Automatic Sensor Management

2024-2025

Noël Jumin
Clément Gauché

Noël Jumin
Clément Gauché

Introduction

In a world that is getting more and more connected, good architectural decisions are the foundation of designing modular, scalable, and high-performance applications. Service-Oriented Architecture, Resource-Oriented Architecture, and Microservices have totally changed the way one designs and deploys software solutions. This report is a follow-up of the project where we studied and applied theoretical foundations on concrete examples.

In this work, we go one step further to describe the design of a complete room management application in a university building. The developed application is based on a distributed architecture, composed of eight microservices, a local MariaDB database, an intuitive user interface, and an automation script, so as to manage the usage scenarios. The main objective here was to apply the principles learned so far in a joined way, while meeting the needs of a real-world environment, making sure modularity and interoperability remain guaranteed, along with system resilience.

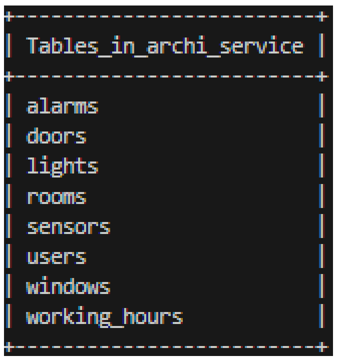
Project definition

The goal of this project is to develop a web application to manage the rooms of an INSA building. This application will be based on several Java services, a Python script acting as a state machine to simulate normal operations, and a graphical interface to interact with and monitor the status of the rooms. Below, we will define the three main components: Microservices, Interface, and Script Automation, which we have integrated.

The scenario we have defined is as follows:
Several rooms are available in a building at INSA. Each room can be equipped with lights, doors, windows, alarms, and presence sensors. All rooms follow the same working hours schedule. During these hours, the doors and windows are automatically opened at the start of the day. When at least one user enters the room, the lights are automatically turned on, and they turn off when no users remain in the room. This behavior relies on data from the presence sensor, which is active when at least one user is present in the room and inactive otherwise. Outside working hours, the doors and windows close, and the lights are turned off. If a user enters a room outside these hours, the presence sensor detects them, and the lights turn on. However, if the room is equipped with an alarm, it will be triggered and start ringing.

Microservices

For this project, we needed to integrate 8 microservices, each corresponding to one of the previously mentioned modules.
To store the data, we decided to implement our own database. Unlike in the previous project, this database is not remote, as it depends on INSA's infrastructure, which could become inaccessible. Therefore, we chose to implement our own local database using MariaDB.



```
+------------------------+
| Tables_in_archi_service |
+------------------------+
| alarms                 |
| doors                  |
| lights                 |
| rooms                  |
| sensors                |
| users                  |
| windows                |
| working_hours          |
+------------------------+
```

Figure 1. Snapshot of the database and the tables present in it

As you can see, each microservice has its own data table, allowing us to separate the data for each microservice.

Working Hours

To simulate a university building as accurately as possible, we decided to implement opening and closing hours that will impact the subsequent services based on the defined schedule. For this, we set two times: the first is the start time of the day, and the second is the end time of the day. These times are configurable by the user. Additionally, a third time is available, providing the current time. This time will be updated by the simulation file, which will increment it to simulate the progression of a normal day.
Here's how this service is implemented at the database level:

```sql
CREATE TABLE IF NOT EXISTS working_hours (
    id BIGINT AUTO_INCREMENT PRIMARY KEY,
    working_hour_name VARCHAR(255) UNIQUE NOT NULL,
    start_time TIME NOT NULL,
    end_time TIME NOT NULL,
    current_time_value TIME NOT NULL
);
```

At the WorkingHours service level, it's possible to add a working hour (for our project, only one working hour will be used), delete it (though it won't be used in our application case), and access various data such as the start and end times, as well as the current time. Additionally, the user will be able to modify the start and end times, as well as adjust the current time in the simulator to let it evolve.

This service is central to the simulation since the script's evolution depends on the working hours.

Rooms

Each of the following elements we will present is linked to a room. In fact, each room will be equipped with lights, doors, etc., and if a room is destroyed, all elements present in it will automatically be deleted from the database as well.
This is possible due to the following lines in the database:

```
CREATE TABLE IF NOT EXISTS rooms (
    id BIGINT AUTO_INCREMENT PRIMARY KEY,
    room_name VARCHAR(255) UNIQUE NOT NULL
);

CREATE TABLE IF NOT EXISTS alarms (
    id BIGINT AUTO_INCREMENT PRIMARY KEY,
    alarm_name VARCHAR(255) UNIQUE NOT NULL,
    room_id BIGINT NOT NULL,
    active BOOLEAN NOT NULL,
    FOREIGN KEY (room_id) REFERENCES rooms(id) ON DELETE CASCADE
);
```

The Rooms microservice is central to the script's progression since it allows, through its id that is distributed across other elements, to locate these elements. This enables the interface to correctly display each element in its designated place and allows the script, for example, to know if a presence sensor has detected someone.
Regarding the Rooms microservice itself, it only contains a few parameters:

- An id, which is automatically assigned upon creation
- A name, provided at creation

This service implements only a POST method to create a room, a GET method to retrieve all existing rooms, and another GET method with /{id} to specify the room whose information we want to retrieve.

Actionneurs

We have added to these rooms some actuators such as locks and shutters, which we will refer to as doors and windows for simplicity. Other actuators are also available, such as alarms and lights.
Each of these actuators can be created, deleted, and modified, mainly to change their active or inactive state.
As mentioned earlier, each actuator belongs to a room, so they are dependent on it. This makes it possible to locate them in space, as well as to delete them in a cascade if the room is deleted.

Sensors

Officially, we only have one sensor, which is a presence sensor. In our use case, it will detect if a user has entered the room, and activate various actuators accordingly. There is no significant change compared to the actuators, we can still create, delete, and modify their state.
The real difference between the Sensors service and the actuators services is the way they will be called from the frontend or the scenario code.
At the database level, they are strictly identical:

```
CREATE TABLE IF NOT EXISTS lights (
    id BIGINT AUTO_INCREMENT PRIMARY KEY,
    light_name VARCHAR(255) UNIQUE NOT NULL,
    room_id BIGINT NOT NULL,
    active BOOLEAN NOT NULL,
    FOREIGN KEY (room_id) REFERENCES rooms(id) ON DELETE CASCADE
);

CREATE TABLE IF NOT EXISTS sensors (
    id BIGINT AUTO_INCREMENT PRIMARY KEY,
    sensor_name VARCHAR(255) UNIQUE NOT NULL,
    room_id BIGINT NOT NULL,
    active BOOLEAN NOT NULL,
    FOREIGN KEY (room_id) REFERENCES rooms(id) ON DELETE CASCADE
);
```

Graphical interface

To ensure a more user-friendly experience, we decided to implement a graphical interface. This interface allows several actions. In the next steps we will analyze each possibility it offers.

Setup

At each startup, the frontend calls a Python service that resets the database, allowing for the deletion of previously created elements and returning to a clean slate.
Users are then given the option to choose the building's working hours.

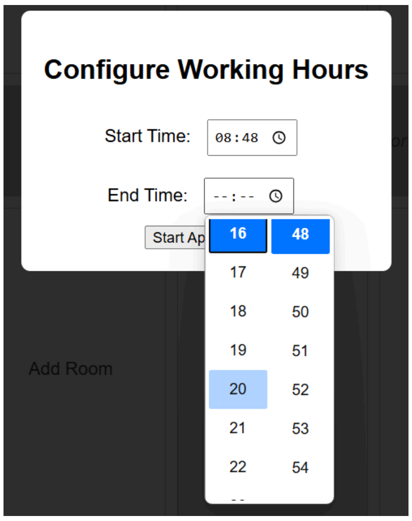


Figure 2. Snapshot of working hours configuration at startup

Upon validation, this sends a creation request to the working hours service. After creation, however, it remains possible to modify the start and end times of the day directly from the interface. This interface also provides an indication of the time of day, whether it is outside working hours (Moon) or within them (Sun).

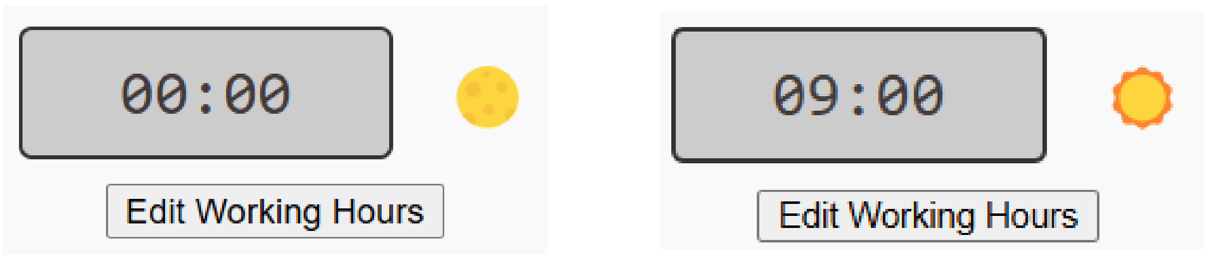


Figure 3. Snapshot of working hours display in the interface

What it displays

The interface is presented like this, with different rooms that can be created through a button.

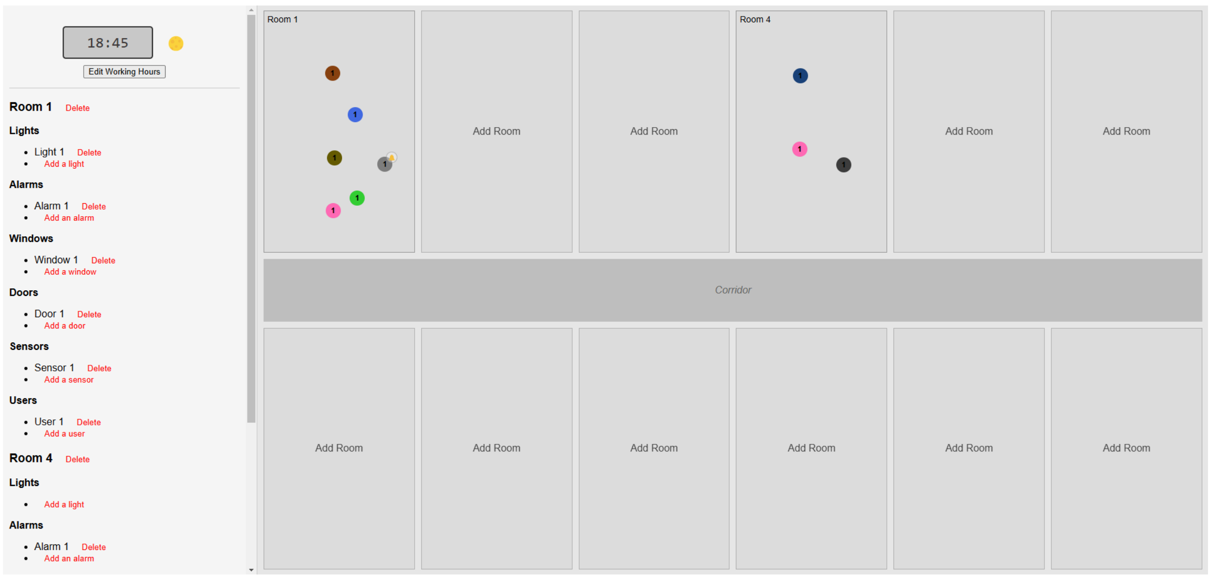


Figure 4. Snapshot of the complete interface with different elements added in different rooms

Once the room is created, elements can be added by drag and drop and managed for deletion directly from the left-hand zone. Each of these elements is linked to a room, which provides more information about the location of each one.
Additionally, as you can see, when hovering over one of the elements, it is possible to see their states, like this:

- A light can be in the “on” (💡) or “off” (⭕) state.
- An alarm can be in an “active” (🔔) state, meaning it’s ringing, or in an “inactive” (🔕) state.

The same functionality has been implemented for windows, doors, and presence sensors.
For a more direct view of these states, the circle representing each element will be more or less bright, depending on whether it is active or not.
Users can also change the state of each actuator by clicking on them, but this is not possible for the presence sensor because we have chosen that it is not possible to change a sensor state manually.

Script automatization

To validate the proper functioning of the project, we have defined a scenario that outlines use cases to verify the correct operation of the various services.
The scenario is as follows, described in the scenario.md file:

```
# Scénario de gestion des lumières, portes, fenêtres et alarmes

## Pendant les heures de travail (entre `start time` et `end time`)
- Les **lumières** s'allument **uniquement** si un capteur de présence est actif.
- Les **alarmes** sont **désactivées**.

## À la fin des heures de travail (à `end time`)
- Les **fenêtres** et les **portes** se **ferment**.

## Pendant les heures hors travail (entre `end time` et `start time`)
- Les **lumières** s'**éteignent**.
- Les **alarmes** sont **désactivées** tant qu'**aucun** capteur de présence ne passe à actif, ou qu'**aucune** porte ou fenêtre n'est ouverte.
- Le **capteur de présence** s'**active** si au moins un utilisateur est présent dans la pièce.

## À la fin des heures hors travail (à `start time`)
- Les **fenêtres** et les **portes** s'**ouvrent**.
```

You can find the scenario's implementation in the scenario.py *file*.

Simulation initialization

After running the *run.bat* file, we arrive at the HTML API, where we can edit the working hours. For instance, I set them from 8:00 AM to 5:00 PM.
We then proceed to the initial HMI.

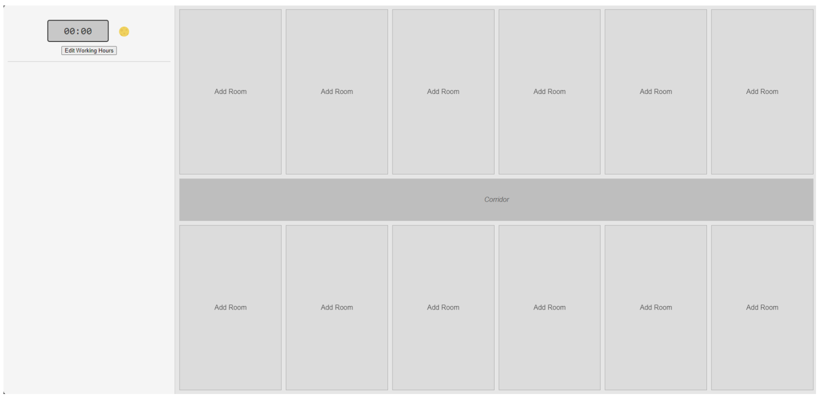


Figure 5: Snapshot of the initial interface

To launch the backend and have the current time increment, we run the *scenario.py* script. If it runs correctly, the current time should increment by 15 minutes every 5 seconds.

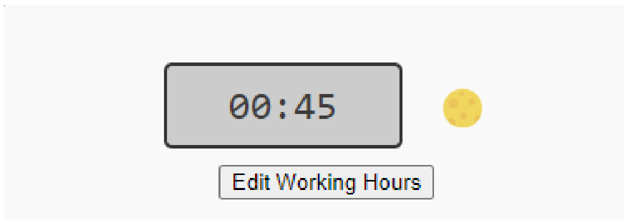
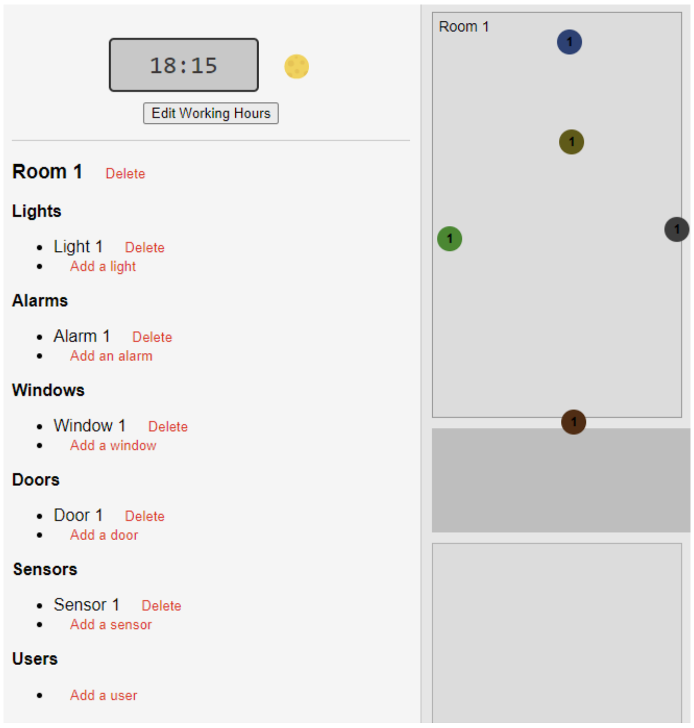


Figure 6: Snapshot of the current time evolving

Through the interface, I add the elements necessary for the scenario (as explained in the previous section).



Room 1 Delete

Lights

- Light 1 Delete
- Add a light

Alarms

- Alarm 1 Delete
- Add an alarm

Windows

- Window 1 Delete
- Add a window

Doors

- Door 1 Delete
- Add a door

Sensors

- Sensor 1 Delete
- Add a sensor

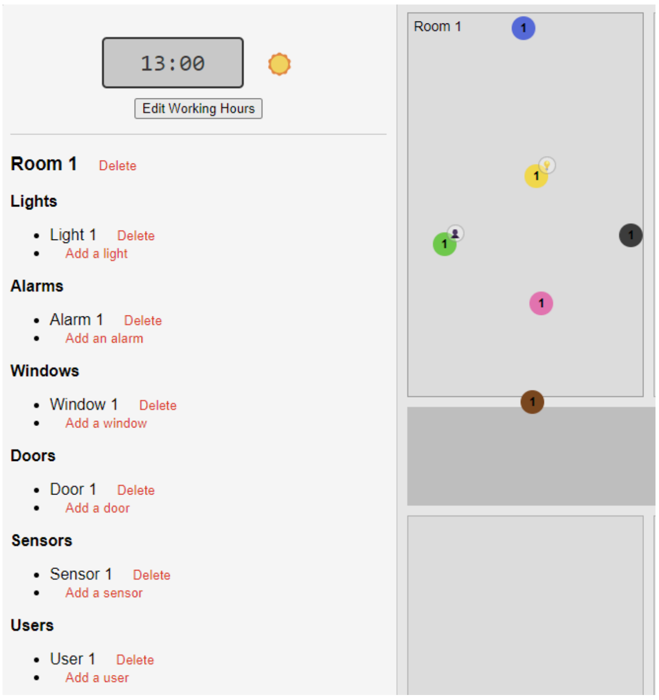
Users

- Add a user

Figure 7: Snapshot of the interface with room1 setted for the scenario

Validation of two main scenario cases

1. During working hours, if a user is present, the presence sensor detects him and the lights turn on.



13:00

Edit Working Hours

Room 1 Delete

Lights

- Light 1 Delete
- Add a light

Alarms

- Alarm 1 Delete
- Add an alarm

Windows

- Window 1 Delete
- Add a window

Doors

- Door 1 Delete
- Add a door

Sensors

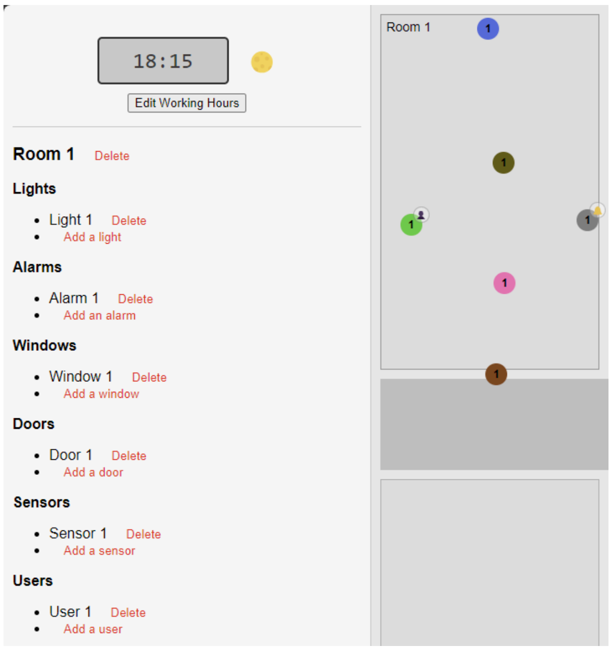
- Sensor 1 Delete
- Add a sensor

Users

- User 1 Delete
- Add a user

Figure 8: Snapshot the case of a user present during the working hours

2. Outside working hours, if a motion sensor is triggered or if a window or a door is opened, the alarm is activated.



18:15

Edit Working Hours

Room 1 Delete

Lights

- Light 1 Delete
- Add a light

Alarms

- Alarm 1 Delete
- Add an alarm

Windows

- Window 1 Delete
- Add a window

Doors

- Door 1 Delete
- Add a door

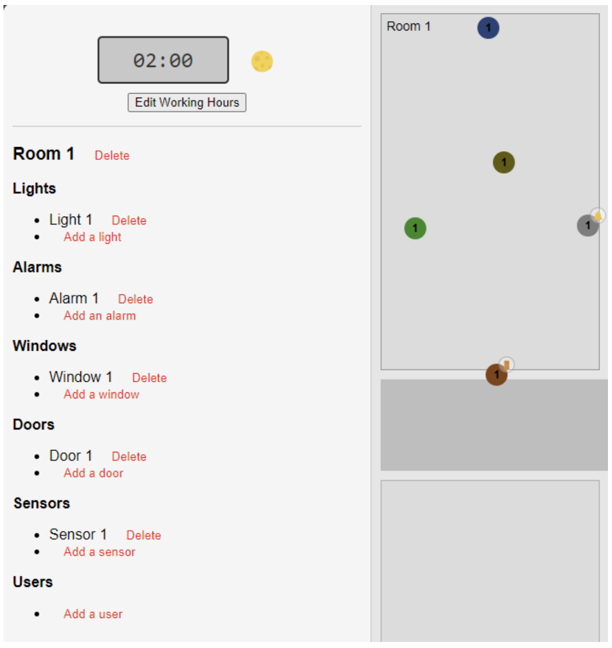
Sensors

- Sensor 1 Delete
- Add a sensor

Users

- User 1 Delete
- Add a user

Figure 9: Snapshot of a user present outside working hours



02:00

Edit Working Hours

Room 1 Delete

Lights

- Light 1 Delete
- Add a light

Alarms

- Alarm 1 Delete
- Add an alarm

Windows

- Window 1 Delete
- Add a window

Doors

- Door 1 Delete
- Add a door

Sensors

- Sensor 1 Delete
- Add a sensor

Users

- Add a user

Figure 10: Snapshot of a door oppened outside working hours

It is also possible to test the scenario in which windows and doors automatically close at the end of the working hours and reopen at the end of the non-working hours.

You can have fun testing them by yourself by manipulating the interface.

Conclusion

The present project extends the concepts debated in our first report, where we analyzed SOA, ROA, and microservices paradigms. During this second phase, we implemented a room management application that included the realization of eight different microservices, each one responsible for one function: management of schedules, sensors, actuators, and automated scenarios. A local MariaDB database was used in order to avoid problems with data consistency and interservice communications, stemming from the use of a remote database during the previous project.
Moreover, this work deepened our knowledge about DevOps principles, mainly by automating crucial tasks such as database initialization and running predefined scenarios. In this respect, these tools and methodologies have enabled us to meet the challenges inherent in a more complex and realistic distributed system while reinforcing both our technical and organizational know-how.
This project picks up where the first report left off, providing an implementable solution for real-world challenges. Scalability, interoperability, and robustness of the system have been major focus points, modularity, and methodical design for particular needs.

Github link

Here the github link for the project presented:
https://github.com/NoNo47400/Projet_Automatic_Management_Service