National Institute of Applied Sciences
Department of Electronic and Computer Engineering
Innovative & Smart Systems

Sandra BEJAOUI , Imane EL GHAIT

5A Innovative Smart Systems

Supervised by : Nawal GUERMOUCH

INSA Room Management

2025/2026

Table of contents

1. Introduction

1.1. Context

The management of rooms at INSA currently relies on manual operations : opening or closing doors and windows, controlling lighting, and monitoring occupancy. These manual tasks are time-consuming and often lead to energy waste or security gaps.

With the rise of IoT (Internet of Things) and modern distributed architectures, these tasks can be automated using a network of sensors and actuators coordinated by intelligent software. This project proposes a Proof-of-Concept (PoC) to automate room management using a microservice architecture.

1.2. Project Objectives

The main objective of this project is to design and develop an application that can automatically manage INSA's rooms using data collected from simulated or real sensors. The application should be able to:

- Analyze the state of a room (presence, working hours, doors/windows open or closed, lights, etc.).
- Trigger automatic actions on actuators based on predefined scenarios.
- Use a microservices architecture for scalability, modularity, and testability.
- Store historical actions for monitoring .

1.3. Selected Scenario

For this mini-project, we chose the scenario:

> *"Closing doors, windows, and lights outside working hours, and triggering an alarm if presence is detected after 10 PM or before 8 AM."*

This scenario allows implementing multiple microservices interacting with each other while maintaining simplicity for a proof-of-concept.

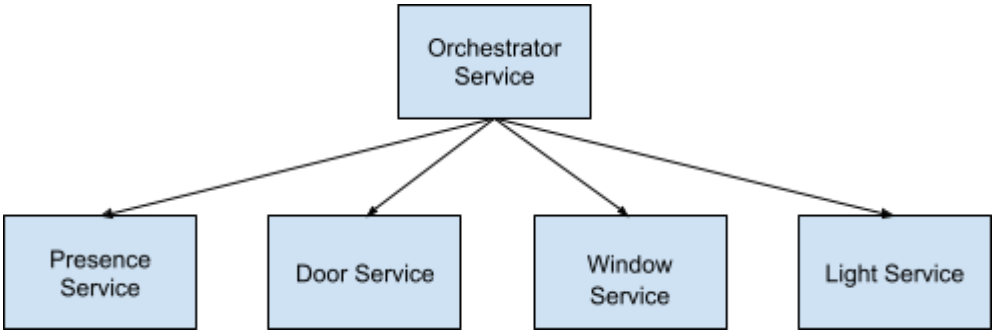
2. Global Architecture of the Application

2.1. General principle

The system is composed of specialized microservices that handle specific functionalities, including:

- Reading sensor data (Presence).
- Controlling actuators (doors, windows, lights).
- A central orchestrator that evaluates the room state and triggers actions.

2.2. Microservices architecture



The application architecture includes the following microservices :

- **Presence Service:** Monitors presence in each room.
- **Door Service:** Controls doors.
- **Window Service:** Controls windows.
- **Light Service:** Controls lights.
- **Orchestrator Service:** Central brain that coordinates actions.

2.3. Communication between services

The architecture relies on a decoupled communication model where each microservice operates independently but interacts through a centralized logic.

2.3.1 RESTful Architecture (Spring Boot + REST Controllers)

The communication is based on the REST (Representational State Transfer) architectural style. Each microservice (Door, Window, Light,Presence) exposes a set of REST Controllers.

- **Protocol :** We use HTTP as the transport protocol.
- **Data Exchange:** Information is exchanged using JSON payloads. This ensures lightweight data transfer and easy integration between the Spring Boot back-end services.

2.3.2 Direct Service Interaction (Orchestration)

The Orchestrator acts as the central intelligence of the system. Instead of services talking to each other (Choreography), we implemented the Orchestration pattern :

- The Orchestrator triggers requests to the actuators' endpoints (e.g., POST /api/doors/action).
- This approach centralizes the business logic (the INSA room management rules), making it easier to monitor and modify the automation workflow in one single place.

2.3.3 Static Service Discovery (PoC Approach)

For this Proof-of-Concept, we implemented Static Service Discovery.

- **Implementation:** Since the environment is controlled and the number of instances is fixed, the network locations (Domain Name /Port) of the services are defined in the application.properties configuration file of the Orchestrator.
- **Justification:** While dynamic discovery (like Netflix Eureka) is preferred for scalable production environments, a static configuration is sufficient for this PoC to demonstrate the functional communication between the "logic" and the "hardware simulation" without adding unnecessary infrastructure overhead. We did however later on implement a dynamic discovery with Eureka.

3. Detailed description of microservices

This section describes the internal structure and responsibilities of each microservice developed for the INSA room management system.

3.1. Door Service

- The **DoorMS** is a specialized microservice designed to simulate and manage the state of a room's door. It follows a decoupled architecture using the **Controller-Service-Model** pattern to separate concerns and ensure maintainability

Packages :

- DoorController → Defines the REST API endpoints. It handles incoming HTTP requests from the Orchestrator and maps them to service actions. POST /door/close : Commands the door to lock/close, and GET /door/state : Returns the current status ("CLOSE", "OPEN").
- DoorService → Contains the business logic. It manages the lifecycle and state transitions (Opening/Closing) of the door object.
- DoorState → The model, It represents the physical status of the door using a boolean attribute (closed/opened).
- DoorMsApplication → The entry point of the Spring Boot application, annotated with @EnableDiscoveryClient to allow automatic registration with the Eureka Server.

In a nutshell, the microservice exposes three main operations through its REST interface :

1. **Status Retrieval (**GET /door/status**)**: Returns the current state of the door in JSON format.
2. **Open Action (**POST /door/open**)**: Updates the internal state to closed = false.
3. **Close Action (**POST /door/close**)**: Updates the internal state to closed = true.

3.2. Window Service and Light Service

These two microservices follow the exact same architectural pattern as the DoorMS, demonstrating a modular and standardized design across the entire system. Each service is responsible for managing a specific type of hardware state within the room.

3.2.1. Light Microservice (LightMS)

The LightMS handles the illumination state. While the structure is the same, the terminology and logic are adapted for lighting.

Packages :

- Model : Defines a LightState with a boolean attribute on (defaulting to false).
- Service : Provides methods turnOn() and turnOff() to modify the state.
- Controller : Exposes the following REST endpoints:
    - GET /light/status: Returns the current lighting state.
    - POST /light/on: Switches the light to the "On" state.
    - POST /light/off: Switches the light to the "Off" state.

3.2.2. Window Microservice (WindowMS)

The WindowMS manages the room's automated windows or shutters.

Packages :

- Model: Defines a WindowState with a boolean attribute closed (defaulting to true).
- Service: Provides methods openWindow() and closeWindow().
- Controller: Exposes the following REST endpoints:
    - GET /window/status: Returns the current window status.
    - POST /window/open: Sets the window state to "Open".
    - POST /window/close: Sets the window state to "Closed".

3.2.3 Common Technical Features

For both services, the implementation is similar to DoorMs and includes :

- **@EnableDiscoveryClient**: This annotation in the main Application class ensures that both services automatically register their presence on the **Eureka Server** upon startup.
- **Decoupled Logic**: By separating the Controller from the Service, we ensure that the business logic (the state change) is independent of the communication protocol (HTTP/REST).
- **JSON Communication**: All data exchanged with the Orchestrator is serialized in JSON format, facilitating easy integration.

3.3. Presence Service

The **PresenceMS** acts as the "eyes" of the system. While the actuator services (Door, Light, Window) receive orders to change the environment, the Presence

microservice provides the critical data that the Orchestrator needs to make informed decisions.

Packages :

- **PresenceController** → Provides endpoints to check for activity and manually simulate sensor triggers.(e.g GET /presence/status:Returns a boolean (true/false) indicating if someone is detected.
- **PresenceService** → Maintains the current state of occupancy in the memory**.**
- **PresenceState** → The Data Model representing whether movement is currently detected (detected boolean).
- **PresenceMsApplication** → Enables the service to be discoverable by the Eureka Server via @EnableDiscoveryClient

Functional Logic & Simulation :

Since this project is a Proof-of-Concept, the PresenceMS includes endpoints to simulate real-world sensor behavior:

- Status Check (**GET /presence/status**): Used by the Orchestrator to decide if it is safe to close the room or if an alarm should be triggered.
- Simulate Detection (**POST /presence/detec**t): Simulates a person entering the room or moving, setting detected to true.
- Clear Presence (**POST /presence/clear**): Simulates the room becoming empty, setting detected to false.

3.4. Orchestrator Service

The **OrchestratorMS** is the master component of the architecture. Unlike peripheral microservices that manage a single state (a door or a light), the Orchestrator coordinates actions across all services, manages the MySQL database persistence, and handles time-based automation logic.

3.4.1 Internal Package :

Controller→ Exposes high-level API endpoints (e.g., /night, /morning) to trigger full room scenarios.

Service → Contains the core business logic (OrchestratorService) and the automated task engine (TimerService).

Model → Defines persistent entities (Room, RoomState, ServiceUrl) mapped to MySQL tables.

Repository → The class manages the flow of data between the Java code and the **MySQL database** through three repositories:

- RoomRepository: To know which rooms exist.

It is linked to the following table, as can be seen each room has a name, a certain number of doors, windows and lights that need to be turned on or off :

id	name	doors	windows	lights
1	gei102	2	3	4
2	gei107	1	2	2
3	gei105	4	10	20
4	gei12	1	2	3
5	gei13	3	4	8
NULL	NULL	NULL	NULL	NULL

Figure 2 : insa_rooms table in MySQL Benchwork framework

- ServiceUrlRepository: To find the addresses of the other microservices.

It is linked to the following table :

id	service_name	url
1	light_service	http://LIGHTMS
2	door_service	http://DOORMS
3	window_service	http://WINDOWMS
4	presence_service	http://PRESENCEMS
NULL	NULL	NULL

Figure 3 : service_urls table in MySQL Benchwork framework

- RoomStateRepository: To save the history of every action taken.

It is linked to the following table :

id	room_name	service_name	state	timestamp
1	gei13	door_service	{"closed":true}	2026-01-14 14:59:48
2	gei13	door_service	{"closed":true}	2026-01-14 15:02:00
3	gei13	window_service	{"closed":true}	2026-01-14 15:02:00
4	gei13	light_service	{"on":false}	2026-01-14 15:02:00
5	gei13	door_service	{"closed":true}	2026-01-14 15:03:00
6	gei13	window_service	{"closed":true}	2026-01-14 15:03:00
7	gei13	light_service	{"on":false}	2026-01-14 15:03:00
8	gei105	door_service	{"closed":true}	2026-01-14 15:06:40
9	gei105	window_service	{"closed":true}	2026-01-14 15:06:40
10	gei105	light_service	{"on":false}	2026-01-14 15:06:40
11	gei13	door_service	{"closed":true}	2026-01-14 15:14:03
12	gei13	window_service	{"closed":true}	2026-01-14 15:14:03
13	gei13	light_service	{"on":false}	2026-01-14 15:14:03

Figure 4 : room_state table in MySQL Benchwork framework

3.4.2 Decision Engine and Automation

The Orchestrator is more than a simple API, it is an autonomous system capable of independent action thanks to TimerService that allows us to automate certain tasks.

1- TimerService :

The TimerService is the heartbeat of the system. In a microservices environment, automation often requires a component that tracks time without waiting for an external user request.

- Autonomous Triggering : Using Spring Boot's @Scheduled annotation, this service operates independently of any API call. It acts as an internal "alarm clock" for the building.
- Cron Expressions : The service uses "Cron" syntax to define execution patterns. the cron ="*/20 * * * * ?" allows for rapid testing by triggering the logic every 20 seconds for testing purposes, while cron ="0 0 22 * * ?" targets the actual 10:00 PM security deadline.
- Scalability : By calling nightModeForAllRooms(), the TimerService ensures that the logic is not limited to one room but iterates through every entry in the insa_rooms database table, ensuring a global building sweep.

2- The OrchestratorService :

The Orchestrator is the most important part of the code because it contains the intelligence part. While other microservices (light , door , window) just wait for orders , the Orchestrator make the decisions :

The service acts as a bridge between the Database and the communication with other microservices , it uses roomRepository , serviceUrlRepository and roomStateRepository to fetch room data , find service addresses and save the history of every action taken.

The RestTemplate allows the Orchestrator to send HTTP requests to other microservices

Every time the sendCommand method is executed, the service automatically creates a RoomStateRespository. It records:

- Which room was affected.
- Which service was called.
- What the result was (e.g., "closed", "on", or "Alarm activated").
- The exact timestamp of the event.

4. Database Implementation (MySQL)

4.1. Role of the Database

In a microservices architecture, data persistence is crucial for moving from a volatile "in-memory" state to a permanent record. In this project, the MySQL database serves three main purposes:

- Room State (Mandatory): It keeps a reliable audit trail of every decision made by the Orchestrator (when a door was closed or an alarm triggered).
- Persistence : It ensures that even if the microservices are restarted or the power is cut, the history of what happened in the INSA rooms is preserved.

4.2. Database Structure & Schema

For this Proof-of-Concept, we designed a streamlined schema focused on the Room State table. We used Spring Data JPA to automatically generate the table structure from our Java Entity.

4.3. Integration: Linking Services to the Database

4.3.1. Technical Implementation (Spring Data JPA)

The connection between the OrchestratorMS and the MySQL Server is handled by the spring-boot-starter-data-jpa dependency.

=> The Entity (Room State): A Java class annotated with @Entity that maps exactly to the database table.

4.3.2. Connectivity & Credentials (INSA Server)

Since the database is hosted on the INSA server (srv-bdens.insa-toulouse.fr), the configuration in application.properties is necessary.

```
spring.application.name=OrchestratorMS
server.port=8086


spring.datasource.url=jdbc:mysql://srv-bdens.insa-toulouse.fr:3306/projet_gei_005
spring.datasource.username=projet_gei_005
spring.datasource.password=oWeib2ai


spring.jpa.hibernate.ddl_auto=none
spring.jpa.show-sql=true
spring.jpa.database-platform=org.hibernate.dialect.MySQLDialect
```

4.4. Workflow of a Database Record

This workflow example illustrates how a single business decision in the Java code is transformed into a persistent entry in our MySQL database.

- **Trigger:** The TimerService detects a scheduled interval (e.g., every 20 seconds during testing or 22:00 in production) and calls orchestratorService.nightModeForAllRooms().
- **Logic:** The OrchestratorService executes the sendCommand method. It calls the PresenceMS via RestTemplate. If presence is detected, the logic determines the result: "Presence detected! Alarm activated".
- **Instantiation:** A new **RoomState** object is instantiated in Java. The service populates the object using the setter methods:
    - log.setRoomName("gei15");
    - log.setServiceName("presence_service");
    - log.setState("Presence detected! Alarm activated");
- **Persistence:** The Orchestrator calls the **RoomStateRepository** using the command: roomStateRepository.save(log);.

There is also a orchestratorService.morningModeForAllRooms() function that is called at 8 am but for simplicity reasons, we will only mention one use case.

5. Communication and orchestration

5.1. Service Discovery: From Static to Dynamic (Eureka)

One of the main challenges in a microservices architecture is "Service Discovery" (knowing where each service is located on the network).

5.1.1. Current Implementation: Static Discovery

For this Proof-of-Concept, we implemented **Static Service Discovery**. The network locations (IP addresses and ports like localhost:8081) are hardcoded in the Orchestrator's configuration. This approach was chosen for its simplicity and speed of deployment during the initial development phase.

Here are the ports for all of our microservices :

- DoorMS: 8081
- WindowMS: 8082
- LightMS: 8083
- PresenceMS: 8084
- OrchestratorMS: 8086
- Eureka: 8761

5.2.2. Advanced Evolution: Netflix Eureka

To make the system production-ready, we studied the integration of **Netflix Eureka**.

- **What is Eureka?** It is a "Service Registry." Every time a microservice (like DoorMS) starts, it automatically registers its name and address in the Eureka Server.
- **Why use it?** It eliminates the need for hardcoded ports. If a service changes its port or if we start multiple instances of the same service, the Orchestrator simply asks Eureka: *"Where is the 'door-service'?"* and Eureka returns the correct address.
- **Project Benefit:** Using Eureka would make our INSA Room Manager more resilient. If a service crashes and restarts on a different port, the Orchestrator would still be able to find it without any code modification.

5.2. Linking the Database to Services

The persistence layer is integrated specifically into the **OrchestratorMS** to centralize the "Journaling" of the system.

- **Driver:** We used the mysql-connector-j to bridge Java and the remote INSA MySQL server.
- **JPA/Hibernate:** We implemented **Spring Data JPA**. This allows the Orchestrator to interact with the database using Java objects instead of writing complex SQL queries manually.
- **Automatic Schema Update:** We enabled hibernate.ddl-auto=update, which automatically created the RoomState table on the INSA server as soon as the Orchestrator was launched.

6. Validation & Testing Phase

To ensure the reliability and the proper synchronization of the distributed system, a multi-layered testing strategy was implemented.

6.1. Unitary Testing (Individual Service Validation)

*The first phase focused on the isolation of each microservice. Each actuator (Door, Light, Window) and the Presence sensor were tested independently to ensure their internal **Controller-Service-Model** logic was functional.*

- ***Method:** Manual HTTP requests using **cURL** or a **Web Browser**.*
- ***Example:** Sending a request to `http://localhost:8081/door/status` to verify that the service is alive and returning the correct JSON state.*

6.2. Logic Testing (Orchestration Flow)

*Once the individual nodes were validated, we tested the "Brain" of the system. This phase ensures that the **OrchestratorMS** correctly interprets business rules before sending commands.*

- ***Method:** Triggering the `/orchestrator/night` endpoint manually via cURL.*
- ***Verification:** Checking if the Orchestrator correctly switches between the "Automation" path (closing devices) and the "Security" path (activating alarms) based on the sensor data it receives.*

6.3. Integration Testing (Chain of Command)

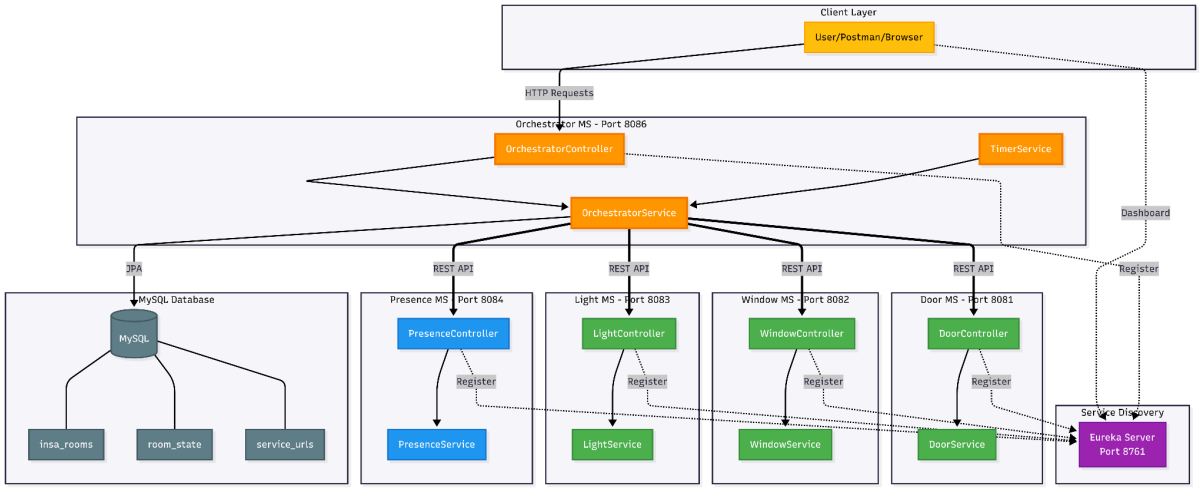
*The final phase validated the full ecosystem. We monitored the **inter-service communication** and the persistence layer simultaneously.*

- ***Method:** Simultaneous monitoring of **5 terminal windows** (one per service + Eureka).*
- ***The "Chain of Command" Verification:***
  1. ***Trigger:** The `TimerService` reaches the scheduled time.*
  2. ***Discovery:** The Orchestrator locates services via the **Eureka Registry**.*
  3. ***Sensing:** The Orchestrator queries `PresenceMS`.*
  4. ***Action:** Commands are dispatched to actuators (`DoorMS`, `LightMS`, etc.).*
  5. ***Persistence:** The final result is successfully saved into the **MySQL Database**.*

7. Diagrams

7.1. Global Architecture Diagram

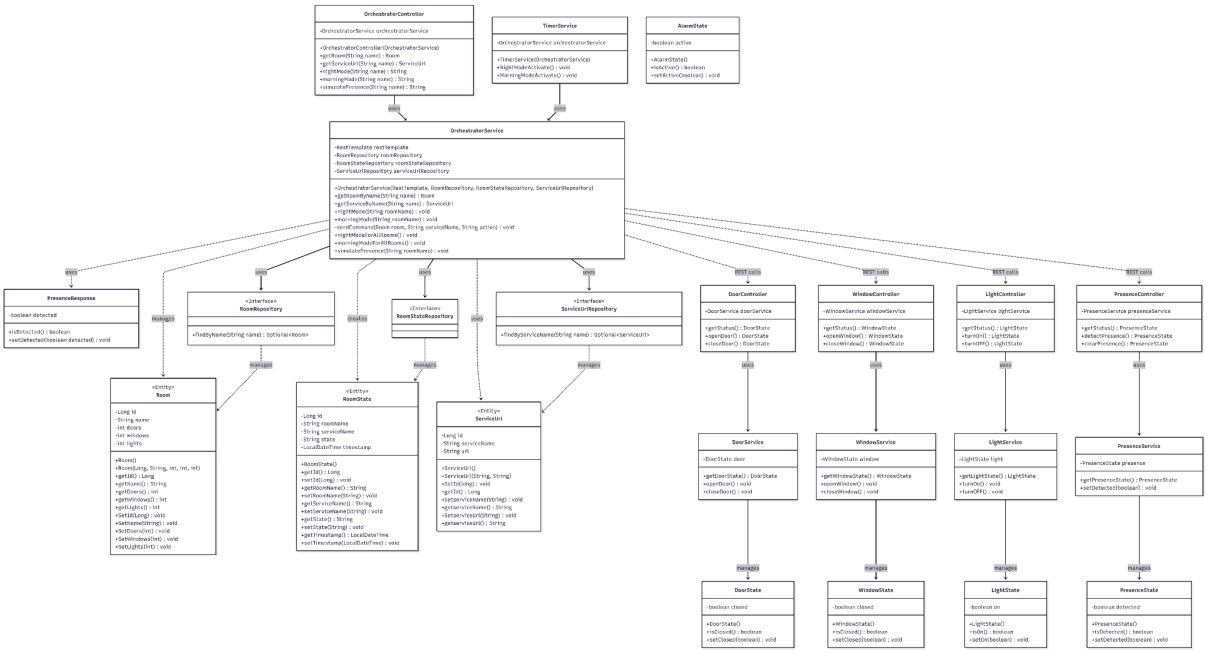
This diagram represents the high-level ecosystem of the project. It illustrates the decentralized nature of the system where independent services communicate through a central hub.



Microservices Layer: DoorMS, LightMS, WindowMS, and PresenceMS acting as independent nodes.
Orchestration Layer: The OrchestratorMS acting as the "Brain" and the Eureka Server acting as the "Registry".
Data Layer: The MySQL Database provides persistence for room configurations, service locations, and event logs.

7.2. Class Diagram

The following class diagram illustrates the complete object-oriented architecture of the INSAroom Manager system. It shows the structural organization of classes across all microservices, their attributes, methods, and relationships. The diagram follows a layered architecture pattern with clear separation between Controllers, Services, Models, and Repositories.



7.3. Sequence Diagram

The sequence diagrams illustrate the dynamic interactions between the different components of the INSAroom Manager system during the execution of main use cases. They show how microservices collaborate to automate intelligent room management at INSA.

Scenario 1 : Automatic night mode activation

Night mode is triggered automatically every 20 seconds (test configuration) by the **TimerService**. In production, this would be triggered at 10:00 PM (22:00). This mode secures all rooms by closing doors and windows, turning off lights, and checking for unauthorized presence. Here goes the execution flow :

1. **TimerService** triggers via `@Scheduled` annotation (`cron = "*/20 * * * * ?"`)
2. **OrchestratorService** calls `nightModeForAllRooms()`
3. **Query MySQL** to retrieve all rooms from `insa_rooms` table
4. **For each room**, the system executes:
    - Queries `service_urls` table to get the logical service name (e.g., "DOORMS")
    - Asks **Eureka** for the actual IP address and port of the service
    - Sends **POST /door/close** to DoorMS via REST
    - Logs the action in `room_state` table with timestamp
    - Repeats for WindowMS (close) and LightMS (off)
    - Checks **PresenceMS** via GET /presence/status
    - If presence detected: logs "Presence detected! Alarm activated"
    - If no presence: logs "No presence detected"

A presence can be simulated via the /orchestrator/room/{name}/simPresence end point.

Scenario 2 : Automatic morning mode activation

Morning mode is triggered automatically at 8:00 AM by the **TimerService**. This mode prepares all rooms for daily operations by opening doors and windows, and turning on lights. The execution flow goes as follow :

1. **TimerService** triggers via `@Scheduled` annotation (`cron = "0 0 8 * * ?"`)
2. **OrchestratorService** calls `morningModeForAllRooms()`
3. **Query MySQL** to retrieve all rooms
4. **For each room**, the system executes:
    - Discovers DoorMS via Eureka
    - Sends **POST /door/open** to DoorMS
    - Logs action in database
    - Discovers WindowMS via Eureka
    - Sends **POST /window/open** to WindowMS
    - Logs action in database

- Discovers LightMS via Eureka
- Sends **POST /light/on** to LightMS
- Logs action in database

Unlike night mode, morning mode does NOT check presence detection since the goal is to prepare rooms for expected occupancy, not to detect intrusions.

Scenario 3 : Presence detection with alarm

This scenario shows the conditional logic when the Orchestrator checks for presence in a room. The system must distinguish between normal operation and security breaches.

1. **OrchestratorService** sends GET request to **PresenceMS** `/presence/status`
2. **PresenceMS** returns `PresenceResponse` object with `detected` boolean
3. **Decision point**:
   - **If presence detected** (`detected = true`):
     - Set result = "Presence detected! Alarm activated"
     - Log alarm state to `room_state` table
     - This indicates a potential security issue during night mode
   - **If no presence** (`detected = false`):
     - Set result = "No presence detected"
     - Log normal state to `room_state` table
     - Room is secure and unoccupied as expected

Scenario 4 : Presence simulation

This scenario demonstrates the test endpoint that allows developers to manually simulate presence detection in a specific room. This is essential for testing the alarm functionality without requiring physical sensors.

1. **User** sends POST request: `/orchestrator/room/gei13/simPresence`
2. **OrchestratorController** receives request and calls `simulatePresence("gei13")`
3. **OrchestratorService** queries MySQL for presence service URL
4. **OrchestratorService** asks **Eureka** for PresenceMS location
5. **OrchestratorService** sends **POST /presence/detect** to PresenceMS
6. **PresenceMS** updates its internal state: `setDetected(true)`
7. **PresenceMS** returns `PresenceState` with `detected = true`
8. **Response** bubbles back: User receives "Presence Detected, alarm activated!"

Here is an example usage :

curl -X POST http://localhost:8086/orchestrator/room/gei13/simPresence

Other scenarios include manual night and morning mode for a specific room, as well as a room information query.

They can all be found in the following sequence diagram that encapsulates all of the diagrams => Orchestrator Night Mode Flow-2026-01-21-205859.png

8. Conclusion

This project successfully designed and implemented an intelligent system based on a modular microservices architecture using Spring Boot, Eureka, and MySQL. The clear separation between the orchestration layer and the microservices layer ensures a well-structured system in which each service, including DoorMS, LightMS, WindowMS, and PresenceMS, operates as an independent, loosely coupled, and easily testable component. This architectural approach improves maintainability, enhances fault isolation, and enables independent evolution of services without affecting the overall system.

The proposed architecture demonstrates strong scalability and extensibility capabilities. It allows the system to be easily expanded to manage additional rooms or entire buildings by updating the service registry and adapting the persistence models, without requiring major structural changes. Furthermore, the system is designed to support the future integration of additional environmental sensors, such as carbon dioxide and temperature sensors, which would contribute to more advanced building intelligence, improved energy efficiency, and enhanced occupant comfort.

Although the core functional requirements of the system have been successfully implemented, the development of a comprehensive graphical user interface remains an important perspective for future work. Due to project time constraints , no dedicated graphical interface was implemented. A complete web dashboard would provide users with an intuitive and centralized view of stored data, real-time sensor monitoring, and advanced manual control capabilities. Such an interface would significantly enhance system usability and facilitate its adoption in real-world smart building environments.