INSTITUTO POLITÉCNICO DE LISBOA
INSTITUTO SUPERIOR DE ENGENHARIA DE LISBOA

SensiFlow

48267 Teodosie Pienescu a48267@alunos.isel.pt

48282 Francisco Costa a48282@alunos.isel.pt

48265 Tiago Filipe a48265@alunos.isel.pt

Orientador: Prof. Nuno Leite

Projeto e Seminário

Report

July 2023

Abstract

The use of surveillance cameras has become more widespread due to the increase in criminal activity. Nowadays, these devices are found in almost every establishment. With advancements in technology, these cameras now offer various useful features, including the ability to alert authorities about specific behaviors and utilize facial recognition. However, upgrading existing devices to support these enhanced features can be expensive. Our project aims to add a feature to devices that do not have it by default.

In this work we describe a software development project aimed at enabling organizations to effectively manage person detection in their cameras and devices. The project leverages machine learning models to make inferences from camera feeds, offering valuable insights that aid in making strategic decisions based on space occupancy. The primary goal of the project is to develop a comprehensive and scalable system capable of meeting the specific requirements of implementing person detection in various organizational settings.

The system incorporates state-of-the-art machine learning techniques to accurately identify and track individuals within camera frames. By analyzing the data gathered from the camera feeds, the software provides real-time information on space occupation. This enables companies to optimize resource allocation, and make informed decisions regarding space utilization and layout planning.

Through this project, the developed software system aims to empower organizations with valuable insights derived from person detection, facilitating data-driven decision-making and enabling strategic planning based on accurate occupancy information.

Keywords: Software development, Person detection, Machine learning models, Camera feeds, Strategic decision-making, Resource allocation, Scalable system.

List of Acronyms

AI	Artificial Intelligence
API	Application Programming Interface
CI	Continuous integration
CRUD	Create, Read, Update and Delete
CSR	Client Side Rendering
CSRF	Cross-site request forgery
DAL	Data Access Layer
DASH	Dynamic Adaptive Streaming over HTTP
DOM	Document Object Model
ESM	EcmaScript Modules
FPS	Frames Per Second
GCP	Google Cloud Platform
GPU	Graphics Processing Unit
gRPC	Google Remote Procedure Call
HLS	HTTP Live Streaming
hmr	Hot Module Reload
HTTPS	Hyper Text Transfer Protocol Secure

IPC	Inter-Process communication
JDBI	Java Database Interface
JPA	Java Persistence API
NPM	Node Package Manager
ORM	Object-Relational Mapping
R-CNN	Region-based Convolutional Neural Networks
RBAC	Role Based Access Control
RTCP	Real-time Control Protocol
RTP	Real-time Transport Protocol
RTSP	Real Time Streaming Protocol
RTSPS	Real Time Streaming Protocol Secure
SPA	Single Page Application
SQL	Structured Query Language
SSE	Server-sent events
TLS	Transport Layer Security
URL	Uniform Resource Locator
WebRTC	Web Real-Time Communications
XSS	Cross-site scripting
YOLO	You Only Look Once

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Nowadays with all the technology available and ways of obtaining data, data analysis has gained a significant importance across several domains, including retail, healthcare, and transportation. We understanding that the flow and density of people is crucial in improving efficiency, safety, and customer satisfaction, and since there are already was of obtaining the data to perform this analysis we decided to create SensiFlow. SensiFlow is an organization targeted app that uses person detection technology on received video feed to calculate metrics such as occupancy rates in a given area. By leveraging state-of-the-art deep learning approaches, SensiFlow can detect individuals within a space and provide real-time occupancy data and insights. Mostly it provides the ability to perform analysis on the given data.

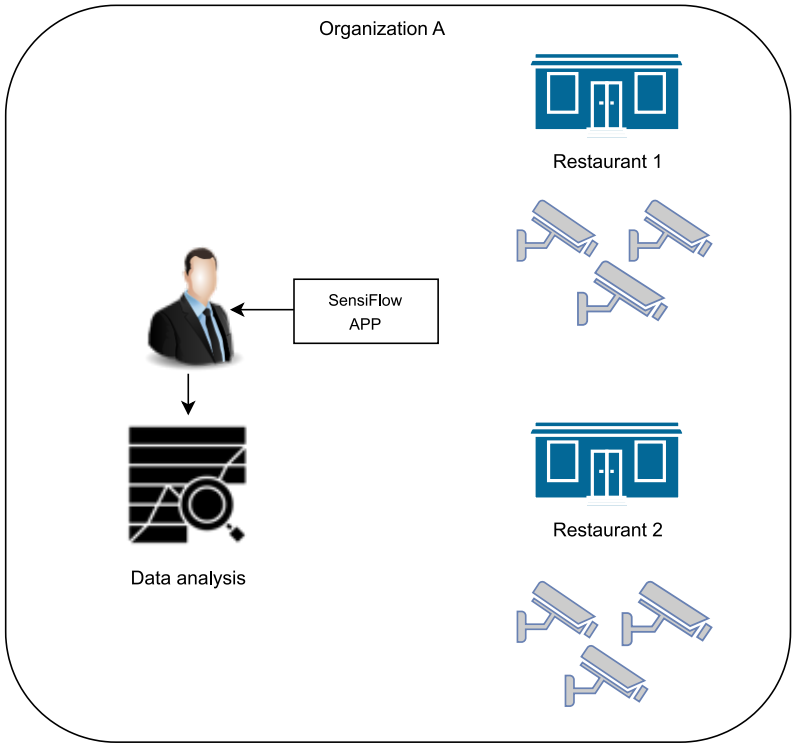

Figure 1.1: Use case for the SensiFlow App

Figure 1.1 illustrates a potential use case for an organization responsible for managing restaurants. Since it is common for business establishments to have surveillance cameras for safety purposes, we can leverage these cameras as a valuable source of information for our application. By analyzing patterns in occupancy rates and customer movement, restaurants can gain insights into peak hours, popular seating areas, and high traffic zones.

This valuable information can then be used to make informed decisions regarding staffing, table arrangements, and optimizing the restaurant layout to enhance customer flow.

The app uses a client-server architecture, in which the frontend is a web application, and the backend is comprised of 3 components, a Web API, a Media Server and an Instance Manager. These will be explained later, on the next chapter.

The remainder of the report is organized into seven chapters. Chapter 2 presents the Problem Analysis and outlines the requirements for the application. Chapter 3 details the Approach to solving the problem, together with the architecture used. In Chapter 4, the development of the API is described, followed by Chapter 5, which focuses on the development of the Instance Manager. Chapter 6 delves into the development of the client Web Application. In Chapter 7 it's discussed some features involving all the modules and some overall evaluations. Finally, Chapter 8 concludes the report and discusses potential avenues for future work.

Chapter 2

Problem Analysis

Contents

2.1 State of the art

Object and person detection technologies have made significant advancements in recent years, enabling various applications across industries. Using sophisticated algorithms and deep learning techniques, these technologies can accurately identify and classify objects and individuals in real-time. In the retail and market sector, person detection plays a crucial role in enhancing customer experience and security. For instance, smart surveillance systems equipped with person detection algorithms can monitor foot traffic and customer behavior, enabling store owners to optimize store layouts and product placements.

2.1.1 Detection Algorithm

At the initial stage of the project we planned to work on our own object detection model using the Haar Cascade Classifier Team (2023). We started by training a cascade function by feeding it positive and negative images with numerous people and different locations from a data set with over 2000 images, however we noticed this algorithm was very prone to false positives. As we further defined the project's focus, we realized that the available time was insufficient to achieve the desired performance for our model. As a result, we explored existing algorithms as potential tools to support the main idea of the project without much focus on the algorithm implementation.

The *You Only Look Once* (YOLO) Jocher (2020) algorithm has emerged as one of the best object detection algorithms in recent years. Unlike previous approaches that relied on multiple stages, YOLO introduced a single-stage, end-to-end framework, making it faster and more efficient.YOLO's key advantage lies in its ability to simultaneously perform object localization and classification in real-time, enabling real-time video processing at high frame rates.

Compared to previous algorithms such as *Region-based Convolutional Neural Networks* (R-CNN) Girshick, Donahue, Darrell & Malik (2014) and Fast R-CNN Girshick (2015), YOLO is able to offer significant speed improvements without compromising accuracy. By eliminating the need for region proposal networks and subsequent refinement stages, YOLO achieves near real-time performance, making it ideal for applications that require fast and accurate object detection. Redmon & Farhadi (2018)

Considering these advantages, we have chosen to utilize the YOLO algorithm for person detection in our project. Its real-time processing capabilities and accurate detection performance are well-suited for our application requirements, enabling us to efficiently detect and track individuals in various scenarios.

However there are multiple versions of the YOLO algorithm, we have decided to use

the $yolov5$ due to the fact that it was more lightweight than some newer versions tested, such as the $yolov7$, and it is not an embryonic version such as the new v8 version.

2.2 Requirements

In order for the application to achieve the purpose stated in the state-of-the-art section using the detection algorithm described above, the system has to support a set of requirements, functional and non-functional, that match the client's needs to perform data analysis effortlessly, efficiently and safely.

2.2.1 Mandatory Functional requirements

The mandatory functional requirements are as follows:

- REST API
- Dashboard for data visualization.
- People detection and count from a given camera stream.
- Role based permissions.
- Transmit a stream with bounding boxes applied to detected people.
- Receive metrics associated to a given device.
- Possibility for a user to manage devices and device groups.

2.2.2 Mandatory Non functional requirements

These requirements improve the project quality and are as follows:

- Code should be modular.
- *Scalability* of the system.
- Secure transmission of stream with detected people.
- Secure communication between the API and clients.
- The user interface should be simple and intuitive

2.2.3 Optional Requirements

The optional requirements are as follows:

- Deployment of the application.
- Ability to configure zones in a space (restricted or unrestricted).
- Limit number of persons in a space and warn on violation.
- Email verification on password update and registration.
- Allow the admin to define groups of users who have restricted access to a limited group of cameras.
- Allow a worker to be configured (resources used, other classes to detect, the ability to choose if you want to stream back the detection or not).

2.3 Related work

In our exploration of the computer vision market and similar applications, we came across Viso.ai Viso.Ai (2023), a low-code platform for computer vision development. However, we found that user friendly applications in the field of computer vision are still lacking. In this section, we will discuss the unique differentiating factors between Sensiflow and Viso.ai, highlighting how Sensiflow stands out in managing device detection without the need for coding or device management capabilities. Viso.ai is a platform that offers low-code solutions for developing computer vision applications. It provides a streamlined approach to building computer vision models and applications, reducing the complexity and coding requirements traditionally associated with such development. While similar to our application in its focus on computer vision, there are distinct differences between Viso.ai and our application, Sensiflow.

Viso.ai provides a generic approach that allows users to configure and build computer vision applications according to their specific requirements. In contrast, Sensiflow takes a different approach by focusing on reaching organizations of any size, including small businesses, and ensuring accessibility for users without computer science expertise.

Unlike Viso.ai, Sensiflow is a no-code application that specifically focuses on managing device detection rather than managing the devices themselves. Sensiflow assumes that the company already has a camera surveillance system in place and provides a user-friendly interface to easily set up and manage the detection of individuals. Sensiflow

simplifies the process by eliminating the need for coding or complex configuration, making it accessible for users without extensive technical knowledge.

In summary, while Viso.ai offers a low-code platform for computer vision application development, Sensiflow differentiates itself as a no-code solution focused on managing device detection specifically, assuming the presence of an existing camera surveillance system.

Chapter 3

Approach

Contents

3.1 Architecture

Figure 3.1 represents the system's architecture which is composed of 3 main modules: The Web Client, Web API and Instance Manager.

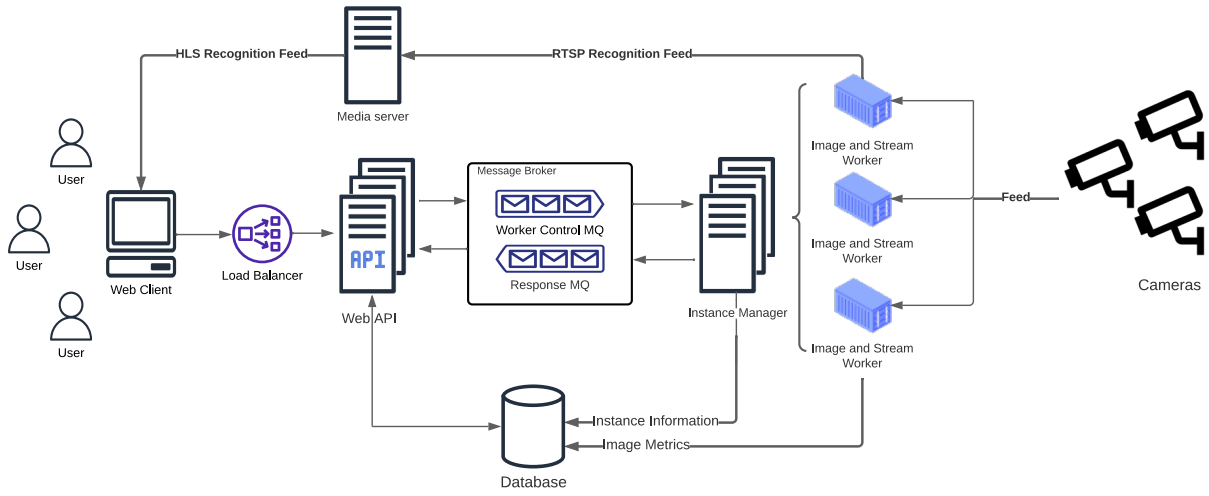

Figure 3.1: Proposed Architecture

Architecture Applied To Use case

Given the use case illustrated in the figure 1.1 which represents an organization responsible for managing restaurants. By applying the proposed architecture to the problem, the outcome will be an organization that has a Web API, a Media Server, a Database and multiple Instance Managers. assuming that each restaurant has a local *Real Time Streaming Protocol* (RTSP) server that the cameras are streaming to, these can be registered by a worker as devices. By having this setup, the organization workers may perform analysis on the received data.

3.1.1 Architectural Justification

In this section we dive into the thought process behind our chosen software architecture, we explain the strategic decisions and key benefits that drove its selection, aligning our development approach with the project goals and requirements.

Scalability

In software systems, horizontal scalability refers to the ability of a system to handle increasing loads by adding more hardware resources, such as servers, rather than increasing the power of existing hardware, this method is preferred since it is more expensive to perform vertical scaling. Since the detection process consumes a great deal of computer resources, the system architecture was designed with this issue into account hence scalability became a defining factor on our architecture. It is important to keep in mind that the instance manager cannot be scaled on the same machine as it will utilize the same computer resources, including the same docker daemon. To make this system scalable horizontally, several factors were considered:

- **Load Balancing** - Implementing a load balancer that distributes incoming requests across multiple servers, ensuring that the workload is evenly distributed. This allows the system to handle more concurrent users and facilitates the process of adding additional servers as the client communicates with the load balancer and does not know the amount of existing servers.
- **Stateless Architecture** - Designing the system with a stateless architecture ensures that no client session data is stored on the server. Instead, all the necessary information to process a request is contained within the request itself. This allows any server in the system to handle a request, enabling easy scaling by adding or removing servers.

In order to achieve horizontal scalability in our system, we have prioritized these factors. Firstly, all of our main components, such as the WebAPI and Instance Manager, are designed to be "stateless." This means that they do not maintain any application domain state specific to a single server. Instead, all the state information is stored in an external database, ensuring that servers do not need to store any data locally.

To facilitate load balancing, we have implemented a message queue that acts as a load balancer for the instance managers in our system. This allows for effective distribution of workload across multiple instances, ensuring that the system can handle increased traffic and demands without overwhelming any single component. To perform load balacing for the API instances **NGINX** was used, the clients communicate with the load balancer interface which handles the distribution of each connection to the available API instances, it is important to note that to serve the static files obtained from the web client **NGINX** was also used, another option was to serve the static files using Spring. However this was discarded since every instance of the API would have duplicate static files, and this would result in a waste of space.

By adopting these measures, we have provided to possibility to achieve horizontal scalability for the WebAPI and Instance Manager, enabling our system to efficiently scale and handle increased loads by distributing the workload. The other components, such as the Media server and Database must be vertically scaled as it is harder to perform horizontal scalability on this components, since each instance will have different data.

However to scale the Media server horizontaly it is possible to create a module with multiple media server instances, this module would acts as an interface and would use its media servers APIs to redirect the incoming requests to the correct inner server.

Media server

This architecture uses a media server to support streaming of media content enabling the Web Client to receive feed and the Image and Stream Workers to send feed. However this media server is reserved for our application necessities, hence it is expected that the cameras are already streaming their video feed to an external RTSP server. Having this in mind, in order to get the stream from the respective server it is necessary that the client provides the respective *Uniform Resource Locator* (URL) that the camera is publishing to. This has originated the need to provide an interface that allows the communication between the backend and client.

Web Client

To solve the need of an user interface, a Web client application was created, which gives the user a friendly interface to interact with. The chosen client architecture is a *Single Page Application* (SPA) with *Client Side Rendering* (CSR). This approach was chosen since it provides a faster user navigation and experience once the app is loaded. This component communicates with the Web API that provides static assets and a RESTful API.

Web API

The Web API's purpose is to handle requests from the Web client and send back responses. It also acts as an intermediary between the rest of the system and the Web Client. It provides authorization and authentication ensuring that only authorized users can access the system and perform actions based on their roles and permissions. To communicate with the Instance manager it sends messages with an appropriate schema to the Message Queue.

Instance Manager

On our initial attempt of developing this module we considered integrating it directly into the API. However this would restrict the possible languages of the API to python as the subsection 3.2.4 states, which may not be a problem if that change brought us benefits. Combining them would mix their domains and purposes, requiring separate scaling strategies. Dividing the modules provides the advantage of independent scaling for each component, allowing them to grow and adapt separately. It also enables the independent management of each component, facilitating better control and flexibility in addressing specific needs and requirements.

Due to the nature of the problem that this architecture tries to solve, which involves the need of person detection using *Artificial Intelligence* (AI), a great deal of processing power is required, preferably by a *Graphics Processing Unit* (GPU) as these are optimized to process many pieces of data simultaneously. Taking this into consideration, it is important that the module responsible for the detection is easily *scalable*.

To address this requirement, the load balancing functionality of a message queue is used to distribute the workload across one or more *Instance Manager* components, each responsible for managing their respective workers. An instance manager can have as much workers as long as the system has enough resources. The person detection processing that happens inside a Stream Worker is a containerized process.

Communication channel between instance manager and Web API

To perform data exchange between the instance manager and the Web API, and considering the need of a full-duplex communication we considered four potential options, which were:

- *Inter-Process communication* (IPC).
- API on Instance Manager.
- Asynchronous Message queue.
- *Google Remote Procedure Call* (gRPC).

Among the four potential options considered for data exchange between the *Instance Manager* and the web API, the asynchronous message queue emerges as the most suitable choice. Despite the other options (IPC, API on the instance manager and Remote procedure call) are suitable ways of communicating, they do not provide the same level of flexibility and efficiency as an asynchronous message queue. IPC, although capable of

facilitating data exchange between processes, may introduce complexity and dependencies, making it harder to maintain and scale the system. Additionally, IPC often requires tight coupling between components, which can inhibit flexibility and modularity. Furthermore, it is important to note that both components must be operating on the same machine, which significantly restricts our ability to achieve horizontal *scalability*. This is a limitation that we aim to avoid.

An API on the instance manager could allow for bidirectional communication but may introduce unnecessary overhead and latency. It would require the instance manager to constantly listen for requests from the Web Api, resulting in a less efficient use of resources. Another disadvantage is that this approach does not promote *scalability* as it is necessary to have an extra component, a load balancer, since each instance of the instance manager will have a different IP address or port.

While gRPC is a valid option as a communication channel, it may introduce unnecessary complexity for the current requirements. It is a powerful framework suitable for efficient communication between services but may not be the most straightforward choice when it comes to decoupling and asynchronous communication.

On the other hand, an asynchronous message queue provides a decoupled and scalable solution for full-duplex communication. It allows the Web API to send messages to the instance manager without requiring it to actively listen for incoming requests. This decoupling enhances maintainability and allows for easy integration of additional components or services in the future. The asynchronous nature of the message queue ensures efficient utilization of system resources and minimizes latency, as messages can be processed independently and asynchronously by the instance manager.

Furthermore, the use of a message queue facilitates horizontal *scalability*. Multiple instance managers can be added to handle the processing load efficiently. The message queue acts as a buffer, distributing the incoming requests among the available instance managers, ensuring optimal utilization of resources and improving overall system performance. It is also important to note that this is a critical component and messages cannot be lost when it goes down, this can be achieved with rabbitMQ as it enables the persistence of the messages to the disk.

3.2 Technologies and tools

Given the requirements above, the following paragraphs elaborate on what we are using, why and what were the considered alternatives.

3.2.1 Web API

The Web API was built using the Spring framework with Kotlin. We chose **Kotlin** as the language for the *Application Programming Interface* (API) as it is the language we had the most experience with. Spring was chosen as the framework for the API since we had previous experience with it, and provides a lot of useful features and support for various modules. Another reason is the fact that can also help speed up development and reduce the amount of boilerplate code we have to write.

The main alternative to it was using Javascript with Node.js and Express framework for the API. However, we decided against it because Kotlin with Spring would easily allow us to create a REST API with a lot less boilerplate code than Node.js and Express, and we would like to make use of spring features regarding authentication and authorization.

3.2.2 Database

The first step in choosing a database was deciding between a document based database or a relational database. We determined that a relational database would be the appropriate choice since our data nature is more relational than document based, because it has a fixed and well defined structure. However, due to the large volume of metrics being saved in the database within a short period of time, this results in a significant number of total entries. PostgreSQL was chosen as this was the database we had most experience with, and we did not consider the importance of the large volume of metrics at the time. However, later we recognized that to improve this issue we could store the metrics in a non-relational database to enable an easier horizontal scalability.
Given this, to improve the database module in the future, we are planning to use a non-relational database for the metrics and a relational database for the rest of the data as it has a well defined structure.

3.2.3 Communication with database

Java Persistence API (JPA) offers a robust *Object-Relational Mapping* (ORM) capability that maps Java/Kotlin objects to database tables. This feature allowed us to concentrate on the core business logic of our application without concerning ourselves with the intricate details of database implementation.

We also evaluated an alternative, called *Java Database Interface* (JDBI), which is a lighter framework compared to JPA and offers more control over *Structured Query Language* (SQL) queries. However, we made the decision to forego JDBI because the ORM functionality provided by JPA would be more advantageous for our application.

Together with the Spring framework, JPA provides a powerful abstraction layer over the database. Additionally, JPA simplifies the execution of *Create, Read, Update and Delete* (CRUD) operations on the database, eliminating the need to write manual SQL queries and reducing the amount of repetitive code.

3.2.4 Instance Manager

While choosing a language for the Instance Manager, a crucial requirement was having support for an API to communicate with the Docker daemon since this was the containerization approach chosen, enabling us to perform operations on containers and images.

After careful consideration, we decided to go with **Python**. Our preference was to choose either Python or Go as these are officially supported languages for Docker-SDK (2023b), while the others would require us to rely on unofficial libraries. Given these 2 options the main factor behind this decision was that we had no previous experience with Go.

3.2.5 Message broker

Considering the choice of a message queue and the already chosen languages and framework for the communication member modules the chosen service was **RabbitMQ**, since the RabbitMQ team offers official support to Spring and to a Python library. The alternatives considered were Apache Kafka and ActiveMQ. We decided to discard the Kafka library since it is more directed to handling large amounts of data and has no official support for python. ActiveMQ was also discarded because there is better documentation and support for spring with RabbitMQ rather than ActiveMQ.

3.2.6 Web Client

We chose React with TypeScript for our project based on several criteria. Firstly, we felt comfortable using these tools, which is crucial for productivity and ease of development. Secondly, we wanted to utilize modern technologies that are widely adopted in the industry for building applications. Lastly, we sought tools that enable us to work efficiently and a maintain a clean and understandable code structure. React, coupled with TypeScript, offered the perfect balance, allowing us to leverage an ecosystem of pre-built components through libraries like MUI, saving time and effort. The combination of React and TypeScript offers strong type checking, enhanced code readability, and improved development workflow, making it an ideal choice for our project.

3.2.7 Media Server

The Media server used to support our app is an open source project "MediaMTX" from Bluenviron (2023), a “ready-to-use and zero-dependency server and proxy that allows users to publish, read and proxy live video and audio streams.” This specific server was chosen, since it supports various streaming protocols and also the transcoding between them. This means that it is possible to publish a RTSP stream and read it as an *HTTP Live Streaming* (HLS) stream. The supported protocols are available in its github page. Another reason for this choice is that it offers an extensive amount of possible configurations and a good documentation.

As the Media server will be deployed on a network, it is crucial to authenticate the readers and publishers to ensure data security.

To achieve this, the MediaMTX server allows the following authentication methods:

- Fixed User and Password Authentication.
- External authentication via an external server.

The Fixed User and Password allows publishers and readers to authenticate via predefined credentials. This provides simplicity but some limitations as we won’t be able to fine grain the data. This option is optimal for small scale deployments where the publishers and readers have similar access requirements.
The External authentication allows publishers and readers to authenticate via an external server. This involves sending a request to the MediaMTX server with the credentials, and the MediaMTX server will then send a request to a configured URL with a body containing the credentials and what data is being requested. The external server will then respond with a status code indicating if the access is granted or not. This option provides more flexibility by allowing integration with existing authentication systems, and also provides an enhanced control of the access requirements, as the external server can decide what data is accessible to the publisher or reader. In our case, if we want to allow users to use our media server as a publish destination for their cameras we would choose to use the External Authentication method, but since we will not do that, we chose to use a Fixed User And Password Authentication. However by choosing this method we are aware of its vulnerabilities such as the fact that it alone does not provide confidentiality, since the credentials are sent as clear text, the request may be intercepted and the credentials are leaked, given this, it is necessary that the media server uses https to solve this issue. However since the credentials do not change for each user, if any user reads the request

and save the credentials he will always have access to the media server, even if the user gets deleted from the system. We are aware of this vulnerability and a change to external authentication will be made in the future.

3.3 Data Model

Figure 3.2 illustrates the chosen data model.

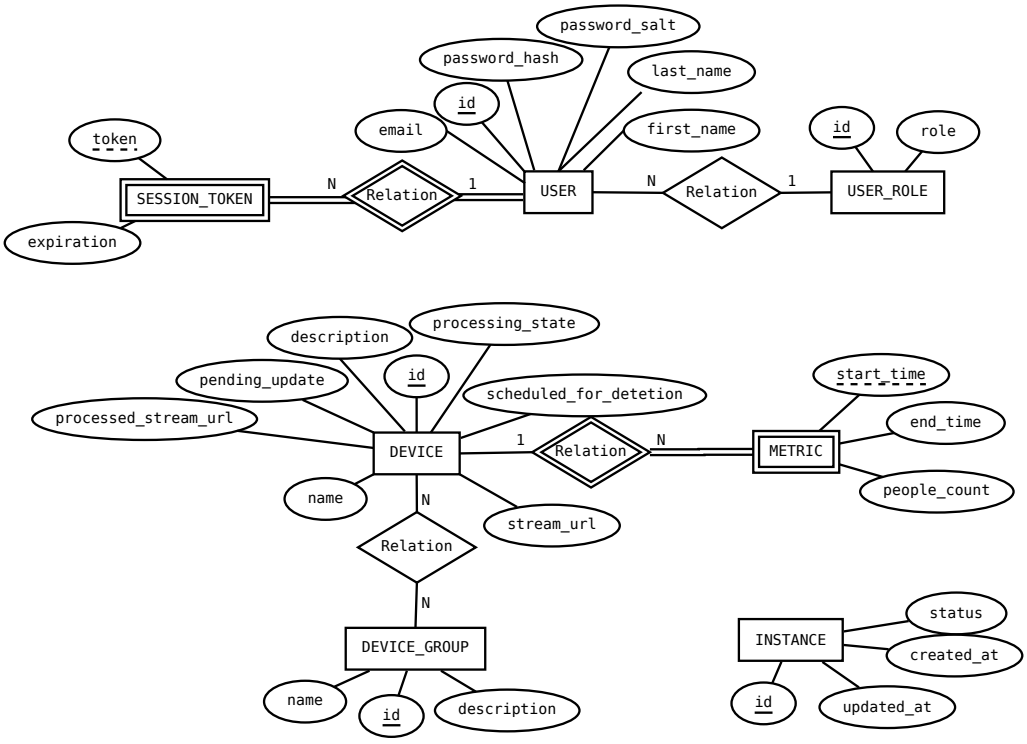

Figure 3.2: Relational Model

The entities and relationships are explained in the sequel.

- User – Represents an app user and contains an id, last name, first name, an email, a password hash and password salt to have extra security on storing a representation of the password.
- User_Role – Represents the role of the user. Since it is possible to add new roles, this identity can't be part of the User. User Role contains an id and the role.
- Session_Token – Represents the time that the user account is going to be active without needing to log in again. Session Token contains an expiration and the token,

it relation to the user is N to 1 to support a user having multiple session tokens.

- Device – Represents a camera and contains a name, description, the processing state and the pending update. These last two are related to because the pending update changes when the user changes the processing state of the device. Furthermore, the Device also has an url for the input stream, another url for the processed stream and a schedule for deletion flag.
- Metric – Represents the number of people in a time period. This contains the people count, time start and end time.
- Device_Group – Represents a group of cameras and contains the name of the group, an id and a description.
- Instance – Represents an image processor instance.

Chapter 4

Web API Module

Contents

This chapter provides a description of the Web API, one of the main modules of our project. All the actions on domain entities are performed through HTTP API requests.

External Modules

The external modules used are:

- *Spring* spr (2023) - Framework the API used.
- *Junit* jun (2023) - Framework to perform the tests.
- *Jackson* jac (2023) - Module that adds support for serialization/deserialization of Kotlin classes and data classes.
- *Klint* ktl (2023) - Linter for maintaining code quality.
- *jpa* jpa (2023) - Data persistence framework .
- *Spring Rabbit* sra (2023) - Spring module to assist with the usage of RabbitMQ.

4.1 Architecture

In Sourcemaking (2023), the authors suggests the following good pratices: "In the context of architecturing a Web API, design patterns can speed up the development process by providing tested and proven development paradigms. Effective software design requires considering issues that may not become visible until later in the implementation." Having that in mind, the Web API was designed using patterns already known, being the Presentation-Domain-Data-Layering and Domain Driven Design Fowler (2020), as well as other concepts like SOLID. The layering allows to separate the code in three topics relatively independently as the following figure 4.1 illustrates. By separating these elements it narrows the scope of thinking in each piece, which makes it easier to follow what it is needed to do.

In order to assure we have a backwards compatibility, we made sure to have API versioning. This grants independency between the versions as it provides the possibility to update an API version and make breaking changes while having multiple stable API versions.

The main traits of the chosen architecture are:

- Testability.
- Independent of the UI.

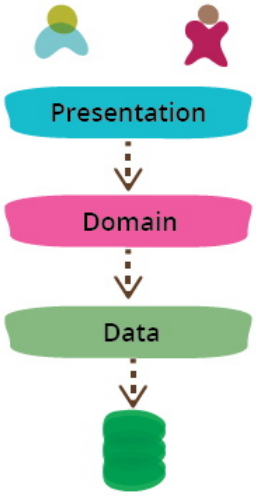

Figure 4.1: API Layered Structure Fowler (2015)

- Independent of the Database.

Directory structure

The high-level directory structure of our project's source directory. Each layer in the architecture has a dedicated directory housing the corresponding code for that particular layer. Each one explained in their respective section below.

```
amqp ..... Message Broker Communication Layer
http ..... Controller/Presentation Layer
model ..... Data Access Layer
services ..... Domain Logic Layer
```

4.1.1 Non functional queries

To see what endpoints support the following queries, it is possible to check the *Sensiflow*'s documentation.

Pagination Requests that return multiple items are paginated to 10 items by default. It can be requested further pages with the page parameter. It is also possible to set a custom page size up to 50 with the size parameter. The table 4.1 indicates the pagination parameters.

Table 4.1: Pagination Query parameters.

Parameter	Type	Default	Max
page	number	1	—
size	number	10	50

Expanding Many endpoints allow the request of additional information as an expanded response by using the expanded request parameter, additional information on the expanded parameter can be seen in the table 4.2. This enables the possibility to get more information from the same endpoint without having to make additional requests.

Table 4.2: Expanded Query parameter.

Parameter	Type	Required	Default
expanded	boolean	No	false

4.2 Controller layer

The API, as mentioned earlier, uses spring as its framework. HTTP requests associated endpoints to certain triggers the execution of handlers available on the controllers. Each handler proceeds to call the correspondent operation in the service layer, these services are supplied by the dependency injection engine that Spring provides.These services use repositories to manipulate data.
This layer is where the responses and requests are received and sent to the client as output entities. In case of any error, a Problem Detail representation is sent. This representation is described in the Akamai & Wilde (2016) Problem Detail RFC.

Input verification Every entity representing user input, from the request's body, is verified using the $@Valid$ annotation on the respective controller. This annotation marks a method parameter for validation cascading. Each user input entity has constraints defined on its properties and those are validated. If a constraint is broken an exception is thrown.

4.2.1 Authentication

The API uses cookie-based session authentication, which means that the user maintains a session with the API, and the API uses the session to identify the user. The client uses a cookie to maintain the session information with the API. The session is created when the user logs in, and is destroyed when the user logs out. A user can only have one session at a time, which means that if the user logs in again, the previous session is destroyed.

The authentication is supported by Cookies because it can prevent *Cross-site scripting* (XSS) attacks. This is done by setting the "HttpOnly" flag to the cookie, meaning that it can only be accessed by the API, and not by client side scripts. It also prevents *Cross-site request forgery* (CSRF) attacks. This is done by setting the SameSite=Strict flag to the cookie so it can only be sent to the sites into the same domain.

For safety measures, the password is not stored as clear text in the database, instead it is concatenated with a salt which is a randomly generated string, and then it is hashed and stored in the database together with the salt. In the login process, to verify if the user's password is the same as the one received, the same process is applied and the result is compared with the one stored in the database. This procedure is used to raise the resistance to dictionary attacks and rainbow table attacks.This process is illustrated in Figure 4.2. In the authentication process there is a vulnerability if the API is using http, as if someone is listening the network it can obtain the credentials on the login / register request, this issue can be fixed by using HTTPS, or by estabilishing a protocol to encode

the client credentials

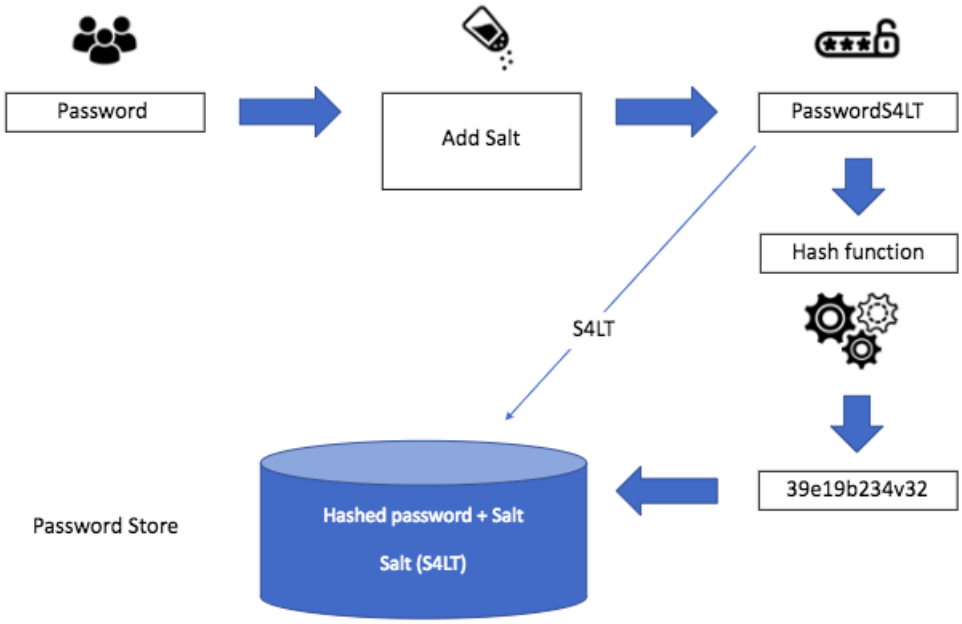

Figure 4.2: Password storage process. Jung (2021)

4.2.2 Authorization

A *Role Based Access Control* (RBAC) is implemented, limiting the access to the API to only authorized users. The following roles are available:

- Admin.
- Moderator.
- User.

The Admin role is the highest role. An Admin user is created by default when the API is deployed. This is due to the fact that to create a new user it is required a role with enough privileges considering that this project is meant for organizations. Having the following login credentials:

```
Email: admin@gmail.com
Password: Admin123.
```

These credentials can then be changed in the web interface or through API requests. Having in mind that the project was built with the purpose of being used individually by organizations, the Admin, being the first and only user, can then create new users and assign them roles. A new user cannot register himself/herself, the registration is made by an Admin or Moderator.

4.2.3 Spring pipeline

By using Spring, we have access to a pipeline that allows the interception of data before or after reaching the handlers and the filtering of information. In this context, the next interceptors, argument resolvers were added as it is shown in Figure 4.3:

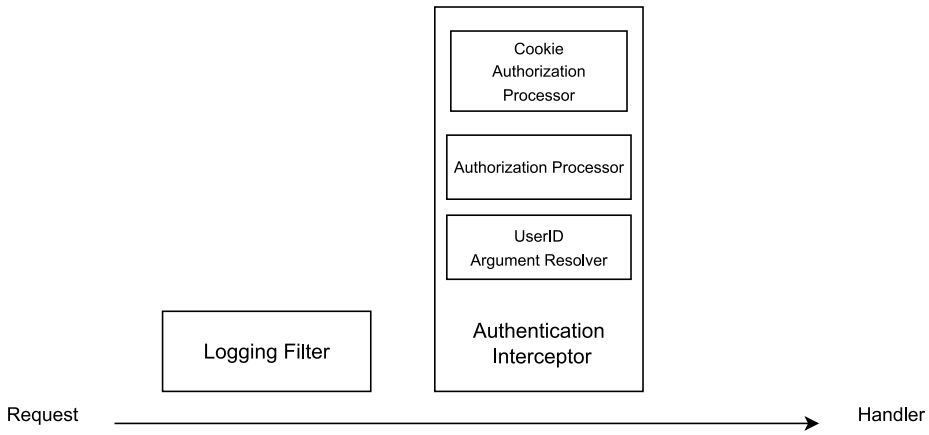

Figure 4.3: Spring Request Pipeline

Info Filter

This filter logs information about the requests and responses. This information includes:

- Time to process the request;
- The amount of cookies;
- The endpoint and method where the request was made to;
- The query parameters;
- The request body.

It is important to notice that to prevent the logging of critical information such as passwords, the request body is not logged for any request dedicated to authentication containing user sensitive information.

Authentication Interceptor

Handles authorization and authentication for endpoints. This is done with the $@Authentication$ annotation. This annotation is used in any handler that requires Authentication, if the id of the authorized user is needed it can be supplied on the handler with an argument of the type "UserID" and named "userID".

To verify if a user has authentication, an interceptor was created. The *Authentication Interceptor* which is responsible for intercepting the request and with the help the *CookieAuthenticationProcessor* class, checks if the given information on the cookie is valid, if it is not, an Unauthenticated exception will be thrown. If it is valid the user id will be added to the request attributes allowing it to be supplied later on and finally the execution chain proceed. This interceptor also verifies authorization after the user authenticity is verified. This process is confirmed by the *AuthorizationProcessor* class which compares if the user role is higher than the role received in the annotation.

To provide the user id to the specific handler argument we created the class *UserIDArgumentResolver* extending Spring's *HandlerMethodArgumentResolver*. This resolver verifies if a handler has the required conditions to receive the necessary data and injects it.

Error Handler

This class handles the errors that occur. This is done by having a Controller class dedicated for errors using Spring annotations and overriding Spring error handlers to catch the exceptions that are thrown while processing the outcome of a request. In order to catch Spring-related exceptions the following options were added to the $application.properties$ file:

```
spring.mvc.throw-exception-if-no-handler-found=true
```

To associate our app exceptions with the correct HTTP status code, they are mapped when building the error response. This error response uses Problem detail to inform the user of the occurred an error. To do this Spring provides an implementation of a Problem Detail class. When an error occurs in the app and its handled in the Error Handler Class, the response sent to the client is a problem detail class. This Problem Detail class used is provided by spring when the following option is added to the $application.properties$ file:

```
spring.mvc.problemdetails.enabled=true
```

To be able to catch the exception thrown when a request was made to an non existing endpoint the following option was necessary:

```
spring.web.resources.add-mappings=false
```

However this option disabled the possibility to serve static content on the API, but since the static content will be served with the assist of NGINX, it was not a problem to use these options. However if the serving of static content would take place on the API, a replacement for this option would be possible, such as adding a fallback endpoint for every request beginning with "/api/v1/".

4.2.4 Server-sent events

Since changing a device processing state is an asynchronous call more information on these types of calls can be obtained in Section 7.1, when changing it to a pending state, the client needs to know when the change happens. For that reason, and since it was needed an unidirectional communication between the API and the client, we ruled out Web Sockets for being more resource intensive and bidirectional. Another option to implement was Long Polling but it has a massive network overhead as it involves frequent request-response cycles. For those reasons, *Server-sent events* (SSE) were chosen. Whenever a change of processing state is requested, a subscription to the change is emitted on a different coroutine and a Server-Side Event emitter is returned. The coroutine will be up until it is canceled or the job is completed, this way it is not necessary a reference to the $emmiter$ as it is mantained by the use of the coroutine. Furthermore there are mechanisms to ensure the $emmiter$ is closed when an error or timeout occurs. It is important to notice that the server sent events used are auto cancellable, this means that when a certain condition is met, the server sent event closes itself, for example, the sse to listen for a change of processing state will notify the client and be closed when that change occurs. This process is used to prevent problems resulting from the need of unsubscribing an event, which can be problematic when using a load balancer for multiple instances.

Another instance where SSE are used is to obtain the number o people in real time.

4.2.5 Thoughts and Reflections on the Controller

During the design of the controller's endpoints, a doubt was brought up relating to the endpoint to get the *DeviceGroup*'s information. In a RESTful API, it is generally a good practice to separate related resources into distinct endpoints whenever it makes sense. In the case of a *DeviceGroup* and its associated *Devices*, it could make sense to have a sepa-

rate endpoint for retrieving the *Devices* of a given *DeviceGroup*, like $/groups/id/devices$ and another to get the group information.

This approach allows for a more fine-grained control over the resources being retrieved, and can simplify the representation of each resource. It also aligns with the principle of "single responsibility", which suggests that each endpoint should represent a single resource or action.

Having a separate $/groups/id/devices$ endpoint also enables support for additional features, such as filtering, sorting, or pagination, that may be useful when working with large collections of *Devices*.

However, since retrieving the *Devices* of a *DeviceGroup* is a common operation, if the group has few *Devices*, it could also be reasonable to include them in the response body of the $/groups/id$ endpoint. This approach can reduce the number of requests needed to retrieve all the relevant data and simplify the client's logic.

Considering that the API includes operations for adding and removing *Devices* from a group, opting for a single endpoint means that the mentioned operations happen on group update. However, this approach would require fetching all the devices from the requested group when performing a PUT request, which disregards the benefits of pagination. To address this issue, two possible endpoints were made:

- /groups/id
- /groups/id/devices

By separating into endpoints paths, it is no longer necessary to fetch all the devices on update. Instead, $POST$ and $DELETE$ requests are used to update the group's device list.

In the current version, the API has the endpoints presented in the Figure 4.4.

4.3 Service Layer

The services contain all the business logic of the API, including the data transaction management and the process of verification of the necessary requirements. When an operation requires data it can be provided through the *Data Access Layer* (DAL) which abstracts the process to obtain data by providing a repository responsible for the interaction with the data.

AUTHENTICATION		
Logs the user in	**POST**	/users/login
Logs the user out	**POST**	/users/logout
USERS		
Create a new user	**POST**	/users
Get an user	**GET**	/users/{id}
Get all users	**GET**	/users
Edit an user	**PUT**	/users/{id}
Delete an user	**DELETE**	/users/{id}
Update an user role	**PUT**	/users/{id}/role
DEVICES		
Create a new device	**POST**	/devices
Get a device	**GET**	/devices/{id}
Get all devices	**GET**	/devices
Edit a device	**PUT**	/devices/{id}
Delete devices	**DELETE**	/devices
Update the device's processing state	**PUT**	/devices/{id}/processing-state
Get device's stats	**GET**	/devices/{id}/stats
Get devices's processed stream	**GET**	/devices/{id}/processed-stream
GROUPS		
Create a new group	**POST**	/groups
Get a group	**GET**	/groups/{id}
Get all groups	**GET**	/groups
Update a group	**PUT**	/groups/{id}
Delete a group	**DELETE**	/groups/{id}
Get group's devices	**GET**	/groups/{id}/devices
Add devices to group	**POST**	/groups/{id}/devices
Remove devices from group	**DELETE**	/groups/{id}/devices
SERVER EVENTS		
Processing state	/devices/{id}/server-events/processing-state	
People count	/devices/{id}/server-events/people-count	

Figure 4.4: API endpoints available

4.3.1 Transaction Management

The service layer may call different repositories to perform database operations. In situations where the transaction management is made in the data layer and there are multiple entities operations in a service method, if the first operation on a repository failed, other two may still pass and the database state will be inconsistent. In order to avoid that, the transactional management is made in the service layer with the annotation provided by Spring $@Transactional$. This annotation supports further configuration:

- Isolation Level.
- readOnly flag.
- Rollback rules.

Every service method has an Isolation Level. Methods with only read accesses have an Isolation Level of Read Committed ensuring that only data that is already commited is

read, meanwhile read/write accesses have an Isolation Level of Repeatable Read ensuring that the transaction sees a consistent snapshot of the data, preventing other transactions from modifying the read data until the transaction completes. This aims to reduce performance degradation and improve the system scalability.

For the read only methods, the *readOnly* flag is also used, allowing for corresponding optimization at runtime. Lastly, for methods where an exception is necessary to be propagated and not rollback, it is used a *noRollbackFor* rule that it is associated to the exception.

4.4 Data Layer

Data access By using *Java Persistence API* (JPA) and its annotations we can map entities to its respective database table. This process is called *Object-Relational Mapping* (ORM), and provides an abstraction of the details of communication to the data source, such as not needing to deal directly with SQL queries. After the mapping process is done, it is possible to create repositories for each entity. These are created by extending from Spring's *JpaRepository<T,ID>* interface, which requires two arguments, *T* the type of the entity to manage and *ID* representing the type used to identify the entity. This provides *Create, Read, Update and Delete* (CRUD) operations to manipulate the T entity and also provides support for pagination. The goal of the repository is to provide an abstraction to significantly reduce the amount of boilerplate code required to access the data on the respective data source.

4.5 Communication with the Queue

With the official rabbit dependency from Spring, this process is eased by Spring's helper classes, requiring only some configuration to be able to establish a successful connection and exchange messages. This interaction is needed for operations requiring actions from the Instance Manager, there are three occasions when this happens: when a user wants to delete a device, when a user changes a device's processing state, or when a device's source streamURL is edited.

The first one is necessary since a worker may still be processing feed for a device therefore it needs to be shut down. For that reason the device is not deleted from the database right away, instead it is set a flag for deletion. The deletion is executed only when the queue returns a message that confirms that the processing has stopped.
The second instance is due to the fact that changing a device's processing state changes

the worker's processing state. As the Instance Manager is updating the worker's state, the device's processing state is pending. When a message with confirmation on the update is returned to the API the device's processing state is finally updated.
The last occurrence is necessary due to the fact that a user may change the device's streamURL while a worker may be processing the stream from the previous streamURL. Given this, when this update happens, a message is sent to the instance manager to stop the processing device and remove the instance to guarantee that the consistency is maintained.

Figure 4.5 illustrates the messages format and the queues available to establish communication.

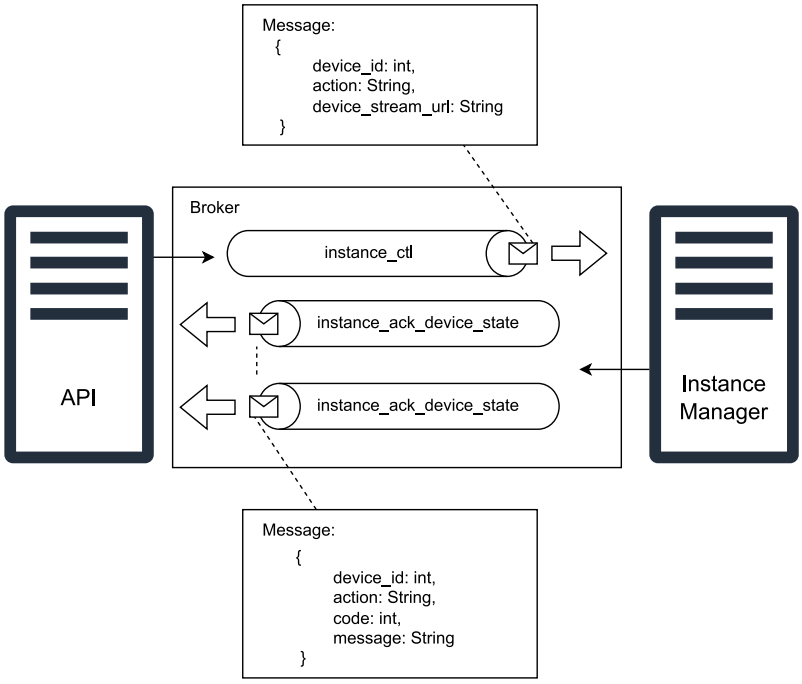

Figure 4.5: Message Channel between API and Instance Manager

Spring Configuration To support communication with RabbitMQ, configuration had to be added to the $application.properties$ file. The configuration includes the host, port, credentials and the name of the queues necessary for the application to work properly.

Rabbit Listener Since the process of receiving a message is asynchronous, the processing of these messages happens inside a class annotated with the $@RabbitListener$ annotation which receives a queue to listen from. The messages exchanged in the queue have a pre-defined format that is verified when a message object is built. There are two types of messages received:

- messages that notify about the update of a device's state;
- messages that notify about inconsistencies.

Both cases have proper handling and keep the device's state updated. The handler contains a dispatcher that allows the addition or removal of actions that can be executed on a message without changing existing code. If a third party sends messages to the queues Spring is listening to, and rabbit goes down, the messages do not reach Spring however they are not lost as they are rabbit has mechanisms to persist the messages that are in a queue. This way, when rabbit comes back up, the messages will be available.

Chapter 5

Instance manager Module

Contents

This chapter provides a description of the Instance manager Module, one of the main modules of our project. This module is responsible for managing the containers processing the video input from the devices.

5.1 Structure

This module is composed of the following components:

- Instance Manager
- Image Processor Worker
- Instance Manager Scheduler

External Modules

The external modules used are:

- *Aio pika* Mosquito (2023) - Package used to communicate asynchronously with the RabbitMQ message broker.
- *docker-py* dev team (2023a) - Module to interact with the Docker daemon.
- *Psycopg3* team (2023) - Module for asynchronous communication with a PostgreSQL database.
- *Flake8* Ziadé, sottile & Cordasco (2023) - Linter for maintaining code quality.
- *tox* dev team (2023d) - Package used to automate and standardize testing in Python.
- *Pytest* dev team (2023b) - Framework for writing tests.
- *Pytest-env* dev team (2023c) - Pytest module for simplifying the definition of environment variables on tests.
- *ffmpeg* ffm (2023) - External program capable of recording, converting and streaming audio and video.

Build tool and dependency manager

To simplify the process of managing project dependencies, creating virtual environments, and building and publishing packages, the tool Poetry poe (2023) is used. Poetry is a

popular dependency manager and build tool for Python projects. This tool has a dependency resolver, intuitive and easy commands to use and it is isolated from the system. The modules above are managed by this dependency manager.

5.1.1 Architecture

To ensure horizontal scalability of the Instance Manager, we have adopted a pull-style architecture as it is illustrated in Figure 5.1. This design choice allows the developer to select and configure the number of messages to receive simultaneously, optimizing resource utilization and performance.

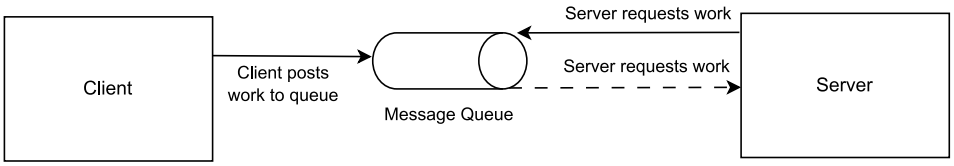

Figure 5.1: Pull architecture

Another choice would have been a push-style architecture as it is illustrated in Figure 5.2. In this architecture the Instance Manager would request work from the API whenever it was available. This option would require to continuously request work from the API, meaning that if the Instance Manager fails to request work from the API, it might miss out on potential tasks.

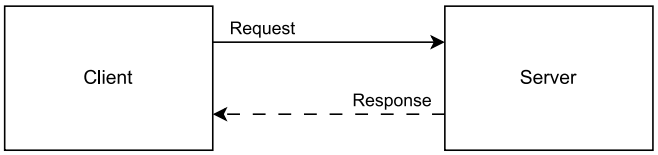

Figure 5.2: Push architecture

Both push-style and pull-style architectures can be designed to scale horizontally. However, a pull-style architecture offers more control over resource allocation Poole (2018).

Directory Structure

The directory structure of this module is as follows:

```
configs
├── docker ..... Worker dockerfiles
├── src ..... source code of the project
│   ├── config ..... Configuration parser and provider
│   ├── database ..... Database connection and transaction management utilities
│   ├── docker_manager ..... Docker Engine project's interface
│   ├── image_processor ..... Worker component
│   ├── instance_manager ..... Instance Manager Component
│   ├── rabbitmq ..... Rabbitmq wrappers and utilities
│   └── exceptions.py ..... Application common exceptions
├── tests
├── run.py ..... Instance Manager entrypoint
├── scheduler.py ..... Scheduler entrypoint
└── transmit.py ..... Image Processor Worker entrypoint
```

All of the components are residing within the same project. These components have been grouped together due to their extensive codebase overlap and shared functionality. By placing them in a single project, we can efficiently manage and maintain the common codebase while leveraging the synergies between these closely related modules. This approach promotes code reuse, simplifies updates, and enhances overall project cohesion.

5.2 Instance Manager

The Instance Manager is responsible for managing the image processor workers. It communicates with the API module to receive commands to start, stop, or pause workers. It is a consumer of the Instance Controller Queue from the message broker to receive device state updates and device delete requests from the API. It also publishes messages to acknowledge device state updates and device delete requests using the *ACK Device State Queue* and *ACK Device Delete Queue* respectively. When starting the instance manager, an image from a Dockerfile, dedicated for either CPU or GPU, is built to be used by the workers, this is done to prevent the overhead otherwise experienced when creating the first *image processor worker*. The configuration dictates whether to use hardware acceleration or not. The model used for the machine learning algorithm is also downloaded in the initialization of the Instance Manager, this process is done so that each *image processor worker* can access it without wasting time downloading it.

5.2.1 Controlling the Instance Manager

Communication with the Control Queue

We tried implementing an efficient and scalable approach in the communication with the control queue, where the control messages are sent. Leveraging the power of the library "Aio Pika", we have developed an *AsyncRabbitMQManager* helper class to establish an asynchronous connection with the message queue. Additionally, we have created an *AsyncRabbitMQClient* that provides an asynchronous interface to consume messages asynchronously, facilitating efficient handling and processing. By implementing these classes, the instance manager is able to process multiple messages at a time, effectively improving overall throughput and responsiveness. This robust communication setup provides the flexibility and control needed to efficiently manage the control messages and integrate them into our system with ease.
The processing of the messages from the Control Queue depends on the type of message sent, as if the message is to "START" a processor, the message is sent in a worker pattern wpa (2023), while if the message is any other, the processing is made in a Fan out pattern, requiring each **Instance Manager** to have a queue associated with it. These two patterns are used due to the fact that there would be routing issues if only the worker pattern would be used, as it is possible to see in the situation of having multiple **Instance Managers**, each one will manage their own image workers, and a control message may end up in the wrong manager that has no access to the desired image worker. This is described in Figure 5.3.

By using both patterns and therefore two queues that each **Instance Manager** gets the messages from, we assure that each message will be handled correctly as it is illustrated in Figure 5.4. It is important to notice that in our case, a worker for the Start Queue is an **Instance Manager** instance. However each **Instance Manager** will still receive messages that are not meant for them to handle and can be addressed by discarding these messages and acknowledging them to make sure they are removed from the queue. An **Instance Manager** can be aware if the message has to be handled by using the docker API and checking whether it has the container with the associated device ID. This is not the best solution as there will be unnecessary computations for this check. Given the necessity to have a queue for each **Instance Manager**, we have to guarantee its creation and removal. Nevertheless this is not the optimal solution as there is still an edge case where an error happens, that is when a container is paused on a given instance manager and a "START" message is sent to start it. However this message will be sent to the queue using Work pattern, in which there is a possiblity for the wrong instance manager to try and start the

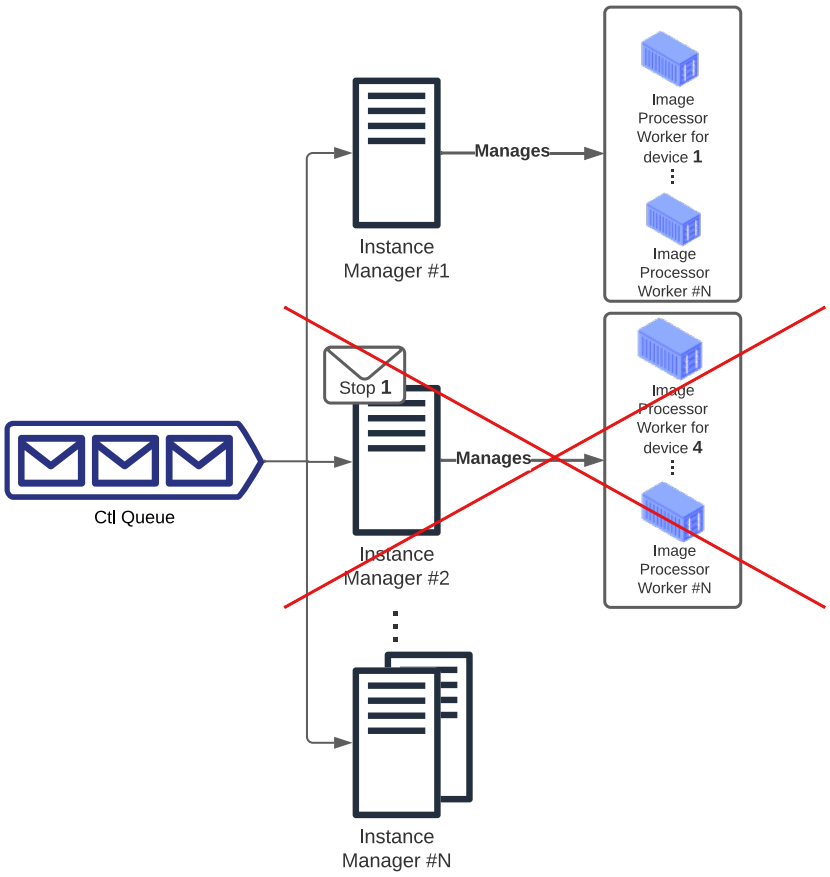

Figure 5.3: Worker Pattern applied to our problem

already existing processor on another instance.
In order to address this issue, the following adjustment was implemented, for every message received from the queue created by the instance manager, the procedure to check if the message is meant for it is still in place. However an additional step has been added where "START" messages are also sent to this queue if the corresponding device is paused. "START" messages when the device status is stopped will still be sent to the worker queue.

Flow of execution

To efficiently handle various actions based on the action type specified in incoming messages, our team has implemented the dispatcher pattern. By employing this pattern, it is possible to seamlessly map different types of actions to their corresponding functions. This allows the execution of specific actions dynamically and effortlessly by simply examining the action field within the message.

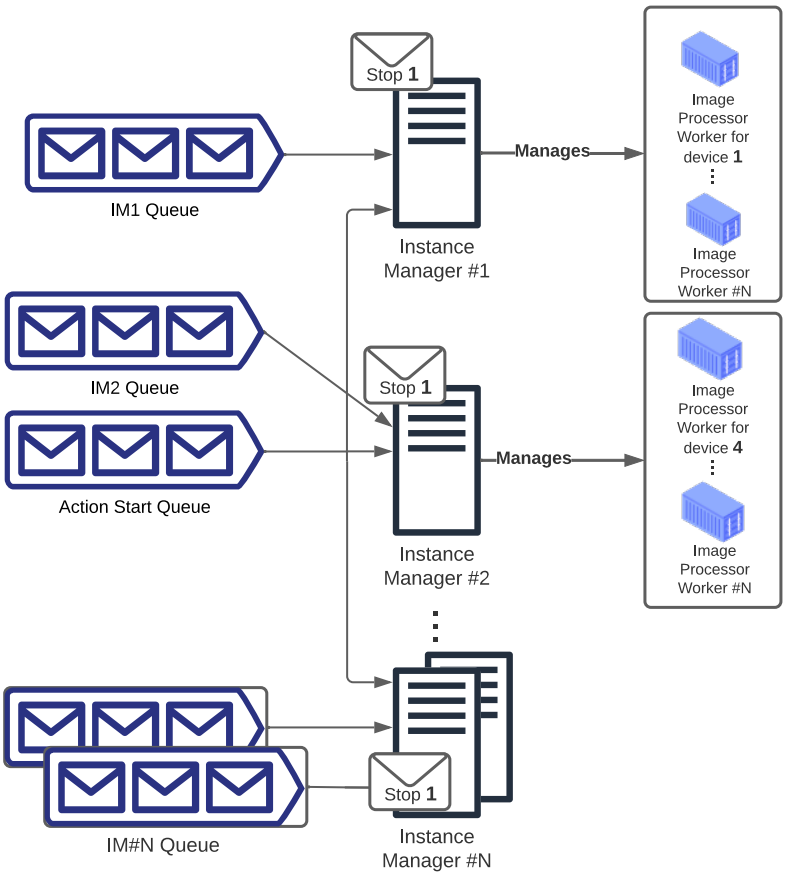

Figure 5.4: Multiple Pattern applied to our problem

The dispatcher pattern in software engineering serves as an action mapper, where specific functions are associated with different types of actions. When a particular type of action is received, the dispatcher identifies the corresponding function and invokes it with the provided arguments. In the context of the given scenario, a message queue is utilized to receive messages in a specific format, such as :

```
{
    "action": "START",
    "device_id": 1
}
```

Diagram 5.5 illustrates the flow of this approach. The process begins with the arrival of a message in the message queue. The consumer (*RabbitMQClient*) pulls the message from the queue, removing it. It then calls the dispatcher handler with the message as an argument. The dispatcher, responsible for handling these messages, extracts the action

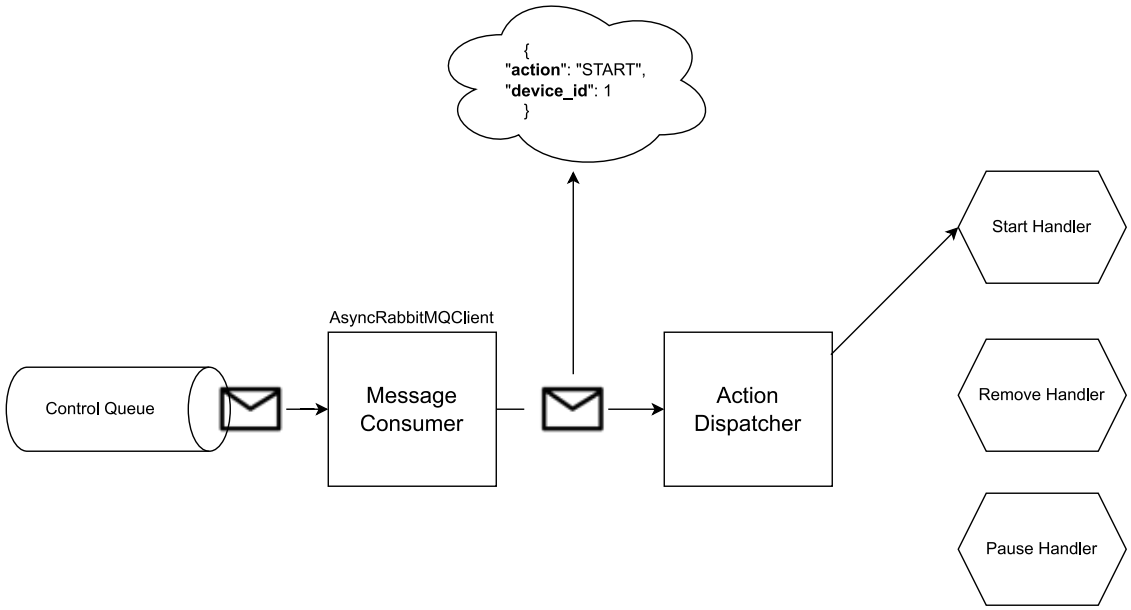

Figure 5.5: Dispatcher pattern

type from the message, which in this case is "START".

Next, the dispatcher looks up the associated function for the "START" action. The function identified is then called, passing the provided arguments (in this case, the device ID). The function executes the necessary logic to handle the "START" action, such as creating a worker for the specified device.

In the case of "START" action, the worker initially is created from an image generated by a Dockerfile. Then proceeds to run an entrypoint to start processing the video feed. If the entrypoint runs without error, it logs a message indicating success reaching the goal. Since this log is used by the instance manager to check if the worker started up correctly, an acknowledge message is only sent with the feedback to the API when the goal has been reached or failed.

The benefits of using this approach lie in its flexibility and extensibility. By mapping actions to functions, the system becomes highly modular and adaptable. Adding new actions requires defining a new function and updating the action-to-function mapping, rather than modifying existing code. This promotes code reuse and simplifies maintenance. Additionally, this approach enhances code readability and separation of concerns, as each function is dedicated to handling a specific action, making it easier to understand and maintain the codebase.

5.2.2 Post message handle

After processing the received message from the Control Queue, two case scenarios can occur: success or error. In the case of success the Instance Manager sends a success message to the *ACK Device State Queue* or *ACK Device Delete Queue* respectively, depending on the action that was requested. It also acknowledges the message so it can be removed from the Controller Queue. This case is illustrated in Figure 5.6.

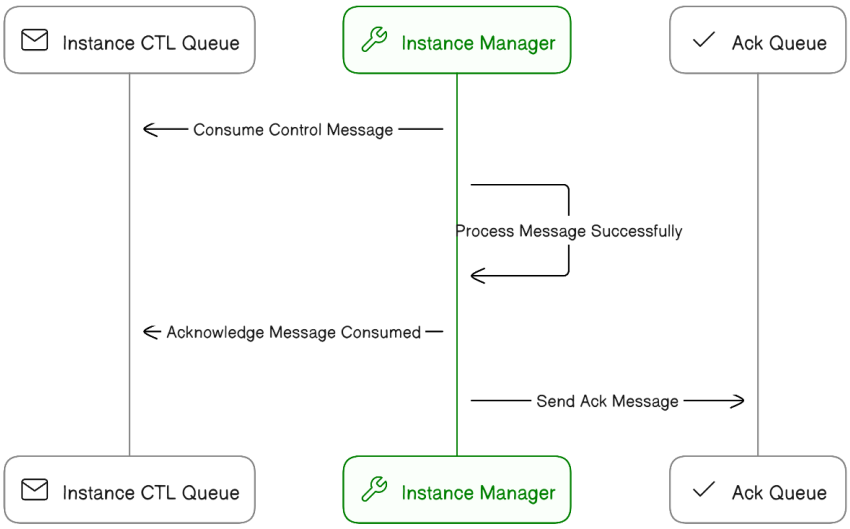

Figure 5.6: Case 1. Message processed correctly

In the case of error, we categorize it into two types: App Error and Internal Error. The App Error occurs due a violation of an application constraint, while an Internal Error occurs when unexpected exceptions happen, such as the Database or Docker being down.

Both types of errors are handled differently. The App Error is acknowledged and removed from the Controller Queue. Additionally, an error message is sent to either the *ACK Device State Queue* or the *ACK Device Delete Queue*, based on the requested action. This case is illustrated in the figure 5.7.

However, for Internal Errors, we have implemented a retry mechanism. Since the cause of the error is unrelated to the message itself, it was important to ensure consistency. For example, in the event of an Internal Error during device deletion, we don't want to notify the clients that the deletion failed and ask them to retry, as it would lead to an inconsistent user experience. To address this, we introduced a retry mechanism triggered by not acknowledging the message and requeuing it. However, this approach resulted in a problem of multiple rapid requeues, which could cause scalability issues if the Database or Docker were down.

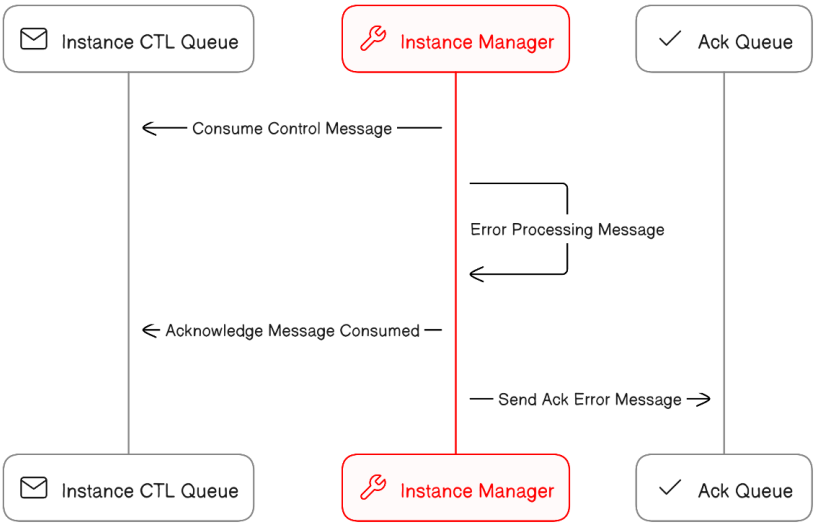

Figure 5.7: Case 2. Error but the message could never be processed

To solve this problem, we implemented the use of Dead Letter Exchanges, a mechanism supported by RabbitMQ. A new queue was created specifically for storing dead messages. By rejecting the messages without the requeue flag, they are redirected to the Instance Control Retry Queue. Here, the messages wait for a pre-defined time before being returned to the Instance Control Queue for retry purposes. This last case is illustrated in Figure 5.8

5.2.3 Environment and configuration

The *Instance Manager* and each of its *Image Processor Worker* is configured by environment variables and a configuration file.

The *Image Processor Worker* config file contains the database and Media server information. The configuration file cannot be changed through environment variables, as this is a bad pratice and should be changed in the future.

The ENVIRONMENT environment variable is used to determine which configuration file the *Instance Manager* will use. The configuration file is located in the $configs$ directory and is named *ENVIRONMENT*.ini. The configuration file reference can be found at *Sensiflow*'s documentation.

It is in this configuration file that is dictated whether the **Image Processor Workers** use hardware acceleration or not and where the Database and RabbitMQ configurations are set.

Two distinct configuration files were used. However this is a bad practice, since the

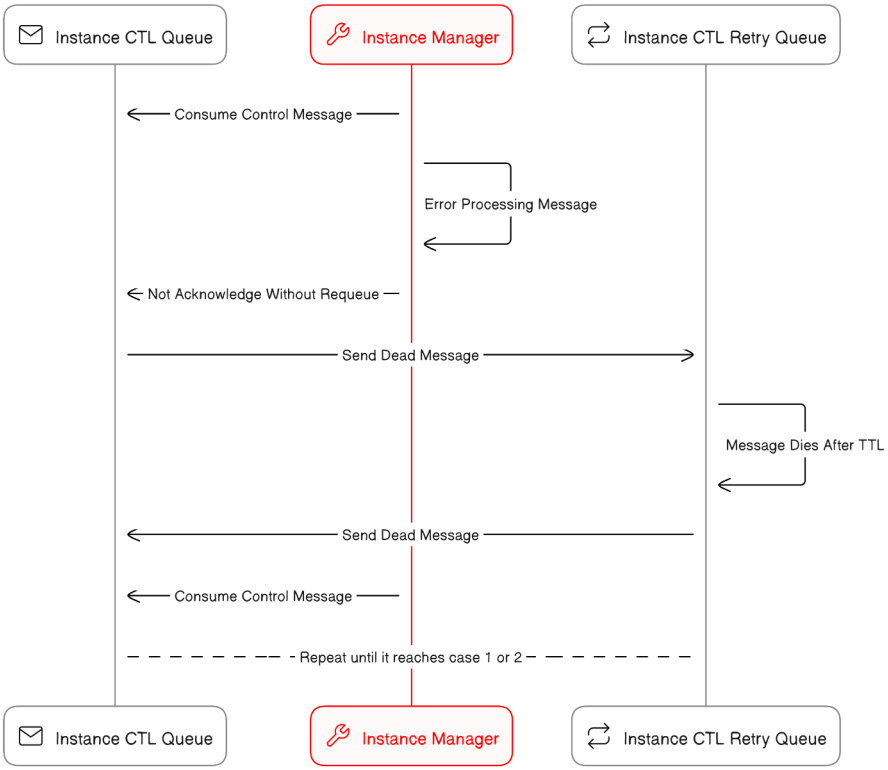

Figure 5.8: Case 3. Error but a message can eventually be processed

worker config file must be manually changed and does not benefit from flexibility of the environment variable. To improve this in the future, we suggest the following options: (1) merge them into one configuration file for both the *Image Processor Worker* and the *Instance Manager*; (2) create an environment variable to dictate what worker config file to use. The first options, if it is possible to implement, is the preferrable one.

This approach has multiple benefits, namely:

- **Flexibility**.
- **Scalability**.
- **Collaboration**.

Flexibility: By using the ENVIRONMENT environment variable, it is possible to easily switch between different environments such as development, testing, and production. Each environment can have its own specific configuration file, allowing the configuration of the settings tailored to the needs of that particular environment.

Scalability: This approach is highly scalable because it allows the addition of new environments or configurations without modifying the code. It is just necessary to create a new configuration file with the appropriate settings, set the ENVIRONMENT environment variable, and the application will automatically load the correct configuration.

Collaboration: This approach promotes collaboration among developers working on the same project. Each developer can use their own environment-specific configuration file without affecting others. It allows developers to customize their development environment while maintaining consistency with the production environment.

5.3 Image Processor Worker

The Image Processor Worker is responsible for processing the video stream, streaming the video feed with bounding boxes on the objects found and writing the metrics to the database. This Worker may use the GPU or CPU to run the machine learning inferences. This option can be configured from the instance manager, as it has full control over the image processor worker. Each device can have up to 1 worker, and these workers may be started/created by the instance manager. There is one worker running for each device with the ACTIVE state. However, there can be less workers than devices with the STOPPED or PAUSED state, because the scheduler might have stopped or deleted some instances to free up resources. The worker runs in a Docker container and runs machine learning inferences supported by yoloV5 vision AI by Jocher (2020) to detect objects in the video stream. The insights obtained are then written to the metrics database table. During the startup of the worker it writes specific logs informing the current steps that is performing.

5.3.1 Metrics Saving Strategy

In our project, we have a worker responsible for saving metrics obtained from a machine learning model to a database. To address the potential issue of frequent and unnecessary database interactions, we implemented a throttling mechanism. Throttling limits the frequency at which the worker updates the database with metrics, optimizing performance while maintaining an acceptable level of precision.

Instead of saving metrics for every inference made by the machine learning model (which occurs every 10 milliseconds on average), we throttle the database interaction to a 1-second interval, this is illustrated in Figure 5.9. Since our objective is to count people in frames, it is unlikely for individuals to enter or leave the frame within milliseconds. By

accumulating metrics over a 1-second period, we strike a balance between precision and efficiency.

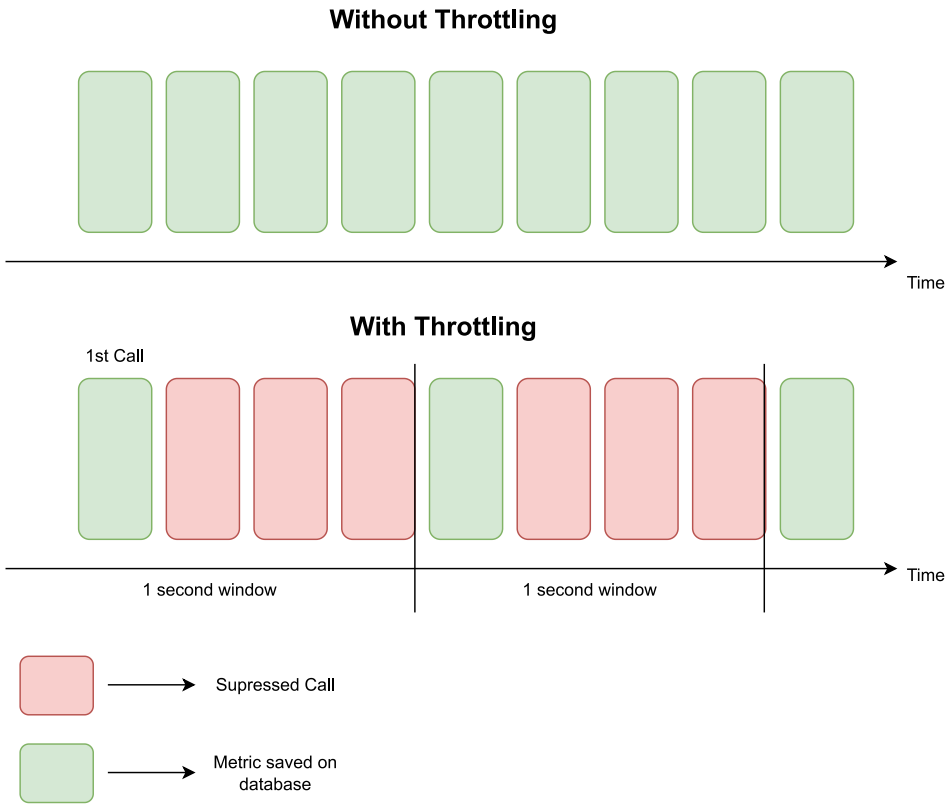

Figure 5.9: Throttling the metrics writing to the database

Throttling provides several benefits, including reduced database load, optimal resource utilization, and streamlined data storage. By reducing the frequency of metric updates, we alleviate the strain on the database and improve its overall performance. Throttling also allows for efficient resource utilization by minimizing the overhead associated with establishing and tearing down database connections. Additionally, consolidating metrics within a 1-second interval simplifies analysis and reporting processes.

5.3.2 Detection stream

Streamer Object A "Streamer" class was made to help abstract the streaming process inside of the image processor worker, this class uses an external program, *FFMPEG*, to perform the video stream. An interface for a "Streamer" was made, which allows in the future the use of a different Streaming class implementation. To start a stream, a subprocess runs an *FFMPEG* command that is able to receive information, this information

are input frames that are received via a "Streamer" function. The stream is done without hardware acceleration. This was done to offer a wider support since not every GPU has the support to perform it with the used configurations.

Stream details In our streaming setup, we have chosen to use a frame rate of 30 *Frames Per Second* (FPS) for the viewer's optimal viewing experience. This decision ensures that the streamed content appears smooth and visually appealing to the user. However, it's important to note that we are only storing statistics in our database at a rate of 1 per second. This means that if there is no need to stream the output, we can reduce the frame rate of the input stream to just 1 fps. This would be part of the ability to configure the worker, as stated in the future work section.

By adjusting the input stream to 1 fps, we can effectively reduce the workload of inferences made by the model matching the inference execution time with the throttling delay. This approach allows us to optimize resource usage and minimize storage requirements. Since we are primarily interested in capturing statistics rather than real-time visual representation, the lower frame rate for the input stream does not significantly impact the overall user experience. Ultimately, this setup strikes a balance between providing a seamless viewing experience and efficiently managing data storage for statistical analysis.

5.4 Instance Manager Scheduler

The scheduler component was built to recover from unexpected errors that might occur in the Instance Manager another purpose is to clean up some instances that might have been left in a PAUSED or STOPPED state, to free up resources automatically. However we had to disable this feature since as the way it was implemented presented a problem to the scalability of the module, leaving it for an improvement in future work. Additionally, it checks and solves inconsistencies between the containers and instances in the database, informing the API if there are inconsistencies that need to be solved relating to the device in question. Such inconsistencies can happen when a stream that a worker is reading from is terminated while processing.

For every machine that is running an instance manager, there needs to be running a scheduler too. This is required because the containers are in the local machine, and use the local docker daemon. In order to have only one scheduler, it is necessary to perform remote access to the docker daemons. This is possible and can be configured as it is referred in the docker official documentation Docker-SDK (2023a). To simplify this process in the future we separated the entrypoint code for the scheduler from the instance

manager. However we did not have time to perform this change, leaving it for future work.

Chapter 6

Web Client Module

Contents

In the web client chapter of our report, we focused on building a *Single Page Application* (SPA) using React.

6.1 Structure

A SPA is a web application that dynamically updates its content without requiring full page reloads. This approach provides a seamless user experience as the application behaves more like a desktop application, with smooth navigation between different sections and pages. In our case, we incorporated a dashboard with a sidebar navigation, and by leveraging React Router, we ensured that users could navigate between pages within the dashboard without the need for page reloads. This enhances usability and responsiveness, making the overall user journey more efficient and enjoyable.

External Modules

- *react-hook-form* rea (2023b) - Provides forms with easy-to-use validation.
- *react-player* rea (2023a) - The packaged used to support the video player.
- *react-router-dom* rea (2023c) - Package used to configure routes for our app.
- *react-pro-sidebar* Azouaoui (2023) - Package with a base sidebar component.
- *axios* axi (2023) - Promise based HTTP client for the browser and node.js.
- *MUI* mui (2023) - A component library.
- *echarts* team (2023) - The library used for the charts.
- *jest* jes (2023) - Testing framework.
- *js-cookie* js- (2023) - API for handling Cookies.
- *hls.js* hls (2023) - Library which implements an HTTP Live Streaming client.

Build tool and dependency manager

To develop the Web application the *Node.js* environment was used, since it offered a dependency manager tool, and an infrastructure for the code to run on that grants reusability, isolation. However since the module system used by Node.JS, CommonJS, is not supported by the browser, a tool was required to convert the code's module system from CommonJS to *EcmaScript Modules* (ESM) which is supported natively by the browser.

To manage the dependencies, the *Node Package Manager* (NPM) tool was used, which is the default dependency manager tool of *Node.js*.

The bundle tool Webpack was used since the language used for the Web application was typescript, since the browsers don´t support it and also the fact that the module system used is not also supported, it was required the use of Webpack to convert and bundle the code into a single javascript module. In order to simplify the development, webpack also provided an HTTP server to serve files on, it also provided the capacity to perform *Hot Module Reload* (hmr), which updates the code changed, without the necessity to reload the page.

Directory Structure

Since we are using react with webpack our project has the common *nodejs* project structure.

```
src ..... source code of the project
├── api ..... API access module
├── assets ..... Static assets such as images, icons, fonts
├── components ..... Reusable UI components
├── pages ..... Routes within the application
│   ├── auth ..... Authentication pages e.g login
│   └── dashboardspa ..... Dashboard SPA pages
├── model ..... Domain Entities
├── logic
│   ├── context ..... Application contexts and respective providers
│   ├── events ..... Custom event types
│   ├── hooks ..... App custom hooks
│   └── reducers ..... Reducers used in the application
├── theme.ts ..... Application theme context and provider
├── index.tsx ..... Entry point
├── app.tsx ..... Outer app router (Between login and dashboard)
└── index.css ..... App main styles
```

6.1.1 Application navigation

In our application, navigation is achieved through the use of the React Router package. React Router provides declarative routing capabilities, allowing the developer to define the navigation paths and associate them with specific components.

The router is configured using the *BrowserRouter* component, which sets up the necessary infrastructure for client-side routing.

Within the <*Routes*> component, we define various routes that correspond to different pages or views in our application. Each route is associated with a specific URL path and

an associated component that should be rendered when that path is matched.

In our case, we have routes for the login page, as well as the dashboard section. The dashboard section is protected, meaning that the user must be authenticated to access it. This process is achieved by using a custom made *<ProtectedRoute>* component, which provides support to allow only authenticated and authorized users to access specific routes, the required authorization may be configured through the component parameters.

Additionally, we have nested routes within the dashboard section for the following pages:

- Home page.
- Devices Page.
- Device Page.
- Groups Page.
- User management Page.

These nested routes allow for a hierarchical organization of our application's pages.

Navigation graph

The figure 6.1 represents the navigation graph for our application based on the provided router configuration.

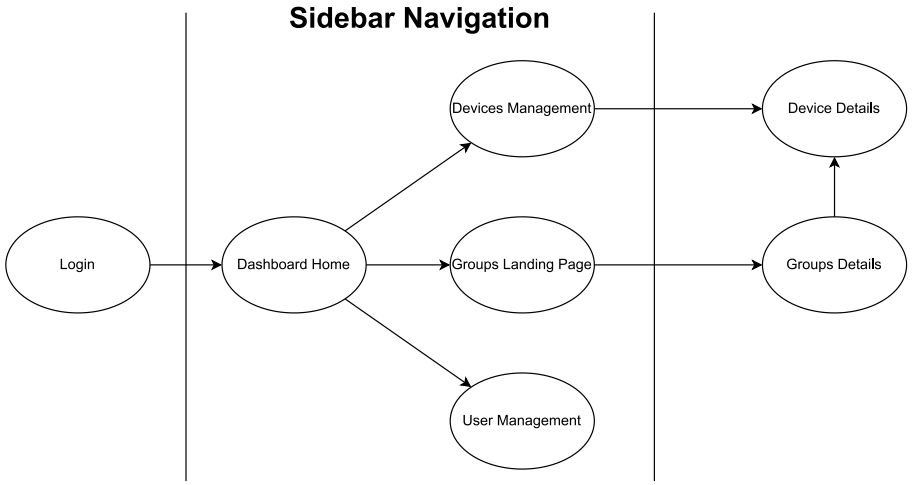

Figure 6.1: Web application navigation graph

6.1.2 Authentication and Authorization handling

To authenticate a user it is used the */login* endpoint of the API, this provides a session cookie however this cookie is *HTTPOnly* meaning that is cannot be accessed by Javascript, this brings the problem of not knowing when the user session will expire, however this can be easily solved by storing some information that indicates the user is logged in, this information has to expire in a bit before the session cookie expiry time, so that the user does not make requests to endpoints requiring authentication when we does not have really have it. To store this information we came up with three alternatives, browser's localStorage or sessionStorage or Cookies. The table 6.1 illustrates the difference between each of these options

Table 6.1: State preservation options available on a browser

	Cookies	Local Storage	Session Storage
Capacity	4KB	10MB	5MB
Accessible from	Any Window	Any Window	Same tab
Expires	Manually set	Never	On tab close
Storage location	Browser and Server	Browser only	Browser only
Sent with requests	Yes	No	No

The information stored must be accessible from any window as it makes sense for the user to change windows and maintain its authentication, a manually expire time must also be set. Given these limitations, the cookie was used for the Web client to be aware of the user authentication status. The figure 6.2 represents how the web client is aware of the authentication, where the Cookie B is just a representation of Cookie A, having no sensitive information crucial to authentication, and Cookie A is the cookie that handles the real authentication on the API and has the crucial information to authenticate the user.

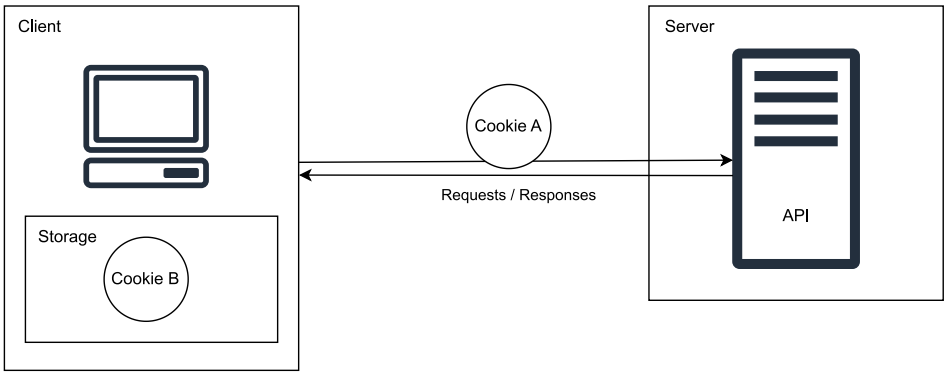

Figure 6.2: Web client side authentication

6.2 React

6.2.1 Custom hooks

Use of custom hooks allows us to reuse code and keep our components clean and easy to read.

useWindowSize: This custom hook is designed to track and provide the current width and height of the browser window to components in a React application. Since, the resize event can be triggered multiple times rapidly in short amount of time, the debounce hook was used to ensure that it will only update after a time of inactivity.

Overall, the *useWindowSize* custom hook enables components to dynamically adapt their behavior based on the current size of the browser window, providing a responsive user experience.

useCookie: This hook is responsible for persisting the received state into a cookie using a given key, it also provides operations to update/remove the cookie and the capacity to execute a callback when its cookie expires.

useSSE: The purpose of this custom hook is to facilitate the integration of *Server-sent events* (SSE) functionality into a React component. This hook streamlines the process of setting up and managing SSE connections by handling the creation, event listening, error handling, and cleanup logic.

The *useSSE* hook establishes and maintains an SSE connection when the component is active. It automatically opens the connection, listens for specific events, and handles any errors that may occur. When the component is no longer active or when the dependencies change, the hook closes the connection and performs the necessary cleanup. This abstracts the complexities of SSE integration, making it easier for React components to incorporate real-time updates from the server in a controlled and efficient manner.

6.2.2 State Management

In this section it is described how the application state was managed.

Context

The context is used to access information without having the need of passing it down the *Document Object Model* (DOM) tree. For that purpose, the higher-level components

that need access to the shared information within, must be wrapped with the Provider component. The Provider accepts a value property that contains the data to be available. The diagram 6.3 represents how a user can use the context to obtain data.

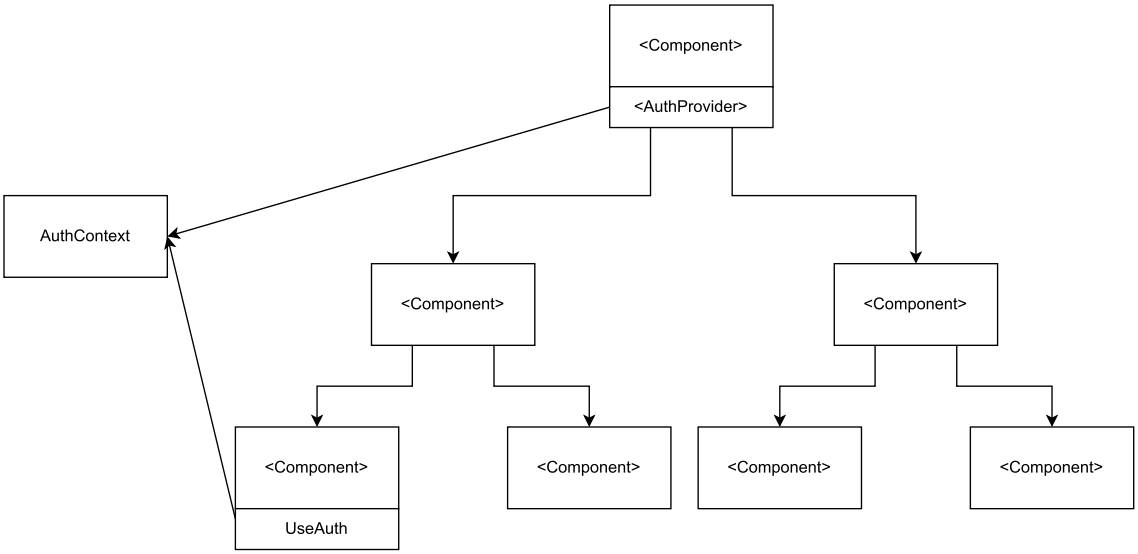

Figure 6.3: Context data flow

Given this, the custom context *AuthContext* was made, it is also complemented with a custom hook to simplify the use of the contexts. The hook just return the current context value, as given by its context provider.

AuthContext This context was made due to the necessity of accessing user information in various components at different levels of the tree, and also to avoid repeating fetching the user information from the API. It also provides operations to act upon this user information and to login/logout the user.

A custom wrapper for its provider was made, this wrapper is responsible for storing the current user's information in the a cookie as previously mentioned in the "Authorization and Authentication" section on 6.1.2 this is done with the help of the custom made *useCookie* hook.

By storing the user´s information in the cookie, it is not lost and can be retrieved after closing and opening the application, however the access to this information is limited by the user session, this means that in order for the information to stay stored the user must be authenticated, if its session expires, the stored user data will be deleted. The provided operations are the following:

- login - Operation to execute the login operation by making a request to the API and storing the user information.

- logout - Operation to execute the logout operation comunicating it to the API and deleting the stored user information.
- isLoggedIn - Returns a boolean that dictates whether a user has a valid session.
- user - Provides the current authenticated user
- fetchCurrentUser - Operation to get the authenticated user's information from the API and save it.

Dialog Reducer

Given the fact that lots of pages required dialog components and every dialog required to have an associated state to control whether it is open or closed. For simplicity reasons a state management component was made, it is the generic dialog state manager. This reducer allows the management of multiple dialogs.

6.3 Reading a processed stream

6.3.1 Player

The video player used supports playing HLS and DASH streams, this player was chosen as it is one of the biggest player libraries compatible with react.

Authentication

Since the Media server is using Basic Authentication, the credentials must be supplied when trying to access a stream video the player and since the player does not allow the direct configuration of the credentials. To achieve this, the options of the "hls.js" package were changed to set the request header with the Basic credentials hashed in Base64.

6.3.2 Streaming protocols

Real Time Streaming Protocol (RTSP), is a network control protocol designed for controlling and delivering real-time multimedia data. It is the standard streaming protocol used on video surveillance systems, where it enables the transmission of live video streams from cameras to monitoring systems.

Unlike HTTP-based streaming protocols such as *Dynamic Adaptive Streaming over HTTP* (DASH), *HTTP Live Streaming* (HLS), and *Web Real-Time Communications* (WebRTC), which are designed for web-based environments, RTSP operates differently.

RTSP does not handle the actual streaming of the video content, the responsible for this is actually the *Real-time Transport Protocol* (RTP) that RTSP uses internally together with *Real-time Control Protocol* (RTCP). Given this, RTSP, it is responsible for controlling the stream and establishing the communication between the client and server

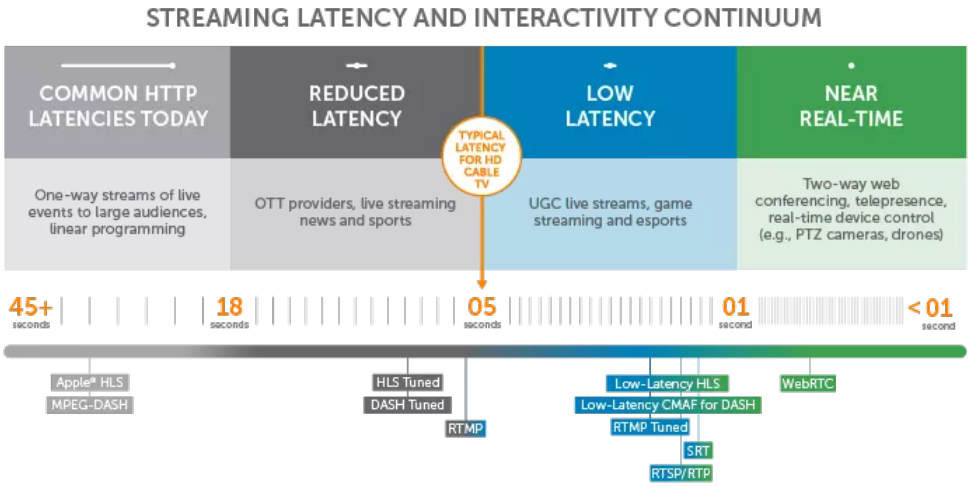

Figure 6.4: Average latency for each streaming protocol

In contrast, HTTP-based streaming protocols are designed to work within web browsers, making them more suited for our web app.

From the HTTP-based streaming protocols, the MediaMTX server supports transcoding RTSP into HLS or WebRTC and since DASH is quite similar to HLS, those were the two options for the HTTP-based streaming protocol to use.

When deciding between HLS and WebRTC as the HTTP-based streaming protocol to use for the project, the advantages and disadvantages of each were evaluated. HLS is an HTTP-based streaming protocol that is widely supported across browsers and devices, making it a reliable choice. However, this protocol introduces a 2 second delay in the video playback.

On the other hand, WebRTC is a real-time communication protocol that offers low latency (around 0.5s) and high-quality streaming capabilities. It is ideal for applications that require real-time video communication, such as video conferencing and live streaming. However, it is not as widely supported across browsers as HLS, and it requires more configuration and setup.

In this project, since the delay is not a defining factor and real-time viewing is not required, HLS was chosen as the streaming protocol. Its widespread support and reliability make it a suitable choice for delivering video content from the cameras to the browser-

based client.

Chapter 7

Discussion

Contents

7.1 Special operations

The most intricate and interconnected part of our project involves complex operations that require seamless communication between all components. This segment plays a crucial role in achieving high levels of efficiency, reliability, and performance.

7.1.1 Device processing State Update

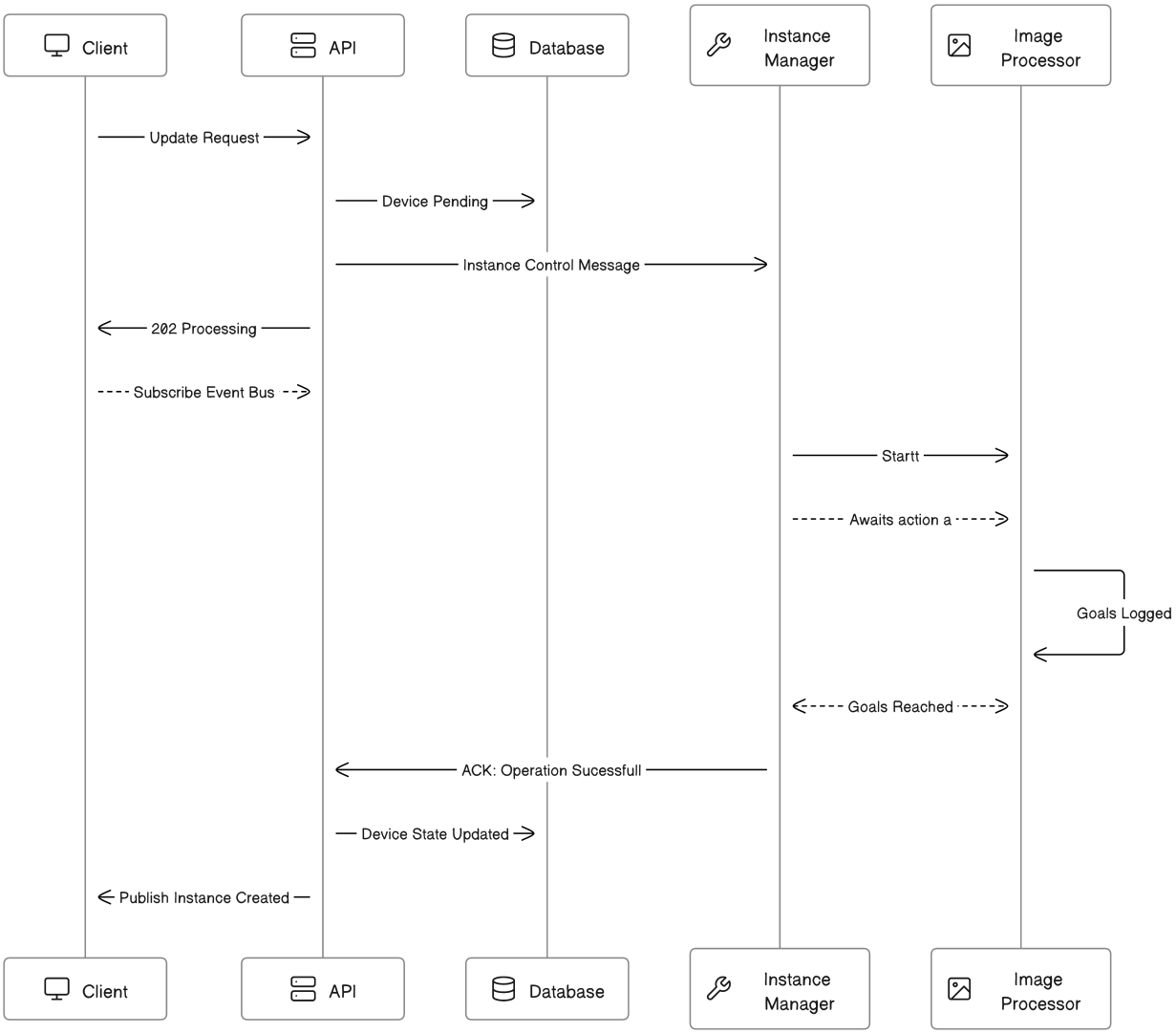

Figure 7.1: Device processing State Update Mechanism

The following Figure 7.2 describes the process of updating a device state. The data flow within our system begins when the Client initiates an Update Request, sending the request to the API. The API then interacts with the Database, marking the corresponding

Device as Pending. Simultaneously, the API communicates with the Instance Manager, sending an Instance Control Message to coordinate further actions. To keep the Client informed, the API responds with a 202 Processing message.

At this point, the Client subscribes to the Event Bus, establishing a communication channel. The Instance Manager, upon receiving the necessary instructions, triggers the Image Processor component to start its operations, denoted by the Start message. The Image Processor awaits an action, marked as "a," before proceeding. During this time, the Image Processor logs its goals internally for reference.

As the Image Processor progresses, it communicates bidirectionally with the Instance Manager to update the progress and any achieved goals. This ongoing communication ensures synchronization and allows for efficient coordination. Meanwhile, the Client remains engaged in the flow.

Upon successful completion of the operation, the Instance Manager sends an acknowledgment (ACK) to the API, indicating the successful execution of the operation. The API then updates the Device State in the Database accordingly. Lastly, the API notifies the Client that the processing state has been updated.

7.1.2 Device Deletion Process

Deleting a device may initially seem straightforward as a simple CRUD operation - deleting the device entry in the database and removing all related entries in tables where the device is referenced. However, a complication arises when active metrics are still being written for the device within the API. Due to the deviceid serving as a foreign key in the metrics table, attempting to delete the device would violate this constraint and lead to unexpected errors. This challenge could be overcome by decoupling the constraints, such as storing metrics in a separate database, which is a topic discussed in our future work section. As a workaround, we devised a solution to delete the instance prior to deleting the device. By doing so, we ensure that no metrics are being written to the database when a device is successfully removed.

Device deletion dataflow

In the described flow, the Client initiates a Delete Request, sending it to the API. The API then updates the status in the Database, marking the Device as Scheduled for removal. Simultaneously, the API communicates with the Instance Manager, sending an Instance Control Message to coordinate the deletion process.

To inform the Client about the status, the API responds with a 204 No Content mes-

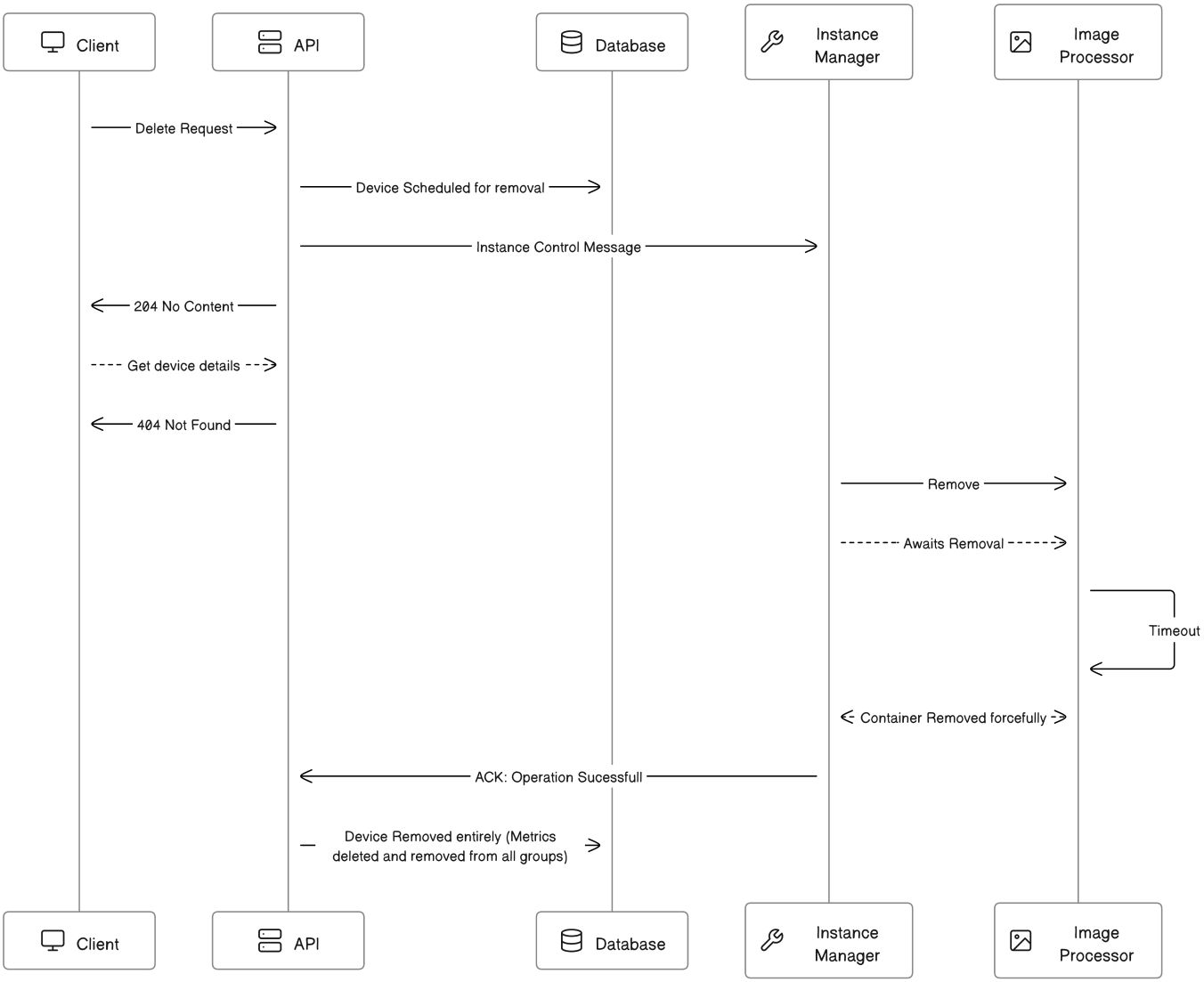

Figure 7.2: Device processing State Update Mechanism

sage, indicating the successful acceptance of the request. The Client, still engaged in the flow, proceeds to request device details from the API.

However, since the Device is scheduled for removal, the API responds with a 404 Not Found message, indicating that the Device no longer exists. At this stage, the Instance Manager takes action, triggering the Image Processor component to remove the associated container. The Instance Manager then awaits the removal process to complete.

During the removal process, the Image Processor has a timeout to remove the container until it is forcefully removed by the manager. This ensures the removal process continues smoothly.

Throughout this operation, the Client remains involved in the flow. After the container

is removed, the Instance Manager sends an acknowledgment (ACK) to the API, indicating the successful execution of the operation. Subsequently, the API proceeds to remove the Device entirely from the Database, including its associated metrics and removing it from all relevant groups.

7.2 DevOps

Each main module has a linter plugin, which helps to keep a consistent code style and flags errors and common bugs. The development of the project modules is assisted by the use of *Continuous integration* (CI) provided by github actions.

In each module, we made a workflow config file to run every time code is pushed to a pre-defined branch, in this config file it configured the necessary infrastructure to run at least 2 checks

- Lint.
- Test.

This process helps us by improving the code quality and maintaining a tested code base. Another great advantage to the use of CI is that it ensured portability since the tests ran on a different environment rather than our own machine.

7.3 Security

In order to grant privacy for our clients, it was necessary to encrypt the communication content between the client and servers, as someone might be listening to the connection and intercept sensitive data. To solve this issue a solution is to use secure connections, these can be obtained with *Transport Layer Security* (TLS) certificates.

7.3.1 Obtaining Certificates

To obtain a trusted TLS certificated it is required that it is signed by trusted Certificate Authorities, in order to achieve this, it is necessary to own a domain. Given this we obtained a domain via the *Google Cloud Platform* (GCP) platform. This allowed us to get trusted certificates via a third party (Let's Encrypt) let (2023). The certificates were obtained following the tutorial available on the Let's Encrypt website.

7.3.2 Using certificates

By having Secure connections throughout the modules, we can guarantee client privacy. However every vulnerable point must be secured, this is because a system is only as secure as its weakest link.

Web API

To use TLS on Spring it was required to install another dependency, “Spring Security”. However to use *Hyper Text Transfer Protocol Secure* (HTTPS) it was required a Keystore with the certificate and private key bundled. This was made with the assist of the command “keytool”. After obtaining the Keystore, the application.properties was properly configured and every http request was redirected to https.
This feature can be turned off using environment variables.

Media server

The use of TLS on the Media server allowed the use of protocols such as *Real Time Streaming Protocol Secure* (RTSPS) and *HTTP Live Streaming* (HLS) secure, it was just necessary to fine-tune the media server configuration and provide the certificates, more information on the configuration can be obtained in the media server’s github. By changing the Media server protocols to be secure, it is necessary to change environment variables throughout the modules to grant that,

Load balancer

Since the interface with the clients is the load balancer, this piece of software needs to be secure, to perform this, it was just necessary to change the NGINX configurations, it is important to note that a Secure connection can be configured to be toggled with the use of environment variables.

Instance Manager

The only security vulnerability point in the instance manager is the streaming to the media server. However this can be configured to be safe via the instance manager config file, it is important to note that to enable this Secure option, the media server must be also secure.

Web Client

There is no need to tweak the Web Client. As all the connections it uses are already secure if properly configured.

Chapter 8

Conclusions and Future Work

Contents

8.1 Conclusion

SensiFlow solves key problems by utilizing person detection technology and data analysis capabilities. It addresses the lack of real-time occupancy data, eliminates the limitations of manual analysis, improves operational efficiency and safety, provides valuable insights for decision-making, and enhances customer satisfaction. By automating occupancy monitoring and enabling data-driven strategies, SensiFlow empowers organizations to optimize resources and implement effective crowd management. Every mandatory requirement was successfully fulfilled.

Through the transformation of requirements into tasks and the utilization of diverse project management tools, we have successfully attained a high level of autonomy among the team members, enabling simultaneous progress. Also, by initially adopting CI we could significantly reduce the time and effort required for manual testing, improve code quality and ensure a more stable and reliable product.

Lastly, it is crucial to emphasize that the comprehensive documentation and dedicated effort invested in this project have brought us significantly closer to the realization of a future product-oriented company. This accomplishment not only instills a deep sense of pride among the team members but also serves as a powerful source of motivation.

8.2 Future Work

Having in mind that the focus of the application development was more on the robustness, flexibility and scalability of the system, the optional requirements were not fulfilled. However, with the current architecture, it would be easy to implement additional features without changing it. There are lot of options for future work, but some of them are:

- All the previous optional requirements.
- Update YOLOv5 to YOLOv8.
- Use WebRTC Protocol to receive video on the Web Client video Player.
- Change the authentication method of the Media server to external authentication.
- Use a non relational database to store the metrics.
- Provide elasticity for the instance manager module
- Use remote access to docker daemon on the scheduler

- Utilize the scheduler abilities to free resources automatically.
- Improve the web application's overall user experience and interaction.

References

(2023). Axios. URL: https://github.com/axios/axios.

(2023). Delightful javascript testing. URL: https://github.com/jestjs/jest.

(2023). Ffmpeg. URL: https://ffmpeg.org/download.html.

(2023). Getting started. URL: https://letsencrypt.org/getting-started/.

(2023). hls.js. URL: https://github.com/video-dev/hls.js/.

(2023). Jackson. URL: https://github.com/FasterXML/jackson-module-kotlin.

(2023). Javascript cookie. URL: https://github.com/js-cookie/js-cookie.

(2023). Jpa. URL: https://spring.io/projects/spring-data-jpa.

(2023). Junit4. URL: https://junit.org/junit4/.

(2023). Klint. URL: https://pinterest.github.io/ktlint/0.50.0/.

(2023). Material ui. URL: https://mui.com/.

(2023). Poetry. URL: https://python-poetry.org/.

(2023a). A react component for playing a variety of urls. URL: https://www.npmjs.com/package/react-player.

(2023b). React hook form. URL: https://www.react-hook-form.com/.

(2023c). React router. URL: https://reactrouter.com/en/main.

(2023). Spring. URL: https://spring.io/.

(2023). Spring amqp. URL: https://spring.io/projects/spring-amqp.

(2023). Work queue systems. URL: https://www.oreilly.com/library/view/designing-distributed-systems/9781491983638/ch10.html.

Akamai, M. N., & Wilde, E. (2016). Problem details for http apis. Internet Requests for Comments. URL: https://www.rfc-editor.org/rfc/rfc7807.txt.

Azouaoui, M. (2023). React pro sidebar. URL: https://github.com/azouaoui-med/react-pro-sidebar.

Bluenviron (2023). Media mtx server. https://github.com/bluenviron/mediamtx, last acessed on 2/06/23.

Docker-SDK (2023a). Configure remote access for docker daemon. https://docs.docker.com/config/daemon/remote-access/, last acessed on 25/06/23.

Docker-SDK (2023b). Develop with docker engine sdks. https://docs.docker.com/engine/api/sdk/, last accessed on 14/04/23.

Fowler, M. (2015). Presentation domain data layering. URL: https://martinfowler.com/bliki/PresentationDomainDataLayering.html.

Fowler, M. (2020). Domain driven design. URL: https://martinfowler.com/bliki/DomainDrivenDesign.html.

Girshick, R. (2015). Fast r-cnn. In *International Conference on Computer Vision (ICCV)*.

Girshick, R., Donahue, J., Darrell, T., & Malik, J. (2014). Rich feature hierarchies for accurate object detection and semantic segmentation. In *Computer Vision and Pattern Recognition*.

Jocher, G. (2020). Yolov5 by ultralytics. URL: https://github.com/ultralytics/yolov5. doi:10.5281/zenodo.3908559.

Jung, J. (2021). What are salted passwords and password hashing? https://www.okta.com/blog/2019/03/what-are-salted-passwords-and-password-hashing/, last acessed on 2/05/23.

Mosquito (2023). Aio pika. URL: https://github.com/mosquito/aio-pika.

Poole, J. (2018). Push vs pull architectures. URL: https://medium.com/@_JeffPoole/thoughts-on-push-vs-pull-architectures-666f1eab20c2.

Redmon, J., & Farhadi, A. (2018). Yolov3: An incremental improvement. *arXiv*, . URL: https://pjreddie.com/darknet/yolo/.

Sourcemaking (2023). Design patterns. URL: https://sourcemaking.com/design_patterns.

team, A. (2023). Echarts. URL: https://github.com/apache/echarts.

dev team, D. S. (2023a). Docker sdk for python. URL: https://github.com/docker/docker-py.

Team, O. (2023). Cascade classifier. URL: https://docs.opencv.org/3.4/db/d28/tutorial_cascade_classifier.html.

team, P. (2023). Psycopg 3 – postgresql database adapter for python. URL: https://github.com/psycopg/psycopg.

dev team, P. (2023b). Pytest framework. URL: https://github.com/pytest-dev/pytest.

dev team, P. (2023c). Pytest plugin for environment variables. URL: https://github.com/pytest-dev/pytest-env.

dev team, T. (2023d). Tox. URL: https://github.com/tox-dev/tox.

Ziadé, T., sottile, A., & Cordasco, I. (2023). Flake8: Your tool for style guide enforcement. URL: https://github.com/PyCQA/flake8.