Revue

Mattia Matteini
mattia.matteini@studio.unibo.it

Alberto Paganelli
alberto.paganelli3@studio.unibo.it

March 2023

Revue is a real-time video surveillance and environment monitoring system. It is designed to be used in multiple scenarios such as home, office or warehouses following the user needs. Revue permits controlling the environment through sensors and cameras, providing real-time data and video streaming. It is also possible to configure security rules, to get notified when a sensor data exceeds a specified range, or when camera recognise an intrusion.

1 Requirements

The goal of the project is to develop a distributed software system which is able to monitor the environment of a certain area through sensors and cameras, providing real-time data and video streaming.

Moreover, to enhance the usefulness of the system, it should also be able to notify the user when specific conditions are met. These conditions include detecting whether sensor data exceed a predetermined range or cameras recognise a particular object. This notification feature ensures that the user is promptly informed about critical events or anomalies in the monitored environment.

The outcome should be a reliable system adaptable to different scenarios, such as smart cities, industrial, or simply home monitoring.

In the following are listed the main requirements of the system.

1.1 User Requirements

1. The user can authenticate to the system through a web interface.
2. The user can monitor the environment data produced by the sensors.
3. The user can monitor the video stream produced by the cameras.

4. The user can add/delete a device to the system.
5. The user can enable and disable a device.
6. The user can modify a device configuration.
7. The user can add/delete a security rule regarding a camera/sensor.
8. The user can modify a security rule.
9. The user can delete received notifications.
10. The user can consult the history of produced data.

1.2 System Requirements

1. The system grants access only to authenticated users.
2. The system provides a web interface as an entry point for the user.
3. The system generates video and data streams.
4. The system monitors streams to detect anomalies.
5. The system notifies the user when a security rule is violated.
6. The system persistently stores produced data.

1.3 Non-functional Requirements

1. The system should be modular and reliable. In particular:
   a) The system should work even though the recognition component is down or not deployed.
   b) The system should work even though the component responsible for the storage of the data is down or not deployed.
   c) The system should work even though the component responsible for the authentication is down.
2. The system should be as much as possible available (and also replicable).
3. The system should be usable, with a user-friendly and minimal interface.

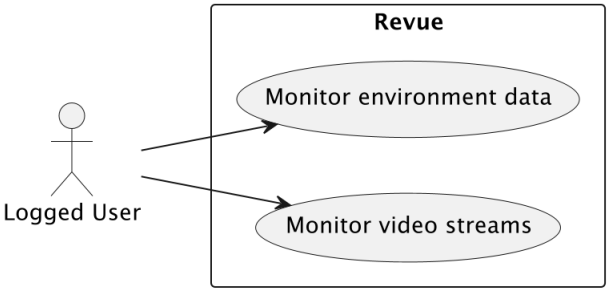

Figure 1: Simple Use Case

1.4 Implementation Requirements

1. The recognition component of the system should be implemented in Python.
2. The frontend of the system should be implemented using Vue 3 and Typescript.
3. The storage of data should be implemented using a NoSQL database.
4. The system should be implemented using a microservices architecture.
5. The system should be deployed using Docker.

1.5 Scenarios

The system can be used in various scenarios, depending on the user needs. This system is designed to be used by multiple types of users, from a private user to a company director.

In the sections below, are explained two main possible scenarios in which the system can be used.

In the simplest scenario, the system is used by a private user, who wants just to monitor his home or a particular environment without the necessity to recognise the objects in the video. In this case, the user can just rely on the camera, monitoring the home or a proprietary field. The user is free to monitor the live video by the camera whenever and wherever he/she wants, using the browser on the smartphone. The user can also set up sensors to monitor data from the environment.

A more complex scenario could involve both sensor and camera usages with the support of n intelligent feature to detect intrusion. For example, the director of food wholesale could monitor the temperature of the warehouse and the presence of unauthorized people during the night. In this case, the recognition part of the system is necessary to detect whenever an intrusion occurs.

Moreover, supply chain monitoring can be done for who's needs to ensure that the temperature of the warehouse is always in the right range and be alerted whenever the temperature exceeds a certain range to detect or prevent problems.

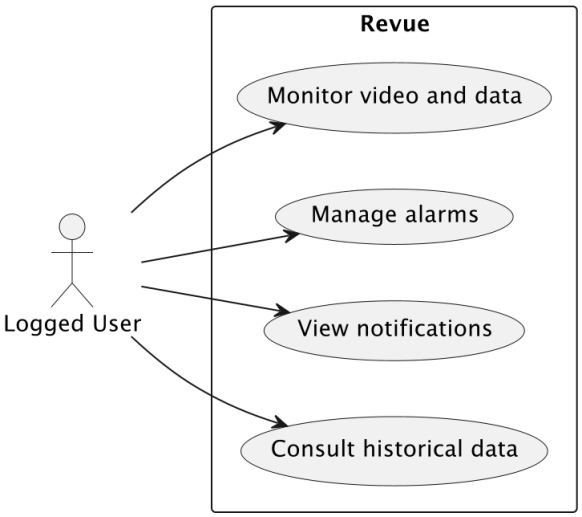

Figure 2: Advanced Use Case

1.6 Self-assessment policy

The project can be considered completed when **all** requirements are satisfied.

In particular, non-functional requirements aim to grant a good general quality of the system. This will be achieved also using automated tests that will ensure, among other things, the quality of code production.

2 Requirements Analysis

Drawing up requirements, one relevant considered topic is the recognition feature. This is supposed to be developed using Python and exploiting its main libraries for video processing and object recognition, to minimize the abstraction gap.

Another important aspect is the handling of video streams. In fact, to facilitate the development, an implicit requirement is necessary: the use of an ad hoc *media server*. This permits also improving the compatibility permitting to produce and consume using different protocols.

Eventually, internal communications between devices and services need to be guaranteed. This implicitly leads to the use of a message broker (*Kafka*) which guarantees better scalability and at least one message delivery policy.

3 Design

3.1 Ubiquitous Language

For the initial design phase, it's useful to define a common language that permits referring to the concepts with coherence and no ambiguity. This is a fundamental part of Domain

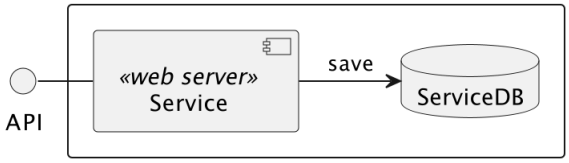

Figure 3: Microservice components

Driven Design philosophy.

In table 1 is shown the designed ubiquitous language, while in table 2 are reported the synonyms.

3.2 Architecture

The overall system is designed with microservices architecture. This choice helped us to increase modularity, scalability, reliability and fault tolerance. The system is composed of the following microservices:

- **Authentication Service**: responsible for the authentication and access control.
- **Monitoring Service**: responsible for managing devices and environment data.
- **Recognition Service**: responsible for the recognition of objects in the video streams.
- **Alarm Service**: responsible for the management of the security rules and anomalies.
- **Notification Service**: responsible for sending notifications to the user.
- **Log Service**: responsible for the persistent saving of data.

Each microservice consists of 1. Web server exposing REST APIs 2. Database to store its data (except for the **Recognition Service**) (figure 3).

Moreover, other components are necessary to make the system work:

- **Frontend**: provides to the user the web interface to interact with the system.
- **Sensors**: captures the environment data and send them to the rest of the system.
- **Cameras**: captures the video streams and send them to the rest of the system.
- **Media Server**: used to consume the produced video streams and make them available using different protocols.
- **Broker**: used to manage some internal communications.

In figure 4 is presented the whole architecture diagram.

Term	Meaning
Camera	Device that generates a video stream and sends it to the other parts of the system
Sensor	Device capturing sensing data from an environment (e.g. temperature)
Device	Either a camera or a sensor
Video Stream	Stream of video data produced by a camera
Environment Data	Data produced by a sensor
User	User that can access the system
Detection	Recognition of an object in a video stream
Intrusion	Detection of an unauthorized object
Exceeding	Environment value exceeding user defined ranges
Anomaly	Either an intrusion or an exceeding
Security Rule	Rule defined by the supervisor to trigger anomalies in a defined range of time
Notification	An alert sent to the user to inform that an anomaly has been triggered.

Table 1: Ubiquitous Language

Term	Synonyms
Camera	Video camera
Video Stream	Video, Transmission
Environment Data	Data, Sensing Data
User	Supervisor, Admin
Security Rule	Rule

Table 2: Synonyms

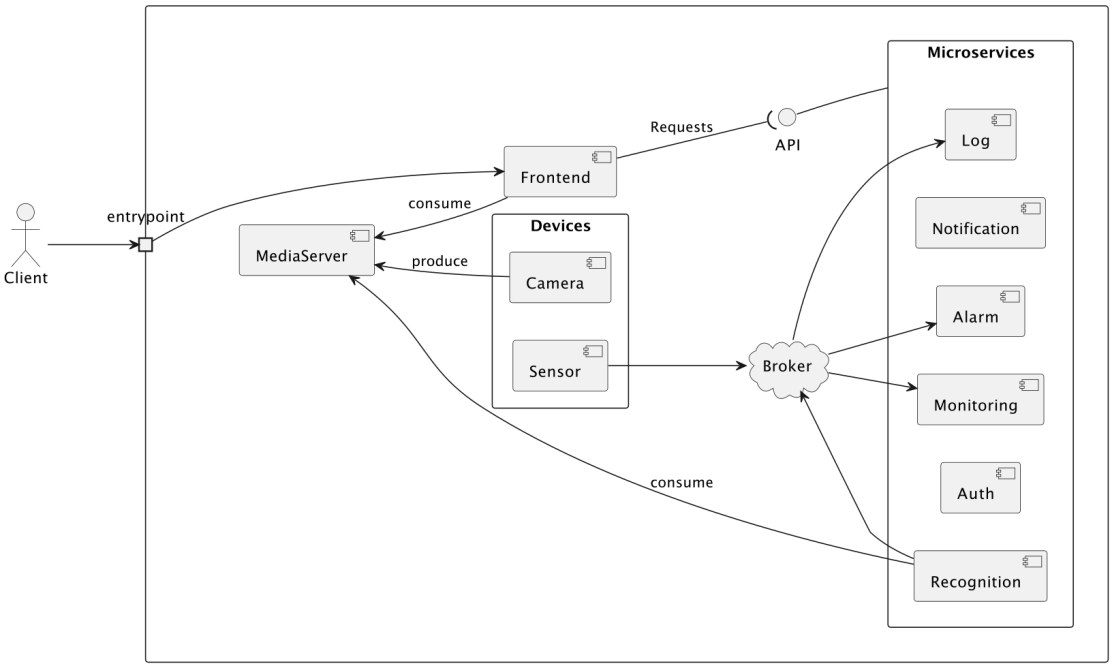

Figure 4: Revue Architecture

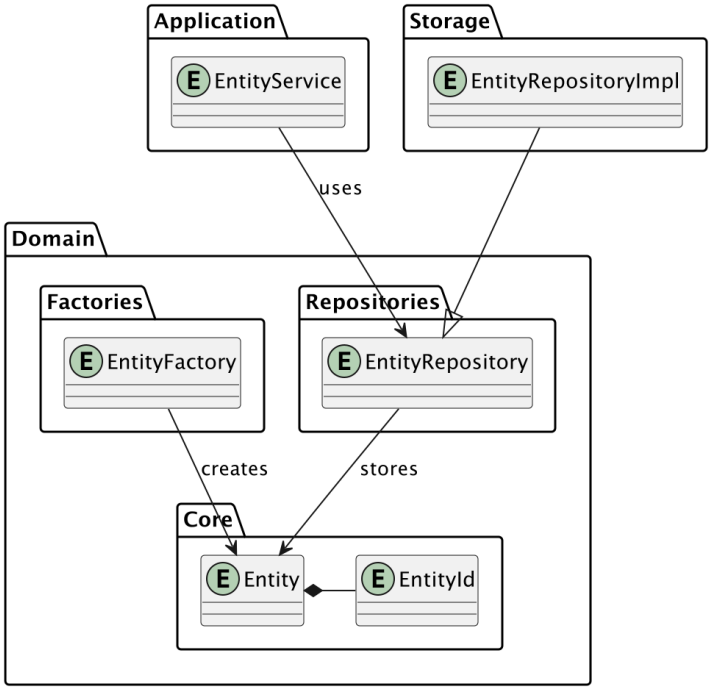

Figure 5: General structure of microservice

3.3 Structure

For the design phase, Domain Driven Design (DDD) and Hexagonal Architecture (HA) have been adopted. Actually, the adherence to DDD and HA can be considered a little bit rough (for example, because of the lack of an ad hoc presentation layer) and it will be improved in the future.

To briefly specify the layers' dependencies:

- **Domain** layer cannot depend on any other layer.
- **Application** layer (which contains business logic) can depend only on Domain layer.
- **Storage** layer depends on the layers above. It is the outer layer since it contains platform-dependent implementation.

Each microservice has the same structure in terms of classes and packages. Furthermore, it is responsible for a specific subset of domain entities.

In figure 5, is represented the general packages/classes structure.

To not replicate all the code in the microservices, a shared module called **domain** has been created. This is possible due to the fact most of the microservices share the same platform (Node.js). The exception is the **Recognition** service, which is implemented in Python, and it required a reimplementation of some domain entities.

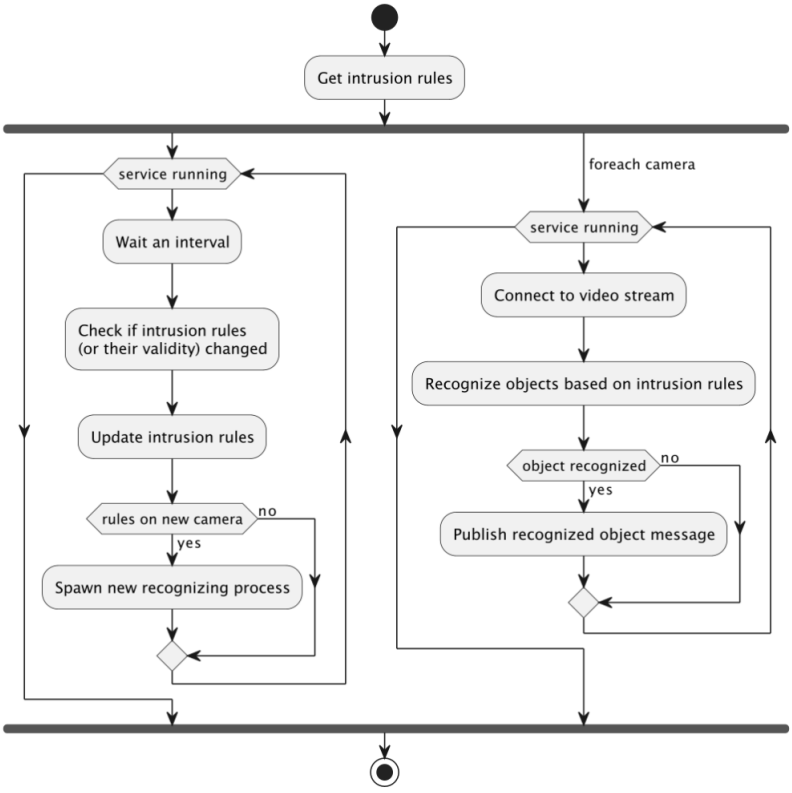

Figure 6: Recognition service activity diagram

3.4 Behaviour

3.4.1 Monitoring Service

Monitoring is the microservice in charge of retrieving environment data from the **Broker** produced by the sensors, and to provide them to the user through the **Frontend**. It allows also adding, delete or modify a device configuration, and it is responsible for the management of the devices.

3.4.2 Recognition Service

Recognition microservice performs in loop object recognition on cameras' video streams. Initially, it gets the intrusion rules from the **Alarm** service. Then, it connects to the **Media Server** to consume the video streams. When a predefined object is recognised, it sends it to the **Alarm** service.

Periodically, it checks if the intrusion rules, or their validity, have changed.

Foreach camera subject to an active rule, it spawns a new process to perform the recognition. When there is no more active rule for a camera, the process terminates.

An activity diagram is shown in figure 6.

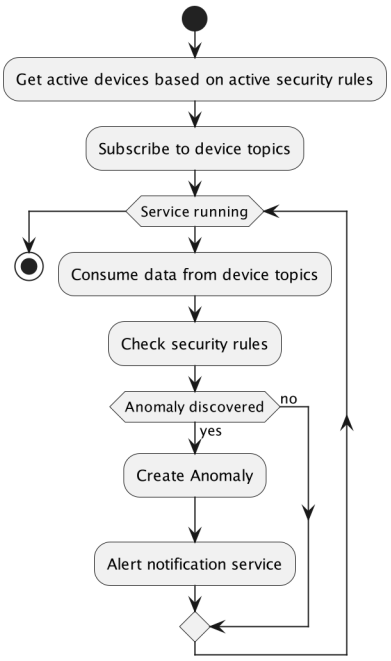

Figure 7: Alarm service activity diagram

3.4.3 Alarm Service

Alarm microservice is responsible for the anomalies' detection. It is able to check the data coming from the sensors or the recognised objects on cameras' streams. The service needs to check active rules for active devices. It is able to create an anomaly, store it, and notify the **Notification** service if the data is not compliant with the rules (figure 7).

Moreover, this service is in charge of the security rules management, allowing the user to modify their configuration.

3.4.4 Notification Service

This microservice is responsible to notify the user in case of anomalies. When the **Alarm** service detects an intrusion or an exceeding, it sends a message to the **Notification** service that will notify the user through an email or an SMS.

If the user is logged in through **Frontend**, the notification will be sent in real-time using sockets.

3.4.5 Log Service

Microservice responsible for the storage of the environment data produced by sensors. It is simply listening to the **Broker** and storing the published data in the database. It also allows the user to consult the historical data of the environment.

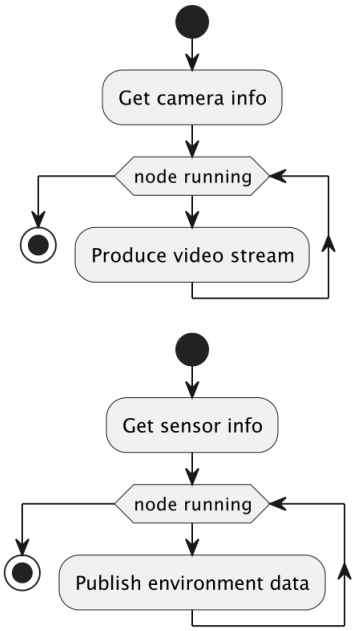

Figure 8: Device activity diagram

3.4.6 Devices

Devices, both cameras and sensors, are responsible for producing data and sending them to the **Broker**. Once initialised, they start to produce data, in particular:

- Sensors publish environment data on a topic.
- Cameras publish video streams with RTSP protocol to the **Media Server**.

Actually, both devices are simulated. Sensor nodes produce random data, while the camera nodes stream a local video file.

3.5 Interaction

3.5.1 Authentication

The **Auth** service is responsible for the authentication process (figure 9).

When a login request is performed, the **Auth** service checks the credentials and returns an access token if the authentication is successful. Whenever a request to another service is performed, the access token is sent in the header. Every service contains a validation middleware through which the token is checked.

This process has been implemented using the Json Web Token (JWT) and sharing the secret key between federated services.

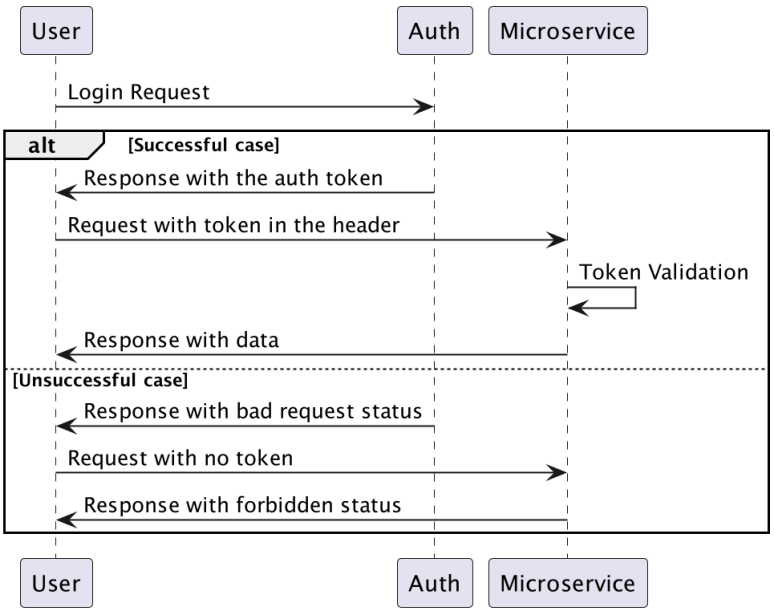

Figure 9: Auth service sequence diagram

3.6 Video Stream

The handling of video stream involves a set of components as shown in figure 10.

First of all, the camera sends the video stream to the **Media Server** using RTSP protocol. Then, the **Media Server** is responsible for making the stream available to whoever wants to consume it.

In this case, the **Recognition** service consumes the stream using RTSP protocol, while the **Frontend** consumes it using WebRTC protocol.

Recognition service starts the image recognition process on the stream and sends the recognised objects to the **Alarm** service. The latter, in case of an intrusion (table 1), will alert the **Notification** service to notify the user.

3.7 Sensor Data Stream

In the sensor data stream scenario (figure 11), what happens is analogous to the video stream scenario.

Sensors continuously produce data, publishing them to the **Broker**. In the standard case, three services will consume the data:

- **Monitoring** service, to make the data available to the user through the **Frontend**.
- **Alarm** service, to check the data against the security rules.
- **Log** service, to store the data persistently.

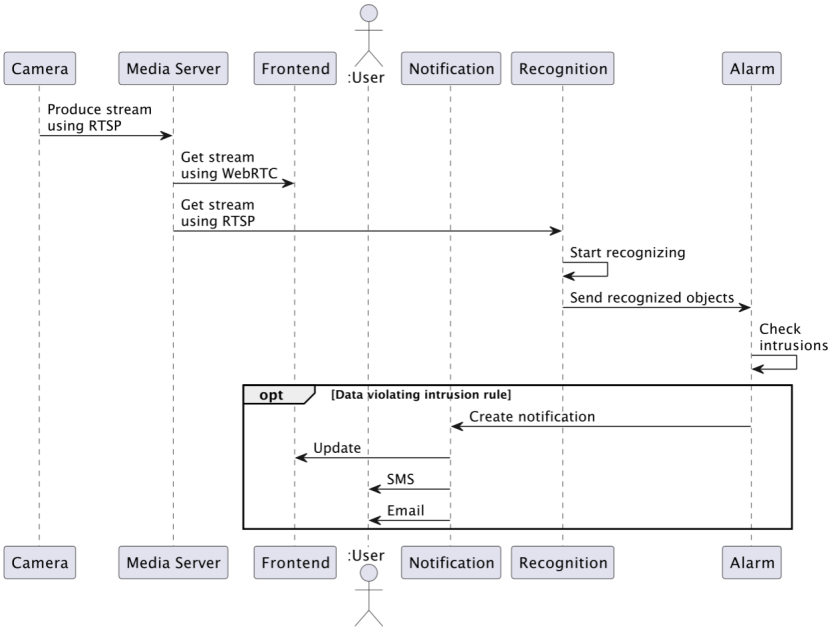

Figure 10: Video stream sequence diagram

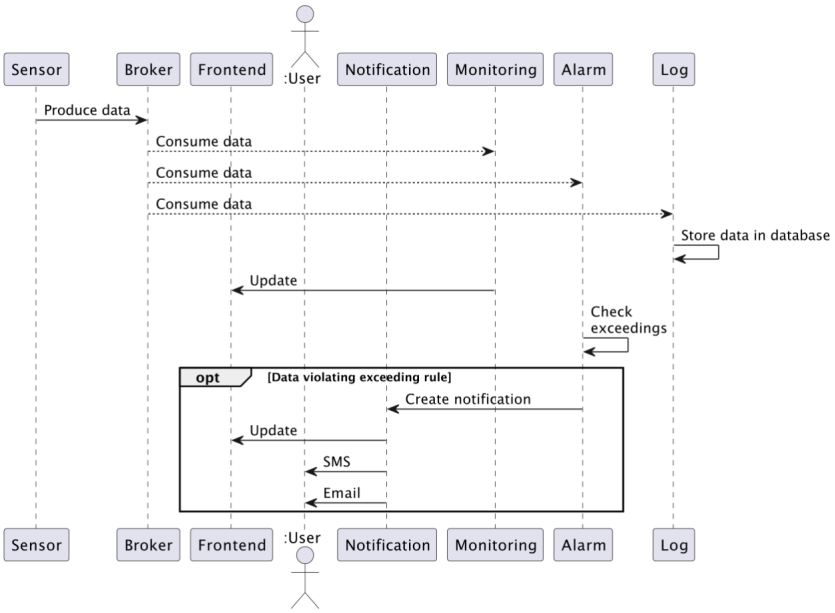

Figure 11: Sensor data sequence diagram

In case of an exceeding (table 1), the same notification process as the video stream scenario will be triggered.

4 Implementation Details

4.1 Kafka

Apache Kafka is the technology chosen to handle intra-system real-time communications.

In particular, KafkaJS and kafka-python clients have been used respectively for the Node.js and Flask services.

Kafka is an open-source distributed event streaming platform capable of handling plenty of events per second.

In the following, is reported the code for producing (Listing 1) and consuming (Listing 2) messages using KafkaJS.

Listing 1: Producing using KafkaJS

```
const kafka: Kafka = new Kafka({
  clientId: `SENSOR_${SENSOR_CODE}`,
  brokers: [`${kafkaHost}:${kafkaPort}`]
})

export const produce = async (): Promise<void> => {
  const producer: Producer = kafka.producer({ createPartitioner:
      Partitioners.LegacyPartitioner })
  await producer.connect()
  setInterval(async (): Promise<void> => {
    await producer.send({
      topic: `SENSOR_${sourceSensor.deviceId.code}`,
      messages: [
        {
          value: JSON.stringify(values)
        }
      ]
    })
  }, sourceSensor.intervalMillis)
}
```

Listing 2: Consuming using KafkaJS

```
const consumer: Consumer = kafkaManager.createConsumer('alarmConsumer')

export const setupConsumer = async (): Promise<void> => {
  await consumer.connect()
  await consumer.subscribe({ topics: await getTopics(), fromBeginning:
      false })
  consumer
    .run({
      eachMessage: async ({ topic, message }): Promise<void> => {
        // message arrived!
      }
```

```
    })
    .then(() => console.log('Consumer running'))
}
```

4.2 Sockets

Socket IO is a library that enables real-time, bidirectional and event-based communication between the browser and the server. It has been exploited to implement the real-time communication between the user and the system, in particular for pushing notifications or sending the environment data to the user. A security layer has been added to socket servers to ensure secure connections. Also for this feature, the JWT token generated when the user logs in has been used.

Listing 3: Socket Server middleware

```
io.use(function (socket, next): void {
    if (socket.handshake.query && socket.handshake.query.token) {
      if (jwtManager.verify(socket.handshake.query.token as string)) next
          ()
    } else {
      next(new Error('Authentication error'))
    }
  })
```

4.3 Object Recognition with YOLO

You only look once (YOLO) is a state-of-the-art, efficient real-time object detection algorithm. YOLO is able to recognise classes of objects and its position in the video.

This algorithm is used within the **Recognition** service to perform object recognition on video stream produced by the cameras, according to active security rules.

In Listing 4 is reported the responsible code for this task.

Listing 4: Python class performing object recognition with YOLO

```
class Recognizer:

    def __init__(self, camera_code: str, rtsp_stream_url: str):
        yolo_resource: str = "app/resources/yolov3"
        with open(f"{yolo_resource}/coco.names", "r") as f:
            self.classes: [str] = [line.strip() for line in f.readlines()
                ]
        self._camera_code: str = camera_code
        self._rtsp_stream_url: str = rtsp_stream_url
        self._is_recognizing: bool = False
        self._recognized_objects: [str] = []
        self._net = cv.dnn_DetectionModel(
            f"{yolo_resource}/yolov3.weights", f"{yolo_resource}/yolov3.
                cfg"
        )

```

```
    def start_recognizing(self) -> None:
        os.environ["OPENCV_FFMPEG_CAPTURE_OPTIONS"] = "rtsp_transport;tcp
            "

        # load capture
        capture = cv.VideoCapture(self._rtsp_stream_url, cv.CAP_FFMPEG)
        capture.set(cv.CAP_PROP_FRAME_WIDTH, 640)
        capture.set(cv.CAP_PROP_FRAME_HEIGHT, 480)

        self._is_recognizing = True
        while self._is_recognizing:
            ret, frame = capture.read()
            if not ret:
                break

            # Detecting objects
            class_ids, confidences, _ = self._net.detect(frame,
                confThreshold=0.5)

            # ... process the detected objects and publish to Kafka

        capture.release()

    def stop_recognizing(self) -> None:
        self._is_recognizing = False

    def is_recognizing(self) -> bool:
        return self._is_recognizing
```

4.4 Media Server

MediaMTX has been used to allow the system to support more than one protocol without implementing a single procedure for each of them. All the cameras connect to the media server, it can be used to retrieve the video streams using different protocols.

In this case, RTSP protocol is used from cameras to produce video streams and from **Recognition** to consume video streams. Instead, WebRTC protocol is used to consume video streams from the **Frontend**.

4.5 WebRTC

To simulate the camera streams, the WebRTC protocol takes place. It has been selected for its ability to stream video and audio in real-time with simplicity. It does not require any plugin or software installation, it is supported by all the major browsers, and it is open-source. Developed by Google, it implements the Google Congestion Control (GCC) algorithm that allows to stream video and audio in real-time with a low latency.

4.6 Single Sign-On

The Single Sign-On (SSO) has been implemented to allow the user to access the system with a single set of credentials. The **Auth** service is responsible for the authentication of the user and the generation of the Json Web Token (JWT) token that will be used to authenticate the user in the other services. The generated token can be validated from each microservice to ensure that the user is authenticated. Following this approach, in case of failure of the authentication service, the user can still access the system until its token validity expires. Moreover, the single point of failure is avoided. This process has been implemented through the use of JWT and sharing the secret key between federated services.

Listing 5: Middleware for the microservices

```
authenticate(req: Request, res: Response, next: NextFunction) {
    const authHeader = req.headers['authorization']
    const token = authHeader && authHeader.split(' ')[1]
    if (token == null) return res.status(HttpStatusCode.FORBIDDEN)

    console.log('Authentication token: ' + token)
    this.jwt.verify(token, this.secret, (err: any, _user: any) => {
      if (err) return res.sendStatus(HttpStatusCode.UNAUTHORIZED)
      next()
    })
  }
```

5 Self-assessment / Validation

5.1 Quality Assurance

Two main tools have been used to ensure the quality of the code produced:

- Prettier is a code formatter that supports many languages. It enforces a consistent style by parsing code and re-writing it according to the configuration rules.
- ESLint is a tool that statically analyses code to find suboptimal patterns and errors.

Both tools have been integrated into the Continuous Integration pipeline (section 5.5) to keep the high-quality code production.

5.2 Architectural Testing

In order to ensure that layers' dependencies are respected, Dependency Cruiser framework has been exploited.

Essentially, the configured rules check that:

- **Domain** layer does not access to any other layer.

- **Application** layer can access only the Domain layer.
- **Presentation** layer can access only Domain and Application layers.

5.3 API Testing

API testing has been performed using Vitest framework.

To be able to execute the tests, the database has been mocked using MongoDB Memory Server.

In Listing 6 is reported a simplified example of an API test.

Listing 6: API testing example

```
describe('GET /devices/', (): void => {
  beforeAll(async (): Promise<void> => {
    await connectToMock()
    await populateDevices()
  })

  describe('GET /devices/sensors', (): void => {
    it('responds with a forbidden status if no auth token is provided',
        async (): Promise<void> => {
      const sensors: Response = await monitoringService.get('/devices/
          sensors')
      expect(sensors.status).toBe(HttpStatusCode.FORBIDDEN)
    })

    it('responds with the sensors otherwise', async (): Promise<void> =>
        {
      const sensors: Response = await monitoringService
        .get('/devices/sensors/')
        .set('Authorization', `Bearer ${TOKEN}`)
      expect(sensors.status).toBe(HttpStatusCode.OK)
      expect(sensors.type).toBe('application/json')
    })
  })

  afterAll(async (): Promise<void> => {
    await disconnectFromMock()
  })
})
```

5.4 Fault Tolerance Testing

Since reliability is an important non-functional requirement (requirement 1.3.1), a specific script has been set up to test the system behaviour in case of faults.

In Listing 7 is reported the main part of the script used to 1. Tearing up the system 2. Tearing down some running services 3. Executing fault tolerance tests.

Nevertheless, script and tests could be improved by adding new different fault scenarios.

Listing 7: API testing example

```
echo "Tearing up the system..."
./deploy.sh > /dev/null 2>&1
sleep 2

tear_down_services "log"
execute_test "monitoring" "log"

tear_down_services "auth"
execute_test "notification" "auth"

tear_down_services "notification"
execute_test "alarm" "notification"

tear_down_services "sensor-1" "sensor-2"
execute_test "monitoring" "sensor"

tear_down_system 0
```

5.5 Continuous Integration

Each test previously described in this section has been integrated into the Continuous Integration pipeline.

6 Deployment

6.1 Prerequisites

- Docker

6.2 Starting the system

To deploy the whole system:

1. Clone the project from Revue.
2. Navigate to the project root.
3. Run the *deploy.sh* script.

Once the system is up and running, the web interface entrypoint is available at http://localhost:8080. The credentials of the example user are:

- Username: `user`
- Password: `user`

To tear down the system, run the *undeploy.sh* script.

In addiction to already cited entities, is present a Zookeeper container that is necessary for Kafka to work properly.

To simulate running devices, two containers for each device are deployed. Note that if more devices are needed, it is necessary to manually add more containers to the docker-compose file in devices' modules (**sensor** and **camera**). It is also necessary, when adding new devices through the web interface, inserting the same device code used in the docker-compose file.

6.3 Starting sub-parts of the system

Other scripts are available to start the system differently, in order to:

- Deploy only some services with their databases, using the *compose-service.sh* script.
- Deploy only databases, using the *compose-db.sh* script.

Usage examples:

- `./scripts/compose-service.sh --up auth monitoring frontend log`
- `./scripts/compose-db.sh --up auth monitoring frontend log`

NB: Scripts have to be launched from the root of the project.

6.4 Configurations

In the root of the project, there is a *.env* that contains the environment variables needed to correctly configure the system. Without modifying the *.env* file, the services will be exposed on different ports according to table 3.

7 Usage Examples

For every usage, the system requires a login for security reasons.

In the simplest scenario, the user can see the section designated for sensor environment data or the camera video streams consultation.

In the more complex scenario, the user can add security rules to detect exceeding values or unauthorized objects in the video streams.

When adding a new intrusion rule, the user can choose the object class to detect, the camera, and the time slot on which the rule will be active.

Instead, when adding a new exceeding rule, the user specifies the range value (for a specific measure), the target sensor and the time slot validity.

Both usages of the system consent to the user to add, delete or modify a device or a security rule configuration.

Moreover, the user can consult the history of produced data or all the notifications received by the system.

Service name	Service Port	Database Port
frontend	8080	—
auth	4000	27017
monitoring	4001	27018
alarm	4002	27019
log	4003	27020
notification	4004	27021
recognition	4005	—
media-server	8554 rtsp	—
kafka	9092	—
zookeper	2181	—

Table 3: Services binding ports

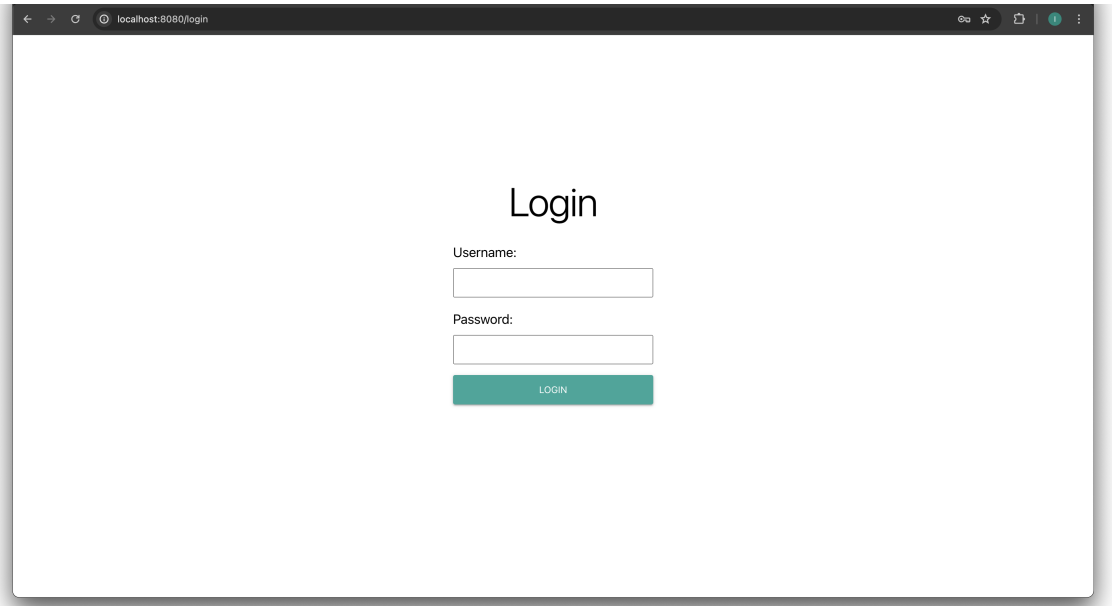

Figure 12: Login view

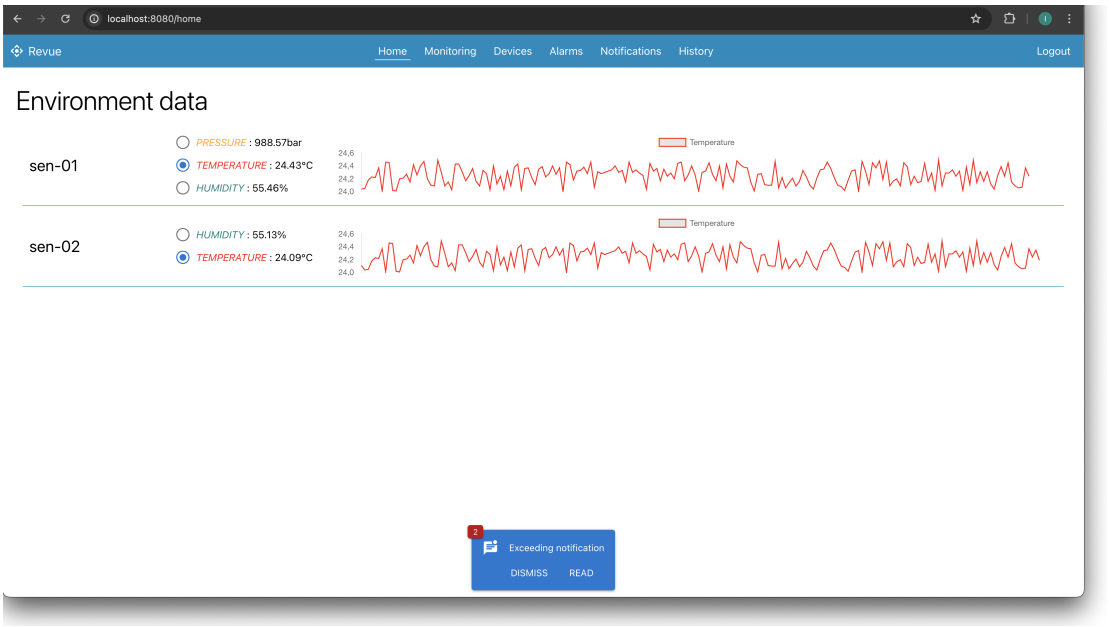

Figure 13: Home view

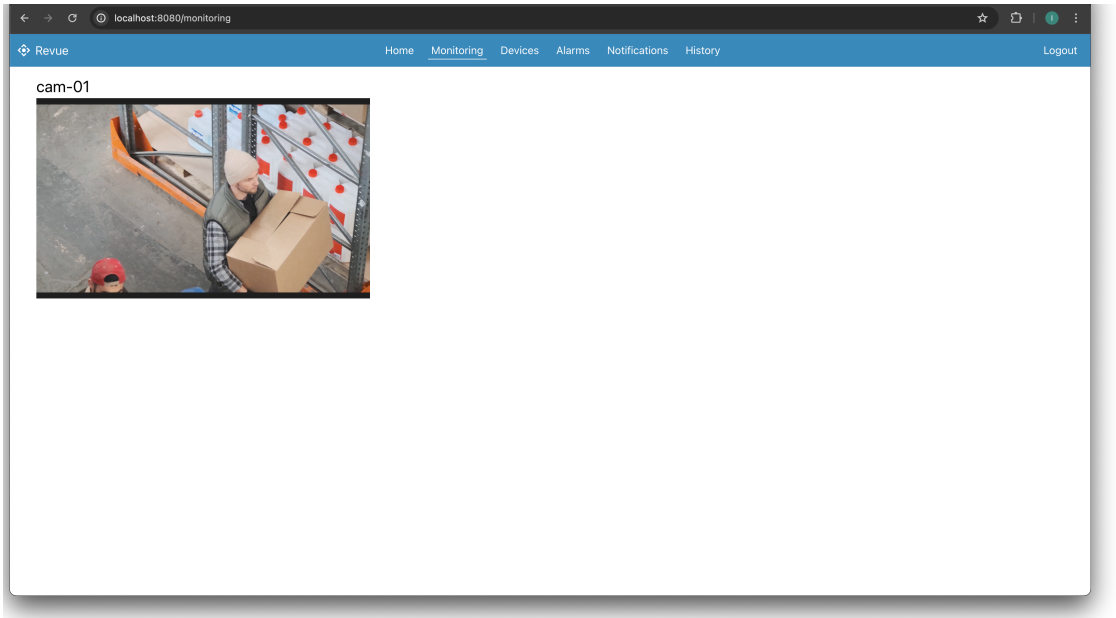

Figure 14: Monitoring view

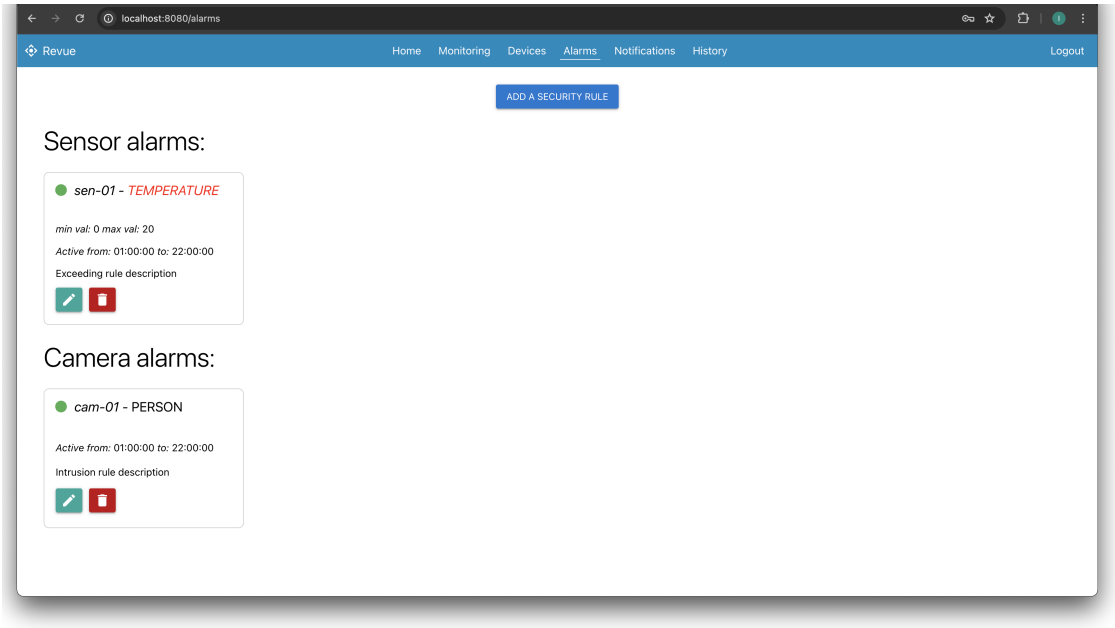

Figure 15: Security rule view

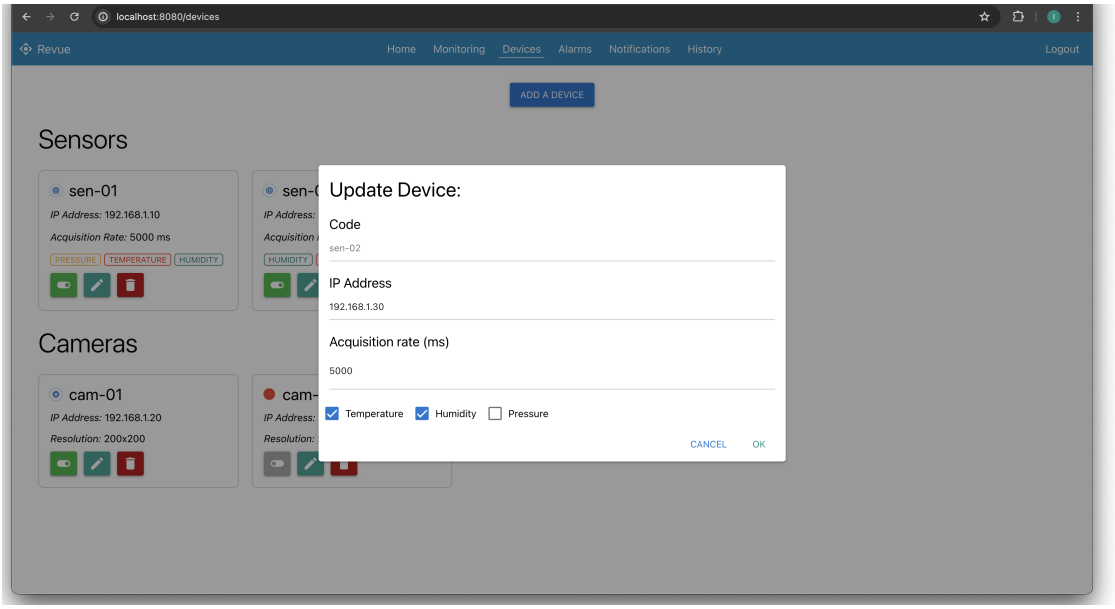

Figure 16: Updating a device

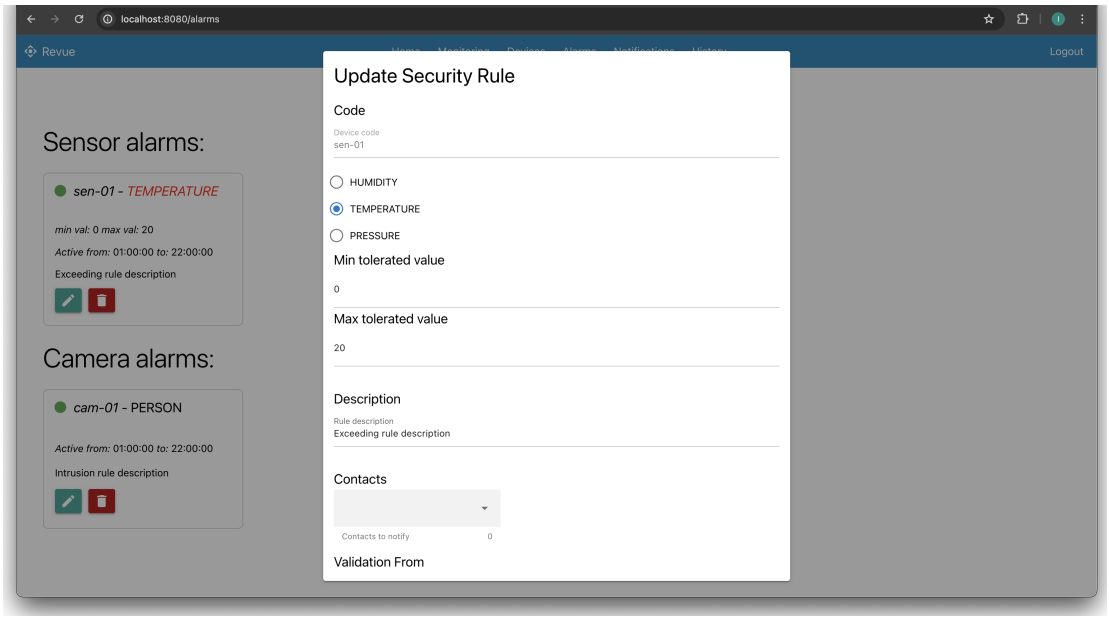

Figure 17: Updating a security rule

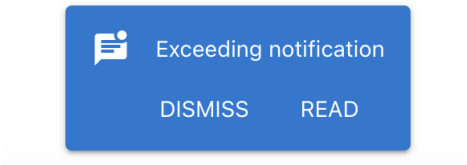

Figure 18: Notification alert

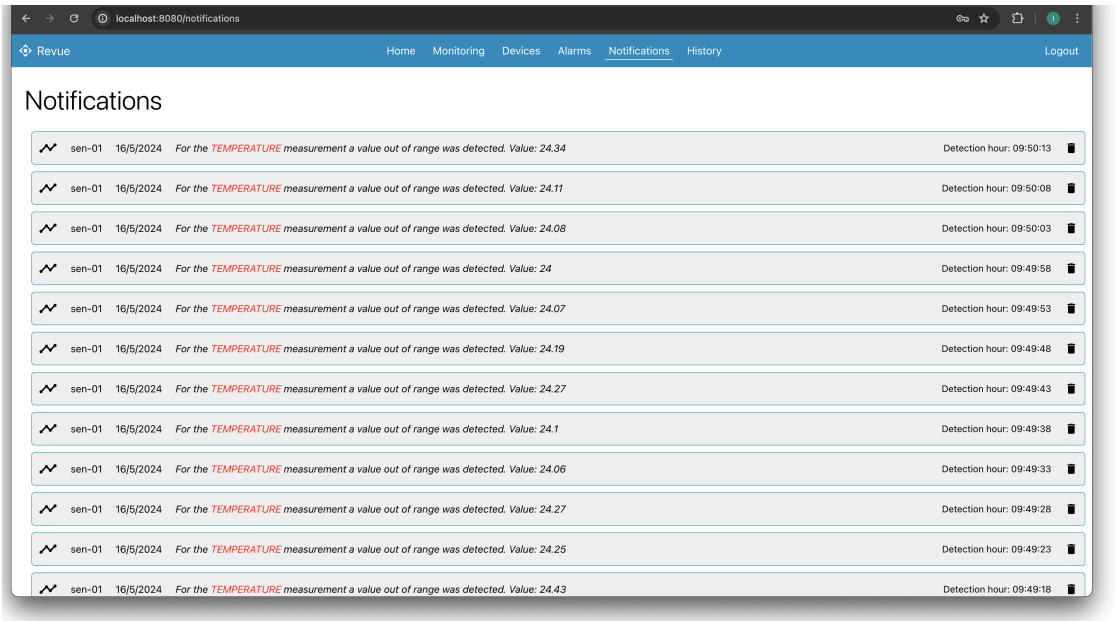

Figure 19: Security rule view

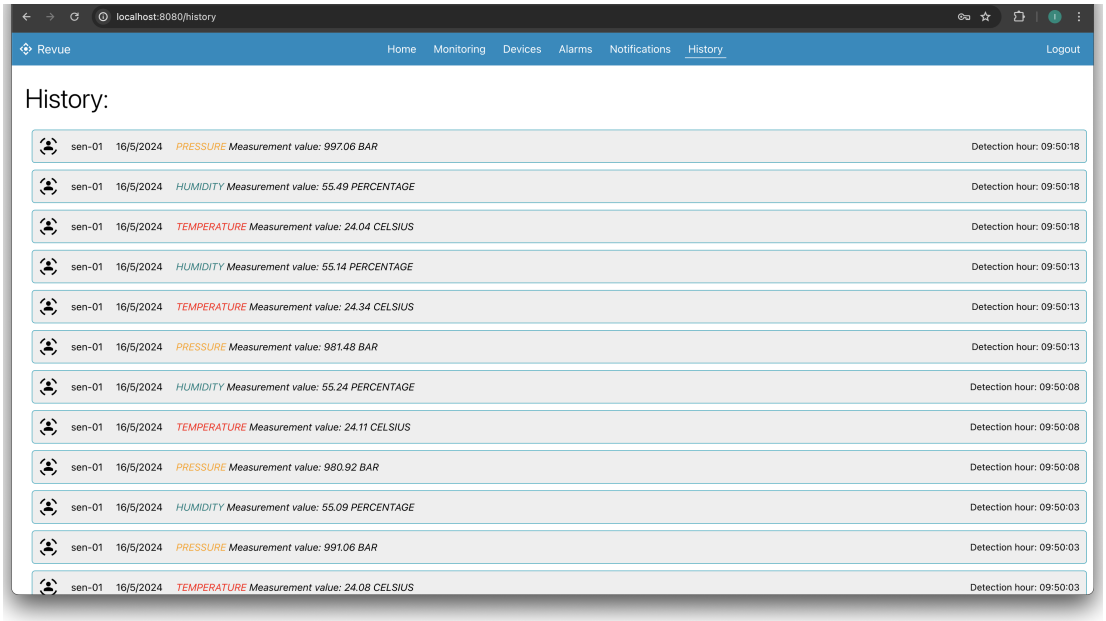

Figure 20: Security rule view

8 Conclusions

8.1 Future Works

Future developments of this system will certainly have to enrich the user experience to exploit all the possible features offered by the system. Certainly a well-designed user section and the introduction of roles could open up to better configurability.

Real-time video consultation could be improved with the introduction of video recording functions as well as a recorded history, which has been put aside for now due to time restrictions.

Furthermore, Web of Things perspective is the objective for next courses' exams. This will lead to the refactor of existent API, the addition of Thing Descriptions, etc. Also, an upgrade to the RESTful API could be considered to make them HATEOAS compliant.

Another point that could certainly be worked on would be a refinement of the system design to make it even more scalable and maintainable. In particular, device management, which is currently part of the monitoring service, surely will be improved with the introduction of a dedicated service.

For the deployment of the system, the use of Kubernetes (in particulare k3s) will be considered to manage the containers in a more efficient way. The goal is to deploy the system in a Raspberry PI cluster.

8.2 What did we learned

This project allowed us to deepen our knowledge of the microservices architecture and to understand the advantages and disadvantages of this approach.

Moreover, it helped us to experiment with new technologies such as Kafka, and more in-depth testing.

Fault tolerance testing has been particularly interesting and has allowed us to understand how to manage the system in case of failure of one or more services, validating the reliability of the system.