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

June 2022

Abstract

With the rise in criminal activity, the deployment of surveillance cameras has become increasingly prevalent. Nowadays, these devices can be found almost everywhere, in almost every establishment. And with the advancement in technology, these devices now support plenty of useful features, such as alerting authorities to specific behaviors and facial recognition. However these improved devices are costly, our project aims to add a feature to devices that do not support 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
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
GPU	Graphics Processing Unit
gRPC	Google Remote Procedure Call
HLS	HTTP Live Streaming
IPC	Inter-Process communication
JDBI	Java Database Interface
JPA	Java Persistence API
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
SPA	Single Page Application
SQL	Structured Query Language
SSE	Server-sent events
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

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. Data analysis has gained significant importance across several domains, including retail, healthcare, and transportation. Understanding the flow and density of people is crucial in improving efficiency, safety, and customer satisfaction.

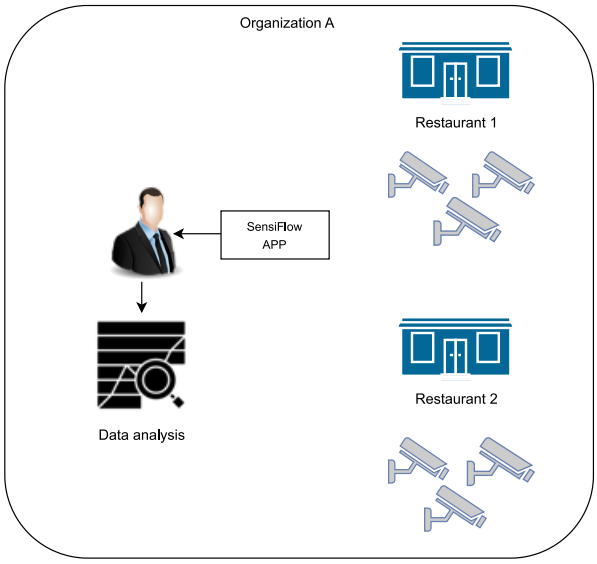

Figure 1.1: Use case for the SensiFlow App

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 its discussed what we learned by making this project 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

The *You Only Look Once* (YOLO) 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) and Fast R-CNN, YOLO offers 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.

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.

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

- 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.
- Dashboard for data visualization.

2.2.2 Mandatory Non functional requirements

- *Scalability* of the system
- Secure transmission of stream with detected people.
- HTTPS on the API
- Deployment of the application

2.2.3 Optional Requirements

- 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

2.3 Related work

In our exploration of the computer vision market and similar applications, we came across Viso.ai, 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

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

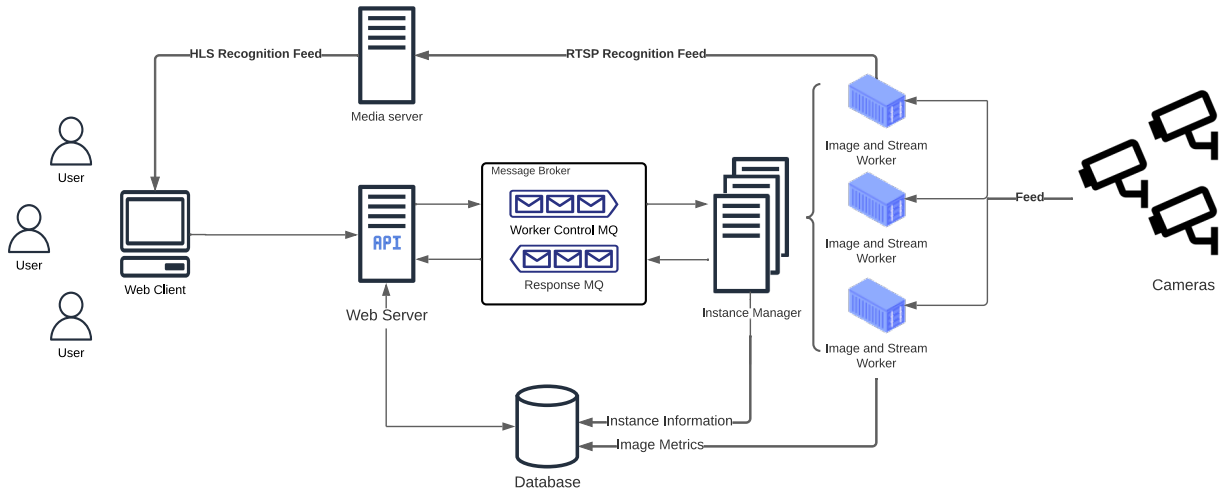

Figure 3.1: Proposed Architecture

3.1.1 Architectural Justification

In this section, we present the Architectural Justification. Diving 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 project goals and requirements.

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/receive feed. Having this in mind, in order to get the stream from the server it is necessary that the client provides the respective *Uniform Resource Locator* (URL) that the camera is publishing to. That 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 server that provides static assets and a RESTful API.

Web Server

The Web Server'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, however combining these modules is not right, since it would mix their domains, they would have different purposes and would need to scale their own way. Given this, dividing these two modules provides a greater scalability, and component decoupling.

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*. Given this necessity, with the help of the message queue as a load balancer to one or more Instance Manager components that tackle this problem by managing its 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 Server, 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 server, the asynchronous message queue emerges as the most suitable choice. While IPC, an API on the instance manager and Remote procedure call offer means of communication, 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 server, 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 or port.

While gRPC is a valid option as 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 server 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.

3.2 Technologies and tools

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

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 theAPI 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 as the language 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 this option is not as *scalable* as a noSQL database, but since our app is designed for organizations there is not a need for *scalability* in the database since it won´t be overloaded with the large amounts of data. Hence the language chosen was PostgresSQL as this was the database we had most experience with.

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 lighter in weight 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 (2023), 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 however Kafka was discarded 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 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 an *Real Time Streaming Protocol* (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 and 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 a 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 are 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.

3.3 Data Model

The fig 3.2 illustrates the chosen data model.

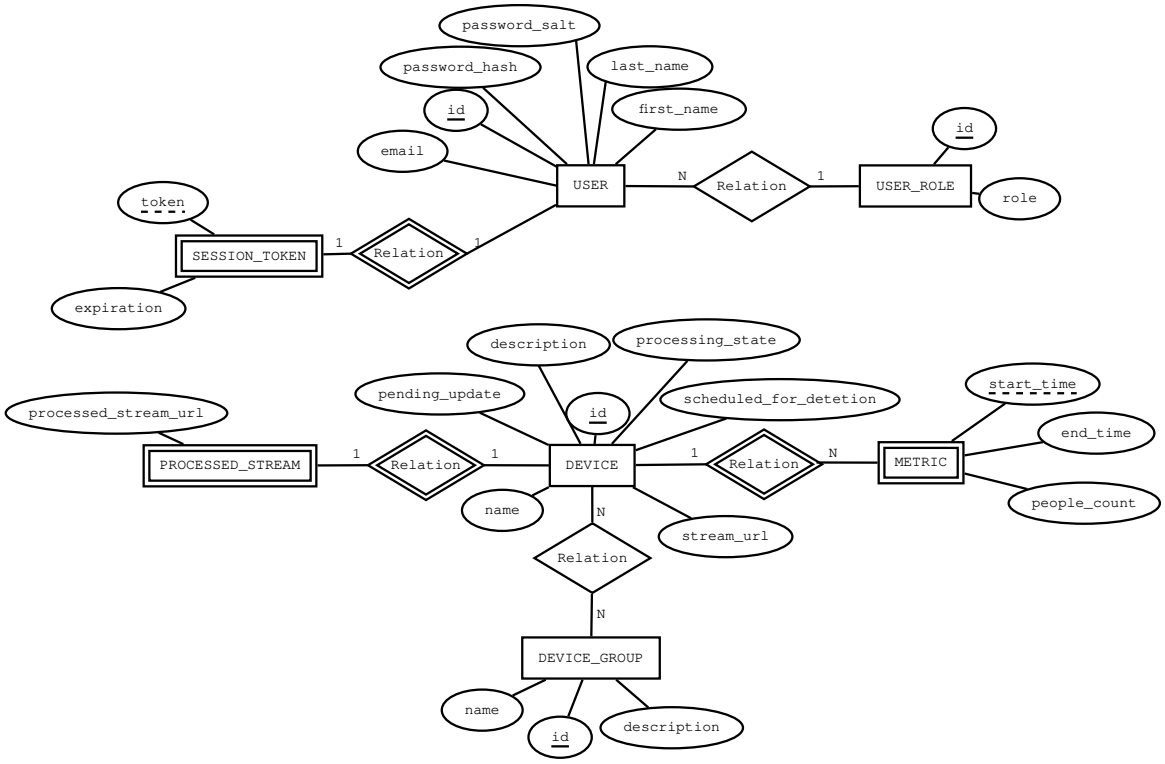

Figure 3.2: Relational Model

- 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;
- Processed_Stream – Represents a processed video stream and contains the URL of that stream;
- Device - Represents a camera and contains a name, description, the processing state and the pending update. This 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, 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.

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.

4.1 Architecture

"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." Sourcemaking (2023) Having that in mind, the Web API was designed using patterns already known, being the Presentation-Domain-Data-Layering 4.1 and Domain Driven Design, as well as other concepts like SOLID. The layering allows to separate the code in three topics relatively independently. 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.

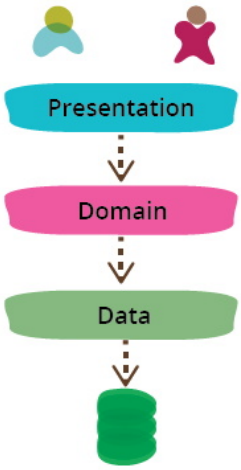

Figure 4.1: API Layered Structure

The main traits of the chosen architecture are:

- Testability
- Independent of the UI
- Independent of the Database

Directory structure

This is 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.

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. 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. Which after configuration associates endpoints to handlers available on the controllers, these are called by HTTP requests. 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 malicious 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.

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, 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 the following figure 4.2:

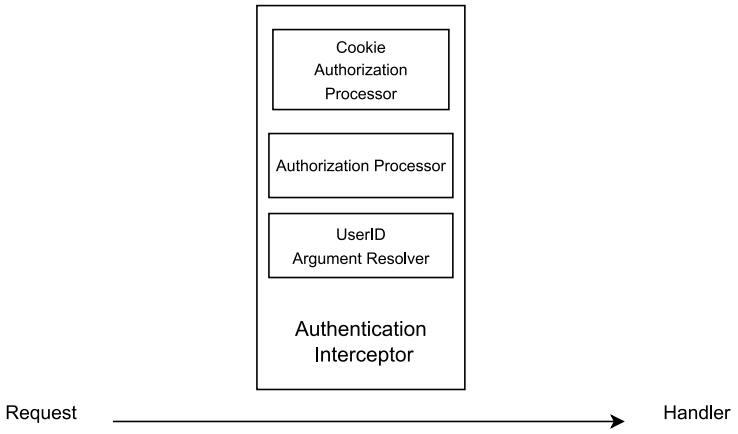

Figure 4.2: Spring Request Pipeline

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

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
```

4.2.4 Server-sent events

Since changing a device processing state is an asynchronous call, when changing it to a pending state, the client needs to know when the change happens. For that reason, and since it was needed a 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. 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 separate 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 Figure4.3.

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	

Figure 4.3: API endpoints available

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.

4.3.1 Transaction Management

The service layer may call different repositories to perform DB 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 1st operation on an 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 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 we want to throw an exception and not rollback, it is used a *noRollbackFor* rule that it is associated to an 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), this 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, *ID* 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 two occasions when this happens, when a user wants to delete a device, or when a user changes a device's processing state.

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 other 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 figure 4.4 illustrates the messages format and the queues available to establish communication.

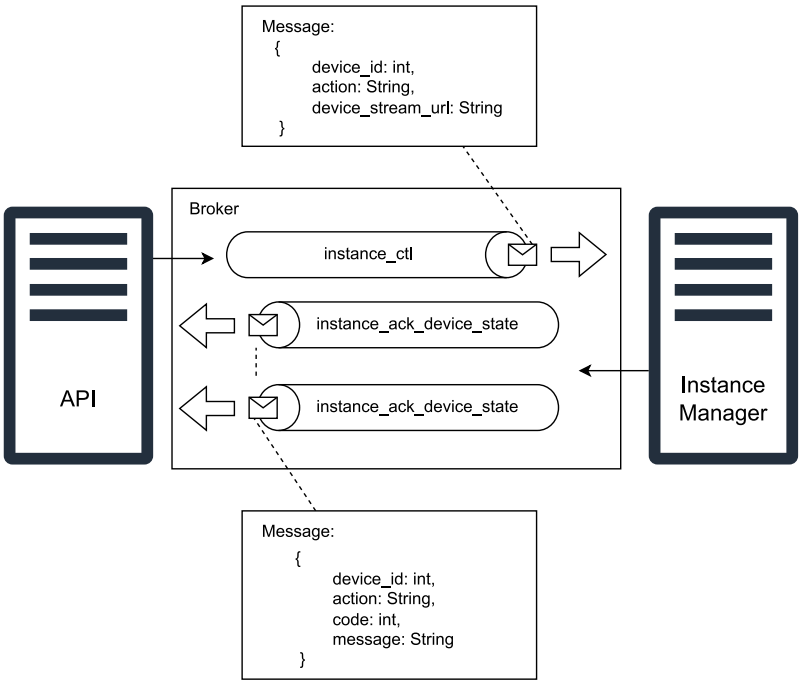

Figure 4.4: Message Channel between API and Instance Manager

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 necessaries for the application 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.

Chapter 5

Instance manager Module

Contents

5.1 Structure

This module is composed of the following components:

- Instance Manager
- Image Processor Worker
- Instance Manager Scheduler

External Modules

- *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.

Build tool and dependency manager

To simplify the process of managing project dependencies, creating virtual environments, and building and publishing packages it is used poetry. 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 the 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.

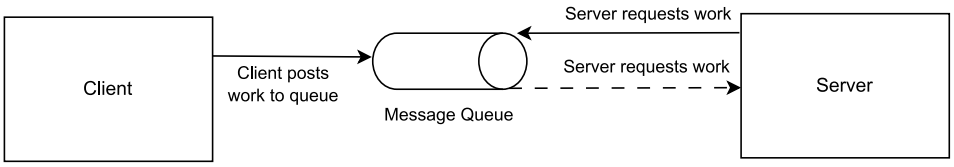

Figure 5.1: Pull architecture

Another choice would have been a push-style architecture as it is illustrated in the 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.

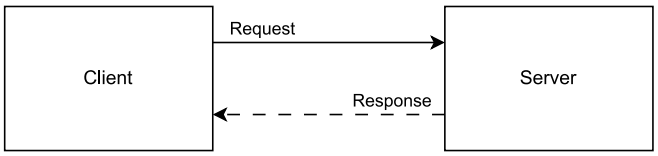

Figure 5.2: Push architecture

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

```
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.

5.2.1 Controlling the Instance Manager

Communication with the Control Queue

In the communication with the control queue, where the control messages are sent, we have implemented an efficient and scalable approach. 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 seamlessly integrate them into our system.

The processing of the messages from Control Queue is made in a Worker Pattern, by using this pattern it is possible to have N Workers receiving messages to process at a time, in our case, a Worker is an **Instance Manager** instance. This pattern is used to grant a scalable approach. The figure 5.3 illustrates the worker pattern applied when the number of instances of the textbfInstance Manager is scaled up.

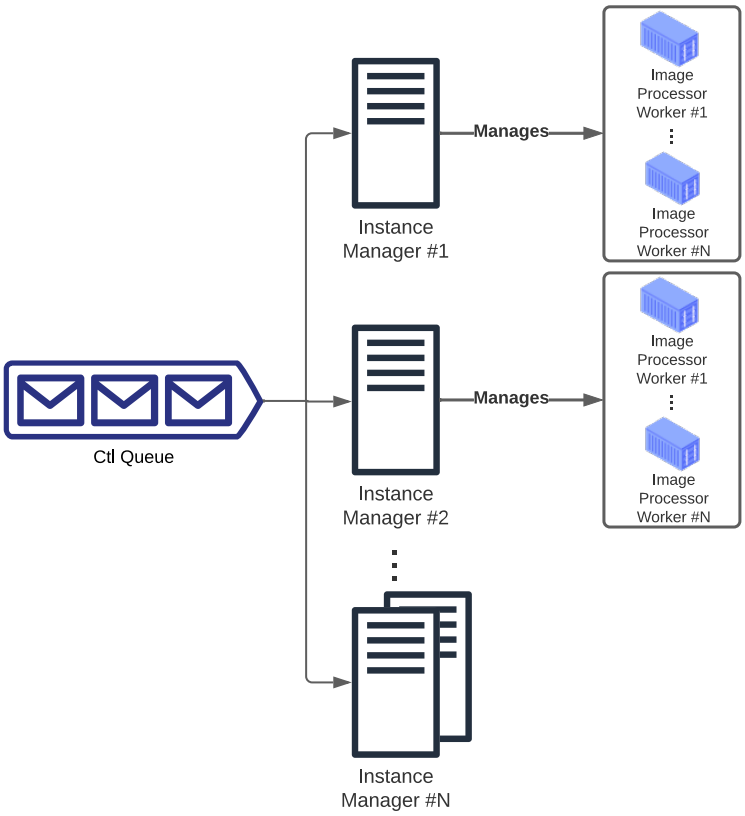

Figure 5.3: Worker Pattern applied to our problem

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.

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
}
```

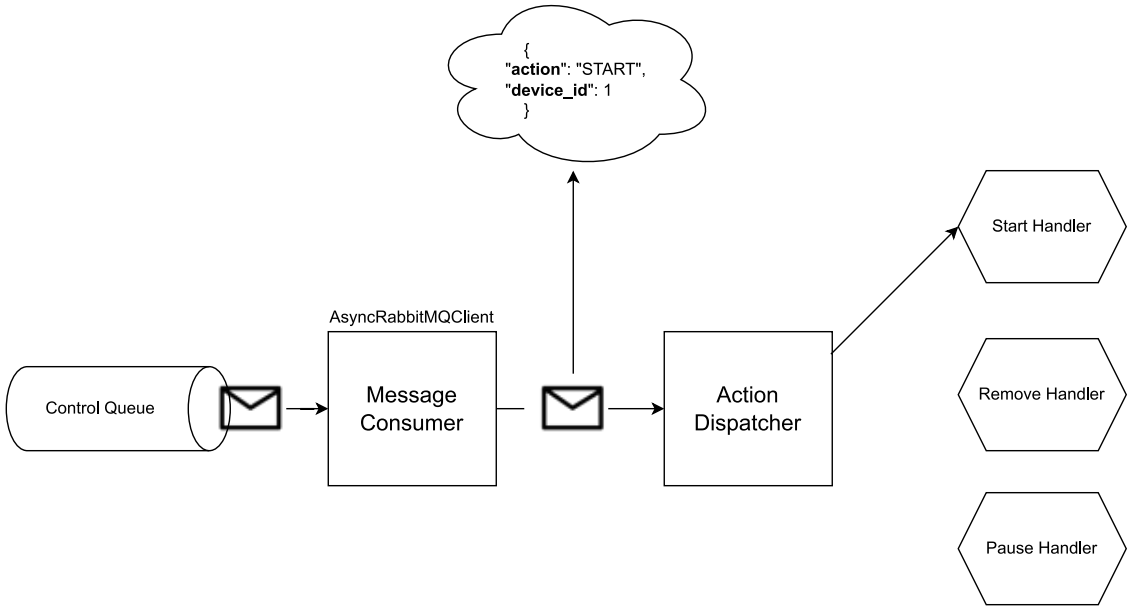

Figure 5.4: Dispatcher pattern

Diagram 5.4 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 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.

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 the figure 5.5.

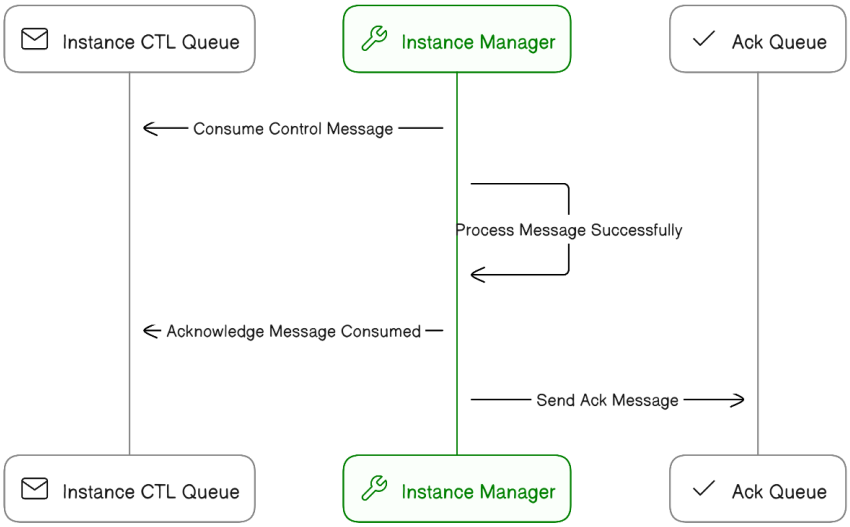

Figure 5.5: Case 1. Message processed correctly

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 5.6.

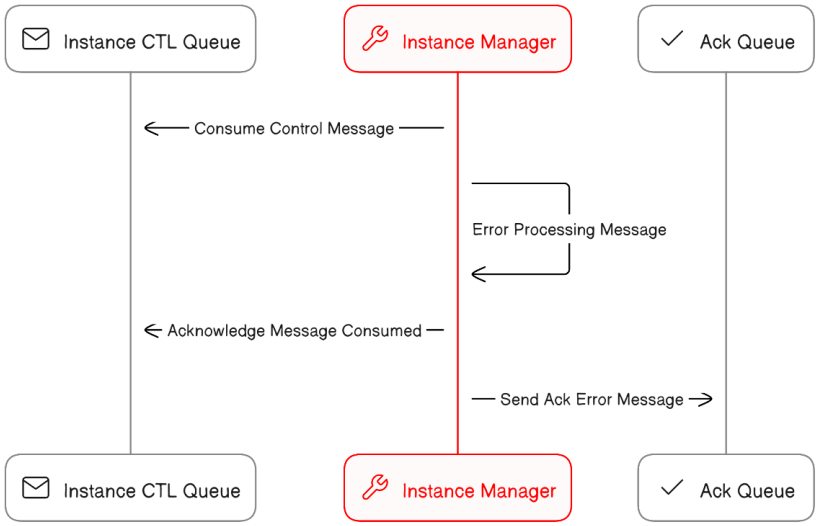

Figure 5.6: Case 2. Error but the message could never be processed

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 client 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.

To solve this problem, we implemented the use of Dead Letter Exchanges, a supported mechanism 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 the figure 5.7

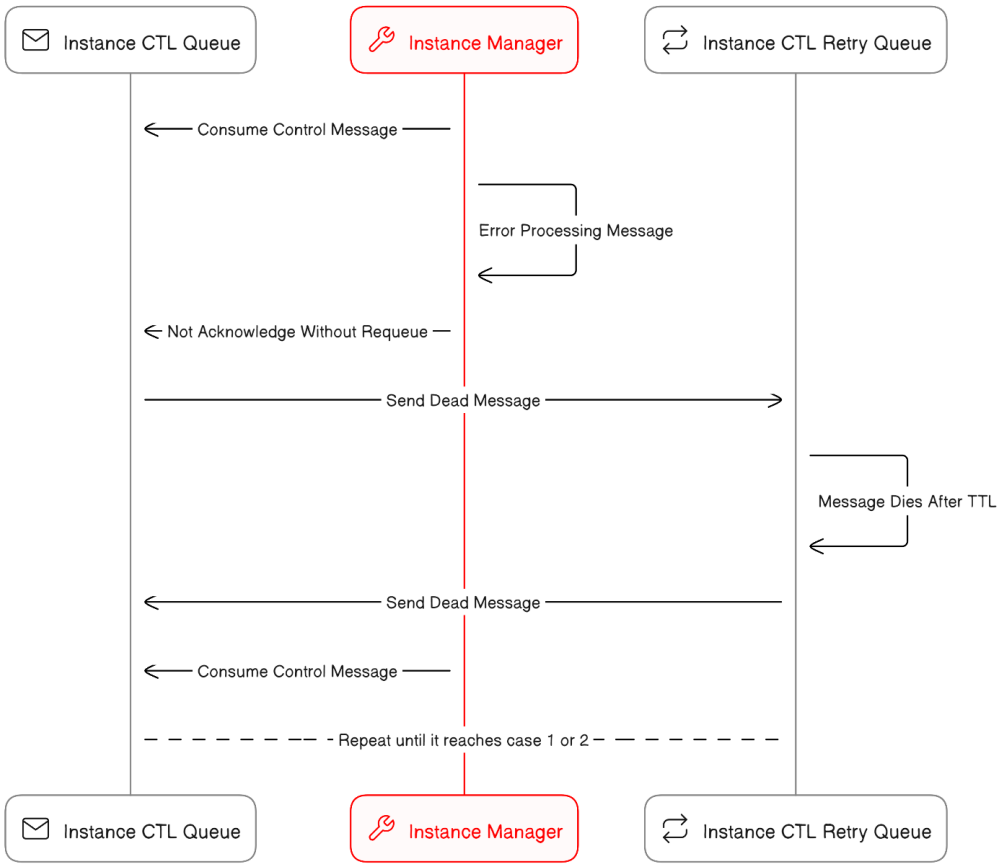

Figure 5.7: Case 3. Error but a message can eventually be processed

5.2.3 Environment and configuration

The *Instance Manager* is configured by environment variables and a configuration file.

The ENVIRONMENT environment variable is used to determine which configuration file to 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.

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. Each device can have up to 1 worker, 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.

5.4 Instance Manager Scheduler

This scheduler checks for long paused instances and stops them. It also checks for long stopped instances and deletes them. The scheduler component was built to recover from unexpected errors that might occur in the Instance Manager and to clean up some instances that might have been left in a PAUSED or STOPPED state, to free up resources automatically.

Chapter 6

Web Client Module

Contents

6.1 Structure

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

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 the 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.

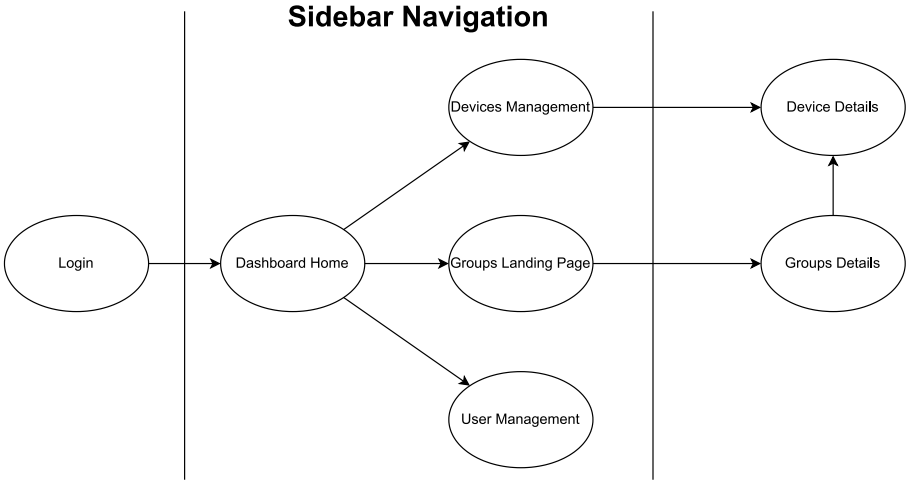

Figure 6.1: Web application navigation graph

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.

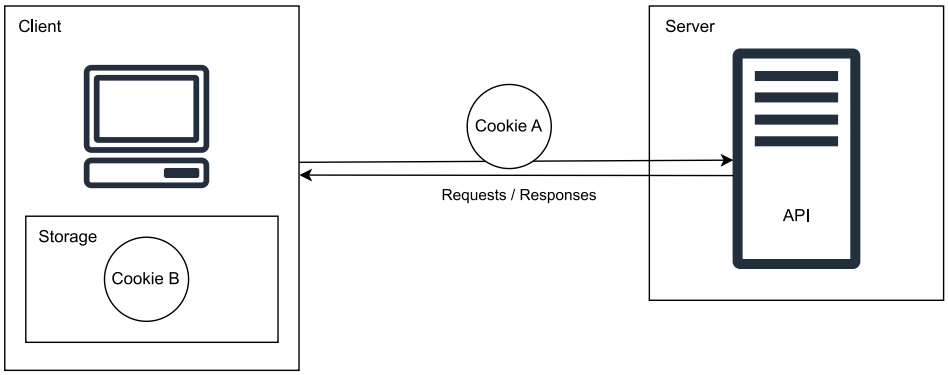

Figure 6.2: Web client side authentication

6.1.3 React

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.

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 prop that contains the data to be available.

Given this, two custom context were made: *CurrentUserContext* and textAuthContext.

For each context, a custom hook was also made to simplify the use of the mentioned contexts. The hooks just return the current context value, as given by its context provider.

CurrentUserContext 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.

A custom wrapper for its provider was made, this wrapper is responsible for storing the current user's information in the browser's local storage as previously mentioned in the Authorization and Authentication section on 6.1.2 this is done with the help of the *useLocalStorageState* hook, from an external library, this hook's job is to make sure the given state is persisted into localStorage using a given key. By storing the user´s information in the local storage, 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 this information to stay stored the user must be authenticated, if its session expires, the user data will be deleted on the next time the user opens the

application.

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

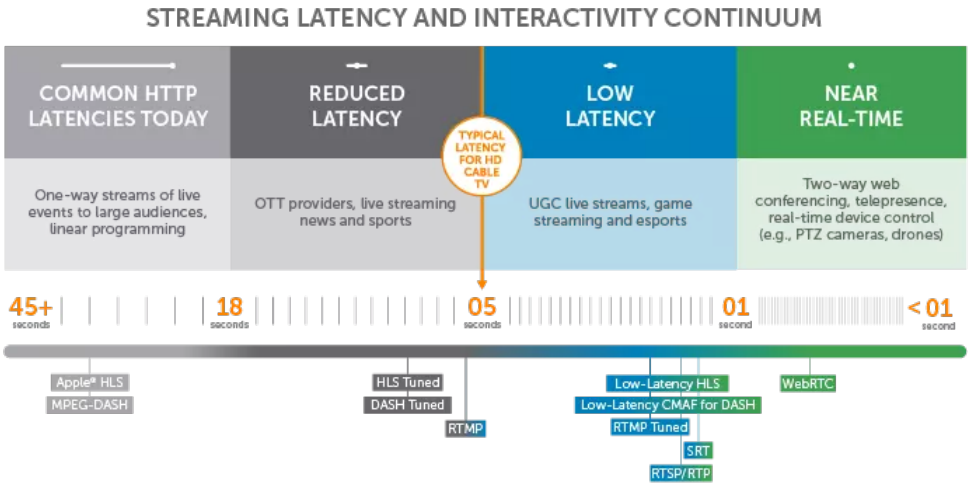

Figure 6.3: Average latency for each streaming protocol

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 , there were the two choices on what HTTP-based 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 for delivering video content over the web. However, HLS introduces a 2 second delay in the video playback due to its use of chunks of video files, which can be a concern if real-time viewing is required.

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.

5+

Chapter 7

Conclusions and Future Work

Contents

7.1 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.

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.

7.2 Future Work

Having in mind that the 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
- Improve the web application's overall user experience and interaction

References

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

Bluenviron (2023). Media mtx server. URL: https://github.com/bluenviron/mediamtx.

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

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

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.

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

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

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.