PiSentry - IP Camera with Real-Time Object Recognition

Daniel Roduit

Haaga-Helia University of Applied Sciences
Bachelor of Business Information Technology
Research thesis
2024

Abstract

Author
Daniel Roduit

Degree
Bachelor of Business Information Technology

Report/Thesis Title
PiSentry - IP Camera with Real-Time Object Recognition

Number of pages and appendix pages
44 + 9

The market for connected objects is booming and the latter are taking up more and more space in our lives. Due to their ease of acquisition and use, we do not always realize their complexity and the high technology that these devices embed.

This thesis aims to learn more about the creation process as well as the functioning of connected objects through the development of a security camera system. The camera uses inexpensive and accessible hardware equipment and can be set up by any hobbyist with basic computer knowledge.

The development process was driven in accordance with the Agile methodology. The workload was broken down and planned in small work units. The tasks to be carried out were then organized using the Kanban system.

Constrained by full-time professional commitments, the author conducted the thesis from September 2022 to May 2024, 18 months of which were needed to complete the practical project.

The final product shows that it is possible to successfully set up a fairly complex operational system in view of the limited means used. The system is easy to deploy and requires virtually no maintenance once up and running.

In conclusion, the current system fulfills the basic functions for which one would want to use a security camera. The system could be further improved by adding more electronic components which would allow to reach a product competing with the best surveillance cameras currently on the market.

Key words
IP camera, object recognition, motion detection, Raspberry Pi, Internet of Things

Table of contents

1 Introduction 1
1.1 Personal challenge 2
1.2 Significance and benefits 2
2 Project overview 4
2.1 Objectives 4
2.2 Out of scope 4
2.3 Development methodology 5
3 Architecture 6
3.1 Hardware 6
3.1.1 Platform 6
3.1.2 Camera module 8
3.1.3 Cooling system 9
3.2 Software 11
3.2.1 Camera software 11
3.2.2 Media server 13
3.2.3 Backend API 14
3.2.4 Web application 17
4 Implementation 21
4.1 Live streaming 21
4.1.1 Sending the camera stream over the network 22
4.1.2 Receiving and transmuxing video streams on the server 24
4.1.3 Viewing the live stream 25
4.2 Area surveillance 26
4.2.1 Motion detection 26
4.2.2 Object recognition 30
5 Results 33
5.1 Problems encountered 33
5.2 Current limitations and possible improvements 36
6 Conclusion 41
6.1 Project assessment 41
6.2 Personal assessment 41
Sources 43
Appendices 45
Appendix 1. PiSentry database Entity Relationship model 45
Appendix 2. Web application – Settings page 46

Appendix 3. Web application – Camera settings page 47
Appendix 4. Web application – Rename camera page 48
Appendix 5. Web application – Notifications page 49
Appendix 6. Web application – Object detection page 50
Appendix 7. Web application – Detection actions page 51
Appendix 8. Web application – PWA splash screen on Android 52
Appendix 9. Web application – PWA push notification on iOS 53

Table of figures

Figure 1: Camera hardware components 6
Figure 2: Summary diagram of the interactions between the software bricks of PiSentry 11
Figure 3: Camera software file tree 12
Figure 4: Media server file tree 14
Figure 5: Backend API file tree 16
Figure 6: Web application – Home page 18
Figure 7: Live streaming process 21
Figure 8: Implementation of live streaming in the camera software 22
Figure 9: Implementation of transmuxing in the media server 24
Figure 10: Implementation of the live video player in the web application 25
Figure 11: Motion detection result 27
Figure 12: Artefacts cleaning process 28
Figure 13: Movement isolation process 28
Figure 14: Movement refinement process 29
Figure 15: Movement delimiting process 30
Figure 16: Object recognition result 30
Figure 17: Implementation of object recognition 31
Figure 18: Hierarchization of recognized object types 32
Figure 19: Custom implementation of `FfmpegOutput` class with exception handling 35
Figure 20: Image captured with infrared cut-off filter (left) and without (right) 37
Figure 21: Raspberry Pi Camera Module 2 FOV in Full HD in video (left) and photo (right) modes38
Figure 22: Visual of currently implemented detection areas and their JSON equivalent 39

Table of abbreviations

IoT: Internet of Things
IP: Internet Protocol
RAM: Random Access Memory
GB: Gigabyte
USB: Universal Serial Bus
Full HD: Full High Definition
CSI: Camera Serial Interface
API: Application Programming Interface
HTTP: HyperText Transfer Protocol
HLS: HTTP Live Streaming
RTMP: Real-Time Messaging Protocol
SQL: Structured Query Language
SSH: Secure Shell
NAT: Network Address Translation
PWA: Progressive Web App
FLV: Flash Video
FOV: Field of View
JSON: JavaScript Object Notation
PIR: Passive Infrared
GPIO: General Purpose Input/Output

1 Introduction

Over the past ten years and following the gradual advent of connected objects (Ericsson 2023), video surveillance cameras have become more and more present in our daily lives. In Switzerland for example, in 2016 public space was observed by more than 21,000 cameras (Radio Télévision Suisse 2016). This represented one camera per 400 inhabitants. Also, in 2020, England recorded a record number for a European country of more than five million cameras (Ratcliffe 2020). That is approximately one camera for every 13 people, more than 30 times the number in Switzerland. In 2024, with the constant evolution of technology and its democratization, this figure could probably only evolve.

Over time, these cameras have gained functionality in many areas. They are no longer content to simply film passively and no longer require large infrastructures to operate. The vast majority of cameras now have the ability to be controlled remotely, to film at night, to alert their owners when events occur or to recognize objects. For the most advanced of them, it is even possible to ask the camera to follow people's movements, perform facial recognition, detect specific noises, etc. Cameras have moved beyond the strictly imaging field to incorporate elements from the fields of computing, acoustics and even robotics. However, all this technology is not without consequences.

As cameras have become ever smaller, cheaper to acquire and easier to handle, their proliferation has exploded and unfortunately we did not have to wait long to see stories in the news of people making inappropriate or even criminal uses of them. In 2018, a couple was unknowingly filmed during intimate relations in a San Diego AirBnB. The owner of the apartment had hidden two cameras in the bathroom and one in the bedroom ceiling (Rivas 2019). In 2019, four people were arrested in Seoul for secretly filming around 1,600 guests in hotel rooms. Cameras had been placed in TV boxes, wall outlets and hairdryer stands (Paul 2019). Even more recently, in August 2022, a couple was filmed without their knowledge in an AirBnB in the city of Silver Spring. The couple found two cameras hidden in smoke detectors (Fox 2023). All these events show that cameras can now pose serious privacy problems. But let's not be fooled, because we must remember that cameras are ultimately just inert objects with no will of their own. It is above all the use made of it that counts. And when cameras are used for good causes, they can benefit society as a whole.

This is for example the case for cameras placed in landscapes such as ski resorts, mountain villages or Alpine peaks and accessible from the internet. These allow anyone and from anywhere in the world to see the weather conditions on site, to witness the life of the locals during parties, to discover new places to go on vacation, etc. We can also note cameras placed around animal habitats to

observe their habits and learn more about them, or at home, to keep an eye on our pets when we are away. Cameras have countless benefits to offer - they just have to be used properly.

1.1 Personal challenge

This work is the result of my strong interest in the fields of computer science and electronics. Anyone with a passion for computers knows that it naturally drives us to always seek to create new projects and explore new technologies. If we take this passion and mix it with a nascent but not least interest in the world of microcontrollers and sensors, it gives birth to what is called the Internet of Things (IoT).

The term "Internet of Things" refers to all physical objects that have the ability to exchange data with other systems through the Internet network. You have probably come across such objects during your last shopping trip to the mall, perhaps even without realizing it. Robot vacuum cleaner, thermostat, light bulb, shower head, plug and switch, key ring, scale, pet food dispenser, watering system, indoor vegetable garden, bicycle, security camera, the list of connected objects could go on.

Today more than ever, these objects are taking a growing place in our homes, in our companies, in our cities and are giving birth to new and exciting use cases every day. In view of their attractiveness but also of the concerns they can generate, rightly or wrongly, I am curious to learn more about the functioning and development of these objects. As they say, or at least as I believe, one learns best by doing. To introduce myself to the world of IoT, I decided to develop a surveillance camera.

A surveillance camera is an ideal and stimulating product to make a first experience in the field of IoT because it gathers very well the computer science and electronic fields in a single object while leaving the possibility of prioritizing one or the other as desired. Indeed, a modern surveillance camera suitable for home use, which can be purchased in any electronics store, is nowadays closer to a multi-function gadget than a simple camera.

1.2 Significance and benefits

I consider this work not only as the accomplishment of three years of study, but also as a real work situation that mixes both existing and new knowledge. I know that the skills acquired during the development of the project will serve me both on a professional and personal level. Whether through a job or a hobby, I have the desire to pursue my journey in the world of IoT and this project is the foundation stone.

As my work is intended to be freely available and used, by myself and hopefully by others as well, I wanted to give my full commitment to make my finished work usable and best represent my skills as a developer. I believe that nothing is a better resume than a functional project, cared for and maintained by its developer. The quality of my work is therefore of high importance to me.

2 Project overview

2.1 Objectives

This project has the ambition to create a simplified version of a modern surveillance camera which retains the key features that make it an attractive and useful product for any household. The type of camera we are aiming to develop is specifically an IP camera, meaning a camera that can be reached at any time and from anywhere via the internet.

The camera should be able to recognize certain types of objects, in particular people, vehicles and animals, and react to their detection. The reaction should be implemented in the form of automatic video recording of the moment when the object was detected as well as in the form of a notification that alerts of the detection.

The camera's live video stream and detection recordings should be viewable via a web application. This latter should allow to configure the camera's behavior regarding at least object detection and notifications. The precise parameters will depend on what may be of interest to a user and above all what is possible to do, whether in terms of technique or time constraints.

This written thesis serves to document and explain the development and the functioning of the main features of the camera. For the covered features, I am thinking of the live streaming and the implementation of motion detection and object recognition. The goal of the thesis is to have at the end a project that works and can be used in real conditions.

2.2 Out of scope

The project focuses on the key functionalities of a surveillance IP camera which are related to the camera module. External and optional features such as lighting, speakers, cloud services, etc., are not included in the project objectives.

Also, the aspect of data processing and security, user authentication and data access authorization will not be covered. These themes, while important in a product or service intended to be publicly accessible and thus handling other people's data, are less crucial for private use in a controlled environment and are also not essential to the proper functioning of the surveillance camera. Their development was therefore not planned within the framework of this work.

2.3 Development methodology

This project was conducted using the Agile methodology. Agile is a software development method open and reactive to evolving requirements throughout the development process. By continuously evaluating which tasks should be given priority, it puts more emphasis on moving forward quickly than on having perfect results on the first try.

This way of designing software breaks away from the more traditional Waterfall methodology, where requirements and features are thought through, defined and fixed once and are not expected to change during the subsequent development phases. It suits well this project considering the uncertainty over whether the technologies initially planned would be suitable and sufficient for the tasks or would need to be changed along the way. Not having good prior knowledge of the subject, and therefore unable to gain an overview of the development process in advance, Agile was the natural choice to make.

Another positive aspect considered in choosing the Agile methodology is the possibility of using the Kanban system. The latter enables the workflow to be organized visually using a board on which tasks are placed and then moved from one state of progress to another as they are developed. This is a very efficient way to always keep an overview of the completed and remaining tasks and thus be able to better assess deadlines for each task.

3 Architecture

3.1 Hardware

This section addresses the hardware used to make the camera. More specifically, it focuses on the central components which, if changed or missing, would directly impact the quality or operation of the product. The hardware is the tangible part of the project. It naturally comes with physical and technological limitations that condition what can be done. It is therefore important to carefully consider the resulting implications before making a decision. Let's review the thoughts that led to the choice of the components.

Figure 1: Camera hardware components

3.1.1 Platform

To help me choose the platform on which I was going to base all my work, I thought about making a list of requirements. It was important that the platform not only met the required technical criteria but also that it was in line with the vision of the project.

In this context, the minimal technical requirements came down to a device powerful enough to support a significant workload, capable of connecting to the internet, having an interface to connect a camera and offering enough storage space for the photo and video files produced. In terms of accessibility, and to respect the initial idea of a project that could be reproduced by an amateur, it was desirable for the chosen platform to be well-known, well-documented and easy to use.

This is precisely where the Raspberry Pi comes into play.

Renowned for being versatile and suitable for a wide range of applications, the Raspberry Pi is used for both hobby projects and commercial products and services. It is very popular among individuals such as students and makers as well as in the industry. And its success is easily explained in view of its value proposition.

In its version 4 Model B, which is the one used in this project, it ticks all the boxes in the list of criteria. Among all the technologies it embeds, we can cite among others a powerful chip capable of running a full 64-bit operating system, RAM memory up to 8GB, plenty of storage capacity by default using a simple Micro-SD card, as well as numerous connectors making it easy to plug in various peripherals, including the camera in our case.

Although certain other systems also have technical specifications that could come close, I am thinking in particular of microcontrollers such as the ESP32 and its variants or even the Arduinos, both of which are very well known and enjoy a good reputation in the world of IoT, they just do not play in the same category. These development boards are primarily designed to be integrated into more complex solutions where interfacing with other components is largely done manually. They are intended to be small, portable and as energy efficient as possible. They do not seek to satisfy every need on their own. In this sense, they do not at all offer the same all-in-one, plug-and-play package as a Raspberry Pi. The user experience and development process are incomparable.

The Raspberry Pi is a single board computer. It can be used both graphically and from the command line. It can be remotely accessed, which is very practical for debugging a program without having to physically reach the device, if it has been placed in a location that is difficult to access for example. Its operating system is derived from Linux, which allows it to support by default many other programming languages in addition to C and C++. This is a considerable advantage, enabling as many people as possible to develop software on the Raspberry Pi using knowledge already acquired in other languages.

All these factors have a significant impact on productivity and thus weighed in the balance when making the choice. A platform like the Raspberry Pi allows you to spend as little time as possible on the technical set up of the device and to fully concentrate on developing your own software, and that is exactly what I was looking for.

3.1.2 Camera module

The camera being literally the heart of the project, I wanted to make sure that it delivered good image quality that was up to the standards of a modern security camera. To achieve this, the main criterion was a camera that could film in Full HD resolution at 30 frames per second.

Due to the use of the Raspberry Pi for the platform, which provides several different connectors, two types of cameras were possible. The first is a USB camera. This is the most common type for webcams available on the market because USB is a versatile interface that is widespread and available on a huge number of devices. Nowadays, it can be found on all computers, whether laptop or desktop, and even on single board computers. The second type is a CSI camera. This one is more commonly used on embedded and mobile systems, so it is by no means as well known to the general public as USB.

From there, I started doing some research to determine the best choice to make, and I realized that the decision should perhaps not be based on the hardware interface used but rather on the software support available for each type of interface.

Indeed, at the beginning of 2022, the Raspberry Pi Foundation released the picamera2 library, which is their Python library specially designed for Raspberry Pi cameras. It offers a high-level API for interacting with the camera. Being an official project of the foundation, it has first-class support in the ecosystem. Exhaustive documentation as well as numerous example code snippets are made available. Furthermore, the library is continually updated to add new features or resolve bugs reported by the community. All of this made it an ideal library to use for developing our project.

However, although the library supports Raspberry Pi cameras very well, which are CSI cameras, this is not the case for USB cameras. As explicitly stated in its documentation, picamera2 has limited support for USB cameras and users should not expect all features to be available (Raspberry Pi Ltd 2023). This therefore complicates the use of a USB camera. Worse still, to my knowledge there is actually no library like picamera2 tailored to USB cameras. It is possible to interact with them using a library such as OpenCV, but many features provided out of the box by picamera2 would then have to be implemented manually.

With that in mind, I opted for the stability and security offered by an official Raspberry Pi camera. The last camera released at that time was the Raspberry Pi Camera Module 2. It has an 8-megapixel sensor and can film in Full HD resolution at 30 frames per second, which exactly meets our quality criterion. I was able to get it for 28.85 CHF.

Looking back, I am convinced that this was a wise decision that allowed me to concentrate my efforts on developing the project's functionalities rather than loosing time solving interoperability issues with third-party hardware. That being said, I remain aware that making the project work with any camera would be an undeniable asset. I would easily understand if people preferred to use an old USB camera lying at the bottom of a drawer rather than having to buy specific equipment, myself included. So, this is a compatibility that I would like to develop in the future.

3.1.3 Cooling system

The addition of a cooling device was not a deliberate choice to improve the comfort of the Raspberry Pi. It turned out to be a necessity for the proper functioning of the project, at least at the current stage of development. I found myself forced to add one because the camera program requires a lot of resources, and a heavy workload means a lot of heat production. The only way to keep the Raspberry Pi within a temperature range below the throttling threshold of 80 degrees Celsius (Alasdair 2023) was to install such a device.

The cooling system that was installed is the official one for the Raspberry Pi 4, produced by the Raspberry Pi Foundation. It is composed of a heat sink and a fan.

The heat sink is a passive system. It is stuck to the processor and plays a heat transfer role. Thanks, among other things, to its materials, its design and its large surface area, it conducts and dissipates heat into the air much more efficiently than the processor alone. The fan, on the other hand, is an active system. It is placed in the case and allows the air to be renewed by drawing hot air out through the gaps in the case. The heat sink and the fan each play different but complementary roles in dissipating heat.

There are many other cooling solutions for the Raspberry Pi but this one appealed to me mainly for two main reasons.

First, its simplicity. Being an official product, it is natively supported in the operating system, therefore very simple and quick to configure, and it can be directly integrated into the official case without any modification. The installation process is also very well documented, which gives confidence in the

product. The second reason is its low cost. I was able to get it for as little as 4.45 CHF, where other systems, although sometimes more efficient, can cost two to four times more. So it was a very reasonable price to test the solution, and if it had not met the needs of the project, that would not have been a big deal.

Unfortunately, such a system also involves dealing with the potential nuisance it may cause, specifically from the fan. The latter, due to its high speed, produces a dull and high-pitched noise which can be annoying to hear. If the Raspberry Pi is completely enclosed or placed in a poorly ventilated area, the processor will quickly heat up and the fan will often be in operation. Fortunately, the latter still has the merit of fulfilling its function effectively, as the processor quickly drops from 75 degrees Celsius, which is the temperature set for the fan to start up, to below 65 degrees Celsius, before rising more slowly again. So there are more moments when it is not running than when it is. What is more, as the camera is most of the time intended to be installed outdoors, the noise can be barely heard, if at all, from indoors.

3.2 Software

The software is just as important as the hardware because it is what brings the project to life. While the hardware components are responsible for providing raw functionality, the software layer is essential to take full advantage of them. Unlike hardware, which is very static, software can be seen as the dynamic part, which applies a logic and allows to react differently depending on the situation. It is thanks to software that we can really build a custom solution, even using the same components. Let's take a look at the different parts that make up the PiSentry project, how they are designed and interconnected.

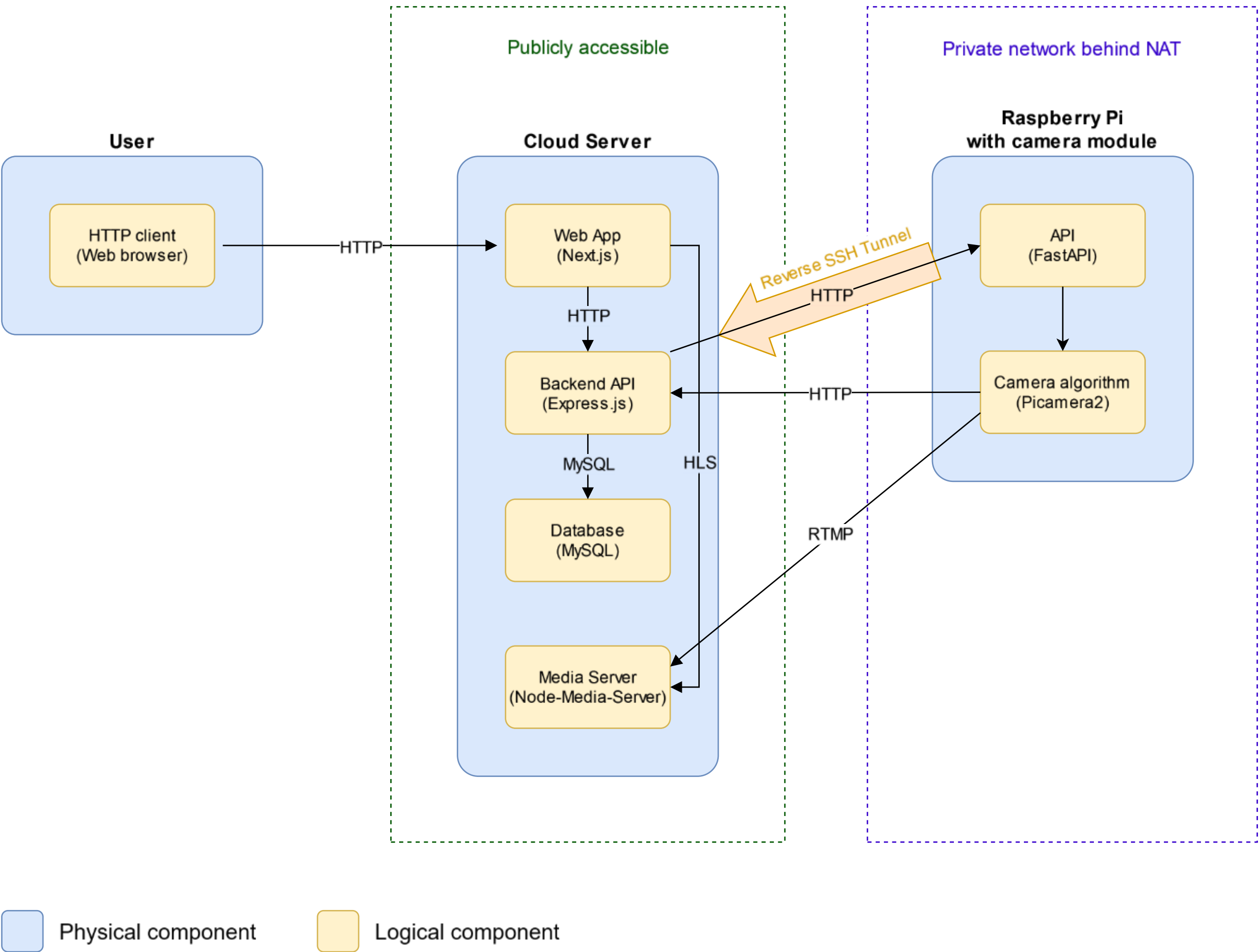


Figure 2: Summary diagram of the interactions between the software bricks of PiSentry

3.2.1 Camera software

The camera software is the centerpiece of PiSentry. This is the project that runs on the Raspberry Pi and acts as an interface with the camera hardware and all its peripherals. It is entirely programmed in Python.

As can be seen in the right-hand column in Figure 2, in the *Raspberry Pi with camera module* box, and also in the project folders in Figure 3, the camera software has two sub-parts, each with very distinct tasks. These are the API and the camera algorithm.

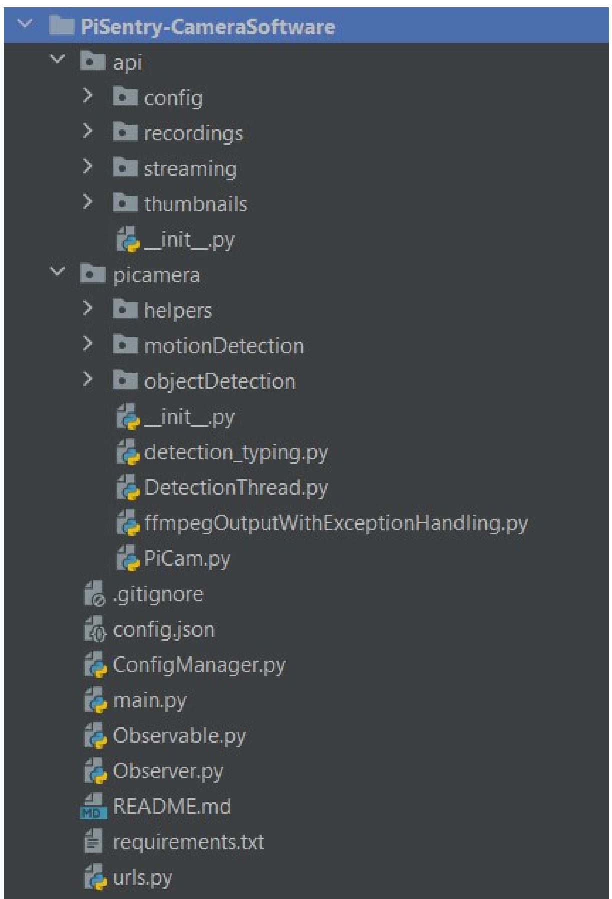


Figure 3: Camera software file tree

First, the camera algorithm, as its name implies, handles everything related to the camera, and therefore the camera module in our case. It is responsible for publishing the live stream, performing motion detection and object recognition, recording videos, taking photos, save events in the database, notify a new detection and so on. When it needs to send the live video stream, it establishes a direct connection with the media server. The subject of live streaming will be covered in more detail in the next chapter on implementation in section 4.1. Similarly, when it wants to interact with the database or trigger a notification to signal a new detection, it sends requests to the backend API.

To carry out its numerous and diverse tasks, it uses several libraries and frameworks. These include, among others, Picamera2 for managing the camera module, OpenCV for image processing and FFmpeg for video processing. Picamera2 was chosen because it is, as already stated previously, the Raspberry Pi Foundation's official library for interfacing with Raspberry Pi camera modules. It offers all the features needed to easily configure and interact with the camera module, making it one

of the most necessary and used libraries throughout the project. Then, OpenCV was chosen because it alone allows both motion detection and object recognition to be implemented, not to mention the other additional functionalities used from time to time throughout the project. It is an extremely versatile library. Finally, the FFmpeg framework is used to both stream the live video feed and encapsulate the raw H.264 video recordings produced by the camera in a more convenient MP4 container.

Second, the API, acts as an intermediary between the camera and other projects. It can be seen as the camera software's secretary. It receives requests from other projects, in our case mainly from the backend API, and depending on the request sends back data or instructs the camera algorithm to perform actions. Figure 3 shows that the `api/` folder contains several other folders. These represent the available endpoints. Here is the function of each of them:

- `/config`: reload configuration parameters stored in database.
- `/recordings`: return video recordings, start and stop recording on demand.
- `/streaming`: start and stop live streaming.
- `/thumbnails`: if necessary create, then return, the thumbnails of the videos and the live stream.

The API was developed using the FastAPI framework and runs on the Uvicorn web server. Although FastAPI is by no means the only solution for developing APIs in Python, it has the merit of communicating successfully. Its documentation is clear and neat and is really focused on API development. This is particularly what pushed me to choose it over the Flask framework. Its simplicity and its specialization in APIs also made it preferable to a framework like Django, which is too feature-rich for this use case.

3.2.2 Media server

The media server is the element that makes it possible to view the live stream regardless of the browser or device used. In the interaction diagram (Figure 2), it is represented as an intermediary between the camera algorithm and the web application. Its role is to act as a bridge between them with the smallest possible footprint. The less you notice it, the better. It receives the live video stream sent to it by the camera and transforms it, in real time, into a new live video stream that the web application will be able to display in the browser. Indeed, the video stream coming from the camera cannot be directly displayed in a browser, and this is due to the video format and transmission protocol chosen for the camera’s live stream. Of course, this choice was not made without thought, and the reason for it is explained in detail in section 4.1.

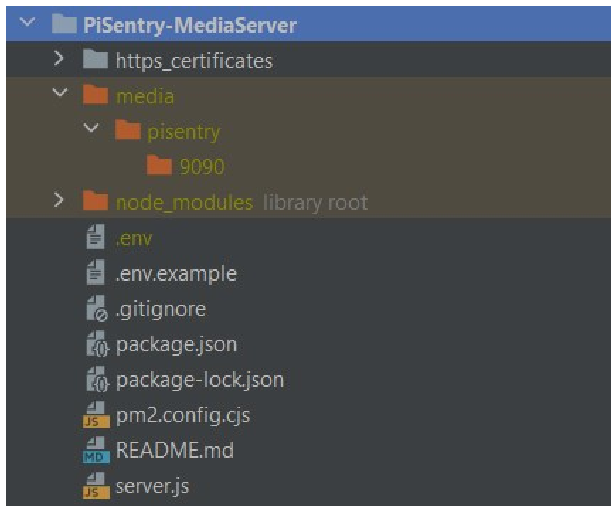


Figure 4: Media server file tree

As can be seen in the file tree in Figure 4 above, there are very few files in the project. The only items of interest are the `server.js` file, which contains the code to configure and launch the media server, and the folder tree under the `media/` folder, which is used to temporarily store the newly created live video stream. The `pisentry/` subfolder represents the project under which the live stream is received, and the `9090/` subfolder represents a specific live stream. All other files are configuration files.

The media server project is developed entirely in JavaScript. It uses a single module, called Node-Media-Server. The latter is a media server implementation for the Node.js runtime. It is responsible for doing the heavy lifting when creating a new live stream from the source live stream. It can be configured to meet exact requirements in terms of the protocols used to receive and transmit video streams, the video format generated, and so on. It also includes a graphical interface to keep an eye on what is happening on the server and an API for interacting programmatically with the server. It is a media server that suits the basic needs of the project.

In our deployed architecture, the media server is hosted on a cloud server, which allows it to take advantage of good computing power for its operations and a large storage space for the live stream.

3.2.3 Backend API

The backend API is the project that takes care of inter-project communication and data management. In the PiSentry architecture, it fulfills several central roles.

First, just like the media server, it acts as an intermediary between the web application and the camera. It enables them to communicate and exchange data. However, unlike the media server, it

is not the camera algorithm that initiates the requests but the web application. In this case, the recipient is also no longer the web application but the camera API. The direction of the data flow is therefore reversed. The backend API receives a request from the web application and forwards it to the camera API, which then responds appropriately. In this case, the request can be used to ask the camera to return a resource, such as a video recording, or to perform a specific action, such as starting live streaming.

The intermediary mechanism may seem surprising at first glance, as it is legitimate to wonder why it is necessary to go through an intermediary API to ultimately transfer the request to another API. Why not shorten the path and talk directly to the target API? This architecture was chosen to facilitate access to the camera. Indeed, the camera is located in a private network that cannot be reached from the Internet. It is therefore necessary to establish a tunnel beforehand in order to communicate with it. In this context, it was then simpler to use the backend API as the single point of contact at the end of the tunnel that any project could use to communicate with the camera, rather than having to implement different logic for this task in every project that needs it.

Secondly, it has the role of data provider for all projects. It is the only one with access to the database. As a result, whether it is the web application or the camera algorithm, they both have to go through it to perform CRUD (Create, Read, Update, Delete) operations on the database. The web application uses it to retrieve data relating to cameras, recordings, live stream, configuration parameters and to modify the latter if necessary. The camera algorithm uses it to retrieve configuration parameters and add and delete recordings and detections from the database. Due to this role, it also performs numerous data checks and validations. It ensures that the data received is actually what it is supposed to be and returns the appropriate error if it is not.

Its third and final role is to send push notifications when new detections are made by the camera. The latter, via the camera algorithm, will send a request to the backend API so that it notifies this new detection to all browsers subscribed to the notifications. The API will then retrieve the subscriber data from the database and attempt to send them the push notification via their dedicated push server.

The backend API is entirely programmed in JavaScript. It mainly uses three libraries and frameworks to get the job done. The first framework is Express.js, used to implement the API itself. Next is the MySQL2 library, which takes care of the connection and interaction with the MySQL database. Finally, the Web-Push library, which simplifies the configuration required to send push notifications.

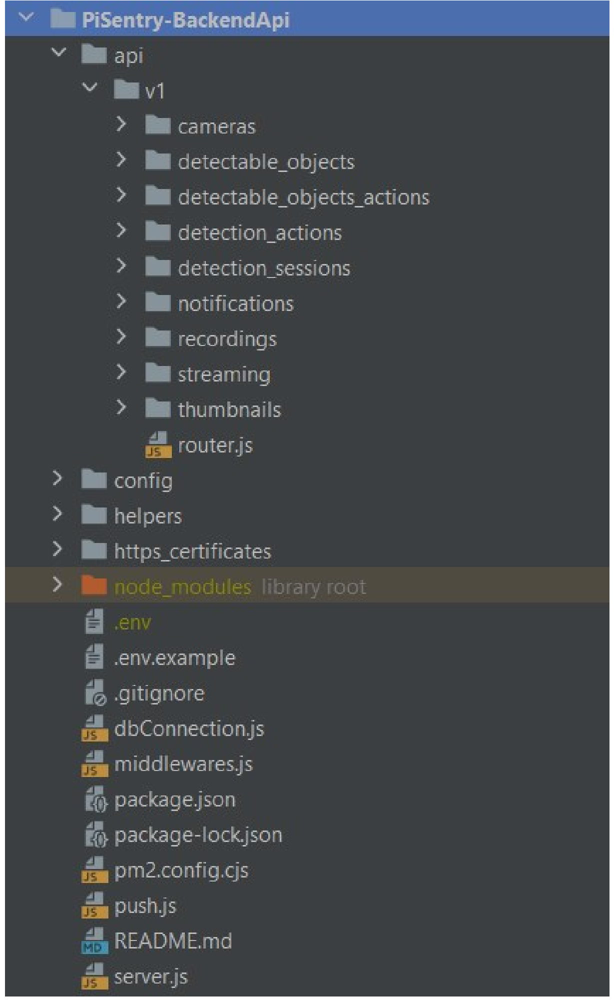


Figure 5: Backend API file tree

As can be seen in the project file tree in Figure 5, the API is versioned, with currently only one version and identifiable by the `v1/` folder. This folder itself contains several sub-folders, each of which represents a callable endpoint. Here are their functions:

- `/cameras`: returns or modifies camera data. Used by the web application and the camera algorithm.
- `/detectable-objects`: returns the types of objects to be recognized by the camera algorithm. Used by the web application.
- `/detectable-objects-actions`: returns or modifies the actions to be performed depending on the types of objects recognized. Used by the web application and the camera algorithm.
- `/detection-actions`: returns the list of actions which may be performed in response to a detection. Used by the web application.

- `/detection-sessions`: creates detection sessions, that is to say groups of several detections. Used by the web application and the camera algorithm.
- `/notifications`: subscribes or unsubscribes a browser to push notifications, returns subscription data, sends a push notification to a subscribed browser. Used by the camera algorithm.
- `/recordings`: creates, deletes or returns recordings data, returns video recordings. Used by the web application and the camera algorithm.
- `/streaming`: ask to start or stop live streaming. Used by the web application.
- `/thumbnails`: returns video thumbnails. Used by the web application.

Among all the endpoints, there are three that serve the intermediary role mentioned earlier, namely `/recordings`, `/streaming` and `/thumbnails`. Those are used at the very least by the web application and can also be found in the camera API under the same exact name. Of course, acting as intermediaries is not necessarily their only purpose, as can be seen with the `/recordings` endpoint, which is also used by the camera algorithm.

3.2.4 Web application

The web application is the visible face of PiSentry. It is the platform that allows users to interact in a simple way with the complex underlying system. Simply put, this is the front end of the project.

The frontend is an important element because it is, in a way, the business card of the project. Users often judge an entire project solely by what they see of it. If the frontend is neat, pleasant to use, functional and logical, then they will form a positive opinion of the project. They will want to use it, recommend it to others, and might even consider paying for the service. In the opposite case, that is to say if the frontend is disorganized and complicated to use, then the user will not want to waste time with it, even if the underlying system works. This made the frontend a part that required a lot of thought throughout the development process. Choices had to be made regarding the user interface, the user experience and the functionalities available in the interface.

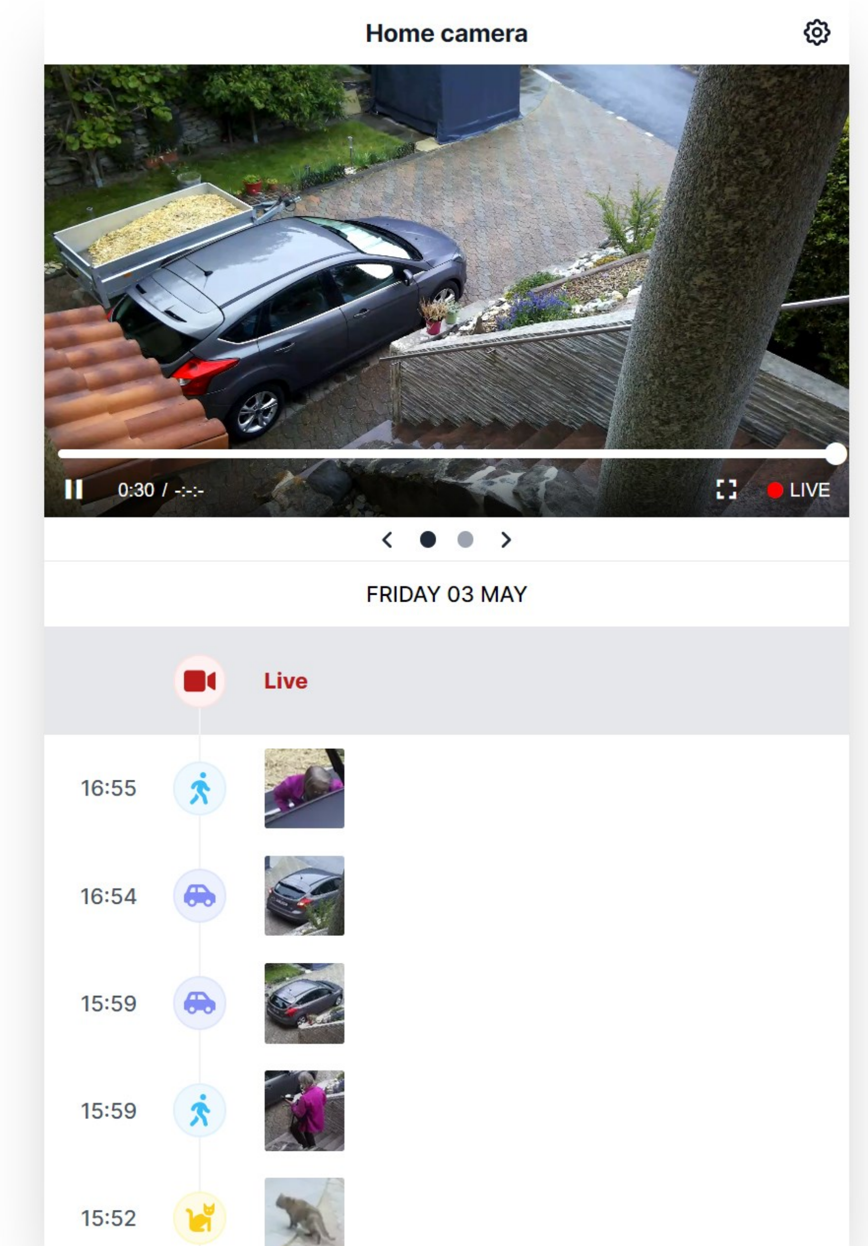


Figure 6: Web application – Home page

Currently, the web application allows the user to do three things.

First, it allows viewing the camera's live stream. To do this, the web application asks the camera, as always via the backend API, to start live streaming. Then, thanks to the media server that receives the live video stream from the camera and generates a live stream in a format supported by browsers, the web application can retrieve this new live stream and display it in a video player. The display of the live stream in the player can be seen at the top of Figure 6.

Second, the web application provides access to the videos recorded following the detections. This is done in three steps. The web application begins by asking the backend API to return the list of

available recordings which is stored in the database. Once obtained, it displays the recordings sorted by date and ordered chronologically from the most recent to the oldest. When the user clicks on a recording, the web application then only has to ask the camera to return the chosen recording and display it in the video player. The list of recordings can be seen below the video player in Figure 6.

Finally, the web application enables the camera's behavior to be configured in real time. For the time being, three settings can be modified and each of them has a dedicated page in the application. A visual of each page is available in Appendices 4 to 7.

First of all, it is possible to rename the camera (Appendix 4). The name of the camera, in addition to being displayed above the live video player to identify which camera the live stream is coming from, also serves as the title of push notifications. This is very practical, especially when managing several cameras, to be informed of which camera had done the detection without even having to open the application.

The second setting concerns notifications (Appendix 5). These can be enabled or disabled. However, it should be noted that the activation state is specific to each browser, which means that a user who activates notifications in the PiSentry application on their smartphone will not automatically receive notifications in the PiSentry application on their computer. Each browser is represented by a unique push endpoint. If notifications are enabled, it is also possible to define a time range during which notifications will be received. No notifications will be sent outside of this range. This can be useful to avoid being disturbed during working hours, for example.

The third and final setting relates to object recognition (Appendices 6 and 7). Here again, it is possible to define a time range during which the detection process, which is motion detection followed by object recognition, is active. This range was put in place because the camera does not currently support night vision and therefore does not see anything throughout the night. There is therefore no point in continuing the detection process in these conditions. Defining the range of activity only during the day when the camera has enough light to see avoids giving the camera unnecessary work during the night. However, when the camera supports night vision, the existence of this range could be called into question if there is no other good reason to pause the detection process. Normally, the whole point of a security camera is to be able to monitor 24 hours a day.

In addition to the activity range, the object recognition setting also offers the option of choosing, for each type of object, what action should be performed after the object has been recognized. Three actions are available. The first is to simply ignore, so nothing will happen when the object is recognized. The second is to make a video recording of the recognition. We will therefore be able to find

the video of the moment when the object was recognized in the application. The third is to make a video recording and notify the user of the recognition via a push notification.

The web application has been developed as a Progressive Web App (PWA), which gives it additional capabilities compared to a traditional website. In our case, to make it more practical to use on a day-to-day basis, we made it installable like any native application on operating systems that support PWAs (Appendix 8). Similarly, push notifications are a core feature that we took advantage of and without which the project would not have had the same usefulness. It is truly the feature that makes PiSentry real time (Appendix 9).

In terms of the technologies used, the web application was developed in JavaScript using the Next.js framework. The latter is a modern, fully featured framework that greatly accelerated the development process. It was chosen in particular for its management of client-side rendering and server-side rendering, which makes it flexible, powerful and pleasant to work with. What is more, as I already had some knowledge of the React library, it was more appropriate to choose Next.js, which relies on React, rather than other full-stack frameworks such as Nuxt or SvelteKit, which rely on other libraries.

On the design side, the CSS framework Tailwind CSS was used to design the interface, and the result speaks for itself. It is a library that becomes essential once you have tried it because it tremendously simplifies and speeds up styling.

As far as data is concerned, the SWR and Redux Toolkit libraries were used respectively for some data fetching and state management on the client. SWR was created by the same team that develops Next.js and therefore benefits from first-class support. Redux Toolkit is widely known for its centralized client-side state management model.

Finally, the Video.js framework is used as the video player to play the live stream and recordings. Video.js stood out mainly for the fact that it is open source, for its quality documentation and guides, and finally for its numerous customization possibilities. This makes it a highly scalable video player that is well suited to our use case.

4 Implementation

This chapter is dedicated to the technical implementation of the core features of the PiSentry project. It presents step by step, in a concrete manner using real code snippets and commands, the inner workings of each feature. When relevant, the different paths that were explored during development and the resulting decisions are also discussed and explained thoroughly. Let's dive in.

4.1 Live streaming

Live streaming can be described as sending content in real time allowing that same content to also be played back in real time. In this sense, one could say that television or radio are examples of live streaming. Although most often video and audio, the content of a live stream can theoretically be of any nature.

Even though we are nowadays accustomed to consuming content in real time on video platforms such as YouTube for example, we often do not produce content ourselves. It is therefore not necessarily easy to understand how all this works in practice.

In the PiSentry project, live streaming represents a significant feature where we had to set up everything ourselves, that is to say playing the role of content producer as well as final consumer, including everything in between. For PiSentry, the live streaming process is as follows:

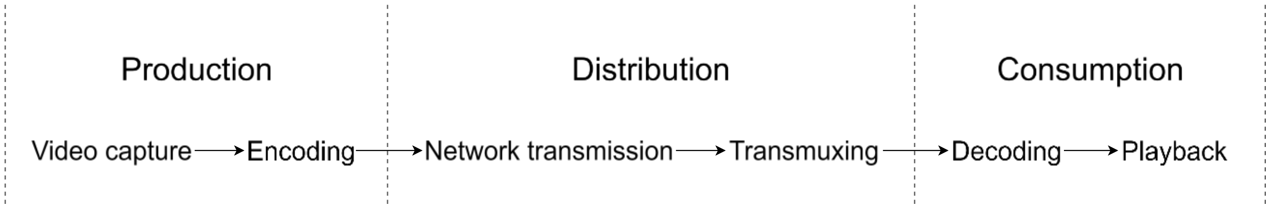


Figure 7: Live streaming process

This figure shows the successive steps involved in the live streaming process. It teaches us that live streaming is not as simple as just talking about producers and consumers. A set of intermediate operations are necessary for the proper functioning of live streaming as we know it.

It all starts with content production, that is, capturing images using the camera module. Once the images have been produced, they need to be encoded. This means processing them to output less voluminous video data and overall usable by the following actors in the process. Then, the video stream is transferred on the network. It is sent to the media server which will, based on this video

stream coming from the camera, create a new video stream. This is the transmuxing step. Finally, this new video stream can be decoded and consumed by video players.

4.1.1 Sending the camera stream over the network

The following piece of code (Figure 8) is taken from the camera software project. It contains the code needed to perform the first three steps of live streaming, from capturing the video to sending it to the media server. All operations are performed using the Picamera2 library.

PiCam.py

```
1 ...
2
3 class PiCam:
4     def __init__(self):
5         self._picam2 = Picamera2()
6
7         video_config = self._picam2.create_video_configuration(
8             main={"format": "RGB888", "size": (1920, 1080)}
9         )
10        self._picam2.configure(video_config)
11
12        self._encoder = H264Encoder(repeat=True, iperiod=15)
13        self._picam2.start_encoder(self._encoder)
14        self._picam2.start()
15
16        camera_port = configManager.config.camera.port
17
18        self._streamingOutput = FfmpegOutputWithExceptionHandling(
19            f"-f flv {mediaserver_publish_livestream_url}/pisentry/{camera_port}"
20        )
21        self._recordingOutput = CircularOutput(
22            buffersize=300
23        )  # 300 means 30 images/second * 10 seconds
24        self._encoder.output = [self._streamingOutput, self._recordingOutput]
25
26        ...
27
28    def start_streaming(self):
29        if not self._streamingOutput.recording:
30            self.stop_streaming()
31            self._streamingOutput.start()
32
33    def stop_streaming(self):
34        self._streamingOutput.stop()
35
36    ...
```

Figure 8: Implementation of live streaming in the camera software

First, lines 5 to 10 initialize and configure the camera. Then, on lines 12 and 13, an encoder is assigned to the camera. This is the unit that will perform the encoding task. As can be deduced from its name, `H264Encoder`, this encoder uses the H.264 video compression standard. The H.264 standard is the most widely used at the moment and makes it possible to generate high quality videos while keeping a reasonable file size (Cloudflare s.a.a). Although this encoder is not the only one provided by Picamera2, it was chosen because it is the only one that uses a compression standard supported by the HLS streaming protocol (Apple s.a.) that we use to play the live stream in browsers.

Further on, on lines 18 to 24, an output called `_streamingOutput` is created and then allocated to the encoder. An output is a destination to which the video stream is passed. What happens to the video stream once it reaches its destination depends on the output chosen. The video stream could be stored, sent over the network or even played back directly. In this case, the output is used to transport the camera's video stream to the media server.

As is apparent from the name of the output, `FfmpegOutputWithExceptionHandling`, the FFmpeg library is used as the destination. This means that it is FFmpeg that will receive the video stream and decide what to do with it. In our code, the command passed to FFmpeg asks it to send the video stream to the network address of the media server using the RTMP protocol. Unfortunately, the protocol prefix ”rtmp://” is not visible on the image, as it is written directly in the media server address. The command also asks to encapsulate the video to be sent in the Flash Video (FLV) container.

Transporting the video stream from the camera to the media server was not a random choice. As mentioned earlier, we ultimately need to use the HLS streaming protocol to be able to display the live stream in browsers. So why not generate an HLS live stream directly from the camera? Technically, the FFmpeg output would be capable of doing this. Why do we have to go through the media server?

In fact, it would actually have been possible to create the HLS video stream on-camera, and this is approach was tested. However, it quickly became clear that this was not a viable solution for the future. Indeed, in the case where the camera had itself produced the live stream for the browsers, in addition to having to generate the HLS video stream in real time, it would have had to store the resulting video segments, at least temporarily. This was not a desirable situation, as the camera needs storage space for the video recordings. It would then have been complicated to ensure that enough space was available to store the live stream as well.

To keep things simple, for the sake of efficiency and to avoid adding to the already heavy workload given to the Raspberry Pi, it was decided to delegate this responsibility to a specialized external server, the media server.

4.1.2 Receiving and transmuxing video streams on the server

The main purpose of the media server is to handle transmuxing, which is the fourth step in the live streaming process.

Transmuxing is an operation that involves changing the way data is arranged in a video stream. It is commonly referred to as changing the video format. In our project, this procedure is necessary because the camera's video stream is in FLV format. However, this format is not supported for video playback in browsers. We are therefore going to remedy this by transmuxing in real time the FLV video stream into an MPEG Transport Stream (MPEG-TS), which is a container format supported for HTTP Live Streaming (Apple s.a.).

server.js

```
1 import NodeMediaServer from 'node-media-server';
2
3 ...
4
5 const config = {
6     rtmp: {
7         port: process.env.RTMP_PORT,
8         chunk_size: 2000,
9         gop_cache: false,
10        ping: 30,
11        ping_timeout: 60,
12    },
13    http: {...},
14    trans: {
15        ffmpeg: process.env.FFMPEG_PATH,
16        tasks: [{
17            app: 'pisentry',
18            hls: true,
19            hlsFlags: '[hls_time=2:hls_list_size=20 \
20              :hls_flags=delete_segments+discont_start:hls_allow_cache=0]',
21        }],
22    },
23 };
24
25 ...
26
27 const mediaServer = new NodeMediaServer(config);
28 mediaServer.run();
29
30 ...
```

Figure 9: Implementation of transmuxing in the media server

In the code snippet above (Figure 9), we can see that the media server code is very short. This is because the Node-Media-Server library abstracts all the complexity of video processing for us. The most interesting part is the configuration, because this is what specifies all the actions to be carried out. All we have to do is add the configuration values to the configuration object and the server will understand that it has to start this or that service.

In the configuration shown, the media server is asked to do 3 things. The first, from lines 5 to 12, is to launch a server dedicated to receiving the RTMP stream. Options relating to RTMP stream management are passed to it. The second, on line 13, is to launch an HTTP server. This server provides access to a graphical interface for monitoring the operation of the media server. Finally, the third, from lines 14 to 23, is the transmuxing operation. We can see that a path to an FFmpeg executable is specified and that the value of the `hls` key is set to true. It is this part of the configuration that will create the video stream that can be displayed in browsers.

4.1.3 Viewing the live stream

Once the HLS video stream is made available by the media server, the web application simply needs to fetch it and display it in a video player. The video player is therefore responsible for the last two steps of the live streaming process, namely decoding and playback.

CamerasCarousel.jsx

```
1 ...
2
3 const renderCameraVideoPlayer = (cameraData) => (
4     <VideoPlayer
5         options={{
6             id: `${cameraData.camera_id}`, // id for videojs player must be a string
7             sources: {
8                 src: `${pisentryLivestreamEndpoint}/${cameraData.port}/index.m3u8`,
9                 type: 'application/x-mpegURL',
10            },
11            poster: `${thumbnailsApiEndpoint}/${cameraData.camera_id}/live`,
12        }}
13        onReady={(player) => onReadyVideoPlayer(player, cameraData)}
14        afterInstantiation={(player) => afterVideoPlayerInstantiation(player, cameraData)}
15    />
16 );
17
18 ...
```

Figure 10: Implementation of the live video player in the web application

The code snippet in Figure 10 shows the React component `<VideoPlayer/>`, which is our custom component created to wrap the logic of the Video.js framework. The component is straight forward to use, taking just three properties. However, only the `options` property is worth a closer look. This is the one used to specify the source of the live stream to be played.

We can see that the `src` key points to the address of a PiSentry live stream and retrieves an "index.m3u8" file. This file is not a video file but a plain text file. It contains information about the playlist, in other words the live stream, to be played. The video player will use the data in this file to retrieve the video segments of the live stream and play them back to the user. It is also thanks to this file that the video player is able to determine whether it is live, whether it has fallen behind, whether the live stream has been stopped, and so on.

The decoding and playback process is therefore entirely and seamlessly managed by the video player.

4.2 Area surveillance

The surveillance process may seem magical at first glance. Without any prior knowledge in this field, it is indeed complicated to conceive how a camera can identify objects. But in fact, there is nothing magical about it. The camera simply mimics what a human being would have done naturally if given the same task. If a human was asked to watch for something appearing in its field of vision and to determine what it is, let's say an intruder for example, it would proceed as follows: it would continuously observe the given scene, and if the event that we were asked to watch for occurred, that is the appearance of an unknown person, then it would react.

The camera will be programmed to work in the same way. It will observe the scene and then, to be able to know if the event has occurred or not in order to react to it, several algorithms will be used. This chapter explains how the camera, with its own mechanisms, can emulate the ability to understand the world around us that seems so innate in humans.

4.2.1 Motion detection

The first step of the surveillance process is motion detection. This one is special because it was not initially planned. In my mind, the only step that had to be implemented to recognize objects was the object recognition algorithm itself. For that matter, object recognition was initially the only part that had been developed.

However, a logical problem soon became apparent. If the camera only performs object recognition on the images it captures, it will certainly identify the objects present in the image, but it will not be able to distinguish between objects that were already there a long time ago and objects that have just appeared in the scene. This means, for example in the case of monitoring a parking space, that the camera will only be able to tell if there is a car in the space, but not to alert when the car is leaving or arriving. That is a shame, because that is precisely what we would like a security camera to do.

To overcome this problem, motion detection proved necessary. By setting up a process where motion detection is first carried out to see what has moved in the image, and then passing that information to object recognition, it is possible to tell for sure that a particular object was not there before and that it just appeared. As a result, motion detection was added to the project in the course of development. Let's see how it works.

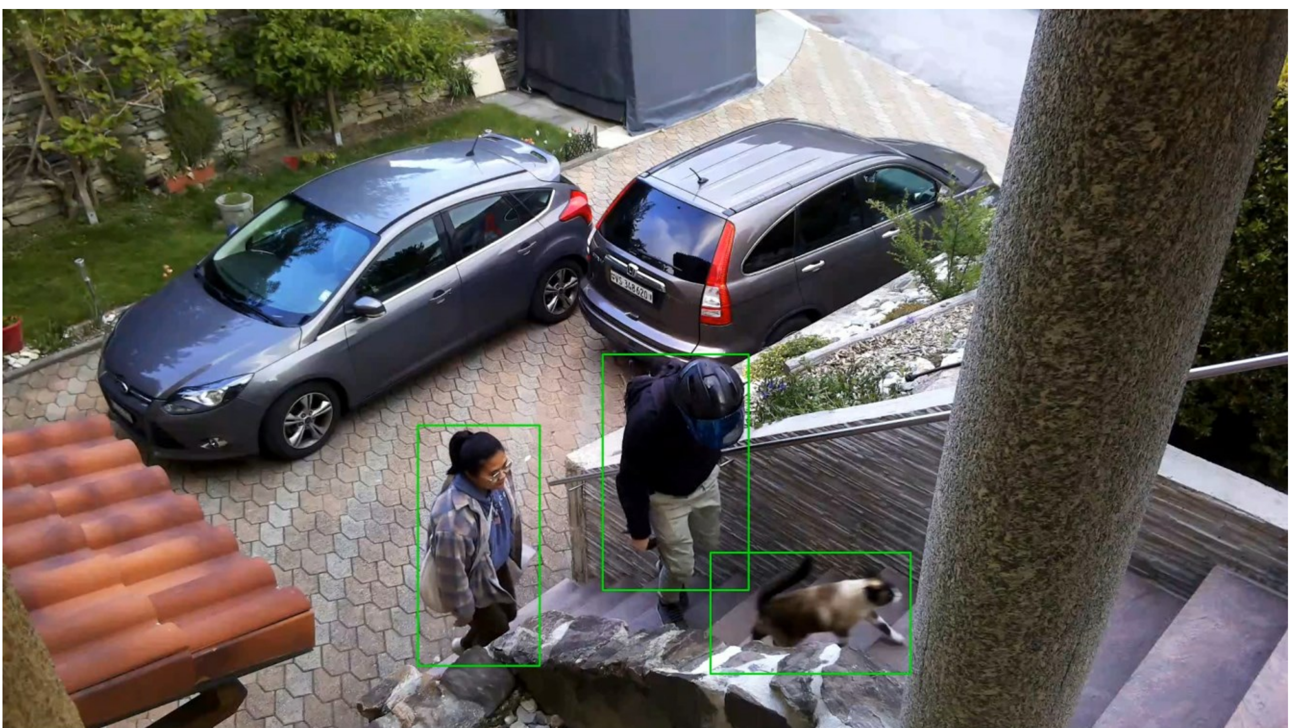


Figure 11: Motion detection result

Generally speaking, motion detection is a comparison of pixels between two images, and if the pixels have changed too much, then something must have moved in the image. To determine exactly which pixels have changed, and thus determine whether there has been movement and which part of the image has moved, a set of operations are carried out.

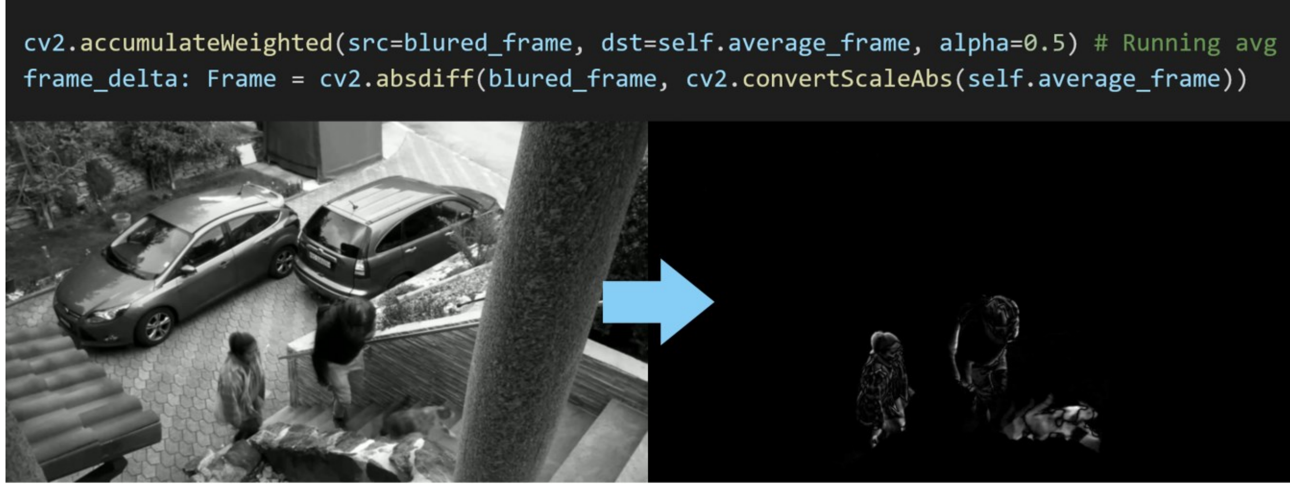
The first step is to clean the image of all unnecessary artefacts, that is, to retain only the information needed for the subsequent operations involved in the process (Figure 12).

Base frame Grayed out + blurred frame

Figure 12: Artefacts cleaning process

First, the base image will be grayed out because color data is neither necessary nor desirable for the rest of the process. Then, the grayed out image is blurred to reduce sensitivity to movement. This implies that for a movement to be detected, it must have a certain minimum intensity. This operation has the advantage of acting as a filter which eliminates all the very small changes such as those of leaves in the wind for example or all the small solitary pixels whose value changes due to the physical sensor of the camera module but which in reality do not represent real movement.

The second step is to find the movement in the image (Figure 13).



Running average Frame difference

Figure 13: Movement isolation process

We start by creating what we call a running average, which is the average image calculated on the basis of the previous images and the most current image from the camera given as input. This produces a result that looks like an image taken with a long exposure time, where what has moved quickly appears dimmed and relatively transparent, and what has not move is sharp and opaque. Then, once this average image has been generated, we calculate the difference between the average image and the most recent image from the camera, the one in which we are looking for movement. The result is what we call a motion mask, which is the regions of the image that have changed.

The third stage consists of refining the movement by making it more defined, making it stand out more (Figure 14).

Figure 14: Movement refinement process

We start by applying a binary threshold, which will transform all the pixels in shades of grey into either black or white pixels. The value of the threshold represents the limit separating the grey intensities that will become white pixels and those that will become black pixels. This separation between black and white strongly highlights the movement but disconnects de facto certain regions which could be part of the same movement. To compensate for this, we dilate the white pixels to enlarge them and thus obtain a more uniform motion mask.

Finally, the last step consists of delimiting the movement (Figure 15).

Figure 15: Movement delimiting process

We start by finding the contours of the motion mask highlighted earlier. Then all that is left to do is to compute the minimum rectangle that encompasses each contour and we are done. Having now identified the coordinates of the areas where motion is detected, we can feed these areas of the image to the object recognition algorithm.

4.2.2 Object recognition

Object recognition is the second stage of the area surveillance and is carried out after motion detection. Its purpose is to provide information about the types of objects present in the image, the level of confidence it has in its prediction and the coordinates of the areas in which the objects are located.

Figure 16: Object recognition result

To achieve this objective, object recognition is performed through the use of a model. The model used in this project is MobileNetV3. It is a convolutional neural network, a type of neural network particularly well suited to identifying patterns within images (IBM s.a.). This model in particular was chosen because it was created with the intent of being used on low-power devices such as mobiles or embedded systems, which is perfect for the Raspberry Pi (Zhu 2017).

The model's job is to do what is known as inference, that is, to make predictions about what it sees in our image based on what it has seen in the past when trained on a dataset (Cloudflare s.a.b). The model we use was trained on the COCO (Common Objects in Context) dataset, which is a dataset that includes 80 different types of objects. The objects that the model is able to detect are therefore predefined and will always be objects contained in this dataset. For the purposes of our project, we are only interested in 6 types of objects: person, bicycle, car, motorcycle, cat and dog.

Let's take a look at how object detection is implemented in practice.

ObjectDetector.py

```
1 ...
2
3 class ObjectDetector:
4     ...
5
6     def configure_neural_network(...) -> -
7         self._net = cv2.dnn_DetectionModel(
8             'frozen_inference_graph.pb', 'ssd_mobilenet_v3_large_coco_2020_01_14.pbtxt'
9         )
10        self._net.setInputSize((320, 320))
11        self._net.setInputScale(1.0 / 127.5)
12        self._net.setInputMean((127.5, 127.5, 127.5))
13        self._net.setInputSwapRB(True)
14
15    def detect(...) -> list[ObjectDetectionResult]:
16        ...
17
18        class_ids, confidences, bounding_boxes = self._net.detect(
19            frame = sub_frame,
20            confThreshold = 0.65,
21            nmsThreshold = 0.1
22        )
23
24        ...
```

Figure 17: Implementation of object recognition

To begin with, the code will call the `configure_neural_network(...)` method to create the detection model and configure it. In the creation phase, OpenCV's Deep Neural Network module is used to load the pre-trained model. The first argument to the `cv2.dnn_DetectionModel(...)` function is the binary file containing the trained weights, while the second argument is the neural network configuration file. Once the detection model has been created, it is configured by providing it with information about how it should process the image given as input. The values supplied are generally determined by the way the model used was pre-trained.

When the detection model is ready, the code will be able to use it by calling the `detect()` method. The model will then perform inference on the input image and return its prediction. Note that in addition to the image, a confidence threshold and a non-maximum suppression threshold are passed as arguments to the method. This ensures that the model returns only the most relevant detections.

An interesting case that must be handled when performing object detection is having several different objects detected for the same movement. Figure 18 below shows an example of such a case.

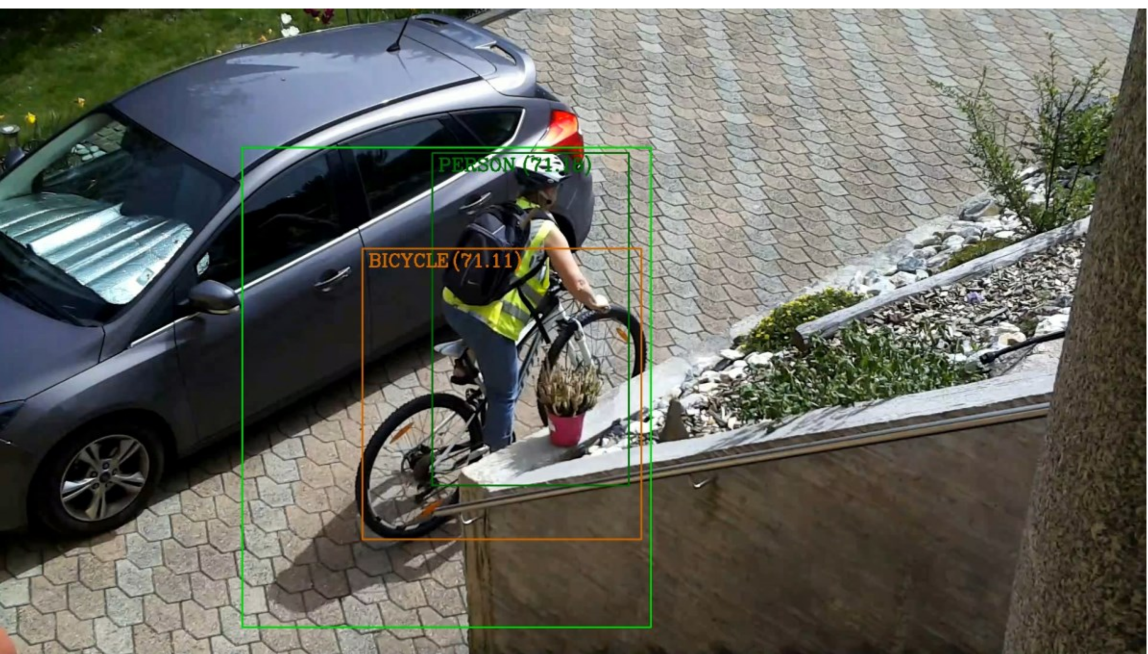


Figure 18: Hierarchization of recognized object types

For our use case, it was decided that if several objects were recognized for the same movement, the object with the largest bounding box in terms of area would be considered to be the source of the detection trigger. This is based on the natural assumption that the larger the object, the easier it is to recognize. The chances of making a mistake are therefore lower. Conversely, the smaller the object, the easier it is to misrecognize it. We thus prefer a larger bounding box to a higher confidence rate.

5 Results

5.1 Problems encountered

PiSentry suffers from two main problems, which appeared fairly early on in development and unfortunately remain unresolved despite many hours spent trying to debug to find the cause.

The camera software hangs randomly

It sometimes happens that the features related to the camera module, that is the detection algorithm and live streaming, suddenly stop working. When I investigated to see where the problem was coming from, I noticed that the thread that performs the detection, which is separate from the program's main thread, was frozen.

I first thought that this could be the consequence of too much workload for the Raspberry Pi. Without a means of cooling, the processor easily rises to over 80 degrees, so perhaps the processor would have become too hot and would have simply decided to stop the responsible thread. But this hypothesis was ruled out after the problem persisted even with the installation of the fan, which allows the Raspberry Pi to remain between 65 and 75 degrees.

If it was not the temperature, then maybe it was because of the RAM, which would be full and could no longer take in new data. But here again, monitoring of the RAM and the swap file shows that even under a heavy workload and over a period of several days of operation, only a quarter of the RAM is used at most. So, this is another route that can be dismissed.

My most current hypothesis is that the problem must be related to a software layer that manages the camera module. I am led to believe this because the camera no longer responds to any commands, but when trying to initialize it again without first killing the whole process, the camera software tells us that it is already busy. There must therefore be a mechanism that creates this lock.

The problem occurs, not only but most often, in the morning when the camera has to resume operation after remaining inactive during the night. There may therefore be an interaction timeout with the camera where, if it is not used within this period of time, it becomes unusable. I also have the impression that this problem occurs more frequently when the program is run as a systemd service than manually in a shell, but I have no concrete evidence which would suggest that the source of the issue comes from the way of launching the program. But it could perhaps exacerbate the problem in some way.

That being said, by looking through the systemd documentation, I may have found a way to mitigate the problem, failing to solve it.

Although I have not tested, it might be possible to take advantage of systemd's watchdog timeout feature. The watchdog is activated when the program starts and checks that the latter regularly sends a signal within the defined timeout period. If it does not receive the signal from the program, it considers the program to have failed, terminates it and can even restart it for us. It turns out that these are exactly the operations that we must carry out manually when we want to recover from the program hang. It would therefore be very useful if the watchdog could automatically carry out the task as soon as the program stopped responding.

This particular watchdog would only work in the case where the program is executed via systemd, but if the solution proves satisfactory, then we could certainly implement such a mechanism directly in the software so that it works regardless of the way the program is run.

The camera stops live streaming to the media server

This problem is just as mysterious as the previous one. It occurs when interacting with the API while the camera is live streaming. After the API has received requests, the FFmpeg child process throws an exception and breaks the connection between the Raspberry Pi and the media server, abruptly terminating the live stream. After that, it is no longer possible to reuse the FFmpeg output to restart live streaming. The error message that appears in the console is "[Errno 32] Broken pipe".

The very first time this problem appeared, I obviously thought it was a dumb mistake in my code. Maybe an API endpoint was calling a function to stop live streaming that it should not and I had implemented it accidentally. But after having extensively rechecked the actions of each endpoint and their interactions with the other bricks of the code, no obvious problem with the logic stood out to me. So I had to resort to experimentation.

I set out with the idea of reproducing the problem. If I found the conditions under which the problem was certain to appear, then I might be able to deduce the cause. The test consisted of sending several sets of different requests. I first wanted to see if it was a particular type of resource that was causing the problem. I asked the API to return light files, very heavy files, photos, videos and to perform actions. I then tried to see if it was perhaps the number of requests, sending a lot in a very short period of time, or on the contrary just a few. But all these tests did not help. Regardless of the type and number of requests, live streaming can break as much after 4 requests as after 50, after a few seconds or after 10 minutes.

The latest and current lead is that my program must somehow be mishandling deeper networking/communication mechanisms. While researching what could be causing the error message I was getting in the console, I came across two main recurring reasons. The first is trying to write data to a socket when the connection has already been interrupted, and the second is writing data to a pipe whose buffer is already full. In our case, the first reason seems unlikely to me, as the error first appears in our program before the media server detects that the connection has been closed. It is therefore the camera software which interrupts the connection first and not the other way around. With this in mind, there remains the second reason. The latter would perhaps be possible, but I have no concrete element to back it up or discredit it, and unfortunately not having enough knowledge in this area, my investigation of the problem remained fairly superficial on this side and was not pursued for long.

Fortunately, even if it is not always possible to fix the source of a problem, there is often a way around it. This is what I did by reimplementing part of the `FfmpegOutput` class provided by the picamera2 library to add exception handling.

FfmpegOutputWithExceptionHandling.py

```python
from picamera2.outputs import FfmpegOutput

class FfmpegOutputWithExceptionHandling(FfmpegOutput):
    def stop(self):
        try:
            super().stop()
        except Exception as e:
            print('Exception caught while stopping FfmpegOutput:', e)

    def outputframe(self, frame, keyframe=True, timestamp=None):
        try:
            super().outputframe(frame, keyframe, timestamp)
        except Exception as e:
            print('Exception caught in FfmpegOutput while outputting frame:', e)
            self.stop()
```

Figure 19: Custom implementation of `FfmpegOutput` class with exception handling

Although very basic, this allowed me to catch the exception before it terminated the FFmpeg child process and made it unusable. Coupled with an automatic retry mechanism on the web app side to start live streaming even if it experiences an error, this workaround saved the live streaming functionality as this problem was making it impossible to use.

5.2 Current limitations and possible improvements

The project finds itself in a situation where certain choices that have been made make it difficult, if not impossible, to use the camera in other scenarios, such as at night. The same applies to the development of certain functionalities, which is limited by the technologies that have been used. This is a consequence of decisions commonly referred to in software development as technical debt. This can arise from various aspects, whether technical, organizational, material or even human, and in our case, it is clear that it is above all the lack of projection at the start of the project that is to blame.

When defining needs and objectives, I did not have the knowledge and confidence to claim I could do better. So, I chose goals that I considered ambitious but achievable. Now that the project has come to an end, and building on new knowledge acquired in the relevant areas, I feel comfortable looking back and talking about the limitations of the project in its current state and the ways I see to remove these barriers. Let's review some of these ideas.

Use a camera capable of filming day and night

The camera currently used does the job very well during the day, when there is enough light. But as soon as the sun goes down, the image quickly becomes too noisy and then too dark to be able to detect movement, let alone recognize objects.

This situation is due to the fact that the camera currently used is only suitable for daytime operation. It contains an infrared cut-off filter which allows to capture only the spectrum of light visible to the human eye in order to render an image more faithful to what we would see with our own eyes. However, this filter, as its name suggests, reduces the sensitivity of the camera to infrared light, and effectively prevents the camera from seeing at night. Indeed, a technique commonly used by cameras to see in the dark is to illuminate the scene using infrared radiation. If we block this light with an infrared filter, the photographic sensor does not receive the light and therefore sees everything black.

A start of a solution would be to use a camera without an infrared filter. This would make it possible to see infrared light at night, but would then cause odd colors to appear during the day.

Figure 20: Image captured with infrared cut-off filter (left) and without (right)

The complete solution would be to add a switchable infrared filter to the camera. This could be placed in front of the sensor during the day to have beautiful images with natural colors and removed at night when we need to capture infrared light.

Note that to illuminate the scene at night, a source of infrared radiation is necessary. To implement this solution, we should not only change the camera but also add infrared lighting.

Use a camera which can film with a wider angle of view

The field of view (FOV) is an important feature to consider when choosing the camera, all the more for a security camera. The wider the angle of view, the larger the area that can be observed. It is therefore useful, especially outdoors, to have a camera with a wide enough angle of view so as not to need several cameras to cover a single area.

It is clearly a drawback of the current camera. The latter has a field of view of 62.2 degrees horizontally and 48.8 degrees vertically. In our case, by carefully choosing where to place the camera, this configuration is mathematically sufficient to completely cover the desired area. However, when filming the area, the image appears very zoomed in and no longer shows the entire area. As a result, when performing motion detection for example, we often find ourselves with subjects rapidly moving in and out of the captured area, complicating the object detection process and video recording as a whole.

The problem is actually linked to the image processing done by the camera in video mode. The camera has a physical sensor of 3280 x 2464 pixels which gives it a native 4:3 aspect ratio. When we want to film in Full HD resolution, that is 1920 x 1080 pixels and which gives a 16:9 aspect ratio, the camera only chooses a rectangle of 1920 x 1080 pixels in the center of the complete image and discards all the other pixels around. This is what gives the zoomed-in effect. A video in Full HD format therefore no longer allows you to see the entire area.

This situation does not, however, apply in photo mode, where an image taken in Full HD resolution corresponds to what we generally expect to obtain, that is an image downscaled in width from 3280 to 1920 pixels, then cropped in height to preserve a 16:9 aspect ratio.

Despite my research, I did not find the exact reason for this differentiated treatment between photo and video, but I think it has to do with the camera's performance. In video mode, it is important for the camera to be able to shoot a high number of frames per second, known as the frame rate, to ensure smooth videos. And the simpler it is to process each image, the greater the number of frames per second the camera can generate. It is therefore preferable for the camera, when it has to generate images which do not have the same aspect ratio as the native one, to avoid carrying out complex operations in order to be able to offer an acceptable frame rate. In photo mode, the frame rate is not relevant because only a single image is captured. The camera can afford to do heavier and more time-consuming processing.

To resolve this problem of too narrow a field of view, two solutions could be considered.

The first would keep the current camera and would consist of filming in a resolution that supports full field of view, that is two possible choices: 1640 x 1232 or 3280 x 2464. But in both cases we would have an image in 4:3 format, which we do not want to. Furthermore, the second choice limits the maximum frame rate to 22 frames per second due to its large size.

The second and preferable solution would involve using another camera whose physical sensor would have a native aspect ratio of 16:9. To stay in the world of Raspberry Pi, this is for example the case of the new Camera Module 3 released in 2023. It would allow filming in Full HD resolution with full field of view while having a frame rate up to 50 frames per second.

Figure 21: Raspberry Pi Camera Module 2 FOV in Full HD in video (left) and photo (right) modes

Implement detection areas in the web app

When focusing the camera on a particular area, things that do not interest us may still appear in the field of view. It can be a road with traffic, a neighbor's house or even trees moving in the wind. This would be even more true if a camera with a wider field of view were used.

These areas are often sources of frequent motion detections that we would like to avoid. They give a greater workload to the camera which has repercussions in terms of performance, heat production and energy consumption. The impact is all the more significant when motion detection is carried out actively on the software side, as is the case in this project, and not passively using a hardware sensor. To mitigate these consequences, it would be useful to be able to define areas in which detection will be made, and only in these areas, in order to prevent unnecessary detections.

At this stage of the project, this feature has already been implemented in the motion detection algorithm of the camera, but unfortunately no interface has been developed for the web application, so users can not interact with it graphically yet. The only way to define these areas is to manually create them directly in the database in JSON format. The areas are in the shape of rectangles and are represented as arrays of four integers. Each array follows the format convention "[x, y, width, height]" where x and y are respectively the abscissa and ordinate of the top left corner of the rectangle, and where width and height are the dimensions of the rectangle.

Figure 22: Visual of currently implemented detection areas and their JSON equivalent

Use sensors

In the current state of the project, all the data we are working with, which is not generated by the user, comes from the camera. This is the only external hardware component that the Raspberry Pi needs. We use the camera's video feed for motion detection, object recognition and live streaming.

This simplicity can be considered both an advantage and a disadvantage. An advantage because setting up is quick and easy, with affordable and accessible equipment. But also a disadvantage,

because the camera module cannot cover a wide range of functionalities on its own and is therefore not the best way to solve all problems. Adding certain sensors could greatly improve the security camera, both in terms of efficiency and value proposition.

The first sensor I think of is a passive infrared sensor (PIR sensor). It works by detecting changes in the amount of infrared radiation coming from the area it is observing. Widely used to detect movement in security applications or lighting and door opening automation, we could imagine using it to relieve our software-based motion detection solution. Indeed, the latter currently analyzes continuously the images from the camera to determine whether something has moved or not. If the PIR sensor could do this job in place of the camera, the software-based motion detection could be used only to locate movement in the image and would only be needed for a very short period of time. This would significantly reduce the workload on the Raspberry Pi and consequently result in much lower power consumption. This is not negligible in case we wanted to power the Raspberry Pi via batteries.

The second sensor is a microphone. Since our camera module does not have a built-in microphone, all we can get is a video feed with no sound. This is detrimental to the user experience and is quickly noticed when watching videos or the live stream. The quickest solution would probably be to use a camera module with a built-in microphone, but other options would also be possible, such as using an external microphone connected via USB or managed via the GPIO pins.

6 Conclusion

6.1 Project assessment

Conducting this project was challenging. Considering the numerous features to be developed and my lack of knowledge of the related fields, the learning has been substantial. I had so far never done anything related to the world of video, be it live streaming, transmuxing, computer vision, and so on. This has been therefore my first experience in a lot of domains.

The same goes for accessing remotely the camera. Managing remote access to many devices could be viewed more like the job of a network administrator than a software developer. However, in the field of IoT, it is crucial to have this skill to ensure that you keep a permanent connection to the devices to be able to monitor them correctly. The implementation of such a remote access system has been again a first for me.

In fact, the whole project has been a discovery, and while this is great from a learning perspective, it has also led to taking much longer than expected. In an overly optimistic mood, I underestimated the amount of work that was going to be required to build on technologies that I did not master. The end of the project has even been postponed several times because of this.

Fortunately, after this demanding phase of intensive learning and development, the result is all the more appreciable. The camera is currently in operation 24 hours a day, and has been since January 2024. Even if it sometimes needs a little maintenance to restart processes which have crashed or which have become stalled, at this stage it is already virtually autonomous. In addition, I never need to manipulate the hardware directly. The problems mentioned earlier are all software-related and can be solved remotely.

As is often said, a software project is never completely finished. There are always optimizations, improvements and new features that can be made. This is also very true for this project. There are still many aspects that I would like to continue developing. Some features could be improved, some new capabilities could be added. But as it stands, PiSentry meets all the objectives set at the start of the project. I therefore consider it achieved within the framework of this thesis.

6.2 Personal assessment

If I had to summarize what I draw from this project in two words, it would be satisfaction and pride. Looking at the final result, I am proud to have successfully completed a project that was initially a challenge. When I started it 18 months ago, I had everything to learn. A working security camera

was just an ideal, a wish. Today, the finished product lives up to my expectations. It works even better than I could have imagined when I embarked on this adventure.

Of course, this personal conviction has not always been unshakeable over these past few months, far from it. Due to the length of the project, I sometimes found myself demotivated, wondering not when the project would be finished but whether it would ever be finished. Things never progressed as quickly as I wanted to. It was certainly a period rich in learning but nevertheless complicated to manage. Fortunately, the project was being built step by step, and every time a small feature was completed and working, the motivation was back to keep moving forward.

Speaking of motivation, it has never been higher than it is right now. This project is proof that great things can be achieved if one put heart and determination into it. That is exactly the mindset I intend to apply to future projects as well. The desire to continue not only this project but to develop new ones based on the knowledge acquired is great. I cannot wait to see where it leads me.

Sources

Alasdair, A. 2023. Heating and cooling Raspberry Pi 5. URL: https://www.raspberrypi.com/news/heating-and-cooling-raspberry-pi-5/. Accessed: 29 April 2024.

Apple s.a. HTTP Live Streaming (HLS) authoring specification for Apple devices. URL: https://developer.apple.com/documentation/http-live-streaming/hls-authoring-specification-for-apple-devices. Accessed: 18 May 2024.

Cloudflare s.a.a. What is H.264? | Advanced Video Coding (AVC). URL: https://www.cloudflare.com/learning/video/what-is-h264-avc/. Accessed: 18 May 2024.

Cloudflare s.a.b. AI inference vs. training: What is AI inference? URL: https://www.cloudflare.com/learning/ai/inference-vs-training/. Accessed: 19 May 2024.
Ericsson 2023. Number of connected devices worldwide in 2014 and 2028, by device (in billions) [Graph]. In Statista. URL: https://www.statista.com/statistics/512650/worldwide-connected-devices-amount/. Accessed: 4 April 2024.

Fox, S. 2023. Texas couple sues Maryland man after Airbnb nightmare. URL: https://www.fox5dc.com/news/couple-files-lawsuit-after-finding-hidden-cameras-in-silver-spring-airbnb. Accessed: 12 March 2024.

IBM s.a. What are convolutional neural networks? URL: https://www.ibm.com/topics/convolutional-neural-networks. Accessed: 19 May 2024.

Paul, D. 2019. Spy cameras live-streamed 1,600 hotel guests for subscribers. Then police caught on. The Washington Post. URL: https://www.washingtonpost.com/world/2019/03/20/spy-cameras-secretly-live-streamed-hotel-guests-subscribers-then-police-caught/?itid=lk_inline_manual_18. Accessed: 12 March 2024.

Radio Télévision Suisse 2016. L’espace public suisse compte 21’000 caméras de surveillance. URL: https://www.rts.ch/info/suisse/7619786-lespace-public-suisse-compte-21000-cameras-de-surveillance.html. Accessed: 11 March 2024.

Raspberry Pi Ltd 2023. The Picamera2 Library. URL: https://datasheets.raspberrypi.com/camera/picamera2-manual.pdf. Accessed: 29 April 2024.

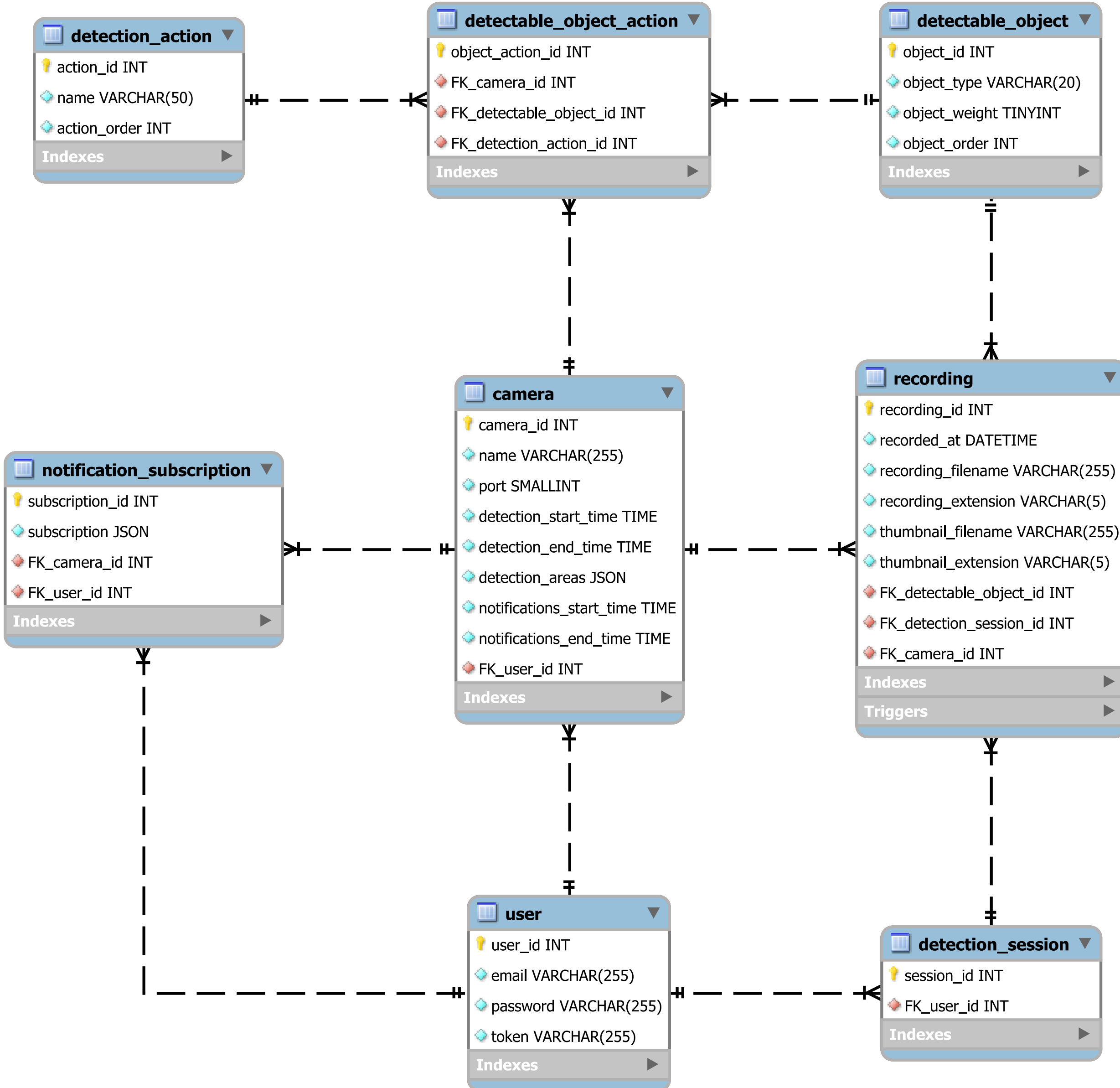
Ratcliffe, J. 2020. Number of CCTV Cameras in the UK reaches 5.2 million. URL: https://cctv.co.uk/number-of-cctv-cameras-in-the-uk-reaches-5-2-million/. Accessed: 11 March 2024.

Rivas, A. 2019. Airbnb Couple Finds Hidden Cameras in Bathroom, Bedroom: Lawsuit. URL: https://www.nbcsandiego.com/news/local/airbnb-guests-find-hidden-cameras-in-bathroom-bedroom-lawsuit/2109603/. Accessed: 12 March 2024.

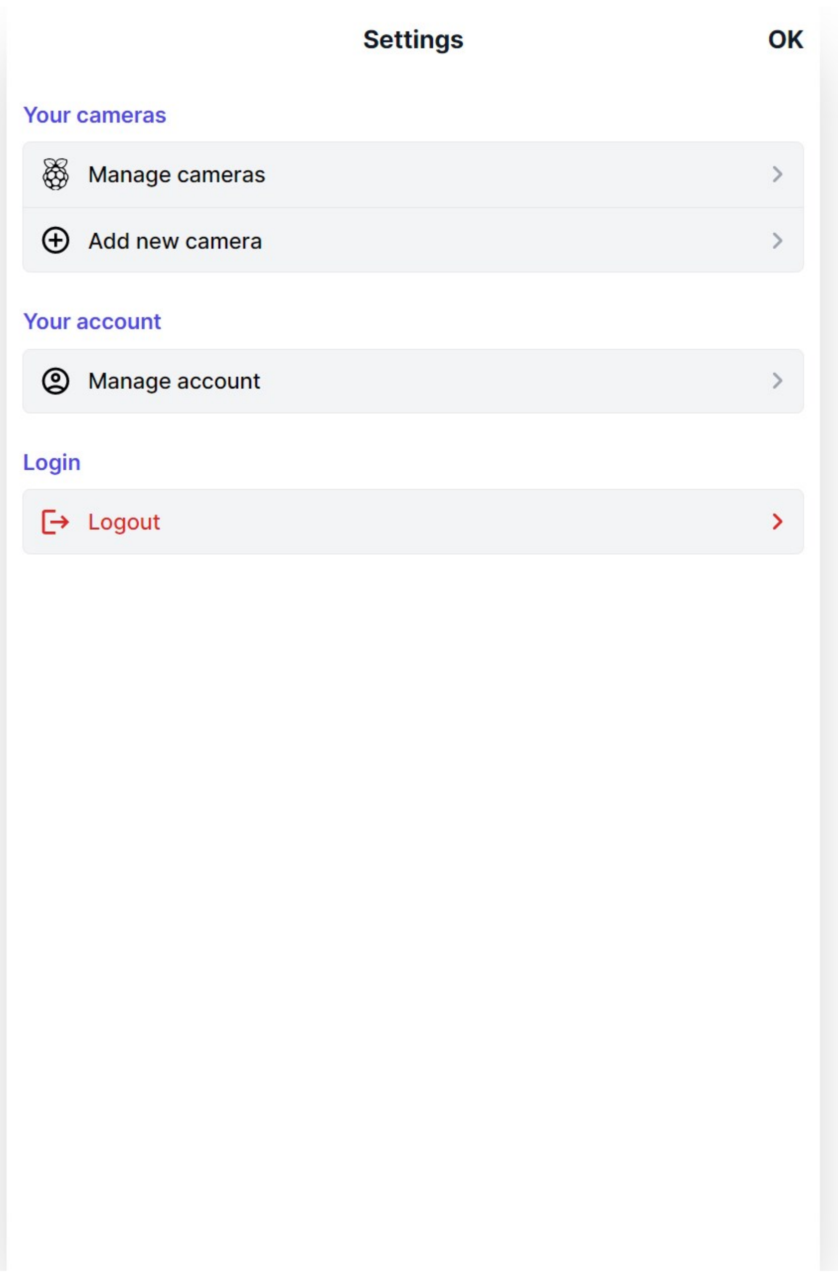
Zhu, M. 2017. MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications. URL: https://arxiv.org/abs/1704.04861. Accessed: 19 May 2024.

Appendices

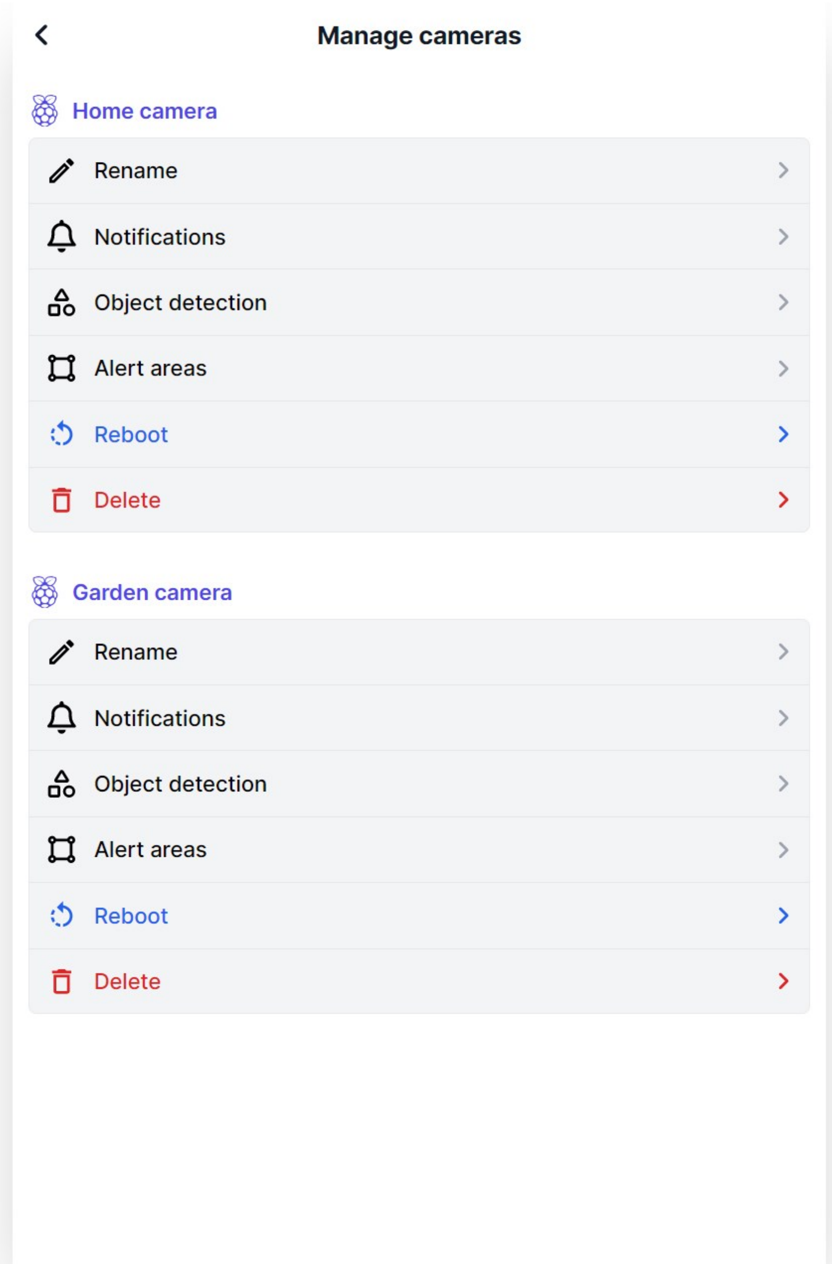
Appendix 1. PiSentry database Entity Relationship model



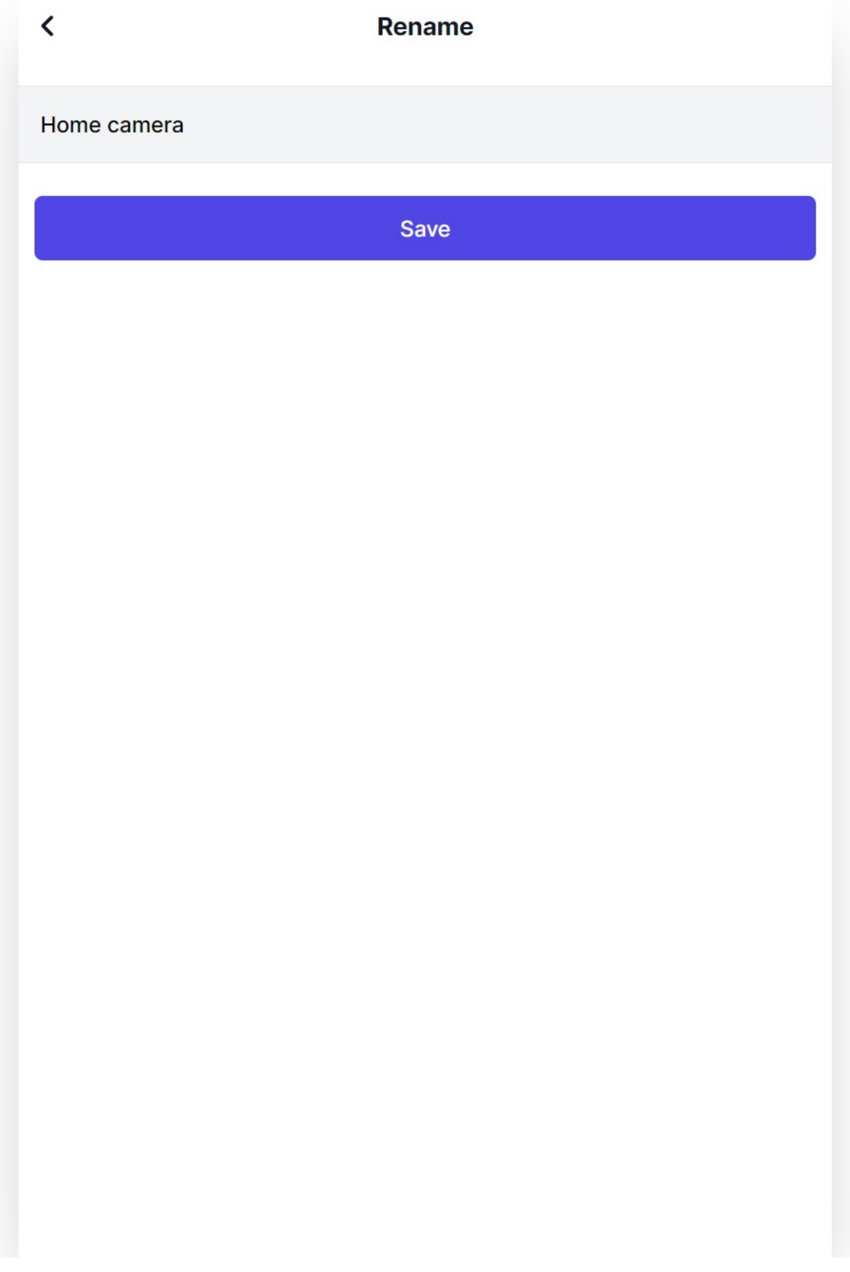
Appendix 2. Web application – Settings page



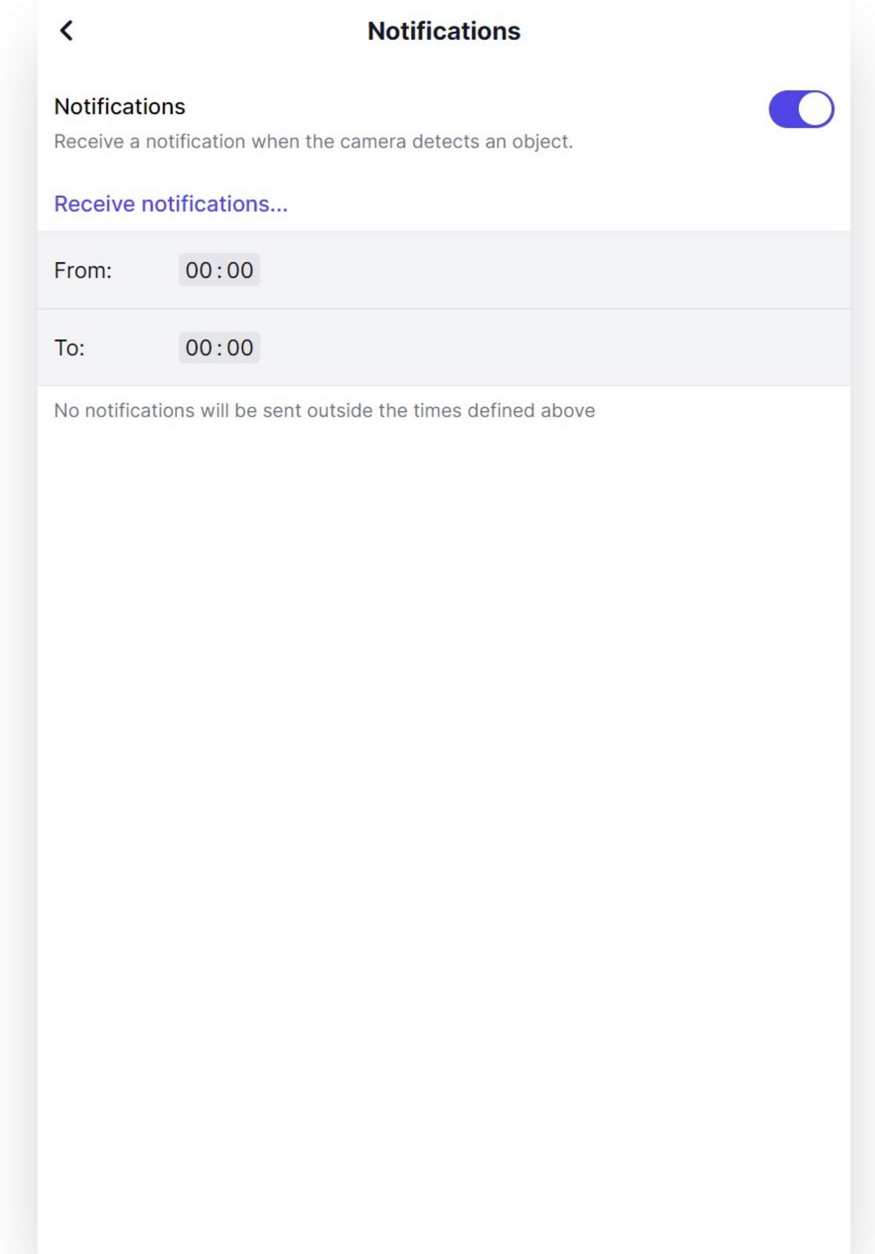
Appendix 3. Web application – Camera settings page



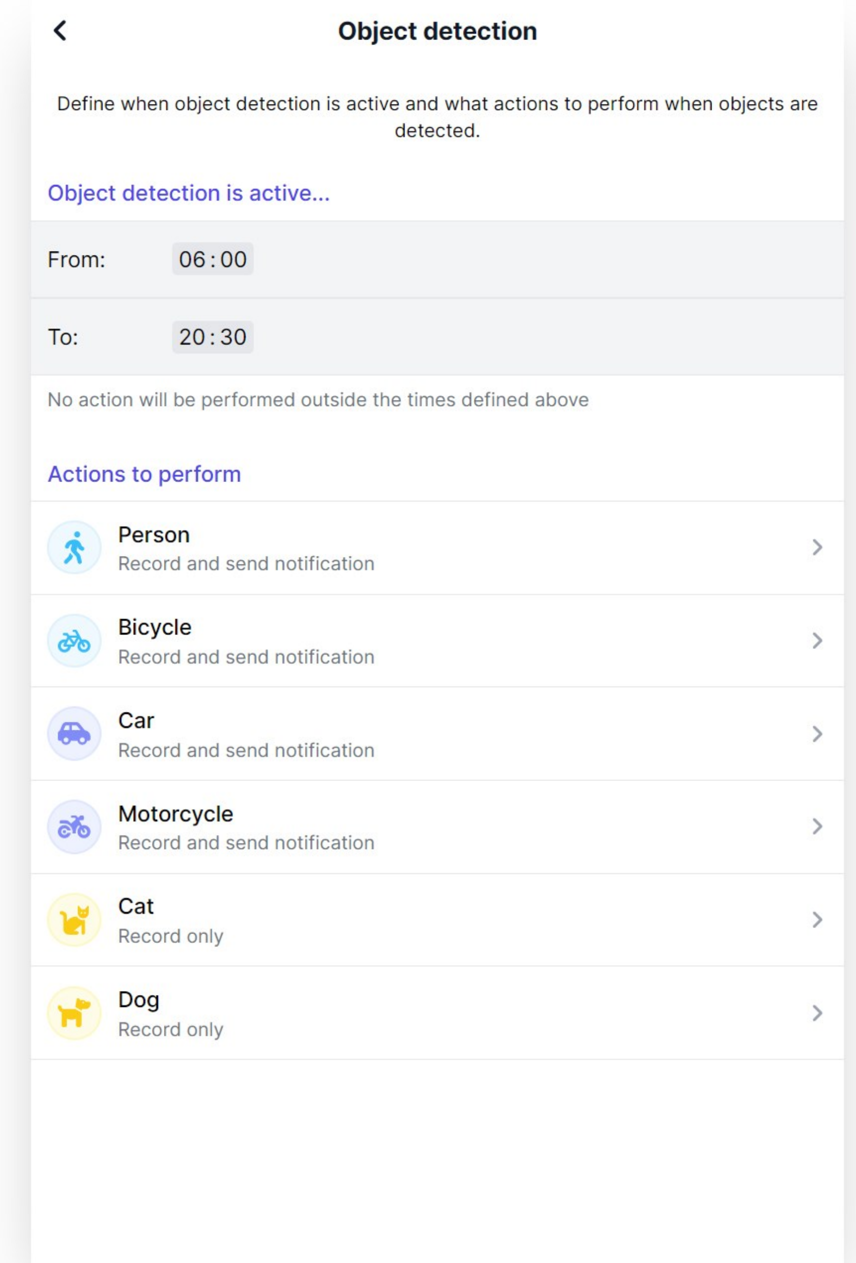
Appendix 4. Web application – Rename camera page



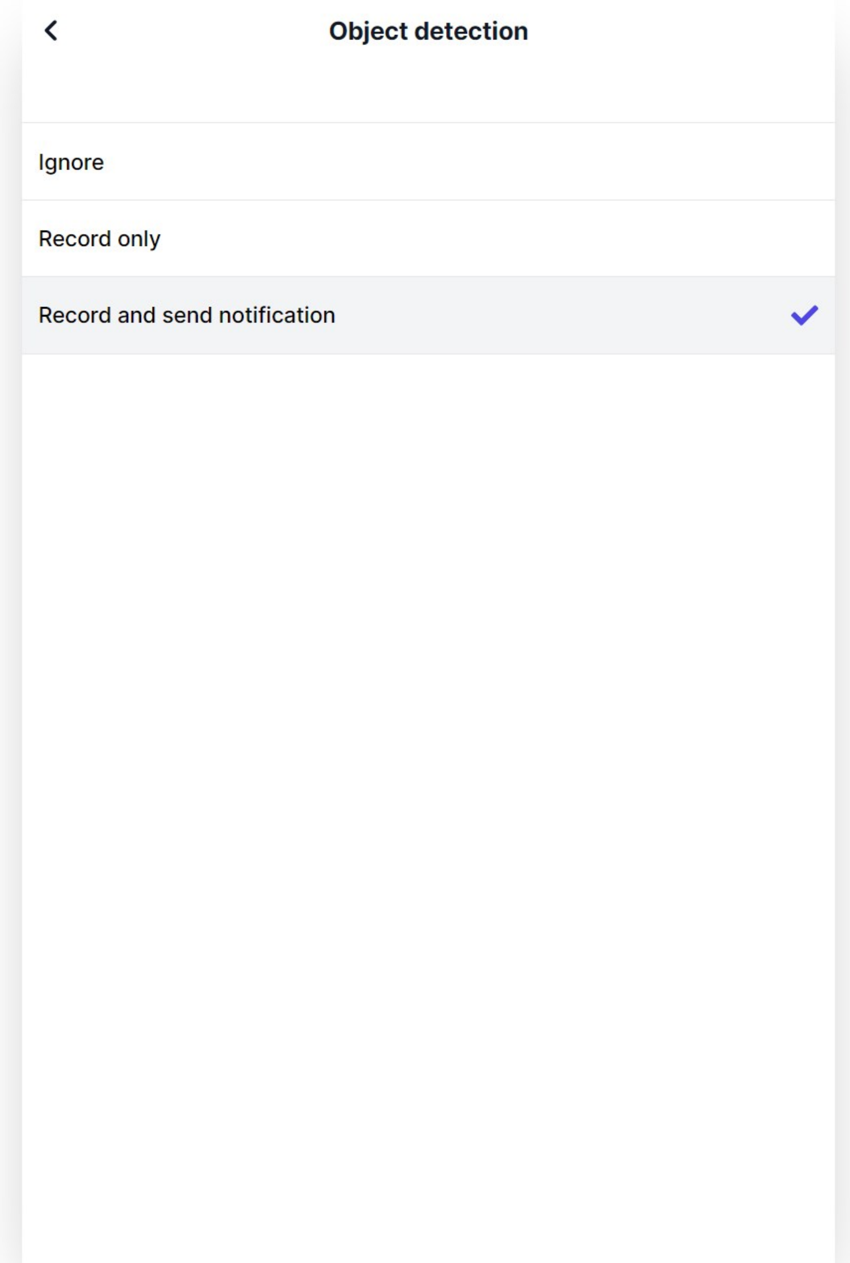
Appendix 5. Web application – Notifications page



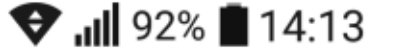
Appendix 6. Web application – Object detection page



Appendix 7. Web application – Detection actions page

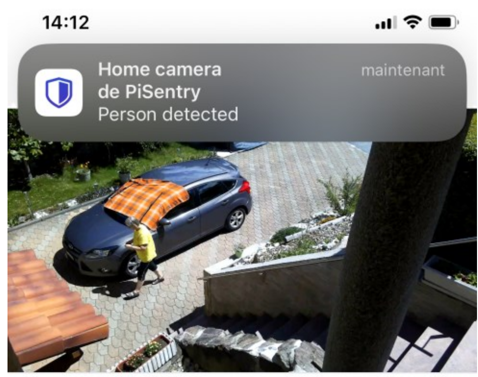


Appendix 8. Web application – PWA splash screen on Android



PiSentry

Appendix 9. Web application – PWA push notification on iOS



FRIDAY 17 MAY