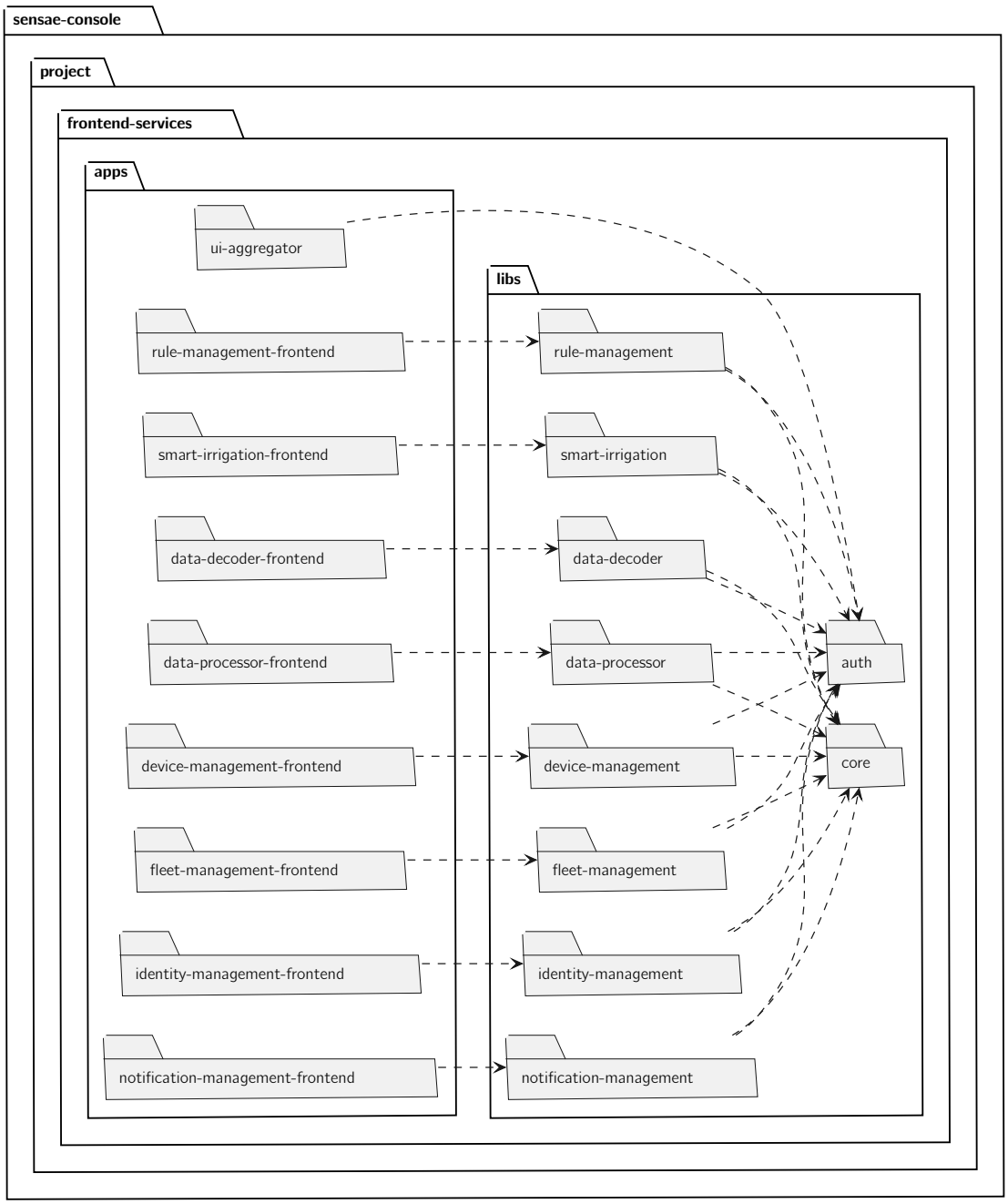

Figure 5.34: Container Level - Frontend Services - Development View Diagram

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 textitauth package.

Backend services are organized according to the diagram in Figure 5.35.

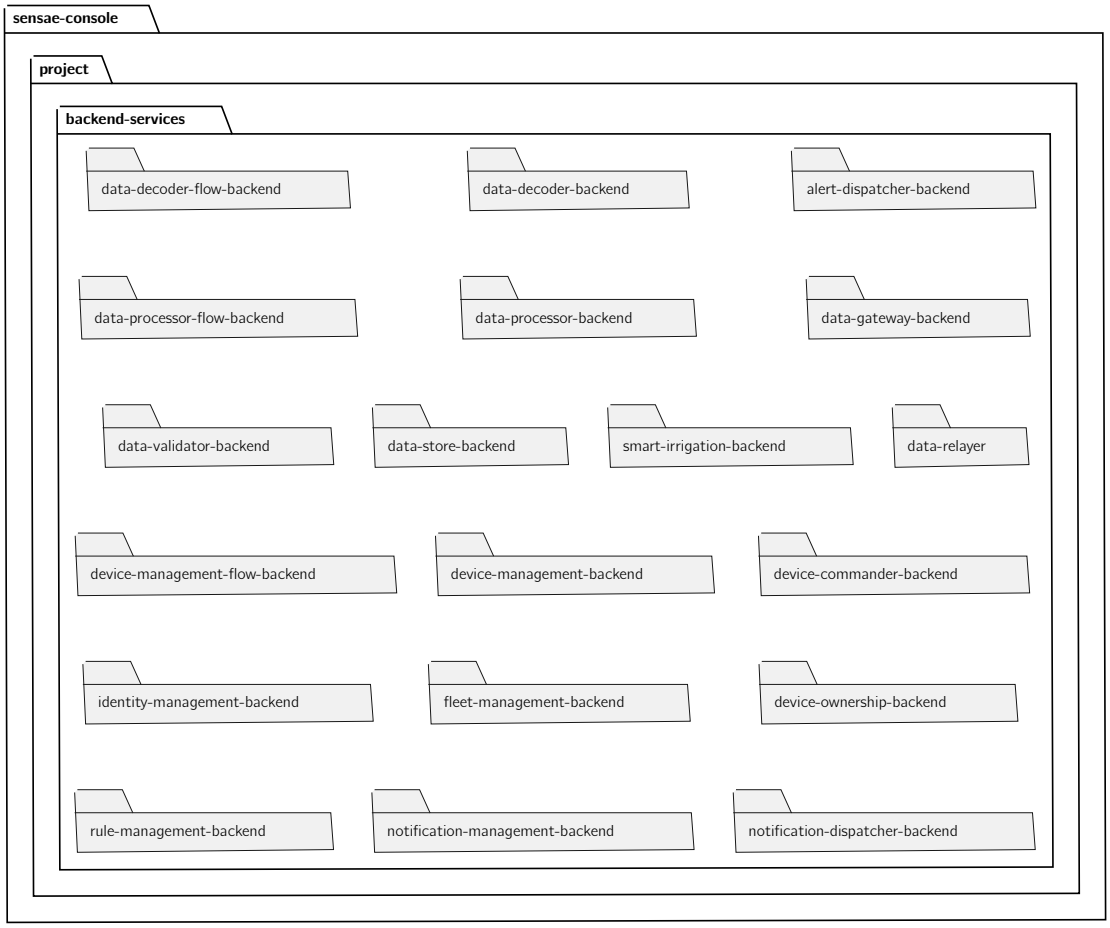

Figure 5.35: Container Level - Backend Services - Development View Diagram

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 5.35.

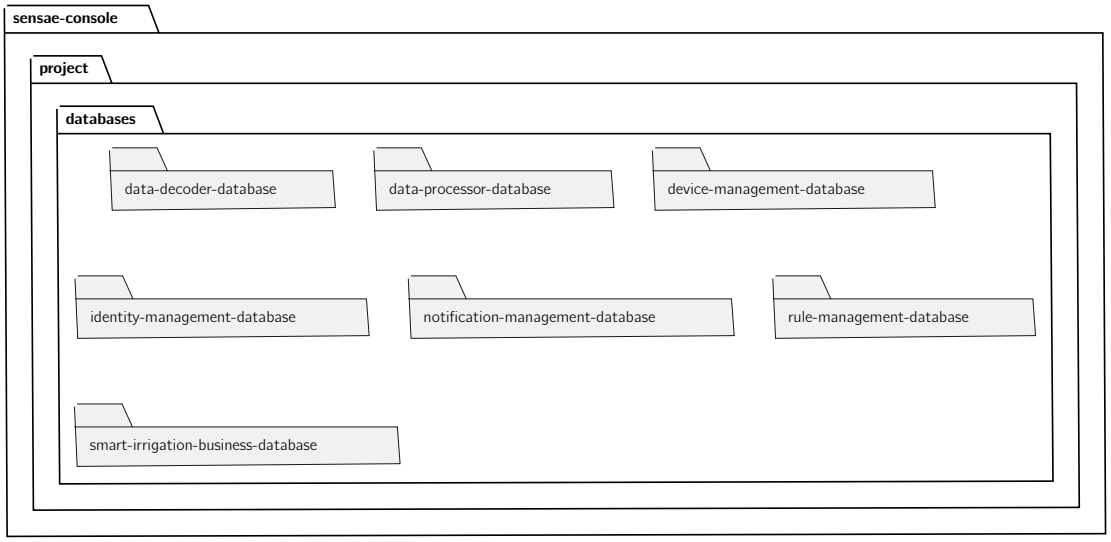

Figure 5.36: Container Level - Database Services - Development View Diagram

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 5.37), 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.

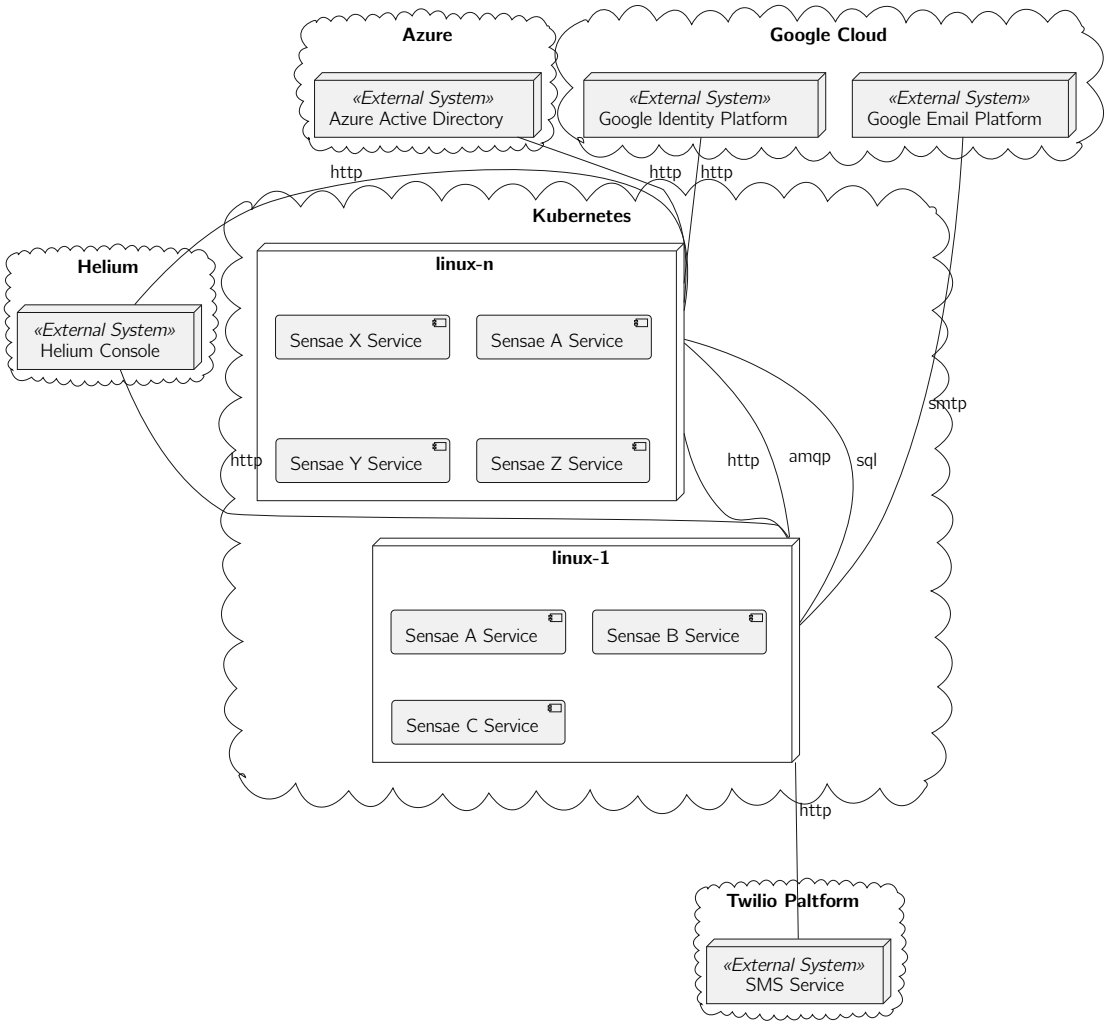

Figure 5.37: Container Level - Physical View Diagram

Due to time constrains the environment was not deployed to *Kubernetes* and the solution is instead orchestrated using *Docker Compose* in a single node/server.

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

5.4.3 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 5.38.

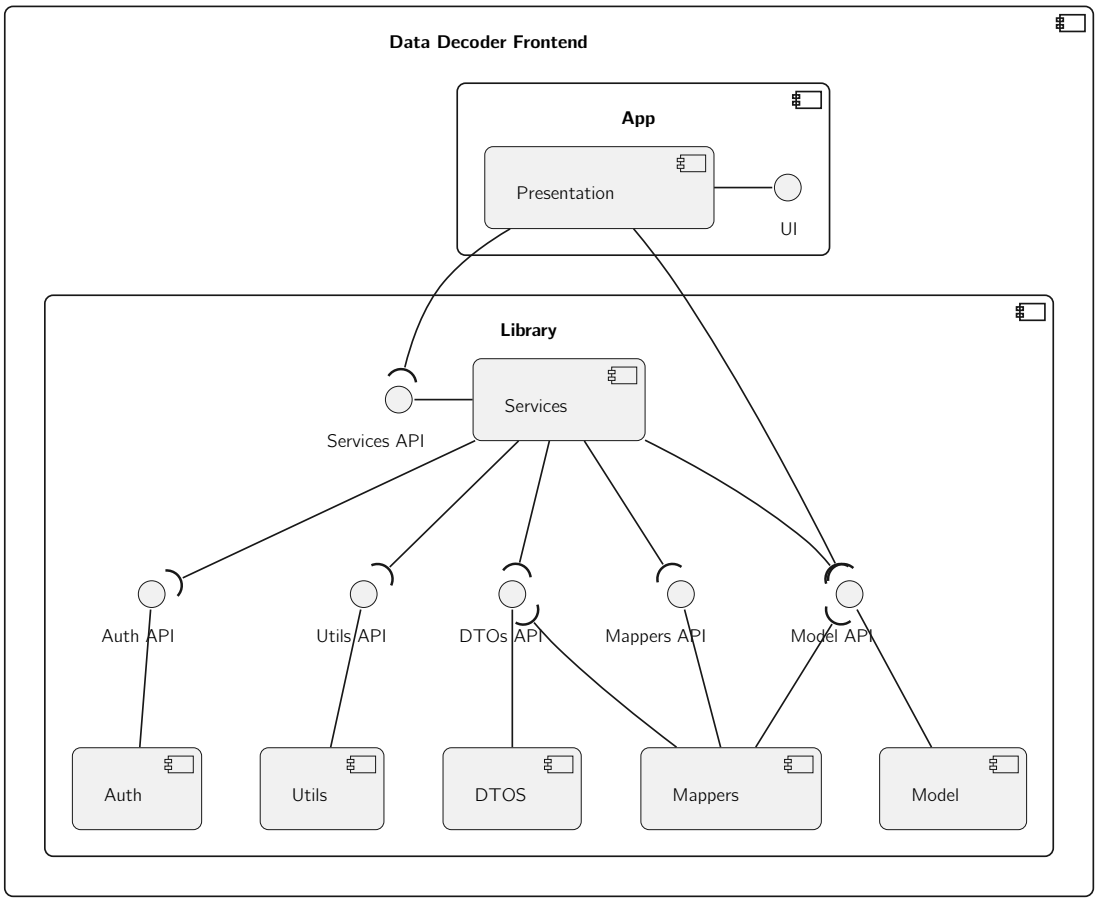

Figure 5.38: Component Level - Data Decoder Frontend - Logical View Diagram

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 5.39.

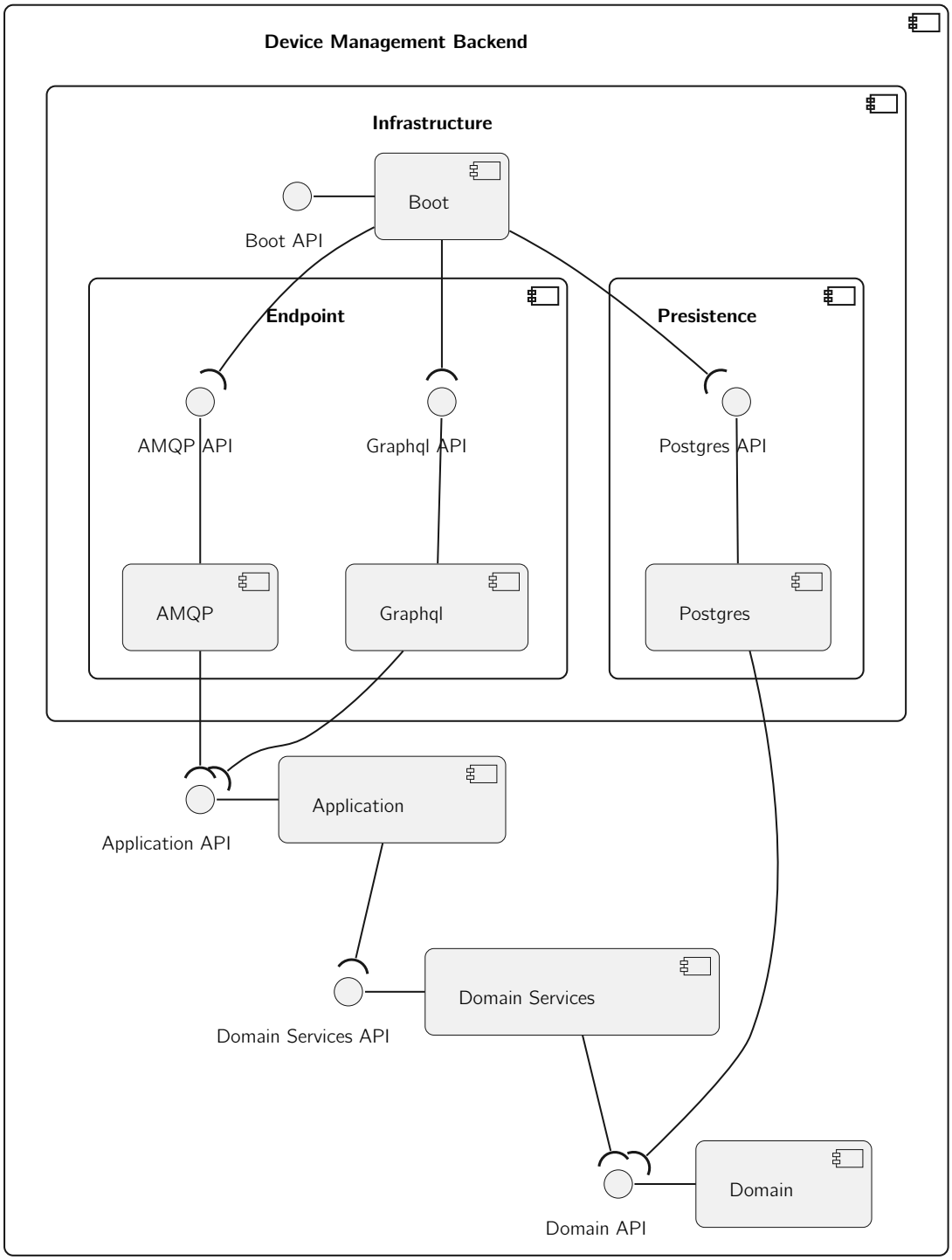

Figure 5.39: Component Level - Device Management Backend - Logical View Diagram

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 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 5.40.

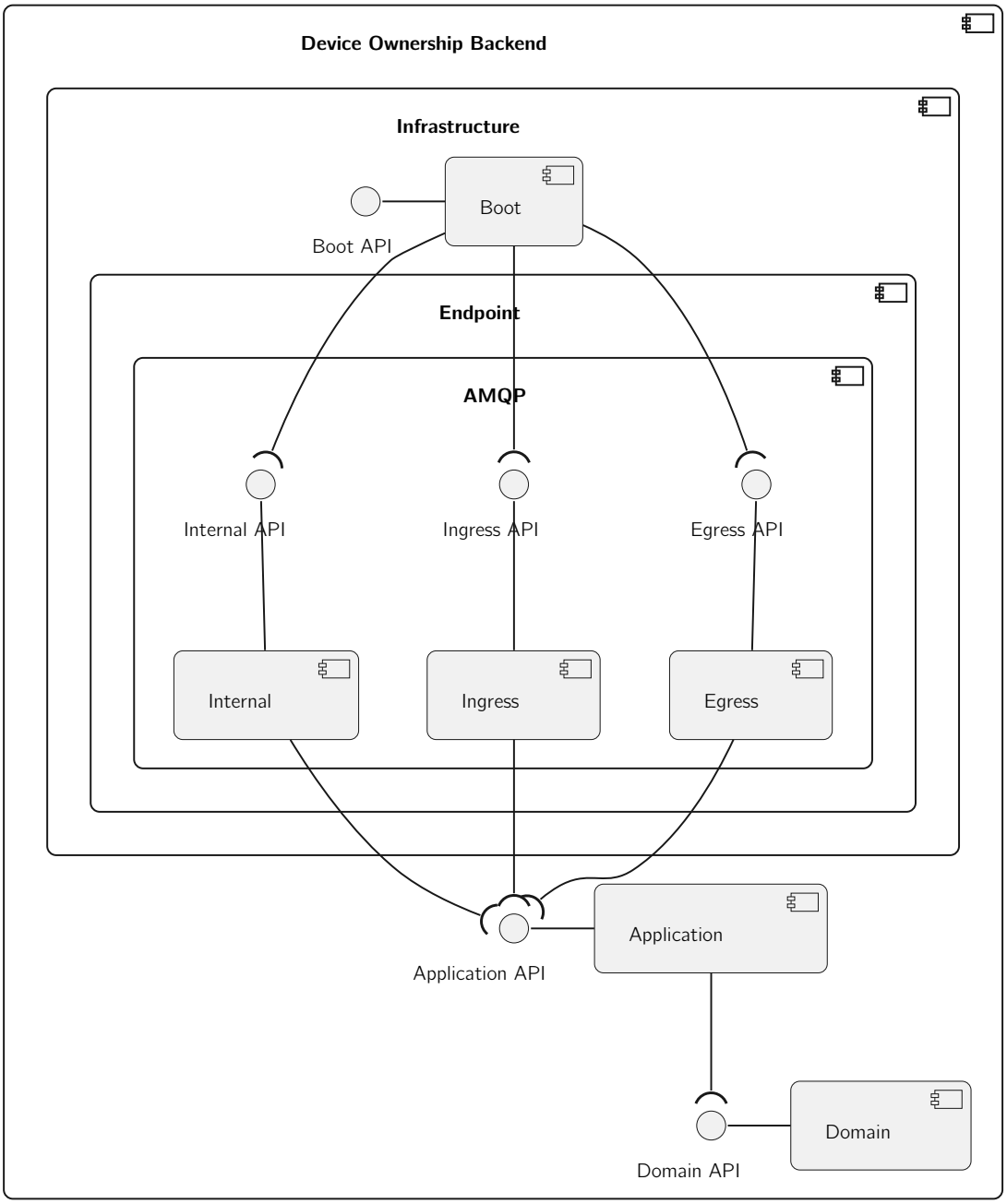

Figure 5.40: Component Level - Device Ownership Backend - Logical View Diagram

This architecture is used on the containers: (i) Device Management Flow Backend, (ii) Data Decoder Flow Backend, (iii) Data Processor Flow Backend, (iv) Alert Dispatcher Backend, (v) Device Ownership Backend, and (vi) Data Validator 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 Data Gateway, Device Commander and Data Store backend containers have architectures that derive from this one and can be consulted in Appendix ***TODO***.

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 5.41, details this process.

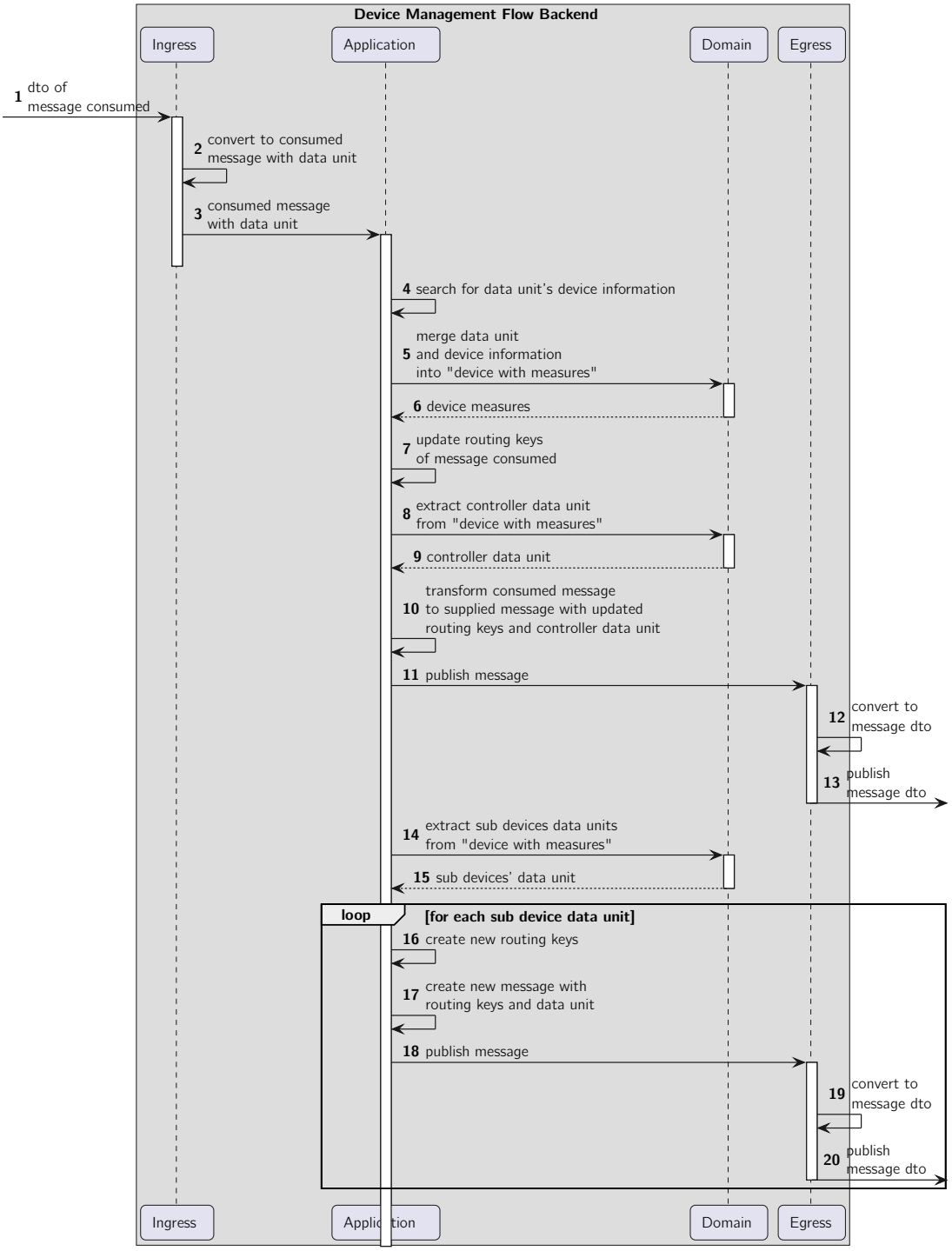

Figure 5.41: Component Level - Process Data Unit in Device Management Flow Backend - Process View Diagram

As presented in the diagram, for each sub device of the controller a new data unit with that device measures is published in the system.

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 5.42, tackles this concern.

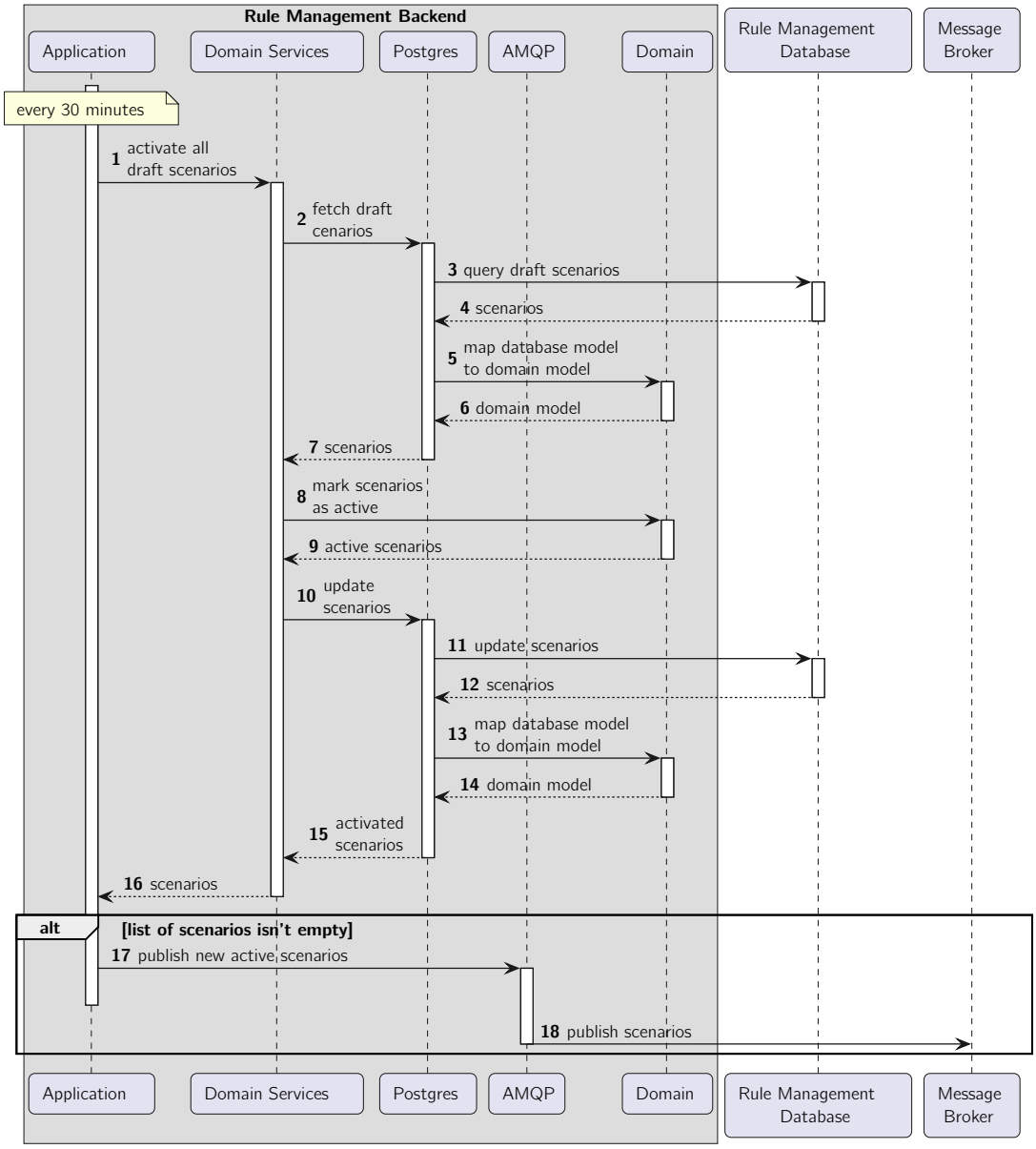

Figure 5.42: Component Level - Deploy Draft Rule Scenarios in Rule Management Backend - Process View Diagram

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 5.43, Figure 5.44 and Figure 5.45 describe this view at the components level.

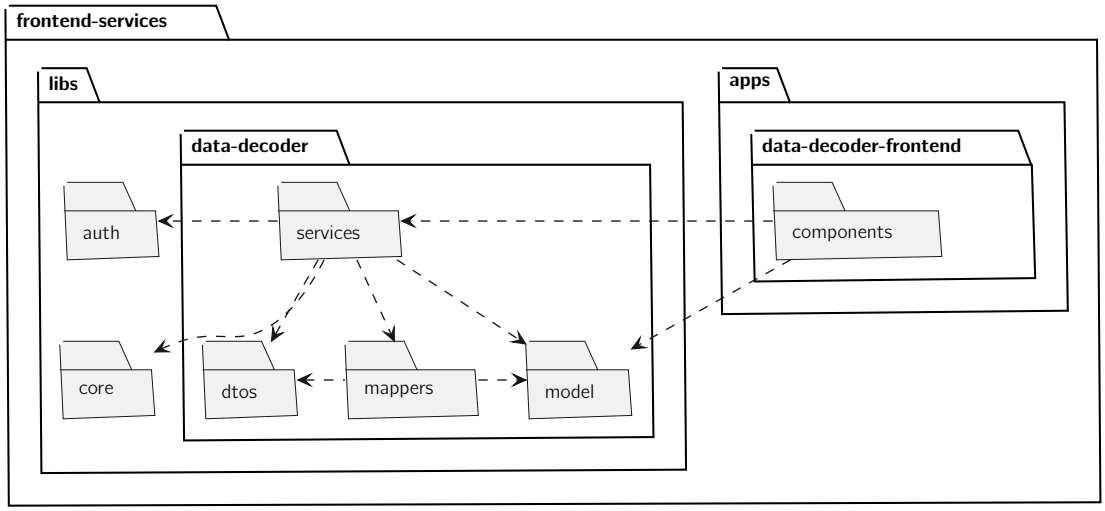

Figure 5.43: Component Level - Data Decoder Frontend - Development View Diagram

The packages presented correspond to the components described in the logical view (Figure 5.38). 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.

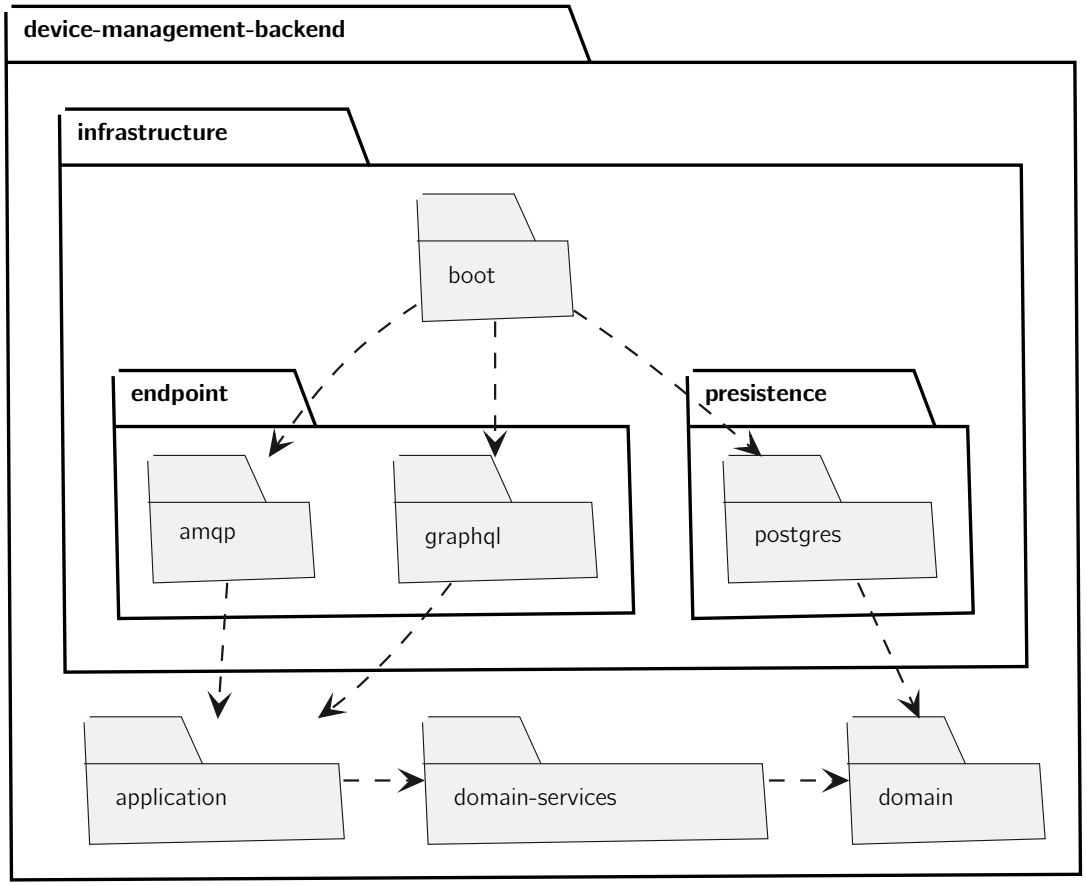

Figure 5.44: Component Level - Device Management Backend - Development View Diagram

The packages presented correspond to the components described in the logical view (Figure 5.39). The names given in both views differ only on the case used.

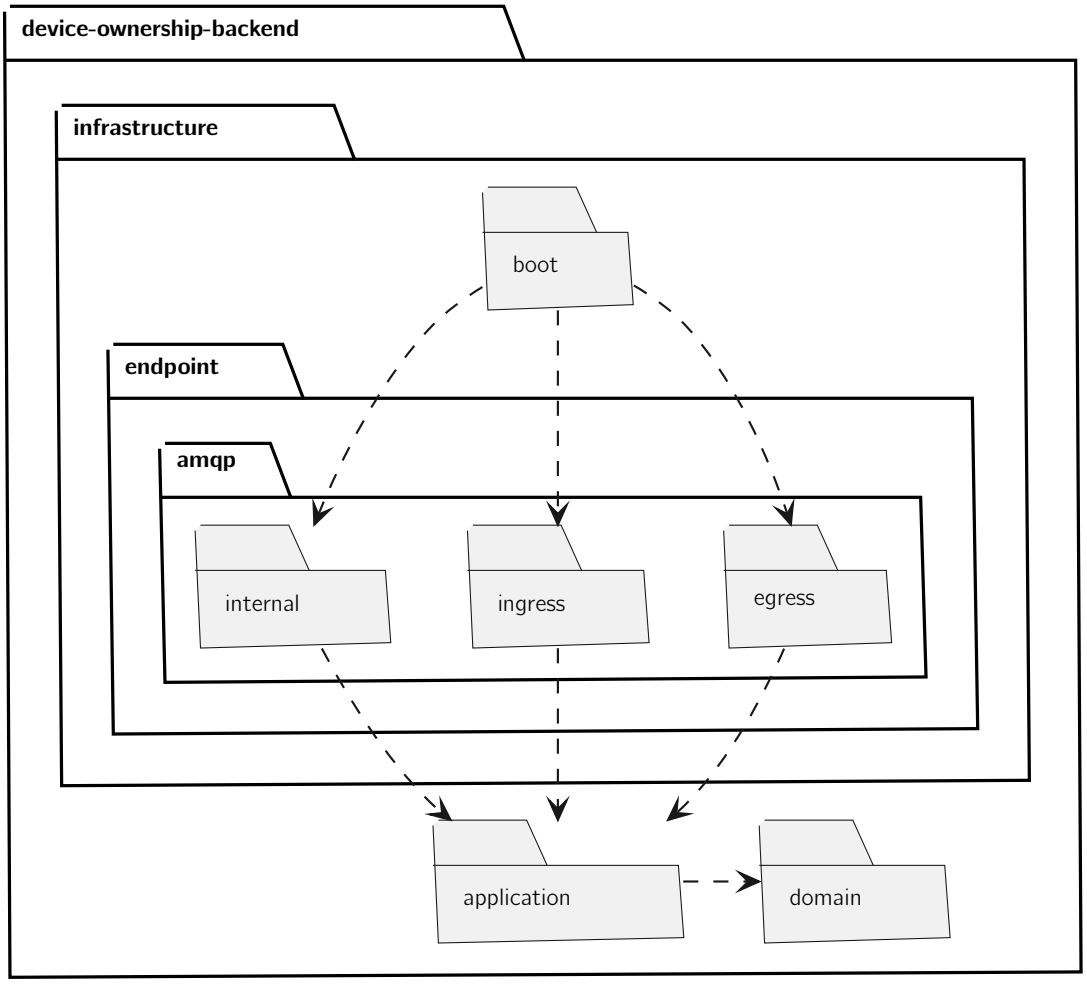

Figure 5.45: Component Level - Device Ownership Backend - Development View Diagram

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.

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.

5.5 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.

5.5.1 Backend Segregation

There are three main architectural approaches to this topic: Monolithic Backend - Richardson 2021b -, Service Oriented Architecture (SOA) - IBM 2021b - or Microservices - Fowler and 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 2021b, 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 5.46, summarizes how each architect scales, the SOA behaves similarly to the microservices architecture presented.

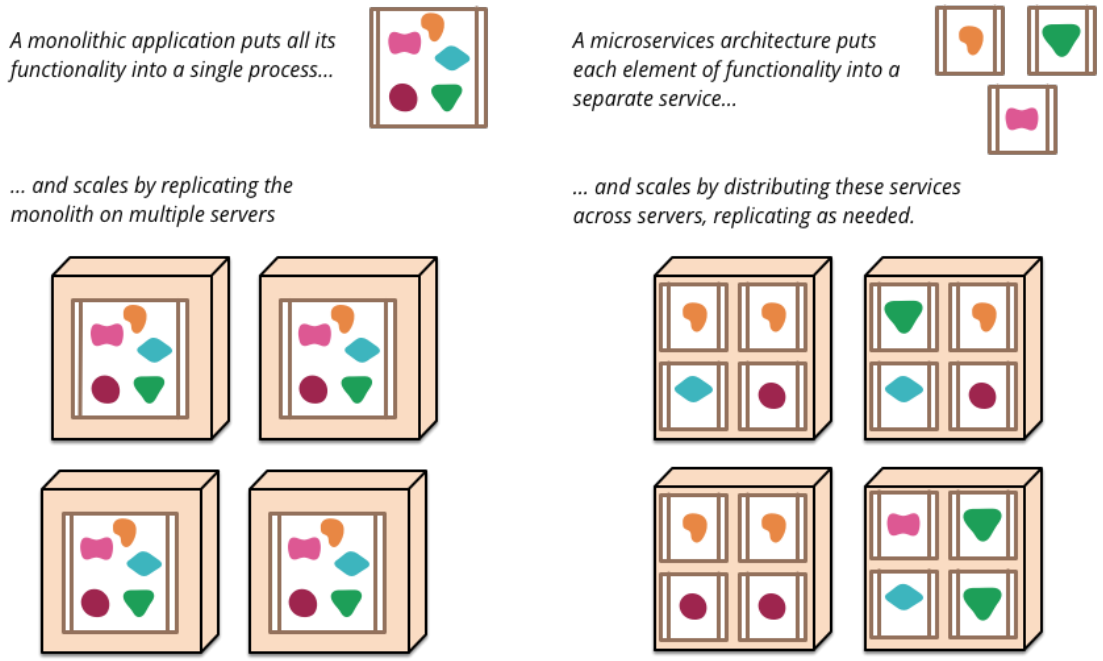

Figure 5.46: Monoliths and Microservices by Fowler and Lewis 2014

The final decision was to follow an architecture based on Microservices even tho this decision comes with 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:

- Well defined boundaries between the various business processes;
- Need to scale the solution early on the road due to high volumes of glsIoT data to process and store;
- High number of independent IoT services to develop;
- High number of costumers with different 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 auditing, security, logging, data storage, data presentation, or business processes. All these services usually communicate via Enterprise Service Bus (ESB). A new requirement or business process requires

every 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. Microservices are more easily extended when/if needed compared with SOA.

5.5.2 Frontend Segregation

5.5.3 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 5.47,presents the normal environment flow for this alternative.

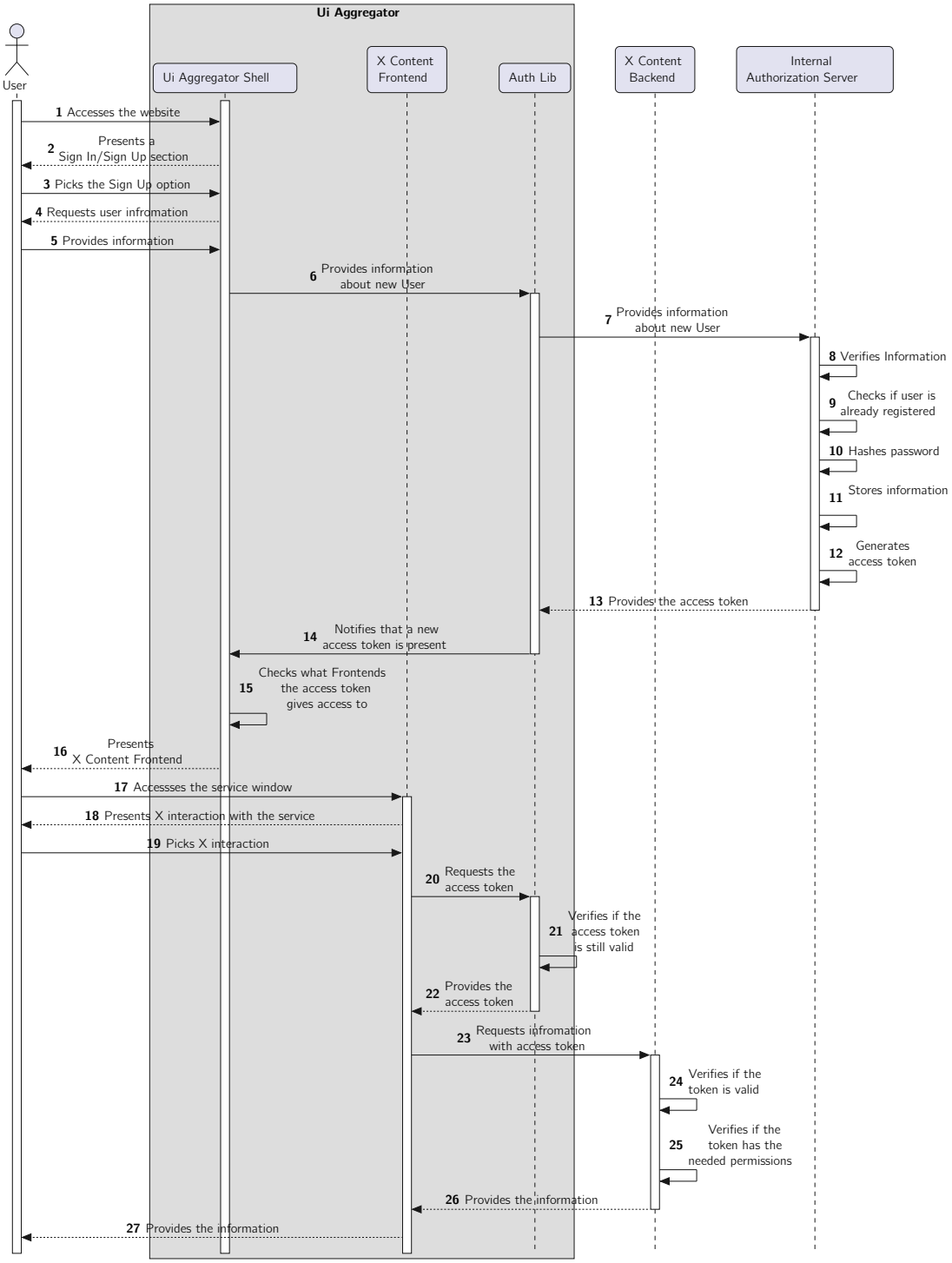

Figure 5.47: User Authorization/Authentication - Internal Authorization Server Alternative - Sequence Diagram

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 5.48, presents the normal environment flow for this alternative.

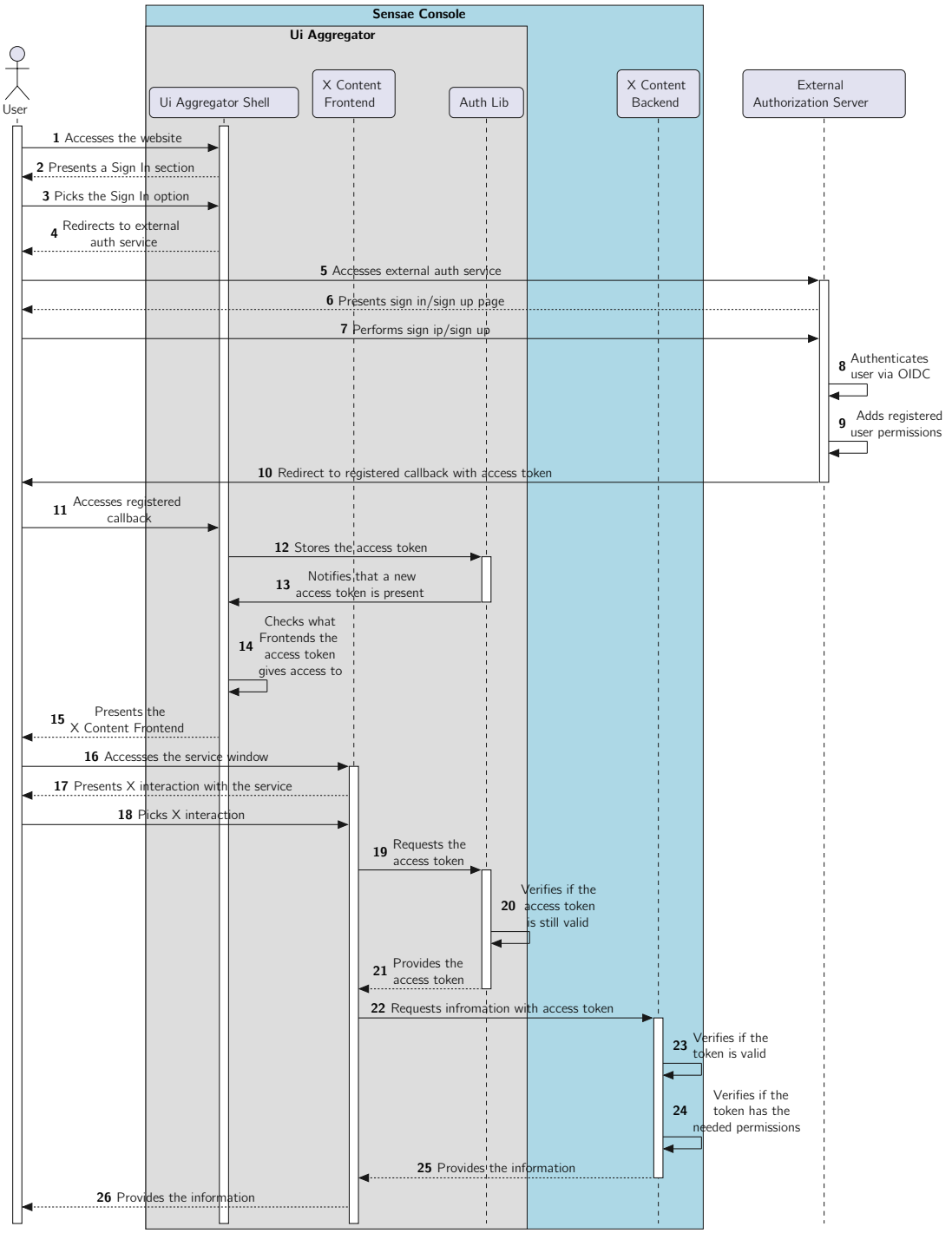

Figure 5.48: User Authorization/Authentication - External Authorization Server Alternative - Sequence Diagram

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 Auth0 2022, (ii) Google Google n.d., (iii) Okta Okta 2022, (iv) Amazon Cognito Amazon 2022 and (v) Azure Active Directory B2C Azure 2022.

The platform Auth0 was tested and it 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 5.49, presents the normal environment flow for this alternative.

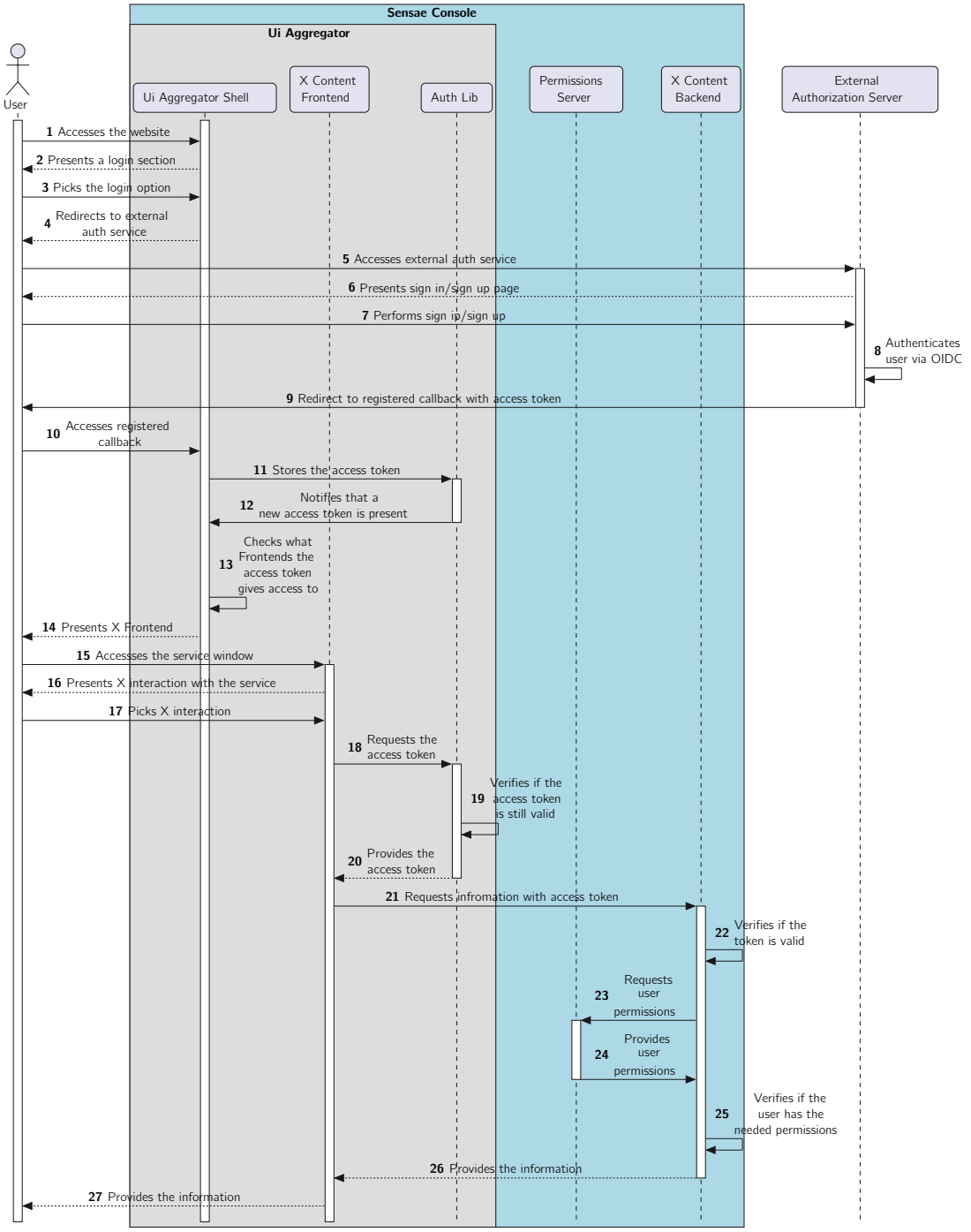

Figure 5.49: User Authorization/Authentication - External Authorization Server with Internal Permissions Server Alternative - Sequence Diagram

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 5.29 and Figure 5.30 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.

5.5.4 Data Streaming/Pipeline

5.5.5 Internal Communication

5.6 Synopsis

Chapter 6

Implementation

Chapter 7

Evaluation of the Solution

Chapter 8

Conclusion

8.1 Achievements

8.2 Unfulfilled Results

8.3 Future Work

8.4 Synopsis

Bibliography

Amazon (2022). *Amazon Cognito*. url: https://aws.amazon.com/cognito/.

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/.

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].

D. Hardt, Ed. (2012). *The OAuth 2.0 Authorization Framework*. url: https://datatracker.ietf.org/doc/html/rfc6749.

Evans, E. (2014). *Domain-Driven Design Reference: Definitions and Pattern Summaries*. Dog Ear Publishing. isbn: 9781457501197. url: https://books.google.pt/books?id=ccRsBgAAQBAJ.

Fowler, Martin and James Lewis (2014). *Microservices*. url: https://www.martinfowler.com/articles/microservices.html.

Google (n.d.). *Google Identity Platform*. url: https://cloud.google.com/identity-platform/.

Harris, Chandler (n.d.). *Microservices vs. monolithic architecture*. url: https://www.atlassian.com/microservices/microservices-architecture/microservices-vs-monolith.

IBM (Jan. 2020a). *What are Message Brokers?* url: https://www.ibm.com/cloud/learn/message-brokers.

– (Jan. 2020b). *What are Message Brokers?* url: https://www.ibm.com/cloud/learn/three-tier-architecture.

– (Mar. 2021a). *Microservices*. url: https://www.ibm.com/cloud/learn/microservices#toc-anti-patte-uScI1WAE.

– (Apr. 2021b). *SOA (Service-Oriented Architecture)*. url: https://www.ibm.com/cloud/learn/soa.

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.

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.

Newman, S. (2021). *Building Microservices*. O'Reilly Media. isbn: 9781492033974. url: https://books.google.pt/books?id=ZvM5EAAAQBAJ.

Okta (2022). *Okta Customer Identity*. url: https://www.okta.com/solutions/secure-ciam/.

OpenID (2014). *OpenID Connect*. url: https://openid.net/connect/.

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

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/.

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.

Richardson, Chris (2021a). *Pattern: Microservice Architecture*. url: https://microservices.io/patterns/microservices.html.

– (2021b). *Pattern: Monolithic Architecture*. url: https://microservices.io/patterns/monolithic.html.

Appendix A

Appendix Title Here

Write your Appendix content here.